计 算 机 科 学 丛 书

Pearson

原书第4版

C++程序设计语言

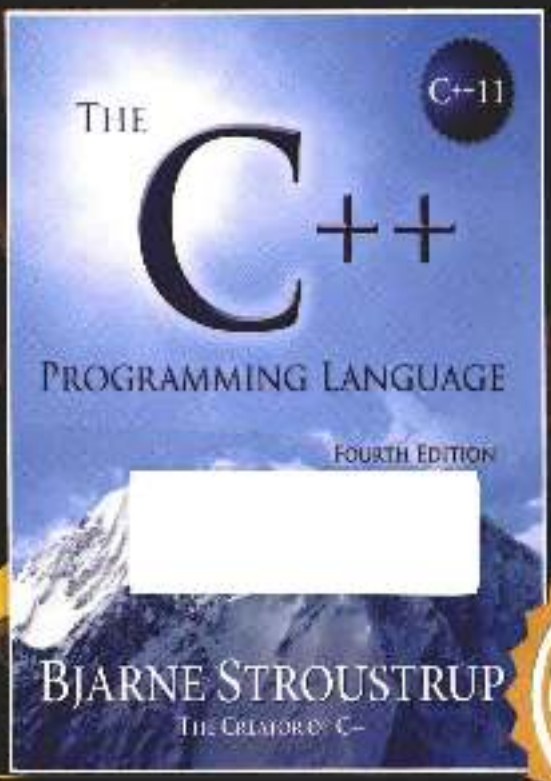
（第1~3部分）

[美] 本贾尼·斯特劳斯特鲁普（Bjarne Stroustrup） 著

王刚 杨巨峰 译

The C++ Programming Language

Fourth Edition



C++语言之父
经典名著
新版本

机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

原书第4版

C++程序设计语言

（第1~3部分）

[美] 本贾尼·斯特劳斯特鲁普（Bjarne Stroustrup） 著

王刚 杨巨峰 译

The C++ Programming Language

Fourth Edition



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

C++ 程序设计语言（第 1 - 3 部分）（原书第 4 版）/（美）本贾尼·斯特劳斯特鲁普（Bjarne Stroustrup）著；王刚，杨巨峰译．—北京：机械工业出版社，2016.6
（计算机科学丛书）
书名原文：The C++ Programming Language，Fourth Edition

ISBN 978-7-111-53941-4

Ⅰ. C… Ⅱ. ①本… ②王… ③杨… Ⅲ. C 语言－程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2016）第 121243 号

本书版权登记号：图字：01-2013-4811

Authorized translation from the English language edition, entitled *The C++ Programming Language, Fourth Edition*, 9780321563842 by Bjarne Stroustrup, published by Pearson Education, Inc., Copyright © 2013.

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 China Machine Press Copyright © 2016.

本书中文简体字版由 Pearson Education（培生出版教育集团）授权机械工业出版社在中华人民共和国境内（不包括中国台湾地区和中国香港、澳门特别行政区）独家出版发行，未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

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

《C++ 程序设计语言》（原书第 4 版）是 C++ 领域最经典的参考书，介绍了 C++11 的各项新特性和新功能。全书共分四部分。第一部分（第 1 ~ 5 章）是引言，包括 C++ 的背景知识，C++ 语言及其标准库的简要介绍；第二部分（第 6 - 15 章）介绍 C++ 的内置类型和基本特性，以及如何用它们构造程序；第三部分（第 16 - 29 章）介绍 C++ 的抽象机制及如何用这些机制编写面向对象程序和泛型程序；第四部分（第 30 - 44 章）概述标准库并讨论一些兼容性问题。

由于篇幅问题，原书中文版分两册出版，分别对应原书的第一至三部分和第四部分，这一册为第一至三部分。

本书适合计算机及相关专业本科生用作 C++ 课程的教材，也适合 C++ 程序设计新手和开发人员阅读。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：关　敏　　责任校对：殷　虹
印　　刷：北京市荣盛彩色印刷有限公司　　版　　次：2016 年 6 月第 1 版第 1 次印刷
开　　本：185mm×260mm　1/16　　印　　张：46.75
书　　号：ISBN 978-7-111-53941-4　　定　　价：139.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

版权所有·侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光／邹晓东

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出，独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅奠划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brian W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章教育

华章科技图书出版中心

译者序

The C++ Programming Language, Fourth Edition

历时近两年，终于翻译完了《C++程序设计语言》（原书第4版）。全书包含44章，英文原版共有1300多页，是C++语言之父Bjarne Stroustrup的一部呕心沥血之作。

这部巨著有几个特点：

一是知识结构完整，对C++语言的介绍非常全面。作者按照“基本功能”→“抽象机制”→“标准库”的递进层次组织全书，由浅入深地把C++语言的方方面面呈现在读者的面前。各种水平、各种背景的读者都能在书中找到适合自己的切入点和学习路径。

二是对细节的讲解非常深入，有利于读者了解和掌握语言的精华。作为C++语言的发明者和主要维护者，Bjarne Stroustrup在撰写本书时绝不仅仅满足于阐明语法和知识点本身。他试图向读者揭示各个语言功能的设计初衷，以及他对各种制约因素是如何考虑并妥协的。对于大多数读者来说，这种视角新奇而有趣。他们不再只是被动的学习者，在知道了“是什么”和“为什么”之后，还可以大胆地猜测“C++语言接下来该如何继续发展”。不得不说，这是本书与其他C++书籍的最大区别。

三是作者在写作中融入了很多自己的工程实践经验。学习程序设计语言与学习文化课有很大的不同。设计程序的过程是一门艺术，程序语言只是完成艺术作品所需的工具。举个例子来说，由于各种各样的原因，在C++中存在一些语言特性，它们的功能和作用非常类似。那么这些特性之间是何关系？在遇到某类实际问题时应该如何聪明地选择？本书很好地回答了此类问题。

以译者的浅见，程序员应该是艺术家（Artist），而非匠人（Worker）——后者只会堆砌代码，而前者能创造出美好的作品。这也应该是Bjarne Stroustrup写作本书时所追求的吧！

这本译著的出版凝结了很多人的智慧和心血，绝非译者二人独力可为。感谢机械工业出版社的朱劼、关敏等编辑在本书译校和出版过程中的辛勤付出，她们给予了我们很多无私的帮助。由于译者水平有限，书中难免有一些不当之处，恳请读者不吝批评指正。

译者

2016年春于南开园

所有计算机科学问题，
都可以通过引入一个新的间接层次来解决，
那些已有过多间接层次的问题除外。
——David J. Wheeler

与C++98标准相比，C++11标准让我可以更清晰、更简洁而且更直接地表达自己的想法。而且，新版本的编译器可以对程序进行更好的检查并生成更快的目标程序。因此，C++11给人的感觉就像是一种新语言一样。

在本书中，我追求完整性（completeness）。我会介绍专业程序员可能需要的每个语言特性和标准库组件。对每个特性或组件，我将给出：

- 基本原理：设计这个特性（组件）是为了帮助解决哪类问题？其设计原理是什么？它有什么根本的局限？
- 规范：它该如何定义？我将以专业程序员为目标读者来选择内容的详略程度，对于要求更高的C++语言研究者，有很多ISO标准的文献可供查阅。
- 例子：当单独使用这个特性或与其他特性组合使用时，如何用好它？其中的关键技术和习惯用法是怎样的？在程序的可维护性和性能方面是否有一些隐含的问题？

多年来，无论是C++语言本身还是它的使用，都已经发生了巨大改变。从程序员的角度，大多数改变都属于语言的改进。与之前的版本相比，当前的ISO C++标准（ISO/IEC 14882-2011，通常称为C++11）在编写高质量代码方面无疑是一个好得多的工具。但是它好在哪里？现代C++语言支持什么样的程序设计风格和技术？这些技术靠哪些语言特性和标准库特性来支撑？精练、正确，可维护性好、性能高的C++代码的基本构建单元是怎样的？本书将回答这些关键问题。很多答案已经不同于1985、1995或2005等旧版本的C++语言了；C++在进步。

C++是一种通用程序设计语言，它强调富类型、轻量级抽象的设计和使用。C++特别适合开发资源受限的应用，例如可在软件基础设施中发现的那些应用。那些花费时间学习高质量代码编写技术的程序员将会从C++语言受益良多。C++是为那些严肃对待编程的人而设计的。人类文明已经严重依赖软件，编写高质量的软件非常重要。

目前已经部署的C++代码达到数十亿行，因此程序稳定性备受重视——很多1985年和1995年编写的C++代码仍然运行良好，而且还会继续运行几十年。但是，对所有这些应用程序，都可以用现代C++语言写出更好的版本；如果你墨守成规，将来写出的代码将会是低质量、低性能的。对稳定性的强调还意味着，你现在遵循标准写出的代码，在未来几十年中会运行良好。本书中所有代码都遵循2011 ISO C++标准。

本书面向三类读者：

- 想知道最新的ISO C++标准都提供了哪些新特性的C++程序员。

- 好奇 C++ 到底提供了哪些超越 C 语言的特性的 C 程序员。
- 具备 Java、C#、Python 和 Ruby 等编程语言背景，正在探寻“更接近机器”的语言，即更灵活、提供更好的编译时检查或是更好性能的语言的程序员。

自然，这三类读者可能是有交集的——一个专业软件开发者通常掌握多门编程语言。

本书假定目标读者是程序员。如果你想问“什么是 for 循环？”或是“什么是编译器？”，那么本书现在还不适合你，我向你推荐我的另一本书《C++ 程序设计原理与实践》[⊖]，这本书适合作为程序设计和 C++ 语言的入门书籍。而且，我假定读者是较为成熟的软件开发者。如果你的问题是“为什么要费力进行测试？”或者认为“所有语言基本都是一样的，给我看语法就可以了”，甚至确信存在一种适合所有任务的完美语言，那么本书也不适合你。

相对于 C++98，C++11 提出了哪些改进和新特性呢？适合现代计算机的机器模型会涉及大量并发处理。为此，C++11 提供了用于系统级并行编程（如使用多核）的语言和标准库特性。C++11 还提供了正则表达式处理、资源管理指针、随机数、改进的容器（包括哈希表）以及其他很多特性。此外，C++11 还提供了通用和一致的初始化机制、更简单的 for 语句、移动语义、基础的 Unicode 支持、lambda 表达式、通用常量表达式、控制类缺省定义的能力、可变参数模板、用户定义的字面值常量和其他很多新特性。请记住，这些标准库和语言特性的目标就是支撑那些用来开发高质量软件的程序设计技术。这些特性应该组合使用——将它们看作盖大楼的砖，而不应该相互隔离地单独使用来解决特定问题。计算机是一种通用机器，而 C++ 在其中起着重要作用。特别是，C++ 的设计目标就是足够灵活和通用，以便处理那些连它的设计者都未曾想象过的未来难题。

致谢

除了本书上一版致谢提及的人之外，我还要感谢 Pete Becker、Hans-J. Boehm、Marshall Clow、Jonathan Coe、Lawrence Crowl、Walter Daugherty、J. Daniel Garcia、Robert Harle、Greg Hickman、Howard Hinnant、Brian Kernighan、Daniel Krügler、Nevin Liber、Michel Michaud、Gary Powell、Jan Christiaan van Winkel 和 Leor Zolman。没有他们的帮助，本书的质量要差得多。

感谢 Howard Hinnant 为我解答很多有关标准库的问题。

Andrew Sutton 是 Origin 库的作者，模板相关章节中很多模拟概念的讨论都是基于这个测试平台的。他还是 Matrix 库的作者，这是第 29 章的主题。Origin 库是开源的，在互联网上搜索“Origin”和“Andrew Sutton”就能找到。

感谢我指导的毕业设计班，他们从第一部分中找出的问题比其他任何人都多。

假如我能遵照审阅人的所有建议，毫无疑问会大幅度提高本书的质量，但篇幅上也会增加数百页。每个专家审阅人都建议增加技术细节、进阶示例和很多有用的开发规范；每个新手审阅人（或教育工作者）都建议增加示例；而大多数审阅人都（正确地）注意到本书的篇幅可能过长了。

⊖ 该书原文影印版及中文翻译版已由机械工业出版社出版，书号分别为 ISBN 978-7-111-28248-8 和 ISBN 978-7-111-30322-0。——编辑注

感谢普林斯顿大学计算机科学系，特别感谢 Brian Kernighan 教授，在我利用部分休假时间撰写此书时给予我热情接待。

感谢剑桥大学计算机实验室，特别感谢 Andy Hopper 教授，在我利用部分休假时间撰写此书时给予我热情接待。

感谢我的编辑 Peter Gordon 以及他在 Addison-Wesley 的出版团队，感谢你们的帮助和耐心。

Bjarne Stroustrup
于得克萨斯大学城

第3版前言

The C++ Programming Language, Fourth Edition

去编程就是去理解。

——Kristen Nygaard

我觉得用C++编程比以往更令人愉快。在过去这些年里，C++在支持设计和编程方面取得了令人振奋的进步，针对其使用的大量新技术已经被开发出来了。然而，C++并不只是好玩。普通的程序员在几乎所有种类和规模的开发项目上，在生产率、可维护性、灵活性和质量方面都取得了显著的进步。到今天为止，C++已经实现了我当初期望中的绝大部分，还在许多我原来根本没有梦想过的工作中取得了成功。

本书介绍的是标准C++[⊖]以及由C++所支持的关键编程技术和设计技术。与本书第1版所介绍的那个C++版本相比，标准C++是一个经过了更仔细推敲的更强大的语言。各种新的语言特征，如名字空间、异常、模板，以及运行时类型识别，使人能以比过去更直接的方式使用许多技术，标准库使程序员能够从比基本语言高得多的层面上起步。

本书第2版中大约有三分之一的内容来自第1版。第3版则重写了更大的篇幅。它提供的许多东西是大部分有经验的程序员也需要的，与此同时，本书也比它的以前版本更容易让新手入门。C++使用的爆炸性增长和由此带来的海量经验积累使这些成为可能。

一个功能广泛的标准库定义使我能以一种与以前不同的方式介绍C++的各种概念。与过去一样，本书对C++的介绍与任何特定的实现都没有关系；与过去一样，教材式的各章还是采用“自下而上”的方式，使每种结构都是在定义之后才使用。无论如何，使用一个设计良好的库远比理解其实现细节容易得多。因此，假定读者在理解标准库的内部工作原理之前，就可以利用它提供许多更实际、更有趣的例子。标准库本身也是程序设计示例和设计技术的丰富源泉。

本书将介绍每种主要的C++语言特征和标准库，它是围绕着语言和库功能组织起来的。当然，各种特征都将在使用它们的环境中介绍。也就是说，这里所关注的是将语言作为一种设计和编程的工具，而不是语言本身。本书将展示那些使C++卓有成效的关键技术，讲述为掌握它们所需要的基本概念。除了专门阐释技术细节的那些地方之外，其他示例都取自系统软件领域。另一本与本书配套出版的书《带标注的C++语言标准》(*The Annotated C++ Language Standard*)，将给出完整的语言定义，所附标注能使它更容易理解。

本书的基本目标就是帮助读者理解C++所提供的功能将如何支持关键的程序设计技术。这里的目标是使读者能远远超越简单地复制示例并使之能够运行，或者模仿来自其他语言的程序设计风格。只有对隐藏在语言背后的思想有了很好的理解之后，才能真正掌握这个语言。如果有一些具体实现的文档的辅助，这里所提供的信息就足以对付具有挑战性的真实世界中的重要项目。我的希望是，本书能帮助读者获得新的洞察力，使他们成为更好的程序员和设计师。

⊖ ISO/IEC 14882，C++程序设计语言标准。

致谢

除了第1版和第2版的致谢中所提到的那些人之外，我还要感谢 Matt Austern、Hans Boehm、Don Caldwell、Lawrence Crowl、Alan Feuer、Andrew Forrest、David Gay、Tim Griffin、Peter Juhl、Brian Kernighan、Andrew Koenig、Mike Mowbray、Rob Murray、Lee Nackman、Joseph Newcomer、Alex Stepanov、David Vandevoorde、Peter Weinberger 和 Chris Van Wyk，他们对第3版各章的初稿提出了许多意见。没有他们的帮助和建议，这本书一定会更难理解，包含更多的错误，没有这么完整，当然也可能稍微短一点。

我还要感谢C++标准化委员会的志愿者们，是他们完成了规模宏大的建设性工作，才使C++具有它今天这个样子。要罗列出每个人会有一点不公平，但一个也不提就更不公平，所以我想特别提及 Mike Ball、Dag Brück、Sean Corfield、Ted Goldstein、Kim Knuttila、Andrew Koenig、Dmitry Lenkov、Nathan Myers、Martin O'Riordan、Tom Plum、Jonathan Shopiro、John Spicer、Jerry Schwarz、Alex Stepanov 和 Mike Vilot，他们中的每个人都在C++及其标准库的某些方面直接与我合作过。

在这本书第一次印刷之后，许多人给我发来电子邮件，提出更正和建议。我已经在本书的结构中响应了他们的建议，使后来出版的版本大为改善。将本书翻译到各种语言的译者也提供了许多澄清性的意见。作为对这些读者的回应，我增加了附录D和附录E。让我借这个机会感谢他们之中特别有帮助的几位：Dave Abrahams、Matt Austern、Jan Bielawski、Janina Mincer Daszkiewicz、Andrew Koenig、Dietmar Kühl、Nicolai Josuttis、Nathan Myers、Paul E. Sevinç、Andy Tenne-Sens、Shoichi Uchida、Ping-Fai（Mike）Yang 和 Dennis Yelle。

Bjarne Stroustrup
于新泽西莫里山

第 2 版前言

The C++ Programming Language, Fourth Edition

前路漫漫。

——Bilbo Baggins

正如在本书的第 1 版中所承诺的，C++ 为满足其用户的需要正在不断地演化。这一演化过程得益于许多有着极大的背景差异，在范围广泛的应用领域中工作的用户们的实际经验的指导。在第 1 版出版后的六年中，C++ 的用户群体扩大了不止百倍，人们学到了许多东西，发现了许多新技术并通过了实践的检验。这些技术中的一些也在这一版中有所反映。

在过去六年里所完成的许多语言扩展，其基本宗旨就是将 C++ 提升为一种服务于一般性的数据抽象和面向对象程序设计的语言，特别是提升为一个可编写高质量的用户定义类型库的工具。一个“高质量的库”是指这样的库，它以一个或几个方便、安全且高效的类的形式，给用户提供了一个概念。在这个环境中，安全意味着这个类在库的使用者与它的供方之间构成了一个特殊的类型安全的界面；高效意味着与手工写出的 C 代码相比，这种库的使用不会给用户强加明显的运行时间上或空间上的额外开销。

本书介绍的是完整的 C++ 语言。从第 1 章到第 10 章是一个教材式的导引，第 11 章到第 13 章展现的是一个有关设计和软件开发问题的讨论，最后包含了完整的 C++ 参考手册。自然，在原来版本之后新加入的特征和变化已成为这个展示的有机组成部分。这些特征包括：经过精化后的重载解析规则和存储管理功能，以及访问控制机制、类型安全的连接、const 和 static 成员函数、抽象类、多重继承、模板和异常处理。

C++ 是一个通用的程序设计语言，其核心应用领域是最广泛意义上的系统程序设计。此外，C++ 还被成功地用到许多无法称为系统程序设计的应用领域中。从最摩登的小型计算机到最大的超级计算机上，以及几乎所有操作系统上都有 C++ 的实现。因此，本书描述的是 C++ 语言本身，并不想试着去解释任何特殊的实现、程序设计环境或者库。

本书中给出的许多类的示例虽然都很有用，但也还是应该归到“玩具”一类。与在完整的精益求精的程序中做解释相比，这里所采用的解说风格能更清晰地呈现那些具有普遍意义的原理和极其有用的技术，在实际例子中它们很容易被细节所淹没。这里给出的大部分有用的类，如链接表、数组、字符串、矩阵、图形类、关联数组等，在广泛可用的各种商品和非商品资源中，都有可用的“防弹”或“金盘”版本。那些“具有工业强度”的类和库中的许多东西，实际上不过是在这里可以找到的玩具版本的直接或间接后裔。

与第 1 版相比，这一版更加强调本书在教学方面的作用。然而，这里的叙述仍然是针对有经验的程序员，并努力不去轻视他们的智慧和经验。有关设计问题的讨论有了很大的扩充，作为对读者在语言特征及其直接应用之外的要求的一种回应。技术细节和精确性也有所增强。特别是，这里的参考手册体现了在这个方向上多年的工作。我的目标是提供一本具有足够深度的书籍，使大部分程序员能在多次阅读中都有所收获。换句话说，这本书给出的是 C++ 语言，它的基本原理，以及使用时所需要的关键性技术。欢迎欣赏！

致谢

除了在第1版前言的致谢里所提到的人们之外，我还要感谢Al Aho、Steve Buroff、Jim Coplien、Ted Goldstein、Tony Hansen、Lorraine Juhl、Peter Juhl、Brian Kernighan、Andrew Koenig、Bill Leggett、Warren Montgomery、Mike Mowbray、Rob Murray、Jonathan Shopiro、Mike Vilot和Peter Weinberger，他们对第2版的初稿提出了许多意见。许多人对C++从1985年到1991年的开发有很大影响。我只能提及其中几个：Andrew Koenig，Brian Kernighan，Doug McIlroy和Jonathan Shopiro。还要感谢参考手册“外部评阅”的许多参与者，以及在X3J16的整个第一年里一直在其中受苦的人们。

Bjarne Stroustrup
于新泽西贝尔里山

第1版前言

The C++ Programming Language, Fourth Edition

语言磨砺了我们思维的方式，
也决定着我们思考的范围。
——B. L. Whorf

C++是一种通用的程序设计语言，其设计就是为了使认真的程序员工作得更愉快。除了一些小细节之外，C++是C程序设计语言的一个超集。C++提供了C所提供的各种功能，还为定义新类型提供了灵活而有效的功能。程序员可以通过定义新类型，使这些类型与应用中的概念紧密对应，从而把一个应用划分成许多容易管理的片段。这种程序构造技术通常被称为数据抽象。某些用户定义类型的对象包含着类型信息，这种对象就可以方便而安全地用在那种对象类型无法在编译时确定的环境中。使用这种类型的对象的程序通常被称为是基于对象的。如果用得好，这些技术可以产生出更短、更容易理解，而且也更容易管理的程序。

C++里的最关键概念是类。一个类就是一个用户定义类型。类提供了对数据的隐藏，数据的初始化保证，用户定义类型的隐式类型转换，动态类型识别，用户控制的存储管理，以及重载运算符的机制等。在类型检查和表述模块性方面，C++提供了比C好得多的功能。它还包含了许多并不直接与类相关的改进，包括符号常量、函数的在线替换、默认函数参数、重载函数名、自由存储管理运算符，以及引用类型等。C++保持了C高效处理硬件基本对象（位、字节、字、地址等）的能力。这就使用户定义类型能够在相当高的效率水平上实现。

C++及其标准库也是为了可移植性而设计的。当前的实现能够在大多数支持C的系统上运行。C的库也能用于C++程序，而且大部分支持C程序设计的工具也同样能用于C++。

本书的基本目标就是帮助认真的程序员学习这个语言，并将它用于那些非平凡的项目。书中提供了有关C++的完整描述，许多完整的例子，以及更多的程序片段。

致谢

如果没有许多朋友和同事持之以恒的使用、建议和建设性的批评，C++绝不会像它现在这样成熟。特别地，Tom Cargill、Jim Coplien、Stu Feldman、Sandy Fraser、Steve Johnson、Brian Kernighan、Bart Locanthi、Doug McIlroy、Dennis Ritchie、Larry Rosler、Jerry Schwarz和Jon Shopiro对语言发展提供了重要的思想。Dave Presotto写出了流I/O库的当前实现。

此外，还有几百人对C++及其编译器的开发做出了贡献：给我提出改进的建议，描述所遇到的问题，告诉我编译中的错误等。我只能提及其中的很少几位：Gary Bishop、Andrew Hume、Tom Karzes、Victor Milenkovic、Rob Murray、Leonie Rose、Brian Schmult和Gary Walker。

许多人在本书的撰写过程中为我提供了帮助，特别值得提出的是Jon Bentley、Laura

Eaves、Brian Kernighan、Ted Kowalski、Steve Mahaney、Jon Shopiro，以及参加1985年7月26～27日俄亥俄州哥伦布贝尔实验室C++课程的人们。

Bjarne Stroustrup
于新泽西莫里山

目 录

The C++ Programming Language, Fourth Edition

出版者的话
译者序
前言
第 3 版前言
第 2 版前言
第 1 版前言

第一部分 引言

第 1 章 致读者 …… 2

1.1 本书结构 …… 2
1.1.1 引言 …… 2
1.1.2 基本特性 …… 3
1.1.3 抽象机制 …… 4
1.1.4 标准库 …… 5
1.1.5 例子和参考文献 …… 5
1.2 C++ 的设计 …… 7
1.2.1 程序设计风格 …… 8
1.2.2 类型检查 …… 11
1.2.3 C 兼容性 …… 12
1.2.4 语言、库和系统 …… 12
1.3 学习 C++ …… 14
1.3.1 用 C++ 编程 …… 15
1.3.2 对 C++ 程序员的建议 …… 16
1.3.3 对 C 程序员的建议 …… 16
1.3.4 对 Java 程序员的建议 …… 17
1.4 C++ 的历史 …… 18
1.4.1 大事年表 …… 19
1.4.2 早期的 C++ …… 19
1.4.3 1998 标准 …… 21
1.4.4 2011 标准 …… 23
1.4.5 C++ 的用途 …… 26
1.5 建议 …… 27
1.6 参考文献 …… 28

第 2 章 C++ 概览：基础知识 …… 32

2.1 引言 …… 32
2.2 基本概念 …… 33
2.2.1 Hello, World! …… 33
2.2.2 类型、变量和算术运算 …… 34
2.2.3 常量 …… 36
2.2.4 检验和循环 …… 37
2.2.5 指针、数组和循环 …… 38
2.3 用户自定义类型 …… 40
2.3.1 结构 …… 41
2.3.2 类 …… 42
2.3.3 枚举 …… 43
2.4 模块化 …… 44
2.4.1 分离编译 …… 45
2.4.2 名字空间 …… 46
2.4.3 错误处理 …… 47
2.5 附记 …… 50
2.6 建议 …… 50

第 3 章 C++ 概览：抽象机制 …… 51

3.1 引言 …… 51
3.2 类 …… 51
3.2.1 具体类型 …… 52
3.2.2 抽象类型 …… 56
3.2.3 虚函数 …… 58
3.2.4 类层次 …… 59
3.3 拷贝和移动 …… 62
3.3.1 拷贝容器 …… 63
3.3.2 移动容器 …… 64
3.3.3 资源管理 …… 66
3.3.4 抑制操作 …… 66
3.4 模板 …… 67
3.4.1 参数化类型 …… 67
3.4.2 函数模板 …… 69
3.4.3 函数对象 …… 69

3.4.4 可变参数模板 …… 71
3.4.5 别名 …… 72
3.5 建议 …… 73

第 4 章 C++ 概览：容器与算法 …… 74

4.1 标准库 …… 74
4.1.1 标准库概述 …… 75
4.1.2 标准库头文件与名字空间 …… 75
4.2 字符串 …… 77
4.3 I/O 流 …… 78
4.3.1 输出 …… 78
4.3.2 输入 …… 79
4.3.3 用户自定义类型的 I/O …… 80
4.4 容器 …… 81
4.4.1 vector …… 81
4.4.2 list …… 84
4.4.3 map …… 85
4.4.4 unordered_map …… 86
4.4.5 容器概述 …… 86
4.5 算法 …… 87
4.5.1 使用迭代器 …… 88
4.5.2 迭代器类型 …… 90
4.5.3 流迭代器 …… 91
4.5.4 谓词 …… 93
4.5.5 算法概述 …… 93
4.5.6 容器算法 …… 94
4.6 建议 …… 94

第 5 章 C++ 概览：并发与实用功能 … 96

5.1 引言 …… 96
5.2 资源管理 …… 96
5.2.1 unique_ptr 与 shared_ptr …… 97
5.3 并发 …… 99
5.3.1 任务和 thread …… 99
5.3.2 传递参数 …… 100
5.3.3 返回结果 …… 100
5.3.4 共享数据 …… 101
5.3.5 任务通信 …… 103
5.4 小工具组件 …… 106
5.4.1 时间 …… 106
5.4.2 类型函数 …… 107
5.4.3 pair 和 tuple …… 109
5.5 正则表达式 …… 110
5.6 数学计算 …… 111
5.6.1 数学函数和算法 …… 111
5.6.2 复数 …… 111
5.6.3 随机数 …… 112
5.6.4 向量算术 …… 113
5.6.5 数值限制 …… 113
5.7 建议 …… 114

第二部分 基本功能

第 6 章 类型与声明 …… 116

6.1 ISO C++ 标准 …… 116
6.1.1 实现 …… 117
6.1.2 基本源程序字符集 …… 118
6.2 类型 …… 118
6.2.1 基本类型 …… 119
6.2.2 布尔值 …… 119
6.2.3 字符类型 …… 121
6.2.4 整数类型 …… 124
6.2.5 浮点数类型 …… 126
6.2.6 前缀和后缀 …… 127
6.2.7 void …… 128
6.2.8 类型尺寸 …… 128
6.2.9 对齐 …… 130
6.3 声明 …… 131
6.3.1 声明的结构 …… 133
6.3.2 声明多个名字 …… 134
6.3.3 名字 …… 134
6.3.4 作用域 …… 136
6.3.5 初始化 …… 138
6.3.6 推断类型：auto 和 decltype() …… 141
6.4 对象和值 …… 144
6.4.1 左值和右值 …… 144
6.4.2 对象的生命周期 …… 145
6.5 类型别名 …… 146
6.6 建议 …… 147

第 7 章　指针、数组与引用 …… 148

7.1　引言 …… 148
7.2　指针 …… 148
7.2.1　void* …… 149
7.2.2　nullptr …… 150
7.3　数组 …… 150
7.3.1　数组的初始化器 …… 152
7.3.2　字符串字面值常量 …… 152
7.4　数组中的指针 …… 155
7.4.1　数组漫游 …… 156
7.4.2　多维数组 …… 158
7.4.3　传递数组 …… 159
7.5　指针与 const …… 161
7.6　指针与所有权 …… 163
7.7　引用 …… 163
7.7.1　左值引用 …… 164
7.7.2　右值引用 …… 167
7.7.3　引用的引用 …… 169
7.7.4　指针与引用 …… 170
7.8　建议 …… 172

第 8 章　结构、联合与枚举 …… 173

8.1　引言 …… 173
8.2　结构 …… 173
8.2.1　struct 的布局 …… 175
8.2.2　struct 的名字 …… 176
8.2.3　结构与类 …… 177
8.2.4　结构与数组 …… 178
8.2.5　类型等价 …… 180
8.2.6　普通旧数据 …… 180
8.2.7　域 …… 182
8.3　联合 …… 183
8.3.1　联合与类 …… 185
8.3.2　匿名 union …… 186
8.4　枚举 …… 188
8.4.1　enum class …… 188
8.4.2　普通的 enum …… 191
8.4.3　未命名的 enum …… 192
8.5　建议 …… 193

第 9 章　语句 …… 194

9.1　引言 …… 194
9.2　语句概述 …… 194
9.3　声明作为语句 …… 195
9.4　选择语句 …… 196
9.4.1　if 语句 …… 196
9.4.2　switch 语句 …… 198
9.4.3　条件中的声明 …… 200
9.5　循环语句 …… 201
9.5.1　范围 for 语句 …… 201
9.5.2　for 语句 …… 202
9.5.3　while 语句 …… 203
9.5.4　do 语句 …… 203
9.5.5　退出循环 …… 204
9.6　goto 语句 …… 204
9.7　注释与缩进 …… 205
9.8　建议 …… 207

第 10 章　表达式 …… 208

10.1　引言 …… 208
10.2　一个桌面计算器示例 …… 208
10.2.1　分析器 …… 209
10.2.2　输入 …… 213
10.2.3　底层输入 …… 216
10.2.4　错误处理 …… 217
10.2.5　驱动程序 …… 217
10.2.6　头文件 …… 218
10.2.7　命令行参数 …… 218
10.2.8　关于风格 …… 220
10.3　运算符概述 …… 220
10.3.1　结果 …… 224
10.3.2　求值顺序 …… 224
10.3.3　运算符优先级 …… 225
10.3.4　临时对象 …… 226
10.4　常量表达式 …… 227
10.4.1　符号化常量 …… 229
10.4.2　常量表达式中的 const …… 229
10.4.3　字面值常量类型 …… 229
10.4.4　引用参数 …… 230

10.4.5 地址常量表达式 …… 231
10.5 隐式类型转换 …… 231
10.5.1 提升 …… 231
10.5.2 类型转换 …… 232
10.5.3 常用的算术类型转换 …… 234
10.6 建议 …… 235

第 11 章 选择适当的操作 …… 236

11.1 其他运算符 …… 236
11.1.1 逗号运算符 …… 236
11.1.2 位逻辑运算符 …… 236
11.1.3 条件表达式 …… 238
11.1.4 逗号与逗点 …… 238
11.2 自由存储 …… 240
11.2.1 内存管理 …… 241
11.2.2 数组 …… 243
11.2.3 获取内存空间 …… 244
11.2.4 重载 new …… 245
11.3 列表 …… 247
11.3.1 实现模型 …… 248
11.3.2 限定列表 …… 249
11.3.3 未限定列表 …… 249
11.4 lambda 表达式 …… 251
11.4.1 实现模型 …… 251
11.4.2 lambda 的替代品 …… 252
11.4.3 捕获 …… 254
11.4.4 调用与返回 …… 257
11.4.5 lambda 的类型 …… 257
11.5 显式类型转换 …… 258
11.5.1 构造 …… 259
11.5.2 命名转换 …… 261
11.5.3 C 风格的转换 …… 262
11.5.4 函数形式的转换 …… 262
11.6 建议 …… 263

第 12 章 函数 …… 264

12.1 函数声明 …… 264
12.1.1 为什么使用函数 …… 265
12.1.2 函数声明的组成要件 …… 265
12.1.3 函数定义 …… 266
12.1.4 返回值 …… 267
12.1.5 inline 函数 …… 269
12.1.6 constexpr 函数 …… 269
12.1.7 [[noreturn]] 函数 …… 271
12.1.8 局部变量 …… 272
12.2 参数传递 …… 273
12.2.1 引用参数 …… 273
12.2.2 数组参数 …… 275
12.2.3 列表参数 …… 277
12.2.4 数量未定的参数 …… 278
12.2.5 默认参数 …… 281
12.3 重载函数 …… 282
12.3.1 自动重载解析 …… 283
12.3.2 重载与返回类型 …… 284
12.3.3 重载与作用域 …… 285
12.3.4 多实参解析 …… 285
12.3.5 手动重载解析 …… 286
12.4 前置与后置条件 …… 286
12.5 函数指针 …… 288
12.6 宏 …… 292
12.6.1 条件编译 …… 294
12.6.2 预定义宏 …… 295
12.6.3 编译指令 …… 296
12.7 建议 …… 296

第 13 章 异常处理 …… 297

13.1 错误处理 …… 297
13.1.1 异常 …… 298
13.1.2 传统的错误处理 …… 299
13.1.3 渐进决策 …… 300
13.1.4 另一种观点看异常 …… 301
13.1.5 何时不应使用异常 …… 302
13.1.6 层次化错误处理 …… 303
13.1.7 异常与效率 …… 304
13.2 异常保障 …… 305
13.3 资源管理 …… 307
13.3.1 finally …… 310
13.4 强制不变式 …… 311
13.5 抛出与捕获异常 …… 315
13.5.1 抛出异常 …… 315

13.5.2 捕获异常 …… 318
13.5.3 异常与线程 …… 324
13.6 vector 的实现 …… 324
13.6.1 一个简单的 vector …… 325
13.6.2 显式地表示内存 …… 328
13.6.3 赋值 …… 331
13.6.4 改变尺寸 …… 332
13.7 建议 …… 335

第 14 章 名字空间 …… 337

14.1 组合问题 …… 337
14.2 名字空间 …… 338
14.2.1 显式限定 …… 339
14.2.2 using 声明 …… 340
14.2.3 using 指示 …… 341
14.2.4 参数依赖查找 …… 342
14.2.5 名字空间是开放的 …… 344
14.3 模块化和接口 …… 345
14.3.1 名字空间作为模块 …… 346
14.3.2 实现 …… 348
14.3.3 接口和名字 …… 349
14.4 组合使用名字空间 …… 351
14.4.1 便利性与安全性 …… 351
14.4.2 名字空间别名 …… 352
14.4.3 组合名字空间 …… 352
14.4.4 组合与选择 …… 353
14.4.5 名字空间和重载 …… 354
14.4.6 版本控制 …… 356
14.4.7 名字空间嵌套 …… 358
14.4.8 无名名字空间 …… 359
14.4.9 C 头文件 …… 359
14.5 建议 …… 360

第 15 章 源文件与程序 …… 362

15.1 分离编译 …… 362
15.2 链接 …… 363
15.2.1 文件内名字 …… 365
15.2.2 头文件 …… 366
15.2.3 单一定义规则 …… 368
15.2.4 标准库头文件 …… 369
15.2.5 链接非 C++ 代码 …… 370
15.2.6 链接和函数指针 …… 372
15.3 使用头文件 …… 373
15.3.1 单头文件组织 …… 373
15.3.2 多头文件组织 …… 376
15.3.3 包含保护 …… 380
15.4 程序 …… 381
15.4.1 非局部变量初始化 …… 381
15.4.2 初始化和并发 …… 382
15.4.3 程序终止 …… 383
15.5 建议 …… 384

第三部分 抽象机制

第 16 章 类 …… 386

16.1 引言 …… 386
16.2 类基础 …… 387
16.2.1 成员函数 …… 388
16.2.2 默认拷贝 …… 389
16.2.3 访问控制 …… 389
16.2.4 class 和 struct …… 390
16.2.5 构造函数 …… 391
16.2.6 explicit 构造函数 …… 393
16.2.7 类内初始化器 …… 395
16.2.8 类内函数定义 …… 395
16.2.9 可变性 …… 396
16.2.10 自引用 …… 399
16.2.11 成员访问 …… 400
16.2.12 static 成员 …… 401
16.2.13 成员类型 …… 403
16.3 具体类 …… 403
16.3.1 成员函数 …… 406
16.3.2 辅助函数 …… 408
16.3.3 重载运算符 …… 410
16.3.4 具体类的重要性 …… 410
16.4 建议 …… 411

第 17 章 构造、清理、拷贝和移动 … 413

17.1 引言 …… 413
17.2 构造函数和析构函数 …… 415

17.2.1 构造函数和不变式 …… 415
17.2.2 析构函数和资源 …… 416
17.2.3 基类和成员析构函数 …… 417
17.2.4 调用构造函数和析构函数 …… 418
17.2.5 virtual 析构函数 …… 419
17.3 类对象初始化 …… 420
17.3.1 不使用构造函数进行初始化 …… 420
17.3.2 使用构造函数进行初始化 …… 421
17.3.3 默认构造函数 …… 424
17.3.4 初始化器列表构造函数 …… 425
17.4 成员和基类初始化 …… 429
17.4.1 成员初始化 …… 429
17.4.2 基类初始化器 …… 431
17.4.3 委托构造函数 …… 431
17.4.4 类内初始化器 …… 432
17.4.5 static 成员初始化 …… 434
17.5 拷贝和移动 …… 435
17.5.1 拷贝 …… 435
17.5.2 移动 …… 441
17.6 生成默认操作 …… 444
17.6.1 显式声明默认操作 …… 444
17.6.2 默认操作 …… 445
17.6.3 使用默认操作 …… 446
17.6.4 使用 delete 删除的函数 …… 449
17.7 建议 …… 451

第 18 章 运算符重载 …… 452

18.1 引言 …… 452
18.2 运算符函数 …… 453
18.2.1 二元和一元运算符 …… 454
18.2.2 运算符的预置含义 …… 455
18.2.3 运算符与用户自定义类型 …… 456
18.2.4 传递对象 …… 456
18.2.5 名字空间中的运算符 …… 457
18.3 复数类型 …… 459
18.3.1 成员和非成员运算符 …… 459
18.3.2 混合模式运算 …… 460
18.3.3 类型转换 …… 461
18.3.4 字面值常量 …… 463
18.3.5 访问函数 …… 464
18.3.6 辅助函数 …… 465
18.4 类型转换 …… 466
18.4.1 类型转换运算符 …… 466
18.4.2 explicit 类型转换运算符 …… 467
18.4.3 二义性 …… 468
18.5 建议 …… 469

第 19 章 特殊运算符 …… 471

19.1 引言 …… 471
19.2 特殊运算符 …… 471
19.2.1 取下标 …… 471
19.2.2 函数调用 …… 472
19.2.3 解引用 …… 473
19.2.4 递增和递减 …… 475
19.2.5 分配和释放 …… 477
19.2.6 用户自定义字面值常量 …… 478
19.3 字符串类 …… 481
19.3.1 必备操作 …… 481
19.3.2 访问字符 …… 482
19.3.3 类的表示 …… 483
19.3.4 成员函数 …… 485
19.3.5 辅助函数 …… 487
19.3.6 应用 String …… 489
19.4 友元 …… 490
19.4.1 发现友元 …… 491
19.4.2 友元与成员 …… 492
19.5 建议 …… 493

第 20 章 派生类 …… 495

20.1 引言 …… 495
20.2 派生类 …… 496
20.2.1 成员函数 …… 498
20.2.2 构造函数和析构函数 …… 499
20.3 类层次 …… 500
20.3.1 类型域 …… 500
20.3.2 虚函数 …… 502
20.3.3 显式限定 …… 504
20.3.4 覆盖控制 …… 505
20.3.5 using 基类成员 …… 508

20.3.6 返回类型放松 …… 511
20.4 抽象类 …… 512
20.5 访问控制 …… 514
20.5.1 protected 成员 …… 517
20.5.2 访问基类 …… 518
20.5.3 using 声明与访问控制 …… 519
20.6 成员指针 …… 520
20.6.1 函数成员指针 …… 520
20.6.2 数据成员指针 …… 522
20.6.3 基类和派生类成员 …… 523
20.7 建议 …… 523

第 21 章 类层次 …… 524

21.1 引言 …… 524
21.2 设计类层次 …… 524
21.2.1 实现继承 …… 525
21.2.2 接口继承 …… 527
21.2.3 替代实现方式 …… 529
21.2.4 定位对象创建 …… 532
21.3 多重继承 …… 533
21.3.1 多重接口 …… 533
21.3.2 多重实现类 …… 533
21.3.3 二义性解析 …… 535
21.3.4 重复使用基类 …… 538
21.3.5 虚基类 …… 539
21.3.6 重复基类与虚基类 …… 544
21.4 建议 …… 546

第 22 章 运行时类型信息 …… 547

22.1 引言 …… 547
22.2 类层次导航 …… 547
22.2.1 dynamic_cast …… 548
22.2.2 多重继承 …… 551
22.2.3 static_cast 和 dynamic_cast … 552
22.2.4 恢复接口 …… 553
22.3 双重分发和访客 …… 557
22.3.1 双重分发 …… 557
22.3.2 访客 …… 559
22.4 构造和析构 …… 561
22.5 类型识别 …… 561
22.5.1 扩展类型信息 …… 563
22.6 RTTI 的使用和误用 …… 564
22.7 建议 …… 565

第 23 章 模板 …… 566

23.1 引言和概述 …… 566
23.2 一个简单的字符串模板 …… 568
23.2.1 定义模板 …… 569
23.2.2 模板实例化 …… 571
23.3 类型检查 …… 571
23.3.1 类型等价 …… 572
23.3.2 错误检测 …… 573
23.4 类模板成员 …… 574
23.4.1 数据成员 …… 574
23.4.2 成员函数 …… 575
23.4.3 成员类型别名 …… 575
23.4.4 static 成员 …… 575
23.4.5 成员类型 …… 576
23.4.6 成员模板 …… 577
23.4.7 友元 …… 580
23.5 函数模板 …… 582
23.5.1 函数模板实参 …… 583
23.5.2 函数模板实参推断 …… 584
23.5.3 函数模板重载 …… 586
23.6 模板别名 …… 590
23.7 源码组织 …… 591
23.7.1 链接 …… 593
23.8 建议 …… 594

第 24 章 泛型程序设计 …… 595

24.1 引言 …… 595
24.2 算法和提升 …… 596
24.3 概念 …… 599
24.3.1 发现概念 …… 599
24.3.2 概念和约束 …… 602
24.4 具体化概念 …… 604
24.4.1 公理 …… 607
24.4.2 多实参概念 …… 607

24.4.3 值概念 …… 608
24.4.4 约束检查 …… 609
24.4.5 模板定义检查 …… 610
24.5 建议 …… 612

第 25 章 特例化 …… 613

25.1 引言 …… 613
25.2 模板参数和实参 …… 614
25.2.1 类型作为实参 …… 614
25.2.2 值作为实参 …… 615
25.2.3 操作作为实参 …… 616
25.2.4 模板作为实参 …… 618
25.2.5 默认模板实参 …… 619
25.3 特例化 …… 621
25.3.1 接口特例化 …… 623
25.3.2 主模板 …… 624
25.3.3 特例化顺序 …… 625
25.3.4 函数模板特例化 …… 626
25.4 建议 …… 628

第 26 章 实例化 …… 629

26.1 引言 …… 629
26.2 模板实例化 …… 630
26.2.1 何时需要实例化 …… 630
26.2.2 手工控制实例化 …… 631
26.3 名字绑定 …… 632
26.3.1 依赖性名字 …… 633
26.3.2 定义点绑定 …… 635
26.3.3 实例化点绑定 …… 636
26.3.4 多实例化点 …… 638
26.3.5 模板和名字空间 …… 639
26.3.6 过于激进的 ADL …… 639
26.3.7 来自基类的名字 …… 641
26.4 建议 …… 643

第 27 章 模板和类层次 …… 645

27.1 引言 …… 645
27.2 参数化和类层次 …… 646
27.2.1 生成类型 …… 647
27.2.2 模板类型转换 …… 649
27.3 类模板层次 …… 650
27.3.1 模板作为接口 …… 651
27.4 模板参数作为基类 …… 652
27.4.1 组合数据结构 …… 652
27.4.2 线性化类层次 …… 655
27.5 建议 …… 660

第 28 章 元编程 …… 661

28.1 引言 …… 661
28.2 类型函数 …… 663
28.2.1 类型别名 …… 665
28.2.2 类型谓词 …… 666
28.2.3 选择函数 …… 668
28.2.4 萃取 …… 668
28.3 控制结构 …… 670
28.3.1 选择 …… 670
28.3.2 迭代和递归 …… 673
28.3.3 何时使用元编程 …… 674
28.4 条件定义：Enable_if …… 675
28.4.1 使用 Enable_if …… 676
28.4.2 实现 Enable_if …… 678
28.4.3 Enable_if 与概念 …… 678
28.4.4 更多 Enable_if 例子 …… 679
28.5 一个编译时列表：Tuple …… 681
28.5.1 一个简单的输出函数 …… 683
28.5.2 元素访问 …… 684
28.5.3 make_tuple …… 686
28.6 可变参数模板 …… 686
28.6.1 一个类型安全的 printf() …… 687
28.6.2 技术细节 …… 689
28.6.3 转发 …… 691
28.6.4 标准库 tuple …… 692
28.7 国际标准单位例子 …… 694
28.7.1 Unit …… 695
28.7.2 Quantity …… 696
28.7.3 Unit 字面值常量 …… 697
28.7.4 工具函数 …… 698
28.8 建议 …… 700

第 29 章　一个矩阵设计 …… 701
29.1　引言 …… 701
29.1.1　Matrix 的基本使用 …… 701
29.1.2　对 Matrix 的要求 …… 703
29.2　Matrix 模板 …… 704
29.2.1　构造和赋值 …… 705
29.2.2　下标和切片 …… 706
29.3　Matrix 算术运算 …… 708
29.3.1　标量运算 …… 709
29.3.2　加法 …… 710
29.3.3　乘法 …… 711
29.4　Matrix 实现 …… 712
29.4.1　slice() …… 713
29.4.2　Matrix 切片 …… 713
29.4.3　Matrix_ref …… 714
29.4.4　Matrix 列表初始化 …… 715
29.4.5　Matrix 访问 …… 717
29.4.6　零维 Matrix …… 719
29.5　求解线性方程组 …… 720
29.5.1　经典高斯消去法 …… 721
29.5.2　旋转 …… 722
29.5.3　测试 …… 723
29.5.4　综合运算 …… 723
29.6　建议 …… 725

第一部分

The C++ Programming Language, Fourth Edition

引　言

这一部分会概述C++语言及其标准库的主要概念和特性。还会介绍本书的主要内容并解释本书对语言特性及其使用的描述方式。此外，本部分还会介绍有关C++的设计以及C++的使用的一些背景知识。

"……而你，马库斯，已经给了我很多；现在我要给你一个忠告：尝试不同的生活，放弃墓守马库斯·科科扎的游戏。你总是过分担心马库斯·科科扎，以至于变成了自己的奴隶和囚犯。你做任何事之前都要考虑会不会影响马库斯·科科扎的幸福和声望。你一直过分害怕马库斯可能做蠢事，或是感到厌烦。但这真的重要吗？世上所有人都会做蠢事……我希望你放轻松，希望你的小心脏再次被点燃。从现在开始，你必须尝试更多彩的生活，和你能想象的一样多彩……"

——卡伦·布里克森

《七个神奇的故事》中的"梦想家"一章（1934）

第 1 章

The C++ Programming Language, Fourth Edition

致 读 者

欲速则不达。

——屋大维，恺撒·奥古斯都

- 本书结构
  引言；基本特性；抽象机制；标准库；例子和参考文献
- C++ 的设计
  程序设计风格；类型检查；C 兼容性；语言、库和系统
- 学习 C++
  用 C++ 编程；对 C++ 程序员的建议；对 C 程序员的建议；对 Java 程序员的建议
- C++ 的历史
  大事年表；早期的 C++；1998 标准；2011 标准；C++ 的用途
- 建议
- 参考文献

1.1 本书结构

纯粹的入门教材通常会这样组织其内容——所有概念都会先介绍再应用，因此必须从第一页开始顺序阅读。与之相反，纯粹的参考手册则可以从任何地方开始查阅，因为每个主题的描述都简明扼要，辅以指向相关主题的（向前或向后）引用。阅读一本纯粹的入门教材原则上不需要任何预备知识，因为教材中会仔细地描述所有内容。而纯粹的参考手册则只适合那些已经熟悉所有基本概念和技术的人使用。本书兼具这两类书的特点。如果你已经了解大多数概念和技术，就可以按需要只阅读特定章甚至特定节。如果你不了解这些基础知识，则可以从头开始阅读，但注意不要陷入细节中。要善用索引和交叉引用。

令一本书的各部分形成一定程度的自包含意味着要有一些重复内容，这些重复内容也起到让顺序阅读的读者进行回顾的作用。本书包含大量的交叉引用，不仅引用本书内容还引用 ISO C++ 标准库。有经验的程序员可以阅读 C++（相对的）快速“指南”来了解本书的概貌，以便将本书作为参考手册使用。本书包含以下四个部分。

第一部分　第 1 章（本章）是本书的导引，会介绍一点 C++ 的背景知识。第 2 ~ 5 章对 C++ 语言及其标准库进行简要介绍。

第二部分　第 6 ~ 15 章介绍 C++ 的内置类型和基本特性以及如何用它们构造程序。

第三部分　第 16 ~ 29 章介绍 C++ 的抽象机制及如何用这些机制编写面向对象和泛型程序。

第四部分　第 30 ~ 44 章概述标准库并讨论一些兼容性问题。

1.1.1 引言

本章会概述本书的结构和内容，给出使用本书的一些提示，并介绍一些有关 C++ 语

言及其使用的背景知识。建议快速浏览本章，只阅读那些看起来有意思的部分，然后在学习了本书其他部分后再回到本章。请不要误认为必须仔细阅读本章然后才能继续阅读。

接下来的几章将简要介绍 C++ 程序设计语言及其标准库的主要概念和特性。

第 2 章 C++ 概览：基础知识。介绍 C++ 的内存模型、计算模型和错误处理模型。

第 3 章 C++ 概览：抽象机制。介绍用来支持数据抽象、面向对象编程以及泛型编程的语言特性。

第 4 章 C++ 概览：容器与算法。介绍标准库提供的字符串、简单 I/O、容器和算法等特性。

第 5 章 C++ 概览：并发与实用功能。概述与资源管理、并发、数学计算、正则表达式以及其他一些方面相关的标准库工具。

这几章对 C++ 特性的概览是想让读者初步领略 C++ 都提供了哪些功能。特别是让读者看到，从本书第 1 版出版到第 2 版以及第 3 版出版到现在，C++ 已经取得了巨大的进展。

1.1.2 基本特性

C++ 支持传统的 C 语言编程风格（也被其他一些相似的语言所采用）。第二部分重点介绍支持 C 编程风格的 C++ 子集，包括类型、对象、作用域和存储的基本概念，计算的基础（表达式、语句和函数），以及支撑模块化的特性——名字空间、源文件和异常处理。

第 6 章 类型与声明。基础类型、命名、作用域、初始化、简单类型推断、对象生命周期和类型别名。

第 7 章 指针、数组与引用。

第 8 章 结构、联合与枚举。

第 9 章 语句。声明语句、选择语句（if 和 switch）、迭代语句（for、while 和 do）、goto 语句和注释语句。

第 10 章 表达式。桌面计算器例子、运算符、常量表达式和隐式类型转换。

第 11 章 选择适当的操作。逻辑运算符、条件表达式、递增和递减、自由空间（new 和 delete）、{} 列表、lambda 表达式和显式类型转换（static_cast 和 const_cast）。

第 12 章 函数。函数声明和定义、inline 函数、constexpr 函数、实参传递、重载函数、前置和后置条件、函数的指针和宏。

第 13 章 异常处理。错误处理风格、异常保证、资源管理、强制不变量、throw 和 catch、一个 vector 的实现。

第 14 章 名字空间。namespace、模块化和接口；使用名字空间组织代码。

第 15 章 源文件与程序。分离编译、链接、使用头文件及程序启动和结束。

我假定读者熟悉第一部分中用到的大多数程序设计概念。例如，我会解释用于表达递归和循环的 C++ 特性，但我不会深入讨论技术细节或是花很多时间解释递归和循环如何有用。

唯一的例外是异常处理。很多程序员缺乏异常处理的经验，或者有限的经验都来自资源管理和异常处理机制不完整的语言（例如 Java）。因此，异常处理一章（第 13 章）会介绍 C++ 异常处理和资源管理的基本理念，并深入讨论一些技术细节，重点介绍“资源获取即初始化”（Resource Acquisition Is Initialization，RAII）技术。

1.1.3　抽象机制

第三部分介绍的C++特性用来支持不同形式的抽象，包括面向对象编程和泛型编程。所有章节可以粗略分为三组：类、类继承和模板。

前四章集中讨论类机制。

第16章　类。用户自定义类型，也就是类的概念，是所有C++抽象机制的基础。

第17章　构造、清理、拷贝和移动。展示了程序员如何定义类对象创建和初始化操作的含义。此外，拷贝、移动和析构的含义同样可由程序员来定义。

第18章　运算符重载。介绍了为用户自定义类型指定运算符含义的规则，重点介绍常用的算术和逻辑运算符，例如+、*和&等。

第19章　特殊运算符。讨论用户自定义的非算术运算符的使用，例如，用于下标的[]、用于函数对象的()和用于“智能指针”的->。

类可以按层次化组织。

第20章　派生类。介绍构建类层次的基本语言特性及其基本使用方法。我们可以实现接口（抽象类）与其实现（派生类）的完全分离，两者间的联系通过虚函数提供。本章还会介绍C++访问控制模型（public、protected和private）。

第21章　类层次。讨论有效地使用类层次的方法。本章还会介绍多重继承的概念，即一个类有多个直接基类。

第22章　运行时类型信息。介绍如何使用存储在对象中的数据实现在类层次中导航。我们可以使用dynamic_cast查询一个基类对象是否是作为派生类对象定义的，还可以用typeid获得一个对象的最基本信息（例如它的类名）。

那些最灵活、最高效、最有用的抽象，很多都是用其他类型（类）和算法（函数）对某类型（类）和算法（函数）进行参数化。

第23章　模板。介绍隐藏在模板及其使用方法之下的基本原理，还会介绍类模板、函数模板和模板别名。

第24章　泛型程序设计。介绍设计泛型程序所需的基本技术。其中，核心是从很多具体代码示例中提升（lift）抽象算法的技术，而概念（concept）则指明了一个泛型算法对其实参的要求。

第25章　特例化。介绍特例化（specialization）技术，即如何利用给定的一组模板参数，从模板生成类和函数。

第26章　实例化。主要介绍名字绑定规则。

第27章　模板和类层次。介绍模板层次和类层次如何结合使用。

第28章　元编程。介绍如何用模板生成程序。模板提供了一种生成代码的图灵完备机制。

第29章　一个矩阵设计。给出了一个稍大的例子，展示怎样组合使用已经学到的语言特性来解决复杂的设计问题：设计一个N维矩阵，几乎支持任意元素类型。

在第三部分中，我都是给出抽象技术的相关介绍，然后再描述支持这些技术的语言特性。这是与第二部分的明显不同之处，在这里我不再假定读者已经了解所介绍的技术。

1.1.4 标准库

介绍标准库的章节比介绍语言特性的章节“教学味”更少。你可以按照任意顺序阅读这一部分，实际上这一部分可以当作标准库组件的用户手册来使用。

第 30 章 标准库概览。给出标准库的概览，列出标准库头文件，并介绍语言支持和程序诊断方面的支持，如 exception 和 system_error。

第 31 章 STL 容器。介绍迭代器、容器和算法框架（称为标准模板库，STL）中的容器，包括 vector、map 和 unordered_set。

第 32 章 STL 算法。介绍 STL 中的算法，包括 find()、sort() 和 merge()。

第 33 章 STL 迭代器。介绍 STL 中的迭代器和其他工具，包括 reverse_iterator、move_iterator 和 function。

第 34 章 内存和资源。介绍内存和资源管理相关的工具组件，如 array、bitset、pair、tuple、unique_ptr、shared_ptr、分配器和垃圾收集接口。

第 35 章 工具。介绍一些重要性稍低的工具组件，如时间工具、类型特征以及多种类型函数。

第 36 章 字符串。介绍标准库 string，包括字符特征——它是使用不同字符集的基础。

第 37 章 正则表达式。介绍正则表达式语法和使用它进行字符串匹配的不同方法，包括用 regex_match() 匹配一个完整的字符串，用 regex_search() 在一个字符串中查找一个模式，用 regex_replace() 进行简单替换，以及用 regex_iterator 对一个字符流进行遍历。

第 38 章 I/O 流。介绍标准库 I/O 流，包括格式化和非格式化输入输出、错误处理以及缓冲。

第 39 章 区域设置。介绍类 locale 和它的各种 facet，这些 facet 提供了对区域设置功能的支持，包括处理字符集中的文化差异、格式化数值、格式化日期和时间及其他很多功能。

第 40 章 数值计算。介绍用于数值计算的标准库工具（如 complex、valarray、随机数和通用数值算法）。

第 41 章 并发。介绍 C++ 基本内存模型和 C++ 所提供的支持无锁并发编程的工具。

第 42 章 线程和任务。介绍支持“线程和锁风格”并发编程的类（如 thread、timed_mutex、lock_guard 和 try_lock()）和支持基于任务的并发编程模式的类（如 future 和 async()）。

第 43 章 C 标准库。介绍纳入 C++ 标准库的 C 标准库特性（包括 printf() 和 clock()）。

第 44 章 兼容性。讨论 C 和 C++ 的关系以及标准 C++（也称为 ISO C++）与早期 C++ 版本的关系。

1.1.5 例子和参考文献

本书的重点是介绍程序组织而非算法设计，因此我避开了那些巧妙的或难理解的算法。平凡的算法通常更适合阐述语言特性或是程序结构中的某个点。例如，我可能选择用希尔（Shell）排序来介绍语言特性，但在实际代码中，采用快速排序可能更好。通常，用更适合的算法重新实现程序的工作会留作练习。在实际代码中，调用库函数的方式通常比书中用来

阐述语言特性的代码更好。

教材中的程序示例带给学生的关于软件开发的观念必然是不正确的——由于示例程序都经过净化和简化，程序规模所带来的复杂性就荡然无存了。为了获得对程序设计和程序设计语言的正确观念，编写实际规模的程序仍然是唯一途径。本书关注语言特性和标准库工具，它们是构造所有程序的基础，我会详细阐述利用它们构造程序的原则和技术。

本书所选择的例子反映了我在编译器、基础库和仿真领域的背景，一些重点例子则反映了我对系统编程的兴趣。所有例子都是真实代码的简化版本。简化是必要的，以免编程语言和设计要点的介绍迷失在细节中。我心目中理想的例子应该在阐述清楚一个设计原理、一个程序设计技术、一个语言结构或是一个标准库特性的基本要求下，做到最短、最清晰。本书中没有凭空造出的“精巧”例子。对于那些纯粹的语言技术性的例子，我会将变量命名为 x 和 y，将类型命名为 A 和 B，将函数命名为 f() 和 g()。

只要可能，我都会结合使用场景来介绍 C++ 语言和标准库特性，而不是以干巴巴的手册方式给出。本书中所介绍的语言特性和描述它们的细节内容，大致反映了我对“如何高效使用 C++”这一问题的观点。介绍这些内容是为了让你了解如何使用一个语言特性，这种使用通常不是孤立的，而是与其他特性相结合的。对于写出好程序的目标而言，了解一个语言特性或标准库组件的所有语言技术性细节是不必要的，也是不够的。实际上，痴迷于了解每个小细节只会导致过分精致、过分聪明的糟糕代码。为了写出好的程序，真正需要的是对设计和编程技术的理解以及对应用领域的了解。

我假定你可以访问网络资源。语言和标准库规则的最终依据是 ISO C++ 标准［C++, 2011］，你可以在互联网上找到它。

本书中有很多对书中其他部分的引用，它们遵循 2.3.4 节（第 2 章，第 3 节，第 4 小节）和 iso.5.3.1 节（ISO C++ 标准的 5.3.1 节）这样的格式。我会有节制地使用楷体来强调某些内容（如，“不接受一个字符串字面值常量”），对第一次出现的重要概念（如，多态（polymorphism））我也使用楷体。

为了环保（节约纸张）以及简化附加内容，我将本书的数百道习题放在网络上，请在 www.stroustrup.com 中查阅。

本书中使用的语言和库是 C++ 标准［C++, 2011］所定义的“纯粹 C++”。因此，书中代码示例在所有最新的 C++ 实现中应该都能运行。书中的主要程序片段都已在多个 C++ 实现上进行了实验，那些使用了新特性的代码在某些编译器上会编译失败。但我认为指出某某编译器不能编译某某例子没有什么意义，这些信息很快就会过时，因为编译器设计者都在努力工作以确保他们的编译器能正确支持所有 C++ 特性。第 44 章对如何应对旧版本 C++ 编译器以及如何处理 C 代码提出了一些建议。

当我发现在某个地方 C++11 特性最适合时，我就会使用 C++11 特性。例如，我倾向于使用 {} 风格的初始化方式以及使用 using 定义类型别名。有时，这些用法可能会让“老程序员”惊讶。但是，惊讶通常是促使你开始学习新知识的很好的诱因。另一方面，我不会仅仅因为一个特性是新特性就不加分辨地使用它。对于语言特性，我理想中的使用方式是能最具体地表达基本思想，并且能很好地使用那些在 C++ 甚至 C 中已经使用了很长时间的东西。

显然，如果你不得不使用旧版本的编译器（比方说，由于你的一些客户还未升级到支持最新标准的编译器），就必须避开新特性。但是，不要因为旧特性是成熟的而且是你所熟悉

的就认为“走老路”更好而且更简单。44.2 节概述了 C++98 标准和 C++11 标准间的差异。

1.2 C++ 的设计

程序设计语言的目的就是帮助我们用代码来表达思想。因此，一种程序设计语言要完成两个相关的任务：为程序员提供一个工具，用来指明需要由计算机执行什么动作；为程序员提供一组概念，用于思考能做些什么。对于第一个目标，理想情况是语言更“靠近机器”，使得程序员能很容易地找到方法来简单高效地处理计算机所有重要的方面。C 语言最初就是出于这种考虑而设计的。第二个目标理想情况下要求语言更“接近待求解的问题”，这样就能直接而具体地表达问题求解方案的概念。在创造 C++ 时向 C 添加的那些特性，如函数实参检查、const、类、构造函数和析构函数、异常及模板，就是从这个角度考虑而设计的。因此，C++ 的设计理念是同时提供

- 将内置操作和内置类型直接映射到硬件，从而提供高效的内存利用和高效的底层操作；
- 灵活且低开销的抽象机制，使得用户自定义类型无论是符号表达、使用范围还是性能都能与内置类型相当。

最初，通过将源自 Simula 语言的思想应用到 C 语言中，C++ 实现了这一理念。多年来，这些简单思想的进一步应用催生了更为通用、高效且灵活的语言特性集合。这些语言特性支持多种程序设计风格的综合，从而同时实现高效率（efficient）和优雅风格（elegant）。

C++ 的设计一直都重点关注那些处理基本概念的程序设计技术，这些基本概念包括内存、易变性、抽象、资源管理、算法的表达、错误处理及模块化。这些都是一个系统程序员最为关注的问题，也是资源受限系统和高性能系统程序员普遍关注的问题。

通过定义类库、类层次和模板，你可以在比本书中展示的更高的抽象层次上编写 C++ 程序。例如，C++ 广泛应用于金融系统、游戏开发以及科学计算（见 1.4.5 节）。为了使高级应用的编程更加高效和方便，我们需要库。如果只能使用语言的内置特性，几乎所有编程工作都会很痛苦，所有通用编程语言都是如此。相反，只要有了合适的库，几乎任何编程工作都可以是很愉悦的。

我介绍 C++ 的标准方式通常像下面这样开始：

- C++ 是一种通用程序设计语言，偏重于系统程序设计。

这一描述现在仍然是正确的。这么多年来的变化是 C++ 抽象机制的重要性、能力和灵活性在不断提高：

- C++ 是一种通用程序设计语言，它提供了直接且高效的硬件模型，并结合了定义轻量级抽象的工具。

或者更精练地表达为：

- C++ 是一种用来开发和使用优雅而高效的抽象的程序设计语言。

“通用程序设计语言”想表达的意思是，C++ 的应用范围很广。而它确实已经用于非常广泛的场景之中（从微控制器到大型分布式商用系统），但关键在于，C++ 并不是为了任何一个特定的应用领域而专门设计的。任何程序设计语言都不可能完美地适合所有应用领域和所有程序员，但 C++ 的设计理念是更好地支持尽可能多的应用领域。

系统程序设计（system programming）的含义是编写直接使用硬件资源的、严重受限于资源的代码，或是编写的代码与这类代码联系紧密。特别是软件基础设施的实现（如设备驱

动程序、通信协议栈、虚拟机、操作系统、业务支持系统、编程环境以及基础库）大部分都属于系统程序设计。长期以来，我对 C++ 的描述都会加上“偏重于系统程序设计”，这是很重要的，C++ 从来没有因为希望更适合于其他应用领域就做出妥协，就简化掉那些支持对硬件和系统资源进行专家级使用的语言特性。

当然，你在编程时也可以完全隐藏硬件细节，使用代价更高的抽象机制（例如，每个对象都从自由存储区分配空间，每个操作都设计为虚函数），使用不优雅的风格（例如，过度抽象），你也可以根本不使用抽象（“荣耀的汇编代码”）。但是，很多程序设计语言都能做到这些，因此这些并不是 C++ 的特征。

《C++ 语言的设计和演化》一书［Stroustrup，1994］（大家所熟知的 D&E）更为详细地概括了 C++ 的理念和设计目标，其中有两个基本原则是最重要的。

- 不给比 C++ 更底层的语言留任何余地（在极少的情况下汇编语言是例外）。因为，如果你能用一种更底层的语言编写出更高效的代码，那意味着这种语言很可能比 C++ 更适合系统程序设计。
- 你不使用它，就不要为它付出代价。如果程序员能够手工编写出很不错的代码，来模拟一个语言特性或是一个基础的抽象机制，甚至性能更好一些，那么一些人就真的会去编写这种代码，而很多人就会效仿。因此，与等价的替代方法相比，我们设计的语言特性或是基础的抽象机制必须不浪费哪怕一个字节或是一个处理器时钟周期。这就是众所周知的零开销原则（zero-overhead principle）。

这两个原则很苛刻，但在某些（显然不是全部）场景下是必要的。特别是零开销原则不断地引导 C++ 变得更简单、更优雅，并催生出比最初预期更为强大的语言特性。STL 就是一个例子（见 4.1.1 节、4.4 节、4.5 节、第 31 – 33 章）。在人们不断努力提高程序设计水平的过程中，这两个原则已被证明是非常重要的。

1.2.1 程序设计风格

现有的语言特性为程序设计风格提供了支持。请不要将单个语言特性作为解决方案来看待，而应将其看作一个多变的特性集合中的基本单元，我们可以组合多个特性来表达解决方案。

我们可以简单描述软件设计和编程的基本理念：

- 用代码直接表达想法。
- 无关的想法应独立表达。
- 用代码直接描述想法之间的关联。
- 可以自由地组合用代码表达的想法，但仅在这种组合有意义时。
- 简单的想法应简单表达。

这些理念已被很多人分享，但支持这些理念的语言在设计上可能天差地别。根本原因是，一种编程语言包含了很多工程上的折中，这些折中反映了多种多样的个体和社区的不同需求、审美以及历史。对于设计上的一些普遍性挑战，C++ 有着自己的答案，这些答案的形成归结于 C++ 系统程序设计的起源（可以追溯到 C 和 BCPL[Richards，1980]），通过抽象解决程序复杂性问题的目标（可以追溯到 Simula），以及它的历史。

C++ 语言特性直接支持四种程序设计风格：

- 过程式程序设计；

- 数据抽象；
- 面向对象程序设计；
- 泛型程序设计。

但是，重点不在于对单个程序设计风格的支持，而在于有效地组合它们。对于大多数非平凡的问题而言，最好的（最易维护的、最易读的、最小的、最快的，等等）解决方案通常是这些风格某些方面的组合。

就像计算机领域里的很多重要术语一样，这些术语也有五花八门的叫法流行于计算机业界和学术界的不同领域。例如，我称为“程序设计风格”，其他人可能称之为“程序设计技术”或“范型”。而我更喜欢用“程序设计技术”表示那些特定语言相关的更具体的内容。对于“范型”一词，由于其自命不凡以及（始自 Kuhn 最初定义的）排他性暗示，我感到很不舒服。

我理想中的语言特性应该能优雅地组合使用，来支持连续统一的程序设计风格和各种各样的程序设计技术。

- 过程式程序设计：这种风格专注于处理和设计恰当的数据结构。支持这种风格也是 C 语言（以及 Algol、Fortran 和很多其他语言）的设计目标。C++ 对这种风格的支持体现为内置类型、运算符、语句、函数、struct 和 union 等特性。除少数例外，C 可以看作 C++ 的子集。与 C 相比，C++ 对过程式程序设计的支持更强，这体现在很多额外的语言特性和一个更严格、更灵活且对过程式编程支持更好的类型系统。
- 数据抽象：这种风格专注于接口的设计以及一般实现细节的隐藏和特殊的表示方式。C++ 支持具体类和抽象类。一些语言特性可直接用来定义具有私有实现细节、构造函数和析构函数以及相关操作的类。而抽象类则为完全的数据隐藏提供了直接支持。
- 面向对象程序设计：这种风格专注于类层次的设计、实现和使用。除了允许定义类框架之外，C++ 还提供了各种各样的特性来支持类框架中的导航以及简化由已有的类来定义新的类。类层次提供了运行时多态（见 20.3.2 节、21.2 节）和封装（见 20.4 节、20.5 节）机制。
- 泛型程序设计：这种风格专注于通用算法的设计、实现和使用。在这里，“通用”的含义是，一个算法可以设计成能处理多种类型，只要这些类型满足算法对其实参的要求即可。C++ 支持泛型编程的主要特性是模板。模板提供了（运行时）参数多态。

几乎任何可以提高类的灵活性或效率的语言特性都会增强对这些程序设计风格的支持。因此，C++ 可以（而且已经）被称为面向类（class oriented）的程序设计语言。

上述这些设计和编程风格的强大在于它们的综合，每种风格都对综合起到了重要作用，而这种综合实际上就是 C++。因此，只关注一种风格是错误的；除非你只编写一些玩具程序，否则只关注一种风格会导致开发工作的浪费，产生非最优的（不灵活的、冗长的、性能低下的、不易维护的，等等）程序。

有两种观点我非常不赞同：一是将 C++ 描绘为只适合上述风格中的一种（例如，“C++ 是一种面向对象语言”）；或是用某种术语（例如，“混合的”或“混合范型”）来暗示某种局限性更强的语言更好。前一种观点的问题是忽略了这样一个事实：上述所有程序设计风格都对综合有某些方面的重要贡献。后一种观点则否定了风格综合的有效性。上述这些风格并非可相互替代的不同选择：每种风格都为表达力更强、效率更高的程序设计风格贡献了重要的技术，而 C++ 则为这些风格的组合使用提供了直接支持。

自诞生之初，C++ 的设计目标就是多种编程和设计风格的综合。即使在最早的 C++ 著作［Stroustrup，1982］中，也给出了组合使用这些不同风格的例子以及支持这种组合的语言特性：

- 类支持上述所有风格；这依赖于用户如何将想法表示为用户自定义类型或是自定义类型的对象。
- 公有 / 私有访问控制支持数据抽象和面向对象程序设计——清晰地分离接口和实现。
- 成员函数、构造函数、析构函数以及用户自定义赋值运算符为对象提供了一个清晰的功能接口，这是数据抽象和面向对象程序设计所需要的。这些特性还提供了一种统一的符号表示，这是泛型编程所需要的。更一般的重载机制直到 1984 年才产生，而一致初始化机制直到 2010 年才出现。
- 函数声明为成员函数和独立函数提供了特殊的具备静态检查的接口，因此支持上述所有程序设计风格，而这也是重载所需要的。当时，C 还没有“函数原型”，但 Simula 已经有了函数声明和成员函数。
- 泛型函数和参数化类型（当时是使用宏从函数和类生成的）支持泛型程序设计。模板直到 1988 年才产生。
- 基类和派生类为面向对象程序设计和某些形式的数据抽象提供了基础。虚函数直到 1983 年才产生。
- 内联使得在进行系统程序设计以及构造运行时间和空间都很高效的库时，使用上述语言特性的代价可以接受。

这些早期的 C++ 特性着眼于提供通用的抽象机制，而非支持不相关的程序设计风格。当今的 C++ 对基于轻量级抽象的设计和编程提供了好得多的支持，但支持编写优雅而高效的代码这一目标从诞生之初一直延续至今。1981 年以来 C++ 的发展，使得这些最初就已关注的程序设计风格（范式）的综合得到了更好的支持，并且极大地提高了综合的效率。

C++ 中的基本对象具有唯一的身份，即它们位于内存中的特定位置，而且可以通过比较地址与（可能）具有相同值的其他对象区分开来。表示这种对象的表达式被称为*左值*（lvalue，见 6.4 节）。但早在 C++ 的祖先［Barron，1963］所在的年代，就已经有了没有身份的对象（对于这类对象，不存在一个安全存储的地址可供随后使用）。在 C++11 中，这一*右值*（rvalue）的概念发展为一个新的概念——不能以低开销进行移动的值（见 3.3.2 节、6.4.1 节、7.7.2 节）。以这种对象为基础的技术很像函数式程序设计中所用的技术（在函数式程序设计中，有身份的对象的概念是令人反感的）。这一新概念对泛型程序设计技术和语言特性（如 lambda 表达式）是很好的补充。它还很好地解决了与“简单抽象数据类型”相关的一些问题，例如，如何从一个操作（如矩阵 +）优雅而高效地返回一个大矩阵。

从很早开始，C++ 程序以及 C++ 本身的设计就已经开始关注资源管理了。理想的资源管理（到现在仍）是这样的：

- 简单（对实现者，特别是使用者而言）；
- 通用（资源可以是任何须从某处进行申请并稍后释放的东西）；
- 高效（服从零开销原则，见 1.2 节）；
- 完善（任何资源泄漏都是不可接受的）；
- 静态类型安全。

很多重要的 C++ 类，例如标准库中的 vector、string、thread、mutex、unique_ptr、fstream

和 regex，都是用来处理资源的。标准库之外的基础库和应用库也提供了很多例子，例如 Matrix 和 Widget。支持资源处理概念的第一步，是“带类的 C”草案中就已有的构造函数和析构函数。随后很快就出现了拷贝控制特性，允许用户自定义赋值运算符和拷贝构造函数。C++11 引入的移动构造函数和移动赋值运算符（见 3.3 节）完善了这一思路，它们允许在作用域之间（见 3.3.2 节）以低代价移动大对象以及简单地控制多态或共享对象的生命周期（见 5.2.1 节）。

支持资源管理的语言特性也能使不处理资源的抽象受益。任何建立并维护不变量的类都依赖于这些特性的一个子集。

1.2.2 类型检查

我们用来思考/编程的语言与我们能够想象的问题/解决方案间的联系是非常紧密的。为此，以消除程序员的错误为目的限制语言特性是无意义的，最好情况也只是一种危险的理念。一种语言为程序员提供了一组概念性的工具；如果这些工具不足以完成一项任务，程序员就会忽略它们。因此，仅仅靠增加或减少特定语言特性是不能保证程序员不犯错误、创造出好的设计的。不过，C++ 还是提供了一些语言特性和一个类型系统来帮助程序员准确而简洁地用代码表达设计。

静态类型和编译时类型检查的概念对高效使用 C++ 是极为重要的。静态类型的使用是可表达性、可维护性和性能的关键。在 C++ 中，用户自定义类型需要有完整的接口，在编译时进行类型检查，这借鉴自 Simula，是 C++ 可表达性的关键。C++ 的类型系统是可扩展的，但并不简单（见第 3 章、第 16 章、第 18 章、第 19 章、第 21 章、第 23 章、第 28 章、第 29 章），其目标是对内置类型和用户自定义类型提供同等的支持。

C++ 的类型检查和数据隐藏特性依赖编译时的程序分析来防止数据的意外损坏。但对于故意破坏规则的人，它们无法提供数据保密或保护，即：C++ 能防止意外，但不能防止欺骗。不过，我们可以随意使用这些特性，而不会产生运行时间和空间上的额外开销。其中蕴含的设计思想是，为了成为有用的语言特性，不仅要优雅，还必须在程序的实际运行环境中有较低的运行开销。

C++ 的静态类型系统很灵活，而且效率很高——简单用户自定义类型即使有额外使用开销的话，也很小。其目标是支持这样一种程序设计风格：将不同的想法表示为不同类型，如整数、浮点数、字符串、“原始内存”和“对象”，而不是到处使用泛型。一个类型丰富的程序设计风格能使代码更加易读、易维护、易分析。一个简单的类型系统只能进行简单的分析，而一个类型丰富的程序设计风格则为复杂的错误检测和优化提供了可能。C++ 编译器和开发工具支持这种基于类型的分析［Stroustrup，2012］。

C++ 保留了大多数 C 语言特性作为子集，并且保留了语言特性到硬件的直接映射，这是大多数要求较高的低层系统程序设计任务所需要的，这些特性的保留意味着会打破静态类型系统。但我们的理想（一直）是完整的类型安全。在此，我同意 Dennis Ritchie 的观点：“C 是一种强类型、但弱检查的语言。”注意，Simula 既是类型安全的，也很灵活。实际上，当我开始设计 C++ 时，我的理想是“带类的 Algol68”，而不是“带类的 C”。但是，有很多充分的理由阻止我将工作建立在类型安全的 Algol68［Woodward，1974］的基础上，这些理由可以列出一个长长的令人痛苦的列表。因此，完美的类型安全只是一个理想，C++ 作为一种编程语言，只能接近而难以达到这一理想。但 C++ 程序员（特别是库的编写者）应该向着

这一理想而努力。多年以来，支持这一理念的语言特性集合、标准库组件以及相关技术在不断发展。现在，在低层代码（有希望用类型安全的接口加以隔离），遵守不同语言规范的代码的接口代码（如操作系统调用的接口），以及基础抽象的实现（例如，string 和 vector）之外，已经很少需要类型不安全的代码了。

1.2.3 C 兼容性

C++ 从 C 语言发展而来，它保留了 C 的特性（除了极少例外）作为子集。以 C 为基础的主要原因是，我希望 C++ 建立在一组久经考验的低层语言特性之上，并成为技术社区的一部分。这样，保持与 C 语言的高度兼容就变得非常重要［Koenig，1989］［Stroustrup，1994］（见第 44 章）；但不幸的是，这会阻碍对 C 语法的清理。C 和 C++ 不间断的或多或少并行的演化已经成为持续受到关注的源泉［Stroustrup，2002］。但是，由两个委员会致力保持两个广泛使用的语言“尽可能兼容”，并不是一种特别好的工作组织方式。特别是，对于兼容的价值、好的程序设计由什么构成以及好的程序设计需要什么样的支持，还存在意见分歧。仅仅通过保持两个委员会间的交流来消除分歧，工作量太大。

与 C 语言百分之百兼容从来也不是 C++ 的目标，因为这需要在类型安全及用户自定义类型与内置类型的同等地位等方面做出妥协。不过，C++ 的定义已经反复修订，来消除无谓的不兼容；因此，现在 C++ 与 C 的兼容性更胜当初。C++98 采纳了 C89 的很多细节（见 44.3.1 节）。当 C 接着从 C89［C，1990］演化到 C99［C，1999］时，C++ 采纳了几乎所有新的特性，只排除了可变长度数组（VLA）和指定初始化，前者是因为性能不佳，而后者是多余的。C 中用于低层系统程序设计任务的特性得以保留和加强；例如内联（见 3.2.1.1 节、12.1.5 节和 16.2.8 节）和 constexpr（见 2.2.3 节，10.4 节和 12.1.6 节）。

与之相对，现代 C 语言也采纳了（忠实原版和性能保持的程度不同）很多源自 C++ 的特性（如 const，函数原型和内联，见［Stroustrup，2002］）。

C++ 的定义已经经过反复修订，保证既符合 C 语法也符合 C++ 语法的程序结构在两种语言中具有相同的含义（见 44.3 节）。

C 语言的最初目标之一是代替汇编语言来进行大多数要求较高的系统程序设计任务。在 C++ 设计之初，我们就非常谨慎，确保这方面的优势不会打折。C 和 C++ 间的区别主要在于对类型和结构的强调程度不同。C 语言有很强的表达力，也很自由。而通过类型系统的深入使用，C++ 实现了更强的表达力，同时又没有损失性能。

掌握 C 语言并不是学习 C++ 的先决条件。很多 C 语言编程所鼓励的技术和技巧，在有了 C++ 语言特性后，就变得不再必要了。例如，相比于 C，C++ 更不需要显式类型转换（见 1.3.3 节）。但是，好的 C 程序往往很符合 C++ 程序的标准。例如，Kernighan 和 Ritchie 的《C 程序设计语言（第 2 版）》［Kernighan，1988］中的每个程序同时也都是一个 C++ 程序。有任何静态类型编程语言的经验对学习 C++ 都是有帮助的。

1.2.4 语言、库和系统

C++ 的基本（内置）类型、运算符和语句都是计算机硬件能直接处理的：数字、字符和地址。C++ 没有内置的高级数据类型，也没有高级操作原语。例如，C++ 语言不提供支持逆矩阵运算的矩阵类型，也不提供支持连接运算的字符串类型。如果用户需要，可以利用语言本身的特性来定义这些类型。实际上，定义一个新的通用类型或是特定应用类型是 C++ 最

基本的程序设计工作。一个精心设计的用户自定义类型与内置类型的区别仅仅在于定义的方式，而使用方式则是完全一样的。C++ 标准库（见第 4 章、第 5 章、第 30 章、第 31 章等）提供了很多这种类型及其使用的例子。从用户的角度来看，内置类型和标准库提供的类型几乎没有区别。除了历史遗留下的几个不幸的且不重要的事故，C++ 标准库都是用 C++ 语言编写的。用 C++ 语言来编写 C++ 标准库是对 C++ 类型系统和抽象机制的重要测试；对于大多数要求较高的系统程序设计任务，它们必须（也确实）足够强大（表达力强）且足够高效（代价可接受）。这确保它们可以用于更大的多层抽象系统中。

C++ 回避了那些甚至不使用时都会引发时间或内存额外开销的特性。例如，那些在每个对象中都必须保存“管理信息”的结构都未被采纳。因此，如果用户声明了一个由两个 16 位值组成的结构，该结构将会放入一个 32 位的寄存器中。除了 new、delete、typeid、dynamic_cast 和 throw 这几个运算符以及 try 块之外，其他 C++ 表达式和语句都不需要运行时支持。这对嵌入式应用和高性能应用来说是非常必要的。特别是，这意味着 C++ 的抽象机制对嵌入式、高性能、高可靠性和实时系统等应用场景是适用的。因此，开发这些应用的程序员不必使用低层语言特性（意味着易出错、创造性差且生产力低）来编写程序。

C++ 被设计成用于传统的编译和运行时环境，即 UNIX 系统［UNIX，1985］上的 C 语言编程环境。幸运的是，C++ 从未被局限于 UNIX 系统；它只是用 UNIX 和 C 作为展示语言、库、编译器、链接器、执行环境等之间关系的一个范本。基本上，这个最小范本帮助 C++ 在几乎所有计算平台上都取得了成功。但是，某些情况下还是有充足的理由在提供更多运行时支持的环境中使用 C++。这时，诸如动态载入、增量编译和类型定义数据库等机制都能得以施展，而不会影响语言本身。

并不是所有代码都能做到结构良好、硬件无关、易读等。C++ 拥有一些特性，其设计目的就是为了直接高效地操纵硬件设备，同时又不必担心安全性以及是否易于理解。而 C++ 的另外一些特性又能将这类代码隐藏在优雅而安全的接口之后。

C++ 在大型程序开发中的应用很自然地令大量程序员使用它。C++ 对模块化、强类型接口以及灵活性的强调在此获得了回报。但是，随着程序变得庞大，开发和维护方面遇到的问题逐渐从局部的语言问题转变为更全局的工具和管理问题。

本书着重介绍的技术可提供通用的特性、广泛使用的类型、库、等等。这些技术既能为小型程序开发者所用，也能为大型程序的程序员所用。而且，由于所有非平凡的程序都是由很多半独立的部分组成的，用来编写这种组成部分的技术可为所有应用的程序员所用。

我以标准库组件（如 vector）的实现和使用为例。这些例子介绍了标准库组件及它们的基本设计理念和实现技术。这些例子展示了程序员如何设计并实现他们自己的库。但是，如果对某个问题标准库已经提供了相应的组件，那么使用这个组件来解决问题几乎总是比重新构造你自己的组件更好。即使标准组件对特定问题可能比自定义组件稍差些，但它很可能适用范围更广、更容易获取且更广为人知。长期而言，标准组件（可能通过一个方便的自定义接口来访问）更有利于降低维护、移植、调优以及培训成本。

你可能怀疑用一个更复杂的类型结构来编写程序会导致程序源码的大小（乃至生成的目标代码的大小）增大。但对 C++ 并非如此。一个用类等特性来声明函数参数类型的 C++ 程序通常比不使用这些特性但等价的 C 程序要短一点儿。而使用库的 C++ 程序则会比等价的 C 程序短得多（当然，假定要用 C 编写与库功能相同的代码）。

C++ 支持系统程序设计。这意味着 C++ 代码能高效地与系统中用其他语言编写的代码

进行互操作。用单一的程序设计语言来编写所有软件的想法只是一个空想而已。C++ 从一开始就被设计成能与 C、汇编和 Fortran 简单而高效地交互。在这里，简单高效交互的意思是：一个 C++、C、汇编或 Fortran 函数可以调用其他语言编写的函数，没有额外开销，相互之间传递的数据结构也不必进行转换。

C++ 被设计成在单一地址空间内进行操作。多进程和多地址空间的使用完全依赖于（语言之外的）操作系统的支持。特别是，我假定一个 C++ 程序员能够使用操作系统命令启动进程在系统中运行。最初，这依赖 UNIX Shell 命令实现，但实际上几乎所有“脚本语言”都能做到。因此，C++ 不提供对多地址空间和多进程的支持，但从最早期它就被用在依赖这些特性的系统中。C++ 的设计目的之一就是成为一个大规模多语言并行系统的一部分。

1.3　学习 C++

现实中不存在完美的程序设计语言。但幸运的是，为了成为开发优秀系统的好工具，程序设计语言不必是完美的。实际上，一种通用程序设计语言也不可能在它所应用的所有任务中都是完美的。对于一个任务是完美的，通常对另一个任务就会有严重缺陷，因为对一个领域完美通常意味着专门化。因此，C++ 的设计目标就是在各种各样的系统的开发中都能成为好工具，并且能够直接表达各种各样的想法。

并不是所有想法都能用语言的内置特性直接表达。实际上，这甚至并不是我们所追求的理想。语言特性的存在是为了支持各种程序设计风格和技术。因此，语言的学习应该更关注掌握其固有的、内在的风格，而不是试图了解每个语言特性的所有细节。编程练习是基础，因为理解一种程序设计语言并不只是智力训练，将想法付诸实践更为重要。

在实际的程序设计中，了解最生僻的语言特性或是能使用最多数量的特性并没有什么优势。孤立的单个特性没什么意思，一个特性只有在编程技术和其他特性所提供的语境中才有意义。因此，当阅读后续章节时，请记住学习 C++ 细节知识的真正目的是：在良好设计所提供的语境中，有能力组合使用语言特性和库特性来支持好的程序设计风格。

没有任何一个重要的系统是单纯由语言特性实现的。我们需要构建并使用库来简化程序设计任务，提高系统的质量。我们使用库来提高可维护性、可移植性以及性能。我们将应用程序的基本概念表示为库中的抽象（例如，类、模板以及类层次），很多最基础的程序设计概念都可以用标准库来表达。因此，学习标准库是学习 C++ 不可分割的一部分。标准库是一个知识库，保存了大量辛苦获得的如何高效使用 C++ 的知识。

C++ 广泛用于教学和研究。那些指责 C++ 并非有史以来最小或最纯净的语言（这个观点是正确的）的人会对此感到惊讶。但是 C++：

- 对于基本设计和编程概念的成功教学来说已足够纯净；
- 对于高级概念和技术的教学来说是足够全面的工具；
- 对高要求的开发项目来说足够实用、高效、灵活；
- 若将所学用于非学术场景，它是足够好的商业化工具；
- 完全适用于依赖多样的开发和执行环境的机构与合作单位。

总之，C++ 是一种你可以与之一道成长的语言。

学习 C++ 最重要的是重视基本概念（例如类型安全、资源管理和不变式）和程序设计技术（例如使用限定作用域的对象进行资源管理以及在算法中使用迭代器），还要注意不要迷失在语言技术性细节中。学习一门程序设计语言的目的是成为一个更好的程序员，即，能更

高效地设计和实现新系统、维护旧系统。为此，领悟编程和设计技术比了解所有细节重要得多。对技术性细节不必过分担心，只要你付出时间不断练习，自然而然就掌握了。

C++程序设计基于强静态类型检查，大多数技术的目标是实现程序员想法的高层抽象和直接表达。而这些技术与低层编程技术相比，也并未在运行时间和空间效率上有所妥协。为了从C++的这些优势受益，从其他语言转到C++的程序员必须学习并吸收常用的C++程序设计风格和技术，对使用表达力较低的早期C++版本的程序员也是如此。

将一种语言中很高效的技术草率地用于另一种语言通常会导致令人难堪的糟糕性能和难以维护的代码。编写这种代码也最令人沮丧，因为每一行代码和每一个编译器错误信息都在提醒程序员正在使用的语言不同于“老语言”。你可以用Fortran、C、Lisp、Java等的风格来编写任何语言的程序，但用理念完全不同的语言编写程序既不愉快也不经济。每种语言都可以为如何编写C++程序提供丰富的思想。但是，这些思想必须转换为适合C++通用结构和类型系统的东西，才能在C++中高效应用。如果超越了一种语言的基本类型系统，即使成功了也必然是皮洛士式的胜利[⊖]。

关于是否应该在学习C++之前学习C语言，一直存在争论，我坚信最好的方式是直接学习C++。C++是一种更安全、表达力更强的语言，而且它降低了关注低层技术的要求。当你已经接触了C和C++的共同子集以及C++直接支持的一些高层编程技术后，再来学习C语言中那些为了弥补高层特性的缺乏而设计的部分（也是C语言中最复杂的部分）会更简单。第44章给出了程序员从C++迁移到C的指南，或者说是C++程序员如何处理旧程序的指南。关于如何向初学者讲授C++，我在[Stroustrup，2008]中有详细阐述。

当前，有不少成熟的C++的实现，它们都包含丰富的工具、库和软件开发环境。为了掌握所有这些内容，你可以查阅教材、手册和令人眼花缭乱的各种网络资源。如果你计划认真地使用C++，我强烈建议你查阅一些这类资源。每种实现都有自己的重点和偏好，因此最好使用至少其中两种。

1.3.1 用C++编程

“如何用C++编写出好程序？”这个问题与“如何写出好的英语散文？”很相似。对于第二个问题，人们的答案是：“弄清楚你想要说什么”及“练习，模仿好的写作方式”。这两点看起来对C++编程也是适用的，但也同样难以遵循。

C++程序设计的主要理念与大多数高级语言编程一样：用代码直接表达从设计而来的概念（想法、意图等）。我们试图保证我们所讨论的概念——在白板上用方框和箭头表示的概念，在我们的（非程序设计）教材中找到的概念——在我们的程序中都有直接且明显的对应：

[1] 用代码直接表达想法。

[2] 用代码直接表达想法之间的关联（例如，层次化、参数化以及所属关系）。

[3] 无关的想法独立用代码表达。

[4] 保持简单（但不会令复杂的事无法实现）。

更具体的：

[5] （如果适用的话）尽量使用静态类型检查。

[6] 保持信息局部性（例如，避免全局变量、尽量减少指针的使用）。

⊖ 指付出极大代价才获得的胜利。——译者注

[7] 不要过分抽象化（即，没有明显的需求时不要使用泛型、引入类层次或是进行参数化）。

1.3.2 节给出了更具体的建议。

1.3.2 对 C++ 程序员的建议

到目前为止，已经有很多人使用 C++ 十几二十年了。而更多人正在单一的环境中使用 C++，并忍受着早期编译器和第一代库所强加的限制。通常，一个有经验的 C++ 程序员多年间可能会忽略的并非新特性本身的引入，而是特性间关系的改变，而恰好是这种改变令基础性的新程序设计风格成为可能。换句话说，在你最初学习 C++ 时不予考虑的或是发现不可行的东西，如今恰恰可能是先进的方法。你只能通过复习基础知识来发现这些。

按顺序通读所有章节。如果你已经了解了某一章的内容，那么花几分钟就能读完这一章。如果你尚不了解，就会学到一些新知识。为了写这本书，我学习了很多新知识，我怀疑几乎没有 C++ 程序员了解本书介绍的所有特性和技术，因此你总会学到一些未曾见过的新知识。而且，为了更好地使用一种语言，你需要对特性和技术有一个全局视角，梳理出它们的顺序。通过恰当的内容组织和充分的实例，本书就提供了这样一个全局视角。

学习本书的一个重点是，利用 C++11 的新特性所提供的机会来更新你的设计和编程技术，使之更加现代化：

[1] 使用构造函数建立不变式（见 2.4.3.2 节、13.4 节和 17.2.1 节）。

[2] 配合使用构造 / 析构函数来简化资源管理（RAII；见 5.2 节和 13.3 节）。

[3] 避免“裸的” new 和 delete（见 3.2.1.2 节和 11.2.1 节）。

[4] 使用容器和算法而不是内置数组和专用代码（见 4.4 节、4.5 节、7.4 节和第 32 章）。

[5] 优先使用标准库特性而非自己开发的代码（见 1.2.4 节）。

[6] 使用异常而非错误代码来报告不能局部处理的错误（见 2.4.3 和 13.1 节）。

[7] 使用移动语义来避免拷贝大对象（见 3.3.2 节和 17.5.2 节）。

[8] 使用 unique_ptr 来引用多态类型的对象（见 5.2.1 节）。

[9] 使用 shared_ptr 来引用共享对象，即，不只有一个所有者负责其析构的对象（见 5.2.1 节）。

[10] 使用模板来保持静态类型安全（消除类型转换）并避免类层次的不必要使用（见 27.2 节）。

对 C 和 Java 程序员的建议（1.3.3 节和 1.3.4 节）可能也是很好的参考。

1.3.3 对 C 程序员的建议

程序员对 C 语言掌握得越好，似乎就越难避免用 C 风格编写 C++ 程序，从而失去了 C++ 的很多潜在优势。关于 C 和 C++ 之间的差异，请查阅第 44 章。我对 C 程序员学习本书的建议是：

[1] 不要将 C++ 看作增加了一些特性的 C。你可以这样来使用 C++，但这将导致次最优的结果。为了真正发挥 C++ 相对于 C 的优势，你需要采用不同的设计和实现风格。

[2] 不要用 C++ 来写 C 程序；这通常会令可维护性和性能都非常不好。

[3] 将C++标准库作为学习新技术和新程序设计风格的老师。注意它与C标准库的差异（例如，字符串拷贝用=而不是strcpy()以及字符串比较用==而不是strcmp()）。

[4] C++几乎从来不需要宏替换。作为替代，使用const（见7.5节）、constexpr（见2.2.3节和10.4节）、enum或enum class（见8.4节）来定义明示常量；使用inline（见12.1.5节）来避免函数调用的开销；使用template（见3.4节和第23章）来指明函数族和类型族；使用namespace（见2.4.2节和14.3.1节）来避免名字冲突。

[5] 在真正需要一个变量时再声明它，且声明后立即进行初始化。声明可以出现在语句可以出现的任何位置，以及for-语句初始化部分（见9.5节）和条件中（见9.4.3节）。

[6] 不要使用malloc()、new运算符（见11.2节）可以完成相同的工作，而且完成得更好。同样，不要使用realloc()，尝试用vector（见3.4.2节）。但注意不要简单地用“裸的”new和delete来代替malloc()和free()（见3.2.1.2节和11.2.1节）。

[7] 避免使用void*、联合以及类型转换，除非在某些函数和类的深层实现中。使用这些特性会限制你从类型系统得到的支持，而且会损害性能。在大多数情况下，一次类型转换就暗示着一个设计错误。如果你必须使用显式类型转换，尝试使用命名的显式类型转换（例如static_cast；见11.5.2节），这能更精确地表达你的意图。

[8] 尽量减少数组和C-风格字符串的使用。与这种传统的C风格程序相比，通常可以用C++标准库中的string（见4.2节）、array（见8.2.4节）和vector（见4.4.1节）写出更简单也更易维护的代码。一般而言，如果标准库中已经提供了相应的功能，就尽量不要自己重新构造代码。

[9] 除非是在非常专门的代码中（例如内存管理器），或是进行简单的数组遍历（例如++p），否则要避免对指针进行算术运算。

[10] 不要认为用C风格（回避诸如类、模板和异常等C++特性）辛苦写出的程序会比一个简短的替代程序（例如，使用标准库特性写出的代码）更高效。实际情况通常（当然并不是绝对的）正好相反。

为了遵守C的连接规范，C++函数必须声明为使用C连接方式（见15.2.5节），才能与C程序连接在一起。

1.3.4 对Java程序员的建议

C++和Java具有相似的语法，但其实是相当不同的语言。它们的目标和应用领域有巨大差异。Java并不是C++的直接继任者，因为从一般意义上讲，继任者应该能做和前任相同的事，而且做得更好也更多。为了更好地使用C++，你应该采用适合C++的编程和设计技术，而不是试图用C++语言来编写Java程序。并不是记住你用new创建的对象都要delete就可以了，而是要了解你已不能再依赖垃圾收集器：

[1] 不要简单地用C++模仿Java风格，这通常会令可维护性和性能都非常不好。

[2] 使用C++的抽象机制（例如类和模板）；不要由于虚假的熟悉感而退回到C语言程序设计风格。

[3] 将 C++ 标准库作为学习新技术和新程序设计风格的老师。
[4] 不要马上为所有类创造一个唯一的基类（Object 类）。对很多 / 大多数类来说，没有它你通常会做得更好。
[5] 尽量不使用引用和指针变量，作为替代，使用局部变量和成员变量（见 3.2.1.2 节、5.2 节、16.3.4 节和 17.1 节）。
[6] 记住："一个变量隐含地是一个引用"不再成立了。
[7] 将指针看作 Java 的引用在 C++ 中的等价物（C++ 引用的局限性更强，不允许改变地址）。
[8] 函数不再默认是 virtual 的了。并非每个类都必然被继承。
[9] 将抽象类作为类层次的接口，避免"脆弱的基类"，即，带数据成员的基类。
[10] 只要有可能就使用限定作用域的资源管理（"资源获取即初始化"，RAII）。
[11] 使用构造函数建立类的不变式（如果失败就抛出异常）。
[12] 如果对象释放时（例如，离开作用域）需要进行清理动作，使用析构函数。不要模仿 finally（这么做太特殊化了，而且长期来看要做的工作比析构函数多得多）。
[13] 避免使用"裸的" new 和 delete，应该使用容器（例如，vector、string 和 map）和句柄类（例如，lock 和 unique_ptr）。
[14] 使用独立函数（非成员函数）来最小化耦合（见标准库算法），使用名字空间（见 2.4.2 节和第 14 章）来限制独立函数的作用域。
[15] 不要使用异常规范（noexcept 除外，见 13.5.1.1 节）。
[16] C++ 嵌套类对外围类的对象没有访问权限。
[17] C++ 提供最小化的运行时反射：dynamic_cast 和 type_id（见第 22 章）。因此应更多依靠编译时特性（例如编译时多态；见第 27 章和第 28 章）。

大多数建议对 C# 程序员也适用。

1.4　C++ 的历史

我发明了 C++，制定了最初的定义，并完成了第一个实现。我选择并制定了 C++ 的设计标准，设计了大多数语言特性，设计或帮助设计了早期标准库中的很多内容，并在 C++ 标准委员会中负责处理扩展提案。

C++ 的设计目的是为程序的组织提供 Simula 的特性［Dahl，1970］［Dahl，1972］，同时为系统程序设计提供 C 的效率和灵活性［Kernighan，1978］［Kernighan，1988］。Simula 是 C++ 抽象机制的最初来源。类的概念（以及派生类和虚函数的概念）也是从 Simula 借鉴而来的。不过，模板和异常则是稍晚引入 C++ 的，灵感的来源也不同。

讨论 C++ 的演化，总是要针对它的使用来谈。我花了大量时间倾听用户的意见，搜集有经验的程序员的观点。特别是，我在 AT&T 贝尔实验室的同事在 C++ 的第一个十年中对其成长贡献了重要力量。

本节是一个简单概览，不会试图讨论每个语言特性和库组件，而且也不会深入细节。更多的信息，特别是更多贡献者的名字，请查阅［Stroustrup，1993］［Stroustrup，2007］和［Stroustrup，1994］。我在 ACM 程序设计语言历史大会上发表的两篇论文和我的《C++ 语言的设计和演化》一书（人们熟知的"D&E"）详细介绍了 C++ 的设计和演化以及 C++ 受到的其他程序设计语言的影响。

大多数作为 ISO C++ 标准一部分而生成的文档材料都可以在网上找到［WG21］。在我的常见问题解答（FAQ）中，我设法维护着一个列表，列出每个标准库特性的提出者和改进者［Stroustrup，2010］。C++ 并非一个不露面的匿名委员会或是一个想象中的万能的“终身独裁者”的作品，而是千万个甘于奉献的、有经验的、辛勤工作的人的劳动结晶。

1.4.1 大事年表

创造 C++ 的工作始于 1979 年秋天，当时的名字是“带类的 C”。下面是简要的大事年表：

1979 “带类的 C”的工作开始。最初的特性集合包括类、派生类、公有 / 私有访问控制、构造函数和析构函数以及带实参检查的函数声明。最初的库支持非抢占的并发任务和随机数发生器。

1984 “带类的 C”被重新命名为 C++。在那个时候，C++ 已经引入了虚函数、函数与运算符重载、引用以及 I/O 流和复数库。

1985 C++ 第一个商业版本发布（10 月 14 日）。标准库中已经包含了 I/O 流、复数和多任务（非抢占调度）特性。

1985 《C++ 程序设计语言》出版（“TC++PL”，10 月 14 日）［Stroustrup，1986］。

1989 《C++ 参考手册批注版》出版（“the ARM”）。

1991 《C++ 程序设计语言（第 2 版）》出版［Stroustrup，1991］，提出了使用模板的泛型编程和基于异常的错误处理（包括资源管理理念“资源管理即初始化”）。

1997 《C++ 程序设计语言（第 3 版）》出版［Stroustrup,1997］，引入了 ISO C++ 标准，包括名字空间、dynamic_cast 和模板的很多改进。标准库加入了标准库模板库（STL）框架，包括泛型容器和算法。

1998 ISO C++ 标准发布。

2002 标准的修订工作开始，这个版本俗称 C++0x。

2003 ISO C++ 标准的一个“错误修正版”发布。一个 C++ 技术报告引入了新的标准库组件，诸如正则表达式、无序容器（哈希表）和资源管理指针，这些内容后来成为 C++0x 的一部分。

2006 ISO C++ 性能技术报告发布，回答了主要与嵌入式系统程序设计相关的代价、可预测性和技术问题。

2009 C++0x 的特性完成。它引入了统一初始化、移动语义、可变模板参数、lambda 表达式、类型别名、一种适合并发的内存模型以及其他很多特性。标准库增加了一些组件，包括线程、锁机制和 2003 年技术报告中的大多数组件。

2011 ISO C++11 标准正式被批准。

2012 第一个完整的 C++11 实现出现。

2012 未来 ISO C++ 标准（被称为 C++14 和 C++17）的制定工作开始。

2013 《C++ 程序设计语言（第 4 版）》出版，增加了 C++11 的新内容。

在制定过程中，C++11 被称为 C++0x——就像其他大型项目中也会出现的情况一样，我们过于乐观地估计了完工日期。

1.4.2 早期的 C++

我最初设计和实现一种新语言的原因是希望在多处理器间和局域网内（现在被称为多核

与集群）发布 UNIX 内核的服务。为此，我需要一些事件驱动的仿真程序，Simula 是写这类程序的理想语言，但性能不佳。我还需要直接处理硬件的能力和高性能并发编程机制，C 很适合编写这类程序，但它对模块化和类型检查的支持很弱。我将 Simula 风格的类机制加入 C 中，结果就得到了“带类的 C”，它的一些特性适合于编写具有最小时间和空间需求的程序，在一些大型项目的开发中，这些特性经受了严峻的考研。“带类的 C”缺少运算符重载、引用、虚函数、模板、异常以及很多很多特性［Stroustrup，1982］。C++ 第一次用于研究机构之外是在 1983 年 7 月。

C++ 这个名字（发音为“see plus plus”）是由 Rick Mascitti 在 1983 年夏天创造的，我们选用它来取代我创造的“带类的 C”。这个名字体现了这种新语言的进化本质——它是从 C 演化而来的，其中“++”是 C 语言的递增运算符。一个稍短的名字“C+”是一个语法错误，它也曾被用于命名另一种不相干的语言。C 语义的行家可能会认为 C++ 不如 ++C。新语言没有被命名为 D 的原因是，它是 C 的扩展，它并没有试图通过删除特性来解决存在的问题，另一个原因是已经有好几个自称 C 语言继任者的语言被命名为 D 了。C++ 这个名字还有另一个解释，请查阅［Orwell，1949］的附录。

最初设计 C++ 的目的之一是让我的朋友们和我不必再用汇编语言、C 语言以及当时各种流行的高级语言编写程序。其主要目标是能让程序员更简单、更愉快地编写好程序。在最初，C++ 并没有“图纸设计”阶段，其设计、文档编写和实现都是同时进行的。当时既没有“C++ 项目”，也没有“C++ 设计委员会”。自始至终，C++ 的演化都是为了处理用户遇到的问题，主导演化的主要是我的朋友、同事和我之间的讨论。

1.4.2.1　语言特性和标准库特性

C++ 最初的设计（当时还叫“带类的 C”）包含带实参类型检查和隐式类型转换的函数声明、具备接口和实现间 public/private 差异的类机制、派生类以及构造函数和析构函数。我使用宏实现了原始的参数化机制，并一直沿用至 1980 年代中期。当年年底，我提出了一组语言特性来支持一套完整的程序设计风格（见 1.2.1 节）。回顾往事，我认为引入构造函数和析构函数是最重要的。用当时的术语来说“构造函数为成员函数创建执行环境，而析构函数则完成相反的工作”。这是 C++ 资源管理策略的根源（导致了对异常的需求），也是许多让用户代码更简洁清晰的技术的关键。我没有听说过（到现在也没有）当时有其他语言支持能执行普通代码的多重构造函数。而析构函数则是 C++ 新发明的特性。

C++ 第一个商业化版本发布于 1985 年 10 月。到那时为止，我已经增加了内联（见 12.1.5 节和 16.2.8 节）、const（见 2.2.3 节、7.5 节和 16.2.9 节）、函数重载（见 12.3 节）、引用（见 7.7 节）、运算符重载（见 3.2.1.1 节、第 18 章和第 19 章）和虚函数（见 3.2.3 节和 20.3.2 节）等特性。在这些特性中，以虚函数的形式支持运行时多态在当时是最受争议的。我是从 Simula 中认识到其价值的，但我发现几乎不可能说服大多数系统程序员也认识到它的价值。系统程序员总是对间接函数调用抱有怀疑，而熟悉其他支持面向对象编程的语言的人则很难相信 virtual 函数快到足以用于系统级代码中。与之相对，很多有面向对象编程背景的程序员在当时很难习惯（现在很多人仍不习惯）这样一个理念：你使用虚函数调用只是为了表达一个必须在运行时做出的选择。虚函数当时受到很大阻力，这可能与另一个理念也遇到阻力相关：你可以通过一种程序设计语言所支持的更正规的代码结构来实现更好的系统。因为当时很多 C 程序员似乎已经接受：真正重要的是彻底的灵活性和程序的每个细节都仔细地人工打造。而当时我的观点是（现在也是）：我们从语言和工具获得的每一点帮助都

很重要，我们正在创建的系统的内在复杂性总是处于我们能（否）表达的边缘。

C++ 的很多设计都是在我的同事的黑板上完成的。在早期，Stu Feldman、Alexander Fraser、Steve Johnson、Brian Kernighan、Doug McIlroy 和 Dennis Ritchie 都给出了宝贵的意见。

在 20 世纪 80 年代的后半段，作为对用户反馈的回应，我继续添加新的语言特性。其中最重要的是模板［Stroustrup，1988］和异常处理［Koenig，1990］。在标准制定工作开始时，这两个特性还都处于实验性状态。在设计模板的过程中，我被迫在灵活性、效率和提早类型检查之间做出决断。在那时，没人知道如何同时实现这三点，也没人知道如何与 C-风格代码竞争高要求的系统应用开发任务。我觉得应该选择前两个性质。回顾往事，我认为这个选择是正确的，模板类型检查尚未有完善的方案，对它的探索一直在进行中［Gregor，2006］［Sutton，2011］［Stroustrup，2012a］。异常的设计则关注异常的多级传播、将任意信息传递给一个异常处理程序以及异常和资源管理的融合（使用带析构函数的局部对象来表示和释放资源，我笨拙地称之为“资源获取即初始化”，见 13.3 节）等问题。

我推广了 C++ 的继承机制，使之支持多重基类［Stroustrup，1987a］。这种机制被称为多重继承（multiple inheritance），它被认为是很有难度且有争议的。我认为它远不如模板和异常重要。当前，支持静态类型检查和面向对象程序设计的语言普遍支持虚基类（通常称为接口（interface））的多重继承。

C++ 语言的演化与一些关键库特性紧紧联系在一起，本书介绍了这些特性。例如，我设计了复数类［Stroustrup，1984］、向量类、栈类和（I/O）流类［Stroustrup，1985］以及运算符重载机制。第一个字符串和列表类是由 Jonathan Shopiro 和我开发的，是我们共同工作的成果之一。Jonathan 的字符串和列表类得到了广泛应用，这是库的特性第一次得到广泛应用。标准库中的字符串类就源于这些早期的工作。［Stroustrup，1987b］中描述了任务库，它是 1980 年编写的第一版“带类的 C”的一部分。我编写这个库及其相关的类是为了支持 Simula 风格的仿真。不幸的是，我一直等到 2011 年（已经过去了 30 年！）才等到并发特性进入标准并被 C++ 实现普遍支持（见 1.4.4.2 节、5.3 节和第 41 章）。模板机制的发展受到了 vector、map、list 和 sort 等各种模板的影响，这些模板是由 Andrew Koenig、Alex Stepanov、我以及其他一些人设计的。

C++ 的成长环境中有着众多成熟的和实验性的程序设计语言（例如 Ada［Ichbiah，1979］、Algol 68［Woodward，1974］和 ML［Paulson，1996］）。那时，我畅游在大约 25 种语言之中，它们对 C++ 的影响都记录在［Stroustrup，1994］和［Stroustrup，2007］中。但是，决定性的影响总是来自于我遇到的应用。这是一个深思熟虑的策略，它令 C++ 的发展是“问题驱动”的，而非模仿性的。

1.4.3 1998 标准

C++ 的使用爆炸式增长，这导致了一些变化。1987 年的某个时候，事情变得明朗，C++ 的正式标准化已是必然，我们必须开始为标准化做好准备了［Stroustrup，1994］。因此，我们有意识地保持 C++ 编译器实现者和主要用户之间的联系。这是通过文件和电子邮件以及 C++ 大会上和其他场合下的面对面会议实现的。

AT&T 贝尔实验室允许我与 C++ 实现者和用户共享 C++ 参考手册修订版本的草案，这对 C++ 及其社区做出了重要贡献。由于这些实现者和用户中很多人都供职于可视为 AT&T 竞争者的公司中，这一贡献的重要性绝对不应被低估。一个不甚开明的公司可能不会这样

做，从而导致严重的语言碎片化问题。正是由于 AT&T 这样做了，使得来自数十个机构的大约一百人阅读草案并提出了意见，使之成为被普遍接受的参考手册和 ANSI C++ 标准化工作的基础文献。这些人的名字可以在《C++ 参考手册批注版》（"the ARM"）[Ellis，1989] 中找到。ANSI 的 X3J16 委员会于 1989 年 12 月筹建，是由 HP 公司发起的。1991 年 6 月，这一 ANSI（美国国家）C++ 标准化工作成为 ISO（国际）C++ 标准化工作的一部分，并被命名为 WG21。自 1990 年起，这些联合的标准委员会逐渐成为 C++ 语言演化及其定义完善工作的主要论坛。我自始至终在这些委员会中任职。特别是，作为扩展工作组（后来改称演化工作组）的主席，我直接负责处理 C++ 重大变化和新特性加入的提案。最初标准草案的公众预览版于 1995 年 4 月发布。1998 年，第一个 ISO C++ 标准（ISO/IEC 14882-1998）[C++，1998] 被批准，投票结果是 22 个国家赞成 0 个国家反对。此标准的"错误修正版"于 2003 年发布，因此你有时会听人提到 C++03，但它与 C++98 本质上是相同的语言。

1.4.3.1 语言特性

在 ANSI 和 ISO 标准化工作开始时，大多数主要的语言特性都已成熟，并记录在 ARM [Ellis，1989] 中。因此，大多数工作都是对特性及其规范的完善。模板机制从这些细节工作中受益很多。名字空间在这期间被引入，来应对 C++ 程序不断增长的规模和不断增加的库。在 HP 公司的 Dmitry Lenkov 的推进下，引入了最少的运行时类型信息（RTTI，见第 22 章）相关的特性。我之前将这些特性排除在 C++ 之外，原因是我发现它们在 Simula 中被滥用了。我尝试将一种可选的保守垃圾收集机制引入标准，但失败了，直到 2011 年它才进入标准。

显然，C++98 在语言特性方面，特别是规范细节方面，要远胜过 1989 年的版本。但是，并非所有的变化都是改进。现在回想起来，除了一些不可避免的小错误，有两个主要的新特性当时也不应该加入：

- 异常说明可以指定一个函数运行时可以抛出哪些异常。这个特性是在 Sun 微系统公司的人的积极推动下加入的。异常说明已经被证明对于提高可读性、可靠性和性能是有害无益的，在 2011 标准中已被弃用（计划将来删除）。2011 标准引入了 noexcept（见 13.5.1 节），它可以解决那些本来希望异常说明能解决的问题，但更简单。
- 显然，将模板的编译和使用分离是理想的方式 [Stroustrup，1994]，但由于模板实际使用带来的一些限制，如何实现这一方式就一点儿也不显然了。委员会经过长时间的争论达成了妥协——将模板 export 作为 1998 标准的一部分。对此问题这并不是一个优雅的解决方案，只有一家厂商实现了 export（爱迪生设计集团），2011 标准中已将此特性删除。我们仍在寻找更好的解决方案。我的观点是，根本问题不在于分离编译本身，而是模板的接口和实现之间的差别并不明确。因此，export 解决的是一个错误的问题。未来，通过提供模板需求的准确说明可能会对语言支持"概念"（见 24.3 节）有所帮助。目前这个领域的研究和设计都很活跃 [Sutton，2011] [Stroustrup，2012a]。

1.4.3.2 标准库

1998 标准中最大且最重要的革新是引入了 STL，这是标准库中一个算法和容器的框架（见 4.4 节、4.5 节、第 31 章、第 32 章和第 33 章）。它是 Alex Stepanov（和 Dave Musser、Meng Lee 等人）在泛型编程方面超过十年长期工作的结果。Andrew Koenig、Beman Dawes

和我为帮助STL被广泛接受做了很多工作[Stroustrup，2007]。现在，STL在C++社区和更大范围内已经有了巨大的影响力。

除了STL之外，标准库的其他组件有一点儿大杂烩的感觉，并没有统一的设计。在C++ 1.0版本[Stroustrup，1993]发布时，我曾想同时推出一个足够大的基础库，但失败了。后来，一位AT&T的（非研究）经理阻止了我的同事和我在2.0版发布时改正这个错误。这意味着在标准制定工作开始时，每个主要的机构（如Borland、IBM、微软以及德州仪器）都有自己的基础库。因此，委员会的工作受到了很大局限，只能在已有的库（例如complex库）组件、那些不会对主要厂商的库造成影响的新组件以及确保不同非标准库之间协同工作的必要组件的基础上进行一些拼拼补补的工作。

标准库string（见4.2节和第36章）源于Jonathan Shopiro和我在贝尔实验室的早期工作，但在标准化过程中一些个人和组织对其进行了修改和扩展。varlarray库用于数值计算（见40.5节），它主要是Kent Budge的工作。Jerry Schwarz利用Andrew Koenig的操纵符技术（见38.4.5.2节）和其他一些想法将我的流库（见1.4.2.1节）转换为iostream库（见4.3节和第38章）。在标准化期间，iostream库又进一步被改进，其中大部分工作是由Jerry Schwarz、Nathan Myers和Norihiro Kumagai完成的。

以商业标准来看，C++98标准库太小了。例如，它不包含图形用户界面（GUI）、数据库访问组件或是网络应用组件。现在很多C++实现都提供这类组件，但它们并不是ISO标准的一部分。其原因并不是技术上的，而是实际应用和商业上的。不过，很多有影响力的人一直都以C标准库为标准来评价一个标准库，而与之相比，C++标准库显然庞大很多。

1.4.4 2011标准

当前的C++标准是C++11，它曾经多年被称为C++0x，是WG21的成员的工作成果，委员会的工作流程和程序日益繁重，但这都是自愿增加的。这些流程可能导致更好的（也更严格的）规范，但也限制了创新[Stroustrup，2007]。这一版标准最初草案的公众预览版于2009年发布，正式的ISO C++标准（ISO/IEC 14882-2011）[C++，2011]于2011年8月被批准，投票结果是21票赞成，0票反对。

造成两个版本的标准之间漫长的时间间隔的原因是，大多数委员会成员（包括我）都对ISO的规则有一个错误印象，以为在一个标准发布之后，在开始新特性的标准化工作之前要有一个“等待期”。结果造成新语言特性的重要工作2002年才开始。其他原因包括现代语言及其基础库日益增长的规模。以标准文本的页数来衡量，语言的规模增长了30%，而标准库则增长了100%。规模的增长大部分都是由更加详细的规范而非新功能造成的。而且，新C++标准的工作显然要非常小心，不能因为不兼容而导致旧代码产生问题。委员会不可以破坏数十亿行正在使用的C++代码。

C++11制定工作的总体目标是：

- 使C++成为系统程序设计和构造库的更好语言。
- 使C++更容易教和学。

这些目标在[Stroustrup，2007]中有记载和详细介绍。

C++11标准制定的一项主要工作是实现并发系统程序设计的类型安全和可移植性。这包括一个内存模型（见41.2节）和一组无锁编程特性（见41.3节）。这些工作主要是由Hans Boehm、Brian McKnight和其他一些人完成的。在此基础上，我们添加了thread库。Pete

Becker、Peter Dimov、Howard Hinnant、William Kempf、Anthony Williams 和其他一些人为此做了大量工作。为了提供一个例子来展示在这些基础并发特性上可以实现什么，我提议进行“一种在任务之间交换信息而无须显式使用锁的方法”的工作，这一工作后来形成了 future 和 async()（见 5.3.5 节），其中大部分工作是由 Lawrence Crowl 和 Detlef Vollmann 完成的。并发领域是如此庞大，以至于完整、详细地列出谁做了什么以及为什么做就可以形成一篇很长的论文。在这里，我不会尝试这么做。

1.4.4.1 语言特性

C++11 相对于 C++98 增加的语言特性和标准库特性的列表见 44.2 节。除了并发性支持之外，其他每个新增特性都可以视为“次要的”，但这种看法没有抓住要点：新语言特性要组合使用才能写出更好的程序。这里“更好的”的意思是更易读易写、更优雅、更不易出错、更易维护、运行得更快、消耗更少资源等。

下面列出了影响 C++11 代码风格的“砖瓦”中我认为用处最广的几个，每个特性都列出了相关章节的引用及其主要作者：

- 默认操作的控制，=delete 和 =default；3.3.4 节、17.6.1 节和 17.6.4 节；Lawrence Crowl 和 Bjarne Stroustrup。
- 从初始化器推断对象的类型，=auto；2.2.2 节和 6.3.6.1 节；Bjarne Stroustrup。我于 1983 年首先设计并实现了 auto，但当时由于和 C 语言的兼容性问题不得不删除了它。
- 推广的常量表达式求值（包括字面值常量）、constexpr；2.2.3 节、10.4 节和 12.1.6 节；Gabriel Dos Reis 和 Bjarne Stroustrup［DosReis，2010］。
- 类内成员初始化器；17.4.4 节；Michael Spertus 和 Bill Seymour。
- 继承的构造函数；20.3.5.1 节；Bjarne Stroustrup、Michael Wong 和 Michel Michaud。
- Lambda 表达式，一种在表达式中隐式定义函数对象供使用的方法；3.4.3 节和 11.4 节；Jaakko Jarvi。
- 移动语义，一种无拷贝传输信息的方法；3.3.2 节和 17.5.2 节；Howard Hinnant。
- 一种声明函数不能抛出异常的方法，noexcept；13.5.1.1 节；David Abrahams、Rani Sharoni 和 Doug Gregor。
- 空指针的一个更适合的名字；7.2.2 节；Herb Sutter 和 Bjarne Stroustrup。
- 范围 for 语句；2.2.5 节和 9.5.1 节；Thorsten Ottosen 和 Bjarne Stroustrup。
- 覆盖控制，final 和 override；20.3.4 节。Alisdair Meredith、Chris Uzdavinis 和 Ville Voutilainen。
- 类型别名，一种为类型或模板提供别名的机制，特别是，一种通过绑定另一个模板的某些实参来定义一个新模板的方法；3.4.5 节和 23.6 节；Bjarne Stroustrup 和 Gabriel Dos Reis。
- 类型和限定作用域的枚举，enum class；8.4.1 节；David E. Miller、Herb Sutter 和 Bjarne Stroustrup。
- 通用和统一的初始化（包括任意长度的初始化器列表和防止窄化转换）；2.2.2 节、3.2.1.3 节、6.3.5 节、17.3.1 节和 17.3.4 节；Bjarne Stroustrup 和 Gabriel Dos Reis。
- 可变参数模板，一种向模板传递任意数量任意类型实参的机制；3.4.4 节和 28.6 节；Doug Gregor 和 Jaakko Jarvi。

还有很多人应该被提及。委员会的技术报告［WG21］和我的C++11 FAQ［Stroustrup，2010a］给出了其中很多人的名字。委员会工作组的会议纪要中提及了更多人。我的名字多次出现的原因（我希望）不是虚荣心作怪，纯粹是因为我选择参与我认为重要的那些特性。在好程序中，这些特性将无处不在。它们的主要作用是充实C++特性集，以便更好地支持各种程序设计风格（见1.2.1节）。它们是程序设计风格综合（也就是C++11）的基础。

大家在一个提案上付出了很多劳动，但它并未进入标准，这就是“概念”。这是一种指定和检查模板实参要求的特性［Gregor，2006］，它基于前期研究（如［Stroustrup，1994］［Siek，2000］和［DosReis，2006］）和委员会的大量工作。我们设计了这个特性，给出了规范说明，实现并测试了它，但委员会以大多数票认定这个提案尚未做好准备。假如我们当初能及时改进“概念”，它可能已经成为C++11中最重要的一个特性了（这个头衔的唯一竞争者是并发性支持）。但是，基于“概念”的复杂性、使用难度和编译时性能，委员会决定拒绝它［Stroustrup，2010b］。我认为我们（委员会）做出了正确的决定，但这个特性确实是“跑掉的大鱼”，关于它的研究和设计已经形成一个活跃的领域［Sutton，2011］［Stroustrup，2012a］。

1.4.4.2 标准库

关于哪些特性将进入C++11标准库的工作是以一个标准委员会技术报告（“TR1”）开始的。最初，Matt Austern是标准库工作组的负责人，后来Howard Hinnant接管了这项工作，直至2011年我们推出最终的标准草案。

与语言特性类似，我只列出几个标准库组件，给出相关章节的引用和主要贡献者的名字。更详细的列表请见44.2.2节。某些组件，例如unordered_map（哈希表），是C++98标准发布时我们未能及时完成的。很多其他组件，例如unique_ptr和function则是技术报告（TR1）（基于Boost库）的一部分。Boost是一个志愿者组织，其创建目的是提供基于STL的有用的库组件［Boost］。

- 哈希容器，如unordered_map：31.4.3节；Matt Austern。
- 基础的并发库组件，如thread、mutex和lock：5.3节和42.2节；Pete Becker，Peter Dimov，Howard Hinnant，William Kempf，Anthony Williams和其他很多人。
- 异步计算发射和结果返回，future、promise和async()：5.3.5节和42.4.6节；Detlef Vollmann，Lawrence Crowl，Bjarne Stroustrup和Herb Sutter。
- 垃圾收集接口：34.5节；Michael Spertus和Hans Boehm。
- 正则表达式库regexp：5.5节和第37章；John Maddock。
- 随机数库：5.6.3节和40.7节；Jens Maurer和Walter Brown。这个组件的加入其实只是时间问题。我在1980年就已经随“带类的C”推出了第一个随机数库。

下面是一些从Boost中提炼出的工具组件：

- 一种用于简单高效传递资源的指针，unique_ptr：5.2.1节和34.3.1节；Howard E. Hinnant。它最初被称为move_ptr，假如当初我们就已经知道如何设计它的话，C++98中的auto_ptr就应该具备这样的功能。
- 一种可以表示共享所有权的指针，shared_ptr：5.2.1节和34.3.2节；Peter Dimov。C++98中的counted_ptr（Greg Colvin所提议）的继任者。
- tuple库：5.4.3节，28.5节和34.2.4.2节；Jaakko Jarvi和Gary Powell。他们感谢了一长串的贡献者，包括Doug Gregor、David Abrahams和Jeremy Siek。

- 通用的 bind()：33.5.1 节；Peter Dimov。他的致谢名单是名副其实的 Boost 名人录（包括 Doug Gregor、John Maddock、Dave Abrahams 和 Jaakko Jarvi）。
- 用于保存可调用对象的 function 类型：33.5.3 节；Doug Gregor。他感谢了 William Kempf 和其他人的贡献。

1.4.5 C++ 的用途

到目前为止，C++ 几乎用在任何地方：你的电脑中、你的手机中、你的汽车里、甚至还可能在你的相机里，但你通常不会看到它。C++ 是一种系统程序设计语言，它最普遍的用途是在系统架构深处，用户是永远看不到那里的。

C++ 被数以百万计的程序员用于几乎每个应用领域。现在有数十亿行 C++ 代码部署在各个地方。如此大量的应用是由几个独立的实现、数千个库、数百本教材和几十个网站所支撑的。各种层次的培训和教学都有丰富的资源。

C++ 早期更多用于强系统程序设计。例如，有若干早期的操作系统是用 C++ 编写的：[Campbell，1987]（学术研究），[Rozier，1988]（实时操作系统），[Berg，1995]（高吞吐量 I/O）。很多当前的操作系统（如 Windows、苹果操作系统、Linux 和大多数便携式设备的操作系统）都用 C++ 编写了系统的某些部分。你的移动电话和因特网路由器的系统很可能是用 C++ 编写的。我认为在低层应用的效率上毫不妥协对 C++ 是非常重要的。这样我们就可以用 C++ 编写驱动程序和其他需要直接操纵硬件且要求实时性的软件。对这类代码，性能的可预测性与单纯的运行速度至少一样重要，通常系统的紧凑性也同等重要。C++ 的设计目标之一就是每个语言特性都适用于编写有严格时间和空间限制的代码（见 1.2.4 节）[Stroustrup，1994，4.5 节]。

一些当前最常见、使用最广泛的系统用 C++ 编写了其关键部分。例如英扎特（航空售票系统）、亚马逊（网络商务系统）、彭博社（金融信息系统）、谷歌（网页搜索系统）和 Facebook（社交媒体系统）。还有很多程序设计语言和技术都是用 C++ 实现的，以获得好的性能和可靠性。这方面的例子包括使用最为广泛的 Java 虚拟机（如 Oracle 的 HotSpot）、JavaScript 解释器（如谷歌的 V8）、浏览器（如微软的 Internet Explorer、Mozilla 的 Firefox、苹果的 Safari 和谷歌的 Chrome）和应用框架（如微软的 .NET 网络服务框架）。我认为 C++ 在基础架构领域的优势是独一无二的 [Stroustrup，2012a]。

大多数应用都有一部分影响性能的关键代码。但是，绝大多数代码并不属于这部分。对大多数代码而言，可维护性、易于扩展以及易于测试才是关键。C++ 对这些方面的支持使它广泛用于那些可靠性必不可少的应用和那些需要随着时间推移进行重大改进的应用中。例如金融系统、电信系统、设备控制和军事应用。数十年来，美国长途电话系统的中央控制都依赖于 C++，而且每一通 800 电话（即被叫方付费的通话）都是由 C++ 程序完成路由的 [Kamath，1993]。这类应用很多都很庞大，软件生命周期也很长。因此，稳定性、兼容性和伸缩性已经成为 C++ 发展过程中不变的关注点。在用 C++ 编写的程序中，数百万行规模很常见。

游戏领域需要许多语言和工具协作，而其中必须有一种语言毫不妥协地提供高效率（通常在“不寻常的”硬件上）。因此，游戏已经成为 C++ 的另一个主要应用领域。

人们常说的系统程序设计广泛用于嵌入式系统中，因此在高要求的嵌入式项目中发现 C++ 的大量使用并不奇怪，包括计算机 X 线断层摄影术（CAT 扫描）、航空控制软件（例如

洛克希德－马丁）、火箭控制、舰船引擎（例如曼恩动力设备公司制造的世界上最大的船用柴油引擎的控制系统）、汽车驾驶软件（例如宝马）以及风力涡轮机控制（例如维斯塔斯）。

C++并不是以数值计算为目的而特殊设计的。但是，很多数值、科学和工程计算代码都是用C++编写的。主要原因是传统的数值计算工作通常必须与图形化相结合，还必须与其他计算相结合，而这些计算所依赖的数据结构不适合传统Fortran语言模式（如[Root, 1995]）。我非常高兴看到C++被用于重要的科学计算项目中，例如人类基因组项目、美国国家航空和宇宙航行局的火星探测器、欧洲核子研究委员会的宇宙起源探索项目以及其他很多项目。

C++能高效地用于那些多领域综合的应用的开发，这是一个很重要的长处。一个应用同时涉及局域网和广域网、数值计算、图形化、用户交互和数据库是很常见的。传统上，这些应用领域被认为是独立的，由使用各种程序设计语言的不同技术社区所支撑。但C++被广泛用于所有这些领域中，还有其他很多领域。C++的设计目的之一就是让其代码能与其他语言编写的代码协作。在这里，C++几十年来的稳定性再次显示了重要作用。而且，任何现实世界中的重要系统都不会百分之百用一种语言来编写。因此，C++的初始设计目标之一——互操作性就变得非常重要了。

重要应用是不会仅仅用裸语言来编写的。C++有大量（除ISO C++标准库之外）的支持库和工具集，例如Boost[Boost]（可移植基础库）、POCO（网站开发库）、QT（跨平台应用开发库）、wxWidgets（跨平台图形用户界面库）、WebKit（网页浏览器布局引擎库）、CGAL（计算几何库）、QuickFix（金融信息交换库）、OpenCV（实时图像处理库）和Root[Root, 1995]（高能物理库）。现在有数以千计的C++库，跟上所有这些库的变化是不可能的。

1.5 建议

本书的每一章都有“建议”一节，给出与该章内容相关的一些具体建议。这些建议都是一些经验法则，而非不变的定律。每条建议只应该用于合理的地方。智慧、经验、常识和好的风格是无可取代的。

我发现把建议写成“绝对不要这样做”是无益的。因此，大多数建议都表述成“该做什么”的形式。否定的建议通常不会以绝对禁止的语气表述，而是设法给出替代的肯定建议，据我了解还没有哪个C++主要特性不能给出正面的使用建议。“建议”一节并不包含解释，而是对每条建议附加一个指向相应章节的引用。

对于初学者，下面列出了一些来自C++的设计、学习和历史这几节的建议：

[1] 用代码直接表达想法（概念），例如，表达为一个函数、一个类或是一个枚举；1.2节。

[2] 编写代码应以优雅且高效为目标；1.2节。

[3] 不要过度抽象；1.2节。

[4] 设计应关注提供优雅且高效的抽象，可能的情况下以库的形式呈现；1.2节。

[5] 用代码直接表达想法之间的关联，例如，通过参数化或类层次；1.2节。

[6] 无关的想法应用独立的代码表达，例如，避免类之间的相互依赖；1.2.1节。

[7] C++并不只是面向对象的；1.2.1节。

[8] C++并不只是用于泛型编程；1.2.1节。

[9] 优选可以进行静态检查的方案；1.2.1节。

［10］ 令资源是显式的（将它们表示为类对象）；1.2.1 节和 1.4.2.1 节。
［11］ 简单的想法应简单表达；1.2.1 节。
［12］ 使用库，特别是标准库，不要试图从头开始构建所有东西；1.2.1 节。
［13］ 使用类型丰富的程序设计风格；1.2.2 节。
［14］ 低层代码不一定高效；不要因为担心性能问题而回避类、模板和标准库组件；1.2.4 节和 1.3.3 节。
［15］ 如果数据具有不变量，封装它；1.3.2 节。
［16］ C++ 并非 C 的简单扩展；1.3.3 节。

总之：编写好程序需要智慧、风格和耐心。你不可能第一次就成功，要不断尝试！

1.6 参考文献

[Austern,2003] Matt Austern et al.: *Untangling the Balancing and Searching of Balanced Binary Search Trees*. Software – Practice & Experience. Vol 33, Issue 13. November 2003.

[Barron,1963] D. W. Barron et al.: *The main features of CPL*. The Computer Journal. 6 (2): 134. (1963). comjnl.oxfordjournals.org/content/6/2/134.full.pdf+html.

[Barton,1994] J. J. Barton and L. R. Nackman: *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley. Reading, Massachusetts. 1994. ISBN 0-201-53393-6.

[Berg,1995] William Berg, Marshall Cline, and Mike Girou: *Lessons Learned from the OS/400 OO Project*. CACM. Vol. 38, No. 10. October 1995.

[Boehm,2008] Hans-J. Boehm and Sarita V. Adve: *Foundations of the C++ concurrency memory model*. ACM PLDI'08.

[Boost] The Boost library collection. www.boost.org.

[Budge,1992] Kent Budge, J. S. Perry, and A. C. Robinson: *High-Performance Scientific Computation Using C++*. Proc. USENIX C++ Conference. Portland, Oregon. August 1992.

[C,1990] X3 Secretariat: *Standard – The C Language*. X3J11/90-013. ISO Standard ISO/IEC 9899-1990. Computer and Business Equipment Manufacturers Association. Washington, DC.

[C,1999] ISO/IEC 9899. *Standard – The C Language*. X3J11/90-013-1999.

[C,2011] ISO/IEC 9899. *Standard – The C Language*. X3J11/90-013-2011.

[C++,1998] ISO/IEC JTC1/SC22/WG21: *International Standard – The C++ Language*. ISO/IEC 14882:1998.

[C++Math,2010] *International Standard – Extensions to the C++ Library to Support Mathematical Special Functions*. ISO/IEC 29124:2010.

[C++,2011] ISO/IEC JTC1/SC22/WG21: *International Standard – The C++ Language*. ISO/IEC 14882:2011.

[Campbell,1987] Roy Campbell et al.: *The Design of a Multiprocessor Operating System*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Coplien,1995] James O. Coplien: *Curiously Recurring Template Patterns*. The C++ Report. February 1995.

[Cox,2007] Russ Cox: *Regular Expression Matching Can Be Simple And Fast*. January 2007. swtch.com/~rsc/regexp/regexp1.html.

[Czarnecki,2000] K. Czarnecki and U. Eisenecker: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley. Reading, Massachusetts. 2000. ISBN 0-201-30977-7.

[Dahl,1970] O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language*. Norwegian Computing Center S-22. Oslo, Norway. 1970.

[Dahl,1972] O-J. Dahl and C. A. R. Hoare: *Hierarchical Program Construction* in *Structured Programming*. Academic Press. New York. 1972.

[Dean,2004] J. Dean and S. Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04: Sixth Symposium on Operating System Design and Implementation. 2004.

[Dechev,2010] D. Dechev, P. Pirkelbauer, and B. Stroustrup: *Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs*. 13th IEEE Computer Society ISORC 2010 Symposium. May 2010.

[DosReis,2006] Gabriel Dos Reis and Bjarne Stroustrup: *Specifying C++ Concepts*. POPL06. January 2006.

[DosReis,2010] Gabriel Dos Reis and Bjarne Stroustrup: *General Constant Expressions for System Programming Languages*. SAC-2010. The 25th ACM Symposium On Applied Computing. March 2010.

[DosReis,2011] Gabriel Dos Reis and Bjarne Stroustrup: *A Principled, Complete, and Efficient Representation of C++*. Journal of Mathematics in Computer Science. Vol. 5, Issue 3. 2011.

[Ellis,1989] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.

[Freeman,1992] Len Freeman and Chris Phillips: *Parallel Numerical Algorithms*. Prentice Hall. Englewood Cliffs, New Jersey. 1992. ISBN 0-13-651597-5.

[Friedl,1997] Jeffrey E. F. Friedl: *Mastering Regular Expressions*. O'Reilly Media. Sebastopol, California. 1997. ISBN 978-1565922570.

[Gamma,1995] Erich Gamma et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, Massachusetts. 1994. ISBN 0-201-63361-2.

[Gregor,2006] Douglas Gregor et al.: *Concepts: Linguistic Support for Generic Programming in C++*. OOPSLA'06.

[Hennessy,2011] John L. Hennessy and David A. Patterson: *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann. San Francisco, California. 2011. ISBN 978-0123838728.

[Ichbiah,1979] Jean D. Ichbiah et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices. Vol. 14, No. 6. June 1979.

[Kamath,1993] Yogeesh H. Kamath, Ruth E. Smilan, and Jean G. Smith: *Reaping Benefits with Object-Oriented Technology*. AT&T Technical Journal. Vol. 72, No. 5. September/October 1993.

[Kernighan,1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*. Prentice Hall. Englewood Cliffs, New Jersey. 1978.

[Kernighan,1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language, Second Edition*. Prentice-Hall. Englewood Cliffs, New Jersey. 1988. ISBN 0-13-110362-8.

[Knuth,1968] Donald E. Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Massachusetts. 1968.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. Vol. 1, No. 7. July 1989.

[Koenig,1990] A. R. Koenig and B. Stroustrup: *Exception Handling for C++ (revised)*. Proc USENIX C++ Conference. April 1990.

[Kolecki,2002] Joseph C. Kolecki: *An Introduction to Tensors for Students of Physics and Engineering*. NASA/TM-2002-211716.

[Langer,2000] Angelika Langer and Klaus Kreft: *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley. 2000. ISBN 978-0201183955.

[McKenney] Paul E. McKenney: *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org. Corvallis, Oregon. 2012. http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html.

[Maddock,2009] John Maddock: *Boost.Regex*. www.boost.org. 2009.

[Orwell,1949] George Orwell: *1984*. Secker and Warburg. London. 1949.

[Paulson,1996] Larry C. Paulson: *ML for the Working Programmer*. Cambridge University Press. Cambridge. 1996. ISBN 0-521-56543-X.

[Pirkelbauer,2009] P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup: *Design and Evaluation of C++ Open Multi-Methods*. Science of Computer Programming. Elsevier Journal. June 2009. doi:10.1016/j.scico.2009.06.002.

[Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL – The Language and Its Compiler*. Cambridge University Press. Cambridge. 1980. ISBN 0-521-21965-5.

[Root,1995] *ROOT: A Data Analysis Framework*. root.cern.ch. It seems appropriate to represent a tool from CERN, the birthplace of the World Wide Web, by a Web address.

[Rozier,1988] M. Rozier et al.: *CHORUS Distributed Operating Systems*. Computing Systems. Vol. 1, No. 4. Fall 1988.

[Siek,2000] Jeremy G. Siek and Andrew Lumsdaine: *Concept checking: Binding parametric polymorphism in C++*. Proc. First Workshop on C++ Template Programming. Erfurt, Germany. 2000.

[Solodkyy,2012] Y. Solodkyy, G. Dos Reis, and B. Stroustrup: *Open and Efficient Type Switch for C++*. Proc. OOPSLA'12.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Technical Report HPL-94-34 (R. 1). 1994.

[Stewart,1998] G. W. Stewart: *Matrix Algorithms, Volume I. Basic Decompositions*. SIAM. Philadelphia, Pennsylvania. 1998.

[Stroustrup,1982] B. Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Sigplan Notices. January 1982. The first public description of "C with Classes."

[Stroustrup,1984] B. Stroustrup: *Operator Overloading in C++*. Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment. September 1984.

[Stroustrup,1985] B. Stroustrup: *An Extensible I/O Facility for C++*. Proc. Summer 1985 USENIX Conference.

[Stroustrup,1986] B. Stroustrup: *The C++ Programming Language*. Addison-Wesley. Reading, Massachusetts. 1986. ISBN 0-201-12078-X.

[Stroustrup,1987] B. Stroustrup: *Multiple Inheritance for C++*. Proc. EUUG Spring Conference. May 1987.

[Stroustrup,1987b] B. Stroustrup and J. Shopiro: *A Set of C Classes for Co-Routine Style Programming*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Stroustrup,1988] B. Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver. 1988.

[Stroustrup,1991] B. Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Massachusetts. 1991. ISBN 0-201-53992-6.

[Stroustrup,1993] B. Stroustrup: *A History of C++: 1979-1991*. Proc. ACM History of Programming Languages conference (HOPL-2). ACM Sigplan Notices. Vol 28, No 3. 1993.

[Stroustrup,1994] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-54330-3.

[Stroustrup,1997] B. Stroustrup: *The C++ Programming Language, Third Edition*. Addison-Wesley. Reading, Massachusetts. 1997. ISBN 0-201-88954-4. Hardcover ("Special") Edition. 2000. ISBN 0-201-70073-5.

[Stroustrup,2002] B. Stroustrup: *C and C++: Siblings*, *C and C++: A Case for Compatibility*, and *C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. July-September 2002. www.stroustrup.com/papers.html.

[Stroustrup,2007] B. Stroustrup: *Evolving a language in and for the real world: C++ 1991-2006*. ACM HOPL-III. June 2007.

[Stroustrup,2008] B. Stroustrup: *Programming – Principles and Practice Using C++*. Addison-Wesley. 2009. ISBN 0-321-54372-6.

[Stroustrup,2010a] B. Stroustrup: *The C++11 FAQ*. www.stroustrup.com/C++11FAQ.html.

[Stroustrup,2010b] B. Stroustrup: *The C++0x "Remove Concepts" Decision*. Dr. Dobb's Jour-

nal. July 2009.

[Stroustrup,2012a] B. Stroustrup and A. Sutton: *A Concept Design for the STL*. WG21 Technical Report N3351—12-0041. January 2012.

[Stroustrup,2012b] B. Stroustrup: *Software Development for Infrastructure*. Computer. January 2012. doi:10.1109/MC.2011.353.

[Sutton,2011] A. Sutton and B. Stroustrup: *Design of Concept Libraries for C++*. Proc. SLE 2011 (International Conference on Software Language Engineering). July 2011.

[Tanenbaum,2007] Andrew S. Tanenbaum: *Modern Operating Systems, Third Edition*. Prentice Hall, Upper Saddle River, New Jersey. 2007. ISBN 0-13-600663-9.

[Tsafrir,2009] Dan Tsafrir et al.: *Minimizing Dependencies within Generic Classes for Faster and Smaller Programs*. ACM OOPSLA'09. October 2009.

[Unicode,1996] The Unicode Consortium: *The Unicode Standard, Version 2.0*. Addison-Wesley. Reading, Massachusetts. 1996. ISBN 0-201-48345-9.

[UNIX,1985] *UNIX Time-Sharing System: Programmer's Manual. Research Version, Tenth Edition*. AT&T Bell Laboratories, Murray Hill, New Jersey. February 1985.

[Vandevoorde,2002] David Vandevoorde and Nicolai M. Josuttis: *C++ Templates: The Complete Guide*. Addison-Wesley. 2002. ISBN 0-201-73484-2.

[Veldhuizen,1995] Todd Veldhuizen: *Expression Templates*. The C++ Report. June 1995.

[Veldhuizen,2003] Todd L. Veldhuizen: *C++ Templates are Turing Complete*. Indiana University Computer Science Technical Report. 2003.

[Vitter,1985] Jefferey Scott Vitter: *Random Sampling with a Reservoir*. ACM Transactions on Mathematical Software, Vol. 11, No. 1. 1985.

[WG21] ISO SC22/WG21 The C++ Programming Language Standards Committee: *Document Archive*. www.open-std.org/jtc1/sc22/wg21.

[Williams,2012] Anthony Williams: *C++ Concurrency in Action – Practical Multithreading*. Manning Publications Co. ISBN 978-1933988771.

[Wilson,1996] Gregory V. Wilson and Paul Lu (editors): *Parallel Programming Using C++*. The MIT Press. Cambridge, Mass. 1996. ISBN 0-262-73118-5.

[Wood,1999] Alistair Wood: *Introduction to Numerical Analysis*. Addison-Wesley. Reading, Massachusetts. 1999. ISBN 0-201-34291-X.

[Woodward,1974] P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide*. Her Majesty's Stationery Office. London. 1974.

第 2 章
The C++ Programming Language, Fourth Edition

C++ 概览：基础知识

首要任务，干掉所有语言专家。

——《亨利六世》(第二部分)

- 引言
- 基本概念
  Hello，World!；类型、变量和算术运算；常量；检验和循环；指针、数组和循环
- 用户自定义类型
  结构；类；枚举
- 模块化
  分离编译；名字空间；错误处理
- 附记
- 建议

2.1 引言

本章以及接下来 3 章的主要目的是在不涉及过多细节的前提下，带领读者初步了解 C++ 语言。其中，第 2 章介绍 C++ 的符号系统、C++ 的存储和计算模型以及如何将代码组织成程序，这些特性可以支持 C 语言中常见的编程模式，即面向过程的程序设计（procedural programming）。接下来，第 3 章介绍 C++ 的抽象机制，第 4、5 两章则给出一些标准库功能的示例。

你最好在有了一些编程经验之后再阅读本章。如果没有，建议读者先找一本入门教材学习一下，比如《Programming：Principles and Practice Using C++》[Stroustrup，2009]。即便你编写过程序，你使用的语言或编写的应用也可能在风格或形式上与本书介绍的 C++ 的相距甚远。因此，如果你发现接下来的“快速导览”不那么容易理解，不妨直接跳到第 6 章，从那儿开始我们将会对知识介绍得更加系统和有条理。

通过学习 C++ 概览的内容，我们能在早期就使用大量 C++ 语言的功能，而不必非要一步一步学完语言和标准库的所有特性再使用它们。例如，本书直到第 10 章才会详细讨论关于循环的内容，但是在此之前就已经使用循环语句了。同样，关于类、模板、自由存储和标准库的详细描述分散在不同章节中，但是只要有助于更好地呈现示例代码，我会在任何章节中使用 vector、string、complex、map、unique_ptr 和 ostream 等标准库类型。

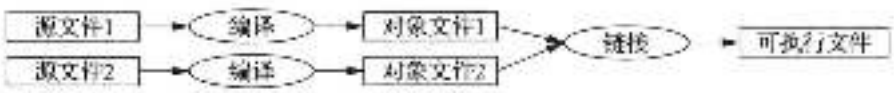
不妨用城市观光的例子来说明这一点，比方说参观哥本哈根（Copenhagen）或者纽约（New York）。在短短几个小时之内，你可能会匆匆游览几个主要的景点、听到一些有趣的传说或故事，然后被告知接下来应该参观哪里。但是仅靠这样一段旅程，你无法真正了解这座城市，甚至对听到和看到的东西也是一知半解。要想认识并融入一座城市，必须在其中生活很多年。不过，如果运气好的话，在这样的浏览之后你会对城市的总体情况有一些了解，知

道城市的特殊之处，并且对某些方面产生兴趣。接下来，你就能进入真正的探索旅程了。

本书的概览部分把C++作为一个整体呈现在读者面前，而非像千层糕一样一层一层仔细介绍。因此，在这里我们不会细分某项语言特性归属于C、C++98还是C++11，这些语言的历史信息在第1.4节和第44章可以找到。

2.2 基本概念

C++是一种编译型语言。顾名思义，要想运行一段C++程序，需要首先用编译器把源文件转换为对象文件，然后再用链接器把这些对象文件组合生成可执行程序。一个C++程序通常包含许多源代码文件（通常简称为源文件）。



一个可执行程序适用于一种特定的硬件/系统组合，是不可移植的。例如，可执行程序无法直接从Mac移植到Windows PC。当我们谈论C++程序的可移植性时，通常是指源代码的可移植性。也就是说，同一份源代码可以在不同系统上成功编译并运行。

ISO的C++标准定义了两种实体：

- 核心语言功能，比如内置类型（如char和int）和循环（如for语句和while语句）；
- 标准库组件，比如容器（如vector和map）和I/O操作（如<<和getline()）。

每个C++实现都会提供标准库组件，其实它们也是非常普通的C++代码。换句话说，C++标准库可以用C++语言本身实现（仅在实现线程上下文切换这样的功能时才使用少量机器代码）。这意味着C++在面对绝大多数要求较高的系统编程任务时高效且有足够的表达能力。

C++是一种静态类型语言，这意味着编译器在处理任何实体（如对象、值、名称和表达式）时，都必须清楚它的类型。对象的类型决定了能在该对象上执行哪些操作。

2.2.1 Hello，World!

最小的C++程序如下所示：

```
int main() { }        // 最小的 C++ 程序
```

这段代码定义了一个名为main的函数，该函数不接受任何参数也不做任何实际工作（见15.4节）。

在C++中，花括号{}表示成组的意思，此例中它指示出函数体的首尾位置。从双斜线//开始直到该行结束是注释，注释只是供读者阅读的，编译器不处理注释。

在每个C++程序中有且只有一个名为main()的全局函数，在执行一个程序时首先执行该函数。如果main()返回一个int值，则这个值将作为程序给“系统”的返回值。如果main()没有返回任何值，则系统也将收到一个表示程序成功完成的值。来自main()的非零返回值表示程序执行失败。并非所有操作系统和执行环境都会用到这个返回值：基于Linux/Unix的环境通常会用到，而基于Windows的环境一般不会用到。

通常情况下，一个程序会产生某些输出结果。下面的程序输出Hello, World!：

```
#include <iostream>

int main()
```

```
{
    std::cout << "Hello, World!\n";
}
```

代码行 #include<iostream> 指示编译器把 iostream 中涉及标准流 I/O 功能的声明包含（include）进来。没有这些声明的话，表达式

```
std::cout << "Hello, World!\n"
```

将无法正确执行。运算符 <<（"输出"）把它的第 2 个参数写入到第 1 个参数。在这个例子中，字符串字面值 "Hello, World!\n" 被写入到标准输出流 std::cout。字符串字面值是指一对双引号当中的字符序列。在字符串字面值中，反斜线 \ 紧跟一个其他字符代表某种"特殊字符"。在这个例子中，\n 是换行符，因此输出的字符是 Hello, World!，后面紧跟一个换行。

std:: 指定名字 cout 所在的标准库名字空间（见 2.4.2 节和第 14 章）。本书在讨论标准特性时通常会省略掉 std::，2.4.2 节将介绍在不使用显式限定符的情况下如何让名字空间中的名字可见。

基本上所有可执行代码都要放在函数中，并且被 main() 直接或间接地调用。例如：

```
#include <iostream>
using namespace std;          // 使得 std 中的名字无须 std:: 就变得可见（见 2.4.2 节）

double square(double x)       // 计算一个双精度浮点数的平方
{
    return x*x;
}

void print_square(double x)
{
    cout << "the square of " << x << " is " << square(x) << "\n";
}

int main()
{
    print_square(1.234);      // 打印：1.234 的平方是 1.52276
}
```

"返回类型" void 表示该函数不返回任何值。

2.2.2　类型、变量和算术运算

每个名字和每个表达式都有一个类型，类型决定所能执行的操作。例如，如下的声明

```
int inch;
```

将 inch 的类型指定为 int，也就是说，inch 是一个整型变量。

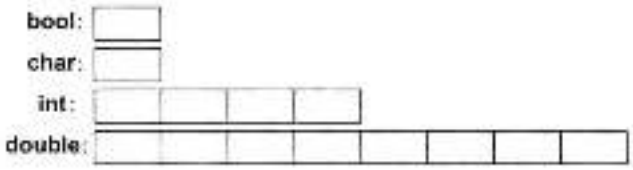
声明（declaration）是一条语句，负责为程序引入一个新的名字，并指定该命名实体的类型：

- 类型（type）定义一组可能的值以及一组（对象上的）操作；
- 对象（object）是存放某类型值的内存空间；
- 值（value）是一组二进制位，具体的含义由其类型决定；
- 变量（variable）是一个命名的对象。

C++ 提供了若干基本类型，例如：

```
bool      // 布尔值，可取 true 或 false
char      // 字符，如 'a'、'z' 和 '9'
int       // 整数，如 1、42 和 1066
double    // 双精度浮点数，如 3.14 和 299793.0
```

每种基本类型都与硬件特性直接相关，其尺寸固定不变，决定了其中所能存储的值的范围：



一个 char 变量的尺寸取决于在给定的机器上存放一个字符所需的空间（通常是一个 8 位的字节），其他类型的尺寸则是 char 尺寸的整数倍。类型的实际尺寸是依赖于实现的（即在不同机器上可能不同），我们使用 sizeof 运算符可以得到它；例如，sizeof(char) 等于 1，sizeof(int) 常常是 4。

算术运算符可用于上述这些类型的组合：

```
x+y       // 加法
+x        // 一元加法
x-y       // 减法
-x        // 一元减法
x*y       // 乘法
x/y       // 除法
x%y       // 整数取余（取模）
```

比较运算符也是如此：

```
x==y      // 相等
x!=y      // 不相等
x<y       // 小于
x>y       // 大于
x<=y      // 小于等于
x>=y      // 大于等于
```

在赋值运算和算术运算中，C++ 会在基本类型间进行所有有意义的类型转换（见 10.5.3 节），以便它们自由地进行混合运算：

```
void some_function()    // 返回空值的函数
{
    double d = 2.2;     // 初始化浮点数
    int i = 7;          // 初始化整数
    d = d+i;            // 把求和结果赋给 d
    i = d*i;            // 把乘积结果赋给 i（d*i 是一个 double 型，在这里被截成一个 int 值）
}
```

注意 = 是赋值运算符，而 == 用于相等性判断。

C++ 提供了好几种表示初始化的符号，比如上面用到的 =，此外还有一种更加通用的形式，这种形式使用的是花括号内的一组初始化器列表：

```
double d1 = 2.3;
double d2 {2.3};

complex<double> z = 1;          // 复数为双精度浮点数的复数
```

```
complex<double> z2 {d1,d2};
complex<double> z3 = {1,2};        // 当使用 {...} 的时候，符号 = 是可选的

vector<int> v {1,2,3,4,5,6};       // 由整数构成的向量
```

符号 = 是一种比较传统的形式，最早被 C 语言使用。但是如果拿不准的话，最好在 C++ 中使用更通用的 {} 列表形式（见 6.3.5.2 节）。抛开其他因素不谈，使用初始化器列表的形式至少可以确保不会发生某些可能导致信息丢失的类型转换（窄化类型转换，第 10.5 节）：

```
int i1 = 7.2;          // i1 变成了 7（意料之外的情况？）
int i2 {7.2};          // 错误：试图执行浮点数向整数的类型转换
int i3 = {7.2};        // 错误：试图执行浮点数向整数的类型转换（这里的符号 = 是多余的）
```

常量（见 2.2.3 节）在声明时不能不进行初始化，普通变量也只应在极有限的情况下不进行初始化。换句话说，在引入一个新名字时最好已经有了一个合适的值。用户定义的类型（如 string、vector、Matrix、Motor_controller 和 Orc_warrior）可以在定义时进行隐式初始化（见 3.2.1.1 节）。

在定义一个变量时，如果变量的类型可以由初始化器推断得到，则我们无须显式指定其类型：

```
auto b = true;         // 变量的类型是 bool
auto ch = 'x';         // 变量的类型是 char
auto i = 123;          // 变量的类型是 int
auto d = 1.2;          // 变量的类型是 double
auto z = sqrt(y);      // z 的类型是 sqrt(y) 的返回类型
```

我们可以使用 = 的初始化形式与 auto 配合，因为在此过程中不存在可能引发错误的类型转换（见 6.3.6.2 节）。

当我们没有明显的理由需要显式指定数据类型时，一般使用 auto。在这里，“明显的理由”包括：

- 该定义位于一个较大的作用域中，我们希望代码的读者清楚地知道其类型；
- 我们希望明确规定某个变量的范围和精度（比如希望使用 double 而非 float）。

通过使用 auto 可以帮助我们避免冗余，并且无须再书写长类型名。这一点在泛型编程中尤其重要，因为在泛型编程中程序员很难知道对象的确切类型，况且类型名字可能相当长（见 4.5.1 节）。

除了传统的算术运算符和逻辑运算符（见 10.3 节）之外，C++ 还提供了其他一些可用于改变变量值的运算符：

```
x+=y    // x=x+y
++x     // 递增：x = x+1
x-=y    // x=x-y
--x     // 递减：x = x-1
x*=y    // 缩放：x = x*y
x/=y    // 缩放：x = x/y
x%=y    // x=x%y
```

这些运算符简洁明了，被广泛使用。

2.2.3　常量

C++ 支持如下两种不变性概念（见 7.5 节）。

- const：大致意思是“我承诺不改变这个值（见7.5节）”。主要用于说明接口，这样在把变量传入函数时就不必担心变量会在函数内被改变了。编译器负责确认并执行const的承诺。
- constexpr：大致意思是“在编译时求值（见10.4节）”。主要用于说明常量，作用是允许将数据置于只读内存中（不太可能被破坏）以及提升性能。

例如：

```
const int dmv = 17;                                  // dmv是一个命名的常量
int var = 17;                                        // var不是常量
constexpr double max1 = 1.4*square(dmv);             // 如果square(17)是常量表达式，则正确
constexpr double max2 = 1.4*square(var);             // 错误：var不是常量表达式
const double max3 = 1.4*square(var);                 // OK，可在运行时求值
double sum(const vector<double>&);                   // sum不会更改它的参数的值（5.2.5节）
vector<double> v {1.2, 3.4, 4.5};                    // v不是常量
const double s1 = sum(v);                            // OK：在运行时求值
constexpr double s2 = sum(v);                        // 错误：sum(v)不是常量表达式
```

如果某个函数用在常量表达式（constant expression）中，即该表达式在编译时求值，则函数必须定义成constexpr。例如：

```
constexpr double square(double x) { return x*x; }
```

要想定义成constexpr，函数必须非常简单：函数中只能有一条用于计算某个值的return语句。constexpr函数可以接受非常量实参，但此时其结果将不会是一个常量表达式。当程序的上下文不需要常量表达式时，我们可以使用非常量表达式实参来调用constexpr函数，这样我们就不用把同一个函数定义两次了：其中一个用于常量表达式，另一个用于变量。

在有的场合中，常量表达式是语言规则所必需的（如数组的界（见2.2.5节和7.3节）、case标签（见2.2.4节和9.4.2节）、某些模板实参（见25.2节）和使用constexpr声明的常量）。另一些情况下，编译时求值对程序的性能非常重要，所以需要使用常量。即使不考虑性能因素，不变性概念（对象状态不发生改变）也是程序设计中要考虑的一个重要问题（见10.4节）。

2.2.4 检验和循环

C++提供了一套用于表示选择和循环结构的常规语句。例如，下面是一个简单的函数，它首先向用户提问，然后根据用户的回应返回一个布尔值：

```
bool accept()
{
    cout << "Do you want to proceed (y or n)?\n";    // 向用户提问

    char answer = 0;
    cin >> answer;                                   // 读入用户的回答

    if (answer == 'y') return true;
    return false;
}
```

与<<运算符（输出）对应，>>运算符用于输入，cin是标准输入流。>>的右侧运算对象是输入操作的目标，该运算对象的类型决定了>>能够接受什么样的输入。输出字符串末尾的\n字符表示换行（见2.2.1节）。

我们可以进一步完善上面的代码，使其能够处理用户输入 n（表示“no”）的情况：

```
bool accept2()
{
    cout << "Do you want to proceed (y or n)?\n";      // 向用户提问

    char answer = 0;
    cin >> answer;                                     // 读入用户的回答
    switch (answer) {
    case 'y':
        return true;
    case 'n':
        return false;
    default:
        cout << "I'll take that for a no.\n";
        return false;
    }
}
```

switch 语句检查一个值是否存在于一组常量中。case 常量彼此之间不能重复，如果检验值不等于任何 case 常量，则执行 default 分支。如果程序没有提供 default，则当检验值不等于任何 case 常量时什么也不做。

绝大多数程序都含有循环。例如，我们可能允许用户进行多次输入：

```
bool accept3()
{
    int tries = 1;
    while (tries<4) {
        cout << "Do you want to proceed (y or n)?\n";      // 向用户提问
        char answer = 0;
        cin >> answer;                                     // 读入用户的回答

        switch (answer) {
        case 'y':
            return true;
        case 'n':
            return false;
        default:
            cout << "Sorry, I don't understand that.\n";
            ++tries;                                       // 递增
        }
    }
    cout << "I'll take that for a no.\n";
    return false;
}
```

在上面的程序中，while- 语句重复执行直到条件变为 false。

2.2.5　指针、数组和循环

元素类型为 char 的数组可以声明如下：

```
char v[6];          // 含有 6 个字符的数组
```

类似地，指针可以声明如下：

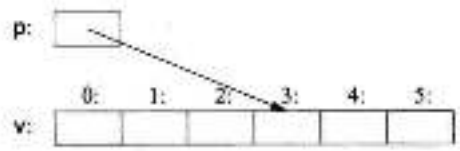
```
char* p;            // 该指针指向字符
```

在声明语句中，[ ] 表示“……的数组”，而 * 表示“指向……”。所有数组的下标都从 0 开始，

因此 v 包含 6 个元素 v[0] 到 v[5]。数组的大小必须是一个常量表达式（见 2.2.3 节）。指针变量中存放着一个相应类型对象的地址：

```
char* p = &v[3];        // p 指向 v 的第 4 个元素
char x = *p;            // *p 是 p 所指向的对象
```

在表达式中，前置一元运算符 * 表示“……的内容”，而前置一元运算符 & 表示“……的地址”。我们可以用下面的图形来表示上述初始化定义的结果：



考虑把一个数组的 10 个元素拷贝给另一个数组：

```
void copy_fct()
{
    int v1[10] = {0,1,2,3,4,5,6,7,8,9};
    int v2[10];                 // 将作为 v1 的副本

    for (auto i=0; i!=10; ++i)  // 拷贝元素
        v2[i]=v1[i];
    // ...
}
```

上面的 for 语句可以这样解读：“把 i 置为 0，当 i 不等于 10 时，拷贝第 i 个元素并递增 i”。当作用于一个整型变量时，递增运算符 ++ 执行简单加 1 的操作。C++ 还提供了一种更简单的 for 语句，即范围 for 语句，它可以用最简单的方式遍历一个序列：

```
void print()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};

    for (auto x : v)            // 对于 v 当中的每个 x
        cout << x << '\n';

    for (auto x : {10,21,32,43,54,65})
        cout << x << '\n';
    // ...
}
```

上面的第一个范围 for 语句可以解读为“对于 v 的每个元素，将其从头到尾依次放入 x 并打印”。注意，当我们使用一个列表初始化数组时无须指定其大小。范围 for 语句可用于任意的元素序列（见 3.4.1 节）。

如果我们不希望把 v 的值拷贝到变量 x 中，而只想令 x 指向一个元素，则可以书写如下的代码：

```
void increment()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};

    for (auto& x : v)
        ++x;
    // ...
}
```

在声明语句中，一元后置运算符 & 表示“……的引用”。引用类似于指针，唯一的区别是我们无须使用前置运算符 * 访问所引用的值。同样，一个引用在初始化之后就不能再引用其他对象了。当用于声明语句时，运算符（如 &、* 和 [ ]）称为声明运算符（declarator operators）：

```
T a[n];    // T[n]: n 个 T 组成的数组 (7.3 节)
T* p;      // T*: 指向 T 的指针 (7.2 节)
T& r;      // T&: T 的引用 (7.7 节)
T f(A);    // T(A): 是一个函数，接受 A 类型的实参，返回 T 类型的结果 (2.2.1 节)
```

我们的目标是确保指针永远指向某个对象，这样解引用该指针的操作就是合法的。当确实没有对象可指或者我们希望表达一种“没有可用对象”的含义时（比如在列表的末尾），我们令指针取值为 nullptr（“空指针”）。所有指针类型都共享同一个 nullptr：

```
double* pd = nullptr;
Link<Record>* lst = nullptr;    // 指向一个 Record 的 Link 的指针
int x = nullptr;                // 错误：nullptr 是个指针，不是整数
```

通常情况下，当我们希望指针实参指向某个东西时，最好检查一下是否确实如此：

```
int count_x(char* p, char x)
    // 统计在 p[] 中 x 出现的次数
    // 假定 p 指向一个字符数组，该数组的结尾处是 0；或者 p 不指向任何东西
{
    if (p==nullptr) return 0;
    int count = 0;
    for (; *p!=0; ++p)
        if (*p==x)
            ++count;
    return count;
}
```

有两点值得注意：一是我们可以使用 ++ 将指针移动到数组的下一个元素；二是在 for 语句中如果不需要初始化操作，则可以省略它。

count_x() 的定义假定 char* 是一个 C 风格字符串，也就是说指针指向了一个字符数组，该数组的结尾处是 0。

在旧式代码中，0 和 NULL 都可以用来替代 nullptr 的功能（见 7.2.2 节）。不过，使用 nullptr 能够避免在整数（如 0 或 NULL）和指针（如 nullptr）之间发生混淆。

2.3　用户自定义类型

我们把可以通过基本类型（见 2.2.2 节）、const 修饰符（见 2.2.3 节）和声明运算符（见 2.2.5 节）构造出的类型称为内置类型（built-in type）。C++ 语言的内置类型及其操作的集合非常丰富，不过相对来说更偏重底层编程。这些内置类型的优点是能够直接有效地展现出传统计算机硬件的特性，但是并不能向程序员提供便于书写高级应用程序的上层特性。为此，C++ 语言扩充了这些内置类型和操作，提供了一套成熟的抽象机制（abstraction mechanism），程序员可以使用这套机制实现其所需的上层功能。C++ 抽象机制的目的主要是让程序员能够设计并实现他们自己的数据类型，这些类型具有恰当的表现形式和操作，程序员可以简单优雅地使用它们。为了与内置类型区别开来，我们把利用 C++ 的抽象机制构建的新类型称为用户自定义类型（user-defined types），诸如类和枚举等等。本书的大部分内容都与用户自定义类型的设计、实现和使用有关。本章的剩余部分将呈现其中最简单也是最基

础的内容。第3章将对抽象机制及其支持的编程风格进行更加详细的描述。第4章和第5章介绍标准库的基本情况，因为标准库主要是由用户自定义类型组成的，所以这两章也从另一个角度阐述了我们到底能用第2章和第3章介绍的语言特性及编程技术完成哪些任务。

2.3.1 结构

构建一种新类型的第一步通常是把所需的元素组织成一种数据结构。下面是一个struct的示例：

```
struct Vector {
    int sz;          // 元素的数量
    double* elem; // 指向元素的指针
};
```

这是Vector的第一个版本，其中包含一个int和一个double*。

一个Vector类型的变量可以通过下述形式进行定义：

```
Vector v;
```

仅就v本身而言，它的用处似乎不大，因为v的elem指针并没有指向任何实际的内容。为了让它变得更有用，我们需要令v指向某些元素。例如，我们可以构造一个如下所示的Vector：

```
void vector_init(Vector& v, int s)
{
    v.elem = new double[s]; // 分配一个数组，它包含s个double值
    v.sz = s;
}
```

也就是说，v的elem成员被赋予了一个由new运算符生成的指针，而sz成员的值则是元素的个数。Vector&中的符号&指定我们通过非常量引用（见2.2.5节和7.7节）的方式传递v，这样vector_init()就能修改传入其中的向量了。

new运算符从一块名为自由存储（free store）（又称为动态内存（dynamic memory）或堆（heap），见11.2节）的区域中分配内存。

Vector的一个简单应用如下所示：

```
double read_and_sum(int s)
    // 从cin读入s个整数，然后返回这些整数的和；其中，假定s是正的
{
    Vector v;
    vector_init(v,s);                  // 为v分配s个元素
    for (int i=0; i!=s; ++i)
        cin>>v.elem[i];                // 读入元素

    double sum = 0;
    for (int i=0; i!=s; ++i)
        sum+=v.elem[i];                // 计算元素的和
    return sum;
}
```

显然，在灵活性和优雅程度上我们的Vector与标准库vector还有很大差距，尤其是Vector的使用者必须清楚地知道它的所有细节。本章余下的部分以及下一章将把Vector当作呈现语言特性和技术的一个示例，一步步地完善它。作为对比，第4章将介绍标准库vector，在其中蕴含着很多漂亮的改进，第31章将结合其他标准库功能呈现完整的vector。

本书使用 vector 和其他标准库组件作为示例，试图达到以下两个目的：

- 展现语言特性和程序设计技术；
- 帮助读者学会使用这些标准库组件。

忠告：与其试着重写 vector 和 string 等标准库组件，不如直接将它们拿来使用。

访问 struct 成员的方式有两种：一种是通过名字或引用，这时我们使用 .（点运算符）；另一种是通过指针，这时用到的是 ->。例如：

```
void f(Vector v, Vector& rv, Vector* pv)
{
    int i1 = v.sz;          // 通过名字访问
    int i2 = rv.sz;         // 通过引用访问
    int i4 = pv->sz;        // 通过指针访问
}
```

2.3.2 类

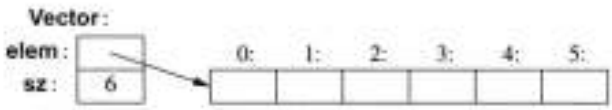
上面这种分割数据及其操作的做法有其优势，比如我们可以非常自由地使用它的数据部分。不过对于用户自定义类型来说，为了将其所有属性捏合在一起，形成一个“真正的类型”，在表示形式和操作之间建立紧密的联系还是很有必要的。特别是，我们常常希望自己构建的类型易于使用和修改，数据的使用具有一致性，并且表示形式最好对用户是不可见的。此时，最理想的做法就是把类型的接口（所有代码都可使用的部分）与其实现（对其他不可访问的数据具有访问权限）分离开来。在 C++ 中，实现上述目的的语言机制被称为类（class）。类含有一系列成员（member），可能是数据、函数或者类型。类的 public 成员定义该类的接口，private 成员则只能通过接口访问。例如：

```
class Vector {
public:
    Vector(int s) :elem{new double[s]}, sz{s} { }   // 构建一个 Vector
    double& operator[](int i) { return elem[i]; }   // 通过下标访问元素
    int size() { return sz; }
private:
    double* elem; // 指向元素的指针
    int sz;       // 元素的数量
};
```

在此基础上，我们定义一个 Vector 类型的变量：

```
Vector v(6);    // 该 Vector 对象含有 6 个元素
```

下图解释了这个 Vector 对象的含义：



总的来说，Vector 对象是一个“句柄”，它包含指向元素的指针（elem）以及元素的数量（sz）。在不同 Vector 对象中元素的数量可能不同（本例是 6），即使同一个 Vector 对象在不同时刻也可能含有不同数量的元素（见 3.2.1.3 节）。不过，Vector 对象本身的大小永远保持不变。这是 C++ 语言处理可变数量信息的一项基本技术：一个固定大小的句柄指向位于“别处”（即通过 new 分配的自由空间，见 11.2 节）的一组可变数量的数据。第 3 章的主题就是学习如何设计并使用这样的对象。

在这里，我们只能通过 Vector 的接口访问其表示形式（成员 elem 和 sz）。Vector 的接口是由其 public 成员提供的，包括 Vector()、operator[]() 和 size()。2.3.1 节的 read_and_sum() 示例可简化为：

```
double read_and_sum(int s)
{
    Vector v(s);                    // 创建一个包含 s 个元素的向量
    for (int i=0; i!=v.size(); ++i)
        cin>>v[i];                  // 读入元素

    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=v[i];                  // 计算元素的和
    return sum;
}
```

与所属类同名的“函数”称为构造函数（constructor），即，它是用来构造类的对象的函数。因此构造函数 Vector() 替换了第 2.3.1 节的 vector_init()。与普通函数不同，构造函数的作用是初始化类的对象，因此定义一个构造函数可以解决类变量未初始化的问题。

Vector(int) 规定了 Vector 类型的对象的构造方式，此处意味着我们需要一个整数来构造对象，这个整数用于指定元素的数量。该构造函数使用成员初始化器列表来初始化 Vector 的成员：

```
:elem{new double[s]}, sz{s}
```

这条语句的含义是：首先从自由空间获取 s 个 double 类型的元素，并用一个指向这些元素的指针初始化 elem；然后用 s 初始化 sz。

访问元素的功能是由一个下标函数提供的，这个函数名为 opeartor[ ]，它的返回值是对相应元素的引用（double&）。

size() 函数的作用是向使用者提供元素的数量。

显然，在上面的代码中完全没有涉及错误处理，与之有关的内容将在 2.4.3 节提及。同样地，我们也没有提供一种机制来“归还”通过 new 获取的 double 数组，3.2.1.2 节将介绍如何使用析构函数来完成这一任务。

2.3.3 枚举

除了类之外，C++ 还提供了另一种简单形式的用户自定义类型，使得我们可以枚举一系列值：

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red };

Color col = Color::red;
Traffic_light light = Traffic_light::red;
```

其中枚举值（如 red）位于其 enum class 的作用域之内，因此我们可以在不同的 enum class 中重复使用这些枚举值而不致引起混淆。例如，Color::red 是指 Color 的 red，它与 Traffic_light::red 显然不同。

枚举类型常用于描述规模较小的整数值集合。通过使用有指代意义的（且易于记忆的）枚举值名字可提高代码的可读性，降低出错的风险。

enum 后面的 class 指明了枚举是强类型的，且它的枚举值位于指定的作用域中。不同的 enum class 是不同的类型，这有助于防止对常量的意外误用。在上面的例子中，我们不能混用 Traffic_light 和 Color 的值：

```
Color x = red;                  // 错误：哪个 red？
Color y = Traffic_light::red;   // 错误：这个 red 不是 Color 的对象
Color z = Color::red;           // OK
```

同样，我们也不能隐式地混用 Color 和整数值：

```
int i = Color::red;             // 错误：Color::red 不是一个 int
Color c = 2;                    // 错误：2 不是一个 Color 对象
```

如果你不想显式地限定枚举值名字，并且希望枚举值可以是 int（无须显式转换），则应该去掉 enum class 中的 class 而得到一个“普通的”enum（见 8.4.2 节）。

默认情况下，enum class 只定义了赋值、初始化和比较（即 == 和 <，见 2.2.2 节）操作。然而，既然枚举类型是一种用户自定义类型，那么我们就可以为它定义别的运算符：

```
Traffic_light& operator++(Traffic_light& t)
    // 前置递增运算符 ++
{
    switch (t) {
    case Traffic_light::green:      return t=Traffic_light::yellow;
    case Traffic_light::yellow:     return t=Traffic_light::red;
    case Traffic_light::red:        return t=Traffic_light::green;
    }
}

Traffic_light next = ++light;       // next 变成了 Traffic_light::green
```

C++ 也提供了次强类型的“普通的”enum（见 8.4.2 节）。

2.4 模块化

一个 C++ 程序可能包含许多独立开发的部分，例如函数（见 2.2.1 节和第 12 章）、用户自定义类型（见 2.3 节，3.2 节和第 16 章）、类层次（见 3.2.4 节和第 20 章）和模板（见 3.4 节和第 23 章）等。因此构建 C++ 程序的关键就是清晰地定义这些组成部分之间的交互关系。第一步也是最重要的一步，是将某个部分的接口和实现分离开来。在语言层面，C++ 使用声明来描述接口。声明（declaration）指定了使用某个函数或某种类型所需的所有内容。例如：

```
double sqrt(double);    // 这个平方根函数接受一个 double，返回值也是一个 double

class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem; // elem 指向一个数组，该数组包含 sz 个 double
    int sz;
};
```

这里的关键点是函数体，即函数的定义（definition）位于“其他某处”。在此例中，我们可能也想让 Vector 的描述位于“其他某处”，不过，我们将稍后再介绍相关内容（抽象类型，见 3.2.2 节）。sqrt() 的定义形如下面的形式：

```
double sqrt(double d)          // sqrt() 的定义
{
    // ……求解平方根的算法，与数学教科书中的非无二致……
}
```

对于 Vector 来说，我们需要定义全部 3 个成员函数：

```
Vector::Vector(int s)                    // 构造函数的定义
    :elem{new double[s]}, sz{s}          // 初始化成员
{
}

double& Vector::operator[](int i)        // 下标运算符的定义
{
    return elem[i];
}

int Vector::size()                       // size() 的定义
{
    return sz;
}
```

我们必须定义 Vector 的函数，而不必定义 sqrt()，因为它是标准库的一部分。但是这没什么本质区别：库其实就是一些“我们碰巧用到的其他代码”，编写这些代码的语言功能与我们平常使用的那些没什么区别。

2.4.1 分离编译

C++ 支持一种名为分离编译的概念，用户代码只能看见所用类型和函数的声明，它们的定义则放置在分离的源文件里，并被分别编译。这种机制有助于将一个程序组织成一组半独立的代码片段。其优点是编译时间减到最少，并且强制要求程序中逻辑独立的部分分离开来（从而将发生错误的几率降到最低）。一个库通常是一组分离编译的代码片段（如函数）的集合。

一般情况下，我们把描述模块接口的声明放置在一个特定的文件中，文件名常常指示模块的预期用途。例如：

```
// Vector.h:

class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem;      // elem 指向一个数组，该数组包含 sz 个 double
    int sz;
};
```

这段声明被置于文件 Vector.h 中，我们称这种文件为头文件（header file），用户将其包含（include）进程序以访问接口。例如：

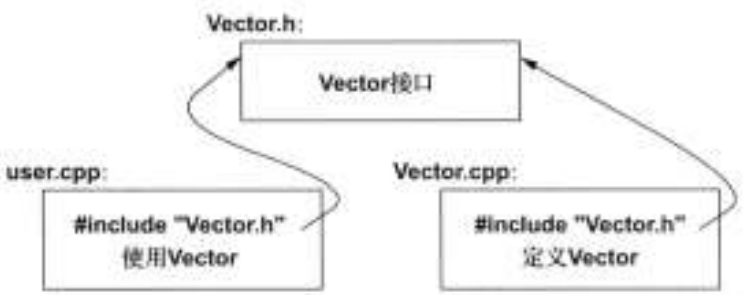
```
// user.cpp:

#include "Vector.h"        // 获得 Vector 的接口
#include <cmath>           // 获得标准库数学函数接口，其中含有 sqrt()
using namespace std;       // 令 std 成员可见（见 2.4.2 节）
```

```
double sqrt_sum(Vector& v)
{
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=sqrt(v[i]);                    // 平方根的和
    return sum;
}
```

为了帮助编译器确保一致性，负责提供 Vector 实现部分的 .cpp 文件同样应该包含提供接口的 .h 文件：

```
// Vector.cpp:

#include "Vector.h" // 获得接口

Vector::Vector(int s)
    :elem{new double[s]}, sz{s}
{
}

double& Vector::operator[](int i)
{
    return elem[i];
}

int Vector::size()
{
    return sz;
}
```

user.cpp 和 Vector.cpp 中的代码共享 Vector.h 提供的 Vector 接口信息，但这两个文件是相互独立的，可以被分离编译。上述程序片段用图形化的方式呈现如下：



严格来说，使用分离编译并不是一个语言问题；而是关于如何以最佳方式利用特定语言实现的问题。不管怎么说，分离编译机制在实际的编程过程中非常重要。最好的方法是最大限度地模块化，逻辑上通过语言特性描述模块，而后在物理上通过划分文件及高效分离编译来充分利用模块化（见第 14、15 章）。

2.4.2 名字空间

在函数（见 2.2.1 节和第 12 章）、类（见第 16 章）和枚举（见 2.3.3 节和 8.4 节）之外，C++ 还提供了一种称为名字空间（namespace，见第 14 章）的机制，一方面表达某些声明是属于一个整体的，另一方面表明它们的名字不会与其他名字空间中的名字冲突。例如，我们尝试利用自己定义的复数类型（见 3.2.1.1 节，18.3 节和 40.4 节）进行实验：

```
namespace My_code {
    class complex { /* ... */ };
    complex sqrt(complex);
    // ...
    int main();
}

int My_code::main()
{
    complex z {1,2};
    auto z2 = sqrt(z);
    std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
    // ...
};

int main()
{
    return My_code::main();
}
```

通过将代码放在名字空间 My_code 中，就可以确保我们的名字不会和名字空间 std（见 4.1.2 节）中的标准库名字冲突。因为标准库确实支持 complex 算术运算（见 3.2.1.1 节和 40.4 节），所以提前设置这样的预防措施显然是非常明智的。

要想访问其他名字空间中的某个名字，最简单的方法是在这个名字前加上名字空间的名字作为限定（例如 std::cout 和 My_code::main）。"真正的 main()" 定义在全局名字空间中，换句话说，它不属于任何自定义名字空间、类或函数。要想获取标准库名字空间中名字的访问权，我们应该使用 using 指示（见 14.2.3 节）：

```
using namespace std;
```

名字空间主要用于组织较大规模的程序组件，最典型的例子是库。使用名字空间，我们就可以很容易地把若干独立开发的部件组织成一个程序。

2.4.3 错误处理

错误处理是一个略显繁杂的主题，它的内容和影响都远远超越了语言特性的层面，而应被归结为程序设计技术和工具的范畴。不过 C++ 还是提供了一些有益的功能，其中最主要的一个工具就是类型系统本身。在构建应用程序时，通常的做法不是仅仅依靠内置类型（如 char、int 和 double）和语句（如 if、while 和 for），而是建立更多适合应用的新类型（如 string、map 和 regex）和算法（如 sort()、find_if() 和 draw_all()）。这些高层次的结构简化了程序设计，减少了产生错误的机会（你大概不会把遍历树的算法应用在对话框上），同时也增加了编译器捕获错误的概率。大多数 C++ 的结构都致力于设计并实现优雅而高效的抽象模型（例如用户自定义类型以及基于这些自定义类型的算法）。这种模块化和抽象机制（特别是库的使用）的一个重要影响就是运行时错误的捕获位置与错误处理的位置被分离开来。随着程序规模不断增长，特别是库的应用越来越广泛，处理错误的规范和标准变得愈加重要。

2.4.3.1 异常

让我们重新考虑 Vector 的例子。对 2.3.2 节中的向量，当我们试图访问某个越界的元素时，应该做什么呢？

- Vector 的作者并不知道使用者在面临这种情况时希望如何处理（通常情况下，Vector 的作者甚至不知道向量将在何种程序场景中使用）。
- Vector 的使用者不能保证每次都检测到问题（如果他们能做到的话，越界访问也就不会发生了）。

因此最佳的解决方案是由 Vector 的实现者负责检测可能的越界访问，然后通知使用者。之后 Vector 的使用者可以采取适当的应对措施。例如，Vector::operator[]() 能够检测到潜在的越界访问错误并且抛出一个 out_of_range 异常：

```
double& Vector::operator[](int i)
{
    if (i<0 || size()<=i) throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

throw 负责把程序的控制权从某个直接或间接调用了 Vector::operator[]() 的函数转移到 out_of_range 异常处理代码。为了完成这一目标，实现部分需要展开函数调用栈以便返回主调函数的上下文（见 13.5.1 节）。例如：

```
void f(Vector& v)
{
    // ...
    try { // 此处的异常将被后面定义的处理模块处理

        v[v.size()] = 7; // 试图访问 v 末尾之后的位置
    }
    catch (out_of_range) {    // 糟糕！发生了越界错误
        // ... 在此处处理越界错误 ...
    }
    // ...
}
```

我们把可能处理异常的程序放在一个 try 块当中。显然，对 v[v.siz e()] 的赋值操作将出错。因此，程序进入到提供了 out_of_range 错误处理代码的 catch 从句中。out_of_range 类型定义在标准库中（在 <stdexcept> 中），事实上，它也被一些标准库容器访问函数使用。

通过使用异常处理机制，错误处理变得更简单，条理性和可读性也得到了加强。第 13 章将就相关内容进行更深入的探讨，也会介绍更多细节和示例。

2.4.3.2　不变式

使用异常机制通报越界访问错误是函数检查实参的一个示例，此时，因为基本假设，即所谓的前置条件（precondition）没有得到满足，所以函数将拒绝执行。在正式说明 Vector 的下标运算符时，我们应该规定类似于“索引值必须在 [0:size()) 范围内”的规则，这一规则是在 operator[]() 内被检查的。无论什么时候只要我们试图定义一个函数，就应该考虑它的前置条件是什么，以及检验该条件的过程是否足够简洁（见 12.4 节和 13.4 节）。

然而在上面的定义中，operator[]() 作用于 Vector 类型的对象并且只在 Vector 的成员有“合理”的值时它才有意义。特别是，我们说过“elem 指向一个含有 sz 个 double 的数组”，但这只是注释中的说明而已。对于类来说，这样一条假定某事为真的声明称为类的不变式（class invariant），简称为不变式（invariant）。建立类的不变式是构造函数的任务（从而成员函数可以依赖于该不变式），它的另一个作用是确保当成员函数退出时不变式仍然成立。不幸的是，我们的 Vector 构造函数只履行了一部分职责。它正确地初始化了 Vector 成员，

但是没有检验传入的实参是否有效。考虑如下情况：

```
Vector v(-27);
```

这条语句很可能会引起混乱。

与原来的版本相比，下面的定义更好：

```
Vector::Vector(int s)
{
    if (s<0) throw length_error{};
    elem = new double[s];
    sz = s;
}
```

我使用标准库异常 length_error 报告元素数目为非正数的错误，因为一些标准库操作也是这么做的。如果 new 运算符找不到可分配的内存，就会抛出 std::bad_alloc。我们可以接着书写：

```
void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error) {
        // 处理负值问题
    }
    catch (std::bad_alloc) {
        // 处理内存耗尽问题
    }
}
```

你可以自定义异常类，然后让它们把指定的信息从检测到异常的点传递到处理异常的点（见13.5节）。

通常情况下，当遇遇异常后函数就无法继续完成工作了。此时，“处理”异常的含义仅仅是做一些简单的局部资源清理，然后重新抛出异常。

不变式的概念是设计类的关键，而前置条件也在设计函数的过程中起到同样的作用。不变式

- 帮助我们准确地理解想要什么；
- 强制要求具体而明确地描述设计，而这有助于确保代码正确（在调试和测试之后）。

不变式的概念是C++当中由构造函数（见2.3.2节）和析构函数（见3.2.1.2节和5.2节）支撑的资源管理概念的基础，相关内容在13.4节、16.3.1节和17.2节还会有详细介绍。

2.4.3.3 静态断言

程序异常负责报告运行时发生的错误。如果我们能在编译时发现错误，显然效果更好。这是大多数类型系统以及自定义类型接口的主要目的。不过，我们也能对其他一些编译时可知的属性做一些简单检查，并以编译器错误消息的形式报告所发现的问题。例如：

```
static_assert(4<=sizeof(int), "integers are too small");  // 检查整数的尺寸
```

如果4<=siz eof(int)不满足，将输出integers are too small的信息。也就是说，如果在当前系统上一个int占有的空间不足4个字节，就会报错。我们把这种表达某种期望的语句称为断言（assertion）。

static_assert机制能用于任何可以表达为常量表达式的东西（见2.2.3节和10.4节）。

例如：

```
constexpr double C = 299792.458;                    // km/s

void f(double speed)
{
    const double local_max = 160.0/(60*60);         // 160 km/h == 160.0/(60*60) km/s

    static_assert(speed<C,"can't go that fast");    // 错误：速度必须是个常量
    static_assert(local_max<C,"can't go that fast"); // OK

    // ...
}
```

通常情况下，static_asser t(A,S) 的作用是当 A 不为 true 时，把 S 作为一条编译器错误信息输出。

static_assert 最重要的用途是为泛型编程中作为形参的类型设置断言（见 5.4.2 节和 24.3 节）。

关于运行时检查的断言，请见 13.4 节。

2.5　附记

本章涉及的主题与第二部分（第 6 ~ 15 章）的内容大致契合，它们是 C++ 所有程序设计技术的基础。有经验的 C 和 C++ 程序员请注意，这部分基础知识并不密切对应 C++（即 C++11）的 C 和 C++98 子集。

2.6　建议

［1］ 不必慌张，一切知识都会随着时间推移变得逐渐清晰；2.1 节。

［2］ 即使你没有掌握 C++ 的所有细节，也能写出漂亮的程序；1.3.1 节。

［3］ 请关注编程技术，而非语言特性；2.1 节。

第3章

The C++ Programming Language, Fourth Edition

C++概览：抽象机制

不必惊慌失措！

——道格拉斯·亚当斯

- 引言
- 类
  具体类型；抽象类型；虚函数；类层次
- 拷贝和移动
  拷贝容器；移动容器；资源管理；抑制操作
- 模板
  参数化类型；函数模板；函数对象；可变参数模板；别名
- 建议

3.1 引言

本章的目标是在不涉及过多细节的前提下向读者展现C++是如何支持抽象和资源管理的。我们将会介绍如何定义并使用新类型（用户自定义类型），重点介绍与具体类、抽象类和类层次结构有关的基本属性、实现技术以及语言特性。模板是一种用（其他）类型和算法对类型和算法进行参数化的机制。用户自定义类型与内置类型的计算表现为函数，有时泛化为模板函数和函数对象。这些语言特性用于支持面向对象编程和泛型编程等编程风格。接下来的两章将展示一些标准库功能及其用法的例子。

在阅读本章之前，你最好已经有一些编程经验。如果没有，建议读者先找一本入门教材学习一下，比如《Programming：Principles and Practice Using C++》[Stroustrup，2009]。即便编写过程序，你使用的语言或编写的应用也可能在风格或形式上与本书展示的C++风格相距甚远。因此，如果你发现接下来的“快速导览”不那么容易理解，不妨直接跳到第6章，从那儿开始我们将对知识介绍得更加系统和有条理。

与第2章一样，本章的概览试图把C++作为一个整体呈现在读者面前，而非像千层糕一样一层一层地介绍。因此，在这里我们不会细分某项语言特性归属于C、C++98还是C++11。这些语言的历史信息在1.4节和第44章可以找到。

3.2 类

C++最核心的语言特性就是类。类是一种用户自定义的数据类型，用于在程序代码中表示某种概念。无论何时，只要我们想为程序设计一个有用的概念、想法或实体，都应该设法把它表示为程序中的一个类，这样我们的想法就能表达成代码，而不是仅存在于我们的头脑中、设计文档里或者注释里。对于一个程序来说，不论是用易读性还是正确性来衡量，使用一组精挑细选的类都要比直接用内置类型完成所有任务更好，尤其是当选用由库提供的类

时更是如此。

从本质上来说，基础类型、运算符和语句之外的所有语言特性存在的目的就是帮助我们定义更好的类以及更方便地使用它们。在这里，“更好”的意思包括更加正确、更容易实现、更有效率、更优雅、更易用以及更易推断。大多数编程技术依赖于某些特定类的设计与实现。程序员要完成的任务千差万别，因此对类的支持也应该是宽泛和丰富的。接下来，我们只考虑对三种重要的类的基本支持：

- 具体类（见 3.2.1 节）；
- 抽象类（见 3.2.2 节）；
- 类层次中的类（见 3.2.4 节）。

很多有用的类都可以归为这三个类别，其他类也可以看成是这些类别的简单变形或组合。

3.2.1 具体类型

具体类的基本思想是它们的行为“就像内置类型一样”。例如，一个复数类型和一个无穷精度整数与内置的 int 非常相像，当然它们有自己的语义和操作集合。同样，vector 和 string 也很像内置的数组，只不过在可操作性上更胜一筹（见 4.2 节、4.3.2 节和 4.4.1 节）。

具体类型的典型特征是，它的表现形式是其定义的一部分。在很多重要例子（如 vector）中，表现形式只不过是一个或几个指向保存在别处的数据的指针，但这种表现形式出现在具体类的每一个对象中。这使得实现可以在时空上达到最优，尤其是它允许我们：

- 把具体类型的对象置于栈、静态分配的内存或者其他对象中（见 6.4.2 节）；
- 直接引用对象（而非仅仅通过指针或引用）；
- 创建对象后立即进行完整的初始化（比如使用构造函数，见 2.3.2 节）；
- 拷贝对象（见 3.3 节）。

类的表现形式可以被限定为私有的（就像 Vector 一样，见 2.3.2 节），只能通过成员函数访问，但它确实存在。因此，一旦表现形式发生了任何明显的改动，使用者就必须重新编译。这也是我们试图使具体类型尽可能接近内置类型而必须付出的代价。对于那些不常改动的类型和那些局部变量提供了迫切需要的清晰性和效率的类型来说，这种特性是可以接受的，而且通常会很理想。如果想提高灵活性，具体类型可以将其表现形式的主要部分放置在自由存储（动态内存、堆）中，然后通过存储在类对象内部的另一部分访问它们。vector 和 string 的实现机理正是如此，我们可以把它们看做是带有精致接口的资源管理器。

3.2.1.1 一种算术类型

一种“经典的用户自定义算术类型”是 complex：

```
class complex {
    double re, im; // 表现形式：两个双精度浮点数
public:
    complex(double r, double i) :re{r}, im{i} {}    // 由两个标量构建该复数
    complex(double r) :re{r}, im{0} {}              // 由一个标量构建该复数
    complex() :re{0}, im{0} {}                      // 默认的复数是 {0,0}

    double real() const { return re; }
    void real(double d) { re=d; }
    double imag() const { return im; }
    void imag(double d) { im=d; }
```

```
    complex& operator+=(complex z) { re+=z.re, im+=z.im; return *this; }   // 加到 re
                                                                            // 和 im 上然后返回
    complex& operator-=(complex z) { re-=z.re, im-=z.im; return *this; }

    complex& operator*=(complex);   // 在类外的某处进行定义
    complex& operator/=(complex);   // 在类外的某处进行定义
};
```

这是对标准库 complex（见 40.4 节）略作简化后的版本，类定义本身仅包含需要访问其表现形式的操作。它的表现形式非常简单，也是大家约定俗成的。由于编程实践的要求，它必须兼容 50 年前 Fortran 语言提供的版本，还需要一些常规的运算符。除了满足逻辑上的要求外，complex 还必须足够高效，否则依然没有实用价值。这意味着我们应该将简单的操作设置成内联的。也就是说，在最终生成的机器代码中，一些简单的操作（如构造函数、+= 和 imag() 等）不能以函数调用的方式实现。定义在类内部的函数默认是内联的。一个工业级的 complex（就像标准库中的那个一样）必须精心实现，并且恰当地使用内联。

无须实参就可以调用的构造函数称为默认构造函数，complex() 是 complex 的默认构造函数。通过定义默认构造函数，可以有效防止该类型的对象未初始化。

在负责返回复数实部和虚部的函数中，const 说明符表示这两个函数不会修改所调用的对象。

很多有用的操作并不需要直接访问 complex 的表现形式，因此它们的定义可以与类的定义分离开来：

```
complex operator+(complex a, complex b) { return a+=b; }
complex operator-(complex a, complex b) { return a-=b; }
complex operator-(complex a) { return {-a.real(), -a.imag()}; }     // 一元负号
complex operator*(complex a, complex b) { return a*=b; }
complex operator/(complex a, complex b) { return a/=b; }
```

此处我们利用了 C++ 的一个特性，即，以传值方式传递实参实际上是把一份副本传递给函数，因此我们修改形参（副本）不会影响主调函数的实参，并可以将结果作为返回值。

== 和 != 的定义非常直观且易于理解：

```
bool operator==(complex a, complex b)       // 相等
{
    return a.real()==b.real() && a.imag()==b.imag();
}

bool operator!=(complex a, complex b)       // 不等
{
    return !(a==b);
}

complex sqrt(complex);

// ...
```

我们可以像下面这样使用 complex：

```
void f(complex z)
{
    complex a {2.3};            // 用 2.3 构建出 {2.3, 0.0}
    complex b {1/a};
    complex c {a+z*complex{1,2.3}};
    // ...
```

```
    if (c != b)
        c = -(b/a)+2*b;
}
```

编译器自动地把计算 complex 值的运算符转换成对应的函数调用，例如 c!=b 意味着 operator!=(c,b)，而 1/a 意味着 operator/(complex{1},a)。

在使用用户自定义的运算符（“重载运算符”）时，我们应该尽量小心谨慎，并且尊重其常规的使用习惯。你不能定义一元运算符 /，因为其语法在语言中已被固定。同样，你不可能改变一个运算符操作内置类型时的含义，因此不能重新定义运算符 + 令其执行 int 的减法。

3.2.1.2　容器

容器（container）是指一个包含若干元素的对象，因为 Vector 的对象都是容器，所以我们称 Vector 是一种容器类型。如 2.3.2 节中定义，Vector 作为 double 的容器具有许多优点：它易于理解，建立有用的不变式（见 2.4.3.2 节），提供了包含边界检查的访问功能（见 2.4.3.1 节），并且提供了 size() 以允许我们遍历其元素。然而，它还是存在一个致命的缺陷：它使用 new 分配了元素但是从来没有释放这些元素。这显然是个糟糕的设计，因为尽管 C++ 定义了一个垃圾回收的接口（见 34.5 节），可将未使用的内存提供给新对象，但 C++ 并不保证垃圾收集器总是可用的。在某些情况下你不能使用回收功能，而且有的时候出于逻辑或性能的考虑你宁愿使用更精确的资源释放控制（见 13.6.4 节）。因此，我们迫切需要一种机制以确保构造函数分配的内存一定会被销毁，这种机制就叫做析构函数（destructor）：

```
class Vector {
private:
    double* elem;       // elem 指向一个包含 sz 个 double 的数组
    int sz;
public:
    Vector(int s) :elem{new double[s]}, sz{s}       // 构造函数：请求资源
    {
        for (int i=0; i!=s; ++i) elem[i]=0;         // 初始化元素
    }

    ~Vector() { delete[] elem; }                    // 析构函数：释放资源

    double& operator[](int i);
    int size() const;
};
```

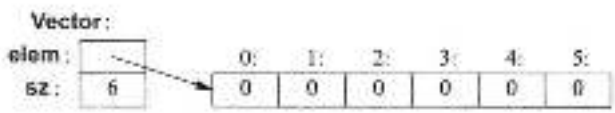
析构函数的命名规则是一个求补运算符 ~ 后接类的名字，从含义上来说它是构造函数的补充。Vector 的构造函数使用 new 运算符从自由存储（也称为堆或动态存储）分配一些内存空间，析构函数则使用 delete 运算符释放该空间以达到清理资源的目的。这一切都无须 Vector 的使用者干预，他们只需要像使用普通的内置类型变量那样使用 Vector 对象就可以了。例如：

```
void fct(int n)
{
    Vector v(n);

    // ... 使用 v ...

    {
        Vector v2(2*n);
```

```
        // ... 使用 v 和 v2 ...
    } // v2 在此处被销毁

    // ... 使用 v ...

} // v 在此处被销毁
```

Vector 与 int 和 char 等内置类型遵循同样的命名、作用域、分配空间、生命周期等规则。6.4 节将介绍如何控制对象的生命周期。另外出于简化的考虑，Vector 没有涉及错误处理，相关内容可以参考 2.4.3 节。

构造函数/析构函数的机制是很多优雅技术的基础，尤其是大多数 C++ 通用资源管理技术（见 5.2 节和 13.3 节）的基础。以下是一个 Vector 的图示：



构造函数负责分配元素空间并正确地初始化 Vector 成员，析构函数则负责释放空间。这就是所谓的数据句柄模型（handle-to-data model），常用来管理在对象生命周期中大小会发生变化的数据。在构造函数中请求资源，然后在析构函数中释放它们的技术称为资源获取即初始化（Resource Acquisition Is Initialization），简称 RAII，它使得我们得以规避“裸 new 操作”的风险；换句话说，该技术可以避免在普通代码中分配内存，而是把分配操作隐藏在行为良好的抽象的实现内部。同样，也应该避免“裸 delete 操作”。避免裸 new 和裸 delete 可以使我们的代码远离各种潜在风险，避免资源泄漏（见 5.2 节）。

3.2.1.3 初始化容器

容器的作用是保存元素，因此我们需要找到一种便利的方式将元素存入容器中。为了做到这一点，一种可能的方式是先用若干元素创建一个 Vector，然后再依次为这些元素赋值。显然这不是最好的办法，下面列举两种更简洁的途径。

- *初始化器列表构造函数*（Initializer-list constructor）：使用元素的列表进行初始化；
- push_back()：在序列的末尾添加一个新元素。

它们的声明形式如下所示：

```
class Vector {
public:
    Vector(std::initializer_list<double>);      // 使用一个列表进行初始化
    // ...
    void push_back(double);                     // 在末尾添加一个元素，容器的长度加 1
    // ...
};
```

其中，push_back() 可用于添加任意数量的元素。例如：

```
Vector read(istream& is)
{
    Vector v;
    for (double d; is>>d;)          // 将浮点值读入 d
        v.push_back(d);             // 把 d 加到 v 当中
    return v;
}
```

上面的循环负责执行输入操作，它的终止条件是遇到文件末尾或者格式错误。在此之

前，每个读入的数依次添加到 Vector 的尾部，最后 v 的大小就是读取的元素数量。我们使用了一个 for 语句而不是 while 语句以便将 d 的作用域限制在循环内部。push_back() 的具体实现将在 13.6.4.3 节介绍。3.3.2 节则将介绍移动构造函数，可以从 read() 返回非常巨大的数据量而代价又很低。

用于定义初始化器列表构造函数的 std::initializer_list 是一种标准库类型，编译器可以辨识它：当我们使用 {} 列表时，如 {1,2,3,4}，编译器会创建一个 initializer_list 类型的对象并将其提供给程序。因此，我们可以书写：

```
Vector v1 = {1,2,3,4,5};              // v1 包含 5 个元素
Vector v2 = {1.23, 3.45, 6.7, 8};     // v2 包含 4 个元素
```

Vector 的初始化器列表构造函数可以定义成如下的形式：

```
Vector::Vector(std::initializer_list<double> lst)     // 用一个列表初始化
    :elem{new double[lst.size()]}, sz{lst.size()}
{
    copy(lst.begin(),lst.end(),elem);          // 从 lst 复制内容到 elem 中
}
```

3.2.2　抽象类型

complex 和 Vector 等类型之所以被称为具体类型（concrete type），是因为它们的表现形式属于定义的一部分。在这一点上，它们与内置类型很相似。相反，抽象类型（abstract type）则将使用者与类的实现细节完全隔离开来。为了做到这一点，我们分离接口与表现形式并且放弃了纯局部变量。因为我们对抽象类型的表现形式一无所知（甚至对它的大小也不了解），所以必须从自由存储（见 3.2.1.2 节和 11.2 节）为对象分配空间，然后通过引用或指针（见 2.2.5 节，7.2 节和 7.7 节）访问对象。

首先，我们为 Container 类设计接口，Container 类可以看成是比 Vector 更抽象的一个版本：

```
class Container {
public:
    virtual double& operator[](int) = 0;      // 纯虚函数
    virtual int size() const = 0;              // 常量成员函数（见 3.2.1.1 节）
    virtual ~Container() {}                    // 析构函数（见 3.2.1.2 节）
};
```

对于后面定义的那些特定容器来说，上面这个类是个纯粹的接口。关键字 virtual 的意思是“可能随后在其派生类中重新定义”。意料之中，我们把这种用关键字 virtual 声明的函数称为虚函数（virtual function）。Container 的派生类负责为 Container 接口提供具体实现。看起来有点奇怪的 =0 说明该函数是纯虚函数，意味着 Container 的派生类必须定义这个函数。因此，我们不能单纯定义一个 Container 的对象，Container 只是作为接口出现，它的派生类负责具体实现 operator[]() 和 size()。含有纯虚函数的类称为抽象类（abstract class）。

Container 的用法是：

```
void use(Container& c)
{
    const int sz = c.size();

    for (int i=0; i!=sz; ++i)
        cout << c[i] << '\n';
}
```

请注意 use() 是如何在完全忽视实现细节的情况下使用 Container 接口的。它使用了 size() 和 [ ]，却根本不知道是哪个类型实现了它们。如果一个类负责为其他一些类提供接口，那么我们把前者称为多态类型（polymorphic type，见 20.3.2 节）。

作为一个抽象类，在 Container 中没有构造函数，毕竟它没有什么数据需要初始化。另一方面，Container 含有一个析构函数，而且该析构函数是 virtual 的。这也不难理解，因为抽象类需要通过引用或指针来操纵，而当我们试图通过一个指针销毁 Container 时，我们并不清楚它的实现部分到底拥有哪些资源，关于这一点在 3.2.4 节有详细介绍。

一个容器为了实现抽象类 Container 接口所需的函数，可以使用具体类 Vector：

```
class Vector_container : public Container {    // Vector_container 实现了 Container
    Vector v;
public:
    Vector_container(int s) : v(s) { }    // 含有 s 个元素的 Vector
    ~Vector_container() {}

    double& operator[](int i) { return v[i]; }
    int size() const { return v.size(); }
};
```

:public 可读作“派生自”或“是…的子类型”。我们说 Vector_container 类派生（derived）自 Container 类，而 Container 类是 Vector_container 类的基类（base）。还有另外一种叫法，分别把 Vector_container 和 Container 叫做子类（subclass）和超类（superclass）。派生类从它的基类继承成员，所以我们通常把基类和派生类的这种关联关系叫做继承（inheritance）。

成员 operator[]() 和 size() 覆盖（override）了基类 Container 中对应的成员（见 20.3.2 节）。析构函数 ~Vector_container() 则覆盖了基类的析构函数 ~Container()。注意，成员 v 的析构函数（~Vector()）被其类的析构函数（~Vector_container()）隐式调用。

对于像 use(Container&) 这样的函数来说，可以在完全不了解一个 Container 实现细节的情况下使用它，但还需另外某个函数（g）为其创建可供操作的对象。例如：

```
void g()
{
    Vector_container vc {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
    use(vc);
}
```

因为 use() 只知道 Container 的接口而不了解 Vector_container，因此对于 Container 的其他实现，use() 仍能正常工作。例如：

```
class List_container : public Container { // List_container 实现了 Container
    std::list<double> ld;    // 一个 double 类型的标准库 list（见 4.4.2 节）
public:
    List_container() { }    // 空列表
    List_container(initializer_list<double> il) : ld{il} { }
    ~List_container() {}
    double& operator[](int i);
    int size() const { return ld.size(); }

};

double& List_container::operator[](int i)
{
    for (auto& x : ld) {
```

```
            if (i==0) return x;
            --i;
        }
        throw out_of_range("List container");
    }
```

在这段代码中，类的表现形式是一个标准库 list<double>。一般情况下，我们不会用 list 实现一个带下标的容器，毕竟 list 取下标的性能很难与 vector 相比。不过，本例中我们只是用它完成一个与前一个实现完全不同的版本。

我们可以通过一个函数创建一个 List_container，然后让 use() 使用它：

```
void h()
{
    List_container lc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    use(lc);
}
```

这段代码的关键点是 use(Container&) 并不清楚它的实参是 Vector_container、List_container，还是其他什么容器，它也根本不需要知道。它只要了解 Container 定义的接口就可以了。因此，不论 List_container 的实现发生了改变还是我们使用了 Container 的一个全新派生类，都不需要重新编译 use(Container&)。

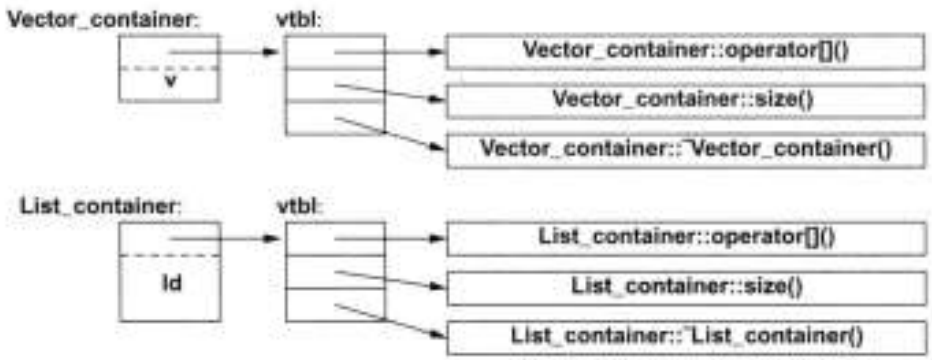
灵活性背后唯一的不足是，我们只能通过引用或指针操作对象（见 3.3 节和 20.4 节）。

3.2.3　虚函数

我们进一步思考 Container 的用法：

```
void use(Container& c)
{
    const int sz = c.size();

    for (int i=0; i!=sz; ++i)
        cout << c[i] << '\n';
}
```

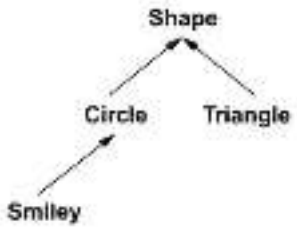
一个有趣的问题是：use() 中的 c[i] 是如何解析到正确的 operator[]() 的？当 h() 调用 use() 时，必须调用 List_container 的 operator[]()；而当 g() 调用 use() 时，必须调用 Vector_container 的 operator[]()。要想达到这种效果，Container 对象就必须包含一些有助于它在运行时选择正确函数的信息。常见的做法是编译器将虚函数的名字转换成函数指针表中对应的索引值，这张表就是所谓的虚函数表（virtual function table）或简称为 vtbl。每个含有虚函数的类都有它自己的 vtbl 用于辨识虚函数，其工作机理如下图所示：



即使调用函数不清楚对象的大小和数据布局，vtbl 中的函数也能确保对象被正确使用。调用函数的实现只需要知道 Container 中 vtbl 指针的位置以及每个虚函数对应的索引就可以了。这种虚调用机制的效率非常接近“普通函数调用”机制（相差不超过 25%），而它的空间开销包括两部分：如果类包含虚函数，则该类的每个对象需要一个额外的指针；另外每个这样的类需要一个 vtbl。

3.2.4 类层次

Container 是一个非常简单的类层次的例子，所谓类层次（class hierarchy）是指通过派生（如 :public）创建的一组类，在框架中有序排列。我们使用类层次表示具有层次关系的概念，比如“消防车是卡车的一种，卡车是车辆的一种”以及“笑脸是一个圆，圆是一个形状”。在实际应用中，大的类层次动辄包含上百个类，不论深度还是宽度都很大。不过在本节，我们只考虑一个半写实的小例子，那就是屏幕上的形状：



箭头表示继承关系。例如，Circle 类派生自 Shape 类。要想把上面这个简单的图例写成代码，我们首先需要说明一个类，令其定义所有这些形状的公共属性：

```
class Shape {
public:
    virtual Point center() const =0;        // 纯虚函数
    virtual void move(Point to) =0;

    virtual void draw() const = 0;          // 在当前“画布”上绘制
    virtual void rotate(int angle) = 0;

    virtual ~Shape() {}                     // 析构函数
    // ...
};
```

这个接口显然是一个抽象类：对于每种 Shape 来说，它们的表现形式基本上各不相同（除了 vtbl 指针的位置之外）。基于上面的定义，我们就能编写函数令其操纵由形状指针组成的向量了：

```
void rotate_all(vector<Shape*>& v, int angle) // 将 v 的元素按照指定角度旋转
{
    for (auto p : v)
        p->rotate(angle);
}
```

要定义一种具体的形状，首先必须指明它是一个 Shape，然后再规定其特有的属性（包括虚函数）：

```
class Circle : public Shape {
public:
    Circle(Point p, int rr);            // 构造函数
```

```
    Point center() const { return x; }
    void move(Point to) { x=to; }

    void draw() const;
    void rotate(int) {}                    // 一个简单明了的示例算法
private:
    Point x;    // 圆心
    int r;      // 半径
};
```

到目前为止，Shape 和 Circle 涉及的语法知识并不比 Container 和 Vector_container 多多少，但是我们可以继续构造：

```
class Smiley : public Circle {  // 使用 Circle 作为笑脸的基类
public:
    Smiley(Point p, int r) : Circle{p,r}, mouth{nullptr} { }

    ~Smiley()
    {
        delete mouth;
        for (auto p : eyes) delete p;
    }
    void move(Point to);

    void draw() const;
    void rotate(int);

    void add_eye(Shape* s) { eyes.push_back(s); }
    void set_mouth(Shape* s);
    virtual void wink(int i);          // 眨眼数 i

    // ...

private:
    vector<Shape*> eyes;               // 通常包含两只眼睛
    Shape* mouth;
};
```

成员函数 push_back() 把它的实参添加给 vector（此处是 eyes），每次将向量的长度加 1。

接下来通过调用 Smiley 的基类（Circle）的 draw() 和 Smiley 的成员（eyes）的 draw() 来定义 Smiley::draw()：

```
void Smiley::draw()
{
    Circle::draw();
    for (auto p : eyes)
        p->draw();
    mouth->draw();
}
```

请注意，Smiley 把它的 eyes 放在标准库 vector 中，然后在析构函数里把它们释放掉了。Shape 的析构函数是个虚函数，Smiley 的析构函数覆盖了它。对于抽象类来说，因为其派生类的对象通常是通过抽象基类的接口操纵的，所以基类中必须有一个虚析构函数。当我们使用一个基类指针释放派生类对象时，虚函数调用机制能够确保我们调用正确的析构函数，然后该析构函数再隐式地调用其基类的析构函数和成员的析构函数。

在上面这个简化的示例中，在表示人脸的圆圈中恰当地放置眼睛和嘴的任务交给程序员去完成。

当我们通过派生的方式定义新类时，可以向其中添加数据成员或者新的操作，或者都添加。这种机制一方面提供了巨大的灵活性，同时也可能带来混淆，从而造成糟糕的设计。第21章会详细介绍相关内容。总的来说，类层次提供了两种便利：

- 接口继承（Interface inheritance）：派生类对象可以用在任何需要基类对象的地方。也就是说，基类看起来像是派生类的接口一样。Container 和 Shape 就是很好的例子，这样的类通常是抽象类。
- 实现继承（Implementation inheritance）：基类负责提供可以简化派生类实现的函数或数据。Smiley 使用 Circle 的构造函数和 Circle::draw() 就是例子，这样的基类通常含有数据成员和构造函数。

具体类，尤其是表现形式不复杂的类，其行为非常类似于内置类型：我们将其定义为局部变量，通过它们的名字访问它们，随意拷贝它们，等等。类层次中的类则有所区别：我们倾向于通过 new 在自由存储中为其分配空间，然后通过指针或引用访问它们。例如，我们设计这样一个函数，它首先从输入流中读入描述形状的数据，然后构造对应的 Shape 对象：

```
enum class Kind { circle, triangle, smiley };

Shape* read_shape(istream& is)    // 从输入流 is 中读取形状描述信息
{
    // ... 从 is 中读取形状描述信息，找到形状的种类 k ...

    switch (k) {
    case Kind::circle:
        // 读入 circle 数据 {Point,int} 到 p 和 r
        return new Circle{p,r};
    case Kind::triangle:
        // 读入 triangle 数据 {Point,Point,Point} 到 p1, p2 和 p3
        return new Triangle{p1,p2,p3};
    case Kind::smiley:
        // 读入 smiley 数据 {Point,int,Shape,Shape,Shape} 到 p, r, e1 ,e2 和 m
        Smiley* ps = new Smiley{p,r};
        ps->add_eye(e1);
        ps->add_eye(e2);
        ps->set_mouth(m);
        return ps;
    }
}
```

程序使用该函数的方式如下所示：

```
void user()
{
    std::vector<Shape*> v;
    while (cin)
        v.push_back(read_shape(cin));
    draw_all(v);                    // 对每个元素调用 draw()
    rotate_all(v,45);               // 对每个元素调用 rotate(45)
    for (auto p : v) delete p;      // 注意最后要删除掉元素
}
```

上面这个例子显然非常简单，尤其是并没有做任何错误处理。不过我们还是能从中看出

user() 并不知道它操纵的具体是哪种形状。user() 的代码只需编译一次就可以使用随后添加到程序中的新 Shape。我们注意到在 user() 外没有任何指向这些形状的指针，因此 user() 应该负责释放掉它们。这项工作由 delete 运算符完成并且完全依赖于 Shape 的虚析构函数。因为该析构函数是虚函数，所以，delete 会调用最终派生类的析构函数。这一点非常关键：因为派生类可能有很多种需要释放的资源（如文件句柄、锁、输出流等）。此例中，Smiley 需要释放掉它的 eyes 和 mouth 对象。

有经验的程序员可能已经发现，上面的程序有两处漏洞：

- 使用者有可能未能 delete 掉 read_shape() 返回的指针。
- Shape 指针容器的拥有者可能没有 delete 指针所指的对象。

从这层意义上来看，函数返回一个指向自由存储上的对象的指针是非常危险的。

一种解决方案是不要返回一个“裸指针”，而是返回一个标准库 unique_ptr（见 5.2.1 节），并且把 unique_ptr 存放在容器中：

```
unique_ptr<Shape> read_shape(istream& is) // 从输入流 is 读取形状描述信息
{
    // ... 从 is 中读取形状描述信息，找到形状的种类 k...

    switch (k) {
    case Kind::circle:
        // 读入 circle 数据 {Point,int} 到 p 和 r
        return unique_ptr<Shape>{new Circle{p,r}};        // 5.2.1 节
    // ...
}

void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));
    draw_all(v);                // 对每个元素调用 draw()
    rotate_all(v,45);               // 对每个元素调用 rotate(45)
} // 所有形状被隐式销毁
```

这样对象就由 unique_ptr 拥有了。当不再需要对象时，换句话说，当对象的 unique_ptr 离开了作用域时，unique_ptr 将释放掉所指的对象。

要令 unique_ptr 版本的 user() 能够正确运行，必须首先构建接受 vector<unique_ptr<Shape>> 的 draw_all() 和 rotate_all()。写太多这样的 _all() 函数过于繁琐和乏味，因此 3.4.3 节将提供另一种可选的方案。

3.3　拷贝和移动

默认情况下，我们可以拷贝对象，不论用户自定义类型的对象还是内置类型的对象都是如此。拷贝的默认含义是逐成员的复制，即依次复制每个成员。例如，使用 3.2.1.1 节的 complex：

```
void test(complex z1)
{
    complex z2 {z1};    // 拷贝初始化
    complex z3;
    z3 = z2;            // 拷贝赋值
    // ...
}
```

因为赋值和初始化操作都复制了 complex 的全部两个成员，所以在上述操作之后 z1、z2、z3 的值变得完全一样。

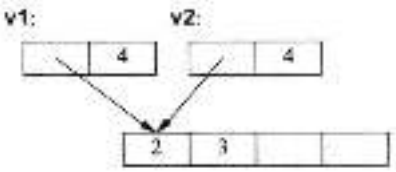
当我们设计一个类时，必须仔细考虑对象是否会被拷贝以及如何拷贝的问题。对于简单的具体类型来说，逐成员的复制通常符合拷贝操作的本来语义。然而对于某些像 Vector 一样的复杂具体类型，逐成员的复制常常是不正确的；抽象类型更是如此。

3.3.1 拷贝容器

当一个类作为资源句柄（resource handle）时，换句话说，当这个类负责通过指针访问一个对象时，采用默认的逐成员复制方式通常意味着错误。逐成员复制将违反资源句柄的不变式（见 2.4.3.2 节）。例如，下面所示的默认拷贝将产生 Vector 的一份拷贝，而这个拷贝所指向的元素与原来的元素是同一个：

```
void bad_copy(Vector v1)
{
    Vector v2 = v1;      // 把 v1 的表现形式复制给 v2
    v1[0] = 2;           // v2[0] 现在也是 2 了！
    v2[1] = 3;           // v1[1] 现在也是 3 了！
}
```

假设 v1 包含四个元素，则结果如下图所示：



幸运的是，Vector 的析构函数可以发现默认逐成员复制的语义错误从而引发编译器针对上述示例的报警（见 17.6 节）。这提醒我们应该为其定义更好的拷贝语义。

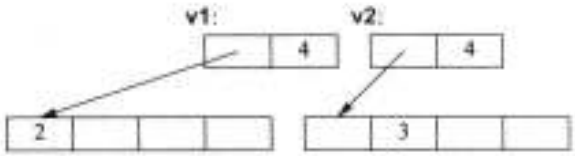
类对象的拷贝操作可以通过两个成员来定义：拷贝构造函数（copy constructor）与拷贝赋值运算符（copy assignment）：

```
class Vector {
private:
    double* elem; // elem 指向含有 sz 个 double 的数组
    int sz;
public:
    Vector(int s);                          // 构造函数：建立不变式，请求资源
    ~Vector() { delete[] elem; }            // 析构函数：释放资源

    Vector(const Vector& a);                // 拷贝构造函数
    Vector& operator=(const Vector& a);     // 拷贝赋值运算符

    double& operator[](int i);
    const double& operator[](int i) const;

    int size() const;
};
```

对于 Vector 来说，拷贝构造函数的正确定义应该首先为指定数量的元素分配空间，然后把元素复制到空间中。这样在复制完成后，每个 Vector 就拥有自己的元素副本了：

```
Vector::Vector(const Vector& a)      // 复制构造函数
    :elem{new double[sz]},           // 为元素分配空间
    sz{a.sz}
{
    for (int i=0; i!=sz; ++i)        // 复制元素
        elem[i] = a.elem[i];
}
```

在新的示例中 v2=v1 可以表示成：



当然，在拷贝构造函数之外我们还需要一个拷贝赋值运算符：

```
Vector& Vector::operator=(const Vector& a)      // 拷贝赋值运算符
{
    double* p = new double[a.sz];
    for (int i=0; i!=a.sz; ++i)
        p[i] = a.elem[i];
    delete[] elem;          // 删除旧元素
    elem = p;
    sz = a.sz;
    return *this;
}
```

其中，名字 this 预定义在成员函数中，它指向调用该成员函数的那个对象。

类 X 的拷贝构造函数和拷贝赋值运算符接受的实参类型通常是 const X&。

3.3.2 移动容器

我们能通过定义拷贝构造函数和拷贝赋值运算符来控制拷贝过程，但是对于大容量的容器来说拷贝过程有可能耗费巨大。以下面的代码为例：

```
Vector operator+(const Vector& a, const Vector& b)
{
    if (a.size()!=b.size())
        throw Vector_size_mismatch{};

    Vector res(a.size());
    for (int i=0; i!=a.size(); ++i)
        res[i]=a[i]+b[i];
    return res;
}
```

要想从 + 运算符返回结果，需要把局部变量 res 的内容拷贝到调用者可以访问的地方。我们可能这样使用 +：

```
void f(const Vector& x, const Vector& y, const Vector& z)
{
    Vector r;
    // ...
    r = x+y+z;
    // ...
}
```

这时就需要拷贝 Vector 对象至少两次（每个 + 一次）。如果 Vector 容量比较大的话，比方说含有 10 000 个 double，那么显然上述过程会让人头疼不已。最不合理的地方是 operator+() 中的 res 在拷贝后就不再使用了。事实上我们并不真的想要一个副本；我们只想把计算结果从函数中取出来；相对于拷贝（copy）一个 Vector 对象，我们更希望移动（move）它。幸运的是，C++ 为我们的想法提供了支持：

```
class Vector {
    // ...

    Vector(const Vector& a);              // 拷贝构造函数
    Vector& operator=(const Vector& a);   // 拷贝赋值运算符

    Vector(Vector&& a);                   // 移动构造函数
    Vector& operator=(Vector&& a);        // 移动赋值运算符
};
```

基于上述定义，编译器将选择移动构造函数（move constructor）来执行从函数中移出返回值的任务。这意味着 r=x+y+z 不需要再拷贝 Vector，只是移动它就足够了。

定义 Vector 移动构造函数的过程非常简单：

```
Vector::Vector(Vector&& a)
    :elem{a.elem},      // 从 a 中“夺取元素”
     sz{a.sz}
{
    a.elem = nullptr;          // 现在 a 已经没有元素了
    a.sz = 0;
}
```

符号 && 的意思是“右值引用”，我们可以给该引用绑定一个右值（见 6.4.1 节）。“右值”的含义与“左值”正好相反，左值的大致含义是“能出现在赋值运算符左侧的内容”，因此右值大致上就是我们无法为其赋值的值，比如函数调用返回的一个整数。进一步，右值引用的含义就是引用了一个别人无法赋值的内容。Vector 的 operator+() 运算符的局部变量 res 就是一个示例。

移动构造函数不接受 const 实参；毕竟移动构造函数最终要删除掉它实参中的值。移动赋值运算符（move assignment）的定义与之类似。

当右值引用被用作初始化器或者赋值操作的右侧运算对象时，程序将使用移动操作。

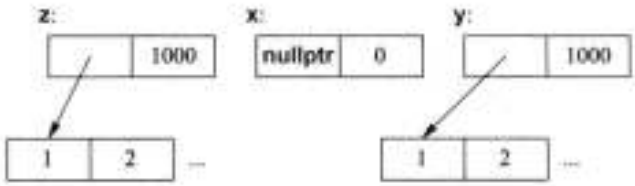
移动之后，源对象所进入的状态应该能允许运行析构函数。通常，我们也应该允许为一个移动操作后的源对象赋值（见 17.5 节和 17.6.2 节）。

程序员可以知道一个值在什么地方不再被使用，但是编译器做不到这一点，因此程序员最好在程序中写得明确一些：

```
Vector f()
{
    Vector x(1000);
    Vector y(1000);
    Vector z(1000);
    // ...
    z = x;                  // 执行拷贝操作
    y = std::move(x);       // 执行移动操作
    // ...
    return z;               // 执行移动操作
};
```

其中，标准库函数 move() 负责返回实参的右值引用。

在 return 语句执行之前的状态是：



当 z 被销毁时，事实上它也被 return 语句移走了，因此和 x 一样它也变为空（没有任何元素）。

3.3.3 资源管理

通过定义构造函数、拷贝操作、移动操作和析构函数，程序员就能对受控资源（比如容器中的元素）的全生命周期进行管理。而且移动构造函数还允许对象从一个作用域简单便捷地移动到另一个作用域。采取这种方式，我们不能或不希望拷贝到作用域之外的对象就能简单高效地移动出去了。不妨以表示并发活动的标准库 thread（见 5.3.1 节）和含有百万个 double 的 Vector 为例，前者“不能”执行拷贝操作，而后者我们“不希望”拷贝它。

```
std::vector<thread> my_threads;

Vector init(int n)
{
    thread t {heartbeat};                  // 同时运行 heartbeat（在它自己的线程上）
    my_threads.push_back(move(t));         // 把 t 移动到 my_threads
    // ... 其他初始化部分 ...
    Vector vec(n);
    for (int i=0; i<vec.size(); ++i) vec[i] = 777;
    return vec;                                    // 把 vec 移动到 init() 之外
}

auto v = init(); // 启动 heartbeat，初始化 v
```

在很多情况下，用 Vector 和 thread 这样的资源句柄比用指针效果要好。事实上，以 unique_ptr 为代表的“智能指针”本身就是资源句柄（见 5.2.1 节）。

我们使用标准库 vector 存放 thread 的原因是，在 3.4.1 节之前我们还接触不到用一种元素类型参数化 Vector 的方法。

就像替换掉程序中的 new 和 delete 一样，我们也可以把指针转化为资源句柄。在这两种情况下，都将得到更简单也更易维护的代码，而且没什么额外的开销。特别是我们能实现强资源安全（strong resource safety），换句话说，对于一般概念上的资源，这种方法都可以消除资源泄漏。比如存放内存的 vector、存放系统线程的 thread 和存放文件句柄的 fstream。

3.3.4 抑制操作

对于层次中的类来说，使用默认的拷贝或移动操作常常意味着风险：因为只给出一个基类的指针，我们无法了解派生类有什么样的成员（见 3.2.2 节），当然也就不知道该如何操作它们。因此，最好的做法是删除掉默认的拷贝和移动操作，也就是说，我们应该尽量避免使用这两个操作的默认定义：

```
class Shape {
public:
    Shape(const Shape&) =delete;                    // 没有拷贝操作
    Shape& operator=(const Shape&) =delete;

    Shape(Shape&&) =delete;                         // 没有移动操作
    Shape& operator=(Shape&&) =delete;

    ~Shape();
    // ...
};
```

现在编译器将捕获拷贝 Shape 的意图。如果你确实希望拷贝类层次中的某个对象，可以编写一些克隆函数（见 22.2.4 节）。

在这个特殊的例子中，即使忘了 delete 拷贝和移动操作也没什么问题。如果使用者在类中显式地声明了析构函数，则移动操作将不会隐式地生成。而且在本例中拷贝操作的生成也被禁止了（见 44.2.3 节）。这就是为什么即使编译器可以隐式地提供析构函数，也最好显式地自己定义一个析构函数的原因（见 17.2.3 节）。

在有些情况下我们不希望拷贝类的对象，类层次中的基类就是很好的例子。通常，我们无法通过拷贝成员的方式来拷贝资源句柄（见 5.2 节和 17.2.2 节）。

这种 =delete 的机制是通用的，也就是说，我们可以用它抑制任何操作（见 17.6.4 节）。

3.4 模板

显然，需要使用向量的人不一定总是使用 double 向量。向量是个通用的概念，不应局限于浮点数。因此，向量的元素类型应该独立表示。一个模板（template）就是一个类或一个函数，但需要我们用一组类型或值对其进行参数化。我们使用模板表示那些通用的概念，然后通过指定实参（比如指定元素的类型为 double）生成特定的类型或函数。

3.4.1 参数化类型

对于我们之前使用的 double 向量，只要将其改为 template 并且用一个形参替换掉特定类型 double，就能将其泛化成任意类型的向量了。例如：

```
template<typename T>
class Vector {
private:
    T* elem;  // elem 指向含有 sz 个 T 类型元素的数组
    int sz;
public:
    Vector(int s);                  // 构造函数：建立不变式，获取资源
    ~Vector() { delete[] elem; }    // 析构函数：释放资源

    // ... 拷贝和移动操作 ...

    T& operator[](int i);
    const T& operator[](int i) const;
    int size() const { return sz; }
};
```

前缀 template<typename T> 指明 T 是该声明的形参，它是数学上“对所有 T”或“对所有类型 T”的 C++ 表达。

成员函数的定义方式与之类似：

```
template<typename T>
Vector<T>::Vector(int s)
{
    if (s<0) throw Negative_size{};
    elem = new T[s];
    sz = s;
}

template<typename T>
const T& Vector<T>::operator[](int i) const
{
    if (i<0 || size()<=i)
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

基于上述定义，我们定义 Vector 的方式变为：

```
Vector<char> vc(200);          // 含有 200 个字符的向量
Vector<string> vs(17);         // 含有 17 个字符串的向量
Vector<list<int>> vli(45);     // 含有 45 个整数列表的向量
```

其中，最后一行 Vector<list<int>> 中的 >> 表示嵌套模板实参的结束，并不是输入运算符被放错了地方，不用非得像 C++98 那样在两个 > 之间加个空格。

我们使用 Vector 的方式是：

```
void write(const Vector<string>& vs)          // 字符串的向量
{
    for (int i = 0; i!=vs.size(); ++i)
        cout << vs[i] << '\n';
}
```

为了让我们的 Vector 支持范围 for 循环，需要为之定义适当的 begin() 和 end() 函数：

```
template<typename T>
T* begin(Vector<T>& x)
{
    return &x[0];          // 指针指向第一个元素
}

template<typename T>
T* end(Vector<T>& x)
{
    return x.begin()+x.size(); // 指针指向末尾元素的下一位置
}
```

在此基础上，我们就能编写如下的代码了：

```
void f2(const Vector<string>& vs)  // 字符串组成的 Vector
{
    for (auto& s : vs)
        cout << s << '\n';
}
```

类似地，我们也能将列表、向量、映射（也就是关联数组）等定义成模板（见 4.4 节，23.2 节和第 31 章）。

模板是一种编译时的机制，因此与“手工编写的代码”相比，并不会产生任何额外的运

行时开销（见23.2.2节）。

3.4.2 函数模板

模板的用途当然远不只是用元素类型参数化容器，我们用模板能参数化标准库中的很多类型和算法（见4.4.5节和4.5.5节）。例如，下面这段程序可以计算任意容器中元素的和：

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{
    for (auto x : c)
        v+=x;
    return v;
}
```

模板参数Value和函数参数v使得调用者可以指定累加器（用于求和的变量）的类型和初始值：

```
void user(Vector<int>& vi, std::list<double>& ld, std::vector<complex<double>>& vc)
{
    int x = sum(vi,0);                          // 求整数向量的和（累加整数）
    double d = sum(vi,0.0);                     // 求整数向量的和（累加浮点数）
    double dd = sum(ld,0.0);                    // 求浮点数列表的和
    auto z = sum(vc,complex<double>{});         // 求 complex<double> 向量的和
                                                // 初始值是 {0.0,0.0}
}
```

将一些int值累加到double变量中的做法让我们可以得体地处理超出int表示范围的数值。请注意sum<T,V>的模板实参类型是如何根据函数实参推断出来的。幸运的是，我们无须显式地指定这些类型。

这里的sum()可以看做是标准库accumulate()（见40.6节）的简化版本。

3.4.3 函数对象

模板的一个特殊用途是函数对象（function object，有时也称为函子functor），我们可以像调用函数一样使用函数对象。例如：

```
template<typename T>
class Less_than {
    const T val;                                    // 待比较的值
public:
    Less_than(const T& v) :val(v) { }
    bool operator()(const T& x) const { return x<val; } // 调用运算符
};
```

其中，名为operator()的函数实现了“函数调用”“调用”或“应用”运算符()。

我们能为某些实参类型定义Less_than类型的命名变量：

```
Less_than<int> lti {42};                // lti(i) 将使用 < 比较 i 和 42（i<42）
Less_than<string> lts {"Backus"};       // lts(s) 将使用 < 比较 s 和 "Backus"（s<"Backus"）
```

接下来，我们就能像调用函数一样调用该对象了：

```
void fct(int n, const string & s)
{
    bool b1 = lti(n);       // 如果 n<42 则为真
    bool b2 = lts(s);       // 如果 s<"Backus" 则为真
```

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

这样的函数对象经常作为算法的实参出现。例如，我们可以像下面这样统计令断言返回 true 的值的个数：

```
template<typename C, typename P>
int count(const C& c, P pred)
{
    int cnt = 0;
    for (const auto& x : c)
        if (pred(x))
            ++cnt;
    return cnt;
}
```

一个谓词（predicate）的返回值或者是 true，或者是 false。例如：

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
    cout << "number of values less than " << x
        << ": " << count(vec,Less_than<int>{x})
        << '\n';
    cout << "number of values less than " << s
        << ": " << count(lst,Less_than<string>{s})
        << '\n';
}
```

其中，Less_than<int>{x} 构造的对象将与名为 x 的 int 比较，而 Less_than<string>{s} 构造的对象将与名为 s 的 string 比较。函数对象的精妙之处在于它们随身携带着准备与之比较的值。我们无须为每个值（或者每种类型）单独编写函数，更不必把值保存在让人厌倦的全局变量中。同时，像 Less_than 这样的简单函数对象很容易内联，因此调用 Less_than 比间接函数调用更有效率。正是因为函数对象具有可携带数据和高效这两个特性，我们经常用其作为算法的实参。

用于指明通用算法关键操作含义的函数对象（如 Less_than 之于 count()）被称为策略对象（policy object）。

我们必须把 Less_than 的定义和使用分离开来。这么做看起来有点儿麻烦，因此，C++ 提供了一个隐式生成函数对象的表示法：

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
    cout << "number of values less than " << x
        << ": " << count(vec,[&](int a){ return a<x; })
        << '\n';
    cout << "number of values less than " << s
        << ": " << count(lst,[&](const string& a){ return a<s; })
        << '\n';
}
```

这里的 [&](int a){ return a<x; } 被称为 lambda 表达式（lambda expression，见 11.4 节），它生成一个函数对象，就像 Less_than<int>{x} 一样。[&] 是一个捕获列表（capture list），它指明所用的局部名字（如 x）将通过引用访问。如果我们希望只“捕获”x，则可以写成 [&x]；如果希望给生成的函数对象传递一个 x 的拷贝，则写成 [=x]。什么也不捕获是 [ ]，捕获所有通过引用访问的局部名字是 [&]，捕获所有以值访问的局部名字是 [=]。

使用 lambda 虽然简单便捷，但也有可能稍显晦涩难懂。对于复杂的操作（不是简单的一条表达式），我们更愿意给该操作起个名字，以便更加清晰地表述它的目的并且在程序中随处使用它。

在 3.2.4 节中，我们不得不编写很多像 draw_all() 和 rotate_all() 这样的函数来执行针对指针 vector 或 unique_ptr vector 中元素的操作。函数对象（尤其是 lambda）能在一定程度上解决这一问题，其核心思想是把容器的遍历和对每个元素的具体操作分离开来。

首先，我们需要定义一个函数，它负责把某个操作应用于指针容器的元素所指的每个对象：

```
template<class C, class Oper>
void for_all(C& c, Oper op)          // 假定 C 是一个指针容器
{
    for (auto& x : c)
        op(*x);                      // 为给 op() 每个元素所指对象的引用
}
```

接下来，我们改写 3.2.4 节中的 user()，而无须编写一大堆 _all() 函数：

```
void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));
    for_all(v,[](Shape& s){ s.draw(); });        // draw_all()
    for_all(v,[](Shape& s){ s.rotate(45); });    // rotate_all(45)
}
```

我给 lambda 传入 Shape 的引用，这样 lambda 就无须考虑容器中对象的存储方式了。特别是，即使把 v 改成一个 vector<Shape*>，那些 for_all() 调用也仍然能够正常工作。

3.4.4 可变参数模板

定义模板时可以令其接受任意数量任意类型的实参，这样的模板称为可变参数模板（variadic template）。例如：

```
template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
    g(head);   // 对 head 做某些操作
    f(tail...);   // 再次处理 tail
}

void f() { }      // 不执行任何操作
```

实现可变参数模板的关键是：当你传给它多个参数时，谨记把第一个参数和其他参数区分对待。此处，我们首先处理第一个参数（head），然后使用剩余参数（tail）递归地调用 f()。省略号…表示列表的“剩余部分”。最终，tail 将变为空，我们需要另外一个独立的函数来处理它。

调用 f() 的形式如下所示：

```
int main()
{
    cout << "first: ";
    f(1,2.2,"hello");
```

```
    cout << "\nsecond: "
    f(0.2,'c',"yuck!",0,1,2);
    cout << "\n";
}
```

上面的程序首先调用 f(1,2.2,"hello")，然后调用 f(2.2,"hello")，接着调用 f("hello")，最终会调用 f()。g(head) 又会做什么呢？显然，在一个真实的程序中，它将完成我们希望对每个实参执行的操作。例如，我们想要输出实参（这里是 head）：

```
template<typename T>
void g(T x)
{
    cout << x << " ";
}
```

则其输出的内容是：

```
first: 1 2.2 hello
second: 0.2 c yuck! 0 1 2
```

从效果上看，f() 类似于 printf() 的简单变形，仅用 3 行代码及相应的声明就实现了打印任意列表和值的功能。

可变参数模板有时也简称为可变参数（variadic），它的优势是可以接受我们希望传递给它的任意实参，而缺点则是接口的类型检查会比较复杂。相关细节读者可以参阅 28.6 节；34.2.4.2 节（N 元组）和第 29 章（N 维矩阵）则介绍了一些示例。

3.4.5　别名

在很多情况下，我们应该为类型或模板引入一个同义词（见 6.5 节）。例如，标准库头文件 <cstddef> 包含别名 size_t 的定义：

```
using size_t = unsigned int;
```

其中 size_t 的实际类型依赖于具体实现，在另外一个实现中 size_t 可能变成 unsigned long。使用 size_t，程序员就能写出易于移植的代码。

参数化的类型经常为与其模板实参关联的类型提供别名，例如：

```
template<typename T>
class Vector {
public:
    using value_type = T;
    // ...
};
```

事实上，每个标准库容器都提供了 value_type 作为其值类型的名字（见 31.3.1 节），这样我们编写的代码就能在任何一个服从这种规范的容器上工作了。例如：

```
template<typename C>
using Element_type = typename C::value_type;

template<typename Container>
void algo(Container& c)
{
    Vector<Element_type<Container>> vec; // 保存结果
    // ... 使用 vec ...
}
```

通过绑定某些或全部模板实参，我们就能使用别名机制定义新的模板。例如：

```
template<typename Key, typename Value>
class Map {
    // ...
};

template<typename Value>
using String_map = Map<string,Value>;

String_map<int> m; // m 是一个 Map<string,int>
```

相关内容请参阅 23.6 节。

3.5 建议

[1] 直接用代码表达你的想法；3.2 节。
[2] 在代码中直接定义类来表示应用中的概念；3.2 节。
[3] 用具体类表示那些简单的概念或性能关键的组件；3.2.1 节。
[4] 避免“裸的” new 和 delete 操作；3.2.1.2 节。
[5] 用资源句柄和 RAII 管理资源；3.2.1.2 节。
[6] 当接口和实现需要完全分离时使用抽象类作为接口；3.2.2 节。
[7] 用类层次表示具有固有的层次关系的概念；3.2.4 节。
[8] 在设计类层次时，注意区分实现继承和接口继承；3.2.4 节。
[9] 控制好对象的构造、拷贝、移动和析构操作；3.3 节。
[10] 以值的方式返回容器（依赖于移动操作以提高效率）；3.3.2 节。
[11] 注意强资源安全，也就是说，不要泄漏任何你认为是资源的东西；3.3.3 节。
[12] 使用容器保存同类型值的集合，将其定义为资源管理模板；3.4.1 节。
[13] 使用函数模板表示通用的算法；3.4.2 节。
[14] 使用包括 lambda 表达式在内的函数对象表示策略和动作；3.4.3 节。
[15] 使用类型别名和模板别名为相似类型或可能在实现中变化的类型提供统一的符号表示法；3.4.5 节。

第 4 章
The C++ Programming Language, Fourth Edition

C++ 概览：容器与算法

当无知只是瞬间，
又何必浪费时间学习呢？

——霍布斯（漫画人物）

- 标准库
  标准库概述；标准库头文件与名字空间
- 字符串
- I/O 流
  输出；输入；用户自定义类型的 I/O
- 容器
  vector；list；map；unordered_map；容器概述
- 算法
  使用迭代器；迭代器类型；流迭代器；谓词；算法概述；容器算法
- 建议

4.1 标准库

从来没有任何一个重要的程序是用“裸语言”写成的。人们通常先开发出一系列库，随后把它们作为进一步编程工作的基础。如果只用裸语言编写程序，大多数情况下程序将是非常乏味的。而有了好的库，几乎所有编程工作都会变得更简单。

承接第 2 ~ 3 章，本章和下一章将对重要的标准库设施给出一个快速导览。在阅读本章之前，你最好已经有一些编程经验。如果没有，建议读者先找一本入门教材学习一下，比如《Programming: Principles and Practice Using C++》[Stroustrup, 2009]。即便你编写过程序，你使用的语言或编写的应用也可能在风格或形式上与本书展示的 C++ 风格相距甚远。因此，如果你发现接下来的“快速导览”不那么容易理解，不妨直接跳到第 6 章，从那儿开始我们将对知识介绍得更加系统和有条理。尤其是从第 30 章开始，我们将为读者系统介绍标准库的知识。

我将简要介绍常用的标准库类型，如 string、ostream、vector、map（本章）、unique_ptr、thread、regex 和 complex（第 5 章），并介绍它们最常见的用法。这么做的好处是便于我在接下来的章节中更好地举例。与在第 2 ~ 3 章中一样，我们强烈建议你不要因为对某些细节理解不够充分而心烦或气馁。本章的目的是让读者对那些最有用的标准库设施的基本知识有个初步认识，而非对它们进行详细介绍。

在 ISO C++ 标准中，标准库规范几乎占了 2/3。在学习 C++ 的过程中，你应努力探寻标准库的相关知识，尽量使用已有的标准库设施而不是自己再做一份。因为标准库的设计已经凝结了太多精妙的思想，还有更多思想体现在其实现中，并且未来还会有大量的精力投入

到标准库的维护和扩展中。

本书介绍的标准库设施，在任何一个完整的C++实现中都是必备的部分。当然，除了标准库组件外，大多数C++实现还提供“图形用户接口”系统（GUI）、Web接口、数据库接口等。类似地，大多数应用程序开发环境还会提供“基础库”，来完善企业级或工业级的“标准”开发和运行环境。但在本书中，我不会介绍这类系统和库。本书的目标还是为读者提供一个自包含的C++语言介绍，它基于C++标准定义，同时保证程序范例都是可移植的（特别指出的除外）。当然，我们鼓励程序员去探索那些常见的非C++标准设施。

4.1.1 标准库概述

标准库提供的设施可以分为以下几类：

- 运行时语言支持（例如，对资源分配和运行时类型信息的支持）；见30.3节。
- C标准库（进行了微小的修改，以便尽量减少与类型系统的冲突）；见第43章。
- 字符串和I/O流（包括对国际字符集和本地化的支持）；见第36、38和39章。I/O流是一个可扩展的输入输出框架，用户可向其中添加自己的流、缓冲策略和字符集。
- 一个包含容器（如vector和map）和算法（如find()、sort()和merge()）的框架；见4.4节、4.5节、第31～33章。人们习惯上称这个框架为标准模板库（STL）[Stepanov, 1994]。用户可向其中添加自己定义的容器和算法。
- 对数值计算的支持（例如标准数学函数、复数、支持算术运算的向量以及随机数发生器），见3.2.1.1节和第40章。
- 对正则表达式匹配的支持，见5.5节和第37章。
- 对并发程序设计的支持，包括thread和lock机制，见5.3节和第41章。在此基础上，用户就能够以库的形式添加新的并发模型。
- 一系列工具，它们用于支持模板元编程（如类型特性，见5.4.2节、28.2.4节和35.4节）、STL-风格的泛型程序设计（如pair，见5.4.3节和34.2.4.1节）和通用程序设计（如clock，见5.4.1节和35.2节）。
- 用于资源管理的“智能指针”（如unique_ptr和shared_ptr，见5.2.1节和34.3节）和垃圾收集器接口（见34.5节）。
- 特殊用途容器，例如array（见34.2.1节）、bitset（见34.2.2节）和tuple（见34.2.4.2节）。

判断是否应该将一个类纳入标准库的主要依据包括：

- 它几乎对所有C++程序员（包括初学者和专家）都有用；
- 它能以一种通用的形式提供给程序员，并且与简单版本相比没有严重的额外开销；
- 易学易用（相对于编程任务的内在复杂性而言）。

本质上来说，C++标准库提供了最常用的基本数据结构以及运行在之上的基础算法。

4.1.2 标准库头文件与名字空间

每个标准库设施都是通过若干标准库头文件提供的，例如：

```
#include<string>
#include<list>
```

包含这两个头文件后，程序中就可以使用string和list了。

标准库定义在一个名为 std 的名字空间中（见 2.4.2 节和 14.3.1 节）。为了使用标准库设施，可以加上 std:: 前缀：

```
std::string s {"Four legs Good; two legs Baaad!"};
std::list<std::string> slogans {"War is peace", "Freedom is Slavery", "Ignorance is Strength"};
```

为简洁起见，我在书中的例子中很少显式使用 std:: 前缀，我也不会显式地给出 #include 语句以包含必要的头文件。为了正确编译并运行本书中的程序片段，读者需要自行补上恰当的 #include 语句（4.4.5 节、4.5.5 节和 30.2 节分别列出了一些常用的头文件），以让标准库名字变得可用。例如：

```
#include<string>          // 令标准库 string 可用
using namespace std;      // 令 std 中所有名字可用而不必使用 std:: 前缀

string s {"C++ is a general-purpose programming language"};    // OK: 此处的 string 意即 std::string
```

一般来说，把一个名字空间（std）中所有的名字都暴露在全局名字空间中不是什么好的编程习惯。但是仅就本书而言，我基本上只用到了标准库，因此读者很容易知道它从何而来，又能为我们提供哪些功能。基于上述原因，我既不会在每次用到标准库名字时都加上 std::，也不会在每个示例中都 #include 所需的头文件。我假设我的读者已经对这些心知肚明了。

下面是一些标准库头文件，它们所含的声明都位于名字空间 std 中：

部分标准库头文件			
<algorithm>	copy()，find()，sort()	32.2 节	iso.25
<array>	array	34.2.1 节	iso.23.3.2
<chrono>	duration，time_point	35.2 节	iso.20.11.2
<cmath>	sqrt()，pow()	40.3 节	iso.26.8
<complex>	complex，sqrt()，pow()	40.4 节	iso.26.8
<fstream>	fstream，ifstream，ofstream	38.2.1 节	iso.27.9.1
<future>	future，promise	5.3.5 节	iso.30.6
<iostream>	istream，ostream，cin，cout	38.1 节	iso.27.4
<map>	map，multimap	31.4.3 节	iso.23.4.4
<memory>	unique_ptr，shared_ptr，allocator	5.2.1 节	iso.20.6
<random>	default_random_engine，normal_distribution	40.7 节	iso.26.5
<regex>	regex，smatch	第 37 章	iso.28.8
<string>	string，basic_string	第 36 章	iso.21.3
<set>	set，multiset	31.4.3 节	iso.23.4.6
<sstream>	istrstream，ostrstream	38.2.2 节	iso.27.8
<thread>	thread	5.3.1 节	iso.30.3
<unordered_map>	unordered_map，unordered_multimap	31.4.3.2 节	iso.23.5.4
<utility>	move()，swap()，pair	35.5 节	iso.20.1
<vector>	vector	31.4 节	iso.23.3.6

此列表远未囊括所有标准库头文件，30.2 节将会介绍更多有关信息。

4.2 字符串

标准库提供的 string 类型弥补了字符串字面常量的不足。string 类型提供了很多有用的字符串操作，最典型的比如连接操作。下面是一个例子：

```
string compose(const string& name, const string& domain)
{
    return name + '@' + domain;
}

auto addr = compose("dmr","bell-labs.com");
```

在本例中，addr 被初始化为字符序列 dmr@bell-labs.com。函数 compose 中的字符串“加法”表示连接操作。你可以将一个 string 对象、一个字符串字面常量、一个 C-风格字符串或是一个字符连接到 string 上。标准库 string 定义了一个移动构造函数，因此，即使是以传值方式而不是传引用方式返回一个很长的 string 也会很高效（见 3.3.2 节）。

在很多应用中，连接操作最常见的用法是在一个 string 的末尾添加一些内容。这可以直接通过 += 操作来实现。例如：

```
void m2(string& s1, string& s2)
{
    s1 = s1 + '\n';  // 追加换行
    s2 += '\n';      // 追加换行
}
```

这两种向 string 末尾添加内容的方法在语义上是等价的，但我更倾向于使用后者，因为它更明确、更简洁地表达了要做什么，而且可能也更高效。

string 对象是可变的。除了 = 和 += 外，string 还支持下标操作（使用 [ ]）和子串操作。第 36 章将介绍标准库 string 的详细信息。它为其他有用的特性提供了操纵子串的能力。例如：

```
string name = "Niels Stroustrup";

void m3()
{
    string s = name.substr(6,10);      // s = "Stroustrup"
    name.replace(0,5,"nicholas");      // name 变为 "nicholas Stroustrup"
    name[0] = toupper(name[0]);        // name 变为 "Nicholas Stroustrup"
}
```

substr() 操作返回一个 string，保存实参指定的子串的拷贝。第一个参数是一个下标，指向 string 中的一个位置，第二个参数指出所需子串的长度。由于下标从 0 开始，因此上面的程序中 s 得到的值是 Stroustrup。

replace () 操作替换子串内容。在本例中，要替换的是从 0 开始长度为 5 的子串，即 Niels，它被替换为 nicholas。最后，我将首字母变为大写。因此，name 的最终值为 Nicholas Stroustrup。注意，替换的内容和被替换的子串不必一样长。

自然地，string 之间可以相互比较，也可以与字符串字面常量比较，例如：

```
string incantation;

void respond(const string& answer)
{
    if (answer == incantation) {
```

```
            // 执行一些操作
        }
        else if (answer == "yes") {
            // ...
        }
        // ...
}
```

第 36 章将介绍标准库 string 的详细信息，实现 string 用到的关键技术在 String 例子中有所体现（见 19.3 节）。

4.3 I/O 流

标准库 iostream 提供了格式化字符的输入输出功能，其中的输入操作类型敏感且能被扩展以处理用户自定义的类型。本节将简要介绍 iostream 的用法，第 38 章会完整介绍 iostream 标准库设施。

其他形式的用户交互，如图形化 I/O，是通过相应的库来进行处理的。这些库并不是 ISO 标准库的一部分，因此本书并未涉及。

4.3.1 输出

I/O 流库为所有内置类型都定义了输出操作。而且，为用户自定义类型定义输出操作也很简单（见 4.3.3 节）。运算符 << （“放到”）是输出运算符，它作用于 ostream 类型的对象；cout 是标准输出流，cerr 是报告错误的标准流。默认情况下，写到 cout 的值被转换为一个字符序列。例如，为了输出十进制数 10，可编写函数如下：

```
void f()
{
    cout << 10;
}
```

此代码将字符 1 放到标准输出流中，接着又放入字符 0。

另一种等价的写法是：

```
void g()
{
    int i {10};
    cout << i;
}
```

不同类型值的输出可以用一种很直观的方式组合在一起：

```
void h(int i)
{
    cout << "the value of i is ";
    cout << i;
    cout << '\n';
}
```

调用 h(10) 会输出：

```
the value of i is 10
```

如果像上面这样输出多个相关的项，你肯定很快就厌倦了不断重复输出流的名字。幸运的是，输出表达式的返回结果是输出流的引用，因此可用来继续进行输出，例如：

```
void h2(int i)
{
    cout << "the value of i is " << i << '\n';
}
```

h2() 的输出结果与 h() 完全一样。

字符常量就是被单引号包围的一个字符。注意，输出一个字符的结果是其字符形式，而不是其数值。例如：

```
void k()
{
    int b = 'b';      // 注意：char 隐式转换为 int
    char c = 'c';
    cout << 'a' << b << c;
}
```

字符 'b' 对应的整数值是 98（我所使用的 C++ 实现中的 ASCII 编码值），因此这个函数的输出结果为 a98c。

4.3.2 输入

标准库提供了 istream 来实现输入。与 ostream 类似，istream 处理内置类型的字符串表示形式，并能被很容易地扩展以支持用户自定义的类型。

运算符 >>（"从…获取"）实现输入功能；cin 是标准输入流。>> 右侧的运算对象决定了输入什么类型的值，以及输入的值保存在哪里。例如：

```
void f()
{
    int i;
    cin >> i;         // 读取一个 int 保存在 i 中

    double d;
    cin >> d;         // 读取一个双精度浮点数保存在 d 中
}
```

这段代码从标准输入读取一个数，如 1234，保存在整型变量 i 中。然后读取一个浮点数，如 12.34e5，保存在双精度浮点型变量 d 中。

我们常常要读取一个字符序列，最简单的方法是读入一个 string。例如：

```
void hello()
{
    cout << "Please enter your name\n";
    string str;
    cin >> str;
    cout << "Hello, " << str << "!\n";
}
```

如果你键入 Eric，程序将回应：

```
Hello, Eric!
```

默认情况下，空格等空白符（见 7.3.2 节）会终止输入。因此，如果你键入 Eric Bloodaxe 冒充不幸的约克王，程序的回应仍会是：

```
Hello, Eric!
```

你可以用函数 getline() 来读取一整行（包括结束时的换行符），例如：

```
void hello_line()
{
    cout << "Please enter your name\n";
    string str;
    getline(cin,str);
    cout << "Hello, " << str << "!\n";
}
```

运行这个程序，再输入 Eric Bloodaxe 就会得到想要的输出：

```
Hello, Eric Bloodaxe!
```

行尾的换行符被丢弃掉了，因此接下来再 cin 的话会从下一行开始。

标准库字符串有一个很好的性质——可以自动扩充空间来容纳你存入的内容。这样，你就无须预先计算所需的最大空间。因此，即使你输入几兆字节的分号，上述程序也能正确执行，回应给你一页页的分号。

4.3.3　用户自定义类型的 I/O

除了支持内置类型和标准库 string 的 I/O 之外，iostream 库还允许程序员为自己的类型定义 I/O 操作。例如，考虑一个简单的类型 Entry，我们用它来表示电话簿中的一条记录：

```
struct Entry {
    string name;
    int number;
};
```

我们可以定义一个简单的输出运算符，以类似于初始化代码的形式 {"*name*", *number*} 来打印一个 Entry：

```
ostream& operator<<(ostream& os, const Entry& e)
{
    return os << "{\"" << e.name << "\", " << e.number << "}";
}
```

一个用户自定义的输出运算符接受它的输出流（通过引用）为第一个参数，输出完毕后，返回此流的引用。详细信息请见 38.4.2 节。

对应的输入运算符要复杂得多，因为它必须检查格式是否正确并处理可能发生的错误：

```
istream& operator>>(istream& is, Entry& e)
    // 读取 { "name", number } 对。注意，正确格式包含 { " " 和 }。
{
    char c, c2;
    if (is>>c && c=='{' && is>>c2 && c2=='"') { // 以一个 { " 开始
        string name;                    // string 的默认值是空字符串 ""
        while (is.get(c) && c!='"')     // " 之前的任何内容都是名字的一部分
            name+=c;

        if (is>>c && c==',') {
            int number = 0;
            if (is>>number>>c && c=='}') { // 读取数和一个 }
                e = {name,number};      // 把读入的值赋予 Entry 对象
                return is;
            }
        }
    }
    is.setf(ios_base::failbit);         // 将流状态置为 fail
```

```
    return is;
}
```

输入运算符返回它所操作的 istream 对象的引用。该引用可用来检测操作是否成功。例如，当用作一个条件时，is>>c 表示“我们从 is 读取数据存入 c 的操作成功了吗？”

is>>c 默认跳过空白符，而 is.get(c) 则不会。因此，上面的 Entry 的输入运算符忽略（跳过）名字字符串外围的空白符，但不会忽略其内部的空白符。例如：

```
{ "John Marwood Cleese" , 123456      }
{"Michael Edward Palin",987654}
```

我们可以用下面的代码从输入流读取这样的值对，然后存入 Entry 对象中：

```
for (Entry ee; cin>>ee; ) // 从 cin 读取数据存入 ee
    cout << ee << '\n';   // 将 ee 的值写入 cout
```

则输出为：

```
{"John Marwood Cleese", 123456}
{"Michael Edward Palin", 987654}
```

关于如何为用户自定义类型编写输入运算符的更多技术细节和手段请参考 38.4.1 节。5.5 节和第 37 章将讲解在字符流中识别模式的更系统的方法（正则表达式匹配）。

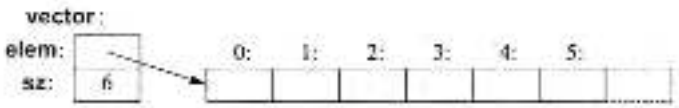
4.4 容器

大多数计算任务都会涉及创建值的集合然后对这些集合进行操作。一个简单的例子是，先读取若干字符到 string 中，然后打印这个 string。如果一个类的主要目的是保存一些对象，那么我们通常称之为容器。为给定的任务提供合适的容器以及之上有用的基本操作，是构建任何程序的重要步骤。

我们通过一个保存名字和电话号码的简单示例程序来介绍标准库容器。这是一个对于任何背景的人都显得“简单而直观”程序。我们用 4.3.3 节中的 Entry 类来保存一个电话簿的表项。在本例中，我们特意忽略掉很多现实世界中的复杂因素，例如，很多电话号码其实不能简单地用一个 32 位 int 来表示。

4.4.1 vector

最有用的标准库容器当属 vector。一个 vector 就是一个给定类型元素的序列，元素在内存中是连续存储的：



3.2.2 节和 3.4 节的 Vector 示例给出了标准库 vector 的实现思想，13.6 节和 31.4 节则会讨论更多细节内容。

我们可以用一组值来初始化 vector，当然，值的类型必须与 vector 元素类型吻合：

```
vector<Entry> phone_book = {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

我们可以通过下标运算符访问元素：

```
void print_book(const vector<Entry>& book)
{
    for (int i = 0; i!=book.size(); ++i)
        cout << book[i] << '\n';
}
```

照例，下标从 0 开始，因此 book[0] 保存的表项是 David Hume。vector 的成员函数 size() 返回元素的数目。

vector 的所有元素构成了一个范围，因此我们可以对其使用范围 for 循环（见 2.2.5 节）：

```
void print_book(const vector<Entry>& book)
{
    for (const auto& x : book)      // 关于 "auto"，请查阅 2.2.2 节
        cout << x << '\n';
}
```

定义一个 vector 时，为它设定一个初始大小（初始的元素数目）：

```
vector<int> v1 = {1, 2, 3, 4};          // size 为 4
vector<string> v2;                      // size 为 0
vector<Shape*> v3(23);                  // size 为 23；元素初值是 nullptr
vector<double> v4(32,9.9);              // size 为 32；元素初值是 9.9
```

我们可以在一对圆括号中显式地给出 vector 的大小，如 (23)。默认情况下，元素被初始化为其类型的默认值（例如，指针初始化为 nullptr，整数初始化为 0）。如果不想要默认值，你可以通过构造函数的第 2 个实参来指定一个值（例如，将 v4 的 32 个元素初始化为 9.9）。

vector 的初始大小随着程序的执行可以被改变。vector 最常用的一个操作就是 push_back()，它向 vector 末尾追加一个新元素，从而将 vector 的规模增大 1。例如：

```
void input()
{
    for (Entry e; cin>>e;)
        phone_book.push_back(e);
}
```

这段程序从标准输入读取 Entry，保存到 phone_book 中，直至遇到输入结束标识（如文件尾）或是输入操作遇到一个格式错误。标准库 vector 经过了精心设计，即使不断调用 push_back() 来扩充 vector 也会很高效。

在赋值和初始化时，vector 可以被拷贝。例如：

```
vector<Entry> book2 = phone_book;
```

如 3.3 节所述，拷贝和移动 vector 是通过构造函数和赋值运算符实现的。vector 的赋值过程包括拷贝其中的元素。因此，在 book2 初始化完成后，它和 phone_book 各自保存每个 Entry 的一份副本。当一个 vector 包含很多元素时，这样一个看起来无害的赋值或初始化操作可能非常耗时。因此，当拷贝并非必要时，我们应该优先使用引用或指针（见 7.2 节和 7.7 节）或是移动操作（见 3.3.2 节和 17.5.2 节）。

4.4.1.1　元素

和所有标准库容器一样，vector 也是元素类型为 T 的容器，即 vector<T>。几乎任意一种数据类型都可以作为容器的元素类型，它们包括：内置数值类型（如 char、int 和 double）、用户自定义类型（如 string、Entry、list<int> 和 Matrix<double, 2>）以及指针类

型（如 const char *、Shape *、和 double *）。当插入一个新元素时，它的值被拷贝到容器中。例如，当你把一个整型值 7 存入容器中时，结果元素确实就是一个值为 7 的整型对象，而不是指向某个整型对象 7 的引用或指针。这样的策略促成了精巧、紧凑、访问快速的容器。对于在意内存大小和运行时性能的人来说，这是非常关键的。

4.4.1.2 范围检查

标准库 vector 并不进行范围检查（见 31.2.2 节）。例如：

```
void silly(vector<Entry>& book)
{
    int i = book[ph.size()].number;      // book.size() 越界
    // ...
}
```

这个初始化操作有可能将某个随机值存入 i 中，而不是产生一个错误。这并不是我们所期望的，而这种越界错误又非常常见。因此，我通常使用 vector 的一个简单改进版本，它增加了范围检查：

```
template<typename T>
class Vec : public std::vector<T> {
public:
    using vector<T>::vector; // 使用 vector 的构造函数（包含字是 Vec）；见 20.3.5.1 节

    T& operator[](int i)                    // 范围检查
        { return vector<T>::at(i); }

    const T& operator[](int i) const        // 常量版本；见 3.2.1.1 节
        { return vector<T>::at(i); }
};
```

Vec 继承了 vector 除下标运算符之外的所有内容，它重定义了下标运算符来进行范围检查。vector 的 at() 函数同样负责下标操作，但它会在参数越界时抛出一个类型为 out_of_range 的异常（见 2.4.3.1 节和 31.2.2 节）。

对于一个 Vec 对象来说，越界访问会抛出一个用户可捕获的异常，例如：

```
void checked(Vec<Entry>& book)
{
    try {
        book[book.size()] = {"Joe",999999};      // 会抛出一个异常
        // ...
    }
    catch (out_of_range) {
        cout << "range error\n";
    }
}
```

这段程序会抛出一个异常，然后将其捕获（见 2.4.3.1 节和第 13 章）。如果用户不捕获异常，则程序会以一种定义良好的方式退出，而不是继续执行或者以一种未定义的方式终止。一种尽量弱化未捕获异常影响的方法是使用以 try- 块作为 main() 函数的函数体。例如：

```
int main()
try {
    // 你的代码
}
```

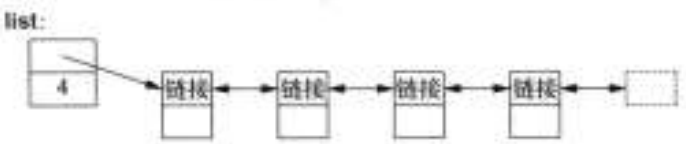
```
catch (out_of_range) {
    cerr << "range error\n";
}
catch (...) {
    cerr << "unknown exception thrown\n";
}
```

这段代码提供了默认的异常处理措施。当我们未能成功捕获某些异常时，就会进入默认异常处理程序，在标准错误流 cerr 上打印一条错误消息（见 38.1 节）。

某些 C++ 实现提供带范围检查功能的 vector（例如，作为一个编译选项提供），从而免去你定义 Vec（或等价的类）的麻烦。

4.4.2　list

标准库提供了一个名为 list 的双向链表：



如果我们希望在一个序列中添加和删除元素的同时无须移动其他元素，则应该使用 list。对电话簿应用而言，插入删除操作可能非常频繁，因此 list 适合保存电话簿。例如：

```
list<Entry> phone_book = {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

当使用链表时，我们通常并不想像使用向量那样使用它，即，不会用下标操作来访问链表元素，而是想进行“在链表中搜索具有给定值的元素”这类操作。为了完成这样的操作，我们可以利用“list 是序列”这样一个事实（如 4.5 节所述）：

```
int get_number(const string& s)
{
    for (const auto& x : phone_book)
        if (x.name==s)
            return x.number;
    return 0; // 用 0 表示“未找到所需值”
}
```

这段代码从链表头开始搜索 s，直至找到 s 或到达 phone_book 的末尾为止。

我们有时需要在 list 中定位一个元素。例如，我们可能想删除这个元素或是在这个元素之前插入一个新元素。为此，我们需要使用迭代器（iterator）：一个 list 迭代器指向 list 中的一个元素，它可以用来遍历 list（它也正是因此得名）。每个标准库容器都提供 begin() 和 end() 函数，分别返回一个指向首元素的迭代器和一个指向尾后位置的迭代器（见 4.5 节和 33.1.1 节）。我们可以改写 get_number() 函数，令其显式地使用迭代器遍历 list，当然这个版本显然稍微有些繁琐：

```
int get_number(const string& s)
{
    for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
```

```
        if (p->name==s)
            return p->number;
    return 0; // 用 0 表示"未找到所需值"
}
```

使用范围 for 循环的 get_number() 函数更简练、更不容易出错，但实际上，迭代器版本差不多就是编译器最终实现范围 for 的方式。给定一个迭代器 p，*p 表示它所指向的元素，++p 令 p 指向下一个元素。当 p 指向一个类且该类有一个成员 m 时，p->m 等价于 (*p).m。

向 list 中添加元素以及从 list 中删除元素都很简单：

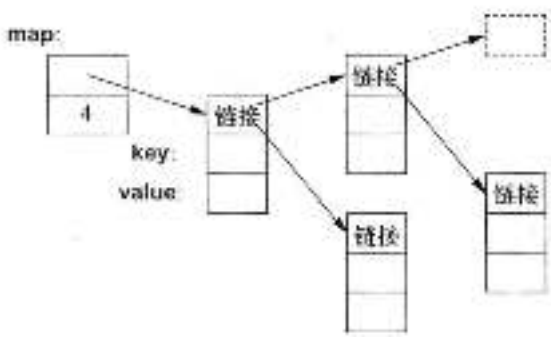
```
void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)
{
    phone_book.insert(p,ee);      // 将 ee 添加到 p 指向的元素之前
    phone_book.erase(q);          // 删除 q 指向的元素
}
```

31.3.7 节会介绍更多关于 inser t() 和 erase() 的内容。

上面这些 list 的例子都可以写成等价的使用 vector 的版本，而且令人惊讶（除非你了解机器的体系结构）的是，当数据量较小时，vector 版本的性能会优于 list 版本。当我们想要的只是一个元素序列时，我们可以在 vector 和 list 之间选择。除非你有充分的理由选择 list，否则就应该使用 vector。vector 无论是遍历（如 find() 和 count()）性能还是排序和搜索（如 sort() 和 binary_search()）性能都优于 list。

4.4.3 map

编写程序在一个（名字，数值）对的列表中查找给定名字，是一项很烦人的工作。而且，除非列表很短，否则顺序搜索是非常低效的。标准库提供了一个名为 map 的搜索树（红黑树）：



在某些情境下，map 也被称为关联数组或字典。map 通常用平衡二叉树实现。

标准库 map（见 31.4.3 节）是值对的容器，经过特殊优化来提高搜索性能。我们可以像初始化 vector 和 list 那样来初始化 map（见 4.4.1 节和 4.4.2 节）：

```
map<string,int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

map 也支持下标操作，给定的下标值应该是 map 的第一个类型（称为关键字，key），得到的结果是与关键字关联的值（是 map 的第二个类型，称为值或映射类型）。例如：

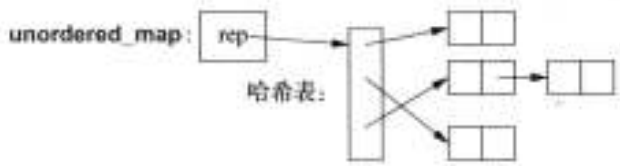
```
int get_number(const string& s)
{
    return phone_book[s];
}
```

换句话说，对 map 进行下标操作本质上是进行一次搜索，我们称为 get_number()。如果未找到 key，则向 map 插入一个新元素，它具有给定的 key 且关联的值是 value 类型的默认值。在本例中，整数类型的默认值是 0，恰好是我用来表示无效电话号码的值。

如果我们希望避免将一个无效号码添加到电话簿中，就应该使用 find() 和 insert() 来代替 [ ]（见 31.4.3.1 节）。

4.4.4 unordered_map

搜索 map 的时间代价是 O(log(n))，其中 n 是 map 元素的数目。通常情况下，这样的性能非常好。例如，考虑一个包含 100 万个元素的 map，我们只需执行约 20 次比较和间接寻址操作即可找到元素。不过，在很多情况下我们还可以做得更好，那就是使用哈希查找，而不是使用基于某种序函数的比较操作（如 <）。标准库哈希容器被称为“无序”容器，因为它们不需要序函数：



例如，我们可以使用 <unordered_map> 中定义的 unordered_map 来编写电话簿程序：

```
unordered_map<string,int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

与 map 类似，我们也可以对 unordered_map 使用下标操作：

```
int get_number(const string& s)
{
    return phone_book[s];
}
```

标准库 unordered_map 为 string 提供了默认的哈希函数。如有必要，你也可以定义自己的哈希函数（见 31.4.3.4 节）。

4.4.5 容器概述

标准库提供了一些既通用又好用的容器类型，程序员可以根据应用的实际需求选择最适合的容器：

标准库容器概述	
vector<T>	可变大小向量（31.4 节）
list<T>	双向链表（31.4.2 节）
forward_list<T>	单向链表（31.4.2 节）

（续）

标准库容器概述	
deque<T>	双端队列（31.2 节）
set<T>	集合（31.4.3 节）
multiset<T>	允许重复值的集合（31.4.3 节）
map<K,V>	关联数组（31.4.3 节）
multimap<K,V>	允许重复关键字的 map（31.4.3 节）
unordered_map<K,V>	采用哈希搜索的 map（31.4.3.2 节）
unordered_multimap<K,V>	采用哈希搜索的 multimap（31.4.3.2 节）
unordered_set<T>	采用哈希搜索的 set（31.4.3.2 节）
unordered_multiset<T>	采用哈希搜索的 multiset（31.4.3.2 节）

无序容器针对关键字（通常是一个字符串）搜索进行了优化，这是通过使用哈希表来实现的。

31.4 节将详细介绍与容器有关的知识。前面提到的容器分别定义在 <vector>、<list> 和 <map> 等头文件中（见 4.1.2 节和 30.2 节），这些容器都属于名字空间 std。此外，标准库还提供了容器适配器 queue<T>（见 31.5.2 节）、stack<T>（见 31.5.1 节）、deque<T>（见 31.4 节）和 priority_queue<T>（见 31.5.3 节）。标准库还提供了一些更特殊的类似容器的类型，如定长数组 array<T,N>（见 34.2.1 节）和 bitset<N>（见 34.2.2 节）。

从符号表示的角度来看，标准库的各种容器和它们的基本操作比较类似。而且，不同容器的操作含义也大致相同。基本操作可用于每一种适用的容器，且可高效实现。例如：

- begin() 和 end() 分别返回指向首元素和尾后位置的迭代器。
- push_back() 可用来（高效地）向 vector、list 及其他容器末尾添加元素。
- size() 返回元素数目。

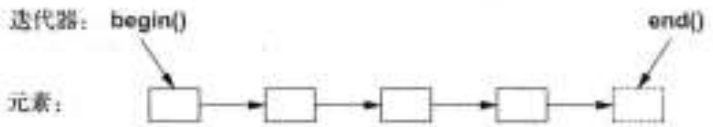
形式和语义上的一致性使得程序员可以设计出与标准库容器在使用方式上非常相似的新的容器类型，包含范围检查功能的向量 Vector（见 2.3.2 节和 2.4.3.1 节）就是一个很好的例子。容器接口的一致性还便于我们设计与容器类型无关的算法。但是，凡事皆有两面性，某种技术的优点和缺点常常并存。例如，下标操作和遍历 vector 的操作很高效也很简单。但另一方面，当我们在 vector 中插入或删除元素时，不得不每次都移动元素，效率不佳；list 则恰好具有相反的特性。请注意，当序列较短且元素尺寸较小时，vector 通常比 list 高效（insert() 和 erase() 操作也是如此）。推荐将标准库 vector 作为存储元素序列的默认类型；只有当你的理由足够充分时，再考虑选择其他容器。

4.5 算法

对于编程任务来说，仅仅定义链表和向量等数据结构是远远不够的。为了使用某种数据结构，我们还需要一些基本访问操作，比如添加和删除元素的操作（就像为 list 和 vector 提供的那样）。而且，我们一般不会只把对象保存在容器中了事，而是需要对它们进行排序、打印、抽取子集、删除元素、搜索等更复杂的操作。因此，标准库在提供最常用的容器类型之外，还为这些容器提供了最常用的算法。例如，下面的代码首先排序一个 vector 对象，然后把所有不重复的 vector 元素拷贝到一个 list 中：

```
bool operator<(const Entry& x, const Entry& y)    // 小于运算符
{
    return x.name<y.name;          // Entry 对象的序由它们的名字确定
}

void f(vector<Entry>& vec, list<Entry>& lst)
{
    sort(vec.begin(),vec.end());                          // 用 < 确定元素的序
    unique_copy(vec.begin(),vec.end(),lst.begin());       // 不拷贝相邻的重复元素
}
```

第 32 章将详细介绍标准库算法的知识。标准库算法都描述为元素序列上的操作。这里的一个序列（sequence）由一对迭代器表示，它们分别指向首元素和尾后位置：



在本例中，迭代器对 vec.begin() 和 vec.end() 定义了一个序列（恰好就是 vector 的全部元素），sort() 对此序列进行排序操作。要想在 list 中写入数据（不妨理解为程序的输出任务），你只需指明要写的第一个元素。如果写入的元素不止一个，则写入的内容会覆盖起始位置之后的那些元素。因此，为了避免写入错误，lst 的空间应该至少能够容纳 vec 的全部不重复元素。

如果我们想把不重复元素存入一个新容器，而不是覆盖一个容器中的旧元素，则可以这样编写程序：

```
list<Entry> f(vector<Entry>& vec)
{
    list<Entry> res;
    sort(vec.begin(),vec.end());
    unique_copy(vec.begin(),vec.end(),back_inserter(res)); // 追加到 res
    return res;
}
```

我们调用 back_inserter() 把元素追加到容器末尾，在追加的过程中扩展容器空间以容纳新元素（见 33.2.2 节）。这样，标准库容器加上 back_inserter() 就提供了一个很好的方案，使我们不必再使用容易出错的 C- 风格的显式内存管理（使用 realloc()，见 31.5.1 节）。标准库 list 具有移动构造函数（见 3.3.2 节和 17.5.2 节），这使得以传值方式返回 res 也很高效（即使 list 中有数千个元素）。

如果你觉得 sort(vec.begin(),vec.end()) 这种使用迭代器对的代码太冗长，你也可以自己定义容器版本的算法，代码就能简化为 sort(vec)（见 4.5.6 节）。

4.5.1 使用迭代器

当你第一次接触容器时，首先需要了解的就是一些指向有用元素的迭代器，比如 begin() 和 end()。此外，很多算法的返回值类型也是迭代器。例如，标准库算法 find 在一个序列中查找一个值，返回的结果是指向找到的元素的迭代器：

```
bool has_c(const string& s, char c)    // s 包含字符 c 吗?
{
    auto p = find(s.begin(),s.end(),c);
```

```
    if (p!=s.end())
        return true;
    else
        return false;
}
```

与很多标准库搜索算法类似，find 返回 end() 来表示“未找到”。has_c() 还有一个更短的等价版本：

```
bool has_c(const string& s, char c)    // s 包含字符 c 吗?
{
    return find(s.begin(),s.end(),c)!=s.end();
}
```

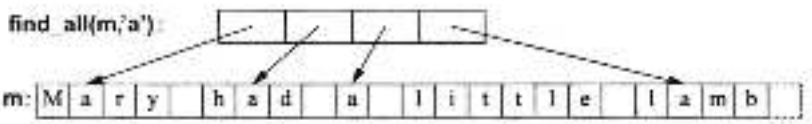
一个更有意思的练习是在字符串中查找一个字符出现的所有位置。我们可以返回一个 string 迭代器的 vector，其中保存出现位置的集合。因为 vector 提供了移动语义（见 3.3.1 节），所以返回 vector 是很高效的。如果我们希望能够修改找到的内容，则应该传递一个非 const 字符串：

```
vector<string::iterator> find_all(string& s, char c)        // 在 s 中查找 c 出现的所有位置
{
    vector<string::iterator> res;
    for (auto p = s.begin(); p!=s.end(); ++p)
        if (*p==c)
            res.push_back(p);
    return res;
}
```

这段代码用一个常规的循环遍历字符串，每个循环步使用 ++ 运算符将迭代器 p 向前移动一个元素，并使用解引用运算符 * 查看元素值。我们可以这样来测试 find_all()：

```
void test()
{
    string m {"Mary had a little lamb"};
    for (auto p : find_all(m,'a'))
        if (*p!='a')
            cerr << "a bug!\n";
}
```

find_all() 调用可以图示如下：



迭代器和标准库算法在所有标准库容器上的工作方式都是相同的（前提是它们适用于这种容器）。因此，我们可以泛化 find_all()：

```
template<typename C, typename V>
vector<typename C::iterator> find_all(C& c, V v)        // 在容器 c 中查找值 v 出现的所有位置
{
    vector<typename C::iterator> res;
    for (auto p = c.begin(); p!=c.end(); ++p)
        if (*p==v)
            res.push_back(p);
    return res;
}
```

这里的 typename 必不可少，它通知编译器 C 的 iterator 是一种类型，而非某种类型的值，比如说整数 7。我们可以通过引入一个类型别名（见 3.4.5 节）Iterator 来隐藏这些实现细节：

```
template<typename T>
using Iterator<T> = typename T::iterator;        // T 的迭代器

template<typename C, typename V>
vector<Iterator<C>> find_all(C& c, V v)          // 在 c 中查找 v 出现的所有位置
{
    vector<Iterator<C>> res;
    for (auto p = c.begin(); p!=c.end(); ++p)
        if (*p==v)
            res.push_back(p);
    return res;
}
```

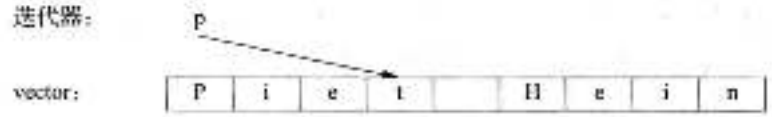
现在我们就可以编写下面的代码来完成一些搜索任务了：

```
void test()
{
    string m {"Mary had a little lamb"};
    for (auto p : find_all(m,'a'))               // p 是一个 string::iterator
        if (*p!='a')
            cerr << "string bug!\n";

    list<double> ld {1.1, 2.2, 3.3, 1.1};
    for (auto p : find_all(ld,1.1))
        if (*p!=1.1)
            cerr << "list bug!\n";

    vector<string> vs { "red", "blue", "green", "green", "orange", "green" };
    for (auto p : find_all(vs,"green"))
        if (*p!="green")
            cerr << "vector bug!\n";

    for (auto p : find_all(vs,"green"))
        *p = "vert";
}
```

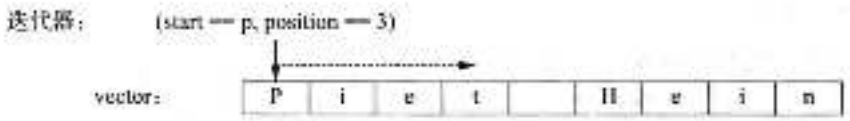
迭代器的一个重要作用是分离算法和容器。算法通过迭代器来处理数据，但它对存储元素的容器一无所知。反之亦然，容器对处理其元素的算法也是一无所知，它所做的全部事情就是按需求提供迭代器（如 begin() 和 end()）。这种数据存储和算法分离的模型催生出非常通用和灵活的软件。

4.5.2　迭代器类型

迭代器本质上是什么？当然，任何一种特定的迭代器都是某种类型的对象。不过，迭代器的类型非常多，毕竟每个迭代器都是与某个特定容器类型相关联的。它需要保存一些必要的信息，以便我们对容器执行某些特定的任务。因此，有多少种容器就有多少种迭代器，有多少种特殊要求就有多少种迭代器。例如，vector 的迭代器可能就是一个普通指针，因为就引用 vector 中的元素而言指针再恰当不过了：

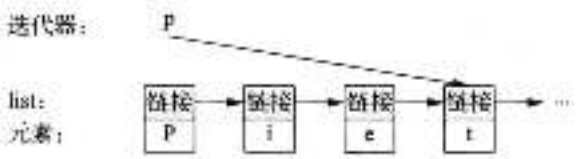


或者，vector的迭代器也可以实现为指向vector（存储空间起始地址）的指针再加上一个索引：



采用这种实现方式便于进行范围检查。

list迭代器必须是某种比普通指针更复杂的东西，这里的普通指针是指那些指向某个元素的指针。因为list的元素通常不知道它的下一个元素在哪里，所以list迭代器应该指向某个链接：



所有迭代器类型的语义及其操作的命名都是相似的。例如，对任何迭代器使用++运算符都会得到一个指向下一个元素的迭代器，而*运算符则得到迭代器所指的元素。实际上，任何符合这些简单规则的对象都能被看成是迭代器（见33.1.4节）。用户不需要知道某个特定迭代器的类型，迭代器“知道”它自己的迭代器类型是什么，而且都能通过规范的名字iterator和const_iterator来正确声明自己的类型。例如，list<Entry>::iterator是list<Entry>的迭代器类型，程序员很少需要操心“这些类型是如何定义的？”等细节问题。

4.5.3 流迭代器

迭代器是处理容器中元素序列的一个很有用的通用概念。但是，容器并非容纳元素序列的唯一场所。例如，一个输入流产生一个值的序列，我们还可以将一个值的序列写入一个输出流。因此，将迭代器的概念应用到输入输出是很有用的。

为了创建一个ostream_iterator，我们需要指出使用哪个流，以及输出的对象类型。例如：

```
ostream_iterator<string> oo {cout};      // 将字符串写入 cout
```

向*oo赋值的作用是把所赋的值输出到cout。例如：

```
int main()
{
    *oo = "Hello, ";      // 等价于 cout<<"Hello,"
    ++oo;
    *oo = "world!\n";     // 等价于 cout<<"world!\n"
}
```

我们得到了一种向标准输出写入消息的新方法，其中++oo的工作方式类似于用指针向数组中写入值。

类似地，istream_iterator 允许我们将一个输入流当作一个只读容器来使用。与之前一样，我们必须指明从哪个流读取数据以及数据的类型是什么：

```
istream_iterator<string> ii {cin};
```

我们需要用一对输入迭代器表示一个序列，因此我们必须提供一个表示输入结束的 istream_iterator。默认的 istream_iterator 就起到这个作用：

```
istream_iterator<string> eos {};
```

我们通常不直接使用 istream_iterator 和 ostream_iterator，而是将它们作为实参传递给算法。例如，我们可以写出一个简单的程序，它从文件中读取数据，排序读入的单词，去除重复单词，最后把结果写到另一个文件中：

```
int main()
{
    string from, to;
    cin >> from >> to;                      // 获取源文件和目标文件名

    ifstream is {from};                     // 对应文件 "from" 的输入流
    istream_iterator<string> ii {is};       // 输入流的迭代器
    istream_iterator<string> eos {};        // 输入哨兵

    ofstream os{to};                        // 对应文件 "to" 的输出流
    ostream_iterator<string> oo {os,"\n"};  // 输出流的迭代器

    vector<string> b {ii,eos};              // b 是一个 vector，用输入进行初始化
    sort(b.begin(),b.end());                // 排序缓冲区中的单词

    unique_copy(b.begin(),b.end(),oo);      // 将不重复的单词拷贝到输出，丢弃重复值

    return !is.eof() || !os;                // 返回错误状态（见 2.2.1 节和 38.3 节）
}
```

ifstream 是一个可以绑定到文件的 istream，而 ofstream 就是一个可以绑定到文件的 ostream。其中，ostream_iterator 的第 2 个参数指定输出的间隔符。

实际上，这个程序本不必这么长。该版本首先读取字符串并存入一个 vector，然后用 sort() 对它们排序，最终将不重复的单词写入输出。一个更简洁的方案是根本不保存重复单词。我们可以将 string 保存在一个 set 中，而 set 是不会保留重复值的，而且能维护值的顺序（见 31.4.3 节）。这样，我们就能把使用 vector 的两行代码改写成使用 set 的一行代码，而且也不必再使用 unique_copy()，用更简单的 copy() 就可以了：

```
set<string> b {ii,eos};             // 从输入流采集字符串
copy(b.begin(),b.end(),oo);         // 把缓冲区中的单词拷贝到输出
```

ii、eos 和 oo 都只使用了一次，因此我们可以继续减小程序的规模：

```
int main()
{
    string from, to;
    cin >> from >> to;              // 获取源文件和目标文件名

    ifstream is {from};             // 对应文件 "from" 的输入流
    ofstream os {to};               // 对应文件 "to" 的输出流

    set<string> b {istream_iterator<string>{is},istream_iterator<string>{}}; // 读取输入
```

```
    copy(b.begin(),b.end(),ostream_iterator<string>{os,"\n"});          // 标准库输出

    return !is.eof() || !os;                    // 返回错误状态（见 2.2.1 节和 38.3 节）
}
```

至于最终的简化版本是否真的提高了可读性，就完全依赖个人偏好和编程经验了。

4.5.4 谓词

在前面的例子中，算法都是对序列中每个元素简单地进行“内置”操作，但我们常常需要把操作也作为算法的参数。例如，find 算法（见 32.4 节）提供了一种查找给定值的便捷途径。而对于查找满足特定要求元素的问题，还有一种更通用的方法，称为谓词（predicate，见 3.4.2 节）。例如，我们可能需要在一个 map 中搜索第一个大于 42 的值。我们在访问 map 的元素时，实际访问的是（关键字，值）对的序列。因此，我们可以将任务转化为在 map<string,int> 中搜索一个特定的 pair<const string,int>，并要求它的 int 部分大于 42：

```
void f(map<string,int>& m)
{
    auto p = find_if(m.begin(),m.end(),Greater_than{42});
    // ...
}
```

此处，Greater_than 是一个函数对象（见 3.4.3 节），它保存着要比较的值（42）：

```
struct Greater_than {
    int val;
    Greater_than(int v) : val{v} { }
    bool operator()(const pair<string,int>& r) { return r.second>val; }
};
```

我们也可以使用 lambda 表达式（见 3.4.3 节）：

```
int cxx = count_if(m.begin(), m.end(), [](const pair<string,int>& r) { return r.second>42; });
```

4.5.5 算法概述

算法的一个更一般性的定义是“一个有限规则集合，给出了一个操作序列，用来求解一组特定问题［且］具有 5 个重要特性：有限性……确定性……输入……输出……有效性”［Knuth，1968，1.1 节］。在 C++ 标准库的语境中，算法就是一个对元素序列进行操作的函数模板。

标准库提供了很多算法，它们都定义在头文件 <algorithm> 中且属于名字空间 std。这些标准库算法的输入都是序列（见 4.5 节）。一个从 b 到 e 的半开序列表示为 [b:e)。下面是一些特别有用的算法的简介。

部分标准库算法	
p=find(b,e,x)	p 是 [b:e) 中第一个满足 *p==x 的迭代器
p=find_if(b,e,f)	p 是 [b:e) 中第一个满足 f(*p)==true 的迭代器
n=count(b,e,x)	n 是 [b:e) 中满足 *q==x 的元素 *q 的数目
n=count_if(b,e,f)	n 是 [b:e) 中满足 f(*q)==true 的元素 *q 的数目
replace(b,e,v,v2)	将 [b:e) 中满足 *q==v 的元素 *q 替换为 v2
replace_if(b,e,f,v2)	将 [b:e) 中满足 f(*q)==true 的元素 *q 替换为 v2

（续）

部分标准库算法	
p=copy(b,e,out)	将 [b:e) 拷贝到 [out:p)
p=copy_if(b,e,out,f)	将 [b:e) 中满足 f(*q)==true 的元素 *q 拷贝到 [out:p)
p=unique_copy(b,e,out,f)	将 [b:e) 拷贝到 [out:p)，不拷贝连续的重复元素
sort(b,e)	排序 [b:e) 中的元素，用 < 作为排序标准
sort(b,e,f)	排序 [b:e) 中的元素，用谓词 f 作为排序标准
(p1,p2)=equal_range(b,e,v)	[p1:p2) 是已排序序列 [b:e) 的子序列，其中元素的值都等于 v，本质上等价于二分搜索 v
p=merge(b,e,b2,e2,out)	将两个序列 [b:e) 和 [b2:e2) 合并，结果保存到 [out:p)

这些算法以及其他很多算法（见第 32 章）可以用于容器、string 或内置数组。

4.5.6　容器算法

序列是通过一对迭代器 [begin:end) 定义的，算法操作序列的方式具有通用性且非常灵活。但很多时候，算法所操作的序列就是容器本身的内容。例如：

```
sort(v.begin(),v.end());
```

既然这样的话，为什么我们不直接用 sort(v) 呢？支持这样的简写形式并不困难：

```
namespace Estd {
    using namespace std;

    template<class C>
    void sort(C& c)
    {
        sort(c.begin(),c.end());
    }

    template<class C, class Pred>
    void sort(C& c, Pred p)
    {
        sort(c.begin(),c.end(),p);
    }

    // ...
}
```

我把容器版本的 sort()（和其他容器版本的算法）放在它们自己的名字空间 Estd 中（“扩展的 std”），这样就可以避免与其他程序员对名字空间 std 的使用相互干扰了。

4.6　建议

[1] 没必要推倒重来，直接使用标准库是最好的选择；4.1 节。

[2] 除非万不得已，大多数时候先考虑使用标准库，再考虑别的库；4.1 节。

[3] 标准库绝非万能；4.1 节。

[4] 如果你用到了某些标准库设施，记得把头文件 #include 进来；4.1.2 节。

[5] 标准库设施位于名字空间 std 中；4.1.2 节。

[6] 标准库 string 优于 C 风格字符串（即 char*，见 2.2.5 节）；4.2 节，4.3.2 节。

[7] iostream 具有类型敏感、类型安全和可扩展等特点；4.3 节。
[8] 与 T[] 相比，vector<T>、map<K,T> 和 unordered_map<K,T> 更优；4.4 节。
[9] 一定要了解各种标准库容器的设计思想和优缺点；4.4 节。
[10] 优先选用 vector 作为你的容器类型；4.4.1 节。
[11] 数据结构应力求小巧；4.4.1 节。
[12] 如果你拿不准会不会越界，记得使用带边界检查的容器（比如 Vec）；4.4.1.2 节。
[13] 用 push_back() 或者 back_inserter() 给容器添加元素；4.4.1 节，4.5 节。
[14] 在 vector 上使用 push_back() 要比在数组上使用 realloc() 更好；4.5 节。
[15] 在 main() 函数中捕获常见的异常；4.4.1.2 节。
[16] 了解常用的标准库算法，用它们替代你自己手写的循环；4.5.5 节。
[17] 如果用迭代器实现某算法过于冗长，不妨改写成使用容器的版本；4.5.6 节。

第5章
The C++ Programming Language, Fourth Edition

C++概览：并发与实用功能

要想让别人听得明白，言辞必须简洁。

——西塞罗

- 引言
- 资源管理
  unique_ptr与shared_ptr
- 并发
  任务与thread；传递参数；返回结果；共享数据；任务通信
- 小工具组件
  时间；类型函数；pair和tuple
- 正则表达式
- 数学计算
  数学函数和算法；复数；随机数；向量算术；数值限制
- 建议

5.1 引言

从使用者的角度出发，理想的标准库应该为所有可能的需求都提供直接支持。对于某个特定领域来说，一个超大规模的商业库也许能够无限接近这种理想状态，但这绝非C++标准库的目标。显然，一个易于管理且具有普适性的库不可能同时满足每个人的每个要求。C++标准库的真正目标是为大多数人在大多数领域中的需求提供必要的组件。换句话说，标准库关注的是所有需求的交集而非并集。此外，标准库也尝试为一些特别重要的应用领域（如数学计算和文本操作）提供支持。

5.2 资源管理

所有程序都包含一项关键任务：管理资源。所谓资源是指程序中符合先获取后释放（显式或者隐式）规律的东西，比如内存、锁、套接字、线程句柄和文件句柄等。对于长时间连续运行的程序来说，如果不能及时地释放掉资源（即造成了泄漏），就有可能大大降低程序的运行效率甚至造成程序崩溃。即使在规模较小的程序中，资源泄漏也可能造成严重的后果：由于系统资源短缺，所以程序的运行时间会成倍地增长。

标准库组件不会出现资源泄漏的问题。在设计标准库组件时，设计者使用成对的构造函数/析构函数等基本语言特性来管理资源，确保资源依存于其所属的对象，而不会超过对象的生命周期。举一个例子，3.2.1.2节介绍的Vector就是使用构造函数/析构函数的机制管理元素的，所有标准库容器的实现方式也都与之类似。此外，这种管理资源的方式通常通过抛出和捕获异常来进行错误处理。例如标准库当中的锁：

```
mutex m; // 用于确保共享数据被正确访问
// ...
void f()
{
    unique_lock<mutex> lck {m}; // 获取资源
    // ... 操作共享数据 ...
}
```

lck 的构造函数首先获取它的 mutex，m（见 5.3.4 节），然后 thread 开始处理。最后析构函数负责释放掉资源。在上面的例子中，当控制线程离开 f() 时（通过 return 语句跳转到函数末尾，或者因为抛出异常而离开函数），unique_lock 的析构函数负责释放掉 mutex。

这是“资源获取即初始化”（RAII，见 3.2.1.2 节和 13.3 节）技术的一个典型应用。RAII 是 C++ 处理资源的基础，容器（比如 vector 和 map）、string 和 iostream 管理资源（比如文件句柄和缓冲区）的方式都十分相似。

5.2.1 unique_ptr 与 shared_ptr

之前的例子都是关于定义在作用域内的对象的，它们可以在作用域结束的时候释放掉资源。但是如果对象是在自由存储上分配的呢？在 <memory> 当中，标准库提供了两种“智能指针”来管理自由存储上的对象：

[1] unique_ptr 对应所有权唯一的情况（见 34.3.1 节）。

[2] shared_ptr 对应所有权共享的情况（见 34.3.2 节）。

这些“智能指针”最基本的作用是防止由于编程疏忽而造成的内存泄漏。例如：

```
void f(int i, int j)     // 对比 X* 和 unique_ptr<X>
{
    X* p = new X;                 // 分配一个新的 X
    unique_ptr<X> sp {new X};     // 分配一个新的 X，把它的指针赋给 unique_ptr
    // ...
    if (i<99) throw Z();          // 可能会抛出异常
    if (j<77) return;             // 可能会“过早地”返回
    p->do_something();            // 可能会抛出异常
    sp->do_something();           // 可能会抛出异常
    // ...
    delete p;                     // 销毁 *p
}
```

在这段代码中，如果 i<99 或者 j<77，我们会“忘记”释放掉指针 p。另一方面，unique_ptr 确保不论我们以哪种方式（通过抛出异常，或者通过执行 return 语句，或者跳转到了函数末尾）退出 f() 都会释放掉它的对象。其实换个角度思考一下，如果我们干脆不使用指针也不使用 new，那么上面的问题也就不复存在了：

```
void f(int i, int j)     // 变量是局部变量，而非指针
{
    X x;
    // ...
}
```

不幸的是，越来越多的程序员喜欢不加节制地滥用 new（以及指针和引用）。

如果你确实需要使用指针，那么与内置指针相比，unique_ptr 是更好的选择。后者是一种轻量级的机制，消耗的时空代价并不比前者大。通过使用 unique_ptr，我们还可以把自由存储上申请的对象传递给函数或者从函数中传出来：

```
unique_ptr<X> make_X(int i)
    // 创建一个 X，然后立即把它赋给 unique_ptr
{
    // ... 检查 i 以及其他操作 ...
    return unique_ptr<X>{new X(i)};
}
```

unique_ptr 是一个独立对象或数组的句柄，就像 vector 是对象序列的句柄一样。这二者都以 RAII 的机制控制其他对象的生命周期，并且都通过移动操作使得 return 语句简单高效。

shared_ptr 在很多方面都和 unique_ptr 非常相似，唯一的区别是 shared_ptr 的对象使用拷贝操作而非移动操作。某个对象的多个 shared_ptr 共享该对象的所有权，只有当最后一个 shared_ptr 被销毁时对象才被销毁。例如：

```
void f(shared_ptr<fstream>);
void g(shared_ptr<fstream>);

void user(const string& name, ios_base::openmode mode)
{
    shared_ptr<fstream> fp {new fstream(name,mode)};
    if (!*fp) throw No_file{};  // 检查文件是否被正确打开

    f(fp);
    g(fp);
    // ...
}
```

fp 的构造函数打开的文件将会被使用了 fp 的最后一个函数（显式地或者隐式地）关闭。其中 f() 或者 g() 有可能含有 fp 的一份拷贝，而这份拷贝直到 user() 执行完还在使用。因此，与使用析构函数管理内存对象的资源管理方式相比，shared_ptr 提供的垃圾回收机制需要慎重使用。这与时空代价无关，而是说 shared_ptr 使得对象的生命周期变得不那么容易掌控了。我们的建议是：除非你确实需要共享所有权，否则别轻易使用 shared_ptr。

通过使用 unique_ptr 和 shared_ptr，我们就能在很多程序中实现完全“没有裸 new”的目标（见 3.2.1.2 节）。不过，这些“智能指针”从概念上讲仍然是指针，因此我在管理资源时只把它们当成次优选择——把容器和其他可以在更高的概念层次上管理资源的类型作为第一选择效果更好。还有一点值得注意，shared_ptr 本身没有制定任何规则用以指明共享指针的哪个拥有者有权读写对象。因此尽管在一定程度上解决了资源管理的问题，但是数据竞争（见 41.2.4 节）和其他形式的数据混淆依然存在。

那么什么情况下我们才应该选择“智能指针”（比如 unique_ptr）而非带有特定操作的资源句柄（比如 vector 和 thread）呢？显然，答案应该是“当我们需要使用指针的语义时”。

- 当我们共享某个对象时，需要让多个指针或者引用指向被共享的对象，此时选择 shared_ptr 是显而易见的（除非所有人都知道资源有且只有一个拥有者）。
- 当我们指向一个多态对象时，很难确切地知道对象到底是什么类型（甚至连对象的大小都不知道），所以应该使用指针或者引用，此时 unique_ptr 成为必然的选择。
- 共享的多态对象通常会用到 shared_ptr。

当我们需要从函数返回对象的集合时，不必用指针，使用容器能让这个任务更加简单高效（见 3.3.2 节）。

5.3 并发

并发，也就是多个任务同时执行，被广泛用于提高吞吐率（用多个处理器共同完成单个运算）和提高响应速度（允许程序的一部分在等待响应时，另一部分继续执行）。所有现代程序设计语言都提供了对并发的支持。C++ 标准库并发设施的前身在 C++ 中已应用超过 20 年了，经过对可移植性和类型安全的改进，成为标准库的一部分，它几乎适用于所有现代硬件平台。标准库并发设施重点提供系统级并发机制，而不是直接提供复杂的高层并发模型。基于标准库并发设施可以构建出这类高层并发模型，并以库的形式提供。

标准库直接支持在单一地址空间内并发执行多个线程。为了实现这一目的，C++ 提供了一个适合的内存模型（见 41.2 节）和一套原子操作（见 41.3 节）。但是，大多数用户眼中的并发就是标准库设施以及建立在之上的其他并发库。因此，本节将简要介绍一些关键的标准库并发设施：thread、mutex、lock() 操作、packaged_task 和 future，给出一些示例。这些特性直接建立在操作系统并发机制之上，与系统原始机制相比，这些特性并不会带来额外的性能开销，当然也不保证性能有显著提升。

5.3.1 任务和 thread

我们称那些可以与其他计算并行执行的计算为任务（task）。线程（thread）是任务在程序中的系统级表示。若要启动一个与其他任务并发执行的任务，我们可以构造一个 std::thread（在 <thread> 中）并将任务作为它的实参。这里的任务是以函数或函数对象的形式出现的：

```
void f();               // 函数

struct F {              // 函数对象
    void operator()();  // F 调用运算符（见 3.4.3 节）
};

void user()
{
    thread t1 {f};      // f() 在独立的线程中执行
    thread t2 {F()};    // F()() 在独立的线程中执行

    t1.join();          // 等待 t1 完成
    t2.join();          // 等待 t2 完成
}
```

join() 保证我们在线程完成后才退出 user()，其中“join”的意思是“等待线程结束”。

一个程序的所有线程共享单一地址空间。在这一点上线程与进程不同，进程间通常不直接共享数据。由于共享单一地址空间，因此线程间可通过共享对象（见 5.3.4 节）相互通信。通常通过锁或其他防止数据竞争（对变量的不受控制的并发访问）的机制来控制线程间通信。

编写并发任务可能非常棘手。任务 f（函数）和 F（函数对象）可能会写成如下的形式：

```
void f() { cout << "Hello "; }

struct F {
    void operator()() { cout << "Parallel World!\n"; }
};
```

这是一个典型的严重错误：在本例中，f 和 F() 都使用了对象 cout，但没有采取任何形式

的同步措施。因此，输出结果将是不可预测的，而且程序每一次执行都可能得到不同结果，毕竟两个任务中的操作的执行顺序根本无法确定。程序可能会产生下面这样“奇怪的”输出：

```
PaHeralllllel o World!
```

在定义一个并发程序的任务时，我们的目标是保持任务的完全隔离，唯一的例外是任务间通信的部分，这种通信应该以一种简单而明显的方式进行。思考一个并发任务的最简单的方式是把它看作一个可以与调用者并发执行的函数。为此，我们只需传递实参、获取结果并保证两者不会同时使用共享数据（不存在数据竞争）即可。

5.3.2　传递参数

任务通常需要处理数据，我们可以将数据（或指向数据的指针或引用）作为参数传递给任务，例如：

```
void f(vector<double>& v);     // 处理 v 的函数

struct F {                     // 处理 v 的函数对象
    vector<double>& v;
    F(vector<double>& vv) :v{vv} { }
    void operator()();         // 调用运算符，见 3.4.3 节
};

int main()
{
    vector<double> some_vec {1,2,3,4,5,6,7,8,9};
    vector<double> vec2 {10,11,12,13,14};

    thread t1 {f,some_vec};    // f(some_vec) 在一个独立线程中执行
    thread t2 {F{vec2}};       // F(vec2)() 在一个独立线程中执行

    t1.join();
    t2.join();
}
```

显然，F{vec2}将一个指向参数（一个向量）的引用保存在 F 中。F 现在就可以使用向量了，并希望在它运行的时候其他任务不会访问 vec2——将 vec2 以传值方式传递就可以消除这个风险。

上面代码用 {f,some_vec} 初始化一个线程，它使用了 thread 的可变参数模板构造函数，接受一个任意的参数序列（见 28.6 节）。编译器检查第一个参数（函数或函数对象）是否可用后续的参数来调用，如果检查通过，就构造一个必要的函数对象并传递给线程。因此，F::operator()() 与 f() 执行相同的算法，两个任务的处理大致相同：它们都为 thread 构造了一个函数对象来执行任务。

5.3.3　返回结果

在 5.3.2 节的例子中，我通过一个非 const 引用向线程传递参数。只有当希望任务有权修改引用所引的数据时，我才会这么做（见 7.7 节）。这种返回结果的方法有点不正规，但并不少见。一种不那么晦涩的技术是将输入数据以 const 引用的方式传递，并将保存结果的内存地址作为第二个参数传递给线程。

```
void f(const vector<double>& v, double* res);// 从 v 获取输入，将结果放入 *res

class F {
public:
    F(const vector<double>& vv, double* p) :v{vv}, res{p} { }
    void operator()();              // 将结果放入 *res
private:
    const vector<double>& v;        // 输入源
    double* res;                    // 输出目标
};

int main()
{
    vector<double> some_vec;
    vector<double> vec2;
    // ...

    double res1;
    double res2;

    thread t1 {f,some_vec,&res1}; // f(some_vec,&res1) 在一个独立线程中执行
    thread t2 {F{vec2,&res2}};          // F{vec2,&res2}() 在一个独立线程中执行

    t1.join();
    t2.join();

    cout << res1 << ' ' << res2 << '\n';
}
```

我不认为通过参数返回结果是一种很优雅的方法。我们将在 5.3.5.1 节再次讨论这个问题。

5.3.4　共享数据

有时任务间需要共享数据。此时数据访问必须进行同步，以确保在同一时刻至多有一个任务能访问数据。有经验的程序员可能认为这是一种简单化的方法（例如，很多任务同时读取不变的数据是没有任何问题的），但无论如何，确保在同一时刻至多有一个任务可以访问给定的对象是很有意义的。

解决此问题的基础是“互斥对象”mutex。thread 使用 lock() 操作来获取一个互斥对象：

```
mutex m; // 控制共享数据访问的 mutex
int sh;  // 共享的数据

void f()
{
    unique_lock<mutex> lck {m}; // 获取 mutex
    sh += 7;                    // 处理共享数据
}   // 隐式释放 mutex
```

unique_lock 的构造函数获取了互斥对象（通过调用 m.lock()）。如果另一个线程已经获取了互斥对象，则当前线程会等待（“阻塞”）直至那个线程完成对共享数据的访问。一旦线程完成了对共享数据的访问，unique_lock 会释放 mutex（通过调用 m.unlock()）。互斥和锁机制在头文件 <mutex> 中提供。

共享对象和 mutex 间是一种常规的对应关系：程序员只需知道哪个 mutex 对应哪个数据即可。显然这很容易出错，我们最好努力借助多种语言特性来使这样的对应关系更为清晰。例如：

```
class Record {
public:
    mutex rm;
    // ...
};
```

对于一个名为 rec 的 Record，不难猜测 rec.rm 是一个 mutex，在访问 rec 的其他数据前应该先获取这个互斥对象。可见，通过注释或好的命名方式可以提高程序的可读性。

需要同时访问多个资源来执行一个操作的情况并不罕见，这可能导致死锁。例如，如果 thread1 获取了 mutex1 然后试图获取 mutex2，而同时 thread2 已经获取了 mutex2 然后试图获取 mutex1，则两个任务都无法继续执行了。标准库提供了一个同时获取多个锁的操作，可以帮助解决这个问题：

```
void f()
{
    // ...
    unique_lock<mutex> lck1 {m1,defer_lock};    // 推迟加锁：还未尝试获取 mutex
    unique_lock<mutex> lck2 {m2,defer_lock};
    unique_lock<mutex> lck3 {m3,defer_lock};
    // ...
    lock(lck1,lck2,lck3);                       // 获取全部三个锁
    // ... 处理共享数据 ...
} // 隐式释放所有 mutex
```

lock() 调用只有在获取了全部 mutex 实参后才会继续执行，当它持有 mutex 时，绝不会阻塞（“睡眠”），当然也就不会导致死锁。unique_lock 的析构函数保证了当 thread 离开作用域时 mutex 会被释放。

通过共享数据进行通信是一种很底层的方式。特别是，程序员必须想方设法了解不同的任务已经做了哪些工作，又有哪些工作尚未完成。在这方面，使用共享数据不如调用 – 返回模式。另一方面，有些人深信数据共享肯定比参数拷贝和结果返回更高效。如果处理大量数据，这种观点可能确实是对的，但同时加锁和解锁也是代价相当高的操作。而且，现代计算机拷贝数据的效率已经很高，特别是紧凑的数据，如 vector 的元素。因此，不要为了所谓“效率”就不经思考、不经测试地选择使用共享数据的方式来进行线程间通信。

5.3.4.1 等待事件

有时候 thread 需要等待某种外部事件，比如另一个 thread 完成了任务或是已经过去了一段时间。最简单的“事件”就是时间流逝。请考虑如下的代码：

```
using namespace std::chrono;    // 见 35.2 节

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20});
auto t1 = high_resolution_clock::now();
cout << duration_cast<nanoseconds>(t1-t0).count() << " nanoseconds passed\n";
```

注意，我甚至没有启动一个 thread，this_thread 默认指向唯一的线程（见 42.2.6 节）。

我使用 duration_cast 将时钟单位调整为期望的纳秒。要想尝试更多关于时间的操作，请先阅读 5.4.1 节和 35.2 节。C++ 的时间功能位于 <chrono> 头文件中。

通过外部事件实现线程间通信的基本方法是使用 condition_variable，它定义在 <condition_variable> 中（见 42.3.4 节）。condition_variable 提供了一种机制，允许一个 thread 等待另一个 thread。特别是，它允许一个 thread 等待某个条件（condition，通常称为一个事件，event）发生，这种条件通常是其他 thread 完成工作产生的结果。

考虑两个 thread 通过 queue 传递消息的经典例子。为简单起见，我声明 queue 对象，以及生产者、消费者共享 queue 同时避免竞争条件的机制如下：

```
class Message {     // 通信的对象
    // ...
};

queue<Message> mqueue;          // 消息的队列
condition_variable mcond;       // 通信用的条件变量
mutex mmutex;                   // 锁机制
```

其中的类型 queue、condition_variable 和 mutex 由标准库提供。

consumer() 读取并处理 Message：

```
void consumer()
{
    while(true) {
        unique_lock<mutex> lck{mmutex};          // 获取 mmutex
        while (mcond.wait(lck)) /* do nothing */; // 释放 lck 并等待
                                                 // 被唤醒后重新获取 lck
        auto m = mqueue.front();                 // 获取消息
        mqueue.pop();
        lck.unlock();                            // 释放 lck
        // ... 处理 m ...
    }
}
```

此例中，我通过一个 mutex 上的 unique_lock 显式保护对 queue 和 condition_variable 的操作。线程在 condition_variable 上等待时，会释放已持有的锁，直至被唤醒后（此时队列非空）重新获取锁。

对应的 producer 可以这样编写：

```
void producer()
{
    while(true) {
        Message m;
        // ... 填入消息 ...
        unique_lock<mutex> lck {mmutex};         // 保护队列上的操作
        mqueue.push(m);
        mcond.notify_one();                      // 通知
    }                                            // 释放锁（在作用域结束）
}
```

使用 condition_variable 可以帮助我们完成很多既优雅又有效的数据共享，但是绝非一直如此（见 42.3.4 节）。

5.3.5　任务通信

标准库提供了一些特性，允许程序员在抽象的任务层（工作并发执行）进行操作，而不是在底层的线程和锁的层次直接进行操作。

［1］ future 和 promise 用来从一个独立线程上创建出的任务返回结果。

［2］ packaged_task 是帮助启动任务以及连接返回结果的机制。

［3］ async() 以非常类似调用函数的方式启动一个任务。

这些特性都定义在 <future> 中。

5.3.5.1 future 和 promise

future 和 promise 的关键点是它们允许在两个任务间传输值，而无须显式使用锁——“系统”高效地实现了这种传输。基本思路很简单：当一个任务需要向另一个任务传输某个值时，它把值放入 promise 中。具体的 C++ 实现以自己的方式令这个值出现在对应的 future 中，然后就可以从其中读取这个值了（通常是任务的启动者读取此值）。这种模式如下图所示：



如果我们有一个名为 fx 的 future<X>，则可以使用 get() 得到一个类型为 X 的值：

```
X v = fx.get();  // if necessary, wait for the value to get computed
```

如果值还未准备好，线程会阻塞直至值准备好。如果值无法正确地计算出来，则 get() 会抛出一个异常（可能是系统抛出的，也可能是从使用 get() 得到数据的任务传递来的）。

promise 的主要目的是提供与 future 的 get() 相匹配的简单的“放置”操作（名为 set_value() 和 set_exception()）。“期货”（future）和“承诺”（promise）的命名是历史遗留问题，所以请不要批判或赞美我。现实中像这样的双关语有很多。

如果你有一个 promise，并且需要把类型为 X 的结果发送给 future，那么你要么传递一个值，要么传递一个异常。例如：

```
void f(promise<X>& px)  // 一个任务：将结果放在 px 中
{
    // ...
    try {
        X res;
        // ... 计算一个值，保存在 res 中 ...
        px.set_value(res);
    }
    catch (...) {           // 糟糕：不能正确计算 res
        // 将异常传递给 future 的线程
        px.set_exception(current_exception());
    }
}
```

current_exception() 表示捕获的异常（见 30.4.1.2 节）。

为了处理经过 future 传递的异常，get() 的调用者必须准备好在某处捕获它。例如：

```
void g(future<X>& fx)       // 一个任务：从 fx 获取结果
{
    // ...
    try {
        X v = fx.get();  // 如必要，等待值准备好
```

```
        // ... 使用 v ...
    }
    catch (...) {          // 糟糕：v 不能正确计算
        // ... 处理错误 ...
    }
}
```

5.3.5.2 packaged_task

我们应该如何向一个需要结果的任务引入 future？又如何向一个生成结果的线程引入对应的 promise 呢？标准库提供了 packaged_task 类型简化任务连接 future 和 promise 的设置。packaged_task 提供了一层包装代码，负责把某个任务的返回值或异常放入一个 promise 中（就像 5.3.5.1 节中代码所做的那样）。如果通过调用 get_future() 来向一个 packaged_task 发出请求，它会返回给你对应 promise 的 future。例如，我们可以将两个任务连接起来，它们各自使用标准库 accumulate()（见 3.4.2 节和 40.6 节）算法将一个 vector<double> 中的一半元素累加起来：

```
double accum(double* beg, double* end, double init)
    // 计算 [beg,end) 中元素的和，计算的初始值是 init
{
    return accumulate(beg,end,init);
}

double comp2(vector<double>& v)
{
    using Task_type = double(double*,double*,double);          // 任务的类型

    packaged_task<Task_type> pt0 {accum};                      // 打包任务（即 accum）
    packaged_task<Task_type> pt1 {accum};

    future<double> f0 {pt0.get_future()};                      // 获取 pt0 的 future
    future<double> f1 {pt1.get_future()};                      // 获取 pt1 的 future

    double* first = &v[0];
    thread t1 {move(pt0),first,first+v.size()/2,0};            // 为 pt0 启动一个线程
    thread t2 {move(pt1),first+v.size()/2,first+v.size(),0};   // 为 pt1 启动一个线程

    // ...

    return f0.get()+f1.get();                                  // 获取结果
}
```

packaged_task 模板接受模板参数表示任务的类型（本例中为 Task_type，即 double (double*,double*,double) 的别名），并接受构造函数参数作为任务（本例中为 accum）。因为 packaged_task 不能被拷贝，所以 move() 操作是必需的。

请注意这段代码没有显式地使用锁：通过使用 packaged_task，我们可以集中精力于要完成的任务，而不必操心该如何管理它们之间的通信。两个任务运行于两个独立的线程，因此可以并行执行。

5.3.5.3 async()

我在本章中所遵循的思路是：将任务当作可以与其他任务并发执行的函数来处理，这也是在所有思路中我认为最简单的，但同时又不失其强大性。它并非 C++ 标准库所支持的唯一模型，但它能很好地满足广泛的需求。一些更为微妙和复杂的模型，如依赖于共享内存的

程序设计风格，可以在需要时使用。

如需启动可异步运行的任务，我们可以使用 async()：

```
double comp4(vector<double>& v)
    // 如果 v 足够大，则创建很多任务
{
    if (v.size()<10000) return accum(v.begin(),v.end(),0.0);

    auto v0 = &v[0];
    auto sz = v.size();

    auto f0 = async(accum,v0,v0+sz/4,0.0);          // 第一个四分之一
    auto f1 = async(accum,v0+sz/4,v0+sz/2,0.0);     // 第二个四分之一
    auto f2 = async(accum,v0+sz/2,v0+sz*3/4,0.0);   // 第三个四分之一
    auto f3 = async(accum,v0+sz*3/4,v0+sz,0.0);     // 第四个四分之一

    return f0.get()+f1.get()+f2.get()+f3.get(); // 收集并组合结果
}
```

async() 将一个函数调用的“调用部分”和“获取结果部分”分离开来，并将这两部分与任务的实际执行分离开来。通过使用 async()，你不必再操心线程和锁，而只需考虑可能异步执行的任务。这种做法显然受到了限制：不要试图对共享资源且需要用锁机制的任务使用 async()——使用 async()，你甚至不知道要使用多少个 thread，因为这是由 async() 来决定的，它根据它所了解的调用发生时系统可用资源量来确定使用多少个 thread。例如，async() 会先检查有多少可用核（处理器）再确定启动多少 thread。

请注意，async() 并非专门为并行计算提高性能所设计的机制。例如，我们还可以用它来创建一个任务从用户那里获取信息，而让“主程序”继续进行其他计算（见 42.4.6 节）。

5.4 小工具组件

并非所有标准库组件都有个像“容器”和“I/O”这样响当当的名字，本节介绍几种不太显眼但是应用非常广泛的组件。

- clock 和 duration，用于度量时间。
- iterator_traits 和 is_arithmetic 等类型函数，用于获取关于类型的信息。
- pair 和 tuple，用于表示规模较小且由异构数据组成的集合。

这里提到的函数或者类型不需要太复杂，也不必与其他函数或者类型有太多牵连，它们本身就非常有用。这类库组件经常用于实现更重要的库功能，或者用于组成标准库的其他组件。

5.4.1 时间

标准库提供了一些功能，我们可以利用这些功能完成与时间有关的任务。例如，下面这段程序实现了最基本的计时：

```
using namespace std::chrono;     // 见 35.2 节

auto t0 = high_resolution_clock::now();
do_work();
auto t1 = high_resolution_clock::now();
cout << duration_cast<milliseconds>(t1-t0).count() << "msec\n";
```

系统时钟返回一个 time_point 类型的值（时间点），两个 time_point 相减的结果是 duration（时间段）。不同的时钟得到的时间单位各有不同（这里用到的时钟单位是 nanoseconds），所以在实际使用时，最好把 duration 统一转换成一个公认的单位。在上面的代码中，duration_cast 负责完成这一任务。

处理时间的标准库功能定义在 <chrono> 头文件中，属于子名字空间 std::chrono。

判断程序“效率”最有效的办法是统计程序运行的时间，仅靠猜测很难做出正确的判断。

5.4.2 类型函数

类型函数（type function）是指在编译时求值的函数，它接受一个类型作为实参或者返回一个类型作为结果。标准库提供了大量的类型函数，这些函数可以帮助库的实现者及程序员在编写代码时充分利用语言、标准库以及其他代码的优势。

对于数字类型来说，<limits> 的 numeric_limits 提供了一些有用的信息（见 5.6.5 节）。例如：

```
constexpr float min = numeric_limits<float>::min();      // 最小的正浮点数（见 40.2 节）
```

与之类似，我们可以使用内置的 sizeof 运算符（见 2.2.2 节）获取对象的大小。例如：

```
constexpr int szi = sizeof(int); // int 所占的字节数量
```

类型函数是 C++ 的编译时计算机制的一部分，它允许程序进行更严格的类型检查以获取更优的性能。我们通常把这种用法称为元编程（metaprogramming）或者模板元编程（template metaprogramming，当含有模板时，见第 28 章）。接下来，我们介绍标准库提供的两种有用功能：iterator_traits（见 5.4.2.1 节）和类型谓词（见 5.4.2.2 节）。

5.4.2.1 iterator_traits

标准库 sort() 函数接受一对迭代器作为参数，这对迭代器通常表示序列的两端（见 4.5 节）。而且，这两个迭代器必须提供对序列的随机访问，也就是说它们必须是随机访问迭代器（random-access iterator）。某些容器（比如 forward_list）无法提供满足要求的迭代器。尤其是，forward_list 是一个单链表，对它进行取下标操作的代价非常昂贵，而且要想访问当前元素的前一个元素也不太容易。不过和大多数容器一样，forward_list 提供了前向迭代器（forward iterator），这样其他算法和 for 语句就能遍历序列的元素了（见 33.1.1 节）。

我们可以用标准库提供的 iterator_traits 机制检查当前容器支持哪种迭代器，这样我们就能让 4.5.6 节的 sort() 函数既支持 vector 又支持 forward_list 了。例如：

```
void test(vector<string>& v, forward_list<int>& lst)
{
    sort(v);   // 排序 vector
    sort(lst); // 排序单链表
}
```

显然，如果有某种技术能让上面的代码合法和有效，那这样的技术会非常有用。

为了实现这一目的，我们首先编写两个辅助器函数。它们分别接受一个额外的实参以区分是用于随机访问迭代器还是前向迭代器。其中，接受随机访问迭代器的版本没什么特别之处：

```
template<typename Ran>                                          // 对于随机访问迭代器的情况
void sort_helper(Ran beg, Ran end, random_access_iterator_tag)  // 能够使用下标运算符随机访问
{                                                               // [beg:end) 内的元素
    sort(beg,end);      //执行排序操作

}
```

接受前向迭代器的版本的工作机理是：在其内部先把列表拷贝给 vector，接着在 vector 上执行排序操作，最后再拷贝回列表：

```
template<typename For>                                  // 对于前向迭代器的情况
void sort_helper(For beg, For end, forward_iterator_tag)  // 能够依次遍历 [beg:end) 内的元素
{
    vector<decltype(*beg)> v {beg,end};   // 用 [beg:end) 内的元素初始化一个 vector
    sort(v.begin(),v.end());
    copy(v.begin(),v.end(),beg);          // 把元素拷贝回列表
}
```

decltype() 是内置类型函数，返回其实参的已声明类型（见 6.3.6.3 节）。我们得到的 v 是一个 vector<X>，其中 X 是输入序列的元素类型。

真正的“类型魔法”发生在选择辅助器函数时：

```
template<typname C>
void sort(C& c)
{
    using Iter = Iterator_type<C>;
    sort_helper(c.begin(),c.end(),Iterator_category<Iter>{});
}
```

在这里我们使用了两个类型函数：Iterator_type<C> 返回 C 的迭代器类型（即 C::iterator），而 Iterator_category<Iter>{} 构建了一个“标签”值以指示提供的是哪种迭代器：

- 如果 C 的迭代器支持随机访问，则取值为 std::random_access_iterator_tag。
- 如果 C 的迭代器支持前向访问，则取值为 std::forward_iterator_tag。

有了这个标签值，就能在编译时从两种排序算法中选择一种供我们使用了。这种技术称为标签分发（tag dispatch），它在标准库和其他地方经常被用到以提高程序的灵活性和效率。

标准库对于像标签分发这种迭代器技术的支持是以简单类模板 iterator_traits 的形式提供的，iterator_traits 定义在 <iterator> 中（见 33.1.3 节）。我们可以在 sort() 内部很容易地定义类型函数：

```
template<typename C>
    using Iterator_type = typename C::iterator;   // C 的迭代器类型

template<typename Iter>
    using Iterator_category = typename std::iterator_traits<Iter>::iterator_category;  // Iter 的类别
```

如果你对于在标准库中使用了什么样的“编译时类型魔法”不感兴趣，大可以忽略掉像 iterator_traits 这样的功能。不过与此同时，你也就没办法利用它们来改进你的代码了。

5.4.2.2 类型谓词

标准库类型谓词是一种简单的类型函数，它负责回答一个关于类型的问题。例如：

```
bool b1 = Is_arithmetic<int>();      // Yes，int 是一种算术类型
bool b2 = Is_arithmetic<string>();   // No，std::string 不是一种算术类型
```

读者可以在 <type_traits> 中找到类似的谓词，35.4.1 节会介绍相关的知识。一些典型

的例子包括 is_class、is_pod、is_literal_type、has_virtual_destructor 和 is_base_of。我们在编写模板时经常会用到这些谓词，例如：

```
template<typename Scalar>
class complex {
    Scalar re, im;
public:
    static_assert(Is_arithmetic<Scalar>(), "Sorry, I only support complex of arithmetic types");
    // ...
};
```

为了提高代码的可读性，使其与直接使用标准库相当，我们不妨定义一个类型函数：

```
template<typename T>
constexpr bool Is_arithmetic()
{
    return std::is_arithmetic<T>::value ;
}
```

旧式代码习惯于直接使用 ::value 而非 ()，不过前者既不美观又容易暴露实现的细节，不建议读者使用。

5.4.3 pair 和 tuple

在有些情况下，我们希望数据就是数据。换句话说，我们想要的仅仅是一组值，而非定义了良好语义的类的对象或者含有值的变量（见 2.4.3.2 节和 13.4 节）。此时，我们可以自己定义一个简单的 struct，并且为它的每个成员起个合适的名字，也可以让标准库帮我们定义。例如，标准库算法 equal_range（见 32.6.1 节）返回迭代器的一个 pair，表示一个满足给定谓词的子序列：

```
template<typename Forward_iterator, typename T, typename Compare>
    pair<Forward_iterator,Forward_iterator>
    equal_range(Forward_iterator first, Forward_iterator last, const T& val, Compare cmp);
```

给定一个有序序列 [first:last)，equal_range() 返回表示某个子序列的 pair，该子序列中元素都满足谓词 cmp。我们可以用它在一个有序的 Record 序列中进行搜索：

```
auto rec_eq = [](const Record& r1, const Record& r2) { return r1.name<r2.name;};// 比较名字的大小

void f(const vector<Record>& v)          // 假定 v 的元素根据 "name" 字段排好了序
{
    auto er = equal_range(v.begin(),v.end(),Record{"Reg"},rec_eq);
    for (auto p = er.first; p!=er.second; ++p)      // 输出所有匹配的记录
        cout << *p;                                 // 假定 Record 定义了 << 操作
}
```

pair 的第一个成员是 first，第二个成员是 second。这样的命名方式看起来怪怪的，一点新意也没有，不过当我们编写某些具有通用性的代码时会从中受益良多。

标准库 pair（定义在 <utility> 中）被用于实现很多其他标准库组件。pair 提供了一些运算符，比如 =、== 和 <，不过前提是它的元素得支持这些运算。我们可以用 make_pair() 函数快捷地创建一个 pair，而无须显式指定它的类型（见 34.2.4.1 节）。例如：

```
void f(vector<string>& v)
{
    auto pp = make_pair(v.begin(),2);   // pp 的类型是 pair<vector<string>::iterator, int>
    // ...
}
```

如果你用到的元素个数不止两个（或者不足两个），则应该使用 tuple（定义在 <utility> 中，见 34.2.4.2 节）。tuple 表示任意形式的元素序列，例如：

```
tuple<string,int,double> t2("Sild",123, 3.14); // 显式地指定了类型

auto t = make_tuple(string("Herring"),10, 1.23);    // 隐式地推断出类型是
                                                    // tuple<string,int,double>

string s = get<0>(t); // 获取 tuple 的第一个元素："Herring"
int x = get<1>(t);
double d = get<2>(t);
```

tuple 的每个元素都对应一个编号，从 0 开始依次排列；而 pair 的元素有自己的名字（first 和 second）。要想在编译时从 tuple 当中选取元素，只能使用 get<1>(t) 的方式（尽管看起来不够简洁），而不能写成 get(t,1) 或者 t[1]（见 28.5.2 节）。

与 pair 类似，只要 tuple 的元素支持赋值操作和比较操作，我们就能对整个 tuple 赋值和比较。

因为我们常常希望从接口函数中返回两个结果（比如结果本身和一个表示结果好不好的标志位），所以总是会用到 pair。与它相比 tuple 就没那么多用处了，毕竟同时返回三个或者更多结果的时候不太多。tuple 一般用于实现泛型算法。

5.5 正则表达式

正则表达式是一种很强大的文本处理工具，它提供了一种简单、精练的方法描述文本中的模式（形如 TX 77845 的美国邮政编码，或者形如 2009-06-07 的 ISO 风格日期），还提供了在文本中高效查找模式的方法。在 <regex> 中，标准库定义了 std::regex 类及其支持函数，提供对正则表达式的支持。下面是一个模式的定义，你可以从中领略 regex 库的风格：

```
regex pat (R"(\w{2}\s*\d{5}(-\d{4})?)");  // 美国邮政编码模式：XXddddd-dddd 及其变形
cout << "pattern: " << pat << "\n";
```

在其他语言中使用过正则表达式的人会发现 \w{2}\s *\d{5}(-\d{4})? 很熟悉。它指定了一种以 2 个字母开始（\w{2}）的模式，后面是可选的若干空白符 \s *，再接下来是 5 个数字 \d{5}，然后是可选的一个破折号和 4 个数字 -\d{4}。如果你还不熟悉正则表达式，别犹豫了，马上开始（[Stroustrup，2009][Maddock，2009] 和 [Friedl，1997]）。37.1.1 节会概述与正则表达式有关的内容。

为了表达模式，我使用了一个*原始字符串字面常量*（raw string literal，见 7.3.2.1 节），它以 R"(开始，以 ") 结束。原始字符串字面常量的好处是可以直接包含反斜线和引号而无须转义，因此非常适合表示正则表达式。

正则表达式最常见的应用场景是在数据流中搜索符合某一模式的字符串：

```
int lineno = 0;
for (string line; getline(cin,line);) {          // 把数据读入缓冲区
    ++lineno;
    smatch matches;                              // 用于存放匹配的字符串
    if (regex_search(line,matches,pat))          // 在一行字符串中查找符合模式 pat 的子串
        cout << lineno << ": " << matches[0] << '\n';
}
```

regex_search(line,matches,pat) 负责在读入的 line 中查找所有符合模式 pat 的子

串。如果找到了，把它们存入 matches 中；如果没找到，regex_search(line,matches,pat) 返回 false。matches 的类型是 smatch，其中字符“s”表示“子（sub）”或“字符串（string）”。所以 smatch 就是一个包含 string 类型匹配子串的 vector。代码中的第 1 个元素 matches[0] 是匹配得到的结果。

关于正则表达式的更多细节请读者参阅第 37 章。

5.6 数学计算

最初设计 C++ 语言时，数值计算并非关注的焦点。然而，数值计算在很多场景中都发挥着重要作用，因此标准库提供了很多对数值计算的支持。

5.6.1 数学函数和算法

在 <cmath> 中包含着很多“有用的数学函数”，如 sqrt()、log() 和 sin() 等，它们支持各种各样的实参类型（float、double、long double，见 40.3 节）。这些函数的复数版本则定义在 <complex> 中（见 40.4 节）。

在 <numeric> 中有一些泛化的数值算法，比如 accumulate()，它的使用示例是：

```
void f()
{
    list<double> lst {1, 2, 3, 4, 5, 9999.99999};
    auto s = accumulate(lst.begin(),lst.end(),0.0); // 求和操作
    cout << s << '\n';                               // 输出 10014.9999
}
```

这些算法可以作用于任意一种标准库序列，同时接受某种运算符作为其实参（见 40.6 节）。

5.6.2 复数

标准库提供了一系列复数类型，其形式与 2.3 节描述的 complex 类有些类似。为了让复数的标量可以取单精度浮点数（float）、双精度浮点数（double）等不同类型，标准库把 complex 定义成了模板：

```
template<typename Scalar>
class complex {
public:
    complex(const Scalar& re ={}, const Scalar& im ={});
    // ...
};
```

标准库复数类型支持常见的算术操作和数学函数，例如：

```
void f(complex<float> fl, complex<double> db)
{
    complex<long double> ld {fl+sqrt(db)};
    db += fl*3;
    fl = pow(1/fl,2);
    // ...
}
```

<complex> 定义了一些常见的数学函数，sqrt() 和 pow()（求幂指数）即在其中。更多细节请参阅 40.4 节。

5.6.3　随机数

随机数在测试、游戏、仿真和安全等很多问题中都非常有用。为了适应各种各样的应用需求，标准库在 <random> 中提供了很多种不同的随机数生成器。随机数生成器包括两部分：

［1］一个引擎（engine），负责生成一组随机值或者伪随机值。

［2］一种分布（distribution），负责把引擎产生的值映射到某个数学分布上。

常用的分布包括 uniform_int_distribution（生成的所有整数概率相等）、normal_distribution（正态分布，又名“钟鸣曲线”）和 exponential_distribution（指数增长），它们的范围各不相同。例如：

```
using my_engine = default_random_engine;                // 引擎类型
using my_distribution = uniform_int_distribution<>;     // 分布类型

my_engine re {};                                        // 默认引擎
my_distribution one_to_six {1,6};                       // 该分布把随机数映射到 1 - 6 的范围
auto die = bind(one_to_six,re);                         // 得到一个随机数生成器

int x = die();                                          // 掷骰子：x 得到的值位于 1 - 6 之间
```

标准库函数 bind() 生成一个函数对象，它会把第 2 个参数（re）作为实参绑定到第一个参数（one_to_six 函数对象）的调用中（见 33.5.1 节）。因此，调用 die() 等价于调用 one_to_six(re)。

在设计和实现标准库随机数组件时，我们非常注重它的泛化能力和性能，因此曾经有一位专家把它评价为“实现随机数库的榜样和标杆”。不过它对于入门级的程序员来说稍显繁杂且不够友好。在上面的代码中，我们用 using 语句稍微解释了一下生成随机数的过程，当然也可以写成下面的形式：

```
auto die = bind(uniform_int_distribution<>{1,6}, default_random_engine());
```

至于说哪个版本更易读完全取决于代码上下文和读者自己的感觉。

对于初学者来说，完整版本的随机数生成器显得有些过于正式且不易使用，有一个简易版本可作为替代。例如：

```
Rand_int rnd {1,10};        // 构建一个随机数生成器，生成 1 - 10 之间的随机数
int x = rnd();              // x 是 1 - 10 之间的随机数
```

我们该如何得到这个新版本的随机数生成器呢？显然必须在 Rand_int 类的内部实现与 die() 类似的功能才行：

```
class Rand_int {
public:
    Rand_int(int low, int high) :dist{low,high} { }
    int operator()() { return dist(re); }          // 得到一个 int
private:
    default_random_engine re;
    uniform_int_distribution<> dist;
};
```

Rand_int() 的定义仍然是“专家级的”，不过使用起来已经变得很容易了，甚至初学者在 C++ 课程的第一周就能学会使用它。例如：

```
int main()
{
    Rand_int rnd {0,4};        // 创建一个随机数生成器
```

```
    vector<int> histogram(5);           // 构建一个相应尺寸的 vector
    for (int i=0; i!=200; ++i)
        ++histogram[rnd()];             // 用 [0:4] 之间每个数字出现的次数填充 histogram

    for (int i = 0; i!=mn.size(); ++i) {    // 输出直方图
        cout << i << '\t';
        for (int j=0; j!=mn[i]; ++j) cout << '*';
        cout << endl;
    }
}
```

输出结果是如下所示的一个均匀分布（统计差异在合理范围之内）：

```
0    ********************************************************
1    ********************************************************
2    ********************************************************
3    ********************************************************
4    ********************************************************
```

因为 C++ 没有标准图形库，所以我们使用了“ASCII 图形”。众所周知，有很多为 C++ 设计的开源或者商业的 GUI 库，但是在本书中我们尽量只使用 ISO 标准之内的功能。

关于随机数的更多内容请参阅 40.7 节。

5.6.4 向量算术

4.4.1 节介绍的 vector 被设计成一种通用的机制，它可以存放值的序列并且足够灵活，也能够适应容器、迭代器和算法的体系结构，但是它不支持数学意义上的向量运算。为 vector 提供这类运算并不难，但是 vector 对于通用性和灵活性的要求限制了数值计算所需的优化操作，因此，标准库在 <valarray> 中提供了一个类似于 vector 的模板 valarray。与 vector 相比，valarray 的通用性不强，但是对于数值计算进行了必要的优化：

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

valarray 支持常见的算术运算和大多数数学函数，例如：

```
void f(valarray<double>& a1, valarray<double>& a2)
{
    valarray<double> a = a1*3.14+a2/a1;         // 适用于数字序列的算术运算 *、-、/ 和 =
    a2 += a1*3.14;
    a = abs(a);
    double d = a2[7];
    // ...
}
```

更多信息请参阅 40.5 节。值得注意的是，valarray 为实现多维运算提供了足够的支持。

5.6.5 数值限制

在 <limits> 中，标准库提供了描述内置类型属性的类，比如 float 的最高阶以及 int 所占的字节数等（见 40.2 节）。举个例子，我们可以用下面的语句断言 char 是带符号的类型：

```
static_assert(numeric_limits<char>::is_signed,"unsigned characters!");
static_assert(100000<numeric_limits<int>::max(),"small ints!");
```

请注意，因为 numeric_limits<int>::max() 是一个 constexpr 函数（见 2.2.3 节和 10.4 节），所以第 2 个断言是有效的。

5.7　建议

［1］ 用资源句柄管理资源（RAII）；5.2 节。
［2］ 用 unique_ptr 访问多态类型的对象；5.2.1 节。
［3］ 用 shared_ptr 访问共享的对象；5.2.1 节。
［4］ 用类型安全的机制处理并发；5.3 节。
［5］ 最好避免共享数据；5.3.4 节。
［6］ 不要为了所谓“效率”而不经思考、不经测试地选择使用共享数据；5.3.4 节。
［7］ 从并发执行任务的角度进行设计，而不是直接从 thread 角度思考；5.3.5 节。
［8］ 一个库是否有用，与它的规模和复杂程度无关；5.4 节。
［9］ 别轻易抱怨程序的效率低下，记得用事实说话；5.4.1 节。
［10］ 编写代码时可以显式地令其利用某些类型的属性；5.4.2 节。
［11］ 利用正则表达式简化模式匹配任务；5.5 节。
［12］ 进行数值计算时优先使用库而非语言本身；5.6 节。
［13］ 用 numeric_limits 访问数值类型的属性；5.6.5 节。

第二部分

The C++ Programming Language, Fourth Edition

基本功能

本部分主要介绍C++的内置类型以及编写程序所要用到的基本功能。其中既包括C++的C子集，也包括C++对传统程序设计风格的进一步支持。我们还将讨论从问题的逻辑和物理构成出发，使用那些基本功能能组合生成对应的C++程序。

"……长久以来，我对哲学先贤们的大多数见解都持怀疑的态度。在我身上有一种倾向，即对于别人的结论总是愿意争论而非苟同。哲学家们常犯的一个错误是，他们总是对提出的准则进行过多限定，而忽视了事物的多样性，殊不知丰富多彩、不拘一格才是自然界的本质。哲学家们一旦提出一条"心爱的"理论并且找到一些证据，就想当然地认为这条理论是放之四海而皆准的，全然不顾他们的推理过程是多么简单粗暴和荒谬无理……"

——大卫·休谟

《道德、政治与文学论文集（第1卷）》，1752年

第 6 章
The C++ Programming Language, Fourth Edition

类型与声明

只有接近崩溃的那一刻，
才能达到完美。

——C. N. 帕金森

- ISO C++ 标准
  实现；基本源程序字符集
- 类型
  基本类型；布尔值；字符类型；整数类型；浮点数类型；前缀和后缀；void；类型尺寸；对齐
- 声明
  声明的结构；声明多个名字；名字；作用域；初始化；推断类型：auto 和 decltype()
- 对象和值
  左值和右值；对象的生命周期
- 类型别名
- 建议

6.1 ISO C++ 标准

C++ 语言和标准库对应的 ISO 标准是 ISO/IEC 14882:2011，本书记为 § iso.23.3.6.1。如果读者担心书中某处的内容不准确、不完整，或者存在错误，请随时参阅 ISO 标准文档。不过对于大多数非专业人员来说，要想读懂标准文档可不像看看参考书那么简单。

即使在编程时你严格遵循 C++ 语言和库的规范，也不能确保写出的代码一定是简洁、高效和可移植的。所谓“标准”不能用来判断一段代码是好是坏，它仅仅规定了程序员能做什么和不能做什么。有时候，我们需要访问某些 C++ 无法直接操纵或者依赖于特殊实现细节的系统接口或硬件功能，这种任务很难，因此程序员也许会编写出虽然符合标准但非常糟糕的代码。此外，大多数实际应用所依赖的语言特性也不一定是可移植的。

在 C++ 标准之下，很多重要的功能都是依赖于实现的（implementation-defined）。这意味着对于语言的某个概念来说，每个实现版本都必须为之设定恰当的、定义良好的语法行为，同时详细记录行为规范。例如：

```
unsigned char c1 = 64;      // 定义良好：在任何情况下 char 都至少包含 8 个二进制位，肯定能容下 64
unsigned char c2 = 1256;    // 依赖于实现的：如果当前情况下 char 只占 8 位，则值将被截断
```

因为在任何情况下 char 都至少包含 8 个二进制位，所以 c1 的初始化操作是定义良好的。与之相反，c2 的初始化操作则依赖于实现，毕竟 char 到底占多少位在不同的实现版本中可能不一样。如果 char 只占 8 位，则 1256 被截断成 232（见 10.5.2.1 节）。大多数依赖于实现的功能都与运行程序的硬件系统密切相关。

还有一种行为是不确定的（unspecified）。意思是几种行为都有可能发生，但是实现者没有说明实际发生的是其中哪种。当由于某些原因造成确切的行为无法预知时，我们可以认为它是不确定的。例如，new 运算符的返回结果是不确定的。另一个例子是，假如两个线程同时为某个变量赋值而我们又没有提供同步机制以防止数据竞争（见 41.2 节），则该变量的值也将是不确定的。

在编写实际的应用程序时，常常会用到那些依赖于实现的行为。这一点都不奇怪，毕竟这样做可以让我们在各种系统上都取得较好的执行效率。例如，C++ 如果强制限定所有字符都占 8 位而所有指针都占 32 位，那么仅从语法上来说当然会简单很多。但是这种想法一点都不可行，毕竟 16 位和 32 位的字符集有很多，而使用 16 位或 64 位指针的机器也并不少见。

为了尽量满足可移植性的要求，聪明的做法是首先明确哪些特性是依赖于实现的，然后把这些敏感的部分整理在一起，放在程序的某个带有明显标记的位置中。一种被普遍认可的做法是，程序员常常把所有依赖于硬件的类型尺寸及定义放在头文件中。为了支持这一技术，标准库提供了 numeric_limits（见 40.2 节）。我们通过设置静态断言的方式（见 2.4.3.3 节）检查某些特性是否确实是依赖于实现的。例如：

```
static_assert(4<=sizeof(int),"sizeof(int) too small");
```

与依赖于实现的行为相比，不确定的行为是一种更糟糕的情况。如果具体实现无法为某一概念指定明确合理的行为，则 C++ 标准会认为它是未定义的。比如，某些我们习以为常的实现技术实际上会让程序包含未定义的行为，从而产生意想不到的结果。例如：

```
const int size = 4*1024;
char page[size];

void f()
{
    page[size+size] = 7;//未定义的
}
```

上述代码可能产生两种效果：把数值写入到一个毫不相关的内存区域或者导致一个硬件错误或异常。对于某个具体的实现来说，它不必非得在可能的结果中做出抉择。但是一旦我们使用了强有力的优化工具，未定义行为的实际效果就变得不可预知了。如果存在一组合理的且易于实现的结果，则我们认为这样的特性是不确定的或依赖于实现的，而不认为它是未定义的。

程序员应该投入更多精力以确保程序中没有不确定的或未定义的部分。很多时候一些现有的工具可以帮助我们完成这一任务。

6.1.1 实现

C++ 的一个具体实现可以有两种形式：宿主式（hosted）和独立式（freestanding）（§iso.17.6.1.3）。在宿主式实现中包含了 C++ 标准（见 30.2 节）和本书描述的所有标准库功能；独立式实现包含的标准库功能可能会少一些，但是肯定包含下面列举的这些。

独立式实现所含的头文件

类型	<cstddef>	10.3.1 节
实现属性	<cfloat> <limits> <climits>	40.2 节

（续）

独立式实现所含的头文件		
整数类型	<cstdint>	43.7 节
开始和终止	<cstdlib>	43.7 节
动态内存管理	<new>	11.2.3 节
类型信息	<typeinfo>	22.5 节
异常处理	<exception>	30.4.1.1 节
初始化器列表	<initializer_list>	30.3.1 节
其他运行时支持	<cstdalign> <cstdarg> <cstdbool>	12.2.4 节和 44.3.4 节
类型特性	<type_traits>	35.4.1 节
原子	<atomic>	41.3 节

在那些只提供最基本的操作系统功能的环境中，常采用独立式实现。其他很多实现也会设置一个非标准的可选项，它允许程序员在规模较小且主要用于硬件操作的程序中把异常处理模块排除在外。

6.1.2　基本源程序字符集

C++ 标准以及本书用到的示例程序都是基于基本源程序字符集（basic source character set）写成的，该字符集包括字母、数字、图形化字符和空白字符等。这些字符源于国际 7 位字符集 ISO 646-1983 的美国版本，也就是我们熟悉的 ASCII（ANSI3.4-1968）字符集。如果程序运行的环境使用的是其他字符集，可能会产生一些问题：

- ASCII 字符集含有标点符号和操作符号（比如 [、} 和 !），但是其他字符集可能没有。
- 有的字符本身没有一种可见的表示形式，我们需要为它们设计符号（比如换行符和“值为 17 的符号”）。
- ASCII 字符集中不包含那些英语之外的其他语言的字符（比如 ñ、Þ 和 Æ）。

要想在源代码中使用扩展字符集，编程环境需要把扩展字符集映射为基本源程序字符集。映射的方式有几种，其中之一是使用通用字符集名字（见 6.2.3.2 节）。

6.2　类型

观察下面的式子：

```
x = y+f(2);
```

要想让这个式子在 C++ 程序中有效，必须提前声明好名字 x、y 和 f。换句话说，程序员必须确保名字 x、y 和 f 对应的实体确实存在，且对于它们的类型来说 =（赋值）、+（加法）和 ()（函数调用）是有意义的。

C++ 程序中的每个名字（标识符）都对应一种数据类型。该类型决定了这个名字（即该名字代表的实体）能执行哪些运算以及如何执行这些运算。例如：

```
float x;        // x 是一个浮点型变量
int y = 7;      // y 是一个整型变量，它的初始值是 7
float f(int);   // f 是一个函数，它接受一个整数类型的参数，返回一个浮点数
```

有了这几条声明语句，一开始的那个式子就有意义了。因为我们把 y 声明成 int 类型，所以能给它赋值，也能把它作为 + 的运算对象；因为我们把 f 声明成接受 int 参数的函数，所以

能用整数值 2 调用它。

本章介绍基本类型（见 6.2.1 节）和声明（见 6.3 节），其中的示例程序仅用于描述语法特性，并不能解决什么实际问题。后面的章节会陆续引入一些应用面广且有实际意义的例子。本章介绍的内容是构成 C++ 程序所需的最基本元素，读者必须了解这些元素以及与之有关的技术和语法，这样才有能力用 C++ 编写出一段真正的代码，同时也才有可能读懂别人编写的代码。不过，也不是说必须掌握本章提到的每一个细节之后才能继续学习后续章节。建议读者先泛读本章，记下有哪些主要的概念，在以后用到的时候再返回来深入研究。

6.2.1 基本类型

C++ 包含一套基本类型（fundamental type），这些类型对应计算机最基本的存储单元并且展现了如何利用这些单元存储数据。

6.2.2 节 布尔值类型（bool）

6.2.3 节 字符类型（比如 char 和 wchar_t）

6.2.4 节 整数类型（比如 int 和 long long）

6.2.5 节 浮点数类型（比如 double 和 long double）

6.2.7 节 void 类型，用以表示类型信息缺失

基于上述类型，我们可以用声明符构造出更多类型：

7.2 节 指针类型（比如 int*）

7.3 节 数组类型（比如 char[]）

7.7 节 引用类型（比如 double& 和 vector<int>&&）

除此之外，用户还能自定义类型：

8.2 节 数据结构和类（第 16 章）

8.4 节 枚举类型，用以表示特定值的集合（enum 和 enum class）。

其中，布尔值、字符和整数统称为整型（integral type），整型和浮点型进一步统称为算术类型（arithmetic type）。我们把枚举类型和类（第 16 章）称为用户自定义类型（user-defined type），因为用户必须先定义它们，然后才能使用；这一点显然与基本类型无须声明可以直接使用的方式不同。与之相反，我们把基本类型、指针和引用统称为内置类型（built-in type）。标准库提供了很多种精妙的用户自定义类型（第 4 章和第 5 章）。

整型和浮点型包含的具体类型很多，它们的尺寸各不相同，程序员可以根据计算任务所需的存储空间大小、精度和表示范围选择适当的类型（见 6.2.8 节）。在设计这些类型时，我们假设计算机系统的字节可以存放字符、字可以存放和计算整数值、某些实体可以执行浮点计算，而内存地址可以用来引用或指向上述实体。C++ 的基本类型以及指针和数组以一种与实现无关的方式把这些机器级别的思想和概念呈现在程序员面前，供他们使用。

对于大多数应用来说，我们用 bool 表示布尔值、用 char 表示字符、用 int 表示整数值、用 double 表示浮点数。其他基本类型可以看做上述类型的变形，只有当程序在性能优化、兼容性或其他方面有所要求时才会用到它们；一般情况下，前面四种类型已经足够了。

6.2.2 布尔值

一个布尔变量（bool）的取值或者是 true 或者是 false，布尔变量常用于表示逻辑运算的结果。例如：

```
void f(int a, int b)
{
    bool b1 {a==b};
    // ...
}
```

如果 a 和 b 的值相等，则 b1 变成 true；否则 b1 的值是 false。

有的函数被用来检验某个条件（谓词）是否成立，我们常常将这类函数的返回值类型设为 bool。例如：

```
bool is_open(File*);

bool greater(int a, int b) { return a>b; }
```

根据定义，当我们把布尔值转换成整数时，true 转为 1 而 false 转为 0。反之，整数值也能在需要的时候隐式地转换成布尔值，其中非 0 整数值对应 true 而 0 对应 false。例如：

```
bool b1 = 7;    // 因为 7!=0，所以 b 被赋值为 true
bool b2 {7};    // 错误：发生了窄化转换（见 2.2.2 节和 10.5 节）

int i1 = true;  // i1 被赋值为 1
int i2 {true};  // i2 被赋值为 1
```

如果你既想使用 {}- 初始化器列表防止窄化转换的发生，同时又确实想把 int 转换成 bool，则可以显式声明如下：

```
void f(int i)
{
    bool b {i!=0};
    // ...
};
```

在算术逻辑表达式和位逻辑表达式中，bool 被自动转换成 int，编译器在转换后的值上执行整数算术运算以及逻辑运算。如果最终的计算结果需要转换回 bool，则与之前介绍的一样，0 转换成 false 而非 0 值转换成 true。例如：

```
bool a = true;
bool b = true;

bool x = a+b;  // a+b 的结果是 2，因此 x 的最终取值是 true
bool y = a||b; // a||b 的结果是 1，因此 y 的值是 true ("||" 的含义是 " 或 ")
bool z = a-b;  // a-b 的结果是 0，因此 z 的最终取值是 false
```

如有必要，指针也能被隐式地转换成 bool（见 10.5.2.5 节）。其中，非空指针对应 true，值为 nullptr 的指针对应 false。例如：

```
void g(int* p)
{
    bool b = p;                 // 窄化成 true 或 false
    bool b2 {p!=nullptr};       // 显式地检查指针是否为非空

    if (p) {    // 等价于 p!=nullptr
        // ...
    }
}
```

与 if (p!=nullptr) 相比，我觉得 if(p) 更好，它不但简洁而且可以直接表达 " p 是否有效" 的含义，使用 if(p) 也不太容易出错。

6.2.3　字符类型

常用的字符集和字符集编码方式有很多。为了反映和描述这种多样性，C++ 提供了一系列字符类型：

- char：默认的字符类型，用于程序文本。char 是 C++ 实现所用的字符集，通常占 8 位。
- signed char：与 char 类似，但是带有符号；换句话说，它既可以存放正值也可以存放负值。
- unsigned char：与 char 类似，但是不带符号。
- wchar_t：用于存放 Unicode（见 7.3.2.2 节）等更大的字符集。wchar_t 的尺寸依赖于实现，确保能够支持实现环境中用到的最大字符集（第 39 章）。
- char16_t：该类型存放 UTF-16 等 16 位字符集。
- char32_t：该类型存放 UTF-32 等 32 位字符集。

这是 6 种不同的字符类型（除了后缀 _t 常用于指代别名之外，见 6.5 节）。在具体的实现版本中，char 类型可能会和 signed char 或者 unsigned char 完全等效。但不管怎么样，我们还是把这 3 个名字看成完全独立的 3 种类型。

一个 char 类型的变量存放一个字符，字符的种类由实现版本所用的字符集决定。例如：

```
char ch = 'a';
```

绝大多数情况下 char 占 8 个二进制位，因此可以保存 256 个不同的值。一般来说，该字符集是 ISO-646 的某个变种，比如 ASCII，你所用的键盘上的字符应该都会包含在内。由于该字符集只实现了部分标准化，所以有时候会带来一些问题。

字符集一些可能的变化必须引起我们的重视，比如支持不同自然语言的字符集，以及虽然支持的自然语言是同一种，但实现方式不同的字符集。我们最关心的是这些区别是否会影响 C++ 的语法规则。如何在多语言、多字符集的环境中编程是一个更大也更有趣的命题，我们对它稍有涉及（见 6.2.3 节，36.2.1 节和第 39 章）；但基本上这个问题已经远远超出了本书讨论的范围。

我们不妨认为所有字符集都包含十进制数字、英语的 26 个字母以及一些最基本的标点符号。但是下面这些对字符集的假设不一定成立，有可能带来一些问题：

- 8 位字符集中的字符总数不超过 127 个（某些字符集提供了 255 个字符）。
- 字符集中只包含英文字母，没有其他字母（大多数欧洲大陆的语言都含有更多字母，如 æ、þ 和 ß 等）。
- 字母都是紧密相连的（EBCDIC 在字母 'i' 和 'j' 之间留有空位）。
- 编写 C++ 代码所需的字符都是可用的（某些国家的字符集中不含 {、}、[、]、| 和 \）。
- char 占用一个字节。某些嵌入式处理器没有按字节访问内存的硬件，因此 char 占 4 个字节。另外，程序员也可以用 16 位的 Unicode 字符集编码基本 char 类型。

读者最好不要对对象的表示形式做出任何主观假设，对于字符尤其如此。

在编程环境所用的字符集中，每个字符都对应一个整数值。例如，在 ASCII 字符集中字符 'b' 的值是 98。下面这个小程序的功能是，你可以查看任意字符对应的整数值：

```
void intval()
{
```

```
    for (char c; cin >> c; )
        cout << "the value of '" << c << "' is " << int(c) << '\n';
}
```

我们用符号 int(c) 得到字符 c 对应的整数值（“用 c 构建的 int”）。既然 char 能转换成 int，那么随之而来的问题是：char 是有符号的还是无符号的？一个 8 位的字节所能容纳的 256 个值既可以被看成 0~255，也能被看成是 –127~127。注意：不是像你想象的 –128~127。因为 C++ 标准支持使用补码的硬件设备，而补码会排除掉一个值，所以如果我们使用 –128 的话代码就不容易移植了。不幸的是，char 到底是带符号的还是无符号的是个依赖于实现的问题。为了解决这一问题，C++ 提供了两种含义更明确的字符类型：signed char 存放 –127~127 之间的值；unsigned char 存放 0~255 之间的值。幸运的是，问题只出现在 0~127 之外的值，而绝大多数常用的字符事实上不会受到干扰。

把超过上述范围的值存入一个普通的 char 会带来一定移植性方面的问题。如果你需要使用几种不同的 char 或者你需要把整数值存在 char 变量中，请参阅 6.2.3.1 节。

请注意，字符类型属于整型（见 6.2.1 节）。因此，我们可以在字符类型上执行算术运算和位逻辑运算（见 10.3 节）。例如：

```
void digits()
{
    for (int i=0; i!=10; ++i)
        cout << static_cast<char>('0'+i);
}
```

上面的代码把 10 个阿拉伯数字输出到 cout。字符字面值常量 '0' 先转换成它对应的整数值，再与 i 相加；所得的 int 再转回 char 并被输出到 cout。'0'+i 得到的结果本来是一个 int，因此如果不加上 static_cast<char> 的话，输出的结果将会是 48, 49……，而不是 0, 1……

6.2.3.1　带符号字符和无符号字符

char 类型到底带不带符号是依赖于实现的，这可能会带来一些意料之外的糟糕结果。例如：

```
char c = 255;  // 255 的二进制表示是“全 1 形式”，对应的十六进制是 0xFF
int i = c;
```

i 的值是几？不幸的是，答案是未定义的。如果在运行环境中一个字节占 8 位，则答案依赖于 char 的“全 1 形式”在转换为 int 时是何含义。若机器的 char 是无符号的，则答案是 255；反之，若机器的 char 是带符号的，则答案是 –1。在此例中，因为字面值 255 有可能会转换成 char 值 –1，所以编译器可能会发出警告。但是 C++ 并没有某种通用的机制来检测这种问题。一个可能的解决方案是放弃使用普通 char 而只使用特定的 char 类型。不过像 strcmp() 这样的标准库函数通常只接受普通的 char（见 43.4 节）。

虽然从本质上来说，char 的行为无非与 signed char 一致或者与 unsigned char 一致，但这 3 个名字代表的类型的确各不相同。我们不能混用指向这 3 种字符类型的指针，例如：

```
void f(char c, signed char sc, unsigned char uc)
{
    char* pc = &uc;              // 错误：不存在对应的指针转换规则
    signed char* psc = pc;       // 错误：不存在对应的指针转换规则
    unsigned char* puc = pc;     // 错误：不存在对应的指针转换规则
    psc = puc;                   // 错误：不存在对应的指针转换规则
}
```

3 种 char 类型的变量可以相互赋值，但是把一个特别大的值赋给带符号的 char（见 10.5.2.1 节）是未定义的行为。例如：

```
void g(char c, signed char sc, unsigned char uc)
{
    c = 255;    // 如果普通的 char 是带符号的且占 8 位，则该语句的行为依赖于具体实现
    c = sc;     // OK
    c = uc;     // 如果普通的 char 是带符号的且 uc 的值特别大，则该语句的行为依赖于具体实现
    sc = uc;    // 如果 uc 的值特别大，则该语句的行为依赖于具体实现
    uc = sc;    // OK：转换成无符号类型
    sc = c;     // 如果普通的 char 是无符号的且 uc 的值特别大，则该语句的行为依赖于具体实现
    uc = c;     // OK：转换成无符号类型
}
```

再举个例子，假设 char 占 8 位：

```
signed char sc = -160;
unsigned char uc = sc;    // uc == 116（因为 256-140==116）
cout << uc;               // 输出 't'

char count[256];          // 假设是占 8 位的 char（未初始化的）
++count[sc];              // 严重错误：越界访问
++count[uc];              // OK
```

如果你从始至终都使用普通的 char 并且尽量避免负值，则上面这些潜在的错误不太可能发生。

6.2.3.2　字符字面值常量

字符字面值常量（character literal）是指单引号内的一个字符，如 'a' 和 '0' 等。字符字面值常量的数据类型是 char，它可以隐式地转换成当前机器所用字符集中对应的整数值。例如，如果你的机器使用的是 ASCII 字符集，则 '0' 的值是 48。建议程序员尽量使用字符字面值常量，而不要直接使用对应的十进制数值，显然前者的可移植性更强。

一些字符有一个以反斜线 \ 开头的标准名字，我们称之为转义字符：

字符名字	ASCII 名字	C++ 名字
换行	NL(LF)	\n
横向制表	HT	\t
纵向制表	VT	\v
退格	BS	\b
回车	CR	\r
换页	FF	\f
警告	BEL	\a
反斜线	\	\\
问号	?	\?
单引号	'	\'
双引号	"	\"
八进制数	*ooo*	*ooo*
十六进制数	*hhh*	\x*hhh* ...

不要被它们的外表迷惑，它们都是货真价实的单字符。

我们可以把字符集中的字符表示成一个 1~3 位的八进制数（\ 后紧跟八进制数字）或者

表示成十六进制数（\x 后紧跟十六进制数字）。其中，序列里十六进制数字的数量没有限制。如果遇到了第一个不是八进制数字或十六进制数字的字符，则表明当前的八进制序列或十六进制序列已经结束。例如：

八进制	十六进制	十进制	ASCII
'\6'	'\x6'	6	ACK
'\60'	'\x30'	48	'0'
'\137'	'\x05f'	95	'_'

上述规则使得我们不但可以设法表示出字符集中的每一个字符，而且能把这些字符嵌入到一个长字符串中（见 7.3.2 节）。但是，一旦我们在程序中使用了字符对应的数字形式，这样的程序就无法在使用不同字符集的机器间移植了。

有时候程序员会把多个字符放在一对单引号内，比如 'ab'。这种用法已经过时，它的效果完全依赖于实现，我们最好避免这种用法。多字符字面值常量的数据类型是 int。

当我们在字符串中嵌入八进制数字常量时，常规的做法是使用 3 个数字。这样的用法稳定且易于解读，我们不必担心常量之后的字符到底是不是数字。对于十六进制数字常量来说，我们使用两个数字。考虑如下的示例：

```
char v1[] = "a\xah\129";      // 6 个字符：'a' '\xa' 'h' '\12' '9' '\0'
char v2[] = "a\xah\127";      // 5 个字符：'a' '\xa' 'h' '\127' '\0'
char v3[] = "a\xad\127";      // 4 个字符：'a' '\xad' '\127' '\0'
char v4[] = "a\xad\0127";     // 5 个字符：'a' '\xad' '\012' '7' '\0'
```

宽字符字面值常量形如 L'ab'，它的数据类型是 wchar_t。单引号内字符的数量及其含义依赖于具体实现。

C++ 程序可以操作 Unicode 等其他字符集，这些字符集的规模远不止 ASCII 的 127 个字符这么多。大字符集中的字面值常量通常表示成 4 个或 8 个十六进制数字，其前缀是 u 或者 U。例如：

```
U'\UFADEBEEF'
u'\uDEAD'
u'\xDEAD'
```

对于任意的十六进制数字 X 而言，较短的表示形式 u'\uXXXX' 与较长的表示形式 U'\U0000XXXX' 是等价的。但是长度值不能是 4 和 8 之外的其他数字，否则会造成词法错误。ISO/IEC 10646 标准定义了上述十六进制数值的含义，这样的值称为通用字符名字（universal character name）。在 C++ 标准中，§ iso.2.2、§ iso.2.3、§ iso.2.14.3、§ iso.2.14.5 和 § iso.E 等处解释了通用字符名字。

6.2.4　整数类型

与 char 类似，整数类型也包含“普通的” int、signed int 和 unsigned int。整数还可以划分成另外 4 种形式：short int、“普通的” int、long int 和 long long int。其中，long int 即 long，而 long long int 即 long long。类似地，short 是 short int 的同义词，unsigned 是 unsigned int 的同义词，而 signed 是 signed int 的同义词。读者要注意，千万不能想当然地揣测 long short int 等价于 int，压根儿没有 long short int 这种类型。

当我们把存储空间看成是二进制位的数组时，可以考虑使用 unsigned 整数类型。但是

有的程序员试图用 unsigned 取代 int 来表示正整数，并且宣称这样做可以多利用 1 位，这样的想法有点不切实际。尽管他们试图通过声明 unsigned 类型的变量来确保某些值为正，但是这种担保并不可靠，因为程序中随处都是隐式类型转换（见 10.5.1 节和 10.5.2.1 节），一旦发生了隐式类型转换，谁也不知道后果会怎样。

一个不加修饰的 int 通常是带符号的，这一点与 char 不太一样。带符号的 int 和普通 int 不是两种类型，前者更像是后者的等价词，只不过语义上更明确一些。

如果你需要更精细地控制整数的尺寸，可以使用 <cstdint> 中定义的别名（见 43.7 节）。这些别名包括 int64_t（明确规定占用 64 位的带符号整数）、uint_fast16_t（至少占用 16 位的无符号整数，一般被认为是最快的整数）和 int_least32_t（至少占用 32 位的带符号整数，类似于 int）等。事实上，常规的几种整数类型都对最小尺寸做了很好的定义（见 6.2.8 节），因此 <cstdint> 显得有点儿多余，建议程序员谨慎使用。

在标准整数类型之外，一个具体的实现还可能提供某些扩展整数类型（extended integer type，带符号的以及无符号的）。这些新类型的行为必须与标准整数类似并且可以参与类型转换，也对应整数字面值常量。唯一的区别是它们的表示范围更大（占用更多空间）。

6.2.4.1 整数字面值常量

整数字面值常量分为 3 种：十进制、八进制和十六进制。其中十进制字面值常量最常见，也最符合用户的使用习惯：

```
7   1234   976   123456789012345678901
```

当字面值常量表示的数值太大以至于在 C++ 中无法表达时，编译器会发出警告；但是只有使用 {} 初始化器的形式才会报错（见 6.3.5 节）。

以 x 或 X（0x 或 0X）开头的字面值常量表示一个十六进制数值（基是 16）；以 0 开头但是后面没有 x 或 X 的字面值常量表示一个八进制数值（基是 8）。例如：

十进制	八进制	十六进制
	0	0x0
2	02	0x2
63	077	0x3f
83	0123	0x63

在十六进制中，a、b、c、d、e、f 及其大写形式分别表示 10、11、12、13、14 和 15。用八进制和十六进制表示二进制比较有效，但如果用它们表示纯数字，可能会产生意想不到的后果。举个例子，如果某台机器用 16 位补码表示 int，则 0xffff 对应十进制数 –1；但如果表示 int 的二进制位不只 16 位，则 0xffff 对应的值就可能变成 65535 了。

后缀 U 用于显式指定 unsigned 字面值常量，与之类似，后缀 L 用于显式指定 long 字面值常量。例如，3 是一个 int，3U 的类型是 unsigned int 而 3L 的类型是 long int。

多个后缀可以组合在一起使用，例如：

```
cout << 0xF0UL << ' ' << 0LU << '\n';
```

如果没有显式地指定后缀，编译器会根据整数字面值常量的值以及当前实现的整数尺寸为它分配一种合适的类型（见 6.2.4.2 节）。

不要滥用含义不明显的常量，最好只在给 const（见 7.5 节）、constexpr（见 10.4 节）

和枚举（见 8.4 节）赋初值时使用。

6.2.4.2　整数字面值常量的类型

通常情况下，整数字面值常量的类型由它的形式、取值和后缀共同决定：

- 如果它是十进制数且没有后缀，则它的类型是下面几种类型中能够表达它的值且尺寸最小的那个：int，long int，long long int。
- 如果它是八进制数或十六进制数且没有后缀，则它的类型是下面几种类型中能够表达它的值且尺寸最小的那个：int，unsigned int，long int，unsigned long int，long long int，unsigned long long int。
- 如果它的后缀是 u 或 U，则它的类型是下面几种类型中能够表达它的值且尺寸最小的那个：unsigned int，unsigned long int，unsigned long long int。
- 如果它是十进制数且后缀是 l 或 L，则它的类型是下面几种类型中能够表达它的值且尺寸最小的那个：long int，long long int。
- 如果它是八进制数或十六进制数且后缀是 l 或 L，则它的类型是下面几种类型中能够表达它的值且尺寸最小的那个：long int，unsigned long int，long long int，unsigned long long int。
- 如果它的后缀是 ul，lu，uL，Lu，Ul，lU，UL 或 LU，则它的类型是下面几种类型中能够表达它的值且尺寸最小的那个：unsigned long int，unsigned long long int。
- 如果它是十进制数且后缀是 ll 或 LL，则它的类型是 long long int。
- 如果它是八进制数或十六进制数且后缀是 ll 或 LL，则它的类型是下面几种类型中能够表达它的值且尺寸最小的那个：long long int，unsigned long long int。
- 如果它的后缀是 llu，llU，ull，Ull，LLu，LLU，uLL 或 ULL，则它的类型是 unsigned long long int。

例如，对于字面值常量 100000 来说，在 32 位 int 的机器上它的类型是 int，而在 16 位 int 和 32 位 long 的机器上它的类型是 long int。类似地，0XA000 在 32 位 int 的机器上类型是 int 而在 16 位 int 的机器上的类型是 unsigned int。我们可以使用后缀来规避上述对于实现的依赖性：在任何机器上 100000L 的类型都是 long int，0XA000U 的类型都是 unsigned int。

6.2.5　浮点数类型

浮点数类型用于表示浮点数。浮点数是实数在有限内存空间上的一种近似表示。有 3 种浮点数类型：float（单精度）、double（双精度）和 long double（扩展精度）。

所谓单精度、双精度和扩展精度的确切含义是依赖于具体实现的。程序员只有对浮点运算有非常深刻的理解才能在解决实际问题时做出最好的选择。如果你做不到这一点，最好向有经验的程序员寻求建议或者自学。实在不行就优先选择 double 类型，这是一种折中的选择，比较稳妥。

6.2.5.1　浮点数字面值常量

默认情况下，浮点数字面值常量的类型是 double。再说一次，编译器应该会在发现数据类型不足以表示给定值的时候发出警告。下面是一些浮点数字面值常量的示例：

```
1.23  .23  0.23  1.  1.0  1.2e10  1.23e-15
```

谨记在浮点数字面值常量内部不允许出现空格。例如，65.43 e-21 不是一个浮点数字面值常

量，它更像是4个独立的词汇单元（并且会导致语法错误）：

```
65.43 e - 21
```

如果你希望定义一个 float 类型的浮点数字面值常量，则必须加上后缀 f 或 F：

```
3.14159265f 2.0f 2.997925F 2.9e-3f
```

类似地，如果你希望定义一个 long double 类型的浮点数字面值常量，加上后缀 l 或 L：

```
3.14159265L 2.0L 2.997925L 2.9e-3L
```

6.2.6 前缀和后缀

有一些前缀和后缀常被用来限定字面值常量的类型：

算术字面值常量的前缀和后缀

符号		前/中/后缀	含义	示例	参见	ISO
0		前缀	八进制	0776	6.2.4.1 节	§ iso.2.14.2
0x	0X	前缀	十六进制	0xff	6.2.4.1 节	§ iso.2.14.2
u	U	后缀	unsigned	10U	6.2.4.1 节	§ iso.2.14.2
l	L	后缀	long	20000L	6.2.4.1 节	§ iso.2.14.2
ll	LL	后缀	long long	20000LL	6.2.4.1 节	§ iso.2.14.2
f	F	后缀	float	10f	6.2.5.1 节	§ iso.2.14.4
e	E	中缀	浮点数	10e-4	6.2.5.1 节	§ iso.2.14.4
.		中缀	浮点数	12.3	6.2.5.1 节	§ iso.2.14.4
'		前缀	char	'c'	6.2.3.2 节	§ iso.2.14.3
u'		前缀	char16_t	u'c'	6.2.3.2 节	§ iso.2.14.3
U'		前缀	char32_t	U'c'	6.2.3.2 节	§ iso.2.14.3
L'		前缀	wchar_t	L'c'	6.2.3.2 节	§ iso.2.14.3
"		前缀	字符串	"mess"	7.3.2 节	§ iso.2.14.5
R"		前缀	原始字符串	R"(\b)"	7.3.2.1 节	§ iso.2.14.5
u8"	u8R"	前缀	UTF-8 字符串	u8"foo"	7.3.2.2 节	§ iso.2.14.5
u"	uR"	前缀	UTF-16 字符串	u"foo"	7.3.2.2 节	§ iso.2.14.5
U"	UR"	前缀	UTF-32 字符串	U"foo"	7.3.2.2 节	§ iso.2.14.5
L"	LR"	前缀	wchar_t 字符串	L"foo"	7.3.2.2 节	§ iso.2.14.5

请注意，上表中的“字符串”是指“字符串字面值常量”（见 7.3.2 节），而非数据类型“std::string”。

我们当然可以把 . 和 e 看成是中缀，同时把 R" 和 u8" 看成分隔符的一部分，不过怎么命名并不重要。通过上表，我们的最终目标是把字面值常量的各种情况总结在一起，供读者了解和学习。

后缀 l 和 L 可以与 u 和 U 结合在一起使用，表达的数据类型是 unsigned long。例如：

```
1LU     // unsigned long
2UL     // unsigned long
3ULL    // unsigned long long
4LLU    // unsigned long long
5LUL    // 错误
```

同样，后缀 l 和 L 也能用于表示浮点数字面值常量，表达的类型是 long double。例如：

```
1L          // long int
1.0L        // long double
```

几种前缀 R、L 和 u 能结合在一起，如 uR"**(foo\(bar))**"。读者一定要对符号 U 的两种用法加以区分：一种是字符的前缀 U，表示 unsigned，另一种是字符串的前缀 U，表示 UTF-32 编码（见 7.3.2.2 节）。

此外，用户可以为自定义类型定义新的后缀。例如，我们可以定义一个新的字面值常量运算符（见 19.2.6 节）：

```
"foo bar"s      // 是一个 std::string 类型的字面值常量
123_km          // 是一个 Distance 类型的字面值常量
```

不以 _ 开始的后缀仅存在于标准库中。

6.2.7　void

从语法结构上来说，void 属于基本类型。但是它只能被用作其他复杂类型的一部分，不存在任何 void 类型的对象。void 有两个作用：一是作为函数的返回类型用以说明函数不返回任何实际的值；二是作为指针的基本类型部分以表明指针所指对象的类型未知。例如：

```
void x;         // 错误：不存在 void 类型的对象
void& r;        // 错误：不存在 void 的引用
void f();       // 函数 f 不返回任何实际的值（见 12.1.4 节）
void* pv;       // 指针所指的对象类型未知（见 7.2.1 节）
```

当我们声明一个函数时，必须指明返回结果的数据类型。从逻辑上来说，如果某个函数不返回任何值，也许我们会希望直接忽略掉返回值部分。但其实这种想法并不可行，它会违反 C++ 的语法规则（§ iso.A）。因此，我们使用 void 表示函数的返回值为空，此时 void 可以看成是一种“伪返回类型”。

6.2.8　类型尺寸

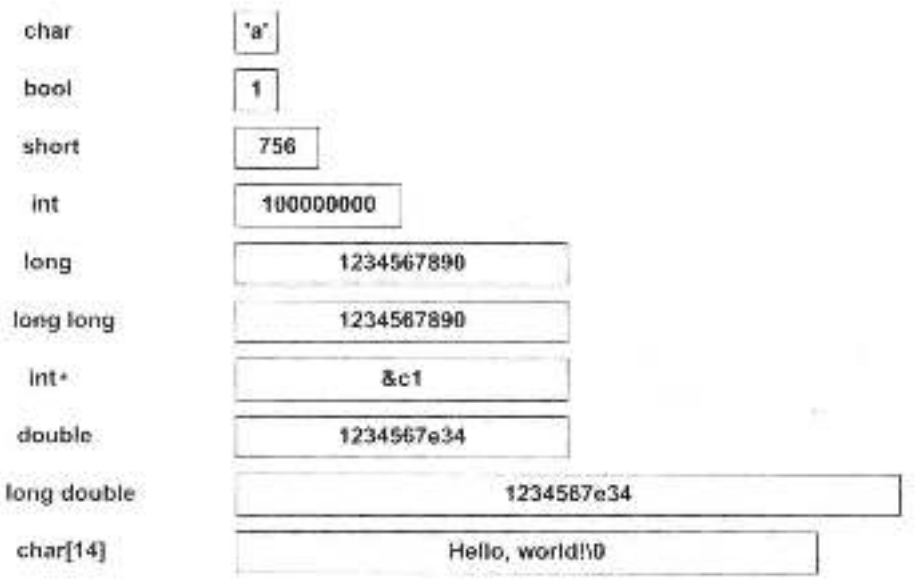
C++ 基本类型的某些方面是依赖于实现的（见 6.1 节），其中一个例子是 int 类型的尺寸。我曾经不止一次指出过这种依赖性的存在，并且建议程序员应该尽量避免依赖性带来的问题或者设法减少它对程序结果的影响。为什么呢？通常，如果程序员在几种不同的系统中编程或者使用不同的编译器编程，则他们必须特别关注依赖性的问题，否则的话，他们有可能得花费大量时间来定位并修改许多隐藏较深、不易察觉的程序错误。有的人宣称他们不太介意可移植性的问题，原因是这些人基本上只在一种系统上编程，而且武断地认为“当前编译器实现的就是真正的 C++ 语言，没有其他了”。这显然是非常狭隘和短视的观点。如果你的程序真的有用，它不可避免地会被移植到其他系统中，此时别的程序员就不得不费时费力地去寻找和改正程序中受实现依赖影响的部分。此外，在一个大系统中，某部分程序常常需要用其他编译器编译；而且即使是你一直使用的编译器，它的不同版本之间也可能会有差异。显然，在编写程序的时候就对实现依赖性的问题给予足够重视并设法减少其负面影响要比事后弥补容易得多。

相对来说，限制依赖于实现的语言特性的影响比较容易；而要想限制依赖于系统的标准

库功能就难多了。一种可行的措施是尽量使用那些比较通用的标准库功能。

我们之所以为整数类型、无符号类型和浮点数类型都分别设计了几种不同形式，目的是允许程序员从中选择最恰当的一种以充分利用硬件的特性。在许多机器上，不同的基本类型之间差异很大，这些差异体现在内存需求、内存访问时间和计算时间等方面。如果你深入了解某一机器，则做出抉择的过程会比较容易，比如很容易为某一变量选择一种适用于当前机器的整数类型。显然，要想编写真正具有可移植性的程序非常困难。

下图展示了一组基本类型的集合以及一个字符串字面值常量（见7.3.2节）：



如果以上图的比例来计算（0.2英寸[1]对应1个字节），则1M内存大概要向右延伸3英里（5千米）。

所有C++对象的尺寸都可以表示成char尺寸的整数倍，因此如果我们令char的尺寸为1，则使用sizeof运算符（见10.3节）就能得到任意类型或对象的尺寸。下面是C++对于基本类型尺寸的一些规定：

- 1 = sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)
- 1 ≤ sizeof(bool) ≤ sizeof(long)
- sizeof(char) ≤ sizeof(wchar_t) ≤ sizeof(long)
- sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)
- sizeof(N) = sizeof(signed N) = sizeof(unsigned N)

其中，最后一行的N可以是char、short、int、long或者long long。C++规定char至少占8位，short至少占16位，long至少占32位。char应该能存放机器字符集中的任意字符，它的实际类型依赖于实现并确保是当前机器上最适合保存和操作字符的类型。通常情况下，char占据一个8位的字节。与之类似，int的实际类型也是依赖于实现的，并确保是当前机器上最适合保存和操作整数的类型。int通常占据一个4字节（32位）的字。上面这些假设是比较恰当的，但是我们很难做更多设定。比如，我们只能说char“通常”占据8位，因为确实也存在char占32位的机器。又比如我们决不能假定int和指针的尺寸一样大，因为在很多机器上（“64位体系结构”）指针的尺寸比整数大。最后请注意，下面两条假设并

[1] 1英寸=0.0254m。——编辑注

不成立：sizeof(long)<sizeof(long long) 和 sizeof(double)<sizeof(long double)。

通过使用 sizeof 函数，我们能发现基本类型的某些依赖于实现的特性，更多这样的特性包含在 <limits> 中。例如：

```
#include <limits>    // 见 40.2 节
#include <iostream>

int main()
{
    cout << "size of long " << sizeof(1L) << '\n';
    cout << "size of long long " << sizeof(1LL) << '\n';

    cout << "largest float == " << std::numeric_limits<float>::max() << '\n';
    cout << "char is signed == " << std::numeric_limits<char>::is_signed << '\n';
}
```

因为 <limits> 中定义的函数（见 40.2 节）是 constexpr（见 10.4 节），所以可以用在需要常量表达式的上下文中，并且不会带来额外的运行时开销。

在赋值语句及表达式中可以自由地使用基本类型，编译器随时随地计算并转换变量的值以尽量做到不损失信息（见 10.5 节）。

如果某个值 v 能用 T 类型的变量确切地表达，则把 v 的类型转换成 T 是值保护的（value-preseving）。我们最好避免使用那些做不到值保护的类型转换（见 2.2.2 节和 10.5.2.6 节）。

如果你需要使用某种特定尺寸的整数类型（比如 16 位的整数），应该事先 #include 标准库头文件 <cstdint>。在 <cstdint> 中定义了很多类型（或类型别名，见 6.5 节），例如：

```
int16_t x {0xaabb};                    // 2 字节
int64_t xxxx {0xaaaabbbbccccdddd};     // 8 字节
int_least16_t y;                       // 至少 2 字节（与 int 类似）
int_least32_t yy                       // 至少 4 字节（与 long 类似）
int_fast32_t z;                        // 是最快的整数类型，至少包含 4 个字节
```

在标准库头文件 <cstddef> 中定义了一个别名，它被广泛用于标准库声明及用户代码：size_t 是一个依赖于实现的无符号整数类型，用于表示任意对象所占的字节数。我们可以在需要保存对象尺寸的时候使用 size_t，例如：

```
void* allocate(size_t n);  // 获得 n 个字节
```

类似地，<cstddef> 还定义了一个带符号的整数类型 ptrdiff_t，两个指针相减所得的元素数量可以保存在 ptrdiff_t 中。

6.2.9　对齐

对象首先应该有足够的空间存放对应的变量，但这还不够。在一些机器的体系结构中，存放变量的字节必须保持一种良好的对齐（alignment）方式，以便硬件在访问数据资源时足够高效（在极端情况下一次性访问所有数据）。例如，4 字节的 int 应该按字（4 字节）的边界排列，而 8 字节的 double 有时也应该按字（8 字节）的边界排列。当然这些约定都是依赖于实现且用户不可见的，你也许写了几十年漂亮的 C++ 代码却从来没有为对齐问题担心过。对齐只有在涉及对象布局的问题中比较明显：有时候我们会让 struct 包含一些“空洞”以提升整齐程度（见 8.2.1 节）。

alignof() 运算符返回实参表达式的对齐情况，例如：

```
auto ac = alignof('c');      // char 的对齐情况
auto ai = alignof(1);        // int 的对齐情况
auto ad = alignof(2.0);      // double 的对齐情况

int a[20];
auto aa = alignof(a);        // int 的对齐情况
```

有时我们需要在声明语句中使用对齐，但是不允许形如 alignof(x+y) 的表达式；此时，我们可以使用类型说明符 alignas: alignas(T)，它的含义是“像 T 那样对齐”。例如，我们用下面的语句为一些类型 X 的变量留出未初始化的存储空间：

```
void user(const vector<X>& vx)
{
    constexpr int bufmax = 1024;
    alignas(X) buffer[bufmax];      // 未初始化的

    const int max = min(vx.size(),bufmax/sizeof(X));
    uninitialized_copy(vx.begin(),vx.begin()+max,buffer);
    // ...
}
```

6.3 声明

在 C++ 程序中要想使用某个名字（标识符），必须先对其进行声明。换句话说，我们必须指定它的类型以便编译器知道这个名字对应的是何种实体。例如：

```
char ch;
string s;
auto count = 1;
const double pi {3.1415926535897};
extern int error_number;

const char* name = "Njal";
const char* season[] = { "spring", "summer", "fall", "winter" };
vector<string> people { name, "Skarphedin", "Gunnar" };
struct Date { int d, m, y; };
int day(Date* p) { return p->d; }
double sqrt(double);
template<class T> T abs(T a) { return a<0 ? -a : a; }

constexpr int fac(int n) { return (n<2)?1:n*fac(n-1); }   // 可能的编译时求值（见 2.2.3 节）
constexpr double zz { ii=fac(7) };                        // 编译时初始化

using Cmplx = std::complex<double>;                       // 类型别名（见 3.4.5 节和 6.5 节）
struct User;                                              // 类型名字
enum class Beer { Carlsberg, Tuborg, Thor };
namespace NS { int a; }
```

从上面这些例子可以看出，声明语句的作用不止把类型和名字关联起来这么简单。大多数声明（declaration）同时也是定义（definition）。我们可以把定义看成是一种特殊的声明，它提供了在程序中使用该实体所需的一切信息。尤其是当实体需要内存空间来存储某些信息时，定义语句把所需的内存预留了出来。还有一种观点认为声明是接口的一部分，而定义属于实现的范畴。在这种视角下，我们尽量用声明语句组成程序的接口。其中，同一个声明可以在不同文件中重复出现（见 15.2.2 节）。负责申请内存空间的定义语句不属于接口。

假定这些声明语句位于全局作用域中（见 6.3.4 节），则：

```
char ch;                                    // 为一个 char 类型的变量分配内存空间并赋初值 0
auto count = 1;                             // 为一个 int 类型的变量分配内存空间并赋初值 1
const char* name = "Njal";                  // 为一个指向 char 的指针分配内存空间
                                            // 为字符串字面值常量 "Njal" 分配内存空间
                                            // 用字符串字面值常量的地址初始化指针

struct Date { int d, m, y; };               // Date 是一个 struct，它包含 3 个成员
int day(Date* p) { return p->d; }           // day 是一个函数，它执行某些既定的代码

using Point = std::complex<short>;// Point 是类型 std::complex<short> 的别名
```

在上面这些声明语句中，只有 3 个不是定义：

```
double sqrt(double);                // 函数声明
extern int error_number;            // 变量声明
struct User;                        // 类型名字声明
```

也就是说，要想使用它们对应的实体，必须先在其他某处进行定义。例如：

```
double sqrt(double d) { /* ... */ }
int error_number = 1;
struct User { /* ... */ };
```

在 C++ 程序中每个名字可以对应多个声明语句，但是只能有一个定义（关于 #include 的影响见 15.2.3 节）。

在同一实体的所有声明中，实体的类型必须保持一致。因此，下面的小片段包含两处错误：

```
int count;
int count;      // 错误：重定义
extern int error_number;
extern short error_number;          // 错误：类型不匹配
```

下面的语句则是正确的（关于 extern 的用法见 15.2 节）：

```
extern int error_number;
extern int error_number; // OK：多次声明
```

有的定义语句为它们定义的实体显式地赋“值”，例如：

```
struct Date { int d, m, y; };
using Point = std::complex<short>;      // Point 是类型 std::complex<short> 的别名
int day(Date* p) { return p->d; }
const double pi {3.1415926535897};
```

对于类型、别名、模板、函数和常量来说，这个“值”是不变的；而对于非 const 数据类型来说，初始值可能会在稍后被改变。例如：

```
void f()
{
    int count {1};                      // 把 count 初始化为 1
    const char* name {"Bjarne"}; // name 是个变量，它所指的对象是个常量（见 7.5 节）
    count = 2;                          // 把 2 赋给 count
    name = "Marian";
}
```

在这些定义语句中，只有两个没有指定值：

```
char ch;
string s;
```

关于变量何时以及以何种方式被赋予默认值请见6.3.5节和17.3.3节。对于任意的声明语句来说，只要它为变量指定了值，就是一条定义语句。

6.3.1 声明的结构

C++语法规定了声明语句的结构（§iso.A）。这套语法最早从C的语法演化而来，历经四十多年的发展，变得相当复杂。在不做什么根本性简化的前提下，我们可以认为一条声明语句（依次）包含5个部分：

- 可选的前置修饰符（比如static和virtual）
- 基本类型（比如vector<double>和const int）
- 可选的声明符，可包含一个名字（比如p[7]、n和*(*)[]）
- 可选的后缀函数修饰符（比如const和noexcept）
- 可选的初始化器或函数体（比如={7,5,3}和{return x;}）

除了函数和名字空间的定义外，其他声明语句都以分号结束。请考虑下面这个C风格字符串数组的定义语句：

```
const char* kings[] = { "Antigonus", "Seleucus", "Ptolemy" };
```

此例中，基本类型是const char，声明符是*kings[]，初始化器是=及其后的{}列表。

修饰符是指声明语句中最开始的关键字，如virtual（见3.2.3节和20.3.2节）、extern（见15.2节）和constexpr（见2.2.3节）等。修饰符的作用是指定所声明对象的某些非类型属性。

声明符由一个名字和一些可选的声明运算符组成。最常用的声明运算符包括：

声明运算符		
前缀	*	指针
前缀	*const	常量指针
前缀	*volatile	volatile指针
前缀	&	左值引用（见7.7.1节）
前缀	&&	右值引用（见7.7.2节）
前缀	auto	函数（使用后置返回类型）
后缀	[]	数组
后缀	()	函数
后缀	->	从函数返回

如果上面这些声明符都是前缀或者都是后缀的话，它们的用法就简单了。但实际上，*、[]和()的用法与在表达式中一致（见10.3节）。因此，*是前缀，而[]和()是后缀。后缀声明符的绑定效果比前缀声明符更紧密，所以char*kings[]是char指针的数组，而char(*kings)[]是指向char数组的指针。如果我们想声明“数组的指针”或者“函数的指针”，则必须使用括号加以限定，具体的例子请见7.2节。

请注意，在声明语句中不允许省略数据类型。例如：

```
const c = 7;    // 错误：缺少数据类型

gt(int a, int b)  // 错误：缺少数据类型
```

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

unsigned ui;    // OK："unsigned" 即 "unsigned int"
long li;        // OK："long" 即 "long int"
```

早期版本的 C 和 C++ 允许前两个例子所示的用法，它们认为当程序员没有指定数据类型时，int 是默认的类型（见 44.3 节）；但是标准 C++ 不允许这样做。所谓的“隐式 int”规则会让源程序充满不可捉摸的错误，并且显得杂乱无章。

有的类型名字包含多个关键字，比如 long long 和 volatile int；还有一些类型名字看起来不像个名字，比如 decltype(f(x))（表示函数 f(x) 的返回值类型，见 6.3.6.3 节）。

41.4 节介绍 volatile 修饰符。

6.2.9 节介绍 alignas() 修饰符。

6.3.2　声明多个名字

C++ 允许在同一条声明语句中声明多个名字，其中包含逗号隔开的多个声明符即可。例如，我们能以如下方式声明两个整数：

```
int x, y;       // int x; int y;
```

读者千万要注意，在声明语句中，运算符只作用于紧邻的一个名字，对于后续的其他名字是无效的。例如：

```
int* p, y;      // 准确的含义是 int* p; int y; 而非 int* y;
int x, *q;      // int x; int* q;
int v[10], *pv; // int v[10]; int* pv;
```

上面这些示例在同一条声明语句中包含了多个名字且加入了某些特殊的声明符，这会让程序看起来有点难懂，实际编程过程中最好避免这种用法。

6.3.3　名字

一个名字（标识符）包含若干字母和数字。第一个字符必须是字母，其中，我们把下划线 _ 也看成是字母。C++ 对于名字中所含的字符数量未作限定。但是在具体实现中，某些部分并不受编译器的控制（尤其是链接器），而这些部分有可能会限制名字中字符的多少。某些运行时环境要求扩展或缩减能出现在标识符中的字符集规模。此时，对可接受字符的扩充（比如允许名字中出现字符 $）会造成程序无法移植。C++ 关键字（比如 new 和 int，见 6.3.3.1 节）不能用作用户自定义实体的名字。一些有效的标识符如下所示：

```
hello       this_is_a_most_unusually_long_identifier_that_is_better_avoided
DEFINED     foO     bAr     u_name      HorseSense
var0        var1    CLASS   _class      ___
```

下面这些字符序列不能作为标识符：

```
012         a fool      $sys        class       3var
pay.due     foo~bar     .name       if
```

以下划线开头的非局部名字表示具体实现及运行时环境中的某些特殊功能，应用程序中不应

该使用这样的名字。类似地，包含双下划线（__）的名字和以下划线并头紧跟大写字母的名字（比如 _Foo）都有特殊用途（见 17.6.4.3 节）。

编译器在编译代码时总是优先寻找能构成名字的最长的字符串。因此，var10 整体是个名字，而不是名字 var 后跟着数字 10。同样，elseif 也是个名字，而非关键字 else 后跟着关键值 if。

标识符的命名区分大小写，因此 Count 和 count 是两个不同的名字。显然，仅靠字母的大小写来区分名字不太合适。一般情况下，我们应该尽量避免使用过于相似的名字。举个例子，在很多字体中，字母“o”的大写形式（O）与数字 0 很难区分，字母“L”的小写形式（l），字母“i”的大写形式（I）与数字 1 也很难区分。因此，l0、lO、l1、ll 和 lIl 显然是一组非常糟糕的名字。不是所有字体都有这个问题，但大多数确实如此。

在一个范围较大的作用域中，我们应该使用相对较长且有明确含义的名字，比如 vector、Window_with_border 和 Department_number。然而，在范围较小的作用域中使用一些长度较短但是约定俗成的名字也不失为一种好的选择，这些名字包括 x、i、p 等。函数（第 12 章）、类（第 16 章）和名字空间（见 14.3.1 节）可以帮助我们限定一个较小的作用域。通常，我们令那些频繁使用的名字相对较短，而让较长的名字对应一些很少用到的实体。

在为实体命名时，我们应该尽量让名字反映实体的含义而非其实现细节。例如，当我们用 vector 存储电话号码时（见 4.4 节），变量的名字用 phone_book 比用 number_book 更好。在使用某些具有动态类型系统或弱类型系统的语言编程时，程序员习惯在实体的名字中掺杂进类型信息（比如用 pcname 作为某个 char* 的名字，或者用 icount 作为某个 int 计数变量的名字），但是在 C++ 中我们不建议这样做：

- 把类型信息加到名字里降低了程序的抽象水平，尤其是不利于泛型编程（基本机理是其中的某个名字可以指向不同类型的实体）。
- 编译器比程序员更擅长记录和追踪类型信息。
- 一旦你想改变某个名字的类型（用 std::string 存放名字），就必须更改程序中所有用到该名字的地方（否则，已经嵌入名字的类型信息就名不副实了）。
- 随着你用到的类型越来越多，你设计的缩写集会越来越大，有时含糊不清，有时过于啰嗦。

简而言之，为标识符命名称得上是一门艺术。

程序员最好遵循某些约定俗成的命名风格，并且坚持下去，不要轻易改变。例如，用户自定义类型名的首字母大写，非类型实体名的首字母小写（比如 Shape 和 current_token）。又如在宏定义中全都使用大写字母（前提是当你不得不使用宏时，见 12.6 节，比如 HACK），在其他场合绝对不要这样做（即使非宏常量也不行）。用下划线把标识符中的单词隔开，number_of_elements 比 numberOfElements 的可读性更好。然而，保持命名风格的统一也不是一件容易的事，毕竟程序通常是由很多来源不同的片段组合而成的，它们遵循的风格可能各不相同，各有各的道理。谨记对缩写的使用应该保持一致。请注意，C++ 语言和标准库中的类型名字都是小写，有时候这条线索可以帮助我们判断某个类型名字是否来源于 C++ 标准。

6.3.3.1 关键字

C++ 的关键字如下表所示：

C++ 关键字					
alignas	alignof	and	and_eq	asm	auto
bitand	bitor	bool	break	case	catch
char	char16_t	char32_t	class	compl	const
constexpr	const_cast	continue	decltype	default	delete
do	double	dynamic_cast	else	enum	explicit
extern	false	float	for	friend	goto
if	inline	int	long	mutable	namespace
new	noexcept	not	not_eq	nullptr	operator
or	or_eq	private	protected	public	register
reinterpret_cast	return	short	signed	sizeof	static
static_asser t	static_cast	struct	switch	template	this
thread_local	throw	true	try	typedef	typeid
typename	union	unsigned	using	virtual	void
volatile	wchar_t	while	xor	xor_eq	

此外，export 被留在以后使用。

6.3.4　作用域

声明语句为作用域引入了一个新名字，换句话说，某个名字只能在程序文本的某个特定区域使用。

- *局部作用域*（local scope）：函数（第 12 章）或 lambda 表达式（见 11.4 节）中声明的名字称为*局部名字*（local name）。局部名字的作用域从声明处开始，到声明语句所在的块结束为止。其中块（block）是指用一对 {} 包围的代码片段。对于函数和 lambda 表达式最外层的块来说，参数名字是其中的局部名字。
- *类作用域*（class scope）：如果某个类位于任意函数、类（第 16 章）和枚举类（见 8.4.1 节）或其他名字空间的外部，则定义在该类中的名字称为*成员名字*（member name）或*类成员名字*（class member name）。类成员名字的作用域从类声明的 { 开始，到类声明的结束为止。
- *名字空间作用域*（namespace scope）：如果某个名字空间位于任意函数（第 12 章）、lambda 表达式（见 11.4 节）、类（第 16 章）和枚举类（见 8.4.1 节）或其他名字空间的外部，则定义在该名字空间中的名字为*名字空间成员名字*（namespace member name）。名字空间成员名字的作用域从声明语句开始，到名字空间结束为止。名字空间名字能被其他翻译单元访问（见 15.2 节）。
- *全局作用域*（global scope）：定义在任意函数、类（第 16 章）、枚举类（见 8.4.1 节）和名字空间（见 14.3.1 节）之外的名字称为*全局名字*（global name）。全局名字的作用域从声明处开始，到声明语句所在的文件末尾为止。全局名字能被其他翻译单元访问（见 15.2 节）。从技术上来说，全局名字空间也是一种名字空间。因此，我们可以把全局名字看成是一种特殊的名字空间成员名字。
- *语句作用域*（statement scope）：如果某个名字定义在 for、while、if 和 switch 语句的 () 部分，则该名字位于语句作用域中。它的作用域范围从声明处开始，到语句结束为止。语句作用域中的所有名字都是局部名字。

- 函数作用域（function scope）：标签（见9.6节）的作用域是从声明它开始到函数体结束。

在块内声明的名字能隐藏外层块及全局作用域中的同名声明。换句话说，一个已有的名字能在块内被重新定义以指向另外一个实体。退出块后，该名字恢复原来的含义。例如：

```
int x;                  // 全局变量 x

void f()
{
    int x;              // 局部变量 x 隐藏了全局变量 x
    x = 1;              // 为局部变量 x 赋值
    {
        int x;          // 隐藏了上一个局部变量 x
        x = 2;          // 为第二个局部变量 x 赋值
    }
    x = 3;              // 为第一个局部变量 x 赋值
}

int* p = &x;            // 获取全局变量 x 的地址
```

隐藏名字的现象在规模较大的程序中比较普遍，很难避免。然而，程序的读者经常会遗忘某个名字被隐藏（或者说遮住，shadowed）的事实。由名字隐藏造成的程序错误不太多，因此一旦出错极难发现。程序员应该尽量避免隐藏名字。如果你给全局变量或者大函数中的局部变量起类似于 i 或 x 的名字，无异于自找麻烦。

我们可以使用作用域解析运算符 :: 访问被隐藏了的全局名字，例如：

```
int x;

void f2()
{
    int x = 1; // 隐藏全局变量 x
    ::x = 2;   // 为全局变量 x 赋值
    x = 2;     // 为局部变量 x 赋值
    // ...
}
```

我们无法使用被隐藏的局部名字。

非类成员名字的作用域始于它的声明点，即完整的声明符之后且初始化器之前的位置。这一规定意味着我们甚至能用某个名字作为它自己的初始值。例如：

```
int x = 97;

void f3()
{
    int x = x;          // 不合理：赋给 x 的值是它自己的未初始化的值
}
```

一个严谨的编译器应该能在遇到变量未初始化即使用的现象时发出警告。

有一种现象看起来有点奇怪，但却是合理的，即在同一个块内有可能同一个名字所指的是两个完全不同的实体，并且我们没有使用 :: 运算符。例如：

```
int x = 11;

void f4()               // 不合理：在同一个作用域内使用了两个名字都是 x 的对象
{
    int y = x;          // 使用全局变量 x，结果是 y = 11
```

```
    int x = 22;
    y = x;          // 使用局部变量 x，结果是 y = 22
}
```

再次提醒，在你的程序中最好避免出现这种小问题。

我们通常认为函数的实参是声明在函数的最外层块中的，例如：

```
void f5(int x)
{
    int x;      // 错误
}
```

因为 x 在同一个作用域中定义了两次，所以上述程序存在错误。

在 for 语句中引入的名字是该语句的局部名字（位于语句作用域内）。因此，在同一个函数内，我们可以在好几个循环中使用同一个便于理解的名字。例如：

```
void f(vector<string>& v, list<int>& lst)
{
    for (const auto& x : v) cout << x << '\n';
    for (auto x : lst) cout << x << '\n';
    for (int i = 0, ii=v.size(); ++i) cout << v[i] << '\n';
    for (auto i : {1, 2, 3, 4, 5, 6, 7}) cout << i << '\n';
}
```

在这个函数中，不存在名字冲突。

如果在 if 语句的分支中有一条声明语句，并且它是该分支唯一的语句，则这种用法是不允许的（见 9.4.1 节）。

6.3.5　初始化

顾名思义，初始化器就是对象在初始状态下被赋予的值。初始化器有四种可能的形式：

```
X a1 {v};
X a2 = {v};
X a3 = v;
X a4(v);
```

在这些形式中，只有第一种不受任何限制，在所有场景中都能使用。我强烈建议程序员使用这种形式为变量赋初值，它含义清晰，与其他形式相比不太容易出错。不过，第一种初值形式 (a1) 在 C++11 新标准中刚刚被提出，因此在老代码中使用的都是后面三种形式。其中，使用 = 的两种形式是从 C 语言继承而来的。俗话说习惯成自然，即使是我，也会（不总是）在遇到用简单值初始化简单变量的时候，不自觉地使用 =。例如：

```
int x1 = 0;
char c1 = 'z';
```

然而，在面对稍微复杂一点的情况时，我还是建议读者使用 {}。使用 {} 的初始化称为*列表初始化*（list initialization），它能防止*窄化转换*（§ iso.8.5.4）。这句话的意思是：

- 如果一种整型存不下另一种整型的值，则后者不会被转换成前者。例如，允许 char 到 int 的类型转换，但是不允许 int 到 char 的类型转换。
- 如果一种浮点型存不下另一种浮点型的值，则后者不会被转换成前者。例如，允许 float 到 double 的类型转换，但是不允许 double 到 float 的类型转换。
- 浮点型的值不能转换成整型值。

- 整型值不能转换成浮点型的值。

例如：

```
void f(double val, int val2)
{
    int x2 = val;       // 如果 val==7.9，则 x2 的值变为 7
    char c2 = val2;     // 如果 val2==1025，则 c2 的值变为 1

    int x3 {val};       // 错误：不能发生收窄
    char c3 {val2};     // 错误：可能发生窄化转换

    char c4 {24};       // OK：24 能准确地表达成一个 char
    char c5 {264};      // 错误（假定 char 占 8 位）：264 不能表示成一个 char

    int x4 {2.0};       // 错误：不允许 double 到 int 的类型转换

    // ...
}
```

关于内置类型的转换规则请见 10.5 节。

当我们使用 auto 关键字从初始化器推断变量的类型时，没必要采用列表初始化的方式。而且如果初始化器是 {} 列表，则推断得到的数据类型肯定不是我们想要的结果（见 6.3.6.2 节）。例如：

```
auto z1 {99};   // z1 的类型是 initializer_list<int>
auto z2 = 99;   // z2 的类型是 int
```

因此当使用 auto 的时候应该选择 = 的初始化形式。

当我们构建某些类的对象时，可能有两种形式：一种是提供一组初始值；另一种是提供几个实参，这些实参不一定是实际存储的值，可能有别的含义。一个典型的例子是存放整数的 vector：

```
vector<int> v1 {99};    // v1 包含 1 个元素，该元素的值是 99
vector<int> v2(99);     // v2 包含 99 个元素，每个元素都取默认值 0
```

在上述代码中，我采用 (99) 的形式显式调用构造函数以实现第二种效果。大多数数据类型不提供这种语义含糊的初始化形式，即使是很多 vector 也不会；例如：

```
vector<string> v1("hello!");    // v1 含有 1 个元素，该元素的值是 "hello!"
vector<string> v2("hello!");    // 错误：vector 的任何构造函数都不接受字符串字面值常量作为参数
```

因此，除非你有充分的理由，否则最好使用 {} 初始化。

空初始化器列表 {} 指定使用默认值进行初始化，例如：

```
int x4 {};          // x4 被赋值为 0
double d4 {};       // d4 被赋值为 0.0
char* p {};         // p 被赋值为 nullptr
vector<int> v4{};   // v4 被赋值为一个空向量
string s4 {};       // s4 被赋值为 ""
```

大多数数据类型都有默认值。对于整数类型来说，默认值是数字 0 的某种适当形式。指针的默认值是 nullptr（见 7.2.2 节）。用户自定义类型的默认值（如果存在的话）由该类型的构造函数决定（见 17.3.3 节）。

对于用户自定义类型来说，直接初始化（允许隐式类型转换）和拷贝初始化（不允许隐式类型转换）可能会有所不同。相关细节请见 16.2.6 节。

特定类型对象的初始化问题将在后续章节中逐一介绍：

- 指针，见 7.2.2 节、7.3.2 节和 7.4 节。
- 引用，见 7.7.1 节（左值）和 7.7.2 节（右值）。
- 数组，见 7.3.1 节和 7.3.2 节。
- 常量，见 10.4 节。
- 类，见 17.3.1 节（不使用构造函数），17.3.2 节（使用构造函数），17.3.3 节（默认构造函数），17.4 节（成员和基类），17.5 节（拷贝和移动）。
- 用户定义的容器，见 17.3.4 节。

6.3.5.1　缺少初始化器

包括内置类型在内的很多类型都可能遇到缺少初始化器的情况。如果这真的发生了（事实上经常发生），那么事情会变得有点复杂。如果你不想面对这种复杂的情况，一定要时时记得初始化变量。未初始化变量的一种最有用的场景是当我们使用一个大的输入缓冲区时，例如：

```
constexpr int max = 1024*1024;
char buf[max];
some_stream.get(buf,max);   // 读入最多 max 个字符到 buf
```

要想初始化 buf 其实很容易：

```
char buf[max] {};           // 把每个字符都初始化成 0
```

但是这样做显然是多余的，而且可能会对程序性能造成非常严重的影响。无论如何，程序员应该尽量避免直接操作冲区，并且除非能百分之百确定（比如通过度量时间）未初始化缓冲区远优于初始化缓冲区，否则不要轻易地让缓冲区处于未初始化的状态。

如果没有指定初始化器，则全局变量（见 6.3.4 节）、名字空间变量（见 14.3.1 节）、局部 static 变量（见 12.1.8 节）和 static 成员（见 16.2.12 节）（统称为静态对象（static object））将会执行相应数据类型的列表 {} 初始化。例如：

```
int a;          // 等同于 ``int a{};''，因此 a 的值变为 0
double d;       // 等同于 ``double d{};''，因此 d 的值变为 0.0
```

对于局部变量和自由存储上的对象（有时也称为动态对象（dynamic object）或堆对象（heap object），见 11.2 节）来说，除非它们位于用户自定义类型的默认构造函数中（见 17.3.3 节），否则不会执行默认初始化。例如：

```
void f()
{
    int x;                          // x 没有一个定义良好的值
    char buf[1024];                 // buf[i] 没有一个定义良好的值

    int* p {new int};               // *p 没有一个定义良好的值
    char* q {new char[1024]};       // q[i] 没有一个定义良好的值

    string s;                       // 因为 string 的默认构造函数，所以 s=""
    vector<char> v;                 // 因为 vector 的默认构造函数，所以 v={}

    string* ps {new string};        // 因为 vector 的默认构造函数，所以 *ps 是 ""
    // ...
}
```

如果你想对内置类型的局部变量或者用 new 创建的内置类型的对象执行初始化，使用 {} 的

形式。例如：

```
void ff()
{
    int x {};                       // x 的值变为 0
    char buf[1024]{};               // 对于任意 i，buf[i] 的值变为 0

    int* p {new int{10}};           // *p 的值变为 10
    char* q {new char[1024]{}};     // 对于任意 i，q[i] 的值变为 0

    // ...
}
```

对于上面所示的数组和类来说，其成员将执行默认初始化。

6.3.5.2 初始化器列表

到目前为止，我们已经讨论了没有初始化器和只有一个初始化器的情况。复杂一点的对象可能需要多于一个初始化器，此时就要用到以一对花括号 {} 界定的初始化器列表了。例如：

```
int a[] = { 1, 2 };                         // 数组初始化器
struct S { int x; string s };
S s = { 1, "Helios" };                      // 结构的初始化器
complex<double> z = { 0, pi };              // 使用构造函数
vector<double> v = { 0.0, 1.1, 2.2, 3.3 };  // 使用列表构造函数
```

C 风格的数组初始化方式见 7.3.1 节，C 风格的结构初始化方式见 8.2 节，使用构造函数初始化用户自定义类型的方式见 2.3.2 节和 16.2.5 节，初始化器列表构造函数见 17.3.4 节。

在上面的例子中，符号 = 实际上是多余的。不过有的人愿意保留 =，以说明我们是在用一组值初始化一组成员变量。

在有的例子中，我们也可以使用函数风格的实参列表（见 2.3 节和 16.2.5 节），例如：

```
complex<double> z(0,pi);    // 使用构造函数
vector<double> v(10,3.3);   // 使用构造函数：v 包含 10 个元素，每个元素都初始化为 3.3
```

在声明语句中，一对空括号 () 通常表示“函数”（见 12.1 节）。因此，如果想显式地表达“执行默认初始化”的意愿，你需要使用 {}。例如：

```
complex<double> z1(1,2);    // 函数风格的初始化器（用构造函数执行初始化）
complex<double> f1();       // 函数声明

complex<double> z2 {1,2};   // 用构造函数初始化成 {1,2}
complex<double> f2 {};      // 用构造函数初始化成默认值 {0,0}
```

请注意，当我们使用 {} 符号进行初始化时，不会进行窄化转换（见 6.3.5 节）。

在使用了 auto 的语句中，{} 列表的类型被推断为 std::initializer_list<T>。例如：

```
auto x1 {1,2,3,4};          // x1 的类型是 initializer_list<int>
auto x2 {1.0, 2.25, 3.5 };  // x2 的类型是 initializer_list of<double>
auto x3 {1.0,2};            // 错误：无法推断 {1.0,2} 的类型（见 6.3.6.2 节）
```

6.3.6 推断类型：auto 和 decltype()

C++ 语言提供了两种从表达式中推断数据类型的机制：

- auto 根据对象的初始化器推断对象的数据类型，可能是变量、const 或者 constexpr 的类型。

- decltype(expr) 推断的对象不是一个简单的初始化器，有可能是函数的返回类型或者类成员的类型。

这里所谓的推断其实非常简单：auto 和 decltype() 只是简单地报告一个编译器已知的表达式的类型。

6.3.6.1　auto 类型修饰符

当声明语句中的变量含有初始化器时，我们无须显式地指定变量的类型，只要让变量取其初始化器的类型即可。例如：

```
int a1 = 123;
char a2 = 123;
auto a3 = 123; // a3 的类型是 int
```

整数字面值常量 123 的类型是 int，因此 a3 的类型就是 int。换句话说，我们可以把 auto 看成是初始化器类型的占位符。

在像 123 这样简单的表达式中，用 auto 代替 int 看起来没什么大不了的。但是，表达式的类型越难读懂、越难书写，auto 就越有用。例如：

```
template<class T> void f1(vector<T>& arg)
{
    for (vector<T>::iterator p = arg.begin(); p!=arg.end(); ++p)
        *p = 7;

    for (auto p = arg.begin(); p!=arg.end(); ++p)
        *p = 7;
}
```

对于上面的程序来说，使用 auto 显然是更好的选择。它易读易写，且能在一定程度上适应代码的变化。例如，如果我们把 arg 的类型更改成 list，则使用 auto 的循环仍然可以正常工作，而第一个循环需要重写。因此，在较小的作用域中，建议程序员优先选择使用 auto。

如果作用域的范围较大，则显式地指定类型有助于定位错误。换句话说，与使用明确的类型名相比，使用 auto 可能会使得定位类型错误的难度增大。例如：

```
void f(double d)
{
    constexpr auto max = d+7;
    int a[max];          // 错误：数组边界不是整数
    // ...
}
```

为了解决 auto 可能造成的影响，最常规的做法是保持函数的规模较小，这也被证明是一种行之有效的方法（见 12.1 节）。

我们可以为推断出的类型添加修饰符或说明符（见 6.3.1 节），比如 const 和 &（引用，见 7.7 节）。例如：

```
void f(vector<int>& v)
{
    for (const auto& x : v) {  // x 的类型是 const int&
        // ...
    }
}
```

在此例中，auto 推断为 v 的元素的类型，即 int。

请注意，表达式的类型永远不会是引用类型，因为表达式会隐式地执行解引用操作（见 7.7 节）。例如：

```
void g(int& v)
{
    auto x = v;     // x 的类型是 int，而不是 int&
    auto& y = v;    // y 的类型是 int&
}
```

6.3.6.2　auto 与 {} 列表

如果我们正在初始化某个对象，那么当提到它的类型时必须同时考虑两部分：对象本身的类型以及初始化器的类型。例如：

```
char v1 = 12345;    // 12345 的类型是 int
int v2 = 'c';       // 'c' 的类型是 char
T v3 = f();
```

使用 {} 初始化器列表可以尽可能减少意料之外的类型转换：

```
char v1 {12345};    // 错误：窄化转换
int v2 {'c'};       // 正确：显式从 char->int 类型转换
T v3 {f()};         // 当且仅当 f() 的类型能被显式地转换成 T 时，该语句有效
```

当我们使用 auto 关键字时，只涉及一种类型（初始值的类型），此时使用 = 是安全的，不会有什么问题：

```
auto v1 = 12345;    // v1 的类型是 int
auto v2 = 'c';      // v2 的类型是 char
auto v3 = f();      // v3 将会是某种适当的类型
```

事实上，当声明语句中有 auto 关键字时，= 是比 {} 更好的选择，因为前者的结果可能并非我们所愿：

```
auto v1 {12345};    // v1 的类型是 int 的列表
auto v2 {'c'};      // v2 的类型是 char 的列表
auto v3 {f()};      // v3 是某种类型的列表
```

这是符合逻辑的。请考虑如下情况：

```
auto x0 {};         // 错误：无法推断类型
auto x1 {1};        // int 的列表，包含 1 个元素
auto x2 {1,2};      // int 的列表，包含 2 个元素
auto x3 {1,2,3};    // int 的列表，包含 3 个元素
```

由同一种类型 T 的元素组成的列表类型是 initializer_list<T>（见 3.2.1.3 节和 11.3.3 节）。应该庆幸 x1 的类型没有被推断成 int，否则的话我们真不知道 x2 和 x3 该怎么办了。

总之，只要我们不是期望得到某种“列表”类型，就应该选择 = 而非 {}。

6.3.6.3　decltype() 修饰符

当有一个合适的初始化器的时候可以使用 auto。但是很多时候我们既想推断得到类型，又不想在此过程中定义一个初始化的变量，此时，我们应该使用声明类型修饰符 decltype(expr)。其中，推断所得的结果是 expr 的声明类型。这种用法在泛型编程中很有效。请考虑这样一个问题：如果我们想编写一个函数令其执行两个矩阵的加法运算，但是两个矩阵的元素类型可能不同，那么相加之后所得结果的类型应该是什么呢？当然是矩阵，但是这个结果矩阵的元素是什么类型？最自然的回答是：结果矩阵的元素类型应该是对应元素求和后的类型。因此，我们的声明如下所示：

```
template<class T, class U>
auto operator+(const Matrix<T>& a, const Matrix<U>& b) -> Matrix<decltype(T{}+U{})>;
```

在这个声明中我使用了后置返回类型语法（见 12.1 节），以便通过 Matrix<decltype(T{}+U{})> 推断出函数的返回类型。换句话说，函数的结果是一个 Matrix，Matrix 的元素类型由 T{}+U{} 推断得到。

在该函数的定义部分，我再一次使用 decltype() 来表示 Matrix 的元素类型：

```
template<class T, class U>
auto operator+(const Matrix<T>& a, const Matrix<U>& b) -> Matrix<decltype(T{}+U{})>
{
    Matrix<decltype(T{}+U{})> res;
    for (int i=0; i!=a.rows(); ++i)
        for (int j=0; j!=a.cols(); ++j)
            res(i,j) += a(i,j) + b(i,j);
    return res;
}
```

6.4　对象和值

我们可以分配并使用没有名字的对象（比如用 new 创建的对象），也能为某些看起来不太寻常的表达式赋值（如，*p[a+10]=7）。因此，我们需要用一个名字来表示“内存中的某个东西”。这就是对象一词最简单和最基本的含义。换句话说，对象（object）是指一块连续存储区域，左值（lvalue）是指向对象的一条表达式。“左值”的字面意思是“能用在赋值运算符左侧的东西”，但其实不是所有左值都能用在赋值运算符的左侧，左值也有可能指示某个常量（见 7.7 节）。未被声明成 const 的左值称为可修改的左值（modifiable lvalue）。此处我们提到的对象的最简单和最基本的含义不应该与类的对象或多态类型的对象混淆（见 3.2.2 节和 20.3.2 节）。

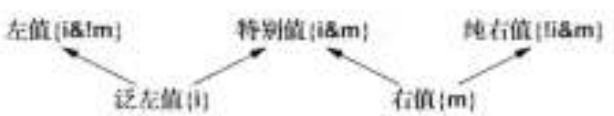
6.4.1　左值和右值

为了补充和完善左值的含义，我们相应地定义了右值（rvalue）。简单来说，右值是指“不能作为左值的值”，比如像函数返回值一样的临时值。

如果你希望技术性更强一些（比如你想读一下 ISO C++ 标准），那就需要更新看待左值和右值的视角了。当考虑对象的寻址、拷贝、移动等操作时，有两种属性非常关键。

- 有身份（Has identity）：在程序中有对象的名字，或指向该对象的指针，或该对象的引用，这样我们就能判断两个对象是否相等或者对象的值是否发生了改变。
- 可移动（Is movable）：能把对象的内容移动出来（比如，我们能把它的值移动到其他某处，剩下的对象处于合法但未指定的状态，与拷贝是有差别的，见 17.5 节）。

在上述两个属性的四种组合形式中，有三种需要用 C++ 语言规则精确地描述（既没有身份又不能移动的对象不重要）。我们“用 m 表示可移动”，且“用 i 表示有身份”，从而把表达式的分类表示成下图所示的形式：



从图中可知，一个经典的左值有身份但不能移动（因为我们可能会在移动后仍然使用它），

而一个经典的右值是允许执行移出操作的对象。其他一些有关的术语还包括纯右值（prvalue）、泛左值（glvalue）和特别值（xvalue，又称为专家值，人们对于这个“x”的解释极具想象力[⊖]）。例如：

```
void f(vector<string>& vs)
{
    vector<string>& v2 = std::move(vs);    // 移动 vs 到 v2
    // ...
}
```

此处，std::move(vs) 是一个特别值。它明显有身份（我们能像 vs 一样引用它），并且我们显式地给予了将其值移出的许可，方式是调用 std::move()（见 3.3.2 节和 35.5.1 节）。

在实际编程过程中，考虑左值和右值就足够了。一条表达式要么是左值，要么是右值，不可能两者都是。

6.4.2 对象的生命周期

对象的生命周期（lifetime）从对象的构造函数完成的那一刻开始，直到析构函数执行为止。对于那些没有声明构造函数的类型（比如 int），我们可以认为它们拥有默认的构造函数和析构函数，并且这两个函数不执行任何实际操作。

我们从生命周期的角度把对象划分成以下类别。

- 自动（automatic）对象：除非程序员特别说明（见 12.1.8 节和 16.2.12 节），否则在函数中声明的对象在其定义处被创建，当超出作用域范围时被销毁。这样的对象被称为自动（automatic）对象。在大多数实现中，自动对象被分配在栈空间上。每调用一次函数，获取新的栈帧（stack frame）以存放它的自动对象。
- 静态（static）对象：在全局作用域或名字空间作用域（见 6.3.4 节）中声明的对象以及在函数（见 12.1.8 节）或类（见 16.2.12 节）中声明的 static 成员只被创建并初始化一次，并且直到程序结束之前都“活着”（见 15.4.3 节）。这样的对象被称为静态（static）对象。静态对象在程序的整个执行周期内地址唯一。在多线程环境中，静态对象可能会造成某些意料之外的问题。因为所有线程都共享静态对象，所以必须为其加锁以避免数据竞争（见 5.3.1 节和 42.3 节）。
- 自由存储（free store）对象：用 new 和 delete 直接控制其生命周期的对象。
- 临时（temporary）对象：比如计算的中间结果或用于存放 const 实参引用的值的对象。临时对象的生命周期由其用法决定。如果临时对象被绑定到一个引用上，则它的生命周期就是引用的生命周期；否则，临时对象的生命周期与它所处的完整表达式一致。其中，完整表达式（full expression）不属于任何其他表达式。通常情况下，临时对象也是自动对象。
- 线程局部（thread-local）对象，或者说声明为 thread_local（见 42.2.8 节）的对象：这样的对象随着线程的创建而创建，随着线程的销毁而销毁。

其中，静态和自动被称为存储类（storage class）。

数组元素和非静态类成员的生命周期由它们所属的对象决定。

⊖ 此句需要在英文语境下理解，特别值的原词是 extraordinary value，专家值的原词是 expert only value，两个名字都有字母 x，与 xvalue 吻合，是一种巧妙的解释。——译者注

6.5　类型别名

有时，我们需要为某种类型起个新名字。可能的动机包括：

- 原来的名字太长、太复杂或者太难看（在某些程序员眼中）。
- 某项程序设计技术要求在同一段上下文中，不同类型有相同的名字。
- 在某处提及某种类型仅仅是为了便于后期维护。

例如：

```
using Pchar = char*;          // 字符串指针
using PF = int(*)(double);    // 函数指针，该函数接受一个 double 且返回一个 int
```

相似类型可以定义同一个名字作为成员别名：

```
template<class T>
class vector {
    using value_type = T;          // 每个容器都有一个 value_type
    // ...
};

template<class T>
class list {
    using value_type = T;          // 每个容器都有一个 value_type
    // ...
};
```

无论如何，类型别名绝不代表一种新类型，它只是某种已有类型的同义词。换句话说，别名就是类型的另外一个名字而已。例如：

```
Pchar p1 = nullptr;       // p1 的类型是 char*
char* p3 = p1;            // 正确
```

如果你想实现一种新类型，并且它的语义和表达形式与某种已有类型一致，应该使用枚举（见 8.4 节）或者类（第 16 章）。

早期还有一种语法也可以用在类似的语境中，我们使用 typedef 关键字，然后把要声明的类型别名放在一般声明语句中变量所在的位置上。例如：

```
typedef int int32_t;          // 等价于 "using int32_t = int;"
typedef short int16_t;        // 等价于 "using int16_t = short;"
typedef void(*PtoF)(int);     // 等价于 "using PtoF = void(*)(int);"
```

使用别名有助于我们把代码与机器细节分离开来。名字 int32_t 明确指出我们想用它表示占 32 个二进制位的整数。与“普通的 int”相比，使用 int32_t 可以让我们把代码平滑地移植到一台 sizeof(int)==2 的机器上。我们要做的只是修改一处 int32_t 的定义令其表示一个尺寸更大的整数类型即可：

```
using int32_t = long;
```

后缀 _t 通常用来表示类型别名（源自 typedef 关键字）。int16_t、int32_t 以及其他一些类型别名都定义在 <cstdint> 中（见 43.7 节）。类型的别名应该尽量反映出该类型的目的和作用，而不是类型的实现细节（见 6.3.3 节）。

using 关键字可用于引入一个 template 别名（见 23.6 节），例如：

```
template<typename T>
    using Vector = std::vector<T, My_allocator<T>>;
```

不允许在类型别名前加修饰符（如 unsigned），例如：

```
using Char = char;
using Uchar = unsigned Char;      // 错误
using Uchar = unsigned char;      // OK
```

6.6 建议

[1] 如果想了解语言的更多细节和权威解释，请翻阅 ISO C++ 标准；6.1 节。

[2] 尽量避免不确定的和未定义的行为；6.1 节。

[3] 如果某些代码必须依赖于具体实现，记得把它们与程序的其他部分分离开来；6.1 节。

[4] 对于字符对应的数字值不要乱作假定；6.2.3.2 节，10.5.2.1 节。

[5] 以 0 开头的整数是八进制的；6.2.4.1 节。

[6] 不要使用“魔法常量”；6.2.4.1 节。

[7] 不要对整数的尺寸妄加猜测；6.2.8 节。

[8] 不要对浮点数的精度和表示范围妄加猜测；6.2.8 节。

[9] 尽量使用普通的 char，而非 signed char 或者 unsigned char；6.2.3.1 节。

[10] 注意带符号类型和无符号类型之间的转换；6.2.3.1 节。

[11] 在一条声明语句中只声明一个名字；6.3.2 节。

[12] 常用的、局部的名字尽量短；不常用的、非局部的名字可以长一些；6.3.3 节。

[13] 起的名字不要太相似；6.3.3 节。

[14] 对象的名字应该尽量反映对象的含义而非类型；6.3.3 节。

[15] 坚持一种统一的命名风格；6.3.3 节。

[16] 避免使用全大写的名字；6.3.3 节。

[17] 作用域宜小不宜大；6.3.4 节。

[18] 最好不要在一个作用域以及它的外层作用域中使用相同的名字；6.3.4 节。

[19] 使用指定类型声明时最好用 {} 初始化器语法；6.3.5 节。

[20] 使用 auto 声明时最好用 = 语法；6.3.5 节。

[21] 避免使用未初始化的变量；6.3.5.1 节。

[22] 当内置类型被用来表示一个可变值时，不妨给它起个能反映其含义的别名；6.5 节。

[23] 用别名作为类型的同义词；用枚举和类定义新类型；6.5 节。

指针、数组与引用

崇高与荒谬，
往往就在一线之间。

——托马斯·潘恩

- 引言
- 指针
  void*；nullptr
- 数组
  数组的初始化器；字符串字面值常量
- 数组中的指针
  数组漫游；多维数组；传递数组
- 指针与 const
- 指针与所有权
- 引用
  左值引用；右值引用；引用的引用；指针与引用
- 建议

7.1 引言

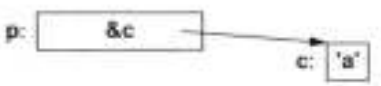
本章介绍 C++ 语言中指示某块内存区域的基本机制。显然，我们能通过名字使用对象。然而在 C++ 中，大多数对象都“有身份”；也就是说对象位于内存的某个地址中。如果我们知道对象的地址和类型，就能访问它。在 C++ 语言中存放及使用内存地址是通过指针和引用完成的。

7.2 指针

对于类型 T 来说，T* 是表示“指向 T 的指针”的类型。换句话说，T* 类型的变量能存放 T 类型对象的地址。例如：

```
char c = 'a';
char* p = &c;        // p 存放着 c 的地址，& 是取地址运算符
```

这两条语句的图形化表示是：



对指针的一个基本操作是解引用（dereferencing），即引用指针所指的对象。这个操作也称为间接取值（indirection）。解引用运算符是个前置一元运算符，对应的符号是 *。例如：

```
char c = 'a';
char* p = &c;   // p 存放着 c 的地址，& 是取地址运算符
char c2 = *p;   // c2 == 'a'; * 解引用运算符
```

指针 p 所指的对象是 c，c 中存储的值是 'a'，因此我们把 *p 赋给 c2 等价于给 c2 赋值 'a'。

当指针指向数组中的元素时，C++ 允许对这类指针执行某些算术运算（见 7.4 节）。

指针的具体实现应该与运行程序的机器的寻址机制同步。大多数机器支持逐字节访问内存，其他机器则需要从字中抽取字节。很少有机器能直接寻址到一个二进制位。因此，能独立分配且用内置指针指向的最小对象是 char 类型的对象。有一点请读者注意：bool 占用的内存空间至少和 char 一样多（见 6.2.8 节）。如果想把更小的值存得更紧密，可以使用位逻辑操作（见 11.1.1 节）、结构中的位域（见 8.2.7 节）或者 bitset（见 34.2.2 节）。

符号 * 在用作类型名的后缀时表示“指向”的含义。如果我们想表示指向数组的指针或者指向函数的指针，需要使用稍微复杂一些的形式：

```
int* pi;            // 指向 int 的指针
char** ppc;         // 指向字符指针的指针
int* ap[15];        // ap 是一个数组，包含 15 个指向 int 的指针
int (*fp)(char*);   // 指向函数的指针，该函数接受一个 char* 实参，返回一个 int
int* f(char*);      // 该函数接受一个 char* 实参，返回一个指向 int 的指针
```

6.3.1 节解释了如何声明一个指针，§ iso.A 则包含完整的语法信息。

指向函数的指针很有用，相关内容将在 12.5 节讨论；20.6 节将介绍指向类成员的指针。

7.2.1　void*

在某些偏向底层的代码中，我们偶尔需要在不知道对象确切类型的情况下，仅通过对象在内存中的地址存储或传递对象。此时，我们会用到 void*。void* 的含义是“指向未知类型对象的指针”。

除了函数指针（见 12.5 节）和指向类成员的指针（见 20.6 节），指向其他任意类型对象的指针都能被赋给一个 void* 类型的变量。此外，一个 void* 能被赋给另一个 void*，两个 void* 能比较是否相等，我们还能把 void* 显式地转换成其他类型。因为编译器事实上并不清楚 void* 所指的对象到底是什么类型，所以对它执行其他操作可能不太安全并且会引发编译器错误。要想使用 void*，我们必须把它显式地转换成某一特定类型的指针。例如：

```
void f(int* pi)
{
    void* pv = pi;  // ok：发生了从 int* 到 void* 的隐式类型转换
    *pv;            // 错误：不允许解引用 void*
    ++pv;           // 错误：不允许对 void* 执行递增操作（所指向对象尺寸未知）

    int* pi2 = static_cast<int*>(pv);          // 显式转换回 int*

    double* pd1 = pv;                          // 错误
    double* pd2 = pi;                          // 错误
    double* pd3 = static_cast<double*>(pv);    // 不安全（见 11.5.2 节）
}
```

一般情况下，如果某个指针已经被转换成（强制类型转换）指向一种与实际所指对象类型完全不同的新类型，则使用转换后的指针是不安全的行为。例如，某个机器可能假定 double 沿着 8 字节边界分配内存，如果指向 int 的 pi 分配内存的方式与之不同，将造成意想不到的

后果。这种显式类型转换既不安全也不自然，我们在设计 static_cast（见 11.5.2 节）的时候考虑到了这一点：static_cast 的字面形式比较特殊，一旦出现了与之有关的错误，程序员很容易定位。

void* 最主要的用途是当我们无法假定对象的类型时，向函数传递指向该对象的指针；它还用于从函数返回未知类型的对象。要想使用这样的对象，必须先进行显式类型转换。

用到 void* 指针的函数通常位于系统的最底层，这些函数的作用大多是操作硬件资源。例如：

```
void* my_alloc(size_t n);      // 从特定的堆上分配 n 个字节的内存空间
```

在系统的较上层代码中很少用到 void*，一旦出现了你就要认真核实是不是存在设计上的错误。当用于优化的目的时，void* 能隐藏在类型安全的接口（见 27.3.1 节）中。

函数指针（见 12.5 节）和指向类成员的指针（见 20.6 节）不能被赋给 void*。

7.2.2 nullptr

字面值常量 nullptr 表示空指针，即不指向任何对象的指针。我们可以把 nullptr 赋给其他任意指针类型，但是不能赋给其他内置类型：

```
int* pi = nullptr;
double* pd = nullptr;
int i = nullptr;              // 错误：i 不是指针
```

nullptr 只有一个，它可以用于任意指针类型，C++ 并没有为每种指针类型各设计一个空指针。

在 nullptr 被引入之前，人们使用数字 0 表示空指针。例如：

```
int* x = 0; // x 的值是 nullptr
```

任何对象都不会分配到地址 0 上，0（所有位全 0 的模式）是 nullptr 最常见的表示形式。0 本身是一个 int，但是标准类型转换规则（见 10.5.2.3 节）允许我们把 0 当成是一个指针常量或指向成员的类型的常量。

在原来的代码中，很多人习惯于定义一个宏 NULL 来表示空指针。例如：

```
int* p = NULL; // 使用宏 NULL
```

然而，在不同的具体实现中 NULL 的定义有所差别；例如，NULL 可能是 0，也可能是 0L。在 C 语言中，NULL 通常是 (void*)0，这种用法在 C++ 中是非法的（见 7.2.1 节）：

```
int* p = NULL; // 错误：不能把 void* 赋给 int*
```

使用 nullptr 的好处很多，首先它的可读性更强，其次当一组重载函数既可以接受指针也可以接受整数时（见 12.3.1 节），用 nullptr 能够避免语义混淆。

7.3 数组

假设有类型 T，T[size] 的含义是“包含 size 个 T 类型元素的数组”。元素的索引值范围是 0 到 size-1。例如：

```
float v[3];      // 包含 3 个 float 的数组，分别是 v[0], v[1], v[2]
char* a[32];     // 包含 32 个 char 指针的数组，依次是 a[0] ... a[31]
```

你可以使用下标运算符 [] 或指针（运算符 * 或运算符 []，见 7.4 节）访问数组中的元素，

例如：

```
void f()
{
    int aa[10];
    aa[6] = 9;      // 为数组 aa 的第 7 个元素赋值
    int x = aa[99]; // 未定义的行为
}
```

越界访问数组是一种未定义的行为，而且很有可能会引发严重的程序错误。在 C++ 语言中，运行时边界检查既不常见，也无法保证。

数组中元素的数量（即数组的边界）必须是常量表达式（见 10.4 节）。如果你希望边界可变，最好使用 vector（见 4.4.1 节和 31.4 节）。例如：

```
void f(int n)
{
    int v1[n];              // 错误：数组的大小不是常量表达式
    vector<int> v2(n);      // OK：包含 n 个 int 元素的 vector
}
```

多维数组表现为数组的数组（见 7.4.2 节）。

数组是 C++ 表示内存中对象序列最基本的方式。如果你用到的只是内存中一个固定大小、固定元素类型的序列，那么数组完全可以满足你的要求。对于其他要求，数组就不一定可靠了。

C++ 允许静态地分配数组空间，也允许在栈上或者在自由存储上（见 6.4.2 节）分配数组空间。例如：

```
int a1[10];                 // 静态存储中的 10 个 int

void f()
{
    int a2 [20];            // 栈上的 20 个 int
    int* p = new int[40];   // 自由存储上的 40 个 int
    // ...
}
```

C++ 的内置数组本质上是语言的一种底层功能，我们常常用数组来实现标准库 vector 和 array 等更高层级上的、行为定义更好的数据结构。数组不能执行赋值操作，一旦需要，数组名就会隐式地转换成指向数组首元素的指针（见 7.4 节）。特别要注意避免在接口中使用数组（比如作为函数的参数，见 7.4.3 节和 12.2.2 节），因为数组名隐式转换成指针是 C 代码和 C 风格的 C++ 代码中很多错误的根源。如果是在自由存储上分配数组的，切记一定要在最后一次使用数组之后把对应的指针 delete[] 掉（见 11.2.2 节）。要让程序员时刻遵守这一要求并不容易，最简便且最可靠的办法是用资源句柄（比如 string（见 19.3 节和 36.3 节）、vector（见 13.6 节和 34.2 节）和 unique_ptr（见 34.3.1 节））控制自由存储上的数组的生命周期。如果你是静态地分配数组或者是在栈上分配数组，一定不要 delete[] 它。显然，因为 C 语言本身缺乏封装数组的能力，所以 C 程序员很难完全遵循上述建议；但是这些建议在 C++ 程序中非常有用，不存在任何适用性的问题。

以 0 作为终止符的 char 数组是应用最广泛的一种数组。这是 C 语言存储字符串的基本方式，因此我们常把以 0 作为终止符的 char 数组称为 C 风格字符串（C-style string）。C++ 的字符串字面值常量沿用了这一传统（见 7.3.2 节），并且某些标准库函数（比如 strcpy() 和

strcmp()，见 43.4 节）也是建立在这一用法之上的。通常情况下，char* 和 const char* 指向以 0 结尾的字符序列。

7.3.1　数组的初始化器

我们能用值的列表初始化一个数组，例如：

```
int v1[] = { 1, 2, 3, 4 };
char v2[] = { 'a', 'b', 'c', 0 };
```

如果声明数组的时候没有指定它的大小但是给出了初始化器列表，则编译器会根据列表包含的元素数量自动计算数组的大小。因此，v1 和 v2 的类型分别是 int[4] 和 char[4]。如果我们指定了数组的大小，但是提供的初始化器列表元素数量过多，则程序会发生错误。例如：

```
char v3[2] = { 'a', 'b', 0 };      // 错误：提供的初始化器过多
char v4[3] = { 'a', 'b', 0 };      // OK
```

如果初始化器提供的元素数量不足，则系统自动把剩余的元素赋值为 0。例如：

```
int v5[8] = { 1, 2, 3, 4 };
```

等价于

```
int v5[] = { 1, 2, 3, 4 , 0, 0, 0, 0 };
```

C++ 没有为数组提供内置的拷贝操作。不允许用一个数组初始化另一个数组（即使两个数组的类型完全一样也不行），因为数组不支持赋值操作：

```
int v6[8] = v5;  // 错误：不允许拷贝数组（不允许把 int* 赋给数组）
v6 = v5;         // 错误：不存在数组的赋值操作
```

同样，不允许以传值方式传递数组（见 7.4 节）。

如果你想给一组对象赋值，可以使用 vector（见 4.4.1 节，13.6 节和 34.2 节）、array（见 8.2.4 节）或者 valarry（见 40.5 节）。

我们可以用字符串字面值常量（见 7.3.2 节）初始化字符的数组。

7.3.2　字符串字面值常量

字符串字面值常量（string literal）是指双引号内的字符序列：

```
"this is a string"
```

字符串字面值常量实际包含的字符数量比它表现出来的样子多一个。它以一个取值为 0 的空字符 '\0' 结尾，例如：

```
sizeof("Bohr")==5
```

字符串字面值常量的类型是“若干个 const 字符组成的数组”，因此 "Bohr" 的类型是 const char[5]。

在 C 和旧式的 C++ 代码中，允许把字符串字面值常量赋给一个非常量 char*：

```
void f()
{
    char* p = "Plato";  // 错误，但是被 C++11 之前的代码接受
    p[4] = 'e';         // 错误：试图为常量赋值
}
```

显然上面的赋值语句是不安全的。这种用法有出错的风险，因此如果某些老代码因为这个原

因无法编译通过再正常不过了。令字符串字面值常量的内容保持不变是一种显而易见的选择，这样做有助于在具体实现环节对字符串字面值常量的存储和访问方式加以改进。

如果我们希望字符串能被修改，最好把字符放在一个非常量的数组中：

```
void f()
{
    char p[] = "Zeno";   // p是含有5个字符的数组
    p[0] = 'R';          // OK
}
```

字符串字面值常量是静态分配的，因此函数返回字符串字面值常量是很安全的行为，不会有什么问题。例如：

```
const char* error_message(int i)
{
    // ...
    return "range error";
}
```

调用 error_message() 后，存放 "rang e error" 的内存区域不会消失。

两个完全一样的字符串字面值常量是在同一个数组中还是在两个不同的数组中依赖于实现（见 6.1 节），例如：

```
const char* p = "Heraclitus";
const char* q = "Heraclitus";

void g()
{
    if (p == q) cout << "one!\n";        // 结果依赖于实现
    // ...
}
```

当符号 == 作用于指针时，比较的是地址（指针本身的值）而非指针所指的值。

空字符串记作一对紧挨着的双引号 ""，其类型是 const char[1]。空字符串中唯一的一个字符是结束符 '\0'。

只要表示的是非图形化字符，反斜线（见 6.2.3.2 节）就能用在字符串中。这使得我们可以在字符串中表示双引号（"）和转义字符反斜线（\）。我们最常用的是换行符 '\n'，例如：

```
cout<<"beep at end of message\a\n";
```

转义字符 '\a' 是 ASCII 字符集中的 BEL（警告，alert），它的作用是发出报警的声音。

在一个非原始字符串字面值常量中，不存在“真正的”换行：

```
"this is not a string
but a syntax error"
```

我们可以用空格把一条长字符串分割开以使程序文本显得整洁美观，例如：

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz"
               "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

编译器会自动把相邻的字符串连接起来，因此用下面这条长字符串字面值常量初始化 alpha，从效果上来说是等价的：

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

允许在字符串中间存在空字符，但是大多数程序都会忽略空字符之后的内容。例如，标准库

函数 strcpy() 和 strlen() 都把字符串 "Jens\000Munk" 当成 "Jens" 处理（见 43.4 节）。

7.3.2.1　原始字符串

要想在字符串字面值常量中表示反斜线（\）或者双引号（"），我们需要在这些符号的前面再加一个反斜线。这种做法合情合理，并且在大多数时候都有效。然而，如果我们要表达的字符串字面值常量中含有太多的反斜线和双引号，问题就变得比较复杂了。特别是在正则表达式中反斜线会频繁地出现，它既用来表示转义字符，又能表示某些字符类别（见 37.1.1 节）。我们无权更改或优化正则表达式的规则，毕竟包括 C++ 在内的很多编程语言都在使用它，几乎已经形成传统了。因此，当你在使用标准 regex 库（第 37 章）书写正则表达式时，反斜线可以表示转义字符这一事实很可能会造成某些潜在的错误。请思考一个问题，如果我们想表示两个用反斜线隔开的单词，可能需要这么写：

```
string s = "\\w\\\\w";      // 这么多反斜线？祈祷自己千万别写错！
```

显然这种约定俗成的正则表达式太容易出错了，为了解决这个问题，C++ 提供了原始字符串字面值常量（raw string literal）。在原始字符串字面值常量中，反斜线就是反斜线，双引号就是双引号，上面的例子变成了：

```
string s = R"(\w\\w)";      // 嗯，没问题，肯定不会写错的！
```

原始字符串字面值常量使用 R"(ccc)" 的形式表示字符序列 ccc，其中，开头的 R 用于把原始字符串字面值常量和普通的字符串字面值常量区别开来。一对括号的作用是允许我们使用非转义的双引号。例如：

```
R"("quoted string")"      // 字符串的内容是 "quoted string"
```

进一步，如果我们想在原始字符串字面值常量中加入字符序列)" 该怎么办呢？这个要求并不常见，不过即使真的遇到了也有解决的办法。因为 "(和)" 并不是唯一的分隔符，在 "(...)" 的框架中我们还可以在 (之前和) 之后加入其他分隔符。例如：

```
R"***("quoted string containing the usual terminator (")")"***"
    // "quoted string containing the usual terminator (")"
```

规则要求：符号) 后面的字符序列必须与符号 (前面的序列完全一致。采用这种措施，我们就能处理几乎所有复杂的字符串模式了。

除非你正在处理正则表达式，否则原始的字符串字面值常量不会有太大的用处（顶多算是“茴香豆的茴的一种写法”）。但是正则表达式本身是非常非常有用的，读者不妨思考现实世界中的一个例子：

```
"('(?:['\\\"]|\\\\.)+'|\"(?:['\\\\\"]|\\\\.)+\")|"      // 这些反斜线的用法对吗？
```

面对这个例子，即使是有经验的程序员也很难做出判断，此时原始字符串字面值常量就派上用场了。

与普通的字符串字面值常量不同，在原始字符串字面值常量中允许出现换行（真正的换行，而非换行符）。例如：

```
string counts {R"(1
22
333)"};
```

等价于

```
string x {"1\n22\n333"};
```

7.3.2.2　大字符集

前缀是 L 的字符串（比如 L"angst"）由宽字符（见 6.2.3 节）组成，它的类型是 const wchar_t[]。类似地，前缀是 LR 的字符串（比如 LR"(angst)"）也是由宽字符组成的（见 7.3.2.1 节），它的类型同样是 const wchar_t[]，它属于原始字符串字面值常量。这样的字符串以字符 L'\0' 结束。

有 6 种字符字面值常量支持 Unicode（称为 Unicode 字面值常量，Unicode literal）。这听起来有点多，但是要知道 Unicode 本身有至少 3 种编码方式：UTF-8、UTF-16 和 UTF-32。对于每种编码方式，分别支持原始字符串以及“普通”字符串。因为每种 UTF 编码方式都支持全部 Unicode 字符，所以到底选用哪种编码方式主要看程序所要依赖的系统是什么要求。基本上，所有 Internet 应用（比如浏览器和电子邮件）都支持至少一种 Unicode 编码方式。

UTF-8 是一种可变宽度的编码方式：常用字符占据 1 个字节，不常用的字符（根据使用的情况度量）占据 2 个字节，特别罕见的字符占据 3 或 4 个字节。众所周知，ASCII 字符占 1 个字节，这些字符在 UTF-8 中的编码（对应的整数值）与在 ASCII 中完全一致。拉丁字母、希腊文、斯拉夫语、希伯来文、阿拉伯文以及其他字符占 2 个字节。

UTF-8 字符串的结尾是 '\0'，UTF-16 是 u'\0'，UTF-32 是 U'\0'。

英文字符串的表示方式有很多种，考虑以反斜线作为分隔符的文件名：

```
"folder\\file"        // 基于实现字符集的字符串
R"(folder\file)"      // 基于实现字符集的原始字符串
u8"folder\\file"      // UTF-8 字符串
u8R"(folder\file)"    // UTF-8 原始字符串
u"folder\\file"       // UTF-16 字符串
uR"(folder\file)"     // UTF-16 原始字符串
U"folder\\file"       // UTF-32 字符串
UR"(folder\file)"     // UTF-32 原始字符串
```

上面这些字符串的输出效果都是一样的，但是除了“普通”字符串和 UTF-8 字符串外，其他字符串的内在表现形式略有区别。

显然，Unicode 字符串的最终目的是处理 Unicode 字符，例如：

```
u8"The official vowels in Danish are: a, e, i, o, u, \u00E6, \u00F8, \u00E5 and y."
```

输出该字符串所得的结果是：

```
The official vowels in Danish are: a, e, i, o, u, æ, ø, å and y.
```

\u 之后的十六进制数是一个 Unicode 编码点（§ iso.2.14.3）[Unicode，1996]。编码点独立于编码方式，事实上，在不同编码方式下编码点的表现形式会有所不同。例如，u'0430'（斯拉夫语的小写字母“а”）在 UTF-8 中是 2 字节的十六进制值 D0B0，在 UTF-16 中是 2 字节的十六进制值 0430，在 UTF-32 中是 4 字节的十六进制值 00000430。这些十六进制值称为通用字符名字（universal character name）。

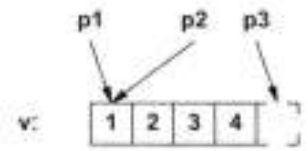
前缀 u 和 R 对于顺序和大小写敏感：RU 和 Ur 都不是合法的字符串前缀。

7.4　数组中的指针

在 C++ 语言中，指针与数组密切相关。数组名可以看成是指向数组首元素的指针，例如：

```
int v[] = { 1, 2, 3, 4 };
int* p1 = v;          // 指向数组首元素的指针（隐式转换）
int* p2 = &v[0];      // 指向数组首元素的指针
int* p3 = v+4;        // 指向数组尾后位置的指针
```

或者表示成图形的形式：



令指针指向数组的最后一个元素的下一个位置（尾后位置）是有效的，这对于很多算法（见4.5节和33.1节）非常重要。不过，因为该指针事实上指向的并不是数组中的任何一个元素，所以不能对它进行读写操作。试图获取和使用数组首元素之前或者尾元素之后的地址都是未定义的行为，应该尽量避免。例如：

```
int* p4 = v-1;  // 首元素之前的位置是未定义的，应该避免使用
int* p5 = v+7;  // 尾元素之后的位置是未定义的，应该避免使用
```

数组名向数组首元素指针的隐式类型转换在C风格代码的函数调用中广泛使用，例如：

```
extern "C" int strlen(const char*);        // 定义在 <string.h> 中

void f()
{
    char v[] = "Annemarie";
    char* p = v;    // char[] 到 char* 的隐式类型转换
    strlen(p);
    strlen(v);      // char[] 到 char* 的隐式类型转换
    v = p;          // 错误：不允许给数组直接赋值
}
```

两次调用传入标准库函数strlen()的值是相同的。唯一的问题是这种隐式类型转换无法避免，注定会发生。换句话说，我们不可能让函数接受整个数组v。不过幸运的是，不存在从指针向数组的显式或隐式类型转换。

数组实参向指针的隐式类型转换意味着数组的大小在调用函数时丢掉了，然而函数经常需要以某种方式得到数组的大小以便执行某些必要的操作。和其他接受字符指针的C标准库函数一样，strlen()也用0作为字符串的结束符，strlen(p)返回的是字符串中除了结束符0之外其他字符的总数。以上提及的都是一些非常底层的细节。标准库vector（见4.4.1节，13.6节和31.4节）、array（见8.2.4节和34.2.1节）和string（见4.2节）不受这些问题困扰。这些标准库类型的元素数量可以通过size()得到，不需要每次都重新计算。

7.4.1　数组漫游

如何便捷高效地访问数组（以及类似的数据结构）是很多算法的关键（见4.5节和第32章）。我们既可以通过指向数组的指针加上一个索引值来访问数组元素，也可以通过直接指向数组元素的指针进行访问。例如：

```
void f(char v[])
{
    for (int i = 0; v[i]!=0; ++i)
        use(v[i]);
}
```

```
void fp(char v[])
{
    for (char* p = v; *p!=0; ++p)
        use(*p);
}
```

前置 * 运算符执行解引用运算，因此 *p 是指针 p 所指的字符，++ 运算令 p 指向数组的下一个元素。

这两个版本的代码不存在谁比谁更快的问题。在现代编译器中，两个例子编译生成的代码应该是一样的（大多数情况下也确实一样）。程序员可以从逻辑性和优美程度出发自由选择。

内置数组的取下标操作是通过组合指针的 + 和 * 两种运算得到的。对于内置数组 a 和数组范围之内的整数 j，有下式成立：

```
a[j] == *(&a[0]+j) == *(a+j) == *(j+a) == j[a]
```

人们常常会纠结于为什么 a[j]==j[a]，比如 3["Texas"]=="Texas"[3]=='a'，其实这种小聪明在实际的代码中并没有多少展示的空间。上面这些等价关系属于非常底层的规则，并不适用于 array 和 vector 等标准库容器。

把 +、-、++、-- 等算术运算符用在指针上得到的结果依赖于指针所指对象的数据类型。当我们对 T* 类型的指针 p 执行算术运算时，p 指向 T 类型的数组元素，p+1 指向数组的下一个元素，p-1 指向上一个元素。上述规则意味着 p+1 对应的整数值比 p 对应的整数值大 sizeof(T)。例如：

```
template<typename T>
int byte_diff(T* p, T* q)
{
    return reinterpret_cast<char*>(q)-reinterpret_cast<char*>(p);
}

void diff_test()
{
    int vi[10];
    short vs[10];
    cout << vi << ' ' << &vi[1] << ' ' << &vi[1]-&vi[0] << ' ' << byte_diff(&vi[0],&vi[1]) << '\n';
    cout << vs << ' ' << &vs[1] << ' ' << &vs[1]-&vs[0] << ' ' << byte_diff(&vs[0],&vs[1]) << '\n';
}
```

这段代码的输出结果是

```
0x7fffaef0 0x7fffaef4 1 4
0x7fffaedc 0x7fffaede 1 2
```

指针值以默认的十六进制形式输出，从上面的结果我们可以知道，在我所用的 C++ 实现版本中 sizeof(short) 是 2，sizeof(int) 是 4。

指针的减法运算只有当参与运算的两个指针指向的是同一个数组中的元素时才有效（尽管 C++ 语言本身并没有一种机制可以快速地检测该条件是否满足）。当计算两个指针 p 和 q 的差值 (q-p) 时，所得结果是序列 [p:q) 中的元素数量（一个整数）。我们可以给指针加上一个整数或者从指针中减去一个整数，得到的结果都是指针。如果该指针指向的位置既不是原数组中的元素，也不是尾后元素，那我们不能使用它，否则会产生未定义的行为。例如：

```
void f()
{
    int v1[10];
    int v2[10];

    int i1 = &v1[5]-&v1[3];    // i1 = 2
    int i2 = &v1[5]-&v2[3];    // 结果是未定义的

    int* p1 = v2+2;            // p1 = &v2[2]
    int* p2 = v2-2;            // *p2 是未定义的
}
```

复杂的指针算术运算通常没什么必要，最好避免使用。此外，直接把两个指针相加没有实际意义，C++ 也不允许这样做。

因为数组的元素数量不一定能与数组本身存储在一起，所以数组不具有自解释性。当我们需要遍历一个数组并且它不像 C 风格的字符串那样具有明确的终结符时，我们必须以某种方式提供元素的数量。例如：

```
void fp(char v[], int size)
{
    for (int i=0; i!=size; ++i)
        use(v[i]);             // 祈祷数组 v 至少包含 size 个元素，否则就会越界
    for (int x : v)
        use(x);                // 错误：范围 for 循环对指针无效

    const int N = 7;
    char v2[N];
    for (int i=0; i!=N; ++i)
        use(v2[i]);
    for (int x : v2)
        use(x);                // 当已知数组的大小时，可以使用范围 for 循环
}
```

数组是一个底层的语言概念，标准库容器 array（见 8.2.4 节和 34.2.1 节）具有内置数组的绝大多数优点，同时规避掉了它的很多缺点。某些 C++ 实现为数组提供了可选的范围检查操作，然而这种范围检查的时空开销可能会非常大，因此大多数时候我们只把它作为开发的辅助工具，而不会真的包含在最终代码中。如果你不打算使用范围检查功能，则一定要设法以一种切实有效的措施确保访问元素不会越界。我的建议是使用 vector 等更高层次的容器类型管理数组，元素的有效范围非常明确，我们一般不会用错。

7.4.2　多维数组

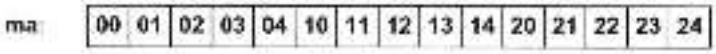
多维数组是指数组的数组。我们可以用下面的语句声明一个 3*5 的数组：

```
int ma[3][5];    // 3 行，每行 5 个 int
```

初始化 ma 的语句是：

```
void init_ma()
{
    for (int i = 0; i!=3; i++)
        for (int j = 0; j!=5; j++)
            ma[i][j] = 10*i+j;
}
```

或者表示成图形的形式：



如果数组 ma 包含 3 行且每行有 5 个 int，则我们可以把它看成是连续 15 个 int。在内存中不存在一个表示矩阵 ma 的单独的对象，我们只存储了数组的元素。维度 3 和 5 只在编译器源代码中有效。当我们编写代码时，必须时刻谨记数组的维度并且在需要使用的时候提供出来。例如，我们用下面的代码输出 ma 的内容：

```
void print_ma()
{
    for (int i = 0; i!=3; i++) {
        for (int j = 0; j!=5; j++)
            cout << ma[i][j] << '\t';
        cout << '\n';
    }
}
```

在别的某些编程语言中，有时候用逗号分隔数组的边界。C++ 不允许这样做，因为逗号（,）是表示序列的运算符（见 10.3.2 节）。好在编译器能捕获大多数此类错误，例如：

```
int bad[3,5];           // 错误：计量表达式中不能使用逗号
int good[3][5];         // 3 行，每行 5 个 int
int ouch = good[1,4];   // 错误：试图用 int* 初始化 int（good[1,4] 的效果是 good[4]，它的类型是指针 int*）
int nice = good[1][4];
```

7.4.3　传递数组

不能以值传递的方式直接把数组传给函数，我们通常传递的是指向数组首元素的指针。例如：

```
void comp(double arg[10])          // arg 的类型是 double*
{
    for (int i=0; i!=10; ++i)
        arg[i]+=99;
}

void f()
{
    double a1[10];
    double a2[5];
    double a3[100];

    comp(a1);
    comp(a2);       // 严重错误！
    comp(a3);       // 只使用了前 10 个元素
};
```

上面的代码看似正确，实则不然。它虽然能通过编译，但是调用 comp(a2) 试图向 a2 的合法边界之外的区域写入内容。此外，如果你期望数组以值传递的方式传给函数，恐怕也要大失所望了：对 arg[i] 执行写操作实际上是直接向 comp() 的实参的元素写内容，而不是工作在该实参的一份副本上。comp() 函数的等价形式是：

```
void comp(double* arg)
{
    for (int i=0; i!=10; ++i)
        arg[i]+=99;
}
```

现在问题更加明显了。当数组作为函数的实参时，我们完全把数组的第一维当成指针使用，而忽略了数组的边界。因此，如果你想在给函数传入一组元素的同时不丢掉数组的大小，就不能使用内置数组类型。你可以把数组放在类中作为类的成员（类似于 std::array），或者直接定义一个句柄类（类似于 std::string 和 std::vector）。

如果苛刻点儿评价的话，使用内置数组有百弊而无一利。当我们需要定义一个接受二维矩阵的函数时，如果编译时知道数组的具体维度当然没有问题：

```
void print_m35(int m[3][5])
{
    for (int i = 0; i!=3; i++) {
        for (int j = 0; j!=5; j++)
            cout << m[i][j] << '\t';
        cout << '\n';
    }
}
```

实参从形式上看虽然是用多维数组表示的矩阵，但实际传入函数的是个指针（而非矩阵的副本，见 7.4 节）。数组的第一个维度与定位元素无关，它只负责指明当前类型（此处是 int[5]）包含几个元素（此处是 3）。例如在前面提到的 ma 中，只要知道第二个维度是 5，我们就能定位任意的 ma[i][5]。此时，可以把数组的第一个维度当成实参传入函数：

```
void print_mi5(int m[][5], int dim1)
{
    for (int i = 0; i!=dim1; i++) {
        for (int j = 0; j!=5; j++)
            cout << m[i][j] << '\t';
        cout << '\n';
    }
}
```

但是当需要传入两个维度时，“显而易见的解决方案”并不有效：

```
void print_mij(int m[][], int dim1, int dim2)    // 预期的结果并不一致
{
    for (int i = 0; i!=dim1; i++) {
        for (int j = 0; j!=dim2; j++)
            cout << m[i][j] << '\t';    // 意料之外的结果!
        cout << '\n';
    }
}
```

好在编译器会因实参声明 m[][] 非法而报错，因为多维数组的第二个维度必须是已知的，这样我们才能准确定位其中的元素。然而，表达式 m[i][j] 会被编译器理解成 *(*(m+i)+j)，尽管这绝非程序员的原意。一种正确的解决方案是：

```
void print_mij(int* m, int dim1, int dim2)
{
    for (int i = 0; i!=dim1; i++) {
        for (int j = 0; j!=dim2; j++)
            cout << m[i*dim2+j] << '\t'; // 有点儿难懂
        cout << '\n';
    }
}
```

其中用来访问数组成员的表达式就是编译器在已知最后一个维度的时候所用的方式。

要想调用该函数，我们只需传入一个代表矩阵的指针即可：

```
int test()
{
    int v[3][5] = {
        {0,1,2,3,4}, {10,11,12,13,14}, {20,21,22,23,24}
    };
    print_m35(v);
    print_mi5(v,3);
    print_mij(&v[0][0],3,5);
}
```

请注意，在最后一个调用中我们使用了 v[0][0]。此处使用 v[0] 也是可以的，因为它与 v[0][0] 等价；但是直接用传入 v 会引发类型错误。这样的代码含义微妙、用法凌乱，还是越少出现越好。如果你必须直接处理多维数组，那么记得把有关的代码封装起来，这样其他程序员在用到你的代码时会容易上手一些。最好的办法是给多维数组提供适当的下标运算符，这样用户就不必被数组中元素的分布情况困扰了（见 29.2.2 节和 40.5.2 节）。

标准库 vector（见 31.4 节）完美地解决了上述问题。

7.5 指针与 const

C++ 提供了两种与“常量”有关的概念：

- constexpr：编译时求值（见 2.2.3 节和 10.4 节）。
- const：在当前作用域内，值不发生改变（见 2.2.3 节）。

基本上，constexpr 的作用是指示或确保在编译时求值，而 const 的主要任务是规定接口的不可修改性。本节主要关注第二点：接口说明。

很多对象的值一旦初始化就不会再改动：

- 使用符号化常量的代码比直接使用字面值常量的代码更易维护。
- 我们经常通过指针读取数据，但是很少通过指针写入数据。
- 绝大多数函数的参数只负责读取数据，很少写入数据。

为了表达一经初始化就不可修改的特性，我们可以在对象的定义中加上 const 关键字。例如：

```
const int model = 90;           // model 是一个 const
const int v[] = { 1, 2, 3, 4 };  // v[] 是一个 const
const int x;                    // 错误：缺少初始化器
```

因为我们无法给 const 对象赋值，所以它必须初始化。

一旦我们把某物声明成 const，就确保它的值在其作用域内不会发生改变：

```
void f()
{
    model = 200;  // 错误
    v[2] = 3;     // 错误
}
```

使用 const 会改变一种类型。所谓改变不是说改变了常量的分配方式，而是限制了它的使用方式。例如：

```
void g(const X* p)
{
    // 此处无法修改 *p
}
```

```
void h()
{
    X val;    // 此处可以修改 val 的值
    g(&val);
    // ...
}
```

一个指针牵扯到两个对象：指针本身以及指针所指的对象。在指针的声明语句中“前置”const 关键字将令所指的对象而非指针本身成为常量。要想令指针本身成为常量，应该用 *const 代替普通的 *。例如：

```
void f1(char* p)
{
    char s[] = "Gorm";

    const char* pc = s;          // 指向常量的指针
    pc[3] = 'g';                 // 错误：pc 指向常量
    pc = p;                      // OK

    char *const cp = s;          // 常量指针
    cp[3] = 'a';                 // OK
    cp = p;                      // 错误：cp 是一个常量

    const char *const cpc = s;   // 指向常量的常量指针
    cpc[3] = 'a';                // 错误：cpc 指向常量
    cpc = p;                     // 错误：cpc 本身是一个常量
}
```

声明运算符 *const 的作用是令指针本身成为常量。不存在形如 const* 的声明运算符。相反，出现在 * 前面的 const 是基本类型的一部分。例如：

```
char *const cp;          // 指向 char 的常量指针
char const* pc;          // 指向常量 const 的指针
const char* pc2;         // 指向常量 char 的指针
```

要想理解上述声明的含义，一个小技巧是按照从右向左的顺序读。例如，“cp 是指向 char 的 const 指针”，而“pc2 是指向 char const 的指针”。

对于同一个对象来说，通过一个指针访问它时是常量并不妨碍在其他情况下它是个变量。这一点在涉及函数的实参时特别有用。我们可以把指针类型的实参声明成 const，这样就能阻止函数修改该指针所指的对象了。例如：

```
const char* strchr(const char* p, char c);    // 找到在字符串 p 中字符 c 第一次出现的位置
char* strchr(char* p, char c);                // 找到在字符串 p 中字符 c 第一次出现的位置
```

第一个函数的参数是常量字符串，函数无权修改其中的元素；它的返回值是指向 const 的指针，也不允许修改其所指的对象。第二个函数没有这些限制。

C++ 允许把非 const 变量的地址赋给指向常量的指针，这不会造成什么不可接受的后果。相反，常量的地址不能被赋给某个不受限的指针，因为如果这样的话，用户有可能通过该指针修改对象的值，这显然是不被允许的。例如：

```
void f4()
{
    int a = 1;
    const int c = 2;
    const int* p1 = &c;  // OK
    const int* p2 = &a;  // OK
```

```
    int* p3 = &c;          // 错误：用 const int* 初始化 int*
    *p3 = 7;               // 试图改变 c 的值
}
```

C++ 允许通过显式类型转换的方式（见 16.2.9 节和 11.5 节）显式地移除掉对于指针指向常量的限制，但是一般情况下不要这么做。

7.6　指针与所有权

资源必须先分配后释放（见 5.2 节）。我们用 new 分配内存，用 delete 释放内存（见 11.2 节）；用 fopen() 打开文件，用 fclose() 关闭文件（见 43.2 节），因此内存和文件都是资源。指针是最常用的资源句柄。这一点不太容易理解，毕竟在程序中指针随处可见，而且作为资源句柄的指针和不作为资源句柄的指针似乎没什么差别。例如：

```
void confused(int* p)
{
    // 释放掉 p?
}

int global {7};

void f()
{
    X* pn = new int{7};
    int i {7};
    int q = &i;
    confused(pn);
    confused(q);
    confused(&global);
}
```

如果在 confused() 中 delete 掉 p，则当执行后面两个调用时程序将发生非常严重的错误，原因是对于不是由 new 分配的对象，我们无权 delete 它（见 11.2 节）。然而，如果 confused() 没有 delete 掉 p，又会造成程序的资源泄露（见 11.2.1 节）。在此例中，很明显应该由 f() 负责管理它创建在自由存储上的对象的生命周期。但是通常情况下，在一个规模较大的程序中我们需要使用某种简单有效的策略追踪和维护那些需要 delete 的资源。

一种比较好的策略是把表示某种所有权的指针全都置于 vector、string 和 unique_ptr 等资源句柄类中。此时，我们就能假定所有不在资源句柄中的指针都不负责管理资源，因此也不必对它们执行 delete 操作。第 13 章将详细讨论资源管理的细节。

7.7　引用

通过使用指针，我们就能以很低的代价在一个范围内传递大量数据，与直接拷贝所有数据不同，我们只需要传递指向这些数据的指针的值就行了。指针的类型决定了我们能对指针所指的对象进行哪些操作。使用指针与使用对象名存在以下差别：

- 语法形式不同，*p 和 p->m 分别取代了 obj 和 obj.m。
- 同一个指针在不同时刻可以指向不同对象。
- 使用指针要比直接使用对象更小心：指针的值可能是 nullptr，也可能指向一个我们并不想要的对象。

这些差别有时候很烦人。例如，程序员常常受困于到底该用 f(&x) 还是 f(x)。更糟糕

的是，程序员必须花费大量精力去管理变化多端的指针变量，而且还得时时防范指针取值为 nullptr 的情况。此外，当我们重载运算符时（比如 +），肯定希望写成 x+y 的形式而不是 &x+&y。解决这些问题的语言机制是使用引用（reference）。和指针类似，引用作为对象的别名存放的也是对象的机器地址。与指针相比，引用不会带来额外的开销。引用与指针的区别主要包括：

- 访问引用与访问对象本身从语法形式上看是一样的。
- 引用所引的永远是一开始初始化的那个对象。
- 不存在“空引用”，我们可以认为引用一定对应着某个对象（见 7.7.4 节）。

引用实际上是对象的别名。引用最重要的用途是作为函数的实参或返回值，此外，它也被用于重载运算符（第 18 章）。例如：

```
template<class T>
class vector {
    T* elem;
    // ...
public:
    T& operator[](int i) { return elem[i]; }                    // 返回元素的引用
    const T& operator[](int i) const { return elem[i]; }        // 返回常量元素的引用

    void push_back(const T& a);                                 // 通过引用传入待添加的元素
    // ...
};

void f(const vector<double>& v)
{
    double d1 = v[1];    // 把 v.operator[](1) 所引的 double 的值拷给 d1
    v[2] = 7;            // 把 7 赋给 v.operator[](2) 所引的 double

    v.push_back(d1);     // 给 push_back() 传入 d1 的引用
}
```

以引用的形式给函数传递实参的思想非常经典，它的历史和高级程序设计语言一样悠久（Fortran 语言最早的版本就用到了这种思想）。

为了体现左值 / 右值以及 const/ 非 const 的区别，存在三种形式的引用：

- *左值引用*（lvalue reference）：引用那些我们希望改变值的对象。
- const 引用（const reference）：引用那些我们不希望改变值的对象（比如常量）。
- *右值引用*（rvalue reference）：所引对象的值在我们使用之后就无须保留了（比如临时变量）。

这三种形式统称为引用，其中前两种形式都是*左值引用*。

7.7.1　左值引用

在类型名字中，符号 X& 的意思是“X 的引用”；它常用于表示左值的引用，因此称为*左值引用*。例如：

```
void f()
{
    int var = 1;
    int& r {var};    // r 和 var 对应同一个 int
    int x = r;       // x 的值变为 1
```

```
    r = 2;          // var 的值变为 2
}
```

为了确保引用对应某个对象（即把它绑定到某个对象），我们必须初始化引用。例如：

```
int var = 1;
int& r1 {var};      // OK：初始化 r1
int& r2;            // 错误：缺少初始化器
extern int& r3;     // OK：r3 在别处初始化
```

初始化引用和给引用赋值是完全不同的操作。除了形式上的区别外，事实上没有专门针对引用的运算符。例如：

```
void g()
{
    int var = 0;
    int& rr {var};
    ++rr;           // var 的值加 1
    int* pp = &rr;  // pp 指向 var
}
```

在这段代码中，++rr 的含义并不是递增引用 rr，相反它的作用是给 rr 所引的 int（即 var）加 1。因此，引用本身的值一旦经过初始化就不能再改变了：它永远都指向一开始指定的对象。我们可以使用 &rr 得到一个指向 rr 所引对象的指针。但是我们既不能令某个指针指向引用，也不能定义引用的数组。从这个意义上来说，引用不是对象。

显然，引用的实现方式应该类似于常量指针，每次使用引用实际上是对该指针执行解引用操作。绝大多数情况下像这样理解引用是没问题的，不过程序员必须谨记：引用不是对象，而指针是一种对象。例如：



有时候编译器能对引用进行优化，使得在运行时无须任何对象表示该引用。

当初始值是左值时（你能获取地址的对象，见 6.4 节），引用的初始化过程没什么特殊之处。提供给“普通”T& 的初始值必须是 T 类型的左值。

const T& 的初始值不一定非得是左值，甚至可以不是 T 类型的。此时：

[1] 首先，如果必要的话先执行目标为 T 的隐式类型转换（见 10.5 节）。

[2] 然后，所得的值置于一个 T 类型的临时变量中。

[3] 最后，把这个临时变量作为初始值。

考虑如下的情况：

```
double& dr = 1;             // 错误：此处需要左值
const double& cdr {1};      // OK
```

后一条语句的初始化过程可以理解为：

```
double temp = double{1};    // 首先用给定的值创建一个临时变量
const double& cdr {temp};   // 然后用这个临时变量作为 cdr 的初始值
```

用于存放引用初始值的临时变量的生命周期从它创建之处开始，到它的引用作用域结束为止。

普通变量的引用和常量的引用必须区分开来。为变量引入一个临时量充满了风险，当我

们为该变量赋值时，实际上是在为一个转瞬即逝的临时量赋值。常量的引用则不存在这一问题，函数的实参经常定义成常量的引用（见 18.2.4 节）。

我们常用引用作为函数的实参类型，这样函数就能修改传入其中的对象的值了。例如：

```
void increment(int& aa)
{
    ++aa;
}

void f()
{
    int x = 1;
    increment(x);        // x = 2
}
```

实参传递在本质上与初始化过程非常相似。因此当调用函数 increment 时，实参 aa 变成了另一个名字 x。从代码的可读性角度出发，尽量避免让函数更改它的实参值，我们可以让函数显式地返回一个值来达到同样的目的：

```
int next(int p) { return p+1; }

void g()
{
    int x = 1;
    increment(x);        // x = 2
    x = next(x);         // x = 3
}
```

函数 increment(x) 从形式上看不出 x 的值已经被改变，相反 x=next(x) 可以。因此，除非函数名字能明显地表达修改实参的意思，否则不要轻易使用“普通”引用。

引用还能作为函数的返回值类型，此时，该函数既能作为赋值运算符的左侧运算对象，也能作为赋值运算符的右侧运算对象。一个典型的示例是如下所示的 Map：

```
template<class K, class V>
class Map {          // 一个简单的示例 map 类
public:
    V& operator[](const K& v);    // 返回与键值 v 对应的值

    pair<K,V>* begin() { return &elem[0]; }
    pair<K,V>* end() { return &elem[0]+elem.size(); }
private:
    vector<pair<K,V>> elem;       // {key,value} 对
};
```

实现标准库 map（见 4.4.3 节和 31.4.3 节）所用的数据结构通常是一棵红黑树。但是为了避开琐碎的细节，我在这里使用最简单的线性搜索实现 Map：

```
template<class K, class V>
V& Map<K,V>::operator[](const K& k)
{
    for (auto& x : elem)
        if (k == x.first)
            return x.second;

    elem.push_back({k,V{}});      // 在末尾添加一对（见 4.4.2 节）
    return elem.back().second;    // 返回新元素的默认值
}
```

我在传递键值实参 k 的时候使用了引用，因为键值的类型可能太大以至于不便拷贝。类似地，返回结果也设为引用类型，因为拷贝函数返回的值也可能代价过于昂贵。之所以把 k 的类型设为 const 引用是因为我不希望函数修改它的值，而且这样做还允许我给函数传入一个字面值常量或者临时对象。因为 Map 的用户几乎肯定会使用和更改找到的值，所以函数的返回值应该是一个非 const 引用。例如：

```
int main()//统计输入流中每个单词出现的次数
{
    Map<string,int> buf;

    for (string s; cin>>s;) ++buf[s];

    for (const auto& x : buf)
        cout << x.first << ": " << x.second << '\n';
}
```

每次执行程序，输入循环负责从标准输入流 cin 读入单词到字符串 s 中（见 4.3.2 节），同时更新该单词对应的计数值。最后，输入的每个单词以及它们各自出现的次数以表格形式输出出来。假设输入是：

```
aa bb bb aa aa bb aa aa
```

则程序的运行结果将是：

```
aa: 5
bb: 3
```

因为我们的 Map 像标准库 map 一样定义了 begin() 和 end()，所以可以使用范围 for 循环遍历其元素。

7.7.2　右值引用

C++ 之所以设计了几种不同形式的引用，是为了支持对象的不同用法：

- 非 const 左值引用所引的对象可以由用户写入内容。
- const 左值引用所引的对象从用户的角度来看是不可修改的。
- 右值引用对应一个临时对象，用户可以修改这个对象（通常确实会修改它），并且认定这个对象以后不会被用到了。

我们最好事先判断引用所引的是否是临时对象，如果是的话，我们就能用比较廉价的移动操作代替昂贵的拷贝操作了（见 3.3.2 节，17.1 节和 17.5.2 节）。对于像 string 和 list 这样的对象来说，它们本身所含的信息量可能非常庞大，但是用于指向这些信息的描述符（比如引用）可能非常小。此时，如果我们确认以后不会再用到该信息，则执行廉价的移动操作是最好的选择。一个典型的例子是，编译器清楚地知道函数返回的局部变量的值不会再被用到了（见 3.3.2 节）。

右值引用可以绑定到右值，但是不能绑定到左值。从这一点上来说，右值引用与左值引用正好相反。例如：

```
string var {"Cambridge"};
string f();

string& r1 {var};          //左值引用，r1 绑定到 var（左值）上
string& r2 {f()};          //左值引用，错误：f() 是右值
string& r3 {"Princeton"};  //左值引用，错误：不允许绑定到临时变量
```

```
string&& rr1 {f()};          // 右值引用，正确：rr1 绑定到一个右值（临时变量）
string&& rr2 {var};          // 右值引用，错误：var 是左值
string&& rr3 {"Oxford"};     // rr3 引用的是一个临时变量，它的内容是 "Oxford"

const string cr1& {"Harvard"};// OK：创建一个临时变量，然后把它绑定到 cr1
```

声明符 && 表示“右值引用”。我们不使用 const 右值引用，因为右值引用的大多数用法都是建立在能够修改所引对象的基础上的。const 左值引用和右值引用都能绑定右值，但是它们的目标完全不同：

- 右值引用实现了一种“破坏性读取”，某些数据本来需要被拷贝，使用右值引用可以优化其性能。
- const 左值引用的作用是保护参数内容不被修改。

右值引用所引对象的使用方式与左值引用所引的对象以及普通变量没什么区别，例如：

```
string f(string&& s)
{
    if (s.size())
        s[0] = toupper(s[0]);
    return s;
}
```

有时，程序员明确知道某一对象不再有用了，但是编译器并不知道这一点。例如：

```
template<class T>
swap(T& a, T& b)          // “旧式的 swap 函数”
{
    T tmp {a};// 此时，我们拥有了两份 a
    a = b;    // 此时，我们拥有了两份 b
    b = tmp;  // 此时，我们拥有了两份 tmp（即 a）
}
```

如果 T 是 string 和 vector 等拷贝操作非常昂贵的类型，则上面这个 swap() 函数会非常昂贵。注意一个事实：其实我们根本没打算拷贝什么东西，我们想要的只是在 a、b 和 tmp 间移动数据而已。我们可以告诉编译器我们的初衷：

```
template<class T>
void swap(T& a, T& b)     // “（几乎）完美的 swap 函数”
{
    T tmp {static_cast<T&&>(a)}; // 初始化的同时对 a 写操作
    a = static_cast<T&&>(b);     // 赋值的同时对 b 写操作
    b = static_cast<T&&>(tmp);   // 赋值的同时对 tmp 写操作
}
```

static_cast<T&&>(x) 的结果值是 T&& 类型的右值引用，引用的对象是 x。现在我们可以把右值引用的优化操作用在 x 上了。当类型 T 含有移动构造函数（见 3.3.2 节和 17.5.2 节）或者移动赋值运算符时，上述优化操作将发挥作用。以 vector 为例：

```
template<class T> class vector {
    // ...
    vector(const vector& r); // 拷贝构造函数（拷贝 r 的表示）
    vector(vector&& r);      // 移动构造函数（“窃取” r 的表示）
};

vector<string> s;
vector<string> s2 {s};              // s 是左值，使用拷贝构造函数
vector<string> s3 {s+"tail"};       // s+"tail" 是右值，使用移动构造函数
```

在 swap() 函数中使用 static_cast 显得有点累赘，程序员有时还可能拼错，因此标准库提供了一个名为 move() 的函数：move(x) 等价于 static_cast<X&&>(x)，其中 x 的类型是 X。通过使用 move()，我们就能让 swap() 的形式变得清晰简洁：

```
template<class T>
void swap(T& a, T& b)    // "(几乎) 完美的 swap 函数"
{
    T tmp {move(a)};     // 从 a 中移出值
    a = move(b);         // 从 b 中移出值
    b = move(tmp);       // 从 tmp 中移出值
}
```

与最初的 swap() 相比，最新版本无须执行任何拷贝操作，它使用移动操作完成所需的功能。

因为 move(x) 实际上并不真的移动 x（它只是为 x 创建了一个右值引用），所以其实给它起名 rval() 的话更贴切，不过 move() 的名字已经使用了太长时间，程序员已经习惯了。

我之所以认为这个 swap() 是“几乎完美的”，是因为它只能交换左值。例如：

```
void f(vector<int>& v)
{
    swap(v,vector<int>{1,2,3});    // 用 1,2,3 替换 v 的元素
    // ...
}
```

有的时候人们确实需要用一组有序默认值替换容器的当前内容，但是上面的 swap() 函数做不到。一种解决方案是再增加两个重载函数：

```
template<class T> void swap(T&& a, T& b);
template<class T> void swap(T& a, T&& b)
```

最后一个版本可以满足我们的要求。标准库为 string 和 vector 等类型（见 31.3.3 节）提供了 shrink_to_fit() 和 clear()，以使得 swap() 可以处理右值参数：

```
void f(string& s, vector<int>& v)
{
    s.shrink_to_fit();          // 令 s.capacity()==s.size()
    swap(s,string{s});          // 令 s.capacity()==s.size()
    v.clear();                  // 清空 v
    swap(v,vector<int>{});      // 清空 v
    v = {};                     // 清空 v
}
```

右值引用还可用于实参转发（见 23.5.2.1 节和 35.5.1 节）。

所有标准库容器都提供了移动构造函数和移动赋值运算符（见 31.3.2 节）。它们用于插入新元素的操作，比如 insert() 和 push_back()，都提供了接受右值引用的版本。

7.7.3　引用的引用

如果你让引用指向某类型的引用，那么你得到的还是该类型的引用，即非特殊的引用的引用类型。但你得到的到底是哪种引用呢，左值引用还是右值引用？考虑如下情况：

```
using rr_i = int&&;
using lr_i = int&;
using rr_rr_i = rr_i&&;    // "int && &&" 的类型是 int&&
using lr_rr_i = rr_i&;     // "int && &" 的类型是 int&
using rr_lr_i = lr_i&&;    // "int & &&" 的类型是 int&
using lr_lr_i = lr_i&;     // "int & &" 的类型是 int&
```

总之，永远是左值引用优先。这种规定合情合理：不管我们怎么做都无法改变左值引用绑定左值的事实。有时候，我们把这种现象称为引用合并（reference collapse）。

C++ 不允许下面的语法形式：

```
int && & r = i;
```

引用的引用只能作为别名（见 3.4.5 节和 6.5 节）的结果或者模板类型的参数（见 23.5.2.1 节）。

7.7.4　指针与引用

指针和引用是两种无须拷贝就能在别处使用对象的机制。它们的图形化表示如下所示：



指针和引用各有优势，也都存在不足之处。

如果你需要更换所指的对象，应该使用指针。你可以用 =、+=、-=、++ 和 -- 改变指针变量的值（见 11.1.4 节）。例如：

```
void fp(char* p)
{
    while (*p)
        cout << ++*p;
}

void fr(char& r)
{
    while (r)
        cout << ++r;    // 哎呀：增加的是所引用的 char 的值，而非引用本身
                        // 很可能是个死循环！
}

void fr2(char& r)
{
    char* p = &r;       // 得到一个指向所引用对象的指针
    while (*p)
        cout << ++*p;
}
```

反之，如果你想让某个名字永远对应同一个对象，应该使用引用。例如：

```
template<class T> class Proxy {         // Proxy 引用初始化它的那个对象
    T& m;
public:
    Proxy(T& mm) :m{mm} {}
    // ...
};

template<class T> class Handle {    // Handle 引用当前对象
    T* m;
public:
    Proxy(T* mm) :m{mm} {}
    void rebind(T* mm) { m = mm; }
    // ...
};
```

如果你想自定义（重载）一个运算符（见 18.1 节），使之用于指向对象的某物，应该使用引用。例如：

```
Matrix operator+(const Matrix&, const Matrix&);    // OK
Matrix operator-(const Matrix*, const Matrix*);    // 错误：不是用户自定义类型参数

Matrix y, z;
// ...
Matrix x = y+z;        // OK
Matrix x2 = &y-&z;  // 容易且存在错误
```

C++ 不允许重新定义指针等内置类型的运算符含义（见 18.2.3 节）。

如果你想让一个集合中的元素指向对象，应该使用指针：

```
int x, y;
string& a1[] = {x, y};                  // 错误：引用的数组
string* a2[] = {&x, &y};                // OK
vector<string&> s1 = {x , y};           // 错误：引用的向量
vector<string*> s2 = {&x, &y};          // OK
```

除非 C++ 对于某些情况做出了明确的规定，我们不得不照做；其他大多数时候，程序员有权在指针和引用中进行选择，这个过程有点像艺术创作：需要点儿智慧，也需要点儿美感。理论上，我们应该尽量减少错误的风险，并且增加代码的可读性。

如果你需要表示“值空缺”，则应该使用指针。指针提供了 nullptr 作为“空指针”，但是并没有“空引用”与之对应。例如：

```
void fp(X* p)
{
    if (p == nullptr) {
        // 指针的值为空
    }
    else {
        // 使用 *p
    }
}

void fr(X& r)    // 常规形式
{
    // 假定 r 合法，然后使用它
}
```

当确实需要的时候，也可以为特定的类型构造一个“空引用”。例如：

```
void fr2(X& r)
{
    if (&r == &nullX) {    // 或者是 r==nullX
        // 引用为空
    }
    else {
        // 使用 r
    }
}
```

显然，你需要让 nullX 有良好的定义。但不管怎么说，这种用法并不符合语言习惯，我不建议程序员使用。默认情况下，程序员可以认定他所使用的引用是有效的。除非有人故意创建一个无效的引用，否则这种情况很难遇到。例如：

```
char* ident(char * p) { return p; }

char& r {*ident(nullptr)}; // 无效代码
```

这是无效的 C++ 代码。即使你的编程环境暂时没有发现，也最好不要这样写。

7.8 建议

［1］ 使用指针时越简单直接越好；7.4.1 节。
［2］ 不要对指针执行稀奇古怪的算术运算；7.4 节。
［3］ 注意不要越界访问数组，尤其不要在数组之外的区域写入内容；7.4.1 节。
［4］ 不要使用多维数组，用合适的容器替代它；7.4.2 节。
［5］ 用 nullptr 代替 0 和 NULL；7.2.2 节。
［6］ 与内置的 C 风格数组相比，优先选用容器（比如 vector、array 和 valarray）；7.4.1 节。
［7］ 优先选用 string，而不是以 0 结尾的 char 数组；7.4 节。
［8］ 如果字符串字面值常量中包含太多反斜线，则使用原始字符串；7.3.2.1 节。
［9］ const 引用比普通引用更适合作为函数的实参；7.7.3 节。
［10］ 只有当需要转发和移动时才使用右值引用；7.7.2 节。
［11］ 让表示所有权的指针位于句柄类的内部；7.6 节。
［12］ 在底层代码之外尽量不要使用 void*；7.2.1 节。
［13］ 用 const 指针和 const 引用表示接口中不允许修改的部分；7.5 节。
［14］ 引用比指针更适合作为函数的实参，不过当需要处理“对象缺失”的情况时例外；7.7.4 节。

结构、联合与枚举

> 想造一个更完美的 Union[⊖]。
>
> ——人们的呼声

- 引言
- 结构
  struct 的布局；struct 的名字；结构与类；结构与数组；类型等价；普通旧数据；域
- 联合
  联合与类；匿名 union
- 枚举
  enum class；普通的 enum；未命名的 enum
- 建议

8.1 引言

用户自定义类型是能否有效使用 C++ 的关键，本章介绍三种用户自定义类型的初级形式：

- struct（结构）是由任意类型元素（即成员，member）构成的序列。
- union 是一种 struct，同一时刻只保存一个元素的值。
- enum（枚举）是包含一组命名常量（称为枚举值）的类型。
- enum class（限定作用域的枚举类型）是一种 enum，枚举值位于枚举类型的作用域内，不存在向其他类型的隐式类型转换。

这些类型在 C++ 的早期版本中就已经存在了，它们主要关注数据如何表示的问题，构成了大多数 C 程序的基本框架。这里描述的 struct 其实是一种简单的 class（见 3.2 节和第 16 章）。

8.2 结构

数组是相同类型元素的集合。相反，struct 是任意类型元素的集合。例如：

```
struct Address {
    const char* name;       // "Jim Dandy"
    int number;             // 61
    const char* street;     // "South St"
    const char* town;       // "New Providence"
    char state[2];          // 'N' 'J'
    const char* zip;        // "07974"
};
```

⊖ "We the people, in order to form a more perfect union." 是美国马其次演讲的主题。"union" 在原语境中的含义是 "合众国"。作者像变魔术，巧妙地引用了这句话，但其实 "union" 却是指 C++ 的语法概念 "联合"。——译者注

该结构定义了一个名为 Address 的类型，它包含给身处美国的某人发信所需的全部信息。注意不要忽略结构最后的分号。

声明 Address 类型的变量与声明其他类型变量的方式一模一样。我们可以用点运算符（.）为每个成员分别赋值。例如：

```
void f()
{
    Address jd;
    jd.name = "Jim Dandy";
    jd.number = 61;
}
```

struct 类型的变量能使用 {} 的形式初始化（见 6.3.5 节）。例如：

```
Address jd = {
    "Jim Dandy",
    61, "South St",
    "New Providence",
    {'N','J'}, "07974"
};
```

请注意，我们不能用字符串 "NJ" 初始化 jd.state。字符串以符号 '\0' 结尾，因此 "NJ" 实际上包含 3 个字符，比 jd.state 需要的多出了一个。在这里，我故意将结构的成员定义成底层数据类型，这样读者就能对如何定义结构以及可能面临哪些问题有切身体会了。

我们通常使用 -> 运算符（struct 指针解引用）访问结构的内容，例如：

```
void print_addr(Address* p)
{
    cout << p->name << '\n'
         << p->number << ' ' << p->street << '\n'
         << p->town << '\n'
         << p->state[0] << p->state[1] << ' ' << p->zip << '\n';
}
```

如果 p 是一个指针，则 p->m 等价于 (*p).m。

除此之外，我们也能以引用的方式传递 struct，并且使用 . 运算符（struct 成员访问）访问它：

```
void print_addr2(const Address& r)
{
    cout << r.name << '\n'
         << r.number << ' ' << r.street << '\n'
         << r.town << '\n'
         << r.state[0] << r.state[1] << ' ' << r.zip << '\n';
}
```

关于实参传递的内容将在 12.2 节详细讨论。

结构类型的对象可以被赋值、作为实参传入函数，或者作为函数的结果返回。例如：

```
Address current;

Address set_current(Address next)
{
    address prev = current;
    current = next;
    return prev;
}
```

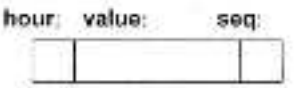
默认情况下，比较运算符（== 和 !=）等一些似是而非的操作并不适用于结构类型。当然，用户有权自定义这些运算符（见 3.2.1.1 节和第 18 章）。

8.2.1　struct 的布局

在 struct 的对象中，成员按照声明的顺序依次存放。例如，我们存储一个简单的读取器的方式可能如下所示：

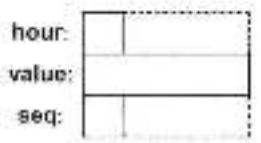
```
struct Readout {
    char hour;      // [0:23]
    int value;
    char seq;       // 序列标识 ['a':'z']
};
```

你可以设想 Readout 对象的成员在内存中的分布情况形如下图所示：



在内存中为成员分配空间时，顺序与声明结构的时候保持一致。因此，hour 的地址一定在 value 的地址之前，见 8.2.6 节。

然而，一个 struct 对象的大小不一定恰好等于它所有元素大小的累积之和。因为很多机器要求一些特定类型的对象沿着系统结构设定的边界分配空间，以便机器能高效地处理这些对象。例如，整数通常沿着字的边界分配空间。在这类机器上，我们说对象对齐（aligned，见 6.2.9 节）得很好。这种做法会导致在结构中存在“空洞”。在 4 字节 int 的机器上，Readout 的布局很可能是：

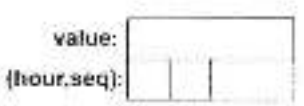


在预测 Readout 所占的空间大小时，很多人简单地把每个成员的尺寸加在一起，得到结果是 6。其实，在此例中 sizeof(Readout) 真正的结果是 12，在很多机器上都是如此。

你也可以把成员按照各自的尺寸排序（大的在前），这样能在一定程度上减少空间浪费。例如：

```
struct Readout {
    int value;
    char hour;      // [0:23]
    char seq;       // 序列标识 ['a':'z']
};
```

此时，Readout 的存储方式是：



在 Readout 中仍然包含一个 2 字节的“空洞”（未使用空间），sizeof(Readout)==8。这一点也不奇怪，毕竟当我们将来把两个 Readout 对象放在一起时（构成数组），肯定也希望它们

是对齐的。包含 10 个 Readout 对象的数组大小是 10*sizeof(Readout)。

通常情况下，我们应该从可读性的角度出发设计结构成员的顺序。只有当需要优化程序的性能时，才按照成员的大小排序。

在结构中如果用到了多个访问修饰符（即，public、private 和 protected），有可能影响布局（见 20.5 节）。

8.2.2　struct 的名字

类型名字只要一出现就能马上使用了，无须等到该类型的声明全部完成。例如：

```
struct Link {
    Link* previous;
    Link* successor;
};
```

但是，只有等到 struct 的声明全部完成，才能声明它的对象。例如：

```
struct No_good {
    No_good member;  // 错误：递归定义
};
```

因为编译器无法确定 No_good 的大小，所以程序会报错。要想让两个或更多 struct 互相引用，必须提前声明好 struct 的名字。例如：

```
struct List;          // 结构名字声明：稍后再定义 List

struct Link {
    Link* pre;
    Link* suc;
    List* member_of;
    int data;
};

struct List {
    Link* head;
};
```

如果没有一开始声明 List，则在稍后声明 Link 时使用 List* 类型的指针将造成错误。

我们可以在真正定义一个 struct 类型之前就使用它的名字，只要在此过程中不使用成员的名字和结构的大小就行了。然而，直到 struct 的声明全部完成之前，它都是一个不完整的类型。例如：

```
struct S;  // "S" 是类型的名字

extern S a;
S f();
void g(S);
S* h(S*);
```

我们必须先定义 S 才能继续使用上面这些声明：

```
void k(S* p)
{
    S a;          // 错误：还没有定义 S，分配空间需要用到 S 的尺寸

    f();          // 错误：还没有定义 S，返回值需要用到 S 的尺寸
    g(a);         // 错误：还没有定义 S，传递实参需要用到 S 的尺寸
```

```
    p->m = 7;       // 错误：还没有定义 S，成员名字未知

    S* q = h(p);    // ok：允许分配和传递指针
    q->m = 7;       // 错误：还没有定义 S，成员名字未知
}
```

为了符合 C 语言早期的规定，C++ 允许在同一个作用域中分别声明一对同名的 struct 和非 struct。例如：

```
struct stat { /* ... */ };
int stat(char* name, struct stat* buf);
```

此时，普通的名字（stat）默认是非 struct 的名字，要想表示 struct 必须在 stat 前加上前缀 struct。类似地，我们还可以让关键字 class、union（见 8.3 节）和 enum（见 8.4 节）作为名字的前缀以避免二义性。我的建议是程序员应该尽量避免使用这种同名实体。

8.2.3　结构与类

struct 是一种 class，它的成员默认是 public 的。struct 可以包含成员函数（见 2.3.2 节和第 16 章），尤其是构造函数。例如：

```
struct Points {
    vector<Point> elem;// 必须至少包含一个 Point
    Points(Point p0) { elem.push_back(p0);}
    Points(Point p0, Point p1) { elem.push_back(p0); elem.push_back(p1); }
    // ...
};

Points x0;                          // 错误：缺少默认构造函数
Points x1{ {100,200} };             // 一个 Point
Points x1{ {100,200}, {300,400} };  // 两个 Point
```

如果你只想按照默认的顺序初始化结构的成员，则不需要专门定义一个构造函数，例如：

```
struct Point {
    int x, y;
};

Point p0;       // 危险的行为：如果位于局部作用域中，则 p0 未初始化（见 6.3.5.1 节）
Point p1 {};    // 以默认方式构造：{{},{}}；即 {0,0}
Point p2 {1};   // 以默认方式构造第二个成员：{1,{}}；即 {1,0}
Point p3 {1,2}; // {1,2}
```

但是如果你需要改变实参的顺序、检验实参的有效性、修改实参或者建立不变式（见 2.4.3.2 节和 13.4 节），则应该编写一个专门的构造函数。例如：

```
struct Address {
    string name;        // "Jim Dandy"
    int number;         // 61
    string street;      // "South St"
    string town;        // "New Providence"
    char state[2];      // 'N' 'J'
    char zip[5];        // 07974

    Address(const string n, int nu, const string& s, const string& t, const string& st, int z);
};
```

我添加了一个构造函数以确保每个成员都被初始化。同时，我能够输入 string 和 int 作为邮

政编码，而不必再伪造一些字符。例如：

```
Address jd = {
    "Jim Dandy",
    61, "South St",
    "New Providence",
    "NJ", 7974              // 07974 是八进制数值（见 6.2.4.1 节）
};
```

Address 的构造函数可以定义成下面所示的形式：

```
Address::Address(const string& n, int nu, const string& s, const string& t, const string& st, int z)
                    // 检验邮政编码的有效性
    :name{n},
     number{nu},
     street{s},
     town{t}
{
    if (st.size()!=2)
        error("State abbreviation should be two characters")
    state = {st[0],st[1]};      // 把邮政编码的简称存成字符
    ostringstream ost;          // 输出字符串流，见 38.4.2 节
    ost << z;                   // 从 int 中抽取字符
    string zi {ost.str()};
    switch (zi.size()) {
    case 5:
        zip = {zi[0], zi[1], zi[2], zi[3], zi[4]};
        break;
    case 4:   // 以 '0' 开始
        zip = {'0', zi[0], zi[1], zi[2], zi[3]};
        break;
    default:
        error("unexpected ZIP code format");
    }
    // ... 检查编码是否是有意义的 ...
}
```

8.2.4　结构与数组

很自然地，我们可以构建 struct 的数组，也可以让 struct 包含数组。例如：

```
struct Point {
    int x,y
};

Point points[3] {{1,2},{3,4},{5,6}};
int x2 = points[2].x;

struct Array {
    Point elem[3];
};

Array points2 {{1,2},{3,4},{5,6}};
int y2 = points2.elem[2].y;
```

把内置数组置于 struct 的内部意味着我们可以把该数组当成一个对象来使用：我们可以在初始化（包括函数传参及函数返回）和赋值时直接拷贝 struct。例如：

```
Array shift(Array a, Point p)
{
    for (int i=0; i!=3; ++i) {
        a.elem[i].x += p.x;
        a.elem[i].y += p.y;
    }
    return a;
}

Array ax = shift(points2,{10,20});
```

Array 的定义还很初级，存在很多问题：为什么 i!=3？为什么反复使用 .elem[i]？为什么只有 Point 类型的元素？标准库在固定尺寸的数组的基础上做了进一步的提升，设计了 std::array（见 34.2.1 节）。标准库 array 是一种 struct，与内置数组相比，它更完善，设计思想也更巧妙：

```
template<typename T, size_t N >
struct array {    // 简化版本（见 34.2.1 节）
    T elem[N];

    T* begin() noexcept { return elem; }
    const T* begin() const noexcept {return elem; }
    T* end() noexcept { return elem+N; }
    const T* end() const noexcept { return elem+N; }

    constexpr size_t size() noexcept;

    T& operator[](size_t n) { return elem[n]; }
    const T& operator[](size_type n) const { return elem[n]; }

    T* data() noexcept { return elem; }
    const T* data() const noexcept { return elem; }

    // ...
};
```

这里的 array 是个模板，它可以存放任意数量、任意类型的元素。它还可以直接处理异常（见 13.5.1.1 节）和 const 对象（见 16.2.9.1 节）。我们基于 array 继续编写下面的程序：

```
struct Point {
    int x,y
};

using Array = array<Point,3>; // 包含 3 个 Point 的 array

Array points {{1,2},{3,4},{5,6}};
int x2 = points[2].x;
int y2 = points[2].y;

Array shift(Array a, Point p)
{
    for (int i=0; i!=a.size(); ++i) {
        a[i].x += p.x;
        a[i].y += p.y;
    }
    return a;
}
```

```
Array ax = shift(points,{10,20});
```

与内置数组相比，std::array 有两个明显的优势：首先它是一种真正的对象类型（可以执行赋值操作），其次它不会隐式地转换成指向元素的指针：

```
ostream& operator<<(ostream& os, Point p)
{
    cout << '{' << p[i].x << ',' << p[i].y << '}';
}

void print(Point a[],int s) // 必须指定元素的数量
{
    for (int i=0; i!=s; ++i)
        cout << a[i] << '\n';
}

template<typename T, int N>
void print(array<T,N>& a)
{
    for (int i=0; i!=a.size(); ++i)
        cout << a[i] << '\n';
}

Point point1[] = {{1,2},{3,4},{5,6}};          // 3 个元素
array<Point,3> point2 = {{1,2},{3,4},{5,6}};   // 3 个元素

void f()
{
    print(point1,4);    // 4 是一个糟糕的错误
    print(point2);
}
```

std::array 也有不足，我们无法从初始化器的长度推断元素的数量：

```
Point point1[] = {{1,2},{3,4},{5,6}};          // 3 个元素
array<Point,3> point2 = {{1,2},{3,4},{5,6}};   // 3 个元素
array<Point> point3 = {{1,2},{3,4},{5,6}};     // 错误：未指定元素的数量
```

8.2.5　类型等价

对于两个 struct 来说，即使它们的成员相同，它们本身仍是不同的类型。例如：

```
struct S1 { int a; };
struct S2 { int a; };
```

S1 和 S2 是两种类型，因此：

```
S1 x;
S2 y = x; // 错误：类型不匹配
```

struct 本身的类型与其成员的类型不能混为一谈，例如：

```
S1 x;
int i = x;  // 错误：类型不匹配
```

在程序中，每个 struct 只能有唯一的定义（见 15.2.3 节）。

8.2.6　普通旧数据

有时候，我们只想把对象当成“普通旧数据”（内存中的连续字节序列）而不愿考虑那些

高级语义概念，比如运行时多态（见 3.2.3 节和 20.3.2 节），用户自定义的拷贝语义（见 3.3 节和 17.5 节）等。这么做的主要动机是在硬件条件允许的范围内尽可能高效地移动对象。例如，要执行拷贝含有 100 个元素的数组的任务，调用 100 次拷贝构造函数显然不像直接调用 std::memcpy() 有效率，毕竟后者只需要使用一个块移动机器指令即可。即使构造函数是内联的，对于优化器来说要想发现这样的优化机会也并不容易。这种“小把戏”在实现 vector 等容器以及底层 I/O 程序时都很常见，并且非常重要。但是在高层代码中就没什么必要了，应该尽量避免使用。

POD（“普通旧数据”）是指能被“仅当作数据”处理的对象，程序员无须顾及类布局的复杂性以及用户自定义的构造、拷贝和移动语义。例如：

```
struct S0 { };                                          // 是 POD
struct S1 { int a; };                                   // 是 POD
struct S2 { int a; S2(int aa) : a(aa) { } };            // 不是 POD（不是默认构造函数）
struct S3 { int a; S3(int aa) : a(aa) { } S3() {} };    // 是 POD（用户自定义的默认构造函数）
struct S4 { int a; S4(int aa) : a(aa) { } S4() = default; };  // 是 POD
struct S5 { virtual void f(); /* ... */ };              // 不是 POD（含有一个虚函数）

struct S6 : S1 { };             // 是 POD
struct S7 : S0 { int b; };      // 是 POD
struct S8 : S1 { int b; };      // 不是 POD（数据成员于 S1 也属于 S8）
struct S9 : S0, S1 {};          // 是 POD
```

我们如果想把某个对象“仅当作数据”处理（当作 POD），则要求该对象必须满足下述条件：

- 不具有复杂的布局（比如含有 vptr，见 3.2.3 节和 20.3.2 节）。
- 不具有非标准（用户自定义的）拷贝语义。
- 含有一个最普通的默认构造函数。

显然，我们在定义 POD 时必须慎之又慎，从而确保在不破坏任何语言规则的前提下使用这些优化措施。正式的规定是（§ iso.3.9，§ iso.9）：POD 必须是属于下列类型的对象：

- 标准布局类型（standard layout type）
- 平凡可拷贝类型（trivially copyable type）
- 具有平凡默认构造函数的类型

一个与之有关的概念是平凡类型（trivial type），它具有以下属性：

- 一个平凡默认构造函数
- 平凡拷贝和移动操作

通俗地说，当一个默认构造函数无须执行任何实际操作时（如果需要定义一个默认构造函数，使用 =default，见 17.6.1 节），我们认为它是平凡的。

一个类型具有标准布局，除非它：

- 含有一个非标准布局的非 static 成员或基类；
- 包含 virtual 函数（见 3.2.3 节和 20.3.2 节）；
- 包含 virtual 基类（见 21.3.5 节）；
- 含有引用类型（见 7.7 节）的成员；
- 其中的非静态数据成员有多种访问修饰符（见 20.5 节）；
- 阻止了重要的布局优化：
  - 在多个基类中都含有非 static 数据成员，或者在派生类和基类中都含有非 static 数据成员，或者

- 基类类型与第一个非 static 数据成员的类型相同。

基本上，标准布局类型是指与 C 语言的布局兼容的类型，并且应该能被常规的 C++ 应用程序二进制接口（ABI）处理。

除非在类型内部含有非平凡的拷贝操作、移动操作或者析构函数（见 3.2.1.2 节和 17.6 节），否则该类型就是平凡可拷贝的类型。通俗地说，如果一个拷贝操作能被实现成逐位拷贝的形式，则它是平凡的。那么，是什么原因让拷贝、移动和析构函数变得不平凡呢？

- 这些操作是用户定义的。
- 这些操作所属的类含有 virtual 函数。
- 这些操作所属的类含有 virtual 基类。
- 这些操作所属的类含有非平凡的基类或者成员。

内置类型的变量都是平凡可拷贝的，且拥有标准布局。同样，由平凡可拷贝对象组成的数组是平凡可拷贝的，由标准布局对象组成的数组拥有标准布局。思考下面的例子：

```
template<typename T>
    void mycopy(T* to, const T* from, int count);
```

已知 T 是 POD，我们希望对它进行优化。一种优化的思路是只对 POD 调用 mycopy() 函数，但是这么做充满了风险：如果使用了 mycopy()，你能确保代码的用户永远不会对非 POD 调用该函数吗？现实情况是，谁也无法做出这种保证。另一种可行的措施是调用 std::copy()，标准库在实现该函数时应该已经进行了必要的优化。无论如何，下面所示的是经过优化后的代码：

```
template<typename T>
void mycopy(T* to, const T* from, int count)
{
    if (is_pod<T>::value)
        memcpy(to,from,count*sizeof(T));
    else
        for (int i=0; i!=count; ++i)
            to[i]=from[i];
}
```

is_pod 是一个标准库类型属性谓词（见 35.4.1 节），它定义在 <type_traits> 中，我们可以通过它在代码中提问："T 是 POD 吗？"程序员能从 is_pod<T> 中受益良多，尤其是不必再记忆那些判断 T 是否是 POD 所需的繁琐规则。

请注意，增加或删除非默认构造函数不会影响布局和性能（在 C++98 标准中可不是这样）。

如果你确实对 C++ 语言的深层次内容有非常浓厚的兴趣，不妨花点时间研究一下 C++ 标准中对布局和平凡性概念的规定（§ iso.3.9，§ iso.9），并且思考这些规定是如何影响程序员和编译器作者的。当然，思考这些问题可能需要占用你的很多时间，坚持下去，别轻易放弃。

8.2.7　域

看起来用一整个字节（一个 char 或者一个 bool）表示一个二元变量（比如 on/off 开关）有点浪费，但是 char 已经是 C++ 中能独立分配和寻址的最小对象了（见 7.2 节）。我们也可以把这些微小的变量组织在一起作为 struct 的域（field）。域也称为位域（bit-field）。我们只

要指定成员所占的位数，就能把它定义成域了。C++ 允许未命名的域。未命名的域不会干扰命名域的含义，同时能以某种依赖于机器的方式优化布局：

```
struct PPN {          // R6000 物理页编号
    unsigned int PFN : 22;   // 页框编号
    int : 3;                 // 未使用
    unsigned int CCA : 3;          // 缓存一致性算法
    bool nonreachable : 1;
    bool dirty : 1;
    bool valid : 1;
    bool global : 1;
};
```

这个例子还展示了域的另外一个作用：即，为外部设定的布局中的部件命名。域必须是整型或枚举类型（见 6.2.1 节）。我们无法获取域的地址，除此之外，域的用法和其他变量一样。我们能用一个单独的二进制位表示 bool 域，在操作系统内核及调试器中，类型 PPN 的用法与之类似：

```
void part_of_VM_system(PPN* p)
{
    // ...
    if (p->dirty) {  // 更改了内容
        // 拷贝至磁盘
        p->dirty = 0;
    }
}
```

出乎人们意料之外的事实是，用域把几个变量打包在一个字节内并不一定能节省空间。这种做法虽然节省了数据空间，但是负责管理和操作这些变量的代码在绝大多数机器上都会更长。经验表明，当二进制变量的存储方式从位域变成字符时，程序的规模会显著缩小！同时，直接访问 char 或者 int 也比访问位域更快。位域不过是用位逻辑运算符（见 11.1.1 节）从字中提取信息或插入信息的一种便捷手段罢了。

8.3　联合

union 是一种特殊的 struct，它的所有成员都分配在同一个地址空间上。因此，一个 union 实际占用的空间大小与其最大的成员一样。自然地，在同一时刻 union 只能保存一个成员的值。例如，考虑一个符号表项存放名字和值对的情况：

```
enum Type { str, num };

struct Entry {
    char* name;
    Type t;
    char* s;  // 如果 t==str，使用 s
    int i;    // 如果 t==num，使用 i
};

void f(Entry* p)
{
    if (p->t == str)
        cout << p->s;
    // ...
}
```

在这个例子中，成员 s 和 i 永远不会被同时使用，因此空间被浪费了。我们可以把它们指定成 union 的成员，以解决上述问题：

```
union Value {
    char* s;
    int i;
};
```

语言本身并不负责追踪和管理 union 到底存的是哪种值，这是程序员的责任：

```
struct Entry {
    char* name;
    Type t;
    Value v;  // 如果 t==str，使用 v.s；如果 t==num，使用 v.i
};

void f(Entry* p)
{
    if (p->t == str)
        cout << p->v.s;
    // ...
}
```

为了避免可能出现的错误，程序员最好把 union 封装起来，从而确保访问 union 成员的方式与该成员的类型永远保持一致（见 8.3.2 节）。

联合有时候会被误用于“类型转换”的目的，这种误用的情况常常发生在一些特定的程序员身上，他们曾经使用的编程语言缺少显式类型转换的功能，因此不得不采用这种方式。例如，下面所示的 int 向 int* 的“转换”以这两种类型逐位等价为前提：

```
union Fudge {
    int i;
    int* p;
};

int* cheat(int i)
{
    Fudge a;
    a.i = i;
    return a.p;     // 错误的用法
}
```

这根本算不上是一种类型转换。在一些机器环境中，int 和 int* 占用的空间大小并不一样；而在另外一些机器中，整数的地址不能是奇数。因此，像这样使用 union 不但危险，而且无法移植。如果你确实需要类似的转换，最好使用显式类型转换符（见 11.5.2 节），这样读者就能清楚地知道到底发生了什么。例如：

```
int* cheat2(int i)
{
    return reinterpret_cast<int*>(i);    // 虽然这个转换本身既不美观，又很容易出错
}
```

无论如何，在上面的代码中，如果转换前后对象的尺寸不一致，那么编译器至少有机会给出报错信息；它比使用 union 的版本强多了。

使用 union 的目的无非是让数据更紧密，从而提高程序的性能。然而，大多数程序即使用了 union 也不会提高太多；同时，使用 union 的代码更容易出错。因此，我认为 union 是

一种被过度使用的语言特性，最好不要出现在你的程序中。

8.3.1　联合与类

在很多非平凡的 union 中，存在一个不太常用的成员，它的尺寸比其他常用成员的尺寸都大得多。因为 union 的尺寸与它最大的成员一样大，所以不可避免地存在空间浪费的情况。我们可以使用一组派生类（见 3.2.2 节和第 20 章）代替 union，从而避免空间的浪费。

从技术上来说，union 是一种特殊的 struct（见 8.2 节），而 struct 是一种特殊的 class（第 16 章）。然而，很多提供给类的功能与联合无关，因此对 union 施加了一些限制：

[1] union 不能含有虚函数。

[2] union 不能含有引用类型的成员。

[3] union 不能含有基类。

[4] 如果 union 的成员含有用户自定义的构造函数、拷贝操作、移动操作或者析构函数，则此类函数对于 union 来说被 delete 掉了（见 3.3.4 节和 17.6.4 节）。换句话说，union 类型的对象不能含有这些函数。

[5] 在 union 的所有成员中，最多只能有一个成员包含类内初始化器（见 17.4.4 节）。

[6] union 不能被用作其他类的基类。

这些约束规则有效地阻止了很多错误的发生，同时简化了 union 的实现过程。后面一点非常重要，因为 union 的主要作用是优化代码的性能，所以我们肯定不希望在使用 union 的过程中引入“隐形的代价”。

如果 union 的成员含有构造函数（及其他），则必须 delete 掉这些函数。这条规则使得简单的 union 使用起来确实简单。如果需要用到更复杂的操作，那么由程序员来实现。例如，因为 Entry 的成员不含构造函数、析构函数及赋值操作，所以我们能自由地创建 Entry 的副本：

```
void f(Entry a)
{
    Entry b = a;
};
```

如果对一个复杂的 union 执行同样的操作，则不仅实现起来很难，而且容易出错：

```
union U {
    int m1;
    complex<double> m2;    //复数含有构造函数
    string m3;             //string 含有构造函数（维护一个字符的不变量）
};
```

要想拷贝 U，程序员必须决定使用哪个拷贝操作。例如：

```
void f2(U x)
{
    U u;                   //错误：哪个默认构造函数？
    U u2 = x;              //错误：哪个拷贝构造函数？
    u.m1 = 1;              //给 int 成员赋值
    string s = u.m3;       //经典灾难：从 string 成员中读取内容
    return;                //错误：x、u 和 u2 使用的是哪个析构函数？
}
```

一般来说，先把值写入某个成员然后读取另一个成员的值通常是非法的，但是人们常常忽略

这一点（从而产生了错误）。在此例中，程序以无效实参调用了 string 的拷贝构造函数。程序员应该庆幸这段代码无法通过编译，否则后果不堪设想。如果确实需要，用户可以定义一个包含 union 的类，该类可以正确地处理 union 中含有构造函数、析构函数和赋值操作（见 8.3.2 节）的成员。同时，该类还能防止先把值写入某个成员然后读取另一个成员的错误。

C++ 允许为联合的最多一个成员指定类内初始化器。此时，该初始化器被用于默认初始化。例如：

```
union U2 {
    int a;
    const char* p {""};
};

U2 x1;          // 执行默认初始化，使得 x1.p == ""
U2 x2 {7};      // x2.a == 7
```

8.3.2　匿名 union

下面的程序建立了 Entry（见 8.3 节）的一个变形，从中可以看出如何编写一个类来解决误用 union 带来的问题：

```
class Entry2 {  // 用 union 规定了两种不同的表示形式
private:
    enum class Tag { number, text };
    Tag type;  // 判别式

    union {    // 表示形式
        int i;
        string s;  // string 有默认构造函数、拷贝操作及析构函数
    };
public:
    struct Bad_entry { };      // 用于处理异常

    string name;

    ~Entry2();
    Entry2& operator=(const Entry2&);      // 因为存在 string 变量，所以是必需的
    Entry2(const Entry2&);
    // ...

    int number() const;
    string text() const;

    void set_number(int n);
    void set_text(const string&);
    // ...
};
```

我不是 get/set 函数的拥趸，但是在这个例子中，我们确实需要为每种访问操作自定义非平凡的形式。我用 Tag 的取值为“get”函数命名，并且在“set”函数前加上 set_ 前缀。这是一种我自己比较习惯的命名方式。

执行读操作的函数的定义如下所示：

```
int Entry2::number() const
{
```

```
    if (type!=Tag::number) throw Bad_entry{};
    return i;
};

string Entry2::text() const
{
    if (type!=Tag::text) throw Bad_entry{};
    return s;
};
```

这两个访问函数首先检查 type 标签，如果是我们想执行的访问，则返回对应值的引用；否则，抛出异常。这样的 union 称为标签联合（tagged union）或者可判别联合（discriminated union）。

执行写操作的函数检查 type 标签的方式与执行读操作的函数基本相同，但是在设置新值的时候必须考虑之前的值的情况：

```
void Entry2::set_number(int n)
{
    if (type==Tag::text) {
        s.~string();                    // 显式地销毁 string（见 11.2.4 节）
        type = Tag::number;
    }
    i = n;
}

void Entry2::set_text(const string& ss)
{
    if (type==Tag::text)
        s = ss;
    else {
        new(&s) string{ss};        // new 的作用是显式地构造 string（见 11.2.4 节）
        type = Tag::text;
    }
}
```

union 的用法使得我们必须用其他一些晦涩的、底层的语言特性（显式构造函数和析构函数）来管理 union 的元素的生命周期。这是应该避免使用 union 的另一个原因。

在 Entry2 中声明的 union 没有命名，它是一个匿名联合（anonymous union）。匿名联合是一个对象而非一种类型，我们无须对象名就能直接访问它的成员。因此，我们使用匿名联合的成员的方式与使用类成员的方式完全一样，只要谨记同一时刻只能使用 union 的一个成员就可以了。

Entry2 含有一个 string 类型的成员，而在 string 类型中有用户自定义的赋值运算符，因此 Entry2 的赋值运算符被 delete 掉了（见 3.3.4 节和 17.6.4 节）。要想为 Entry2 的对象赋值，就必须先定义 Entry2::operator=()。赋值运算兼具读/写两种操作的复杂性，但是它在逻辑上与访问函数很相似：

```
Entry2& Entry2::operator=(const Entry2& e)  // 因为含有 string 变量，所以是必要的
{
    if (type==Tag::text && e.type==Tag::text) {
        s = e.s;            // 常规的 string 赋值
        return *this;
    }

    if (type==Tag::text) s.~string(); // 显式地销毁（见 11.2.4 节）
```

```
        switch (e.type) {
        case Tag::number:
            i = e.i;
            break;
        case Tag::text:
            new(&s)(e.s);  // new 的作用是显式地构造 string（见 11.2.4 节）
            type = e.type;
        }

        return *this;
    }
```

构造函数与移动赋值操作的定义方式可以很类似。我们至少需要一到两个构造函数来建立 type 标签和值之间的对应关系。析构函数必须能处理 string 类型：

```
Entry2::~Entry2()
{
    if (type==Tag::text) s.~string(); // 显式地销毁（见 11.2.4 节）
}
```

8.4　枚举

枚举（enumeration）类型用于存放用户指定的一组整数值（§iso.7.2）。枚举类型的每种取值各自对应一个名字，我们把这些值叫做枚举值（enumerator）。例如：

```
enum class Color { red, green, blue };
```

上述代码定义了一个名为 Color 的枚举类型，它的枚举值是 red、green 和 blue。“一个枚举类型”简称“一个 enum”。

枚举类型分为两种：

[1] enum class，它的枚举值名字（比如 red）位于 enum 的局部作用域内，枚举值不会隐式地转换成其他类型。

[2] “普通的 enum”，它的枚举值名字与枚举类型本身位于同一个作用域中，枚举值隐式地转换成整数。

通常情况下，建议程序员使用 enum class，它很少会产生我们意想不到的结果。

8.4.1　enum class

enum class 是一种限定了作用域的强类型枚举，例如：

```
enum class Traffic_light { red, yellow, green };
enum class Warning { green, yellow, orange, red };// 火警等级

Warning a1 = 7;                 // 错误：不存在 int 向 Warning 的类型转换
int a2 = green;                 // 错误：green 位于它的作用域之外
int a3 = Warning::green;        // 错误：不存在 Warning 向 int 的类型转换
Warning a4 = Warning::green;    // OK

void f(Traffic_light x)
{
    if (x == 9) { /* ... */ }                   // 错误：9 不是一个 Traffic_light
    if (x == red) { /* ... */ }                 // 错误：当前作用域中没有 red
    if (x == Warning::red) { /* ... */ }        // 错误：x 不是一个 Warning
    if (x == Traffic_light::red) { /* ... */ }  // OK
}
```

两个 enum 的枚举值不会互相冲突，它们位于各自 enum class 的作用域中。

枚举常用一些整数类型表示，每个枚举值是一个整数。我们把用于表示某个枚举的类型称为它的基础类型（underlying type）。基础类型必须是一种带符号或无符号的整数类型（见 6.2.4 节），默认是 int。我们可以显式地指定：

```
enum class Warning : int { green, yellow, orange, red }; // sizeof(Warning)==sizeof(int)
```

如果你认为上述定义太浪费空间，可以用 char 代替 int：

```
enum class Warning : char { green, yellow, orange, red };    // sizeof(Warning)==1
```

默认情况下，枚举值从 0 开始，依次递增。因此，我们可以得到：

```
static_cast<int>(Warning::green)==0
static_cast<int>(Warning::yellow)==1
static_cast<int>(Warning::orange)==2
static_cast<int>(Warning::red)==3
```

在有了 Warning 之后，用 Warning 变量代替普通的 int 变量使得用户和编译器都能更好地理解该变量的真正用途。例如：

```
void f(Warning key)
{
    switch (key) {
    case Warning::green:
        // ... 相应的操作 ...
        break;
    case Warning::orange:
        // ... 相应的操作 ...
        break;
    case Warning::red:
        // ... 相应的操作 ...
        break;
    }
}
```

用户很容易发现程序缺少了对 yellow 的处理，编译器也能发现这一点。因为 Warning 的四个值中只处理了三个，所以编译器会发出一条警告信息。

我们可以用整型（见 6.2.1 节）常量表达式（见 10.4 节）初始化枚举值，例如：

```
enum class Printer_flags {
    acknowledge=1,
    paper_empty=2,
    busy=4,
    out_of_black=8,
    out_of_color=16,
    //
};
```

我们特意为 Printer_flags 选取了一些特殊的枚举值，以便能用位运算符把它们组合在一起。enum 属于用户自定义的类型，因此我们可以为它定义 | 和 & 运算符（见 3.2.1.1 节和第 18 章）。例如：

```
constexpr Printer_flags operator|(Printer_flags a, Printer_flags b)
{
    return static_cast<Printer_flags>(static_cast<int>(a)|static_cast<int>(b));
}
```

```
constexpr Printer_flags operator&(Printer_flags a, Printer_flags b)
{
    return static_cast<Printer_flags>(static_cast<int>(a)&static_cast<int>(b));
}
```

因为 enum class 不支持隐式类型转换，所以我们必须在这里使用显式的类型转换。在为 Printer_flags 定义了 | 和 & 之后：

```
void try_to_print(Printer_flags x)
{
    if (x&Printer_flags::acknowledge) {
        // ...
    }
    else if (x&Printer_flags::busy) {
        // ...
    }
    else if (x&(Printer_flags::out_of_black|Printer_flags::out_of_color)) {
        // 缺墨：黑白或彩色
        // ...
    }
    // ...
}
```

我把 operator|() 和 operator&() 定义成了 constexpr 函数（见 10.4 节和 12.1.6 节），这样就能把它们用于常量表达式了。例如：

```
void g(Printer_flags x)
{
    switch (x) {
    case Printer_flags::acknowledge:
        // ...
        break;
    case Printer_flags::busy:
        // ...
        break;
    case Printer_flags::out_of_black:
        // ...
        break;
    case Printer_flags::out_of_color:
        // ...
        break;
    case Printer_flags::out_of_black&Printer_flags::out_of_color:
        // 缺墨：黑白或彩色
        // ...
        break;
    }

    // ...
}
```

C++ 允许先声明一个 enum class，稍后再给出它的定义（见 6.3 节）。例如：

```
enum class Color_code : char;       // 声明
void foobar(Color_code* p);         // 使用声明
// ...
enum class Color_code : char {      // 定义
    red, yellow, green, blue
};
```

一个整数类型的值可以显式地转换成枚举类型。如果这个值属于枚举的基础类型的取值范

围，则转换是有效的；否则，如果超出了合理的表示范围，则转换的结果是未定义的。例如：

```
enum class Flag : char{ x=1, y=2, z=4, e=8 };

Flag f0 {};                          // f0 的默认值是 0
Flag f1 = 5;                         // 类型错误：5 不属于 Flag 类型
Flag f2 = Flag{5};                   // 错误：不允许隐式转换成 enum class 类型
Flag f3 = static_cast<Flag>(5);      // "不近人情" 的转换
Flag f4 = static_cast<Flag>(999);    // 错误：999 不是一个 char 类型的值（也许会丰润数不到）
```

最后一条赋值语句很好地展示了为什么不允许从整数到枚举类型的隐式转换，因为绝大多数整数值根本不在某一枚举类型的合理表示范围之内。

每个枚举值对应一个整数，我们可以显式地把这个整数抽取出来。例如：

```
int i = static_cast<int>(Flag::y);          // i 的值变为 2
char c = static_cast<char>(Flag::e);        // c 的值变为 8
```

我们在这里提到的枚举的取值概念与 Pascal 语言家族中的枚举概念不同。然而，像 Printer_flags 这样的位操作枚举类型在 C 和 C++ 的历史中已经存在很长时间了，它要求枚举值之外的其他值也应该是定义良好的。

对 enum class 执行 sizeof 的结果是对其基础类型执行 sizeof 的结果。如果没有显式指定基础类型，则枚举类型的尺寸等于 sizeof(int)。

8.4.2　普通的 enum

"普通的 enum" 是指 C++ 在提出 enum class 之前提供的枚举类型，在很多 C 和 C++98 的代码中都存在普通的 enum。普通的 enum 的枚举值位于 enum 本身所在的作用域中，它们隐式地转换成某些整数类型的值。我们把 8.4.1 节的例子去掉 class 关键字后变成下面的形式：

```
enum Traffic_light { red, yellow, green };
enum Warning { green, yellow, orange, red }; // 火警等级

// 错误：yellow 被重复定义（取值相同）
// 错误：red 被重复定义（取值不同）

Warning a1 = 7;                      // 错误：不存在 int 向 Warning 的类型转换
int a2 = green;                      // OK：green 位于其作用域中，其已经转换成 int 类型
int a3 = Warning::green;             // OK：Warning 与 int 的类型兼容
Warning a4 = Warning::green;         // OK

void f(Traffic_light x)
{
    if (x == 9) { /* ... */ }                    // OK（但是 Traffic_light 并不包含枚举值 9）
    if (x == red) { /* ... */ }                  // 错误：作用域中有两个 red
    if (x == Warning::red) { /* ... */ }         // OK（没错！）
    if (x == Traffic_light::red) { /* ... */ }   // OK
}
```

我们在同一个作用域的两个普通枚举中都定义了 red，从而"很幸运地"避免了一个难以发现的错误。我们可以人为地消除枚举值的二义性，以实现对于普通 enum 的"清理"（在小规模程序中很容易做到，但是在规模较大的程序中就很难做到了）：

```
enum Traffic_light { tl_red, tl_yellow, tl_green };
enum Warning { green, yellow, orange, red }; // 火警等级
```

```
void f(Traffic_light x)
{
    if (x == red) { /* ... */ }                    // OK（哎哟！）
    if (x == Warning::red) { /* ... */ }           // OK（哎哟！）
    if (x == Traffic_light::red) { /* ... */ }     // 错误：red 不是一个 Traffic_light 类型的值
}
```

从编译器的角度来看，x==red 是合法的，但它几乎肯定是一个程序缺陷。把名字注入外层作用域（当使用 enum 时会发生这种情况，但是使用 enum class 和 class 不会）的行为称为名字空间污染（namespace pollution），在规模较大的程序中应该尽量避免这样做（第 14 章）。

你可以为普通的枚举指定基础类型，就像你对 enum class 所做的一样。此时，允许先声明枚举类型，稍后再给出它的定义。例如：

```
enum Traffic_light : char { tl_red, tl_yellow, tl_green };  // 基础类型是 cha

enum Color_code : char;      // 声明
void foobar(Color_code* p);  // 使用声明
// ...
enum Color_code : char { red, yellow, green, blue }; // 定义
```

如果没有指定枚举的基础类型，则不能把它的声明和定义分开。此时，枚举的基础类型需要通过一个相对复杂的算法推算出来：如果所有枚举值都是非负值，则该枚举类型的范围是 $[0:2^k-1]$，其中 2^k 是 2 的最小整数次幂并且保证所有枚举值都位于该范围内。如果存在负值，则范围是 $[-2^k:2^k-1]$。该算法定义了能够存放所有枚举值的最小位域，其中，枚举值是用传统的二进制补码表示的。例如：

```
enum E1 { dark, light };                    // 范围 0:1
enum E2 { a = 3, b = 9 };                   // 范围 0:15
enum E3 { min = -10, max = 1000000 };       // 范围 -1048576:1048575
```

整数到普通 enum 的显式类型转换规则与转换为 enum class 的规则一样。稍有的一点区别是，当没有显式地指定基础类型时，除非该值位于枚举类型的范围之内，否则转换的结果是未定义的。例如：

```
enum Flag { x=1, y=2, z=4, e=8 };   // 范围 0:15

Flag f0 {};                         // f0 的默认值是 0
Flag f1 = 5;                        // 类型错误：5 不是一个 Flag
Flag f2 = Flag{5};                  // 错误：不存在 int 向 Flag 的显式类型转换
Flag f2 = static_cast<Flag>(5);     // OK：5 在 Flag 的取值范围之内
Flag f3 = static_cast<Flag>(z|e);   // OK：12 在 Flag 的取值范围之内
Flag f4 = static_cast<Flag>(99);    // 未定义的：99 不在 Flag 的取值范围之内
```

因为普通的 enum 和其基础类型之间存在隐式类型转换，所以我们不需要为它专门定义运算符 |：z 和 e 会自动转换成 int，因此 z|e 能够正常求值。对枚举类型求 sizeof 的结果等价于对其基础类型求 sizeof 的结果。如果没有显式地指定基础类型，则除非其中的枚举值不能表示成 int 或者 unsigned int，否则该枚举将取某种既能保存其范围内的值又不超过 sizeof(int) 的整数类型作为其类型。例如在 sizeof(int)==4 的机器上，sizeof(Flags) 可能是 1，也可能是 4，但不会是 8。

8.4.3　未命名的 enum

一个普通的 enum 可以是未命名的，例如：

```
enum { arrow_up=1, arrow_down, arrow_sideways };
```

如果我们需要的只是一组整型常量，而不是用于声明变量的类型，则可以使用未命名的enum。

8.5 建议

[1] 如果想紧凑地存储数据，则把结构中尺寸较大的成员布局在较小的成员之前；8.2.1节。

[2] 用位域表示由硬件决定的数据布局；8.2.7节。

[3] 不要天真地认为仅靠把几个值打包在一个字节中就能轻易地优化内存；8.2.7节。

[4] 用union减少内存空间的使用（表示一组候选项），不要将它用于类型转换；8.3节。

[5] 用枚举类型表示一组命名的常量；8.4节。

[6] enum class比“普通的”enum可靠；8.4节。

[7] 为枚举类型定义一些适当的操作，以便我们能够既安全又便捷地使用它；8.4.1节。

语　句

程序员就是一台能把咖啡因变成代码的机器。

——某位程序员

- 引言
- 语句概述
- 声明作为语句
- 选择语句
  if 语句；switch 语句；条件中的声明
- 循环语句
  范围 for 语句；for 语句；while 语句；do 语句；退出循环
- goto 语句
- 注释与缩进
- 建议

9.1 引言

C++ 提供了一组既符合传统又灵活易用的语句。基本上，与表达式和声明有关的知识要么相当有趣，要么异常复杂。一个声明就是一条语句，表达式的末尾加上一个分号也是一条语句。

与表达式不同，语句本身没有值。语句的主要作用是指定执行的顺序。例如：

```
a = b+c;        // 表达式语句
if (a==7)       // if 语句
    b = 9;      // 当且仅当 a==7 时，执行这条语句
```

从逻辑角度来说，a=b+c 的执行发生在 if 之前，这符合人们的预期。为了提高程序的性能，编译器可能会在确保执行结果不变的前提下调整代码的顺序。

9.2 语句概述

这里是 C++ 语句的形式化定义：

```
语句：
    声明
    表达式opt ;
    { 语句列表opt }
    try { 语句列表opt } 处理模块列表

    case 常量表达式 : 语句
    default : 语句
    break ;
    continue ;

    return 表达式opt ;
```

```
    goto 标识符;
    标识符:语句

    选择语句
    循环语句

选择语句:
    if ( 条件 ) 语句
    if ( 条件 ) 语句 else 语句
    switch ( 条件 ) 语句

循环语句:
    while ( 条件 ) 语句
    do 语句 while ( 表达式 );
    for ( for 初始化语句 条件可选 ; 表达式可选 ) 语句
    for ( for 初始化声明 : 表达式 ) 语句

语句列表:
    语句 语句列表opt

条件:
    表达式
    类型修饰符 声明符 = 表达式
    类型修饰符 声明符 { 表达式 }

处理模块列表:
    处理模块 处理模块列表opt

处理模块:
    catch ( 表达式声明 ) { 语句列表opt }
```

分号本身也是一条语句，即空语句（empty statement）。

"花括号"（{}）括起来的一个可能为空的语句序列称为块（block）或者复合语句（compound statement）。块中声明的名字的作用域到块的末尾就结束了（6.3.4 节）。

声明（declaration）是一条语句，没有赋值语句或过程调用语句；赋值和函数调用不是语句，它们是表达式。

for 初始化语句（for-init-statement）要么是声明，要么是一条表达式语句（expression-statement），它们都以分号结束。

for 初始化声明（for-init-declaration）必须是一个未初始化变量的声明。

try 语句块（try-block）的作用是处理异常，我们将在 13.5 节介绍它。

9.3 声明作为语句

一个声明就是一条语句。除非变量被声明成 static，否则在控制线程传递给当前声明语句的同时执行初始化器（见 6.4.2 节）。允许把声明当成一条语句使用（当然还能用在其他一些场合，见 9.4.3 节和 9.5.2 节）的目的是尽量减少由未初始化变量造成的程序错误，并且让代码的局部性更好。在绝大多数情况下，如果没有为变量找到一个合适的值，暂时不要声明它。例如：

```
void f(vector<string>& v, int i, const char* p)
{
    if (p==nullptr) return;
    if (i<0 || v.size()<=i)
        error("bad index");
    string s = v[i];
    if (s == p) {
```

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

对于很多常量和单赋值形式的程序设计（对象的值一旦初始化就不再改变）来说，能否把声明置于可执行代码之后至关重要。同样，对于用户自定义的数据类型，先确定一个合适的初始化器再定义变量能获得更好的程序性能。例如：

```
void use()
{
    string s1;
    s1 = "The best is the enemy of the good.";
    // ...
}
```

这段代码先把 s1 默认初始化成空字符串再对它赋值，这么做显然不如直接用给定的值初始化效率高：

```
string s2 {"Voltaire"};
```

声明一个缺少初始化器的变量，常常是因为我们需要一条专门的语句给变量赋值。其中一种情况是等待用户输入数据：

```
void input()
{
    int buf[max];
    int count = 0;
    for (int i; cin>>i;) {
        if (i<0) error("unexpected negative value");
        if (count==max) error("buffer overflow");
        buf[count++] = i;
    }
    // ...
}
```

假定 error() 不会令函数结束并返回，否则的话，这段代码将造成缓冲区溢出。通常情况下 push_back()（见 3.2.1.3 节、13.6 节和 31.3.6 节）能以更好的方式实现上述功能。

9.4　选择语句

if 语句和 switch 语句都需要首先检测一个值：

```
if ( 条件 ) 语句
if ( 条件 ) 语句 else 语句
switch ( 条件 ) 语句
```

条件（condition）可能是一个表达式，也可能是一个声明（9.4.3 节）。

9.4.1　if 语句

在 if 语句中，如果条件为真，则执行第一条（或者唯一的一条）语句；否则，执行第二条语句（如果有的话）。即使条件的求值结果不是布尔值，也能尽量隐式地转换成 bool 类型。因此，算术类型及指针类型的表达式都能作为条件。例如，如果 x 是一个整数，则

```
if (x) // ...
```

等价于

```
if (x != 0) // ...
```

对于指针 p 来说，

```
if (p) // ...
```

是一种非常直接的表达，它的含义是："指针 p 指向一个有效的对象（假定被正确地初始化了）吗？"它等价于

```
if (p != nullptr) // ...
```

"普通的" enum 可以先隐式地转换成整数，然后再转换成 bool 类型，但是 enum class 不能（见 8.4.1 节）。例如：

```
enum E1 { a, b };
enum class E2 { a, b };

void f(E1 x, E2 y)
{
    if (x)          // OK
        // ...
    if (y)          // 错误：无法转换成 bool 类型
        // ...
    if (y==E2::a)   // OK
        // ...
}
```

逻辑运算符

```
&&  ||  !
```

经常在条件中出现。对于运算符 && 和 || 来说，除非必需，否则运算符右侧的运算对象不会被求值。例如：

```
if (p && 1<p->count) // ...
```

只有当 p 不是 nullptr 的时候才会检查 1<p->count。

如果要在两项中选择一项，则与 if 语句相比，条件表达式（见 11.1.3 节）在表达程序意图的能力上更胜一筹。例如：

```
int max(int a, int b)
{
    return (a>b)?a:b;    // 返回 a 和 b 中较大的一个
}
```

一个名字只能在声明它的作用域中使用。在 if 语句中，一个分支声明的名字不能在另一个分支中直接使用。例如：

```
void f2(int i)
{
    if (i) {
        int x = i+2;
        ++x;
        // ...
    }
    else {
        ++x; // 错误：x 在其作用域之外
    }
```

```
        ++x;        // 错误：x 在其作用域之外
    }
```

if 语句的一个分支不能仅有一条声明语句，没有别的语句。如果我们想在一个分支中引入一个新名字，则该声明语句必须包含在块中（见 9.2 节）。例如：

```
void f1(int i)
{
    if (i)
        int x = i+2;            // 错误：if 语句分支的声明
}
```

9.4.2　switch 语句

switch 语句在一组候选项（case 标签）中进行选择。case 标签中出现的表达式必须是整型或枚举类型的常量表达式。在同一个 switch 语句中，一个值最多被 case 标签使用一次。例如：

```
void f(int i)
{
    switch (i) {
    case 2.7: // 错误：在 case 中使用浮点数
        // ...
    case 2:
        // ...
    case 4-2: // 错误：在 case 标签中使用了两次常量 2
        // ...
};
```

switch 语句可以用一组 if 语句等价地替换，例如：

```
switch (val) {
case 1:
    f();
    break;
case 2:
    g();
    break;
default:
    h();
    break;
}
```

可以将它等价转换为：

```
if (val == 1)
    f();
else if (val == 2)
    g();
else
    h();
```

上面两种形式的含义相同，但是相对来说 switch 版本更好，它清晰地表达出了操作的本质（检查给定的值是一组常量中的哪一个）。我们没有必要反复检查同一个值，因此 switch 语句产生的代码更好。尤其是在处理一些非平凡的示例时，switch 语句更容易读懂。可以使用跳表实现上述功能。

谨记 switch 语句的每一个分支都应该有一条结束语句，否则程序将会继续执行下一个

分支的内容。考虑下面的情况：

```
switch (val) {                    // 注意
case 1:
    cout << "case 1\n";
case 2:
    cout << "case 2\n";
default:
    cout << "default: case not found\n";
}
```

假设 val==1，则实际的输出结果可能会有点出人意料：

```
case 1
case 2
default: case not found
```

有一种好的解决办法：在那些我们确实希望继续执行下一个分支的地方（这种情况可能很少发生）加上注释，指明程序的意图，那么如果我们发现了在某处缺少结束语句而又没有明确的注释，则可以判断这是一处错误。例如：

```
switch (action) {                 // 处理 (action,value) 对
case do_and_print:
    act(value);
    // 不跳出；继续执行输出操作
case print:
    print(value);
    break;
// ...
}
```

要想结束一个分支，最常用的是 break 语句，有时候也可以用 return 语句（见 10.2.1 节）。

switch 语句在何种情况下应该包含一个 default 分支呢？很难用一句话回答这个问题。一种用法是让 default 处理最常出现的情况；另一种用法恰恰相反，default 只是用来处理取值错误的情况，所有有效的取值都包含在 case 分支中了。然而，有一种情况下不应该使用 default：switch 语句希望它的每个分支对应枚举类型中的一个枚举值。如果是这样的话，最好不要使用 default 语句，应该让编译器负责发现并报告 case 分支与枚举值未能完全匹配的问题。例如，下面的代码几乎肯定是错误的：

```
enum class Vessel { cup, glass, goblet, chalice };

void problematic(Vessel v)
{
    switch (v) {
    case Vessel::cup:       /* ... */   break;
    case Vessel::glass:     /* ... */   break;
    case Vessel::goblet:    /* ... */   break;
    }
}
```

如果在程序的后期维护和升级过程中，我们给枚举类型添加了一个新的枚举值，则很容易引发上述错误。

检测一个“不可能的”枚举值的操作最好与其他代码分离开来。

9.4.2.1 case 分支中的声明

C++ 允许在 switch 语句的块内声明变量，但是不能不初始化。例如：

```
void f(int i)
{
    switch (i) {
    case 0:
        int x;          // 未初始化
        int y = 3;      // 错误：程序有可能跳过该声明（显式初始化）
        string s;       // 错误：程序有可能跳过该声明（隐式初始化）
    case 1:
        ++x;            // 错误：试图使用未初始化的对象
        ++y;
        s = "nasty!";
    }
}
```

在上面的代码中，如果 i==1，程序将跳过 y 和 s 的初始化操作。因此，f() 无法通过编译。不幸的是，因为 int 无须初始化，所以 x 的声明语句不会报错。然而，语法上不错不代表用法上不错：实际上我们读取了一个未初始化的变量。仅靠编译器无法解决这一问题，因为编译器只对未初始化的变量进行警告而不是报错，所以不能可靠地捕获出现的问题。程序员应该避免使用未初始化的变量（见 6.3.5.1 节）。

如果我们确实需要在 switch 语句中使用变量，最好把该变量的声明和使用限定在一个块中。相关示例请见 10.2.1 节的 prim()。

9.4.3　条件中的声明

要想避免不小心误用变量，最好的办法是把变量的作用域限定在一个较小的范围内。此外，我们应该尽量延迟局部变量的定义，直到能给它赋初值为止。这样做能避免很多不必要的麻烦。

有个很简洁的例子能满足上述原则，我们在条件中声明变量：

```
if (double d = prim(true)) {
    left /= d;
    break;
}
```

首先声明 d 并给它赋了初值，然后把初始化后的 d 的值作为条件的值进行检查。d 的作用域从声明处开始，到条件控制的语句结束为止。假设还有一个 else 分支与上面的 if 分支对应，则 d 在两个分支中都有效。

另一种比较传统的做法是在条件之前声明 d。不过这么做很可能造成 d 的实际作用域变得太大，远远超出我们预期的范围，并且可能使得 d 未经初始化就被使用：

```
double d;
// ...
d2 = d;   // 哎哟！出错了！
// ...
if (d = prim(true)) {
    left /= d;
    break;
}
// ...
d = 2.0;  // 与 d 的预期用途毫无关系
```

在条件中声明变量一方面逻辑性更强，另一方面也让代码显得更紧凑。

条件中的声明语句只能声明并初始化一个变量或 const。

9.5 循环语句

循环语句能表示成 for、while 和 do 的形式：

```
while ( 条件 ) 语句
do 语句 while ( 表达式 ) ;
for ( for 初始化语句 条件可选 ; 表达式可选 ) 语句
for ( for 声明 : 表达式 ) 语句
```

for 初始化语句（for-init-statement）要么是一个声明，要么是一条表达式语句（expression-statement）。它们都以分号结束。

for 语句的语句（称为受控语句（controlled statement）或者循环体（loop body））重复执行，直到条件变为 false 或者程序员以其他方式退出循环（比如 break、return、throw 和 goto）为止。

复杂的循环可以用算法加上 lambda 表达式来表示（见 11.4.2 节）。

9.5.1 范围 for 语句

最简单的循环是范围 for 语句，它使得程序员可以依次访问指定范围内的每个元素。例如：

```
int sum(vector<int>& v)
{
    int s = 0;
    for (int x : v)
        s+=x;
    return s;
}
```

for (int x : v) 读作“对于范围 v 中的每个元素 x”，或者干脆说“对于 v 中的每个 x”。程序从头到尾依次访问 v 的全部元素。

命名元素的变量的作用域是整个 for 语句。

冒号之后的表达式必须是一个序列（一个范围），换句话说，如果我们对它调用 v.begin() 和 v.end() 或者 begin(v) 和 end(v)，得到的应该是迭代器（见 4.5 节）：

[1] 编译器首先尝试寻找并使用成员 begin 和 end。如果找到了 begin 和 end，但是它们不能表示一个范围（比如，begin 有可能是变量而非函数），则当前的范围 for 是错误的。

[2] 如果没有找到，则编译器继续在外层作用域寻找 begin/end 成员。如果找不到或者找到的不能用（比如 begin 不接受当前序列类型的实参），则范围 for 是错误的。

对于内置数组 T v[N] 来说，编译器使用 v 和 v+N 代替 begin(v) 和 end(v)。<iterator> 头文件为内置数组和所有标准库容器提供了 begin(c) 和 end(c)。我们也可以为自己的序列定义 begin() 和 end()，只要它们的工作方式与标准库容器中的一样就可以了（见 4.4.5 节）。

控制变量 x 指向当前正在处理的元素，它等价于 for 语句中的 *p：

```
int sum2(vector<int>& v)
{
    int s = 0;
    for (auto p = begin(v); p!=end(v); ++p)
```

```
        s+=*p;
    return s;
}
```

如果想在范围 for 循环中修改元素的值，则应该使用元素的引用。例如，我们用下面的代码把 vector 的每个元素都加 1：

```
void incr(vector<int>& v)
{
    for (int& x : v)
        ++x;
}
```

引用还可以用于尺寸较大的元素，特别是当直接拷贝元素的值代价昂贵时。例如：

```
template<class T> T accum(vector<T>& v)
{
    T sum = 0;
    for (const T& x : v)
        sum += x;
    return sum;
}
```

范围 for 循环是一种非常简单的语法结构。例如，你无法用它同时访问两个元素，也不能同时遍历两个序列。这种简单性是语言的设计者有意为之，如果想处理比较复杂的情况，程序员应该选用普通的 for 语句。

9.5.2　for 语句

除了范围 for 语句之外，还有一种更通用的 for 语句，它允许程序员对循环过程控制得更细。其中，循环变量、终止条件和负责更新循环变量的表达式被显式地“预先”放置在第一行。例如：

```
void f(int v[], int max)
{
    for (int i = 0; i!=max; ++i)
        v[i] = i*i;
}
```

它等价于

```
void f(int v[], int max)
{
    int i = 0;          // 引入循环变量
    while (i!=max) {    // 检验终止条件
        v[i] = i*i;     // 执行循环体
        ++i;            // 递增循环变量
    }
}
```

我们可以在 for 语句的初始化器部分声明变量。如果初始化器部分是一个声明，则该声明引入的变量的作用域直至 for 语句的末尾才会结束。

很多时候，我们并不清楚 for 循环的控制变量应该是什么类型，此时 auto 关键字就派上用场了：

```
for (auto p = begin(c); c!=end(c); ++p) {
    // ... 通过迭代器 p 依次访问容器 c 的每个元素 ...
}
```

如果在 for 循环结束后还需要使用最终的索引值，则必须在 for 循环之外声明索引变量（见 9.6 节）。

如果不需要初始化过程，则初始化语句可以为空。

如果缺少了递增循环变量的表达式，则我们必须在循环体内或者别的某处以适当的方式更新循环变量。如果循环不符合“引入一个循环变量，检验条件，更新循环变量”的模式，则它更适合用 while 语句表示。考虑下面这个简洁的变形：

```
for (string s; cin>>s;)
    v.push_back(s);
```

其中，cin>>s 既负责读入数据，也具有检验终止条件是否成立的功能，因此无须再声明一个显式的循环变量。另一方面，for 语句允许我们把“当前元素”s 的作用域限定在 for 语句本身的范围内，这是 while 语句做不到的。

for 语句还可以表示没有显式终止条件的循环：

```
for (;;) {   // "永远"
    // ...
}
```

但是很多人不喜欢上面的用法，认为它的含义有些含糊不清，他们更愿意用下面的语句：

```
while(true) {   // "永远"
    // ...
}
```

9.5.3 while 语句

while 语句重复执行它的受控语句直到条件部分变成 false，例如：

```
template<class Iter, class Value>
Iter find(Iter first, Iter last, Value val)
{
    while (first!=last && *first!=val)
        ++first;
    return first;
}
```

与 for 语句相比，while 语句更适合处理以下两种情况：一是没有一个明显的循环变量，二是程序员觉得把负责更新循环变量的语句置于循环体内更自然。

for 语句（见 9.5.2 节）很容易改写成等价的 while 语句，反之亦然。

9.5.4 do 语句

do 语句与 while 语句非常相似，唯一的区别是 do 语句的条件位于循环体之后。例如：

```
void print_backwards(char a[], int i)    // i 必须为正
{
    cout << '{';
    do {
        cout << a[--i];
    } while (i);
    cout << '}';
}
```

我们调用该函数的方式可能是：print_backwards(s,strlen(s));程序的执行过程很容易出现非常严重的错误。例如，s 如果是空字符串怎么办？

以我的经验来看，do 语句很容易造成代码混乱甚至错误。在 do 语句中，不管条件如何循环体都肯定会被执行一次，但是要想确保循环体正常工作，我们必须确保一些类似于条件的东西成立，即使是第一次执行循环也不例外。现实情况往往事与愿违，无论是第一次编写和检验程序还是后来位于循环前面的代码被修改了，都有可能造成条件无法达成。因此，我更愿意把条件“提前放在我能看得见的地方”。建议尽量避免使用 do 语句。

9.5.5　退出循环

如果 for 语句、while 语句或者 do 语句缺少了条件（condition）部分，则除非用户显式地使用了 break、return（见 12.1.4 节）、goto（见 9.6 节）、throw（见 13.5 节），或者采取了像 exit()（见 15.4.3 节）这样不太明显的手段，其他情况下循环都将无法正常终止。break 语句负责“跳出”最近的外层 switch 语句（见 9.4.2 节）或者循环语句。例如：

```
void f(vector<string>& v, string terminator)
{
    char c;
    string s;
    while (cin>>c) {
        // ...
        if (c == '\n') break;
        // ...
    }
}
```

当我们需要“中途”离开循环体的时候，可以使用 break 语句。通常情况下，应该让完整退出的条件位于 while 语句和 for 语句的条件部分，只要这么做不会违背循环本身的逻辑就行（比如需要引入一个额外的变量）。

有时候，我们并不想完全退出循环，我们只想到达循环体的末尾。continue 可以跳过循环语句循环体的剩余部分，例如：

```
void find_prime(vector<string>& v)
{
    for (int i = 0; i!=v.size(); ++i) {
        if (!prime(v[i]) continue;
        return v[i];
    }
}
```

continue 之后继续执行递增循环变量的语句（如果有的话），然后检验循环条件是否满足（如果有的话）。因此，find_prime() 等价于下面的形式：

```
void find_prime(vector<string>& v)
{
    for (int i = 0; i!=v.size(); ++i) {
        if (!prime(v[i]) {
            return v[i];
        }
    }
}
```

9.6　goto 语句

C++ 支持臭名昭著的 goto 语句：

```
goto 标识符;
标识符：语句
```

goto 语句在大多数由程序员直接完成的高级程序设计任务中都没什么用，但是在由别的程序生成的 C++ 代码中非常重要。例如，goto 可能会出现在一个由解析生成器生成的解析程序中。

标签的作用域是标签所处的函数（见 6.3.4 节）。这意味着你能用 goto 从块的范围跳进跳出，唯一的限制是不能跳过初始化器或者跳入到异常处理程序中（见 13.5 节）。

在一般的代码中，goto 可以用来跳出嵌套的循环或者 switch 语句（break 只能跳出最内层的循环或者 switch 语句），这是它为数不多的有意义的用法之一。例如：

```
void do_something(int i, int j)
    // 操作一个名为 nm 的二维矩阵
{
    for (i = 0; i!=n; ++i)
        for (j = 0; j!=m; ++j)
            if (nm[i][j] == a)
                goto found;
    // 无关代码
    // ...
found:
    // nm[i][j] == a
}
```

这个 goto 直接跳过了（退出了）整个循环。它既不会开启另一次循环，也不会进入到一个新作用域中。因此，goto 的这种用法几乎不会给程序员带来任何麻烦和困扰。

9.7 注释与缩进

通过为代码添加适当的注释并且保持缩进格式规范有序，能让阅读和理解程序的任务变得轻松惬意，不再那么枯燥。缩进格式有几种常见的风格，很难说一种比另一种更好（尽管如同大多数程序员一样，我也有自己的倾向，并且读者可以从本书中窥见一斑）。注释的风格也是同样的道理。

反过来说，注释如果用不好也会严重影响程序的可读性。因为编译器不负责理解注释的内容，所以它无法确保一段注释：

- 是有意义的；
- 确实描述了当前的程序；
- 是最新的。

大多数程序都含有一些难以理解的、具有二义性的并且充斥着错误的注释。糟糕的注释还不如没有注释。

如果语言本身能说清楚某件事，那就不要放在注释中，而应该让语言来完成。这条建议的反例可能是：

```
// 变量 "v" 必须初始化
// 变量 "v" 只能用在函数 "f()" 中
// 在当前文件中先调用函数 "init()"，再调用其他函数
// 记得在程序结束前调用函数 "cleanup()"
// 不要调用函数 "weird()"
// 函数 "f(int...)" 接受 2、3 个实参
```

这样的注释其实根本没有必要，相反应该由 C++ 代码本身负责完成这些要求。

语言如果已经把一件事情描述清楚了，就不要在注释中再提一次。例如：

```
a = b+c;  // a 的值变为 b+c
count++; // 递增计数器
```

这样的注释很糟糕，远远不只是冗余这么简单。它们增加了代码读者的阅读量，使得程序的结构杂乱无章，甚至有可能是错误的。然而，在程序设计语言的教科书中（包括本书在内），这类注释被大量使用。这也是教科书程序与真实程序的区别之一。

好注释负责指明一段代码应该实现什么功能（代码的意图），而代码本身负责完成该功能（完成的方式）。最好的方式是，注释的语言应该保持在一个较高层次的抽象水平上，这样便于人们理解而无须纠结于过多技术细节。

关于注释，我的习惯是：

- 在针对每个源文件的注释中指明：该文件中的声明有何共同点、对应的参考手册条目、程序员的名字以及维护该文件所需的其他信息。
- 为每个类、模板和名字空间分别编写注释。
- 为每个非平凡的函数分别编写注释并指明：函数的目的、用到的算法（如果很明显的话可以不用提），以及该函数对其应用环境所做的某些设定。
- 为全局和名字空间内的每个变量及常量分别编写注释。
- 为某些不太明显或不可移植的代码编写注释。
- 其他情况，则几乎不需要注释了。

例如：

```
//    tbl.c: 实现符号表

/*
      Gaussian elimination with partial pivoting.
      见 Ralston: "A first course ..." ，第 411 页
*/

//    scan(p,n,c) 要求 p 指向一个至少包含 n 个元素的数组

// sort(p,q) 用 < 运算符排序序列 [p:q) 中的元素

// 2013-2-29 日修订[⊖]：处理无效日期。修订者：Bjarne Stroustrup
```

经过精心设计和编写的注释是完美程序必不可少的部分。编写漂亮的注释就像艺术创作一样需要灵感和日积月累的经验，其难度一点儿也不亚于编写程序本身。

注意，/* */ 形式的注释不支持嵌套。例如：

```
/*
      删除吊销的检查
      if (check(p,q)) error("bad p q") /* 永远不会发生 */
*/
```

这种嵌套的写法将会因为最后一个未匹配的 */ 而报错。

⊖ 事实上 2013 年 2 月只有 28 天，因此 2013-2-29 本身就是无效日期，这可以理解为作者的一个小幽默。——译者注

9.8 建议

[1] 直到有了合适的初始值再声明变量；9.3 节，9.4.3 节，9.5.2 节。
[2] 如果可能的话，优先选用 switch 语句而非 if 语句；9.4.2 节。
[3] 如果可能的话，优先选用范围 for 语句而非普通的 for 语句；9.5.1 节。
[4] 当存在明显的循环变量时，优先选用 for 语句而非 while 语句；9.5.2 节。
[5] 当没有明显的循环变量时，优先选用 while 语句而非 for 语句；9.5.3 节。
[6] 避免使用 do 语句；9.5 节。
[7] 避免使用 goto；9.6 节。
[8] 注释应该简短直接；9.7 节。
[9] 代码能说清楚的事就别放在注释中；9.7 节。
[10] 注释应该表明程序的意图；9.7 节。
[11] 坚持一种缩进风格，不要轻易改变；9.7 节。

第 10 章

The C++ Programming Language, Fourth Edition

表 达 式

程序设计就像性爱一样：
尽管它们可以带来一些实际的成果，
但那并不是我们喜欢做它们的原因。

——抱歉，理查德·费曼先生[⊖]

- 引言
- 一个桌面计算器示例
  分析器；输入；底层输入；错误处理；驱动程序；头文件；命令行参数；关于风格
- 运算符概述
  结果；求值顺序；运算符优先级；临时对象
- 常量表达式
  符号化常量；常量表达式中的 const；字面值常量类型；引用参数；地址常量表达式
- 隐式类型转换
  提升；类型转换；常用的算术类型转换
- 建议

10.1 引言

本章讨论表达式的细节。在 C++ 语言中，一次赋值是一个表达式、一次函数调用是一个表达式、构造一个对象是一个表达式、传统算术表达式求值之外的许多其他操作也是表达式。为了在一个上下文情境中讲解表达式的用法，我在这里提供了一个规模较小但是比较完整的示例程序，即，“桌面计算器”。然后会列举出 C++ 的全部运算符以及它们作用于内置类型时的简单释义。关于运算符的更多内容将在第 11 章进一步讨论。

10.2 一个桌面计算器示例

我们的桌面计算器程序提供了四种标准算术操作，它们以中缀运算符的形式出现，可以作用于浮点数。此外，用户还可以定义变量。例如，给定输入

```
r = 2.5
area = pi * r * r
```

（预先定义了 pi）计算器程序将会输出

```
2.5
19.635
```

其中，2.5 是第一行输入的结果，19.635 是第二行的结果。

⊖ 诺贝尔物理学奖得主理查德·费曼曾经说过，“物理学就如同性爱一样，尽管它们可以带来一些实际的成果，但那并不是我们喜欢做它们的原因。”作者在这里套用了费曼的说法。——译者注

计算器包含四个部分：分析器、输入函数、符号表和驱动。实际上，它的功能有点类似于一个微型编译器；其中分析器负责分析语法，输入函数负责处理输入及词法分析，符号表存放永久信息，驱动处理初始化、输出和错误。我们当然可以为该计算器再添加一些功能，使得它更贴近实用。但是那样的话代码就太长了，并且额外添加的功能也不会涉及 C++ 的更多语法细节。

10.2.1 分析器

下面是计算器程序遵循的一套语法：

```
program:
    end                          // end 是输入的结束
    expr_list end

expr_list:
    expression print             // print 是换行或者分号
    expression print expr_list

expression:
    expression + term
    expression - term
    term

term:
    term / primary
    term * primary
    primary

primary:
    number                       // number 是一个浮点型字面值常量
    name                         // name 是一个标识符
    name = expression
    - primary
    ( expression )
```

换句话说，程序就是以分号隔开的一段表达式序列。表达式的基本单元是数字、名字以及运算符 *、/、+、-（一元和二元）和 =（赋值）等。其中，名字不需要在使用之前提前声明。

我使用的是一种名为递归下降（recursive descent）的语法分析机制，这是一种被广泛接受且含义明确的自顶向下的技术。在 C++ 等函数调用相对廉价的编程语言中，递归下降的效率很高。在符合语法规则的每个生成结果中，总有一个函数负责调用其他函数。词法分析器负责识别终结符（end、number、+ 和 -），语法分析函数负责识别非终结符（expr()、term() 和 prim()）。只要（子）表达式的两个运算对象已知，该表达式就会被求值。在真实的编译环境中，将于此时生成代码。

在处理输入的环节，分析器使用了 Token_stream，它负责把读入字符的过程以及字符的组成情况封装到 Token 中。也就是说，Token_stream 的作用是“单词化”：它负责把字符流（比如 123.45）转换成 Token。其中，Token 是一个形如 {单词类型，值} 的对，在 {数字，123.45} 的例子中，123.45 被转换成浮点数值。分析器的主体部分只需要知道 Token_stream 的名字是 ts 以及如何从中获取 Token 就可以了。函数 ts.get() 负责读取下一个 Token；函数 ts.current() 可以获得刚刚读进来的 Token（“当前单词”）。在提供单词化服务的过程中，Token_stream 会隐藏字符的真实来源。这些来源可能包括用户直接在 cin 录

入的内容、程序命令行以及任何其他输入流（见 10.2.7 节）。

Token 的定义如下所示：

```
enum class Kind : char {
    name, number, end,
    plus='+', minus='-', mul='*', div='/', print=';', assign='=', lp='(', rp=')'
};

struct Token {
    Kind kind;
    string string_value;
    double number_value;
};
```

把每个单词表示成字符对应的整数形式是一种便捷有效的手段，有助于程序员进行调试。只要输入字符对应的整数值不是一个枚举值，上面的方式就是可行的。并且据我所知，在目前常用的字符集中，没有任何一个可打印的字符对应的整数值只含一个数字。

Token_stream 接口的形式是：

```
class Token_stream {
public:
    Token get();                // 读取并返回下一个单词
    const Token& current();     // 刚刚读入的单词
    // ...
};
```

实现细节将在 10.2.2 节介绍。

每个分析函数接受一个名为 get 的 bool 类型（见 6.2.2 节）实参，它指明函数是否需要调用 Token_stream::get() 以获取下一个单词。每个分析函数求值“它的”表达式并返回相应的值。函数 expr() 处理加法和减法，它包含一个循环，依次寻找加法和减法所需的项：

```
double expr(bool get)           // 加法和减法
{
    double left = term(get);

    for (;;) {                          // 读作“forever”
        switch (ts.current().kind) {
        case Kind::plus:
            left += term(true);
            break;
        case Kind::minus:
            left -= term(true);
            break;
        default:
            return left;
        }
    }
}
```

这个函数本身并不能做多少事。就像一个大程序的顶层函数一样，它通过调用其他函数来执行具体的操作。

switch 语句（见 2.2.4 节和 9.4.2 节）的条件位于 switch 关键字之后的一对括号内，我们检查该条件的值是否是一组常量中的某一个。break 语句用于跳出 switch 语句。如果被检验的值与所有 case 标签都不匹配，则执行 default 分支。程序员无须提供 default。

注意，语法规定：形如 2-3+4 的式子，其真实的求值过程是 (2-3)+4。

for(;;) 表示一个死循环，它可以读作英文单词“forever”的读音（见 9.5 节），死循环的另一种形式是 while(true)。switch 语句不断重复执行直到遇到了 + 和 - 之外的符号，此时默认分支中的 return 语句令程序跳出循环。

运算符 += 和 -= 用于执行加法和减法。我们也可以使用 left=left+term(true) 和 left=left-term(true) 表达相同的含义，但是与之相比，left+=term(true) 和 left-=term(true) 不但更短小精干，而且能够更直接地表达程序的意图。每个赋值运算符都是词法意义上的一个单独的单词，因此由于在 a + = 1; 中的 + 和 = 之间存在一个空格，所以该语句存在语法错误。

C++ 规定赋值运算符可与下面的二元运算符一起使用：

```
+   -   *   /   %   &   |   ^   <<  >>
```

因此，下面这些新的赋值运算符都是合法的：

```
=   +=  -=  *=  /=  %=  &=  |=  ^=  <<= >>=
```

其中，% 是取模或取余运算符，&、| 和 ^ 分别是位逻辑与、或和异或运算符，<< 和 >> 分别是左移和右移运算符。10.3 节将详细归纳和介绍运算符的含义。对于作用在内置类型运算对象上的二元运算符 @ 来说，表达式 x@=y 等价于 x=x@y，唯一的区别是前者只对 x 求值一次。

函数 term() 执行乘法和除法操作，其方式与 expr() 处理加法和减法的方式一样：

```
double term(bool get)          // 乘法和除法
{
    double left = prim(get);

    for (;;) {
        switch (ts.current().kind) {
        case Kind::mul:
            left *= prim(true);
            break;
        case Kind::div:
            if (auto d = prim(true)) {
                left /= d;
                break;
            }
            return error("divide by 0");
        default:
            return left;
        }
    }
}
```

除数为 0 的除法其结果是未定义的，通常会引发严重的程序错误。因此，我们在实际执行除法操作之前预先检验除数是否为 0，如果是 0 的话调用 error() 函数。10.2.4 节介绍关于 error() 的细节。

当我们确实需要使用变量 d 并且将立即对它进行初始化时，才会把该变量引入程序中。在条件中引入的名字的作用域范围恰好是该条件所控制的语句范围，所得的结果值就是条件的值（见 9.4.3 节）。因此，当且仅当 d 不等于 0 时执行除法赋值语句 left/=d。

函数 prim() 处理初等项（primary）的方式很像 expr() 和 term()。当然，因为我们需要

进入调用层次的下一层，所以在这里需要多做一点事情，而且不必使用循环：

```
double prim(bool get)          // 处理初等项
{
    if (get) ts.get(); // 读取下一个单词

    switch (ts.current().kind) {
    case Kind::number:      // 浮点数常量
    {   double v = ts.current().number_value;
        ts.get();
        return v;
    }
    case Kind::name:
    {   double& v = table[ts.current().string_value];      // 找到对应的项
        if (ts.get().kind == Kind::assign) v = expr(true);  // 看到了 '='：赋值运算
        return v;
    }
    case Kind::minus:       // 一元减法
        return -prim(true);
    case Kind::lp:
    {   auto e = expr(true);
        if (ts.current().kind != Kind::rp) return error("')' expected");
        ts.get();       // 吃掉了 ')'
        return e;
    }
    default:
        return error("primary expected");
    }
}
```

当发现 Token 是一个 number 时（也就是整型或浮点型字面值常量），它的值被置于它的 number_value 中。与之类似，当发现 Token 是一个 name 时（不管如何定义的，见 10.2.2 节和 10.2.3 节），它的值被置于对应的 string_value 中。

相对于分析初等项表达式实际所需的 Token 数量来说，prim() 永远多读取一个。这是因为在某些情况下它必须这么做（比如考察某个名字是否被赋值），因此从一致性的考虑出发，它干脆在所有情况下都统一了起来。当分析函数只想移动到下一个 Token 时，它不需要使用 ts.get() 的返回值。此时，我们可以从 ts.current() 获取结果。如果顾虑忽略掉 get() 的返回值可能会造成某种不便，我们可以添加一个 read() 函数，令其负责更新 current() 而无须返回任何值；或者直接显式地“扔掉”结果：void(ts.get())。

在对某个名字执行具体操作之前，计算器首先向前查看该名字是否被赋值或者只是从中读取了内容。两种情况下都要用到符号表，符号表的类型是 map（见 4.4.3 节和 31.4.3 节）：

```
map<string,double> table;
```

也就是说，当 table 以 string 作为索引时，所得的结果值是与该 string 对应的 double。例如，假定用户的输入是

```
radius = 6378.388;
```

则计算器将进入 case Kind::name 并且执行

```
double& v = table["radius"];
// ... expr() 计算将要赋予的值
v = 6378.388;
```

引用 v 用于保存与 radius 关联的 double，expr() 从输入的字符序列计算得到值 6378.388。

第 14 章和第 15 章将详细讨论如何把程序组织成一组模块。不过在本例中，我们能把计算器示例程序用到的所有声明排列成序，同时保证每个名字都只被声明了一次且声明的位置恰好在使用之前。唯一的一个例外是 expr()，它调用了 term()，term() 调用了 prim()，prim() 又调用了 expr()。在这个循环调用的过程中一定会破坏之前的规则。声明

```
double expr(bool);
```

最好位于 expr() 的定义之前。

10.2.2 输入

读取输入的数据通常是程序中最麻烦的部分。要想和用户进行良好的数据通信，程序必须能应付用户的习惯、产生的各种奇思妙想以及看似随机发生的错误。强迫用户以适合于机器的方式行动不太合乎情理，也很难实现。底层输入模块的任务是读取字符并从数据中生成高一级的单词。这些单词又作为更高层级模块的输入单元。在这里，底层输入由 ts.get() 完成。编写底层输入模块并不难，很多系统都为之提供了标准函数。

首先，我们给出 Token_stream 的完整定义：

```
class Token_stream {
public:
    Token_stream(istream& s) : ip{&s}, owns{false} { }
    Token_stream(istream* p) : ip{p}, owns{true} { }

    ~Token_stream() { close(); }

    Token get();          // 读取并返回下一个单词
    Token& current();     // 刚刚读入的单词

    void set_input(istream& s) { close(); ip = &s; owns=false; }
    void set_input(istream* p) { close(); ip = p; owns = true; }

private:
    void close() { if (owns) delete ip; }

    istream* ip;                  // 指向输入流的指针
    bool owns;                    // Token_stream 有流吗？
    Token ct {Kind::end} ;        // 当前单词
};
```

我们用一个输入流（见 4.3.2 节和第 38 章）初始化 Token_stream，它从该输入流中获取它的字符。Token_stream 占有（并且在最后删除掉，见 3.2.1.2 节和 11.2 节）一个以指针方式传递的 istream，而不是以引用方式传递的 istream，这一点与人们的习惯相符。可能对于我们目前的示例程序来说，这么做有点过于复杂了；但是当指针指向需要析构的资源，而这样的指针被包含在类的内部时，上述技术就显得非常有用了。

Token_stream 保存三个值：一个指向其输入流的指针（ip）、一个用于指示输入流所有权的布尔值（ows）和当前的单词（ct）。

我给 ct 赋予了一个默认值。我们不应该在调用 get() 之前调用 current()，但即使我们真的这样做了，也能得到一个定义良好的 Token。我选择 Kind::end 作为 ct 的初始值，因此，当程序误用 current() 时，得到的值仍然是位于输入流中的，而不是未定义的。

我分两个阶段呈现 Token_stream::get()。首先，我提供一个看似很简单的版本，但是用户要想使用这个版本需要自己做很多事情。因此，接下来我把它修改了一下，修改之后的版本简洁程度略有降低，但是更方便用户使用了。get() 的任务是读入一个字符，根据该字符判断需要组成何种单词，接着读入所需的更多字符，最后返回对应于已读入字符的 Token。

一开始的语句从 *ip（ip 所指的流）读取第一个非空白字符到 ch，并且检查读取操作是否成功：

```
Token Token_stream::get()
{
    char ch = 0;
    *ip>>ch;

    switch (ch) {
    case 0:
        return ct={Kind::end};    // 赋值并返回
```

默认情况下，>> 运算符会跳过空白（即，空格、制表符、换行等），并当输入操作失败时不更改 ch 的值。因此，ch==0 代表输入过程结束。

赋值是一种运算符，赋值的结果是变量被赋予的值。因此，我可以把 {Kind::end} 赋给 curr-tok，然后在同一条语句中返回该值。一条语句比等价的两条语句更易维护。如果赋值和 return 在代码中被分开的话，程序员有可能会更新其中一条而忘记更新另一条。

还要注意 {} 列表形式（见 3.2.1.3 节和 11.3 节）是如何用在赋值运算符右侧的。也就是说，它是一条表达式。我有可能把 return 语句写成下面的形式：

```
ct.kind = Kind::end; // 赋值
return ct;           // 返回
```

然而，我认为赋值一个完整的对象 {Kind::end} 要比单独处理 ct 的每个成员在逻辑上更清晰明了。{Kind::end} 等价于 {Kind::end,0,0}。如果我们很在意后两个成员的值，则第二种形式更好；反之，如果我们更在意性能，则第一种形式更好。两种形式都没有在接下来的程序中出现，但是一般情况下，处理完整的对象与单独维护每个数据成员相比更清晰、更不易出错。下面的例子采用的是另一种策略。

在我们完整地实现函数之前，不妨先分开来考虑一些个例。其中，表达式终结符（;）、括号和运算符的处理方式很简单，直接返回它们的值即可：

```
case ';':  // 表达式终结符，输出
case '*':
case '/':
case '+':
case '-':
case '(':
case ')':
case '=':
    return ct={static_cast<Kind>(ch)};
```

因为在 char 和 Kind 之间没有隐式类型转换规则（见 8.4.1 节），所以必须使用 static_cast（见 11.5.2 节）。仅有一少部分字符对应 Kind 的值，因此我们必须“确保”此例中的 ch 是符合要求的。

数字的处理方式是：

```
case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9':
case '.':
ip->putback(ch);          // 把第一个数字（或者.）放回到输入流中
>ip >> ct.number_value; // 把数字读入 ct
ct.kind=Kind::number;
return ct;
```

通常情况下我们不应该横排 case 标签，因为与纵排方式相比，横排的标签很难阅读和理解。但是在此例中，如果每行只保留一个数字的话看起来更繁琐。由于 >> 运算符的定义，它可以把浮点值读入 double 中，因此上面的代码与我们习惯的操作方式没什么两样。一开始的字符（数字或者点）被放回 cin，然后浮点值被读入 ct.number_value 中。

如果单词不是输入结束符、运算符、标点符号或者数字，则它必然是一个名字。处理名字的方式与数字类似：

```
default:                  // 名字，名字=，或者错误
    if (isalpha(ch)) {
        ip->putback(ch);              // 把第一个字符放回输入流中
        >ip>>ct.string_value;         // 把 string 读入 ct
        ct.kind=Kind::name;
        return ct;
    }
```

最后，我们也可能会得到一个错误。处理错误的一种虽然简单但非常有效的方法是调用 error() 函数，然后返回一个 print 单词：

```
error("bad token");
return ct={Kind::print};
```

标准库函数 isalpha()（见 3.6.2.1 节）可以令我们不必把每个字符作为一个单独的 case 标签。>> 运算符在字符串（此例中是 string_value）内连续读取直到遇到一个空白符时停止。因此，在运算符使用某个名字作为它的运算对象之前，用户必须加上一个空白符以表示名字的结束。这种方式似乎还不太理想，我们将在 10.2.3 节继续讨论这个问题。

下面是完整的输入函数：

```
Token Token_stream::get()
{
    char ch = 0;
    >ip>>ch;

    switch (ch) {
    case 0:
        return ct={Kind::end};          // 赋值并返回
    case ';':  // 表达式终结符，输出
    case '*':
    case '/':
    case '+':
    case '-':
    case '(':
    case ')':
    case '=':
        return ct=={static_cast<Kind>(ch)};
    case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9':
    case '.':
        ip->putback(ch);              // 把第一个数字（或者.）放回输入流中
        >ip >> ct.number_value;       // 把数字读入 ct
```

```
            ct.kind=Kind::number;
            return ct;
        default:                    // 名字，名字=，或者错误
            if (isalpha(ch)) {
                ip->putback(ch);            // 把第一个字符放回输入流中
                *ip>>ct.string_value;       // 把 string 读入 ct
                ct.kind=Kind::name;
                return ct;
            }

            error("bad token");
            return ct={Kind::print};
        }
    }
```

因为我们把运算符的 kind 定义成该运算符对应的整数值（见 10.2.1 节），所以运算符向其 Token 值的类型转换非常自然，没什么特别之处。

10.2.3　底层输入

到目前为止，我们实现的计算器还有一些不足。首先，为了输出表达式的值，我们必须在表达式之后加上一个分号；其次，用空白符分隔名字的方式会带来很多麻烦。例如，在目前的系统中 x=7 是一个标识符，而非标识符 x 之后跟上运算符 = 和数字 7。要想得到我们希望的结果，必须在 x 后面加上一个空格：x =7。解决这两个问题的办法是：在函数 get() 中，用依次读取单个字符的方式替换面向类型的默认输入操作。

首先，我们用换行等价替代分号指示表达式的结束：

```
Token Token_stream::get()
{
    char ch;

    do { // 跳过除 '\n' 之外的其他空白符
        if (!ip->get(ch)) return ct={Kind::end};
    } while (ch!='\n' && isspace(ch));

    switch (ch) {
    case ';':
    case '\n':
        return ct={Kind::print};
```

在这里，我使用了 do 语句。它与 while 语句非常类似，唯一的区别是循环的受控语句部分至少会执行一次。当调用 ip->get(ch) 时，从输入流 *ip 读取一个字符存放到 ch 中。默认情况下，get() 不会像 >> 那样跳过空白符。如果 cin 中没有字符可供读取，则 if (!ip->get(ch)) 的条件满足；此例中，程序返回 Kind::end 以结束计算器的执行。因为当 get() 执行成功时返回 true，所以这里需要加上一个 ! 运算符（非）。

标准库函数 isspace() 提供了对空白符的标准检测方法（见 36.2.1 节）；如果 c 是一个空白字符，则 isspace(c) 返回一个非 0 值；否则，返回 0。标准检测采用的是表格查找的方式，因此使用 isspace() 比单独检测每个空白字符快得多。类似的检测函数还有 isdigit()（是否是数字）、isalpha()（是否是字母）和 isalnum()（是否是数字或者字母）。

跳过空白符后，下一个字符决定接下来的单词是什么类型。

>> 运算符的机制是读取字符串的内容直到遇到空白符为止。为了解决这一问题，我们

每次只读取一个字符，当该字符不是字母或者数字时终止读取过程：

```
default:                    // 名字，名字=，或者错误
    if (isalpha(ch)) {
        string value = ch;
        while (ip->get(ch) && isalnum(ch))
            string_value += ch; // 把 ch 加到 string_value 的末尾
        ip->putback(ch);
        return ct={Kind::name};
    }
```

幸运的是，我们只需要修改代码的一个区域就能实现上述两种对读取操作的提升。构造一个程序使得后续所做的修改都能限制在较小的局部范围内，是程序设计的重要目标。

你可能会担心把字符逐一添加到 string 末尾的做法不太高效。它可能会是一个长的 string，但是所有现代的 string 实现都提供了“小字符串优化”（见 19.3.3 节）。这意味着我们在处理计算器（甚至是编译器）中的名字时不会包含任何低效的操作。尤其是使用一个短 string 不会涉及任何自由存储空间。在这里，短 string 允许的最大字符数是依赖于实现的，但 14 是个比较理想的值。

10.2.4 错误处理

任何程序都必须具备检测并报告错误的功能。然而就这个程序来说，一个简单的异常处理策略就足够了。error() 函数负责统计错误数量、输出错误消息，然后返回：

```
int no_of_errors;

double error(const string& s)
{
    no_of_errors++;
    cerr << "error: " << s << '\n';
    return 1;
}
```

其中，cerr 是一个不带缓冲的输出流，常用于报告错误（见 38.1 节）。

error() 函数之所以要返回一个值，是因为错误通常发生在表达式的求值过程中，所以，我们要么放弃整个求值过程，要么返回一个不会造成后续错误的值。后者对于简单计算器程序来说足够了。令 Token_stream::get() 持续追踪行号，则 error() 可以通过错误消息报告错误发生的具体位置。当计算器以无交互的方式运行时，这一设定尤其有用。

另一种更规范且更通用的错误处理策略是把错误检测和错误恢复分离开来，我们用异常（见 2.4.3.1 节和第 13 章）来实现这种策略。但是仅就一个 180 行的计算器程序而言，上面的 error() 函数已经足够了。

10.2.5 驱动程序

在程序的所有模块都编写完成后，我们还需要一个驱动程序控制程序开始执行。我实现了两个函数：main() 负责启动程序及报告错误；calculate() 负责完成实际的计算任务：

```
Token_stream ts {cin};  // 使用 cin 中的输入流

void calculate()
{
    for (;;) {
```

```
            ts.get();
            if (ts.current().kind == Kind::end) break;
            if (ts.current().kind == Kind::print) continue;
            cout << expr(false) << '\n';
        }
    }

    int main()
    {
        table["pi"] = 3.1415926535897932385;    // 插入预先定义的名字
        table["e"] = 2.7182818284590452354;

        calculate();

        return no_of_errors;
    }
```

通常情况下，main() 函数返回 0 表示程序被正确执行，返回非 0 表示程序出现了错误（见 2.2.1 节）。我们之所以在 error() 中统计错误出现的数量，就是为了这个目的。程序开始执行后，唯一需要的初始化操作就是把预先定义的名字插入到符号表中。

主循环（在函数 calculate() 中）的核心任务是读取表达式并输出答案，我们用下面的代码行来实现：

```
cout << expr(false) << '\n';
```

实参 false 告诉 expr() 它无须调用 ts.get() 来读取单词。

在程序中检测 Kind::end 可以确保当 ts.get() 遇到输入错误或程序末尾时循环可以正常结束。break 语句跳出离它最近的外层 switch 语句或者循环（见 9.5 节）。检测 Kind::print（即，'\n' 和 ';'）可以让 expr() 不必再专门处理空字符串的情形。continue 语句的作用是直接跳到当前循环的末尾。

10.2.6　头文件

计算器程序使用了标准库功能。因此，在程序中必须 #include 用到的头文件：

```
#include<iostream> // I/O
#include<string>   // string
#include<map>      // map
#include<cctype>   // isalpha() 等
```

这些头文件提供的功能都位于 std 名字空间中，因此要想使用它们提供的名字，我们要么显式地使用限定符 std::，要么通过下面的语句把这些名字引入到全局名字空间中：

```
using namespace std;
```

为了把关于表达式和模块化的讨论区分开来，我将在后面的章节中专门介绍模块化的内容。第 14 章和第 15 章讨论如何用名字空间的方式把计算器组织成模块以及如何把这些模块进一步组合成源文件。

10.2.7　命令行参数

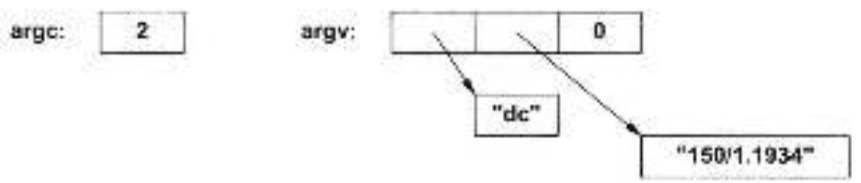
在编写完程序并且测试通过之后，接下来的问题是如何启动程序、键入准备求值的表达式、最后退出程序。该程序最常用的形式是接受一条表达式并且求它的值，因此如果我们能

把该表达式表示成一个命令行参数，则可以减少一些不必要的录入操作。

程序通过调用 main() 启动（见 2.2.1 节和 15.4 节）。之后 main() 被传入两个实参，分别是：argc 指明实参的数量，argv 代表实参的数组。这里的实参是 C 风格的字符串（见 2.2.5 节和 7.3 节），因此 argv 的类型是 char*[argc+1]。argv[0] 表示程序的名字（当它出现在命令行时），因此 argc 的值至少是 1。实参列表以 0 作为结束符，即，argv[argc]==0。例如，对于命令

```
dc 150/1.1934
```

实参的值包括：



因为以调用 main() 函数启动程序的方式是从 C 语言继承而来的，所以我们用到了 C 风格的数组和字符串。

基本的思想是像从输入流读取数据一样读取命令行字符串的内容，从字符串读取数据的流当然应该叫做 istringstream（见 38.2.2 节）。因此，我们只要让 Token_stream 从正确的 istringstream 读取数据，就能计算存在于命令行中的表达式了：

```
Token_stream ts {cin};

int main(int argc, char* argv[])
{
    switch (argc) {
    case 1:                                          // 从标准输入读取数据
        break;
    case 2:                                          // 从实参字符串读取数据
        ts.set_input(new istringstream(argv[1]));
        break;
    default:
        error("too many arguments");
        return 1;
    }

    table["pi"] = 3.1415926535897932385;             // 输入预先定义好的名字
    table["e"] = 2.7182818284590452354;

    calculate();

    return no_of_errors;
}
```

把头文件 <sstream> 包含进程序中来，这样才能使用 istringstream。

稍作修改就能让 main() 函数接受多个命令行实参，但是这么做其实没什么必要，毕竟一个实参也可以传递多个表达式：

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

在我的 UNIX 系统中，符号 ; 表示命令分隔符；其他系统有可能使用另外的符号。

尽管 argc 和 argv 从形式到用法都非常简单，但它们仍有可能造成一些微小但恼人的问题。为了避免这些问题发生，尤其是为了便于传递和分发程序的实参，我习惯用一个简单的函数创建一个 vector<string>：

```
vector<string> arguments(int argc, char* argv[])
{
    vector<string> res;
    for (int i = 0; i!=argc; ++i)
        res.push_back(argv[i]);
    return res;
}
```

对于一个分析实参的函数来说，这种规模已经足够，再复杂的话就不实用了。

10.2.8　关于风格

对于不熟悉关联数组的程序员来说，采用标准库 map 作为符号表似乎有些投机取巧。事实当然不是这样。标准库和其他库本来就是供人使用的。通常情况下，设计并实现一个库所付出的努力，远远多于程序员在自己的程序中手工编写一个代码片段所需的投入。二者不可等量齐观。

请看计算器的代码，特别是第一个版本。我们可以看到这里并没有多少 C 风格的、低级别的代码。许多传统的、技术性的细节都被标准库中的类 ostream、string 和 map（见 4.3.1 节，4.2 节，4.4.3 节，31.4 节，第 36 章和第 38 章）取代了。

请注意，这里的算术运算、循环和赋值操作都比较少。对于那些并不直接操作硬件或者实现低级抽象的代码来说，这是应有的形式。

10.3　运算符概述

本节简要介绍表达式并给出一些示例。每个运算符对应一个或者几个常用的名字，以及一个用来解释其用法的示例。在表格中：

- *名字*（name）可能是一个标识符（比如 sum 和 map），一个运算符名字（比如 operator int、operator+ 和 operator"" km），或者是一个模板特例的名字（比如 sort<Record> 和 array<int,10>）。名字前面可能用 :: 加以限定（std::vector 和 vector<T>::operator[]）。
- *类名字*（class-name），包括 decltype(expr) 在内（其中 expr 表示一个类）。
- *成员*（member）是指成员的名字（包括析构函数的名字及成员模板的名字）。
- *对象*（object）是指产生类对象的表达式。
- *指针*（pointer）是指产生指针的表达式（包括 this 以及支持指针操作的类型的对象）。
- *表达式*（expr），包括字面值常量（比如 17、"mouse" 和 true）。
- *表达式列表*（expr-list），可能为空。
- *左值*（lvalue）是指对应一个可修改对象（见 6.4.1 节）的表达式。
- *类型*（type），如果类型出现在一对括号内，则它可以是一个完整的通用类型名字（含有 *、() 等）；否则，对它有一些严格的限制（§ iso.A）。
- lambda *声明符*（lambda-declarator）是（可能为空的、以逗号隔开的）参数列表，后面可跟 mutable 修饰符、noexcept 修饰符或者返回类型（见 11.4 节）。

- 捕获列表（capture-list）是一个可能为空的列表，指定上下文相关的内容（见11.4节）。
- 语句列表（stmt-list），可能为空（见2.2.4节和第9章）。

表达式的语法与运算对象的类型无关。下表列出的是当运算对象是内置类型时对应的运算符含义（见6.2.1节）此外，你可以指定这些运算符用于用户自定义类型时的含义（见2.3节和第18章）。

下表近似地概括了语法规则，更详细的信息请见 § iso.5 和 § iso.A。

运算符一览（§iso.5.1）		
括号表达式	(*expr*)	
lambda 表达式	[*capture-list*] *lambda-declarator* { *stmt-list* }	11.4 节
作用域解析	*class-name* :: *member*	16.2.3 节
作用域解析	*namespace-name* :: *member*	14.2.1 节
全局	:: *name*	14.2.1 节
成员选择	*object* . *member*	16.2.3 节
成员选择	*pointer* -> *member*	16.2.3 节
取下标	*pointer* [*expr*]	7.3 节
函数调用	*expr* (*expr-list*)	12.2 节
值构造	*type* { *expr-list* }	11.3.2 节
函数形式的类型转换	*type* (*expr-list*)	11.5.4 节
后置递增	*lvalue* ++	11.1.4 节
后置递减	*lvalue* --	11.1.4 节
类型识别	typeid (*type*)	22.5 节
运行时类型识别	typeid (*expr*)	22.5 节
运行时检查的类型转换	dynamic_cast < *type* > (*expr*)	22.2.1 节
编译时检查的类型转换	static_cast < *type* > (*expr*)	11.5.2 节
不检查的类型转换	reinterpret_cast < *type* > (*expr*)	11.5.2 节
const 转换	const_cast < *type* > (*expr*)	11.5.2 节
对象尺寸	sizeof *expr*	6.2.8 节
类型尺寸	sizeof (*type*)	6.2.8 节
参数包尺寸	sizeof... *name*	28.6.2 节
类型对齐	alignof (*type*)	6.2.9 节
前置递增	++ *lvalue*	11.1.4 节
前置递减	-- *lvalue*	11.1.4 节
补	~ *expr*	11.1.2 节
非	! *expr*	11.1.1 节
一元负号	- *expr*	2.2.2 节
一元正号	+ *expr*	2.2.2 节
取地址	& *lvalue*	7.2 节
解引用	* *expr*	7.2 节
创建（分配）	new *type*	11.2 节
创建（分配并初始化）	new *type* (*expr-list*)	11.2 节
创建（分配并初始化）	new *type* { *expr-list* }	11.2 节

（续）

运算符一览（§iso.5.1）		
创建（放置）	new (*expr-list*) *type*	11.2.4 节
创建（放置并初始化）	new (*expr-list*) *type* (*expr-list*)	11.2.4 节
创建（放置并初始化）	new (*expr-list*) *type* { *expr-list* }	11.2.4 节
销毁（释放）	delete *pointer*	11.2 节
销毁数组	delete [] *pointer*	11.2.2 节
允许表达式抛出异常吗	noexcept (*expr*)	13.5.1.2 节
强制类型转换	(*type*) *expr*	11.5.3 节
成员选择	*object* .* *pointer-to-member*	20.6 节
成员选择	*pointer* ->* *pointer-to-member*	20.6 节
乘法	*expr* * *expr*	10.2.1 节
除法	*expr* / *expr*	10.2.1 节
取模（取余）	*expr* % *expr*	10.2.1 节
加法	*expr* + *expr*	10.2.1 节
减法	*expr* – *expr*	10.2.1 节
左移	*expr* << *expr*	11.1.2 节
右移	*expr* >> *expr*	11.1.2 节
小于	*expr* < *expr*	2.2.2 节
小于等于	*expr* <= *expr*	2.2.2 节
大于	*expr* > *expr*	2.2.2 节
大于等于	*expr* >= *expr*	2.2.2 节
等于	*expr* == *expr*	2.2.2 节
不等于	*expr* != *expr*	2.2.2 节
位与	*expr* & *expr*	11.1.2 节
位异或	*expr* ^ *expr*	11.1.2 节
位或	*expr* | *expr*	11.1.2 节
逻辑与	*expr* && *expr*	11.1.1 节
逻辑或	*expr* || *expr*	11.1.1 节
条件表达式	*expr* ? *expr* : *expr*	11.1.3 节
列表	{ *expr-list* }	11.3 节
抛出异常	throw *expr*	13.5 节
简单赋值	*lvalue* = *expr*	10.2.1 节
相乘并赋值	*lvalue* *= *expr*	10.2.1 节
相除并赋值	*lvalue* /= *expr*	10.2.1 节
取模并赋值	*lvalue* %= *expr*	10.2.1 节
相加并赋值	*lvalue* += *expr*	10.2.1 节
相减并赋值	*lvalue* –= *expr*	10.2.1 节
左移并赋值	*lvalue* <<= *expr*	10.2.1 节
右移并赋值	*lvalue* >>= *expr*	10.2.1 节
位与并赋值	*lvalue* &= *expr*	10.2.1 节
位或并赋值	*lvalue* |= *expr*	10.2.1 节
位异或并赋值	*lvalue* ^= *expr*	10.2.1 节
逗号（序列）	*expr* , *expr*	10.3.2 节

同一个方块中的运算符具有相同的优先级，运算符所在的方块位置越靠前，它的优先级越高。例如，N::x.m 的含义是 (N::m).m 而非 N::(x.m)。

例如，后置 ++ 的优先级高于一元 *，因此 *p++ 的意思是 *(p++) 而非 (*p)++。

再比如，因为 * 的优先级高于 +，所以 a+b*c 的意思是 a+(b*c) 而非 (a+b)*c。

一元运算符和赋值运算符是右结合的，其他所有运算符都是左结合的。例如，a=b=c 的意思是 a=(b=c)，而 a+b+c 的意思是 (a+b)+c。

有些语法规则无法通过优先级（也称为绑定强度）和结合律表达出来。例如，a=b<c?d=e:f=g 的含义是 a=((b<c)?(d=e):(f=g))，但是你需要查阅相关的语法规定（§iso.A）才能正确理解它的含义。

在应用语法规则之前，先用字符构成单词。我们选择最长可能的字符序列作为单词。例如，&& 是一个运算符，而非两个 & 运算符；a++++1 的意思是 (a ++) + 1。这种构词方式称为最长匹配规则（Max Munch rule）。

单词一览（§iso.2.7）

单词类别	示例	出处
标识符	vector, foo_bar, x3	6.3.3 节
关键字	int, for, virtual	6.3.3.1 节
字符字面值常量	'x', '\n', U'\UFADEFADE'	6.2.3.2 节
整数字面值常量	12, 012, 0x12	6.2.4.1 节
浮点数字面值常量	1.2, 1.2e-3, 1.2L	6.2.5.1 节
字符串字面值常量	"Hello!", R"("World"!)"	7.3.2 节
运算符	+=, %, <<	10.3 节
标点符号	;, ., {, }, ( )	
预处理符号	#, ##	12.6 节

空白字符（比如空格、制表符和换行）能用来分隔单词（比如 int count 表示一个关键字跟着一个标识符，而不是 intcount），在其他情况下则被直接忽略掉。

基本源字符集（见 6.1.2 节）中的某些字符在有的键盘上不易键入，而且有的程序员不喜欢用 && 和 ~ 等符号表示逻辑运算。因此，我们设计了一组关键词作为等效的替代。

替代形式（§iso.2.12）

and	and_eq	bitand	bitor	compl	not	not_eq	or	or_eq	xor	xor_eq
&&	&=	&	|	~	!	!=	||	|=	^	^=

例如

```
bool b = not (x or y) and z;
int x4 = ~ (x1 bitor x2) bitand x3;
```

等价于

```
bool b = !(x || y) && z;
int x4 = ~(x1 | x2) & x3;
```

请注意，and= 与 &= 不等价；如果你想用一个关键词表示 &=，应该用 and_eq。

10.3.1　结果

算术运算符的结果类型由一组称为“常见算术类型转换”的规则决定（见 10.5.3 节）。这些规则的基本目标是产生“最大的”运算对象类型的结果。例如，如果一个二元运算符有一个浮点型运算对象，则相应的运算基于浮点数运算规则执行，所得的结果也是一个浮点值。类似地，如果它有一个 long 运算对象，则运算基于长整型运算规则进行，所得的结果类型是 long。在开始执行运算前，尺寸小于 int 的运算对象（比如 bool 和 char）先转换成 int 类型。

关系运算符（==、<= 等）的结果是布尔值。用户自定义运算符的含义和结果类型由运算符的声明本身决定（见 18.2 节）。

只要逻辑上说得通，对于接受左值运算对象的运算符来说，它的结果是一个表示该左值运算对象的左值。例如：

```
void f(int x, int y)
{
    int j = x = y;          //x=y 的值是 x 在执行赋值运算之后的结果值
    int* p = &++x;          //p 指向 x
    int* q = &(x++);        //错误：x++ 不是一个左值（它不是存储在 x 中的值）
    int* p2 = &(x>y?x:y);   //具有较大值的 int 的地址
    int& r = (x<y)?x:1;     //错误：1 不是左值
}
```

如果 ?: 的第二个和第三个运算对象都是左值且类型相同，则该运算符的运算结果是一个同类型的左值。在这种方式下，左值的性质得以保留，因而在使用运算符时拥有更大的灵活性。尤其当我们需要以统一高效的方式同时处理内置类型和用户自定义的类型时（比如编写模板和生成 C++ 代码的程序），上述方式显得特别有用。

sizeof 的结果是名为 size_t 的无符号整数类型，该类型定义在 <cstddef> 中。两个指针相减的结果是名为 ptrdiff_t 的带符号整数类型，它同样定义在 <cstddef> 中。

C++ 的具体实现既不必也不会检查算术运算溢出的问题，例如：

```
void f()
{
    int i = 1;
    while (0 < i) ++i;
    cout << "i has become negative!" << i << '\n';
}
```

这段代码不断递增 i 的值，直到最终超过 i 的最大表示范围。接下来发生的事情是未定义的，但是通常 i 的值会“绕一圈”变成一个负值（在我的机器上是 –2147483648）。类似地，除数为 0 的结果也是未定义的，遇到这种情况时程序一般会突然中断。下溢、上溢和除数为 0 的错误不会抛出标准异常（见 30.4.1.1 节）。

10.3.2　求值顺序

C++ 没有明确规定表达式中子表达式的求值顺序，尤其请注意，你不能假定表达式是按照从左到右的顺序求值的。例如：

```
int x = f(2)+g(3);      // 到底先调用 f() 还是先调用 g() 并没有明确的规定
```

不限定表达式的求值顺序有助于生成更好的代码，但是有时也会带来一些问题。例如：

```
int i = 1;
v[i] = i++; // 未定义的结果
```

其中的赋值操作既可能执行为 v[1]=1，也可能执行为 v[2]=1，甚至会产生非常奇怪的运行结果。编译器也许能发现并报告这类二义性操作，但事实上大多数编译器都会置之不理。因此，一定要避免在同一条表达式中同时读写一个对象。除非你只用到了一个运算符（比如 ++ 和 +=），或者显式地表达出了序列的含义（比如使用了逗号、&& 或者 ||）。

逗号运算符（,）、逻辑与运算符（&&）和逻辑或运算符（||）规定它们的左侧运算对象先被求值，然后才是右侧运算对象。例如，b=(a=2,a+1) 的意思是把 3 赋给 b。10.3.3 节会介绍 || 和 && 的用法示例。对于内置类型来说，只有当 && 的第一个运算对象是 true 时才会求第二个运算对象的值，同理，只有当 || 的第一个运算对象是 false 时才会求第二个运算对象的值。这种现象称为短路求值（short-circuit evaluation）。请注意，序列运算符（逗号）与调用函数时分隔实参的逗号在逻辑上完全是两回事。例如：

```
f1(v[i],i++);        // 两个实参
f2( (v[i],i++) );    // 一个实参
```

f1 的调用语句包含两个实参 v[i] 和 i++，C++ 并没有明确规定这两个实参表达式的求值顺序。因此，这种用法应该尽量避免。依赖实参表达式求值顺序会产生未定义的结果，很不可取。f2 的调用语句只含有一个实参，即逗号表达式 (v[i],i++)，它的效果等价于 i++。这种写法具有一定的迷惑性，也应该尽量避免。

我们用括号把表达式的某一部分强行结合在一起。例如，a*b/c 本身的含义是 (a*b)/c，因此要想求 a*(b/c) 的值必须加上括号。只有当用户也分辨不出其中的差异时，a*(b/c) 才会以 (a*b)/c 的顺序求值。但是对于很多浮点类型来说，a*(b/c) 和 (a*b)/c 根本不一样，所以编译器会根据用户书写的形式求值。

10.3.3 运算符优先级

优先级层级和结合律法则影响着很多常见的用法，例如：

```
if (i<=0 || max<i) // ...
```

它的含义是“如果 i 小于等于 0 或者如果 max 小于 i”，也就是说它等价于：

```
if ( (i<=0) || (max<i) ) // ...
```

而不是下面这个貌似合法，但其实是一派胡言的式子：

```
if (i <= (0||max) < i) // ...
```

然而，如果程序员对某些规则不太有把握，最好使用括号来把他的意思表达清楚。当子表达式变得很复杂时，我们通常会加上一些括号；但是复杂的子表达式通常意味着错误的几率也随之上升。因此，如果你感觉需要使用括号了，其实最好的办法反而是通过一个额外的变量把长表达式截断成较短的表达式。

存在一些情况，运算符实际的优先级与看上去“很明显的”理解不符，例如：

```
if (i&mask == 0)    // 错误！ —— 表达式是 & 的一个运算对象
```

这行代码实际上并不是把 mask 添加到 i 上，然后判断结果是否为 0。因为 == 的优先级高于 &，因此该表达式的含义是 i&(mask==0)。幸运的是，编译器很容易发现并报告此类错误。在此例中，括号显得非常重要：

```
if ((i&mask) == 0) // ...
```

请特别注意，下面的式子虽然从数学的角度看很合理，但在程序中完全不是这样：

```
if (0 <= x <= 99) // ...
```

该式在语法上是正确的，但它的含义是 (0<=x)<=99。即，第一个不等式的结果或者是 false 或者是 true，该布尔值隐式地转换成 0 或者 1，然后再与 99 比较大小，最终的结果是 true。要想检查 x 是否在 0...99 之间，应该写成：

```
if (0<=x && x<=99) // ...
```

新手常犯的一个错误是在条件语句中把 ==（相等）错写成 =（赋值）：

```
if (a = 7)  // 哎哟！在条件语句中赋了一个常量值
```

因为符号 = 在很多语言中都表示“相等”，所以这个错误一点儿也不奇怪。再说一次，编译器很容易发现并报告这类错误，而且很多编译器确实会这样做。我不建议你为了让代码通过编译而改变常规的书写方式，尤其是下面这种形式不值得提倡：

```
if (7 == a) // 试图避免误用 =，不推荐这种写法
```

10.3.4　临时对象

通常情况下，编译器必须引入一个对象，用以保存表达式的中间结果。例如，对于表达式 v=x+y*z 来说，在把 y*z 的结果加到 x 上之前必须暂时存在某处。内置类型使用临时对象（temporary object，简称为临时量）实现上述要求，并且该临时对象是用户不可见的。然而，对于含有某些资源的用户自定义类型而言，了解并掌握临时量的生命周期非常重要。除非我们把临时对象绑定到引用上或者用它初始化一个命名对象，否则大多数时候在临时对象所在的完整表达式末尾，它就被销毁了。完整表达式（full expression）不是任何其他表达式的子表达式。

标准库 string 有一个名为 c_str()（见 36.3 节）的成员，该成员返回的结果是一个指向以 0 结束的字符数组（见 2.2.5 节和 43.4 节）的 C 风格指针。另外，它还定义运算符 + 来执行字符串连接的操作。这些都是 string 很有用的功能。然而，如果简单地把这些功能组合在一起有可能会带来意料之外的问题。例如：

```
void f(string& s1, string& s2, string& s3)
{
    const char* cs = (s1+s2).c_str();
    cout << cs;
    if (strlen(cs=(s2+s3).c_str())<8 && cs[0]=='a') {
        // 使用 cs
    }
}
```

也许你的第一反应是“别这么写”，我深表同意。但是类似的代码确实存在，因此我们还是有必要研究一下它到底是什么含义。

程序创建了一个临时的 string 对象保存 s1+s2 的结果，然后从该对象引出一个 C 风格字符串的指针。在表达式的末尾，该临时对象被删除。然而，c_str() 返回的 C 风格字符串是作为保存 s1+s2 结果的临时对象的一部分被分配的，我们无法确保当临时对象被销毁后，一定会退出该部分区域。因此，cs 可能会指向一片已被释放的存储空间，而输出操作 cout<<cs 是否能正常工作就完全看运气了。编译器能检测并报告很多类似的错误。

if 语句的问题更微妙一些。因为保存 s2+s3 结果的临时对象所在的完整表达式是在条件内部创建的，所以该条件会按照我们预期的方式工作。不过，在程序进入受控语句之前，这个临时变量就被销毁了，所以在受控语句内对 cs 的使用无法保证有效。

请注意，就像很多其他场合一样，这个例子之所以在使用临时变量时发生了问题，是因为程序以一种低层级的方式使用了高层级的数据类型。如果采用一种更清晰的程序设计风格，则不但能得到更易理解的程序片段，而且可以完全避免由临时对象带来的问题。例如：

```
void f(string& s1, string& s2, string& s3)
{
    cout << s1+s2;
    string s = s2+s3;
    if (s.length()<8 && s[0]=='a') {
        // 使用 s
    }
}
```

临时量可以用作 const 引用或者命名对象的初始化器，例如：

```
void g(const string&, const string&);

void h(string& s1, string& s2)
{
    const string& s = s1+s2;
    string ss = s1+s2;

    g(s,ss);   // 我们可以在此处使用 s 和 ss
}
```

这段代码非常完美。当临时量对应的引用或者命名对象超出了作用域范围时，该临时量被销毁。切记试图返回局部变量的引用会造成程序错误（见 12.1.4 节），并且也不允许把一个临时变量绑定到非 const 左值引用上（见 7.7 节）。

我们可以使用构造函数（见 11.5.1 节）在表达式内部显式地创建临时对象，例如：

```
void f(Shape& s, int n, char ch)
{
    s.move(string(n,ch));    // 构造一个有 n 个 ch 的拷贝的字符串，传递给 Shape::move()
    // ...
}
```

销毁这类临时量的方式与销毁隐式生成的临时量的方式完全一致。

10.4 常量表达式

C++ 提供了两种与“常量”有关的概念：

- constexpr：编译时求值（见 2.2.3 节）。
- const：在作用域内不改变其值（见 2.2.3 节和 7.5 节）。

基本上，constexpr 的作用是启用并确保编译时求值，而 const 的主要作用是在接口中规定某些成分不可修改。本节主要关注第一个问题：编译时求值。

常量表达式（constant expression）是指由编译器求值的表达式。它不能包含任何编译时未知的值，也不能具有其他副作用。一条常量表达式由整数值（见 6.2.1 节）、浮点数值（见 6.2.5 节）或者枚举值（见 8.4 节）等成分构成，我们可以用运算符或者（接受这些值，反过来又）生成常量值的 constexpr 函数把这些基本成分组合在一起。另外，某些常量表达式中

还可以出现地址值。出于简化问题的考虑，我将在 10.4.5 节再专门讨论这一内容。

在很多情况下，人们希望使用命名的常量而非字面值常量或者变量中的值。这是因为：

[1] 命名常量使得代码易于理解和维护。

[2] 变量的值可能会被修改（因此与常量相比，我们在推导时必须更小心）。

[3] C++ 语言要求数组的尺寸、case 标签和 template 值实参使用常量。

[4] 嵌入式系统程序员喜欢把不可修改的数据置于只读内存中，因为与动态内存相比，只读内存更廉价（一方面指价格本身，另一方面指所需的能源消耗）、空间也更大。此外，即使系统崩溃，只读内存的数据也基本上不受影响。

[5] 如果在编译时完成了初始化操作，则即使在多线程系统中也不会发生针对该对象的数据竞争。

[6] 有时，在编译时求值一次比在运行时求值上百万次的效率高得多。

请注意，原因 [1]、[2]、[5] 和原因 [4] 的一部分都是出于逻辑上的考虑。我们之所以使用常量表达式并非因为我们只关注程序的性能问题。很多情况下，常量表达式明显更符合系统的要求。

作为数据项（此处，我特意没有使用“变量”这个词）定义的一部分，constexpr 表达了编译时求值的意愿。如果 constexpr 的初始化器无法在编译时求值，则编译器将报错。例如：

```
int x1 = 7;
constexpr int x2 = 7;

constexpr int x3 = x1;          // 错误：初始化器不是常量表达式
constexpr int x4 = x2;          // OK

void f()
{
    constexpr int y3 = x1;      // 错误：初始化器不是常量表达式
    constexpr int y4 = x2;      // OK
    // ...
}
```

聪明的编译器能推断出作为 x3 初始化器的 x1，其值为 7。但是，显然我们不应该把程序的对错完全寄托在编译器的“聪明程度”上。在大型程序中，要想在编译时确定变量的值通常非常困难，基本上不可能。

常量表达式的表达能力异常丰富。我们可以使用整数、浮点数和枚举值，还可以使用任何不会修改状态的运算符（比如 +、?: 和 [] 可以，但是 = 和 ++ 不行）。通过使用 constexpr 函数（见 12.1.6 节）和字面值类型（见 10.4.3 节），能显著提升类型安全性以及表达能力，这一点比常用的宏（见 12.6 节）强出很多。

条件表达式运算符 ?: 的含义是根据一条常量表达式的值进行选择。例如，我们能在编译时计算某个整数的平方根：

```
constexpr int isqrt_helper(int sq, int d, int a)
{
    return sq <= a ? isqrt_helper(sq+d,d+2,a) : d;
}

constexpr int isqrt(int x)
{
```

```
    return isqrt_helper(1,3,x)/2 - 1;
}

constexpr int s1 = isqrt(9);          // s1 的值变为 3
constexpr int s2 = isqrt(1234);
```

我们先对 ?: 表达式的条件进行求值，然后继续对选出的项求值。没有被选中的项不会求值，甚至可能不是一条常量表达式。类似地，运算符 && 和 || 的运算对象如果没有被求值，也不必是常量表达式。这一特性对于 constexpr 函数尤其有用，因为它有时被用作常量表达式，有时又不是。

10.4.1 符号化常量

常量（constexpr 或者 const 值）最重要的一个用处是为值提供符号化的名字。我们应该在代码中有意识地使用符号化常量以避免出现“魔法数字”。散布在代码中的字面值给维护工作带来极大的困难。以表示数组规模的数字为例，如果它在程序中反复出现，则一旦需要修改该数字的值，我们将不得不找到它每一次出现的位置然后修改。使用符号化的名字可以起到把信息局部化的作用。通常情况下，一个数字型常量表示对程序的一个假定。例如，4 可能表示整数所占的字节数，128 是缓冲输入所需的字节数，6.24 则是计麦克朗和美元的汇率。如果程序中到处是这样的数字，那么任何人都很难理解它们的含义并且维护该程序。此外，很多数字随着时间的发展需要发生改变。但是人们往往忽略这一点，也许程序已经被移植到其他平台，或者程序的其他改变违背了我们当初对于数字的假定，在这些情况下被忽视的数字值都可能演变成程序错误。通过把这种假定表示成命名良好的符号化常量，可以有效地规避维护代码时可能遇到的困难。

10.4.2 常量表达式中的 const

const 常用于表示接口（见 7.5 节）。同时，const 也可以表示常量值。例如：

```
const int x = 7;
const string s = "asdf";
const int y = sqrt(x);
```

以常量表达式初始化的 const 可以用在常量表达式中。与 constexpr 不同的是，const 可以用非常量表达式初始化，但是此时该 const 将不能用作常量表达式。例如：

```
constexpr int xx = x;          // OK
constexpr string ss = s;       // 错误：s 不是常量表达式
constexpr int yy = y;          // 错误：sqrt(x) 不是常量表达式
```

发生错误的原因是 string 不是字面值常量类型（见 10.4.3 节），sqrt() 不是一个 constexpr 函数（见 12.1.6 节）。

通常情况下，当定义简单的常量时，constexpr 比 const 好。但 constexpr 是 C++11 新增加的，因此在旧代码中存在大量 const。很多时候，枚举值（见 8.4 节）可以替代 const。

10.4.3 字面值常量类型

在常量表达式中可以使用简单的用户自定义类型，例如：

```
struct Point {
    int x,y,z;
```

```
    constexpr Point up(int d) { return {x,y,z+d}; }
    constexpr Point move(int dx, int dy) { return {x+dx,y+dy}; }
    // ...
};
```

含有 constexpr 构造函数的类称为字面值常量类型（literal type）。构造函数必须足够简单才能声明成 constexpr，其中，“简单”的含义是它的函数体必须为空且所有成员都是用潜在的常量表达式初始化的。例如：

```
constexpr Point origo {0,0};
constexpr int z = origo.x;

constexpr Point a[] = {
    origo, Point{1,1}, Point{2,2}, origo.move(3,3)
};
constexpr int x = a[1].x;              // x 的值变为 1

constexpr Point xy{0,sqrt(2)};         // 错误：sqrt(2) 不是常量表达式
```

请注意，即使把数组声明成 constexpr，我们仍然能访问该数组的元素及对象成员。

自然而然地，我们可以定义 constexpr 函数使其接受字面值常量类型的实参。例如：

```
constexpr int square(int x)
{
    return x*x;
}

constexpr int radial_distance(Point p)
{
    return isqrt(square(p.x)+square(p.y)+square(p.z));
}

constexpr Point p1 {10,20,30};             // 默认构造函数是 constexpr 的
constexpr p2 {p1.up(20)};                  // Point::up() 是 constexpr 的
constexpr int dist = radial_distance(p2);
```

在上面的代码中，因为没有一个便于使用的 constexpr 浮点型平方根函数，所以用了 int 而非 double。

对于成员函数来说，constexpr 隐含了 const 的意思，所以下面的写法没有必要：

```
constexpr Point move(int dx, int dy) const { return {x+dx,y+dy}; }
```

10.4.4　引用参数

当你使用 constexpr 时，谨记 constexpr 是一个关于值的概念。此时，任何对象都无法改变值或者造成其他什么影响：constexpr 实际上提供了一种微型的编译时函数式程序设计语言。基于此，你可能会猜测 constexpr 不能接受引用参数，但其实不尽然，因为 const 引用引用的是值，因此也能作为 constexpr 函数的参数。考虑在标准库中将通用的 complex<T> 特例化成 complex<double> 的过程：

```
template<> class complex<double> {
public:
    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(const complex<float>&);
    explicit constexpr complex(const complex<long double>&);
```

```
    constexpr double real();        // 读取实部
    void real(double);              // 设置实部
    constexpr double imag();        // 读取虚部
    void imag(double);              // 设置虚部

    complex<double>& operator= (double);
    complex<double>& operator+=(double);
    // ...
};
```

显然，= 和 += 等用于修改对象的操作不能是 constexpr 的。相反，real() 和 imag() 等简单读取对象内容的操作可以是 constexpr 的，我们在编译时用一条给定的常量表达式对它们求值。有趣的成员是另一种 complex 类型的模板构造函数。例如：

```
constexpr complex<float> z1 {1,2};          // 注意：是 <float> 而非 <double>
constexpr double re = z1.real();
constexpr double im = z1.imag();
constexpr complex<double> z2 {re,im};       // z2 变为 z1 的副本
constexpr complex<double> z3 {z1};          // z3 变为 z1 的副本
```

其中的拷贝构造函数之所以有效，是因为编译器识别出引用（const complex<float>&）所引的是一个常量值，而我们使用的仅仅是这个值本身（既不是指针或者引用，也不是其他什么高级的东西）。

字面值常量类型允许类型丰富的编译时程序设计。传统地，C++ 编译时求值被严格限定为只能使用整数值（不能是函数）。此规定使得人们不得不把每种信息都编码成整数，从而生成异常复杂又充满风险的代码。模板元编程（第 28 章）的某些用法即是如此。有的程序员为了避开这种复杂性，干脆选择运行时求值作为替代。

10.4.5 地址常量表达式

全局变量等静态分配的对象（见 6.4.2 节）的地址是一个常量。然而，该地址值是由链接器赋值的，而非编译器。因此，编译器并不知道这类地址常量的值到底是多少。这就限制了指针或者引用类型的常量表达式的使用范围。例如：

```
constexpr const char* p1 = "asdf";
constexpr const char* p2 = p1;          // OK
constexpr const char* p2 = p1+2;        // 错误：编译器不知道 p1 本身的值是多少
constexpr char c = p1[2];               // OK, c=='d'; 编译器知道 p1 所指的值
```

10.5 隐式类型转换

整数和浮点数类型（见 6.2.1 节）可以在赋值语句及表达式中自由地混合使用。在可能的情况下，值的类型会自动转换以避免损失信息。不幸的是，在这一过程中也可能会发生某些隐式的损失值（“窄化”）的类型转换。如果我们转换了某个值的类型，然后能够把它再转回原类型并且保持初始值不变，则称该转换是值保护的。相反，如果某个转换做不到这一点，那么它被称为窄化类型转换（narrowing conversion，见 10.5.2.6 节）。本节简要介绍类型转换的规则、遇到的问题及解决方案。

10.5.1 提升

保护值不被改变的隐式类型转换通常称为提升（promotion）。在执行算术运算之前，通

常先把较短的整数类型通过整型提升（integral promotion）成 int。提升的结果一般不会是 long（除非运算对象的类型是 char16_t、char32_t、wchar_t 或者本身比 int 大的一个普通枚举类型）或 long double。这反映了 C 语言中类型提升的本质：把运算对象变得符合算术运算的“自然”尺寸。

整型提升的规则是：

- 如果 int 能表示类型为 char、signed char、unsigned char、short int 或者 unsigned short int 的数据的值，则将该数据转换为 int 类型；否则，将它转换为 unsigned int 类型。
- char16_t、char32_t、wchar_t（见 6.2.3 节）或者普通枚举类型（见 8.4.2 节）的数据转换成下列类型中第一个能够表示其全部值的类型：int、unsigned int、long、unsigned long 或者 unsigned long long。
- 如果位域（见 8.2.7 节）的全部值都能用 int 表示，则它转换为 int；否则，如果全部值能用 unsigned int 表示，则它转换为 unsigned int；如果 int 和 unsigned int 都不行，则不执行任何整型提升。
- bool 值转换成 int，其中，false 变为 0 而 true 变为 1。

提升是常规算术类型转换（见 10.5.3 节）的一部分。

10.5.2　类型转换

基本类型之间可能发生各种各样的隐式类型转换（§ iso.4）。以我的观点来看，C++ 语言允许的类型转换有点太多了。例如：

```
void f(double d)
{
    char c = d;          // 当心：这是双精度浮点数向字符类型的转换
}
```

当编写代码的时候，应该谨记不要产生未定义的行为，并且时刻提防在不知不觉中丢失信息的类型转换（“窄化类型转换”）。

编译器能够发现并报告很多不可靠的类型转换，幸运的是，很多编译器也确实会这样做。

使用 {} 列表能防止窄化计算的发生（见 6.3.5 节），例如：

```
void f(double d)
{
    char c {d};          // 错误：编译器发现程序试图把双精度浮点数转换成字符类型
}
```

如果潜在的窄化类型转换确实无法避免，则程序员应该考虑使用一些在运行时执行检查的类型转换函数（比如 narrow_cast<>()，见 11.5 节）。

10.5.2.1　整数类型转换

整数能被转换成其他整数类型。一个普通的枚举类型值也能转换成整数类型（见 8.4.2 节）。

如果目标类型是 unsigned 的，则结果值所占的二进制位数以目标类型为准（如有必要，会丢掉靠前的二进制位）。更准确地说，转换前的整数值对 2 的 n 次幂取模后所得的结果值就是转换结果，其中 n 是目标类型所占的位数。例如：

```
unsigned char uc = 1023;// 二进制 1111111111；uc 的值变为二进制 11111111，即，255
```

如果目标类型是 signed 的，则当原值能用目标类型表示时，它不发生改变；反之，结果值依赖于具体实现：

```
signed char sc = 1023;  // 依赖于实现
```

结果可能是 127 或者 –1（见 6.2.3 节）。

布尔值或者普通枚举类型的值能隐式地转换成等值的整数类型（见 6.2.2 节和 8.4 节）。

10.5.2.2 浮点数类型转换

给定一个浮点值，我们能把它转换成其他浮点类型的值。如果原值能用目标类型完整地表示，则所得的结果与原值相等。如果原值介于两个相邻的目标值之间，则结果取它们中的一个。其他情况下，结果是未定义的。例如：

```
float f = FLT_MAX;          // 最大的单精度浮点值
double d = f;               // OK: d == f

double d2 = DBL_MAX;        // 最大的双精度浮点值
float f2 = d2;              // 如果 FLT_MAX<DBL_MAX，则结果是未定义的

long double ld = d2;        // OK: ld == d2
long double ld2 = numeric_limits<long double>::max();
double d3 = ld2;            // 如果 sizeof(long double)>sizeof(double)，则结果是未定义的
```

DBL_MAX 和 FLT_MAX 定义在 <limits> 中；numeric_limits 定义在 <limits> 中（见 40.2 节）。

10.5.2.3 指针和引用类型转换

任何指向对象类型的指针都能隐式地转换成 void*（见 7.2.1 节）。指向派生类的指针（或引用）能隐式地转换成指向其可访问的且明确无二义的基类（见 20.2 节）的指针（或引用）。请注意，指向函数的指针和指向成员的指针不能隐式地转换成 void*。

求值结果为 0 的常量表达式（见 10.4 节）能隐式地转换成任意指针类型的空指针。类似地，求值结果为 0 的常量表达式也能隐式地转换成指向成员的指针类型（见 20.6 节）。例如：

```
int* p = (1+2)*(2*(1-1)); // 正确，但是让人感觉很奇怪
```

最好直接使用 nullptr（见 7.2.2 节）。

T* 可以隐式地转换成 const T*（见 7.5 节）。类似地，T& 能隐式地转换成 const T&。

10.5.2.4 指向成员的指针的类型转换

指向成员的指针或引用的类型转换规则将在 20.6.3 节介绍。

10.5.2.5 布尔值类型转换

指针、整数和浮点数都能隐式地转换成 bool 类型（见 6.2.2 节）。非 0 的值对应 true，0 对应 false。例如：

```
void f(int* p, int i)
{
    bool is_not_zero = p;       // 如果 p!=0 则为真
    bool b2 = i;                // 如果 i!=0 则为真
    // ...
}
```

指针向布尔值的类型转换在条件中很有用，但是其他情况下可能造成误解：

```
void fi(int);
void fb(bool);
```

```
void ff(int* p, int* q)
{
    if (p) do_something(*p);              // OK
    if (q!=nullptr) do_something(*q);     // 正确，但是显得啰嗦
    //...
    fl(p);                                // 错误：不存在指针向整数的类型转换
    fb(p);                                // OK: 指针到布尔值的类型转换（感觉有些吧？）
}
```

我们希望编译器能发现 fb(p) 的问题并给出警告。

10.5.2.6　浮点向整数类型转换

当浮点值转换成整数值时，浮点值的小数部分被忽略掉了。换句话说，从浮点数向整数的类型转换会导致截断。例如，int(1.6) 的值是 1。如果被截断的值不能用目标类型表示，则该行为是未定义的。例如：

```
int i = 2.7;          // i 的值变为 2
char b = 2000.7;      // 当 char 占 8 位时是未定义的：2000 不能用 8 位的字符表示
```

反之，只要硬件条件允许，那么从整数向浮点数的转换从数学意义上来说就是合法的。只有当某个整数值无法用浮点类型完整表示时，才会出现精度的损失。例如：

```
int i = float(1234567890);
```

在一台 int 和 float 都用 32 位表示的机器上，转换后的 i 值是 1234567936。

显然，我们应该尽量避免可能损失值的隐式类型转换。事实上，编译器能够发现并报告某些明显非常危险的转换操作，比如浮点数转换为整数以及 long int 转换为 char。然而，完全依靠编译器进行检查并不现实，因此，程序员自己也必须加倍小心。当“加倍小心”不够的时候，程序员就需要加入一些显式的检查了。例如：

```
char checked_cast(int i)
{
    char c = i;      // 警告：不可移植（见 10.5.2.1 节）
    if (i != c) throw std::runtime_error("int-to-char check failed");
    return c;
}

void my_code(int i)
{
    char c = checked_cast(i);
    //...
}
```

25.2.5.1 节将介绍一种检查类型转换更常用的技术。

通过使用 numeric_limits（见 40.2 节），能确保截断以一种可移植的方式进行。在初始化过程中，{} 初始化器形式有助于避免截断的发生（见 6.3.5 节）。

10.5.3　常用的算术类型转换

下面这些转换规则适用于二元运算符的运算对象，目的是把它们转换成一种常见的类型，并且用该类型作为运算结果的类型：

[1] 如果一个运算对象的类型是 long double，则另一个也转换成 long double。

- 否则，如果一个运算对象的类型是 double，则另一个也转换成 double。
- 否则，如果一个运算对象的类型是 float，则另一个也转换成 float。

- 否则，两个运算对象都执行整型提升（见10.5.1节）。

[2] 否则，如果一个运算对象的类型是unsigned long long，则另一个也转换成unsigned long long。

- 否则，如果一个运算对象的类型是long long int，而另一个运算对象的类型是unsigned long int，则当long long int能表示所有unsigned long int的值时，该unsigned long int转换成long long int；否则，两个运算对象都转换成unsigned long long int。
- 否则，如果一个运算对象的类型是long int，而另一个运算对象的类型是unsigned int，则当long int能表示所有unsigned int的值时，该unsigned int转换成long int；否则，两个运算对象都转换成unsigned long int。
- 否则，如果一个运算对象的类型是long，则另一个也转换成long。
- 否则，如果一个运算对象的类型是unsigned，则另一个也转换成unsigned。
- 否则，两个运算对象都转换成int。

上述规则使得转换后的结果要么是无符号整型，要么是在实现中尺寸更大的带符号整型。这也是我们要求避免在同一条表达式中混用无符号整数和带符号整数的原因之一。

10.6 建议

[1] 优先使用标准库，然后是其他库，最后才是“手工打造的代码”；10.2.8节。

[2] 尽可能不要使用字符级的输入；10.2.3节。

[3] 读取数据的时候，一定要考虑格式错误的可能；10.2.3节。

[4] 优先使用合适的抽象概念（类、算法等），然后才考虑直接使用语言功能（比如int、语句）；10.2.8节。

[5] 避免使用复杂的表达式；10.3.3节。

[6] 如果对运算符的优先级存疑，可以用括号括起来；10.3.3节。

[7] 避免未定义求值顺序的表达式；10.3.2节。

[8] 避免窄化类型转换；10.5.2节。

[9] 定义符号化常量以防止出现“魔法常量”；10.4.1节。

选择适当的操作

如果有人说，
“希望有这么一种编程语言，
我所要做的就是说出我想实现的东西”，
那么，赶紧给他个棒棒糖吧。

——艾伦·佩利

- 其他运算符
  逻辑运算符；位逻辑运算符；条件表达式；递增与递减
- 自由存储
  内存管理；数组；获取内存空间；重载 new
- 列表
  实现模型；限定列表；未限定列表
- lambda 表达式
  实现模型；lambda 的替代品；捕获；调用与返回；lambda 的类型
- 显式类型转换
  构造；命名转换；C 风格的转换；函数形式的转换
- 建议

11.1 其他运算符

本节介绍几种简单的运算符：逻辑运算符（&&、|| 和 !）、位逻辑运算符（&、|、~、<< 和 >>）、条件表达式（? :）以及递增递减运算符（++ 和 --）。这些运算符在细节上差异很大，且与之前提到的运算符关联性不强，所以放在这里集中讨论。

11.1.1 逻辑运算符

逻辑运算符 &&（与）、||（或）和 !（非）接受算术类型以及指针类型的运算对象，将其转换为 bool 类型，最后返回一个 bool 类型的结果。只有当逻辑上确实需要时，&& 和 || 才会对其第二个实参求值；因此，这两个运算符具有控制求值顺序的功能。例如：

```
while (p && !whitespace(*p)) ++p;
```

此时，如果 p 是 nullptr，则不会对其执行解引用的操作。

11.1.2 位逻辑运算符

位逻辑运算符 &（与）、|（或）、^（异或）、~（非）、>>（右移）和 <<（左移）作用于整型对象，即，char、short、int、long、long long 及其对应的 unsigned 版本，以及 bool、wchar_t、char16_t 和 char32_t 等类型。一个普通的 enum（而非 enum class）可被隐式地转换成

整数类型，从而作为位逻辑运算符的运算对象。算术类型转换（见 10.5.3 节）决定了结果的类型。

位逻辑运算符常用于实现一个小集合的概念（位向量）。此时，无符号整数的每一位表示集合的一个成员，位的数量限制了成员的数量。二元运算符 & 表示求交操作，| 表示求并操作，^ 表示异或操作，~ 表示求补操作。我们可以使用枚举类型命名该集合的成员，下面是从 ostream 的实现借鉴而来的一个小例子：

```
enum ios_base::iostate {
    goodbit=0, eofbit=1, failbit=2, badbit=4
};
```

流的实现通常以下面的方式设置并检查其状态：

```
state = goodbit;
// ...
if (state&(badbit|failbit)) // 流的状态不好
```

因为 & 的优先级高于 |（见 10.3 节），所以在上面的代码中，if 语句内部的括号必不可少。

如果某个函数到达了输入的末尾，则它可能以下面的形式报告这一状态：

```
state |= eofbit;
```

|= 能在现有的状态上添加内容，相反，简单的赋值语句（state=eofbit）将会清除掉其他所有位。

我们能在流的实现之外观察并使用这些流的状态标志位。例如，使用下面的代码可以发现两个流的状态有何不同：

```
int old = cin.rdstate();    // rdstate() 返回状态
// ...使用 cin...
if (cin.rdstate()^old) {    // 有变化吗?
    // ...
}
```

计算流状态的差别并不多见，但是对于其他类似的类型来说，常常需要计算差别。例如，不妨考虑这样一个任务：我们需要比较两个位向量，其中一个表示正在处理的中断的集合，另一个表示等待处理的中断的集合。

请注意，这个关于位状态更改的例子应该是从输入输出流的实现中挖掘出来的，而非源自用户接口。便捷的位操作非常重要，但是从可靠性、可操作性和可移植性等角度出发，这项功能还是应该置于系统的底层。关于集合的更多概念，请参见标准库 set（见 31.4.3 节）和 bitset（见 34.2.2 节）。

我们能通过位逻辑运算从字中抽取位域。例如，下面的代码可以从一个 32 位的 int 中提取出中间的 16 位：

```
constexpr unsigned short middle(int a)
{
    static_assert(sizeof(int)==4,"unexpected int size");
    static_assert(sizeof(short)==2,"unexpected short size");
    return (a>>8)&0xFFFF;
}

int x = 0xFF00FF00; // 假定 sizeof(int)==4
short y = middle(x); // y = 0x00FF
```

对于类似的移动和取模操作来说，位域（见 8.2.7 节）是一种便捷的实现方式。

不要把位逻辑运算符与逻辑运算符（&&、|| 和 !）混为一谈。后者的返回值是 true 或 false，且常用于 if、while 和 for 语句的条件部分（见 9.4 节和 9.5 节）。例如，!0（非 0）的值是 true，转换后得到 1；而 ~0（0 的补）在位模式下表示为全 1，对应的补码值是 −1。

11.1.3　条件表达式

某些 if 语句可以改写成条件表达式（conditional-expression），例如：

```
if (a <= b)
    max = b;
else
    max = a;
```

这段代码可以更加直观地表示为：

```
max = (a<=b) ? b : a;
```

其中，条件部分的括号并非必需，但是加上后能使代码更易读。

条件表达式能用在常量表达式（见 10.4 节）中，这一点非常重要。

在条件表达式 c?e1:e2 中使用了一对可选的表达式 e1 和 e2，这对表达式的类型必须相同，或者它们都能隐式地转成同一种类型 T。对于算术类型来说，可使用常规的算术类型转换规则（见 10.5.3 节）找到公共类型 T；对于其他类型，要求 e1 能隐式转换成 e2 的类型，或者 e2 能隐式转换成 e1 的类型。此外，throw 表达式（见 13.5.1 节）也能作为条件表达式的一个分支。例如：

```
void fct(int* p)
{
    int i = (p) ? *p : std::runtime_error{"unexpected nullptr};
    // ...
}
```

11.1.4　递增与递减

++ 运算符可以直接表达递增的含义，而无须通过加法运算和赋值运算的组合来间接地表达。与 ++ 运算符一起出现的 lvalue 不会产生任何副作用，++lvalue 的含义是 lvalue+=1，即 lvalue=lvalue+1。在该表达式中，执行递增运算的对象只求值一次。类似地，递减运算符表示为 --。

++ 和 -- 既可以作为前缀运算符，也可以作为后缀运算符。++x 的值是 x 的新值（即，x 递增之后的值）。例如，y=++x 等价于 y=(x=x+1)。与之相反，x++ 的值是 x 的旧值。例如，y=x++ 等价于 y=(t=x,x=x+1,t)，其中，t 是一个与 x 类型相同的变量。

类似对指针加和减一个 int，当 ++ 和 -- 作用于指针时，将会直接操作指针所指的数组中的元素。p++ 令 p 指向下一个元素（见 7.4.1 节）。

当程序需要递增或递减循环变量时，++ 和 -- 运算符特别有用。例如，我们可以使用下面的循环拷贝一个以 0 结尾的 C 风格字符串：

```
void cpy(char* p, const char* q)
{
    while (*p++ = *q++) ;
}
```

这种简练且面向表达式的编码方式对于 C 和 C++ 来说都是利弊各半，让程序员又爱又恨。

考虑其中的循环部分：

```
while (*p++ = *q++) ;
```

对于不熟悉 C 语言的程序员来说，他们很难理解这条语句的确切含义。但是，这样的语句在程序中并不少见，因此我们有必要详细讨论一下其细节。首先用传统的方式实现拷贝字符数组的功能：

```
int length = strlen(q);
for (int i = 0; i<=length; i++)
    p[i] = q[i];
```

这段代码存在浪费。为了求出一个以 0 结尾的字符串的长度，我们必须依次读取字符串的每一个元素以寻找结束符 0。这样我们就读了两次字符串：一次获取它的长度，另一次拷贝它的内容。一种可供替代的解决方案是：

```
int i;
for (i = 0; q[i]!=0 ; i++)
    p[i] = q[i];
p[i] = 0;          // 添加 0 作为结束符
```

进一步，因为 p 和 q 都是指针类型，所以我们可以去掉索引变量 i：

```
while (*q != 0) {
    *p = *q;
    p++;       // 指向下一个字符
    q++;       // 指向下一个字符
}
*p = 0;        // 添加 0 作为结束符
```

因为后置递增运算符允许我们先使用值再递增它，所以我们能把循环重写成下面的形式：

```
while (*q != 0) {
    *p++ = *q++;
}
*p = 0; // 添加 0 作为结束符
```

*p++ = *q++ 的值是 *q，因此继续改写该例的程序：

```
while ((*p++ = *q++) != 0) { }
```

在此例中，我们先把 *q 拷贝给 *p 并递增 p 的值，然后才判断 *q 是否为 0。因此，之前版本中最后一条执行结尾字符 0 赋值的语句可以省略掉。我们进一步发现：空白循环体根本没必要出现，并且由于当整数作为条件时无论如何都会与 0 进行比较，所以 !=0 也是冗余的。基于这些分析，最后我们把循环简写成下面的形式：

```
while (*p++ = *q++) ;
```

这个版本的易读性比之前的版本差吗？至少对于经验丰富的 C 或 C++ 程序员来说并非如此。那么这个版本在时空效率上优于之前的版本吗？它比一开始那个调用了 strlen() 的版本更优，但是与其他版本相比就不一定了，通常情况下，它们在性能上几乎是等价的，甚至有可能生成完全一样的代码。

拷贝一个以 0 结尾的字符串的最有效方式是采用标准的 C 风格字符串拷贝函数：

```
char* strcpy(char*, const char*);   // 来自于 <string.h>
```

如果要执行更一般的拷贝任务，应该使用标准库 copy 算法（见 4.5 节和 32.5 节）。我们的

原则是尽可能利用标准库功能，而非用指针和字节自行实现。有些标准库函数是内联的（见12.1.3节），有些甚至是用特定的机器指令实现的。因此，在决定选用手工编写的代码之前，最好确认它的性能优于标准库函数。即便如此，这种暂时的优越性也可能随着硬件和编译器的改变而不复存在。另外对于程序的维护者来说，使用手工编写的代码而非标准库函数会让他们头疼不已。

11.2　自由存储

命名对象的生命周期由其作用域决定（见6.3.4节）。然而，某些情况下我们希望对象与创建它的语句所在的作用域独立开来。例如，很多时候我们在函数内部创建了对象，并且希望在函数返回后仍能使用这些对象。运算符new负责创建这样的对象，运算符delete则负责销毁它们。new分配的对象“位于自由存储之上”（或者说“在堆上”或“在动态内存中”）。

让我们设想一下该为桌面计算器（见10.2节）编写一个怎样的编译器。它的语法分析函数应该会构建一棵表达式树以供代码生成器使用：

```
struct Enode {
    Token_value oper;
    Enode* left;
    Enode* right;
    // ...
};
Enode* expr(bool get)
{
    Enode* left = term(get);

    for (;;) {
        switch (ts.current().kind) {
        case Kind::plus:
        case Kind::minus:
            left = new Enode {ts.current().kind,left,term(true)};
            break;
        default:
            return left;                    // 返回节点
        }
    }
}
```

在Kind::plus和Kind::minus分支中，我们在自由存储上新建了一个Enode并将其初始化为{ts.current().kind,left,term(true)}。所得的指针赋给left并最终从expr()返回回来。

我使用{}列表的形式传递实参，当然也可以使用传统的()列表形式指定初始化器。但是，如果试图用符号=初始化一个用new创建的对象，将会引发程序错误：

```
int* p = new int = 7; // 错误
```

如果某一类型含有默认构造函数，则我们可以省略掉初始化器。但是对内置类型这么做的话，其变量将会处于未初始化的状态。例如：

```
auto pc = new complex<double>;   // 该复数被初始化为 (0,0)
auto pi = new int;               // 该 int 未被初始化
```

上述设定可能会让人感到有些困惑，更好的办法是使用{}，这样可以确保变量执行默认初始化。例如：

```
auto pc = new complex<double>{}; // 该复数被初始化为 {0,0}
auto pi = new int{};             // 该 int 被初始化为 0
```

代码生成器先使用 expr() 创建的 Enode，然后将其删除：

```
void generate(Enode* n)
{
    switch (n->oper) {
    case Kind::plus:
        // 使用 n
        delete n; // 从自由存储中删除一个 Enode
    }
}
```

对于一个用 new 创建的对象来说，我们必须用 delete 显式地将它销毁，否则它将一直存在。只有将它销毁了，它占用的空间才能被其他 new 使用。有一种思路是建立一个“垃圾回收器”，由它负责看管未引用的对象并使得 new 能重新使用这些对象所占的空间，但是 C++ 的具体实现并不能确保这一点。因此，我假设 new 创建的对象需要由 delete 手动地释放。

delete 运算符只能作用于 new 返回的指针或者 nullptr，不过对 nullptr 使用 delete 不产生什么实际效果。

如果被删除的对象的类型是一个含有析构函数的类（见 3.2.1.2 节和 17.2 节），则 delete 将调用该析构函数，然后释放该对象所占的内存空间以供后续使用。

11.2.1　内存管理

自由存储的问题主要包括：

- 对象泄漏（leaked object）：使用 new，但是忘了用 delete 释放掉分配的对象。
- 提前释放（premature deletion）：在尚有其他指针指向该对象并且后续仍会使用该对象的情况下过早地 delete。
- 重复释放（double deletion）：同一对象被释放两次，两次调用对象的析构函数（如果有的话）。

对象泄露是一种潜在的严重错误，因为它可能会令程序面临资源耗尽的情况。与之相比，提前释放更容易造成恶果，因为指向“已删对象”的指针所指的可能已经不是一个有效的对象了（此时读取的结果很可能与预期不符），又或者该内存区域已经存放了其他对象（此时对该区域执行写入操作将会影响本来无关的对象）。下面是一段非常糟糕的代码：

```
int* p1 = new int{99};
int* p2 = p1;                  // 存在潜在的麻烦
delete p1;                     // 此时，p2 所指的不再是一个有效对象
p1 = nullptr;                  // 造成代码资源的错觉
char* p3 = new char{'x'};      // 此时，p3 有可能指向了 p2 所指的内存区域
*p2 = 999;                     // 该行代码可能会造成错误
cout << *p3 << '\n';           // 输出的内容可能不是 x
```

重复释放的问题在于资源管理器通常无法追踪资源的所有者。例如：

```
void sloppy()  // 非常糟糕的代码
{
    int* p = new int[1000];    // 请求内存
    // ... 使用 *p ...
    delete[] p;                // 释放内存
```

```
    // ... 完成一些其他操作 ...

    delete[] p;                    // 此时，sloppy() 已经不拥有 *p 了
}
```

在执行第 2 个 delete[] 的时候，*p 对应的内存区域可能已经被重新分配了，此时重新分配的内容可能会受到影响。如果把该段示例代码中的 int 替换成 string 的话，我们就能看到 string 的析构函数试图先读取一块已经被释放并且可能已被其他代码重写的内存区域，然后再 delete 这块区域，这显然是错误的。通常情况下，重复释放属于未定义的行为，将产生不可预知的结果，甚至引发程序灾难。

程序中之所以会存在上述错误，并非有人故意为之，甚至连程序员的疏忽过错都算不上。真正的原因是在一个规模较大的程序中要想确保准确释放掉分配的每一个对象（只释放一次且确保释放点正确）实在太难了。对于初学者来说，仅仅分析程序的局部很难发现此类问题，因为错误通常会涉及程序的几个不同部分。

有两种方法可以避免上述问题，我建议程序员使用这两种方法代替“裸”new 和 delete：

[1] 除非万不得已不要把对象放在自由存储上，优先使用作用域内的变量。

[2] 当你在自由存储上构建对象时，把它的指针放在一个管理器对象（manager object，有时也称为句柄）中，此类对象通常含有一个析构函数，可以确保释放资源。例如 string、vector 等标准库容器，以及 unique_ptr（见 5.2.1 节，34.3.1 节）和 shared_ptr（见 5.2.1 节，34.3.2 节）等。尽可能让这个管理器对象作为作用域内的变量出现。很多习惯于使用自由存储的场合其实都可以用移动语义（见 3.3 节，17.5.2 节）替代，只要从函数中返回一个表示大对象的管理器对象就可以了。

其中，规则 [2] 简称为 RAII（“资源获取即初始化”，见 5.2 节和 13.3 节）。RAII 是一项避免资源泄漏的基本技术，它让我们可以安全便捷地使用异常机制来处理错误。

标准库 vector 是其中的一个示例：

```
void f(const string& s)
{
    vector<char> v;
    for (auto c : s)
        v.push_back(c);
    // ...
}
```

vector 的元素位于自由存储上，但是它把分配和释放资源的操作都限定在其内部进行。在此例中，push_back() 负责执行 new（为元素分配空间）和 delete（释放不再需要的空间）的操作。vector 的用户无须了解具体的实现细节，他们可以完全信任 vector 不会导致内存泄漏。

计算器示例的 Token_stream 是一个更简单的例子（见 10.2.2 节），用户使用 new 并且令所得的指针指向 Token_stream 以便于管理：

```
Token_stream ts{new istringstream{some_string}};
```

如果我们仅仅希望从函数中得到一个大对象，则不必使用自由存储。例如：

```
string reverse(const string& s)
{
```

```
        string ss;
        for (int i=s.size()-1; 0<=i; --i)
            ss.push_back(s[i]);
        return ss;
}
```

与 vector 类似，string 实际上也是其元素的一个句柄。因此，我们可以直接把 ss 移动到 reverse() 之外，而无须拷贝任何元素（见 3.3.2 节）。

上述思想的另一个比较深入的例子是用于资源管理的“智能指针”（unique_ptr 和 shared_ptr，见 5.2.1 节和 34.3.1 节）。例如：

```
void f(int n)
{
        int* p1 = new int[n];                    // 存在潜在的风险
        unique_ptr<int[]> p2 {new int[n]};
        // ...
        if (n%2) throw runtime_error("odd");
        delete[] p1;                        // 程序有可能运行不到此处
}
```

对于 f(3) 来说，p1 所指的内存发生了泄漏，但是 p2 所指的内存以隐式的方式正确释放了。

关于 new 和 delete，我的经验是应该尽量确保“没有裸 new”，即，令 new 位于构造函数或类似的函数中，delete 位于析构函数中，由它们提供内存管理的策略。此外，new 常用作资源句柄的实参。

如果所有措施都无能为力（例如，某人的旧代码规模庞大且对 new 的使用非常混乱），C++ 最后还提供了一个垃圾回收器的标准接口（见 34.5 节）。

11.2.2　数组

new 还能用来创建对象的数组，例如：

```
char* save_string(const char* p)
{
        char* s = new char[strlen(p)+1];
        strcpy(s,p);            // 从 p 拷贝到 s
        return s;
}

int main(int argc, char* argv[])
{
        if (argc < 2) exit(1);
        char* p = save_string(argv[1]);
        // ...
        delete[] p;
}
```

“普通”delete 用于删除单个对象，delete[] 负责删除数组。

除非必须直接使用 char*，否则一般情况下，标准库 string 是更好的选择，它可以简化 save_string()：

```
string save_string(const char* p)
{
        return string{p};
}
```

```
int main(int argc, char* argv[])
{
    if (argc < 2) exit(1);
    string s = save_string(argv[1]);
    // ...
}
```

关键之处是我们无须纠结于 new[] 和 delete[] 了。

delete 和 delete[] 必须清楚分配的对象有多大，才能准确地释放 new 分配的空间。这意味着用 new 的标准实现分配的对象要比静态对象所占的空间稍大一点。超出的部分至少要能存得下对象的尺寸。通常情况下，对于每次分配，我们需要两个或更多字来管理自由存储。绝大多数现代计算机都使用 8 字节的字。如果我们分配的很多对象组成了一个数组或者大对象，则消耗的管理空间完全可以接受；但是如果我们在自由存储上分配了很多个小对象（比如很多 int 或者 Point），那么额外的空间就显得有点太多了。

请注意，vector（见 4.4.1 节和 31.4 节）本身就是一个对象，因此我们可以使用普通的 new 和 delete 分配和释放 vector。例如：

```
void f(int n)
{
    vector<int>* p = new vector<int>(n);    // 单个对象
    int* q = new int[n];                    // 数组
    // ...
    delete p;
    delete[] q;
}
```

delete[] 只能用于两种情况，一种是指向由 new 创建的数组的指针，另一种是空指针（见 7.2.2 节）。delete[] 作用于空指针时什么也不做。

切记不要用 new 创建局部对象，例如：

```
void f1()
{
    X* p =new X;
    // ... 使用 *p ...
    delete p;
}
```

这种用法冗长、低效且极易出错（见 13.3 节）。如果先有 return 语句或者抛出异常的语句后有 delete，则可能导致内存泄漏（除非辅以其他代码）。相反，使用局部变量可以解决这一问题：

```
void f2()
{
    X x;
    // ... 使用 x ...
}
```

在退出 f2 之前，先隐式地销毁局部变量 x。

11.2.3　获取内存空间

自由存储运算符 new、delete、new[] 和 delete[] 的实现位于 <new> 头文件中：

```
void* operator new(size_t);          // 为单个对象分配空间
void operator delete(void* p);       // 如果 p 为真，释放 new() 分配的全部空间
```

```
void* operator new[](size_t);          // 为数组分配空间
void operator delete[](void* p);       // 如果 p 为真，释放 new[]() 分配的全部空间
```

当运算符 new 需要为对象分配空间时，它调用 operator new() 分配适当数量的字节。类似地，当运算符 new 需要为数组分配空间时，调用 operator new[]()。

operator new() 和 operator new[]() 的标准实现不负责初始化得到的内存。

分配和释放函数负责处理无类型且未初始化的内存（通常称为“原始内存”），而非类型明确的对象。因此，其实参和返回值的类型都是 void*。无类型的内存层和带类型的对象层的映射关系由运算符 new 和 delete 负责。

当 new 发现没有多余的内存可供分配时会发生什么呢？默认情况下，分配器会抛出一个标准库 bad_alloc 异常（别的处理方式见 11.2.4.1 节）。例如：

```
void f()
{
    vector<char*> v;
    try {
        for (;;) {
            char * p = new char[10000];    // 申请一些内存空间
            v.push_back(p);                // 将申请的空间加入向量中以供将来使用
            p[0] = 'x';                    // 使用新申请的内存
        }
    }
    catch(bad_alloc) {
        cerr << "Memory exhausted!\n";
    }
}
```

不论在我们的机器上有多少内存，上述代码都必须包含处理 bad_alloc 的部分。有一点需要明确：new 运算符并不保证在耗尽物理主存后一定会抛出异常。因此，如果系统设置了虚拟内存，则该程序将可能消耗大量的磁盘空间，在很长一段时间后才抛出异常。

我们可以规定当内存资源耗尽时 new 的行为，参见 30.4.1.3 节。

除了 <new> 中定义的函数之外，用户还可以为某个特定的类自定义 operator new() 等函数（见 19.2.5 节）。根据一般的作用域规则，类成员 operator new() 的优先级高于 <new> 中的函数。

11.2.4　重载 new

默认情况下，new 运算符在自由存储上创建它的对象。但是如果我们想在别的地方分配对象该怎么办呢？以一个简单的类为例：

```
class X {
public:
    X(int);
    // ...
};
```

如果我们想把对象放置在别的地方，可以提供一个含有额外实参的分配函数（见 11.2.3 节），然后在使用 new 的时候传入指定的额外实参：

```
void* operator new(size_t, void* p) { return p; }    // 放式运算符，将对象置于别处

void* buf = reinterpret_cast<void*>(0xF00F);         // 一个明确的地址
X* p2 = new(buf) X;                                  // 在 buf 处构建 X
                                                     // 调用：operator new(sizeof(X),buf)
```

由于这种用法的存在，我们通常把提供额外的实参给 operator new() 的 new(buf) X 语法称为放置语法（placement syntax）。请注意，每个 operator new() 都接受一个尺寸作为它的第一个实参，而该尺寸的对象是隐式提供的（见 19.2.5 节）。编译器根据常规的实参匹配规则（见 12.3 节）确定 new 运算符到底使用哪个 operator new()。每个 operator new() 都以 size_t 作为它的第一个实参。

其中，“放置式” operator new() 是最简单的一个，它的定义位于 <new> 头文件中：

```
void* operator new (size_t sz, void* p) noexcept;        // 将大小为 sz 的对象置于 p 处
void* operator new[](size_t sz, void* p) noexcept;       // 将大小为 sz 的对象置于 p 处

void operator delete (void* p, void*) noexcept;          // 如果 p 为真，令 *p 无效
void operator delete[](void* p, void*) noexcept;         // 如果 p 为真，令 *p 无效
```

“放置式 delete” 可能会告知垃圾回收器当前删掉的指针不再安全（见 34.5 节），除此之外就什么也不做了。

放置式 new 还能用于从某一特定区域分配内存：

```
class Arena {
public:
    virtual void* alloc(size_t) =0;
    virtual void free(void*) =0;
    // ...
};

void* operator new(size_t sz, Arena* a)
{
    return a->alloc(sz);
}
```

现在，我们就能在不同 Arena 里分配任意类型的对象了。例如：

```
extern Arena* Persistent;
extern Arena* Shared;

void g(int i)
{
    X* p = new(Persistent) X(i);    // 在某持续性存储上分配 X
    X* q = new(Shared) X(i);        // 在共享内存上分配 X
    // ...
}
```

把对象置于一块标准自由存储管理器不（直接）控制的区域，意味着我们在销毁此类对象时必须特别小心。处理这一问题的常规做法是显式调用一个析构函数：

```
void destroy(X* p, Arena* a)
{
    p->~X();        // 调用析构函数
    a->free(p);     // 释放内存
}
```

请注意，除非我们实现的是资源管理类，否则应该尽量避免显式调用析构函数。甚至绝大多数资源句柄都能用 new 和 delete 构建。然而，如果不通过显式的析构函数调用，我们将很难按照标准库 vector（见 4.4.1 节和 31.3.3 节）的方式去实现有效的通用容器类。作为新手，在决定显式调用析构函数之前一定要仔细斟酌，必要的话可以向更有经验的同事寻求一些建议。

关于放置式 new 如何与异常处理模块交互的例子，请参见 13.6.1 节。

没有专门放置数组的语法；而且因为放置式 new 能分配任意类型的对象，所以也没有必要再设计这种语法。不过，我们可以为数组定义一个 operator delete[]()（见 11.2.3 节）。

11.2.4.1　nothrow new

有的程序不允许出现异常（见 13.1.5 节），此时，我们可以使用 nothrow 版本的 new 和 delete。例如：

```
void f(int n)
{
    int* p = new(nothrow) int[n];        // 在自由存储上分配 n 个 int
    if (p==nullptr) {// 无可用内存
        // ... 处理分配内存错误 ...
    }
    // ...
    operator delete(nothrow,p);          // 释放 *p
}
```

其中，nothrow 是标准库类型 nothrow_t 的对象，该类型有助于消除二义性。nothrow 和 nothrow_t 的声明位于 <new> 中。

其实现细节也在 <new> 中：

```
void* operator new(size_t sz, const nothrow_t&) noexcept;  // 分配 sz 个字节；
                                                           // 如果分配失败，返回 nullptr
void operator delete(void* p, const nothrow_t&) noexcept;  // 释放 new 分配的空间

void* operator new[](size_t sz, const nothrow_t&) noexcept; // 分配 sz 个字节；
                                                           // 如果分配失败，返回 nullptr
void operator delete[](void* p, const nothrow_t&) noexcept; // 释放 new 分配的空间
```

如果没能分配到有效的内存，上面这些 operator new 不会抛出 bad_alloc，而是返回一个 nullptr。

11.3　列表

我们能用 {} 列表初始化命名变量（见 6.3.5.2 节）。此外，在很多（但并非所有）地方 {} 列表还能作为表达式出现。它们的表现形式有两种：

[1] 限定为某种类型，形如 T{……}，意思是“创建一个 T 类型的对象并用 T{……} 初始化它”；参见 11.3.2 节。

[2] 未限定的 {……}，其类型根据上下文确定；参见 11.3.3 节。

例如：

```
struct S { int a, b; };
struct SS { double a, b; };

void f(S);      // f() 接受一个 S

void g(S);
void g(SS);     // 重载了 g()

void h()
{
    f{1,2};         // OK: 调用 f(S{1,2})

    g{1,2};         // 错误：存在二义性
    g(S{1,2});      // OK: 调用 g(S)
```

```
    g(SS{1,2});     // OK: 调用 g(SS)
}
```

当我们用列表初始化命名变量时（见 6.3.5 节），列表中可以包含 0 个、1 个或者多个元素。{} 列表构建的是某种类型的对象，因此其中包含的元素数量和类型都必须符合构建该类型对象的要求。

11.3.1　实现模型

{} 列表的实现模型由三部分组成：

- 如果 {} 列表被用作构造函数的实参，则其实现过程与使用 () 列表类似。除非列表的元素以传值的方式传给构造函数，否则我们不会拷贝列表的元素。
- 如果 {} 列表被用于初始化一个聚合体（一个数组或者一个未提供构造函数的类）的元素，则列表的每个元素分别初始化聚合体中的一个元素。除非列表的元素以传值的方式传给聚合体元素的构造函数，否则我们不会拷贝列表的元素。
- 如果 {} 列表被用于构建一个 initializer_list 对象，则列表的每个元素分别初始化 initializer_list 的*底层数组*（underlying array）的一个元素。通常情况下，我们把元素从 initializer_list 拷贝到实际使用它们的地方。

请注意，以上只是我们用以理解 {} 列表语义的一般化模型。只要这层含义不被破坏，编译器有权采取一些更好的优化措施。

考虑如下情况：

```
vector<double> v = {1, 2, 3.14};
```

标准库 vector 含有一个接受初始化器列表的构造函数（见 17.3.4 节），因此初始化器列表 {1,2,3.14} 可以理解成一个临时数组，其构造过程和用法如下所示：

```
const double temp[] = {double{1}, double{2}, 3.14 } ;
const initializer_list<double> tmp(temp,sizeof(temp)/sizeof(double));
vector<double> v(tmp);
```

也就是说，编译器构建了一个数组，初始化器被转换成期望的类型（此处是 double）后被包含在该数组中。这个数组作为一个 initializer_list 被传给 vector 的接受初始化器列表的构造函数，该构造函数再把值从数组拷贝到它自己的存放元素的数据结构中。请注意，initializer_list 本身是个小对象（可能只占两个字空间），因此以传值方式传递它完全可行。

这个底层数组是不可修改的，所以在两次使用同一个 {} 列表期间，列表的含义不会发生变化（在标准规则范围内）。考虑如下的代码：

```
void f()
{
    initializer_list<int> lst {1,2,3};
    cout << *lst.begin() << '\n';
    *lst.begin() = 2;                   // 错误：lst 不可修改
    cout << *lst.begin() << '\n';
}
```

尤其是，{} 列表不可修改还意味着接受列表元素的容器必须使用拷贝操作，而不能使用移动操作。

{} 列表（及其对应的底层数组）的生命周期由使用该列表的作用域决定（见 6.4.2 节）。当列表被用于初始化 initializer_list<T> 类型的变量时，它的生命周期与该变量相同。而当

列表被用在一条表达式之中时（包括作为其他类型变量的初始化器，比如 vector<T>），在完整表达式的结尾之处销毁该列表。

11.3.2　限定列表

把初始化器列表用作表达式的基本思想是：如果你能用下面的语句初始化一个变量 x

```
T x {v};
```

那么你也能用 T{v} 或者 new T{v} 的形式创建一个对象并将其当成一条表达式。使用 new 会把目标对象置于自由存储之上，并返回一个指向该对象的指针；相反，“普通的 T{v}”仅在局部作用域中创建一个临时对象（见 6.4.2 节）。例如：

```
struct S { int a, b; };

void f()
{
    S v {7,8};          // 直接初始化一个变量
    v = S{7,8};         // 用限定列表进行赋值
    S* p = new S{7,8};  // 使用限定列表在自由存储上构建对象
}
```

使用限定列表构建对象与直接初始化（见 16.2.6 节）规则相同。

如果某个限定列表中只含有一个元素，则其含义基本上等同于把该元素转换成另外一种类型。例如：

```
template<class T>
T square(T x)
{
    return x*x;
}

void f(int i)
{
    double d = square(double{i});
    complex<double> z = square(complex<double>{i});
}
```

这一点在 11.5.1 节将有更详细的阐述。

11.3.3　未限定列表

当我们明确知道所用类型时，可以使用未限定列表。它只能被用作一条表达式，并且仅限于以下场景：

- 函数实参
- 返回值
- 赋值运算符（=、+=、*= 等）的右侧运算对象
- 下标

例如：

```
int f(double d, Matrix& m)
{
    int v {7};          // 初始化器（直接初始化）
    int v2 = {7};       // 初始化器（拷贝初始化）
```

```
    int v3 = m[{2,3}];      // 假设 m 接受一个值对作为其下标

    v = {8};                // 赋值运算的右侧运算对象
    v += {88};              // 赋值运算的右侧运算对象
    {v} = 9;                // 错误：不能作为赋值运算的左侧运算对象
    v = 7+{10};             // 错误：不能作为非赋值运算符的运算对象
    f({10.0});              // 函数实参
    return {11};            // 返回值
}
```

我们之所以不允许未限定列表出现在赋值运算的左侧，主要是因为 C++ 语法允许 { 出现在该位置表示复合语句（块）。如果这样做了，一方面程序的可读性会下降，另一方面编译器也会遇到二义性的问题。虽然这样的问题并非完全无法解决，但因为发现了存在此类风险，所以我们就没有让 C++ 扩展这项功能。

当我们在不使用 = 的前提下，把未限定的 {} 列表用作命名对象的初始化器时（就像上面的 v 那样），该列表直接执行初始化（见 16.2.6 节）。其他情况下，它执行拷贝初始化（见 16.2.6 节）。初始化语句中冗余的 = 限制了我们只能用给定的 {} 列表执行某些特定的初始化操作。

标准库类型 initializer_list<T> 用于处理长度可变的 {} 列表（见 12.2.3 节）。我们常把它用于用户自定义容器的初始化器列表（见 3.2.1.3 节），但是除此之外也可以直接使用它。例如：

```
int high_value(initializer_list<int> val)
{
    int high = numeric_traits<int>lowest();
    if (val.size()==0) return high;

    for (auto x : val)
        if (x>high) high = x;

    return high;
}

int v1 = high_value({1,2,3,4,5,6,7});
int v2 = high_value({-1,2,v1,4,-9,20,v1});
```

{} 列表是处理同质、变长列表的最简单的方法，但是注意 0 个元素的情况是个例外。此时，我们应该使用默认的构造函数（见 17.3.3 节）。

只有当 {} 列表的所有元素类型相同时，我们才能推断该列表的类型。例如：

```
auto x0 = {};       // 错误（缺少元素类型）
auto x1 = {1};      // initializer_list<int>
auto x2 = {1,2};    // initializer_list<int>
auto x3 = {1,2,3};  // initializer_list<int>
auto x4 = {1,2.0};  // 错误：元素类型不相同
```

可惜，我们无法通过推断未限定列表的类型使其作为普通模板的实参。例如：

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

f({});          // 错误：初始化器的类型未知
f({1});         // 错误：未限定的列表与“普通的 T”不匹配
f({1,2});       // 错误：未限定的列表与“普通的 T”不匹配
f({1,2,3});     // 错误：未限定的列表与“普通的 T”不匹配
```

我之所以说“可惜”，是因为这只是语言的限制，而非一条基本规则。仅从技术上来说，我们完全可以像那些 auto 的初始化器一样把上述 {} 列表的类型推断为 initializer_list<int>。

类似地，当容器的元素类型是模板时，我们无法推断它。例如：

```
template<class T>
void f2(const vector<T>&);

f2({1,2,3});                    // 错误：无法推断 T
f2({"Kona","Sidney"});          // 错误：无法推断 T
```

这一规定同样让人感觉很可惜的，不过从语言技术角度来看相对容易理解：毕竟要用到 vector 的情况实在太多了。要想推断 T 的类型，编译器需要首先检查用户是否真的要用到 vector，随后需要深入到 vector 的定义内部检查它是否包含可以接受 {1,2,3} 的构造函数。一般情况下，这一过程会用到 vector 的实例化（见 26.2 节）。这种思路并非行不通，但是实在太消耗编译器的时间了。此外，如果 f2() 有重载版本的话，我们还可能面临二义性的问题。如果要调用 f2()，最好写得明确一些：

```
f2(vector<int>{1,2,3});                 // OK
f2(vector<string>{"Kona","Sidney"});    // OK
```

11.4　lambda 表达式

lambda 表达式（lambda expression）有时也称为 lambda 函数（lambda function），或者直接简称为 lambda（严格来说后者并不正确，但符合口语习惯）。它是定义和使用匿名函数对象的一种简便的方式。人们习惯的传统方式是先定义一个含有 operator() 的命名类，随后再创建该类的一个对象并通过该对象调用函数；与之相比，lambda 表达式就像文字速记法一样简单易行。尤其是当我们想把操作当成实参传给算法时，这种便捷性显得尤其重要。在图形用户界面（以及其他场合）中，这样的操作常被称为回调（callback）。本节主要探讨 lambda 技术层面的问题，lambda 的应用实例则在其他章节介绍（见 3.4.3 节、32.4 节和 33.5.2 节）。

一条 lambda 表达式包含以下组成要件：

- 一个可能为空的捕获列表（capture list），指明定义环境中的哪些名字能被用在 lambda 表达式内，以及这些名字的访问方式是拷贝还是引用。捕获列表位于 [] 内（见 11.4.3 节）。
- 一个可选的参数列表（parameter list），指明 lambda 表达式所需的参数。参数列表位于 () 内（见 11.4.4 节）。
- 一个可选的 mutable 修饰符，指明该 lambda 表达式可能会修改它自身的状态（即，改变通过值捕获的变量的副本，见 11.4.3.4 节）。
- 一个可选的 noexcept 修饰符。
- 一个可选的 -> 形式的返回类型声明（见 11.4.4 节）。
- 一个表达式体（body），指明要执行的代码，表达式体位于 {} 内（见 11.4.3 节）。

在 lambda 的概念中，传参、返回结果以及定义表达式体等环节都与函数的相应概念是一致的，这些内容将在第 12 章介绍。区别在于函数没有提供局部变量“捕获”的功能，这意味着 lambda 可以作为局部函数使用，而普通函数不能。

11.4.1　实现模型

我们可以用很多种不同的方式实现 lambda 表达式，并且优化的途径也有很多。然而我

发现，如果把 lambda 表达式看成是一种定义并使用函数对象的便捷方式，将非常有助于我们理解 lambda 表达式的语义。有个相对简单的例子：

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // 如果 v[i]%m==0，则输出 v[i] 到 os
{
    for_each(begin(v),end(v),
        [&os,m](int x) { if (x%m==0) os << x << '\n'; }
    );
}
```

要想理解上述代码的含义，我们不妨定义一个等价的函数对象：

```
class Modulo_print {
    ostream& os;  // 用于存放捕获列表的成员
    int m;
public:
    Modulo_print(ostream& s, int mm) :os(s), m(mm) {}     // 捕获
    void operator()(int x) const
        { if (x%m==0) os << x << '\n'; }
};
```

其中，捕获列表 [&os,m] 变成了两个成员变量以及一个用于初始化它们的构造函数。os 之前的 & 表示它是一个引用，m 之前没有 & 则表示它是一个副本。这种使用 & 的方式与函数实参声明的使用方式完全一致。

lambda 的主体部分变为了 operator()() 的函数体。因为 lambda 并不返回值，所以 operator()() 是 void。默认情况下，operator()() 是 const，因此在 lambda 体内部无法修改捕获的变量，这也是目前为止最常见的情况。如果你确实希望在 lambda 的内部修改其状态，则应该把它声明为 mutable（见 11.4.3.4 节）。当然，此时对应的 operator()() 就不能声明为 const 了。

我们把由 lambda 生成的类的对象称为闭包对象（closure object，或者简称为闭包）。一开始的那个函数将改写成如下形式：

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // 如果 v[i]%m==0，则输出 v[i] 到 os
{
    for_each(begin(v),end(v),Modulo_print{os,m});
}
```

如果 lambda 通过引用（使用捕获列表 [&]）捕获它的每个局部变量，则其闭包对象可以优化为简单地包含一个指向外层栈框架的指针。

11.4.2　lambda 的替代品

print_modulo() 的最终版本实际上非常棒，并且为重要操作命名也不失为一个好主意。同时，单独定义的类为我们添加注释留出了足够的空间；如果我们在实参列表中嵌入 lambda 表达式，显然无法做到这一点。

然而，很多 lambda 表达式很小且只用一次。此时，一种比较现实的做法是定义一个局部类，并且定义的位置就在使用之前。例如：

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // 如果 v[i]%m==0，则输出 v[i] 到 os
{
```

```
        class Modulo_print {
            ostream& os;  // 用于存放捕获列表的成员
            int m;
        public:
            Modulo_print (ostream& s, int mm) :os(s), m(mm) {}    // 捕获
            void operator()(int x) const
                { if (x%m==0) os << x << '\n'; }
        };

        for_each(begin(v),end(v),Modulo_print(os,m));
    }
```

与之相比，使用 lambda 明显更优。如果我们确实需要一个名字，那就命名 lambda：

```
    void print_modulo(const vector<int>& v, ostream& os, int m)
        // 如果 v[i]%m==0，则输出 v[i] 到 os
    {
        auto Modulo_print = [&os,m] (int x) { if (x%m==0) os << x << '\n'; };

        for_each(begin(v),end(v),Modulo_print);
    }
```

通常情况下，命名 lambda 是一种有效的手段。这么做可以让我们把注意力集中在如何设计操作本身上，同时，代码布局更加简单，也能使用递归了（见 11.4.5 节）。

对于使用 for_each() 的 lambda 表达式来说，我们可以编写一个 for 循环作为替代。例如：

```
    void print_modulo(const vector<int>& v, ostream& os, int m)
        // 如果 v[i]%m==0，则输出 v[i] 到 os
    {
        for (auto x : v)
            if (x%m==0) os << x << '\n';
    }
```

读者会感觉这个版本比所有 lambda 版本都更简洁明了。但是，for_each 毕竟是种太特殊的算法，同时 vector<int> 也只是一种特殊的容器。我们泛化 print_modulo()，令其可以处理更多容器类型：

```
    template<class C>
    void print_modulo(const C& v, ostream& os, int m)
        // 如果 v[i]%m==0，则输出 v[i] 到 os
    {
        for (auto x : v)
            if (x%m==0) os << x << '\n';
    }
```

这个版本可以很好地处理 map：C++ 的范围 for 语句专门处理从序列头遍历到序列尾的特殊情况。STL 容器使得此类遍历易于执行，且具有通用性。例如，用 for 语句遍历一个 map 会进行深度优先遍历。我们应该如何进行宽度优先遍历呢？ for 循环版本的 print_modulo 难以修改为宽度优先遍历，因此我们必须将 for 循环改写为一个算法。例如：

```
    template<class C>
    void print_modulo(const C& v, ostream& os, int m)
        // 如果 v[i]%m==0，则输出 v[i] 到 os
    {
        breadth_first(begin(v),end(v),
```

```
        [&os,m](int x) { if (x%m==0) os << x << '\n'; }
    );
}
```

其中，我们以算法的形式表示了一个泛化的循环 / 遍历，而 lambda 表达式是它的“主体”。使用 for_each 代替 breadth_first 就会进行深度优先遍历。

当 lambda 作为遍历算法的实参时，其性能与对应的循环等价（通常完全一致）。这一结论与具体实现或者平台无关。因此，当我们需要在“算法 +lambda”和“for 语句”之间进行抉择时，评判标准应该是格式的优劣以及对于可扩展性和可维护性的评估。

11.4.3　捕获

lambda 的主要用途是封装一部分代码以便于将其用作参数。lambda 允许我们“内联地”这么做，而无须命名一个函数（或者函数对象）然后在别处使用它。有些 lambda 无须访问它的局部环境，这样的 lambda 使用空引入符 [] 定义。例如：

```
void algo(vector<int>& v)
{
    sort(v.begin(),v.end());    // 排列值
    // ...
    sort(v.begin(),v.end(),[](int x, int y) { return abs(x)<abs(y); });        // 排列绝对值
    // ...
}
```

如果我们需要访问局部名字，就必须明确指出，否则会产生错误：

```
void f(vector<int>& v)
{
    bool sensitive = true;
    // ...
    sort(v.begin(),v.end(),
        [](int x, int y) { return sensitive ? x<y : abs(x)<abs(y); }    // 错误：无权访问
    );
}
```

我使用了 lambda 引入符（lambda introducer）[]，它是最简单的 lambda 引入符，不允许 lambda 访问其调用环境中的名字。对于一条 lambda 表达式来说，它的第一个字符永远是 [。lambda 引入符的形式有很多种：

- []：空捕获列表。这意味着在 lambda 内部无法使用其外层上下文中的任何局部名字。对于这样的 lambda 表达式来说，其数据需要从实参或者非局部变量中获得。
- [&]：通过引用隐式捕获。所有局部名字都能使用，所有局部变量都通过引用访问。
- [=]：通过值隐式捕获。所有局部名字都能使用，所有名字都指向局部变量的副本，这些副本是在 lambda 表达式的调用点获得的。
- [*捕获列表*]：显式捕获；*捕获列表*是通过值或者引用的方式捕获的局部变量（即，存储在对象中）的名字列表。以 & 为前缀的变量名字通过引用捕获，其他变量通过值捕获。捕获列表中可以出现 this，或者紧跟 ... 的名字以表示元素。
- [&, *捕获列表*]：对于名字没有出现在捕获列表中的局部变量，通过引用隐式捕获。捕获列表中可以出现 this。列出的名字不能以 & 为前缀。捕获列表中的变量名通过值的方式捕获。

- [=, 捕获列表]：对于名字没有出现在捕获列表中的局部变量，通过值隐式捕获。捕获列表中不允许包含 this。列出的名字必须以 & 为前缀。捕获列表中的变量名通过引用的方式捕获。

请注意，以 & 为前缀的局部名字总是通过引用捕获，相反地，不以 & 为前缀的局部名字总是通过值捕获。只有通过引用的捕获允许修改调用环境中的变量。

有时候，我们会指定捕获列表（capture-list）。这么做有助于我们细粒度地管理和控制调用环境中的哪些名字能被使用以及如何使用。例如：

```
void f(vector<int>& v)
{
    bool sensitive = true;
    // ...
    sort(v.begin(),v.end()
        [sensitive](int x, int y) { return sensitive ? x<y : abs(x)<abs(y); }
    );
}
```

通过在捕获列表中列出 sensitive，我们就可以在 lambda 的内部访问它了。除此之外我们没有指定任何其他内容，因此可以确保 sensitive 的捕获是通过“值”的方式进行的。这一点与函数传参非常相似，对于参数传递来说，默认情况下传递的是副本。如果我们希望以“引用”的方式捕获 sensitive，则应该在捕获列表的 sensitive 之前加一个 &：[&sensitive]。

到底该通过值还是通过引用捕获名字呢？选择的依据其实与函数实参完全一致（见 12.2 节）。如果我们希望向捕获的对象写入内容，或者捕获的对象很大，则应该使用引用。然而，对于 lambda 来说，还应该注意 lambda 的有效期可能会超出它的调用者（见 11.4.3.1 节）。当把 lambda 传递给其他线程时，一般来说通过值捕获（[=]）更优；通过引用或者指针访问其他线程的栈内容是一种危险的操作（对于性能和正确性都是如此），更严重的是，试图访问一个已终止线程的栈内容会引发极难发现的程序错误。

如果你想捕获可变模板实参（见 28.6 节），可以使用 ...，例如：

```
template<typename... Var>
void algo(int s, Var... v)
{
    auto helper = [&s,&v...] { return s*(h1(v...)+h2(v...)); }
    // ...
}
```

千万不要滥用捕获机制。很多时候我们既可以捕获，也可以传递实参。捕获的方式虽然精简，但是极易引发混淆。

11.4.3.1　lambda 与生命周期

lambda 的生命周期可能比它的调用者更长。当我们把 lambda 传递给另外一个线程或者被调用者把 lambda 存在别处以供后续使用时，这种情况就会发生。例如：

```
void setup(Menu& m)
{
    // ...
    Point p1, p2, p3;
    // ... 计算 p1, p2 和 p3 的位置 ...
    m.add("draw triangle",[&]{ m.draw(p1,p2,p3); });    // 可能会发生悬空指针错误
    // ...
}
```

假定 add() 负责把一个（名字，动作）对添加到菜单中，并且 draw() 操作是有效的，则上述程序无异于埋下了一颗定时炸弹：setup() 完成之后——也许要到好几分钟之后——用户点了 draw triangle 按钮，此时 lambda 将会试图访问一个早已不存在的局部变量。如果在某些程序中 lambda 需要向通过引用捕获的变量写入内容，情况就更糟糕了。

因此，如果我们发现 lambda 的生命周期可能比它的调用者更长，就必须确保所有局部信息（如果有的话）都被拷贝到闭包对象中，并且这些值应该通过 return 机制（见 12.1.4 节）或者适当的实参返回。对于 setup() 的例子来说，很容易做到这一点：

```
m.add("draw triangle",[=]{ m.draw(p1,p2,p3); });
```

为了便于理解，不妨把捕获列表看成闭包对象的初始化器列表，同时把 [=] 和 [&] 看成一种速记符号（见 11.4.1 节）。

11.4.3.2　名字空间名字

因为名字空间变量（包括全局变量）永远是可访问的（确保在作用域内），所以我们无须“捕获”它们。例如：

```
template<typename U, typename V>
ostream& operator<<(ostream& os, const pair<U,V>& p)
{
    return os << '{' << p.first << ',' << p.second << '}';
}

void print_all(const map<string,int>& m, const string& label)
{
    cout << label << ":\n{\n";
    for_each(m.begin(),m.end(),
        [](const pair<string,int>& p) { cout << p << '\n'; }
    );
    cout << "}\n";
}
```

在这里，我们无须捕获 cout 或者 pair 的输出运算符。

11.4.3.3　lambda 与 this

当 lambda 被用在成员函数中时，我们该如何访问类对象的成员呢？我们的做法是把 this 添加到捕获列表中，这样类的成员就位于可被捕获的名字集合中了。当我们希望在成员函数的实现中使用 lambda 时，这种做法特别有效。例如，我们构建了一个名为 Request 的类，它的作用是建立请求并查询结果：

```
class Request {
    function<map<string,string>(const map<string,string>&)> oper;      // 操作
    map<string,string> values;                  // 参数
    map<string,string> results;                 // 目标
public:
    Request(const string& s);                   // 解析并保存请求

    void execute()
    {
        [this]() { results=oper(values); }      // 根据结果执行相应的操作
    }
};
```

成员通过引用的方式捕获。也就是说，[this] 意味着成员是通过 this 访问的，而非拷贝到

lambda 中。不幸的是，[this] 和 [=] 互不兼容，因此稍有不慎就可能在多线程程序中产生竞争条件（见 42.4.6 节）。

11.4.3.4　mutable 的 lambda

通常情况下，人们不希望修改函数对象（闭包）的状态，因此默认设置为不可修改，换句话说，生成的函数对象（见 11.4.1 节）的 operator()() 是一个 const 成员函数。只有在极少数情况下，如果我们确实希望修改状态（注意，不是修改通过引用捕获的变量的状态，参见 11.4.3 节），则可以把 lambda 声明成 mutable 的。例如：

```
void algo(vector<int>& v)
{
    int count = v.size();
    std::generate(v.begin(),v.end(),
        [count]()mutable{ return --count; }
    );
}
```

其中，--count 负责递减闭包中 v 的副本的尺寸。

11.4.4　调用与返回

向 lambda 传递参数的规则与向函数传递参数是一样的（见 12.2 节），从 lambda 返回结果也是如此（见 12.1.4 节）。实际上，除了关于捕获的规则之外（见 11.4.3 节），lambda 的大多数规则都是从函数和类借鉴而来的。然而，有两点需要注意：

[1] 如果一条 lambda 表达式不接受任何参数，则其参数列表可被忽略。因此，lambda 表达式的最简形式是 []{}。

[2] lambda 表达式的返回类型能由 lambda 表达式本身推断得到，然而函数无法做到这一点。

如果在 lambda 的主体部分不包含 return 语句，则该 lambda 的返回类型是 void。如果 lambda 的主体部分只包含一条 return 语句，则该 lambda 的返回类型是该 return 表达式的类型。其他情况下，我们必须显式地提供一个返回类型。例如：

```
void g(double y)
{
    [&]{ f(y); }                                    // 返回类型是 void
    auto z1 = [=](int x){ return x+y; }             // 返回类型是 double
    auto z2 = [=,y]{ if (y) return 1; else return 2; }
                                                    // 错误：lambda 主体部分过于复杂，
                                                    // 无法推断其类型
    auto z3 =[y]() { return 1 : 2; }                // 返回类型是 int
    auto z4 = [=,y]()->int { if (y) return 1; else return 2; }
                                                    // OK：显式的返回类型
}
```

如果使用了后缀返回类型，则可以忽略参数列表。

11.4.5　lambda 的类型

为了适应可能对 lambda 表达式进行的优化，我们没有定义 lambda 表达式的类型。然而，在 11.4.1 节呈现的形式中它被定义为函数对象的类型。这个类型称为闭包类型（closure type），它对于 lambda 表达式来说是唯一的。因此，任意两个 lambda 的类型都不相同。一旦两个 lambda 具有相同的类型，模板实例化机制就无法辨识它们了。lambda 是一种局部类类型，它含有一个构造函数以及一个 const 成员函数 operator()()。lambda 除了能作为参数外，

还能用于初始化一个声明为 auto 或者 std::function<R(AL)> 的变量。其中，R 是 lambda 的返回类型，AL 是它的类型参数列表（见 33.5.3 节）。

例如，我试图编写一个能更改 C 风格字符串中字符的 lambda：

```
auto rev = [&rev](char* b, char* e)
                { if (1<e-b) { swap(*b,*--e); rev(++b,e); } };          // 错误
```

然而，由于我们无法在推断出一个 auto 变量的类型之前使用它，因此上面的写法根本行不通。相反，我们应该先引入一个新的名字，然后使用它：

```
void f(string& s1, string& s2)
{
    function<void(char* b, char* e)> rev =
        [&](char* b, char* e) { if (1<e-b) { swap(*b,*--e); rev(++b,e); } };
    rev(&s1[0],&s1[0]+s1.size());
    rev(&s2[0],&s2[0]+s2.size());
}
```

这样做，我们就可以确保在使用 rev 之前知道它的类型了。

如果我们只是想给 lambda 起个名字，而不会递归地使用它，则可以考虑使用 auto：

```
void g(vector<string>& vs1, vector<string>& vs2)
{
    auto rev = [&](char* b, char* e) { while (1<e-b) swap(*b++,*--e); };

    rev(&s1[0],&s1[0]+s1.size());
    rev(&s2[0],&s2[0]+s2.size());
}
```

如果一个 lambda 什么也不捕获，则我们可以将它赋值给一个指向正确类型函数的指针。例如：

```
double (*p1)(double) = [](double a) { return sqrt(a); };
double (*p2)(double) = [&](double a) { return sqrt(a); };        // 错误：lambda 捕获了内容
double (*p3)(int) = [](int a) { return sqrt(a); };               // 错误：参数类型不匹配
```

11.5　显式类型转换

有时候，我们必须把一种类型的值转换成另一种类型的值。很多（可以说太多）这种类型转换都是根据语言的规则隐式执行的（见 2.2.2 节和 10.5 节）。例如：

```
double d = 1234567890;  // 整数转换成浮点数
int i = d;              // 浮点数转换成整数
```

但是在另外一些情况下，我们必须显式地转换类型。

出于历史的原因，当然也有一些逻辑方面的考虑在内，C++ 提供了多种显式类型转换的操作，这些操作在便利程度和安全性上都有所不同：

- 构造，使用 {} 符号提供对新值类型安全的构造（见 11.5.1 节）
- 命名的转换，提供不同等级的类型转换：
  - const_cast，对某些声明为 const 的对象获得写入的权利（见 7.5 节）
  - static_cast，反转一个定义良好的隐式类型转换（见 11.5.2 节）
  - reinterpret_cast，改变位模式的含义（见 11.5.2 节）
  - dynamic_cast，动态地检查类层次关系（见 22.2.1 节）
- C 风格的转换，提供命名的类型转换或其组合（见 11.5.3 节）

- 函数化符号，提供 C 风格转换的另一种形式（见 11.5.4 节）

其中，我是按照自己的偏好和使用的安全性来排列这些转换的。

除了 {} 构造符号之外，我对上面这些转换都没什么好感。当然，至少 dynamic_cast 执行了运行时检查。对于发生在两种标量数字类型之间的转换，我倾向于使用一个自制的显式类型转换函数 narrow_cast；此时，值可能会被窄化：

```
template<class Target, class Source>
Target narrow_cast(Source v)
{
    auto r = static_cast<Target>(v);          // 把值转换成目标类型
    if (static_cast<Source>(r)!=v)
        throw runtime_error("narrow_cast<>() failed");
    return r;
}
```

也就是说，如果把某个值转换成目标类型而在这个过程中发生了窄化运算，则把结果转换成原类型，并且恢复原值。我对这种处理效果非常满意，它可以看作是对 {} 初始化器的语言规则的推广（见 6.3.5.2 节）。例如：

```
void test(double d, int i, char* p)
{
    auto c1 = narrow_cast<char>(64);
    auto c2 = narrow_cast<char>(-64);      // 如果字符是无符号的，则抛出异常
    auto c3 = narrow_cast<char>(264);      // 如果字符是 8 位，则抛出异常

    auto d1 = narrow_cast<double>(1/3.0F); // OK
    auto f1 = narrow_cast<float>(1/3.0);   // 很可能抛出异常

    auto c4 = narrow_cast<char>(i);        // 可能抛出异常
    auto f2 = narrow_cast<float>(d);       // 可能抛出异常

    auto p1 = narrow_cast<char*>(i);       // 编译时错误
    auto i1 = narrow_cast<int>(p);         // 编译时错误

    auto d2 = narrow_cast<double>(i);      // 可能抛出异常（在最大多数系统下不会）
    auto i2 = narrow_cast<int>(d);         // 可能抛出异常
}
```

根据浮点数具体用法的不同，有可能对浮点数转换应该使用范围检查而非 !=。我们可以通过使用特例化（见 25.3.4.1 节）或者类型萃取（见 35.4.1 节）来做到这一点。

11.5.1　构造

用值 e 构建一个类型为 T 的值可以表示为 T{e}（§ iso.8.5.4），例如：

```
auto d1 = double{2};          // d1==2.0
double d2 {double{2}/4};      // d2==0.5
```

符号 T{v} 有一个好处，就是它只执行“行为良好的”类型转换。例如：

```
void f(int);
void f(double);

void g(int i, double d)
{
    f(i);                     // 调用 f(int)
```

```
    f(double{i});                    // 错误：{} 拒绝执行整数向浮点数的类型转换

    f(d);                            // 调用 f(double)
    f(int{d});                       // 错误：{} 拒绝截断的行为
    f(static_cast<int>(d));          // 调用 f(int)，传入的是个截断的值

    f(round(d));                     // 调用 f(double)，传入的是个四舍五入的值
    f(static_cast<int>(lround(d))); // 调用 f(int)，传入的是个四舍五入的值
                                     // 如果 round(d) 溢出 int 的范围，它仍会被截断
}
```

我不认为截断浮点数（比如 7.9 到 7）是“良好的行为”。因此如果你真的希望这么做，就必须显式地指出来。如果你希望四舍五入，可以使用标准库函数 round()；它执行“传统的四舍五入”，比如 7.9 到 8 而 7.4 到 7。

看起来，{} 构造不允许 int 向 double 的转换有点奇怪。但真正的原因是，如果 int 和 double 所占的位数一样（这种情况并不罕见），则其中的某些 int 向 double 转换必将损失信息。考虑下面的情况：

```
static_assert(sizeof(int)==sizeof(double),"unexpected sizes");

int x = numeric_limits<int>::max(); // 可能的最大整数值
double d = x;
int y = x;
```

该程序不会得到 x==y 的结果。不过，我们仍然可以使用一个能明确表示出来的整数字面值常量初始化 double。例如：

```
double d { 1234 };   // 正确
```

一旦我们在程序中使用目标类型作为显式的限定，则在这种情况下就不允许行为不正常的类型转换了。例如：

```
void g2(char* p)
{
    int x = int{p};          // 错误：不存在 char* 向 int 的类型转换
    using Pint = int*;
    int* p2 = Pint{p};       // 错误：不存在 char* 向 int* 的类型转换
    // ...
}
```

对 T{v} 来说，“行为非常良好”的含义是存在 v 向 T 的“非窄化”（见 10.5 节）类型转换或者有一个 T 的类型正确的构造函数（见 17.3 节）。

构造函数符号 T{} 用于表示类型 T 的默认值，例如：

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

void g3()
{
    f(int{});                // 默认的 int 值
    f(complex<double>{});    // 默认的 complex 值
    // ...
}
```

对于内置类型来说，显式使用其构造函数得到的值是该类型对应的 0 值（见 6.3.5 节）。因此，int{} 可以看成是 0 的另一种写法。对于用户自定义类型 T 来说，如果含有默认构造函数（见 3.2.1.1 节和 17.6 节），则 T{} 的结果由默认构造函数定义；否则，由每个成员的默认

构造函数 MT{} 定义。

显式构造的未命名对象是临时对象，而且（除非绑定到引用上）其生命周期仅限于其所在的完整表达式（见 6.4.2 节）。在这一点上，它们与通过 new 创建的未命名对象是有区别的（见 11.2 节）。

11.5.2　命名转换

有的类型转换行为不够良好或者不易进行类型检查，它们所转换的值并非来源于定义良好的值的集合。例如：

```
IO_device* d1 = reinterpret_cast<IO_device*>(0Xff00);  // 0Xff00 处的设备
```

编译器并不能确定整数 0Xff00 是否是一个有效的（I/O 设备寄存器）地址。因此这条转换语句的正确性完全依赖于程序员自己。显式类型转换也称为强制类型转换（casting），只有在极个别的情况下才有用。然而在过去，显式类型转换被严重滥用了，成为了程序错误的主要来源。

显式类型转换的另一个典型应用场景是处理“原始内存”，也就是说编译器对其中所存的对象或者将要存储的对象类型未知的内存。例如，内存分配器（比如 operator new()，见 11.2.3 节）可能会返回一个指向新分配内存的 void*：

```
void* my_allocator(size_t);

void f()
{
    int* p = static_cast<int*>(my_allocator(100));      // 新分配的空间被用作 int
    // ...
}
```

编译器不知道 void* 所指对象的类型。

命名转换背后隐藏的基本思想是令类型转换的含义更明显，并且让程序员有机会表达他们的真实意图：

- static_cast 执行关联类型之间的转换，比如一种指针类型向同一个类层次中其他指针类型的转换，或者整数类型向枚举类型的转换，或者浮点类型向整数类型的转换。它还能执行构造函数（见 16.2.6 节，18.3.3 节，§ iso.5.2.9）和转换运算符（见 18.4 节）定义的类型转换。
- reinterpret_cast 处理非关联类型之间的转换，比如整数向指针的转换以及指针向另一个非关联指针类型的转换（见 § iso.5.2.10）。
- const_cast，参与转换的类型仅在 const 修饰符及 volatile 修饰符上有所区别（见 § iso.5.2.11）。
- dynamic_cast 执行指针或者引用向类层次体系的类型转换，并执行运行时检查（见 22.2.1 节和 § iso.5.2.7）。

这些不同的命名类型转换使得编译器可以执行一些最小化的类型检查，同时程序员很容易发现 reinterpret_cast 表示的非常危险的类型转换。一些 static_cast 是可移植的，但是只有很少 reinterpret_cast 具有这一性质。我们几乎无法为 reinterpret_cast 担保任何事，但是通常情况下它产生的新类型的值与它的实参具有相同的位模式。如果目标值所占的位数不少于原始值，则我们能把结果 reinterpret_cast 回它的原始类型并使用它。只有

当 reinterpret_cast 的结果能被精确地转换回原始类型时，该结果才是可用的。请注意，reinterpret_cast 必须作用于函数指针（见 12.5 节）。考虑如下的情况：

```
char x = 'a';
int* p1 = &x;                          // 错误：不存在 char* 向 int* 的隐式类型转换
int* p2 = static_cast<int*>(&x);       // 错误：不存在 char* 向 int* 的隐式类型转换
int* p3 = reinterpret_cast<int*>(&x);  // OK：责任自负

struct B { /* ... */ };
struct D : B { /* ... */ };            // 见 3.2.2 节和 20.5.2 节

B* pb = new D;                         // OK：D*：向 B* 的隐式类型转换
D* pd = pb;                            // 错误：不存在 B* 向 D* 的隐式类型转换
D* pd = static_cast<D*>(pb);           // OK
```

类指针和类引用之间的类型转换将在 22.2 节讨论。

在决定使用显式类型转换之前，请花时间仔细考虑一下是否真的必须这么做。很多情况下，C（见 1.3.3 节）或者早期版本的 C++（见 1.3.2 节和 44.2.3 节）需要用到显式类型转换，但是现在的 C++ 语言并不需要。在很多程序中，我们完全可以避免使用显式类型转换；即使使用，也应该限定在少量代码片段中。

11.5.3　C 风格的转换

C++ 从 C 继承了符号 (T)e，它可以执行 static_cast、reinterpret_cast 和 const_cast 任意组合之后得到的类型转换，它的含义是从表达式 e 得到类型为 T 的值（见 44.2.3 节）。不幸的是，C 风格的类型转换允许把类的指针转换成指向该类私有基类的指针。切记永远不要这么做，而且最好祈祷你的编译器能发现此类错误。C 风格的类型转换比命名的转换运算符危险得多，原因是我们很难在一个规模较大的程序中定位它，并且不容易理解程序员真实的类型转换意图。也就是说，(T)e 可能是执行关联类型之间的可移植的类型转换，也可能是非关联类型之间的不可移植的类型转换，还有可能是移除掉指针类型前面的 const 修饰符。由于 T 和 e 的类型并不明确，所以我们无法得到确切的结论。

11.5.4　函数形式的转换

用值 e 构建类型 T 的值的过程可以表示为函数形式的符号 T(e)，例如：

```
void f(double d)
{
    int i = int(d);                // 截断
    complex z = complex(d); // 从 d 构建一个 complex
    // ...
}
```

T(e) 有时称为函数形式的转换（function-style cast）。不幸的是，对于内置类型 T 来说，T(e) 等价于 (T)e（见 11.5.3 节）。这意味着对于大多数内置类型来说，T(e) 并不安全。

```
void f(double d, char* p)
{
    int a = int(d);   // 截断
    int b = int(p);   // 不可移植
    // ...
}
```

即使是从一种较长的整数类型向较短的整数类型（比如 long 向 char）的显式类型转换也会导致不可移植的依赖于实现的行为。

建议用 T{v} 处理行为良好的构造，用命名的转换（比如 static_cast）处理其他任务。

11.6 建议

[1] 与后置 ++ 运算符相比，建议优先使用前置 ++ 运算符；11.1.4 节。

[2] 使用资源句柄避免泄漏、提前删除和重复删除；11.2.1 节。

[3] 除非万不得已，否则不要把对象放在自由存储上；优先使用作用域内的变量；11.2.1 节。

[4] 避免使用“裸 new”和“裸 delete”；11.2.1 节。

[5] 使用 RAII；11.2.1 节。

[6] 如果需要对操作添加注释，则应该选用命名的函数对象而非 lambda；11.4.2 节。

[7] 如果操作具有一定的通用性，则应该选用命名的函数对象而非 lambda；11.4.2 节。

[8] lambda 应该尽量简短；11.4.2 节。

[9] 出于可维护性和正确性的考虑，通过引用的方式捕获一定要慎之再慎；11.4.3.1 节。

[10] 让编译器推断 lambda 的返回类型；11.4.4 节。

[11] 用 T{e} 构造值；11.5.1 节。

[12] 避免显式类型转换（强制类型转换）；11.5 节。

[13] 当不得不使用显式类型转换时，尽量使用命名的转换；11.5 节。

[14] 对于数字类型之间的转换，考虑使用运行时检查的强制类型转换，比如 narrow_cast<>()；11.5 节。

第 12 章

The C++ Programming Language, Fourth Edition

函　数

去死吧，狂热者们！

——Paradox

- 函数声明
  为什么使用函数；函数声明的组成要件；函数定义；返回值；inline 函数；constexpr 函数；[[noreturn]] 函数；局部变量
- 参数传递
  引用参数；数组参数；列表参数；数量未定的参数；默认参数
- 重载函数
  自动重载解析；重载与返回类型；重载与作用域；多实参解析；手动重载解析
- 前置与后置条件
- 函数指针
- 宏
  条件编译；预定义宏；编译指令
- 建议

12.1　函数声明

在 C++ 程序中要想做点什么事，最好的办法是调用一个函数来完成它。定义函数的过程就是描述某项操作应该如何执行的过程。我们必须首先声明一个函数，然后才能调用它。

函数声明负责指定函数的名字、返回值（如果有的话）的类型以及调用该函数所需的参数数量和类型。例如：

```
Elem* next_elem();      // 无实参，返回 Elem*
void exit(int);         // int 类型的参数，无返回值
double sqrt(double);    // double 类型的参数，返回 double
```

参数传递的语义与拷贝初始化（见 16.2.6 节）的语义完全一致。编译器检查实参的类型，如果需要的话还会执行隐式参数类型转换。例如：

```
double s2 = sqrt(2);        // 用实参 double{2} 调用函数 sqrt()
double s3 = sqrt("three");  // 错误：函数 sqrt() 需要 double 类型的参数
```

对参数类型的检查和转换非常重要，程序员应给予足够的重视。

在函数声明中可以包含参数的名字，这么做有助于读者理解函数的含义。但是，除非该声明同时也是一条函数定义语句，否则编译器将直接忽略参数的名字。当作为返回类型使用时，void 意味着函数不返回任何值（见 6.2.7 节）。

函数的类型既包括返回类型也包括参数的类型。对于类成员函数（见 2.3.2 节，16.2 节）来说，类的名字也是函数类型的一部分。例如：

```
double f(int i, const Info&);          // 类型：double(int,const Info&)
char& String::operator[](int);         // 类型：char& String::(int)
```

12.1.1 为什么使用函数

长久以来，很多程序员身上都有一种不好的习惯，他们编写的函数很长，通常有几百行。我曾经见到过一个手工编写的函数，居然超过了32 768行。这类长函数的作者忽视了函数的一项重要作用，即，把一个复杂的运算分解为若干有意义的片段，然后分别为它们命名。我们希望代码是易于理解的，因为这是实现可维护性的第一步。而代码易于理解的第一步恰恰就是把计算任务分解为易于理解的小块（表示成函数或者类）并为它们命名。函数提供了计算的基本词汇表，就像数据类型（内置类型以及用户自定义类型）提供了数据的基本词汇表一样。我们可以向C++标准算法（比如find、sort和iota）借鉴和学习如何使用函数（见第32章）。随后，我们就能把那些完成常规任务或者特殊任务的函数组合在一起执行比较复杂的计算了。

通常情况下，代码中错误的数量与代码的规模及复杂程度密切相关。通过使用更多更短小的函数，我们可以在一定程度上解决这一问题。使用函数来执行某项具体任务，可以让我们避免在代码中间人为地嵌入另一段相对独立的代码。通过构建函数，我们就能把注意力集中在为活动命名以及厘清其依赖关系上。同时，函数的调用和返回还能让我们远离那些充满错误风险的控制结构，比如goto（见9.6节）和continue（见9.5.5节）。另外，在你的代码中应该尽量避免使用嵌套的循环，因为除非这类循环写得非常规范，否则它们通常都包含大量错误（例如，应该用点乘而非嵌套的循环来实现矩阵算法，见40.6节）。

关于函数的一条最基本的建议是应该令其规模较小，以便于我们一眼就能知悉该函数的全部内容。如果我们一次只能看到算法的一小部分，那么程序就很可能会发生错误。对于大多数程序员来说，函数的规模最好控制在大约40行以内。对于我自己而言，最理想的函数规模要更小，平均7行左右。

大多数情况下，调用函数所产生的代价并不是影响程序性能的关键因素。一旦我们发现调用函数的代价比较高（比如向量的取下标运算等被频繁使用的函数），可以考虑使用内联机制来消除其影响（见12.1.5节）。程序员应当把函数视作代码的一种结构化机制。

12.1.2 函数声明的组成要件

函数声明除了指定函数的名字、一组参数以及函数的返回类型外，还可以包含多种限定符和修饰符。我们将其总结如下：

- 函数的名字：必选
- 参数列表，可以为空()：必选
- 返回类型，可以是void，可以是前置或者后置形式（使用auto）：必选
- inline，表示一种愿望：通过内联函数体实现函数调用（见12.1.5节）
- constexpr，表示当给定常量表达式作为实参时，应该可以在编译时对函数求值（见12.1.6节）
- noexcept，表示该函数不允许抛出异常（见13.5.1.1节）
- 链接说明，例如static（见15.2节）
- [[noreturn]]，表示该函数不会用常规的调用/返回机制返回结果（见12.1.4节）

此外，成员函数还能被限定为：

- virtual，表示该函数可以被派生类覆盖（见 20.3.2 节）
- override，表示该函数必须覆盖基类中的一个虚函数（见 20.3.4.1 节）
- final，表示该函数不能被派生类覆盖（见 20.3.4.2 节）
- static，表示该函数不与某一特定的对象关联（见 16.2.12 节）
- const，表示该函数不能修改其对象的内容（见 3.2.1.1 节，16.2.9.1 节）

如果你想让代码的读者头疼，不妨把函数声明成下面的形式：

```
struct S {
    [[noreturn]] virtual inline auto f(const unsigned long int *const) -> void const noexcept;
};
```

12.1.3　函数定义

如果函数会在程序中调用，那么它必须在某处定义（只定义一次，见 15.2.3 节）。函数定义是特殊的函数声明，它给出了函数体的内容。例如：

```
void swap(int*, int*);          // 声明

void swap(int* p, int* q)       // 定义
{
    int t = *p;
    *p = *q;
    *q = t;
}
```

函数的定义以及全部声明必须对应同一类型。不过，为了与 C 语言兼容，我们会自动忽略参数类型的顶层 const。例如，下面两条声明语句对应的是同一个函数：

```
void f(int);          // 类型是 void(int)
void f(const int);    // 类型是 void(int)
```

函数 f() 可以定义成：

```
void f(int x) { /* 允许在此处修改 x */ }
```

或者定义成：

```
void f(const int x) { /* 允不允许在此处修改 x */ }
```

不论对于哪种情况而言，允许修改实参也好，不允许修改实参也好，它都只是函数调用者提供的实参的一个副本。因此，调用过程不会破坏调用上下文的数据安全性。

函数的参数名字不属于函数类型的一部分，不同的声明语句中参数的名字无须保持一致。例如：

```
int& max(int& a, int& b, int& c); // 返回一个引用，对应 a、b、c 中较大的一个

int& max(int& x1, int& x2, int& x3)
{
    return (x1>x2)? ((x1>x3)?x1:x3) : ((x2>x3)?x2:x3);
}
```

对于一条非定义的声明语句来说，为参数命名的好处是可以简化代码文档，但是我们不一定非得这么做。相反，我们通常通过不命名某个参数来表示该参数未在函数定义中使用。例如：

```
void search(table* t, const char* key, const char*)
{
    // 无用到第3个参数
}
```

一般来说，未命名的参数有助于简化代码并提升代码的可扩展性。此时，尽管某些参数未被使用，但是为其预留位置可以确保函数的调用者不会受到未来函数变动的影响。

除了函数以外，我们还能调用其他一些东西，它们遵循函数的大多数规则，比如参数传递规则（见12.2节）：

- 构造函数（constructor，见2.3.2节，16.2.5节）严格来说不是函数，它没有返回值，可以初始化基类和成员（见17.4节），我们无法得到其地址。
- 析构函数（destructor，见3.2.1.2节，17.2节）不能被重载，我们无法得到其地址。
- 函数对象（function object，见3.4.3节，19.2.2节）不是函数（它们是对象），不能被重载，但是其operator()是函数。
- lambda表达式（lambda expression，见3.4.3节，11.4节）是定义函数对象的一种简写形式。

12.1.4 返回值

每个函数声明都包含函数的返回类型（除了构造函数和类型转换函数之外）。传统上，在C和C++中，返回类型位于函数声明语句一开始的地方（位于函数名字之前）。然而，我们也可以在函数声明中把返回类型写在参数列表之后。例如，下面的两个声明是等价的：

```
string to_string(int a);          // 前置返回类型
auto to_string(int a) -> string;  // 后置返回类型
```

也就是说，前置的auto关键字表示函数的返回类型放在参数列表之后。后置返回类型由符号->引导。

后置返回类型的必要性源于函数模板声明，因为其返回类型是依赖于参数的。例如：

```
template<class T, class U>
auto product(const vector<T>& x, const vector<U>& y) -> decltype(x*y);
```

实际上，任何函数都可以使用后置返回类型。显然，后置返回类型的语法与lambda表达式的语法（见3.4.3节，11.4节）非常相似。这两个概念未能统一在一起，着实算是一种遗憾。

如果函数不返回任何值，则其“返回类型”是void。

如果函数没有被声明成void的，则它必须返回某个值（main()是个例外，见2.2.1节）。相反，void函数无权返回任何值。例如：

```
int f1() { }              // 错误：未返回任何值
void f2() { }             // OK

int f3() { return 1; }    // OK
void f4() { return 1; }   // 错误：试图在void函数中返回值

int f5() { return; }      // 错误：遗漏了返回值
void f6() { return; }     // OK
```

我们用return语句指定函数的返回值，例如：

```
int fac(int n)
{
```

```
    return (n>1) ? n*fac(n-1) : 1;
}
```

函数如果调用了它自身，我们称之为递归（recursive）。

函数可以包含多条 return 语句：

```
int fac2(int n)
{
    if (n > 1)
        return n*fac2(n-1);
    return 1;
}
```

与参数传递的语义类似，函数返回值的语义也与拷贝初始化（见 16.2.6 节）的语义一致。return 语句初始化一个返回类型的变量。编译器检查返回表达式的类型是否与函数的返回类型吻合，并在必要时执行标准的或者用户自定义的类型转换。例如：

```
double f() { return 1; }        // 1 隐式地转换成 double{1}
```

当我们每次调用函数的时候，重新创建它的实参以及局部（自动）变量的拷贝。一旦函数返回了结果，所占的存储空间就被重新分配了。因此，不应该返回指向局部非 static 变量的指针，我们无法预计该指针所指的位置的内容将发生什么样的改变：

```
int* fp()
{
    int local = 1;
    // ...
    return &local; // 糟糕的用法
}
```

当使用引用时也会发生类似的错误：

```
int& fr()
{
    int local = 1;
    // ...
    return local;   // 糟糕的用法
}
```

幸运的是，编译器能够很容易地发现试图返回局部变量引用的行为并给出警告（并且绝大多数情况下确实会这么做）。

并不存在 void 值。不过，我们可以调用 void 函数令其作为另一个 void 函数的返回值。例如：

```
void g(int* p);

void h(int* p)
{
    // ...
    return g(p);    // OK：等价于 "g(p); return;"
}
```

当编写模板函数且返回类型是模板参数时，这种返回形式有助于避免某些特殊情况。

在函数中，return 语句的形式属于下述 5 种之一：

- 执行一条 return 语句。
- “跳转到函数末尾”，也就是说，直接到达函数体的末端。这种情况只允许出现在无

返回值的函数（即，void 函数）以及 main() 中。此时，跳转到函数末尾意味着函数被成功地执行了（见 12.1.4 节）。

- 抛出一个未被局部捕获的异常（见 13.5 节）。
- 在一个 noexcept 函数中抛出了一个异常并且没有局部捕获，从而造成程序终止（见 13.5.1.1 节）。
- 直接或者间接地请求了一个无返回值的系统函数（比如 exit()，见 15.4 节）。

我们可以将一个未按常规方式（即，通过 return 或者“跳转到函数末尾”）返回的函数标记为 [[noreturn]]（见 12.1.7 节）。

12.1.5 inline 函数

函数可以被定义成 inline 的，例如：

```
inline int fac(int n)
{
    return (n<2) ? 1 : n*fac(n-1);
}
```

inline 限定符告诉编译器，它应该尝试为 fac() 的调用生成内联代码，而非先为 fac() 函数构建代码再通过常规的调用机制调用它。一个聪明的编译器能为 fac(6) 调用生成常量 720。内联函数面临许多复杂的情况，比如内联函数之间互相递归调用、单个内联的递归函数以及函数与输入无关，等等。因此，我们很难确保内联函数的每一次调用都真的是内联的。编译器的聪明程度无法做严格限定，因此有的编译器可能生成 720，另外的编译器可能生成 6*fac(5)，还有的编译器可能非内联地调用 fac(6)。如果你希望在编译时求值，最好把函数声明成 constexpr，并且确保求值过程中用到的所有函数都是 constexpr（见 12.1.6 节）。

我们必须注意一个事实，即，编译器和链接功能并非总是足够智能。因此，要想在智能化程度较低的编译环境中也能内联成功，我们应该令内联函数的定义（而非仅仅是声明）位于作用域内（见 15.2 节）。inline 限定符不会影响函数的语义。尤其是，内联函数仍旧拥有一个唯一的地址，内联函数中的 static 变量也是如此（见 12.1.8 节）。

如果内联函数的定义出现在多个编译单元中（通常是因为该内联函数被定义在头文件中，见 15.2.2 节），则这些定义必须保持一致（见 15.2.3 节）。

12.1.6 constexpr 函数

通常情况下，函数无法在编译时求值，因此也就不能在常量表达式中被调用（见 2.2.3 节，10.4 节）。但是通过将函数指定为 constexpr，我们就能向编译器传递这样的信息，即，如果给定了常量表达式作为实参，则我们希望该函数能被用在常量表达式中。例如：

```
constexpr int fac(int n)
{
    return (n>1) ? n*fac(n-1) : 1;
}

constexpr int f9 = fac(9);          // 必须在编译时求值
```

当 constexpr 出现在函数定义中时，它的含义是“如果给定了常量表达式作为实参，则该函数应该能用在常量表达式中”。而当 constexpr 出现在对象定义中时，它的含义是“在编译时对初始化器求值”。例如：

```
void f(int n)
{
    int f5 = fac(5);            // 可能在编译时求值
    int fn = fac(n);            // 在运行时求值（n 是变量）

    constexpr int f6 = fac(6);  // 必须在编译时求值
    constexpr int fnn = fac(n); // 错误：无法确保在编译时求值（n 是变量）

    char a[fac(4)];             // OK：数组的尺寸必须是常量，而 fac() 恰好是 constexpr
    char a2[fac(n)];            // 错误：数组的尺寸必须是常量，而 n 是一个变量

    // ...
}
```

函数必须足够简单才能在编译时求值：constexpr 函数必须包含一条独立的 return 语句，没有循环，也没有局部变量。同时，constexpr 函数不能有副作用。也就是说，constexpr 函数应该是一个纯函数。例如：

```
int glob;

constexpr void bad1(int a)      // 错误：constexpr 函数不能是 void
{
    glob = a;                   // 错误：在 constexpr 函数中有副作用
}

constexpr int bad2(int a)
{
    if (a>=0) return a; else return -a;    // 错误：在 constexpr 函数中有 if 语句
}

constexpr int bad3(int a)
{
    sum = 0;                                // 错误：在 constexpr 函数中有局部变量
    for (int i=0; i<a; ++i) sum +=fac(i);   // 错误：在 constexpr 函数中有循环
    return sum;
}
```

与普通的 constexpr 函数相比，constexpr 构造函数的规则有所区别（见 10.4.3 节）：只允许简单地执行成员初始化操作。

constexpr 函数允许递归和条件表达式。这意味着如果你确实希望某个函数是 constexpr，就一定能做到。随之而来的结果是，你必须严格遵循 constexpr 函数使用习惯，将其用于相对简单的任务。一旦有所违背，调试工作就会变得异常困难，并且编译的时间也会变得很长。

通过使用字面值类型（见 10.4.3 节），我们可以令 constexpr 函数适应用户自定义的类型。

和内联函数一样，constexpr 函数也遵循 ODR（一次定义法则）。因此，多个编译单元中的定义必须保持一致（见 15.2.3 节）。你可以把 constexpr 函数看成是形式更严格的内联函数（见 12.1.5 节）。

12.1.6.1　constexpr 与引用

constexpr 函数不允许有副作用，因此我们不能向非局部对象写入内容。反过来说，只要我们不向非局部对象写入内容，就能使用它。

```
constexpr int ftbl[] { 1, 2, 3, 5, 8, 13 };

constexpr int fib(int n)
```

```
{
    return (n<sizeof(ftbl)/sizeof(*ftbl)) ? ftbl[n] : fib(n);
}
```

constexpr 函数可以接受引用实参。尽管它不能通过这些引用写入内容，但是 const 引用参数同样有用。例如，我们在标准库（见 40.4 节）中发现：

```
template<> class complex<float> {
public:
// ...
    explicit constexpr complex(const complex<double>&);
    // ...
};
```

这允许我们编写：

```
constexpr complex<float> z {2.0};
```

其中，逻辑上用于存储 const 引用实参的临时变量成了编译器内部可用的一个值。

constexpr 函数可以返回一个引用或者指针，例如：

```
constexpr const int* addr(const int& r) { return &r; }    // OK
```

然而，这种做法违背了 constexpr 函数作为常量表达式求值要件的初衷。要想确定此类函数的结果是否是常量表达式就成了一个非常微妙的任务。请考虑下面的情况：

```
static const int x = 5;
constexpr const int* p1 = addr(x);        // OK
constexpr int xx = *p1;                   // OK

static int y;
constexpr const int* p2 = addr(y);        // OK
constexpr int yy = *y;                    // 错误：试图读取一个变量的值

constexpr const int* tp = addr(5);        // 错误：临时变量 y 的地址
```

12.1.6.2 条件求值

constexpr 函数之外的条件表达式不会在编译时求值，这意味着它可以请求运行时求值。例如：

```
constexpr int check(int i)
{
    return (low<=i && i<high) ? i : throw out_of_range();
}

constexpr int low = 0;
constexpr int  high = 99;

// ...
constexpr int val = check(f(x,y,z));
```

其中，我们假定 low 和 high 是设计时未知而编译时已知的配置参数。此时，f(x,y,z) 计算的是依赖于实现的值。

12.1.7 [[noreturn]] 函数

形如 [[...]] 的概念被称为属性（attribute），属性可以置于 C++ 语法的任何位置。通常情况下，属性描述了位于它前面的语法实体的性质，这些性质依赖于实现。此外，属性也能出

现在声明语句开始的位置。C++ 只包含两个标准属性（§ iso.7.6），[[noreturn]] 就是其中之一，另一个是 [[carries_dependency]]（见 41.3 节）。

把 [[noreturn]] 放在函数声明语句的开始位置表示我们不希望该函数返回任何结果。例如：

```
[[noreturn]] void exit(int);      // exit 永远不会返回任何结果
```

如果我们预先知道某个函数不会返回结果，对于理解程序和生成代码都很有益处。如果函数被设定为 [[noreturn]]，但是在该函数的内部仍然返回了某个值，将产生未定义的行为。

12.1.8　局部变量

定义在函数内部的名字通常称为*局部名字*（local name）。当线程执行到局部变量或常量的定义处时，它们将被初始化。除非我们把变量声明成 static，否则函数的每次调用都会拥有该变量的一份拷贝。相反，如果我们把局部变量声明成 static，则在函数的所有调用中都将使用唯一的一份静态分配的对象（见 6.4.2 节）。该对象在线程第一次到达它的定义处时被初始化。例如：

```
void f(int a)
{
    while (a--) {
        static int n = 0;       // 只初始化一次
        int x = 0;              // 每次调用 f() 都会初始化

        cout << "n == " << n++ << ", x == " << x++ << '\n';
    }
}

int main()
{
    f(3);
}
```

上述代码的输出结果是：

```
n == 0, x == 0
n == 1, x == 0
n == 2, x == 0
```

static 局部变量有一个非常重要的作用，即，它可以在函数的多次调用间维护一份公共信息而无须使用全局变量。如果使用了全局变量，则有可能会被其他不相关的函数访问甚至干扰（见 16.2.12 节）。

除非你进入了一个递归调用的函数或者发生了死锁（§ iso.6.7），通常情况下，static 局部变量的初始化不会导致数据竞争（见 5.3.1 节）。也就是说，C++ 实现必须用某种无锁机制（比如 call_once，见 42.3.3 节）确保局部 static 变量的初始化能被正确执行。递归地初始化一个局部 static 变量将产生未定义的结果。例如：

```
int fn(int n)
{
    static int n1 = n;              // OK
    static int n2 = fn(n-1)+1;      // 未定义的
    return n;
}
```

static 局部变量有助于避免非局部变量间的顺序依赖（见 15.4.1 节）。

不存在“局部函数”；如果你觉得自己需要一个类似的东西，不妨使用函数对象或者 lambda 表达式（见 3.4.3 节，11.4 节）。

如前所述，不要轻易使用标号（见 9.6 节）。一旦你不小心用到了，则无论该标号位于怎样的嵌套结构中，它的作用域都是整个函数。

12.2 参数传递

当程序调用一个函数时（使用后缀 ()，称为调用运算符 call operator 或者应用运算符 application operator），我们为该函数的形参（formal arguments，即，parameters）申请内存空间，并用实参（actual argument）初始化对应的形参。参数传递的语义与初始化的语义一致（严格地说是拷贝初始化，见 16.2.6 节）。编译器负责检查实参的类型是否与对应的形参类型吻合，并在必要的时候执行标准类型转换或者用户自定义的类型转换。除非形参是引用，其他情况下传入函数的是实参的副本。例如：

```
int* find(int* first, int* last, int v)    // 在 [first:last) 内查找等于 v
{
    while (first!=last && *first!=v)
        ++first;
    return first;
}

void g(int* p, int* q)
{
    int* pp = find(p,q,'x');
    // ...
}
```

此时，主调函数提供的实参 p 不会被 find() 函数修改，发生变化的是 find() 函数中与 p 对应的副本 first。该指针是以传值方式传入函数的。

传递数组有特殊的规则（见 12.2.2 节），C++ 还提供了专门的机制来传递未检测的参数（见 12.2.4 节）以及指定默认参数（见 12.2.5 节）。初始化器列表的用法将在 12.2.3 节介绍，23.5.2 节和 28.6.2 节将介绍向模板函数传递实参的方法。

12.2.1 引用参数

考虑如下情况：

```
void f(int val, int& ref)
{
    ++val;
    ++ref;
}
```

当调用 f() 时，++val 递增第一个实参在当前函数内的副本，而 ++ref 递增第二个实参本身。再举另外一个例子：

```
void g()
{
    int i = 1;
    int j = 1;
    f(i,j);
}
```

调用 f(i,j) 递增 j 的值，但是不会递增 i 的值。第一个实参 i 是以传值的方式传入函数的，第二个实参 j 是以传引用的方式传入函数的。如第 7.7 节所述，程序员应该尽量避免修改引用类型的实参（但请参见 18.2.5 节），因为这么做会困扰程序的读者。但是同时请注意，当遇到大对象时，引用传递比值传递更有效。此时，我们应该将该引用类型的参数声明成 const 的，以表明我们之所以使用引用只是出于效率上的考虑，而并非想让函数修改对象的值：

```
void f(const Large& arg)
{
    // 不允许修改 "arg" 的值
    // (除非显式使用类型转换；见 11.5 节)
}
```

如果在函数的声明中有某个引用参数未被指定为 const，则我们倾向于认为函数将修改该参数的值：

```
void g(Large& arg);      // 倾向于认为 g() 会修改 arg 的值
```

类似地，指针类型的参数被声明成 const 意味着该指针所指对象的值不会被函数改变。例如：

```
int strlen(const char*);                          // C 风格字符串中的字符数量
char* strcpy(char* to, const char* from);         // 复制一个 C 风格的字符串
int strcmp(const char*, const char*);             // 比较两个 C 风格的字符串
```

程序规模越大，合理使用 const 参数就显得越重要。

请注意，参数传递的语义与赋值的语义是不同的。这一点对于 const 参数、引用参数以及某些用户自定义类型的参数尤为重要。

与之前介绍的引用初始化规则相同，字面值、常量以及需要执行类型转换的参数可以被传给 const T& 参数，但是不能传给普通的非 const T& 参数。一方面，我们允许向 const T& 的转换以确保凡是能传给 T 类型参数的值都能传给 const T& 类型的参数，实现方式是把值存入临时变量。例如：

```
float fsqrt(const float&); // Fortran 风格的 sqrt 接受一个引用参数

void g(double d)
{
    float r = fsqrt(2.0f);      // 传递存放 2.0f 的临时变量的引用
    r = fsqrt(r);               // 传递 r 的引用
    r = fsqrt(d);               // 传递存放 static_cast<float>(d) 的临时变量的引用
}
```

另一方面，禁止向非 const 引用参数转换可以有效地规避临时变量可能带来的程序错误风险。例如：

```
void update(float& i);

void g(double d, float r)
{
    update(2.0f);       // 错误：常量实参
    update(r);          // 传递 r 的引用
    update(d);          // 错误：需要执行类型转换
}
```

假如这些调用都是合法的，则 update() 将会更新那些即将被释放的临时值，而且这一切操作都会在悄无声息间发生。通常情况下，程序员并不愿意看到这种情况。

其实，引用传递的准确描述应该是左值引用传递，原因是函数不能接受一个右值引用作为它的参数。如 7.7 节所述，右值能绑定到右值引用上（但不能绑定到左值引用上），左值能绑定到左值引用上（但不能绑定到右值引用上）。例如：

```
void f(vector<int>&);                  // 非 const 左值引用参数
void f(const vector<int>&);            // const 左值引用参数
void f(vector<int>&&);                 // 右值引用参数

void g(vector<int>& vi, const vector<int>& cvi)
{
    f(vi);                             // 调用 f(vector<int>&)
    f(cvi);                            // 调用 f(const vector<int>&)
    f(vector<int>{1,2,3,4});           // 调用 f(vector<int>&&);
}
```

我们必须假定函数可能会修改右值参数，除非将它用于析构函数或者重新赋值（见 17.5 节）。对于右值引用来说，最常见的用处是定义移动构造函数或者移动赋值运算（见 3.3.2 节，17.5.2 节）。相信未来的某天，有人能为 const 右值引用参数找到合适的用武之地；但是仅就目前的情况来看，我还没有发现这样的应用场景。

请注意，对于模板参数 T 来说，模板参数类型推断准则令 T&& 的含义与 X&& 之于 X 的含义有很大的区别（见 23.5.2.1 节）。右值引用模板参数常用于实现“完美转发”（见 23.5.2.1 节，28.6.3 节）。

该如何选择参数的传递方式呢？我的经验准则是：

[1] 对于小对象使用值传递的方式。

[2] 对于你无须修改的大对象使用 const 引用传递。

[3] 如果需要返回计算结果，最好使用 return 而非通过参数修改对象。

[4] 使用右值引用实现移动（见 3.3.2 节，17.5.2 节）和转发（见 23.5.2.1 节）。

[5] 如果找不到合适的对象，则传递指针（用 nullptr 表示“没有对象”）。

[6] 除非万不得已，否则不要使用引用传递。

在最后一条经验准则中，“除非万不得已”是基于这样一种观察，即，当我们需要修改对象的值时，传递指针比使用引用更容易表达清楚程序的原意。

12.2.2 数组参数

当数组作为函数的参数时，实际传入的是指向该数组首元素的指针。例如：

```
int strlen(const char*);

void f()
{
    char v[] = "Annemarie";
    int i = strlen(v);
    int j = strlen("Nicholas");
}
```

也就是说，当作为参数被传入函数时，类型 T[] 会被转换成 T*。这意味着如果我们对数组参数的元素赋值，则会改变该数组元素的实际值。换言之，与其他类型不同，数组不是以值传递的方式传入的；相反，传入的是指针（以值传递的方式）。

数组类型的参数与指针类型的参数等价，例如：

```
void odd(int* p);
void odd(int a[]);
void odd(int buf[1020]);
```

这三个声明语句是等价的，它们声明的是同一个函数。通常情况下，参数的名字对函数的类型没有影响（见 12.1.3 节）。7.4.3 节介绍了传递多维数组的规则与技术。

对于被调函数来说，数组的尺寸是不可见的。这是很多错误的根源，但是我们也有一些方法可以避免此类问题。C 风格的字符串以 0 作为结尾，因此我们可以计算其长度（例如，可以调用 strlen()，尽管代价高昂，见 43.4 节）。对于其他数组来说，我们可以再传入一个参数，用它来表示数组的大小。例如：

```
void compute1(int* vec_ptr, int vec_size);    // 一种可行的办法
```

不过这么做顶多算一种变通的方法。更好的做法是传入某些容器（比如 vector，见 4.4.1 节和 31.4 节；array，见 34.2.1 节；或者 map，见 4.4.3 节和 31.4.3 节）的引用。

如果你确实想传入一个数组而非容器或者指向数组首元素的指针，你可以把参数的类型声明成数组的引用。例如：

```
void f(int(&r)[4]);

void g()
{
    int a1[] = {1,2,3,4};
    int a2[] = {1,2};

    f(a1);      // OK
    f(a2);      // 错误：元素个数有误
}
```

对于数组引用类型的参数来说，元素个数也是其类型的一部分。因此，数组引用的灵活性远不如指针或者容器（比如 vector）。我们通常在模板中才会使用数组引用，此时元素的个数可以通过推断得到。例如：

```
template<class T, int N> void f(T(&r)[N])
{
    // ...
}

int a1[10];
double a2[100];

void g()
{
    f(a1);          // T 是 int; N 是 10
    f(a2);          // T 是 double; N 是 100
}
```

这么做的后果是调用 f() 所用的数组类型有多少个，对应的函数定义就有多少个。

多维数组的情况比较复杂（见 7.3 节），我们一般用指针的数组替代，此时无须特殊处理。例如：

```
const char* day[] = {
    "mon", "tue", "wed", "thu", "fri", "sat", "sun"
};
```

一如既往，vector 及类似类型比内置的底层数组和指针更好。

12.2.3 列表参数

一个由 {} 限定的列表可以作为下述形参的实参：

[1] 类型 std::initializer_list<T>，其中列表的值能隐式地转换成 T

[2] 能用列表中的值初始化的类型

[3] T 类型数组的引用，其中列表的值能隐式地转换成 T

从技术上来看，情况[2]可以涵盖所有情况。但是分成三种情况更易于程序员理解，例如：

```
template<class T>
void f1(initializer_list<T>);

struct S {
    int a;
    string s;
};
void f2(S);

template<class T, int N>
void f3(T (&r)[N]);

void f4(int);

void g()
{
    f1({1,2,3,4});    // T 是 int，initializer_list 的大小是 4
    f2({1,"MKS"});    // f2(S{1,"MKS"})
    f3({1,2,3,4});    // T 是 int，N 是 4
    f4({1});          // f4(int{1});
}
```

如果存在二义性，则 initializer_list 参数的函数被优先考虑。例如：

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

struct S {
    int a;
    string s;
};
void f(S);

template<class T, int N>
void f(T (&r)[N]);

void f(int);

void g()
{
    f({1,2,3,4});     // T 是 int，initializer_list 的大小是 4
    f({1,"MKS"});     // calls f(S)
    f({1});           // T 是 int，initializer_list 的大小是 1
}
```

之所以优先选择具有 initializer_list 参数的函数，是因为如果根据列表的元素数量选择函数

的话，会让选择的过程显得非常混乱。在重载解析的时候，很难把所有可能引起混淆的函数形式都排除干净（见 4.4 节，17.4.3.1 节），但是当遇到 {} 列表的参数时给 initializer_list 参数最高的优先级能最大限度地避免混淆。

如果某个具有 initializer_list 参数的函数位于作用域内，但是实际提供的参数列表与之并不匹配，则会选择另外的函数。在上例中，对 f({1,"MKS"}) 的调用属于这种情况。

请注意，上述准则只适用于 std::initializer_list<T> 参数。对于 std::initializer_list<T>& 或者其他碰巧也叫 initializer_list 的类型（在其他作用域中）来说，C++ 并没有制定特殊的规则。

12.2.4　数量未定的参数

对于某些函数来说，很难明确指定调用时期望的参数数量和类型。要实现这样的接口，我们有三种选择：

[1] 使用可变模板（见 28.6 节）：它允许我们以类型安全的方式处理任意类型、任意数量的参数，只要写一个小的模板元程序来解释参数列表的正确含义并采取适当的操作就可以了。

[2] 使用 initializer_list（见 12.2.3 节）作为参数类型。它允许我们以类型安全的方式处理某种类型的、任意数量的参数。在大多数上下文中，这种元素类型相同的参数列表是最常见和最重要的情形。

[3] 用省略号（...）结束参数列表，表示“可能有更多参数”。它允许我们通过使用 <cstdarg> 中的宏处理（几乎）任意类型的、任意数量的参数。这种方案并非类型安全的，并且很难用于复杂的用户自定义类型。但是，从早期的 C 语言开始人们就使用这种机制了。

前两种机制将在其他章节中介绍，因此在这里我只为读者描述第三种机制（尽管在大多数情况下，它相对于前两种机制来说是次优选择）。例如：

```
int printf(const char* ...);
```

这条语句规定对标准库函数 printf()（见 43.3 节）的调用必须至少有一个 C 风格字符串的参数，同时可以有也可以没有其他参数。例如：

```
printf("Hello, world!\n");
printf("My name is %s %s\n", first_name, second_name);
printf("%d + %d = %d\n",2,3,5);
```

这样的函数在解释其参数列表时必须依赖于某些编译器不可知的信息。以 printf() 为例，它的第一个参数是一条包含特殊字符序列的格式化字符串，该字符串为 printf() 正确处理其他参数提供了保障。%s 表示“期望一个 char* 参数”，%d 表示“期望一个 int 参数”。然而通常情况下，编译器并不能保证在某次调用中期望的参数一定出现，也不能保证出现的参数一定是期望的类型。例如：

```
#include <cstdio>

int main()
{
    std::printf("My name is %s %s\n",2);
}
```

这是一段错误的代码，但是大多数编译器都无法发现此类错误。读者不妨试试这段代码，它没什么实际作用，顶多会产生一些奇奇怪怪的输出。

显然，如果某个参数并没有被声明，编译器也就不知道该怎么对它执行标准类型检查和类型转换。此时，char 和 short 被当成 int 传递，float 被当成 double 传递。程序员并不一定乐于见到这种情况。

在一个设计良好的程序中，不应该出现太多参数类型不确定的函数。当程序员想使用未限定类型的参数时，其实应该优先考虑使用重载函数、带默认参数的函数、接受 initializer_list 参数的函数或者可变参数模板。它们不但可以覆盖大多数情况而且能较好地执行类型检查。只有当参数数量和参数类型都不确定且可变模板也不适用时，才考虑使用省略号参数。

省略号最常见的用法是为 C 语言库函数提供接口：

```
int fprintf(FILE*, const char* ...);        // 来自于 <cstdio>
int execl(const char* ...);                 // 来自于 UNIX 头文件
```

<cstdarg> 中包含了一组标准宏，它们可用于在此类函数中访问未限定的函数。考虑编写一个报错函数，它接受一个整数参数表示错误的级别，后面是任意数量的字符串参数。在这个函数中，每个 C 风格的字符串参数用来传递一个单词，所有这些单词组成完整的错误信息。字符串参数列表以空指针结束：

```
extern void error(int ...);
extern char* itoa(int, char[]);  // int 转换为字符串

int main(int argc, char* argv[])
{
    switch (argc) {
    case 1:
        error(0,argv[0],nullptr);
        break;
    case 2:
        error(0,argv[0],argv[1],nullptr);
        break;
    default:
        char buffer[8];
        error(1,argv[0],"with",itoa(argc-1,buffer),"arguments",nullptr);
    }
    // ...
}
```

函数 itoa() 返回一个 C 风格字符串，是其 int 参数的字符串表示。这种用法在 C 语言中很流行，但是并不属于 C 标准的一部分。

我之所以总是传递 argv[0] 是因为按照惯例它代表程序的名字。

请注意，把零数 0 作为终止符的做法是不可移植的：在有的实现中，整数 0 和空指针的表示形式并不一致（见 6.2.8 节）。这从一个侧面证明了如果用省略号抑制了类型检查，则程序员将不得不额外面对很多微妙的工作。

函数 error() 的定义如下所示：

```
#include <cstdarg>

void error(int severity ...) // "severity" 之后紧跟一个以 0 结尾的字符串列表
{
    va_list ap;
```

```
    va_start(ap,severity);      // arg 启动

    for (;;) {
        char* p = va_arg(ap,char*);
        if (p == nullptr) break;
        cerr << p << ' ';
    }

    va_end(ap);                 // arg 结束

    cerr << '\n';
    if (severity) exit(severity);
}
```

首先，定义 va_list 并调用 va_start() 初始化它。宏命令 va_start 接受 va_list 的名字和最后一个正式参数的名字作为它的参数。宏命令 va_arg() 用于按顺序提取未命名的参数。每次调用它时，程序员必须提供一个类型；va_arg() 假定该类型的一个实参被传入了函数，但是通常它无法确保这一点。如果在函数中使用了 va_start()，则在该函数返回前必须先调用 va_end()。这么做的原因是 va_start() 可能会修改栈的内容，从而造成函数无法正常返回；但是 va_end() 可以撤销所有此类修改。

函数 error() 也可以用标准库 initializer_list 定义成如下形式：

```
void error(int severity, initializer_list<string> err)
{
    for (auto& s : err)
        cerr << s << ' ';
    cerr << '\n';
    if (severity) exit(severity);
}
```

要想调用它，必须使用列表记法。例如：

```
switch (argc) {
case 1:
    error(0,{argv[0]});
    break;
case 2:
    error(0,{argv[0],argv[1]});
    break;
default:
    error(1,{argv[0],"with",to_string(argc-1),"arguments"});
}
```

标准库负责提供 int 向 string 的转换函数 to_string()（见 36.3.5 节）。

如果不拘泥于 C 语言的风格，我们可以给函数传入一个容器，这可以进一步简化代码：

```
void error(int severity, const vector<string>& err)  // 与之前几乎一样
{
    for (auto& s : err)
        cerr << s << ' ';
    cerr << '\n';
    if (severity) exit(severity);
}

vector<string> arguments(int argc, char* argv[]) // 把参数打包在一起
{
```

```
    vector<string> res;
    for (int i = 0; i!=argc; ++i)
        res.push_back(argv[i]);
    return res
}

int main(int argc, char* argv[])
{
    auto args = arguments(argc,argv);
    error((args.size()<2)?0:1,args);
    // ...
}
```

辅助函数 arguments() 并不复杂，同时 main() 和 error() 也很简单。main() 和 error() 之间的接口传递了所有的参数，因此通用性更强且使得我们可以进一步改进 error()。此外，与未指定数量的参数相比，使用 vector<string> 出错的可能性大大降低了。

12.2.5 默认参数

一个通用的函数所需要的参数通常比处理简单情况所需的参数要多。举个例子，为了增加灵活性，用于构造对象的函数（见 16.2.5 节）往往会提供几种不同的选项。考虑 3.2.1.1 节的 complex 类：

```
class complex {
    double re, im;
public:
    complex(double r, double i) :re{r}, im{i} {}      // 用两个标量构造 complex
    complex(double r) :re{r}, im{0} {}                // 用一个标量构造 complex
    complex() :re{0}, im{0} {}
// 默认为 complex: {0,0}
    // ...
};
```

complex 的构造函数的行为没什么特殊之处，但从逻辑上来说三个函数（这里是构造函数）完成几乎一样的工作看起来总是怪怪的。同时对于很多类来说，构造函数要做的事情更多，而且几乎是重复的。我们处理这种重复性的策略是认为其中一个构造函数是“真正的那个”，然后在别的构造函数中使用它（见 17.4.3 节）：

```
complex(double r, double i) :re{r}, im{i} {}      // 用两个标量构造 complex
complex(double r) :complex{r,0} {}                // 用一个标量构造 complex
complex() :complex{0,0} {}                        // 默认的 complex: {0,0}
```

现在，如果我们想在 complex 中加入一些调试、跟踪和统计的代码，只要加在一个地方就可以了。上述代码还可以进一步简化：

```
complex(double r ={}, double i ={}) :re{r}, im{i} {}   // 用两个标量构造 complex
```

这行代码的含义很清晰：如果用户提供的参数数量不足，则使用预置的默认参数。它很好地体现了程序员的意愿，即，用一个构造函数加上一些速记符号来涵盖所有情况。

默认参数在函数声明时执行类型检查，在调用函数时求值。例如：

```
class X {
public:
    static int def_arg;
    void f(int =def_arg);
```

```
    // ...
};

int X::def_arg = 7;

void g(X& a)
{
    a.f();          // maybe f(7)
    a.def_arg = 9;
    a.f();          // f(9)
}
```

最好避免使用值可能发生改变的默认参数，因为这么做会引入对上下文的微妙依赖。

我们只能给参数列表中位置靠后的参数提供默认值，例如：

```
int f(int, int =0, char* =nullptr);// OK
int g(int =0, int =0, char*);          // 错误
int h(int =0, int, char* =nullptr);    // 错误
```

其中，* 和 = 之间的空格必不可少（*= 是赋值运算符，见 10.3 节）：

```
int nasty(char*=nullptr);              // 语法错误
```

在同一个作用域的一系列声明语句中，默认参数不能重复或者改变：

```
void f(int x = 7);
void f(int = 7);            // 错误：不允许重复默认参数
void f(int = 8);            // 错误：默认参数不一致
void g()
{
    void f(int x = 9);      // OK: 这个声明覆盖了外层的
    // ...
}
```

在上面的代码中，嵌套作用域内部的名字隐藏了外层作用域中的相同名字，这种用法可能造成程序错误。

12.3　重载函数

大多数情况下我们应该给不同的函数起不一样的名字。但如果不同函数是在不同类型的对象上执行相同概念的任务，则给它们起同一个名字是更好的选择。为不同数据类型的同一种操作起相同的名字称为重载（overloading）。C++ 的基本操作已经采用了这种技术：加法只有一个名字 +，但是它既可以执行整数值的加法，也可以执行浮点数的加法，还能执行这些类型彼此之间的加法。这一思想很容易就能扩展到程序员定义的函数中。例如：

```
void print(int);            // 打印 int
void print(const char*);    // 打印 C 风格字符串
```

对于编译器来说，同名函数唯一的共同点就是名字相同。基本上我们可以认为这些函数是相似的，但是语言本身在这一点上既不会限制程序员，也不会提供什么帮助。因此，重载函数的名字主要是提供了一种便利的表示方法。对于具有约定俗成的名字的函数来说，比如 sqrt、print 或者 open，重载函数的便利性体现得比较明显。如果一个名字具有明显的语义，则它的便利性就显得很重要了。在构造函数（见 16.2.5 节，17.1 节）以及泛型编程（见 4.5 节，第 32 章）中经常会用到重载函数，比如常见的有运算符 +、* 和 << 等。

模板为定义成组的重载函数提供了一种系统的方法（见 23.5 节）。

12.3.1 自动重载解析

当调用函数 fct 时，由编译器决定使用名为 fct 的那组函数中的哪一个。依据是考察实参类型与作用域中名为 fct 的哪个函数的形参类型最匹配。当找到了最佳匹配时，调用该函数；反之，引发编译器错误。例如：

```
void print(double);
void print(long);

void f()
{
    print(1L);      // print(long)
    print(1.0);     // print(double)
    print(1);       // 错误，具有二义性，该选择 print(long(1)) 还是 print(double(1))?
}
```

为了更合理地解决这一问题，我们按如下顺序尝试一系列评判准则：

[1] 精确匹配：也就是说，无须类型转换或者仅需简单的类型转换（例如数组名转换为指针，函数名转换为函数指针，以及 T 转换为 const T）即可实现匹配

[2] 执行提升后匹配：也就是说，执行了整数提升（bool 转换为 int，char 转换为 int，short 转换为 int，上述转换的 unsigned 版本，见 10.5.1 节）以及 float 转换为 double

[3] 执行标准类型转换后实现匹配（比如 int 转换为 double，double 转换为 int，double 转换为 long double，Derived* 转换为 Base*（见 20.2 节），T* 转换为 void*（见 7.2.1 节），以及 int 转换为 unsigned int（见 10.5 节））

[4] 执行用户自定义的类型转换后实现匹配（比如 double 转换为 complex<double>，见 18.4 节）

[5] 使用函数声明中的省略号 ... 进行匹配（见 12.2.4 节）

在该体系中，如果某次函数调用在能找到匹配的最高层级上发现了不止一个可用匹配，则本次调用将因产生了二义性而被拒绝。这些复杂的解析规则主要是考虑到 C 和 C++ 的内置数值类型规则而制定的（见 10.5 节）。例如：

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);

void h(char c, int i, short s, float f)
{
    print(c);       // 精确匹配：调用 print(char)
    print(i);       // 精确匹配：调用 print(int)
    print(s);       // 整型提升：调用 print(int)
    print(f);       // float 转换为 double 的提升：print(double)

    print('a');     // 精确匹配：调用 print(char)
    print(49);      // 精确匹配：调用 print(int)
    print(0);       // 精确匹配：调用 print(int)
    print("a");     // 精确匹配：调用 print(const char*)
    print(nullptr); // nullptr_t 转换为 const char* 的提升：调用 print(const char*)
}
```

因为 0 是 int，所以 print(0) 调用的是 print(int)；因为 'a' 是 char，所以 print('a') 调用的是

print(char)（见 6.2.3.2 节）。我们之所以要把转换和提升区分开来，是因为相对于不安全的类型转换（如 int 转换为 char），我们更倾向于使用安全的类型提升（如 char 转换为 int）。相关内容请参见 12.3.5 节。

重载解析与函数声明的次序无关。

处理函数模板时，我们根据模板参数集将重载解析规则应用于特例化的结果之上（见 23.5.3 节）。对于参数是 {} 列表（初始化器列表具有较高的优先级，见 12.2.3 节，17.3.4.1 节）或者是右值引用模板参数的情况，C++ 制定了专门的重载解析规则（见 23.5.2.1 节）。

重载依赖于一套比较复杂的规则体系，如果程序员使用不慎的话，有可能造成意料之外的结果。既然这样，为什么我们还要自寻烦恼呢？为了回答这个问题，不妨考虑如下的解决方案。既然我们的目标是对不同的数据类型执行类似的操作，那么我们可以把这些函数定义成不同的名字：

```
void print_int(int);
void print_char(char);
void print_string(const char*);      // C 风格字符串

void g(int i, char c, const char* p, double d)
{
    print_int(i);          // OK
    print_char(c);         // OK
    print_string(p);       // OK

    print_int(c);          // OK? 调用 print_int(int(c))，输出一个数字
    print_char(i);         // OK? 调用 print_char(char(i))，执行窄化运算
    print_string(i);       // 错误
    print_int(d);          // OK? 调用 print_int(int(d))，执行窄化运算
}
```

与重载版本的 print() 相比，此时，我们不得不记住好几个不同的名字，而且还得时刻警惕对某些数据类型不要用错了函数。显然这么做费时费力，没什么好处；既不利于泛型编程（见 4.5 节），又容易让程序员受困于比较低层的类型问题不得脱身。此外，由于没有重载，所以这些函数的参数可能执行各种各样的标准类型转换，极易发生错误。在上面的例子中，存在四个语义模糊的函数调用，但是编译器只能发现其中一个；而在剩下的三个中有两个都执行了存在错误风险的窄化运算（见 2.2.2 节，10.5 节）。可见，使用重载技术可以在一定程度上增加编译器发现并拒绝不适当参数的机会。

12.3.2　重载与返回类型

在重载解析过程中不考虑函数的返回类型，这样可以确保对运算符（见 18.2.1 节，18.2.5 节）或者函数调用的解析独立于上下文。例如：

```
float sqrt(float);
double sqrt(double);

void f(double da, float fla)
{
    float fl = sqrt(da);   // 调用 sqrt(double)
    double d = sqrt(da);   // 调用 sqrt(double)
    fl = sqrt(fla);        // 调用 sqrt(float)
    d = sqrt(fla);         // 调用 sqrt(float)
}
```

如果把返回类型也考虑在内的话，对 sqrt() 的重载解析就必须依赖于上下文而无法独立进行了。

12.3.3 重载与作用域

重载发生在一组重载函数集的成员内部，也就是说，重载函数应该位于同一个作用域内。不同的非名字空间作用域中的函数不会重载。例如：

```
void f(int);

void g()
{
    void f(double);
    f(1);           // 调用 f(double)
}
```

对于 f(1) 来说，f(int) 显然是最佳匹配，但是当前只有 f(double) 处于作用域中。在此类情况中，程序员可以通过添加或者删除局部声明来获得想要的结果。一般来说，名字隐藏如果是程序员精心设计的，则可能会有奇效；但如果程序员本没有这样的意图，则会出现让人意料不到的结果。

基类和派生类提供的作用域不同，因此默认情况下基类函数和派生类函数不会发生重载。例如：

```
struct Base {
    void f(int);
};

struct Derived : Base {
    void f(double);
};

void g(Derived& d)
{
    d.f(1);     // 调用 Derived::f(double);
}
```

如果我们希望实现跨类作用域（见 20.3.5 节）或者名字空间作用域（见 14.4.5 节）的重载，应该使用 using 声明或者 using 指示（见 14.2.2 节）。依赖于参数的查找（见 14.2.4 节）也会导致跨名字空间的重载。

12.3.4 多实参解析

对于一组重载函数以及一次调用来说，如果该调用对于各函数的参数类型在计算的效率和精度上差别明显，则我们可以应用重载解析规则从中选出最合适的那个函数。例如：

```
int pow(int, int);
double pow(double, double);
complex pow(double, complex);
complex pow(complex, int);
complex pow(complex, complex);

void k(complex z)
{
    int i = pow(2,2);           // 调用 pow(int,int)
```

```
    double d = pow(2.0,2.0);        // 调用 pow(double,double)
    complex z2 = pow(2,z);          // 调用 pow(double,complex)
    complex z3 = pow(z,2);          // 调用 pow(complex,int)
    complex z4 = pow(z,z);          // 调用 pow(complex,complex)
}
```

当重载函数包含两个或者多个参数时，12.3 节提到的解析规则将作用于每一个参数，并且选出该参数的最佳匹配函数。如果某个函数是其中一个参数的最佳匹配，同时在其他参数上也是更优的匹配或者至少不弱于别的函数，则该函数就是最终确定的最佳匹配函数。如果找不到符合上述条件的函数，则本次调用将因二义性的原因被拒绝。例如：

```
void g()
{
    double d = pow(2.0,2);    // 错误：该调用 pow(int(2.0),2) 还是 pow(2.0,double(2))?
}
```

对于该调用来说，2.0 的最佳匹配函数是 pow(double,double)，2 的最佳匹配函数是 pow(int,int)，因此存在二义性，是一次错误的调用。

12.3.5　手动重载解析

某个函数的重载版本过少或者过多都可能导致二义性，例如：

```
void f1(char);
void f1(long);

void f2(char*);
void f2(int*);

void k(int i)
{
    f1(i);      // 二义性：f1(char) 还是 f1(long)?
    f2(0);      // 二义性：f2(char*) 还是 f2(int*)?
}
```

在可能的情况下，程序员应该尽量把一组重载函数当成整体来看，考察其对于函数的语义来说是否有意义。很多时候我们通过增加一个函数版本来解决二义性的问题。例如，增加

```
inline void f1(int n) { f1(long(n)); }
```

会把所有类似 f1(i) 的二义性调用都解析成接受更大类型 long int 的版本。

程序员还可以利用显式类型转换解析某个特定的调用，例如：

```
f2(static_cast<int*>(0));
```

不过这更像是一种丑陋的临时解决方案。很快就会有另一个类似的二义性调用，我们还需要再去解决。

不同人对于编译器报出的二义性错误态度截然不同。C++ 初学者常常很不服气，抱怨这类检查多此一举；而经验丰富的程序员更愿意接受此类信息，因为他们明白这样的提示有助于发现设计程序时存在的错误。

12.4　前置与后置条件

每个函数都对它的参数或多或少有一些预期。有的预期表达为参数的类型，另外一些预期则依赖于实际传入的参数值以及这些值之间的关系。编译器和链接器能确保参数类型的正

确性，但是当需要决定如何处置“不好”的参数值时，就得依靠程序员了。我们把函数调用时应该遵循的约定称为前置条件（precondition），把函数返回时应该遵循的约定称为后置条件（postcondition）。例如：

```
int area(int len, int wid)
/*
    计算长方形的面积

    前置条件：长方形的长和宽都是正数

    后置条件：返回值是正数

    后置条件：返回值是长方形的面积，其中长方形的长和宽分别是 len 和 wid
*/
{
    return len*wid;
}
```

其中，关于前置条件和后置条件的叙述比函数体本身长得多。这么做看起来有点啰嗦，但是这些信息实际上对程序的实现者、area() 的使用者以及测试人员来说都很有用。例如，上述约束告诉我们 0 和 –12 是无效的参数。我们可以在不违背前置条件的情况下传入一对特别大的参数，但是它们相乘的结果 len*wid 也必须在两个后置条件规定的范围之内。

对于可能出现的调用 area(numeric_limits<int>::max(),2)，我们该做些什么呢？

[1] 应该由函数的调用者负责确保这样的调用不会发生吗？是的，但是如果调用者做不到该怎么办？

[2] 应该由函数的实现者负责确保这样的调用不会发生吗？如果是这样的话，该如何处理此类错误？

这些问题的答案可能不止一个。函数的调用者很可能会犯错，违反前置条件；而函数的实现者也很难以极低的代价快速完整地检查前置条件是否满足。我们期望的做法是，由调用者负责确保遵守前置条件，但同时也提供一种途径负责校验和检查。有一点提醒读者请注意，有的前置条件和后置条件很容易检查（比如 len 是正数以及 len*wid 是正数），而另外一些则属于语义上的描述，很难直接处理。例如，我们该如何检查“返回值是长方形的面积，其中长方形的长和宽分别是 len 和 wid”呢？这属于一种语义上的约束，要想检查函数的返回值是否符合该条件，我们不但要弄清楚“长方形面积”的意思，还得把 len 和 wid 再乘一次。这种乘法操作应该在较高的精度下执行以避免产生溢出，显然这么做的代价非常大。

通过上述分析可知，写出 area() 的前置条件和后置条件有助于帮助我们发现这个简单函数的某些潜在风险。这并不令人意外。在程序设计过程中，写出前置条件和后置条件是一种非常有用的手段，提供了良好的文档描述。13.4 节将详细讨论编制及强化前置 / 后置条件的机制。

如果函数仅仅依赖于它的参数的话，则其前置条件也只与参数有关。但是我们还必须警惕另外一种情况，即，函数受非局部变量影响的情况（比如，成员函数与其对象的状态密切相关）。本质上，我们必须考虑函数以隐式参数形式读取的所有非局部变量值。类似地，有的函数仅仅计算并返回某个值，不会产生其他副作用。另外一些函数则可能向非局部对象中写入值，此时程序员就必须考虑这种情况，并且最好把它当成后置条件记录下来。

函数的设计者可以采取如下处理措施，例如：

[1] 确保每个输入都对应一个有效的处理结果（此时，我们无须添加前置条件）。
[2] 假定前置条件满足（依赖于函数的调用者不犯错误）。
[3] 检查前置条件是否满足，如果不满足的话抛出一个异常。
[4] 检查前置条件是否满足，如果不满足的话终止程序。

如果发生了违背后置条件的情况，意味着前置条件未被充分检查或者程序本身存在错误。13.4 节将详细讨论除检查之外的其他处理策略。

12.5 函数指针

与（数据）对象类似，由函数体生成的代码也置于某块内存区域中，因此它也有自己的地址。既然我们可以让指针指向对象，当然也就可以让指针指向函数。与此同时，出于某些考虑——有的与机器体系结构有关，有的与系统设计有关——我们不允许函数指针修改它所指的代码。程序员只能对函数做两种操作：调用它或者获取它的地址。通过获取函数地址得到的指针能被用来调用该函数。例如：

```
void error(string s) { /* ... */ }

void (*efct)(string);        // 指向函数的指针，该函数接受一个字符串参数，不返回任何东西

void f()
{
    efct = &error;           // efct 指向 error
    efct("error");           // 通过 efct 调用 error
}
```

编译器发现 efct 是个函数指针，因此会调用它所指的函数。也就是说，解引用函数指针时可以用 *，也可以不用；同样，获取函数地址时可以用 &，也可以不用：

```
void (*f1)(string) = &error;     // OK：等价于 = error
void (*f2)(string) = error;      // OK：等价于 = &error

void g()
{
    f1("Vasa");                  // OK：等价于 (*f1)("Vasa")
    (*f1)("Mary Rose");          // OK：等价于 f1("Mary Rose")
}
```

函数指针的参数类型声明与函数本身类似。进行指针赋值操作时，要求完整的函数类型都必须精确匹配。例如：

```
void (*pf)(string);   // 指向 void(string) 的指针
void f1(string);      // void(string)
int f2(string);       // int(string)
void f3(int*);        // void(int*)
void f()
{
    pf = &f1;         // OK
    pf = &f2;         // 错误：返回类型错误
    pf = &f3;         // 错误：参数类型错误

    pf("Hera");       // OK
    pf(1);            // 错误：参数类型错误

    int i = pf("Zeus");   // 错误：试图把 void 赋给 int
}
```

对于直接调用函数和通过指针调用函数这两种情况来说，参数传递的规则是一样的。

C++ 允许把一个函数指针转换成别的函数指针类型，但之后必须把得到的结果指针转换回它原来的类型，否则就会出现意想不到的情况：

```
using P1 = int(*)(int*);
using P2 = void(*)(void);

void f(P1 pf)
{
    P2 pf2 = reinterpret_cast<P2>(pf)
    pf2();                          // 可能发生严重错误
    P1 pf1 = reinterpret_cast<P1>(pf2);   // 把 pf2 "转换回来"
    int x = 7;
    int y = pf1(&x);                // OK
    // ...
}
```

我们用最底层的类型转换 reinterpret_cast 来执行函数指针类型的转换，这么做的原因是一旦我们使用了一个类型错误的函数指针，所得的结果会由系统决定，完全是无法预料的。例如在上面的例子中，被调函数需要向它的参数所指的对象中写入内容，但是调用 pf2() 根本就没有提供任何参数！

函数指针为算法的参数化提供了一种途径。因为 C 语言没有函数对象（见 3.4.3 节）或者 lambda 表达式（见 11.4 节）等机制，所以在 C 风格的代码中，我们经常能看到把函数指针用作函数参数的例子。例如，我们可以用函数指针的形式为一个排序函数提供它所需的比较操作：

```
using CFT = int(const void*, const void*);

void ssort(void* base, size_t n, size_t sz, CFT cmp)
/*
    使用 "cmp" 所指的比较函数把向量 "base" 的 "n" 个元素
    按照升序排列。
    元素的大小是 "sz"

    希尔排序（Knuth《计算机程序设计艺术》第 3 卷，第 84 页）
*/
{
    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i!=n; i++)
            for (int j=i-gap; 0<=j; j-=gap) {
                char* b = static_cast<char*>(base);      // 必要的类型转换
                char* pj = b+j*sz;                       // &base[j]
                char* pjg = b+(j+gap)*sz;                // &base[j+gap]
                if (cmp(pjg,pj)<0) {                // 交换 base[j] 和 base[j+gap]:
                    for (int k=0; k!=sz; k++) {
                        char temp = pj[k];
                        pj[k] = pjg[k];
                        pjg[k] = temp;
                    }
                }
            }
}
```

在上面的例子中，ssort() 不知道它排序的对象的类型，它接受的参数仅限于参与排序的元素数目（数组大小）、每个元素的大小以及一个用于执行比较操作的函数。ssort() 的类型与 C

语言库的标准排序算法 qsort() 一致。将来，一个实际的程序可以使用 qsort()、抑或 C++ 标准库算法 sort()（见 32.6 节），再或者特例化的排序函数。上述编码形式在 C 语言中很普遍，但是在 C++ 中却算不上实现排序算法最简洁的方式（见 23.5 节，25.3.4.1 节）。

这样的排序函数可以用来排列如下的表格：

```
struct User {
    const char* name;
    const char* id;
    int dept;
};

vector<User> heads = {
    "Ritchie D.M.",     "dmr",    11271,
    "Sethi R.",         "ravi",   11272,
    "Szymanski T.G.",   "tgs",    11273,
    "Schryer N.L.",     "nls",    11274,
    "Schryer N.L.",     "nls",    11275,
    "Kernighan B.W.",   "bwk",    11276
};

void print_id(vector<User>& v)
{
    for (auto& x : v)
        cout << x.name << '\t' << x.id << '\t' << x.dept << '\n';
}
```

要想执行排序操作，我们必须首先定义一个合适的比较函数。这个比较函数的作用是：当它的第一个参数小于第二个参数时返回一个负数，当两个参数相等时返回 0，其他情况下返回一个正数：

```
int cmp1(const void* p, const void* q)   // 比较 name 字符串
{
    return strcmp(static_cast<const User*>(p)->name,static_cast<const User*>(q)->name);
}

int cmp2(const void* p, const void* q)   // 比较 dept 数字
{
    return static_cast<const User*>(p)->dept - static_cast<const User*>(q)->dept;
}
```

当对函数指针进行赋值或者初始化操作时不存在参数或者返回类型的隐式类型转换。因此，下面的代码在后续使用时不得不引入强制类型转换，而这种用法既不优雅，又充满了错误风险：

```
int cmp3(const User* p, const User* q)  // 比较 ids
{
    return strcmp(p->id,q->id);
}
```

从 cmp3() 的定义可知，它接受的参数类型应该是 const User*，但是把 cmp3 作为 ssort() 的参数显然会违反这一约定（见 15.2.6 节）。

下面的代码执行排序并输出的操作：

```
int main()
{
    cout << "Heads in alphabetical order:\n";
```

```
    ssort(heads,6,sizeof(User),cmp1);
    print_id(heads);
    cout << '\n';

    cout << "Heads in order of department number:\n";
    ssort(heads,6,sizeof(User),cmp2);
    print_id(heads);
}
```

作为对比，上述功能还可以有别的实现方式：

```
int main()
{
    cout << "Heads in alphabetical order:\n";
    sort(heads.begin(), head.end(),
        [](const User& x, const User& y) { return x.name<y.name; }
    );
    print_id(heads);
    cout << '\n';

    cout << "Heads in order of department number:\n";
    sort(heads.begin(), head.end(),
        [](const User& x, const User& y) { return x.dept<y.dept; }
    );
    print_id(heads);
}
```

在这一版的代码中，既不需要提供尺寸信息，也不需要设计任何辅助函数。如果你不想显式地使用 begin() 和 end()，可以考虑改用一个接受容器作为参数的 sort()（见 14.4.5 节）：

```
sort(heads,[](const User& x, const User& y) { return x.name<y.name; });
```

我们可以使用一个重载函数的地址对函数指针进行赋值或者初始化操作。此时，我们利用目标的类型从一组重载函数中选择一个恰当的版本。例如：

```
void f(int);
int f(char);

void (*pf1)(int) = &f;      // void f(int)
int (*pf2)(char) = &f;      // int f(char)
void (*pf3)(char) = &f;     // 错误：不存在 void f(char)
```

同样，我们也可以获得成员函数的地址（见 20.6 节）。但是指向成员函数的指针与指向非成员函数的指针有很大区别。

我们可以将指向 noexcept 函数的指针声明成 noexcept 的，例如：

```
void f(int) noexcept;
void g(int);

void (*p1)(int) = f;                // OK：但是丢失了有用信息
void (*p2)(int) noexcept = f;       // OK：保留了 noexcept 信息
void (*p3)(int) noexcept = g;       // 错误：我们并不知道 g 不会抛出异常
```

函数指针必须反映函数的链接信息（见 15.2.6 节）。链接说明和 noexcept 都不能出现在类型别名中：

```
using Pc = extern "C" void(int);    // 错误：别名中出现了链接说明
using Pn = void(int) noexcept;      // 错误：别名中出现了 noexcept
```

12.6　宏

宏在C语言中非常重要，但在C++中的作用就小得多了。关于宏的最重要的原则是：除非万不得已，否则不要使用宏。几乎每个宏都意味着一点美中不足甚至是缺陷，这样的瑕疵可能是语言本身的，也可能是程序或者程序员的。宏会重排程序文本，在此之前编译器甚至还没有接触到程序，也不知道程序文本本来的样子是什么。因此，对于很多辅助工具来说，程序中包含宏都是个大麻烦。调试器、交叉引用和性能评测工具很难在含有宏的程序上发挥什么作用。如果你一定要使用宏，请务必仔细阅读当前环境中关于C++预处理程序的参考手册，千万别自作聪明。同时，为了提醒你的代码的读者注意，请遵循书写宏的约定，即在命名宏时尽量使用大写字母。关于宏的语法在 § iso.16.3 中有详细介绍。

我建议，只有在进行条件编译（见 12.6.1 节）尤其是执行包含文件防护（见 15.3.3 节）的任务时再使用宏。

下面是一个简单的宏的例子：

```
#define NAME rest of line
```

其中，NAME 只是一个代记符号，它的实际内容是 rest of line。例如：

```
named = NAME
```

会展开成

```
named = rest of line
```

我们也可以在定义宏时要求它接受参数，例如：

```
#define MAC(x,y) argument1: x argument2: y
```

我们如果想使用 MAC，必须提供两个字符串作为参数。当展开 MAC() 时，这两个参数会替代 x 和 y 的位置。例如：

```
expanded = MAC(foo bar, yuk yuk)
```

会展开成

```
expanded = argument1: foo bar argument2: yuk yuk
```

宏的名字不允许重载，同时，宏预处理代码也没有能力处理递归调用。例如：

```
#define PRINT(a,b) cout<<(a)<<(b)
#define PRINT(a,b,c) cout<<(a)<<(b)<<(c)      /* 会遇到麻烦，不要重载，重新定义一个别的名字 */

#define FAC(n) (n>1)?n*FAC(n-1):1             /* 会遇到麻烦，试图递归宏 */
```

宏对于C++的语法涉及很少，更与C++的类型系统和作用域规则完全无关，它的作用就是操作字符串本身。宏只有在展开后才能被编译器看到，因此，如果在宏里面存在错误，只有当宏展开后才可能发现。编译器无法在定义宏的时候发现错误，这就导致关于宏的报错信息常常晦涩不清，让人不明所以。

下面是一些看起来貌似有用的宏：

```
#define CASE break;case
#define FOREVER for(;;)
```

下面这些宏完全没有存在的必要：

```
#define PI 3.141593
#define BEGIN {
#define END }
```

还有一些非常危险的宏：

```
#define SQUARE(a) a*a
#define INCR_xx (xx)++
```

要想知道它们为什么危险，不妨将其展开：

```
int xx = 0;        // 全局计数器

void f(int xx)
{
    int y = SQUARE(xx+2);    // y=xx+2*xx+2; 即，y=xx+(2*xx)+2
    INCR_xx;                 // 实际增加的是参数 xx 的值，而非全局 xx 的值
}
```

如果必须使用宏，记得对于全局名字一定要使用作用域解析运算符 ::（见 6.3.4 节），同时尽量用括号把宏的参数括起来。例如：

```
#define MIN(a,b) (((a)<(b))?(a):(b))
```

这么做可以解决一些简单的语法问题（编译器可以发现并报告此类问题），但对于宏产生的副作用还是无能为力。例如：

```
int x = 1;
int y = 10;
int z = MIN(x++,y++);          // x 变成了 3; y 变成了 11
```

如果你写的宏特别复杂（前提是你不得不这么做）以至于需要添加注释，我的建议是最好使用 /* */ 形式的注释。C++ 工具有时候会用到老的 C 语言预处理程序，而这些预处理程序是无法辨别 // 注释的。例如：

```
#define M2(a) something(a)    /* 这些注释应该经过认真考虑和组织 */
```

你可以用宏来设计你自己的语言。也许有的人更喜欢这种“改进的语言”而非原来的 C++，但是其他程序员就很难理解你写的程序了。而且，预处理程序本身的能力非常有限。你根本没必要也不可能处理太复杂的宏。在现代 C++ 语言中，auto、constexpr、const、decltype、enum、inline、lambda 表达式、namespace 和 template 机制可以完成原来的预处理机制的大多数功能。例如：

```
const int answer = 42;

template<class T>
inline const T& min(const T& a, const T& b)
{
    return (a<b)?a:b;
}
```

在编写宏时，经常需要命名某些东西，我们可以用 ## 宏运算符把两个字符串拼接成一个。例如：

```
#define NAME2(a,b) a##b

int NAME2(hack,cah)();
```

会展开成

```
int hackcah();
```

在置换字符串中，如果参数的名字前面有一个单独的 #，表示此处是包含宏参数的字符串。例如：

```
#define printx(x) cout << #x " = " << x << '\n';

int a = 7;
string str = "asdf";

void f()
{
    printx(a);      // cout << "a" << " = " << a << '\n';
    printx(str);    // cout << "str" << " = " << str << '\n';
}
```

请注意，我们在上面的代码中写成 #x " = " 而非 #x << " = "，这不是书写错误，而是“比较聪明的代码”。相邻的字符串字面值常量会被连接在一起（见 7.3.2 节）。

指示语句

```
#undef X
```

确保没有任何一个宏定义 X——不论在该指示语句之前是否存在名为 X 的宏。这种机制使得我们可以不必受某些意料之外的宏的影响。不过，有时候我们很难弄清楚 X 到底对一段代码应该有什么样的影响。

宏的参数列表（“置换列表”）可以为空：

```
#define EMPTY() std::cout<<"empty\n"
EMPTY();        // 输出 "empty\n"
EMPTY;          // 错误：缺少宏置换列表
```

我花了很长时间才让自己确信空的宏参数列表不算是一种有错误风险或者恶意的代码。

宏甚至可以是可变参数的，例如：

```
#define err_print(...) fprintf(stderr,"error: %s %d\n", __VA_ARGS__)
err_print("The answer",54);
```

其中，省略号（...）的意思是 __VA_ARGS__ 把实际传入的参数当成一个字符串，因此输出结果是：

```
error: The answer 54
```

12.6.1　条件编译

有一种宏的用法是无法替代的。如果定义了 IDENTIFIER，则指示语句

```
#ifdef IDENTIFIER
```

什么也不做；否则，该语句将忽略下一条 #endif 语句之前的所有输入。例如：

```
int f(int a
#ifdef arg_two
,int b
#endif
);
```

除非我们 #define 了一个名为 arg_two 的宏，否则将得到

```
int f(int a
);
```

大多数辅助工具都假定程序员具有正常的行为逻辑，显然上面这个宏会让这类工具感到困惑不已。

一般情况下，#ifdef 不会有太大的危害。只要按照规范使用，#ifdef 及与之对应的 #ifndef 都不会带来什么问题。详情请参见 15.3.3 节。

必须谨慎选择用于控制 #ifdef 的宏名字，确保它们不会与现有的标识符冲突。例如：

```
struct Call_info {
    Node* arg_one;
    Node* arg_two;
    // ...
};
```

这段代码看起来没什么问题，但是如果有人写了下面的宏，就会产生混淆：

```
#define arg_two x
```

不幸的是，在很多常见的头文件中包含了大量这种既危险又没什么必要的宏。

12.6.2 预定义宏

编译器预定义了一些宏（§iso.16.8，§iso.8.4.1）：

- __cplusplus：在 C++ 编译器中有定义（C 语言编译器没有）。在 C++11 程序中它的值是 201103L，在之前的 C++ 标准中该值相应地小一些。
- __DATE__："yyyy:mm:dd" 格式的日期。
- __TIME__："hh:mm:ss" 格式的时间。
- __FILE__：当前源文件的名字。
- __LINE__：当前源文件的代码行数。
- __FUNC__：是一个由具体实现定义的 C 风格字符串，表示当前函数的名字。
- __STDC_HOSTED__：如果当前实现是宿主式的（见 6.1.1 节）则为 1；否则为 0。

此外，还有一些宏是实现根据具体条件定义的：

- __STDC__：在 C 语言编译器中有定义（C++ 编译器中没有）。
- __STDC_MB_MIGHT_NEQ_WC__：在 wchar_t 的编码体系中，如果基本字符集（见 6.1 节）的成员的值与它作为普通字符字面值常量的值可能不同，则为 1。
- __STDCPP_STRICT_POINTER_SAFETY__：如果当前实现有严格的指针安全机制（见 34.5 节），则为 1；否则是未定义的。
- __STDCPP_THREADS__：如果程序可以有多个执行线程，则为 1；否则是未定义的。

例如：

```
cout << __FUNC__ << "() in file " << __FILE__ << " on line " << __LINE__ << "\n";
```

此外，大多数 C++ 实现允许用户在命令行或者其他形式的编译时环境中定义任意多个宏。例如，除非编译过程工作于（某些依赖于实现的）"调试模式"，否则都定义了 NDEBUG 并被 assert() 宏使用（见 13.4 节）。这么做可能有用，但是同时也意味着读者仅靠阅读源代码还无法完全理解程序的含义。

12.6.3　编译指令

具体的C++实现常常提供一些有别于标准甚至标准之外的功能。显然，标准无法规定这样的额外功能应该以何种方式提供，但是标准的句法应该是以预处理指示#pragma作为前缀的一个符号行。例如：

```
#pragma foo bar 666 foobar
```

如果可能，尽量避免使用#pragma。

12.7　建议

［1］把有用的操作“打包”在一起构成函数，然后认真起个名字；12.1节。
［2］一个函数应该对应逻辑上的一个操作；12.1节。
［3］让函数尽量简短；12.1节。
［4］不要返回指向局部变量的指针或者引用；12.1.4节
［5］如果函数必须在编译时求值，把它声明成constexpr；12.1.6节。
［6］如果函数无法返回结果，把它设置为[[noreturn]]；12.1.7节。
［7］对小对象使用传值的方式；12.2.1节。
［8］如果你想传递无须修改的大值，使用传const引用的方式；12.2.1节。
［9］尽量通过return值返回结果，不要通过参数修改对象；12.2.1节。
［10］用右值引用实现移动和转发；12.2.1节。
［11］如果找不到合适的对象，可以传入指针（nullptr表示“没有对象”）；12.2.1节。
［12］除非万不得已，否则不要传递非const引用；12.2.1节。
［13］const的用处广泛，程序员应该多用；12.2.1节。
［14］我们认为char*或者const char*参数指向的是C风格字符串；12.2.2节。
［15］避免把数组当成指针传递；12.2.2节。
［16］用initializer_list<T>传递元素类型相同但是元素数量未知的列表（用其他容器也可以）；12.2.3节。
［17］避免使用数量未知的参数(...)；12.2.4节。
［18］当几个函数完成的功能在概念上一致，仅仅是处理的类型有区别时，使用重载；12.3节。
［19］在整数类型上重载时，提供一些函数以消除二义性；12.3.5节。
［20］为你的函数指定前置条件和后置条件；12.4节。
［21］与函数指针相比，优先使用函数对象（包括lambda）和虚函数；12.5节。
［22］不要使用宏；12.6节。
［23］如果必须使用宏，一定要用很多大写字母组成宏的名字，尽管这样的名字看起来会很丑陋；12.6节。

异常处理

当我打断别人时，
不要打断我。

——温斯顿 S. 丘吉尔

- 错误处理
  异常；传统的错误处理；渐进决策；另一种视角看异常；何时不应使用异常；层次化错误处理；异常与效率
- 异常保障
- 资源管理
  finally
- 强制不变式
- 抛出与捕获异常
  抛出异常；捕获异常；异常与线程
- vector 的实现
  一个简单的 vector；显式地表示内存；赋值；改变尺寸
- 建议

13.1 错误处理

本章介绍如何用异常进行错误处理。我们必须基于一定的策略综合运用各项语言机制才能高效地处理错误。本章的内容主要分为两部分：一是异常安全保障（exception-safety guarantee），它是程序从运行时错误中快速恢复的关键；另一个是使用构造函数和析构函数进行资源管理的资源获取即初始化（Resource Acquisition Is Initialization，RAII）技术。因为异常安全保障和资源获取即初始化都依赖于不变式（invariant）的规范，所以本章也会介绍一些关于强制断言的内容。

本章提及的语言功能和技术主要是为了处理软件中的错误；异步事件处理属于另外一类问题。

我们对错误的讨论主要集中在那些不能被局部处理（即在一个小函数内处理）的错误上，此类错误通常需要在程序的其他部分通过单独的错误处理活动来处理。在程序中，这类专门处理错误的代码常常具有较强的独立性。因此，我习惯于把此类由程序调用专门处理某项特定任务的模块称为“库”。库也是由普通代码组成的，但是在讨论错误处理问题时，我们应该意识到库的设计者通常并不知道他的库会被用在何种情况中：

- 库的作者能检测到运行时错误，但是不知道如何处理。
- 库的用户可能知道该如何处理运行时错误，但是难以检测到错误（或者该错误已经在用户代码中被处理过了，库根本就看不到）。

我们关于异常的讨论集中在某些特定场景中，这些特定场景包括长时间运行的系统、具有严格依赖关系的系统以及库。不同程序的要求各有区别，我们所做的工作应该体现出这种差异性。例如，如果我不过写了一个两页的程序供自己使用，那么就不可能将本章介绍的所有技术都用到这个程序中。但是，本章提及的很多技术都有助于简化代码，因此我在日常编程工作中还是会用到它们。

13.1.1　异常

异常（exception）的概念可以帮助我们将信息从检测到错误的地方传递到处理该错误的地方。如果函数无法处理某个问题，则抛出（throw）异常，并且寄希望于函数的调用者能直接或者间接地处理该问题。函数如果希望处理某个问题，可以捕获（catch）相应的异常（见2.4.3.1节）：

- 主调组件如果想处理某些失败的情形，可以把这些异常置于try块的catch从句中。
- 被调组件如果无法完成既定的任务，可以用throw表达式抛出一个异常来说明这一情况。

下面的例子虽然简单，但是结构完整，能够说明异常处理的机制：

```
void taskmaster()
{
    try {
        auto result = do_task();
        // 使用 result
    }
    catch (Some_error) {
        // 执行 do_task 时发生错误：处理该问题
    }
}

int do_task()
{
    // ...
    if (/* 能够执行该任务 */)
        return result;
    else
        throw Some_error{};
}
```

在上面的代码中，taskmaster()请求do_task()完成某项工作。如果do_task()能够完成该工作并返回一个正确的结果，则一切正常。否则，do_task ()必须抛出一个异常以报告该错误。taskmaster()准备处理Some_error，但是程序也可能抛出其他种类的异常。例如，do_task()可能调用其他函数来完成一大堆子任务，而其中的某个子函数有可能抛出它自己的异常表示不能完成分配的任务。最终的结果是，如果do_task ()抛出了一个Some_error之外的异常，就意味着taskmaster()无法正常完成它的工作了，调用了taskmaster()的其他代码必须负责处理该问题。

一个被调用的函数不能仅仅报告错误就了事。如果程序想继续运行下去（而非仅仅输出一条错误信息后终止），则该函数返回的同时必须确保程序的状态良好且没有泄漏任何资源。C++的异常处理机制与构造函数/析构函数机制及并发机制一道为这一目标提供了保障（见5.2节）。异常处理机制：

- 是对传统技术的一次改进，传统技术常常低效、粗糙且充满了错误风险。
- 是完备的，它能处理普通代码中发生的各类问题。
- 允许程序员显式地把错误处理代码从“普通代码”中分离出来，从而提高了程序的可读性，也便于使用辅助工具。
- 提供了一种更规范的错误处理机制，使得多个单独编写的程序片段合作起来更容易了。

异常是指被程序抛出的一个对象，它表示在程序中出现了一个错误。异常可以是任意类型的对象，只要它能被拷贝即可。但是，我们强烈建议程序员使用自定义的专门用于表示错误的类型。通过这么做我们可以尽量避免两个完全无关的库使用同一个值（比如17）表示两个完全无关的错误，从而减少错误恢复代码陷入混乱的可能。

在程序中有一些代码表现出对于处理某类特定异常的兴趣（catch从句），由它负责捕获异常。因此，最简单的定义异常的方法就是为一种错误定义一个专门的类，当遇到错误时抛出它。例如：

```
struct Range_error {};

void f(int n)
{
    if (n<0 || max<n) throw Range_error {};
    // ...
}
```

如果你不喜欢这种形式，标准库还定义了一个小型的异常类层次供程序员使用（见13.5.2节）。

异常可以携带一些关于错误的描述信息。异常的类型表示错误的种类，异常携带的数据则记录了错误出现时的情形。例如，在标准库异常中含有一个字符串，它可以告诉我们抛出异常的位置（见13.5.2节）。

13.1.2 传统的错误处理

当函数检测到某个无法局部处理的问题（比如越界访问）并且必须向函数的调用者报告时，除了使用异常机制处理该错误，其他几种传统的处理方式都有各自的不足：

- 终止程序。这是一种非常极端的处理方式，例如：

  ```
  if (something_wrong) exit(1);
  ```

  对于绝大多数错误来说，我们有能力也必须处理得更好。例如，在大多数情况下，我们在终止程序之前至少应该给出一条比较准确的错误信息或者把该错误记录下来。尤其是，如果库不清楚它所处的程序的目的和作用，就不应该简单地调用exit()或者abort()。如果一个程序不允许轻易崩溃，显然在其中不能使用任何采用无条件终止的库。

- 返回错误值。这种策略也并非百试不爽，因为有的时候我们根本就得不到合适的“错误值”。例如：

  ```
  int get_int();  // 从输入中获得下一个整数
  ```

  对于上面这个执行输入操作的函数来说，每个int值都可能是它的结果，因此我们无法指定其中某个值作为输入错误的标识值。一种解决方案是修改get_int()令其返回一对值。然而即使这么做可以应付一些情况，看起来也不怎么方便。以后每次调

用该函数都必须检查一下它返回的错误值，无形中增加了程序的工作量，而且也会使程序的规模翻倍（见 13.1.7 节）。另外，函数的调用者常常会忽略可能出现的错误或者忘记检验返回的值。总之，这种策略不足以系统地检查并处理全部错误。例如，printf()（见 43.3 节）遇到输出错误或者编码错误时会返回一个负数，但是程序员们根本就不会检查它。最后，有一些函数根本就没有返回值，构造函数就是个很明显的例子。

- 返回合法值，而程序却处于“错误状态”。问题是主调函数可能没有意识到程序已经处于错误状态了。例如，许多标准 C 语言库函数设置一个非局部变量 errno 来表示错误（见 43.4 节，40.3 节）：

```
double d = sqrt(-1.0);
```

此时，d 的值本身没什么意义，标准库设置 errno 来表示 -1.0 对于浮点数平方根函数来说是个无效的参数。然而，程序通常很难一直追踪 errno 及与之类似的非局部状态（包括设置及检测这些值），因此很难避免某些错误的调用的返回值引发新的错误。此外，在应用并发机制时，用非局部变量记录错误状态的做法不太奏效。

- 调用错误处理函数。例如：

```
if (something_wrong) something_handler();  // 问题并未得到解决，只是暂时转移了
```

这种策略其实只是之前几种处理措施的变形。我们很容易继续发问：“那么错误处理函数应该干什么呢？”除非这里的错误处理函数能够完整地解决所有问题，否则它还是会陷入之前的境地当中：终止程序、返回一个标识发生了错误的值、设置错误状态或者抛出异常。而且如果错误处理函数能够在不打扰函数调用者的前提下解决问题，我们为什么还要把它当成错误呢？

在旧有程序中，上述几种方法可能毫无组织地同时出现。

13.1.3　渐进决策

在异常处理模式中，对于未处理的错误（未捕获的异常）的最终响应是终止该程序。这一点可能会让一部分程序员稍感意外。我们采取的机制基于渐进决策的思想，并期望得到最优结果。因此看起来异常处理机制让程序变得“脆弱”了，因为我们不得不付出更多努力以使得程序能以一种良好的方式运行。不过，无论如何它也比在开发过程中或者开发完成后程序交到不明真相的用户手中时出现错误强多了。如果不允许终止程序，我们完全可以捕获所有异常（见 13.5.2.2 节）。这样就能确保只有当程序员希望的时候，异常才会终止程序。通常情况下，这要优于在原来的机制中由于未完全恢复系统状态而导致程序的无条件终止。如果在某处终止程序是可以接受的一种选择，则未被捕获的异常将会调用 terminate()（见 13.5.2.5 节）。同样，noexcept 说明符（见 13.5.1.1 节）也可以显式地表达这种意图。

有时候，人们希望通过输出一些错误信息或者弹出对话框寻求用户帮助等方式减少“渐进决策”的负面影响。这类措施主要在调试状态下有用，因为此时程序的用户就是程序员本身，他们对程序的结构非常熟悉。一旦库被移交到开发者之外的人手中，那么再向用户或者操作者（有时甚至都不存在这样的角色）寻求帮助就完全没有意义了。形象地说，库不应该“多嘴多舌”。即使必须向用户提供某种信息，异常处理器也应该尽量组织一段合适的语言（比如向芬兰用户提供芬兰语的信息，或者把错误信息置于 XML 文件中以便日志系统使用）。

异常机制实际上提供了一种途径，如果一部分代码能检测到问题但是无法修复，那么它可以把该问题移交给系统中其他能够解决该问题的部分。只有极少数的代码有可能编制出有意义的错误消息，前提是这部分代码非常清楚程序运行的上下文环境才行。

请读者务必注意，尽管与之前的技术相比异常处理机制已经显得规范多了，但它毕竟不是仅包含局部控制流的语言功能，因此严格来说它的结构化水平还是比较低的。异常处理仍然是一项困难的工作。不管怎么说，C++的异常处理机制使得程序员可以在了解系统结构的前提下以一种很自然的方式处理错误。异常使得错误处理的复杂性大白于天下，但是这种复杂性并不是由异常造成的。打个比方说，你听到了坏消息，但是你不能责怪那个给你传话的人。

13.1.4 另一种视角看异常

对于“异常”这个词，不同的人会有不同的理解。C++的异常处理机制主要用于处理那些无法在局部范围内解决的问题（“异常状况”）。它尤其善于处理由相互独立开发的组件构成的程序中的错误。“异常”这个词有时候会造成一些误导，尤其是当程序的某个部分无法完成它的任务，但是又找不到它有什么真正的异常表现时。程序运行时出现很多次的事件能算异常吗？如果有一种情况，它的出现既在意料之中，又有相应的应对措施，那它能算错误吗？这两个问题的答案都是“Yes”。“异常”不等价于“几乎不会发生”或者“灾难”。

13.1.4.1 异步事件

C++的异常机制主要处理同步异常，比如数组边界检查以及I/O错误等。异步事件（比如键盘中断或者电源失效）不属于异常的范畴，当然也就不能用异常机制直接处理。从本质上来看，异步事件与本章定义的异常有很大区别，我们必须定义专门的机制来彻底、高效地处理它。很多系统提供诸如信号一类的机制来处理异步事件，但是因为这类机制紧密依赖于系统本身，所以我们不做过多介绍。

13.1.4.2 不是错误的异常

异常的意思是“系统的某部分不按我们的期望行事”（见13.1.1节，13.2节）。

在一个系统中，抛出异常的次数不能比调用函数还频繁，否则就会掩盖程序本来的面貌。但是对于大规模程序来说，在其正常的运转过程中肯定会抛出并捕获一些异常。

如果我们能预期到一个异常会出现，并且准备好了相应的措施以确保它不会影响程序的正常行为，那我们还能把它当成是一个错误吗？其实其本质是程序员进行了一种联想，我们把异常想象成是错误，然后把异常处理机制想象成是处理错误的工具。换一种角度思考，我们完全可以把异常处理机制当成一种新的控制结构，它的任务也是向调用者返回某个值。举一个二叉树搜索函数的例子：

```
void fnd(Tree* p, const string& s)
{
    if (s == p->str) throw p;          // 找到；
    if (p->left) fnd(p->left,s);
    if (p->right) fnd(p->right,s);
}

Tree* find(Tree* p, const string& s)
{
    try {
        fnd(p,s);
```

```
        }
        catch (Tree* q) {      // q->str=="s"
            return q;
        }
        return 0;
    }
```

我们应该避免像这样编写代码，也许有的人会认为它比较巧妙，但它很可能会引起混淆并使得程序的效率低下。程序员应该尽一切可能坚持“异常处理是错误处理”的观点。这样做有助于把代码清晰明确地划分成两部分：普通代码和错误处理代码，从而提高代码的可读性。此外，C++ 语言在实现异常机制时对它进行了优化，优化的前提正是明确它的作用是要处理异常。

错误处理是一项很困难的工作。任何有助于建立简洁的错误模型并有效处理错误的举措，都显得弥足珍贵。

13.1.5　何时不应使用异常

异常是唯一一种完全通用的系统化地处理 C++ 程序错误的机制。但是我们也必须认识到有的程序出于历史的或者实践的原因无法使用异常。例如：

- 嵌入式系统中的时间关键型组件，我们必须确保该组件的任务在预定的最大时限内完成。因为到目前为止也没有哪个工具可以准确地计算出 throw 和 catch 在处理异常时所需的时间上限，所以我们必须采用其他处理错误的方法。
- 规模较大的旧系统，它的资源管理非常混乱（比如用“裸”指针、new 和 delete 杂乱地“管理”自由存储），没有采用资源句柄（比如 string 和 vector；见 4.2 节，4.4 节）等系统化的管理模式。

在这些情况下，我们不得不采用“传统的”（异常之前的）技术处理错误。这些程序产生的历史原因各不相同，内在的约束也有很多差异，因此我无法对如何处理它们给出一套行之有效的方法。但是，我可以罗列两种比较流行的技术：

- 模仿 RAII 在每个含有构造函数的类中增加一个 invalid() 操作以返回一些 error_code，并约定 error_code==0 表示执行成功。如果构造函数没能成功地建立类的不变式，则它应确保不产生资源泄漏并且令 invalid() 返回一个非零的 error_code。这种方案使得我们可以从构造函数中获得错误的状态，在每次构造对象后系统地检查 invalid() 的返回值，根据错误的种类进行相应的处理。例如：

```
void f(int n)
{
    my_vector<int> x(n);
    if (x.invalid()) {
        // ... 处理错误 ...
    }
    // ...
}
```

- 构建一个既能返回结果又能抛出异常的函数，它返回的是 pair<Value ,Error_code>（见 5.4.3 节）。这种方案使得我们可以在每次调用函数后系统地检查 error_code 的返回值，并根据错误的种类进行相应的处理。例如：

```
void g(int n)
{
```

```
    auto v = make_vector(n); // 返回一个 pair
    if (v.second) {
        // ... 处理错误 ...
    }
    auto val = v.first;
    // ...
}
```

上述模式及其变形可以有效地处理错误，但是与系统化的异常处理机制相比还是显得有点笨拙。

13.1.6　层次化错误处理

异常处理机制的目的是提供一种措施，使得当某处程序发现有未正常完成的任务时（检测到一个“异常状况”），它可以尽快通知程序的另一部分。我们假设程序的这两个部分是独立完成的，并且其中负责处理异常的部分能对发生的错误采取有效措施。

要想在程序中有效地处理异常，我们必须站在更高的高度上全盘考虑。也就是说，程序的各个部分必须就如何使用异常以及在哪儿处理错误达成一致。异常处理机制本来就是非局部的，因此我们有必要坚持一种全局的策略。这意味着我们应该在设计程序时尽早开始考虑异常处理的问题，并且提出的策略必须简单（相对于程序整体的复杂性而言）、明确。对于像错误恢复这种本身很微妙的事情来说，涉及的东西越简单越好。

成功的容错系统都是多层级的。按照自底向上的顺序，每一层级在它力所能及的范围内处理尽量多的错误，把剩下的错误留给更高层级处理。异常处理遵循这一准则。在正常的处理流程之外，terminate() 和 noexcept 还对一些特殊状况提供了额外的处理措施。前者可以应对由异常处理机制本身的漏洞或者它未被完整执行而造成异常未被捕获的情况；后者则适用于错误恢复不可行的情况。

不是每个函数都应当承担防火墙的作用。换句话说，不是每个函数都有能力检验它的前置条件是否足够充分，并且确保它的后置条件不论遇到任何错误都能达成。其中的原因与程序员和程序本身都有很大的关系。然而，对于大规模程序来说：

［1］ 为了确保完全“可靠”而要付出的工作量实在太大了，根本不可能做到。

［2］ 要让系统以令人满意的方式运行，其时空开销同样非常巨大（有可能需要一遍又一遍地检查同样的错误，比如无效参数）。

［3］ 用其他语言编写的代码可不会遵循同样的规则。

［4］ 为了实现局部“可靠”而造成的程序复杂性实际上会影响整个系统的全局可靠。

从必要性、易用性和经济性的角度考虑，我们有必要把程序分割成行为明确的子系统，它们或者成功执行，或者失败，但它们的行为都是处于定义良好的框架下的。主要的库、子系统、关键的接口函数都应该以这种方式设计，而且在绝大多数系统中，我们也确实能够做到让每个函数都以正确的方式成功执行或者报告失败结果。

通常情况下，我们不可能从零开始设计所有代码。因此要想把一种通用的错误处理策略应用到程序的各个部分上，我们还必须兼顾那些基于其他策略设计的程序片段。我们必须详细了解各个程序片段是如何管理资源的，当它们遇到错误时会到达何种状态。即使不同的程序片段是基于各自的策略开发的，我们也希望从整体上看它们遵循同一种错误处理机制。

偶尔也需要从一种错误报告模式转换成另外一种。例如，我们可能会检查 errno 的值，

在调用C语言库后抛出异常；反之，也可能在从C++库返回C程序前捕获异常并设置errno：

```
void callC()    // 在C++中调用C函数，把errno转换成throw
{
    errno = 0;
    c_function();
    if (errno) {
        // ... 如果可能且必要的话，执行局部清理 ...
        throw C_blewit(errno);
    }
}

extern "C" void call_from_C() noexcept        // 在C语言中调用一个C++函数，把throw转换成errno
{
    try {
        c_plus_plus_function();
    }
    catch (...) {
        // ... 如果可能且必要的话，执行局部清理 ...
        errno = E_CPLPLFCTBLEWIT;
    }
}
```

在这样的例子中，我们必须有系统性的解决方案以确保错误报告风格转换能够完整执行。不幸的是，此类转换常常出现在“杂乱无章的代码”中，它们缺乏明确的错误处理机制，系统性也无从谈起。

错误处理应当尽量层次化。如果函数检测到了一个运行时错误，它就不应该向它的调用者寻求帮助或者请求资源了。这类请求会让系统的依赖关系形成一种环状结构，程序变得难以理解，并且错误处理和恢复的代码有可能会陷入死循环。

13.1.7　异常与效率

原则上讲，我们可以做到当不抛出异常时就不会产生异常处理的开销。而且，这么做也能尽量让抛出异常的代价不一定像调用函数那么高。总的来说，有可能在不显著增加内存负担的前提下保持与C语言调用序列、调试器规则等的互通性，但是比较难。然而，即使我们不使用异常，要想实现同样的错误处理功能也不会是免费的。而且在相当一部分旧系统中，几乎一半的代码都是用来处理错误的。

下面是一个简单的函数f()，它不包含任何异常处理：

```
void f()
{
    string buf;
    cin>>buf;
    // ...
    g(1);
    h(buf);
}
```

但是g()和h()都可能抛出异常，因此f()必须增加专门的代码以确保当异常出现时buf能被正常销毁。

假如g()不抛出异常的话，它需要以别的方式报告错误。因此作为对比，使用传统代码而非异常会表现为如下形式：

```
bool g(int);
bool h(const char*);
char* read_long_string();

bool f()
{
    char* s = read_long_string();
    // ...
    if (g(1)) {
        if (h(s)) {
            free(s);
            return true;
        }
        else {
            free(s);
            return false;
        }
    }
    else {
        free(s);
        return false;
    }
}
```

使用局部缓冲替换 s 可以省去调用 free() 的代码，但是我们必须自己执行边界检查。这么做的复杂性会更高。

人们无法保证总能像上面的代码一样把错误处理得有条不紊，而且有的时候确实也没必要。然而，当我们需要细致地、系统地处理错误时，最好把这种工作留给计算机（也就是异常处理机制）去做。

noexcept 说明符（见 13.5.1.1 节）对于改善生成的代码很有帮助。举个例子：

```
void g(int) noexcept;
void h(const string&) noexcept;
```

这样，f() 生成的代码就有可能改进了。

传统的 C 函数不会抛出异常，因此大多数 C 函数都能声明成 noexcept 的。标准库函数的实现者清楚地知道只有一部分标准库 C 函数（比如 atexit() 和 qsort()）能抛出异常，因此他们可以利用这一点生成更好的代码。

在把一个“C 函数”声明成 noexcept 前，最好先花一点时间仔细想想它是否可能抛出异常。例如，它可能已经被转换成使用 C++ 运算符 new，因此会抛出 bad_alloc，或者它将调用可能会抛出异常的 C++ 库。

当然，在缺乏评判标准的情况下讨论效率毫无意义。

13.2 异常保障

要想从错误中恢复过来，换句话说，要想捕获异常并继续执行程序，我们必须清楚地知道恢复之前和之后程序的确切状态。只有这样，恢复才有意义。如果在通过抛出异常终止某个操作后，程序仍然处于有效状态，则称这个操作是异常安全（exception-safe）的操作。显然，我们必须明确“有效状态”指的到底是什么。同时，在使用异常设计程序的实践过程中，我们也必须把“异常安全”拆解开来，转化成几条具体的保障机制，毕竟这个概念太过泛泛了。

对于对象来说，我们通常假定一个类含有类的不变式（见 2.4.3.2 节，17.2.1 节）。我们假定该不变式由类的构造函数创建，并由所有有权访问对象的表示的函数负责维护，直到我们销毁了该对象为止。因此，有效状态（valid state）是指构造函数已经完成且尚未执行析构函数的状态。对于那些无法看成对象的数据，推导方式也类似。也就是说，如果两个非局部数据存在某种联系，则我们必须考虑建立一个不变式来表示这种联系，并且我们的恢复操作必须保持不变式。例如：

```
namespace Points {      // (vx[i],vy[i]) 表示一个坐标点 i
    vector<int> vx;
    vector<int> vy;
};
```

此处，我们假定 vx.size()==vy.size() 永远成立。但是这一约定仅仅是写在注释中，而编译器并不会阅读注释的内容。因此，这种隐式的不变式很难被发现，也不易于维护。

在 throw 之前，函数必须将所有对象置于有效的状态，但是这种有效状态也许并不符合调用者的要求。例如，令 string 表示一个空字符串或者令容器处于无序状态。因此，要想实现完整的恢复，错误处理程序仅仅生成 catch 从句处的有效值还不够，生成的这个值还必须符合应用程序本身的要求才行。

C++ 标准库为我们设计异常安全的程序组件提供了一套通用的概念框架，标准库为它的操作提供下述保障之一：

- 对所有操作的基本保障（basic guarantee）：维护所有对象的基本不变式，确保内存等系统资源不会泄漏。特别是，所有内置类型和标准库类型的基本不变式都确保我们可以在每个标准库操作之后销毁对象或者为它赋值（§ iso.17.6.3.1）。
- 对关键操作的强保障（strong guarantee）：除了提供基本保障之外，确保操作的结果是成功或者无任何效果。这一规则是为 push_back()、list 的单元素 insert()、uninitialized_copy() 等关键操作提供的。
- 对某些操作的不抛出保障（nothrow guarantee）：除了提供基本保障之外，确保某些操作不抛出异常。这一规则是为两个容器的 swap() 以及 pop_back() 等少数简单操作提供的。

基本保障和强保障的前提是：

- 用户提供的操作（比如赋值以及 swap() 函数）没有将容器元素置于无效状态。
- 用户提供的操作不产生资源泄漏。
- 析构函数不抛出异常（见 17.6.5.12 节）。

违反标准库约束（比如析构函数因抛出异常而退出）不仅在逻辑上等同于违反了基本语言规则（比如解引用空指针），产生的后果也很相似，常常意味着程序灾难。

基本保障和强保障都不允许资源泄漏，这对于无法承受资源泄漏的系统来说是非常必要的。尤其是，一个抛出异常的操作仅仅确保运算对象处于定义良好的状态还不够，它必须释放掉之前申请的全部资源。例如在异常抛出点，所有已分配的内存只能处于两种状态：要么已被释放掉，要么属于某个对象并且保证将来被正确地释放掉。例如：

```
void f(int i)
{
    int* p = new int[10];
    // ...
    if (i<0) {
```

```
            delete[] p;     // 在抛出异常前释放掉，不到将泄露
            throw Bad();
        }
        // ...
    }
```

谨记内存不是唯一一种可能泄漏的资源。我们通常把从系统某处申请并且将（显式地或者隐式地）归还回去的东西统称为资源。文件、锁、网络连接和线程都是系统资源。函数在抛出异常前必须释放这些资源或者把它们移交给其他资源句柄。

C++语言中关于部分构造和析构的规则确保构造子对象和成员时抛出的异常能被正确地处理，而无须来自标准库代码的特殊关注（见17.2.3节）。这一规则是所有异常处理技术的基础。

一般情况下，我们认为每个能抛出异常的函数迟早都会抛出异常。因此，我们必须精心组织自己的代码，以防迷失在乱糟糟的控制流和脆弱的数据结构中。当分析含有潜在错误的代码时，最理想的情况是代码简单、高度结构化且“格式化”。13.6节有一个符合这种要求的实际例子。

13.3 资源管理

当函数请求某种资源时，也就是说，当它打开文件、从自由存储分配一些内存或者请求一个互斥锁时，系统常常要求这些资源能在未来某个时刻被正确地释放掉。所谓“正确地释放掉”是指函数应该在返回它的调用者之前释放掉它请求的资源。例如：

```
void use_file(const char* fn)   // 最初正确的代码
{
    FILE* f = fopen(fn,"r");

    // ... 使用 f ...

    fclose(f);
}
```

这段代码看起来没什么问题，实则暗藏风险。假设在调用了fopen()之后且尚未调用fclose()的某个时刻程序出错了，异常将导致use_file()直接退出，并且再也不会执行fclose()。在不支持异常处理的编程语言中，这种情况时有发生。例如，C语言标准库函数longjmp()就会造成上述问题。一条普通的return语句也可能造成在未关闭f的情况下就退出use_file()。

要想让use_file()可以容忍上述错误，一种初步的解决方案是：

```
void use_file(const char* fn)   // 条理的代码
{
    FILE* f = fopen(fn,"r");
    try {
        // ... 使用 f ...
    }
    catch (...) {           // 捕获所有可能的异常
        fclose(f);
        throw;
    }
    fclose(f);
}
```

使用了文件资源的代码放置在 try 块中，程序捕获所有异常、关闭文件，然后重新抛出异常。

上述代码的问题是啰嗦冗长且潜在的开销会比较昂贵。更糟糕的是，一旦程序需要请求和释放几种资源，代码的复杂性会急剧增长。幸运的是，我们可以选择更聪明的解决方案来解决该问题。它的一般形式是：

```
void acquire()
{
    // 请求资源 1
    // ...
    // 请求资源 n

    // 使用资源 ...

    // 释放资源 n
    // ...
    // 释放资源 1
}
```

通常情况下，释放资源的顺序应该与请求资源的顺序相反，这非常类似于构造函数创建对象以及析构函数销毁对象的行为。因此，我们可以用含有构造函数和析构函数的类的对象来处理请求和释放资源的问题。例如，我们可以定义 File_ptr，它的行为与 FILE* 类似：

```
class File_ptr {
    FILE* p;
public:
    File_ptr(const char* n, const char* a)    // 打开文件 n
        : p{fopen(n,a)}
    {
        if (p==nullptr) throw runtime_error{"File_ptr: Can't open file"};
    }

    File_ptr(const string& n, const char* a)  // 打开文件 n
        :File_ptr{n.c_str(),a}
    {}

    explicit File_ptr(FILE* pp)               // 假定 pp 的所有权
        :p{pp}
    {
        if (p==nullptr) throw runtime_error{"File_ptr: nullptr"};
    }

    // ... 适当的移动和拷贝操作 ...

    ~File_ptr() { fclose(p); }

    operator FILE*() { return p; }
};
```

我们可以用 FILE* 构造 File_ptr，也可以用提供给 fopen() 的参数构造 File_ptr。不管怎样，File_ptr 对象都将在它的作用域末尾被销毁，并由它的析构函数负责关闭文件。如果 File_ptr 无法打开文件，它会抛出一个异常，这样就无须每次使用该文件句柄都检测是否是 nullptr 了。我们的函数简化为如下形式：

```
void use_file(const char* fn)
{
    File_ptr f(fn,"r");
    // ... 使用 f ...
}
```

不论函数是正常退出还是因为抛出异常退出，系统都会调用析构函数。也就是说，异常处理机制使得我们可以把错误处理代码从核心算法中独立出来。剩下的代码得到了简化，与原来的形式相比出错的概率也降低了。

这种使用局部对象管理资源的技术通常称为“资源获取即初始化”（RAII，见5.2节）。这是一种比较通用的技术，其前提是具备了构造函数和析构函数的属性，并且这两个函数与错误处理机制可以有机地融合在一起。

有一种意见认为编写一个“句柄类”（RAII类）过于繁琐，更好的解决方案是为catch(...)动作提供一种更优的语法形式。然而这种想法可能根本行不通，因为如果这样的话，我们就必须留意遇到的每一个资源请求点（在一个大一点的程序中通常有几十上百处），并且需要时刻谨记“捕获并且修正”潜在错误。与之相比句柄类的方式显然更优，因为我们只需要写一次句柄类就可以了。

对于一个对象来说，只有当它的构造函数完成了，我们才认为该对象创建成功了。之后栈展开（见13.5.1节）会为对象调用析构函数。如果对象由若干个子对象组成，则先构造每个子对象，再构造该对象本身；数组的情况与之类似，先构造每个元素，再构造数组（在展开时只销毁完整构造的元素）。

构造函数力争完整和正确地构造它的对象。如果无法达成，则一个好的构造函数会尽可能地把系统状态恢复到创建对象之前的模样。理想状态下，设计良好的构造函数不会让它的对象处于某种“半构造的”状态。要想实现这一目标，我们只需对类的成员应用RAII技术就可以了。

假设有一个类X，它的构造函数负责请求两种资源：文件x和互斥量y（见5.3.4节）。这些请求有可能会失败并抛出异常。对于X的构造函数来说，当它结束的时候，既不能只请求文件而未请求互斥量，也不允许只请求互斥量而未请求文件。另外，在实现上述目标的同时还应该避免增加程序员的编程负担。我们用File_ptr和std::unique_lock这两个类的对象（见5.3.4节）表示请求到的资源。此时，我们可以把请求某项资源的工作转换成初始化表示该资源的局部对象：

```
class Locked_file_handle {
    File_ptr p;
    unique_lock<mutex> lck;
public:
    X(const char* file, mutex& m)
        : p{file,"rw"},          // 请求“file”
          lck{m}                 // 请求“m”
    {
    // ...
};
```

就像处理局部对象时一样，由系统负责记录资源的来龙去脉，用户无须为此烦恼。例如，如果在构造p但是尚未构造lck的时刻发生了异常，则程序将调用p的析构函数，但是不会调用lck的析构函数。

这意味着只要我们使用了类似的资源请求模型，构造函数的作者就不用显式地编写异常处理代码了。

内存是最常用的资源，string、vector 以及其他标准容器使用 RAII 隐式地管理内存的请求和释放。与使用 new 和 delete 管理内存的方式相比，前者不但可以节约大量编程工作，还能避免很多错误。

当涉及对象的指针而非局部对象时，我们使用 unique_ptr 和 shared_ptr（见 5.2.1 节，34.3 节）避免资源泄漏。

13.3.1　finally

在之前的介绍中，我们把资源表示为一个包含析构函数的类的对象，这种做法可能会带来一些困扰。为了编写任意代码以在异常发生后执行清理工作，人们曾经设计了很多“最终的”语言概念。这些技术通常只能用于特定的场景，因此与 RAII 相比并不占优势，但如果确实需要的话，RAII 也可以支持此类技术。首先，我们定义一个类，它在析构函数中执行任意操作。

```
template<typename F>
struct Final_action {
    Final_action(F f): clean{f} {}
    ~Final_action() { clean(); }
    F clean;
};
```

我们通过构造函数的参数提供“最终操作”。

接下来定义一个函数，它可以方便地推断某个操作的类型：

```
template<class F>
Final_action<F> finally(F f)
{
    return Final_action<F>(f);
}
```

最后，我们检验 finally() 的效果：

```
void test()
    // 处理非常规的资源请求任务
    // 该代码证明我们可以在其中嵌入任意操作
{
    int* p = new int{7};                        // 其实应该使用 unique_ptr（见 5.2 节）
    int* buf = (int*)malloc(100*sizeof(int));   // C 风格的资源请求

    auto act1 = finally([&]{    delete p;
                                free(buf);      // C 风格的资源释放
                                cout<< "Goodby, Cruel world!\n";
                           }
                        );

    int var = 0;
    cout << "var = " << var << '\n';

    // 最套的块：
    {
        var = 1;
        auto act2 = finally([&]{ cout<< "finally!\n"; var=7; });
```

```
        cout << "var = " << var << '\n';
    } // 调用 act2

    cout << "var = " << var << '\n';
} // 调用 act1
```

上述代码输出：

```
var = 0
var = 1
finally!
var = 7
Goodby, Cruel world!
```

此外，p 和 buf 请求和指向的内存区域也被正确地删除（delete）和清空（free()）掉了。

在日常生活中，保镖应该与被保护的对象寸步不离，在程序中也是如此。以资源管理为例，我们不妨仔细想想到底什么是资源，以及在资源的作用域末尾应该做什么。与用于资源句柄的 RAII 相比，finally() 与其操作的资源之间仍然是一种点对点的隐式联系，但它已经比在块中随处散置清理代码的做法强多了。

基本上，finally() 对于块的作用与 for 语句中递增部分的作用类似（见 9.5.2 节）：它在块一开始的地方就指定了最终要执行的操作，这么做不仅易于程序员阅读，而且从说明的角度来看它本来就应该在这里。它告诉程序当前作用域结束时应该做什么，这样程序员就不用时时留意控制线程可能在何处退出，并且在每处都编写对应的代码了。

13.4 强制不变式

如果函数的前置条件不满足（见 12.4 节），则该函数无法正确执行它的任务。类似地，如果一个构造函数不能建立它的类的不变式（见 2.4.3.2 节，17.2.1 节），则该对象是不可用的。在这类情况下，我通常会令程序抛出异常。然而对于有的程序来说，抛出异常是不可接受的行为（见 13.1.5 节），而且程序员对于如何处理前置条件不满足的情况（或者其他类似情况）也会有不同的见解：

- 别让此类情况发生：函数的调用者应该确保前置条件满足，如果调用者没能做到这一点，则将产生错误的结果。通过改进设计、调试和测试，我们最终将从系统中消除所有这些错误。
- 终止程序：违反前置条件是一种严重的设计错误，一旦发生此类错误，程序就不应该继续执行了。系统有望从其组件的错误中恢复过来。通过改进设计、调试和测试，我们最终有可能从系统中消除这些错误。

我们该如何在上述两种策略中做出选择呢？第一种策略通常与性能有关：系统地检查前置条件可能会导致对逻辑上非必要条件的重复检测（例如，如果调用者已经验证过数据的合法性，则在上千次函数调用中进行数以百万计的检测从逻辑上来看就是冗余的）。性能方面的代价有可能非常巨大，为了获得在性能方面的提升，即使在测试时遇到再多的程序崩溃也是值得的。显然，这么做的意义在于你假定最终可以从系统中排除掉所有违反前置条件的情况。某些系统的控制权完全掌握在单一的组织手中，对于它们来说，完全有可能实现上述目标。

在其他一些系统中，不太可能耗费很长的时间从前置条件失效的错误中完全恢复过来。换句话说，要想确保完全恢复，系统的设计和实现就会变得异常复杂，根本不能接受。另一

方面，直接终止程序是个不错的选择。例如，如果我们很容易就可以换一组不会产生错误的输入数据和参数来重新运行程序，那么终止当前的错误程序就完全是合情合理的。有些分布式系统适用于该策略（终止的程序只是完整系统的一部分），我们写的很多只为自己使用的小程序也是如此。

在编程实践中，很多系统把异常和上述两种策略结合在一起使用。它们的共同点是都承认应该定义并遵守前置条件，区别在于对强制约束的具体方式和是否应该从错误中恢复的观点不同。根据是否恢复进行划分，程序的结构可能完全是两个样子。在大多数系统中，我们抛出某些异常时根本不需要恢复。例如，我在编写程序时抛出的某些异常仅仅是为了确保在终止程序或者重启某项任务前错误被记录在日志中，又或是为了给用户提供一条描述错误的信息（例如，在 main() 函数的 catch(...) 子句中）。

有很多技术可用于检查预置的条件和不变式。如果我们希望对检查的原因保持中立，则常使用断言（assertion，简写为 assert）。断言是一个逻辑表达式，我们假定断言的值为 true。然而，断言绝不仅仅是一条注释，我们还需要注明一旦它的值为 false 时应该做什么。显然在大量系统中，对于断言有各种各样的需求：

- 我们需要在编译时断言（由编译器求值）和运行时断言（在运行时求值）中做出选择。
- 对于运行时断言我们需要选择处理的方式：抛出异常、终止程序还是直接忽略。
- 除非某些逻辑条件为 true，否则不应生成代码。例如，除非逻辑条件为 true，否则不应对某些运行时断言求值。通常情况下，所谓逻辑条件是指某些类似于调试标识、检查级别或者断言选择范围之类的东西。
- 断言应该易于编写（因为断言常常具有通用性，会用在很多地方）。

不是每个系统都有上面的全部需求，当然也无须处理每一种需求。

C++ 标准提供了两种简单的机制：

- 在 <cassert> 中，标准库提供了 assert(A) 宏。当且仅当未定义宏 NDEBUG（非调试）时（见 12.6.2 节），它在运行时检查断言 A。一旦断言失败，编译器将输出一条错误信息并终止程序。其中，输出的错误信息包含失败的断言、源文件的名字以及行号等。
- C++ 语言使用 static_assert(A,message) 在编译时无条件检查断言 A（见 2.4.3.3 节）。一旦断言失败，编译器将输出 message 以及编译错误信息。

如果在某些情况下 assert() 和 static_assert() 不适用，我们也可以使用普通的代码进行检查。例如：

```
void f(int n)
    // n 应该在 [1:max) 之间
{
    if (2<debug_level && (n<=0 || max<n)
        throw Assert_error("range problem");
    // ...
}
```

然而，使用这样的“普通代码”会使得要检查的对象不太明显。我们是在：

- 对我们的测试条件求值吗？（是，在 2<debug_level 的部分。）
- 对某个条件求值吗？我们希望该条件对某些调用为真，对其他调用不为真？（不，因为我们抛出了异常。除非有人愿意把异常当成一种返回机制，否则答案是否定的；见 13.1.4.2 节。）

- 检查一个应该永远成立的前置条件吗？（是，此处的异常仅仅是我们的回应而已。）

更糟糕的是，检查前置条件（或者不变式）的代码经常散落在其他代码中，既难定位又容易出错。我们想要的是一种检查断言的易于识别的机制，接下来我们就介绍这种机制。它可能显得有一点繁琐，但是它既能表示大量的断言，又能涵盖很多种对于断言失败的响应。首先，我定义一些机制，它们负责决定何时检查以及一旦失败应该做些什么：

```
namespace Assert {
    enum class Mode { throw_, terminate_, ignore_ };
    constexpr Mode current_mode = CURRENT_MODE;
    constexpr int current_level = CURRENT_LEVEL;
    constexpr int default_level = 1;

    constexpr bool level(int n)
        { return n<=current_level; }

    struct Error : runtime_error {
        Error(const string& p) :runtime_error(p) {}
    };

    // ...
}
```

上述代码的主要目的是检查断言的“层级”在何时不高于 current_level。如果断言失败，则用 current_mode 在三种响应模式中选择一种。我们的目的是在决定做什么之前对任何断言都不生成代码，因此 current_level 和 current_mode 被设置成常量。CURRENT_MODE 和 CURRENT_LEVEL 可以看成是在程序的编译环境中设置的编译选项。

程序员使用 Assert::dynamic() 设置断言：

```
namespace Assert {
    // ...

    string compose(const char* file, int line, const string& message)
        // 组合生成包含文件名和行号的消息
    {
        ostringstream os ("(");
        os << file << "," << line << "):" << message;
        return os.str();
    }

    template<bool condition =level(default_level), class Except = Error>
    void dynamic(bool assertion, const string& message ="Assert::dynamic failed")
    {
        if (assertion)
            return;
        if (current_mode == Assert_mode::throw_)
            throw Except(message);
        if (current_mode == Assert_mode::terminate_)
            std::terminate();
    }

    template<>
    void dynamic<false,Error>(bool, const string&)    // 什么也不做
    {
    }
```

```
    void dynamic(bool b, const string& s)          // 默认操作
    {
        dynamic<true,Error>(b,s);
    }

    void dynamic(bool b)                           // 默认消息
    {
        dynamic<true,Error>(b);
    }
}
```

我选用 Assert::dynamic 这个名字（意思是“运行时求值”）来与 static_assert 作为对比（意思是“编译时求值”，见 2.4.3.3 节）。

我们还可以通过其他一些实现技巧来使生成的代码量最少。如果对灵活性要求较高的话，也可以在运行时做更多测试的工作。上面这个 Assert 并不是标准的一部分，我把它列在此处主要是为了讲解有关的问题和技术。要想实现一种放之四海而皆准的断言机制，它需要满足的要求会非常非常多。

Assert::dynamic 的用法如下：

```
void f(int n)
    // n 应该在 [1:max) 之间
{
    Assert::dynamic<Assert::level(2),Assert::Error>(
        (n<=0 || max<n), Assert::compose(__FILE__,__LINE__,"range problem");
    // ...
}
```

其中，__FILE__ 和 __LINE__ 是宏，它们会在源代码的对应位置展开（见 12.6.2 节）。我无法把它们放置在 Assert 的实现中以对用户隐藏其细节。

Assert::Error 是默认的异常，因此我们无须显式地提及它。类似地，如果我们想使用默认的断言级别，也不需要显式地指出来：

```
void f(int n)
    // n 应该在 [1:max) 之间
{
    Assert::dynamic((n<=0 || max<n),Assert::compose(__FILE__,__LINE__,"range problem");
    // ...
}
```

我不认为过分纠结于表示断言的文本长度会有什么帮助，但是通过使用名字空间指示（见 14.2.3 节）和默认消息，我们可以令文本长度最小化：

```
void f(int n)
    // n 应该在 [1:max) 之间
{
    dynamic(n<=0||max<n);
    // ...
}
```

我们可以通过构建选项（例如，控制条件编译）或程序代码选项控制要执行的测试以及对测试的响应。因此，你既可以得到一个执行全部测试的程序版本以便对它进行测试，也可以得到一个几乎不做任何测试的产品级版本。

我个人的习惯是在程序的最终发布版中至少留一些测试功能。例如，保留 Assert 的作用之一是所有标记为 0 级的断言都会被检查。在持续开发和维护一个大型程序的过程中，我

们很难穷尽全部漏洞。而且，即使所有事情都做得尽善尽美了，保留一些“清醒的检查”也有助于处理可能发生的硬件错误。

只有完整系统的最后的构建者有权决定某个断言的失败是否可以接受。库和可重用组件的作者无权无条件地终止程序。对于一般的库代码来说，最好通过抛出异常的方式报告错误。

与之前介绍的一样，析构函数不允许抛出异常，也不能在析构函数中使用抛出异常的Assert()。

13.5 抛出与捕获异常

本节从技术的角度出发介绍与异常有关的内容。

13.5.1 抛出异常

我们可以 throw 任意类型的异常，前提是它能被复制和移动。例如：

```
class No_copy {
    No_copy(const No_copy&) = delete;     // 禁止复制（见 17.6.4 节）
};

class My_error {
    // ...
};

void f(int n)
{
    switch (n) {
    case 0:   throw My_error{};          // OK
    case 1:   throw No_copy{};           // 错误：不允许复制 No_copy
    case 2:   throw My_error;            // 错误：My_error 是一种类型，而非一个对象
    }
}
```

能获的异常对象（见 13.5.2 节）从本质上来说就是被抛出的对象的一份拷贝（尽管我们允许优化器最小化拷贝过程），换句话说，throw x 用 x 初始化了一个 x 类型的临时变量。在我们最终捕获这个临时变量之前，还可能复制它好几次；异常从被调函数传回给上调函数，这个过程直到我们找到一个合适的异常处理程序才会停止。我们在某个 try 块的 catch 从句中使用异常类型来选择合适的处理程序。如果异常对象包含有数据部分，则这些数据通常用来生成错误信息，并且帮助程序从异常中恢复。异常从它的抛出点开始“向上”传递到处理程序的过程称为栈展开（stack unwinding）。当某个作用域结束时，系统会自动调用析构函数以确保每个完整构造的对象都能被正确地销毁掉。例如：

```
void f()
{
    string name {"Byron"};
    try {
        string s = "in";
        g();
    }
    catch (My_error) {
        // ...
    }
}
```

```
void g()
{
    string s = "excess";
    {
        string s = "or";
        h();
    }
}

void h()
{
    string s = "not";
    throw My_error{};
    string s2 = "at all";
}
```

当程序在 h() 中抛出异常后，所有已构造的 string 都按与构造时相反的顺序被依次销毁："not"、"or"、"excess" 和 "in"。但是这个列表中不包括 "at all"（控制线程根本就没有到达这句）和 "Byron"（不受影响）。

因为异常在被捕获前有可能被拷贝很多次，所以我们一般不会在异常中存放太大规模的数据。相对来说，含有少量数据的异常比较普遍。异常传播从语义上来理解类似于初始化，因此如果我们抛出的是含有移动语义类型的对象（如 string），则代价会显得不那么昂贵。某些常见的异常不携带任何信息，它们的类型名字本身就足以说明问题了，我们完全可以用这些名字来报告错误。例如：

```
struct Some_error { };

void fct()
{
    // ...
    if (something_wrong)
        throw Some_error{};
}
```

在标准库中定义了一个规模不大的异常类型层次体系（见 13.5.2 节），我们可以直接使用它，也可以把它作为基类。例如：

```
struct My_error2 : std::runtime_error {
    const char* what() const noexcept { return "My_error2"; }
};
```

runtime_error 和 out_of_range 等标准库异常类接受一个字符串作为其构造函数的参数，然后用虚函数 what() 把该字符串的内容用在别处。例如：

```
void g(int n)    // 抛出某个异常
{
    if (n)
        throw std::runtime_error{"I give up!"};
    else
        throw My_error2{};
}

void f(int n)    // 查看 g() 抛出了什么异常
{
    try {
        void g(n);
```

```
    }
    catch (std::exception& e) {
        cerr << e.what() << '\n';
    }
}
```

13.5.1.1 noexcept 函数

有的函数没有抛出异常，另外一些则永远不会抛出异常。当面对后一种情况时，我们可以把它声明成 noexcept 的。例如：

```
double compute(double) noexcept; // 不会有抛出异常
```

此时，compute() 就不会抛出任何异常了。

把函数声明成 noexcept 的不但有助于程序员推断程序逻辑，而且编译器也可以更好地优化程序。程序员无须再书写任何 try 从句（用于处理 noexcept 函数的错误），优化器也不用再为异常处理可能引发的控制路径变更烦心了。

然而，编译器和链接器并不能完整检查 noexcept 的真实性。一旦程序员“撒谎了”会怎么样呢？即，在 noexcept 函数内部抛出某个异常，但是直到该函数结束时程序都没有捕获这个异常。不管这种行为是故意为之也好，还是碰巧发生也罢，会导致什么情况呢？我们不妨考虑如下示例：

```
double compute(double x) noexcept;
{
    string s = "Courtney and Anya";
    vector<double> tmp(10);
    // ...
}
```

vector 构造函数有可能在获取它的 10 个 double 的内存时失败并抛出 std::bad_alloc。此时，程序将直接终止。它通过调用 std::terminate() 无条件终止执行（见 30.4.1.3 节）。在这个过程中，程序不会触及主调者的析构函数。至于是否会使用 throw 和 noexcept 之间作用域的析构函数（如 compute() 中的 s）则完全依赖于程序实现。程序仅有的行为就是中止执行，因此我们无法以任何方式依赖任何具体对象。一旦我们添加了 noexcept 说明符，就表明该处代码不会处理任何 throw 了。

13.5.1.2 noexcept 运算符

我们可以把函数声明成有条件的 noexcept，例如：

```
template<typename T>
void my_fct(T& x) noexcept(Is_pod<T>());
```

noexcept(Is_pod<T>()) 的含义是，如果谓词 Is_pod<T>() 是 true，则 my_fct 不会抛出异常；反之，如果 Is_pod<T>() 是 false，则 my_fct 有可能抛出异常。举个例子，如果 my_fct() 的作用是拷贝它的参数，则上述用法是有用的。原因是我们都知道拷贝 POD 肯定不会抛出异常，而其他类型（例如 string 或者 vector）就有可能了。

在 noexcept 说明中用到的谓语必须是常量表达式。普通的 noexcept 等价于 noexcept(true)。

标准库提供了很多类型谓词，这些谓词可用于表示函数可能抛出异常的条件（见 35.4 节）。

如果我们想用的谓词不能表示成类型谓词该怎么办呢？例如，假设是否抛出异常的条件是一次函数调用 f(x)，我们该如何处理呢？noexcept() 运算符接受一条表达式作为它的参

数，当编译器“知道”它不能抛出异常时，该运算符返回 true，否则返回 false。例如：

```
template<typename T>
void call_f(vector<T>& v) noexcept(noexcept(f(v[0])))
{
    for (auto x : v)
        f(x);
}
```

连续使用两个 noexcept 看起来有点繁琐，但毕竟 noexcept 是个比较特殊的运算符。

我们不会对 noexcept() 的运算对象求值，因此在上面的例子中，即使传给 call_f() 的是个空的 vector 也不会造成运行时错误。

noexcept(expr) 运算符不会细抠每个细节以决定 expr 是否会抛出异常。它只是粗略地查看 expr 中的每个操作，如果它们都对应求值为 true 的 noexcept 说明，则 expr 就为 true。noexcept(expr) 不会深入到 expr 的每个操作内部去检查其具体定义。

在关于容器的标准库操作中，条件 noexcept 说明和 noexcept() 运算符都很常用并且很重要。例如（§ iso.20.2.2）：

```
template<class T, size_t N>
void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));
```

13.5.1.3　异常说明

在老式的 C++ 代码中有一些异常说明（exception specification），例如：

```
void f(int) throw(Bad,Worse);  // 只能抛出 Bad 或者 Worse
void g(int) throw();           // 不允许抛出异常
```

空异常说明 throw() 的作用与 noexcept 等价（见 13.5.1.1 节）。即，如果抛出了异常，则程序终止。

非空异常说明（比如 throw(Bad,Worse)）的含义是，如果函数（此处是 f()）抛出了一个未提及的异常或者不能由参数项公有派生的异常，则程序将调用不可预期的异常处理程序（unexpected handler）。不可预期异常的缺省效果是终止程序（见 30.4.1.3 节）。非空 throw 说明很难使用，并且由于我们必须在运行时检查抛出的异常是否符合规定，因此它的代价也相当昂贵。总之，这项功能仍不完备，建议读者不要使用。

如果你想动态地检查抛出的是哪种异常，请使用 try 块。

13.5.2　捕获异常

考虑如下示例：

```
void f()
{
    try {
        throw E{};
    }
    catch(H) {
        // 何时到达此处呢？
    }
}
```

当满足下述条件之一时，系统会调用异常处理程序：

[1] 如果 H 与 E 的类型相同；

[2] 如果 H 是 E 的无歧义的公有基类；

[3] 如果 H 和 E 都是指针类型，并且它们所指的类型满足 [1] 或者 [2]；

[4] 如果 H 是引用类型，并且它所引用的类型满足 [1] 或者 [2]。

此外，我们可以在捕获异常所用的类型前加上 const，这种情况类似于函数参数的用法。这么做不会改变捕获的异常集合，它只是确保我们不会修改异常。

异常基本上在抛出的时候被拷贝（见 13.5 节）。具体实现可能运用很多种不同的策略存储和传输异常，但是无论如何都会确保有足够的内存空间使得 new 可以抛出一个内存耗尽异常 bad_alloc（见 11.2.3 节）。

请注意，我们可以通过引用的方式捕获异常。异常类型常常作为类层次的一部分以反映它们表示的错误之间的关系，详情可参见 13.5.2.3 节和 30.4.1.1 节。对于希望通过引用捕获异常的程序员来说，把异常类组织成类层次非常有用。

try 块和 catch 从句中的 {} 都是实实在在的作用域。因此，如果我们想在 try 语句的两个部分使用同一个名字，或者想在 try 块的外部使用某个名字，都必须把该名字声明在 try 块的外部。例如：

```
void g()
{
    int x1;

    try {
        int x2 = x1;
        // ...
    }
    catch (Error) {
        ++x1;       // OK
        ++x2;       // 错误：x2 不在作用域范围内
        int x3 = 7;
        // ...
    }
    catch(...) {
        ++x3;       // 错误：x3 不在作用域范围内
        // ...
    }

    ++x1;           // OK
    ++x2;           // 错误：x2 不在作用域范围内
    ++x3;           // 错误：x3 不在作用域范围内
}
```

“捕获全部”从句 catch(...) 将在 13.5.2.2 节介绍。

13.5.2.1 重新抛出

捕获一个异常之后，异常处理程序经常发现它自己无法完整地处理该错误。此时，异常处理程序先完成在局部能完成的任务，然后再次抛出异常。通过这种方式，错误就能被很好地处理了。甚至当处理错误所需的信息散落在程序的多个部分时，程序也可以协同多个处理程序共同完成恢复操作。例如：

```
void h()
{
    try {
        // ... 此处代码可能抛出异常 ...
    }
    catch (std::exception& err) {
```

```
            if (can_handle_it_completely) {
                // ... 处理异常 ...
                return;
            }
            else {
                // ... 尽力完成 ...
                throw;    // 重新抛出异常
            }
        }
    }
```

我们用不带运算对象的 throw 表示重新抛出。重新抛出可能发生在 catch 从句中，也可能发生在 catch 从句调用的某个函数中。如果在没有异常的情况下强行重新抛出，则系统会调用 std::terminate()（见 13.5.2.5 节）。对于这种情况，编译器能检测到其中一些并给出警告，但是无法确保对每个实例都能检测出。

重新抛出的异常就是一开始我们捕获的那个异常，而不会只是它的一部分（能作为 exception 访问的子对象）。例如，假设程序抛出了一个 out_of_range 异常，则 h() 会按照普通 exception 捕获它；而 throw 仍旧会抛出 out_of_range。假如我在程序中写的是 throw err; 而非 throw;，则异常就会产生切片现象（见 17.5.1.4 节），h() 的调用者将无法捕获到 out_of_range 异常。

13.5.2.2　捕获每个异常

在 <stdexcept> 中，标准库提供了一个规模较小的异常类层次，其基类是 exception（见 30.4.1.1 节）。例如：

```
void m()
{
    try {
        // ... 执行某些操作 ...
    }
    catch (std::exception& err) {        // 处理每个标准库异常
        // ... 清除 ...
        throw;
    }
}
```

这段代码会捕获全部标准库异常。然而，标准库异常只是异常类型的一个子集。因此，你无法通过使用 std::exception 捕获每一个异常。如果有人抛出了一个 int（当然这么做其实不太明智），或者抛出的是用户自定义层次中的一个异常，则它们无法被 std::exception 的处理程序捕获。

事实上，我们是需要处理每一种异常的。假设 m() 中遗留了某些指针处于其初始状态，我们就能在异常处理程序中为这些指针赋予适当的值。在函数中省略号 ... 表示“任意实参”（见 12.2.4 节），因此 catch(...) 的含义是“捕获任意异常”。例如：

```
void m()
{
    try {
        // ... 执行某些操作 ...
    }
    catch (...) {          // 处理每一个异常
        // ... 清除 ...
        throw;
    }
}
```

13.5.2.3 多异常处理程序

一个 try 块可以对应多个 catch 从句（异常处理程序）。因为派生的异常能被多种异常类型的处理程序捕获，所以 try 语句中异常处理程序的书写顺序显得非常重要。程序将依次尝试每段处理代码，例如：

```
void f()
{
    try {
        // ...
    }
    catch (std::ios_base::failure) {
        // ... 处理各种输入输出错误（见 30.4.1.1 节）...
    }
    catch (std::exception& e) {
        // ... 处理各种标准库异常（见 30.4.1.1 节）...
    }
    catch (...) {
        // ... 处理其他异常（见 13.5.2.2 节）...
    }
}
```

编译器了解类层次的情况，因此它可以发现并报告很多逻辑错误。例如：

```
void g()
{
    try {
        // ...
    }
    catch (...) {
        // ... 处理各种错误（见 13.5.2.2 节）...
    }
    catch (std::exception& e) {
        // ... 处理各种标准库异常（见 30.4.1.1 节）...
    }
    catch (std::bad_cast) {
        // ... 处理动态类型转换错误（见 22.2.1 节）...
    }
}
```

在这段代码中，系统永远都不会考虑 exception。即使我们删掉“捕获全部”的处理程序，程序也仍然有错，因为 bad_cast 是从 exception 中派生出来的，所以它永远不会执行。异常类型与 catch 从句的匹配过程是一种快速的运行时操作，其机理与编译时的重载解析并不一样。

13.5.2.4 函数 try 块

函数体可以是一个 try 块，例如：

```
int main()
try
{
    // ... 执行实际操作 ...
}
catch (...) {
    // ... 处理异常 ...
}
```

对于大多数函数来说，使用函数 try 块仅仅是为了方便。然而，try 块允许我们在构造函数中

处理基类或成员初始化器抛出的异常（见 17.4 节）。默认情况下，如果基类或成员初始化器抛出了一个异常，则该异常将传递到调用了该成员的类的构造函数的地方。不过，我们也可以把构造函数的函数体（包括成员初始化器列表在内）放在一个 try 块中，这样该构造函数就能自己捕获异常了。例如：

```
class X {
    vector<int> vi;
    vector<string> vs;

    // ...
public:
    X(int,int);
    // ...
};

X::X(int sz1, int sz2)
try
    :vi(sz1),  // 用 sz1 个 int 构造 vi
     vs(sz2),  // 用 sz2 个 string 构造 vs
{
    // ...
}
catch (std::exception& err) { // 捕获 vi 和 vs 抛出的异常
    // ...
}
```

这样我们就能捕获成员构造函数抛出的异常了。类似地，我们可以在析构函数中捕获成员析构函数抛出的异常（尽管析构函数永远也不会抛出异常）。但是，我们无法“修复”对象并且像异常从未发生一样正常返回：成员构造函数的异常意味着成员的状态应该是无效的。同样，对于其他成员对象来说，它们根本不会被构造，或者它们的析构函数已经位于栈展开过程中了。

对于构造函数和析构函数的 try 块来说，我们在其 catch 从句中最应该做的事情就是抛出异常。默认的操作是当我们到达 catch 从句的尾端时重新抛出一开始的那个异常（§ iso.15.3）。

这些约束对于普通函数的 try 块并不适用。

13.5.2.5　终止

在有些情况下最好不要使用异常处理，基本的原则是：

- 处理异常的时候不要抛出异常。
- 不要抛出一个无法捕获的异常。

如果异常处理发现你违反了上述原则，程序就会终止。

如果你试图在同一时刻令两个异常都处于活跃状态（在同一线程中，该用法被禁止），那么系统就不知道该处理哪个异常了：是你刚刚抛出的异常，还是它已经准备处理的异常？请注意，我们一旦进入到 catch 从句中，就表示准备处理异常了。重新抛出异常（见 13.5.2.1 节）和在 catch 从句中抛出一个新异常都被认为是原来的异常被处理之后的新的抛出动作。你也可以在析构函数中抛出异常（甚至在栈展开期间），前提是在离开析构函数之前必须捕获它。

terminate() 的触发条件是（§ iso.15.5.1）：

- 当没有合适的处理程序可以处理已抛出的异常时；
- 当 noexcept 函数结束时仍然留有 throw；
- 当栈展开期间的析构函数结束时仍然留有 throw；
- 当传播异常的代码（比如拷贝构造函数）结束时仍然留有 throw；
- 当有人试图在当前没有处理异常的情况下重新抛出一个异常（throw;）；
- 当静态分配的或者线程局部的对象的析构函数结束时仍然留有 throw；
- 当静态分配的或者线程局部的对象的初始化器结束时仍然留有 throw；
- 当作为 atexit() 函数调用的函数结束时仍然留有 throw。

在上述情况下，系统都将调用 std::terminate()。此外，如果用户觉得实在找不到其他办法了，也可以主动调用 terminate()。

“结束时留有 throw”的意思是在某处抛出了异常，但是该异常未被捕获，因而运行时系统试图把该异常从函数传递给函数的调用者。

默认情况下，terminate() 会调用 abort()（见 15.4.3 节）。对于大多数用户来说，这个默认选项也是最佳选择，特别在调试期间更是如此。如果用户不接受该选项，也可以通过调用 <exception> 中的 std::set_terminate() 提供一个终止处理程序（terminate handler）：

```
using terminate_handler = void(*)();      // 等于 <exception>

[[noreturn]] void my_handler()            // 终止处理程序无法返回任何值
{
    // 自行处理终止
}

void dangerous()    // 非常危险！
{
    terminate_handler old = set_terminate(my_handler);
    // ...
    set_terminate(old); // 恢复旧的终止处理程序
}
```

返回值是提供给 set_terminate() 的函数。

举个例子，终止处理程序可用于中断一个处理过程或者重新初始化一个系统。terminate() 的出发点是当异常处理机制实现错误恢复策略时，必须采取更极端的措施；是时候切换到新的容错策略了。一旦进入了终止处理程序，任何关于程序数据结构的假设都不再成立，我们无法保证它们不被干扰和损坏。甚至用 cerr 输出一条错误消息都可能出错。同样，就像上面的 dangerous() 说的那样，它并非异常安全的。set_terminate(old) 前面的一个 throw 甚至是一个 return 都有可能把 my_handler 置于意料之外的境地。如果你一定要改变 terminate() 的用法，记得使用 RAII（见 13.3 节）。

终止处理程序无法返回它的调用者，如果它试图这么做的话，terminate() 将调用 abort()。

abort() 表示程序非正常退出。除此之外，我们还可以用函数 exit() 退出程序并且返回一个值，这个值用来告诉周围的系统当前的退出动作是正常的还是非正常的（见 15.4.3 节）。

当程序因未捕获的异常终止时，是否调用析构函数是依赖于具体实现的。在有的系统中不调用析构函数，其目的是让程序从调试状态中恢复过来；而在另外一些系统中，寻找异常处理程序的同时很难不调用析构函数。

如果在发生未捕获的异常时你希望程序能执行某些清理工作，你可以在真正在意的处理程序之外再向 main() 添加一个捕获全部的处理程序（见 13.5.2.2 节）。例如：

```
int main()
try {
    // ...
}
catch (const My_error& err) {
    // ... 处理错误 ...
}
catch (const std::range_error&)
{
    cerr << "range error: Not again!\n";
}
catch (const std::bad_alloc&)
{
    cerr << "new ran out of memory\n";
}
catch (...) {
    // ...
}
```

这段程序将捕获几乎全部异常，只有名字空间和线程局部对象的构造和析构抛出的异常除外（见 13.5.3 节）。任何措施都无法捕获在名字空间和线程局部对象的初始化及析构过程中抛出的异常。这恰恰是我们应该尽量避免使用全局变量的另一个原因。

捕获异常之后，通常我们无法获知它是在哪个确切的点被抛出的。与调试器所了解的程序状态相比，异常机制存在某种程度的信息丢失现象。因此，在某些 C++ 开发环境中，对特定的程序和人来说，如果程序并未设计恢复机制，可能程序不捕获异常为好。

我们可以把 throw 的位置整合到抛出异常的信息中，Assert（见 13.4 节）就是一个示例。

13.5.3　异常与线程

如果异常在线程中未被捕获（见 5.3.1 节、42.2 节），系统将调用 std::terminate()（见 13.5.2.5 节）。因此，如果我们不希望整个程序因线程中的一个错误而终止执行，就必须捕获全部错误并且以某种方式把它们报告给程序中对线程执行结果感兴趣的部分。“捕获全部” catch(...)（见 13.5.2.2 节）有助于实现该目标。

我们可以用标准库函数 current_exception()（见 30.4.1.2 节）把某一线程的异常传递给另一线程的处理程序。例如：

```
try {
    // ... 执行某些操作 ...
}
catch(...) {
    prom.set_exception(current_exception());
}
```

这是 packaged_task 处理用户代码异常的一项基本技术（见 5.3.5.2 节）。

13.6　vector 的实现

标准库 vector 为我们编写异常安全的代码提供了非常好的技术示例，它的实现覆盖了很多环境和解决方案中都会涉及的问题。

显然，vector的实现所依赖的语言功能有很多都是用于支持类的实现和使用的。如果你对C++的类和模板还不熟悉，那么最好先阅读第16、25和26章，然后再来学习本节的内容。本章之前的部分展示了一些有关异常的代码片段，但是要想对C++的异常机制有深入的了解，仅有这些片段还远远不够。

编写异常安全的代码所需的基本工具包括：

- try块（见13.5节）。
- 支持“资源获取即初始化”的技术（见13.3节）。

程序员应该遵循的原则包括：

- 不要随意丢弃信息，直到我们确认有东西可以替代它为止。
- 在抛出或者重抛异常时确保对象处于有效状态。

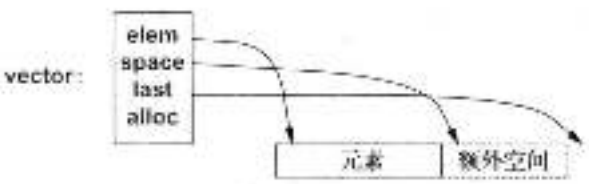
遵循这样的原则，我们就可以确保程序从错误状态中恢复过来。在实践中有时我们会在执行上述原则时遇到一些困难，主要的原因是某些貌似无害的操作（例如<、=和sort()）其实也会抛出异常。很多经验只有通过不断编程实践才能获得。

当你编写标准库文件时，一定要尽量提供强力的异常安全保障（见13.2节）；与之相比，编写特定程序时对异常安全的关注就会少一些了。例如，假设我在编写一个仅供自己使用的简单的数据分析程序，我会希望当发生内存耗尽的问题时直接终止程序。

正确性和基本异常安全紧密相关。尤其是定义和检查不变式（见13.4节）等提供基本异常安全的技术与那些令程序简洁正确的技术非常相似。它的出发点是提供基本安全保障（见13.2节）甚至是强安全保障的代价应该尽量控制在有限的范围内。

13.6.1 一个简单的vector

vector的典型实现（见4.4.1节、31.4节）应该包含一个句柄，它容纳指向首元素的指针、尾元素下一位置的指针以及已分配空间尾后位置的指针（或者表示相同信息的指针及偏移量，见31.2.1节）：



此外，它还包含一个分配器（此处是alloc），vector可以通过它其元素获取内存空间。默认的分配器（见34.4.1节）用new和delete获取及释放内存。

下面是vector的一个简单声明，主要用于讨论异常安全的要件以及如何避免资源泄漏：

```
template<class T, class A = allocator<T>>
class vector {
private:
    T* elem;        // 分配空间的开始
    T* space;       // 元素序列末尾，可扩展空间的开始
    T* last;        // 分配空间的末尾
    A alloc;        // 分配器
public:
    using size_type = unsigned int;                 // 表示 vector 尺寸的数据类型

    explicit vector(size_type n, const T& val = T(), const A& = A());
```

```
    vector(const vector& a);                    // 拷贝构造函数
    vector& operator=(const vector& a);         // 拷贝赋值

    vector(vector&& a);                         // 移动构造函数
    vector& operator=(vector&& a);              // 移动赋值

    ~vector();

    size_type size() const { return space-elem; }
    size_type capacity() const { return last-elem; }
    void reserve(size_type n);                  // 增加容量到 n

    void resize(size_type n, const T& = {});    // 改变 n 的大小
    void push_back(const T&);                   // 在末尾增加元素

    // ...
};
```

先来看构造函数的一个简单实现，它负责把 vector 的 n 个元素初始化成 val：

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a) // 警告：不完整的实现
    :alloc{a}                       // 拷贝分配器
{
    elem = alloc.allocate(n);       // 为元素分配内存（见 34.4 节）
    space = last = elem+n;
    for (T* p = elem; p!=last; ++p)
        a.construct(p,val);         // 在 *p 构造 val 的拷贝（见 34.4 节）
}
```

在这段代码中有两个地方可能引发异常：

［1］ 如果内存不足，allocate() 可能抛出异常。

［2］ 如果 T 的拷贝构造函数无法拷贝 val，那么它会抛出异常。

那么分配器的拷贝呢？我们可以假设它抛出异常，但是事实上标准库对此有特殊要求，它是不允许抛出异常的（§ iso.17.6.3.5）。无论如何代码已经是这样了，因此这一点不在我们的考虑范围之内。

在上述两种抛出异常的情况中，程序都还没有创建 vector 的对象，因此不会调用 vector 的析构函数（见 13.3 节）。

当 allocate() 失败时，throw 退出的时候还没有获取任何资源，也就不会有错误发生。

当 T 的拷贝构造函数失败时，已经获取了一些内存，所以我们必须释放掉这些内存以避免泄漏。更糟糕的是，T 的拷贝构造函数有可能在构造了一部分元素之后抛出异常，这些对象所拥有的资源可能会泄漏。

为了处理上述问题，我们应该随时跟踪哪些元素已经被构造，并且一旦发现错误立即销毁这些元素：

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)      // 详细说明
    :alloc{a}                       // 拷贝分配器
{
    elem = alloc.allocate(n);       // 分配元素的空间

    iterator p;
```

```
        try {
            iterator end = elem+n;
            for (p=elem; p!=end; ++p)
                alloc.construct(p,val);          // 构造元素（见 34.4 节）
            last = space = p;
        }
        catch (...) {
            for (iterator q = elem; q!=p; ++q)
                alloc.destroy(q);                // 销毁已构造的元素
            alloc.deallocate(elem,n);            // 释放内存
            throw;                               // 重新抛出
        }
    }
```

请注意，p 的声明应该位于 try 块之外；否则，我们在 try 部分和 catch 从句都无法访问它。

此处的代价主要是 try 块的开销。在一个好的 C++ 实现中，这部分开销与分配空间和初始化元素所需的开销相比微不足道。对于进入 try 块时产生开销的实现来说，最好在 try 显式地处理（非常常见）空 vector 之前添加一条测试语句 if(n)。

这个构造函数的主要部分类似于 std::uninitializ ed_fill() 的实现：

```
template<class For, class T>
void uninitialized_fill(For beg, For end, const T& x)
{
    For p;
    try {
        for (p=beg; p!=end; ++p)
            ::new(static_cast<void*>(&*p)) T(x);     // 在 *p 中构造 x 的拷贝（见 11.2.4 节）
    }
    catch (...) {
        for (For q = beg; q!=p; ++q)
            (&*q)->~T();                             // 销毁元素（见 11.2.4 节）
        throw;                                       // 重新抛出（见 13.5.2.1 节）
    }
}
```

构造 &*p 主要是为了兼顾非指针的迭代器。在这种情况下，我们需要获取一个解引用的元素的地址来得到指针。显式的全局 ::new 和显式类型转换成 void* 确保我们可以用标准库函数（见 17.2.4 节）而非用户自定义的用于 T* 的 operator new() 调用构造函数。在 vector 的构造函数中，对 alloc.construct() 的调用实际上起到的就是放置式 new 的作用。类似地，alloc.destroy() 调用也可以隐藏显式的析构过程（像 (&*q)->~T() 一样）。这种代码执行的是非常底层的操作，我们很难为它写出真正通用的代码。

幸运的是，我们无须发明 uninitialized_fill()，也不用亲自动手实现它，因为标准库已经为我们提供了（见 32.5.6 节）。对于一般的程序来说，初始化操作的状态只能是以下三种之一：初始化完成、初始化了每个元素、初始化失败但任何元素都没有初始化。因此，标准库提供了 uninitialized_fill()、uninitialized_fill_n() 和 uninitialized_copy()（见 32.5.6 节）作为强力保障（见 13.2 节）。

uninitialized_fill() 算法不处理元素析构器或者迭代器操作抛出的异常（见 32.5.6 节），否则代价将极其昂贵，以至于根本就不可能实现。

uninitialized_fill() 算法可应用于很多种不同的序列，它接受一个前向迭代器（见 33.1.2 节），但是无法确保元素以与构造相反的顺序被销毁。

我们可以用 uninitialized_fill() 简化构造函数：

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)        // 还是显得有点繁琐
    :alloc(a)                                                     // 拷贝分配器
{
    elem = alloc.allocate(n);                                     // 分配元素的空间
    try {
        uninitialized_fill(elem,elem+n,val); // 拷贝元素
        space = last = elem+n;
    }
    catch (...) {
        alloc.deallocate(elem,n);                                 // 释放内存
        throw;                                                    // 重新抛出
    }
}
```

与上一版的构造函数相比，这一版提升非常明显，但是下一节将进一步简化它。

这个构造函数重新抛出了它捕获的异常。它的目的是让 vector 对于异常来说保持透明，这样用户就可以确定问题的真正原因到底是什么了。所有标准库容器都具有该属性。异常的透明性常用于模板或者某些“瘦”软件层次中，这一点与系统的主要部分（“模块”）正好相反，后者倾向于由自己对所有异常负责。换句话说，这类模块的实现者需要罗列出该模块可能抛出的每一种异常。要想做到这一点，通常需要把异常组织成层次（见 13.5.2 节），并使用 catch(...)（见 13.5.2.2 节）。

13.6.2　显式地表示内存

经验显示，用显式的 try 块书写异常安全代码的难度要远远超出人们的预期。事实上，存在一种更简单的做法：“资源获取即初始化”技术（见 13.3 节）不仅可以减少代码量，还能使格式更加规范。在此例中，vector 所需的关键资源是用来存放其元素的内存空间。只要提供一个可以表示 vector 内存的辅助类，我们就能在简化代码的同时大大降低忘记释放内存的可能性：

```
template<class T, class A = allocator<T> >
struct vector_base {                                  // vector 的内存结构
    A alloc;        // 分配器
    T* elem;        // 分配空间的开始
    T* space;       // 元素序列的末尾，可扩展空间的开始
    T* last;        // 已分配空间的末尾

    vector_base(const A& a, typename A::size_type n)
        : alloc{a}, elem{alloc.allocate(n)}, space{elem+n}, last{elem+n} { }
    ~vector_base() { alloc.deallocate(elem,last-elem); }

    vector_base(const vector_base&) = delete;                 // 无拷贝操作
    vector_base& operator=(const vector_base&) = delete;

    vector_base(vector_base&&);                               // 移动操作
    vector_base& operator=(vector_base&&);
};
```

只要 elem 和 last 是正确的，vector_base 就能被销毁。vector_base 处理的不是类型 T 的对象，而是类型 T 的内存。因此，vector_base 的用户必须在已分配的空间上显式地构造全

部对象，并且在 vector_base 被销毁之前销毁掉 vector_base 的所有对象。

vector_base 的唯一目的就是作为 vector 实现的一部分。我们很难预测一个类被用到何时何地，因此我事先约定不允许拷贝 vector_base，并且确保 vector_base 的移动操作可以正确转移已分配内存的所有权：

```
template<class T, class A>
vector_base<T,A>::vector_base(vector_base&& a)
    : alloc{a.alloc},
    elem{a.elem},
    space{a.space},
    last{a.space}
{
    a.elem = a.space = a.last = nullptr; // 不再拥有任何内存
}

template<class T, class A>
vector_base<T,A>::& vector_base<T,A>::operator=(vector_base&& a)
{
    swap(*this,a);
    return *this;
}
```

移动赋值的上述定义使用 swap() 来转移任意已分配内存的所有权。我们没有销毁 T 的对象：vector_base 负责处理内存并且赋予 vector 类型为 T 的对象。

在已知 vector_base 的基础上，我们可以重新定义 vector：

```
template<class T, class A = allocator<T> >
class vector {
    vector_base<T,A> vb;                    // 元素分配器
    void destroy_elements();
public:
    using size_type = unsigned int;

    explicit vector(size_type n, const T& val = T(), const A& = A());

    vector(const vector& a);                // 复制构造函数
    vector& operator=(const vector& a);     // 复制赋值运算

    vector(vector&& a);                     // 移动构造函数
    vector& operator=(vector&& a);          // 移动赋值运算

    ~vector() { destroy_elements(); }

    size_type size() const { return vb.space-vb.elem; }
    size_type capacity() const { return vb.last-vb.elem; }

    void reserve(size_type);                // 增加存储容量

    void resize(size_type, T = {});         // 改变元素个数
    void clear() { resize(0); }             // 清空 vector
    void push_back(const T&);               // 在末尾添加一个元素

    // ...
};

template<class T, class A>
```

```
void vector<T,A>::destroy_elements()
{
    for (T* p = vb.elem; p!=vb.space; ++p)
        p->~T();                    // 销毁元素（见 17.2.4 节）
    vb.space=vb.elem;
}
```

vector 的析构函数为每个元素显式地调用 T 的析构函数。因此，一旦某个元素的析构函数抛出了异常，vector 的析构过程就失败了。如果这种情况发生在异常引起的栈展开期间并且触发了 terminate() 函数（见 13.5.2.5 节），则将引起程序灾难。在正常的析构过程中，析构函数抛出异常通常会导致资源泄漏以及不可预知的代码行为。事实上，没有哪种方法可以切实有效地防止析构函数抛出异常，标准库也无法确保它的析构函数不会抛出异常（见 13.2 节）。

我们可以简单地把构造函数定义成如下形式：

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
    :vb(a,n)        // 为 n 个元素分配空间
{
    uninitialized_fill(vb.elem,vb.elem+n,val);// 拷贝 val
}
```

这一简练的构造函数有助于简化任意与 vector 的构造和初始化有关的操作。例如，在拷贝构造函数中我们把 uninitialized_fill() 替换成了 uninitialized_copy()：

```
template<class T, class A>
vector<T,A>::vector(const vector<T,A>& a)
    :vb(a.alloc,a.size())
{
    uninitialized_copy(a.begin(),a.end(),vb.elem);
}
```

上述构造函数依赖一种基本的语言规则，即，当构造函数抛出异常时，已经完整构造的子对象（包括基类对象）都将正常销毁（见 13.3 节）。uninitialized_fill() 算法及其变种（见 13.6.1 节）为部分构造的序列也提供了类似的保障。

移动操作更简单：

```
template<class T, class A>
vector<T,A>::vector(vector&& a)                    // 移动构造函数
    :vb(move(a.vb))          // 转移所有权
{
}
```

vector_base 的移动构造函数把参数的表示设置为“空”。

对于移动赋值，我们兼顾目标的旧值：

```
template<class T, class A>
vector<T,A>::& vector<T,A>::operator=(vector&& a)          // 移动赋值运算
{
    clear();            // 销毁元素
    swap(*this,a);      // 转移所有权
}
```

严格来说 clear() 是多余的，因为我们可以认为右值 a 在赋值操作后会被立即销毁。然而，谁也无法确保个别程序员不会改动或者重新设置 std::move()，因此现在的写法还是比较稳妥的。

13.6.3 赋值

和往常一样，赋值操作必须和对象的构造区别开来。因为赋值操作需要考虑如何处理旧值。我们先来考虑一种比较直接的实现：

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)        // 提供强保障（见 13.2 节）
{
    vector_base<T,A> b(alloc,a.size());                     // 获取内存空间
    uninitialized_copy(a.begin(),a.end(),b.elem);           // 拷贝元素
    destroy_elements();                                     // 销毁旧元素
    swap(vb,b);                                             // 转移所有权
    return *this;                                           // 隐式地销毁旧值
}
```

vector 的赋值操作提供了强安全保障，但是它重复了大量构造函数和析构函数的代码。我们可以通过下面的形式避免重复：

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)        // 提供强保障（见 13.2 节）
{
    vector temp {a};                    // 拷贝分配器
    std::swap(*this,temp);              // 交换内容
    return *this;
}
```

temp 的析构函数销毁掉了旧元素，temp 的 vector_base 的析构函数则负责释放这些元素所占的内存空间。

我们为 swap() 定义了 vector_base 的移动操作，所以我们可以在 vector_base 中使用标准库 swap()（见 35.5.2 节）。

两个版本的性能应该是相等的，毕竟它们只是实现相同操作的两种方式而已。然而，第二种实现更加简短，而且不会重复其他相关 vector 函数的代码。因此，后者不易出错且维护的代价较小。

请注意，我并没有检查自赋值（v=v）的情况。= 的实现机理是先构造一份拷贝，然后交换二者的内容。因此，即使有自赋值发生也不会有什么问题。加上一条检查语句带来的效益要高于使用另外的 vector 赋值所增加的代价。

在上述两种实现中，还可以进行如下优化：

[1] 如果目标 vector 的容量足够存放新的 vector，则我们无须分配新空间。

[2] 元素赋值的效率显然高于先析构一个元素再构造一个新元素。

把这两点优化因素考虑进去之后，我们得到：

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)        // 优化的版本，实现了基本保障（见 13.2 节）
{
    if (capacity() < a.size()) {  // 分配新的 vector 内容
        vector temp {a};                // 拷贝分配器
        swap(*this,temp);               // 交换内容
        return *this;                   // 隐式地销毁旧值
    }
    if (this == &a) return *this;                           // 优化自赋值

    size_type sz = size();
```

```
        size_type asz = a.size();
        vb.alloc = a.vb.alloc;                                  // 拷贝分配器
        if (asz<=sz) {
            copy(a.begin(),a.begin()+asz,vb.elem);
            for (T* p = vb.elem+asz; p!=vb.space; ++p)          // 销毁多余的元素（见 16.2.6 节）
                p->~T();
        }
        else {
            copy(a.begin(),a.begin()+sz,vb.elem);
            uninitialized_copy(a.begin()+sz,a.end(),vb.space); // 构造额外的元素
        }
        vb.space = vb.elem+asz;
        return *this;
    }
```

这些优化可不是免费的。显然，代码的复杂性大大提高了。虽然我执行了对自赋值的检查，但我的目的是告诉读者该如何做，而并不是这么做真有多大的意义。

copy() 算法（见 32.5.1 节）没有提供强异常安全保障。因此，当 T::operator=() 在 copy() 的过程中抛出异常时，被赋值的 vector 不必是赋值内容的一份拷贝，也无须完全保持不变。例如，我们可以令前 5 个元素是赋值内容的拷贝，而剩下的元素保持不变。而且，当 T::operator=() 抛出异常之后，即使那个正被拷贝的元素既不是它的旧值也不是赋值内容对应的值，也属于正常情况。换句话说，如果 T::operator=() 抛出异常时它的运算对象都处于有效状态，则即使 vector 的状态不是预期中的样子，它也仍然是有效的。

与上面的最后一个实现版本相比，标准库 vector 的赋值提供了一个更弱的基本异常安全保障，同时它潜在的性能提高了。如果你希望抛出异常的时候不要改变 vector 的值，则需要使用提供强保障的标准库实现或者干脆用你自己实现的赋值操作。例如：

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b)     // 简单的 a = b
{
    vector<T,A> temp(b);            // 把元素拷贝给临时量
    swap(a,temp);
}
```

或者也可以用值调用的方式（见 12.2 节）：

```
template<class T, class A>
void safe_assign(vector<T,A>& a, vector<T,A> b)        // 简单的 a = b（注意：b 是值传递的）
{
    swap(a,b);
}
```

至于说最后的这个版本到底是简洁大方还是不切实际（以至于不易维护），目前尚无定论。

13.6.4　改变尺寸

vector 最大的优点是可以改变它的大小以适应我们的需要。改变 vector 尺寸最常用的函数是 v.push_back(x)，它把 x 添加到 v 的末尾后执行 v.resize(s)，其中 s 是 v 的元素数量。

13.6.4.1　reserve()

实现上述函数的关键是 reserve()，它负责在 vector 的末尾增加空间使其扩容。换句话说，reserve() 提升了 vector 的 capacity()。如果新的 vector 比原来的大，则 reserve()

需要分配新的存储空间并且把元素移入其中。我们可以使用未优化的赋值来实现它（见13.6.3节）：

```
template<class T, class A>
void vector<T,A>::reserve(size_type newalloc)      // 第一版，含有缺陷
{
    if (newalloc<=capacity()) return;              // 无法减少空间分配
    vector<T,A> v(capacity());                     // 创建一个新尺寸的 vector
    copy(elem,elem+size(),v.begin());              // 拷贝元素
    swap(*this,v);                                 // 存入新值
} // 隐式地释放旧值
```

这对于提供强保证来说很有用。然而，并非所有类型都有默认值，况且我们也不希望额外的元素都被初始化。所以上述代码存在缺陷。此外，该代码对元素扫描了两次，一次是默认构造，另一次是拷贝值，这种做法显得有点繁琐。因此，我们执行如下的优化操作：

```
template<class T, class A>
void vector<T,A>::reserve(size_type newalloc)
{
    if (newalloc<=capacity()) return;                       // 无法减少空间分配
    vector_base<T,A> b {vb.alloc,newalloc};                 // 获取新空间
    uninitialized_move(elem,elem+size(),b.elem);            // 移动元素
    swap(vb,b);                                             // 存入新值
} // 隐式地释放旧值
```

它的问题是标准库并不提供 uninitialized_move()，因此我们必须写成：

```
template<typename In, typename Out>
Out uninitialized_move(In b, In e, Out oo)
{
    for (; b!=e; ++b,++oo) {
        new(static_cast<void*>(&*oo)) T{move(*b)};  // 移动构造
        b->~T();                                    // 销毁
    }
    return b;
}
```

通常情况下，我们无法从一次失败的移动操作中恢复原来的状态，因此我也不会试图这么做。这个 uninitialized_move() 只提供了基本保障，但是它非常简单，并且在大多数情况下比较高效。此外，标准库 reserve() 也只提供了基本保障。

reserve() 一旦移动了元素，vector 的迭代器可能就会失效了（见31.3.3节）。

移动操作不应该抛出异常。如果遇到了移动操作的实现可能抛出异常的情况，我们应该尽量避免它。移动操作抛出异常的现象非常罕见，属于预期之外的情况，并且对程序的执行逻辑会产生不利影响，应该尽可能避免。其中，标准库 move_if_noexcept() 会对我们有所帮助（见35.5.1节）。

因为编译器并不知道元素 elem[i] 要被销毁，所以我们需要显式地使用 move()。

13.6.4.2 resize()

vector 的成员函数 resize() 改变元素的数量。如果已经有了 reserve()，resize() 的实现会变得非常简单。如果元素的数量增加，我们必须构造新的元素；反之，如果元素的数量减少，我们必须销毁多余的元素：

```
template<class T, class A>
void vector<T,A>::resize(size_type newsize, const T& val)
```

```
{
    reserve(newsize);
    if (size()<newsize)
        uninitialized_fill(elem+size(),elem+newsize,val);    // 构造新元素
    else
        destroy(elem.size(),elem+newsize);                   // 销毁多余元素
    vb.space = vb.last = vb.elem+newsize;
}
```

不存在标准的 destroy()，但是它的结构其实非常简单：

```
template<typename In>
void destroy(In b, In e)
{
    for (; b!=e; ++b)      // 销毁 [b:e)
        b->~T();
}
```

13.6.4.3　push_back()

从异常安全的角度来看，push_back() 和赋值非常相似。当添加新元素失败的时候，我们都必须确保 vector 的内容没有被改变：

```
template< class T, class A>
void vector<T,A>::push_back(const T& x)
{
    if (capacity()==size())                          // 剩余空间不足，重新分配：
        reserve(sz?2*sz:8);                          // 容量扩充一倍，或者初始分配为 8
    vb.alloc.construct(&vb.elem[size()],val);        // 在末尾添加 val
    ++vb.space;                                      // 增加大小
}
```

用于初始化 *space 的拷贝构造函数有可能抛出异常。此时，vector 的值并未改变并且 space 也没有增加。然而，reserve() 可能已经为已有的元素重新分配了空间。

push_back() 的定义中包含两个“魔数”(2 和 8)。尽管符合工业规范的代码不应该这么做，但它还是需要定义初始分配空间大小（此处是 8）和增长比率（此处是 2，规定 vector 每次扩充一倍）。对于这两个值来说，它们取任意数值都是有可能的。一种合理的预期是，既然 vector 用到了 push_back()，那么它很有可能还会用到其他类似的函数。扩容比率 2 比数学上的最小平均内存优化因子 1.618 要大，其目的是当内存容量足够时为系统提供更优的运行时性能。

13.6.4.4　最后的一点思考

请注意，我们在 vector 的实现中没有用到 try 块（除了隐藏在 uninitialized_copy() 中的那个之外）。我们非常小心地设计操作的顺序以确保当抛出异常时，vector 不被改变或者至少处于有效的状态。

与使用 try 块显式地处理错误相比，通过控制操作顺序以及 RAII 技术（见 13.3 节）实现异常安全的方法要更简单有效。但是一旦程序员组织代码的方式有误，那么与缺少异常处理代码相比，前者引发异常安全问题的可能性会大得多。关于代码顺序最基本的规则是先构建好替代者并确保可以对其正常赋值，再销毁现有的信息。

异常通常伴随着人们预期之外的控制流。对于 reserve()、safe_assign() 和 push_back() 等简单局部控制流代码来说，一般很少会出错。当看到这样的代码时，我们很容易

发问“这行代码会抛出异常吗？如果抛出异常的话会怎样呢？”相反，对于包含条件语句或者嵌套循环等复杂控制结构的大型程序来说，这样发问就很困难了。在局部控制结构中添加try块显然会提高它的复杂性并增加出错的风险（见13.3节）。可以断言，局部控制流越简单，排序方法和RAII技术相比使用try块的技术就越有优势。简单且格式化的代码易于理解，一般不会出错。

通过本章这个vector的例子，我们可以观察到异常可能引起哪些问题，以及我们应该如何处理这些问题。C++标准中并没有和本例一模一样的实现，但是它们提供的异常安全保障是一致的。

13.7 建议

[1] 在设计初期尽早确定异常处理策略；13.1节。
[2] 当无法完成既定任务时抛出异常；13.1.1节。
[3] 用异常机制处理错误；13.1.4.2节。
[4] 为特定任务设计用户自定义异常类型（而非内置类型）；13.1.1节。
[5] 如果由于某种原因你无法使用异常，尽量模仿其机制；13.1.5节。
[6] 使用层次化异常处理；13.1.6节。
[7] 保持异常处理的各个部分尽量简洁；13.1.6节。
[8] 不要试图捕获每个函数的每个异常；13.1.6节。
[9] 至少提供基本保障；13.2节，13.6节。
[10] 除非有足够的理由，否则最好提供强保障；13.2节，13.6节。
[11] 让构造函数建立不变式，如果不能，则抛出异常；13.2节。
[12] 抛出异常前先释放局部资源；13.2节。
[13] 谨记在构造函数中抛出异常前释放所有已获取的资源；13.3节。
[14] 如果局部控制结构足以满足要求，不要使用异常；13.1.4节。
[15] 用“资源获取即初始化”技术管理资源；13.3节。
[16] 尽量减少使用try块；13.3节。
[17] 并非所有程序都需要异常安全；13.1节。
[18] 用“资源获取即初始化”技术和异常处理程序维护不变式；13.5.2.2节。
[19] 资源句柄优于弱结构化的finally；13.3.1节。
[20] 为你的不变式设计错误处理策略；13.4节。
[21] 能在编译时检查的东西最好在编译时检查（使用static_assert）；13.4节。
[22] 用你的错误处理策略执行不同层级的检查；13.4节。
[23] 如果函数不会抛出异常，把它声明成noexcept的；13.5.1.1节。
[24] 不要使用异常说明；13.5.1.3节。
[25] 用引用的方式捕获层次体系中的异常；13.5.2节。
[26] 并非每个异常都派生自exception类；13.5.2.2节。
[27] 让main()捕获和报告所有异常；13.5.2.2节，13.5.2.4节。
[28] 销毁信息前先要找到它的替代者；13.6节。
[29] 在赋值运算中抛出异常前要确保运算对象处于有效状态；13.2节。

［30］ 不要让析构函数抛出异常；13.2 节。

［31］ 把普通代码和异常处理代码分离开来；13.1.1 节，13.1.4.2 节。

［32］ 当异常发生时，如果由 new 分配的内存尚未被释放将造成内存泄漏，请注意这一点；13.3 节。

［33］ 函数如果能抛出一个异常，那么它就会抛出这个异常，遵循这一假设；13.2 节。

［34］ 库不应自行终止程序，正确的做法是抛出一个异常然后由调用者决定该怎么做；13.4 节。

［35］ 库不应直接输出面向最终用户的错误诊断信息，正确的做法是抛出一个异常然后由调用者决定该怎么做；13.1.3 节。

名字空间

今年是 787 年!
公元后?

——巨蟒剧团

- 组合问题
- 名字空间
  显式限定；using 声明；using 指示；参数依赖查找；名字空间是开放的
- 模块化和接口
  名字空间作为模块；实现；接口和名字
- 组合使用名字空间
  便利性与安全性；名字空间别名；组合名字空间；组合与选择；名字空间和重载；版本控制；名字空间嵌套；无名名字空间；C 头文件
- 建议

14.1 组合问题

任何实际问题都是由若干独立部分组成的。函数（见 2.2.1 节和第 12 章）和类（见 3.2 节和第 16 章）提供了相对细粒度的关注点分离，而“库”、源文件和编译单元（见 2.4 节和第 15 章）则提供了粗粒度的分离。逻辑上最理想的方式是模块化（modularity），即独立的事物保持分离，只允许通过良好定义的接口访问“模块”。C++ 并不是通过单一语言特性来支持模块的概念，也并不存在模块这种语法构造。取而代之，C++ 通过其他语言特性（如函数、类和名字空间）的组合和源码的组织来表达模块化。

本章和下一章介绍程序的粗粒度结构以及以源文件为单位的物理组织，这两章更多的是关注大规模编程而不是单个类型、算法和数据结构的精致表达。

我们来考虑当模块化设计失败时可能引起的一些问题。例如，一个图形库可能提供不同种类的图形化的 Shape 及函数：

```
// Graph_lib:

class Shape { /* ... */ };
class Line : public Shape { /* ... */ };
class Poly_line: public Shape { /* ... */ };          // 相连的 Line 的序列
class Text : public Shape { /* ... */ };              // 文本标签

Shape operator+(const Shape&, const Shape&);  // 形状组合

Graph_reader open(const char*);                       // 打开 Shape 文件
```

还有另外一个库提供文本处理特性：

```
// Text_lib:

class Glyph { /* ... */ };
class Word { /* ... */ };          // Glyph 序列
class Line { /* ... */ };          // Word 序列
class Text { /* ... */ };          // Line 序列

File* open(const char*);                          // 打开文本文件

Word operator+(const Line&, const Line&);   // 连接
```

现在，我们忽略图形和文本处理的特定设计问题，只考虑如何在一个程序中一起使用 Graph_lib 和 Text_lib 的问题。

假定（足够真实）Graph_lib 的特性定义在头文件 Graph_lib.h 中（见 2.4.1 节），而 Text_lib 的特性定义在头文件 Text_lib.h 中。现在，我可以“无害地”#include 两个头文件，并尝试使用两个库中的特性：

```
#include "Graph_lib.h"
#include "Text_lib.h"
// ...
```

只是简单 #include 两个头文件会导致一些错误消息：Line、Text 和 open() 定义了两次，编译器无法消除这种歧义。继续尝试使用两个库的话，会得到更多错误消息。

已有很多技术可以处理这种*名字冲突*（name clash），例如将一个库的所有特性放在几个类中、使用不太可能重复的名字（如 Text_box 而不是 Text）或对一个库中的名字系统地使用前缀（如 gl_shape 和 gl_line）。这些方法（也被称为“变通方法”和“小技巧”）在某些情况下是可行的，但它们都不是通用方法而且可能给使用带来不便。例如，名字会变得很长，使用很多不同的名字会限制泛型编程（见 3.4 节），等等。

14.2　名字空间

名字空间（namespace）的概念用来直接表示本属一体的一组特性，例如库代码。名字空间的成员都位于相同的作用域中，无须特殊符号即可相互访问，而从名字空间外访问它们就需要显式符号。特别是，我们可以通过将多组声明（如类的接口）划分为若干名字空间来避免名字冲突。例如，我们可以将图形库命名为 Graph_lib：

```
namespace Graph_lib {
    class Shape { /* ... */ };
    class Line : public Shape { /* ... */ };
    class Poly_line: public Shape { /* ... */ };          // 相连的 Line 的序列
    class Text : public Shape { /* ... */ };              // 文本标签

    Shape operator+(const Shape&, const Shape&);   // 形状组合

    Graph_reader open(const char*);                       // 打开 Shape 文件
}
```

类似地，我们的文本库显然可以命名为 Text_lib：

```
namespace Text_lib {
    class Glyph { /* ... */ };
    class Word { /* ... */ };          // Glyph 序列
    class Line { /* ... */ };          // Word 序列
    class Text { /* ... */ };          // Line 序列
```

```
    File* open(const char*);                    // 打开文本文件

    Word operator+(const Line&, const Line&);   // 连接
}
```

只要我们设法取一些独特的名字，如 Graph_lib 和 Text_lib（见 14.4.2 节），就可以一起编译这两组声明而不会产生名字冲突。

一个名字空间应该表达某种逻辑结构：一个名字空间中的声明应该一起提供一些特性，使得在用户看来它们是一个整体，而且能反映一组共同的设计策略。它们应被看成一个逻辑单元，如“图形库”或“文本处理库”，与我们看待一个类的成员相似。实际上，在一个名字空间中声明的实体是作为名字空间的成员被引用的。

一个名字空间就形成了一个（具名的）作用域。在一个名字空间中，稍后的声明可以引用之前定义的成员，但你不能从名字空间之外引用其成员（除非使用特殊方式）。例如：

```
class Glyph { /* ... */ };
class Line { /* ... */ };

namespace Text_lib {
    class Glyph { /* ... */ };
    class Word { /* ... */ };      // Glyph 序列
    class Line { /* ... */ };// Word 序列
    class Text { /* ... */ };// Line 序列

    File* open(const char*);                    // 打开文本文件

    Word operator+(const Line&, const Line&);   // 连接
}

Glyph glyph(Line& ln, int i);   // ln[i]
```

在本例中，Text_lib::operator+() 声明中的 Word 和 Line 引用的是 Text_lib::Word 和 Text_lib::Line。全局的 Line 不会影响局部名字查找。相反，全局 glyph() 声明中的 Glyph 和 Line 引用的则是全局 ::Glyph 和 ::Line。此（非局部）查找也不会受 Text_lib 的 Glyph 和 Line 的影响。

为了引用名字空间的成员，我们可以使用完整的限定名字。例如，如果我们希望 glyph() 使用 Text_lib 中的定义，可以编写代码如下：

```
Text_lib::Glyph glyph(Text_lib::Line& ln, int i);   // ln[i]
```

从名字空间外引用成员的其他方法包括 using 声明（见 14.2.2 节）、using 指示（见 14.2.3 节）和参数依赖查找（见 14.2.4 节）。

14.2.1 显式限定

我们可以在名字空间的定义中声明一个成员，稍后用“名字空间名 :: 成员名”的语法定义它。

名字空间的成员必须用如下语法引入：

```
namespace namespace-name {
    // 声明和定义
}
```

例如：

```
namespace Parser {
        double expr(bool);          // 声明
        double term(bool);
        double prim(bool);
}

double val = Parser::expr();    // 使用

double Parser::expr(bool b)    // 定义
{
        // ...
}
```

我们不能用限定符语法（iso.7.3.1.2）在名字空间的定义之外为其声明一个新成员。这是为了捕获拼写错误和类型不匹配之类的错误，以及便于在名字空间声明中查找所有名字。例如：

```
void Parser::logical(bool);     // 错误：Parser 中没有 logical()
double Parser::trem(bool);      // 错误：Parser 中没有 trem()（拼写错误）
double Parser::prim(int);       // 错误：Parser::prim() 接受一个 bool 类型参数（错误类型）
```

一个名字空间形成一个作用域，通常的作用域规则也适用于名字空间。因此，“名字空间”是一个非常基础、非常简单的概念。程序规模越大，用名字空间表达程序的逻辑划分就越有用。全局作用域也是一个名字空间，可以显式地用 :: 来引用。例如：

```
int f();          // 全局函数

int g()
{
        int f;        // 局部变量：屏蔽了全局函数
        f();          // 错误：不能调用一个整型变量
        ::f();        // 正确：调用全局函数
}
```

类也是名字空间（见 16.2 节）

14.2.2　using 声明

当我们需要在名字空间外频繁使用其名字时，反复用名字空间名进行显式限定很繁琐。考虑下面的代码：

```
#include<string>
#include<vector>
#include<sstream>

std::vector<std::string> split(const std::string& s)
        // 将 s 划分为空白符分隔的子串
{
        std::vector<std::string> res;
        std::istringstream iss(s);
        for (std::string buf; iss>>buf;)
                res.push_back(buf);
        return res;
}
```

反复使用 std 进行限定非常冗长乏味，也容易分散读者的注意力。在如此小的例程中，我们

就反复使用了四次 std::string。为了缓解这个问题，我们可以使用 using 声明来指出在这段代码中 string 表示 std::string：

```
using std::string;   // 其 "string" 表示 "std::string"

std::vector<string> split(const string& s)
    // 将 s 划分为空白符分隔的子串
{
    std::vector<string> res;
    std::istringstream iss(s);
    for (string buf; iss>>buf;)
        res.push_back(buf);
    return res;
}
```

using 声明将一个代用名引入了作用域，最好尽量保持代用名的局部性以避免混淆。

当用于一个重载的名字时，using 声明会应用于其所有重载版本。例如：

```
namespace N {
    void f(int);
    void f(string);
};

void g()
{
    using N::f;
    f(789);          // N::f(int)
    f("Bruce");      // N::f(string)
}
```

有关在类层次中使用 using 声明的内容，请见 20.3.5 节。

14.2.3 using 指示

在 split() 示例中，我们虽然为 std::string 引入了代用名，但仍然使用了一次 std::。我们常常希望不加限定地使用某个名字空间中的每个名字。此时，我们可以为名字空间中的每个名字提供一个 using 声明来实现这一目的，但这种方法冗长乏味，而且每当向名字空间中加入一个新名字或从其中删除一个名字时，都需要再修改 using 声明。作为替代方法，我们可以使用 using 指示，要求编译器允许我们在所在作用域中无须使用限定即可访问某个名字空间中的所有名字。例如：

```
using namespace std;   // 令来自 std 的每个名字都可访问

vector<string> split(const string& s)
    // 将 s 划分为空白符分隔的子串
{
    vector<string> res;
    istringstream iss(s);
    for (string buf; iss>>buf;)
        res.push_back(buf);
    return res;
}
```

使用 using 指示后，使用来自名字空间中的名字就好像它们是声明在名字空间外一样（见 14.4 节）。对那些广为人知、也广泛使用的库中的名字，利用 using 指示使得它们可以在

未加限定的情况下使用，这是一种流行的简化代码的技术。本书中很多代码都利用了这种技术来简化标准库特性的使用。标准库特性定义在名字空间 std 中。

在一个函数中，可以安全地使用 using 指示以方便符号表示，但对全局 using 指示必须小心谨慎，因为过度使用 using 指示会导致名字冲突，而避免名字冲突恰恰是引入名字空间的目的。例如：

```
namespace Graph_lib {
    class Shape { /* ... */ };
    class Line : Shape { /* ... */ };
    class Poly_line: Shape { /* ... */ };    // 相连的 Line 的序列
    class Text : Shape { /* ... */ };        // 文本标签

    Shape operator+(const Shape&, const Shape&);   // 组合

    Graph_reader open(const char*);   // 打开 Shape 文件
}

namespace Text_lib {
    class Glyph { /* ... */ };
    class Word { /* ... */ };      // Glyph 序列
    class Line { /* ... */ };      // Word 序列
    class Text { /* ... */ };      // Line 序列

    File* open(const char*); // 打开文本文件

    Word operator+(const Line&, const Line&);   // 连接
}

using namespace Graph_lib;
using namespace Text_lib;

Glyph gl;                  // Text_lib::Glyph
vector<Shape*> vs;         // Graph_lib::Shape
```

到目前为止一切还好。特别是，我们可以使用不冲突的名字，如 Glyph 和 Shape。但是，只要使用重名实体，就会发生名字冲突，就像根本没有使用名字空间一样。例如：

```
Text txt;                                  // 错误：二义性
File* fp = open("my_precious_data");       // 错误：二义性
```

因此，我们必须小心使用全局作用域中的 using 指示。特别是，除了极特殊的情况外（例如为了帮助代码转换），不要在头文件中将一个 using 指示置于全局作用域中，因为你永远也不知道头文件可能在哪里被 #include。

14.2.4　参数依赖查找

接受参数类型为用户自定义类型 X 的函数常常与 X 定义在相同的名字空间中。因此，如果在使用函数的上下文中找不到函数定义，我们可以在其参数的名字空间中查找它。例如：

```
namespace Chrono {
    class Date { /* ... */ };

    bool operator==(const Date&, const std::string&);

    std::string format(const Date&);   // 创建字符串表示
```

```
    // ...
}

void f(Chrono::Date d, int i)
{
    std::string s = format(d);      // Chrono::format()
    std::string t = format(i);      // 错误：作用域中没有 format() 的定义
}
```

与使用显式限定相比，这种查找规则（称为参数依赖查找，argument-dependent lookup，或简称为 ADL）使程序员可以省去很多输入工作，而且它还不像 using 指示（见 14.2.3）那样会污染名字空间。它对于运算符运算对象（见 18.2.5 节）和模板参数（见 26.3.5 节）特别有用，对这些情况显式限定会非常繁琐。

注意，名字空间本身必须处于使用函数的作用域中，且函数声明必须在函数查找和使用之前。

一个函数自然可以接受来自多个名字空间的参数。例如：

```
void f(Chrono::Date d, std::string s)
{
    if (d == s) {
        // ...
    }
    else if (d == "August 4, 1914") {
        // ...
    }
}
```

在此情况下，我们（照旧）在函数调用的作用域中以及每个参数（包括每个参数的类及其基类）的名字空间中查找函数定义，并对找到的所有函数采用常规的重载解析规则（见 12.3 节）。特别是，对本例中的 d == s 调用，我们在包含 f() 的作用域中、名字空间 std 中（其中有针对 string 的 == 版本的定义）以及名字空间 Chrono 中查找 operator==。标准库中有一个 std::operator==()，但它不接受 Date 参数，因此我们使用接受 Date 参数的 Chrono::operator==()。参见 18.2.5 节。

当一个类成员调用一个命名函数时，编译器会优先选择同一个类的其他成员及其基类而不是基于参数类型查找到的函数（运算符遵循不同的规则；参见 18.2.1 节和 18.2.5 节）。例如：

```
namespace N {
    struct S { int i };
    void f(S);
    void g(S);
    void h(int);
}

struct Base {
    void f(N::S);
};

struct D : Base {
    void mf();

    void g(N::S x)
    {
```

```
        f(x);       // 调用 Base::f()
        mf(x);      // 调用 D::mf()
        h(1);       // 错误：没有可用的 h(int)
    }
};
```

在 C++ 标准中，关于参数依赖查找的规则都有关联名字空间（associated namespace）的措辞（见 iso.3.4.2）。基本上：

- 如果一个参数是一个类成员，关联名字空间即为类本身（包括其基类）和包含类的名字空间。
- 如果一个参数是一个名字空间的成员，则关联名字空间即为外层的名字空间。
- 如果一个参数是内置类型，则没有关联名字空间。

参数依赖查找可以帮助我们避免大量乏味、令人分心的代码输入工作，但偶尔也会带来意外的结果。例如，在查找函数 f() 的声明时，并不优先选择 f() 调用所在的 namespace 中的函数（对于一个 class 中的 f() 调用，则会优先查找同一个类中的声明）：

```
namespace N {
    template<class T>
        void f(T, int);   // N::f()
    class X { };
}

namespace N2 {
    N::X x;

    void f(N::X, unsigned);

    void g()
    {
        f(x,1);     // 调用 N::f(X,int)
    }
}
```

对于 g() 中的 f() 调用，选择 N2::f() 似乎很明显，但结果并不是这样。编译器会应用重载解析规则，并找到最佳匹配：对于 f(x,1)，N::f() 是最佳匹配，因为对于参数 1，类型 int 较之 unsigned 更为匹配。也存在相反的例子，编译器选择了调用者名字空间中的函数，但程序员期待的却是使用一个已知名字空间中的更好的函数（例如，来自 std 中的一个标准库函数）。这可能会造成非常大的困扰，请参阅 26.3.6 节。

14.2.5　名字空间是开放的

名字空间是开放的；即，你可以从多个分离的名字空间声明中向一个名字空间添加名字。例如：

```
namespace A {
    int f();    // 现在 A 包含成员 f()
}

namespace A {
    int g();    // 现在 A 有两个成员，f() 和 g()
}
```

这样，名字空间的成员就不需要连续放置在单一的文件中。当我们转换较旧的程序，改写为

使用名字空间的版本时，这一特点就非常重要了。例如，考虑一个未使用名字空间的头文件：

```
// 我的头文件：

    void mf();      // 我的函数
    void yf();      // 你的函数
    int mg();       // 我的函数
    // ...
```

在此，我们添加声明的方式（很不明智地）未考虑模块化的问题。我们可以改写这段代码而又无须重排声明的顺序：

```
// 我的头文件：

    namespace Mine {
        void mf();      // 我的函数
        // ...
    }

    void yf();          // 你的函数（还未放入一个名字空间中）

    namespace Mine {
        int mg();       // 我的函数
        // ...
    }
```

在编写新代码时，我更喜欢使用很多小的名字空间（见 14.4 节），而不是将大段代码放入单一的名字空间中。但很多软件在转换为使用名字空间的版本时，很难实现这一点。

将一个名字空间的成员定义分散在多个分离的名字空间声明中还有另一个原因：我们有时希望将名字空间作为接口的部分与用来支持简单实现的部分区分开来；14.3 节给出了一个例子。

使用名字空间别名（见 14.4.2 节）无法重新打开名字空间。

14.3 模块化和接口

任何一个实际程序都会由多个分开的部分组成。例如，即使是简单的“Hello, world!”程序也包含至少两个部分：请求打印 Hello, world! 的用户代码和完成实际打印操作的 I/O 系统。

考虑 10.2 节中的桌面计算器例子。它可以看作是由五部分组成的：

[1] 语法分析器，完成语法分析：expr()、term() 和 prim()；

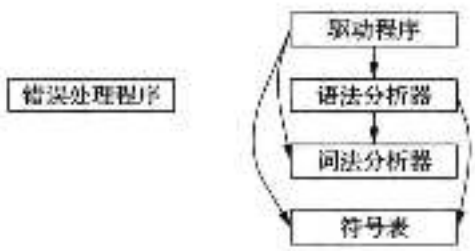
[2] 词法分析器，将字符组合为单词：Kind、Token、Token_stream 和 ts；

[3] 符号表，保存（字符串，值）对：table；

[4] 驱动程序：main() 和 calculate()；

[5] 错误处理程序：error() 和 number_of_errors。

其结构可图示如下：



其中箭头表示“使用”。为了简化图示，我并未表示出每个部分都依赖于错误处理这个事实。实际上，计算器程序最初构思由三部分组成，后来出于完整性的考虑又加入了驱动程序和错误处理程序。

当一个模块使用另一个模块时，无须了解被使用模块的全部细节。理想情况下，一个模块的大部分细节都不应被其使用者所知。因此，我们将模块的实现与其接口分离开来。例如，语法分析器直接依赖于也只依赖于词法分析器接口，而不是完整的词法分析器。词法分析器简单地实现了其接口所发布的那些服务。这一关系可图示如下：



其中虚线表示“实现”。我认为这是程序真正应有的结构，我们程序员的任务就是用代码忠实地实现这种结构。这样得到的代码具有简洁、高效、易理解、易维护等优良性质，因为它直接反映了我们的基础设计。

接下来几节会介绍如何将桌面计算器程序的逻辑结构整理得更为清晰，15.3 节将介绍如何重组程序源码的物理结构来利用这一点。计算器程序规模很小，在“现实生活”中，对于这种程度的名字空间和分离编译（见 2.4.1 节和 15.1 节）的使用，我不会感到困扰。通过将计算器程序的结构整理得更为清晰，我们展示了一些对大规模程序也很有用的技术，这些技术令我们不会被淹没在大量代码中。这很重要，因为在实际程序中，表示为独立名字空间的“模块”通常包含数百个函数、类、模板，等等。

错误处理渗透到程序结构的每一处。当我们将一个程序分解为若干模块或（相反地）将多个模块组合为一个程序时，就必须小心，要尽量减少由错误处理引起的模块间的依赖。C++ 提供了异常机制，将错误检测和错误报告从错误处理中分离（见 2.4.3.1 节和第 13 章）。

除了本章和下一章讨论的内容之外，还有很多模块化的概念。例如，我们可以用并发执行和通信任务（见 5.3 节和第 41 章）或进程来表示模块化的重要方面。类似地，使用分离地址空间和地址空间之间的信息通信也是本章未讨论的重要主题。我认为这些模块化概念很大程度上是相互独立、相互正交的。有意思的是，无论是哪种概念，都能很容易地将一个系统分解为模块，难题其实是提供模块间安全、便捷以及高效的通信机制。

14.3.1　名字空间作为模块

名字空间是表达逻辑分组的一种机制。即，如果按照某些标准判定一些声明逻辑上属于一个整体，则可将它们放置在一个共同的名字空间中，以表达这一点。因此，我们可以用名字空间来表达计算器程序的逻辑结构。例如，桌面计算器（见 10.2.1 节）中的语法分析器的声明可以放在一个名为 Parser 的名字空间中：

```
namespace Parser {
    double expr(bool);
    double prim(bool get) { /* ... */ }
    double term(bool get) { /* ... */ }
```

```
    double expr(bool get) { /* ... */ }
}
```

我们必须先声明函数 expr()，稍后再定义它，这样才能打破 10.2.1 节中所述的依赖循环。

桌面计算器的输入部分也能放入其自己的名字空间中：

```
namespace Lexer {
    enum class Kind : char { /* ... */ };
    class Token { /* ... */ };
    class Token_stream { /* ... */ };

    Token_stream ts;
}
```

符号表部分非常简单：

```
namespace Table {
    map<string,double> table;
}
```

驱动程序不能完全放入一个名字空间中，因为 C++ 语言规则要求 main() 必须是一个全局函数：

```
namespace Driver {
    void calculate() { /* ... */ }
}

int main() { /* ... */ }
```

错误处理模块也很简单：

```
namespace Error {
    int no_of_errors;
    double error(const string& s) { /* ... */ }
}
```

如此使用名字空间将词法分析器和语法分析器提供给用户的功能很清晰地呈现出来。假如我将函数源码也包括进来，这个结构就会显得很乱。如果在一个实际规模的名字空间的声明中包含函数体，你通常就不得不从满屏的信息中艰难地查找程序提供了什么服务，亦即查找接口。

依赖接口分离说明的另一种替代方法是提供一个工具，能从包含实现细节的模块中提取出接口。但我不认为这是一个好的方法。接口说明是基础的设计行为，一个模块可为不同用户提供不同接口，而且通常接口设计应在实现细节落实之前很早就进行。

下面是接口和实现分离的 Parser 的版本：

```
namespace Parser {
    double prim(bool);
    double term(bool);
    double expr(bool);
}

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

注意，作为实现和接口分离的结果，每个函数现在恰好都有一个声明和一个定义。用户只会看到包含声明的接口，而实现（在本例中是函数体）将被放在“其他某处”，用户无须了解。

理想情况下，程序中每个实体都属于某个可识别的逻辑单元（“模块”）。因此，在一个有用的实际程序中，理想情况下每个声明都应置于某个名字空间中，而此名字空间应按其在程序中的逻辑角色命名。main() 是一个例外，它必须是全局的，以便编译器能识别出它这个特殊函数（见 2.2.1 节和 15.4 节）。

14.3.2　实现

如果我们将代码模块化，它看起来将是什么样呢？这依赖于我们决定如何从其他名字空间中访问代码。我们可以一直从“自己的”名字空间中访问名字，就像没使用名字空间时那样。但是，对于其他名字空间中的名字，我们必须选择使用显式限定、using 声明和 using 指示。

对于在实现中名字空间的使用，Parser::prim() 提供了一个很好的测试用例，因为它使用了所有其他名字空间（Driver 除外）。如果使用显式限定，就得到如下代码：

```
double Parser::prim(bool get)        // 处理初等项
{
    if (get) Lexer::ts.get();

    switch (Lexer::ts.current().kind) {
    case Lexer::Kind::number:            // 浮点常量
    {   double v = Lexer::ts.current().number_value;
        Lexer::ts.get();
        return v;
    }
    case Lexer::Kind::name:
    {   double& v = Table::table[Lexer::ts.current().string_value];
        if (Lexer::ts.get().kind == Lexer::Kind::assign) v = expr(true); // 遇到 '='：赋值
        return v;
    }
    case Lexer::Kind::minus:         // 单目减
        return -prim(true);
    case Lexer::Kind::lp:
    {   double e = expr(true);
        if (Lexer::ts.current().kind != Lexer::Kind::rp) return Error::error("')' expected");
        Lexer::ts.get();         // 吃掉 ')'
        return e;
    }
    default:
        return Error::error("primary expected");
    }
}
```

我数了一下，上面这段代码中共出现了 14 次 Lexer::，我不认为更明确地使用模块化提高了代码可读性（虽然这与理论相反）。由于这个函数本身就在名字空间 Parser 内，因此我无须使用 Parser::。

如果使用 using 声明，可得到如下代码：

```
using Lexer::ts;          // 省去了 8 次 "Lexer::"
using Lexer::Kind;        // 省去了 6 次 "Lexer::"
using Error::error;       // 省去了 2 次 "Error::"
using Table::table;       // 省去了 1 次 "Table::"
double prim(bool get)     // 处理初等项
{
    if (get) ts.get();
```

```
        switch (ts.current().kind) {
        case Kind::number:      // 浮点常量
        {   double v = ts.current().number_value;
            ts.get();
            return v;
        }
        case Kind::name:
        {   double& v = table[ts.current().string_value];
            if (ts.get().kind == Kind::assign) v = expr(true);      // 遇到'=': 赋值
            return v;
        }
        case Kind::minus:       // 单目减
            return -prim(true);
        case Kind::lp:
        {   double e = expr(true);
            if (ts.current().kind != Kind::rp) return error("')' expected");
            ts.get();           // 吃掉')'
            return e;
        }
        default:
            return error("primary expected");
        }
    }
```

我觉得对 Lexer:: 使用 using 声明是值得的，但对其他名字空间作用就很小了。

如果使用 using 指示，可以得到如下代码：

```
using namespace Lexer;      // 省去了 14 次 "Lexer::"
using namespace Error;      // 省去了 2 次 "Error::"
using namespace Table;      // 省去了 1 次 "Table::"

double prim(bool get)               // 处理初等项
{
    // 与前一段代码一样
}
```

在符号表示方面，针对 Error 和 Table 的 using 指示并不能带来很大收益，而且还有一种观点认为这会模糊之前限定名字的由来。

因此，显式限定、using 声明和 using 指示间的权衡必须具体情况具体分析。基本原则是：

[1] 如果多个名字确实有相同的限定，则对此名字空间使用 using 指示。

[2] 如果名字空间中的特定名字经常使用某个限定，则对此名字使用 using 声明。

[3] 如果一个限定对某个名字来说并不常用，则在此名字出现的地方使用显式限定，使之更为清晰。

[4] 不要对与用户程序处于相同名字空间中的名字使用显式限定。

14.3.3 接口和名字

很明显，我们为 Parser 设计的名字空间定义并非 Parser 呈现给用户的理想接口。取而代之的是，Parser 设计了一组声明，以便能方便地编写各个语法分析器函数。呈现给用户的 Parser 接口则要简单得多：

```
namespace Parser {// 用户接口
    double expr(bool);
}
```

我们看到，Parser 的名字空间做了两件事：

［1］ 为实现语法分析器的函数提供公共环境。

［2］ 为用户提供语法分析器的外部接口。

因此，驱动程序代码和 main() 只应看到用户接口。

实现语法分析器的函数看到的接口应该是我们确定能最好地表达这些函数的共享环境的那个接口。即：

```
namespace Parser {              // 实现者接口
    double prim(bool);
    double term(bool);
    double expr(bool);

    using namespace Lexer;      // 使用词法分析器提供的所有特性
    using Error::error;
    using Table::table;
}
```

接口和代码之间的关系可图示如下：

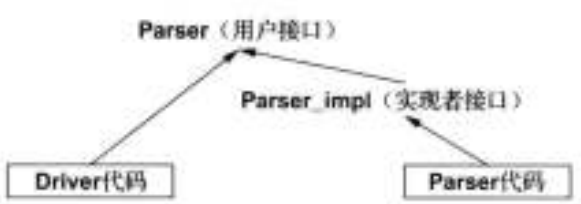


其中箭头表示“依赖于…提供的接口”。

我们可以为用户接口和实现者接口起不同的名字，但（由于名字空间是开放的；见 14.2.5 节）不必这样做。没有独立的名字不会导致混淆，因为程序的物理布局（见 15.3.2 节）自然地提供了独立的名字（文件名）。假如我们决定使用一个独立的实现者名字空间，对于用户而言语法分析器的设计也没有什么不同：

```
namespace Parser {// 用户接口
    double expr(bool);
}

namespace Parser_impl {             // 实现者接口
    using namespace Parser;

    double prim(bool);
    double term(bool);
    double expr(bool);

    using namespace Lexer; // 使用词法分析器提供的所有特性
    using Error::error;
    using Table::table;
}
```

接口和代码的关系可图示如下：



对于大规模程序，我倾向于引入 _impl 接口。

为实现者提供的接口要比为用户提供的接口更大。假如此接口是为真实系统中实际规模的模块设计的，则它会比用户接口更为频繁地被更改。模块的用户（本例中使用 Parser 的 Driver）应与这种更改隔绝，这是很重要的。

14.4 组合使用名字空间

在大规模程序中，我们可能使用很多名字空间。本节介绍用名字空间构造代码的技术。

14.4.1 便利性与安全性

using 声明将名字添加到局部作用域中，而 using 指示则不会，它只是简单地令名字在其所在作用域中可访问。例如：

```
namespace X {
    int i, j, k;
}
int k;

void f1()
{
    int i = 0;
    using namespace X;    // 令来自 X 的名字可访问
    i++;                  // 递增 i
    j++;                  // X::j
    k++;                  // 错误：X 的 k 还是全局的 k？
    ::k++;                // 全局 k
    X::k++;               // X 的 k
}

void f2()
{
    int i = 0;
    using X::i;     // 错误：i 在 f2() 中声明了两次
    using X::j;
    using X::k;     // 隐藏了全局 k

    i++;
    j++;            // X::j
    k++;            // X::k
}
```

一个局部声明的名字（普通声明或用 using 声明）会隐藏同名的非局部声明，而且在声明点上该名字任何不合法的重载都会被检测出来。

请注意 f1() 中 k++ 的二义性错误。如果已使用 using 指示将名字空间中的名字变成全局作用域可访问，那么在名字解析时它们与全局名字是平等的，后者没有任何优势。这为避免意外的名字冲突提供了重要保护，另外很重要的一点是，这确保了污染全局名字空间不会带来任何收益。

当我们使用 using 指示令库中声明的很多名字可被访问时，未用名字的冲突不被认为是错误，这是非常重要的。

14.4.2　名字空间别名

如果用户为其名字空间起了较短的名字，不同名字空间的名字很可能会冲突：

```
namespace A {// 短名字，（最终）可能冲突
    // ...
}

A::String s1 = "Grieg";
A::String s2 = "Nielsen";
```

但长名字在实际代码中可能并不实用：

```
namespace American_Telephone_and_Telegraph {      // 太长
    // ...
}

American_Telephone_and_Telegraph::String s3 = "Grieg";
American_Telephone_and_Telegraph::String s4 = "Nielsen";
```

通过为名字空间的长名起一个别名，我们就可以解决这一两难境地：

```
// 使用名字空间别名缩短名字

namespace ATT = American_Telephone_and_Telegraph;

ATT::String s3 = "Grieg";
ATT::String s4 = "Nielsen";
```

名字空间别名还允许用户引用“库”以及通过单一声明定义使用的具体是哪个库。例如：

```
namespace Lib = Foundation_library_v2r11;

// ...

Lib::set s;
Lib::String s5 = "Sibelius";
```

这极大地简化了替换库版本的任务。你可以使用 Lib 而不是直接使用 Foundation_library_v2r11，这样就能通过修改别名 Lib 的初始化语句并重新编译来实现将库版本更新到“v3r02”。重新编译会捕获源码级的不兼容。但另一方面，过度使用（任何种类的）别名会导致混乱。

14.4.3　组合名字空间

我们通常需要组合已有接口来构造新的接口。例如：

```
namespace His_string {
    class String { /* ... */ };
    String operator+(const String&, const String&);
    String operator+(const String&, const char*);
    void fill(char);
    // ...
}

namespace Her_vector {
    template<class T>
        class Vector { /* ... */ };
    // ...
}
```

```
namespace My_lib {
    using namespace His_string;
    using namespace Her_vector;
    void my_fct(String&);
}
```

有了如上声明，我们就可以用 My_lib 来编写程序了：

```
void f()
{
    My_lib::String s = "Byron";    // 寻找 My_lib::His_string::String
    // ...
}

using namespace My_lib;

void g(Vector<String>& vs)
{
    // ...
    my_fct(vs[5]);
    // ...
}
```

如果一个显式限定的名字（如本例中的 My_lib::String）并未声明在所限定的名字空间中，编译器就会在 using 指示提及的名字空间（如 His_string）中寻找它。

只有当我们需要定义某些实体时，才真的需要了解一个实体的真正名字空间：

```
void My_lib::fill(char c)          // 错误：My_lib 中并未声明 fill()
{
    // ...
}

void His_string::fill(char c)      // 正确：fill() 在 His_string 中声明
{
    // ...
}

void My_lib::my_fct(String& v)// 正确：String 为 My_lib::String，表示 His_string::String
{
    // ...
}
```

理想情况下，一个名字空间应该：

[1] 表达一组逻辑相关的特性；

[2] 不会让用户访问不相关的特性；

[3] 不会给用户增加符号表示上的严重负担。

结合 #include 机制（见 15.2.2 节），本节和下一节中介绍的名字空间组合技术可为实现这三点要求提供强有力的支持。

14.4.4 组合与选择

组合机制（使用 using 指示）与选择机制（using 声明）的结合满足了现实世界中大多数应用实例对灵活性的需求。使用这些机制，我们在访问各种特性时可解决它们的组合所引起的名字冲突和二义性。例如：

```
namespace His_lib {
    class String { /* ... */ };
    template<class T>
        class Vector { /* ... */ };
    // ...
}

namespace Her_lib {
    template<class T>
        class Vector { /* ... */ };
    class String { /* ... */ };
    // ...
}

namespace My_lib {
    using namespace His_lib;    // His_lib 中的所有实体
    using namespace Her_lib;    // Her_lib 中的所有实体

    using His_lib::String;      // 解决潜在冲突：使用 His_lib 中的版本
    using Her_lib::Vector;      // 解决潜在冲突：使用 Her_lib 中的版本

    template<class T>
        class List { /* ... */ };      // 其他内容
    // ...
}
```

当编译器在一个名字空间中进行查找时，在其中显式声明的名字（包括用 using 声明声明的名字）较之通过 using 指示变为可见的名字优先级更高（见 14.4.1 节）。因此，My_lib 的使用者会看到 String 和 Vector 的名字冲突顺利解决，分别使用了 His_lib::String 和 Her_lib::Vector。而 List 则默认解析为 My_lib::List，而不管 His_lib 或 Her_lib 是否提供了 List。

当我将一个名字纳入一个新的名字空间中时，我通常倾向于不改变它的名字。这样，我就不必对同一个实体记忆两个不同的名字了。但有时起一个新的名字是必需的或者更好的。例如：

```
namespace Lib2 {
    using namespace His_lib;    // His_lib 中的所有实体
    using namespace Her_lib;    // Her_lib 中的所有实体

    using His_lib::String;      // 解决潜在冲突：使用 His_lib 中的版本
    using Her_lib::Vector;      // 解决潜在冲突：使用 Her_lib 中的版本

    using Her_string = Her_lib::String;              // 重命名
    template<class T>
        using His_vec = His_lib::Vector<T>;          // 重命名

    template<class T>
        class List { /* ... */ };      // 其他内容
    // ...
}
```

C++ 语言并未提供重命名的通用机制，但对于类型和模板，我们可以通过使用 using 引入别名来实现重命名（见 3.4.5 节和 6.5 节）

14.4.5　名字空间和重载

函数重载（见 12.3 节）机制是跨越名字空间的。这一点很重要，它允许我们以最小的代

码修改代价将现有的库改进为使用名字空间的版本。例如：

```
// 旧 A.h:
    void f(int);
    // ...

// 旧 B.h:
    void f(char);
    // ...

// 旧 user.c:
    #include "A.h"
    #include "B.h"

    void g()
    {
        f('a'); // 调用 B.h 中的 f()
    }
```

我们不必修改实际代码即可将此程序改为使用名字空间的版本：

```
// 新 A.h:

    namespace A {
        void f(int);
        // ...
    }

// 新 B.h:

    namespace B {
        void f(char);
        // ...
    }

// 新 user.c:

    #include "A.h"
    #include "B.h"

    using namespace A;
    using namespace B;

    void g()
    {
        f('a');      // 某证 B.h 中为 f()
    }
```

假如我们希望保持 user.c 完全不变，就要将 using 指示放在头文件中。但是，通常最好避免将 using 指示放在头文件中，因为这样做会大大增加名字冲突的机会。

这种重载规则还提供了一种扩展库的机制。例如，人们经常奇怪，为了使用标准库算法操作容器，为什么必须要显式指定一个序列。例如：

```
sort(v.begin(),v.end());
```

而不能直接对容器操作呢：

```
sort(v);
```

原因在于我们对通用性的追求（见 32.2 节）。但操作容器已是目前为止最常见的情况了，我们可以这样实现直接适用容器的算法：

```
#include<algorithm>

namespace Estd {
    using namespace std;
    template<class C>
        void sort(C& c) { std::sort(c.begin(),c.end()); }
    template<class C, class P>
        void sort(C& c, P p) { std::sort(c.begin(),c.end(),p); }
}
```

Estd（我的“扩展 std”）提供了我们经常需要的容器版本的 sort()。这当然是通过使用 <algorithm> 中的 std::sort() 来实现的。我们可以这样使用它：

```
using namespace Estd;

template<class T>
void print(const vector<T>& v)
{
    for (auto& x : v)
        cout << v << ' ';
    cout << '\n';
}
void f()
{
    std::vector<int> v {7, 3, 9, 4, 0, 1};

    sort(v);
    print(v);
    sort(v,[](int x, int y) { return x>y; });
    print(v);
    sort(v.begin(),v.end());
    print(v);
    sort(v.begin(),v.end(),[](int x, int y) { return x>y; });
    print(v);
}
```

名字空间查找规则和模板重载规则确保我们能找到并调用正确的 sort() 版本，得到所期望的结果：

```
0 1 3 4 7 9
9 7 4 3 1 0
0 1 3 4 7 9
9 7 4 3 1 0
```

如果我们删掉 Estd 中的 using namespace std;，这个例子仍能正常运行，因为通过参数依赖查找（见 12.2.4 节）还是能找到 std 的 sort()。但对于 std 之外我们自己定义的容器，就再也找不到标准 sort() 了。

14.4.6 版本控制

对很多类型的接口而言，最苛刻的测试就是应对一系列的新版本。考虑一个广为使用的接口，如一个 ISO C++ 标准头文件。经过一段时间，标准委员会会定义新的版本，例如 C++98 头文件的 C++11 版本。新版本可能增加了函数、重命名了类、删除了私有扩展（本

不该包含的内容），改变了类型、修改了模板。现实应用中可能有数亿行代码使用了旧版头文件，而新版本的实现者不可能看到或修改这些代码，这就给实现者的生活增加了很多“乐趣”。不消说，破坏这些旧代码会引起强烈不满，后果与不能提供更好的新版本一样严重。除少数情况外，到目前为止已介绍的名字空间特性已能处理这类问题，但当涉及的代码量非常庞大时，“少数情况”仍意味着大量代码。为此，C++提供了一种在两个版本间进行选择的机制，可以简单明确地保证用户看到其中一个特定版本，这就是内联名字空间（inline namespace）：

```
namespace Popular {

    inline namespace V3_2 { // V3_2 提供了 Popular 的默认含义
        double f(double);
        int f(int);
        template<class T>
            class C { /* ... */ };
    }
    namespace V3_0 {
        // ...
    }
    namespace V2_4_2 {
        double f(double);
        template<class T>
            class C { /* ... */ };
    }
}
```

在本例中，Popular 包含三个子名字空间，每一个都定义了一个 Popular 版本。inline 指出 V3_2 是 Popular 的默认含义。因此我们可以编写如下代码：

```
using namespace Popular;

void f()
{
    f(1);           // Popular ::V3_2::f(int)
    V3_0::f(1);         // Popular ::V3_0::f(double)
    V2_4_2::f(1);   // Popular ::V2_4_2::f(double)
}

template<class T>
Popular::C<T*> { /* ... */ };
```

这种 inline namespace 方法是侵入式的。即，为了改变默认版本（子名字空间），必须修改头文件源码。而且，简单地使用这种方法处理版本问题需要复制大量代码（不同版本中的共同代码）。但是，使用 #include 技巧可将复制降到最低。例如：

```
// 文件 V3_common.h:
    // ... 大量声明 ...

// 文件 V3_2.h:

    namespace V3_2 {  // V3_2 提供了 Popular 的默认语义
        double f(double);
        int f(int);
        template<class T>
            class C { /* ... */ };
```

```
        #include "V3_common"
    }

// 文件 V3_0.h:

    namespace V3_0 {
        #include "V3_common"
    }

// 文件 Popular.h:

    namespace Popular {
        inline
        #include "V3_2.h"
        #include "V3_0.h"
        #include "V2_4_2.h"
    }
```

我不推荐这么复杂地使用头文件，除非真的必要。这个例子多次违反了不要包含头文件到非局部作用域的原则和语法构造不要跨越文件边界（使用 inline）的原则，请参考 15.2.2 节。可悲的是，我还曾看到过更糟糕的代码。

大多数情况下，我们可以用侵入性不那么强的方法来实现版本控制。我能想到的唯一一个完全不可能用其他方法实现的例子是显式使用名字空间名的模板（如，Popular::C<T*>）的特例化。但是，很多时候“大多数情况可解决”并不足够好。而且，组合多种技术而得的方案是否完全正确就不那么显然了。

14.4.7　名字空间嵌套

名字空间的一种明显用法是将一组完整的声明和定义封装在一个独立的名字空间中：

```
namespace X {
    // ... 我的所有声明 ...
}
```

这些声明中通常会包含名字空间。名字空间是允许嵌套的，这一规则一方面是出于实践上的考虑，另一方面是因为语法结构应该嵌套，除非有强有力的理由不这么做。例如：

```
void h();

namespace X {
    void g();
    // ...
    namespace Y {
        void f();
        void ff();
        // ...
    }
}
```

对这段代码会应用通常的作用域和限定规则：

```
void X::Y::ff()
{
    f(); g(); h();
}
```

```
void X::g()
{
    f();        // 错误：X中无f()
    Y::f();     // 正确
}

void h()
{
    f();            // 错误：无全局f()
    Y::f();         // 错误：无全局Y
    X::f();         // 错误：X中无f()
    X::Y::f();      // 正确
}
```

关于标准库中的名字空间嵌套的例子，请见 chrono（35.2 节）和 rel_ops（35.5.3 节）。

14.4.8 无名名字空间

有时将一组声明封装在一个名字空间中只是为了防止名字冲突，即，目的是保持代码的局部性而非为用户提供一个接口。例如：

```
#include "header.h"
namespace Mine {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

由于我们不希望在局部环境之外的代码看到名字 Mine，那么创建这么一个全局名字就变成了一个麻烦，可能意外地与其他名字冲突。既然如此，我们可以简单地不为名字空间命名：

```
#include "header.h"
namespace {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

显然，必须提供某种方法实现从无名名字空间之外访问其成员。为此，每个无名名字空间都有一个隐含的 using 指示。前面这个声明等价于：

```
namespace $$$ {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
using namespace $$$;
```

其中 $$$ 是此名字空间所在作用域中的一个独一无二的名字。特别是，不同编译单元中的无名名字空间是不同的。如我们所期望，我们无法从另一个编译单元中为一个无名名字空间的成员命名。

14.4.9 C 头文件

考虑标准的“第一个 C 程序”：

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
}
```

打破这个程序不是好主意，将标准库变成特殊情况也不是好主意。因此，名字空间语言规则的设计原则是：不用名字空间编写程序应比较容易，用名字空间改写程序使其更为结构化也应很简单。实际上，计算器程序（见 10.2 节）就是这样一个例子。

在名字空间中提供 C I/O 特性的一种方法是将 C 头文件 stdio.h 中的声明置于名字空间 std 中：

```
// cstdio:

    namespace std {
        int printf(const char* ... );
        // ...
    }
```

有了这个 <cstdio>，我们就可以通过一个 using 指示提供向后兼容的能力：

```
// stdio.h:

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

有了这个 <stdio.h>，Hello, world! 程序就能编译通过了。但不幸的是，using 指示将 std 中的所有名字都暴露在全局名字空间中。例如：

```
#include<vector>    // 小心避免污染全局名字空间
vector v1;          // 错误：全局作用域中没有 ''vector''
#include<stdio.h>   // 头文件中包含一个 ''using namespace std;''
vector v2;          // 糟糕：不能正常运行
```

因此 C++ 标准要求 <stdio.h> 只将来自 <cstdio> 的名字置于全局作用域中。我们可以为 <cstdio> 中的每个名字提供一个 using 声明：

```
// stdio.h:

    #include<cstdio>
    using std::printf;
    // ...
```

这种方法的另一个优点是 printf() 的 using 声明能防止用户（意外地或故意地）在全局作用域中定义一个非标准 printf()。我将非局部 using 指示主要看作一种代码转换工具。我也将其用于一些重要的基础库，如 ISO C++ 标准库（std）。大多数从外部访问名字空间中名字的代码都可以用显式限定和 using 声明更清晰地表达。

名字空间和链接之间的关系将在 15.2.5 节中介绍。

14.5　建议

［1］ 用名字空间表达逻辑结构；14.3.1 节。

［2］ 将除 main() 之外的所有非局部名字都置于名字空间中；14.3.1 节。

［3］ 设计一个名字空间，以便能方便地使用它避免意外访问到不相关的名字空间；

14.3.3 节。

[4] 不要为名字空间起非常短的名字；14.4.2 节。

[5] 如必要，使用名字空间别名为长名字空间名提供简写；14.4.2 节。

[6] 不要给你的名字空间的使用者增加太多符号表示上的负担；14.2.2 节和 14.2.3 节。

[7] 为接口和实现使用分离的名字空间；14.3.3 节。

[8] 当定义名字空间成员时使用 Namespace::member 表示方式；14.4 节。

[9] 用 inline 名字空间支持版本控制；14.4.6 节。

[10] 将 using 指示用于代码转换、用于基础库（如 std）以及用于局部作用域内；14.4.9 节。

[11] 不要将 using 指示放在头文件中；14.2.3 节。

第 15 章
The C++ Programming Language, Fourth Edition

源文件与程序

形式必须服从功能。

——勒·柯布西耶

- 分离编译
- 链接
  文件内名字；头文件；单一定义规则；标准库头文件；链接非 C++ 代码；链接和函数指针
- 使用头文件
  单头文件组织；多头文件组织；包含保护
- 程序
  非局部变量初始化；初始化和并发；程序终止
- 建议

15.1 分离编译

任何实际程序都由很多逻辑上分离的部分（如名字空间，见第 14 章）组成。为了更好地管理这些组成部分，我们可以将程序表示为一组（源码）文件，其中每个文件包含一个或多个逻辑组件。我们的任务是为程序设计一个（文件集合）物理结构，使得能以一种一致、易理解和灵活的方式表示这些逻辑组件。特别是，我们以接口（如函数声明）与实现（如函数定义）的完全分离为目标。文件是我们传统的（文件系统中的）存储单元，也是传统的编译单元。确实存在一些系统并不以文件集合的方式存储、编译 C++ 程序并将其呈现给程序员。但是，本书重点讨论采用传统文件组织方式的系统。

将完整程序放在一个文件中通常是不可能的。特别是，标准库和操作系统的代码通常不会以源码的形式提供，不会作为用户源程序的一部分。对实际规模的应用程序而言，即使只是将所有用户自己的代码放在单一文件中也是不现实、不方便的。将程序组织为文件的方式能帮助我们强调程序的逻辑结构，帮助程序的读者更好地理解程序，还能帮助编译器强化逻辑结构。如果编译单元是文件，那么对一个文件或它所依赖的程序有任何修改（不管多小）都要重新编译整个文件。因此，即使是中等规模的程序，将其划分为适当规模的文件都能显著节省重编译时间。

当用户将一个源文件（source file）提交给编译器后，首先对文件进行预处理，即，处理宏（见 12.6 节）以及将 #include 指令指定的头文件包含进来（见 2.4.1 节和 15.2.2 节）。预处理的结果称为编译单元（translation unit）。编译单元是编译器真正处理的内容，也是 C++ 语言规则所描述的内容。在本书中，仅当需要区分程序员所看到的内容和编译器所处理的内容时才会区分源文件和编译单元。

为了实现分离编译，程序员所编写的声明必须提供足够的类型信息，以便能够在与程

序剩余部分隔离的情况下分析一个编译单元。如果程序由很多分离编译的部分组成，那么其中的声明必须是一致的，这一点与只包含单一源文件的程序中的声明保持一致的方式完全一样。你的系统会有工具确保这一点。特别是，链接器可以检查出很多种不一致问题。链接器（linker）是将分离编译的多个部分绑定在一起的程序。编译器有时也被（令人迷惑地）称为加载器（loader）。链接可以在程序开始运行前全部完成。但也可以在程序运行中将新代码添加进来（“动态链接”）。

程序的源文件组织通常称为其物理结构（physical structure）。我们必须按照程序的逻辑结构将其在物理上划分为独立的文件。源文件的组织也应遵循程序的名字空间组合所体现的依赖关系。但是，程序的逻辑结构和物理结构不必相同。例如，用多个源文件保存单一名字空间中的函数、在单一源文件中保存一组名字空间定义以及将一个名字空间的定义散布到多个文件中（见 14.3.3 节）等技术都是很有帮助的。

在本节中，我们将首先介绍与链接相关的技术，然后讨论两种将桌面计算器程序（见 10.2 节和 14.3.2 节）划分为源文件的方法。

15.2　链接

除非已显式声明为局部名字，否则函数名、类名、模板名、变量名、名字空间名、枚举名以及枚举值名的使用必须跨所有编译单元保持一致。

程序员必须保证每个名字空间、类、函数等必须在其出现的每个编译单元中都正确声明，且对应相同实体的声明都是一致的。例如，考虑下面两个文件：

```
// file1.cpp:
    int x = 1;
    int f() { /* 进行一些操作 */ }

// file2.cpp:
    extern int x;
    int f();
    void g() { x = f(); }
```

file2.cpp 中的 g() 使用的 x 和 f() 就是 file1.cpp 中所定义的实体。关键字 extern 指出 file2.cpp 中 x 的声明仅仅是一个声明而已，而非一个定义（见 6.3 节）。假如 x 已初始化，extern 将会被忽略，因为带初始值的声明总是被看作一个定义。对象在程序中只能定义一次，它可以声明很多次，但类型必须完全一致。例如：

```
// file1.cpp:
    int x = 1;
    int b = 1;
    extern int c;

// file2.cpp:
    int x;                  // 意味着 "int x = 0;"
    extern double b;
    extern int c;
```

这个程序有 3 个错误：x 被定义了两次，b 的两次声明类型不一致，c 被声明了两次但没有被定义。如果编译器只能同时处理一个文件，就无法检查出这些错误（链接错误）。但大多数这种错误都可以被链接器检查出来。例如，我所知的所有正确的 C++ 实现都能检测出 x 的双重定义。但是，流行的 C++ 实现都捕获不到 b 的声明不一致的问题，遗漏 c 的定义这

一错误通常也只有在 c 被使用时才能捕获到。

注意，如果全局作用域中或名字空间中的变量定义不带初始值，则该变量会使用默认初始值（见 6.3.5.1 节）。非 static 局部变量或创建在自由存储上的对象（见 11.2 节）则不会使用默认初始值。

在类体外，实体必须先声明后使用（见 6.3.4 节）。例如：

```
// file1.cpp:
    int g() { return f()+7; }    // 错误：f()（尚）未声明
    int f() { return x; }        // 错误：x（尚）未声明
    int x;
```

如果一个名字在其定义处之外的编译单元中也可以使用，我们称其具有外部链接（external linkage）。前一个例子中的所有名字都具有外部链接。如果一个名字只能在其定义所在的编译单元中被引用，我们称其具有内部链接（internal linkage）。例如：

```
static int x1 = 1;          // 内部链接：其他编译单元中不可访问
const char x2 = 'a';        // 内部链接：其他编译单元中不可访问
```

在名字空间作用域（包括全局作用域，见 14.2.1 节）中使用关键字 static（有些不合逻辑）表示“不能在其他源文件中访问”（即内部链接）。如果你希望在其他源文件中也能访问 x1（“具有外部链接”），就应去掉 static。关键字 const 暗示默认内部链接，因此如果你希望 x2 具有外部链接，就需要在其定义前加上 extern：

```
int x1 = 1;                 // 外部链接：在其他编译单元中可访问
extern const char x2 = 'a'; // 外部链接：在其他编译单元中可访问
```

链接器看不到的名字，例如局部变量名，被称为*无链接*（no linkage）。

inline 函数（见 12.1.3 节和 16.2.8 节）在其应用的所有编译单元中都必须有完全等价的定义（见 15.2.3 节）。因此，下面这个例子不仅风格糟糕，而且是不合法的：

```
// file1.cpp:
    inline int f(int i) { return i; }

// file2.cpp:
    inline int f(int i) { return i+1; }
```

不幸的是，C++ 实现很难捕获这种错误。而下面的例子中外部链接和内联的组合虽然完全符合逻辑，但却是被禁止的：

```
// file1.cpp:
    extern inline int g(int i);
    int h(int i) { return g(i); }  // 错误：此编译单元中无 g() 定义

// file2.cpp:
    extern inline int g(int i) { return i+1; }
    // ...
```

我们可以通过使用头文件来保持 inline 函数定义的一致性（见 15.2.2 节）。例如：

```
// h.h:
    inline int next(int i) { return i+1; }

// file1.cpp:
    #include "h.h"
    int h(int i) { return next(i); }    // 正确

// file2.cpp:
```

```
#include "h.h"
// ...
```

默认情况下，名字空间中的 const 对象（见 7.5 节）、constexpr 对象（见 10.4 节）、类型别名（见 6.5 节）以及任何声明为 static 的实体（见 6.3.4 节）都具有内部链接。因此，下面这个例子是合法的（虽然可能让人困惑）：

```
// file1.cpp:
    using T = int;
    const int x = 7;
    constexpr T c2 = x+1;

// file2.cpp:
    using T = double;
    const int x = 8;
    constexpr T c2 = x+9;
```

为确保一致性，应该将别名、const 对象、constexpr 对象和 inline 函数放置在头文件中（见 15.2.2 节）。

我们可以通过显式声明为一个 const 对象赋予外部链接：

```
// file1.cpp:
    extern const int a = 77;

// file2.cpp:
    extern const int a;

    void g()
    {
        cout << a << '\n';
    }
```

在本例中，g() 会打印 77。

管理模板定义的技术将在 23.7 节中介绍。

15.2.1 文件内名字

我们一般最好避免使用全局变量，因为这会引起维护问题。特别是，我们很难掌握全局变量在程序中什么位置使用，在多线程程序中全局变量还可能引起数据竞争（见 41.2.4 节），这些都会导致隐藏很深的错误出现。

将变量放在名字空间中会有些帮助，但仍可能引起数据竞争。

如果必须使用全局变量，至少应限制它们只在单一源文件中使用，有两种方法实现这种限制：

[1] 将声明放在无名名字空间中。

[2] 声明实体时使用 static。

使用无名名字空间（见 14.4.8 节）可以令名字成为编译单元的局部名字。无名名字空间的效果非常像内部链接。例如：

```
// file1.cpp:
    namespace {
        class X { /* ... */ };
        void f();
        int i;
```

```
        // ...
    }

// file2.cpp:
    class X { /* ... */ };
    void f();
    int i;
    // ...
```

file1.cpp 中的 f() 与 file2.cpp 中的 f() 不是同一个函数。如果一个名字是一个编译单元的局部名字，我们又用它命名别处的一个具有外部链接的实体，我们就是自找麻烦。

关键字 static（令人困惑地）表示“使用外部链接”（见 44.2.3 节）。这是早期 C 语言遗留下来的一个问题。

15.2.2　头文件

同一个对象、函数、类等的所有声明都要保持类型一致。因此，提交给编译器并随后链接在一起的源码必须保持一致。实现不同编译单元声明一致性的一种不完美但很简单的方法是：在包含可执行代码或数据定义的源文件中 #include 包含接口信息的头文件（header file）。

#include 机制是一种文本处理方式——将源程序片段收集起来形成单一的编译单元（文件）。考虑下面的语句：

```
#include "to_be_included"
```

这条 #include 指令将它所在的这一行替换为文件 to_be_included 的内容。to_be_included 的内容应该是 C++ 源码，因为编译器会继续处理替换后的结果。

包含标准库头文件时应使用尖括号 < 和 > 包围文件名，而不是引号。例如：

```
#include <iostream>          // 来自标准库头文件目录
#include "myheader.h"        // 来自当前目录
```

不幸的是，在包含指令中，<> 或 "" 内的空格不会被忽略：

```
#include < iostream >        // 查找不到 <iostream>
```

这样，每当源文件被其他文件包含时，（在编译那个文件时）它就要被重新编译一次。这看起来很浪费，但源文件可能密集包含大量程序接口信息，而编译器只需分析真正用到的细节（例如，只有在模板实例化时才会完整分析模板体，见 26.3 节）。而且，大多数现代 C++ 实现都提供某种形式的（隐式或显式的）头文件预编译机制，可将重复编译同一个头文件的工作量降到最低。

一般原则是，头文件可包含：

具名的名字空间	namespace N { /* ... */ }
inline 名字空间	inline namespace N { /* ... */ }
类型定义	struct Point { int x, y; };
模板声明	template<typename T> class Z;
模板定义	template<typename T> class V { /* ... */ };
函数声明	extern int strlen(const char*);
inline 函数定义	inline char get(char* p) { /* ... */ }

（续）

constexpr 函数定义	constexpr int fac(int n) { return (n<2) ? 1 : n*fac(n-1); }
数据声明	extern int a;
const 定义	const float pi = 3.141593;
constexpr 定义	constexpr float pi2 = pi*pi;
枚举	enum class Light { red, yellow, green };
名字声明	class Matrix;
类型别名	using value_type = long;
编译时断言	static_asser t(4<=sizeof(int),"small ints");
包含指令	#include<algorithm>
宏定义	#define VERSION 12.03
条件编译指令	#ifdef __cplusplus
注释	/* check for end of file */

上述关于头文件可以包含什么内容的原则并不是 C++ 语言所要求的。它只是一种用 #include 机制表达程序物理结构的合理方法。反过来，头文件中不应包含以下内容：

普通函数定义	char get(char* p) {return *p++; }
数据定义	int a;
集合定义	short tbl[] = { 1, 2, 3 };
无名名字空间	short tbl[] = { 1, 2, 3 };
using 指示	using namespace Foo;

如果一个头文件中含有这些定义，那么包含它就会导致错误或混乱（在使用 using 指示的情况下）。头文件一般采用 .h 后缀，包含函数或数据定义的文件则用 .cpp 后缀，因此它们通常分别被称为“.h 文件”和“.cpp 文件”。其他常用的后缀包括 .c、.C、.cxx、.cc、.hh 和 .hhp。你的编译器手册会详细说明后缀问题。

建议将简单常量定义放在头文件中，但不将集合定义放在头文件中，其原因是 C++ 实现很难避免多个编译单元中重复的集合定义。而且，简单情况更常见，因而对生成高质量代码更为重要。

使用 #include 时过分卖弄聪明是不明智的。我的建议是：

- 只 #include 头文件（不要 #include “包含变量定义和非 inline 函数的普通源码”）。
- 只 #include 完整的声明和定义。
- 只在全局作用域、链接说明块及名字空间定义（转换旧代码时，见 15.2.4 节）中 #include 头文件。
- 将所有 #include 放在其他代码之前，以尽量减少无意造成的依赖关系。
- 避免使用宏技巧。
- 尽量减少在头文件中使用非局部的名字（特别是别名）。

我们可能会间接 #include 一个从来没听说过的头文件，其中定义的宏令程序中的某个名字被替换为完全不同的内容（并非是我们所希望的），从而导致错误，我最不喜欢的一项工作就是查找这种错误。

15.2.3　单一定义规则

每个给定类、枚举和模板等在程序中都只能定义一次。

从实践角度来看这一规则意味着一个定义（比如说一个类的定义）只能唯一存在于某个单一文件中。不幸的是，C++ 语言规则不可能这么简单。例如，一个类的定义可能是由宏扩展形成的（啊！），而且一个类的定义也可以通过 #include 指令（15.2.2 节）包含到两个源文件中。更糟糕的是，“文件”概念并不是 C++ 语言定义的一部分，不将程序保存在源文件中的 C++ 实现也是存在的。

因此，C++ 标准中规定类、模板等的定义必须唯一，这一规则是通过一种更为复杂、更为微妙的方式描述的。它通常被称为单一定义规则（one-definition rule，ODR）。即，一个类、模板或内联函数的两个定义可被接受、被认为是相同唯一定义，当且仅当如下条件成立：

［1］ 它们出现在不同的编译单元中，且

［2］ 它们的源码逐单词对应，完全一样，且

［3］ 这些单词在两个编译单元中的含义完全一样。

例如：

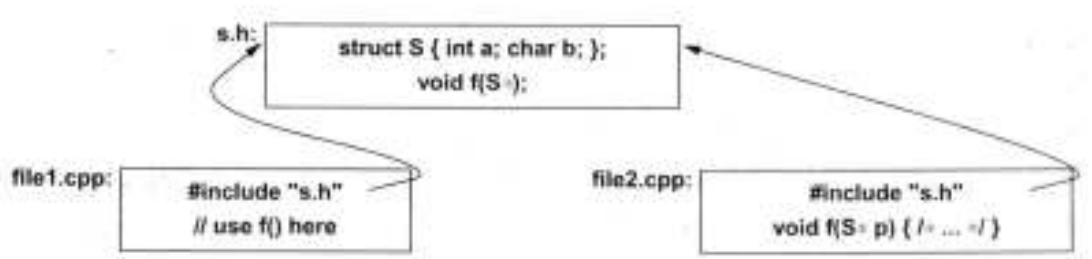
```
// file1.cpp:
    struct S { int a; char b; };
    void f(S*);

// file2.cpp:
    struct S { int a; char b; };
    void f(S* p) { /* ... */ }
```

ODR 认为这个例子是合法的，S 在两个源文件中表示相同的类。但是，像这样一个定义写两次是不明智的。file2.cpp 的维护者会自然地认为 file2.cpp 中的 S 是 S 的唯一定义，从而随意地修改它。这可能引起难以检查的错误。

ODR 的设计意图是允许在不同编译单元中包含来自同一个公共源文件的类定义。例如：

```
// s.h:
    struct S { int a; char b; };
    void f(S*);

// file1.cpp:
    #include "s.h"
    // 使用 f()

// file2.cpp:
    #include "s.h"
    void f(S* p) { /* ... */ }
```

其关系可图示如下：



有三种情况会违反 ODR：

```
// file1.cpp:
    struct S1 { int a; char b; };

    struct S1 { int a; char b; };    // 错误：双重定义
```

这段代码是错误的，因为在单一编译单元中一个 struct 不能定义两次。

```
// file1.cpp:
    struct S2 { int a; char b; };

// file2.cpp:
    struct S2 { int a; char bb; };   // 错误
```

这段代码也是错的，S2 的两个定义有一个成员名不同。

```
// file1.cpp:
    typedef int X;
    struct S3 { X a; char b; };

// file2.cpp:
    typedef char X;
    struct S3 { X a; char b; }; // 错误
```

本例中的两个 S3 定义是逐单词一致的，但这个程序仍然非法，因为两个文件中名字 X 的含义被偷偷地设置为不一致了。

检查多个分离编译单元中的类定义是否一致的问题已经超出了大多数 C++ 实现的能力。因此，违反 ODR 的声明是微妙错误之源。不幸的是，将共享定义放置在头文件然后 #include 头文件的技术并不能防止最后一种违反 ODR 的形式。局部类型别名和宏会改变 #include 声明的含义：

```
// s.h:
    struct S { Point a; char b; };

// file1.cpp:
    #define Point int
    #include "s.h"
    // ...

// file2.cpp:
    class Point { /* ... */ };
    #include "s.h"
    // ...
```

防止这种错误的最好方法是令头文件尽可能地自包含。例如，如果上例中类 Point 声明在头文件 s.h 中，错误就能检查出来了。

只要保持不违反 ODR，一个模板定义就可以在多个编译单元中被 #include，甚至是函数模板定义以及包含成员函数定义的类模板都可以。

15.2.4　标准库头文件

标准库特性是通过一组标准头文件提供的（见 4.1.2 节和 30.2 节）。标准头文件不需要后缀，它们能表明头文件的身份是因为使用了 #include<...> 语法而不是 #include"..."。缺少 .h 后缀并不意味着头文件在存储上有什么特殊之处。<map> 这样的头文件通常保存为

一个名为 map.h 的文本文件，存储在某个标准目录中。另一方面，C++ 标准也不要求标准头文件必须以常规方式保存，它允许 C++ 实现利用所掌握的标准库定义的知识来优化标准库的实现以及标准头文件的处理。例如，一个 C++ 实现可能内置了标准数学库的功能（见 40.3 节），从而将 #include<cmath> 当作触发标准数学函数的开关而无须读取任何文件。

每个 C 标准库头文件 <X.h> 都有一个对应的标准 C++ 头文件 <cX>。例如，#include <cstdio> 提供了 #include<stdio.h> 的功能。一个典型的 stdio.h 示例可能是这样实现的：

```
#ifdef __cplusplus          // 只用于 C++ 编译器（见 15.2.5 节）
namespace std {             // 标准库定义在名字空间 std 中（见 4.1.2 节）
extern "C" {                // stdio 函数采用 C 链接（见 15.2.5 节）
#endif
    /* ... */
    int printf(const char*, ...);
    /* ... */
#ifdef __cplusplus
}
}
// ...
using std::printf;    // 令 printf 在全局名字空间中可用
// ...
#endif
```

即，真正的声明（大多数）是 C++ 和 C 共享的，但必须解决链接和名字空间问题，C 和 C++ 才能共享一个头文件。宏 __cplusplus 是由 C++ 编译器定义的（见 12.6.2 节），可用来区分 C++ 代码和用于 C 编译器的代码。

15.2.5　链接非 C++ 代码

C++ 程序通常包含用其他语言（如 C 或 Fortran）编写的部分。类似的，C++ 代码片段作为主要由某种其他语言（如 Python 或 Matlab）编写的程序的一部分也很常见。用不同语言编写的程序片段间的协同可能很困难，甚至用相同语言编写但用不同编译器编译的程序片段间也很难协同。例如，不同语言和同一种语言的不同实现可能在如何用机器的寄存器保存参数、参数在栈中的布局、字符串和整数等内置类型的内存布局、编译器传递给链接器的名字的格式以及链接器要求的内存检查等方面都有所不同。为了帮助解决此问题，我们可以指定 extern 声明中使用哪种链接（linkage）规范。例如，下面的代码声明了 C 和 C++ 标准库函数 strcpy() 并指定它采用 C 链接规范（系统相关）：

```
extern "C" char* strcpy(char*, const char*);
```

此声明的效果与下面的“普通”声明是不同的

```
extern char* strcpy(char*, const char*);
```

但差别只是调用 strcpy() 时所采用的链接规范不同。

因为 C 和 C++ 关系紧密，extern "C" 指示特别有用。需要注意的是，extern "C" 中的 C 表示的是链接规范而非语言。extern "C" 通常用于将函数链接到恰好符合 C 实现规范的 Fortran 和汇编程序。

一个 extern "C" 指示（仅）指出链接规范，它不影响函数调用的语义。特别是，声明为 extern "C" 的函数仍然遵守 C++ 类型检查和参数转换规则而不是弱一些的 C 规则。例如：

```
extern "C" int f();

int g()
{
    return f(1);          // 错误：不需要参数
}
```

为大量声明添加 extern "C" 很令人厌烦。因此，C++ 提供了一种机制为一组声明指定链接规范。例如：

```
extern "C" {
    char* strcpy(char*, const char*);
    int strcmp(const char*, const char*);
    int strlen(const char*);
    // ...
}
```

这种构造通常称为链接块（linkage block），甚至可用来封装完整 C 头文件，从而使之适用于 C++ 程序。例如：

```
extern "C" {
#include <string.h>
}
```

程序员常常用这种技术从 C 头文件生成 C++ 头文件。另一种创建 C 和 C++ 公用头文件的技术是条件编译（见 12.6.1 节）：

```
#ifdef __cplusplus
extern "C" {
#endif
    char* strcpy(char*, const char*);
    int strcmp(const char*, const char*);
    int strlen(const char*);
    // ...
#ifdef __cplusplus
}
#endif
```

我们用预定义宏 __cplusplus（见 12.6.2 节）确保当头文件用在 C 程序中时，文件中的 C++ 构造会被忽略掉。

任何声明都可以放在链接块中：

```
extern "C" {          // 可以放置任何声明，例如：
    int g1;           // 定义
    extern int g2;    // 声明，非定义
}
```

特别是，变量的作用域和存储类（见 6.3.4 节和 6.4.2 节）不会受到影响，因此 g1 仍是一个全局变量，而且该语句仍然是一个定义而不仅是声明。为了声明一个变量但不定义它，你必须在声明中直接使用 extern 关键字。例如：

```
extern "C" int g3;          // 声明，非定义
extern "C" { int g4; }      // 定义
```

这个例子初看会让人觉得有些奇怪。但是，这样的结果其实很简单：当我们向一个 extern 声明添加 "C" 时，其含义不应被改变；同样，当我们将一个文件封装入一个链接块时，其含义也不应改变。

采用 C 链接规范的名字可声明在名字空间中。名字空间会影响在 C++ 程序中访问名字的方式，但不会影响链接器处理名字的方式。std 中的 printf() 是一个典型的例子：

```
#include<cstdio>

void f()
{
    std::printf("Hello, ");      // 正确
    printf("world!\n");          // 错误：无全局 printf()
}
```

即使被称为 std::printf，它仍然是那个古老的 C printf()（见 43.3 节）。

注意，这一机制允许我们选择在一个名字空间中包含采用 C 链接的库，而不会污染全局名字空间。不幸的是，我们不能同样灵活地在一个头文件中在全局名字空间中定义采用 C++ 链接的函数。原因在于 C++ 实体的链接必须考虑名字空间的因素，以便生成的目标文件能反映是否使用了名字空间。

15.2.6　链接和函数指针

当在一个程序中混合 C 和 C++ 代码片段时，我们有时希望将一种语言定义的函数指针传递给另一种语言定义的函数。如果两种语言的实现共享链接规范和调用机制，这种函数指针的传递就很简单。但是，这种通用性一般很难保证，因此我们必须小心确保一个函数的调用方式符合函数自身的设计预期。

如果在声明中指定了链接方式，则此链接方式会应用于声明中涉及的所有函数类型、函数名和变量名。这令各种各样奇怪的（有时也是必需的）链接方式组合成为可能。例如：

```
typedef int (*FT)(const void*, const void*);                    // FT 采用 C++ 链接

extern "C" {
    typedef int (*CFT)(const void*, const void*);               // CFT 采用 C 链接
    void qsort(void* p, size_t n, size_t sz, CFT cmp);          // cmp 采用 C 链接
}

void isort(void* p, size_t n, size_t sz, FT cmp);               // cmp 采用 C++ 链接
void xsort(void* p, size_t n, size_t sz, CFT cmp);              // cmp 采用 C 链接
extern "C" void ysort(void* p, size_t n, size_t sz, FT cmp);    // cmp 采用 C++ 链接

int compare(const void*, const void*);                          // compare() 采用 C++ 链接
extern "C" int ccmp(const void*, const void*);                  // ccmp() 采用 C 链接

void f(char* v, int sz)
{
    qsort(v,sz,1,&compare);  // 错误
    qsort(v,sz,1,&ccmp);     // 正确

    isort(v,sz,1,&compare);  // 正确
    isort(v,sz,1,&ccmp);     // 错误
}
```

如果一个实现中 C 和 C++ 使用相同的调用规范，那么本例中标记为错误的代码有可能被接受，被当作语言的扩展。但是，即使对兼容 C 和 C++ 的实现而言，std::function（见 33.5.3 节）或带有任意的类型捕获机制的 lambda（见 11.4.3 节）都没办法跨过语言的障碍。

15.3　使用头文件

为了说明头文件的使用，这里给出一些表达计算器程序（见 10.2 节和 14.3.1 节）物理结构的不同方法。

15.3.1　单头文件组织

将一个程序划分为多个文件的最简单的方法就是将定义放在适当数量的 .cpp 文件中，而将所需的类型、函数、类等的声明放在单一的 .h 文件中，每个 .cpp 文件都 #include 它。这是我自己在编写一个简单程序时首先采用的组织方式；如果发现需要某种更精致的组织形式，我会在稍后重新组织。

对计算器程序，我可能使用 5 个 .cpp 文件——lexer.cpp、parser.cpp、table.cpp、error.cpp 和 main.cpp 来保存函数及数据的定义。dc.h 头文件保存着多个 .cpp 文件中所使用的所有名字的声明：

```
// dc.h

#include <map>
#include<string>
#include<iostream>

namespace Parser {
    double expr(bool);
    double term(bool);
    double prim(bool);
}

namespace Lexer {
    enum class Kind : char {
        name, number, end,
        plus='+', minus='-', mul='*', div='/', print=';', assign='=', lp='(', rp=')'
    };

    struct Token {
        Kind kind;
        string string_value;
        double number_value;
    };

    class Token_stream {
    public:
        Token(istream& s) : ip{&s}, owns{false}, ct{Kind::end} { }
        Token(istream* p) : ip{p}, owns{true}, ct{Kind::end} { }

        ~Token() { close(); }

        Token get();        // 读取并返回下一个单词
        Token& current();   // 最近读取的单词

        void set_input(istream& s) { close(); ip = &s; owns=false; }
        void set_input(istream* p) { close(); ip = p; owns = true; }
    private:
        void close() { if (owns) delete ip; }
```

```
        istream* ip;          // 指向输入流的指针
        bool owns;            // Token_stream 拥有这个流吗?
        Token ct {Kind::end}; // 当前的单词
    };

    extern Token_stream ts;
}

namespace Table {
    extern map<string,double> table;
}

namespace Error {
    extern int no_of_errors;
    double error(const string& s);
}

namespace Driver {
    void calculate();
}
```

每个变量的声明中都使用了 extern 关键字以确保当我们在多个 .cpp 文件中 #include dc.h 时不会发生多重定义。对应的定义都放在恰当的 .cpp 文件中。

针对 dc.h 中声明的需要，我增加了标准库头文件，但我并没有仅仅为了方便单个 .cpp 文件而增加声明（例如使用 using 声明）。

不考虑具体实现代码，lexer.cpp 如下所示：

```
// lexer.cpp:

#include "dc.h"
#include <cctype>
#include <iostream>     // 冗余的：dc.h 已经有了

Lexer::Token_stream ts;

Lexer::Token Lexer::Token_stream::get() { /* ... */ }
Lexer::Token& Lexer::Token_stream::current() { /* ... */ }
```

我对每个定义使用了显式限定 Lexer::，而不是简单地将它们都放在名字空间中：

```
namespace Lexer { /* ... */ }
```

这能避免意外地向 Lexer 添加新的成员。另一方面，假如我希望向 Lexer 添加接口之外的成员，我就必须重新打开名字空间（见 14.2.5 节）。

这样使用头文件能确保其中的每个声明都在定义它的文件中被包含。例如，当编译 lexer.cpp 时，编译器会看到如下内容：

```
namespace Lexer { // 来自 dc.h
    // ...
    class Token_stream {
    public:
        Token get();
        // ...
    };
}
// ...
```

```
Lexer::Token Lexer::Token_stream::get() { /* ... */ }
```

这保证编译器能检查出每个名字任何类型上的不一致。例如，假如 get() 声明为返回一个 Token，但定义为返回一个 int，lexer.cpp 的编译就会失败，报告一个类型不匹配错误。如果程序员漏掉了定义，则链接器会发现这个问题。如果漏掉了声明，某些 .cpp 文件就会编译失败。

文件 parser.cpp 会像下面这样：

```
// parser.cpp:

#include "dc.h"

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

文件 table.cpp 会是这样：

```
// table.cpp:

#include "dc.h"

std::map<std::string,double> Table::table;
```

符号表就是一个标准库 map。

文件 error.cpp 是这样：

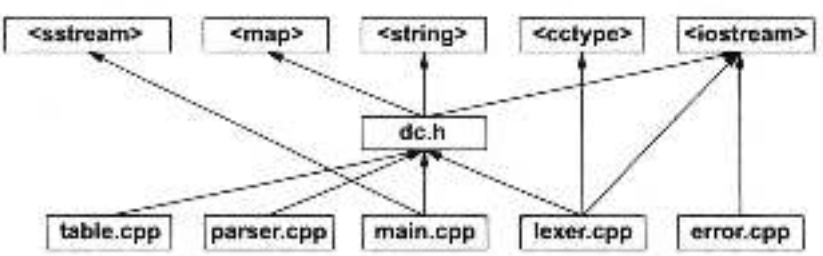
```
// main.cpp:

#include "dg.h"
// 任何其他 #includes 或声明

int Error::no_of_errors;
double Error::error(const string& s) { /* ... */ }
```

最后，文件 main.cpp 会像这样：

```
// main.cpp:

#include "dc.h"
#include <sstream>
#include <iostream>      // 冗余：dc.h 中已经有了

void Driver::calculate() { /* ... */ }

int main(int argc, char* argv[]) { /* ... */ }
```

为了能被识别出是程序的独一无二的那个 main()，main() 必须是一个全局函数（见 2.2.1 节和 15.4 节），因此这里没有使用任何名字空间。

系统的物理结构可图示如下：



顶部的头文件都是标准库特性头文件。多数情况下，进行程序分析时会忽略这些库，因为它们都是众所周知而且稳定的。对很小的程序来说，可以将所有 #include 指令都移到公共头文件中，从而简化程序的结构。类似地，对小规模程序，将 error.cpp 和 table.cpp 从 main.cpp 中分离出来也有些不必要。

当程序比较小且其组成部分不会分开使用时，这种单头文件风格的物理划分是最有用的。注意，如果使用了名字空间，程序的逻辑结构仍然是在 dc.h 内呈现。如果不使用名字空间，结构会很模糊，虽然注释能有所帮助。

对规模更大的程序，这种单头文件方式在传统的基于文件的开发环境中很难奏效。对公共头文件的任何改变都会迫使编译器编译整个程序，而多个程序员一起修改单一的公共头文件很容易出错。除非特别强调依赖名字空间和类的程序设计风格，否则随着程序规模的增长，逻辑结构就会变得糟糕。

15.3.2　多头文件组织

另一种物理组织方式是每个逻辑模块用一个专有的头文件定义其特性。每个 .cpp 文件有一个对应的 .h 文件说明其提供什么（接口）。每个 .cpp 文件包含它自己的 .h 文件，通常也包含其他 .h 文件以指明它需要来自其他模块的什么东西来实现它接口中所宣称的服务。这种物理组织对应模块的逻辑组织，为用户提供的接口放在其 .h 文件中，为实现者提供的接口放在一个后缀为 _impl.h 的文件中，而模块的函数、变量等的定义放在 .cpp 文件中。这样，语法分析器用三个文件表达，其用户接口由 parser.h 提供：

```
// parser.h:

namespace Parser {              // 用户接口
    double expr(bool get);
}
```

实现语法分析器的函数 expr()、prim() 和 term() 的共享上下文由 parser_impl.h 提供：

```
// parser_impl.h:

#include "parser.h"
#include "error.h"
#include "lexer.h"

using Error::error;
using namespace Lexer;

namespace Parser {              // 实现者接口
    double prim(bool get);
    double term(bool get);
    double expr(bool get);
}
```

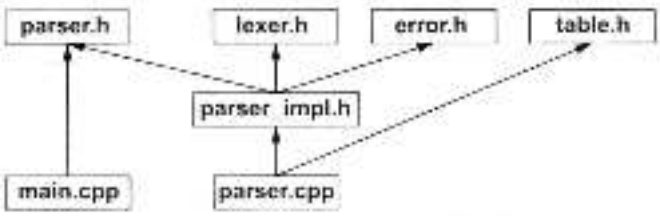
如果我们使用 Parser_impl 名字空间（见 14.3.3 节），用户接口与实现者接口间的区分就会更加清晰。

这里还 #include 了头文件 parser.h 中的用户接口，这就给了编译器检查一致性的机会（见 15.3.1 节）。

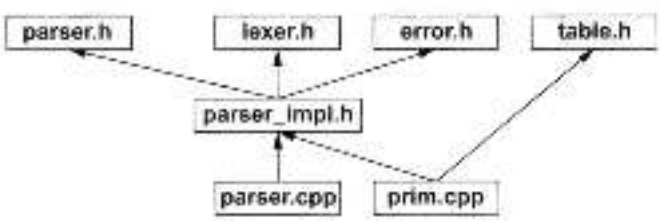
实现语法分析器的函数保存在 parser.cpp 中，其中还 #include 了 Parser 的函数所需要的头文件：

```
// parser.cpp:

#include "parser_impl.h"
#include "table.h"

using Table::table;

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

语法分析器的结构和驱动程序对它的使用可图示如下：



如我们所预期，这种物理结构非常吻合 14.3.1 节中所描述的逻辑结构。为了简化结构，我们可以在 parser_impl.h 中而不是 parser.cpp 中 #include table .h。但是，table .h 并非表达语法分析器函数共享上下文所必需的，它只是实现这些函数所需要的。实际上，它只被 prim() 这一个函数所使用，因此如果真的希望最小化相互依赖，我们应该将 prim() 放置在单独的 .cpp 文件中，而只在此文件中 #include table .h：



如果不是很大规模的模块，如此精心设计其实并没有太大必要。对实际规模的模块，常见的做法是在函数个体需要额外头文件的位置 #include 这些头文件。而且，使用多个 _impl.h 并不常见，因为模块函数的不同子集需要不同的共享上下文。

请注意，_impl.h 命名方式并不是 C++ 标准，甚至不是一种常见规范，这只是我喜欢的一种命名方式而已。

我们为什么自寻烦恼采用这种更复杂的多头文件方案呢？很明显，简单地将所有声明扔进单一头文件，例如 dc.h 中，会少费很多脑细胞。

多头文件组织可以伸缩到比我们的玩具语法分析器大几个数量级的模块以及比我们的计算器大几个数量级的程序。使用这种组织方式的根本原因是它提供了一种更好的关注点局部化的机制。当分析和修改一个大程序时，对程序员而言，能聚焦在一个相对较小的代码片段上是非常重要的。多头文件组织方式能很容易地准确确定语法分析器代码依赖什么，从而忽略程序其他部分。而单头文件方式则会迫使我们分析所有模块使用的每个声明，来确定哪些是相关的。一个简单的事实是，代码维护工作总是在信息不完整、视角受局限的条件下进行的。多头文件组织令我们在仅有局部视角的情况下能成功地“由内而外”地进行代码维护。而单头文件方法与其他任何以全局信息库为中心的方法类似，需要一种自顶向下的方法，而

且总是让我们受困于代码之间的依赖关系。

如果有更好的局部化，那么编译一个模块时所需的信息就更少，从而编译速度更快，其效果可能是非常巨大的。我曾经见到过只是通过简单的依赖分析更好地使用头文件，就令编译速度提高了 1000 倍的情况。

15.3.2.1　计算器程序其他模块

计算器程序其他模块的组织可参考语法分析器。但是，这些模块都太小了，不需要再划分出专有的 _impl.h 文件。只有当一个逻辑模块的实现由需要共享上下文的很多函数（以及提供给用户的内容）组成时，才需要专门的 _impl.h 文件。

错误处理程序通过 error.h 提供接口：

```
// error.h:

#include<string>

namespace Error {
        int Error::number_of_errors;
        double Error::error(const std::string&);
}
```

其实现在 error.cpp 中：

```
// error.cpp:

#include "error.h"

int Error::number_of_errors;
double Error::error(const std::string&) { /* ... */ }
```

词法分析器提供了一个更大且稍显凌乱的接口：

```
// lexer.h:

#include<string>
#include<iostream>

namespace Lexer {

        enum class Kind : char {/* ... */};

        class Token { /* ... */};
        class Token_stream { /* ... */};

        extern Token_stream is;
}
```

除了 lexer.h 之外，词法分析器的实现还依赖于 error.h 以及 <cctype> 中的字符分类函数（见 36.2 节）：

```
// lexer.cpp:

#include "lexer.h"
#include "error.h"
#include <iostream>      // 冗余：在 lexer.h 已经有了
#include <cctype>

Lexer::Token_stream is; // 默认“从 cin 中读”
```

```
Lexer::Token Lexer::Token_stream::get() { /* ... */ };
Lexer::Token& Lexer::Token_stream::current() { /* ... */ };
```

我们可以将 #include error.h 分离出来作为 Lexer 的 _impl.h 文件。但是，我认为对这个小程序而言这是有些过度使用接口分离方式了。

一如以往，我们在模块的实现中 #include 模块提供的接口——在本例中是 lexer.h，从而为编译器提供一个检查一致性的机会。

符号表应该是自包含的，标准库头文件 <map> 可以将所有感兴趣的内容包含进来以实现一个高效的 map 模板类：

```
// table.h:

#include <map>
#include <string>

namespace Table {
    extern std::map<std::string,double> table;
}
```

由于假定每个头文件可能被 #include 到多个 .cpp 文件中，我们必须将 table 的声明与其定义分离开来：

```
// table.cpp:

#include "table.h"

std::map<std::string,double> Table::table;
```

我将驱动程序简单地放入 main.cpp 中：

```
// main.cpp:

#include "parser.h"
#include "lexer.h"   // 以便能设置 ts
#include "error.h"
#include "table.h"   // 以便能预定义名字
#include <sstream> // 以便能将 main() 参数放到一个字符串流中

namespace Driver {
    void calculate() { /* ... */ }
}

int main(int argc, char* argv[]) { /* ... */ }
```

对更大规模的系统，将驱动程序分离出去从而最小化 main() 函数通常是值得的。这样，main() 就会调用另一个独立源文件中的驱动函数。如果你是在编写库代码，这种方式就特别重要。因为如果是开发一个库，我们就不能依赖于 main() 中的代码，并且要做好驱动程序被各种函数调用的准备。

15.3.2.2 头文件的使用

一个程序所使用的头文件的数量由很多因素决定。其中大部分因素更多的是与你的系统如何处理文件相关，而不是与 C++ 语言更相关。例如，如果你的编辑器 / 集成开发环境不能很方便地同时处理多个文件，那么多头文件方案的吸引力就小多了。

提醒一句：通常几十个头文件再加上构建程序执行环境的标准头文件（通常有几百个）

还是可管理的。但是，如果你将一个大程序中的声明划分为逻辑上最小规模的头文件（例如，将每个结构的声明放在独立的头文件中），你就会得到数百个头文件并且很难管理它们，即使是一个很小的项目也会如此。我认为这是对多头文件方式的过度使用。

对大项目而言，多头文件是不可避免的。在这类项目中，数百个文件（不包括标准头文件）是很正常的。真正的混乱始于规模达到上千个文件时。在那样的规模下，本章讨论的技术仍然可以应用，但文件的管理就会变成一项非常艰巨的任务。像依赖关系分析器这样的工具可以带来很大帮助，但如果程序的结构一团糟，这些工具也很难对编译器和链接器的性能有什么帮助。记住，对实际规模的程序而言单头文件风格是不可行的，这类程序会包含多头文件。两种风格间的选择其实（反复）发生在构造程序的组成部分时。

单头文件风格和多头文件风格不是互相替代的，它们是互补的技术，每当我们设计一个重要的模块时，就要考虑如何使用这两种技术，当系统演进时又要重新考虑。记住很关键的一点：一个接口不可能对所有需求都适应良好。分离实现者接口和用户接口通常是必要的。此外，很多大规模系统的构造方式是为大多数用户提供一个简单的接口，另为专家级用户提供一个功能更强的扩展接口，这是一个很好的策略。相对于一般用户所希望了解的特性，专家用户接口（“完整接口”）会 #include 多得多的特性。实际上，一般用户接口的设计方式通常是去掉一些特性，这些特性所在的头文件定义了一些设计者不想让一般用户了解的特性。术语“一般用户”不是贬义的。在那些我不必成为专家的领域中，我非常希望被当作一般用户对待，这样我就能避免很多麻烦。

15.3.3 包含保护

多头文件方法将每个逻辑模块表示为一个一致的、自包含的单元。从程序总体的角度看，很多为了保证逻辑单元完整性而设计的声明其实是冗余的。对大规模程序而言，这种冗余可能导致错误——包含类定义或内联函数的头文件在相同的编译单元中被 #include 两次（见 15.2.3 节）。

对此，我们有两种选择：

[1] 重组我们的程序，去掉冗余，或

[2] 找到一种方法允许重复包含头文件。

我们设计计算器程序的最终版本时就使用了第一种方法，但对于实际规模的程序，这种方法太令人厌烦了，完全不实用。而且我们还是需要这种冗余的，它能令程序的每个组成部分都是独立可理解的。

分析冗余 #include 进而简化程序的好处无论从逻辑角度看还是从减少编译时间的角度看都是非常巨大的。但是，彻底的分析和简化很难做到，因此必须使用某种能允许冗余 #include 的方法。这种方法必须能系统地应用，因为如果依赖用户做冗余分析的话，根本没有方法判断分析是否完整。

传统的解决方法是在头文件中插入包含保护（include guard）。例如：

```
// error.h:

#ifndef CALC_ERROR_H
#define CALC_ERROR_H

namespace Error {
```

```
    //...
}

#endif    // CALC_ERROR_H
```

如果 CALC_ERROR_H 已定义，文件中 #ifndef 和 #endif 间的内容就会被忽略。因此，编译过程中第一次看到 error.h 时，其内容会被读取，CALC_ERROR_H 会被定义。如果在编译过程中编译器再次看到 error.h，其内容会被忽略。这是一种宏技巧，但它很有效，在 C 和 C++ 世界中被普遍使用。所有标准头文件都带有包含保护。

头文件可能在任意上下文中被包含，而且没有名字空间避免宏名冲突。因此，我通常选择很长很丑的名字作为包含保护。

一旦人们习惯了头文件和包含保护，就会直接和间接地包含大量头文件。即使使用的 C++ 实现对头文件的处理进行了优化，包含非常多头文件也是不可取的。这样做可能导致过长的编译时间，而且会将大量声明和宏带入当前作用域中。第二点可能会以不可预测的糟糕方式影响程序的含义。我们只应在必要时包含头文件。

15.4 程序

一组分离编译的单元经由链接器组合就形成了程序。其中用到的每个函数、对象、类型等都必须是唯一定义的（见 6.3 节和 15.2.3 节）。一个程序必须恰好包含一个名为 main() 的函数（见 2.2.1 节）。通过调用全局函数 main() 开始执行程序的主要计算任务，从 main() 返回后程序就终止了。main() 的返回类型是 int，所有 C++ 实现都支持下面两个版本的 main()：

```
int main() { /* ... */ }
int main(int argc, char* argv[]) { /* ... */ }
```

每个程序都只能使用两者之一。此外，C++ 实现还可以支持其他版本的 main()。带 argc、argv 的版本用来从程序运行环境传输参数，参见 10.2.7 节。

main() 返回的 int 作为程序执行的结果被传递给调用 main() 的系统，非零返回值表示发生了一个错误。

这种简单机制对于包含全局变量（见 15.4.1 节）或抛出未捕获异常（见 13.5.2.5 节）的程序必须精心设计才能实现。

15.4.1 非局部变量初始化

原则上，定义在任何函数之外的变量（即，全局变量、名字空间变量以及类 static 变量）在 main() 被调用前初始化。同一个编译单元中的非局部变量按它们的定义顺序进行初始化。如果这种变量没有显式的初始化器，则它们初始化为其类型的默认值（见 17.3.3 节）。内置类型和枚举类型的默认初始化值为 0。例如：

```
double x = 2;        // 非局部变量
double y;
double sqx = sqrt(x+y);
```

在本例中，x 和 y 在被 sqrt() 调用前初始化，因此调用的是 sqrt(2)。

不同编译单元中全局变量的初始化顺序无法保证一致。因此，在不同编译单元的全局变量初始值顺序间建立依赖关系是不明智的做法。此外，全局变量初始化时抛出的异常也不可

能被捕获（见 13.5.2.5 节）。一般来说我们最好尽量减少全局变量的使用，特别是限制使用需要复杂初始化的全局变量。

有多种技术可以强制不同编译单元中全局变量的初始化顺序。但是，没有既高效又可移植的方法。特别是，动态链接库与依赖关系复杂的全局变量不能很好地共存。

通常，返回引用的函数可以很好地替代全局变量。例如：

```
int& use_count()
{
    static int uc = 0;
    return uc;
}
```

现在，调用 use_count() 就像使用全局变量一样，唯一的差别是它在第一次使用时才初始化（见 7.7 节）。例如：

```
void f()
{
    cout << ++use_count();  // 读取并递增
    // ...
}
```

与其他使用 static 的技术类似，这种技术也不是线程安全的。局部 static 本身的初始化是线程安全的（见 42.3.3. 节）。初始值甚至可以是一个常量表达式（见 10.4 节），从而初始化在链接时即完成，不会产生数据竞争（见 42.3.3 节）。但是，本例中的 ++ 可能导致数据竞争。

控制非局部（静态分配的）变量初始化的机制就是 C++ 实现启动 C++ 程序的机制。只有当执行 main() 时这一机制才保证正常工作。因此，在用作非 C++ 程序的片段的 C++ 代码中，我们应该避免使用要求运行时初始化的非局部变量。

注意，用常量表达式（见 10.4 节）初始化的变量不能依赖其他编译单元中对象的值，也不能要求运行时初始化。因而这种变量在所有情况下都是安全的。

15.4.2　初始化和并发

考虑下面的代码：

```
int x = 3;
int y = sqrt(++x);
```

x 和 y 的值会是什么？明显的答案是“3 和 2！”为什么？用一个常量表达式初始化一个静态分配的对象是在链接时完成的，因此 x 的值是 3。但是，y 的初始值不是一个常量表达式（sqrt() 没有 constexpr 版本），因此直到运行时 y 才被初始化。但是，单一编译单元内的静态分配对象的初始化顺序与它们的定义顺序是一致的（见 15.4.1 节）。因此，y 的值是 2。

这一论据的问题在于如果使用了多线程（见 5.3.1 节和 42.2 节），每个线程都会执行运行时初始化。系统并不隐含地应用互斥机制防止数据竞争。这样，一个线程中的 sqrt(++x) 可能发生在另一个线程设法递增 x 之前，也可能发生在其后。因此 y 的值可能是 sqrt(4)，也可能是 sqrt(5)。

为了避免这个问题，我们应该（照例）：

- 尽量减少静态分配对象的使用，并保持它们的初始化尽可能简单。
- 避免依赖其他编译单元中的动态初始化的对象（见 15.4.1 节）

此外，为了避免初始化中的数据竞争，应依次尝试下列技术：

[1] 使用常量表达式进行初始化（注意，在链接时，不带初始化器的内置类型被初始化为零，标准容器和 string 被初始化为空）。
[2] 使用没有副作用的表达式进行初始化。
[3] 在已知是单线程的计算过程的“启动阶段”进行初始化。
[4] 使用某种形式的互斥（见 5.3.4 节和 42.3 节）。

15.4.3 程序终止

程序终止的方式有很多种：
[1] 从 main() 返回。
[2] 调用 exit()。
[3] 调用 abort()。
[4] 抛出一个未捕获的异常。
[5] 违反 noexcept。
[6] 调用 quick_exit()。

此外，还有很多行为不良以及依赖具体实现的方法可令程序崩溃（例如，用一个 double 除以零）。

如果使用标准库函数 exit() 终止一个程序，则会调用已构造的静态对象的析构函数（见 15.4.1 节和 16.2.12 节）。但是，如果程序是使用标准库函数 abort() 终止的，析构函数就不会被调用。注意，这意味着 exit() 不会立即终止程序。在一个析构函数中调用 exit() 会导致无限递归。exit() 的类型为：

```
void exit(int);
```

类似 main() 的返回值（见 2.2.1 节），exit() 的参数会作为程序的结果返回给“系统”，0 表示成功结束。

调用 exit() 意味着调用函数的局部变量及其调用者不会执行各自的析构函数。抛出一个异常并捕获它可确保局部变量被正确销毁（见 13.5.1 节）。而且，调用 exit() 终止一个程序没有给其所在函数的调用者处理此问题的机会。因此，离开一个上下文的最好方式是抛出一个异常，然后让异常处理程序决定接下来做什么。例如，main() 可以捕获所有异常（见 13.5.2.2 节）。

C（和 C++）标准库函数 atexit() 提供了在程序终止过程中执行代码的机会。例如：

```
void my_cleanup();

void somewhere()
{
    if (atexit(&my_cleanup)==0) {
        // 在正常终止时会调用 my_cleanup
    }
    else {
        // 错误：太多的 atexit 函数
    }
}
```

这非常像程序终止时自动调用全局变量的析构函数（见 15.4.1 节和 16.2.12 节）。传给 atexit() 的参数不能传递实参或返回结果，而且不同实现还对 atexit() 函数的数目有不同限

制。如果 atexit() 返回一个非零值，表示已达上限。这些限制使得 atexit() 并没有初看起来那么有用。基本上，atexit() 只是 C 语言没有析构函数的一种变通方法。

在调用 atexit(f) 前构造的静态分配对象的析构函数将在调用 f 之后被调用。在调用 atexit(f) 后构造的对象的析构函数将在调用 f 之前被调用。

函数 quick_exit() 与 exit() 类似，区别在于它不调用任何析构函数。你可以用 at_quick_exit() 注册被 quick_exit() 调用的函数。

函数 exit()、abort()、quick_exit()、atexit() 和 at_quick_exit() 都是在 <cstdlib> 中声明的。

15.5　建议

[1] 用头文件表达接口，强调逻辑结构；15.1 节和 15.3.2 节。

[2] 在实现函数的源文件中 #include 声明函数的头文件；15.3.1 节。

[3] 不要在不同编译单元中定义同名但含义相近却不完全一致的全局实体；15.2 节。

[4] 不要在头文件中定义非内联函数；15.2.2 节。

[5] 只在全局作用域和名字空间中使用 #include；15.2.2 节。

[6] 只 #include 完整的声明；15.2.2 节。

[7] 使用包含保护；15.3.3 节。

[8] 在名字空间中 #include C 头文件以避免全局名字；14.4.9 节和 15.2.4 节。

[9] 令头文件自包含；15.2.3 节。

[10] 区分用户接口和实现者接口；15.3.2 节。

[11] 区分一般用户接口和专家用户接口；15.3.2 节。

[12] 若代码是用作非 C++ 程序的一部分，则应避免需要运行时初始化的非局部对象；15.4.1 节。

第三部分

抽象机制

这一部分介绍定义和使用新类型的C++特性，主要介绍通常称为面向对象程序设计（object-oriented programming）和泛型程序设计（generic programming）的技术。

"……没有什么比建立事物的新秩序更困难、更易受质疑、更充满危险的了。原因是，革新者将那些惯于旧秩序的人们统统放在了敌对位置，而在新秩序下可以顺利过活之人即使拥护你，也是有保留的……"

——尼科洛·马基雅维里（"君主论"第6章）

类

那些类型一点儿也不“抽象”；
它们如此真实，就像 int 和 float 一样。

——道格·麦克罗伊

- 引言
- 类基础
  成员函数；默认拷贝；访问控制；class 和 struct；构造函数；explicit 构造函数；类内初始化器；类内函数定义；可变性；自引用；成员访问；static 成员；成员类型
- 具体类
  成员函数；辅助函数；重载运算符；具体类的重要性
- 建议

16.1 引言

C++ 类是创建新类型的工具，创建出的新类型可以像内置类型一样方便地使用。而且，派生类（见 3.2.4 节，第 20 章）和模板（见 3.4 节，第 23 章）允许程序员表达类之间的（层次和参数化）关系并利用这种关系。

一个类型就是一个概念（一个思想，一个观念，等等）的具体表示。例如，C++ 内置类型 float 及其运算 +、-、* 等等一起提供了数学概念“实数”的一种近似表示。类是用户自定义类型。如果一个概念没有与之直接对应的内置类型，我们就定义一个新类型来表示它。例如，我们可以提供类型 Trunk_line 用于拨号服务处理程序，提供类型 Explosion 用于视频游戏，或是提供类型 list<Paragraph> 用于文本处理程序。如果一个程序提供了与应用中的概念非常匹配的类型，那么它会比其他程序更易理解、更易分析，也更易修改。一组精心挑选的用户自定义类型也会令程序更加简洁，令很多代码分析技术成为可能，特别是令编译器能检测到对象的非法使用。如果没有用户自定义类型，这些错误只能通过穷尽测试来发现。

定义新类型的基本思想是将实现的细节（例如，某种类型对象的数据存储布局）与正确使用它的必要属性（例如，可访问数据的函数的完整列表）分离。这种分离的最佳表达方式是：通过一个专用接口引导数据结构及其内部辅助例程的使用。

本章主要介绍相对简单的“具体的”用户自定义类型，这些类型逻辑上与内置类型没有差别：

16.2 节　介绍定义一个类及其成员的基本特性。

16.3 节　介绍如何设计优雅高效的具体类。

接下来的几章探究更多细节，介绍抽象类和类层次。

第 17 章　介绍各种不同的类对象初始化方法、如何拷贝和移动对象以及如何提供对象

销毁时（如离开作用域）执行的“清理动作”。

第 18 章 介绍如何为用户自定义类型定义一元和二元运算符（如 +、* 和 !）以及如何使用它们。

第 19 章 介绍如何定义和使用一些“特殊”运算符（如 []、()、->、new）。它们通常的使用方式与算术和逻辑运算符不同。特别是，这一章会展示如何定义一个字符串类。

第 20 章 介绍支持面向对象程序设计的基本语言特性，涉及基类、派生类、虚函数和访问控制等内容。

第 21 章 主要介绍如何使用基类和派生类并基于类层次思想高效组织代码。这一章大部分内容都是讨论程序设计技术，但未涉及多重继承（一个类有多个基类）的技术层面的内容。

第 22 章 介绍类层次显式导航技术。特别是，这一章会介绍类型转换操作 dynamic_cast 和 static_cast 以及给定对象的一个基类的条件下确定其类型的操作（typeid）。

16.2 类基础

下面是类的简要概括：

- 一个类就是一个用户自定义类型。
- 一个类由一组成员构成。最常见的成员类别是数据成员和成员函数。
- 成员函数可定义初始化（创建）、拷贝、移动和清理（析构）等语义。
- 对对象使用 .（点）访问成员，对指针使用 ->（箭头）访问成员。
- 可以为类定义运算符，如 +、! 和 []。
- 一个类就是一个包含其成员的名字空间。
- public 成员提供类的接口，private 成员提供实现细节。
- struct 是成员默认为 public 的 class。

例如：

```
class X {
private:                    // 类的表示（实现）是私有的
    int m;
public:                     // 用户接口是公有的
    X(int i =0) :m{i} { }   // 构造函数（初始化数据成员 m）

    int mf(int i)           // 成员函数
    {
        int old = m;
        m = i;              // 设置一个新值
        return old;         // 返回旧值
    }
};

X var {7}; // 一个 X 类型的变量，初始化为 7

int user(X var, X* ptr)
{
    int x = var.mf(7);      // 使用 .（点）访问
    int y = ptr->mf(9);     // 使用 ->（箭头）访问
```

```
    int z = var.m;          // 错误：不能访问私有成员
}
```

接下来的几节会详细介绍这些特性并介绍基本原理。内容的组织是指南风格的：逐步展开思想，细节推迟介绍。

16.2.1　成员函数

考虑用 struct（见 2.3.1 节和 8.2 节）实现日期的概念——定义 Date 的表示方式和操作这种类型的变量的一组函数：

```
struct Date {           // 表示
    int d, m, y;
};

void init_date(Date& d, int, int, int);     // 初始化 d
void add_year(Date& d, int n);              // d 增加 n 年
void add_month(Date& d, int n);             // d 增加 n 个月
void add_day(Date& d, int n);               // d 增加 n 天
```

数据类型、Date 和这些函数之间并无显式关联。我们可以通过将函数声明为成员来建立这种关联：

```
struct Date {
    int d, m, y;

    void init(int dd, int mm, int yy);      // 初始化
    void add_year(int n);                   // 增加 n 年
    void add_month(int n);                  // 增加 n 个月
    void add_day(int n);                    // 增加 n 天
};
```

声明于类定义（struct 也是一种类，见 16.2.4 节）内的函数称为*成员函数*（member function），对恰当类型的特定变量使用结构成员访问语法（见 8.2 节）才能调用这种函数。例如：

```
Date my_birthday;

void f()
{
    Date today;

    today.init(16,10,1996);
    my_birthday.init(30,12,1950);

    Date tomorrow = today;
    tomorrow.add_day(1);
    // ...
}
```

由于不同结构可能有同名成员函数，在定义成员函数时必须指定结构名：

```
void Date::init(int dd, int mm, int yy)
{
    d = dd;
    m = mm;
    y = yy;
}
```

在成员函数中，不必显式引用对象即可使用成员的名字。在此情况下，名字所引用的是调用函数的对象的成员。例如，当对 today 调用 Date::init() 时，m = mm 是对 today.m 赋值。而对 my_birthday 调用 Date::init() 时，m = mm 是对 my_birthday.m 赋值。类成员函数“知道”是哪个对象调用的它。但是，请参考 16.2.12 节中 static 成员的概念。

16.2.2 默认拷贝

默认情况下，对象是可以拷贝的。特别是，一个类对象可以用同类的另一个对象的副本来进行初始化。例如：

```
Date d1 = my_birthday;  // 用副本进行初始化
Date d2 {my_birthday};  // 用副本进行初始化
```

默认情况下，一个类对象的副本是对每个成员逐个拷贝得到的。如果类 X 的这种默认拷贝行为不是我们所希望的，可以提供更恰当的行为（见 3.3 节和 17.5 节）。

类似地，类对象默认也可以通过赋值操作拷贝。例如：

```
void f(Date& d)
{
    d = my_birthday;
}
```

再重复一遍，默认的拷贝语义是逐成员复制。如果对于类 X 这不是正确的选择，用户可以定义一个恰当的赋值运算符（见 3.3 节和 17.5 节）。

16.2.3 访问控制

上一节中的 Date 声明提供了一组处理 Date 对象的函数，但是并未指明是否只有这些函数直接依赖于 Date 的表示方式以及是否只有它们直接访问类 Date 的对象。这种约束可以通过使用 class 而非 struct 来表达：

```
class Date {
    int d, m, y;
public:
    void init(int dd, int mm, int yy);      // 初始化

    void add_year(int n);                   // 增加 n 年
    void add_month(int n);                  // 增加 n 个月
    void add_day(int n);                    // 增加 n 天
};
```

标签 public 将类的主体分为两部分。第一部分中的名字是私有的（private），它们只能被成员函数使用。第二部分是公有的（public），构成类对象的公共接口。struct 就是一个成员默认为公有的 class（见 16.2.4 节）。成员函数的声明和使用是一样的。例如：

```
void Date::add_year(int n)
{
    y += n;
}
```

但是，非成员函数禁止使用私有成员。例如：

```
void timewarp(Date& d)
{
    d.y -= 200;         // 错误：Date::y 是私有的
}
```

现在函数 init() 就非常重要了，因为将数据设定为私有迫使我们提供一种初始化成员的方法。例如：

```
Date dx;
dx.m = 3;                    // 错误：m 是私有的
dx.init(25,3,2011);          // 正确
```

限制只有一组显式声明的函数才能访问一个数据结构可以带来多方面的好处。例如，任何导致对象保存非法数据（例如 2016 年 12 月 36 日）的错误都必然是由成员函数中的代码引起的。这意味着调试的第一阶段——定位——甚至在程序运行之前就完成了。这是一个特例，更一般的情况是，类型 Date 的行为的任何改变都受到且必然受到其成员的改变的影响。特别是，如果我们改变了一个类的表示方式，就只能修改成员函数来利用新的表示方式。用户代码则直接依赖于公共接口，因此无须重写（虽然可能需要重新编译）。另一个好处是潜在用户为了学习类的使用只需观察成员函数的定义。还有一个更微妙但也是最重要的好处是，聚焦于设计一个好的接口能产生更好的代码，因为我们可以对调试投入更多的思考和时间——将精力花费在程序正确使用的相关问题上更有价值。

私有数据的保护依赖于对类成员名的使用限制。因此通过地址操作（见 7.4.1 节）和显式类型转换（见 11.5 节）可以绕过私有保护，当然这是一种欺骗。C++ 只能防止意外而无法防止故意规避（欺骗）。只有硬件可以完美防止对通用语言的恶意使用，而这在实际系统中其实是很难实现的。

16.2.4　class 和 struct

下面的语法结构

```
class X { ... };
```

称为类定义（class definition），它定义了一个名为 X 的类型。由于历史原因，类定义常常被称为类声明（class declaration）。这样叫它的另一个原因是，与其他并非定义的 C++ 声明类似，我们可以在不同源文件中使用 #include 重复类定义而不会违反单一定义规则（见 15.2.3 节）。

根据定义，struct 就是一个成员默认为公有的类，即

```
struct S { /* ... */ };
```

就是下面定义的简写

```
class S { public: /* ... */ };
```

S 的这两个定义是可以互换的，当然坚持一种风格通常更明智。你到底使用哪种风格依赖于具体环境和你的偏好。如果我认为一个类是“简单数据结构”，更喜欢使用 struct。如果我认为一个类是“具有不变式的真正类型”，会使用 class。即使是对 struct 而言，构造函数和访问函数也是非常有用的，但它们只是一种简写而非不变式的保证（见 2.4.3.2 节和 13.4 节）。

class 的成员默认是私有的：

```
class Date1 {
    int d, m, y;             // 默认私有
public:
    Date1(int dd, int mm, int yy);
    void add_year(int n);    // 增加 n 年
};
```

但是，我们也可以使用访问说明符 private: 来指明接下来的成员是私有的，就像用 public: 说明接下来的成员是公有的一样：

```
struct Date2 {
private:
    int d, m, y;
public:
    Date2(int dd, int mm, int yy);
    void add_year(int n);       // 增加 n 年
};
```

除了名字不同之外，Date1 和 Date2 是等价的。

C++ 并不要求在类定义中首先声明数据。实际上，将数据成员放在最后以强调提供公共用户接口的函数（位置在前）通常是很有意义的。例如：

```
class Date3 {
public:
    Date3(int dd, int mm, int yy);
    void add_year(int n);       // 增加 n 年
private:
    int d, m, y;
};
```

实际代码中的公共接口和实现细节通常都比教学用的例子更为复杂，因此我倾向于使用 Date3 的风格。

在一个类声明中可以多次使用访问说明符。例如：

```
class Date4 {
public:
    Date4(int dd, int mm, int yy);
private:
    int d, m, y;
public:
    void add_year(int n);       // 增加 n 年
};
```

像 Date4 这样使用多个公有声明段会让程序显得有些凌乱，而且可能影响对象布局（见 20.5 节）。包含多个私有声明段也有这些问题。但是，允许多个访问说明符对机器生成代码是很有用的。

16.2.5 构造函数

使用像 init() 这样的函数为类对象提供初始化功能既不优雅也容易出错。因为这种方式没有规定一个对象必须进行初始化，程序员可能忘记初始化，或初始化两次（两种情况通常都会带来灾难性后果）。一种更好的方法是允许程序员声明一个函数，它显式表明自己是专门完成对象初始化任务的。由于这种函数的本质是构造一个给定类型的值，因此被称为构造函数（constructor）。构造函数的显著特征是与类具有相同的名字。例如：

```
class Date {
    int d, m, y;
public:
    Date(int dd, int mm, int yy);       // 构造函数
    // ...
};
```

如果一个类有一个构造函数，其所有对象都会通过调用构造函数完成初始化。如果构造函数需要参数，在初始化时就要提供这些参数：

```
Date today = Date(23,6,1983);
Date xmas(25,12,1990);          // 简写形式
Date my_birthday;               // 错误：缺少初始值
Date release1_0(10,12);         // 错误：漏掉了第三个参数
```

由于构造函数定义了类的初始化方式，因此我们可以使用 {} 初始化记法：

```
Date today = Date {23,6,1983};
Date xmas {25,12,1990};         // 简写形式
Date release1_0 {10,12};        // 错误：漏掉了第三个参数
```

我建议优先使用 {} 记法而不是 ()，因为前者明确表明了要做什么（初始化），从而避免了某些潜在错误，而且可以一致地使用（见 2.2.2 节和 6.3.5 节）。有些情况下必须使用 () 记法（见 4.4.1 节和 17.3.2.1 节），但这种情况很少。

通过提供多个构造函数，可以为某类型的对象提供多种不同的初始化方法。例如：

```
class Date {
    int d, m, y;
public:
    // ...

    Date(int, int, int);        // 年，月，日
    Date(int, int);             // 日，月，当前年份
    Date(int);                  // 日，当前月份和年份
    Date();                     // 默认 Date 值：今天的日期
    Date(const char*);          // 字符串表示的日期
};
```

构造函数的重载规则与普通函数（见 12.3 节）相同。只要构造函数的参数类型明显不同，编译器就能选择正确的版本使用：

```
Date today {4};                 // 4, today.m, today.y
Date july4 {"July 4, 1983"};
Date guy {5,11};                // 5, 11 月 , today.y
Date now;                       // 默认初始化为今天
Date start {};                  // 默认初始化为今天
```

在 Date 这个例子中，构造函数的扩展是很典型的。当定义一个类时，程序员总是忍不住增加新的特性，而原因仅仅是可能有人需要。确定哪些特性是真正需要的并在设计中只包含这些特性需要更仔细思考，但这些额外的思考通常会带来更简洁也更容易理解的程序，因此是值得的。减少关联函数的一种方法是使用默认参数（见 12.2.5 节）。对于 Date，我们可以赋予每个参数一个默认值，表示“选择默认值：today”。

```
class Date {
    int d, m, y;
public:
    Date(int dd =0, int mm =0, int yy =0);
    // ...
};

Date::Date(int dd, int mm, int yy)
{
    d = dd ? dd : today.d;
    m = mm ? mm : today.m;
```

```
    y = yy ? yy : today.y;

    // 检查 Date 是否合法
}
```

当一个参数值用来表示“选择默认值”时，此值必须在参数的可能值集合之外。对于 day 和 month，很明显零就可以。但对 year 就不是这样了。幸运的是，欧洲日历没有零年，公元前 1 年（year==-1）之后紧接着是公元元年（year==1）。

另一种方法是直接用默认值作为默认参数：

```
class Date {
    int d, m, y;
public:
    Date(int dd =today.d, int mm =today.m, int yy =today.y);
    // ...
};

Date::Date(int dd, int mm, int yy)
{
    // 检查 Date 是否合法
}
```

但是，我选择使用 0，这样可避免在 Date 的接口中写入具体值，未来我们就有机会优化默认值的实现。

注意，通过确保对象的正确初始化，构造函数极大地简化了成员函数的实现。有了构造函数，其他成员函数就不再需要处理未初始化数据的情况（见 16.3.1 节）。

16.2.6 explicit 构造函数

默认情况下，用单一参数调用一个构造函数，其行为类似于从参数类型到类自身类型的转换。例如：

```
complex<double> d {1};      // d=={1,0}（见 5.6.2 节）
```

这种隐式转换可能非常有用。复数是一个典型的例子，如果忽略虚部，我们就会得到实数轴上的一个复数，这正是数学家所要求的。但在很多情况下，这种转换可能是混乱和错误的主要来源。考虑 Date：

```
void my_fct(Date d);

void f()
{
    Date d {15};    // 似乎合理；d 变为 {15,today.m,today.y}
    // ...
    my_fct(15);     // 含混
    d = 15;         // 含混
    // ...
}
```

这最多只能算是一段含混的代码。不管我们的代码如何错综复杂，数值 15 和 Date 之间并无清晰的逻辑关联。

幸运的是，我们可以指明构造函数不能用作隐式类型转换。如果构造函数的声明带有关键字 explicit，则它只能用于初始化和显式类型转换。例如：

```
class Date {
    int d, m, y;
public:
    explicit Date(int dd =0, int mm =0, int yy =0);
    // ...
};

Date d1 {15};          // 正确：按看作显式类型转换
Date d2 = Date{15};    // 正确：显式类型转换
Date d3 = {15};        // 错误：=方式的初始化不能进行隐式类型转换
Date d4 = 15;          // 错误：=方式的初始化不能进行隐式类型转换

void f()
{
    my_fct(15);        // 错误：参数传递不能进行隐式类型转换
    my_fct({15});      // 错误：参数传递不能进行隐式类型转换
    my_fct(Date{15});  // 正确：显式类型转换
    // ...
}
```

用 = 进行初始化可看作*拷贝初始化*（copy initialization）。一般来说，初始化器的副本会被放入待初始化的对象。但是，如果初始化器是一个右值（见 6.4.1 节），这种拷贝可能被优化掉（取消），而采用移动操作（见 3.3.2 节和 17.5.2 节）。省略 = 会将初始化变为显式初始化。显式初始化也称为*直接初始化*（direct initialization）。

默认情况下，应该将单参数的构造函数声明为 explicit。除非你有很好的理由，否则的话应该按这种默认方式做（例如 complex）。如果定义隐式构造函数，最好写下原因，否则代码的维护者可能怀疑你疏忽了，或是不懂这一原则。

如果一个构造函数声明为 explicit 且定义在类外，则在定义中不能重复 explicit：

```
class Date {
    int d, m, y;
public:
    explicit Date(int dd);
    // ...
};

Date::Date(int dd) { /* ... */ }           // 正确
explicit Date::Date(int dd) { /* ... */ }  // 错误
```

大多数 explicit 起很重要作用的构造函数都接受单一参数。但是，explicit 也可用于无参或多个参数的构造函数。例如：

```
struct X {
    explicit X();
    explicit X(int,int);
};

X x1 = {};          // 错误：隐式的
X x2 = {1,2};       // 错误：隐式的

X x3 {};            // 正确：显式的
X x4 {1,2};         // 正确：显式的

int f(X);
```

```
int i1 = f{};          // 错误：隐式的
int i2 = f{1,2};       // 错误：隐式的

int i3 = f(X{});       // 正确：显式的
int i4 = f(X{1,2});    // 正确：显式的
```

列表初始化（见 17.3.4.3 节）也存在直接初始化和拷贝初始化的区别。

16.2.7 类内初始化器

当使用多个构造函数时，成员初始化可以是重复的。例如：

```
class Date {
    int d, m, y;
public:
    Date(int, int, int);        // 天，月，年
    Date(int, int);             // 天，月，当前年份
    Date(int);                  // 天，当前月份和年份
    Date();                     // 默认 Date: today
    Date(const char*);          // 字符串表示的日期
    // ...
};
```

通过引入默认参数，我们就可以减少构造函数的数量（见 16.2.5 节）来解决此问题。另一种方法是为数据成员添加初始化器：

```
class Date {
    int d {today.d};
    int m {today.m};
    int y {today.y};
public:
    Date(int, int, int);        // 天，月，年
    Date(int, int);             // 天，月，当前年份
    Date(int);                  // 天，当前月份和年份
    Date();                     // 默认 Date: today
    Date(const char*);          // 字符串表示的日期
    // ...
```

现在，每个构造函数都有已初始化的 d，m 和 y，当然它们也可以自己来做初始化。例如：

```
Date::Date(int dd)
    :d{dd}
{
    // 检查 Date 是否合法
}
```

这段代码等价于：

```
Date::Date(int dd)
    :d{dd}, m{today.m}, y{today.y}
{
    // 检查 Date 是否合法
}
```

16.2.8 类内函数定义

如果一个函数不仅在类中声明，还在类中定义，那么它就被当作内联函数处理（见 12.1.5 节），即很少修改且频繁使用的小函数适合类内定义。类似所属类的定义，可在多个

编译单元中使用 #include 重复类内定义的成员函数。无论在哪里使用 #include，其含义都应保持一致（见 15.2.3 节）。

类成员可以访问同类的其他成员，而不管成员是在哪里定义的（见 6.3.4 节）。考虑下面的代码：

```
class Date {
public:
    void add_month(int n) { m+=n; }    // 增加 Date 的 m
    // ...
private:
    int d, m, y;
};
```

即函数和数据成员的声明是不依赖于顺序的。可以编写等价代码如下：

```
class Date {
public:
    void add_month(int n) { m+=n; }    // 增加 Date 的 m
    // ...
private:
    int d, m, y;
};

inline void Date::add_month(int n) // 增加 n 个月
{
    m+=n;    // 增加 Date 的 m
}
```

后一种风格常用来保持类定义更为简单易读。它还实现了类接口和类实现在文本上的分离。

显然，我简化了 Date::add_month 的定义，希望增加 n 能直接得到一个正确日期的想法有些过于天真了（见 16.3.1 节）。

16.2.9 可变性

我们可以定义具名的常量对象或变量对象。换句话说，一个名字指向的既可以是一个保存不可变值的对象，也可以是一个保存可变值的对象。由于精确术语可能有些笨拙，我们最终采用的描述方式是：称某些变量是常量，或者更简单地称之为 const 变量。无论这种描述对一个以英语为母语的人来说有多么奇怪，概念本身还是非常有用的，而且已深深植入了 C++ 类型系统中。系统地使用不可变对象有利于产生更易理解的代码，有利于尽早发现更多错误，而且有时会提高性能。特别是，不可变性在多线程编程中是一个非常有用的特性（见 5.3 节和第 41 章）。

为了使不可变性不局限于内置类型的简单常量的定义，我们必须能定义可操作用户自定义 const 对象的函数。对独立函数而言，这意味着函数可接受 const T& 参数。对类而言，这意味着我们必须能定义操作 const 对象的成员函数。

16.2.9.1 常量成员函数

到目前为止我们所定义的 Date 提供了一些能为 Date 赋值的函数。不幸的是，我们没有提供检查 Date 值的方法。通过增加读取天、月和年的函数，可以很容易地弥补此不足：

```
class Date {
    int d, m, y;
public:
    int day() const { return d; }
```

```
    int month() const { return m; }
    int year() const;

    void add_year(int n);    // 增加n年
    // ...
};
```

函数声明中（空）参数列表后的 const 指出这些函数不会修改 Date 的状态。

自然地，编译器会捕获试图违反此承诺的代码。例如：

```
int Date::year() const
{
    return ++y;    // 错误：试图在 const 函数中改变成员值
}
```

当 const 成员函数定义在类外时，必须使用 const 后缀：

```
int Date::year()    // 错误：在成员函数类型中遗漏了 const
{
    return y;
}
```

换句话说，const 是 Date::day()、Date::month() 和 Date::year() 类型的一部分。

const 和非 const 对象都可以调用 const 成员函数，而非 const 成员函数只能被非 const 对象调用。例如：

```
void f(Date& d, const Date& cd)
{
    int i = d.year();    // 正确
    d.add_year(1);       // 正确

    int j = cd.year();   // 正确
    cd.add_year(1);      // 错误：不能改变 const Date 的值
}
```

16.2.9.2 物理常量性和逻辑常量性

有时，一个成员函数逻辑上是 const，但它仍然需要改变成员的值。即对一个用户而言，函数看起来不会改变其对象的状态，但它更新了用户不能直接观察的某些细节。这通常被称为逻辑常量性（logical constness）。例如，类 Date 可能有一个返回字符串表示的函数。构造这个字符串表示非常耗时，因此，保存一个拷贝，在反复要求获取字符串表示时可以简单地返回此拷贝（除非 Date 的值已被改变），这就很有意义了。更复杂的数据结构常使用这种缓存值的技术，但我们现在只讨论对 Date 如何使用这种技术：

```
class Date {
public:
    // ...
    string string_rep() const;    // 字符串表示
private:
    bool cache_valid;
    string cache;
    void compute_cache_value(); // 填入缓存
    // ...
};
```

从用户的角度来看，string_rep 并未改变其 Date 的状态，因此它显然应该是一个 const 成员函数。但另一方面，有时必须改变成员 cache 和 cache_valid，这种设计才能奏效。

此问题可通过使用类型转换来解决，如 const_cast（见 11.5.2 节）。但是，也存在非常优雅的、不破坏类型规则的方法。

16.2.9.3　mutable

我们可以将一个类成员定义为 mutable，表示即使是在 const 对象中，也可以修改此成员：

```
class Date {
public:
    // ...
    string string_rep() const;                  // 字符串表示
private:
    mutable bool cache_valid;
    mutable string cache;
    void compute_cache_value() const;           // 填入（可变的）缓存
    // ...
};
```

现在，我们显然可以这样定义 string_rep()：

```
string Date::string_rep() const
{
    if (!cache_valid) {
        compute_cache_value();
        cache_valid = true;
    }
    return cache;
}
```

现在，string_rep() 既可用于 const 对象，也可用于非 const 对象。例如：

```
void f(Date d, const Date cd)
{
    string s1 = d.string_rep();
    string s2 = cd.string_rep();            // 正确！
    // ...
}
```

16.2.9.4　间接访问实现可变性

对于小对象的表示形式只有一小部分允许改变的情形，将成员声明为 mutable 是最适合的。但在更复杂的情况下，通常更好的方式是将需要改变的数据放在一个独立对象中，间接地访问它们。如果采用这种技术，字符串缓存例程会变为：

```
struct cache {
    bool valid;
    string rep;
};

class Date {
public:
    // ...
    string string_rep() const;                  // 字符串表示
private:
    cache* c;                                   // 在构造函数中初始化
    void compute_cache_value() const;           // 填入 cache 指向的内存
    // ...
};

string Date::string_rep() const
```

```
{
    if (!c->valid) {
        compute_cache_value();
        c->valid = true;
    }
    return c->rep;
}
```

这种支持缓存的编程技术可推广到各种形式的懒惰求值。

注意，const 不能（传递地）应用到通过指针或引用访问的对象。程序的读者可能会认为这种对象是“某种子对象”，但编译器不能将这种指针或引用与其他指针或引用区分开来。即一个成员指针没有任何与其他指针不同的特殊语义。

16.2.10 自引用

我们定义的状态更新函数 add_year()、add_month() 和 add_day()（见 16.2.3 节）是不返回值的。对这样一组相关的更新函数，一种通常很有用的技术是令它们返回已更新对象的引用，这样这些操作就可以串接起来。例如，我们可能想这样将 d 的天、月、年各增加 1：

```
void f(Date& d)
{
    // ...
    d.add_day(1).add_month(1).add_year(1);
    // ...
}
```

为此，必须将每个函数都声明为返回一个 Date 引用：

```
class Date {
    // ...
    Date& add_year(int n);   // 增加 n 年
    Date& add_month(int n); // 增加 n 个月
    Date& add_day(int n);    // 增加 n 天
};
```

每个（非 static）成员函数都知道是哪个对象调用的它，并能显式引用这个对象。例如：

```
Date& Date::add_year(int n)
{
    if (d==29 && m==2 && !leapyear(y+n)) { // 小心 2 月 29 日
        d = 1;
        m = 3;
    }
    y += n;
    return *this;
}
```

表达式 *this 引用的就是调用此成员函数的对象。

在非 static 成员函数中，关键字 this 是指向调用它的对象的指针。在类 X 的非 const 成员函数中，this 的类型是 X*。但是，this 被当作一个右值，因此我们无法获得 this 的地址或给它赋值。在类 X 的 const 成员函数中，this 的类型是 const X*，以防止修改对象（见 7.5 节）。

this 的使用大多数是隐式的。特别是，每当我们引用类内的一个非 static 成员时，都是依赖于一次 this 的隐式使用来获得恰当对象的该成员。例如，函数 add_year 可以定义为下

面这样等价但更繁琐的形式：

```
Date& Date::add_year(int n)
{
    if (this->d==29 && this->m==2 && !leapyear(this->y+n)) {
        this->d = 1;
        this->m = 3;
    }
    this->y += n;
    return *this;
}
```

this 的一种常见的显式应用是用于链表操作。例如：

```
struct Link {
    Link* pre;
    Link* suc;
    int data;

    Link* insert(int x)    // 在 this 之前插入 x
    {
        return pre = new Link{pre,this,x};
    }
    void remove()  // 删除并销毁 this
    {
        if (pre) pre->suc = suc;
        if (suc) suc->pre = pre;
        delete this;
    }

    // ...
};
```

从一个派生类模板访问基类的成员也需要显式使用 this（见 26.3.7 节）。

16.2.11　成员访问

我们可以通过对类 X 的对象使用 .（点）运算符或对 X 的对象的指针使用 ->（箭头）运算符来访问 X 的成员。例如：

```
struct X {
    void f();
    int m;
};

void user(X x, X* px)
{
    m = 1;          // 错误：作用域中没有 m
    x.m = 1;        // 正确
    x->m = 1;       // 错误：x 不是一个指针
    px->m = 1;      // 正确
    px.m = 1;       // 错误：px 是一个指针
}
```

显然这种语法有些冗余：编译器了解一个名字是指向 X 还是 X*，因此单一的运算符就足够了。但是，程序员可能会感到迷惑，因此从最早期的 C 语言开始就使用两个独立的运算符。

在类的内部访问成员则不需要任何运算符。例如：

```
void X::f()
{
    m = 1;          // 正确：等价于 "this->m = 1;"（见 16.2.10 节）
}
```

即一个不带限定的名字就像加了前缀 this-> 一样。注意，成员函数可以在一个成员声明前就引用它：

```
struct X {
    int f() { return m; }  // 正确：返回此 X 的 m
    int m;
};
```

如果我们希望引用类的一个公共成员，而不是某个特定对象的成员，应该使用类名后接 :: 的限定方式。例如：

```
struct S {
    int m;
    int f();
    static int sm;
};

int X::f() { return m; }        // X 的 f
int X::sm {7};                  // X 的静态成员 sm（见 16.2.12 节）
int (S::*) pmf() {&S::f};       // X 的成员 f
```

最后一个语法结构（成员指针）很少见也很难懂（见 20.6 节）。我在这里提及它只是为了强调 :: 规则的通用性。

16.2.12 static 成员

为 Date 设定默认值的确非常方便，但会带来严重的潜在问题，因为 Date 类会依赖全局变量 today。这样的 Date 类只能用于定义和正确使用 today 的上下文中。这就限制一个类只有在最初编写它的上下文中才有用。尝试使用这种上下文依赖的类会给用户带来很多意料之外的不快，代码维护也会变得很混乱。可能“只是一个小小的全局变量”不是那么难以管理，但这种风格所产生的代码对其编写者之外的人几无用处，因此应该避免。

幸运的是，我们获得这种便利性其实并不需要承担使用可公开访问的全局变量的负担。是类的一部分但不是某个类对象一部分的变量称为 static 成员。static 成员只有唯一副本，而不是像普通非 static 成员那样每个对象都有其副本（见 6.4.2 节）。类似地，需要访问类成员而不需要通过特定对象调用的函数称为 static 成员函数。

下面是重新设计的版本，它保留了 Date 默认构造函数值的语义，又没有依赖全局变量所带来的问题：

```
class Date {
    int d, m, y;
    static Date default_date;
public:
    Date(int dd =0, int mm =0, int yy =0);
    // ...
    static void set_default(int dd, int mm, int yy); // 将 default_date 设置为 Date(dd,mm,yy)
};
```

现在我们可以定义使用 default_date 的 Date 构造函数如下：

```
Date::Date(int dd, int mm, int yy)
{
    d = dd ? dd : default_date.d;
    m = mm ? mm : default_date.m;
    y = yy ? yy : default_date.y;

    // … 检查 Date 是否合法 …
}
```

使用 set_default()，可以在恰当的时候改变默认值。可以像引用任何其他成员一样引用 static 成员。此外，不必提及任何对象即可引用 static 成员，方法是使用其类的名字作为限定。例如：

```
void f()
{
    Date::set_default(4,5,1945);    // 调用 Date 的 static 成员 set_default()
}
```

如果使用了 static 函数或数据成员，我们就必须在某处定义它们。在 static 成员的定义中不要重复关键字 static。例如：

```
Date Date::default_date {16,12,1770};          // Date::default_date 的定义

void Date::set_default(int d, int m, int y)    // Date::set_default 的定义
{
    default_date = {d,m,y};                    // 将新值赋予 default_date
}
```

现在，默认值就变为贝多芬的生日，直到某人决定将其变为其他日期为止。

注意，Date{} 表示 Date::default_date 的值。例如：

```
Date copy_of_default_date = Date{};

void f(Date);

void g()
{
    f(Date{});
}
```

因此，我们不需要一个独立的函数来读取默认值。而且，当目标类型为 Date 无疑时，更简单的 {} 就足够了。例如：

```
void f1(Date);

void f2(Date);
void f2(int);

void g()
{
    f1({});          // 正确：等价于 f1(Date{})
    f2({});          // 错误：二义性，f2(int) 还是 f2(Date)？
    f2(Date{});      // 正确
```

在多线程代码中，static 数据成员需要某种锁机制或访问规则来避免竞争条件（见 5.3.4 节和 41.2.4 节）。多线程现在已经非常常见了，不幸的是旧代码中 static 数据成员的使用非常普

遍，而且使用方式隐含着竞争条件。

16.2.13 成员类型

类型和类型别名也可以作为类的成员。例如：

```
template<typename T>
class Tree {
    using value_type = T;                    // 成员别名
    enum Policy { rb, splay, treeps };       // 成员枚举
    class Node {                             // 成员类
        Node* right;
        Node* left;
        value_type value;
    public:
        void f(Tree*);
    };
    Node* top;
public:
    void g(const T&);
    // ...
};
```

成员类（member class，常称为嵌套类，nested class）可以引用其所属类的类型和 static 成员。当给定所属类的一个对象时，只能引用非 static 成员。为了避免陷入复杂的二叉树结构，我只使用“f()”和“g()”风格的例子。

嵌套类可以访问其所属类的成员（甚至是 private 成员，这方面与成员函数类似），但它没有当前类对象的概念。例如：

```
template<typename T>
void Tree::Node::f(Tree* p)
{
    top = right;                 // 错误：未指定类型为 Tree 的对象
    p->top = right;              // 正确
    value_type v = left->value;  // 正确：value_type 不与某个对象关联
}
```

相反，一个类并没有任何特殊权限能访问其嵌入类的成员。例如：

```
template<typename T>
void Tree::g(Tree::Node* p)
{
    value_type val = right->value;       // 错误：没有 Tree::Node 类型的对象
    value_type v = p->right->value;      // 错误：Node::right 是私有的
    p->f(this);                          // 正确
}
```

成员类更多的是提供了一种符号表示上的便利，而非一种重要的语言特性。另一方面，成员别名非常重要，它是依赖于关联类型（见 28.2.4 节和 33.1.3 节）的泛型编程技术的基础。成员 enum 通常作为 enum class 的替代，以避免与外围作用域中和枚举值同名的实体产生冲突（见 8.4.1 节）。

16.3 具体类

上一节在介绍类定义基本语言特性的过程中讨论了 Date 类的部分设计。在本节中，我将改变重点，讨论如何设计一个简单高效的 Date 类并展示语言特性如何支持这个设计。

在很多应用中经常大量使用较小的抽象。这种例子包括拉丁字符、中文字符、整数、浮点数、复数、点、指针、坐标、变换、(指针，偏移量)对、时间、范围、链接、联合、结点、(值，单元)对、磁盘位置、源码位置、货币值、线、矩形、缩放的定点数、分数、字符串、向量以及数组。每个应用都使用若干这种小的抽象，其中一些简单的具体类型经常被大量使用。一个典型的应用会直接使用一些类型，还有更多的类型是通过库间接使用的。

C++ 直接支持一部分抽象作为其内置类型。但是，大多数抽象并不直接支持，C++ 语言也难以做到这一点，因为数量实在是太多了。而且，一个通用编程语言的设计者不可能预见所有应用的细节需求。因此，语言必须为用户提供定义小的具体类型的机制。这种类型称为*具体类型*（concrete type）或*具体类*（concrete class），以区别于抽象类（见 20.4 节）和类层次中的类（见 20.3 节和 21.2 节）。

如果一个类的表示是其定义的一部分，我们就称它是*具体的*（concrete，或称它是一个*具体类*）。这将它与抽象类（见 3.2.2 节和 20.4 节）区分开来，后者为多种实现提供一个公共接口。在定义中明确类的表示方式令我们能：

- 将对象置于栈、静态分配的内存以及其他对象中；
- 拷贝和移动对象（见 3.3 节和 17.5 节）；
- 直接引用具名对象（与通过指针和引用访问不同）。

这令具体类易于推断，编译器也容易为之生成优化的代码。因此，我们倾向于对频繁使用且性能攸关的小类型使用具体类，例如复数（见 5.6.2 节）、智能指针（见 5.2.1 节）和容器（见 4.4 节）。

很好地支持这种用户自定义类型的定义和使用是 C++ 早期就明确的目标，这是优雅的程序设计的基础。总的来说，简单和平凡要比复杂和精致重要得多。由此，我们来设计一个更好的类：

```
namespace Chrono {

    enum class Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    class Date {
    public:          // 公共接口：
        class Bad_date { };  // 异常类

        explicit Date(int dd ={}, Month mm ={}, int yy ={});          // 表示"选择默认值"
    // 非修改性函数，用于查询 Date:
        int day() const;
        Month month() const;
        int year() const;

        string string_rep() const;                  // 字符串表示
        void char_rep(char s[], in max) const;      // C 风格字符串表示

    // 修改性函数，用于改变 Date:
        Date& add_year(int n);                      // 增加 n 年
        Date& add_month(int n);                     // 增加 n 个月
        Date& add_day(int n);                       // 增加 n 天
    private:
        bool is_valid();                            // 检查 Date 是否表示一个日期
        int d, m, y;                                // 类的表示
    };
```

```
    bool is_date(int d, Month m, int y);              // 对合法日期返回 true
    bool is_leapyear(int y);                          // 若 y 是闰年返回 true

    bool operator==(const Date& a, const Date& b);
    bool operator!=(const Date& a, const Date& b);

    const Date& default_date();                       // 默认日期

    ostream& operator<<(ostream& os, const Date& d);  // 将 d 打印到 os
    istream& operator>>(istream& is, Date& d);        // 从 is 读取 Date 存入 d
} // Chrono
```

这组操作对一个用户自定义类型而言是非常典型的：

[1] 一个构造函数指出此类型的对象 / 变量如何初始化（见 16.2.5 节）。

[2] 一组允许用户检查 Date 的函数。这些函数标记为 const，表明它们不会修改调用它们的对象 / 变量的状态。

[3] 一组允许用户无须了解表示细节也无须摆弄复杂语法即可修改 Date 的函数。

[4] 隐式定义操作，允许 Date 自由拷贝（见 16.2.2 节）。

[5] 类 Bad_date，用来报告错误、抛出异常。

[6] 一组有用的辅助函数。这些函数不是类成员，不能直接访问 Date 的表示，但我们认为它们与名字空间 Chrono 的使用是相关的。

我定义了一个类型 Month 来处理月 / 天顺序的问题，例如避免 6 月 7 日是写成 {6,7}（美国风格）还是写成 {7,6}（欧洲风格）的混淆。

我考虑过引入两个不同的类型 Day 和 Year 来处理 Date{1995,Month::jul,27} 和 Date{27,Month::jul,1995} 这种可能的混淆。但是，这些类型不如类型 Month 那么有用。毕竟几乎所有这类错误都是在运行时被捕获的——公元 27 年 7 月 26 日在我的工作中不是一个常见的日期。处理大约 1800 年之前的历史日期是非常棘手的问题，最好留给历史专家。而且，脱离了年、月是无法正确检查日子的值的。

当上下文已经暗示了年和月时，为了使用户能不必再显式提及年月，我增加了一种提供默认值的机制。注意，对 Month 而言，{} 给出的是（默认）值 0，就像整数的初始化那样，而 0 其实不是一个合法的 Month（见 8.4 节）。但是，在本例中，这正是我们所期望的：用一个非法值表示“选择默认值”。提供默认值（如 Date 对象的默认值）是一个困难的设计问题。某些类型有公认的默认值（如整数的默认值 0）；其他一些类型不存在有意义的默认值；还有一些类型（如 Date），是否应为它们提供默认值的问题不是那么简单。对于这些类型，最好（至少在开始时）不要提供默认值。我为 Date 提供了一个默认值，这主要是为了讨论技术本身。

我忽略了 16.2.9 节中的缓存技术，因为对这么简单的类型并无必要。如果需要的话，我们可以为类型增加缓存机制，这属于实现细节，不会影响用户接口。

下面是一个简单的、有些不自然的使用 Date 的例子：

```
void f(Date& d)
{
    Date lvb_day {16,Month::dec,d.year()};

    if (d.day()==29 && d.month()==Month::feb) {
        // ...
    }
```

```
    if (midnight()) d.add_day(1);

    cout << "day after:" << d+1 << '\n';

    Date dd;  // 初始化为默认值
    cin>>dd;
    if (dd==d) cout << "Hurray!\n";
}
```

这段代码假定已为 Date 声明了加法运算符 +，我将在 16.3.3 节实现它。

注意对 dec 和 feb 使用的显式限定 Month。我专门使用了一个 enum class（见 8.4.1 节）以便能使用月份名的简写，同时也保证这些简写的使用不会模糊或有二义性。

为什么对日期这样简单的东西值得定义一个专门的类型呢？毕竟，我们可以只定义一个简单的数据结构：

```
struct Date {
    int day, month, year;
};
```

然后所有程序员都可以决定用它做什么。但如果我们这样做了，每个用户要么必须直接操作 Date 的组件，要么必须提供独立的函数来完成操作。实际中的效果就是，日期的概念会散布到整个系统中，令代码难以理解、难以编写文档也难以修改。如果我们将一个概念只是实现为一个简单结构，会不可避免地给每个用户带来额外工作。

而且，即使 Date 类型看起来很简单，它也至少体现了一些正确的思想。例如，递增 Date 时必须考虑闰年、每个月天数不同，等等。此外，天 - 月 - 年的表示方式对很多应用来说相当糟糕，如果我们决定修改这种表示方式，只需修改一组指定的函数即可。例如，为了将表示方式改为自 1970 年 1 月 1 日之后的天数，我们只需修改 Date 的成员函数即可。

为简单起见，我决定去掉改变默认日期的功能。这样能消除产生混淆的机会以及在多线程程序中产生竞争条件的可能（见 5.3.1 节）。我认真考虑了同时去掉默认日期的概念。这样做可以强制用户始终显式地初始化他们的 Date。但是，这有些不便和奇怪，而且更重要的是，用于通用代码的公共接口是要求默认构造的（见 17.3.3 节）。这意味着我作为 Date 的设计者必须选定一个默认日期。我选择了 1970 年 1 月 1 日，因为这是 C 和 C++ 标准库时间例程的起始时间（见 35.2 节和 43.6 节）。显然，去掉 set_default_date() 会降低 Date 的通用性。但是，进行设计（包括类的设计）就是要做出决策，而不只是决定推迟决策或将所有选择都留给用户。

为了保留未来进一步优化的机会，我将 default_date() 声明为一个辅助函数：

```
const Date& Chrono::default_date();
```

它并未描述任何有关如何真正设置默认日期的内容。

16.3.1　成员函数

自然地，对每个成员函数我们都必须在某处给出它的实现。例如：

```
Date::Date(int dd, Month mm, int yy)
    :d{dd}, m{mm}, y{yy}
{
    if (y == 0) y = default_date().year();
    if (m == Month{}) m = default_date().month();
```

```
    if (d == 0) d = default_date().day();

    if (!is_valid()) throw Bad_date();
}
```

这个构造函数检查提供的数据是否表示一个合法的Date。如果日期不合法，如{30,Month::feb,1994}，它会抛出一个异常（见2.4.3.1节和第13章），表示产生了错误。如果提供的数据是合法的，就进行相应的初始化。初始化操作相对复杂，因为它包含了数据验证步骤。本例是初始化操作一个很典型的实现。另一方面，一旦Date已创建，即可使用和拷贝而无须再进行检查。换句话说，构造函数建立了类的不变式（在本例中，不变式就是一个合法日期）。其他成员函数可依赖于不变式且必须保持不变式。这种设计技术可以极大地简化代码（见2.4.3.2节和13.4节）。

我使用值Month{}表示“选择默认月份”，它是整数值0，不表示一个月份。我本可选择用Month中的一个枚举值专门表示默认月份，但我认为用一个明显的异常值表示“选择默认月份”比给人一年13个月的错觉要更好。注意，Month{}表示0，是可用来表示月份的，因为它在枚举类型Month保证的取值范围内（见8.4节）。

我使用成员初始化器语法（见17.4节）初始化成员。之后，我检查值是否为0并在需要时修改值。在发生错误时（希望这种情况很少出现）这种方法显然不能提供最优性能，但使用成员初始化器会令代码的结构非常清晰。这种编程风格比其他风格更不容易出错也更易维护。假如我的目标是最优性能，我就必须使用3个独立的构造函数而非带默认参数的单一构造函数。

我考虑过将验证函数is_valid()实现为一个公有函数。但是，我发现得到的用户代码比依赖于异常缓存的代码更为复杂，健壮性更差：

```
void fill(vector<Date>& aa)
{
    while (cin) {
        Date d;
        try {
            cin >> d;
        }
        catch (Date::Bad_date) {
            // ... 我的错误处理代码 ...
            continue;
        }
        aa.push_back(d);    // 参见4.4.2节
    }
}
```

但是，检查值集合{d,m,y}是否是合法日期并不依赖于Date的表示，因此我将is_valid()实现为一个辅助函数：

```
bool Date::is_valid()
{
    return is_date(d,m,y);
}
```

既然定义了is_valid()为什么又要定义is_date()呢？在这个简单的例子中，我们可以只使用其中一个，但我可以想象在更复杂的系统中is_date()（像本例中一样）检查一个(d,m,y)元组是否表示一个合法日期，而is_valid()进一步检查日期是否可以合理表示。例如，is_

valid() 可能拒绝现代日历普遍使用前的日期。

与这类简单具体类型的常见情况一样，Date 的成员函数的定义都是介于简单和不那么复杂之间。例如：

```
inline int Date::day() const
{
    return d;
}
Date& Date::add_month(int n)
{
    if (n==0) return *this;

    if (n>0) {
        int delta_y = n/12;                          // 整年数
        int mm = static_cast<int>(m)+n%12;           // 剩余月数
        if (12 < mm) {                               // 注意：dec 用 12 表示
            ++delta_y;
            mm -= 12;
        }

        // ... 处理 mm 月没有第 d 天的情况 ...

        y += delta_y;
        m = static_cast<Month>(mm);
        return *this;
    }

    // ... 处理负数 n ...

    return *this;
}
```

我不能说 add_month() 的代码是完美的。实际上，如果加入所有细节，它的复杂程度甚至可能接近相对简单的实际代码。这揭示了一个问题：将月份值加 1 在概念上很简单，那么我们的代码为什么会这么复杂？在本例中，原因在于 d,m,y 的表示方式对于我们人类来说很方便，但对于计算机就不是那么方便了。（对很多应用目标来说）更好的表示方式是简单地使用自“零日”（如 1970 年 1 月 1 日）起的天数。这种表示方式令 Date 的计算变得很简单，付出的代价是生成适合人类的输出较为复杂。

注意，Date 默认提供赋值和拷贝初始化（见 16.2.2 节）。而且，Date 不需要析构函数，因为 Date 不拥有资源，因此在离开作用域时不需要清理操作（见 3.2.1.2 节）。

16.3.2　辅助函数

一般而言，一个类都会有一些无须定义在类内的关联函数，因为它们不需要直接访问类的表示。例如：

```
int diff(Date a, Date b);    // [a,b) 或 [b,a) 间的天数

bool is_leapyear(int y);
bool is_date(int d, Month m, int y);

const Date& default_date();
Date next_weekday(Date d);
Date next_saturday(Date d);
```

将这种函数定义在类内会增加类接口的复杂性，还会增加那些考虑修改类表现时需要检查的函数数目。

这种函数如何与类 Date "关联"呢？在早期 C++ 中，与 C 语言一样，这些函数的声明简单地放在类 Date 声明所在的文件中。Date 的用户只需包含定义接口的文件即可使用所有辅助函数（见 15.2.2 节）。例如：

```
#include "Date.h"
```

此外，还有一种替代方法是将类及其辅助函数放在一个名字空间中来显式表明两者的关联（见 14.3.1 节）：

```
namespace Chrono {          // 处理时间的特性

    class Date { /* ... */};

    int diff(Date a, Date b);
    bool is_leapyear(int y);
    bool is_date(int d, Month m, int y);
    const Date& default_date();
    Date next_weekday(Date d);
    Date next_saturday(Date d);
    // ...
}
```

名字空间 Chrono 自然还可能包含 Time 和 Stopwatch 等相关的类及它们的辅助函数。用一个名字空间保存单一类通常过于精细，会导致某些不便。

自然地，我们必须在某处定义辅助函数：

```
bool Chrono::is_date(int d, Month m, int y)
{
    int ndays;

    switch (m) {
    case Month::feb:
        ndays = 28+is_leapyear(y);
        break;
    case Month::apr: case Month::jun: case Month::sep: case Month::nov:
        ndays = 30;
        break;
    case Month::jan: case Month::mar: case Month::may: case Month::jul:
    case Month::aug: case Month::oct: case Month::dec:
        ndays = 31;
        break;
    default:
        return false;
    }

    return 1<=d && d<=ndays;
}
```

我在这里故意有些偏执。Month 不应在 jan 到 dec 的范围之外，但它确实有可能越界（某人可能随意进行了类型转换），因此我检查了 Month 的值。

麻烦的 default_date 最终变为：

```
const Date& Chrono::default_date()
{
    static Date d {1,Month::jan,1970};
    return d;
}
```

16.3.3　重载运算符

添加一些函数使得用户自定义类型能使用人们习惯的符号通常是很有用的。例如，operator==() 定义了 Date 的相等判定运算符 ==：

```
inline bool operator==(Date a, Date b)          // 相等判定
{
    return a.day()==b.day() && a.month()==b.month() && a.year()==b.year();
}
```

其他可能的运算符包括：

```
bool operator!=(Date, Date);          // 不等
bool operator<(Date, Date);           // 小于
bool operator>(Date, Date);           // 大于
// ...

Date& operator++(Date& d) { return d.add_day(1); }          // 增加一天
Date& operator--(Date& d) { return d.add_day(-1); }         // 减少一天

Date& operator+=(Date& d, int n) { return d.add_day(n); }   // 增加 n 天
Date& operator-=(Date& d, int n) { return d.add_day(-n); }  // 减少 n 天

Date operator+(Date d, int n) { return d+=n; }              // 加 n 天
Date operator-(Date d, int n) { return d+=n; }              // 减 n 天

ostream& operator<<(ostream&, Date d);                      // 输出 d
istream& operator>>(istream&, Date& d);                     // 读入 d
```

这些运算符都与 Date 一起定义在 Chrono 中，以避免重载问题以及从参数依赖查找受益（见 14.2.4 节）。

对 Date 而言，这些运算符可以看作一种便利机制。但是，对很多类型而言，例如复数（见 18.3 节）、向量（见 4.4.1 节）和类函数对象（见 3.4.3 节和 19.2.2 节），常规运算符的使用在人们头脑中如此根深蒂固，以至于定义这些运算符几乎是强制的。运算符重载将在第 18 章介绍。

我曾忍不住想将 += 和 -= 实现为 Date 的成员函数来取代 add_day()。假如我这么做了，就遵循了常见的做法（见 3.2.1.1 节）。

注意，赋值和拷贝初始化是默认提供的（见 16.3 节和 17.3.3 节）。

16.3.4　具体类的重要性

我称 Date 这样的简单用户自定义类型为具体类型（concrete type），以区别于抽象类（见 3.2.2 节）和类层次（见 20.4 节），并强调它们与 int 和 char 这样的内置类型相似。具体类的使用就像内置类型一样。具体类型也称为值类型（value type），使用它们编程称为面向值的程序设计（value-oriented programming）。它们的使用模型和设计背后的“哲学”与常

见的面向对象编程（见3.2.4节和第21章）非常不同。

一个具体类型的目标是高效地做好一件相对简单的事情，为用户提供修改其行为的特性通常不是其目标。特别是，展现运行时多态行为也不是其意图（见3.2.3节和20.3.2节）。

如果不喜欢一个具体类型的某些细节，你可以构建一个新的类型，只要实现所需行为即可。如果希望“重用”一个具体类型，你可以用它实现新类型，就像使用int一样。例如：

```
class Date_and_time {
private:
    Date d;
    Time t;
public:
    Date_and_time(Date d, Time t);
    Date_and_time(int d, Date::Month m, int y, Time t);
    // ...
};
```

一种替代方法是派生类机制，可用来从具体类派生新类型，只要描述两者的差异即可，我将在第20章介绍派生类。从vector定义Vec（见4.4.1.2节）就是这种技术的一个例子。但是，我们极少从一个具体类派生新类，即使这样做的话也要特别小心，原因是具体类型缺少虚函数和运行时类型信息（见17.5.1.4节和第22章）。

如果有一个很好的编译器，Date这样的具体类不会有隐含的时空开销。特别是，无须通过指针间接访问具体类对象，也无须在具体类对象中保存“簿记”数据。具体类型的大小在编译时就已知，从而可在运行时栈中分配对象（即，无须自由存储空间操作）。对象的内存布局也在编译时就已知道，从而内联操作很容易实现。类似地，无须特殊努力即可实现与其他语言，如C和Fortran在内存布局上的兼容。

一组好的具体类型能构成应用程序的基础。特别是，使用它们可令接口更为具体、更不容易出错。例如：

```
Month do_something(Date d);
```

比下面的接口更不容易误解，也更不容易误用：

```
int do_something(int d);
```

如果程序员编写代码时不是使用具体类型，而是简单聚合内置类型来表示“简单且频繁使用的”数据结构，并直接操作这种数据结构，就会产生含混的程序，浪费时间。另一方面，在应用程序中不使用“小而高效的类型”，而是使用过分通用且代价高昂的类，会导致明显的运行时间和空间上的低效。

16.4 建议

[1] 将概念表示为类；16.1节。

[2] 将类的接口与实现分离；16.1节。

[3] 仅当数据真的仅仅是数据且数据成员不存在有意义的不变式时才使用公有数据（struct）；16.2.4节。

[4] 定义构造函数来处理对象初始化；16.2.5节。

[5] 默认将单参数构造函数声明为explicit；16.2.6节。

［6］ 将不修改其对象状态的成员函数声明为 const；16.2.9 节。

［7］ 具体类型是最简单的类。只要适用，就应该优先选择具体类型而不是更复杂的类或普通数据结构；16.3 节。

［8］ 仅当函数需要直接访问类的表示时才将其实现为成员函数；16.3.2 节。

［9］ 使用名字空间建立类与其辅助函数间的显式关联；16.3.2 节。

［10］ 将不修改对象值的成员函数定义为 const 成员函数；16.3.2 节。

［11］ 若一个函数需要访问类的表示，但并不需要用某个具体对象来调用，建议将其实现为 static 成员函数；16.2.12 节。

构造、清理、拷贝和移动

无知比知识更常带来信心。

——查尔斯·达尔文

- 引言
- 构造函数和析构函数
  构造函数和不变式；析构函数和资源；基类和成员析构函数；调用构造函数和析构函数；virtual 析构函数
- 类对象初始化
  不使用构造函数进行初始化；使用构造函数进行初始化；默认构造函数；初始化器列表构造函数
- 成员和基类初始化
  成员初始化；基类初始化器；委托构造函数；类内初始化器；static 成员初始化
- 拷贝和移动
  拷贝；移动
- 生成默认操作
  显式声明默认操作；默认操作；使用默认操作；使用 delete 删除的函数
- 建议

17.1 引言

本章主要介绍与对象的“生命周期”有关的技术：我们如何创建对象、如何拷贝对象，如何移动对象以及在对象销毁时如何进行清理工作？首先，“拷贝”和“移动”的恰当定义是什么？例如：

```
string ident(string arg)          // 传值方式传递 string（拷贝到 arg 中）
{
    return arg;                   // 返回 string（将 arg 的值移出 ident()，移动到其调用者中
}

int main ()
{
    string s1 {"Adams"};          // 初始化 string（在 s1 中构造）
    s1 = indet(s1);               // 拷贝 s1 到 ident() 中
                                  // 将 ident(s1) 的结果移动到 s1 中；
                                  // s1 的值为 "Adams"。
    string s2 {"Pratchett"};      // 初始化 string（在 s2 中构造）
    s1 = s2;                      // 将 s2 的值拷贝到 s1 中
                                  // s1 和 s2 的值都为 "Pratchett"。
}
```

显然，调用 ident() 后，s1 的值应该为“Adams”。我们将 s1 的值拷贝到参数 arg 中，然后将 arg 的值移出函数，移回 s1 中。接下来，我们构造 s2，将其初始化为“Pratchett”

并将其拷贝到 s1 中。最后，在 main() 退出时我们销毁变量 s1 和 s2。移动（move）和拷贝（copy）的区别在于，拷贝操作后两个对象具有相同的值，而移动操作后移动源不一定具有其原始值。如果源对象在操作后不再使用，我们就可以使用移动操作。在实现资源移动概念（见 3.2.1.2 节和 5.2 节）时，移动操作特别有用。

这个例子中使用了多个函数：

- 用字符串字面值常量初始化 string 的构造函数（用于 s1 和 s2）
- 拷贝 string 的拷贝构造函数（拷贝到函数参数 arg 中）
- 移动 string 值的移动构造函数（将 arg 的值移出 ident()，移动到保存 ident(s1) 结果的临时变量中）
- 移动 string 值的移动赋值操作（从保存 ident(s1) 结果的临时变量移动到 s1）
- 拷贝 string 值的拷贝赋值操作（从 s2 拷贝到 s1）
- 析构函数释放 s1、s2 和保存 ident(s1) 结果的临时变量所拥有的资源，对移动源——函数参数 arg 不做任何事情。

优化器可能消除掉一部分工作。例如，在这个简单的例子中，临时变量通常会被消除。但是，原则上这些操作都会执行。

构造函数、拷贝和移动赋值操作以及析构函数直接支持生命周期和资源管理的视角。当一个对象的构造函数完成后，它被看作其类型的对象，并保持到其析构函数开始执行。对象生命周期和错误之间的相互作用在 13.2 节和 13.3 节中有进一步的介绍。特别是，本章不讨论半构造和半销毁的对象。

对象的构造在很多设计中起着关键作用，这种广泛应用也反映在语言特性对初始化支持的范围和灵活性上。

一个类型的构造函数、析构函数以及拷贝和移动操作在逻辑上不是相互独立的。我们定义的这组函数必须相互匹配，否则就会引起逻辑问题或性能问题。如果一个类 X 的析构函数执行很重要的任务，如释放自由存储空间或释放锁，那么这个类很可能需要一组完整的函数：

```
class X {
    X(Sometype);                // "普通构造函数"：创建一个对象
    X();                        // 默认构造函数
    X(const X&);                // 拷贝构造函数
    X(X&&);                     // 移动构造函数
    X& operator=(const X&);     // 拷贝赋值运算符：清理目标对象并进行拷贝
    X& operator=(X&&);          // 移动赋值运算符：清理目标对象并进行移动
    ~X();                       // 析构函数：清理
    // ...
};
```

一个对象在 6 种情况下会被拷贝或移动：

- 作为赋值操作的源
- 作为一个对象初始化器
- 作为一个函数实参
- 作为一个函数返回值
- 作为一个异常

在所有这些情况下，都会应用拷贝或移动构造函数（除非它们都可被优化掉）。

除了初始化具名对象和自由存储上的对象，构造函数还用来初始化临时对象（见 6.4.2 节）以及实现显式类型转换（见 11.5 节）

除了“普通构造函数”，这些特殊成员函数都可以由编译器自动生成；参见 17.6 节。

本章充满了规则和术语。为了完全理解本章内容，理解这些规则和术语都是必需的，不过大多数人可以仅从示例来学习一般规则。

17.2　构造函数和析构函数

我们可以通过定义一个构造函数（见 16.2.5 节和 17.3 节）来指出一个类的对象应如何初始化。与构造函数对应，我们还可以定义一个析构函数来确保对象销毁时（例如，当对象离开作用域时）进行恰当的“清理操作”。C++ 中某些最有效的资源管理技术都依赖于构造函数 / 析构函数这对搭档。类似地，其他一些技术也依赖于操作对，如执行 / 撤销、启动 / 停止、前置 / 后置操作，等等。例如：

```
struct Tracer {
    string mess;
    Tracer(const string& s) :mess{s} { clog << mess; }
    ~Tracer() {clog << "~" << mess; }
};

void f(const vector<int>& v)
{
    Tracer tr {"in f()\n"};
    for (auto x : v) {
        Tracer tr {string{"v loop "}+to<string>(x)+"\n"}; // 见 25.2.5.1 节
        // ...
    }
}
```

我们可以尝试调用：

```
f({2,3,5});
```

这个调用会将下面内容打印到日志流：

```
in_f()
v loop 2
~v loop 2
v loop 3
~v loop 3
v loop 5
~v loop 5
~in_f()
```

17.2.1　构造函数和不变式

与类同名的成员称为构造函数（constructor）。例如：

```
class Vector {
public:
    Vector(int s);
    // ...
};
```

构造函数的声明指出其参数列表（与一个函数的参数列表完全一样）。但未指出返回类型。类名不可用于此类内的普通成员函数、数据成员、成员类型，等等。例如：

```
struct S {
    S();                    // 很好
```

```
    void S(int);            // 错误：不能为构造函数指定返回类型
    int S;                  // 错误：类名只能表示构造函数
    enum S { foo, bar };    // 错误：类名只能表示构造函数
};
```

构造函数的任务是初始化该类的一个对象。一般而言，初始化操作必须建立一个类不变式（class invariant），所谓不变式就是当成员函数（从类外）被调用时必须保持的某些东西。考虑下面代码：

```
class Vector {
public:
    Vector(int s);
    // ...
private:
    double* elem; // elem 指向一个数组，保存 sz 个 double
    int sz;       // sz 非负
};
```

在本例中（通常情况下也都是这样做），我们用注释陈述不变式："elem 指向一个数组，保存 sz 个双精度浮点数"以及"sz 非负"。构造函数必须保证这两点为真。例如：

```
Vector::Vector(int s)
{
    if (s<0) throw Bad_size(s);
    sz = s;
    elem = new double[s];
}
```

此构造函数尝试建立不变式，如果失败，它就抛出一个异常。如果构造函数无法建立不变式，则不应创建对象且必须确保没有资源泄漏（见 5.2 节和 13.3 节）。需要获取并在用完后最终（显式或隐式地）归还（释放）的任何东西都是资源，例如内存（见 3.2.1.2 节）、锁（见 5.3.4 节）、文件句柄（见 13.3 节）以及线程句柄（见 5.3.1 节）。

为什么应该定义一个不变式呢？这是为了：

- 聚焦于类的设计工作上（见 2.4.3.2 节）；
- 理清类的行为（如错误状态下的行为；见 13.2 节）；
- 简化成员函数的定义（见 2.4.3.2 节和 16.3.1 节）；
- 理清类的资源管理（见 13.3 节）；
- 简化类的文档。

通常，设计不变式最终会节省我们的总工作量。

17.2.2　析构函数和资源

构造函数初始化对象。换句话说，它创建供成员函数进行操作的环境。创建环境有时需要获取资源，如文件、锁或者一些内存，这些资源在使用后必须释放（见 5.2 节和 13.3 节）。因此，某些类需要一个函数，在对象销毁时保证它会被调用，就像在对象创建时保证构造函数会被调用一样。这样的函数就必然被称为析构函数（destructor）了。析构函数的名字是~后接类名，例如~Vector()。~的一种含义是"补"（见 11.1.2 节），而一个类的析构函数恰好与其构造函数互补。析构函数不接受参数，每个类只能有一个析构函数。当一个自动变量离开作用域时，自由空间中的一个对象被释放时，等等时刻，析构函数会被隐式调用。只有在极少数情况下用户才需要显式调用析构函数（见 17.2.4 节）。

析构函数一般进行清理工作并释放资源。例如：

```
class Vector {
public:
    Vector(int s) :elem{new double[s]}, sz{s} { };          // 构造函数：获取内存
    ~Vector() { delete[] elem; }                            // 析构函数：释放内存
    // ...
private:
    double* elem; // elem 指向一个数组，保存 sz 个 double
    int sz;       // sz 表示
};
```

例如有下面的代码：

```
Vector* f(int s)
{
    Vector v1(s);
    // ...
    return new Vector(s+s);
}

void g(int ss)
{
    Vector* p = f(ss);
    // ...
    delete p;
}
```

在本例中，当退出 f() 时 Vector v1 会被销毁。而且，f() 用 new 在自由存储空间中创建的 Vector 被 g() 通过调用 delete 销毁。在这两种情况下，Vector 析构函数都会被调用，释放构造函数分配的内存。

如果构造函数未能获取足够内存会怎么样？例如，s*sizeof(double) 或 (s+s)*sizeof(double) 可能大于可用内存量（单位为字节）。在此情况下，new 会抛出一个 std::bad_alloc 异常（见 11.2.3 节）。异常处理机制会调用恰当的析构函数，从而所有已获取的内存（也只有这些内存）会被释放掉（见 13.5.1 节）。

这种基于构造函数 / 析构函数的资源管理风格被称为资源获取即初始化（Resource Acquisition Is Initialization）或简称 RAII（见 5.2 节和 13.3 节）。

一对匹配的构造函数 / 析构函数是 C++ 中实现可变大小对象的常用机制。标准库容器，如 vector 和 unordered_map，都使用这种技术的变体来为它们的元素提供存储空间。

没有声明析构函数的类型，如内置类型，被认为有一个不做任何事情的析构函数。

如果程序员为一个类声明了析构函数，那么他还必须决定类对象是否可以拷贝或移动（见 17.6 节）。

17.2.3　基类和成员析构函数

构造函数和析构函数可以很好地与类层次配合（见 3.2.4 节和第 20 章）。构造函数会“自顶向下”地创建一个类对象：

[1] 首先，构造函数调用其基类的构造函数，

[2] 然后，它调用成员的构造函数，

[3] 最后，它执行自身的函数体。

析构函数则按相反顺序“拆除”一个对象：

［1］ 首先，析构函数执行自身的函数体，

［2］ 然后，它调用其成员的析构函数，

［3］ 最后，它调用其基类的析构函数。

特别是，一个 virtual 基类必须在任何可能使用它的基类之前构造，并在它们之后销毁（见 21.3.5.1 节）。这种顺序保证了一个基类或一个成员不会在它初始化完成之前或已销毁之后使用。程序员可以击败这一简单而基本的规则，但只能通过故意规避它才能做到，需要将指向未初始化变量的指针作为参数传递。这样做违反语言规则而且结果通常是灾难性的。

构造函数按声明顺序（而非初始化器的顺序）执行成员和基类的构造函数；如果两个构造函数使用了不同的顺序，析构函数不能保证（即使能保证也会有严重的额外开销）按构造的相反顺序进行销毁。参见 17.4 节。

如果一个类的使用方式要求有默认构造函数，或者类没有其他构造函数，则编译器会尝试生成一个默认构造函数。例如：

```
struct S1 {
    string s;
};

S1 x;    // 正确：x.s 初始化为 ""
```

类似地，如果需要初始化器，可以使用逐成员初始化。例如：

```
struct X { X(int); };

struct S2 {
    X x;
};

S2 x1;    // 错误：没有为 x1.x 提供值
S2 x2 {1}; // 正确：x2.x 用 1 进行初始化
```

请参见 17.3.1 节。

17.2.4　调用构造函数和析构函数

当对象退出作用域或被 delete 释放时，析构函数会被隐式调用。显式调用析构函数通常是不必要的，而且会导致严重的错误。但是，在极少数（但很重要的）情况下我们必须显式调用析构函数。考虑一个容器（如 std::vector）维护一个可增长和缩减（例如使用 push_back() 和 pop_back()）的内存池。当我们添加一个元素时，容器必须对一个特定地址调用其构造函数：

```
void C::push_back(const X& a)
{
    // ...
    new(p) X{a};    // 在地址 p 用值 a 拷贝构造一个 X
    // ...
}
```

构造函数的这种用法被称为“放置式 new”（见 11.2.4 节）。

相反地，当我们删除一个元素时，容器需要调用其析构函数：

```
void C::pop_back()
{
    // ...
```

```
    p->~X();  // 销毁地址 p 中的 X
}
```

语法 p->~X() 对 *p 调用 X 的析构函数。对正常方式销毁的对象（离开其作用域或用 delete 释放）绝不能使用这种语法。

在内存区域中显式管理对象的更完整的例子请见 13.6.1 节。

如果为类 X 声明了一个析构函数，那么每当一个 X 离开作用域或被 delete 释放时，析构函数就会被隐式调用。这意味着我们通过声明 X 的构造函数为 =delete（见 17.6.4 节）或 private，就可以阻止其析构。

在两种方法中，使用 private 更为灵活。例如，我们可以创建一个类，其对象可以显式销毁，但不能隐式销毁：

```
class Nonlocal {
public:
    // ...
    void destroy() { this->~Nonlocal(); }      // 显式析构
private:
    // ...
    ~Nonlocal();                               // 不能隐式析构
};

void user()
{
    Nonlocal x;              // 错误：不能析构一个 Nonlocal
    X* p = new Nonlocal;     // 正确
    // ...
    delete p;                // 错误：不能析构一个 Nonlocal
    p.destroy();             // 正确
}
```

17.2.5　virtual 析构函数

析构函数可以声明为 virtual，而且对于含有虚函数的类通常就应该这么做。例如：

```
class Shape {
public:
    // ...
    virtual void draw() = 0;
    virtual ~Shape();
};

class Circle {
public:
    // ...
    void draw();
    ~Circle();       // 覆盖了 ~Shape()
    // ...
};
```

我们需要一个 virtual 析构函数的原因是，如果通常是通过基类提供的接口来操纵一个对象，那么通常也应通过此接口来 delete 它：

```
void user(Shape* p)
{
    p->draw();    // 调用合适的 draw()
```

```
        // ...
        delete p;       // 调用恰当的析构函数
    };
```

假如 Shape 的析构函数不是 virtual 的，则 delete 在尝试调用恰当的派生类的析构函数（如 ~Circle()）时就会失败。这种失败会导致被释放对象所拥有的资源（如果有的话）泄漏。

17.3　类对象初始化

本节讨论如何初始化一个类的对象，分使用构造函数和不使用构造函数两种情况讨论。本节还会展示如何定义构造函数来接受任意大小的同构初始化器列表（如 {1,2,3} 和 {1,2,3,4,5,6}）。

17.3.1　不使用构造函数进行初始化

我们不能为内置类型定义构造函数，但能用一个恰当类型的值初始化内置类型对象。例如：

```
int a {1};
char* p {nullptr};
```

类似地，我们可以用下列方法初始化一个无构造函数的类的对象

- 逐成员初始化；
- 拷贝初始化；
- 默认初始化（不用初始化器或空初始化列表）。

例如：

```
struct Work {
    string author;
    string name;
    int year;
};

Work s9 { "Beethoven",
          "Symphony No. 9 in D minor, Op. 125; Choral",
          1824
        };                          // 逐成员初始化

Work currently_playing { s9 };      // 拷贝初始化
Work none {};                       // 默认初始化
```

currently_playing 的 3 个成员分别是 s9 的 3 个成员的副本。

使用 {} 进行默认初始化的效果是用 {} 对每个成员进行初始化。因此，none 被初始化为 {{},{},{}}，即 {"","",0}（见 17.3.3 节）。

如果没有声明可接受参数的构造函数，我们也可以完全省去初始化器。例如：

```
Work alpha;

void f()
{
    Work beta;
    // ...
}
```

对于本例，规则不像我们希望的那么清晰。对静态分配的对象（见 6.4.2 节），这种初始化方式与使用 {} 完全一样，因此 alpha 的值是 {""|,"",0}。但是对局部变量和自由存储空间对象，

只对类类型的成员进行默认初始化，内置类型的成员是不进行初始化的，因此 beta 的值是{"","",unknown}。

造成这种复杂性的原因是为了提高极少数关键情况下的性能。例如：

```
struct Buf {
    int count;
    char buf[16*1024];
};
```

可以定义一个 Buf 局部变量，将其作为输入操作的目标，在进行输入操作前并不对它进行初始化。大多数局部变量初始化的性能并不关键，而未初始化局部变量是主要的错误来源之一。如果你希望保证局部变量被初始化或者只是不希望意外结果发生，可以提供一个初始化器，如 {}。例如：

```
Buf buf0;       // 静态分配变量，因此进行默认初始化

void f()
{
    Buf buf1;           // 元素未初始化
    Buf buf2 {};        // 我的编希望将元素清零

    int* p1 = new int;  // *p1 未初始化
    int* p2 = new int{}; // *p2==0
    int* p3 = new int(7); // *p3==7
    // ...
}
```

显然，只有当我们能访问成员时逐成员初始化才奏效。例如：

```
template<class T>
class Checked_pointer { // 控制 T* 的成员的访问
public:
    T& operator*();     // 检查 nullptr 并返回值
// ...
};

Checked_pointer<int> p {new int{7}};   // 错误：不能访问 p.p
```

如果一个类有私有的非 static 数据成员，它就需要一个构造函数来进行初始化。

17.3.2　使用构造函数进行初始化

当逐成员拷贝不能满足需求时，我们可以定义构造函数来初始化对象。特别是，构造函数常用来建立类的不变式并获取必要的资源（见 17.2.1 节）。

如果我们为类声明了构造函数，每个对象就会使用某些构造函数。如果在创建对象时没有提供构造函数所要求的恰当的初始化器，就会导致错误。例如：

```
struct X {
    X(int);
};

X x0;           // 错误：无初始化器
X x1 {};        // 错误：空初始化器
X x2 {2};       // 正确
X x3 {"two"};   // 错误：错误的初始化器类型
X x4 {1,2};     // 错误：初始化器数目不对
X x5 {x4};      // 正确：移动构造函数是隐式定义的（见 17.6 节）
```

注意，当定义了一个接受参数的构造函数后，默认构造函数（见 17.3.3 节）就不存在了；毕竟，X(int) 表明我们需要一个 int 来构造一个 X。但是，拷贝构造函数不会消失（见 17.3.3 节）；假设对象可以被拷贝（在正确构造之后可以拷贝）。在这个假设可能导致问题的情况下（见 3.3.1 节），你可以明确地禁止拷贝（见 17.6.4 节）。

我使用 {} 语法来明确表示正在进行初始化，而不（仅仅）是在赋值、调用函数或是声明函数。只要是在构造对象的地方，我们都可以用 {} 初始化语法为构造函数提供参数。例如：

```
struct Y : X {
    X m {0};                    // 为成员 m 提供默认初始化器
    Y(int a) :X{a}, m{a} { };   // 初始化基类和成员（见 17.4 节）
    Y() : X{0} { };             // 初始化基类和成员
};

X g {1};   // 初始化全局变量

void f(int a)
{
    X def {};                   // 错误：X 没有默认值
    Y de2 {};                   // 正确：使用默认构造函数
    X* p {nullptr};
    X var {2};                  // 初始化局部变量
    p = new X{4};               // 初始化自由空间中的对象
    X a[] {1,2,3};              // 初始化数组元素
    vector<X> v {1,2,3,4};      // 初始化向量元素
}
```

出于这个原因，{} 初始化有时也称为通用（universal）初始化：这种语法可以用在任何地方。而且，{} 初始化还是一致的：无论你在哪里用语法 {v} 将类型 X 的对象初始化为值 v，都会创建相同的值（X{v}）。

与 {} 相反，= 和 () 初始化语法（见 6.3.5 节）不是通用的。例如：

```
struct Y : X {
    X m;
    Y(int a) : X(a), m=a { };   // 错误：不能用 = 进行成员初始化
};

X g(1);   // 初始化全局变量

void f(int a)
{
    X def();                    // 函数返回一个 X（意外吗！？）
    X* p {nullptr};
    X var = 2;                  // 初始化局部变量
    p = new X=4;                // 语法错误：= 不能用于 new
    X a[](1,2,3);               // 错误：不能用 () 进行数组初始化
    vector<X> v(1,2,3,4);       // 错误：不能用 () 初始化列表元素
}
```

= 和 () 初始化语法也不是一致的，但幸运的是这种例子并不显著。如果你坚持使用 = 或 () 初始化语法，就必须记住哪里允许使用以及它们的含义是什么。

构造函数也遵循常规的重载解析规则（见 12.3 节）。例如：

```
struct S {
    S(const char*);
```

```
    S(double*);
};

S s1 {"Napier"};            // S::S(const char*)
S s2 {new double{1.0}};     // S::S(double*);
S s3 {nullptr};             // 二义性：S::S(const char*) 还是 S::S(double*)?
```

注意，{} 初始化器语法不允许窄化转换（见 2.2.2 节）。这是我们更倾向于使用 {} 风格而不是 () 或 = 的另一个原因。

17.3.2.1　用构造函数进行初始化

使用 () 语法，可以请求在初始化过程中使用一个构造函数。即，对一个类，你可以保证用构造函数进行初始化而不会进行 {} 语法也提供的逐成员初始化或初始化器列表初始化（见 17.3.4 节）。例如：

```
struct S1 {
    int a,b;                                    // 无构造函数
};
struct S2 {
    int a,b;
    S2(int a = 0, int b = 0) : a(aa), b(bb) {}          // 构造函数
};

S1 x11(1,2);    // 错误：无构造函数
S1 x12 {1,2};   // 正确：逐成员初始化

S1 x13(1);      // 错误：无构造函数
S1 x14 {1};     // 正确：x14.b 初始化为 0

S2 x21(1,2);    // 正确：使用构造函数
S2 x22 {1,2};   // 正确：使用构造函数

S2 x23(1);      // 正确：使用构造函数和一个默认参数
S2 x24 {1};     // 正确：使用构造函数和一个默认参数
```

{} 初始化的一致使用自 C++11 起才成为现实，旧有 C++ 代码使用 () 和 = 进行初始化。因此，对你来说 () 和 = 可能更熟悉。但是，我想不到有任何合乎逻辑的理由优先选择 () 语法，除非在极少数情况下你需要区分用一个元素列表初始化与用构造函数参数列表初始化。例如：

```
vector<int> v1 {77};        // 用值 77 初始化一个元素
vector<int> v2(77);         // 将 77 个元素均初始化为 0
```

当类型——通常是一个容器——具有一个初始化器列表构造函数（见 17.3.4 节），还有一个接受元素类型参数的“普通构造函数”时，就会产生选择哪种初始化方式的问题。特别是，我们偶尔必须用 () 语法初始化整数或浮点数 vector，但永远不需要用 () 语法初始化字符串或指针 vector：

```
vector<string> v1 {77};         // 77 个元素初始化为默认值 ""
                                // (vector<string>(std::initializer_list<string>) 不接受 {77})
vector<string> v2(77);          // 77 初始化为默认值 ""

vector<string> v3 {"Booh!"};    // 一个元素初始化为 "Booh!"
vector<string> v4("Booh!");     // 错误：没有构造函数接受单一字符串参数

vector<int*> v5 {100,0};        // 100 个 int* 初始化为 nullptr（100 不是一个 int*）
```

```
vector<int*> v6 {0,0};          // 两个 int* 初始化为 nullptr
vector<int*> v7(0,0);           // 空 vector(v7.size()==0)
vector<int*> v8;                // 空 vector (v7.size()==0)
```

对 v6 和 v7 两个例子感兴趣的主要是“语言律师”和编译器测试者。

17.3.3　默认构造函数

无参的构造函数被称为**默认构造函数**（default constructor）。默认构造函数非常常见。例如：

```
class Vector {
public:
    Vector(); // 默认构造函数：无元素
    // ...
};
```

如果构造对象时未指定参数或提供了一个空初始化器列表，则会调用默认构造函数：

```
Vector v1;      // 正确
Vector v2 {};   // 正确
```

如果一个接受参数的构造函数使用了一个默认参数（见 12.2.5 节），它就可能成为一个默认构造函数。例如：

```
class String {
public:
    String(const char* p = "");     // 默认构造函数：空字符串
    // ...
};

String s1;      // 正确
String s2 {};   // 正确
```

标准库 vector 和 string 就有这样的默认构造函数（见 36.3.2 节和 31.3.2 节）。

内置类型被认为具有默认构造函数和拷贝构造函数。但是，对于内置类型的未初始化的非 static 变量（见 17.3 节），其默认构造函数不会被调用。内置整数类型的默认值为 0，浮点类型的默认值为 0.0，指针类型的默认值为 nullptr。例如：

```
void f()
{
    int a0;             // 未初始化
    int a1();           // 函数声明（这是我们的意图吗？）

    int a {};           // a 变为 0
    double d {};        // d 变为 0.0
    char* p {};         // p 变为 nullptr

    int* p1 = new int;  // 未初始化的 int
    int* p2 = new int{}; // int 被初始化为 0
}
```

内置类型的构造函数最常用于模板参数。例如：

```
template<class T>
struct Handle {
    T* p;
    Handle(T* pp = new T{}) :p{pp} { }
```

```
    // ...
};

Handle<int> px;    // 会生成 int()
```

生成的 int 会被初始化为 0。

引用和 const 必须被初始化（见 7.7 节和 7.5 节）。因此，一个包含这些成员的类不能默认构造，除非程序员提供了类内成员初始化器（见 17.4.4 节）或定义了一个默认构造函数来初始化它们（见 17.4.1 节）。例如：

```
int glob {9};

struct X {
    const int a1 {7};      // 正确
    const int a2;          // 错误：需要一个用户自定义构造函数
    const int& r {9};      // 正确
    int& r1 {glob};        // 正确
    int& r2;               // 错误：需要一个用户自定义构造函数
};

X x;      // 错误：X 没有默认构造函数
```

声明数组、标准库 vector 以及类似的容器时可以分配一组默认初始化的元素。在此情况下，被用作 vector 或数组元素类型的类显然需要一个默认构造函数。例如：

```
struct S1 { S1(); };                    // 具有默认构造函数
struct S2 { S2(string); };              // 无默认构造函数

S1 a1[10];                              // 正确：10 个默认元素
S2 a2[10];                              // 错误：不能初始化元素
S2 a3[] { "alpha", "beta" };            // 正确：构造了 2 个元素 S2{"alpha"}, S2{"beta"}

vector<S1> v1(10);                      // 正确：10 个默认元素
vector<S2> v2(10);                      // 错误：不能初始化元素
vector<S2> v3 { "alpha", "beta" };      // 正确：构造了 2 个元素 S2{"alpha"}, S2{"beta"}

vector<S2> v2(10,"");                   // 正确：10 个元素均初始化为 S2{""}
vector<S2> v4;                          // 正确：无元素
```

一个类什么情况下应该具有默认构造函数？一个头脑简单的技术性答案是“当你将它用作数组等的元素类型时。”但是，一个更好的问题是“什么类型有默认值才是有意义的？”或者“此类型是否存在一个我们可以‘自然’用作默认值的‘特殊’值？”字符串有空字符串""，容器有空集 {}，数值有零。我们在确定 Date 的默认值（见 16.3 节）时就有麻烦了，因为不存在“自然”的默认日期（宇宙大爆炸已经太久远而且与我们日常使用的日期并不确切关联）。通常，当尝试创造默认值时不要自作聪明。例如，容器元素没有默认值不是什么大问题，通常最好的解决方式是直到你知道了元素的正确值时再分配它们（如使用 push_back()）。

17.3.4　初始化器列表构造函数

被受单一 std::initializer_list 参数的构造函数被称为初始化器列表构造函数（initializer-list constructor）。一个初始化器列表构造函数使用一个 {} 列表作为其初始化值来构造对象。标准库容器（如 vector 和 map）都有初始化器列表构造函数、初始化器列表赋值运算符等成员（见 31.3.2 节和 31.4.3 节）。考虑下面的代码：

```
vector<double> v = { 1, 2, 3.456, 99.99 };

list<pair<string,string>> languages = {
    {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"}
};

map<vector<string>,vector<int>> years = {
    { {"Maurice","Vincent", "Wilkes"},{1913, 1945, 1951, 1967, 2000} },
    { {"Martin", "Richards"} {1982, 2003, 2007} },
    { {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004} }
};
```

我们想要使用接受一个 {} 列表进行初始化的机制，就要定义一个接受 std::initializer_list<T> 类型参数的函数（通常是一个构造函数）。例如：

```
void f(initializer_list<int>);

f{1,2};
f{23,345,4567,56789};
f{};      // 空列表

f(1,2);   // 错误：缺漏函数调用 ()

years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});
```

一个初始化器列表的长度可以任意，但它必须是同构的。即，所有元素的类型都必须是模板参数 T，或可以隐式转换为 T。

17.3.4.1　initializer_list 构造消除歧义

如果一个类已有多个构造函数，则编译器会使用常规的重载解析规则（见 12.3 节）根据给定参数选择一个正确的构造函数。当选择构造函数时，默认构造函数和初始化器列表构造函数优先。考虑下面的代码：

```
struct X {
    X(initializer_list<int>);
    X();
    X(int);
};

X x0 {};  // 空列表：选择默认构造函数还是初始化器列表构造函数？（默认构造函数）
X x1 {1}; // 一个整数：是一个整型参数还是一个单元素的列表？（初始化器列表构造函数）
```

具体规则如下：

- 如果默认构造函数或初始化器列表构造函数都匹配，优先选择默认构造函数。
- 如果一个初始化器列表构造函数和一个“普通构造函数”都匹配，优先选择列表初始化器构造函数。

第一条规则“优先选择默认构造函数”符合常理：只要可能，就选择最简单的构造函数。而且，如果你定义的初始化器列表构造函数在接受空列表时所做的事情与默认构造函数不同，那么你很可能犯了一个设计错误。

第二条规则“优先选择初始化器列表构造函数”是必要的，可避免依据不同元素数产生不同的解析结果。考虑 std::vector（见 31.4 节）：

```
vector<int> v1 {1};     // 一个元素
vector<int> v2 {1,2};   // 两个元素
vector<int> v3 {1,2,3}; // 三个元素
```

```
vector<string> vs1 {"one"};
vector<string> vs2 {"one", "two"};
vector<string> vs3 {"one", "two", "three"};
```

这段代码中所有初始化操作都使用初始化器列表构造函数。如果我们真的希望调用接受一个或两个整型参数的构造函数，就必须使用 () 语法：

```
vector<int> v1(1);  // 构造一个元素，其值为默认值（0）
vector<int> v2(1,2); // 构造一个值为 2 的元素
```

17.3.4.2　使用 initializer_list

可以将接受一个 initializer_list<T> 参数的函数作为一个序列来访问，即，通过成员函数 begin()、end() 和 size() 访问。例如：

```
void f(initializer_list<int> args)
{
    for (int i = 0; i!=args.size(); ++i)
        cout << args.begin()[i] << "\n";
}
```

不幸的是，initializer_list 不提供下标操作。

initializer_list<T> 是以传值方式传递的。这是重载解析规则所要求的（见 12.3 节），而且不会带来额外开销，因为一个 initializer_list<T> 对象只是一个小句柄（通常是两个字大小），指向一个元素类型为 T 的数组。

上面这个循环等价于：

```
void f(initializer_list<int> args)
{
    for (auto p=args.begin(); p!=args.end(); ++p)
        cout << *p << "\n";
}
```

或是：

```
void f(initializer_list<int> args)
{
    for (auto x : args)
        cout << x << "\n";
}
```

为了显式使用一个 initializer_list，你必须在定义它的地方使用 #include 包含头文件 <initializer_list>。但是，由于 vector、map 等使用 initializer_list，它们的头文件（<vector>、<map> 等）已经 #include 了 <initializer_list>，因此你很少需要直接包含此头文件。

initializer_list 的元素是不可变的，不要考虑修改它们的值，例如：

```
int f(std::initializer_list<int> x, int val)
{
    *x.begin() = val;          // 错误：试图改变初始化器列表元素的值
    return *x.begin();  // 正确
}

void g()
{
    for (int i=0; i!=10; ++i)
        cout << f({1,2,3},i) << '\n';
}
```

假如 f() 中的赋值成功，看起来（{1,2,3} 中）1 的值会改变。这会严重破坏我们某些最基本的概念。由于 initializer_list 元素是不可变的，我们不能对其使用移动构造函数（见 3.3.2 节和 17.5.2 节）。

一个容器可能像下面的代码这样实现初始化器列表构造函数：

```
template<class E>
class Vector {
public:
    Vector(std::initializer_list<E> s); // 初始化器列表构造函数
    // ...
private:
    int sz;
    E* elem;
};

template<class E>
Vector::Vector(std::initializer_list<E> s)
    :sz{s.size()}                                   // 设置 vector 大小
{
    reserve(sz);                                    // 获取足够的内存空间
    uninitialized_copy(s.begin(), s.end(), elem);   // 初始化 elem[0:s.size()) 中的元素
}
```

初始化器列表是通用和一致的初始化设计的一部分（见 17.3 节）。

17.3.4.3　直接和拷贝初始化

{} 初始化也存在直接初始化和拷贝初始化的区别（见 16.2.6. 节）。对一个容器来说，这意味着这种区别对容器自身及其中的元素都有作用：

- 容器的初始化器列表构造函数可以是 explicit，也可以不是。
- 初始化器列表的元素类型的构造函数可以是 explicit，也可以不是。

对一个 vector<vector<double>>，我们可以看到直接初始化与拷贝初始化的区别对元素所起的作用。例如：

```
vector<vector<double>> vs = {
    {10,11,12,13,14},               // 正确：5 个元素的 vector
    {10},                           // 正确：1 个元素的 vector
    10,                             // 错误：vector<double>(int) 是显式的

    vector<double>{10,11,12,13},    // 正确：5 个元素的 vector
    vector<double>{10},             // 正确：1 个元素的 vector，元素值为 10.0
    vector<double>(10),             // 正确：10 个元素的 vector，元素值都为 0.0
};
```

一个容器可以有若干显式的构造函数以及若干非显式的构造函数，标准库 vector 就是一个这样的例子。例如，std::vector<int>(int) 是 explicit，但 std::vector<int>(initialize_list<int>) 不是：

```
vector<double> v1(7);   // 正确：v1 有 7 个元素；注意：使用的是 () 而不是 {}
vector<double> v2 = 9;  // 错误：不能从 int 转换为 vector

void f(const vector<double>&);
void g()
{
    v1 = 9;         // 错误：不能从 int 转换为 vector
    f(9);           // 错误：不能从 int 转换为 vector
}
```

将 () 替换为 {}，我们得到：

```
vector<double> v1 {7};          // 正确：v1 有一个元素（值为 7）
vector<double> v2 = {9};        // 正确：v2 有一个元素（值为 9）

void f(const vector<double>&);
void g()
{
    v1 = {9};           // 正确：v1 现在有一个元素（值为 9）
    f({9});             // 正确：调用 f，将列表 {9} 传递给它
}
```

显然，结果完全不同。

这个例子是精心打造的，以展示最令人迷惑的情况。注意，对更长的列表不会出现明显的歧义（当然，之前的歧义也只是人类眼中的歧义，而不是编译器眼中的）。例如：

```
vector<double> v1 {7,8,9};          // 正确：v1 有 3 个元素，值为 {7,8,9}
vector<double> v2 = {9,8,7};        // 正确：v2 有 3 个元素，值为 {9,8,7}
void f(const vector<double>&);
void g()
{
    v1 = {9,10,11};         // 正确：v1 有 3 个元素，值为 {9,10,11}
    f({9,8,7,6,5,4});       // 正确：调用 f，将列表 {9,8,7,6,5,4} 传递给它
}
```

类似地，对非整数类型元素的列表也不会产生歧义：

```
vector<string> v1 { "Anya"};        // 正确：v1 有一个元素（值为 "Anya"）
vector<string> v2 = {"Courtney"};   // 正确：v2 有一个元素（值为 "Courtney"）

void f(const vector<string>&);
void g()
{
    v1 = {"Gavin"};         // 正确：v1 现在有一个元素（值为 "Gavin"）
    f({"Norah"});           // 正确：调用 f，将列表 {"Norah"} 传递给它
}
```

17.4　成员和基类初始化

构造函数可以建立不变式并获取资源。一般而言，构造函数是通过初始化类成员和基类来完成这些工作的。

17.4.1　成员初始化

考虑下面这个类，它用来保存一个小型组织的信息：

```
class Club {
    string name;
    vector<string> members;
    vector<string> officers;
    Date founded;
    // ...
    Club(const string& n, Date fd);
};
```

Club 的构造函数接受 2 个参数，分别是俱乐部的名字和成立日期。在构造函数的定义中，通过成员初始化器列表（member initialize list）给出成员的构造函数的参数。例如：

```
Club::Club(const string& n, Date fd)
    : name{n}, members{}, officers{}, founded{fd}
{
    // ...
}
```

成员初始化器列表以一个冒号开始，后面的成员初始化器用逗号间隔。

类自身的构造函数在其函数体执行之前会先调用成员的构造函数（见 17.3.2 节）。成员的构造函数按成员在类中声明的顺序调用，而不是按成员在初始化器列表中出现的顺序。为了避免混淆，最好按成员的声明顺序指明初始化器。如果你没有按正确顺序排列初始化器，最好寄希望于编译器给出警告。在类自身的析构函数的函数体执行完毕后，会按相反顺序调用成员的析构函数。

如果一个成员构造函数不需要参数，就不必在成员初始化器列表中提及此成员。例如：

```
Club::Club(const string& n, Date fd)
    : name{n}, founded{fd}
{
    // ...
}
```

此构造函数等价于上一个版本。两个版本都将 Club::officers 和 Club:: members 初始化为空 vector。

显式初始化成员通常是一个好主意。注意，一个“隐式初始化”的内置类型成员其实是未初始化的（见 17.3.1 节）。

一个构造函数可以初始化其类的成员和基类，但不会初始化其成员或基类的成员或基类。例如：

```
struct B { B(int); /* ... */};
struct BB : B { /* ... */ };
struct BBB : BB {
    BBB(int i) : B(i) { };  // 错误：尝试初始化基类的基类
    // ...
};
```

17.4.1.1　成员初始化和赋值

如果对一个类型而言，初始化的含义与赋值不同，那么对其使用成员初始化器就是必要的。例如：

```
class X {
    const int i;
    Club cl;
    Club& rc;
    // ...
    X(int ii, const string& n, Date d, Club& c) : i(ii), cl(n,d), rc(c) { }
};
```

引用成员或 const 成员必须初始化（见 7.5 节、7.7 节和 17.3.3 节）。但是，对大多数类型，程序员可以选择使用初始化器还是使用赋值。对此，我通常倾向于使用成员初始化器语法，这能明确表示我正在进行初始化操作。使用初始化器语法（与使用赋值相比）通常还有性能上的优势。例如：

```
class Person {
    string name;
    string address;
```

```
    // ...
    Person(const Person&);
    Person(const string& n, const string& a);
};

Person::Person(const string& n, const string& a)
    : name{n}
{
    address = a;
}
```

本例中用 n 的一个副本来初始化 name。另一方面，address 首先被初始化为空字符串，然后被赋值为 a 的副本。

17.4.2　基类初始化器

派生类的基类的初始化方式与非数据成员相同。即，如果基类要求一个初始化器，我们就必须在构造函数中提供相应的基类初始化器。如果我们希望进行默认构造，可以显式指出。例如：

```
class B1 { B1(); };   // 具有默认构造函数
class B2 { B2(int); } // 无默认构造函数

struct D1 : B1, B2 {
    D1(int i) :B1{}, B2{i} {}
};

struct D2 : B1, B2 {
    D2(int i) :B2{i} {}          // 隐式使用 B1()
};

struct D1 : B1, B2 {
    D1(int i) { }                // 错误：B2 要求一个 int 初始化器
};
```

与成员初始化类似，基类按声明顺序进行初始化，建议按此顺序指定基类的初始化器。基类的初始化在成员之前，销毁在成员之后（见 17.2.3 节）。

17.4.3　委托构造函数

如果你希望两个构造函数做相同的操作，可以重复代码，也可以定义一个“init() 函数”来执行两者相同的操作。两种“解决方案”都很常见（因为旧版 C++ 没有提供其他更好的方法）。例如：

```
class X {
    int a;
    validate(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
public:
    X(int x) { validate(x); }
    X() { validate(42); }
    X(string s) { int x = to<int>(s); validate(x); }    // 见 25.2.5.1 节
    // ...
};
```

冗长的代码会影响可读性，而重复代码则很容易出错，两者都妨碍了可维护性。一种替代方法是用一个构造函数定义另一个：

```
class X {
        int a;
public:
        X(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
        X() :X{42} { }
        X(string s) :X{to<int>(s)} { }          // 见 25.2.5.1 节
        // ...
};
```

即，使用一个成员风格的初始化器，但用的是类自身的名字（也是构造函数名），它会调用另一个构造函数，作为这个构造过程的一部分。这样的构造函数称为委托构造函数（delegating constructor，有时也称为转发构造函数，forwarding constructor）。

你不能同时显式和委托初始化一个成员。例如：

```
class X {
        int a;
public:
        X(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
        X() :X{42}, a{56} { }       // 错误
        // ...
};
```

在一个构造函数的成员和基类初始化器列表中调用其他构造函数来实现委托初始化，与在构造函数体中显式调用其他构造函数有着很大不同。考虑下面的代码：

```
class X {
        int a;
public:
        X(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
        X() { X{42}; }    // 很可能是错误的
        // ...
};
```

X{42} 简单创建一个新的无名（临时）对象，对它不做任何处理。这种用法多半是错误的，希望编译器能对此给出警告。

直到构造函数执行完毕，对象才被认为完成构造（见 6.4.2 节）。当使用委托构造函数时，委托者执行完毕才表明构造完成。仅仅被委托者执行完毕是不够的。析构函数在构造函数执行完毕后才可能会被调用。

如果你所要做的只是将成员设置为默认值（不依赖于构造函数参数），使用成员初始化器（见 17.4.4 节）可能更为简单。

17.4.4　类内初始化器

我们可以在类声明中为非 static 数据成员指定初始化器。例如：

```
class A {
public:
        int a {7};
        int b = 77;
};
```

出于语法分析和名字查找相关的很隐蔽的技术原因，{} 和 = 语法能用于类内成员初始化器，但 () 语法就不行。

默认情况下，构造函数会使用这种类内初始化器，因此上例等价于下面的这个版本：

```
class A {
public:
    int a;
    int b;
    A() : a{7}, b{77} {}
};
```

类内初始化器的这种用法可以节省一些输入工作量，但真正的收益是在用于具有多个构造函数的更复杂的类时。对同一个成员，多个构造函数通常使用相同的初始化器。例如：

```
class A {
public:
    A() :a{7}, b{5}, algorithm{"MD5"}, state{"Constructor run"} {}
    A(int a_val) :a{a_val}, b{5}, algorithm{"MD5"}, state{"Constructor run"} {}
    A(D d) :a{7}, b{g(d)}, algorithm{"MD5"}, state{"Constructor run"} {}
    // ...
private:
    int a, b;
    HashFunction algorithm;      // 加密哈希，用于所有 A
    string state;                // 字符串，在对象生命周期中指示其状态
};
```

实际上 algorithm 和 state 在所有构造函数中都被初始化为相同的值，但这一点完全隐藏在混乱的代码中，很容易成为代码维护时的一个隐患。为了明确表示相同的初始化值，我们可以为数据成员提炼出唯一的初始化器：

```
class A {
public:
    A() :a{7}, b{5} {}
    A(int a_val) :a{a_val}, b{5} {}
    A(D d) :a{7}, b{g(d)} {}
    // ...
private:
    int a, b;
    HashFunction algorithm {"MD5"};      // 加密哈希，用于所有 A
    string state {"Constructor run"};    // 字符串，在对象生命周期中指示其状态
};
```

如果一个成员既被类内初始化器初始化，又被构造函数初始化，则只执行后者的初始化操作（它“覆盖了”默认值）。因此我们可以进一步简化代码：

```
class A {
public:
    A() {}
    A(int a_val) :a{a_val} {}
    A(D d) :b{g(d)} {}
    // ...
private:
    int a {7};                           // a 的值为 7 表示 …
    int b {5};                           // b 的值为 5 表示 …
    HashFunction algorithm {"MD5"};      // 加密哈希，用于所有 A
    string state {"Constructor run"};    // 字符串，在对象生命周期中指示其状态
};
```

如这段代码所示，默认类内初始化器提供了表明共同初始化处理的机会。

一个类内初始化器可以使用它的位置（在成员声明中）所在作用域中的所有名字。考虑下面这个令人头痛的技术示例：

```
int count = 0;
int count2 = 0;

int f(int i) { return i+count; }

struct S {
    int m1 {count2};    // 即 ::count2
    int m2 {f(m1)};     // 即 this->m1+::count，也就是 ::count2+::count
    S() { ++count2; }   // 非常奇怪的构造函数
};

int main()
{
    S s1;    // {0,0}
    ++count;
    S s2;    // {1,2}
}
```

成员初始化是按成员声明的顺序进行的（见 17.2.3 节），因此 m1 首先被初始化为全局变量 count2。全局变量的值在新 S 对象的构造函数运行时获得，因此它可以改变（本例中确实改变了）。接下来，通过调用全局函数 f() 初始化 m2。

像这样将对全局数据的微妙依赖隐藏于成员初始化器中是一个糟糕的主意。

17.4.5　static 成员初始化

一个 static 类成员是静态分配的，而不是每个类对象的一部分。一般来说，static 成员声明充当类外定义的声明。例如：

```
class Node {
    // ...
    static int node_count;          // 声明
};

int Node::node_count = 0;           // 定义
```

但是，在少数简单的特殊情况下，在类内声明中初始化 static 成员也是可能的。条件是 static 成员必须是整型或枚举类型的 const，或字面值类型的 constexpr（见 10.4.3 节），且初始化器必须是一个常量表达式（constant-expression）。例如：

```
class Curious {
public:
    static const int c1 = 7;            // 正确
    static int c2 = 11;                 // 错误：非 const
    const int c3 = 13;                  // 正确，但非 static（见 17.4.4 节）
    static const int c4 = sqrt(9);      // 错误：类内初始化器不是常量
    static const float c5 = 7.0;        // 错误：类内初始化成员不是整型（应使用 constexpr 而非 const）
    // ...
};
```

当（且仅当）你使用一个已初始化成员的方式要求它像对象一样在内存中存储时，该成员必须在某处（唯一）定义。初始化器不能重复：

```
const int Curious::c1;              // 不重复初始化器
const int* p = &Curious::c1;        // 正确：Curious::c1 已被定义
```

成员常量的主要用途是为类声明中其他地方用到的常量提供符号名称。例如：

```
template<class T, int N>
class Fixed {    // 固定大小数组
public:
    static constexpr int max = N;
    // ...
private:
    T a[max];
};
```

对于整数，枚举值（见 8.4 节）提供了另一种在类声明中定义符号常量的方法。例如：

```
class X {
    enum { c1 = 7, c2 = 11, c3 = 13, c4 = 17 };
    // ...
};
```

17.5　拷贝和移动

当我们需要从 a 到 b 传输一个值的时候，通常有两种逻辑上不同的方法：

- 拷贝（copy）是 x=y 的常规含义；即，结果是 x 和 y 的值都等于赋值前 y 的值。
- 移动（move）将 x 变为 y 的旧值，y 变为某种移出状态（moved-from state）。对我们最感兴趣的情况——容器，移出状态就是“空”。

这种逻辑上的简单区别令人困惑，这一方面是由传统习惯造成的，另一方面是由于我们对移动和拷贝使用相同符号表示而造成的。

一般来说，移动操作不能抛出异常，而拷贝操作则可以（因为拷贝可能需要获取资源），移动操作通常比拷贝操作更高效。当编写一个移动操作时，应该将源对象置于一个合法的但未指明的状态，因为它最终会被销毁，而析构函数不能销毁一个处于非法状态的对象。而且，标准库算法要求能够向一个移出状态的对象进行赋值（使用移动或拷贝）。因此，设计移动操作时不要让它抛出异常，并令源对象处于可析构和赋值的状态。

为了避免乏味的重复性工作，拷贝和移动操作都有默认定义（见 17.6.2 节）。

17.5.1　拷贝

类 X 的拷贝操作有两种：

- 拷贝构造函数：X(const X&)
- 拷贝赋值运算符：X& operator=(const X&)

你可以定义这两个操作接受一些更冒险的参数类型，例如 volatile X&，但不要这么做，这样做只会令你自己和其他人困惑。一个拷贝构造函数应该创建给定对象的副本，而不应修改它。类似地，你可以将 const X& 用作拷贝赋值运算符的返回类型。我的观点是这么做会引起更大的困惑，并不值得，因此我在讨论拷贝操作时假定两个操作都具有常规类型。

考虑下面这个简单二维 Matrix 的代码：

```
template<class T>
class Matrix {
    array<int,2> dim;    // 二维
    T* elem;             // 指向 dim[0]*dim[1] 个类型为 T 的元素
public:
    Matrix(int d1, int d2) :dim{d1,d2}, elem{new T[d1*d2]} {}    // 简化了（无错误处理）
    int size() const { return dim[0]*dim[1]; }
```

```
    Matrix(const Matrix&);              // 拷贝构造函数
    Matrix& operator=(const Matrix&); // 拷贝赋值运算符

    Matrix(Matrix&&);                   // 移动构造函数
    Matrix& operator=(Matrix&&);        // 移动赋值运算符

    ~Matrix() { delete[] elem; }
    // ...
};
```

首先我们注意到默认拷贝（拷贝成员）可能带来灾难性的后果：Matrix 的元素不会被复制，Matrix 的副本只是包含一个指向与源对象相同元素的指针，因而 Matrix 的析构函数会试图释放（共享）元素两次（见 3.3.1 节）。

但是，程序员可以为这些拷贝操作定义任意恰当的含义，对于容器来说常规语义是拷贝容器内的元素：

```
template<class T>
Matrix:: Matrix(const Matrix& m)        // 拷贝构造函数
    : dim{m.dim},
      elem{new T[m.size()]}
{
    uninitialized_copy(m.elem,m.elem+m.size(),elem);      // 拷贝元素
}

template<class T>
Matrix& Matrix::operator=(const Matrix& m)        // 拷贝赋值运算符
{
    if (dim[0]!=m.dim[0] || dim[1]!=m.dim[1])
        throw runtime_error("bad size in Matrix =");
    copy(m.elem,m.elem+m.size(),elem);     // 拷贝元素
}
```

拷贝构造函数与拷贝赋值运算符的区别在于前者初始化一片未初始化的内存，而后者必须正确处理目标对象已构造并可能拥有资源的情况。

Matrix 的拷贝赋值运算符具有一个特性：如果拷贝某个元素时抛出了异常，则赋值的目标对象会变为旧值和新值混合的状态。即，Matrix 的赋值操作提供了基本保障，但未提供强保障（见 13.2 节）。如果我们认为这不可接受，可以用一种基本技术避免该结果——首先创建一个副本，然后交换内容：

```
Matrix& Matrix::operator=(const Matrix& m)        // 拷贝赋值运算符
{
    Matrix tmp {m};          // 创建一个副本
    swap(tmp,*this);         // 交换 tmp 与 *this 的内容
    return *this;
}
```

只有当拷贝成功时才会执行 swap()。显然，这个版本的 operator=() 只有当 swap() 的实现未使用赋值（std::swap() 确实未使用）时才能奏效，参见 17.5.2 节。

一个拷贝构造函数通常需要拷贝每个非 static 成员（见 17.4.1 节）。如果拷贝构造函数无法拷贝一个元素（例如，它需要获取不可用的资源才能完成拷贝），它就会抛出一个异常。

注意，我并未防止使用 Matrix 的拷贝赋值运算符进行自赋值，即 m=m。我没有检测自赋值的原因是成员的自赋值已经是安全的了：对于 m=m，我实现的 Matrix 的拷贝赋值运算

符既正确又高效。而且，自赋值很少见，因此只有当你的确需要时再检测自赋值。

17.5.1.1　小心默认构造函数

当编写一个拷贝操作时，应确保拷贝了每个基类和成员。考虑下面的代码：

```
class X {
    string s;
    string s2;
    vector<string> v;

    X(const X&)          // 拷贝构造函数
        :s{a.s}, v{a.v}  // 可能是正确的结果，而且可能是错误的
    {
    }
    // ...
};
```

在本例中，我“忘了”拷贝 s2，因此它会得到默认初始值（""）。这很可能与人们的预期并不相符。尽管我们不太可能在这么一个简单的类中犯错误，但是对于更大的类，忘记的可能性会上升。更糟的是，当某人在初始设计很久之后向类中添加一个成员时，很容易忘记将它添加到成员拷贝列表中。这也是我们更倾向于使用默认（编译器生成的）拷贝操作的原因之一（见 17.6 节）。

17.5.1.2　拷贝基类

从拷贝的目的来看，一个基类就是一个成员；为了拷贝派生类的一个对象，你必须拷贝其基类。例如：

```
struct B1 {
    B1();
    B1(const B1&);
    // ...
};

struct B2 {
    B2(int);
    B2(const B2&);
    // ...
};

struct D : B1, B2 {
    D(int i) :B1{}, B2{i}, m1{}, m2{2*i} {}
    D(const D& a) :B1{a}, B2{a}, m1{a.m1}, m2{a.m2} {}
    B1 m1;
    B2 m2;
};

D d {1};   // 对 int 参数构造
D dd {d}; // 拷贝构造
```

初始化顺序还是通常的顺序（基类在成员之前），但对于拷贝而言，最好是顺序对结果没有影响。

一个 virtual 基类（见 21.3.5 节）在类层次中可能作为多个类的基类。默认拷贝构造函数（见 17.6 节）能正确拷贝它。如果你定义自己的拷贝构造函数，最简单的技术是重复拷贝 virtual 基类。对于基类对象很小且 virtual 基类在类层次中只出现几次的情况来说，重复拷贝 virtual 基类的做法比试图避免重复拷贝更高效。

17.5.1.3 拷贝的含义

一个拷贝构造函数或拷贝赋值运算符应该怎么做才会被认为是“一次正确的拷贝操作”呢？除了必须声明为正确类型之外，拷贝操作还必须具有正确的拷贝语义。考虑两个相同类型对象的一次拷贝操作 x=y。为了能适合于一般的面向值的程序设计（见 16.3.4 节），以及能适合于与标准库配合使用的特殊情况（见 31.2.2 节），拷贝操作必须满足两个准则：

- 等价性：在 x=y 之后，对 x 和 y 执行相同的操作应该得到相同的结果。特别是，如果它们的类型定义了 ==，应该有 x==y，并且对任何只依赖于 x 和 y 的值的函数 f()（与依赖于 x 和 y 的地址的函数不同）有 f(x)==f(y)。
- 独立性：在 x=y 之后，对 x 的操作不会隐式地改变 y 的状态。即，只要 f(x) 未引用 y，它就不会改变 y 的值。

这是 int 和 vector 都能提供的保证。如果拷贝操作能提供等价性和独立性，代码就会更简单也更易维护。这一点是值得强调的，因为违反这些简单规则的代码并不罕见，而且程序员并不总是能意识到违反这些规则是他们所遇到的一些糟糕问题的根源。提供等价性和独立性的拷贝操作是正规类型概念（见 24.3.1 节）的一部分。

我们首先考虑等价性要求。人们很少故意违反这一要求，而且默认拷贝操作不会违反这一要求，因为默认拷贝操作执行逐成员拷贝（见 17.3.1 节和 17.6.2 节）。但是，人们偶尔会使用一些小花招，例如，令拷贝的含义依赖于不同“选项”，这通常会导致混乱。而且，对象包含的成员不被看作对象值的一部分的情况并不罕见。例如，拷贝一个标准容器时不拷贝其分配器，因为分配器被认为是容器的一部分，但不是容器值的一部分。类似地，用于统计收集和缓存值的计数器有时也不是简单拷贝的。特别是，x=y 应该蕴含 x==y。而且，切片（见 17.5.1.4 节）可能导致行为不同的“副本”，通常是一个糟糕的错误。

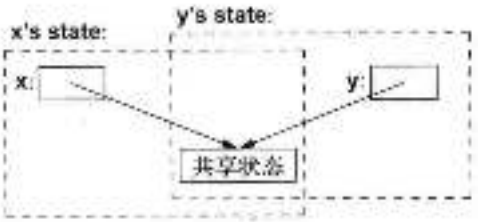
现在我们考虑独立性的要求。大多数与独立性相关（没有独立性）的问题都涉及包含指针的对象。默认拷贝语义是逐成员拷贝，因此一个默认拷贝操作会拷贝指针成员，但不会拷贝指针指向的对象（如果有的话）。例如：

```
struct S {
    int* p;    // 一个指针
};

S x {new int(0)};
void f()
{
    S y {x};              // “拷贝” x

    *y.p = 1;             // 改变 y：影响了 x
    *x.p = 2;             // 改变 x：影响了 y
    delete y.p;           // 影响了 x 和 y
    y.p = new int{3};     // 正确的：改变 y；未影响 x
    *x.p = 4;             // 糟糕：写入已释放的内存
}
```

在本例中我违反了独立性原则。在将 x“拷贝”到 y 后，我们可以通过 y 操纵 x 的部分状态。这有时被称为浅拷贝（shallow copy），人们经常（过度）赞扬浅拷贝在“效率”方面的优势。一个明显的替代方法是拷贝对象的完整状态，这被称为深拷贝（deep copy）。通常，比深拷贝更好的替代方法不是浅拷贝，而是移动操作，它能最小化拷贝量而又不会增加复杂性（见 3.3.2 节和 17.5.2 节）。

一次浅拷贝会令两个对象（本例中的x和y）进入共享状态（shared state），这会带来严重的潜在混乱和错误。如果违反了独立性要求，我们称对象x和y纠缠在一起（entangled）。孤立地分析一个对象是否纠缠是不可能的。例如，从源码中很难分析出对*x.p的两次赋值的效果有什么显著不同。

我们可以图示纠缠对象如下：



注意，很多情况下都能引发纠缠。通常直到问题爆发时才能发现发生了纠缠现象。例如，我们可能不小心将S这样的类型用作另一个行为良好的类的成员。S的原作者可能知道纠缠问题并且准备处理它，但某些人可能会天真地认为拷贝S就意味着拷贝它的完整值，这些人就会对结果感到惊讶，而发现S深层嵌套在其他类中的人也会非常惊讶。

对于与共享子对象生命周期相关的问题，我们可以通过引入某种形式的垃圾收集机制来解决。例如：

```
struct S2 {
    shared_ptr<int> p;
};

S2 x {new int{0}};
void f()
{
    S2 y {x};                   // "拷贝" x

    *y.p = 1;                   // 改变 y；影响了 x
    ^x.p = 2;                   // 改变 x；影响了 y
    y.p.reset(new int{3});      // 改变 y；影响了 x
    ~x.p = 4;                   // 改变 x；影响了 y
}
```

实际上，浅拷贝和这种对象纠缠都是导致需要垃圾收集的原因。如果没有某种形式的垃圾收集（如shared_ptr），对象纠缠会导致代码非常难以管理。

但是，shared_ptr仍然是指针，因此我们不能孤立考虑包含shared_ptr的对象。谁负责更新指向的对象？如何更新？何时更新？如果我们正运行在一个多线程系统中，需要利用同步机制进行共享数据的访问吗？我们如何确认？对象纠缠（在本节中是浅拷贝引起的）是复杂性和错误之源，最好的情况也只能通过垃圾收集（任何形式）部分解决。

注意，不可变的共享状态不是问题。除非我们比较地址，否则不可能判断两个相等的值是一份还是两份副本。这个观察结果是很有用的，因为很多副本永远也不会被修改。例如，我们极少修改以传值方式传递来的对象。这个观察结果还催生了写前拷贝（copy-on-write）的概念。其思想是在共享状态被修改之前，副本其实并不真的需要独立性，因此我们可以推迟共享状态的拷贝，直至首次修改副本前才真正进行拷贝。考虑下面的代码：

```
class Image {
```

```
public:
    // ...
    Image(const Image& a);        // 拷贝构造函数
    // ...
    void write_block(Descriptor);
    // ...
private:
    Representation* clone();      // 拷贝 *rep
    Representation* rep;
    bool shared;
};
```

假定一个 Representation 可能很大，因此与检测 bool 值相比，write_block() 会很耗时。这样，依赖于 Image 的使用，将拷贝构造函数实现为浅拷贝就很有意义了：

```
Image::Image(const Image& a)      // 进行浅拷贝并准备进行写前拷贝
    :rep{a.rep},
     shared{true}
{
}
```

为保护传递给拷贝构造函数的参数，我们在修改它之前拷贝 Representation：

```
void write_block(Descriptor d)
{
    if (shared) {
        rep =clone();         // 拷贝 *rep
        shared = false;       // 不再共享
    }
    // ... 现在我们可以安全地修改我们自己的 rep 副本 ...
}
```

类似任何其他技术，写前拷贝也不是万能灵药，但它能有效地结合真拷贝的简单性和浅拷贝的性能。

17.5.1.4 切片

一个指向派生类的指针可隐式转换为指向其公有基类的指针。当这一简单且必要的规则（见 3.2.4 节和 20.2 节）应用于拷贝操作时，就会导致一个容易让人中招的陷阱。考虑下面的代码：

```
struct Base {
    int b;
    Base(const Base&);
    // ...
};

struct Derived : Base {
    int d;
    Derived(const Derived&);
    // ...
};

void naive(Base* p)
{
    B b2 = *p;        // 可能切片：调用 Base::Base(const Base&)
    // ...
}

void user()
```

```
{
    Derived d;
    naive(&d);
    Base bb = d;   // 切片：调用 Base::Base(const Base&) 而不是 Derived::Derived(const Derived&)
    // ...
}
```

变量 b2 和 bb 包含 d 的 Base 部分的副本，即，d.b 的副本。成员 d.d 不会被拷贝。这种现象称为切片（slicing）。这可能正是你所期望的（例如，参见 17.5.1.2 节中 D 的拷贝构造函数，我们将选出的一些信息传递给一个基类），但这通常是一个微妙的错误。如果你不希望切片，可以采用以下方法防止这种现象：

[1] 禁止拷贝基类：delete 拷贝操作（见 17.6.4 节）。

[2] 防止派生类指针转换为基类指针：将基类声明为 private 或 protected 基类（见 20.5 节）。

方法[1]会令 b2 和 bb 的初始化出现错误；方法[2]会令 naive() 调用和 bb 的初始化出现错误。

17.5.2 移动

将 a 的值给 b 的传统方法是拷贝。对于计算机内存中的一个整数，这几乎是唯一合理的方式：硬件用单一指令即可完成整数的拷贝。但是，从通用和逻辑的角度考虑就不是这样了。考虑下面 swap() 的明显的实现，它完成两个对象值的交换：

```
template<class T>
void swap(T& a, T& b)
{
    const T tmp = a;    // 将 a 的副本放入 tmp
    a = b;              // 将 b 的副本放入 a
    b = tmp;            // 将 tmp 的副本放入 b
};
```

初始化 tmp 之后，我们就拥有了 a 的值的两个副本。为 tmp 赋值后，我们就有了 b 的值的两个副本。为 b 赋值后，我们有了 tmp 的值（即 a 的原值）的两个副本。然后我们销毁了 tmp。这听起来像是做了很多工作，而实际上确实可能需要做很多工作。例如：

```
void f(string& s1, string& s2,
    vector<string>& vs1, vector<string>& vs2,
    Matrix& m1, Matrix& m2)
{
    swap(s1,s2);
    swap(vs1,vs2);
    swap(m1,m2);
}
```

如果 s1 有一千个字符会怎样？如果 vs2 有一千个元素，每个元素有一千个字符会怎样？如果 m1 是一个 1000*1000 的 double 矩阵会怎样？拷贝这些数据结构的代价会非常高。实际上，标准库 swap() 已经过精心设计，对 string 和 vector 能避免这种额外开销。即，标准库的设计者已经努力避免了拷贝（利用了一个事实：string 和 vector 对象实际只是指向其元素的句柄）。如果我们希望 Matrix 的 swap() 也避免严重的性能问题，就要做类似的工作。如果我们只有拷贝操作，就必须要对大量的非标准函数和数据结构做类似的工作。

根本问题是我们其实不希望做任何拷贝；我们只是希望交换一对值。

我们也可以从一个完全不同的角度来考虑拷贝问题：我们通常不会拷贝现实物体，除非绝对需要。如果你希望借我的手机，我会将手机交给你而不是为你创建一个专有副本。如果我将汽车借给你，就会给你车钥匙，然后你开走我的车，而不是复制我的汽车。一旦我给你了某样东西，你就拥有了它而我就不再拥有它了。因此，对现实物体我们会说“送出”“移交”“转移所有权”以及“移动”。计算机中的很多对象更像现实物体（如无必要我们不会进行拷贝，而且只有代价合理才拷贝）而不是整数（我们通常对它进行拷贝，因为比其他方法更简单高效）。这方面的例子有锁、套接字、文件句柄、线程、长字符串以及大向量。

为了允许用户避免拷贝的逻辑和性能问题，C++ 不仅支持拷贝（copy）的概念，也直接支持移动（move）的概念。特别是，我们可以定义移动构造函数（move constructor）和移动赋值操作（move assignment）来移动而非拷贝它们的参数。再次考虑来自 17.5.1 节的简单二维 Matrix：

```
template<class T>
class Matrix {
    std::array<int,2> dim;
    T* elem;  // 指向 sz 个类型为 T 的元素

    Matrix(int d1, int d2) :dim{d1,d2}, elem{new T[d1*d2]} {}
    int size() const { return dim[0]*dim[1]; }

    Matrix(const Matrix&);              // 拷贝构造函数
    Matrix(Matrix&&);                   // 移动构造函数

    Matrix& operator=(const Matrix&); // 拷贝赋值运算符
    Matrix& operator=(Matrix&&);        // 移动赋值运算符

    ~Matrix(); // 析构函数
    // ...
};
```

&& 表示右值引用（见 7.7.2 节）。

移动赋值背后的思想是将左值的处理与右值的处理分离：拷贝赋值操作和拷贝构造函数接受左值，而移动赋值操作和移动构造函数则接受右值。对于 return 值，采用移动构造函数。

我们可以为 Matrix 定义移动构造函数，简单地接受其源对象的表示，并将源对象设置为空 Matrix（销毁代价低）。例如：

```
template<class T>
Matrix<T>::Matrix(Matrix&& a)       // 移动构造函数
    :dim{a.dim}, elem{a.elem}       // 窃取 a 的表示
{
    a.dim = {0,0};                  // 清空 a 的表示
    a.elem = nullptr;
}
```

对于移动赋值操作，我们可以简单地进行一次交换。这种实现方式背后的思想是，源对象即将被销毁，因此我们可以让源对象的析构函数为我们做必要的清理工作：

```
template<class T>
Matrix<T>& Matrix<T>::operator=(Matrix&& a)          // 移动赋值
{
    swap(dim,a.dim);                // 交换两者的表示
    swap(elem,a.elem);
    return *this;
}
```

移动构造函数和移动赋值运算符接受非 const（右值）引用参数；它们可以修改参数，而且通常也确实会这么做，写入实参。但是，移动操作必须总是将其参数置于一种析构函数可处理的状态（而且最好是处理起来很容易、代价很低的状态）。

对于资源句柄，移动操作会比拷贝操作简单、高效得多。特别是，移动操作通常不会抛出异常；它们不获取资源或是执行复杂操作，因此不需要抛出异常。在这一点上，它们与很多拷贝操作是不同的（见 17.5 节）。

编译器如何知道它什么时候可以使用移动操作而不是拷贝操作呢？在少数情况下，例如对返回值，语言规则指出编译器可以使用移动操作（因为下一个操作就是销毁元素）。但是，一般情况下我们必须通过传递右值引用参数告知编译器。例如：

```
template<class T>
void swap(T& a, T& b)    // (几乎是)"完美的 swap"
{
    T tmp = std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}
```

move() 是一个标准库函数，它返回其实参的一个右值引用（见 35.5.1 节）；move(x) 意味着"给我一个 x 的右值引用"。即，std::move(x) 本身不移动任何东西；它只是允许用户移动 x。可能将 move() 改名为 rval() 更好，但人们用名字 move() 来表示这个操作已经有很多年了。

标准库容器都具有移动操作（见 3.3.2 节和 35.5.1 节），其他标准库类型如 pair（见 5.4.3 节和 34.2.4.1 节）和 unique_ptr（见 5.2.1 节和 34.3.1 节）也有移动操作。而且，向标准库容器插入新元素的操作，如 insert() 和 push_back()，也有接受右值引用的版本（见 7.7.2 节）。最终的效果就是标准库容器和算法提供了比使用拷贝操作的版本更好的性能。

如果我们要交换的对象的类型没有移动构造函数怎么办？我们只能进行拷贝操作并付出相应的代价。一般而言，避免过多拷贝是程序员的责任。界定"什么过多""哪些必要"不是编译器的任务。为了将你自己使用拷贝的数据结构优化为使用移动，你就必须提供移动操作（无论是显式还是隐式，见 17.6 节）。

内置类型，如 int 和 double*，被认为具有移动操作，其实就是简单的拷贝。一如以往，你必须小心包含指针的数据结构（见 3.3.1 节）。特别是，不要假定一个移出状态的指针一定被设置为 nullptr 了。

移动操作影响从函数返回大对象的习惯处理方式。考虑下面的代码：

```
Matrix operator+(const Matrix& a, const Matrix& b)
    // 对每个 i 和 j，res[i][j] = a[i][j]+b[i][j]
{
    if (a.dim[0]!=b.dim[0] || a.dim[1]!=b.dim[1])
        throw std::runtime_error("unequal Matrix sizes in +");

    Matrix res{a.dim[0],a.dim[1]};
    constexpr auto n = a.size();
    for (int i = 0; i!=n; ++i)
        res.elem[i] = a.elem[i]+b.elem[i];
    return res;
}
```

Matrix 有一个移动构造函数，因此"传值方式返回结果"会变得简单高效，同时还能保持"自然"的形式。如果没有移动操作，就会产生性能问题，就必须寻求变通方法。我们可以

考虑下面的代码：

```
Matrix& operator+(const Matrix& a, const Matrix& b)    // 小心
{
    Matrix& res = *new Matrix;    // 分配在自由空间中
    // 对每个 i 和 j，res[i][j] = a[i][j]+b[i][j]
    return res;
}
```

在 operator+() 中使用 new 并不值得推荐，它会强制 + 的用户处理麻烦的内存管理问题：

- 如何 delete 掉 new 创建的对象？
- 我们需要一个垃圾收集机制吗？
- 我们应该使用一个 Matrix 池而非通用的 new 吗？
- 我们需要在 Matrix 的表示中增加使用计数吗？
- 我们应该重新设计 Matrix 加法的接口吗？
- operator+() 的调用者必须记得 delete 结果吗？
- 如果计算过程抛出一个异常，新分配的内存会发生什么？

显然，任何候选的解决方案都不优雅也不通用。

17.6　生成默认操作

编写拷贝操作、析构函数这样的常规操作会很乏味也容易出错，因此需要时编译器可为我们生成这些操作。默认情况下，编译器会为一个类生成：

- 一个默认构造函数：X()
- 一个拷贝构造函数：X(const X&)
- 一个拷贝赋值运算符：X& operator=(const X&)
- 一个移动构造函数：X(X&&)
- 一个移动赋值运算符：X& operator=(X&&)
- 一个析构函数：~X()

默认情况下，如果程序需要用到这些操作，编译器就会为我们生成默认的版本。但是，如果程序员选择自己掌控，定义了其中一个或多个操作，那么对应的操作就不会自动生成了：

- 如果程序员为一个类声明了任意构造函数，那么编译器就不会为该类生成默认构造函数。
- 如果程序员为一个类声明了拷贝操作、移动操作或析构函数，则编译器不会为该类生成拷贝操作、移动操作或析构函数。

不幸的是，第二条规则不是完整施行的：出于向后兼容性的需求，即使程序员已经定义了析构函数，编译器还是会自动生成拷贝构造函数和拷贝赋值运算符。但是，这一特性在 ISO 标准中已经弃用了（§ iso.D），你可以期望一个现代编译器能对此给出警告。

如需要，我们可以显式指出希望编译器生成哪些函数（见 17.6.1 节）以及不希望生成哪些函数（见 17.6.4 节）。

17.6.1　显式声明默认操作

由于默认操作的自动生成可以被禁止，因此一定有一种恢复默认操作的方法。而且，一些人更喜欢在程序文本中看到完整的操作列表，即使这个列表并无必要。例如，我们可以编

写下面的代码：

```
class gslice {
        valarray<size_t>  size;
        valarray<size_t> stride;
        valarray<size_t> d1;
public:
        gslice() = default;
        ~gslice() = default;
        gslice(const gslice&) = default;
        gslice(gslice&&) = default;
        gslice& operator=(const gslice&) = default;
        gslice& operator=(gslice&&) = default;
        // ...
};
```

std::gslice 的这个实现片段（见 40.5.6 节）等价于：

```
class gslice {
        valarray<size_t> size;
        valarray<size_t> stride;
        valarray<size_t> d1;
public:
        // ...
};
```

我更喜欢采用后一种方式，但我能理解为什么在经验不足的 C++ 程序员维护的代码库中使用前一种方式：如果看不到，你就可能忘掉。

使用 =default 总是比你自己实现默认语义要好。有些觉得写点儿什么总比什么都不写好，从而写出下面的代码：

```
class gslice {
        valarray<size_t> size;
        valarray<size_t> stride;
        valarray<size_t> d1;
public:
        // ...
        gslice(const gslice& a);
};

gslice::gslice(const gslice& a)
        : size{a.size },
          stride{a.stride},
          d1{a.d1}
{
}
```

这段代码不仅冗长、令 gslice 的定义难以理解，而且为错误提供了机会。例如，我可能忘记拷贝某个成员从而令其进行了默认初始化（而不是拷贝初始化）。而且，当由用户提供一个函数时，编译器就不可能再了解函数的语义，从而就不可能再进行某些优化了。对默认操作，这些优化的效果可能是非常显著的。

17.6.2　默认操作

每个生成的操作的默认含义，像编译器生成它们所用的实现方法一样，就是对类的每个基类和非 static 数据成员应用此操作。即，逐成员拷贝、逐成员默认构造，等等。例如：

```
struct S {
    string a;
    int b;
};

S f(S arg)
{
    S s0 {};    // 默认构造：{"",0}
    S s1 {s0}; // 拷贝构造
    s1 = arg;  // 拷贝赋值
    return s1; // 移动构造
}
```

s1 的拷贝构造函数会拷贝 s0.a 和 s0.b。return s1 会移动 s1.a 和 s1.b，将 s1.a 变为空字符串而 s1.b 保持不变。

注意，如果一个移出对象是内置类型，其值保持不变。这样做对于编译器而言是最简单也是最快的。如果我们希望对类成员做其他操作，就必须编写自己的移动操作。

默认的移出状态是一种能令默认析构函数和默认拷贝赋值函数正确执行的状态。C++ 语言不保证（或者说不要求）任意操作在移出对象上都能正确执行。如果你需要更强的保证，就必须实现自己的操作。

17.6.3　使用默认操作

本节通过一些例子展示拷贝操作、移动操作以及析构函数在逻辑上是如何联系起来的。如果它们并未联系起来，那么你思考它们时的一些很明显的错误就不会被编译器所捕获。

17.6.3.1　默认构造函数

考虑下面的代码：

```
struct X {
    X(int);    // 要求用一个 int 初始化一个 X
};
```

通过声明一个接受整数参数的构造函数，程序员明确地表达出：用户需要提供一个 int 来初始化一个 X。假如允许生成默认构造函数，这个简单的规则就会被打破。对下面的代码：

```
X a {1};   // 正确
X b {};    // 错误：没有默认构造函数
```

如果也希望使用默认构造函数，我们可以定义一个，或声明希望由编译器自动生成一个。例如：

```
struct Y {
    string s;
    int n;
    Y(const string& s); // 用一个字符串初始化 Y
    Y() = default;      // 允许用默认含义进行默认初始化
};
```

默认的（即编译器自动生成的）默认构造函数对每个成员进行默认构造。在本例中，Y() 将 s 设置为空字符串。一个内置类型成员的“默认初始化”其实不会对该成员进行初始化。唉！还是希望编译器能给出一个警告吧。

17.6.3.2　保持不变式

一个类通常都会有一个不变式。如果是这样，我们希望拷贝和移动操作能保持此不变式，而析构函数能释放任何用到的资源。不幸的是，编译器不可能在任何情况下都能了解程

序员所考虑的不变式是什么。考虑下面这个有些牵强的例子：

```
struct Z { // 不变式：
        // my_favorite 是 elem 中我最喜欢的元素的下标
        // largest 指向 elem 中值最大的元素
    vector<int> elem;
    int my_favorite;
    int* largest;
};
```

程序员在注释中说明了不变式，但编译器不可能阅读注释。而且，程序员没有提示如何建立和保持这个不变式。特别是，由于没有声明构造函数或赋值操作，这个不变式是隐含的。结果就是一个 Z 可以被默认操作所拷贝或移动：

```
Z v0;                    // 无初始化（糟糕！存在未定义值的可能性）
Z val {{1,2,3},1,&val[2]};  // 正确，但正确只是偶然
Z v2 = val;              // 拷贝：v2.largest 指向 val
Z v3 = move(val);        // 移动：val.elem 变为空；v3.my_favorite 越界
```

这真是乱七八糟。根本原因是 Z 的设计很糟糕，将关键信息“隐藏”在注释中或是完全遗漏。默认操作的生成规则是启发式的，可以发现常见错误并鼓励系统化的构造、拷贝、移动以及析构操作。只要可能，我们就应该：

［1］ 在构造函数中建立不变式（包括可能的资源获取）。

［2］ 在拷贝和移动操作中保持不变式（利用常用名字和类型）。

［3］ 在析构函数中做任何需要的清理工作（包括可能的资源释放）。

17.6.3.3 资源不变式

不变式很多最关键、最明显的应用都与资源管理相关。考虑一个简单的句柄 Handle：

```
template<class T> class Handle {
    T* p;
public:
    Handle(T* pp) :p{pp} { }
    T& operator*() { return *p; }
    ~Handle() { delete p; }
};
```

其思想是，给定一个用 new 分配的对象的指针，创建一个 Handle。这个 Handle 提供对象访问功能，并负责最终 delete 对象。例如：

```
void f1()
{
    Handle<int> h {new int{99}};
    // ...
}
```

Handle 声明一个接受单参数的构造函数：这禁止了生成默认构造函数。这是一个好的结果，因为默认构造函数会令 Handle<T>::p 未初始化：

```
void f2()
{
    Handle<int> h; // 错误：没有默认构造函数
    // ...
}
```

默认构造函数的缺席令我们免于陷入 delete 一个随机内存地址的情况。

Handle 还声明了一个析构函数：这禁止了生成拷贝和移动操作。这再次令我们免于糟

糕问题的困扰。考虑下面的代码：

```
void f3()
{
    Handle<int> h1 {new int{7}};
    Handle<int> h2 {h1};          // 错误：没有拷贝构造函数
    // ...
}
```

假如 Handle 有一个默认拷贝构造函数，h1 和 h2 包含同一个指针的拷贝，两者都可以 delete 它。结果是未定义的，很可能是一场灾难（见 3.3.1 节）。警告：生成拷贝操作的特性只是被弃用，并未被禁止，因此如果忽略编译器警告，这个例子可能编译成功。一般而言，如果一个类有一个指针成员，就要对默认的拷贝和移动操作保持警惕。如果该指针成员表示所有权，逐成员拷贝就是错误的。如果该指针成员不表示所有权而逐成员拷贝是恰当的，采用显式 =default 并编写必要的注释通常也是好的风格。

如果想要拷贝构造，我们可以定义像下面这样的代码：

```
template<class T>
class Handle {
    // ...
    Handle(const T& a) :p{new T{*a.p}} { }        // 克隆
};
```

17.6.3.4　部分说明的不变式

依赖于不变式但又只是通过构造函数和析构函数部分表达不变式的麻烦例子很少见，但并非前所未闻。考虑下面的代码：

```
class Tic_tac_toe {
public:
    Tic_tac_toe(): pos(9) {}   // 总是 9 个位置

    Tic_tac_toe& operator=(const Tic_tac_toe& arg)
    {
        for(int i = 0; i<9; ++i)
            pos.at(i) = arg.pos.at(i);
        return *this;
    }

    // ... 其他操作 ...

    enum State { empty, nought, cross };
private:
    vector<State> pos;
};
```

这是一个真实程序的一部分。它使用了“魔数”9 来实现拷贝赋值操作，而拷贝赋值操作没有检查参数 arg 是否真的有 9 个元素就直接访问它了。而且，这段代码显式实现了拷贝赋值操作，但没有实现拷贝构造函数。我认为这不是一段好的代码。

这段代码定义了拷贝赋值操作，因此我们还必须定义析构函数。这个析构函数可以是 =default，因为它需要做的就是确保成员 pos 被销毁，而即使没有定义拷贝赋值操作，这个销毁工作也会进行。此时，我们注意到用户自定义的拷贝赋值操作本质上与默认生成的版本没什么区别，因此仍可以采用 =default。再增加一个拷贝构造函数，我们就得到完整定义：

```
class Tic_tac_toe {
public:
    Tic_tac_toe(): pos(9) {}    // 包含 9 个位置
    Tic_tac_toe(const Tic_tac_toe&) = default;
    Tic_tac_toe& operator=(const Tic_tac_toe& arg) = default;
    ~Tic_tac_toe() = default;

    // ... 其他操作 ...

    enum State { empty, nought, cross };
private:
    vector<State> pos;
};
```

观察这段代码，我们发现这些 =default 的最终效果就是去除了移动操作。这是我们所期望的吗？可能不是。当我们将拷贝赋值操作声明为 =default 时，就消除了对魔数 9 的错误依赖，除非还有到目前为止尚未提及的其他 Tic_tac_toe 操作也“与魔数有硬连接”，否则我们可以安全地增加移动操作。最简单的方法是删除显式 =default，然后我们就会看到一个真的极为普通正常的类型：

```
class Tic_tac_toe {
public:
    // ... 其他操作 ...
    enum State { empty, nought, cross };
private:
    vector<State> pos {Vector<State>(9)};        // 包含 9 个位置
};
```

我们从这个例子以及“怪异组合”默认操作的其他例子中得到的一个结论是，应该高度警惕这种类型：这种不合常规的设计通常隐藏着错误。对每个类，我们都应问：

[1] 需要默认构造函数吗（由于默认构造函数不能满足要求或已被另一个构造函数所禁止）？

[2] 需要析构函数吗（例如，由于某种资源需要释放）？

[3] 需要拷贝操作吗（由于默认拷贝语义不能满足需求，例如，由于类是一个基类，或它包含指针，指向的对象必须被类释放）？

[4] 需要移动操作吗（由于默认语义不能满足需求，例如，由于空对象无意义）？

特别是，我们永远不该孤立地考虑这些操作。

17.6.4　使用 delete 删除的函数

我们可以“删除”一个函数；即，我们可以声明一个函数不存在，从而令（隐式或显式）使用它的尝试成为错误。这种机制最明显的应用是消除其他默认函数。例如，防止拷贝基类是很常见的，因为这种拷贝容易导致切片（见 17.5.1.4 节）：

```
class Base {
    // ...
    Base& operator=(const Base&) = delete;// 不允许拷贝
    Base(const Base&) = delete;

    Base& operator=(Base&&) = delete;          // 不允许移动
    Base(Base&&) = delete;
};
```

```
Base x1;
Base x2 {x1}; // 错误：没有拷贝构造函数
```

通常，允许和禁止拷贝及移动更方便的方法是声明我们需要什么（使用 =default，见 17.6.1 节），而不是我们不想要什么（使用 =delete）。但是，我们可以使用 delete 删除任何我们能声明的函数。例如，我们可以将一个特例化版本从函数模板众多可能的特例化版本中删除：

```
template<class T>
T* clone(T* p) // 返回 *p 的副本
{
    return new T{*p};
};

Foo* clone(Foo*) = delete;          // 不要尝试克隆一个 Foo

void f(Shape* ps, Foo* pf)
{
    Shape* ps2 = clone(ps);         // 没问题
    Foo* pf2 = clone(pf);           // 错误：clone(Foo*) 已被删除
}
```

另一种应用是删除不需要的类型转换。例如：

```
struct Z {
    // ...
    Z(double);          // 可以用 double 初始化
    Z(int) = delete;    // 但不能用整数初始化
};

void f()
{
    Z z1 {1};       // 错误：Z(int) 已被删除
    Z z2 {1.0};     // 正确
}
```

进一步的用途是控制在哪里分配类对象：

```
class Not_on_stack {
    // ...
    ~Not_on_stack() = delete;
};

class Not_on_free_store {
    // ...
    void* operator new(size_t) = delete;
};
```

你无法声明一个不能被销毁的局部变量（见 17.2.2 节），如果你已经使用 =delete 删除了类的内存分配运算符（见 19.2.5 节），你也就不能在自由存储空间分配该类的对象了。例如：

```
void f()
{
    Not_on_stack v1;            // 错误：不能销毁
    Not_on_free_store v2;       // 正确

    Not_on_stack* p1 = new Not_on_stack;            // 正确
    Not_on_free_store* p2 = new Not_on_free_store;  // 错误：不能分配
}
```

但是，我们永远不能使用 delete 删除那个 Not_on_stack 对象。另一种替代技术是将析构函

数声明为 private（见 17.2.2 节），以解决此问题。

请注意已使用 =delete 删除的函数与只是未声明的函数之间的差别。在前一种情况下，编译器会发现程序员试图使用已使用 delete 删除的函数的情况并报错。在后一种情况下，编译器会寻找替代方法，如不调用析构函数或使用全局 operator new()。

17.7　建议

[1] 应该将构造函数、赋值操作以及析构函数设计为一组匹配的操作；17.1 节。

[2] 使用构造函数为类建立不变式；17.2.1 节。

[3] 如果一个构造函数获取了资源，那么这个类就需要一个析构函数释放该资源；17.2.2 节。

[4] 如果一个类有虚函数，它就需要一个虚析构函数；17.2.5 节。

[5] 如果一个类没有构造函数，它可以进行逐成员初始化；17.3.1 节。

[6] 优先选择使用 {} 初始化而不是 = 和 () 初始化；17.3.2 节。

[7] 当且仅当类对象有“自然的”默认值时才为类定义默认构造函数；17.3.3 节。

[8] 如果一个类是一个容器，为它定义一个初始化器列表构造函数；17.3.4 节。

[9] 按声明顺序初始化成员和基类；17.4.1 节。

[10] 如果一个类有一个引用成员，它可能需要拷贝操作（拷贝构造函数和拷贝赋值操作）；17.4.1.1 节。

[11] 在构造函数中优先选择成员初始化而不是赋值操作；17.4.1.1 节。

[12] 使用类内初始化器来提供默认值；见 17.4.4 节。

[13] 如果一个类是一个资源句柄，它可能需要拷贝和移动操作；17.5 节。

[14] 当编写一个拷贝构造函数时，小心拷贝每个需要拷贝的元素（小心默认初始化器）；17.5.1.1 节。

[15] 一个拷贝操作应该提供等价性和独立性；17.5.1.3 节。

[16] 小心纠缠的数据结构；17.5.1.3 节。

[17] 优先选择移动语义和写前拷贝而不是浅拷贝；17.5.1.3 节。

[18] 如果一个类被用作基类，防止切片现象；见 17.5.1.4 节。

[19] 如果一个类需要一个拷贝操作或一个析构函数，它可能需要一个构造函数、一个析构函数、一个拷贝赋值操作以及一个拷贝构造函数；17.6 节。

[20] 如果一个类有一个指针成员，它可能需要一个析构函数和非默认拷贝操作；17.6.3.3 节。

[21] 如果一个类是一个资源句柄，它需要一个构造函数、一个析构函数和非默认拷贝操作；17.6.3.3. 节。

[22] 如果一个默认构造函数、赋值操作或析构函数是恰当的，让编译器自动生成它（不要自己重新编写）；17.6 节。

[23] 显式说明你的不变式：用构造函数建立不变式，用赋值操作保持不变式；17.6.3.2 节。

[24] 确保拷贝赋值操作能安全进行自赋值；17.5.1 节。

[25] 当向类添加一个新成员时，检查用户自定义构造函数是否需要更新，以便初始化新加入的成员；17.5.1 节。

运算符重载

我的用词与其字面意义并无二致，
不多也不少。

——蛋头先生

- 引言
- 运算符函数
  二元和一元运算符；运算符的预置含义；运算符与用户自定义类型；传递对象；名字空间中的运算符
- 复数类型
  成员和非成员运算符；混合模式运算；类型转换；字面值常量；访问函数；辅助函数
- 类型转换
  类型转换运算符；explicit 类型转换运算符；二义性
- 建议

18.1 引言

所有技术领域以及几乎所有非技术领域都开发了各自的便捷的简写符号系统，人们可以利用这些简写符号展示并讨论那些常见的概念。比如，我们熟悉的

```
x*y+z
```

比下述形式直观得多：

```
multiply y by z and add the result to x
```

为常见操作设计简写符号是一项非常重要的工作，怎么形容其重要性都不为过。

像其他绝大多数编程语言一样，C++ 也为它的内置数据类型设计了一套运算符。然而，还有很多情况下运算符作用的数据类型并非 C++ 内置类型，我们需要将这些类型表示为用户自定义类型。例如，如果你想在 C++ 中使用复数、矩阵代数、逻辑信号或者字符串，需要自定义一些类来表示它们。我们建议为这些类定义运算符，而非使用基本形式的函数，因为前者可以帮助程序员以更符合常规的方式便捷地操纵类的对象。考虑如下情况：

```
class complex {                    // 非常简单的复数类型
    double re, im;
public:
    complex(double r, double i) :re{r}, im{i} { }
    complex operator+(complex);
    complex operator*(complex);
};
```

上述代码给出了复数类型的一个简单实现。complex 表示为一对双精度浮点数，我们可以用

运算符 + 和 * 操作它。程序员用 complex::operator+() 和 complex::operator*() 分别表示 + 和 * 的含义。假设 b 和 c 的类型是 complex，则 b+c 等价于 b.operator+(c)。在此基础上，我们可以进一步写出 complex 表达式的惯常解释：

```
void f()
{
    complex a = complex(1,3.1);
    complex b {1.2, 2};
    complex c {b};

    a = b+c;
    b = b+c*a;
    c = a*b+complex(1,2);
}
```

根据我们所熟悉的运算符优先级可知，第二句话的意思是 b=b+(c*a)，而非 b=(b+c)*a。

请注意，C++ 语法规定符号 {} 只能表示初始化器，并且只能位于赋值运算符的右侧：

```
void g(complex a, complex b)
{
    a = {1,2};              // OK: 位于赋值运算符的右侧
    a += {1,2};             // OK: 位于赋值运算符的右侧
    b = a+{1,2};            // 语法错误
    b = a+complex{1,2};// OK
    g(a,{1,2});             // 函数参数可以看作初始化器
    {a,b} = {b,a};          // 语法错误
}
```

乍看起来即使允许在更多地方使用 {} 也没什么大不了的，但是在表达式中随意使用 {} 会造成很多问题（比如，分号后面如果紧跟着一个 {，那么它表示的是一条表达式的开始还是一个块的开始呢）。此外，一旦允许随意使用 {} 的话，编译器将很难判别程序正确与否，也很难给出准确的错误消息。

运算符重载最常用于数字类型，但是用户自定义运算符的用处绝不仅仅局限于数字类型。例如，为了设计通用且抽象的访问接口，我们经常需要使用 ->、[]、() 等运算符。

18.2 运算符函数

我们可以声明一些新的函数，令其表示下述运算符（见 10.3 节）：

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

但是，用户无权定义下列运算符：

:: 作用域解析（见 6.3.4 节和 16.2.12 节）；

. 成员选择（见 8.2 节）；

.* 通过指向成员的指针访问成员（见 20.6 节）。

这 3 种运算接受一个名字而非一个值作为其第二个运算对象，主要的作用是指向或者引用成员。假设允许用户重载这些运算符，程序将面临错误风险［Stroustrup,1994］。下列具名“运算符”负责报告其运算对象的某些基本情况，因此也不能重载：

sizeof　对象的尺寸（见 6.2.8 节）；

alignof　对象的对齐方式（见 6.2.9 节）；

typeid　对象的 type_info（见 22.5 节）。

最后，三元条件表达式也不能被重载（没什么特别的原因）：

?:　条件表达式（见 9.4.1 节）。

此外，我们用 operator"" 表示用户定义的字面值常量（见 19.2.6 节）。其实并没有形如 "" 的运算符，所以 operator"" 可以看成是一种语法上的借鉴或者技巧。类似地，operator T() 表示向 T 的类型转换（见 18.4 节）。

C++ 不允许程序员定义新的运算符，但是当现有的运算符集合不够用的时候你可以使用函数调用符号作为补充。比如，你应该使用 pow() 而非 **。这一约束似乎有点过于严厉，但是放松要求的话可能会导致程序的二义性。举个例子，定义运算符 ** 令其表示幂指运算看起来是个不错的主意，但是仔细想想的话这种做法其实并不可行。** 应该绑定到左侧（Fortran 风格）还是右侧（Algol 风格）呢？表达式 a**p 的意思是 a*(*p) 还是 (a)**(p) 呢？当然我们也能找到解决这些问题的办法，但是谁也不知道增加一些微妙的技术规则后会对代码的可读性和可维护性产生什么样的影响。如果你对此存疑，最好使用具名函数。

运算符函数名字的组成规则是在关键字 operator 后面紧跟运算符本身，比如 operator<<。声明和调用运算符函数的方式与其他函数完全一致。使用运算符等价于显式地调用运算符函数，我们可以把前者看成是后者的一种简写形式。例如：

```
void f(complex a, complex b)
{
    complex c = a + b;              // 简写
    complex d = a.operator+(b);     // 显式调用
}
```

在已知 complex 定义的情况下，上面两个初始化器是等价的。

18.2.1　二元和一元运算符

我们可以用接受一个参数的非 static 成员函数定义二元运算符，也可以用接受两个参数的非成员函数定义它。对于任意一种二元运算符 @，aa@bb 可以理解成 aa.operator@(bb) 或者 operator@(aa,bb)。如果这两种形式都被定义了，则由重载解析（见 12.3 节）决定到底使用其中哪一个。例如：

```
class X {
public:
    void operator+(int);
    X(int);
};

void operator+(X,X);
void operator+(X,double);

void f(X a)
{
    a+1;        // a.operator+(1)
    1+a;        // ::operator+(X(1),a)
    a+1.0;      // ::operator+(a,1.0)
}
```

对于一元运算符，不管它是前置的还是后置的，我们既可以用不接受任何参数的非 static 成员函数定义它，也可以用接受一个参数的非成员函数定义它。对于任意一元前置运算符 @，@aa 可以理解成 aa.operator@() 或者 operator@(aa)。如果这两种形式都定义了，则由重载解析（见 12.3 节）决定到底使用其中哪一个。对于任意一元后置运算符 @，aa@ 可以理解成 aa.operator@(int) 或者 operator@(aa,int)。我们将在 19.2.4 节进一步介绍它。如果这两种形式都定义了，则由重载解析（见 12.3 节）决定到底使用其中哪一个。在声明运算符的时候，我们必须确保它的语法与 C++ 标准的规定一致（§ iso.A）。例如，用户无权定义一元的 % 或者三元的 +。考虑如下情况：

```
class X {
public:          // 成员（注意了隐式的 this 指针）：

     X* operator&();          // 前置一元 &（取地址）
     X operator&(X);          // 二元 &（与）
     X operator++(int);       // 后置递增运算符（见 19.2.4 节）
     X operator&(X,X);        // 错误：三元的
     X operator/();           // 错误：一元的 /
};
// 非成员函数：

X operator-(X);               // 前置一元减法
X operator-(X,X);             // 二元减法
X operator--(X&,int);         // 后置递减运算符
X operator-();                // 错误：缺少运算对象
X operator-(X,X,X);           // 错误：三元的
X operator%(X);               // 错误：一元的 %
```

我们将在 19.2.1 节介绍运算符 []，19.2.2 节介绍运算符 ()，19.2.3 节介绍运算符 ->，19.2.4 节介绍运算符 ++ 和 --，11.2.4 节和 19.2.5 节介绍分配和释放运算符。

运算符 operator=（见 18.2. 节）、operator[]（见 19.2.1 节）、operator()（见 19.2.2 节）和 operator->（见 19.2.3 节）必须是非 static 成员函数。

运算符 &&、|| 和 ,（逗号）的默认含义中包含顺序信息：它们的第一个运算对象在第二个运算对象之前求值（&& 和 || 的第二个运算对象有可能不求值）。这一特殊的规则并不适用于用户自定义的 &&、|| 和 ,（逗号），它们与其他二元运算符没什么不一样。

18.2.2 运算符的预置含义

某些内置运算符的含义与其他接受相同参数的运算符组合的含义相同。如果 a 是一个 int，则 ++a 等价于 a+=1 以及 a=a+1。除非有专门的说明，否则这一现象对于用户自定义的运算符并不适用。举个例子，编译器不会根据 Z::operator+() 和 Z::operator=() 的定义生成 Z::operator+=() 的定义。

当作用于类的对象时，运算符 =（赋值）、&（取地址）和 ,（顺序，见 10.3.2 节）自带预置的含义。我们可以去除（"删除"，见 17.6.4 节）这些预置的含义：

```
class X {
public:
     // ...
     void operator=(const X&) = delete;
     void operator&() = delete;
     void operator,(const X&) = delete;
     // ...
};
```

```
void f(X a, X b)
{
    a = b;      // 错误：不存在运算符 operator=()
    &a;         // 错误：不存在运算符 operator&()
    a,b;        // 错误：不存在运算符 operator&()
}
```

我们也可以赋予这些运算符新的含义。

18.2.3　运算符与用户自定义类型

运算符函数应该是成员函数或者至少接受一个用户自定义类型的参数（重新定义 new 和 delete 的函数不必满足上述条件）。这条规则可以确保除非表达式含有用户自定义的类型，否则用户无法改变表达式的含义。尤其不允许定义只处理指针的运算符函数，这可以确保 C++ 可扩展但不产生不确定性（除了类对象的 =、& 和 , 运算符之外）。

如果某个运算符函数接受一个内置类型（见 6.2.1 节）作为它的第一个运算对象，那么该函数不能是成员函数。例如，考虑把复数变量 aa 与整数 2 相加的情况：只要声明了适当的成员函数，我们就能把 aa+2 理解成 aa.operator+(2)；但是不能把 2+aa 看成 2.operator+(aa)，因为类 int 并没有定义这样的 +。就算是有，也需要为 2+aa 和 aa+2 分别定义不同的成员函数。因为编译器不了解用户自定义的 + 的含义，它不清楚该运算符是否满足交换律，当然也就不能把 2+aa 当成 aa+2。上述问题可以通过定义一个或多个非成员函数来解决（见 18.3.2 节和 19.4 节）。

枚举类型是用户自定义的类型，我们可以为其定义运算符。例如：

```
enum Day { sun, mon, tue, wed, thu, fri, sat };

Day& operator++(Day& d)
{
    return d = (sat==d) ? sun : static_cast<Day>(d+1);
}
```

编译器会检查每一条表达式是否具有二义性。只要用户自定义的运算符提供了一种可能的解释，编译器就会按照 12.3 节介绍的重载解析规则检查表达式。

18.2.4　传递对象

我们在定义运算符的时候通常希望提供一种比较便捷的符号，比如 a=b+c。因此，关于向运算符函数传递参数以及返回结果的问题，可供选择的方式比较有限。例如，我们不能请求一个指针类型的参数并且指望程序员使用取地址符，也不能让运算符返回指针并期望解引用它：*a=&b+&c 是不可接受的形式。

关于参数，主要有两种选择（见 12.2 节）：

- 值传递
- 引用传递

对于大小在 1 – 4 个字长之间的小对象来说，采用值传递的方式通常是最好的选择，得到的性能也最好。然而，传递和使用参数的实际性能会受机器的体系结构、编译器接口规范（应用二进制接口，ABI）和参数访问次数（基本上访问值传递的参数比引用传递的参数快）等因素的影响。假设我们用一对 int 表示 Point：

```
void Point::operator+=(Point delta);    // 值传递
```

对于较大的对象，我们一般采用引用传递的方式。例如，因为 Matrix（由 double 构成的一种简单矩阵，见 17.5.1 节）所占的空间通常不止几个字，所以我们采用引用传递的方式：

```
Matrix operator*(const Matrix&, const Matrix&);    // 使用引用传递
```

尤其是，如果传入被调函数的是内容不会被修改的较大对象，则应该采用 const 引用的方式（见 12.2.1 节）。

通常情况下，一个运算符返回一个结果。向一个新创建的对象返回指针或者引用基本上是一种比较糟糕的选择：使用指针会带来符号使用方面的困难，而引用自由存储上的对象（不管使用指针还是引用）会导致资源管理困难。最好的方式是用传值的方式返回对象。对于 Matrix 等较大的对象来说，我们应该定义移动操作以使得值传递的过程足够有效（见 3.3.2 节和 17.5.2 节）。例如：

```
Matrix operator+(const Matrix& a, const Matrix& b)      // 通过传值返回
{
    Matrix res {a};
    return res+=b;
}
```

请注意，如果运算符返回的是其参数对象中的某一个，则该运算符能够并且通常通过引用的方式返回。例如，我们可以把 Matrix 的运算符 += 定义成如下形式：

```
Matrix& Matrix::operator+=(const Matrix& a)      // 通过传引用返回
{
    if (dim[0]!=a.dim[0] || dim[1]!=a.dim[1])
        throw std::exception("bad Matrix += argument");

    double* p = elem;
    double* q = a.elem;
    double* end = p+dim[0]*dim[1];
    while(p!=end)
        *p++ += *q++

    return *this;
}
```

这一现象在被实现为成员函数的运算符中尤其普遍。

如果函数只是把对象简单地传递给另一个函数，应该使用右值引用参数（见 17.4.3 节，23.5.2.1 节，28.6.3 节）。

18.2.5 名字空间中的运算符

运算符要么是类的成员函数，要么定义在某个名字空间（可能是全局名字空间）中。考虑如下所示的标准库字符串 I/O 函数的简化版本：

```
namespace std {          // 简化的 std

    class string {
        // ...
    };
    class ostream {
        // ...
        ostream& operator<<(const char*);          // 输出 C 风格的字符串
    };
```

```
        extern ostream cout;

        ostream& operator<<(ostream&, const string&);    // 输出 std::string
    } // 名字空间 std

    int main()
    {
        const char* p = "Hello";
        std::string s = "world";
        std::cout << p << ", " << s << "!\n";
    }
```

这段代码的输出结果是 Hello, world!，但是为什么呢？请注意，我们并没有通过书写下述语句令 std 的全部内容变得可见：

```
using namespace std;
```

相反，我给 string 和 cout 加上了 std:: 前缀。换句话说，我采取的是最合理的措施，并没有污染全局名字空间或者通过其他方式引入不必要的依赖关系。

C 风格字符串的输出运算符是 std::ostream 的成员，因此

```
std::cout << p
```

意味着

```
std::cout.operator<<(p)
```

但是 std::ostream 并没有可以输出 std::string 的成员函数，因此

```
std::cout << s
```

的含义是

```
operator<<(std::cout,s)
```

我们可以根据运算对象的类型找到名字空间中的运算符，就像根据参数类型可以找到函数一样（见 14.2.4 节）。尤其是，因为 cout 位于名字空间 std 中，所以当我们寻找合适的 << 定义时会考虑 std。此时，编译器找到并且使用

```
std::operator<<(std::ostream&, const std::string&)
```

考虑二元运算符 @ 的情况。假设 x 的类型是 X，y 的类型是 Y，则 x@y 的解析过程是：

- 如果 X 是一个类，查找 X 是否有成员 operator@ 或者 X 的基类是否有成员 operator@；
- 在 x@y 的上下文中查找是否有 operator@ 的声明；
- 如果 X 定义在名字空间 N 中，在 N 的范围内查找 operator@ 的声明；
- 如果 Y 定义在名字空间 M 中，在 M 的范围内查找 operator@ 的声明。

我们有可能会找到多个 operator@ 的声明，此时根据重载解析规则（见 12.3 节）寻找最佳匹配。上述查找机制只有当运算符的至少一个运算对象是用户自定义类型时才会执行。因此，我们把用户自定义的类型转换（见 18.3.2 节和 18.4 节）也考虑在内。请注意，类型别名只是一个同义词，它不属于单独的用户自定义类型（见 6.5 节）。

一元运算符的解析方式与之类似。

在查找运算符时，成员函数与非成员函数相比并没有明显的优势。这一点与具名函数的查找过程不同（见 14.2.4 节）。因为运算符的定义不存在彼此隐藏的问题，所以内置运算符

永远都是可以访问的，用户可以在此基础上赋予运算符新的含义。例如：

```
X operator!(X);

struct Z {
    Z operator!();                    // 不会除截 ::operator!()
    X f(X x) { /* ... */ return !x; }  // 调用 ::operator!(X)
    int f(int x) { /* ... */ return !x; }  // 对 int 调用内置！
};
```

标准库 iostream 定义了 << 成员函数以输出内置类型，用户可以在不修改 ostream 类的前提下定义输出自定义类型的 <<（见 38.4.2 节）。

18.3　复数类型

18.1 节实现的复数类型对于大多数人来说显得过于严格了。例如，我们希望下面的代码是有效的：

```
void f()
{
    complex a {1,2};
    complex b {3};
    complex c {a+2.3};
    complex d {2+b};
    b = c-2-c;
}
```

此外，我们还希望提供一些其他的操作，比如用于比较的 ==，用于输出的 << 以及 sin() 和 sqrt() 等数学函数。

类 complex 是一种具体类型，因此它的设计遵循 16.3 节提出的准则。此外，复数的算术运算非常依赖 complex 定义的运算符，并且遵守运算符重载的基本规则。

本节开发的 complex 使用 double 作为其标量的类型，因而与标准库 complex<double> 几乎是等价的（见 40.4 节）。

18.3.1　成员和非成员运算符

我不太喜欢函数直接操作对象的内容。通过在类内定义只修改第一个参数的值的运算符（比如 +=），可以最大限度地减少此类函数的数量。在类的外部定义那些通过参数计算新值的运算符（比如 +），这些运算符可以使用其些之前定义在类内部的运算符：

```
class complex {
    double re, im;
public:
    complex& operator+=(complex a); // 需要访问类的数据成员
    // ...
};

complex operator+(complex a, complex b)
{
    return a += b; // 通过 += 访问类的数据成员
}
```

operator+() 的参数是通过值传递的方式传入的，因此 a+b 不会修改其运算对象的内容。

基于上述声明，我们可以写出：

```
void f(complex x, complex y, complex z)
{
    complex r1 {x+y+z}; // r1 = operator+(operator+(x,y),z)

    complex r2 {x};     // r2 = x
    r2 += y;            // r2.operator+=(y)
    r2 += z;            // r2.operator+=(z)
}
```

除了可能存在的性能差异外，r1 和 r2 的计算几乎是等价的。

定义复合赋值运算符（比如 += 和 *=）要比定义与其对应的“简化版”（比如 + 和 *）更简单。这个结论可能会让很多人吃惊，但事实是 + 运算要涉及 3 个对象（2 个运算对象和 1 个结果），而 += 运算只涉及 2 个。后者不需要处理临时对象，因此运行时效率有所提升。例如：

```
inline complex& complex::operator+=(complex a)
{
    re += a.re;
    im += a.im;
    return *this;
}
```

该程序不需要使用表示相加结果的临时变量，因此编译器可以很容易对它进行完美的内联。

一个好的优化器也能为普通的 + 运算符生成足够优化的代码，但是我们不一定总能遇到好的优化器，而且也不是所有类型都像 complex 这么简单。因此，我们将在 19.4 节讨论如何通过直接访问类的数据定义运算符。

18.3.2　混合模式运算

为了执行 2+z（其中 z 是一个 complex），我们需要定义一个可以接受不同类型运算对象的 + 运算符。在 Fortran 的术语中，我们称之为*混合模式运算*（mixed-mode arithmetic）。具体做法是为运算符增加一些相应的版本：

```
class complex {
    double re, im;
public:
    complex& operator+=(complex a)
    {
        re += a.re;
        im += a.im;
        return *this;
    }

    complex& operator+=(double a)
    {
        re += a;
        return *this;
    }

    // ...
};
```

operator+() 的 3 个变体可以定义在 complex 的外部：

```
complex operator+(complex a, complex b)
{
    return a += b;  // 调用 complex::operator+=(complex)
}
```

```
complex operator+(complex a, double b)
{
    return {a.real()+b,a.imag()};
}

complex operator+(double a, complex b)
{
    return {a+b.real(),b.imag()};
}
```

其中，访问函数 real() 和 imag() 的定义将在 18.3.6 节介绍。

基于上述关于 + 的声明，我们可以继续编写如下代码：

```
void f(complex x, complex y)
{
    auto r1 = x+y;  // 调用 operator+(complex,complex)
    auto r2 = x+2;  // 调用 operator+(complex,double)
    auto r3 = 2+x;  // 调用 operator+(double,complex)
    auto r4 = 2+3;  // 内置整数加法
}
```

为了体现完整性，我特意添加了一条内置的整数加法运算。

18.3.3　类型转换

为了用标量对 complex 变量赋值和初始化，我们需要进行从标量（整数或者浮点数）向 complex 的类型转换。例如：

```
complex b {3}; // 含义是 b.re=3, b.im=0

void comp(complex x)
{
    x = 4;      // 含义是 x.re=4, x.im=0
    // ...
}
```

我们的做法是实现一个接受单参数的构造函数，它的任务是从参数类型转换为构造函数类型。例如：

```
class complex {
    double re, im;
public:
    complex(double r) :re{r}, im{0} { }   // 从 double 构建一个 complex
    // ...
};
```

该构造函数的效果是在复平面内嵌入一条实线。

构造函数从本质上来说是在创建一个指定类型的值。当需要创建这样的值并且有人提供了合适的初始化器时，就会调用构造函数来实现它。因此，我们无须显式调用接受单参数的构造函数。例如：

```
complex b {3};
```

的意思是：

```
complex b {3,0};
```

系统隐式执行某用户自定义类型转换的前提是该转换具有唯一性（见 12.3 节）。如果你不希

望构造函数被隐式调用，最好把它声明成 explicit 的（见 16.2.6 节）。

显然，除了上面的构造函数之外，我们同样需要接受两个 double 的构造函数，也需要用 {0,0} 作为 complex 初始值的默认构造函数：

```
class complex {
    double re, im;
public:
    complex() : re{0}, im{0} { }
    complex(double r) : re{r}, im{0} { }
    complex(double r, double i) : re{r}, im{i} { }
    // ...
};
```

我们可以用默认参数进一步简写程序：

```
class complex {
    double re, im;
public:
    complex(double r =0, double i =0) : re{r}, im{i} { }
    // ...
};
```

默认情况下，complex 的拷贝操作分别拷贝实部和虚部（见 16.2.2 节）。例如：

```
void f()
{
    complex z;
    complex x {1,2};
    complex y {x}; // y 的值是 {1,2}
    z = x;         // z 的值是 {1,2}
}
```

18.3.3.1　运算对象的类型转换

我们为四则运算分别定义了 3 个不同的版本：

```
complex operator+(complex,complex);
complex operator+(complex,double);
complex operator+(double,complex);
// ...
```

这种做法冗长乏味，并且蕴含着错误风险。如果每个函数的每个参数都有 3 种可用的数据类型会怎么样呢？按照上面的逻辑，我们需要 3 个版本的单参数函数，9 个版本的双参数函数以及 27 个版本的三参数函数。显然这些版本彼此非常相似。事实上，几乎所有版本都包含由参数类型向通用类型的简单类型转换。

要想实现同一个函数的不同参数组合，另一种思路是利用类型转换。例如，我们的 complex 类提供了一个把 double 转换为 complex 的构造函数。因此，我们可以为 complex 定义一个唯一的相等性运算符：

```
bool operator==(complex,complex);

void f(complex x, complex y)
{
    x==y;    // 代表 operator==(x,y)
    x==3;    // 代表 operator==(x,complex(3))
    3==y;    // 代表 operator==(complex(3),y)
}
```

有时人们希望把函数的几个版本区分开来。例如，有些情况下类型转换会提升程序开销，另

一些情况下我们可以为某种特定的参数类型应用更简单的算法。当此类问题并不突出时，一种行之有效的措施是利用类型转换技术为函数提供一个最通用的版本，再辅以少数几种必要的变形。这样做可以完美地解决由混合模式运算带来的组合爆炸问题。

假设函数或者运算符存在几个不同的版本，则编译器需要依据参数类型和可用的（标准或者用户自定义）类型转换做出“最优选择”。如果找不到最佳匹配，就表示该表达式存在二义性，并将造成程序错误（见 12.3 节）。

表达式中通过构造函数显式或者隐式构造的对象是自动对象，一有机会就会销毁它（见 10.3.4 节）。

在 . 或者 -> 的左侧不会执行隐式的用户自定义类型转换，即使 . 本身是隐式的也是如此。例如：

```
void g(complex z)
{
    3+z;                    // OK: complex(3)+z
    3.operator+=(z);        // 错误：3 不是类的对象
    3+=z;                   // 错误：3 不是类的对象
}
```

因此，你可以把需要左值的运算符近似当成是该运算符的左侧运算对象，前提是该运算符是作为成员函数出现的。不过这也只是一种近似，因为我们在访问其中的临时量的时候可能会用到修改操作，比如 operator+=()：

```
complex x {4,5}
complex z {sqrt(x)+={1,2}};          // 类似于 tmp=sqrt(x), tmp+={1,2}
```

如果不想使用隐式类型转换，可以将其声明成 explicit（见 16.2.6 节和 18.4.2 节）。

18.3.4 字面值常量

C++ 有内置类型的字面值常量，比如 1.2 和 12e3 都是 double 类型的字面值常量。通过把构造函数声明成 constexpr（见 10.4 节），我们也能得到非常类似的 complex 字面值常量。例如：

```
class complex {
public:
    constexpr complex(double r =0, double i =0) : re{r}, im{i} { }
    // ...
}
```

就像内置类型字面值常量一样，我们也能在编译时用 complex 的组成部分构造它。例如：

```
complex z1 {1.2,12e3};
constexpr complex z2 {1.2,12e3};   // 编译时编译时初始化
```

如果构造函数很简单而且是内联的，尤其如果构造函数是 constexpr 的，那么我们很容易把用字面值常量参数调用构造函数的结果看成是一个字面值常量。

让我们更进一步，为 complex 类型引入用户自定义的字面值常量（见 19.2.6 节）。我们可以把 i 定义成后缀，它意思是“虚部”。例如：

```
constexpr complex<double> operator "" i(long double d)    // 虚部字面值常量
{
    return {0,d};    // complex 是一种字面值常量类型
}
```

基于上述约定，我们就可以写出：

```
complex z1 {1.2+12e3i};

complex f(double d)
{
    auto x {2.3i};
    return x+sqrt(d+12e3i)+12e3i;
}
```

这种用户自定义的字面值常量与 constexpr 构造函数得到的字面值常量相比有个优势：我们可以在表达式的中间使用用户自定义的字面值常量，而 {} 符号只能出现在有类型限定的地方。上面的例子大体上与接下来的程序相当：

```
complex z1 {1.2,12e3};

complex f(double d)
{
    complex x {0,2.3};
    return x+sqrt(complex{d,12e3})+complex{0,12e3};
}
```

其实我相当怀疑大多数人在选择字面值常量的实现方式时完全是出于个人的美学认知以及工作的领域。标准库 complex 采用的是 constexpr 的方式而非用户自定义字面值常量。

18.3.5　访问函数

到目前为止我们仅仅为 complex 提供了构造函数和算术运算符，尚无法满足实用的要求。我们尤其需要的一项功能是获取并更改实部和虚部的值：

```
class complex {
    double re, im;
public:
    constexpr double real() const { return re; }
    constexpr double imag() const { return im; }

    void real(double r) { re = r; }
    void imag(double i) { im = i; }
    // ...
};
```

事实上，我认为为类的每一个成员都提供单独的访问控制并不是个好主意；而且通常情况下也确实如此。对于很多类型来说，单独的访问控制（有时称为读写函数，get-and-set functions）意味着程序灾难。单独的访问控制一不小心就会破坏不变式，而且要想改变类的表示也会变得更难。例如，对于 16.3 节的 Date 和 19.3 节的 String 来说，为它们的每个成员提供读取器和设置器肯定会大大增加误用的几率。相反，complex 的 real() 和 imag() 就有用多了：很多算法会因为具有独立设置实部和虚部的权限而变得异常简洁。例如，我们可以利用 real() 和 imag() 把一些简单、常用、有用的操作（如 ==）简化为非成员函数（在不增加性能开销的基础上）：

```
inline bool operator==(complex a, complex b)
{
    return a.real()==b.real() && a.imag()==b.imag();
}
```

18.3.6　辅助函数

如果把所有的代码片段组合在一起，complex 类就会变成如下形式：

```
class complex {
    double re, im;
public:
    constexpr complex(double r =0, double i =0) : re(r), im(i) { }

    constexpr double real() const { return re; }
    constexpr double imag() const { return im; }

    void real(double r) { re = r; }
    void imag(double i) { im = i; }

    complex& operator+=(complex);
    complex& operator*=(double);

    // -=, *= 和 /=
};
```

此外，还必须提供一些辅助函数：

```
complex operator+(complex,complex);
complex operator+(complex,double);
complex operator+(double,complex);

// 二元的 -, * 和 /

complex operator-(complex); // 一元减法
complex operator+(complex); // 一元加法

bool operator==(complex,complex);
bool operator!=(complex,complex);

istream& operator>>(istream&,complex&);    // 输入
ostream& operator<<(ostream&,complex);     // 输出
```

请注意，real() 和 imag() 对于定义比较操作必不可少，而且接下来的绝大多数辅助函数的定义也都依赖于 real() 和 imag()。

下列函数有助于我们从极坐标的角度思考问题：

```
complex polar(double rho, double theta);
complex conj(complex);

double abs(complex);
double arg(complex);
double norm(complex);

double real(complex);          // 便于使用
double imag(complex);          // 便于使用
```

最后，我们还必须提供一组标准数学函数：

```
complex acos(complex);
complex asin(complex);
complex atan(complex);
// ...
```

从用户的角度来看，我们提供的 complex 与标准库 <complex> 中的 complex<double> 几乎完全一样（见 5.6.2 节和 40.4 节）。

18.4　类型转换

我们可以通过下述方式实现类型转换

- 接受单参数的构造函数（见 16.2.5 节）；
- 类型转换运算符（见 18.4.1 节）。

在任一种情况中，类型转换都能是：

- explicit，换句话说，只有直接初始化时才会执行类型转换（见 16.2.6 节），比如并未使用 = 的初始化器。
- 隐式的，换句话说，只要不会引起二义性该转换都可能发生（见 18.4.3 节），比如作为函数的参数时。

18.4.1　类型转换运算符

使用接受单参数的构造函数执行类型转换虽然便捷，但有时候并非如人们所愿。并且，构造函数也无法指定：

［1］ 从用户自定义类型向内置类型的隐式转换（因为内置类型不是类）；

［2］ 从新类向已定义类的类型转换（在未对旧类的声明做出修改前）。

我们可以通过为源类型定义一个类型转换运算符（conversion operator）来解决上述问题。成员函数 X::operator T() 定义了从 X 向 T 的类型转换，其中 T 是一个类型名。例如，我们定义一种只占 6 个二进制位的整数 Tiny。我们希望在算术运算中它可以和普通的整数完美地融合在一起，并且当 Tiny 的值超出其表示范围时抛出 Bad_range 异常：

```
class Tiny {
    char v;
    void assign(int i) { if (i&~077) throw Bad_range(); v=i; }
public:
    class Bad_range { };

    Tiny(int i) { assign(i); }
    Tiny& operator=(int i) { assign(i); return *this; }

    operator int() const { return v; }      // 转换成 int 的函数
};
```

用 int 初始化 Tiny 或者给它赋值时进行越界（溢出，包括上溢和下溢）检查。相反，当拷贝 Tiny 的时候无须检查是否越界，因此默认的拷贝构造函数和赋值运算是正确的，无须修改。

为了让 Tiny 可以使用 int 的常规操作，我们定义了从 Tiny 向 int 的隐式类型转换 Tiny::operator int()。请注意，目标类型已经作为运算符名字的一部分出现，因此不必再重复出现在转换函数返回值的位置了：

```
Tiny::operator int() const { return v; }          // 正确
int Tiny::operator int() const { return v; }      // 错误
```

从这层意义上来说，类型转换运算符类似于构造函数。

如果在需要 int 的地方出现了 Tiny，它会自动转换为对应的 int 值。例如：

```
int main()
```

```
{
    Tiny c1 = 2;
    Tiny c2 = 62;
    Tiny c3 = c2-c1;    //c3 = 60
    Tiny c4 = c3;       // 未执行越界检查（没必要）
    int i = c1+c2;      //i = 64

    c1 = c1+c2;         // 越界错误：c1 的值不能取 64
    i = c3-64;          //i = -4
    c2 = c3-64;         // 越界错误：c2 的值不能取 -4
    c3 = c4;            // 未执行越界检查（没必要）
}
```

当数据结构的读取操作（实现为类型转换运算符）不多，而赋值和初始化操作明显更加重要的时候，类型转换函数显得比较有用。

要想使下面的语句成立，istream 和 ostream 需要一个类型转换函数：

```
while (cin>>x)
    cout<<x;
```

输入操作 cin>>x 返回的是 istream&，它被隐式地转换成一个表示 cin 状态的值。我们可以在 while 语句中检验这个值（见 38.4.4 节）。然而，在上述转换过程中丢失了部分信息，因此我们最好不要定义类似的隐式类型转换。

通常情况下，最好尽量避免使用类型转换运算符。一旦过度使用的话可能引起程序的二义性。编译器可以捕获这类二义性的问题，但是很难完美解析。可能最优解决方案是一开始就用像 X::make_int() 一样的具名函数进行类型转换。只有当你觉得显式类型转换函数出现次数太多以至于影响程序的流畅和优雅时，才考虑把它替换成类型转换运算符 X::operator int()。

如果既有用户自定义的类型转换函数，又有用户自定义的运算符，则这二者间可能产生二义性。例如：

```
int operator+(Tiny,Tiny);

void f(Tiny t, int i)
{
    t+i;  // 错误，二义性问题："operator+(t,Tiny(i))" 还是 "int(t)+i" ？
}
```

因此，对于某种给定的数据类型，最好不要同时提供用户自定义的类型转换和用户自定义的运算符，在它们之间选择一种即可。

18.4.2 explicit 类型转换运算符

类型转换运算符也许能用在代码的任何地方。然而，最优的选择是把类型转换运算符声明成 explicit 并且明确只有当直接初始化时才使用它（见 16.2.6 节）。当然我们也可以在此处使用等价的 explicit 构造函数。例如，标准库 unique_ptr（见 5.2.1 节和 34.3.1 节）含有一个转换目标为 bool 的转换运算符：

```
template <typename T, typename D = default_delete<T>>
class unique_ptr {
public:
    // ...
    explicit operator bool() const noexcept;      //*this 含有某个指针（它不是 nullptr）吗？
    // ...
};
```

我们之所以把这个类型转换运算符声明成 explicit 是因为它可能出现在某些意想不到的上下文中。考虑如下情况：

```
void use(unique_ptr<Record> p, unique_ptr<int> q)
{
    if (!p)             // OK: 我们希望如此
        throw Invalid_uninque_ptr{};

    bool b = p;     //错误：并非我们所愿
    int x = p+q;    //错误：我们肯定不想如此
}
```

假设 unique_ptr 向 bool 的转换不是 explicit 的，则后两条定义语句将会编译通过。b 的值会变成 true，x 的值会变成 1 或者 2（由 q 是否有效决定）。这显然是我们不愿意看到的。

18.4.3　二义性

如果类 X 定义了一个赋值运算符 X::operator=(Z)，且类型 V 就是类型 Z 或者存在从 V 向 Z 的唯一类型转换，那么用 V 的值给 X 的对象赋值就是合法的。初始化的情况与之类似。

在某些情况下，目标类型的值需要通过多次重复使用构造函数或者类型转换运算符来构造。此时，我们必须使用显式类型转换。只有同一个层级内的用户自定义隐式类型转换才是合法的。反之，如果目标类型的值可以通过多种不同的方式构建，这样的代码就是非法的。例如：

```
class X { /* ... */ X(int); X(const char*); };
class Y { /* ... */ Y(int); };
class Z { /* ... */ Z(X); };

X f(X);
Y f(Y);

Z g(Z);

void k1()
{
    f(1);               // 错误，存在二义性：f(X(1)) 还是 f(Y(1))?
    f(X{1});            // OK
    f(Y{1});            // OK

    g("Mack");          // 错误：需要用到两个用户自定义类型转换，并没有如期望那样调用 g(Z{X{"Mack"}})
    g(X{"Doc"});        // OK: g(Z{X{"Doc"}})
    g(Z{"Suzy"});       // OK: g(Z{X{"Suzy"}})
}
```

只有当某次调用不得不通过用户自定义的类型转换才能解析时（即，仅靠内置类型转换不足以完成任务），系统才会这么做。例如：

```
class XX { /* ... */ XX(int); };

void h(double);
void h(XX);

void k2()
{
    h(1); // h(double{1}) 还是 h(XX{1})? h(double{1})!
}
```

函数调用 h(1) 实际执行的是 h(double(1))，原因是 h(double) 仅需使用标准类型转换，而 h(XX) 需要用到用户自定义的类型转换（见 12.3 节）。

C++ 选择或者执行类型转换的原则既不是最易于实现，也不是最易于记录，更不是最具有通用性。真正考虑的因素是该种类型转换必须足够安全，并且转换的结果最符合常理。靠人力解析二义性并不难，难的是一旦某些看起来无害的转换引发了错误，该如何查找并定位它。

因为我们采用的是严格的自底向上分析技术，所以返回值类型不能用于重载解析。例如：

```
class Quad {
public:
    Quad(double);
    // ...
};

Quad operator+(Quad,Quad);

void f(double a1, double a2)
{
    Quad r1 = a1+a2;          // 双精度浮点数加法
    Quad r2 = Quad(a1)+a2; // 强制执行自定义的运算
}
```

我们之所以像这样设计代码主要是因为严格的自底向上的分析技术更便于理解，况且猜测程序员到底想在哪种精度下执行加法不属于编译器的职责范畴。

初始化或者赋值操作两侧的数据类型一经确定，它们就会共同被用来进行解析。例如：

```
class Real {
public:
    operator double();
    operator int();
    // ...
};

void g(Real a)
{
    double d = a;   // d = a.double();
    int i = a;      // i = a.int();

    d = a;          // d = a.double();
    i = a;          // i = a.int();
}
```

在上述示例中，类型分析的顺序仍然是自底向上的，只不过同一时刻只考虑一个运算符及其参数类型。

18.5　建议

[1] 定义运算符时应该尽量模仿传统用法；18.1 节。

[2] 如果默认的拷贝操作对于某种类型不适用，应该重新定义或者干脆禁用；18.2.2 节。

[3] 对于较大的运算对象，选用 const 引用类型；18.2.4 节。

[4] 对于较大的返回结果，选用移动构造函数；18.2.4 节。

[5] 对于需要访问类的表示部分的操作，优先将其定义为成员函数；18.3.1 节。

[6] 反之，对于无须访问类的表示部分的操作，优先将其定义为非成员函数；18.3.2 节。

[7] 用名字空间把辅助函数和“它们的”类结合在一起；18.2.5 节。
[8] 把对称的运算符定义成非成员函数；18.3.2 节。
[9] 把需要左值作为其左侧运算对象的运算符定义为成员函数；18.3.3.1 节。
[10] 用用户自定义的字面值常量模仿传统用法；18.3.4 节。
[11] 不要轻易为数据成员提供“set() 和 get() 函数”，除非从语义上确实需要它们；18.3.5 节。
[12] 谨慎使用隐式类型转换；18.4 节。
[13] 避免使用丢失部分信息（“窄化”）的类型转换；18.4.1 节。
[14] 对于同一种类型转换，切勿把它同时定义成构造函数以及类型转换运算符；18.4.3 节。

特殊运算符

我们生而独特，并非凡人。

——阿尔贝·加缪

- 引言
- 特殊运算符
  取下标；函数调用；解引用；递增和递减；分配和释放；用户自定义字面值常量
- 字符串类
  必备操作；访问字符；类的表示；成员函数；辅助函数；应用 String
- 友元
  发现友元；友元与成员
- 建议

19.1 引言

重载并非只发生在算术运算和逻辑运算的范畴。事实上，运算符是容器（比如 vector 和 map，见 4.4 节）、“智能指针”（比如 unique_ptr 和 shared_ptr，见 5.2.1 节）、迭代器（见 4.5 节）以及其他承担资源管理功能的类的设计关键。

19.2 特殊运算符

下列运算符

```
[]  ()  ->  ++  --  new  delete
```

与 +、<、~ 等传统的一元或者二元运算符（见 18.2.3 节）相比有其特殊之处，主要是从这些运算符在代码中的使用到程序员给出的定义的映射与传统运算符有轻微的差别。其中，[]（取下标）和 ()（函数调用）是两种最重要的用户自定义运算符。

19.2.1 取下标

我们可以用 operator[] 函数为类对象的下标赋予某种新的含义。operator[] 函数的第 2 个参数（下标）可以是任意类型的，因此，它常被用于定义 vector、关联数组等类型。

举个例子，我们可以像下面这样定义一个简单的关联数组：

```
struct Assoc {
    vector<pair<string,int>> vec; // vector 的元素是 { 名字，值 } 对

    const int& operator[] (const string&) const;
    int& operator[](const string&);
};
```

Assoc 可存放 std::pair 的向量，它的实现用到了与 7.7 节类似的比较低效的简单搜索方法：

```
int& Assoc::operator[](const string& s)
    // 查找 s，如果找到，返回它的引用；
    // 否则创建一个新 pair {s,0}，并返回它的引用
{
    for (auto x : vec)
        if (s == x.first) return x.second;

    vec.push_back({s,0});          // 初始值：0

    return vec.back().second;      // 返回最后一个元素（见 31.2.2 节）
}
```

Assoc 的用法如下所示：

```
int main()      // 统计输入数据中每个单词出现的次数
{
    Assoc values;
    string buf;
    while (cin>>buf) ++values[buf];
    for (auto x : values.vec)
        cout << '{' << x.first << ',' << x.second << "}\n";
}
```

标准库 map 和 unordered_map 继承了关联数组的思想（见 4.4.3 节和 31.4.3 节），在此基础上进一步发展，并且融合了更高级的实现技术。

operator[]() 必须是非 static 成员函数。

19.2.2 函数调用

函数调用 *expression* (*expression-list*) 可以看成是一个二元运算，它的左侧运算对象是 *expression*，右侧运算对象是 *expression-list*。调用运算符 () 可以像其他运算符一样被重载。例如：

```
struct Action {
    int operator()(int);
    pair<int,int> operator()(int,int);
    double operator()(double);
    // ...
};

void f(Action act)
{
    int x = act(2);
    auto y = act(3,4);
    double z = act(2.3);
    // ...
};
```

operator()() 的参数列表按照常规的参数传递规则进行求值和检查。重载函数调用运算符对于只包含一个操作的数据类型以及有一个主导操作的数据类型来说尤其有用。调用运算符（call operator）又称为应用运算符（application operator）。

运算符 () 最直接也是最重要的目标是为某些行为类似函数的对象提供函数调用语法。其中，行为模式与函数类似的对象称为*类函数对象*（function-like object）或者简称为*函数对象*（function object，见 3.4.3 节）。函数对象使得我们可以接受某些特殊操作为参数。在很

多情况下，函数对象必须保存执行其操作所需的数据。例如，我们定义一个含有 operator()() 的类，它负责把一个预存的值加到它的参数上：

```
class Add {
    complex val;
public:
    Add(complex c) :val{c} { }                          // 保存值
    Add(double r, double i) :val{{r,i}} { }

    void operator()(complex& c) const { c += val; }     // 把值加到参数上
};
```

我们用一个复数数字初始化类 Add 的对象，当通过 () 调用时，它会把那个数字加到参数中去。例如：

```
void h(vector<complex>& vec, list<complex>& lst, complex z)
{
    for_each(vec.begin(),vec.end(),Add{2,3});
    for_each(lst.begin(),lst.end(),Add{z});
}
```

上述代码把 complex{2,3} 加到 vector 的每个元素中，把 z 加到 list 的每个元素中。请注意，Add{z} 创建了一个对象，它被 for_each() 重复地使用：序列中的每个元素都会调用一次 Add{z} 的 operator()()。

上述过程之所以可以正常执行，最根本的原因是 for_each 是个模板，它对第 3 个参数使用 ()，并且毫不在意该参数的内容到底是什么：

```
template<typename Iter, typename Fct>
Fct for_each(Iter b, Iter e, Fct f)
{
    while (b != e) f(*b++);
    return f;
}
```

第一眼看上去这项技术有些不易理解，但其实它简单、高效，非常有用（见 3.4.3 节和 33.4 节）。

请注意，lambda 表达式（见 3.4.3 节和 11.4 节）本质上是定义函数对象的一种方式。例如，我们可以编写如下代码：

```
void h2(vector<complex>& vec, list<complex>& lst, complex z)
{
    for_each(vec.begin(),vec.end(),[](complex& a){ a+={2,3}; });
    for_each(lst.begin(),lst.end(),[](complex& a){ a+=z; });
}
```

在此例中，每个 lambda 表达式都会生成等价的函数对象 Add。

operator()() 还可以用作字符串运算符以及多维数组的取下标运算符（见 29.2.2 节和 40.5.2 节）。

operator()() 必须是非 static 成员函数。

函数调用运算符通常是模板（见 29.2.2 节和 33.5.3 节）。

19.2.3　解引用

解引用运算符 ->（也称为箭头运算符）可以定义成一个一元后置运算符，例如：

```
class Ptr {
    // ...
    X* operator->();
};
```

类 Ptr 的对象可用来访问类 X 的成员，其用法与指针非常相似。例如：

```
void f(Ptr p)
{
    p->m = 7;       // (p.operator->())->m = 7
}
```

对象 p 到指针 p.operator->() 的转换与其所指的成员 m 无关，这正是 operator->() 是一元后置运算符的意义。然而我们并未引入任何新的语法，所以 -> 之后仍然需要一个成员的名字。例如：

```
void g(Ptr p)
{
    X* q1 = p->;                // 语法错误
    X* q2 = p.operator->();     // OK
}
```

重载 -> 的主要目的是创建“智能指针”，即，行为与指针类似的对象，并且当用其访问对象时执行某些操作。标准库“智能指针”unique_ptr 和 shared_ptr（见 5.2.1 节）提供了运算符 ->。

举个例子，我们定义一个访问磁盘对象的类，并将其命名为 Disk_ptr。Disk_ptr 的构造函数接受一个名字，我们通过它查找磁盘上的对象。Disk_ptr::operator->() 负责把对象导入内存，通过 Disk-ptr 访问，Disk_ptr 的析构函数则在最后把更新过的对象写回磁盘：

```
template<typename T>
class Disk_ptr {
    string identifier;
    T* in_core_address;
    // ...
public:
    Disk_ptr(const string& s) : identifier{s}, in_core_address{nullptr} { }
    ~Disk_ptr() { write_to_disk(in_core_address,identifier); }

    T* operator->()
    {
        if (in_core_address == nullptr)
            in_core_address = read_from_disk(identifier);
        return in_core_address;
    }
};
```

Disk_ptr 的用法如下所示：

```
struct Rec {
    string name;
    // ...
};

void update(const string& s)
{
    Disk_ptr<Rec> p {s};        // 为 s 获取 Disk_ptr

    p->name = "Roscoe";         // 更新 s，如有必要，先从磁盘上查找
    // ...
}                               // p 的析构函数负责写回磁盘
```

显然，实际的程序还应该考虑加入异常处理代码，并且选用更高级的方法与磁盘交互。

对于普通的指针来说，使用 -> 和使用 * 及 [] 差不多。已知在类 Y 中 ->、* 和 [] 均有其默认的含义，p 的类型是 Y*，则有：

```
p->m == (*p).m          // 值为 true
(*p).m == p[0].m        // 值为 true
p->m == p[0].m          // 值为 true
```

像往常一样，该规则对于用户自定义的运算符无效。如果确实需要的话，可以人为指定：

```
template<typename T>
class Ptr {
    Y* p;
public:
    Y* operator->() { return p; }          // 解引用以访问成员
    Y& operator*() { return *p; }          // 解引用以访问整个对象
    Y& operator[](int i) { return p[i]; }  // 解引用以访问元素
    // ...
};
```

如果你从上述运算符中选取多个提供给了当前的类，那么最好把等价的操作也实现出来。这一点与某些简单的运算符很相似。比如，如果类 X 实现了 ++、+=、= 和 + 等运算，那么我们应该确保 ++x 和 x+=1 与 x=x+1 的执行效果一致。

重载 -> 绝不仅仅是一个小小的语法点，它对于某些程序的类来说非常重要。众所周知，间接引用（indirection）是 C++ 的一个关键概念，重载 -> 为我们的程序提供了一种简单、直接、高效地表示间接引用的途径。具体示例请参考第 33 章中与迭代器有关的内容。

运算符 -> 必须是非 static 成员函数。此外，它的返回类型必须是指针或者类的对象，我们需要把 -> 作用于它们。因为只有当我们使用模板类成员函数的时候才会检查它的函数体（见 26.2.1 节），所以在定义 operator->() 时无须在意数据类型。举个例子，虽然 -> 作用于 Ptr<int> 没什么实际意义，但是也不会影响我们的定义。

尽管 -> 和 .（点运算符）非常相似，但是我们并不能重载点运算符。

19.2.4 递增和递减

有了“智能指针”之后，接下来就要提供与内置数据类型类似的递增运算符 ++ 和递减运算符 -- 了。与普通指针相比，“智能指针”的语义类似，但是增加了一些运行时错误检查的功能。此时，++ 和 -- 显得尤为重要。举个例子，下面是一段传统代码，其中存在某些错误：

```
void f1(X a)            // 传统用法
{
    X v[200];
    X* p = &v[0];
    p--;
    *p = a;    // 哎呀：p 越界了，但是无法捕获该异常
    ++p;
    *p = a;    // 正确
}
```

在这段代码中最好用类 Ptr<X> 对象取代 X*，只有当前者真的指向一个 X 时它才会被解引用。另外，我还需要确保只有当 p 指向数组中的对象，并且当递增或者递减后所指的对象仍然位于数组中时，才对 p 执行递增 / 递减操作。因此，改进后的程序是：

```
void f2(Ptr<X> a)          // 进行范围检查的版本
{
    X v[200];
    Ptr<X> p(&v[0],v);
    p--;
    *p = a;    // 运行时错误：p 越界
    ++p;
    *p = a;    // 正确
}
```

在 C++ 的所有运算符中，递增运算符和递减运算符是最特别的，因为它们既可以作为前置运算符，也可以作为后置运算符。因此，我们必须为 Ptr<T> 分别定义递增 / 递减运算符的前置和后置版本。例如：

```
template<typename T>
class Ptr {
    T* ptr;
    T* array;
    int sz;
public:
    template<int N>
        Ptr(T* p, T(&a)[N]);        // 绑定到数组 a，sz==N，初始值是 p
    Ptr(T* p, T* a, int s);         // 绑定到数组 a，数组的大小是 s，初始值是 p
    Ptr(T* p);                      // 绑定到单个对象，sz==0，初始值是 p

    Ptr& operator++();              // 前置
    Ptr operator++(int);            // 后置

    Ptr& operator--();              // 前置
    Ptr operator--(int);            // 后置

    T& operator*();                 // 前置
};
```

其中，int 参数表示该函数以后置方式被调用。这个 int 并不会被使用，它只起到区分前置和后置版本的作用。辨别 operator++ 前置版本的方法是在前置版本中没有哑参数，这一点与其他一元的算术运算符和逻辑运算符类似。哑参数只用于表示“奇怪的”后置的 ++ 和 --。

在设计中可以考虑去掉后置的 ++ 和 --。与前置版本的运算符相比，后置版本不仅在语法上有些奇怪，而且还有其他一些缺点，比如较难实现、低效以及应用范围较小等。例如：

```
template<typename T>
Ptr& Ptr<T>::operator++()          // 递增后返回当前对象
{
    // ... 检查 ptr+1 是否有效 ...
    return *++ptr;
}
template<typename T>
Ptr Ptr<T>::operator++(int)        // 递增并返回 Ptr 的旧值
{
    // ... 检查 ptr+1 是否有效 ...
    Ptr<T> old {ptr,array,sz};
    ++ptr;
    return old;
}
```

前置递增运算符返回对象的引用，后置递增运算符返回一个新创建的对象。

使用 Ptr，上面的代码等价于：

```
void f3(T a)          // 进行范围检查的版本
{
    T v[200];
    Ptr<T> p(&v[0],v,200);
    p.operator--(0);          // 后置运算符 p--
    p.operator*() = a;        // 运行时错误：p 越界
    p.operator++();           // 前置运算符 ++p
    p.operator*() = a;        // OK
}
```

我们把 Ptr 类的完整实现留作练习，27.2.2 节介绍了一个可以表现继承关系的指针模板。

19.2.5 分配和释放

运算符 new（见 11.2.3 节）通过调用 operator new() 分配内存。相应地，运算符 delete 通过调用 operator delete() 释放内存。用户可以重新定义全局的 operator new() 和 operator delete()，也可以为特定的类定义 operator new() 和 operator delete()。

全局版本的 operator new() 和 operator delete() 如下所示，其中我们用到了标准库类型别名 size_t（见 6.2.8 节）：

```
void* operator new(size_t);               // 用于单个对象
void* operator new[](size_t);             // 用于数组
void operator delete(void*, size_t);      // 用于单个对象
void operator delete[](void*, size_t);    // 用于数组

// 更多版本请参见 11.2.4 节
```

当 new 需要在自由存储上为类型 X 的对象分配内存空间时，它调用 operator new(sizeof(X))。类似地，当 new 需要在自由存储上为包含 N 个 X 对象的数组分配内存空间时，它调用 operator new[](N*sizeof(X))。new 表达式实际分配的空间也许比 N*sizeof(X) 多一些，超出的部分可以容纳若干字符（即，超出若干字节）。除非你的技术足够精湛，否则不建议改写全局的 operator new() 和 operator delete()。毕竟，其他人也许用到了系统默认提供的版本，或者也为这些函数提供了他们自己的版本。

更好的做法是为特定的类提供这些操作。其中，这个特定的类可以作为很多派生类的基类。例如，我们想要类 Employee 为它自己以及它的派生类提供一对特定的分配和释放函数：

```
class Employee {
public:
    // ...

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

    void* operator new[](size_t);
    void operator delete[](void*, size_t);
};
```

成员函数 operator new() 和 operator delete() 是隐式的 static 成员。因此，它们无法使用 this 指针，也不能修改对象的值。它们提供了一块可供构造函数初始化并由析构函数释放的存储空间。

```
void* Employee::operator new(size_t s)
{
    // 分配 s 个字节的内存空间，返回指向该区域的指针
}

void Employee::operator delete(void* p, size_t s)
{
    if (p) {    // 仅当 p!=0 时释放，见 11.2 节和 11.2.3 节
        // 假定 p 指向 Employee::operator new() 分配的 s 个字节的内存空间，
        // 释放该空间以供重新使用
    }
}
```

此时，我们终于揭开了 size_t 参数的神秘面纱，它表示的是被 delete 对象的大小。删除一个“普通的” Employee 赋予参数的值是 sizeof(Employee)，如果 Employee 的派生类 Manager 本身没定义 operator delete()，则删除 Manager 对应的参数值是 sizeof(Manager)。这种机制使得类相关的分配函数无须保存每次分配的尺寸信息。反过来说，类相关的分配函数当然也可以保存该信息（通用的分配函数必须如此），并且忽略掉 operator delete() 的 size_t 参数。但是这么做的话我们就很难在通用分配函数的基础上提升速度并减少内存消耗，因此也就体现不出类相关的分配函数的优势了。

编译器该如何获知提供给 operator delete() 的正确尺寸是多大呢？ delete 操作中指定的类型需要与实际 delete 的对象的类型匹配。如果我们用基类指针 delete 一个对象，则基类必须包含一个 virtual 析构函数（见 17.2.5 节）：

```
Employee* p = new Manager; // 存在潜在风险（确切类型未知）
// ...
delete p;                  // Employee 应该含有 virtual 析构函数
```

一般来说，释放操作由析构函数（知道类的尺寸）负责完成。

19.2.6　用户自定义字面值常量

C++ 为内置数据类型提供了字面值常量（见 6.2.6 节）：

```
123         // int
1.2         // double
1.2F        // float
'a'         // char
1ULL        // unsigned long long
0xD0        // 十六进制 unsigned
"as"        // C 风格字符串（const char[3]）
```

我们也能为用户自定义类型提供字面值常量，或者更新内置类型字面值常量的形式。例如：

```
"Hi!"s                  // 字符串，并非“以 0 结尾的字符数组”
1.2i                    // 虚数
1010101111000101b       // 二进制数
123s                    // 秒数
123.56km                // 注意此处并非 miles!（单位）
12345678901234567890123456789012345678901234567890x  // 扩展精度数字
```

上述的用户自定义字面值常量（user-defined literal）是通过字面值常量运算符（literal operator）定义的，这类运算符负责把带后缀的字面值常量映射到目标类型。字面值常量运算符的名字由 operator"" 加上后缀组成，例如：

```
constexpr complex<double> operator"" i(long double d)      // 虚数字面值常量
{
    return {0,d};   // 复数是一种字面值常量类型
}

std::string operator"" s(const char* p, size_t n)    // std::string 字面值常量
{
    return string{p,n};   // 需要分配自由存储空间
}
```

这两个运算符分别定义了后缀 i 和 s。我用 constexpr 确保在编译时求值，在此基础上，我们可以编写如下代码：

```
template<typename T> void f(const T&);

void g()
{
    f("Hello");      // 传入 char* 指针
    f("Hello"s);     // 传入包含 5 个字符的字符串对象
    f("Hello\n"s);   // 传入包含 6 个字符的字符串对象

    auto z = 2+1i;   // complex{2,1}
}
```

基本的实现思想是，编译器先解析字面值常量部分，然后检查后缀。用户自定义字面值常量机制允许用户指定新后缀以及对字面值常量可做的操作。C++ 不允许更改内置字面值常量后缀的含义，也不允许更改字面值常量的语法结构。

我们可以在四种字面值常量之后添加后缀以构成用户自定义字面值常量（§ iso.2.14.8）：

- 整型字面值常量（见 6.2.4.1 节）：可用于接受 unsigned long long 或者 const char* 参数的字面值常量运算符，也可用于模板字面值常量运算符。例如，123m 和 12345678901234567890X 属于这种情况。
- 浮点型字面值常量（见 6.2.5.1 节）：可用于接受 long double 或者 const char* 参数的字面值常量运算符，也可用于模板字面值常量运算符。例如，1234567890123 4567890.976543210x 和 3.99s 属于这种情况。
- 字符串字面值常量（见 7.3.2 节）：可用于接受 (const char*,size_t) 参数对的字面值常量运算符。例如，"string"s 和 R"(Foo\bar)"_path 属于这种情况。
- 字符字面值常量（见 6.2.3.2 节）：可用于接受 char、wchar_t、char16_t、char32_t 等字符类型参数的字面值常量运算符。例如，'f'_runic 和 u'BEEF'_w 属于这种情况。

例如，我们定义一种字面值常量运算符，用它存放任意内置整数类型都无法表示的整型值数字：

```
Bignum operator"" x(const char* p)
{
    return Bignum(p);
}

void f(Bignum);

f(12345678901234567890123456789012345678901234567890x);
```

此处的 C 风格字符串 "12345678901234567890123456789012345678901234567890" 作为参数

被传递给 operator""x()。请注意，在代码中我并未在该数字串的两端使用双引号。我们的运算符接受的是 C 风格字符串，由编译器负责把数字转换成所需的类型。

为了从传入字面值常量运算符的程序源文件文本中得到 C 风格的字符串，我们需要用到字符串内容以及字符的数量。例如：

```
string operator"" s(const char* p, size_t n);

string s12 = "one two"s;        // 调用 operator ""s("one two",7)
string s22 = "two\ntwo"s;       // 调用 operator ""s("two\ntwo",7)
string sxx = R"(two\ntwo)"s;    // 调用 operator ""s("two\\ntwo",8)
```

在原始字符串（见 7.3.2.1 节）中，"\n" 代表两个字符 '\' 和 'n'。

此外，之所以需要用到传入字符的数量是基于这样一种假设，即，既然我们想得到“另外一种字符串形式”，那么肯定也想知道字符数到底是多少。

只接受一个 const char* 参数（而无须尺寸信息）的字面值常量运算符既能作用于整型字面值常量，也能作用于浮点型字面值常量。例如：

```
string operator"" SS(const char* p);          // 警告：无法如预期一般

string s12 = "one two"SS;                     // 错误：无可用的字面值常量运算符
string s13 = 13SS;                            // 但是这么做的意义何在？
```

把数字强行转换成字符串的做法实在没什么意义，让人摸不着头脑。

模板字面值常量运算符（template literal operator）将其参数作为模板参数包而非函数参数。例如：

```
template<char...>
constexpr int operator"" _b3();           // 三进制
```

基于该定义，我们可以得到：

```
201_b3   // 意思是 operator"" b3<'2','0','1'>(); 对应的实体数是 2*9+0*3+1 — 19
241_b3   // 意思是 operator"" b3<'2','4','1'>(); 错误：4 不应出现在三进制数中
```

可变参数模板技术（见 28.6 节）可能令人困扰，但它是在编译时将非标准含义赋予数字的唯一方法。

要想定义 operator""_b3()，我们需要先定义几个辅助函数：

```
constexpr int ipow(int x, int n)// 对 n >= 0 求 x 的 n 次方
{
    return (n>0) ? x*ipow(n-1) : 1;
}

template<char c>          // 处理一个三进制数字
constexpr int b3_helper()
{
    static_assert(c<'3',"not a ternary digit");
    return c;
}

template<char c, char... tail>  // 剥离一个数字
constexpr int b3_helper()
{
    static_assert(c<'3',"not a ternary digit");
    return ipow(3,sizeof...(tail))*(c-'0')+b3_helper(tail...);
}
```

基于上述条件，我们就能定义三进制字面值常量运算符了：

```
template<char... chars>
constexpr int operator"" _b3()      // 三进制
{
    return b3_helper(chars...);
}
```

后缀一般都比较简短（例如我们用 s 表示 std::string，i 表示虚部，m 表示米（见 28.7.3 节），x 表示扩展的），因此有可能会产生冲突。应该使用名字空间避免此类冲突：

```
namespace Numerics {
    // ...

    class Bignum { /* ... */};

    namespace literals {
        Bignum operator"" x(char const*);
    }
    // ...
}

using namespace Numerics::literals;
```

不以下划线开始的后缀都是为标准库预留的，因此程序员最好把自己的后缀设计为以下划线开始，否则未来有可能因标准库的扩充而失效：

```
123km       // 标准库预留
123_km      // 程序员可用
```

19.3　字符串类

C++ 提供了几种关键技术以便于程序员设计并实现那些需要重定义传统运算符的类，本节展示的这个简单的字符串类有助于读者学习此类技术。这里的 String 是标准库 string（见 4.2 节，第 36 章）的简化版本。String 提供了值的语义、对于字符的经过检查的 / 未经检查的访问、输入输出流、对范围 for 循环的支持、相等性运算以及连接运算。此外，我还提供了 String 字面值常量，而标准库 std::string 并不具备这一功能。

为了使 String 与 C 风格字符串具有互通性（包括字符串字面值常量，见 7.3.2 节），我把字符串表示为以 0 结尾的字符数组。出于实用的目的，我实现了短字符串优化（short string optimization）。短字符串优化的意思是直接把字符数很少的 String 用对象本身保存，而非置于自由存储上。这样做可以极大地提升短字符串的使用效率。经验显示，绝大多数应用程序用到的字符串的长度都很短。这一优化措施在多线程系统中尤其有用，因为在该类系统中共享指针或者引用很容易，而在自由存储上执行分配和释放的操作则代价昂贵。

为了使 String 可以通过在尾端添加字符的方式实现“生长”，我借鉴 vector 的模式（见 13.6.1）为 String 预留了一些额外空间。这样，String 就可以适应不同形式的输入了。

在真正使用 std::string（第 36 章）并发你的程序之前，不妨先尝试完成一个自己的字符串类或者实现一些自己想到的有用功能，这会是一种很好的编程体验。

19.3.1　必备操作

类 String 提供了构造函数、析构函数以及赋值操作（见 17.1 节）：

```
class String {
public:
    String();                                   // 默认构造函数：x{""}

    explicit String(const char* p);             // 接受 C 风格字符串的构造函数：x{"Euler"}

    String(const String&);                      // 拷贝构造函数
    String& operator=(const String&);           // 拷贝赋值
    String(String&& x);                         // 移动构造函数
    String& operator=(String&& x);              // 移动赋值

    ~String() { if (short_max<sz) delete[] ptr; }   // 析构函数

    // ...
};
```

这个 String 具有值语义。也就是说，执行赋值运算 s1=s2 之后，s1 和 s2 完全是两个独立的字符串，接下来改变其中任何一个都不会影响另外一个。另一种选择是赋予 String 指针语义，即，当我们执行赋值运算 s1=s2 之后，对 s2 的改变也会影响 s1。当值语义有意义时，我倾向于提供值语义，比如 complex、vector、Matrix 和 string 都是如此。同时，为了减少值语义带来的性能开销，我们应该在不需要拷贝的时候采用引用的方式传递 String，并且实现移动语义（见 3.3.2 节和 17.5.2 节）来优化 return。

19.3.3 节将介绍 String 的表示，我们需要实现用户自定义的拷贝和移动操作。

19.3.2　访问字符

要想为字符串设计一套理想的字符访问机制并不容易，它得使用人们习惯的符号 []，还得尽可能优化性能并执行边界检查。我们很难实现全部目标，接下来，我参考标准库的方式提供了一组使用 [] 下标符号的未执行任何检查的操作以及一个执行边界检查的 at() 操作：

```
class String {
public:
    // ...

    char& operator[](int n) { return ptr[n]; }              // 未经检查的元素访问
    char operator[](int n) const { return ptr[n]; }

    char& at(int n) { check(n); return ptr[n]; }            // 执行边界检查的元素访问
    char at(int n) const { check(n); return ptr[n]; }

    String& operator+=(char c);                             // 在末尾添加 c

    const char* c_str() { return ptr; }                     // C 风格字符串的访问
    const char* c_str() const { return ptr; }

    int size() const { return sz; }                         // 元素数量
    int capacity() const                                    // 元素以及可用空间（总容量）
        { return (sz<=short_max) ? short_max : sz+space; }

    // ...
};
```

我们的初衷是用 [] 执行普通的访问操作，例如：

```
int hash(const String& s)
{
    int h {s[0]};
    for (int i {1}; i!=s.size(); i++) h ^= s[i]>>1;        // 未经检查的对 s 的访问
    return h;
}
```

在上面这段代码中，因为我们访问的 s 的范围仅限定在 0 到 s.size()-1 之间，所以不必使用执行边界检查的 at()。

相反，在有可能发生错误的地方使用 at()，例如：

```
void print_in_order(const String& s,const vector<int>& index)
{
    for (x : index) cout << s.at(x) << '\n';
}
```

不幸的是，我们无法保证人们记得在所有可能出错的地方使用 at()。因此有的 std::string 实现（从中借鉴了 []/at() 转换）也会对 [] 执行检查。我个人推荐在开发时使用经过检查的 []。当然，这么做会带来不小的性能开销。

我为访问函数同时提供了 const 和非 const 的版本，这样它们就能处理任意对象了。

19.3.3 类的表示

String 的表示应该满足三个目标：

- 易于把 C 风格的字符串（比如字符串字面值常量）转换成 String，并使得访问 String 的字符就像 C 风格字符串一样容易；
- 尽量减少对自由存储的使用；
- 向 String 末尾添加字符的操作足够高效。

因此，我们最后设计并使用的表示比 {指针，尺寸} 的形式复杂得多，同时也实用得多：

```
class String {
/*
    这是一种比较简单的字符串，它实现了短字符串优化

    size()==sz 是元素的个数
    如果 size()<= short_max 成立，则 String 对象自己保存字符；
    否则用自由存储保存。

    ptr 指向字符序列的起始处
    字符序列以 0 结尾：ptr[size()]==0;
    这样设计的目的是便于我们使用 C 标准库字符串函数以及返回 C 风格的字符串：

    c_str()
    为了便于在末尾添加字符，String 通常把它分配的空间扩充一些；
    capacity() 表示字符可用空间的总量（不包括结尾处的 0）：sz+space
*/
public:
    // ...
private:
    static const int short_max = 15;
    int sz;                         // 字符数量
    char* ptr;
    union {
        int space;                  // 未使用的已分配空间
```

```
        char ch[short_max+1];   // 为结尾的 0 预留空间
    };

    void check(int n) const         // 边界检查
    {
        if (n<0 || sz<=n)
            throw std::out_of_range("String::at()");
    }

    // 其他成员函数：
    void copy_from(const String& x);
    void move_from(String& x);
};
```

我们使用两种不同的表示形式以支持短字符串优化（short string optimization）：

- 如果 sz<=short_max，则 String 对象自己保存字符，存在 ch 数组中。
- 如果 !(sz<=short_max)，则把字符存放在自由存储上，分配一些额外的空间以便于扩展。此时，字符存在 space 中。

在上述两种情况下，我们都用 sz 表示元素的数量，并通过 sz 决定应该用哪种方式存储给定的字符串。

在上述两种情况下，ptr 都指向元素。这一点对于性能至关重要：访问函数无须检查当前使用的是哪一种存储方式，它只要直接使用 ptr 就可以了。只有构造函数、赋值操作、移动操作和析构函数（见 19.3.4 节）需要兼顾两种表示形式。

当 sz<=short_max 的时候我们使用 ch，而当 !(sz<=short_max) 的时候使用 space。因此，从节约空间的角度出发没必要同时为 ch 和 space 分配空间。我们使用 union（见 8.3 节）来避免这种浪费。实际上，我使用的是一种名为匿名联合（anonymous union）的 union（见 8.3.2 节），它的作用是供一个类在几种不同的表示形式间做出选择。匿名联合的全部成员都分配在同一块内存中，并且它们的地址全都相同。同一时刻我们只能使用其中的一个成员，但是从效果上看起来就好像它们是匿名联合的外层作用域中的独立成员一样。程序员需要确保不会误用这些成员。例如，String 的成员函数在使用 space 时必须确保当前状态下 space 有效而 ch 无效。我们通过检验 sz<=short_max 是否满足来实现这一目标。换句话说，String 也可以看成是一个联合，sz<=short_max 是它的判别式。

19.3.3.1　补充函数

除了人们会用到的这些函数之外，我还实现了 3 个作为“基础材料”的补充函数。它们帮助我处理某些微妙的表示问题，并且减少重复代码，从而使得整个代码的结构更加清晰明了。其中的两个需要访问 String 的表示部分，因此它们被设置为成员函数。与此同时，从安全性的角度出发我把它们设置为 private，毕竟它们提供的并非类的公开服务功能。对于很多类来说，程序员要实现的不仅仅是类的表示以及 public 函数。适当添加一些补充函数有助于减少重复代码、提高设计水准并且增加可维护性。

第一个补充函数负责把字符移动到新分配的内存中去：

```
char* expand(const char* ptr, int n)      // 扩展到自由存储
{
    char* p = new char[n];
    strcpy(p,ptr);              // 见 43.4 节
    return p;
}
```

该函数无须访问 String 的表示，因此我没有把它设置为成员函数。

第二个函数是为拷贝操作服务的，它负责把一个 String 的成员的副本提供给另一个 String：

```
void String::copy_from(const String& x)
    // 把 x 拷贝给 *this
{
    if (x.sz<=short_max) {                  // 拷贝 *this
        memcpy(this,&x,sizeof(x));          // 见 43.5 节
        ptr = ch;
    }
    else {                                  // 拷贝元素
        ptr = expand(x.ptr,x.sz+1);
        sz = x.sz;
        space = 0;
    }
}
```

目标 String 的清理工作由 copy_from() 的调用者负责，copy_from() 无条件地改写它的目标。我利用标准库 memcpy()（见 43.5 节）把源字节拷贝给目标。memcpy() 是一个非常底层的操作，因为它对于数据类型一无所知，因此我们必须确保在拷贝的内存中没有任何含有构造函数和析构函数的对象。String 的两个拷贝操作都用到了 copy_from()。

移动操作对应的函数是：

```
void String::move_from(String& x)
{
    if (x.sz<=short_max) {                  // 拷贝 *this
        memcpy(this,&x,sizeof(x));          // 见 43.5 节
        ptr = ch;
    }
    else {                          // 移动元素
        ptr = x.ptr;
        sz = x.sz;
        space = x.space;
        x.ptr = x.ch;               // x = ""
        x.sz = 0;
        x.ch[0]=0;
    }
}
```

它同样无条件地令它的目标成为参数的一份拷贝。不同的是，操作完成后它的参数不再拥有任何自由存储空间。我当然可以在长字符串的情况下使用 memcpy()，但既然长字符串表示只是 String 表示的一部分，那么分别拷贝各个部分也未尝不可。

19.3.4 成员函数

默认的构造函数定义了一个空 String：

```
String::String()                    // 默认构造函数：x{""}
    : sz{0}, ptr{ch}                // ptr 指向元素，ch 是初始位置（见 19.3.3 节）
{
    ch[0] = 0;                      // 以 0 结尾
}
```

如果已经定义了 copy_from() 和 move_from()，那么构造函数、移动运算和赋值运算就变得非常容易实现。接受 C 风格字符串作为参数的构造函数必须计算字符的数量以便正确存储

这些字符：

```
String::String(const char* p)
    :sz{strlen(p)},
     ptr{(sz<=short_max) ? ch : new char[sz+1]},
     space{0}
{
    strcpy(ptr,p);  // 把字符从 p 拷贝给 ptr
}
```

如果参数是一个短字符串，则 ptr 指向 ch；否则，在自由存储上分配 space。不管在哪种情况下，参数字符串中的字符最终都被拷贝到由 String 管理的内存中。

拷贝构造函数只拷贝其参数的表示：

```
String::String(const String& x)      // 拷贝构造函数
{
    copy_from(x); // 从 x 中拷贝其表示部分
}
```

我不打算对源字符串和目标字符串等长的情况进行优化（像针对 vector 做的那样，见 13.6.3 节），因为我不知道这么做是否值得。

类似地，移动构造函数从它的源字符串中移动表示（并且可能会把它的参数设成空串）：

```
String::String(String&& x)      // 移动构造函数
{
    move_from(x);
}
```

类似于拷贝构造函数，拷贝赋值运算也使用 copy_from() 克隆其参数的表示。此外，它还必须 delete 目标已拥有的自由存储并且确保不会陷入自赋值的麻烦中（比如 s=s）：

```
String& String::operator=(const String& x)
{
    if (this==&x) return *this;              // 处理自赋值
    char* p = (short_max<sz) ? ptr : 0;
    copy_from(x);
    delete[] p;
    return *this;
}
```

String 的移动赋值运算先删除它的目标的自由存储（如果有的话），然后执行移动操作：

```
String& String::operator=(String&& x)
{
    if (this==&x) return *this;              // 处理自赋值（x = move(x) 看起来不合常理）
    if (short_max<sz) delete[] ptr;          // 使用 delete 删除目标
    move_from(x);                            // 不抛出异常
    return *this;
}
```

从逻辑上来说，我们可以把源字符串移动到它自己的空间上（比如 s=std::move(s)）。但是这样的操作不合乎常理，因此我们必须防止自赋值的发生。

Sting 最复杂的操作是 +=，它把一个字符添加到当前字符串的末尾，同时把字符串的长度加 1：

```
String& String::operator+=(char c)
{
    if (sz==short_max) {      // 扩展到长字符串
```

```
            int n = sz+sz+2;        // 把空间扩充一倍（+2 的原因是字符串以 0 结尾）
            ptr = expand(ptr,n);
            space = n-sz-2;
        }
        else if (short_max<sz) {
            if (space==0) {             // 在自由存储上扩充
                int n = sz+sz+2;        // 把空间扩充一倍（+2 的原因是字符串以 0 结尾）
                char* p = expand(ptr,n);
                delete[] ptr;
                ptr = p;
                space = n-sz-2;
            }
            else
                --space;
        }
        ptr[sz] = c;            // 在末尾添加 c
        ptr[++sz] = 0;          // 增加长度并设置结束符

        return *this;
    }
```

这儿有很多事情值得注意：operator+=() 必须追踪我们使用的表示类型（长的还是短的）并且判断是否还有额外的空间可供字符串扩容。如果需要申请额外的空间，则调用 expand() 并将原来的字符移动到新的存储空间当中。如果需要释放原来分配的空间，并且 expand() 也把它返回了，+= 可以直接删掉它。一旦内存空间足够使用了，就可以把新字符 c 添加到字符串的尾部，然后再加上结束符 0。

计算 space 可用空间的过程值得我们注意。它在 String 的全部实现过程中需要我们给予最多关注：它极易发生差一错误（off-by-one errors），而且我们使用了好几处令人生厌的“魔数”2。

String 的所有成员都需谨防在新的表示就绪之前就做出任何修改。尤其是需要防止在完成 new 操作之前就先 delete。事实上，String 成员提供了强安全保障（见 13.2 节）。

如果你对 String 这种需要小心翼翼维护的代码不感兴趣，那么可以使用 std::string。标准库功能的意义在很大程度上就是令程序员可以远离底层操作。当然，亲自动手编写一个你自己的字符串类、向量类或者映射类都是很棒的编程体验。不过一旦完成了这类练习，你就会对标准库的价值有更清醒的认识，同时你也更愿意使用标准库而非自定义的类了。

19.3.5 辅助函数

为了完成 String 类，我继续提供一些有用的函数、输入输出流、对范围 for 循环的支持、比较操作以及连接操作。这些功能都借鉴了 std::string 的设计思想。其中，<< 仅负责输出字符，不附加任何格式信息；>> 忽略开头的空白信息，并以下一处空白符作为结束（或者到达流末尾）：

```
ostream& operator<<(ostream& os, const String& s)
{
    return os << s.c_str();     // 见 36.3.3 节
}

istream& operator>>(istream& is, String& s)
{
    s = "";     // 清空目标串
```

```
        is>>ws;  // 跳过空白（见 38.4.1.1 节和 38.4.5.2 节）
        char ch = ' ';
        while(is.get(ch) && !isspace(ch))
            s += ch;
        return is;
    }
```

我实现了两种比较操作 == 和 !=：

```
    bool operator==(const String& a, const String& b)
    {
        if (a.size()!=b.size())
            return false;
        for (int i = 0; i!=a.size(); ++i)
            if (a[i]!=b[i])
                return false;
        return true;
    }

    bool operator!=(const String& a, const String& b)
    {
        return !(a==b);
    }
```

至于增加 < 等操作其实非常简单，不再一一赘述。

要想提供对范围 for 循环的支持，我们需要先实现 begin() 和 end()（见 9.5.1 节）。和往常一样，因为它们无须直接访问 String 实现，所以实现为独立的非成员函数：

```
    char* begin(String& x)          // C 风格字符串的访问
    {
        return x.c_str();
    }

    char* end(String& x)
    {
        return x.c_str()+x.size();
    }

    const char* begin(const String& x)
    {
        return x.c_str();
    }

    const char* end(const String& x)
    {
        return x.c_str()+x.size();
    }
```

因为已经实现了在字符串末尾添加字符的成员函数 +=，所以我们很容易提供一个非成员函数执行连接操作：

```
    String& operator+=(String& a, const String& b)    // 连接
    {
        for (auto x : b)
            a+=x;
        return a;
    }
    String operator+(const String& a, const String& b)// 连接
    {
```

```
        String res (a);
        res += b;
        return res;
    }
```

我感觉自己可能在这里“耍了点小聪明”。我有必要提供向 C 风格字符串末尾添加字符的 += 吗？标准库 string 确实是这样做的，但是即使没有它，连续 C 风格字符串的操作也同样有效。例如：

```
String s = "Njal ";
s += "Gunnar";      // 连续：加到 s 的末尾
```

上述代码可以理解为 operator+=(s,String("Gunnar"))。我还可以实现一个效率更高的 String::operator+=(const char*)，但是这个版本未必在实际应用中真的有意义。在此例中我比较保守，希望表达最精简的设计就可以了。有时候你能做的事未必一定要做，适当舍弃也许是更明智的选择。

类似地，我也没有根据源字符串的尺寸优化 +=。

我们还可以添加 _s 作为字符串字面值常量的后缀：

```
String operator"" _s(const char* p, size_t)
{
    return String(p);
}
```

我们可以接着编写：

```
void f(const char*);        // C 风格的字符串
void f(const String&);      // 我们的字符串

void g()
{
    f("Madden's");              // f(const char*)
    f("Christopher's"_s);       // f(const String&);
}
```

19.3.6　应用 String

接下来的主函数初步练习了 String 的各种操作：

```
int main()
{
    String s ("abcdefghij");
    cout << s << '\n';
    s += 'k';
    s += 'l';
    s += 'm';
    s += 'n';
    cout << s << '\n';
    String s2 = "Hell";
    s2 += " and high water";
    cout << s2 << '\n';

    String s3 = "qwerty";
    s3 = s3;
    String s4 ="the quick brown fox jumped over the lazy dog";
    s4 = s4;
    cout << s3 << " " << s4 << "\n";
```

```
    cout << s + ". " + s3 + String(". ") + "Horsefeathers\n";

    String buf;
    while (cin>>buf && buf!="quit")
        cout << buf << " " << buf.size() << " " << buf.capacity() << '\n';
}
```

本节实现的 String 也许缺少很多你认为重要的功能。它的目的是尽可能模仿 std::string（第 36 章）并说明标准库 string 在实现中用到的各种技术。

19.4 友元

一条普通的成员函数声明语句在逻辑上包含相互独立的三层含义：

[1] 该函数有权访问类的私有成员。

[2] 该函数位于类的作用域之内。

[3] 我们必须用一个含有 this 指针的对象调用该函数。

通过把成员函数声明成 static 的（见 16.2.12 节），我们可以令它只具有前两层含义。通过把非成员函数声明成 friend 的，我们可以令它只具有第一层含义。换句话说，一个 friend 函数可以像成员函数一样访问类的实现，但是在其他层面上与类是完全独立的。

例如，我们可以定义一个计算 Matrix 与 Vector 乘积的运算符。Vector 和 Matrix 会隐藏它们各自的表示部分，并向外提供一组可操作其对象的函数。然而，我们的乘法运算不应是它们之中任何一个的成员，而且我们也不希望用户可以通过底层操作访问 Vector 和 Matrix 的完整表示。为了避免这样，我们可以把 operator* 声明成这两个类的 friend：

```
constexpr rc_max {4};    // 行列的尺寸

class Matrix;

class Vector {
    float v[rc_max];
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};
class Matrix {
    Vector v[rc_max];
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};
```

此时 operator*() 就可以访问 Vector 和 Matrix 的表示部分了。暂时不考虑更复杂的实现技术，我们只提供一个简单的实现版本：

```
Vector operator*(const Matrix& m, const Vector& v)
{
    Vector r;
    for (int i = 0; i!=rc_max; i++) {        // r[i] = m[i] * v;
        r.v[i] = 0;
        for (int j = 0; j!=rc_max; j++)
            r.v[i] += m.v[i].v[j] * v.v[j];
    }
    return r;
}
```

friend 声明既可以位于类的私有部分，也可以位于公有部分，二者没什么差别。就像一般的

成员函数一样，友元函数也应该显式地声明在类的内部，它们共同构成了该类的完整接口。

类的成员函数可以是另一个类的友元，例如：

```
class List_iterator {
    // ...
    int* next();
};

class List {
    friend int* List_iterator::next();
    // ...
};
```

要想令一个类的全部函数都成为另一个类的友元，可以用一种简写方法。例如：

```
class List {
    friend class List_iterator;
    // ...
};
```

这个 friend 声明把 List_iterator 的所有成员函数都声明成 List 的友元。

为类声明 friend 可以授权访问该类的所有函数。这意味着我们其实不了解函数的细节，只有深入友元类的内部专门查看后才能知道到底有哪些函数被赋予了访问权。在这一点上友元类声明与成员函数和友元函数声明有所区别。显然，必须慎用友元类，我们应该只用它表示那些确实有紧密关系的概念。

可以把模板参数设置为 friend：

```
template<typename T>
class X {
    friend T;
    friend class T; // 多余的 "class"
    // ...
};
```

通常情况下，我们可以选择把类设计为成员（嵌套的类）或者非成员的友元（见 18.3.1 节）。

19.4.1 发现友元

友元必须在类的外层作用域中提前声明，或者定义在直接外层非类作用域中。对于在最内层嵌套名字空间作用域内首次声明成 friend 的名字来说，它的友元性到了更外层的作用域就失效了（§ iso.7.3.1.2）。例如：

```
class C1 { };    // 将成为 N::C 的友元
void f1();       // 将成为 N::C 的友元

namespace N {
    class C2 { };        // 将成为 C 的友元
    void f2() { }        // 将成为 C 的友元

    class C {
        int x;
    public:
        friend class C1;     // OK（已经提先定义）
        friend void f1();

        friend class C3;     // OK（已经在外层作用域中定义）
```

```
        friend void f3();
        friend class C4;      // 首次声明出现在名字空间 N 内，因此友元关系只存在于此
        friend void f4();
    };

    class C3 {};              // C 的友元
    void f3() { C x; x.x = 1; } // OK：C 的友元
} // 名字空间 N

class C4 { };                 // 不是 N::C 的友元
void f4() { N::C x; x.x = 1; }  // 错误：x 是私有的并且 f4() 不是 N::C 的友元
```

即使友元函数不是声明在直接外层作用域中，也能通过它的参数找到它（见 14.2.4 节）。例如：

```
void f(Matrix& m)
{
    invert(m);      // Matrix 的友元 invert()
}
```

因此，友元函数应该显式地声明在外层作用域中，或者接受一个数据类型为该类或者其派生类的参数；否则我们无法调用该友元函数。例如：

```
// 该作用域中没有 f()

class X {
    friend void f();          // 没用
    friend void h(const X&);  // 可以通过参数找到
};

void g(const X& x)
{
    f();      // 作用域内找不到 f()
    h(x);     // X 的友元 h()
}
```

19.4.2　友元与成员

到底应该何时使用友元函数，何时把操作定义为成员函数呢？首先，我们应该让有权访问类的表示的函数数量尽可能少，并且确保所选的访问函数准确无误。因此第一个问题不是“它应该是成员、static 成员还是友元”，而是“它真的应该具有访问权限吗？”通常情况下，真正需要访问权的函数集合比我们一开始认为的规模更小。有的操作必须作为成员出现，例如构造函数、析构函数和虚函数（见 3.2.3 节和 17.2.5 节），但是大多数情况下我们可以有不同的选择。因为成员名字位于类的内部，所以除非理由足够充分，否则需要直接访问类的表示部分的函数应该定义成类的成员。

类 X 为同一种操作提供了几种不同的实现形式：

```
class X {
    // ...
    X(int);

    int m1();                 // 成员
    int m2() const;

    friend int f1(X&);        // 友元，而非成员
    friend int f2(const X&);
```

```
    friend int f3(X);
};
```

成员函数只能通过其所属类的对象调用，并且 . 和 -> 的最左侧运算对象不能执行用户自定义的类型转换（见 19.2.3 节）。例如：

```
void g()
{
    99.m1();  // 错误：并未执行到期中的 X(99).m1()
    99.m2();  // 错误：并未执行到期中的 X(99).m2()
}
```

因为非 const 引用参数不会进行隐式类型转换，所以全局函数 f1() 的性质与之类似（见 7.7 节）。然而，f2() 和 f3() 的参数可以进行类型转换：

```
void h()
{
    f1(99);   // 错误：因为该参数应当是非 const 引用，所以并未执行到期的 f1(X(99))
    f2(99);   // OK: f2(X(99)); 参数类型是 const X&
    f3(99);   // OK: f3(X(99)); 参数类型是 X
}
```

因此，需要修改类的对象状态的操作应该定义为成员函数或者接受非 const 引用参数的函数（或者非 const 指针参数）。

需要修改一个运算对象的运算符（比如 =、*= 和 ++）大多被定义成用户自定义类型的成员。相反，如果运算符的所有运算对象都能进行隐式类型转换，则该函数必须定义成接受 const 引用参数或者非引用参数的非成员函数。这种情况对应的运算符函数作用于基本类型时无须左值运算对象（比如 +、- 和 ||）。然而，这些运算符通常需要访问其运算对象所属类的表示部分。因此，二元运算符是最需要实现为友元的一类函数。

除非定义了对应的类型转换，否则没必要用成员函数替换接受引用参数的友元；反之亦然。在某些情况下，程序员可能会倾向于其中某一种调用语法。例如，要想得到 m 的转置矩阵，人们普遍喜欢使用 m2=inv(m) 而非 m2=m.inv()。相反，如果 inv() 只负责转置 m 本身，而并非产生一个新的 Matrix，则最好把它声明为成员函数。

抛开其他因素不谈，我们应该把那些需要直接访问表示的操作定义为成员函数：

- 我们不清楚其他人是否会定义类型转换运算符。
- 成员函数调用语法很清晰地告诉用户对象有可能被修改；与之相比，引用参数就隐晦多了。
- 与实现同样功能的全局函数相比，成员函数体更简短；非成员函数必须使用显式的参数，成员函数可以使用隐式的 this。
- 成员的名字位于类的内部，因此它一般比非成员函数的名字短一些。
- 如果我们已经定义了一个成员函数 f()，之后又感觉需要一个非成员函数 f(x)，则我们可以令其符合 x.f() 的含义。

反之，不需要直接访问表示的操作最好定义成非成员函数，我们将这类函数置于某个名字空间的内部以显式地表示它与类的关系（见 18.3.6 节）。

19.5 建议

[1] 用 operator[]() 执行取下标以及通过单个值查询等操作；19.2.1 节。

[2] 用 operator()() 执行函数调用、取下标以及通过多个值查询等操作；19.2.2 节。
[3] 用 operator-->() 解引用“智能指针”；19.2.3 节。
[4] 前置 ++ 优于后置 ++；19.2.4 节。
[5] 除非万不得已，否则不要定义全局 operator new() 和 operator delete()；19.2.5 节。
[6] 为特定类或者类层次体系定义成员函数 operator new() 和 operator delete()，用它们分配和释放内存空间；19.2.5 节。
[7] 用用户自定义的字面值常量模仿人们习惯的语法表示；19.2.6 节。
[8] 把字面值常量运算符置于单独的名字空间中以便于用户有选择地使用；19.2.6 节。
[9] 在大多数应用场合，建议使用标准库 string（第 36 章）而非你自己的版本；19.3 节。
[10] 如果需要使用非成员函数访问类的表示（比如改进写法，或者同时访问两个类的表示），把它声明成类的友元；19.4 节。
[11] 当需要访问类的实现时，优先选用成员函数而非友元函数；19.4.2 节。

派 生 类

若无必要，勿增实体。

——威廉·奥卡姆[⊖]

- 引言
- 派生类
  成员函数；构造函数和析构函数
- 类层次
  类型域；虚函数；显式限定；覆盖控制；using 基类成员；返回类型放松
- 抽象类
- 访问控制
  protected 成员；访问基类；using 声明与访问控制
- 成员指针
  函数成员指针；数据成员指针；基类和派生类成员
- 建议

20.1 引言

C++ 从 Simula 借鉴了类和类层次的思想。而且，C++ 还借鉴了一个重要的设计思想：类应该用来建模程序员和应用程序世界中的思想。C++ 提供了直接支持这些设计思想的语言特性。反过来，使用支持这些设计思想的语言特性也是高效使用 C++ 的标识。仍使用传统编程风格，只是将 C++ 语言特性当作这种风格语法表示上的支撑，就会错失 C++ 的重要优势。

任何一个概念都不是孤立存在的，都有与之共存的相关概念，而且其强大能力中的大部分都源于与其他概念的关联。例如，请尝试解释汽车是什么。很快你就会聊起轮子、引擎、司机、行人、卡车、救护车、道路、汽油、超速罚单、汽车旅馆等概念。由于我们使用类表示概念，问题就变为如何表示概念之间的关系。但是，我们无法直接用编程语言表达任意关系。即使可以，我们也不想这么做。考虑到实际应用，我们设计的类应该比日常概念范围更窄，但也更精确。

C++ 提供了派生类的概念及相关的语言机制来表达层次关系，即，表达类之间的共性。例如，圆形的概念和三角形的概念是相关的——它们都是形状；即，它们具有形状这一公共概念。因此，我们明确定义类 Circle 和类 Triangle 共同拥有类 Shape。在这个例子中，公共类 Shape 被称为基类（base class）或超类（superclass），而从它派生出的类 Circle 和 Triangle 被称为派生类（derived class）或子类（subclass）。在程序中表示圆形和三角形，但

⊖ 十四世纪前期，英国经院哲学家奥卡姆（William Occam）提出著名的奥卡姆剃刀律（Law of Parsimony），反对在哲学领域滥增实体，提倡如无必要，尽量利用现有概念建设新理论。——译者注

却不涉及形状的概念，就会遗漏某些重要的东西。本章介绍这一简单思想所蕴含的内容，这些内容是我们所熟知的面向对象程序设计（object-oriented programming）的基础。C++ 语言特性支持从已有类构建新的类：

- 实现继承（implementation inheritance）：通过共享基类所提供的特性来减少实现工作量。
- 接口继承（interface inheritance）：通过一个公共基类提供的接口允许不同派生类互换使用。

接口继承常被称为运行时多态（run-time polymorphism，或动态多态，dynamic polymorphism）。相反，模板（见 3.4 节和第 23 章）所提供的类的通用性与继承无关，常被称为编译时多态（compile-time polymorphism，或静态多态，static polymorphism）。

我将分 3 章讨论类层次：

- 派生类（第 20 章）：这一章介绍支持面向对象程序设计的基本语言特性，涉及派生类、虚函数和访问控制等内容。
- 类层次（第 21 章）：这一章聚焦于使用基类和派生类基于类层次概念高效组织代码。这一章的大部分内容都是讨论程序设计技术，但未涉及多重继承（一个类有多个基类）的技术层面的内容。
- 运行时类型识别（第 22 章）：这一章介绍类层次显式导航技术。特别是，这一章会介绍类型转换操作 dynamic_cast 和 static_cast 以及给定一个对象的基类的条件下确定其类型的操作（typeid）。

关于类型层次组织基本思想的简要介绍，请参考第 3 章，其中包括基类和派生类（见 3.2.2 节）以及虚函数（见 3.2.3 节）的简要介绍。接下来的几章将更为细致地介绍这些基本特性以及相关的编程技术和设计技术。

20.2　派生类

考虑设计一个程序，管理一个公司的雇员。这个程序可能有如下所示的数据结构：

```
struct Employee {
    string first_name, family_name;
    char middle_initial;
    Date hiring_date;
    short department;
    // ...
};
```

接下来，我们尝试定义一个表示经理的数据结构：

```
struct Manager {
    Employee emp;                  // 经理的雇员记录
    list<Employee*> group;  // 所管理的人员
    short level;
    // ...
};
```

经理也是一个雇员，其 Employee 数据保存在 Manager 对象的 emp 成员中。这对人类读者来说可能很明显，尤其是对细心的读者，但这段代码并未向编译器或其他工具表达出“一个 Manager 也是一个 Employee”的意思。一个 Manager* 不是一个 Employee*，因此在需要两者之一的地方，不能简单地使用另一个。我们可以对一个 Manager* 使用显式类

型转换，或者将一个 emp 成员的地址放入一个 Employee 列表中。但是，两种方法都不优雅，而且可能含混不清。正确的方法是加入一些信息，显式说明一个 Manager 是一个 Employee：

```
struct Manager : public Employee {
    list<Employee*> group;
    short level;
    // ...
};
```

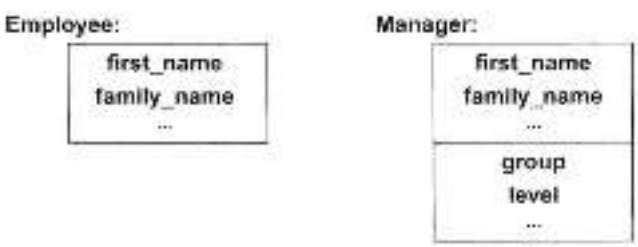
Manager 派生自 Employee，反过来，Employee 是 Manager 的一个基类。除了自己的成员（group、level 等）外，类 Manager 还拥有类型 Employee 的成员（first_name、department 等）。

派生关系通常可以图示为从派生类到其基类的一个箭头，表示派生类引用其基类（而不是相反）：



我们常常称一个派生类继承了来自基类的属性，因此这种关系也称为继承（inheritance）。有时基类也称为超类（superclass），派生类称为子类（subclass）。但是，派生类对象中的数据是其基类对象数据的超集，对于观察到这一现象的人来说，这种术语令人困惑。一个派生类通常比基类保存更多数据、提供更多函数，从这一点来说，它比基类更大（绝不会更小）。

派生类概念的一种流行且高效的实现是将派生类对象表示为基类对象，再加上那些专属于派生类的信息放在末尾。例如：



派生一个类没有任何内存额外开销，所需内存就是成员所需空间。

按这种方式从 Employee 派生 Manager，使得 Manager 成为 Employee 的一个子类型，从而在任何接受 Employee 的地方都可以使用 Manager。例如，我们现在可以创建一个 Employee 列表，其中的元素可以是 Manager：

```
void f(Manager m1, Employee e1)
{
    list<Employee*> elist {&m1,&e1};
    // ...
}
```

一个 Manager（也）是一个 Employee，因此 Manager* 可用作 Employee*。类似地，一个 Manager& 可用作一个 Employee&。但是，一个 Employee 不一定是一个 Manager，因此 Employee* 不能用作 Manager*。一般而言，如果一个类 Derived 有一个公有基类（见 20.5 节）Base，那么我们就可以将一个 Derived* 赋予一个 Base* 类型的变量而无须显式类型转

换。而相反的转换，即从 Base* 到 Derived*，必须是显式的。例如：

```
void g(Manager mm, Employee ee)
{
    Employee* pe = &mm;          // 正确：每个 Manager 都是一个 Employee
    Manager* pm = &ee;           // 错误：并不是每个 Employee 都是一个 Manager

    pm->level = 2;               // 完准：ee 不包含 level

    pm = static_cast<Manager*>(pe); // 暴力转换：可奏效
                                    // 因为 pe 指向 Manager mm

    pm->level = 2;               // 没问题：pm 指向 Manager mm，包含 level
}
```

换句话说，若通过指针和引用进行操作，派生类对象可以当作其基类对象处理，反过来则不能。static_cast 和 dynamic_cast 的使用将在 22.2 节讨论。

将一个类用作基类等价于定义一个该类的（无名）对象。因此，类必须定义后才能用作基类（见 8.2.2 节）：

```
class Employee;    // 只是声明，不是定义

class Manager : public Employee { // 错误：Employee 未定义
    // ...
};
```

20.2.1　成员函数

Employee 和 Manager 这样的简单数据结构实在没什么意思，而且通常不是特别有用。若想提供一个真正的类型，就要为其定义一组恰当的操作，而且不能依赖于特定的表示形式。例如：

```
class Employee {
public:
    void print() const;
    string full_name() const { return first_name + ' ' + middle_initial + ' ' + family_name; }
    // ...
private:
    string first_name, family_name;
    char middle_initial;
    // ...
};

class Manager : public Employee {
public:
    void print() const;
    // ...
};
```

派生类的成员可以使用基类的公有和保护成员（见 20.5 节），就好像它们声明在派生类中一样。例如：

```
void Manager::print() const
{
    cout << "name is " << full_name() << '\n';
    // ...
}
```

但派生类不能访问基类的私有成员：

```
void Manager::print() const
{
    cout << " name is " << family_name << '\n';        // 错误!
    // ...
}
```

Manager::print() 的第二个版本会编译失败，因为它不能访问 family_name。

这有些奇怪，但考虑替代方案：派生类的成员函数可以访问其基类的私有成员。这会令私有成员的概念变得毫无意义，因为程序员简单地从一个类派生出一个新类，就能获得其私有部分的访问权。而且，我们再也不能通过检查成员函数和友元函数就找到私有成员的所有使用之处了。我们必须检查完整程序中涉及派生类的所有源文件，然后检查这些类的每个函数，然后再检查这些类的所有派生类，依此类推。这样的方式不仅是一项烦人的工作，而且实际上通常是不可行的。如果可行，我们可以使用保护的而非私有的成员（见 20.5 节）。

通常，对派生类而言最干净的解决方案是只使用其基类的公有成员。例如：

```
void Manager::print() const
{
    Employee::print();  // 打印 Employee 信息
    cout << level;      // 打印 Manager 特有信息
    // ...
}
```

注意，调用 print() 必须使用 ::，因为它在 Manager 中已经重新定义了。这是一种很典型的名字重用。而粗心的程序员可能写成这样：

```
void Manager::print() const
{
    print();    // 错误
    // 打印 Manager 专有信息
}
```

这个函数的运行结果就是不断地递归调用，直至程序崩溃。

20.2.2 构造函数和析构函数

构造函数和析构函数照例是必不可少的：

- 对象自底向上构造（基类先于成员，成员先于派生类），自顶向下销毁（派生类先于成员，成员先于基类）；见 17.2.3 节。
- 每个类都可以初始化其成员和基类（但不能直接初始化其基类的成员或基类的基类）；见 17.4.1 节。
- 类层次中的析构函数通常应该是 virtual 的；见 17.2.5 节
- 类层次中类的拷贝构造函数须小心使用，以避免切片现象；见 17.5.1.4 节。
- 虚函数调用的解析、dynamic_cast，以及构造函数或析构函数中的 typeid() 反映了构造和析构的阶段（而不是尚未构造完成的对象的类型）；见 22.4 节。

在计算机科学中，“上”和“下”可能令人非常困惑。在源程序中，基类的定义必须出现在其派生类的定义之前。对于一个小型程序，这意味着在屏幕上基类位于派生类上方。而且，在画一棵树时，我们习惯于将根画在顶端。但是，当讨论自底向上构造对象时，我的意思是从最基础的部分（如基类）开始构造，依赖于它的部分（如派生类）稍后构造，即，我们是从根（基类）向叶（派生类）进行构造的。

20.3　类层次

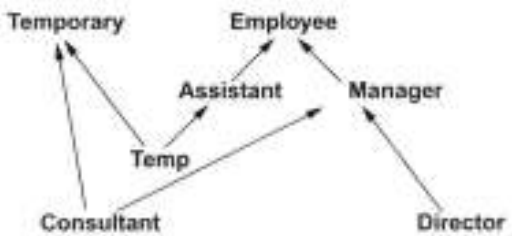
一个派生类自身也可以作为其他类的基类。例如：

```
class Employee { /* ... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };
```

我们习惯称这样一组相关的类为类层次（class hierarchy）。这种层次结构大多数情况下是一棵树，但也可能是更一般的图结构。例如：

```
class Temporary { /* ... */ };
class Assistant : public Employee { /* ... */ };
class Temp : public Temporary, public Assistant { /* ... */ };
class Consultant : public Temporary, public Manager { /* ... */ };
```

它们的层次关系可图示如下：



因此，如21.3节所述，C++可以表达类之间的一个有向无环图结构。

20.3.1　类型域

为了使派生类不至于成为仅仅是一种方便的声明简写方式，我们必须解决一个问题：给定一个Base*类型的指针，它指向的对象的真正派生类型是什么？C++提供了四种基本解决方法：

[1] 保证指针只能指向单一类型的对象（见3.4节和第23章）。

[2] 在基类中放置一个类型域，供函数查看。

[3] 使用dynamic_cast（见22.2节和22.6节）。

[4] 使用虚函数（见3.2.3节和20.3.2节）。

除非使用final（见20.3.4.2节），否则方法[1]依赖于所使用类型的很多知识，比编译器所能掌握的更多。一般而言，不要试图比类型系统更聪明，但方法[1]可用来（特别是与模板组合使用）实现同构容器（如标准库vector和map），以获得非常好的性能。方法[2]、[3]和[4]可用来实现异构列表，即，多种不同类型对象（指针）的列表。方法[3]是方法[2]的一种语言支持的变体，而方法[4]是方法[2]的一种特殊的类型安全的变体。组合使用方法[1]和方法[4]特别有意思也非常强大；在几乎所有情况下，得到的代码都会比方法[2]和方法[3]更干净。

让我们首先考察简单的类型域方法，来看一看为什么最好不要用它。经理/雇员的例子可以重定义如下：

```
struct Employee {
    enum Empl_type { man, empl };
    Empl_type type;
```

```
    Employee() : type(empl) { }

    string first_name, family_name;
    char middle_initial;

    Date hiring_date;
    short department;
    // ...
};

struct Manager : public Employee {
    Manager() { type = man; }

    list<Employee*> group;  // 管理的人
    short level;
    // ...
};
```

有了这个定义，我们现在就可以编写一个函数打印任意 Employee 的信息了：

```
void print_employee(const Employee* e)
{
    switch (e->type) {
    case Employee::empl:
        cout << e->family_name << '\t' << e->department << '\n';
        // ...
        break;
    case Employee::man:
    {   cout << e->family_name << '\t' << e->department << '\n';
        // ...
        const Manager* p = static_cast<const Manager*>(e);
        cout << " level " << p->level << '\n';
        // ...
        break;
    }
    }
}
```

然后用这个函数打印一个 Employee 列表，可能像下面这样：

```
void print_list(const list<Employee*>& elist)
{
    for (auto x : elist)
        print_employee(x);
}
```

这种方法很奏效，特别是对由一个人维护的小程序来说更是如此。但是，它有一个根本缺陷，它依赖于程序员手工操纵类型，编译器无法对此进行检查。此问题通常会变得更糟，因为 print_employee() 这样的函数通常要利用类的通用性：

```
void print_employee(const Employee* e)
{
    cout << e->family_name << '\t' << e->department << '\n';
    // ...
    if (e->type == Employee::man) {
        const Manager* p = static_cast<const Manager*>(e);
        cout << " level " << p->level << '\n';
        // ...
    }
}
```

在一个处理很多派生类的大型函数中找到所有这种类型域的检测是非常困难的。即使都找到了，理解发生了什么也很困难。而且，增加任何一种新的 Employee 都要修改系统中的所有关键函数，只要包含类型域的检测就要修改。在修改之后，程序员还必须考虑所有可能需要检测类型域的函数。这意味着需要检查关键源码，还要检查受影响的代码，这会带来不可避免的额外开销。是否使用显式类型转换可以作为代码是否需要检查的重要提示，这可以在一定程度上减少工作量。

换句话说，使用类型域技术很容易出错，也容易导致维护问题。随着程序规模增大，问题就变得更为严重，因为类型域的使用违反了模块化和数据隐藏的思想。使用类型域的每个函数都必须了解包含类型域的类的派生类的实现细节。

而且，所有派生类都可以访问类型域这样的公共数据，这似乎会诱使人们添加更多这样的数据。公共基类从而变成所有“有用信息”的仓库。这最终会使基类和派生类的实现变得错综复杂，这是最糟糕的。在一个大型类层次中，公共基类中的可访问的（非 private）数据就变成了类层次中的“全局变量”。为了令设计简洁、维护容易，我们还是希望独立的问题保持分离，避免相互依赖。

20.3.2 虚函数

虚函数机制允许程序员在基类中声明函数，然后在每个派生类中重新定义这些函数，从而解决了类型域方法的固有问题。编译器和链接器会保证对象和施用于对象之上的函数之间的正确关联。例如：

```
class Employee {
public:
    Employee(const string& name, int dept);
    virtual void print() const;
    // ...
private:
    string first_name, family_name;
    short department;
    // ...
};
```

关键字 virtual 指出 print() 作为这个类自身定义的 print() 函数及其派生类中定义的 print() 函数的接口。如果派生类中定义了此 print() 函数，编译器会确保对给定的 Employee 对象调用正确的 print()。

为了允许一个虚函数声明能作为派生类中定义的函数的接口，派生类中函数的参数类型必须与基类中声明的参数类型完全一致，返回类型也只允许细微改变（见 20.3.6 节）。虚成员函数有时也称为方法（method）。

首次声明虚函数的类必须定义它（除非虚函数被声明为纯虚函数；见 20.4 节）。例如：

```
void Employee::print() const
{
    cout << family_name << '\t' << department << '\n';
    // ...
}
```

即使没有派生类，也可以使用虚函数，而一个派生类如果不需要自有版本的虚函数，可以不定义它。当派生一个类时，如需要某个函数，定义恰当版本即可。例如：

```
class Manager : public Employee {
public:
    Manager(const string& name, int dept, int lvl);
    void print() const;
    // ...
private:
    list<Employee*> group;
    short level;
    // ...
};

void Manager::print() const
{
    Employee::print();
    cout << "\tlevel " << level << '\n';
    // ...
}
```

如果派生类中一个函数的名字和参数类型与基类中的一个虚函数完全相同，则称它覆盖（override）了虚函数的基类版本。此外，我们也可以用一个派生层次更深的返回类型覆盖基类中的虚函数（见 20.3.6 节）。

除了我们显式说明调用虚函数的哪个版本（如 Employee::print()）之外，覆盖版本会作为最恰当的选择应用于调用它的对象。无论用哪个基类（接口）访问对象，虚函数调用机制都会保证我们总是得到相同的函数。

现在全局函数 print_employee()（见 20.3.1 节）已经没有存在的必要了，因为成员函数 print() 已经取代了它的位置。我们可以像下面这样打印一个 Employee 列表：

```
void print_list(const list<Employee*>& s)
{
    for (auto x : s)
        x->print();
}
```

每个 Employee 会根据其类型正确打印。例如：

```
int main()
{
    Employee e {"Brown",1234};
    Manager m {"Smith",1234,2};

    print_list({&e,&m});
}
```

会输出：

```
Smith 1234
    level 2
Brown 1234
```

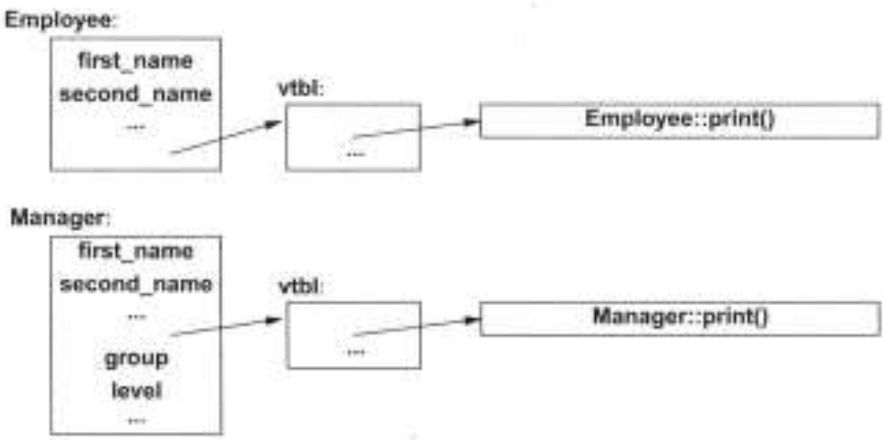
注意，即使 print_list() 是在指定派生类 Manager 构思之前编写并编译的，这段代码也能正确运行。这是类的关键一面。如能正确运用，它会成为面向对象设计的基石，并为程序进化提供一定程度的稳定性。

无论真正使用的确切 Employee 类型是什么，都能令 Employee 的函数表现出“正确的”行为，这称为多态性（polymorphism）。具有虚函数的类型称为多态类型（polymorphic type）或（更精确的）运行时多态类型（run-time polymorphic type）。在 C++ 中为了获得运

行时多态行为，必须调用 virtual 成员函数，对象必须通过指针或引用进行访问。当直接操作一个对象时（而不是通过指针或引用），编译器了解其确切类型，从而就不需要运行时多态了。

默认情况下，覆盖虚函数的函数自身也变为 virtual 的。我们在派生类中可以重复关键字 virtual，但这不是必需的，我的建议是不重复 virtual。如果你希望明确标记覆盖版本，可使用 override（见 20.3.4.1 节）。

显然，为了实现多态性，编译器必须在每个 Employee 类的对象中保存某种类型信息，并利用它选择虚函数 print() 的正确版本。在一个典型的 C++ 实现中，这只会占用一个指针大小的空间（见 3.2.3 节）；常用的编译器实现技术是将虚函数名转换为函数指针表中的一个索引。这个表通常称为虚函数表（the virtual function table）或简称为 vbtl。每个具有虚函数的类都有自己的 vbtl，用来标识它的虚函数。下图展示了这种实现技术：



vbtl 中的函数令对象能正确使用，即使调用者不了解对象的大小和数据布局也没关系。调用者的实现只需了解在一个 Employee 中 vbtl 的位置以及每个虚函数的索引是多少就可以了。这种虚调用机制可以做到与“正常函数调用”几乎一样高效（性能差距在 25% 以内），因此，只要普通函数调用的性能可以接受，那么性能因素就不应成为使用虚函数的障碍。虚调用机制为每个对象带来的额外内存开销是一个指针，再加上每个类一个 vbtl。只有带虚函数的类的对象才需要付出这样的代价。只有你确实需要虚函数所提供的功能时，才应选择付出这种代价。如果你选择使用类型域方法作为替代，也需要大致相当的额外内存开销。

从构造函数或析构函数中调用虚函数能反映出部分构造状态或部分销毁状态的对象（见 22.4 节）。因此从构造函数或析构函数中调用虚函数通常是一个糟糕的主意。

20.3.3　显式限定

使用作用域解析运算符 :: 调用函数（如 Manager::print()）能保证不使用 virtual 机制：

```
void Manager::print() const
{
    Employee::print();  // 不是一个虚调用
    cout << "\tlevel " << level << '\n';
    // ...
}
```

否则，Manager::print() 会面临一个无限递归。使用限定名还有另一个我们需要的效果，即，

如果一个虚函数也是一个 inline（并不罕见），对于使用 :: 限定的调用就可以进行内联替换。这给程序员提供了一种方法高效处理某些重要的特殊情形——一个虚函数对相同对象调用另一个虚函数。函数 Manager::print() 就是这样一个例子。由于在调用 Manager::print() 时已经确定了对象的类型，就没有必要再次动态确定 Employee::print() 了。

20.3.4 覆盖控制

如果你在派生类中声明了一个函数，其名字和类型都与基类中的一个虚函数完全一样，则派生类中的这个函数就覆盖了基类中的版本。这是一个简单有效的规则。但是，在更大的类层次中，很难保证你真的覆盖了你想要覆盖的那个函数。考虑下面的代码：

```
struct B0 {
    void f(int) const;
    virtual void g(double);
};

struct B1 : B0 { /* ... */ };
struct B2 : B1 { /* ... */ };
struct B3 : B2 { /* ... */ };
struct B4 : B3 { /* ... */ };
struct B5 : B4 { /* ... */ };

struct D : B5 {
    void f(int) const;          // 覆盖基类中的 f()
    void g(int);                // 隐藏基类中的 g()
    virtual int h();            // 覆盖基类中的 h()
};
```

这段代码展示了 3 个错误，如果是在一个真实的类层次中，类 B0...B5 有很多成员且散布在很多头文件中，这些错误就很难发现。它们是：

- B0::f() 不是 virtual 的，因此不能覆盖它，只会隐藏它（见 20.3.5 节）。
- D::g() 的参数类型与 B0::g() 不同，因此如果它覆盖了什么东西，也不会是虚函数 B0::g()。最可能的是，D::g() 只是隐藏了 B0::g()。
- 在 B0 中没有名为 h() 的函数，如果 D::h() 覆盖了什么东西，也不会是 B0 中的函数。可能性最大的情况是它引入的是一个全新的虚函数。

我并未给出 B1...B5 是什么，因此这些类中的声明可能导致完全不同的结果。我个人不会对覆盖函数（冗余地）使用 virtual。对小型程序而言（特别是使用的编译器能对常见覆盖相关错误给出得体的警告时），实现正确的覆盖并不困难。但对大型类层次，就要用到特定的控制机制了：

- virtual：函数可能被覆盖（见 20.3.2 节）。
- =0：函数必须是 virtual 的，且必须被覆盖（见 20.4 节）。
- override：函数要覆盖基类中的一个虚函数（见 20.3.4.1 节）。
- final：函数不能被覆盖（见 20.3.2 节）。

如果不使用这些覆盖控制，一个非 static 成员函数为虚函数当且仅当它覆盖了基类中的一个 virtual 函数（见 20.3.2 节）。

编译器能对不一致的覆盖控制给出警告。例如，如果一个类声明只对基类中九个虚函数中的七个使用了 override，就会令维护者感到困惑。

20.3.4.1 override

我们可以显式说明想要进行覆盖：

```
struct D : B5 {
    void f(int) const override;     // 错误：B0::f() 不是虚函数
    void g(int) override;           // 错误：B0::g() 接受一个 double 参数
    virtual int h() override;       // 错误：不存在函数 h() 可覆盖
};
```

这个定义中的 3 个声明都是错误的（假定中间层基类 B1...B5 不提供相关函数）。

在一个有很多虚函数的大型或复杂类层次中，virtual 和 override 最好的使用方式是前者只用来引入新的虚函数，而后者指出函数要覆盖某个虚函数。使用 override 有点儿啰嗦，但能澄清程序员的意图。

说明符 override 出现在声明的最后。例如：

```
void f(int) const noexcept override;// 正确（如果有一个适合的 f() 可覆盖）
override void f(int) const noexcept;// 语法错误
void f(int) override const noexcept;// 错误
```

是的，virtual 为前缀 override 为后缀有些不合逻辑。这是我们为了保证数十年来代码的兼容性和稳定性不得不付出的一部分代价。

说明符 override 不是函数类型的一部分，而且在类外定义中不能重复。例如：

```
class Derived : public Base {
    void f() override;          // 正确，若 Base 有一个 virtual f() 的话
    void g() override;          // 正确，若 Base 有一个 virtual g() 的话
};

void Derived::f() override      // 错误：类外不能使用 override
{
    // ...
}

void g()                        // 正确
{
    // ...
}
```

奇怪的是，override 不是一个关键字；它是所谓的上下文关键字（contextual keyword）。即，override 在某些上下文中有特殊含义，但在其他地方可用作标识符。例如：

```
int override = 7;

struct Dx : Base {
    int override;

    int f() override
    {
        return override + ::override;
    }
};
```

不要沉迷于这种自作聪明的做法，这会使代码维护变得复杂。override 是一个上下文关键字而非普通关键字的唯一原因是，若干年来已有大量代码将它用作了普通标识符。另一个上下文关键字是 final（见 20.3.4.2 节）。

20.3.4.2 final

当我们声明一个成员函数时，就要选择它是 virtual 还是非 virtual（默认情况）。我们对函数使用 virtual，是希望派生类的编写者能定义它或重定义它。我们应根据类的含义（语义）做出选择：

- 是否需要派生类？
- 派生类的设计者是否需要重定义函数来达到某个合理的目标？
- 正确覆盖函数是否直截了当（即，覆盖版本提供虚函数所期望的语义是否相当简单）？

如果 3 个问题的答案都是“否”，我们可以将函数声明为非 virtual 的，这样会使设计更为简单，有时还能获得性能收益（大多数是来自内联）。标准库中有大量这种例子。

极少数情况下，我们开始设计类层次时使用虚函数，但在定义一组派生类后，某个问题的答案变成了“否”。例如，我们可以想象一种编程语言的抽象语法树，其中所有语言结构都定义为具体节点类，这些类派生自若干接口。我们若要修改此语言，只需派生一个新的类即可。在此情况下，我们可能希望阻止用户覆盖虚函数，因为这种覆盖唯一能做的就是改变编程语言的语义。即，我们可能希望关闭我们的设计，不允许用户修改。例如：

```
struct Node {  // interface class
    virtual Type type() = 0;
    // ...
};

class If_statement : public Node {
public:
    Type type() override final;     // 防止进一步覆盖
    // ...
};
```

在一个实际的类层次中，通用接口（本例中的 Node）和表示特定语言结构的派生类（本例中的 If_statement）之间可能有多个中间类。但是，本例的关键之处是 Node::type() 应该被覆盖（这也是它为什么声明为 virtual），而其覆盖版本 If_statement::type() 不应再被覆盖（这也是它为什么声明为 final）。在对一个成员函数使用 final 后，它就不能再被覆盖了，如果你尝试这么做，就会产生一个错误。例如：

```
class Modified_if_statement : public If_statement {
public:
    Type type() override;     // 错误：If_statement::type() 为 final
    // ...
};
```

通过在类名后加上 final，我们可以将一个类的所有 virtual 成员函数都声明为 final 的。例如：

```
class For_statement final : public Node {
public:
    Type type() override;
    // ...
};

class Modified_for_statement : public For_statement {     // 错误：For_statement 为 final
    Type type() override;
    // ...
};
```

这样做既有好的地方，也带来了坏处——它不仅阻止了覆盖，也阻止了从一个类进一步派生

其他类。有的人试图通过使用 final 来获得性能提升，毕竟，非 virtual 函数比 virtual 版本快（在现代 C++ 编译器上可能会快 25%），而且进行内联的机会更大（见 12.1.5 节）。但是，不要盲目地将 final 用于优化目的；它会影响类层次的设计（通常是负面的影响），并且很少能获得显著的性能提升。在声称效率提升之前必须进行认真的测试。正确使用 final 应该能清晰地反映你认为正确的类层次设计。即，final 的使用应该反映语义需求。

final 说明符不是函数类型的一部分，在类外定义中不能重复使用。例如：

```
class Derived : public Base {
    void f() final;         // 正确，如果 Base 有一个 virtual f() 的话
    void g() final;         // 正确，如果 Base 有一个 virtual g() 的话
    // ...
};

void Derived::f() final     // 错误：在类外使用 final
{
    // ...
}
void g() final              // 正确
{
    // ...
}
```

类似 override（见 20.3.4.1 节），final 也是一个上下文关键字。即，final 在一些上下文中有特殊含义，但在其他地方可以用作一个普通标识符。例如：

```
int final = 7;

struct Dx : Base {
    int final;

    int f() final
    {
        return final + ::final;
    }
};
```

不要沉迷于这种自作聪明的做法，这会使代码维护变得复杂。final 是一个上下文关键字而非普通关键字的唯一原因是，若干年来已有大量代码将它用作了普通标识符。另一个上下文关键字是 override（见 20.3.4.1 节）。

20.3.5　using 基类成员

函数重载不会跨越作用域（见 12.3.3 节）。例如：

```
struct Base {
    void f(int);
};

struct Derived : Base {
    void f(double);
};

void use(Derived d)
{
    d.f(1);         // 调用 Derived::f(double)
```

```
    Base& br = d
    br.f(1);        // 调用 Base::f(int)
}
```

这会令一些人困惑，而且我们有时希望重载能保证选择最佳匹配的成员函数。类似名字空间，我们可以用 using 声明将一个函数加入作用域中。例如：

```
struct D2 : Base {
    using Base::f;      // 将 Base 中所有 f 加入 D2
    void f(double);
};
void use2(D2 d)
{
    d.f(1);         // 调用 D2::f(int)，即 Base::f(int)
    Base& br = d
    br.f(1);        // 调用 Base::f(int)
}
```

这是“一个类也可以看作一个名字空间”（见 16.2 节）所带来的简单结果。

我们可使用多个 using 声明从多个基类引入名字。例如：

```
struct B1 {
    void f(int);
};

struct B2 {
    void f(double);
};

struct D : B1, B2 {
    using B1::f;
    using B2::f;
    void f(char);
};

void use(D d)
{
    d.f(1);     // 调用 D::f(int)，即 B1::f(int)
    d.f('a');   // 调用 D::f(char)
    d.f(1.0);   // 调用 D::f(double)，即 B2::f(double)
}
```

我们还可以将构造函数引入派生类作用域，参见 20.3.5.1 节。由 using 声明引入派生类作用域的名字，其访问权限由 using 声明所在位置决定，参见 20.5.3 节。我们不能用 using 指示将一个基类的所有成员都引入一个派生类中。

20.3.5.1 继承构造函数

比如说我希望设计一个类似 std::vector 但保证越界检查的向量。我可以尝试这样设计：

```
template<class T>
struct Vector : std::vector<T> {
    T& operator[](size_type i) { check(i); return this->elem(i); }
    const T& operator[](size_type i) const { check(i); return this->elem(i); }

    void check(size_type i) { if (this->size()<i) throw range_error{"Vector::check() failed"}; }
};
```

不幸的是，我们很快就会发现这个定义很不完整。例如：

```
Vector<int> v { 1, 2, 3, 5, 8 };  // 错误：无初始化器列表构造函数
```

快速检查即可显示 Vector 没有从 std::vector 继承任何构造函数。

这是一条合理的规则：如果一个类向一个基类添加了数据成员或要求一个更严格的类不变式，继承构造函数可能就是一场灾难。但是，Vector 并没有做这些事情，它要求继承构造函数是合理的。

我们可以简单地声明应该继承构造函数，从而解决此问题：

```
template<class T>
struct Vector : std::vector<T> {
    using vector<T>::vector;      // 继承 vector 的构造函数

    T& operator[](size_type i) { check(i); return this->elem(i); }
    const T& operator[](size_type i) const { check(i); return this->elem(i); }

    void check(size_type i) { if (this->size()<i) throw Bad_index(i); }
};

Vector<int> v { 1, 2, 3, 5, 8 };      // 正确：使用来自 std::vector 的初始化器列表构造函数
```

这里 using 的使用与其在普通函数中的使用一样（见 14.4.5 节和 20.3.5 节）。

如果你在派生类中继承了构造函数，又定义了需要显式初始化的新成员变量，就会搬起石头砸自己的脚：

```
struct B1 {
    B1(int) { }
};

struct D1 : B1 {
    using B1::B1; // 隐式声明 D1(int)
    string s;     // string 有默认构造函数
    int x;        // 我们“忘记了”为 x 提供初始值
};

void test()
{
    D1 d {6};     // 糟糕：d.x 未初始化
    D1 e;         // 错误：D1 没有默认构造函数
}
```

D1::s 被初始化而 D1::x 未初始化的原因是继承的构造函数等价于只初始化基类的构造函数。对于本例，上面的版本与下面这个版本是等价的：

```
struct D1 : B1 {
    D1(int i) : B1(i) { }
    string s;     // string 有默认构造函数
    int x;        // 我们“忘记了”为 x 提供初始值
};
```

一种搬开石头的方法是添加类内成员初始化器（见 17.4.4 节）：

```
struct D1 : B1 {
    using B1::B1; // 隐式声明 D1(int)
    int x {0};    // 注意：x 被初始化
};

void test()
```

```
{
    D1 d {6};  // d.x 为 0
}
```

通常，我们最好避免自作聪明，仅对不增加数据成员的简单情况使用继承构造函数。

20.3.6 返回类型放松

覆盖函数的类型必须与它所覆盖的虚函数的类型完全一致，C++ 对这一规则提供了一种放松规则。即，如果原返回类型为 B*，则覆盖函数的返回类型可以为 D*，只要 B 是 D 的一个公有基类即可。类似地，返回类型 B& 可放松为 D&。这一规则有时称为协变返回（covariant return）规则。

这一放松规则只能用于返回类型是指针或引用的情况，但不能是 unique_ptr 这样的“智能指针”（见 5.2.1 节）。特别是，对参数类型没有类似的放松规则，否则会引起类型违背。

考虑一个类层次，它表示不同类型的表达式。除了操纵表达式的运算之外，基类 Expr 还提供创建各种类型的表达式的新对象的特性：

```
class Expr {
public:
    Expr();                     // 默认构造函数
    Expr(const Expr&);          // 拷贝构造函数
    virtual Expr* new_expr() =0;
    virtual Expr* clone() =0;
    // ...
};
```

基本思想是 new_expr() 创建特定类型表达式的一个默认对象，而 clone() 创建已有对象的拷贝。两个函数都返回派生自 Expr 的某个特定类的对象。它们绝不能仅返回一个“普通 Expr”，因为我们故意（当然也是正确的）将 Expr 声明为一个抽象类。

一个派生类可覆盖 new_expr() 或 clone() 来返回其自身类型的对象：

```
class Cond : public Expr {
public:
    Cond();
    Cond(const Cond&);
    Cond* new_expr() override { return new Cond(); }
    Cond* clone() override { return new Cond(*this); }
    // ...
};
```

这意味着给定一个 Expr 类对象，用户可以创建一个“类型完全一样”的新对象。例如：

```
void user(Expr* p)
{
    Expr* p2 = p->new_expr();
    // ...
}
```

赋予 p2 的指针声明为“普通 Expr”指针，但它会指向一个 Expr 的派生类（如 Cond）的对象。

Cond::new_expr() 和 Cond:: clone() 的返回类型是 Cond* 而非 Expr*。这允许我们克隆一个 Cond 而不会损失类型信息。类似地，一个派生类 Addition 的 clone() 可以返回一个 Addition*。例如：

```
void user2(Cond* pc, Addition* pa)
{
    Cond* p1 = pc->clone();
    Addition* p2 = pa->clone();
    // ...
}
```

如果对一个 Expr 使用 clone()，那么我们只知道结果是一个 Expr*：

```
void user3(Cond* pc, Expr* pe)
{
    Cond* p1 = pc->clone();
    Cond* p2 = pe->clone();        // 错误：Expr::clone() 返回一个 Expr*
    // ...
}
```

由于 new_expr() 和 clone() 这样的函数是 virtual 的且（间接）构造对象，它们常被称为虚构造函数（virtual constructor）。这种函数都是简单地使用普通构造函数来创建一个适合的对象。

为了创建一个对象，构造函数需要确切了解要创建的对象的类型。因此，构造函数不能是 virtual 的。而且，一个构造函数并不完全是一个普通函数。特别是，它与内存管理例程的交互方式与普通成员函数不同。因此，你不能接受一个构造函数的指针并将其传递给一个对象创建函数。

通过定义一个函数来调用构造函数并返回构造的对象，我们就可以解决所有这些问题。这很幸运，因为在不了解确切类型的情况下创建一个新对象通常是很有用的。类 Ival_box_maker（见 21.2.4 节）就是一个例子，它专门完成这种工作。

20.4　抽象类

很多类类似 Employee，自身可用，也可用作派生类的接口以及派生类实现的一部分。对于这样的类，20.3.2 节中介绍的技术就足够了。但是，不是所有的类都遵循这种模式。某些类，例如 Shape，表示一个抽象概念，自身不能有具体对象。一个 Shape 仅在作为某个派生类的基类时才有意义，这也体现在我们很难为其虚函数提供有意义的定义方面：

```
class Shape {
public:
    virtual void rotate(int) { throw runtime_error{"Shape::rotate"}; }    // 并不优雅
    virtual void draw() const { throw runtime_error{"Shape::draw"}; }
    // ...
};
```

试图创建一个这种不明类型的形状很愚蠢但却是合法的：

```
Shape s; // 愚蠢："无形形状"
```

这很愚蠢，因为 s 上的每个操作都会导致一个错误。

一种更好的替代方法是将类 Shape 的虚函数声明为纯虚函数（pure virtual function）。通过使用“伪初始化器”=0 就可以将一个虚函数“提纯”：

```
class Shape {        // 抽象类
public:
    virtual void rotate(int) = 0;          // 纯虚函数
    virtual void draw() const = 0;         // 纯虚函数
    virtual bool is_closed() const = 0;    // 纯虚函数
    // ...
    virtual ~Shape();                      // 虚函数
};
```

具有一个或多个纯虚函数的类称为抽象类（abstract class），我们无法创建抽象类的对象：

```
Shape s; // 错误：创建抽象类 Shape 的对象
```

抽象类就是要作为通过指针和引用访问的对象的接口（为保持多态行为）。因此，对一个抽象类来说，定义一个虚析构函数（见 3.2.4 节和 21.2.2 节）通常很重要。由于抽象类提供的接口不能用来创建对象，因此抽象类通常没有构造函数。

抽象类只能用作其他类的接口。例如：

```
class Point { /* ... */ };

class Circle : public Shape {
public:
    void rotate(int) override { }
    void draw() const override;
    bool is_closed() const override { return true; }

    Circle(Point p, int r);
private:
    Point center;
    int radius;
};
```

如果纯虚函数在派生类中未被定义，那么它仍保持是纯虚函数，因此派生类也是一个抽象类。这令我们可以阶段性地构建具体实现：

```
class Polygon : public Shape {          // 抽象类
public:
    bool is_closed() const override { return true; }
    // ... draw 和 rotate 未被覆盖 ...
};

Polygon b {p1,p2,p3,p4};      // 错误：声明抽象类 Polygon 的对象
```

Polygon 是抽象类，因为我们未覆盖 draw() 和 rotate()。只有当它们被覆盖时，我们才得到一个可以创建对象的类：

```
class Irregular_polygon : public Polygon {
    list<Point> lp;
public:
    Irregular_polygon(initializer_list<Point>);

    void draw() const override;
    void rotate(int) override;
    // ...
};

Irregular_polygon poly {p1,p2,p3,p4};   // 假定 p1 .. p4 是某处定义的点
```

抽象类提供接口，但不暴露实现细节。例如，一个操作系统可能将其设备驱动程序的细节隐藏在一个抽象类之后：

```
class Character_device {
public:
    virtual int open(int opt) = 0;
    virtual int close(int opt) = 0;
    virtual int read(char* p, int n) = 0;
    virtual int write(const char* p, int n) = 0;
```

```
    virtual int ioctl(int ...) = 0;          //设备 I/O 控制

    virtual ~Character_device() { }          //虚析构函数
};
```

我们随后就可以通过派生 Character_device 来实现特定驱动程序，并通过此接口操纵各种驱动程序了。

抽象类所支持的设计风格称为接口继承（interface inheritance），它与实现继承（implementation inheritance）相对，后者是由带状态或定义了成员函数的基类所支撑的。两种风格组合使用是可能的。即，我们可以定义并使用既带状态又有纯虚函数的基类。但是，这种混合风格会令人迷惑，也需要特别小心。

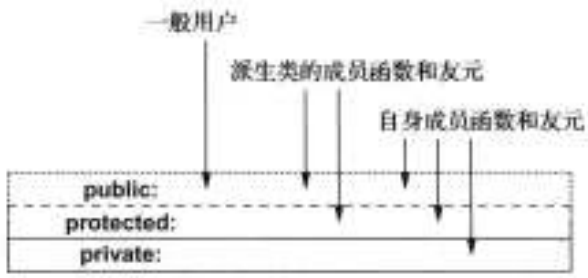
引入抽象类机制后，我们就有了用类作为构建单元编写模块化风格的完整程序的基础技术了。

20.5　访问控制

一个类成员可以是 private、protected 或 public 的：

- 如果它是 private 的，仅可被所属类的成员函数和友元函数所使用。
- 如果它是 protected 的，仅可被所属类的成员函数和友元函数以及派生类的成员函数和友元函数所使用（见 19.4 节）。
- 如果它是 public 的，可被任何函数所使用。

这反映了函数按类访问权限可分为三类：实现类的函数（其友元和成员）、实现派生类的函数（派生类的友元和成员）以及其他函数。这种分类可图示如下：



访问控制对名字的应用是一致的。一个名字引用的是什么并不影响对其访问的控制。这意味着我们不但可以有 private 数据成员，还可以有 private 成员函数、类型、常量，等等。例如，一个高效的非侵入性链表类通常要求数据结构掌握元素的动态。若一个链表不要求修改其元素（例如，要求元素类型有链接域），那么它就是非侵入性的（nonintrusive）。组织链表所用的信息和数据结构可以声明为 private 的：

```
template<class T>
class List {
public:
    void insert(T);
    T get();
    // ...
private:
    struct Link { T val; Link* next; };

    struct Chunk {
```

```
        enum { chunk_size = 15 };
        Link v[chunk_size];
        Chunk* next;
    };
    Chunk* allocated;
    Link* free;
    Link* get_free();
    Link* head;
};
```

公有函数的定义很直接：

```
template<class T>
void List<T>::insert(T val)
{
    Link* lnk = get_free();
    lnk->val = val;
    lnk->next = head;
    head = lnk;
}

template<class T>
T List<T>::get()
{
    if (head == 0)
        throw Underflow(); // Underflow 是我的异常类

    Link* p= head;
    head = p->next;
    p->next = free;
    free = p;
    return p->val;
}
```

支持函数（这里是私有的）的定义照例更复杂一些：

```
template<class T>
typename List<T>::Link* List<T>::get_free()
{
    if (free == 0) {
        // ... 分配一个新块，将其 Link 放在空闲链表中 ...
    }
    Link* p = free;
    free = free->next;
    return p;
}
```

通过在一个成员函数定义中使用 List<T>::，我们就可以进入 List<T> 作用域。但是，由于 get_free() 的返回类型是在函数名 List<T>::get_free() 之前给出的，必须使用全名 List<T>::Link 而不是简写 Link。另一种替代方法是使用返回类型后置语法（见 12.1.4 节）：

```
template<class T>
auto List<T>::get_free() -> Link*
{
    // ...
}
```

非成员函数不能进行这样的访问（友元除外）：

```
template<typename T>
void would_be_meddler(List<T>* p)
{
    List<T>::Link* q = 0;                    // 错误：List<T>::Link 是 private 的
    // ...
    q = p->free;                             // 错误：List<T>::free 是 private 的
    // ...
    if (List<T>::Chunk::chunk_size > 31) {   // 错误：List<T>::Chunk::chunk_size 是 private 的
        // ...
    }
}
```

在一个 class 中，成员默认是 private 的；在一个 struct 中，成员默认是 public 的（见 16.2.4 节）。

成员类型的一种明显的替代方法是将类型放在包含类的名字空间中（而不是类内）。例如：

```
template<class T>
struct Link2 {
    T val;
    Link2* next;
};

template<class T>
class List {
private:
    Link2<T>* free;
    // ...
};
```

Link 用 List<T> 的参数 T 隐式参数化。对 Link2，我们必须显式参数化。

如果一个成员类型不依赖于所有模板类参数，则使用非成员版本可能更好；参见 23.4.6.3 节。

如果嵌入类是不可取的，但被嵌入的类离开嵌入类后自身又没什么用处，那么将（先前的）成员类声明为（先前的）包含类的 friend（见 19.4.2 节）可能是一个好主意：

```
template<class T> class List;

template<class T>
class Link3 {
    friend class List<T>;    // 只有 List<T> 能访问 Link3<T>
    T val;
    Link3* next;
};
template<class T>
class List {
private:
    Link3<T>* free;
    // ...
};
```

对于一个类中多个访问说明符（见 8.2.6 节）分隔开的几段，编译器可能重排它们的顺序。例如：

```
class S {
public:
    int m1;
public:
    int m2;
};
```

编译器可能决定在 S 对象的内存布局中将 m2 放在 m1 之前。这种重排可能会令程序员感到吃惊，而且它依赖于 C++ 具体实现，因此，除非有充足理由，否则不要使用多个访问说明符。

20.5.1 protected 成员

当设计一个类层次时，有时我们提供的函数是供派生类的实现者而非普通用户所用的。例如，我们可能为派生类实现者提供一个（高效的）不进行检查的访问函数，为其他人提供一个（安全的）进行检查的访问函数。我们可以通过将不检查的版本声明为 protected 来达到这一目的。例如：

```
class Buffer {
public:
    char& operator[](int i);    // 检查访问
    // ...
protected:
    char& access(int i);        // 不检查访问
    // ...
};

class Circular_buffer : public Buffer {
public:
    void reallocate(char* p, int s);        // 改变位置和大小
    // ...
};

void Circular_buffer::reallocate(char* p, int s)// 改变位置和大小
{
    // ...
    for (int i=0; i!=old_sz; ++i)
        p[i] = access(i);       // 没有不必要的检查
    // ...
}
void f(Buffer& b)
{
    b[3] = 'b';             // 正确（会检查）
    b.access(3) = 'c';      // 错误：Buffer::access() 是 protected 的
}
```

另一个例子请见 21.3.5.2 节中的 Window_with_border。

一个派生类只能对自身类型的对象访问其基类的保护成员：

```
class Buffer {
protected:
    char a[128];
    // ...
};

class Linked_buffer : public Buffer {
    // ...
};

class Circular_buffer : public Buffer {
    // ...
    void f(Linked_buffer* p)
    {
        a[0] = 0;       // 正确：访问 Circular_buffer 自己的保护成员
```

```
            p->a[0] = 0;    // 错误：访问不同类型的保护成员
        }
    };
```

这能防止一些微妙错误，如一个派生类破坏属于另一个派生类的数据。

20.5.1.1 使用 protected 成员

简单的私有 / 公有数据隐藏模型能很好地用于具体类型（见 16.3 节）。但当使用派生类时，一个类就有两种使用者：派生类和“一般公众”。实现类操作的成员和友元函数是代表这些使用者来操作类对象的。私有 / 公有模型令程序员能清晰地区分实现者和一般公众，但它不能提供一种方法来满足派生类的特殊需要。

声明为 protected 的成员比声明为 private 的成员更容易误用。特别是，将数据成员声明为 protected 通常是一个设计错误。将所有派生类要用到的大量数据都放到一个公共类中令数据更易被破坏。更糟糕的是，类似公有数据，保护数据难以重组，因为没有什么好方法能找到所有使用保护数据的代码。保护数据从而成为软件维护中的一个问题。

幸运的是，你无须使用保护数据；在类中，成员默认是 private 的，而这通常是更好的选择。以我的经验，总是有其他替代方法，从而无须将派生类要用到的大量数据都放到一个公共基类中。

但是，这些反对理由对保护成员函数都不成立：protected 是说明操作用于派生类中的一种很好的方法。21.2.2 节中的 Ival_slider 就是这样一个例子。假如这个例子中的实现类被声明为 private，进一步的派生就不可能了。而另一方面，令基类提供 public 的实现细节，又会招致错误和误用。所以 protected 是最好的选择。

20.5.2 访问基类

类似成员，基类也可以声明为 private、protected 或 public。例如：

```
class X : public B { /* ... */ };
class Y : protected B { /* ... */ };
class Z : private B { /* ... */ };
```

不同的访问说明符满足不同设计需求：

- public 派生令派生类成为基类的一个子类型。例如，X 是一种 B。这是最常见的派生形式。
- private 基类最有用的情形就是当我们定义一个类时将其接口限定为基类，从而可提供更强的保障。例如，B 是 Z 的一个实现细节。25.3 节设计了一个指针 Vector 模板，它是通过向基类 Vector<void*> 添加类型检查而实现的，这就是 private 基类的一个很好的例子。
- protected 基类在类层次中很有用，其中进一步的派生是常态。类似 private 派生，protected 派生也用于表示实现细节。21.2.2 节中的 Ival_slider 就是一个好例子。

基类的访问说明符可以省略，此时，对 class，基类默认为私有的；对 struct，基类默认是公有的。例如：

```
class XX : B { /* ... */ };    // B 是一个公有基类
struct YY : B { /* ... */ };   // B 是一个公有基类
```

人们期望基类是 public 的（即，表达一种子类型关系），因此 class 在缺少访问说明符时的结果可能令人惊讶，而 struct 就不会。

基类的访问说明符控制基类成员的访问以及从派生类类型到基类类型的指针和引用转换。考虑一个类 D 派生自基类 B 的情况：

- 如果 B 是一个 private 基类，其公有和保护成员只能被 D 的成员函数和友元函数使用。只有 D 的友元和成员可以将一个 D* 转换为一个 B*。
- 如果 B 是一个 protected 基类，其公有和保护成员只能被 D 的成员函数和友元函数以及 D 的派生类的成员函数和友元函数使用。只有 D 的成员和友元以及 D 的派生类的成员和友元可以将一个 D* 转换为一个 B*。
- 如果 B 是一个 public 基类，其公有成员可被任何函数访问。而且，其保护成员可被 D 的成员和友元以及 D 的派生类的成员和友元使用。任何函数都可以将一个 D* 转换为一个 B*。

这基本上就是成员访问规则（见 20.5 节）的重复。当设计一个类时，我们为基类选择访问控制的方式与成员一样，具体示例请参见 21.2.2 节中的 lval_slider。

20.5.2.1 多重继承与访问控制

在一个多重继承框架（见 21.3 节）中，如果通过多条路径都可到达一个基类，则，若任何一条路径中此基类可访问，那么在派生类中此基类就可以访问。例如：

```
struct B {
    int m;
    static int sm;
    // ...
};

class D1 : public virtual B { /* ... */ };
class D2 : public virtual B { /* ... */ };
class D12 : public D1, private D2 { /* ... */ };

D12* pd = new D12;
B* pb = pd;          // 正确：可访问——通过 D1
int i1 = pd->m;      // 正确：可访问——通过 D1
```

即使单一实体有多条可达路径，我们仍可以无二义性地引用它。例如：

```
class X1 : public B { /* ... */ };
class X2 : public B { /* ... */ };
class XX : public X1, public X2 { /* ... */ };

XX* pxx = new XX;
int i1 = pxx->m;     // 二义性错误：XX::X1::B::m 还是 XX::X2::B::m？
int i2 = pxx->sm;    // 正确：在一个 XX 中只有唯一的 B::sm（sm 是一个静态成员）
```

20.5.3 using 声明与访问控制

using 声明（见 14.2.2 节和 20.3.5 节）并不能用来获得额外的信息访问权，它只是一种令可访问信息更便于使用的机制。另一方面，如果数据可访问，可将访问权授予其他使用者。例如：

```
class B {
private:
    int a;
protected:
    int b;
public:
```

```
    int c;
};

class D : public B {
public:
    using B::a;          // 错误：B::a 是私有的
    using B::b;          // 通过 D 令 B::b 可公有访问
};
```

当 using 声明与私有和保护派生组合使用时，可为类提供的某些而非全部特性给出其接口。例如：

```
class BB : private B {    // 提供对 B::b 和 B::c 的访问权，但未提供 B::a 的访问权
public:
    using B::b;
    using B::c;
};
```

参见 20.3.5 节。

20.6　成员指针

成员指针是一种类似偏移量的语言构造，允许程序员间接引用类成员。->* 和 .* 可以说是最特殊也最少使用的 C++ 运算符。使用 ->，我们可以访问一个类成员 m，比如说 m*p->m。使用 ->*，我们可以访问一个类成员，其名字保存在一个成员指针中，比如说 ptom:p->*ptom。这允许我们通过函数传递来的成员名访问类成员。在两种情况下，p 都必须是指向恰当类的对象的指针。

成员指针不能赋予 void* 或任何其他普通指针。空指针（如 nullptr）可赋予成员指针，表示“无成员”。

20.6.1　函数成员指针

很多类提供简单的、非常通用的接口，会以多种不同的方式被调用。例如，很多“面向对象”用户界面会定义一组请求，每个在屏幕上呈现的对象都准备好响应这些请求。而且，这些请求由程序直接或间接提供。考虑这一思想的一个简单变形：

```
class Std_interface {
public:
    virtual void start() = 0;
    virtual void suspend() = 0;
    virtual void resume() = 0;
    virtual void quit() = 0;
    virtual void full_size() = 0;
    virtual void small() = 0;

    virtual ~Std_interface() {}
};
```

每个操作的具体含义由调用它的对象所定义。通常，在发出请求的人或程序与接收请求的对象之间还有一层软件。理想情况下，这个中间层软件不必了解 resume() 和 full_size() 这样具体请求的相关信息。否则，每当操作发生改变时，中间层软件也必须更新。因此，这种中间层只是简单地将表示操作的数据从提出请求的源发送到接收请求的对象。

一种简单的实现方法是发送一个表示将被调用的操作的 string。例如，为了调用

suspend()，我可以发送字符串"suspend"。但是，这种方法需要有人创建字符串，还要有人解码字符串来确定对应的操作——如果存在的话。通常，这种间接方式很烦人。一种替代方法是，我们可以简单地发送一个表示操作的整数。例如，可能用2表示suspend()。但是，使用整数可能很方便机器进行处理，但会令人相当困惑。我们还是必须编写代码来确定2表示suspend()，然后调用suspend()。

作为替代，我们可以使用成员指针间接引用类成员。考虑Std_interface的例子。如果我希望对某个对象调用suspend()而又不必直接提及suspend()，我就需要一个引用Std_interface::suspend()的成员指针。我还需要获得希望挂起的对象的指针或引用。考虑一个简单的例子：

```
using Pstd_mem = void (Std_interface::*)();  // 成员指针类型

void f(Std_interface* p)
{
    Pstd_mem s = &Std_interface::suspend;     // 指向 suspend() 的指针
    p->suspend();                             // 直接调用
    (p->*s)();                                // 通过成员指针调用
}
```

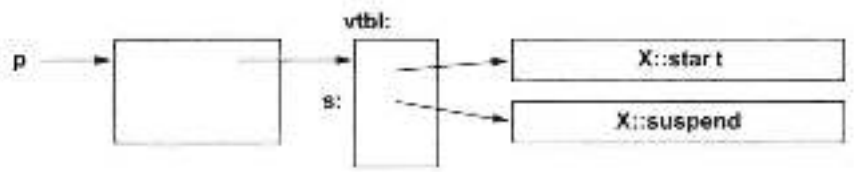
获得成员指针（pointer to member）的方法是对一个完全限定的类成员名使用地址运算符&，例如&Std_interface::suspend。我们可以使用形如X::*的声明符来声明"类X的成员指针"类型的变量。

使用别名来弥补C声明符语法可读性差的问题是一种典型用法。但是，请注意声明符X::*是如何完全匹配传统声明符*的。

成员m的指针可以与对象组合使用，运算符->*和.*允许程序员表达这种组合方式。例如，p->*m将m绑定到p指向的对象，obj.*m将m绑定到对象obj。得到的结果可以根据m的类型进行使用。将一个->*或.*操作的结果保存下来留作后用是不可能的。

自然地，如果我们知道想要调用哪个成员，就可以直接调用它，而不必使用复杂的成员指针。就像普通函数指针一样，成员函数指针的应用场景是在我们需要引用一个函数，但又不知道其名字的时候。但是，一个成员指针并不指向一片内存区域，这一点与变量指针或函数指针是不同的。它更像一个结构内部的偏移量或数组内的下标，但具体C++实现当然会考虑数据成员、虚函数、非虚函数等之间的区别。当一个成员指针与一个恰当类型的对象指针组合使用时，所产生的结果标识着一个特定对象的一个特定成员。

调用p->*s()可图示如下：



由于一个虚成员指针（本例中的s）本质上是一种偏移量，因此它不依赖于某个对象在内存中的位置。这样，我们就可以在不同地址空间之间传递一个虚成员指针，只要两个空间中使用相同的对象布局即可。类似普通函数指针，非虚成员函数指针不能在不同地址空间之间传递。

注意，通过函数指针调用的函数可能是virtual的。例如，通过函数指针调用suspend()时，我们调用的是某个对象的suspend()，该对象就是成员函数指针所应用的对

象。这是成员函数指针非常重要的一方面。

当编写一个解释器程序时，我们可能使用成员指针来调用用字符串表示的函数：

```
map<string,Std_interface*> variable;
map<string,Pstd_mem> operation;

void call_member(string var, string oper)
{
    (variable[var]->*operation[oper])(); // var.oper()
}
```

一个 static 成员不关联某个特定对象，因此 static 成员的指针就是一个普通指针。例如：

```
class Task {
    // ...
    static void schedule();
};

void (*p)() = &Task::schedule;          // 正确
void (Task::* pm)() = &Task::schedule;  // 错误：普通指针赋予成员指针
```

数据成员指针将在 20.6.2 节中介绍。

20.6.2　数据成员指针

自然地，成员指针的概念可用于数据成员和带参数及返回类型的成员函数。例如：

```
struct C {
    const char* val;
    int i;

    void print(int x) { cout << val << x << '\n'; }
    int f1(int);
    void f2();
    C(const char* v) { val = v; }
};

using Pmfi = void (C::*)(int);   // C 的成员函数指针，接收 int
using Pm = const char* C::*;     // C 的 char* 数据成员的指针
void f(C& z1, C& z2)
{
    C* p = &z2;
    Pmfi pf = &C::print;
    Pm pm = &C::val;

    z1.print(1);
    (z1.*pf)(2);
    z1.*pm = "nv1 ";
    p->*pm = "nv2 ";
    z2.print(3);
    (p->*pf)(4);

    pf = &C::f1;    // 错误：返回类型不匹配
    pf = &C::f2;    // 错误：参数类型不匹配
    pm = &C::i;     // 错误：类型不匹配
    pm = pf;        // 错误：类型不匹配
}
```

函数指针的类型检查与其他类型一样。

20.6.3 基类和派生类成员

一个派生类至少包含从基类那里继承来的成员，通常还包含其他成员。这意味着我们可以安全地将一个基类成员指针赋予一个派生类成员指针，但反方向赋值则不行。这一特性常被称为逆变性（contravariance）。例如：

```
class Text : public Std_interface {
public:
        void start();
        void suspend();
        // ...
        virtual void print();
private:
        vector s;
};

void (Std_interface::* pmi)() = &Text::print;    // 错误
void (Text::*pmt)() = &Std_interface::start;     // 正确
```

这一逆变性规则看起来与另一规则是相反的：我们可以将一个派生类指针赋予其基类的指针。实际上，两个规则都是为了提供基本保障：一个指针永远不应指向这样的对象——不能提供指针所承诺的最基本的属性。在本例中，Std_interface::* 可以应用于任意 Std_interface，大多数这种对象可能不是 Text 类型的。因此，它们不包含成员 Text::print，而这是我们用来初始化 pmi 的。编译器拒绝了初始化，从而避免发生一次运行时错误。

20.7 建议

[1] 避免使用类型域；20.3.1 节。
[2] 通过指针和引用访问多态对象；20.3.2 节。
[3] 使用抽象类，以便聚焦于清晰接口的设计应该提供什么；20.4 节。
[4] 在大型类层次中用 override 显式说明覆盖；20.3.4.1 节。
[5] 谨慎使用 final；20.3.4.2 节。
[6] 使用抽象类说明接口；20.4 节。
[7] 使用抽象类保持实现细节和接口分离；20.4 节。
[8] 如果一个类有虚函数，那么它也应该有一个虚析构函数；20.4 节。
[9] 抽象类通常不需要构造函数；20.4 节。
[10] 优先选择 private 成员用于类的细节实现；20.5 节。
[11] 优先选择 public 成员用于接口；20.5 节。
[12] 仅在确实需要时才使用 protected 成员，且务必小心使用；见 20.5.1.1 节。
[13] 不要将数据成员声明为 protected；20.5.1.1 节。

第 21 章
The C++ Programming Language, Fourth Edition

类 层 次

抽象就是选择性无视。

——安德鲁·克尼格

- 引言
- 设计类层次
  实现继承；接口继承；替代实现方式；定位对象创建
- 多重继承
  多重接口；多重实现类；二义性解析；重复使用基类；虚基类；重复基类与虚基类
- 建议

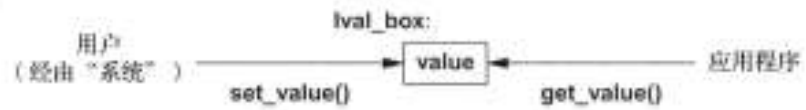
21.1 引言

本章的目标主要是阐述设计技术而非语言功能。本章的示例源于用户界面设计，但是我不会涉及图形用户界面（GUI）系统中常见的事件驱动技术。过分关注如何将屏幕操作转化成对成员函数的调用无益于学习设计类层次的主题，而且会分散读者的注意力：因为它本身就是一个很有趣、很重要的问题。要想理解 GUI，读者可以从众多 C++ GUI 标准库中挑一个出来仔细研究。

21.2 设计类层次

考虑一个简单的设计问题：为程序（“应用程序”）提供一种由用户输入整数的途径。显然有很多措施可以帮助我们实现这一目标。我们希望不受这种多样性的干扰，同时保留在多种设计方案中进行选择的权力。首先定义简单输入操作的程序模型。

我们使用类 Ival_box（“整数值输入框”）指定程序可接受的输入值范围。程序可以从 Ival_box 获取它的值，并在必要的时候给用户适当提示。此外，当用户改变了输入的值时，程序能从 Ival_box 中获取它：



因为实现上述思想的方法有很多，所以必然存在很多不同的 Ival_box，比如滑块、用户直接键入数字的普通文本框、刻度盘以及声音交互，等等。

最通用的方法是为应用程序建立一个“虚拟用户界面系统”。该系统提供与主流用户界面系统类似的服务。出于应用程序代码可移植性的考虑，该系统应该尽量兼容各种环境。显然，通过其他途径也可以实现应用程序与用户界面系统的分离，我之所以选择这种方法是因为：第一，它具有很好的通用性；第二，我可以通过它展示很多实现技术和设计时的考虑；第三，这些技术是实现一个“真实的”用户界面系统必然会用到的；第四，也是最重要的一

点。即使跳出界面系统这一相对比较窄的领域，这些技术也是非常有用的。

本章不涉及任何将用户操作（事件）映射为标准库调用的内容，也不考虑多线程 GUI 系统中锁的问题。

21.2.1 实现继承

我们的第一个方案是使用实现继承的类层次（常见于旧程序代码）。

类 Ival_box 定义了所有 Ival_box 都需要的基本接口以及默认实现，特定的 Ival_box 可以用它们自己的版本覆盖 Ival_box 提供的默认版本。此外，我们还声明了实现基本概念所需的数据：

```
class Ival_box {
protected:
    int val;
    int low, high;
    bool changed {false};          // 用于记录 set_value() 改变
public:
    Ival_box(int ll, int hh) :val{ll}, low{ll}, high{hh} { }

    virtual int get_value() { changed = false; return val; }      // 供应用程序使用
    virtual void set_value(int i) { changed = true; val = i; }    // 供用户使用
    virtual void reset_value(int i) { changed = false; val = i; }  // 供应用程序使用
    virtual void prompt() { }
    virtual bool was_changed() const { return changed; }

    virtual ~Ival_box() {};
};
```

我并未特别认真地考虑上面这些函数的默认实现，它们只要起到说明函数语义的作用就可以了。例如，一个真正的类至少应该进行边界检查。

“Ival 类”的用法如下所示：

```
void interact(Ival_box* pb)
{
    pb->prompt(); // 请求用户
    // ...
    int i = pb->get_value();
    if (pb->was_changed()) {
        // ... 新值，执行某些操作 ...
    }
    else {
        // ... 旧值，可以忽略 ...
    }
}

void some_fct()
{
    unique_ptr<Ival_box> p1 {new Ival_slider{0,5}};    // Ival_slider 派生自 Ival_box
    interact(p1.get());

    unique_ptr<Ival_box> p2 {new Ival_dial{1,12}};
    interact(p2.get());
}
```

绝大多数应用程序使用 Ival_box 的方式与 interact() 类似。此时，应用程序无须深入了解

Ival_box 的众多可能变种。这种特殊类的相关信息都被隔离在少数几个用于创建此类对象的函数之内，从而将用户与派生类的实现隔离开来。大多数代码都并不在意存在不同种类的 Ival_box 这一事实。

我用 unique_ptr（见 5.2.1 节和 34.3.1 节）确保 Ival_box 对象最终被释放（delete）。

为了简化讨论，我们不考虑程序等待输入的问题。可能程序确实在 get_value() 中等待用户输入（比如用 get() 作用于一个 future，见 5.3.5.1 节），也可能程序用事件辅助 Ival_box 并且准备响应回调，又或者程序为 Ival_box 分配一个线程并在稍后请求该线程的状态。这样的讨论在用户界面系统的设计中非常关键，但是如果我们在这里讨论这些细节的话，只会干扰我们对程序设计技术和语言功能的理解。这里描述的设计技术和语言功能不仅仅对用户界面有效，它们还可以应用于很多其他问题。

我们用 Ival_box 的派生类表示不同种类的 Ival_box，例如：

```
class Ival_slider : public Ival_box {
private:
    // ... 用于定义滑块的图形元素 ...
public:
    Ival_slider(int, int);
    int get_value() override; // 接受用户输入的数据，将其存入 val
    void prompt() override;
};
```

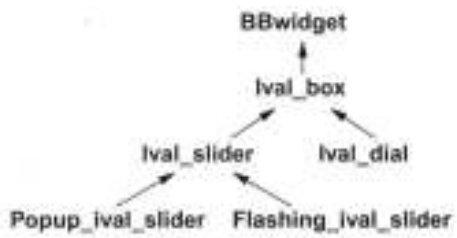
Ival_box 的数据成员被声明成 protected，这样派生类就可以访问它了。Ival_slider::get_value() 可以直接在 Ival_box::val 中存入一个值。protected 的成员可以被类的成员以及派生类的成员访问，但是不能被一般用户访问（见 20.5 节）。

除了 Ival_slider 之外，我们还需要定义 Ival_box 概念的其他变体。这些变体包括 Ival_dial（可以通过旋钮的方式选择值）、Flashing_Ival_slider（当需要 prompt() 的时候闪烁）和 Popup_Ival_slider（通过出现在某处重要的位置来响应 prompt()，这样用户就不能置之不理了）。

我们从哪儿获取图形元素呢？绝大多数用户界面系统都提供了一个专门定义屏幕实体属性的类。因此，如果使用“Big Bucks Inc.”的系统，就必须把我们的 Ival_slider、Ival_dial 等类变成一种 BBwidget。最简单的做法是改写 Ival_box，令其派生自 BBwidget。此时，我们的所有类都会继承 BBwidget 的全部属性。例如，每个 Ival_box 都能置于屏幕之上、遵守图形化风格规则、可以放大缩小、可以来回拖曳，等等，一切都按照 BBwidget 系统设置的标准进行。我们的类层次如下所示：

```
class Ival_box : public BBwidget { /* ... */ };    // 改写，使其可以使用 BBwidget
class Ival_slider : public Ival_box { /* ... */ };
class Ival_dial : public Ival_box { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

或者表示为下面的图形：



21.2.1.1 评价

上述设计有很多优点，对于很多问题来说这种层次体系是不错的解决方案。然而，它也有些细节之处值得改进。

我们把 BBwidget 作为 Ival_box 的基类，这种做法似乎不太妥当（尽管在现实世界中很常见）。事实上，BBwidget 的用法并不在我们讨论的 Ival_box 的基本概念之内，它只是一种实现细节。从 BBwidget 派生 Ival_box 的做法强行把一种实现细节拔高到整个设计决策的最高层级。乍一看能说得通，比如，我们从一个组织如何构建其业务的角度出发使用了“Big Bucks Inc.”定义的环境。但是，如果我们希望 Ival_box 也可以服务于“Imperial Bananas”“Liberated Software”和“Compiler Whizzes”的系统会遇到怎样的情况呢？我们必须为程序维护 4 个不同版本的类：

```
class Ival_box : public BBwidget { /* ... */ };        //BB 版
class Ival_box : public CWwidget { /* ... */ };        //CW 版
class Ival_box : public IBwidget { /* ... */ };        //IB 版
class Ival_box : public LSwindow { /* ... */ };        //LS 版
```

这会让程序员陷入版本控制的噩梦之中。

事实上，很难设计出一种简单、一致、全都是两字母前缀的通用模式。更常见的情况是来自于不同生产者的库通常位于不同的名字空间中，并且对于同一个概念采用完全不同的命名方式，比如 BigBucks::Widget、Wizzies::control 和 LS::window 等。但是这一情况不会影响我们关于类层次的讨论，因此我没有过多在意命名和名字空间的问题。

另外一个问题是每个派生类都会共享 Ival_box 声明的基本数据。这些数据属于实现的细节，但是不经意间混入到 Ival_box 的接口中。从实践的角度出发，很多情况下这样的数据是错误的。例如，Ival_slider 无须专门存储数据。当有人执行 get_value() 时，很容易就能通过滑块的位置计算出来。通常情况下，同时维护两组内在关联紧密的数据等同于自找麻烦。迟早有人会不小心破坏这两组数据之间的同步性。而且经验表明，初级程序员常常会乱用受保护的成员，从而带来很多维护方面的问题。我们最好保持数据成员是私有的，这样派生类的作者就不能随意使用它们了。更好的做法是把数据放在派生类的内部使其定义可以尽量契合实际的需要，并且能保证无关的派生类之间不会发生不必要的联系。在绝大多数情况下，受保护的接口应该仅包含函数、类型和常量。

从 BBwidget 派生的好处是它使得 Ival_box 的用户可以使用 BBwidget 提供的功能。不幸的是，当 BBwidget 有任何改动时，用户都不得不重新编译甚至重写他们的代码以适应这些改动。对于绝大多数 C++ 的实现来说，只要基类的大小发生了改变，所有派生类就必须重新编译。

最后，我们的程序必须运行在一种混合环境中，来自不同用户界面系统的窗口同时存在。当两种系统以某种方式共享同一屏幕或者我们的程序需要与来自不同系统的用户通信时，上述情况都会发生。即使我们把几个用户界面系统“连接”在一起，令其共同作为基类并且提供统一的 Ival_box 接口，也难以完美地解决上述问题。灵活性不足是最大的障碍。

21.2.2 接口继承

既然如此，我们就从头开始建立一种新的类层次以解决传统层次体系存在的问题：

[1] 用户界面系统应该作为一种实现细节对用户隐藏起来，因为用户根本不必了解它。

[2] Ival_box 类不应含有任何数据。

［3］ 用户界面系统的改变不应导致使用了 Ival_box 家族类的代码重新编译。

［4］ 面向不同界面系统的 Ival_box 能在我们的程序中共存。

很多方法都有助于实现上述目标。在这里，我介绍其中的一种，它能用 C++ 完美地实现。首先，指定 Ival_box 作为纯粹的接口：

```
class Ival_box {
public:
        virtual int get_value() = 0;
        virtual void set_value(int i) = 0;
        virtual void reset_value(int i) = 0;
        virtual void prompt() = 0;
        virtual bool was_changed() const = 0;
        virtual ~Ival_box() { }
};
```

这比 Ival_box 原来的声明清楚多了。它去掉了数据以及那些过分简单的成员函数实现。既然没有任何数据需要初始化，因此构造函数也不需要了。相应地，我添加了一个虚析构函数以便将来清除派生类中定义的数据。

Ival_slider 的定义是：

```
class Ival_slider : public Ival_box, protected BBwidget {
public:
        Ival_slider(int,int);
        ~Ival_slider() override;

        int get_value() override;
        void set_value(int i) override;
        // ...
protected:
        // ... 此处覆盖 BBwidget 的虚函数
        // 比如 BBwidget::draw(), BBwidget::mouse1hit()...
private:
        // ... 滑块所需的数据 ...
};
```

派生类 Ival_slider 继承了抽象类 Ival_box，因此 Ival_slider 必须实现其基类的纯虚函数。同时，Ival_slider 也继承了 BBwidget，因此需要实现后者的纯虚函数。因为 Ival_box 提供了派生类的接口，所以派生方式是 public。相反，BBwidget 仅用于辅助实现，因此它的派生方式是 protected（见 20.5.2 节）。这意味着 Ival_slider 的用户无权直接使用 BBwidget 定义的功能。Ival_slider 提供的接口包含继承自 Ival_box 的部分，以及 Ival_slider 显式声明的部分。我使用 protected 派生而非更严格的（通常也更安全的）private 派生，它使得 Ival_slider 的派生类也可以访问 BBwidget。因为这个“窗口部件层次体系”规模较大且情况复杂，所以使用显式的 override 可以避免混淆。

直接从多个类中派生称为多重继承（multiple inheritance，见 21.3 节）。请注意，Ival_slider 必须覆盖 Ival_box 和 BBwidget 中的函数。因此，它必须直接地或者间接地派生自这两个类。如 21.2.1.1 节所述，我们可以通过令 BBwidget 作为 Ival_box 的基类来让 Ival_slider 间接地派生自 BBwidget，但是这么做会产生副作用。类似地，让“实现类”BBwidget 作为 Ival_box 的成员也不可行，因为一个类无法覆盖其成员的虚函数。用 Ival_box 的 BBwidget* 成员表示窗口是一种完全不同的实现思路，并且会引入额外的开销。

对于有的人来说，“多重继承”这个词看起来非常复杂，甚至有点可怕。但是，用一个

基类表示实现细节，用另一个基类表示接口（抽象类）的做法对于所有支持继承和编译时接口检查的编程语言来说都是非常常见的。特别是，抽象类 Ival_box 的用法几乎与 Java 或者 C# 接口的用法完全一致。

有趣的是，上面关于 Ival_slider 的声明允许应用程序代码与之前保持一致，不必做任何改变。我们只是用一种更加符合逻辑的做法重新构造了实现细节而已。

很多类都需要在对象完全失效前进行清理。抽象类 Ival_box 不清楚它的派生类是否需要这样的清理操作，但是它必须假定它是需要的。为了确保清理工作顺利完成，我们在基类中定义了一个 Ival_box::~Ival_box()，然后在派生类中以合理的方式覆盖它。例如：

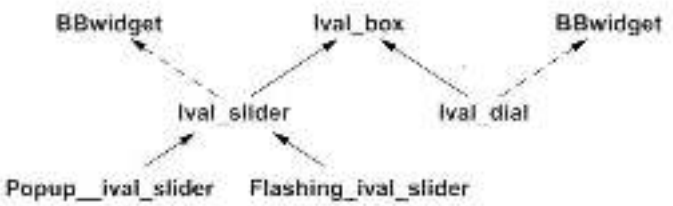
```
void f(Ival_box* p)
{
    // ...
    delete p;
}
```

delete 运算符显式地销毁 p 所指的对象。我们无法确切地了解 p 所指的对象到底属于哪个类，但是得益于 Ival_box 的虚析构函数的存在，系统会正确执行析构函数定义的清理操作。

Ival_box 层次可以定义为：

```
class Ival_box { /* ... */ };
class Ival_slider
    : public Ival_box, protected BBwidget { /* ... */ };
class Ival_dial
    : public Ival_box, protected BBwidget { /* ... */ };
class Flashing_ival_slider
    : public Ival_slider { /* ... */ };
class Popup_ival_slider
    : public Ival_slider { /* ... */ };
```

或者表示为下面的图形：



其中，我用虚线表示受保护的继承（见 20.5.1 节）。一般用户无权访问受保护的基类，因为后者属于实现的一部分。

21.2.3 替代实现方式

与一开始的版本相比，上面这个设计更加清晰，也更容易维护，同时在效率方面也没有损失。但是，它仍然难以解决版本控制的问题：

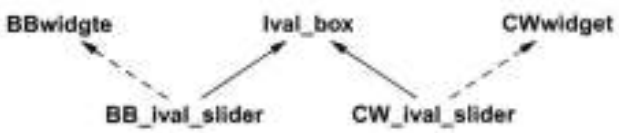
```
class Ival_box { /* ... */ }; // 像原始
class Ival_slider
    : public Ival_box, protected BBwidget { /* ... */ }; // 基于 BB
class Ival_slider
    : public Ival_box, protected CWwidget { /* ... */ }; // 基于 CW
// ...
```

我们无法让为 BBwidget 设计的 Ival_slider 与为 CWwidget 设计的 Ival_slider 共存，即使

这两个用户接口系统本身能够共存也不行。一种最面易见的解决方案是用不同的名字定义几个独立的 Ival_slider 类版本：

```
class Ival_box { /* ... */ };
class BB_ival_slider
        : public Ival_box, protected BBwidget { /* ... */ };
class CW_ival_slider
        : public Ival_box, protected CWwidget { /* ... */ };
// ...
```

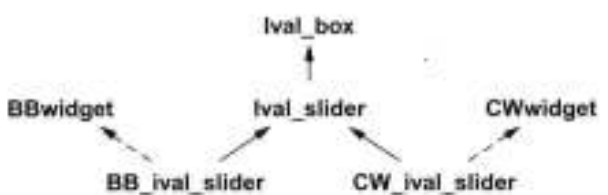
或者表示为下面的图形：



为了展现我们以应用为导向的 Ival_slider 类的更多实现细节，不妨先从 Ival_box 派生出一个抽象的 Ival_slider，再从中派生一个系统特定的 Ival_slider：

```
class Ival_box { /* ... */ };
class Ival_slider
        : public Ival_box { /* ... */ };
class BB_ival_slider
        : public Ival_slider, protected BBwidget { /* ... */ };
class CW_ival_slider
        : public Ival_slider, protected CWwidget { /* ... */ };
// ...
```

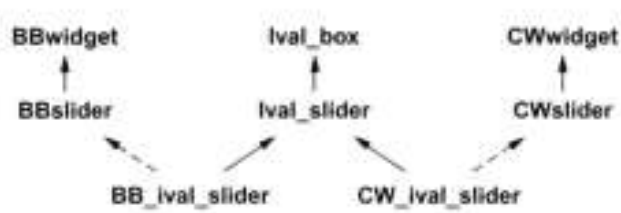
或者表示为下面的图形：



通常情况下，我们可以在实现层次中添加更多专有类以达到进一步优化的目的。例如，如果“Big Bucks Inc.”系统有一个滑块类，我们可以从 BBslider 直接派生出 Ival_slider：

```
class BB_ival_slider
        : public Ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider
        : public Ival_slider, protected CWslider { /* ... */ };
```

或者表示为下面的图形：



当我们的抽象与该实现所服务的系统所提供的版本区别不大时（这种情况并不罕见），这种提升特别有效。此时，程序设计的任务转化成在相似概念之间进行映射。随之而来的结果是

人们很少从 BBwidget 这样的通用基类中直接派生了。

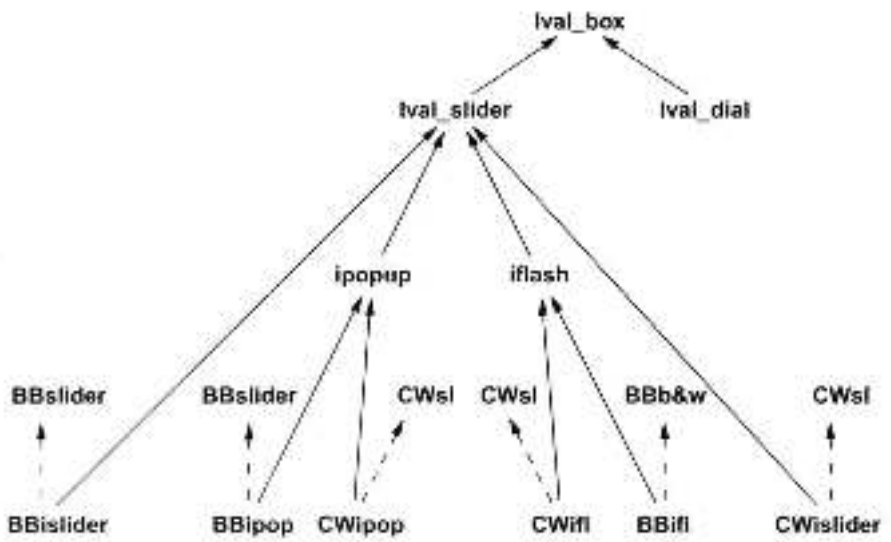
在完整的层次体系中将包含我们原始的以应用为导向的概念化的接口层次体系，它以派生类的形式出现：

```
class Ival_box { /* ... */ };
class Ival_slider
        : public Ival_box { /* ... */ };
class Ival_dial
        : public Ival_box { /* ... */ };
class Flashing_ival_slider
        : public Ival_slider { /* ... */ };
class Popup_ival_slider
        : public Ival_slider { /* ... */ };
```

接下来，该层次体系面向多种用户接口系统的实现也表达为派生类：

```
class BB_ival_slider
        : public Ival_slider, protected BBslider { /* ... */ };
class BB_flashing_ival_slider
        : public Flashing_ival_slider, protected BBwidget_with_bells_and_whistles { /* ... */ };
class BB_popup_ival_slider
        : public Popup_ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider
        : public Ival_slider, protected CWslider { /* ... */ };
// ...
```

该层次体系可以用下面的图形表示，其中用到了一些简写形式：



原来的 Ival_box 未做任何改动，我们只是在它周围添加了一些实现类。

21.2.3.1 评价

抽象类的设计非常灵活，而且处理起来像基于定义用户界面系统的常见基类的设计一样简单。在后一种设计中，窗口类是继承树的根。而对于前一种实现来说，原始的应用类层次负责提供实现，它不做任何改动直接作为整个类层次的根。从应用的角度来看，这两种设计是等价的，因为它们的编码工作几乎没有区别。基于其中任意一种设计，你都可以在不了解窗口实现细节的情况下使用 Ival_box 类家族。例如，当从一个类层次切换到另一个类层次时，无须重写 21.2.1 节的 interact()。

不论基于哪一种设计，当用户界面系统的公共接口发生改变时都必须重写每个 Ival_box 类的实现。然而，在抽象类设计中，几乎所有用户代码都不受实现层次改动的影响，也无须重新编译。对于实现层次的提供者来说，当发布一个新的“几乎完全兼容的”版本时，这一点显得非常重要。此外，与传统的层次体系相比，抽象类层次的用户不太可能陷入某一专有的实现中无法抽身。Ival_box 抽象类应用层次体系的用户无法使用来自实现的功能，因为只有在 Ival_box 层次体系中显式指定的功能才是可访问的；不存在任何从实现相关的基类中隐式继承的东西。

因此，我们的最终结论是一个系统应该用抽象类层次表示，而用传统的层次体系实现。换句话说：

- 用抽象类支持接口继承（见 3.2.3 节和 20.1 节）。
- 用带有虚函数实现的基类支持实现继承（见 3.2.3 节和 20.1 节）。

21.2.4　定位对象创建

我们可以用 Ival_box 接口写出应用程序的绝大部分。然后，派生接口进一步演化以提供比普通的 Ival_box 更多的功能，使得应用可以用 Ival_box、Ival_slider 等接口表达出来。在创建对象时必须使用 CW_ival_dial 和 BB_flashing_ival_slider 等实现相关的名字。我们应该尽量控制这类特定名字出现的频次，而且必须系统地管理对象创建，否则会很难定位它。

和往常一样，正确的解决方案是引入一个指示信息。有很多方式可以实现这一目标，其中一种比较简单的做法是引入一个抽象类来表示创建操作的集合：

```
class Ival_maker {
public:
    virtual Ival_dial* dial(int, int) =0;                    // 创建旋钮
    virtual Popup_ival_slider* popup_slider(int, int) =0;    // 创建弹出式滑块
    // ...
};
```

Ival_maker 为 Ival_box 类家族的每个接口都提供了一个创建对象的函数。这样的类有时被称为工厂（factory），它的函数称为虚构造函数（virtual constructor，有一定的误导性，见 20.3.6 节）。

我们用 Ival_maker 的派生类表示每个用户界面系统：

```
class BB_maker : public Ival_maker {          // 实现 BB 版本
public:
    Ival_dial* dial(int, int) override;
    Popup_ival_slider* popup_slider(int, int) override;
    // ...
};

class LS_maker : public Ival_maker {          // 实现 LS 版本
public:
    Ival_dial* dial(int, int) override;
    Popup_ival_slider* popup_slider(int, int) override;
    // ...
};
```

每个函数创建指定接口和实现类型的一个对象，例如：

```
Ival_dial* BB_maker::dial(int a, int b)
{
    return new BB_ival_dial(a,b);
}

Ival_dial* LS_maker::dial(int a, int b)
{
    return new LS_ival_dial(a,b);
}
```

基于 Ival_maker，用户无须了解具体的用户界面系统信息就能创建对象了。例如：

```
void user(Ival_maker& im)
{
    unique_ptr<Ival_box> pb {im.dial(0,99)};      // 创建适合的按钮
    // ...
}

BB_maker BB_impl;      // 针对 BB 用户
LS_maker LS_impl;      // 针对 LS 用户

void driver()
{
    user(BB_impl);     // 使用 BB
    user(LS_impl);     // 使用 LS
}
```

向这类“虚构造函数”传递参数时情况比较微妙。我们不能覆盖那些在不同派生类中用不同参数表示接口的基类函数。因此，在设计工厂类的接口时必须具有足够的前瞻性。

21.3 多重继承

如 20.1 节所述，继承可以提供下列好处之一：

- 共享接口（shared interface）：通过使用类使得重复代码较少，且代码规格统一。通常称为运行时多态（run-time polymorphism）或者接口继承（interface inheritance）。
- 共享实现（shared implementation）：代码量较少且实现代码的规格统一，通常称为实现继承（implementation inheritance）。

一个类可以综合运用这以上两种风格。

接下来，我们将探讨多重继承更广泛的应用以及与多重基类功能有关的技术问题。

21.3.1 多重接口

抽象类（比如 Ival_box，见 21.2.2 节）是最容易想到的一种表示接口的方式。对一个不含可变状态的抽象类来说，在类层次中被当做单一基类或者多重基类其实没什么差别。对潜在的二义性的解决方案将在 21.3.3 节、21.3.4 节和 21.3.5 节讨论。事实上，任何不含可变状态的类都可以作为多重继承框架中的接口出现，而且不会增加复杂性或者程序开销。最重要的一点是，不含可变状态的类可以被复制，也可以被共享。

在面向对象的程序设计中，用多重抽象类作为接口的做法非常通用（在任何含有接口的编程语言中都是如此）。

21.3.2 多重实现类

考虑对那些绕地球轨道飞行的物体进行仿真，这些物体表示为类 Satellite 的对象。

Satellite 对象包含轨道、尺寸、形状、反照率和密度参数等属性，并且提供轨道计算、属性修改等操作。在我们的定义中，卫星包括岩石、先前的空间设备残骸、通信卫星以及国际空间站。这些不同种类的卫星表示为 Satellite 派生类的对象。这些派生类各自添加一些成员和函数，并且覆盖 Satellite 的某些虚函数以更好地表达各自的含义。

假设我想用图形化的方式展示仿真的结果，并且我的图形系统的（并非不常见）策略是用一个存有图形信息的公共基类显示派生类对象。这个图形类提供在屏幕上放置和缩放图形的操作。为了达到通用、简化且隐藏实际图形系统细节的目标，我把负责图形化（或者非图形化）输出的类称为 Displayed。

接下来，我们提供一个模拟通信卫星的类 Comm_sat：

```
class Comm_sat : public Satellite, public Displayed {
public:
    // ...
};
```

或者表示为下面的图形：



除了 Comm_sat 定义的专门的操作之外，还可以把 Satellite 和 Displayed 的操作结合在一起使用。例如：

```
void f(Comm_sat& s)
{
    s.draw();              // Displayed::draw()
    Pos p = s.center();    // Satellite::center()
    s.transmit();          // Comm_sat::transmit()
}
```

类似地，我们可以给一个需要 Satellite 或者 Displayed 的函数传入 Comm_sat。例如：

```
void highlight(Displayed*);
Pos center_of_gravity(const Satellite*);

void g(Comm_sat* p)
{
    highlight(p);                  // 传递一个指向 Comm_sat 的 Displayed 部分的指针
    Pos x = center_of_gravity(p);  // 传递一个指向 Comm_sat 的 Satellite 部分的指针
}
```

这段代码采用了某些（简单）编译器技术，使得需要 Satellite 的函数与需要 Displayed 的函数看到的 Comm_sat 的部分不一样。虚函数的工作方式与往常类似，例如：

```
class Satellite {
public:
    virtual Pos center() const = 0;      // 质心
    // ...
};

class Displayed {
public:
    virtual void draw() = 0;
    // ...
```

```
};

class Comm_sat : public Satellite, public Displayed {
public:
    Pos center() const override;    // 覆盖 Satellite::center()
    void draw() override;           // 覆盖 Displayed::draw()
    // ...
};
```

这确保 Comm_sat 在调用 Comm_sat::center() 时把自己当成一个 Satellite，在调用 Comm_sat::draw() 时把自己当成一个 Displayed。

为什么我要把 Satellite 和 Displayed 当作 Comm_sat 的一部分，而不是把它们完全分离开来呢？从理论上来说，我可以为 Comm_sat 定义一个 Satellite 成员和一个 Displayed 成员，也可以为 Comm_sat 定义一个 Satellite* 成员和一个 Displayed* 成员，然后让它的构造函数负责建立正确的连接。在很多情况下，我确实会这么做。但是，本例所处的系统是基于含有虚函数的 Satellite 类以及一个单独设计的含有虚函数的 Displayed 类建立的。因此，我们应该通过派生提供自己的卫星及显示对象。尤其是，必须覆盖 Satellite 和 Displayed 各自的虚成员函数以定义你自己的对象的行为。要想多重继承含有状态和实现的基类，我们就很难避免上述情况。变通方法存在很多风险，并且不易维护。

通过多重继承的方式把两个不相关的类强行“粘连”在一起作为第三个类的实现是一种粗鲁且不那么有趣的方法，但是这种做法非常有效，所以显得比较重要。基本上，它使得程序员不必再编写那么多转发函数（以补偿我们只能覆盖基类虚函数的不足）。这项技术不会影响程序的总体设计效果，并且有时候它还会不小心暴露一些实现的细节。不过从整体上来说，没有任何技术能够做到足够完美。

我个人习惯于使用一个实现层次体系，再（在必要时）辅以几个提供接口的抽象类。这种方式比较灵活，也易于系统的演化。但是我们未必总能如愿，尤其是当需要使用现有的类，又不想对它做出任何修改时更是如此（比如，这些类属于别人的库）。

如果只用单继承的话，程序员在实现 Displayed、Satellite 和 Comm_sat 类时会受到很多限制。Comm_sat 可以是一个 Displayed，也可以是一个 Satellite，但是不能兼具二者的性质（除非 Satellite 派生自 Displayed，或者 Displayed 派生自 Satellite）。不管如何选择都必然在灵活性上有一些损失。

人们真的需要 Comm_sat 类吗？与一些人的推测相反，Satellite 的例子是真实存在而且极有价值的。的确曾有程序就是按照多重实现继承的思路构建的，而且现在仍有这样的程序出现。它的作用是研究卫星和地面指挥中心之间的通信问题。事实上，Satellite 源自某个并行任务的早期概念。基于之前的仿真，我们可以回答与通信流有关的问题，在暴风雪的恶劣环境下对地面指挥中心做出响应，权衡卫星连接与地面连接，等等。

21.3.3 二义性解析

两个基类的成员函数可能具有相同的名字，例如：

```
class Satellite {
public:
    virtual Debug_info get_debug();
    // ...
};
```

```
class Displayed {
public:
    virtual Debug_info get_debug();
    // ...
};
```

当我们使用 Comm_sat 时，上面两个函数必须区分开。具体做法是为成员名字加一个类限定符：

```
void f(Comm_sat& cs)
{
    Debug_info di = cs.get_debug();    // 错误：具有二义性
    di = cs.Satellite::get_debug();    // OK
    di = cs.Displayed::get_debug();    // OK
}
```

然而，显式消除二义性比较繁琐，解决此类问题的最佳方式是在派生类中定义一个新函数：

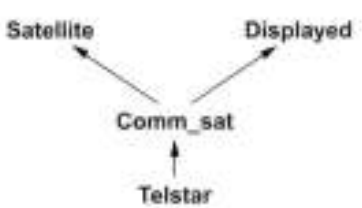
```
class Comm_sat : public Satellite, public Displayed {
public:
    Debug_info get_debug() // 覆盖 Comm_sat::get_debug() 和 Displayed::get_debug()
    {
        Debug_info di1 = Satellite::get_debug();
        Debug_info di2 = Displayed::get_debug();
        return merge_info(di1,di2);
    }
    // ...
};
```

在派生类中声明的函数会覆盖基类中所有同名及同类型的函数。通常情况下，这种效果就是我们需要的，因为在同一个类中的同一个名字不宜有多重含义。virtual 的目标是对于一个调用来说，不管我们是通过哪个接口找到函数的，它的执行效果都应该保持一致（见 20.3.2 节）。

在实现函数覆盖时，通常需要显式地指定类名以获取基类中我们需要的函数版本。一个带限定符的名字（比如 Telstar::draw）既可以表示在 Telstar 中声明的 draw，也可以表示在 Telstar 的基类中声明的 draw。例如：

```
class Telstar : public Comm_sat {
public:
    void draw()
    {
        Comm_sat::draw();          // 查找 Displayed::draw
        // ... 自己的处理 ...
    }
    // ...
};
```

或者表示为下面的图形：



如果 Comm_sat::draw 不能解析为在 Comm_sat 中声明的 draw，则编译器递归地查找 Comm_sat 的基类，也就是说，继续查找 Satellite::draw 和 Displayed::draw。如果需要的

话，还会继续查找 Satellite 和 Displayed 的基类。如果找到了精确匹配的名字，则使用它；否则，Comm_sat::draw 的结果就是未找到或者具有二义性。

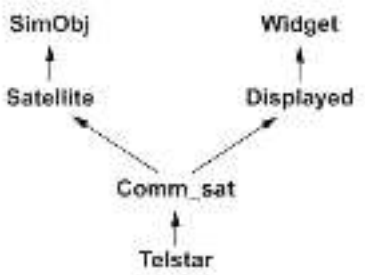
如果我在 Telstar::draw() 中使用了不加任何限定符的 draw()，则该程序将“无限”递归调用 Telstar::draw()。

我可以使用 Displayed::draw()，但是如果有人再添加一个 Comm_sat::draw() 的话代码就会有点小问题；通常情况下，指向直接基类比指向间接基类更好。如果使用 Comm_sat::Displayed::draw() 的话，会显得有些冗余。如果使用 Satellite::draw()，则结果是错误的，因为 draw 在类层次的 Displayed 分支就已经结束了。

在 get_debug() 的例子中，我们假定 Satellite 和 Displayed 至少有某些部分是相通的。不太可能出现名字、参数类型、返回值类型和语义都完全匹配的两个函数。更常见的情况是，程序以不同方式提供了相似功能，而我们需要设法把它们结合在一起以便一同使用。假设有两个类 SimObj 和 Widget，它们都没有提供我们真正想用的功能，并且我们无权修改它们；即使有些功能是我们需要的，但是它们的接口并不相容。此时，我们需要设计 Satellite 和 Displayed 作为我们的接口类，为更高层的类提供一个“映射层”：

```
class Satellite : public SimObj {
    // 把 SimObj 的功能映射到别处以便提供 Satellite
public:
    virtual Debug_info get_debug();    // 调用 SimObj::DBinf() 并抽取信息
    // ...
};

class Displayed : public Widget {
    // 把 Widget 的功能映射到别处以便显示 Satellite 的信息结果
public:
    virtual Debug_info get_debug();    // 读取 Widget 数据，构成 Debug_info
    // ...
};
```

或者表示为下面的图形：



上例非常贴切和生动，它采用的正是之前我们提到的用来消除二义性（两个类提供了一对名字相同但语义不同的函数）的技术，即增加一个专门的接口层。参考那个经典的牛仔视频游戏，它的每个类都有一个 draw() 成员函数：

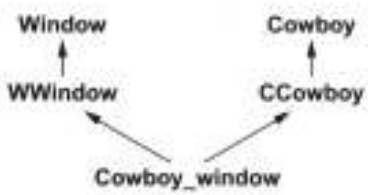
```
class Window {
public:
    void draw();    // 显示图像
    // ...
};

class Cowboy {
```

```
public:
    void draw();    // 从枪套中拔枪
    // ...
};

class Cowboy_window : public Cowboy, public Window {
    // ...
};
```

我们该如何覆盖 Cowboy::draw() 和 Window::draw() 呢？这两个函数的含义（语义）天差地别，但是名字和类型却完全一致；我们必须用两个相互独立的函数覆盖它们。任何语言功能都无法直接解决该问题，只能添加中间类：

```
struct WWindow : Window {
    using Window::Window;                          // 继承构造函数
    virtual void win_draw() = 0;                   // 强制要求派生类必须覆盖
    void draw() override final { win_draw(); }     // 显示图像
};

struct CCowboy : Cowboy{
    using Cowboy::Cowboy;                          // 继承构造函数
    virtual void cow_draw() = 0;                   // 强制要求派生类必须覆盖
    void draw() override final { cow_draw(); }     // 从枪套中拔枪
};

class Cowboy_window : public CCowboy, public WWindow {
public:
    void cow_draw() override;
    void win_draw() override;
    // ...
};
```

或者表示为下面的图形：



其实只要 Window 的设计者稍微仔细一点，把 draw() 指定为 const 的，整个问题就不复存在了。我发现这种情况相当普遍。

21.3.4　重复使用基类

如果每个类只有一个直接基类，则类层次表现为一棵树，并且每个类在树中只能出现一次。如果每个类可以有多个基类，则在层次体系中每个类有可能出现多次。考虑这样一个类，它的目标是在文件中保存状态（比如断点、调试信息、持续时间等），并在稍后修改：

```
struct Storable {    // 持久存储
    virtual string get_file() = 0;
    virtual void read() = 0;
    virtual void write() = 0;

    virtual ~Storable() { }
};
```

这样的类显然非常有用，有可能在类层次中出现多次。例如：

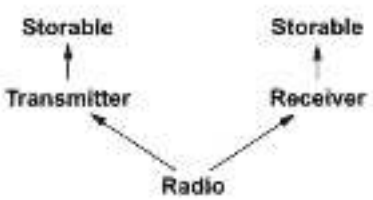
```
class Transmitter : public Storable {
public:
    void write() override;
    // ...
};

class Receiver : public Storable {
public:
    void write() override;
    // ...
};

class Radio : public Transmitter, public Receiver {
public:
    string get_file() override;
    void read() override;
    void write() override;
    // ...
};
```

基于上述代码，可能出现两种情况：

[1] 一个 Radio 对象包含两个 Storable 子对象（一个是 Transmitter 的，另一个是 Receiver 的）。

[2] 一个 Radio 对象包含一个 Storable 子对象（由 Transmitter 和 Receiver 共享）。

上述示例默认提供的是两个子对象的方式。除非有特殊说明，否则只要你把某个类作为基类，就会得到它的一份拷贝。我们可以将其表示为下面的图形：



重复基类的虚函数可以在派生类中被一个（单独的）函数覆盖。通常情况下，这个覆盖的函数先调用其基类的版本，然后执行派生类自己的操作：

```
void Radio::write()
{
    Transmitter::write();
    Receiver::write();
    // ... 写入 radio 特定的信息 ...
}
```

22.2 节将介绍从重复基类向派生类的强制类型转换。如果想在派生类中用不同的函数分别覆盖每个 write()，请参见 21.3.3 节。

21.3.5 虚基类

前面的 Radio 示例之所以能正常工作，是因为 Storable 能以一种安全、便捷和高效的方式重复使用。我们把 Storable 声明成抽象类使其提供纯粹的接口。一个 Storable 对象没有任何自己的数据，这是最简单的情况，也是完成接口和实现分离的最佳方式。事实上，要

想确定在一个 Radio 中是否有两个 Storable 子对象一定会遇到一些困难。

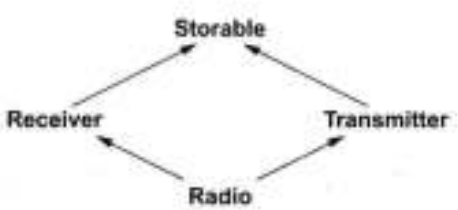
如果 Storable 含有自己的数据，我们应该如何确保它不被重复使用呢？例如，我们希望 Storable 保存用于存储对象的文件的名字：

```
class Storable {
public:
    Storable(const string& s);      // 存在名为 s 的文件中
    virtual void read() = 0;
    virtual void write() = 0;
    virtual ~Storable();
protected:
    string file_name;

    Storable(const Storable&) = delete;
    Storable& operator=(const Storable&) = delete;
};
```

既然这个 Storable 被稍微修改过，我们就必须重新设计 Radio。对象的各个部分必须共享同一个 Storable。否则，就会出现某个东西的两个部分分别派生自两个使用了不同文件的 Storable 的情况。我们应该通过把基类声明成 virtual 来避免重复：因为派生类的每个 virtual 基类都是用同一个（共享）对象表示的。例如：

```
class Transmitter : public virtual Storable {
public:
    void write() override;
    // ...
};

class Receiver : public virtual Storable {
public:
    void write() override;
    // ...
};

class Radio : public Transmitter, public Receiver {
public:
    void write() override;
    // ...
};
```

或者表示为下面的图形：



对比这幅图与 21.3.4 节 Storable 对象的图，它们体现了普通继承与虚继承的区别。在继承结构图中，每个被指定为 virtual 的基类只用该类的一个单独的对象表示。另一方面，非 virtual 基类由其子对象表示。

为什么有的人希望虚基类包含数据呢？我能想到 3 种明显的方式以实现同一个层次体系中的两个类共享数据：

[1] 令数据处于非局部作用域中（位于类的外部，作为全局变量或者名字空间的变量）。

[2] 把数据放在基类中。

[3] 在某处分配对象，然后把指针分别交给两个类。

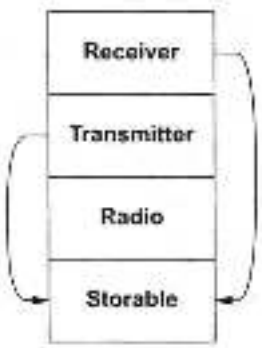
选项[1]使用非局部数据，这通常是一种糟糕的做法，我们无法控制哪些代码可以访问此类数据以及以何种方式访问此类数据。它完全破坏了封装性和局部性的概念。

选项[2]把数据放在基类中，这是最简单的做法。然而，对于单继承来说，该方案把有用的数据（以及函数）都"冒泡"到一个公共基类中，而且通常它会一直"冒泡"到继承树的根部。这意味着类层次的每个成员都获得了对此类数据的访问权。从这层意义上来说它与使用非局部数据的方式非常类似，而且遇到的问题也一样。因此，我们需要一个不是根的公共基类，即虚基类。

选项[3]通过指针共享数据帮助我们实现目标。但是，构造函数需要为共享对象分配一片内存、初始化它并且提供指针以指向共享对象。而这正是构造函数为虚基类所做的。

如果你不需要共享，大可不必使用虚基类，并且你的代码会更简单更漂亮。相反，如果你需要在类层次中实现共享，那么最好使用虚基类的方法，否则你就得构建自己的变量来实现它。

我们可以把一个有虚基类的类的对象表示成下面的形式：



指向表示虚基类 Storable 的共享对象的"指针（箭头）"实际上是偏移量，通过把 Storable 放置在与 Receiver 或者 Transmitter 子对象相对固定的位置上可以实现性能的优化。

21.3.5.1 构造虚基类

我们可以使用虚基类创建复杂的逻辑框架。程序框架当然越简单越好，但是即使它比较复杂，语言也能确保虚基类的构造函数只调用一次。而且，基类（无论是否为虚基类）的构造函数一定是在派生类的构造函数之前调用的，否则就会造成混乱（也就是说，对象还没初始化就被使用了）。为了避免发生这样的混乱，每个虚基类的构造函数都由完整对象的构造函数（最终派生类的构造函数）负责（显式地或者隐式地）调用。这就确保即使在类层次中虚基类被多次提及，它也只会被构造一次。例如：

```
struct V {
    V(int i);
    // ...
};

struct A {
    A();      // 默认构造函数
```

```
    // ...
};

struct B : virtual V, virtual A {
    B() :V{1} { /* ... */ };  // 默认构造函数，必须初始化基类 V
    // ...
};

class C : virtual V {
public:
    C(int i) : V{i} { /* ... */ };    // 必须初始化基类 V
    // ...
};

class D : virtual public B, virtual public C {
    // 从 B 和 C 隐式地获取虚基类 V
    // 从 B 隐式地获取虚基类 A
public:
    D() { /* ... */ }                    // 错误：C 和 V 没有默认构造函数
    D(int i) :C{i} { /* ... */ };        // 错误：V 没有默认构造函数
    D(int i, int j) :V{i}, C{j} { /* ... */ }    // OK
    // ...
};
```

请注意，D 可以而且必须给 V 提供一个初始化器，即使 V 没有被显式地声明成 D 的基类也是如此。关于虚基类的信息以及必须初始化它的义务都被“冒泡”到最终的派生类。虚基类永远被认为是其最终派生类的直接基类。虽然 B 和 C 都初始化 V，但是不会影响最终的结果，因为编译器无法判断这两个初始化器的优劣，它只会使用最终派生类提供的初始化器。

虚基类的构造函数在其派生类的构造函数之前被调用。

在实践中，上述过程很难限定在有限的范围之内。尤其是，如果我们从 D 中派生出另一个类 DD，则 DD 将负责初始化虚基类。除非我们能够简单地继承 D 的构造函数（见 20.3.5.1 节），否则就会带来麻烦。因此，我们应该注意不要过度使用虚基类。

构造函数的这一逻辑问题对于析构函数来说并不存在。析构函数的调用顺序与构造的顺序相反（见 20.2.2 节），而且虚基类的析构函数只执行一次。

21.3.5.2　一次只调用一个虚基类成员

当为具有虚基类的类定义函数时，一般情况下程序员并不知道该基类是否会与其他派生类共享。如果严格要求一次派生类函数调用对应一次基类函数调用，则情况会比较麻烦。必要的话程序员可以模拟构造函数的使用模式以确保对虚基类的调用由其最终派生类完成。例如，假定基类 Window 知道该如何绘制它的内容：

```
class Window {
public:
    // 基本要素
    virtual void draw();
};
```

此外，有很多装饰窗口及添加功能的手段：

```
class Window_with_border : public virtual Window {
    // 边界元素
protected:
    void own_draw();    // 显示边界
public:
```

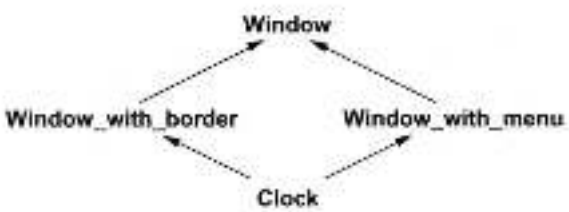
```
    void draw() override;
};

class Window_with_menu : public virtual Window {
    // 菜单元素
protected:
    void own_draw();    // 显示菜单
public:
    void draw() override;
};
```

own_draw() 函数不需要定义成 virtual 的，它一般由虚函数 draw() 负责调用，而后者“知道”什么类型的对象调用了它。

基于上述内容，我们可以进一步创建一个 Clock 类：

```
class Clock : public Window_with_border, public Window_with_menu {
    // 时钟元素
protected:
    void own_draw();    // 显示表盘和指针
public:
    void draw() override;
};
```

或者表示为下面的图形：



此时，我们就可以用 own_draw() 函数定义 draw() 了，并且 draw() 的调用者只执行一次 Window::draw()。这与调用 draw() 的 Window 种类无关：

```
void Window_with_border::draw()
{
    Window::draw();
    own_draw();    // 显示边界
}

void Window_with_menu::draw()
{
    Window::draw();
    own_draw();    // 显示菜单
}

void Clock::draw()
{
    Window::draw();
    Window_with_border::own_draw();
    Window_with_menu::own_draw();
    own_draw();    // 显示表盘和指针
}
```

请注意，一个带限定符的调用（比如 Window::draw()）不会用到虚调用机制。相反，它直接调用显式的具名函数，从而避免了无限递归的情况。

虚基类向派生类的强制类型转换将在 22.2 节讨论。

21.3.6　重复基类与虚基类

使用多重继承实现表示纯接口的抽象类会影响程序设计的方式。21.2.3 节的 BB_ival_slider 类是这样的一个例子：

```
class BB_ival_slider
    : public Ival_slider,       // 接口
      protected BBslider        // 实现
{
    // 使用 BBslider 的功能实现 Ival_slider 和 BBslider 所需的函数
};
```

在本例中，两个基类扮演逻辑上完全相反的两种角色。一个是提供接口的公共抽象类，另一个则是提供实现“细节”的受保护的具体类。它们的角色既受类的形式影响，也受其访问控制权限的影响（见 20.5 节）。在这里多重继承几乎是必不可少的，因为派生类需要同时覆盖接口和实现中的虚函数。

举个例子，让我们重新考虑 21.2.1 节的 Ival_box 类。在最后（见 21.2.2 节），我把所有 Ival_box 类都变成了抽象类以反映其纯接口的角色。这样做可以确保把所有实现细节都放在特定的实现类中。并且，所有共享实现细节的工作都在用于实现的窗口系统的传统层次体系中完成了。

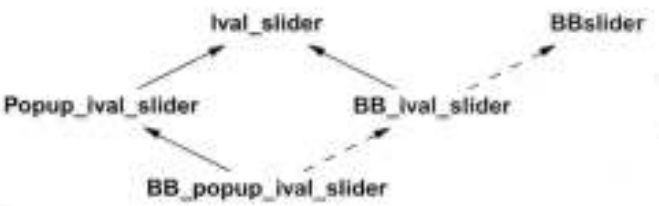
当用抽象类（不含任何共享数据）作为接口时，我们可以选择：

- 重复接口类（在类层次体系中每提到一次创建一个对象）。
- 将接口类设为 virtual，令层次体系中所有提及它的类共享一个简单的对象。

使用 Ival_slider 作为虚基类，我们可以编写下面的代码：

```
class BB_ival_slider
    : public virtual Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider
    : public virtual Ival_slider { /* ... */ };
class BB_popup_ival_slider
    : public virtual Popup_ival_slider, protected BB_ival_slider { /* ... */ };
```

或者表示为下面的图形：

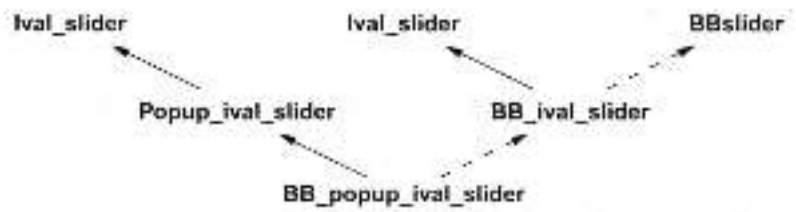


基于上述内容，很容易从 Popup_ival_slider 派生出其他接口，或者从 BB_popup_ival_slider 派生出其他实现类。

我们还可以提供另一种替代方法，它使用重复的 Ival_slider 对象：

```
class BB_ival_slider
    : public Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider
    : public Ival_slider { /* ... */ };
class BB_popup_ival_slider
    : public Popup_ival_slider, protected BB_ival_slider { /* ... */ };
```

或者表示为下面的图形：



出人意料的是，上述两种设计方式尽管存在逻辑差异，但是它们在运行时间及空间上难分伯仲。在重复 Ival_slider 的方式中，BB_popup_ival_slider 不能隐式地转换成 Ival_slider（否则将产生二义性）：

```
void f(Ival_slider* p);

void g(BB_popup_ival_slider* p)
{
    f(p); // 错误：Popup_ival_slider::Ival_slider 还是 BB_ival_slider::Ival_slider?
}
```

另一方面，使用虚基类的方式在某些情境下也会造成从基类向派生类的强制类型转换（见 22.2 节）产生二义性。不过，这种二义性很容易处理。

我们应该选择虚基类的方式还是重复基类的方式表示我们的接口呢？当然在大多数情况下，我们只能适应现有的设计方式而无从选择。但当我们有权选择时，可以考虑如下事实：重复基类的解决方案产生的对象稍小（因为不需要任何数据结构支持共享），并且我们习惯于从“虚构造函数”或者“工厂函数”（见 21.2.4 节）中获取接口对象。例如：

```
Popup_ival_slider* popup_slider_factory(args)
{
    // ...
    return new BB_popup_ival_slider(args);
    // ...
}
```

从实现（此处是 BB_popup_Ival_slider）到其直接接口（此处是 Popup_ival_slider）无须显式的类型转换。

21.3.6.1 覆盖虚基类函数

派生类可以覆盖其直接或者间接虚基类的虚函数。尤其是，两个不同的类可能会覆盖虚基类的不同的虚函数。通过这种方式，几个派生类就能共同为一个虚基类表示的接口提供实现了。例如，Window 类含有函数 set_color() 和 prompt()。此时，Window_with_border 可能覆盖 set_color() 以控制颜色模式，Window_with_menu 可能会覆盖 prompt() 以控制用户交互：

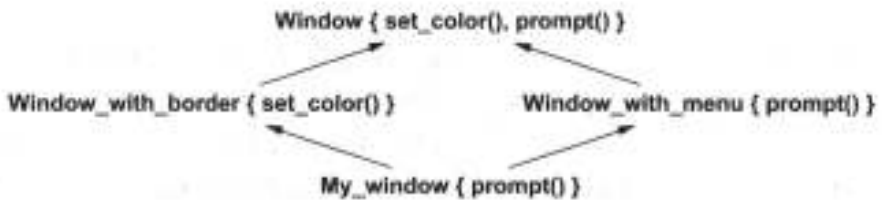
```
class Window {
    // ...
    virtual void set_color(Color) = 0;          // 设置背景颜色
    virtual void prompt() = 0;
};

class Window_with_border : public virtual Window {
    // ...
    void set_color(Color) override;             // 控制背景颜色
};
```

```
class Window_with_menu : public virtual Window {
    // ...
    void prompt() override;              // 控制用户交互
};

class My_window : public Window_with_menu, public Window_with_border {
    // ...
};
```

如果不同的派生类覆盖了同一个函数会怎样呢？当且仅当其中一个覆盖的类派生自其他所有覆盖了该函数的类时，才允许这种情况发生。换句话说，一个函数必须覆盖所有其他版本。例如，My_window 可以覆盖 Window_with_menu 提供的 prompt() 以进行改进：

```
class My_window : public Window_with_menu, public Window_with_border {
    // ...
    void prompt() override;  // 别让基类负责用户交互
};
```

或者表示为下面的图形：



如果两个类都覆盖了同一个基类函数，但是它们彼此之间没有覆盖，则该层次关系存在错误。原因是任何一个函数都无法在为所有调用提供一致语义的同时又不受选用哪个类作为接口的影响。或者用实现术语来说，因为在完整对象上调用该函数具有二义性，所以我们无法构建一张虚函数表。例如，假设 21.3.5 节的 Radio 没有声明 write()，则当我们定义 Radio 时，Receiver 和 Transmitter 声明的 write() 就会产生错误。要想解决此类问题，必须像 Radio 一样给最终派生类增加一个覆盖函数。

为虚基类提供一部分实现（但非全部实现）的类称为混入类（mixin）。

21.4　建议

[1] 为了避免忘记 delete 用 new 创建的对象，建议使用 unique_ptr 或者 shared_ptr；21.2.1 节。

[2] 不要在作为接口的基类中放置数据成员；21.2.1.1 节。

[3] 用抽象类表示接口；21.2.2 节。

[4] 为抽象基类定义一个虚析构函数确保其正确地清理资源；21.2.2 节。

[5] 在规模较大的类层次中用 override 显式地覆盖；21.2.2 节。

[6] 用抽象类支持接口继承；21.2.2 节。

[7] 用含有数据成员的基类支持实现继承；21.2.2 节。

[8] 用普通的多重继承表示特征的组合；21.3 节。

[9] 用多重继承把实现和接口分离开来；21.3 节。

[10] 用虚基类表示层次中一部分（而非全部）类公有的内容；21.3.5 节。

运行时类型信息

不成熟的优化是一切罪恶之源。

——唐纳德·克努斯

另一方面，我们不能忽略效率。

——乔恩·本特利

- 引言
- 类层次导航
  dynamic cast；多重继承；static_cast 和 dynamic_cast；恢复接口
- 双重分发和访客
  双重分发；访客
- 构造和析构
- 类型识别
  扩展类型信息
- RTTI 的使用和误用
- 建议

22.1 引言

一般来说，类是从某类的框架中构造出来的。这种类框架（class lattice）通常被称为类层次（class hierarchy）。我们在设计类时，会努力令使用者不必过分操心一个类是如何由其他类组合出来的。特别是，虚调用机制保证了：当我们对一个对象调用函数 f() 时，对类层次中任何提供了可调用的 f() 声明的类，以及定义了 f() 的类，都会调用此函数。本章将介绍如何在仅有基类提供的接口的情况下获得全部对象信息。

22.2 类层次导航

对 21.2 节中定义的 Ival_box，一个看起来很合理的应用是将其对象传递给一个控制屏幕的系统，当有动作发生时，系统再将对象传回应用程序。这里，系统（system）是指 GUI 库和控制屏幕的操作系统设施的组合。在系统和应用程序间来回传递的对象通常称为小部件（widget）或控件（control）。这就是用户界面的工作方式。从编程语言的角度，很重要的一点是系统不了解我们的 Ival_box 的细节。系统的接口是基于系统自己的类和对象定义的，而不是基于我们的应用程序的类。这是必要的也是恰当的。但是，这种机制也有不那么令人满意的一面：对于我们传递给系统、随后又传回给我们的对象，其类型信息丢失了。

为了恢复“丢失的”对象类型信息，我们需要用某种方法要求对象透露其类型。面对一个对象进行任何操作，都需要使用一个符合该对象类型的指针或引用。因此，在运行时检测对象类型的最明显也最有用的操作就是类型转换。若对象确为预期类型，该操作应返回一个

合法的指针，否则返回一个空指针。dynamic_cast 恰好完成这样的功能。例如，假定“系统”传递给 my_event_handler() 一个指向 BBwindow 的指针，指出用户动作发生的位置（窗口），接下来我就可以用 Ival_box 的 do_something() 调用我的应用代码：

```
void my_event_handler(BBwindow* pw)
{
    if (auto pb = dynamic_cast<Ival_box*>(pw)) { // pw 指向一个 Ival_box 吗?
        // ...
        int x = pb->get_value();  // 使用 Ival_box
        // ...
    }
    else {
        // ... 糟糕！处理意外情况…
    }
}
```

这段程序中发生了什么？可以这么解释：dynamic_cast 将用户界面系统的面向实现的语言转换为应用程序的语言。需要注意的很重要的一点是本例中没有提及对象的真实类型是什么。对象可能是一种特殊的 Ival_box，比如说 Ival_slider，通过一种特殊的 BBwindow，比如说 BBslider 来实现。在这种“系统”和应用程序的交互中，显式使用对象的真实类型既无必要也不恰当。我们设计接口的目的是表示交互的本质特征。尤其是，一个精心设计的接口应该隐藏所有无关紧要的细节。

pb = dynamic_cast<Ival_box*>(pw) 的类型转换过程可以图示如下：



从 pw 和 pb 发出的箭头表示指向传递的对象的指针，而其他箭头表示对象经过的不同部分间的继承关系。

在运行时使用类型信息通常被称为“运行时类型信息”，简写为 RTTI（Run-Time Type Information）。

从基类到派生类的转换通常称为向下转换（downcast），因为我们画继承树的习惯是从根（基类）向下画。类似地，从派生类到基类的转换称为向上转换（upcast）。而从基类到兄弟类的转换，例如从 BBwindow 转换为 Ival_box，则称为交叉转换（crosscast）。

22.2.1　dynamic_cast

运算符 dynamic_cast 接受两个运算对象：被 < 和 > 包围的一个类型和被（和）包围的一个指针或引用。我们首先看一个指针转换的例子：

```
dynamic_cast<T*>(p)
```

如果 p 是 T* 类型，或者是 D* 类型且 T 是 D 的基类，则得到的结果就如同我们简单地将 p 赋予一个 T* 一样。例如：

```
class BB_ival_slider : public Ival_slider, protected BBslider {
    // ...
};
```

```
void f(BB_ival_slider* p)
{
    Ival_slider* pi1 = p;                                   // 正确
    Ival_slider* pi2 = dynamic_cast<Ival_slider*>(p);       // 正确

    BBslider* pbb1 = p;                                     // 错误：BBslider 是一个保护基类
    BBslider* pbb2 = dynamic_cast<BBslider*>(p);            // 正确：pbb2 成为空指针
}
```

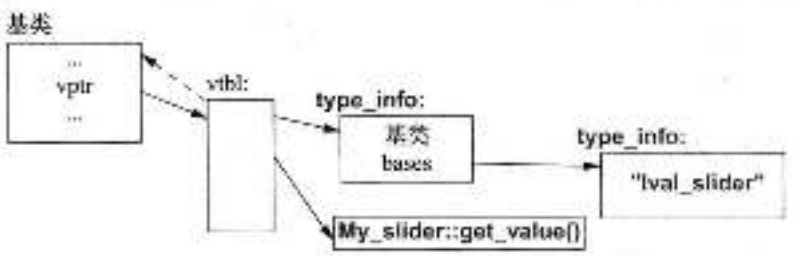
这种向上转换的例子没什么意思。但是，dynamic_cast 不会允许意外地破坏对私有和保护基类的保护，了解到这一点还是会令人安心的。dynamic_cast 在用于向上转换时与简单复赋值到无二致，这意味着使用 dynamic_cast 没有额外开销且对上下文是敏感的。

dynamic_cast 的真正用武之地是编译器无法确定类型转换正确性的情形。在此情况下，dynamic_cast<T*>(p) 查看 p 指向的对象（如果有的话）。如果对象的类型是类 T 或其类型有唯一的基类 T，则 dynamic_cast 返回一个指向该对象的 T* 类型的指针；否则返回 nullptr。如果 p 的值是 nullptr，dynamic_cast<T*>(p) 也会返回 nullptr。注意，类型转换要求必须对可唯一识别的对象进行。我们可以构造出类型转换的例子，其中 p 指向的对象包含不止一个表示基类型 T* 的子对象，由于不满足唯一识别的要求，转换失败并返回 nullptr（见 22.2 节）。

dynamic_cast 要求给定的指针或引用指向一个多态类型，以便进行向下或向上转换。例如：

```
class My_slider: public Ival_slider {       // 多态基类（Ival_slider 有虚函数）
    // ...
};

class My_date : public Date {               // 基类非多态（Date 没有虚函数）
    // ...
};

void g(Ival_box* pb, Date* pd)
{
    My_slider* pd1 = dynamic_cast<My_slider*>(pb);      // 正确（一个 Ival_slider 是一个 Ival_box）
    My_date* pd2 = dynamic_cast<My_date*>(pd);          // 错误：Date 非多态
}
```

对指针类型多态性的要求简化了 dynamic_cast 的实现，因为这样就很容易找到一个保存对象类型必要信息的地方。一个典型的实现会将一个“类型信息对象”（见 22.5 节）附加到对象上，具体方法是将指向类型信息的指针放在对象类的虚函数表中（见 3.2.3 节）。例如：



虚线箭头表示一个偏移量，借助它，仅给定一个指向多态子对象的指针就可以找到整个对象的起始地址。很明显，dynamic_cast 的实现可以非常高效：所要做的只是对表示基类的 type_info 对象进行几次比较操作，无须任何代价昂贵的查询或字符串比较操作。

限制 dynamic_cast 只能转换多态类型在逻辑上也是有道理的——如果一个对象没有虚函数，那么在不了解其确切类型的情况下，是无法安全操作它的。因此，我们在编程时必须十分小心，避免不能获知这种非多态对象的类型却又需要使用它的情形。而另一方面，如果类型已知，我们就不必使用 dynamic_cast 了。

dynamic_cast 的目标类型不必是多态的。这令我们可以在一个多态类型中包含一个具体类型，例如，可以通过一个对象 I/O 系统（见 22.2.4 节）传输具体类型数据，随后将具体类型数据“解包”。如下例：

```
class Io_obj {                  // 对象 I/O 系统的基类
    virtual Io_obj* clone() = 0;
};
class Io_date : public Date, public Io_obj { };

void f(Io_obj* pio)
{
    Date* pd = dynamic_cast<Date*>(pio);
    // ...
}
```

向 void * 进行 dynamic_cast 可以用来确定一个多态类型对象的起始地址。例如：

```
void g(Ival_box* pb, Date* pd)
{
    void* pb2 = dynamic_cast<void*>(pb);  // 正确
    void* pd2 = dynamic_cast<void*>(pd);  // 错误：不是多态类型
}
```

在一个派生类对象中，对应基类（如 Ival_box）的对象并不一定是最底层派生类对象中的第一个子对象。因此，pb 不一定具有和 pb2 一样的地址，使用 dynamic_cast 可以保证获得正确的地址。

这种类型转换只有在与非常底层的函数（如处理 void* 的函数）打交道时才是有用的。用 dynamic_cast 将 void* 转换成其他类型也是不允许的（因为可能不知道去哪里找 vptr，见 22.2.3 节）。

22.2.1.1 用 dynamic_cast 转换引用类型

为了实现多态行为，我们必须通过指针或引用来处理对象。当用 dynamic_cast 转换指针类型时，nullptr 表示转换错误。这对引用类型来说是不可行的，也是不合理的。

当我们获得一个指针时，就必须考虑它是否为 nullptr，即，未指向任何对象。因此，对 dynamic_cast 转换指针得到的结果必须要进行显式检测。对一个指针 p，dynamic_cast<T*>(p) 可以看作一个问题：“p 指向的对象（如果存在的话）类型为 T 吗？”例如：

```
void fp(Ival_box* p)
{
    if (Ival_slider* is = dynamic_cast<Ival_slider*>(p)) {      // p 指向一个 Ival_slider？
        // ... 是的，接着使用转换得到的 is ...
    }
    else {
        // ...*p 不是一个 slider，进行其他处理 ...
    }
}
```

而另一方面，我们可以合理地假定一个引用肯定指向一个对象（见 7.7.4 节）。因此，对一个引用 r,dynamic_cast<T&>(r) 并不是一个问题，而是一个断言：“r 引用的对象的类型为 T。”

对 r 进行 dynamic_cast 得到的结果实际上已经隐含地被 dynamic_cast 实现自身检查过了。如果 dynamic_cast 的引用对象不是所期望的类型，它会抛出一个 bad_cast 异常。例如：

```
void fr(Ival_box& r)
{
    Ival_slider& is = dynamic_cast<Ival_slider&>(r);        // r 应引用一个 Ival_slider !
    // ... 使用 is ...
}
```

一次失败的动态指针转换和一次失败的动态引用转换在结果上的这种差异反映了引用和指针的根本不同。如果用户希望避免错误的引用转换，就必须提供适合的处理程序。例如：

```
void g(BB_ival_slider& slider, BB_ival_dial& dial)
{
    try {
        fp(&slider);    // 指向 BB_ival_slider 的指针作为 Ival_box* 传递
        fr(slider);     // 指向 BB_ival_slider 的引用作为 Ival_box& 传递
        fp(&dial);      // 指向 BB_ival_dial 的指针作为 Ival_box* 传递
        fr(dial);       // dial 作为 Ival_box 传递
    }
    catch (bad_cast) {  // 见 30.4.1.1
        // ...
    }
}
```

对 fp() 的两次调用和第一次 fr() 调用将会正常返回（假定 fp() 的确能处理 BB_ival_dial 的对象），但第 2 次 fr() 调用会导致一个 bad_cast 异常，此异常会被 g() 捕获。

显式检查 nullptr 的代码很容易不小心漏掉。如果这令你烦恼，你可以编写一个转换函数，在转换失败时抛出一个异常而不是返回 nullptr。

22.2.2　多重继承

当只使用单一继承时，一个类及其基类构成一棵树，这棵树以一个单一基类为根。这种类层次很简单，但通常也有很大局限。当使用多重继承时，不存在单一的根结点。本质上，情况并没有变得复杂很多。但是，如果一个类在类层次中出现多次，我们在引用这个类的对象时就必须要小心一点儿。

只要条件允许，我们自然会努力保持类层次的简单性（但不会过分简化）。但是，一旦建立了一个重要的类层次，我们有时就需要在其中导航来寻找要使用的特定的类，这种需求可能以两种形式出现：

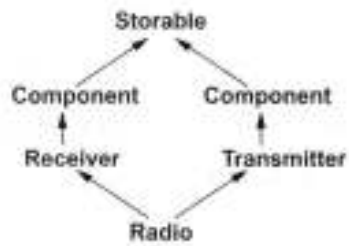
- 有时，我们希望显式地命名一个基类，用作接口。例如，为了解决歧义或是为了在不依赖虚函数机制的情况下调用一个特定函数（显式限定调用请见 21.3.3 节）。
- 有时，我们希望从一个给定的子对象的指针获得类层次另一个子对象的指针。例如，从一个基类指针获得完整派生类对象的指针（向下转换，见 22.2.1 节）或是从一个基类的指针获得另一个基类对象的指针（交叉转换，见 22.2.4 节）。

在本节中，我们考虑一种类层次导航的方法——使用类型转换获得一个所需类型的指针。为了说明转换机制及其背后的规则，我们来考虑一个同时包含重复基类和虚基类的框架：

```
class Component
    : public virtual Storable { /* ... */ };
class Receiver
    : public Component { /* ... */ };
```

```
class Transmitter
    : public Component { /* ... */ };
class Radio
    : public Receiver, public Transmitter { /* ... */ };
```

其结构可图示如下：



在本例中，一个 Radio 对象有两个 Component 类的子对象。因此，在一个 Radio 对象上进行从 Storage 指针到 Component 指针的 dynamic_cast 是有二义性的，会返回 0，因为我们根本无法知道程序员想要哪个 Component：

```
void h1(Radio& r)
{
    Storable* ps = &r;  // 一个 Radio 包含唯一的 Storable
    // ...
    Component* pc = dynamic_cast<Component*>(ps); // pc = 0，因为一个 Radio 包含两个 Component
    // ...
}
```

一般而言（也是典型情况），程序员（或是正在翻译单个程序单元的编译器）并不了解完整的类框架，而只是基于对一些子框架的了解来编写代码。例如，一个程序员可能只了解 Radio 的 Transmitter 部分而编写如下代码：

```
void h2(Storable* ps)          // ps 可能是一个 Component 指针，也可能不是
{
    if (Component* pc = dynamic_cast<Component*>(ps)) {
        // 我们得到了一个 Component！
    }
    else {
        // ps 不是一个 Component
    }
}
```

在本例中，指向 Radio 对象的指针引起的二义性通常在编译时是无法检查出来的。

只有虚基类才需要这种运行时的二义性检查。对于普通基类，当进行向下转换时（即，转换为派生类，见 22.2 节），其结果只可能是唯一的子对象（或是转换失败）。对虚基类来说，进行向上转换（即，转换为基类）也可能发生类似的二义性问题，但这种二义性可以在编译时被捕获。

22.2.3　static_cast 和 dynamic_cast

dynamic_cast 可以从一个多态虚基类转换到一个派生类或是一个兄弟类（见 22.2.1 节）。static_cast 则不行，因为它不检查要转换的对象：

```
void g(Radio& r)
{
    Receiver* prec = &r;                    // Receiver 是 Radio 的普通基类
```

```
    Radio* pr = static_cast<Radio*>(prec);  // 正确，无须检查
    pr = dynamic_cast<Radio*>(prec);        // 正确，运行时检查

    Storable* ps = &r;                      // Storable 是 Radio 的虚基类
    pr = static_cast<Radio*>(ps);           // 错误：不能从虚基类转换
    pr = dynamic_cast<Radio*>(ps);          // 正确，运行时检查
}
```

dynamic_cast 要求运算对象是多态的，因为它需要特定的信息来找到表示基类的子对象，而一个非多态对象中不包含任何这样的信息。特别是，我们可能用其他语言，如 Fortran 或 C 与 C++ 混合编程，当一个类型的对象是由这些语言所确定，而此类型又被用作虚基类时，这种类型的对象就只包含静态类型信息。而运行时类型识别所需的动态信息就包括实现 dynamic_cast 所需的信息。

为什么还有人要用 static_cast 在类层次中进行导航呢？原因在于使用 dynamic_cast 是有运行时额外开销的（见 22.2.1 节）。更重要的是，有数百万行的代码是在 dynamic_cast 产生之前编写的。这些代码依赖于其他方法确保类型转换的有效性，对它们来说基于 dynamic_cast 的检查是多余的。但是，这类代码通常是用 C 风格的类型转换（见 11.5.3 节）编写的，遗留了一些隐藏很深的错误。因此，只要条件允许，尽量使用更安全的 dynamic_cast。

对于一个 void* 所指向的内存，编译器不能做任何假设。这意味着 dynamic_cast 不能将一个 void* 转换为其他类型，因为 dynamic_cast 必须探查一个对象的内部来确定其类型。这时需要使用 static_cast。例如：

```
Radio* f1(void* p)
{
    Storable* ps = static_cast<Storable*>(p);     // 你任我降低
    return dynamic_cast<Radio*>(ps);
}
```

dynamic_cast 和 static_cast 都遵守 const 规则和访问控制规则。例如：

```
class Users : private set<Person> { /* ... */ };

void f2(Users* pu, const Receiver* pcr)
{
    static_cast<set<Person>*>(pu);          // 错误：非法访问
    dynamic_cast<set<Person>*>(pu);         // 错误：非法访问
    static_cast<Receiver*>(pcr);            // 错误：不能强制去除 const
    dynamic_cast<Receiver*>(pcr);           // 错误：不能强制去除 const

    Receiver* pr = const_cast<Receiver*>(pcr);  // 正确
    // ...
}
```

我们不可能用 dynamic_cast 和 static_cast 转换到私有基类，而“强制去除 const”（或 volatile）则需要使用 const_cast（见 11.5.2 节）。但即使是这样，只有当对象最初不是声明为 const（或 volatile）时，转换结果才能安全使用。

22.2.4　恢复接口

从设计的角度来看，dynamic_cast（见 22.2.1 节）可以被看作一种询问对象是否提供了指定接口的机制。

例如，考虑一个简单的对象 I/O 系统。用户想从一个流读取对象，确定对象是否是期望

的类型，如果是，就使用它们：

```
void user()
{
    //... 打开文件，将 istream ss 与之关联，假定文件包含 Shape 对象 ...

    unique_ptr<Io_obj> p {get_obj(ss)}; // 从流读取对象

    if (auto sp = dynamic_cast<Shape*>(p.get())) {
        sp->draw();    // 使用 Shape
        // ...
    }
    else {
        // 错误：处理文件中非 Shape 对象
    }
}
```

函数 user() 只通过抽象类 Shape 来处理形状，因此能使用各种不同的形状。这里使用 dynamic_cast 是必要的，因为对象 I/O 系统能处理很多其他类型的对象，而用户可能意外打开一个包含完好类对象的文件，但对象的类型用户根本都没听说过。

在本例中我使用了 unique_ptr<Io_obj>（见 5.2.1 节和 34.3.1 节），这样就不会忘记释放 get_obj() 分配的对象了。

这个对象 I/O 系统假定读写的每个对象的类型都是派生自 Io_obj 的类。类 Io_obj 必须是一个多态类型，以便允许 get_obj() 的用户用 dynamic_cast 来恢复返回对象的“真正类型”。例如：

```
class Io_obj {
public:
    virtual Io_obj* clone() const =0;    // 多态
    virtual ~Io_obj() {}
};
```

对象 I/O 系统的关键函数是 get_obj()，它从 istream 读取数据创建类对象。假定在输入流中表示对象的数据都以标识对象类型的字符串为前缀，则 get_obj() 的工作就是读取这个字符串，然后调用能读取数据并创建正确类对象的函数。例如：

```
using Pf = Io_obj*(istream&);          // 函数指针，指向的函数返回一个 Io_obj*

map<string,Pf> io_map;                 // 将字符串映射到相应的创建函数

string get_word(istream& is);          // 从 is 读取一个字，若读取失败抛出一个 Read_error

Io_obj* get_obj(istream& is)
{
    string str = get_word(is);         // 读取起始字
    if (auto f = io_map[str])          // 查找 str，获取对应函数
        return f(is);                  // 调用函数
    throw Unknown_class{};             // map 中无匹配 str 的项
}
```

名为 io_map 的 map 保存类的名字字符串对到能创建该类对象的函数的映射。

根据 user() 的需要，我们可以从 Io_obj 派生出 Shape 类：

```
class Shape : public Io_obj {
    // ...
};
```

但是，原封不动地使用已经定义好的 Shape（见 3.2.4 节）会更有趣（而且在很多情况下也更实际）：

```
struct Io_circle : Circle, Io_obj {
    Io_circle(istream&);                                    // 从输入流初始化
    Io_circle* clone() const { return new Io_circle(*this); }   // 使用复制构造函数
    static Io_obj* new_circle(istream& is) { return new Io_circle{is}; } // 供 Io_map 使用
};
```

这个例子展示了，只需很少一点儿预见性来指导设计，我们即可通过一个抽象类将一个类纳入到一个已有的类层次中，而不必起初就将该类定义为类层次中的一个结点。

这段代码中，Io_circle(istream&) 构造函数从 istream 参数读取数据初始化对象。new_circle() 函数则存放在 Io_map 中，以便对象 I/O 系统能处理此类。例如：

```
Io_map["Io_circle"]=&Io_circle::new_circle;   // 放在程序某处
```

其他形状类可以类似构造：

```
class Io_triangle : public Triangle, public Io_obj {
    // ...
};

Io_map["Io_triangle"]=&Io_circle::new_triangle;   // 放在程序某处
```

如果你觉得这样构建对象 I/O 系统的框架有些冗长乏味，使用模板可能会好些：

```
template<class T>
struct Io : T, Io_obj {
public:
    Io(istream&);                                   // 从输入流初始化
    Io* clone() const override { return new Io(*this); }
    static Io* new_io(istream& is) { return new Io{is}; }   // 供 Io_map 使用
};
```

有了这个模板，我们就可以如下定义 Io_circle：

```
using Io_circle = Io<Circle>;
```

我们仍然需要显式定义 Io<Circle>::Io(istream&)，因为它需要了解 Circle 的细节。注意，Io<T>::Io(istream&) 并不需要访问 T 的私有或保护数据。其中的诀窍在于，一个类型 X 的传输格式其实就是使用 X 的构造函数创建一个 X 对象时所要用的数据格式。流中的数据不必是 X 的成员值的序列。

Io 模板展示了如何借助类层次中的一个结点来将一个具体类型纳入到类层次中。Io 派生自它的模板参数和 Io_obj，这样就允许从 Io_obj 进行类型转换。例如：

```
void f(Io<Shape>& ios)
{
    Shape* ps = &ios;
    // ...
}
```

不幸的是，由于派生自模板参数，Io 不能用于内置类型：

```
using Io_date = Io<Date>;     // 创建了一个具体类型
using Io_int = Io<int>;       // 错误：不能派生自内置类型
```

这个问题的一个解决方法是，将用户对象声明为 Io_obj 的一个成员：

```
template<class T>
```

```
struct Io :Io_obj {
    T val;

    Io(istream&);                                          // 从输入流初始化
    Io* clone() const override { return new Io{*this}; }
    static Io* new_io(istream& is) { return new Io{is}; }  // 供 io_map 使用
};
```

现在我们就可以将 Io 用于内置类型了：

```
using Io_int = Io<int>;          // 封装了一个内置类型
```

将值定义为一个成员而非一个基类，我们就不能直接将一个 Io_obj<X> 转换为一个 X 了，而需要提供一个函数完成这种转换：

```
template<typename T>
T* get_val<T>(Io_obj* p)
{
    if (auto pp = dynamic_cast<Io<T>*>(p))
        return &pp->val;
    return nullptr;
}
```

现在 user() 函数变为：

```
void user()
{
    // ... 打开文件，将 istream ss 与之关联，假定文件包含 Shape 对象 ...

    unique_ptr<Io_obj> p {get_obj(ss)}; // 从流读取对象

    if (auto sp = get_val<Shape>(p.get())) {
        sp->draw();    // 使用 Shape
        // ...
    }
    else {
        // ... 糟糕：处理文件中非 Shape 对象 ...
    }
}
```

这个简单的 I/O 系统不能做任何有实际用处的事情，但它通过一页以内的篇幅展示了一些很有用的关键机制。对于一个希望以类型安全的方式在通信信道中传输任意对象的系统，本例给出了“接收端”的一个蓝本。更一般的情况下，本例中所使用的技术还可以用于：基于用户提供的字符串来调用函数；通过运行时类型识别机制发现的接口来处理未知类型的对象。

这种对象 I/O 系统的发送端通常也会使用 RTTI。考虑下面这个类：

```
class Face : public Shape {
public:
    Shape* outline;
    array<Shape*> eyes;
    Shape* mouth;

    // ...
};
```

为了正确写出 outline 指向的 Shape 对象，我们需要解析出它到底是哪种 Shape。这是 typeid() 的工作（见 22.5 节）。一般来说，我们还要维护一个表，保存（指针，唯一标识）对，以便传输链接的数据结构以及避免重复传输被一个以上指针（或引用）所指向的对象。

22.3　双重分发和访客

经典的面向对象程序设计基于这样一种模式：仅给出一个指向接口（基类）的指针或引用，基于动态类型（最底层派生类的类型）来选择一个恰当的虚函数。特别是，C++ 每次可以对一个类型进行这种运行时查找（也称为动态分发，dynamic dispatch）。在这点上，C++ 与 Simula 和 Smalltalk 以及更新的语言如 Java 和 C# 很相似。一个很大的局限是不能根据两个动态类型来选择函数。而且，虚函数必须是成员函数。这意味着，如果不修改提供接口的基类和所有应涉及的派生类，我们是不可能将一个虚函数添加到一个类层次中的。这也会成为一个严重的问题。本节介绍解决这些问题的基本方法：

22.3.1 节展示如何基于两个类型选择一个虚函数。

22.3.2 节展示如何用双重分发只借助类层次中的单一虚函数就能向类层次中添加多个函数。

这些技术的大多数实际应用都是和数据结构处理相关的，如向量、图、指向多态类型对象的指针等数据结构的处理。在处理这些数据结构时，一个对象（如，一个向量元素或一个图结点）的真实类型只能通过（隐式或显式地）检查基类提供的接口来动态获知。

22.3.1　双重分发

考虑如何基于两个参数选择函数，例如：

```
void do_someting(Shape& s1, Shape& s2)
{
    if (s1.intersect(s2)) {
        // 两个形状重叠
    }
    // ...
}
```

我们希望这个函数对任意两个形状类（位于以 Shape 为根的类层次中）都能正确运行，例如 Circle 和 Triangle。

基本策略是调用一个虚函数为 s1 选择正确的函数，然后再进行一次调用来为 s2 选择正确的函数。简单起见，我将略去判断两个形状是否真正相交的代码，只给出选择正确函数的代码框架。首先，我们为 Shape 定义判断相交的函数：

```
class Circle;
class Triangle;

class Shape {
public:
    virtual bool intersect(const Shape&) const =0;
    virtual bool intersect(const Circle&) const =0;
    virtual bool intersect(const Triangle&) const =0;
};
```

接下来，我们需要定义 Circle 和 Triangle，覆盖这些虚函数：

```
class Circle : public Shape {
public:
    bool intersect(const Shape&) const override;
    virtual bool intersect(const Circle&) const override;
    virtual bool intersect(const Triangle&) const override
};
```

```
class Triangle : public Shape {
public:
    bool intersect(const Shape&) const override;
    virtual bool intersect(const Circle&) const override;
    virtual bool intersect(const Triangle&) const override;
};
```

现在每个类都能处理 Shape 层次中所有可能的类了，所以我们只需再决定每种组合应该如何处理即可：

```
bool Circle::intersect(const Shape& s) const { return s.intersect(*this); }
bool Circle::intersect(const Circle&) const { cout <<"intersect(circle,circle)\n"; return true; }
bool Circle::intersect(const Triangle&) const { cout <<"intersect(circle,triangle)\n"; return true; }

bool Triangle::intersect(const Shape& s) const { return s.intersect(*this); }
bool Triangle::intersect(const Circle&) const { cout <<"intersect(triangle,circle)\n"; return true; }
bool Triangle::intersect(const Triangle&) const { cout <<"intersect(triangle,triangle)\n"; return true; }
```

比较有趣的是 Circle::intersect(const Shape& s) 和 Triangle::intersect(const Shape& s) 函数。这两个函数需要处理一个 Shape& 参数，而这个参数必须指向一个派生类对象。这里所使用的技术（花招）是简单地调换两个对象的顺序，对参数 s 调用虚函数。这样，就会在其他四个函数之中选择一个，真正完成相交判定。

为了测试上述设计，我们可以创建一个 vector，保存所有可能的 Shape* 值对，然后对它们调用 intersect()：

```
void test(Triangle& t, Circle& c)
{
    vector<pair<Shape*,Shape*>> vs { {&t,&t}, {&t,&c}, {&c,&t}, {&c,&c} };
    for (auto p : vs)
        p.first->intersect(*p.second);
}
```

使用 Shape* 保证了 intersect() 的选择依赖于运行时类型解析。执行上面的测试函数，我们得到：

```
intersect(triangle,triangle)
intersect(triangle,circle)
intersect(circle,triangle)
intersect(circle,circle)
```

如果你觉得这已是一种很优雅的方法，那就该提升你的标准了。这个设计确实能正确完成任务，但随着类层次变大，所需要的虚函数的数量是呈指数增长的。在大多数情况下，这是不可接受的。而且，将其扩展到三个或更多参数虽很简单，但冗长乏味。最糟的是，当增加新的操作和新的派生类时，需要修改层次中的每个类；这种双重分发技术是高度侵入性的。我宁愿声明一个简单的非成员函数 intersect(Shape&,Shape&)，并为每种需要处理的特定形状组合重写一个专用版本。这种方法是可行的［Pirkelbauer, 2009］，但并不是 C++11 的风格。

双重分发的尴尬并未降低它想解决的问题的重要性。依赖于两个（或更多）参数的类型才能确定的操作，如 intersect(x,y)，在实际工作中并不罕见，经常会遇到。例如，确定两个矩形的相交区域很简单也很高效。因此，对很多应用，人们会发现为每个形状定义一个“边框”然后计算边框的相交区域就够了。例如：

```
class Shape {
public:
    virtual Rectangle box() const = 0;      // 包围形状的矩形
```

```
    // ...
};

class Circle : public Shape {
public:
    Rectangle box() const override;
    // ...
};

class Triangle : public Shape {
public:
    Rectangle box() const override;
    // ...
};

bool intersect(const Rectangle&, const Rectangle&);    // 计算两个矩形
bool intersect(const Shape& s1, const Shape& s2)
{
    return intersect(s1.box(),s2.box());
}
```

还有一种方法是预先计算一个查询表，保存所有类型组合对应的函数［Stroustrup, 1994］：

```
bool intersect(const Shape& s1, const Shape& s2)
{
    auto i = index(type_id(s1),type_id(s2));
    return intersect_tbl[i](s1,s2);
}
```

这种方法及其变形有着广泛的应用——很多应用使用保存在对象中的预计算值来加速类型识别（见 27.4.2 节）。

22.3.2　访客

对虚函数和覆盖版本指数增长的问题以及（过于）简单的双重分发技术令人讨厌的侵入性特性，访客模式［Gamma, 1994］提供了一种部分解决方案。

考虑如何对类层次中的每个类应用两种（或更多）操作。基本上，我们需要对一个结点层次和一个操作层次进行一次双重分发，为正确的结点选择正确的操作。操作被称为访客（visitor），在本节中，它们都定义在类 Visitor 的派生类中。结点层次中的每个结点都是一个类，都定义了一个虚函数 accept()，接受参数 Visitor&。对本例，我们使用一个 Node 层次描述编程语言语法结构，这在基于抽象语法树（abstract syntax trees，AST）的工具中是很常见的。

```
class Visitor;

class Node {
public:
    virtual void accept(Visitor&) = 0;
};

class Expr : public Node {
public:
    void accept(Visitor&) override;
};
```

```
class Stmt : public Node {
public:
    void accept(Visitor&) override;
};
```

到目前为止，一切都还好：Node 层次简单地提供一个虚函数 accept()，它接受参数 Visitor&，表示要对给定类型的 Node 执行什么操作。

这里我没使用 const，因为一个来自 Visitor 的操作通常既会更新它所“访问的”Node，也会更新 Visitor 自身。

现在 Node 的 accept() 执行双重分发并将 Node 本身传递给 Visitor 的 accept()：

```
void Expr::accept(Visitor& v) { v.accept(*this); }
void Stmt::accept(Visitor& v) { v.accept(*this); }
```

Visitor 声明了一组操作：

```
class Visitor {
public:
    virtual void accept(Expr&) = 0;
    virtual void accept(Stmt&) = 0;
};
```

我们可以在 Visitor 的派生类中覆盖函数 accept() 来定义不同的操作。例如：

```
class Do1_visitor : public Visitor {
    void accept(Expr&) { cout << "do1 to Expr\n"; }
    void accept(Stmt&) { cout << "do1 to Stmt\n"; }
};

class Do2_visitor : public Visitor {
    void accept(Expr&) { cout << "do2 to Expr\n"; }
    void accept(Stmt&) { cout << "do2 to Stmt\n"; }
};
```

我们可以创建一个指针 pair 的 vector 来测试这种方法，检查是否真正进行了运行时类型解析：

```
void test(Expr& e, Stmt& s)
{
    vector<pair<Node*,Visitor*>> vn {&e,&do1}, {&s,&do1}, {&e,&do2}, {&s,&do2}};
    for (auto p : vn)
        p.first->accept(*p.second);
}
```

程序运行结果为：

```
do1 to Expr
do1 to Stmt
do2 to Expr
do2 to Stmt
```

与简单的双重分发技术相反，访客模式在实际编程中被大量使用。原因在于它的侵入性适中（仅有 accept() 函数），基于这种基本思想的很多变形也被广泛使用。但是，类层次上的很多操作很难表达为访客。例如，需要访问图中多个不同类型结点的操作就很难简单地实现为一个访客。因此，我认为访客模式是一种不够优雅的解决方案。替代方案是存在的，如［Solodkyy, 2012］中的方法，但普通 C++ 11 中并未提供。

在 C++ 中，访客模式的大多数替代技术都基于对一个同构数据结构（例如，一个向量或一个结点的图，其中保存着指向多态类型对象的指针）的显式遍历。在每个元素或结点

上，可以调用虚函数执行所需操作，也可以基于保存的数据进行某些优化（见 27.4.2 节）。

22.4　构造和析构

一个类对象不仅是一块内存区域那么简单（见 6.4 节）。一个类对象是在“裸内存”上用其构造函数创建出来的，而当其析构函数执行完后，它又回归“裸内存”状态。构造操作是自顶向下的，而析构操作是自底向上的，因而一个类对象就是一个已经被创建或销毁了的对象。这种顺序是必需的，用以确保一个类对象不会在初始化之前被访问。试图通过“聪明地”操纵指针（见 17.2.3 节）来提前或乱序访问某类对象或成员对象是不明智的。构造和析构的顺序反映了 RTTI、异常处理（见 13.3 节）以及虚函数（见 20.3.2 节）的规则。

编程时依赖构造和析构顺序的细节是不明智的，但你可以通过在对象尚未完成的某个点调用虚函数、dynamic_cast（见 22.2 节）或 typeid（见 22.5 节）来观察这种顺序。如果是在构造函数中的某个点，对象的（动态）类型仅反映当前已经构造完成的部分。例如，如果 22.2.2 节的类层次中的 Component 的构造函数调用了一个虚函数，则它会调用 Storable 或 Component 中定义的版本，但不会调用 Receiver、Transmitter 或 Radio 中定义的版本。因为在构造过程中的这个时间点，对象还不是一个 Radio。类似地，从析构函数调用一个虚函数只会反映尚未销毁的部分。因此，最好避免在构造和析构过程中调用虚函数。

22.5　类型识别

dynamic_cast 运算符可以满足大多数运行时对象类型信息获取的需求。特别重要的是，基于它编写的代码能正确处理从程序员明确提及的类派生出的那些类。因此，类似虚函数，dynamic_cast 既保持了灵活性又具有很好的扩展性。

但是，获知一个对象的确切类型有时也是必要的。例如，我们可能想知道对象类的名字或其布局。typeid 可以满足这种需求，它生成一个对象，表示它所处理的类型。如果 typeid() 是一个函数，其声明可能像下面这样：

```
class type_info;
const type_info& typeid(expression);    //伪声明
```

即，typeid() 返回一个指向标准库类型 type_info 的引用（type_info 定义在 <type_info> 中）：

- 给定一个类型名作为运算对象，typeid(type_name) 返回一个表示 type_name 的 type_info 引用；type_name 必须是一个完整定义的类型（见 8.2.2 节）。
- 给定一个表达式 expr 作为运算对象，对于它所指向的对象，typeid(expr) 返回一个表示其类型的 type_info 引用；expr 必须指向一个完整定义的类型（见 8.2.2 节）。如果 expr 的值为 nullptr，则 typeid(expr) 会抛出一个 std::bad_typeid 异常。

typeid() 可以获得一个引用或指针指向的对象的类型：

```
void f(Shape& r, Shape* p)
{
    typeid(r);      //获得 r 引用的对象的类型
    typeid(*p);     //获得 p 指向的对象的类型
    typeid(p);      //获得指针的类型，即 Shape*（这种用法不常见，通常是用错了）
}
```

如果 typeid() 的运算对象是一个值为 nullptr 的多态类型指针或引用，它会抛出一个 std::bad_typeid 异常。如果 typeid() 的运算对象不是多态类型或者不是一个左值，则结果

在编译时即可确定，无须运行时对表达式求值。

如果运算对象是一个解引用的指针或引用，且其类型为多态类型，则返回的 type_info 对应对象的最底层派生类，即定义对象时使用的类型。例如：

```
struct Poly {    // 多态基类
    virtual void f();
    // ...
};

struct Non_poly { /* ... */ }; // 无虚函数

struct D1
    : Poly { /* ... */ };
struct D2
    : Non_poly { /* ... */ };
void f(Non_poly& npr, Poly& pr)
{
    cout << typeid(npr).name() << '\n';  // 打印类似 "Non_poly" 的内容
    cout << typeid(pr).name() << '\n';   // 打印 Poly 或 Poly 派生类的名字
}

void g()
{
    D1 d1;
    D2 d2;
    f(d2,d1);  // 打印 "Non_poly D1"
    f(*static_cast<Poly*>(nullptr),*static_cast<Null_poly*>(nullptr));  // 错误!
}
```

最后一个调用只打印 Non_poly(因为 typeid(npr) 并未被求值)，然后抛出一个 bad_typeid 异常。

type_info 的定义看起来是这样的：

```
class type_info {
    // 数据
public:
    virtual ~type_info();                                   // 多态

    bool operator==(const type_info&) const noexcept;       // 可以比较
    bool operator!=(const type_info&) const noexcept;

    bool before(const type_info&) const noexcept;           // 定义了序
    size_t hash_code() const noexcept;                      // 供 unordered_map 或类似特性所用
    const char* name() const noexcept;                      // 类型名

    type_info(const type_info&) = delete;                   // 阻止拷贝
    type_info& operator=(const type_info&) = delete;        // 阻止拷贝
};
```

函数 before() 允许 type_info 进行排序。特别是，它允许 type_id 用作有序容器（如 map）的关键字。注意，before 定义的关系与继承关系间并不存在关联。此外，函数 hash_code() 允许 type_id 用作哈希表（如 unordered_map）的关键字。

C++ 并不保证系统中每个类型只有一个 type_info 对象。实际上，如果使用了动态链接库，很难避免重复 type_info 对象。因此，我们应该使用 == 比较 type_info 对象是否相等，而不是比较 type_info 指针。

我们有时希望获知一个对象的确切类型，以便对整个对象（而不仅仅是其基类子对象）执行某些服务。理想情况，这种服务是以虚函数形式实现的，从而不必获知对象的确切类型。但在某些情况下，要处理的对象并没有公共接口，绕个弯去获取确切类型就是必需的了（见 22.5.1 节）。type_info 另外一个简单得多的应用是获取一个类的名字用于诊断输出：

```
#include<typeinfo>

void g(Component* p)
{
    cout << typeid(*p).name();
}
```

类名的具体字符表示依赖于实现。得到的 C- 风格字符串保存在系统管理的内存中，因此程序员不要试图 delete[] 它。

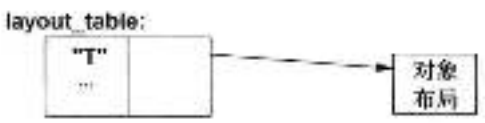
22.5.1　扩展类型信息

一个 type_info 对象只保存着最少的类型信息。因此，我们通常是把查询一个对象的确切类型作为获取和使用类型详细信息的第一步。

考虑一个实现或工具如何在运行时将类型信息提供给用户。假设我们已经有了一个工具，能生成类对象布局的描述。则我们可以将这些描述保存在一个 map 中，以便用户代码查找布局信息：

```
#include <typeinfo>

map<string, Layout> layout_table;

void f(B* p)
{
    Layout& x = layout_table[typeid(*p).name()]; // 基于 *p 的名字查找布局
    // ... 使用 x ...
}
```

得到的数据结构如下图所示：

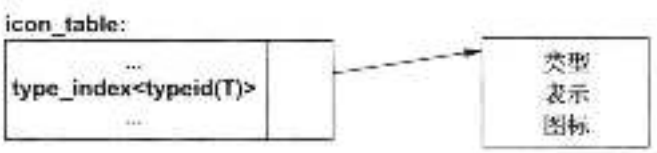


其他代码可能提供一个完全不同的布局信息：

```
unordered_map<type_index,Icon> icon_table;      // 参见 31.4.3.2 节

void g(B* p)
{
    Icon& i = icon_table[type_index(typeid(*p))];
    // ... 使用 i ...
}
```

type_index 是一个标准库类型，用于比较和哈希 type_info 对象（见 35.5.4 节）。

得到的数据结构如下所示：



将 typeid 与信息关联起来而无须修改系统头文件的特性很有用，它允许多人或多种工具将不同信息与相互无关的类型关联起来。这个特性也很重要，因为某个人提出一组信息就能满足所有用户的可能性几乎为 0。

22.6　RTTI 的使用和误用

我们应该在必要时才使用显式运行时类型信息。静态（编译时）类型检查更安全，开销更小，而且（在适用的情况下）会使程序结构更好。基于虚函数的接口结合了静态类型检查和运行时类型查询，而且结合的方式同时保证了类型安全和灵活性。但是，程序员有时会忽略这些方法而不恰当地使用 RTTI。例如，RTTI 可以用来编写稍加伪装的 switch 语句：

```
// 误用运行时类型信息的例子：

void rotate(const Shape& r)
{
    if (typeid(r) == typeid(Circle)) {
        // 什么也不做
    }
    else if (typeid(r) == typeid(Triangle)) {
        // ... 旋转三角形 ...
    }
    else if (typeid(r) == typeid(Square)) {
        // ... 旋转正方形 ...
    }
    // ...
}
```

我们可以用 dynamic_cast 代替 typeid，但也算不上什么改进。无论使用哪种方法，代码在语法上都很丑陋，而且也很低效，因为其中会反复执行一个代价很高的操作。

很不幸的是，这并不是一个虚构的例子，而是现实中真正存在的代码。很多人在编程训练时学习的语言没有类层次和虚函数或类似特性，他们很容易抑制不住冲动将软件组织为一组 switch 语句。一般情况下，我们应该抑制这种冲动，当需要运行时类型识别时，应优先选择虚函数（加 3.2.3 节和 20.3.2 节）而不是 RTTI。

很多 RTTI 的正确使用都发生在这样一个场景中：服务代码实现为一个类，而用户希望通过类派生来添加功能。22.2 节中 Ival_box 的使用就是这样一个例子。在这样的场景中，如果用户愿意而且能够修改库中类的定义，例如 BBwindow，那么就可以避免使用 RTTI；否则，就需要使用 RTTI 了。不过，即使用户愿意修改基类（例如，添加一个虚函数），这种修改也可能导致一些问题。例如，对于那些不需要这些虚函数或者这些函数无意义的类，就有必要引入哑实现。读者可以在 22.2.4 节中找到如何用 RTTI 实现一个简单的对象 I/O 系统。

一些语言严重依赖动态类型检查，如 Smalltalk、前范型 Java 或 Lisp，有这些语言背景的程序员会倾向于将 RTTI 与过分范型化的类型结合使用。例如：

```
// 误用运行时类型信息的例子：

class Object { // 多态
    // ...
};

class Container : public Object {
public:
```

```
    void put(Object*);
    Object* get();
    // ...
};

class Ship : public Object { /* ... */ };

Ship* f(Ship* ps, Container* c)
{
    c->put(ps);                                    // 将 Ship 存入容器
    // ...
    Object* p = c->get();                          // 从容器中获取数据
    if (Ship* q = dynamic_cast<Ship*>(p)) {        // 运行时检查 Object 是否是一个 Ship
        return q;
    }
    else {
        // ...做一些其他事情（或者是错误处理）...
    }
}
```

在本例中，类 Object 完全没有必要。它过于范型化了，因为它并非应用领域中的抽象，而且强迫程序员使用实现层的抽象（Object）。这种问题通常可以用容器模板更好地解决，容器只保存单一类型的指针：

```
Ship* f(Ship* ps, vector<Ship*>& c)
{
    c.push_back(ps);        // 将 Ship 存入容器
    // ...
    return c.pop_back();    // 从容器中获取 Ship
}
```

与单纯基于 Object 的代码相比，这种代码风格更不容易出错（更好的静态类型检查），也更简洁。通过与虚函数结合使用，这种技术可以处理大多数情形。在模板中，模板参数 T 代替了 Object 的作用，并允许静态类型检查（见 27.2 节）。

22.7　建议

[1] 使用虚函数确保无论用什么接口访问对象都执行相同的操作；22.1 节。

[2] 如果在类层次中导航不可避免，使用 dynamic_cast；22.2 节。

[3] 使用 dynamic_cast 进行类型安全的显式类层次导航；22.2.1 节。

[4] 使用 dynamic_cast 转换引用类型，当无法转换到所需类时，会被认为是一个错误；22.2.1.1 节。

[5] 使用 dynamic_cast 转换指针类型，当无法转换到所需类时，只会被认为是一个不同的合法结果；22.2.1.1 节。

[6] 用双重分发或访客模式表达基于两个动态类型的操作（除非你需要优化性能）；22.3.1 节。

[7] 在构造和重构过程中不要调用虚函数；22.4 节。

[8] 使用 typeid 实现扩展的类型信息；22.5.1 节。

[9] 使用 typeid 查询对象的类型（但不要用它查询对象的接口）；22.5 节。

[10] 优选虚函数而不是基于 typeid 或 dynamic_cast 的重复的 switch 语句；22.6 节。

第 23 章
The C++ Programming Language, Fourth Edition

模　板

你的报价在此。

——比雅尼·斯特劳斯特鲁普

- 引言和概述
- 一个简单的字符串模板
  定义模板；模板实例化
- 类型检查
  类型等价；错误检测
- 类模板成员
  数据成员；成员函数；成员类型别名；static 成员；成员类型；成员模板；友元
- 函数模板
  函数模板实参；函数模板实参推断；函数模板重载
- 模板别名
- 源码组织
  链接
- 建议

23.1　引言和概述

模板支持将类型作为参数的程序设计方式，从而实现了对泛型程序设计（见 3.4 节）的直接支持。C++ 模板机制允许在定义类、函数或类型别名时将类型或值作为参数。这提供了一种直接表示各种一般概念的途径，以及组合这些概念的一种简单方法。而且，这样定义的类和函数在运行时间和空间效率上并不逊于手工打造的非通用代码。

模板仅依赖于它真正使用的那些参数类型属性，并不要求参数类型是显式相关的。特别是，模板的参数类型不必是继承层次中的一部分。内置类型作为模板参数类型是允许的，而且也很常见。

模板提供的代码组成是类型安全的（不会隐含地以不符合定义的方式使用对象），但不幸的是，模板对参数的要求并不能简单、直接地用代码表达出来（见 24.3 节）。

所有主要的标准库抽象都是以模板的形式实现的（例如 string、ostream、regex、complex、list、map、unique_ptr、thread、future、tuple 和 function），关键操作也是如此（例如 string 比较、输出运算符 <<、complex 算术运算、list 插入删除以及 sort()）。本书第五部分介绍标准库，其中的章节提供了丰富的模板及相关编程技术示例。

本章内容主要聚焦于设计、实现和使用标准库所需的技术，以此来向读者介绍 C++ 模板的相关知识。较之其他大多数软件，标准库要求更高程度的通用性、灵活性和效率。因此，能用来设计和实现标准库的技术，在设计其他很多问题的求解方案时也会很有效且高

效。这些技术使得程序员能将复杂的实现隐藏在简单的接口之后，而且，当用户需要了解实现细节时，这些技术也有能力将这种复杂性呈现给用户。

本章和接下来的 6 章重点介绍模板和使用模板的基本技术。本章主要介绍最基本的模板特性和使用模板的基本编程技术：

23.2 节　通过一个字符串模板的例子介绍定义和使用类模板的基本机制。

23.3 节　介绍用于模板的类型等价和类型检查的基本规则。

23.4 节　介绍如何定义和使用类模板的成员。

23.5 节　介绍如何定义和使用函数模板以及对函数模板和普通函数如何进行重载解析。

23.6 节　介绍模板别名如何为隐藏实现细节和清理模板符号提供了一种强有力的机制。

23.7 节　介绍如何组织模板代码源文件。

第 24 章将介绍泛型编程的基本技术，并探讨概念（模板对参数的要求）的基本思想：

24.2 节　通过一个例子来介绍从具体（concrete）实例设计泛型（generic）算法的基本技术。

24.3 节　介绍并讨论概念（concept）的基本思想。所谓概念，就是模板为其实参设定的一组要求。

24.4 节　介绍用编译时断言表达概念和使用概念的技术。

第 25 章介绍模板实参传递和特例化的概念：

25.2 节　介绍哪些东西可以作为模板实参：类型、值和模板。并介绍如何指定和使用默认模板实参。

25.3 节　介绍特例化（specialization）。模板针对其一组特定实参的特殊版本称为特例化。特例化可以由编译器从模板生成，也可以由程序员提供。

第 26 章介绍生成模板特例（实例）的名字绑定的一些相关问题：

26.2 节　介绍编译器何时以及如何从模板定义来生成特例以及如何手工指定特例。

26.3 节　指出确定模板定义中使用的一个名字指向哪个实体的规则。

第 27 章讨论由模板支撑的泛型程序设计技术与类层次所支撑的面向对象程序设计技术之间的关系。重点是两者如何组合使用：

27.2 节　介绍模板和类层次是表示一组相关抽象概念的两种方法。我们如何在两者间进行选择呢？

27.3 节　讲解为什么简单地向一个已有的类层次添加模板参数来设计模板类层次通常不是一个好主意。

27.4 节　介绍如何设计类型安全且以性能为目标导向的接口和数据结构。

第 28 章介绍如何将模板用作一种生成函数和类的方法。

28.2 节　介绍类型函数，即接受类型参数或返回类型结果的函数。

28.3 节　介绍如何实现类型函数的选择和递归，并介绍一些使用类型函数的经验法则。

28.4 节　介绍如何有条件地定义函数和使用（几乎）任意谓词重载模板。

28.5 节　介绍如何构建和访问可保存（几乎）任意类型元素的链表。

28.6 节　介绍如何（用一种静态类型安全的方式）定义接受任意数量、任意类型实参的模板。

28.7 节　介绍一个 SI 单元例子，该例组合使用简单元编程技术与其他编程技术设计一个计算库，能在编译时检查计算是否正确使用了米、千克以及秒等单位。

第 29 章展示如何组合使用各种模板特性来解决一个挑战性的设计任务：

29.2 节　介绍如何定义一个 N 维矩阵，具有灵活和类型安全的初始化、下标以及子矩阵操作。

29.3 节　介绍如何实现 N 维矩阵的简单算术运算。

29.4 节　介绍一些有用的实现技术。

29.5 节　介绍一下简单使用矩阵的例子。

在本书中，我很早就介绍了模板（见 3.4.1 节和 3.4.2 节），其使用贯穿本书，因此我假定你对模板已经较为熟悉了。

23.2　一个简单的字符串模板

本节讨论如何实现字符串处理。字符串就是一个能保存字符并提供诸如下标、连接和比较等常见“串”操作的类。我们可能希望为很多不同种类的字符实现这样的类。例如，带符号字符串、无符号字符串、中文字符串、希腊文字符串，等等，这些字符串各有各的应用场景。因此，我们希望“串”概念的表达应该尽量不依赖于特定种类的字符。字符串的定义要求字符是可以拷贝的，还有少量其他要求（见 24.3 节）。因此，我们可以参考 19.3 节中定义的 char 的字符串，将其中的字符类型改为参数，这样就得到一个更通用的字符串模板：

```
template<typename C>
class String {
public:
    String();
    explicit String(const C*);
    String(const String&);
    String operator=(const String&);
    // ...
    C& operator[](int n) { return ptr[n]; }      // 无范围检查的元素访问
    String& operator+=(C c);                     // 将 c 追加到末尾
    // ...
private:
    static const int short_max = 15;             // 用于短字符串优化（见 19.3.3 节）
    int sz;
    C* ptr;    // ptr 指向 sz 个 C
};
```

前缀 template<typename C> 指出将要声明一个模板，而在声明中将用到类型参数 C。像这样引入 C 之后，我们就可以像使用普通类型名一样使用它。C 的作用域一直延伸到以 template<typename C> 为前缀的模板声明的末尾。你也可以使用一个等价但更短的前缀 template<class C>。但即使使用这种形式，C 仍然是一个类型名，而不是一个类名。数学家可能将 template<class C> 看作“对所有 C”或更具体的“对所有类型 C”甚至“对所有是类型的 C”这些习惯陈述的变形。如果沿着这种思路思考，你就会注意到 C++ 缺乏一种完全通用的机制来指明对一个模板参数 C 的要求。即，我们无法用 C++ 陈述“对所有…的 C”，其中“…”表示对 C 的一组要求。换句话说，C++ 没有提供一种直接的方法来陈述希望一个模板参数 C 是什么类型（见 24.3）。

对于一个类模板，如果在其名字后面跟一个用 <> 包围的类型，它就会成为一个类名（由此模板定义的类），我们就可以像使用其他类名一样来使用它。例如：

```
String<char> cs;
String<unsigned char> us;
```

```
String<wchar_t> ws;

struct Jchar { /* ... */ };    // 日文字符

String<Jchar> js;
```

除了名字的特殊语法之外，String<char>与19.3节定义的String类完全一样。将String改为一个模板允许我们将原来为char的String设计的功能扩展到任意种类的字符。例如，如果我们使用标准库map和String模板，19.2.1节中的单词计数程序就变为：

```
int main() // 统计每个单词在输入中出现的次数
{
    map<String<char>,int> m;
    for (String<char> buf; cin>>buf;)
        ++m[buf];
    // ... 输出结果 ...
}
```

针对日文字符的版本为：

```
int main() // 统计每个单词在输入中出现的次数
{
    map<String<Jchar>,int> m;
    for (String<Jchar> buf; cin>>buf;)
        ++m[buf];
    // ... 输出结果 ...
}
```

标准库提供了模板类basic_string，它很像模板化了的String（见19.3节和36.3节）。在标准库中，string是basic_string<char>的一个别名（见36.3节）

```
using string = std::basic_string<char>;
```

这样我们就可以改写单词计数程序如下：

```
int main() // 统计每个单词在输入中出现的次数
{
    map<string,int> m;
    for (string buf; cin>>buf;)
        ++m[buf];
    // ... 输出结果 ...
}
```

一般来说，类型别名（见6.5节）有助于缩短由模板生成的类名的长度。而且，我们通常也并不想了解类型定义的细节，类型别名能帮助我们隐藏类型是由模板生成的这一事实。

23.2.1 定义模板

从类模板生成的类和普通类没什么两样。因此，使用模板并不意味着比一个等价的“手工打造的”类多出一些运行时机制。实际上，使用模板还会减少生成的代码量，因为对类模板来说，只有当一个成员函数被使用时才会为其生成代码（见26.2.1节）

除了类模板之外，C++还提供了函数模板机制（见3.4.2节和23.5节）。我将在介绍类模板时介绍大多数模板相关的技术细节，而将函数模板推迟到23.5节再做介绍。模板本质上是一个说明，描述如何基于给定的恰当的模板实参来生成某些东西。实现这种通用性（实例化（见26.2节）和特例化（见25.3节））的编程语言机制并不太关心到底是生成了一个类还是一个函数。因此，除非特别说明，本章介绍的模板相关的规则都是既适用于类模板，也

适用于函数模板。模板也可以定义为一个别名（见 23.6 节），但 C++ 并不提供其他一些貌似也很合理的语言特性，如名字空间模板。

一些人认为术语类模板（class template）和模板类（template class）在语义上是有差别的。我不同意这种观点，两者即使有差别，也微乎其微，所以，请将这两个术语看作等价的。类似地，我认为函数模板（function template）和模板函数（template function）也是可以互相代替的。

当定义一个类模板时，通常一种好的方式是：先编写调试一个特定类，如 String，然后再将其转换为一个模板，如 String<C>。这样，我们就能针对一个具体实例来处理很多设计问题和大多数代码错误，这种调试对所有程序员来说都很熟悉。而且，较之抽象概念，我们大多数人还是更擅长处理具体实例。随后，我们就可以专心处理那些可能由泛化所引起的问题，而不必再为混杂其中的很多常规错误分心。类似地，当我们尝试理解一个模板时，一个通常很有用的方法是首先设想它对一个特定类型实参如 char 的行为是怎样的，然后再尝试理解它最通用的行为。这也符合我们所习惯的哲学：一个通用组件应该从一个或多个具体实例泛化而得，而不是简单地从第一原理直接设计（见 24.2 节）。

类模板成员的声明和定义与非模板类成员完全一样。模板成员不必定义在模板类中，也可以在外部定义，就像外部定义非模板类成员那样（见 16.2.1 节）。模板类成员本身也是模板，通过所属模板类的参数进行参数化。因此，当在模板类外部定义一个成员时，必须显式声明一个模板。例如：

```
template<typename C>
String<C>::String() // String<C> 的构造函数
    :sz{0}, ptr{ch}
{
    ch[0] = {};// 结尾字符 0，具有恰当的字符类型
}

template<typename C>
String& String<C>::operator+=(C c)
{
    // … 将 c 追加到字符串末尾 …
    return *this;
}
```

像 C 这样的模板参数并不是一个特定类型的名字，而是一个参数，但这并不影响在编写模板代码时将它当作一个类型名来使用。在 String<C> 的作用域中，对模板本身的名字来说限定符 <C> 是多余的，因此构造函数的名字是 String<C>::String。

在一个程序中，一个类成员函数只能由唯一的函数定义，与此类似，在一个程序中，一个类模板成员函数也只能有唯一一个函数模板定义它。不过，特例化（见 25.3 节）使我们能用给定的特定模板实参来提供不同的模板实现。对于类模板成员函数，我们也可以用重载机制为不同实参类型提供不同的函数定义。

我们不能重载一个类模板名，因此，如果在一个作用域中声明了一个类模板，在此作用域中就不能再声明任何其他同名实体了。例如：

```
template<typename T>
class String { /* … */ };

class String { /* … */ };    // 错误：重复定义
```

如果一个类型被用作模板实参，那么它必须提供模板所要求的接口。例如，一个类型如果被用作 Sting 的实参，它就必须提供普通的拷贝操作（见 17.5 节和 36.2.2 节）。注意，同一个模板参数的不同实参并不要求具有继承关系。请参见 25.2.1 节（模板类型参数）、23.5.2 节（模板参数推断）和 24.3 节（对模板实参的要求）

23.2.2 模板实例化

从一个模板和一个模板实参列表生成一个类或一个函数的过程通常被称为模板实例化（template instantiation，见 26.2 节）。一个模板针对某个特定模板实参列表的版本被称为特例化（specialization）。

一般来说，保证从所用的模板实参列表生成模板的特例化是 C++ 实现的责任，而不是程序员的任务。例如：

```
String<char> cs;

void f()
{
    String<Jchar> js;

    cs = "It's the implementation's job to figure out what code needs to be generated";
}
```

对这段代码，C++ 编译器负责为 String<char> 类和 String<Jchar> 类，它们的析构函数和默认构造函数以及 String<char>::operator=(char*) 生成声明。其他成员函数并未使用，因此不会被生成。所生成的类与普通类完全一样，服从普通类的所有基本规则。类似地，生成的函数也和普通函数完全一样，服从普通函数的所有基本规则。

显然，模板提供了一种从相对较短的定义生成大量代码的强有力的方法。但也正因为如此，我们要小心避免几乎相同的函数定义泛滥，占据大量内存（见 25.3 节）。另一方面，模板代码能达到其他方式编写的代码所达不到的质量。特别是，组合使用模板和简单内联来编写程序能消除很多直接或间接的函数调用。例如，关键数据结构上的简单操作（如 sort() 中的 < 操作和矩阵运算中标量的 * 操作）在高度参数化的库中会被约简为单个机器指令。因此，轻率使用模板会生成大量非常相似的函数，从而导致代码膨胀，而正确使用模板则会使很小的函数实现内联，从而和其他方法相比能大幅度缩减代码量，提高运行速度。特别是，为简单的 < 或 [] 生成的代码通常就是单个机器指令，既比任何函数调用都快得多，也比任何需要调用函数取得返回结果的代码短得多。

23.3 类型检查

模板实例化就是从一个模板和一组模板实参来生成代码。由于这些信息在实例化时都能获得，因此从模板定义和模板实参类型来编织这些信息能提供最大程度的灵活性和无与伦比的运行时性能。不幸的是，这种灵活性同时也意味着复杂的类型检查和难以精确报告错误类型。

编译器对模板实例化生成的代码（与程序员手工扩展模板得到的代码完全一样）进行类型检查。生成的代码可能包含很多模板用户听都没听说过的内容（如模板实现细节用到的名字），而在随后的构建过程中，经常是这些内容出现问题。程序员所见 / 所写与编译器所检查之间的这种不匹配可能变成一个大问题，因此我们需要在编写程序时尽量避免此问题带来的不良后果。

模板机制最大的弱点是无法直接表达对模板实参的要求。例如，我们无法这样编写代码：

```
template<Container Cont, typename Elem>
    requires Equal_comparable<Cont::value_type,Elem>()  // 对类型 Cont 和 Elem 的要求
int find_index(Cont& c, Elem e);                        // 在 c 中查找 e 出现的位置
```

即，在 C++ 中我们无法直接陈述 Cont 必须是一个容器类型以及类型 Elem 的值必须能和 Cont 的元素进行比较。将此特性引入未来 C++ 标准的工作已经接近完成（引入此特性不能损失灵活性，运行时性能，也不能显著增加编译时间［Sutton, 2011］），但目前我们还只能用没有此特性的 C++ 编写程序。

有效处理模板实参传递问题的第一步是建立一个用于讨论对模板实参的要求的框架和词汇表。我们可以将一组对模板实参的要求看作一个谓词。例如，可以将“C 必须是一个容器”看作一个谓词，它接受一个类型参数 C，若 C 是一个容器则返回 true（我们应该已经定义了什么是“容器”），否则返回 false。例如，Container<vector<int>>() 和 Container<list<string>>() 应该为真，而 Container<int>() 和 Container<shared_ptr<string>>() 应该为假。我们称这种谓词为概念（concept）。概念（仍然）并非 C++ 中的语言结构，它是一种理念，可以用来推理对模板实参的要求，可以用于注释中，有时可以用我们自己的代码来实现（见 24.3 节）。

初学者可以将一个概念看作一个设计工具：通过一组注释来说明 Container<T>()，指出 T 必须满足什么性质才能使 Container<T>() 为真。例如：

- T 必须有下标运算符（[]）。
- T 必须有成员函数 size()。
- T 必须有成员类型 value_type，它是元素的类型。

注意，这个列表是不完备的（例如，[] 接受什么参数，返回什么结果），也没有说清大多数语义问题（例如，[] 实际完成了什么工作）。但是，即使是要求集合的一个子集也是有用的，即使是很简单的一些说明也能帮助我们手工检查模板的使用、发现一些明显的错误。例如，Container<int>() 显然为假，因为 int 没有下标运算符。在后续章节中我们将回过头来介绍概念的设计（见 24.3 节）、用代码表达概念的技术（见 24.4 节），并给出一个例子展示一些有用的概念（见 24.3.2 节）。现在，你只需知道 C++ 不直接支持概念，但这并不代表概念不存在。对每个可用的模板，其设计者在头脑中都有一些对其实参的概念。正如 Dennis Ritchie 的那句名言：“C 语言是一种强类型、弱检查的语言。”你也可以这样评说 C++ 模板，只不过虽然 C++ 确实会做模板实参要求（概念）的检查，但这种检查是在编译过程中非常晚的时刻进行的，而且是在很低的抽象层次上进行的，帮助有限。

23.3.1　类型等价

给定一个模板，我们可以通过提供模板实参生成类型。例如：

```
String<char> s1;
String<unsigned char> s2;
String<int> s3;

using Uchar = unsigned char;
using uchar = unsigned char;

String<Uchar> s4;
String<uchar> s5;
String<char> s6;
```

```
template<typename T, int N>          // 参见 25.2.2 节
    class Buffer;
Buffer<String<char>,10> b1;
Buffer<char,10> b2;
Buffer<char,20-10> b3;
```

如果对一个模板使用相同的模板实参，我们希望得到相同的生成类型。但是，“相同”的含义是什么？别名并未引入新的类型，因此 String<Uchar> 和 String<uchar> 是与 String<unsigned char> 相同的类型。相反，由于 char 和 unsigned char 是不同类型（见 6.2.3 节），因此 String<char> 和 String<unsigned char> 是不同类型。

编译器可以对常量表达式求值（见 10.4 节），因此 Buffer<char,20-10> 被认为是与 Buffer<char,10> 相同的类型。

对一个模板使用不同模板实参生成的类型是不同类型。特别是，用相关实参生成的类型不一定是相关的。例如，假定 Circle 是一种 Shape：

```
Shape* p {new Circle{p,100}};                  // Circle* 转换为 Shape*
vector<Shape>* q {new vector<Circle>{}};       // 错误：vector<Circle>* 不能转换为 vector<Shape>*
vector<Shape> vs {vector<Circle>{}};           // 错误：vector<Circle> 不能转换为 vector<Shape>
vector<Shape*> vs {vector<Circle*>{}};         // 错误：vector<Circle*> 不能转换为 vector<Shape*>
```

如果允许这些转换，就会导致类型错误（见 27.2.1 节）。如果需要在生成的类之间进行转换，程序员可以定义这种转换操作（见 27.2.2 节）。

23.3.2 错误检测

我们在程序中首先定义模板，随后提供一组模板实参来使用模板。当定义模板时，会检查语法错误和其他可能的错误，当然，这些错误都是与特定的模板实参无关的。例如：

```
template<typename T>
struct Link {
    Link* pre;
    Link* suc                  // 语法错误：漏掉了分号
    T val;
};

template<typename T>
class List {
    Link<T>* head;
public:
    List() :head{7} { }                                  // 错误：用整数初始化指针
    List(const T& t) : head{new Link<T>{0,o,t}} { }      // 错误：未定义标识符 o
    // ...
    void print_all() const;
};
```

编译器可以在模板定义时或稍后使用时检查出简单的语义错误。用户通常希望更早地检查出错误，但并不是所有“简单”错误都能很容易地检测出来。在本例中，我犯了 3 个“错误”：

- 一个简单的语法错误：在一条声明语句的末尾漏掉了分号。
- 一个简单的类型错误：无论模板参数是什么，都不能用整数 7 来初始化一个指针。
- 一个名字查询错误：标识符 o（本应输入 0，误输入了 o）不能作为 Link<T> 的构造函数的实参，因为在此作用域中没有定义这个名字。

模板定义中用到的名字要么是所在作用域中已定义的，要么是明显依赖于模板参数的（见

26.3 节）。最常见、最明显的依赖于一个模板参数 T 的方式就是显式使用名字 T、使用 T 的成员以及接受一个类型为 T 的参数。例如：

```
template<typename T>
void List<T>::print_all() const
{
    for (Link<T>* p = head; p; p=p->suc)    // p 依赖于 T
        cout << *p;                         // << 依赖于 T
}
```

与模板参数使用相关的错误直到使用模板时才能被检测出来。例如：

```
class Rec {
    string name;
    string address;
};
void f(const List<int>& li, const List<Rec>& lr)
{
    li.print_all();
    lr.print_all();
}
```

li.print_all() 完美地通过了类型检查，但 lr.print_all() 给出了一个类型错误，因为 Rec 没有定义 << 输出运算符。与模板参数相关的错误最早也只能在模板第一次使用，给定了特定的模板实参时检测出来。这个时刻被称为*实例化点*（见 26.3.3 节）。C++ 实现实际上可以将所有类型检查都推迟到程序链接时，而确实有一些错误的最早可能发现时刻就是链接时。不管类型检查是什么时候进行的，所应用的检查规则都相同。当然，用户（程序员）还是希望类型检查尽量早进行。

23.4　类模板成员

与普通类一样，模板类可以有几种不同类型的成员：

- 数据成员（变量和常量）；见 23.4.1 节。
- 成员函数；见 23.4.2 节。
- 成员类型别名；见 23.6 节。
- static 成员（函数和数据）；见 23.4.4 节。
- 成员类型（例如，成员类）；见 23.4.5 节。
- 成员模板（例如，成员类模板）；见 23.4.6.3 节。

此外，类模板也可以声明 friend，就像普通类那样，见 23.4.7 节。

类模板成员的规则与生成类成员的规则是一样的。即，如果你想知道一个模板成员的规则有哪些，查找一个普通类成员的规则就行了（见第 16、17 和 20 章）。这样，大部分问题都能找到答案。

23.4.1　数据成员

就像“普通类”一样，类模板可以有任意类型的数据成员。非 static 数据成员可以在其定义时初始化（见 17.4.4 节），也可以在构造函数中初始化（见 16.2.5 节）。例如：

```
template<typename T>
struct X {
    int m1 = 7;
```

```
    T m2;
    X(const T& x) :m2{x} { }
};

X<int> xi {9};
X<string> xs {"Rapperswil"};
```

非 static 数据成员可以是 const 的，但不幸的是不能是 constexpr 的。

23.4.2 成员函数

与“普通类”一样，非 static 成员函数的定义可以在类模板内部，也可以在外部。例如：

```
template<typename T>
struct X {
    void mf1() { /* ... */ }      // 类内定义
    void mf2();
};

template<typename T>
void X<T>::mf2() { /* ... */ }    // 类外定义
```

类似地，类模板的成员函数可以是 virtual 的，也可以不是。但是，一个虚成员函数名不能再用作一个成员函数模板名（见 23.4.6.2 节）。

23.4.3 成员类型别名

我们可以使用 using 或 typedef（见 6.5 节）向类模板引入成员类型别名，它在类模板的设计中起着非常重要的作用。类型别名定义了类的相关类型，定义的方式非常方便类外访问。例如，我们将容器的迭代器和元素类型指定为别名：

```
template<typename T>
class Vector {
public:
    using value_type = T;
    using iterator = Vector_iter<T>;    // Vector_iter 是在其他地方定义的
    // ...
};
```

模板参数名 T 只能被模板自身访问，如果其他代码想使用元素类型，目前我们能用的方法只有提供一个别名。

类型别名在泛型程序设计中起着重要作用，它允许类设计者为来自不同类（和类模板）但具有共同语义的类型提供通用的名字。通过成员别名来表示的类型名通常被称为关联类型（associated type）。在本例中，名字 value_type 和 iterator 的设计借鉴了标准库中的容器的设计（见 33.1.3 节）。如果一个类漏掉了需要的成员别名，可以用类型萃取机制弥补（见 28.2.4 节）。

23.4.4 static 成员

一个类外定义的 staic 数据或函数成员在整个程序中只能有唯一一个定义。例如：

```
template<typename T>
struct X {
    static constexpr Point p {100,250}; // Point 必须是一个字面值常量类型（见 10.4.3 节）
    static const int m1 = 7;
```

```
    static int m2 = 8;                    // 错误：不是 const
    static int m3;
    static void f1() { /* ... */ }
    static void f2();
};

template<typename T> int X<T>::m1 = 88;          // 错误：有两个初始化器
template<typename T> int X<T>::m3 = 99;

template<typename T> void X::<T>::f2() { /* ... */ }
```

与非模板类一样，const 或 constexpr static 的字面值常量类型数据成员可以在类内初始化，不必在类外定义（见 17.4.5 节和 iso.9.2 节）。

一个 static 成员只有真被使用时才需要定义（见 iso.3.2 节、iso.9.4.2 节和 16.2.12 节）。例如：

```
template<typename T>
struct X {
    static int a;
    static int b;
};

int* p = &X<int>::a;
```

如果这些就是程序中所有用到 X<int> 的地方，编译器会报告 X<int>::a "未定义"，而对 X<int>::b 就不会。

23.4.5　成员类型

与"普通类"一样，我们可以将类型定义为类模板的成员。照例，成员类型可以是一个类或是一个枚举。例如：

```
template<typename T>
struct X {
    enum E1 { a, b };
    enum E2;              // 错误：基础类型未知
    enum class E3;
    enum E4 : char;

    struct C1 { /* ... */ };
    struct C2;
};

template<typename T>
enum class X<T>::E3 { a, b };          // 必需的

template<typename T>
enum class X<T>::E4 : char { x, y };   // 必需的

template<typename T>
struct X<T>::C2 { /* ... */ };         // 必需的
```

成员枚举可以在类外定义，但在类内声明中必须给出其基础类型（见 8.4 节）。

非 class 的 enum 的枚举量照例是在枚举类型的作用域中，也就是说，对于一个成员枚举类型来说，枚举量在所在类的作用域中。

23.4.6 成员模板

一个类或一个类模板可以有模板成员，这使得我们表示相关类型时能得到满意的控制度和灵活性。例如，复数类型最好表示为某种标量类型的值对：

```
template<typename Scalar>
class complex {
    Scalar re, im;
public:
    complex() :re{}, im{} {}                    // 默认构造函数
    template<typename T>
    complex(T rr, T ii =0) :re{rr}, im{ii} { }

    complex(const complex&) = default;          // 复制构造函数
    template<typename T>
        complex(const complex<T>& c) : re{c.real()}, im{c.imag()} { }
    // ...
};
```

这种定义允许数学上有意义的复数类型转换，同时禁止不合需要的窄化转换（见 10.5.2.6 节）：

```
complex<float> cf;                  // 缺省值
complex<double> cd {cf};            // 正确：使用 float 向 double 的转换
complex<float> cf2 {cd};            // 错误：不存在隐式的 double->float 转换

complex<float> cf3 {2.0,3.0};       // 错误：不存在隐式的 double->float 转换
complex<double> cd2 {2.0F,3.0F};    // 正确：使用 float 向 double 的转换

class Quad {
    // 没有到 int 的转换
};

complex<Quad> cq;
complex<int> ci {cq};           // 错误：不存在 Quad 向 int 的转换
```

根据 complex 的定义，我们可以从一个 complex<T2> 或是一对 T2 值构造一个 complex<T1> 当且仅当我们可以从一个 T2 构造一个 T1。这看起来是合理的。

要注意的是，从 complex<double> 到 complex<float> 窄化转换的错误直至 complex<float> 的模板构造函数实例化时才会被捕获，而且造成转换错误的唯一原因是我在构造函数的成员初始化列表中使用了 {} 初始化语法（见 6.3.5 节），而这种语法不允许窄化转换。

使用（旧的）() 语法会使我们受到窄化错误的困扰。例如：

```
template<typename Scalar>
class complex {                     // 旧风格
    Scalar re, im;
public:
    complex() :re(0), im(0) { }
    template<typename T>
    complex(T rr, T ii =0) :re(rr), im(ii) { }

    complex(const complex&) = default;      // 复制构造函数
    template<typename T>
        complex(const complex<T>& c) : re(c.real()), im(c.imag()) { }
    // ...
};

complex<float> cf4 {2.1,2.9};  // 很好！窄化转换！
```

```
complex<float> cf5 {cd};      // 哎呀！窄化转换
```

我认为这是应该坚持使用 {} 初始化语法的另一个原因。

23.4.6.1 模板和构造函数

为了尽量减少可能带给读者的困惑，我在上例中显式地添加了一个默认拷贝构造函数。去掉它并不会改变定义的含义：complex 仍然会有一个默认拷贝构造函数。出于技术原因，模板的构造函数从来不会用来生成拷贝构造函数，因此如果我们没有显式声明拷贝构造函数，编译器就会为我们生成一个默认拷贝构造函数。类似地，我们也必须定义非模板的拷贝赋值运算符、移动构造函数以及移动赋值运算符（见 17.5.1 节、17.6 节和 19.3.1 节），否则，编译器就会为我们生成默认的版本。

23.4.6.2 模板和 virtual

成员模板不能是 virtual 的，例如：

```
class Shape {
    // ...
    template<typename T>
        virtual bool intersect(const T&) const =0;      // 错误：虚模板
};
```

这段代码是不合法的。如果 C++ 允许这样的代码，用于实现虚函数机制的传统虚函数表技术（见 3.2.3 节）就无法使用了。每当有人用新的实参类型调用 intersect() 时，链接器就必须向 Shape 类的虚函数表中添加一个对应项。这样增加连接器的复杂性显然是不可接受的。特别是，如果允许虚模板，处理动态链接就会需要与传统方法非常不同的实现技术。

23.4.6.3 使用嵌套

尽量保持信息的局部性通常是一个好主意。这样，就更容易找到一个名字，并且更不容易与程序中的其他东西相互干扰。这种思路就引出了成员类型。将类型定义为成员通常是一种好方法。但是，对于类模板的成员，我们必须考虑参数化对成员类型是否恰当。更形式化地说，一个模板成员依赖于所有模板实参，当成员的行为实际上并未使用所有模板实参时，这种依赖就会不幸产生副作用。一个著名的例子是链表的链接类型。考虑如下定义：

```
template<typename T, typename Allocator>
class List {
private:
    struct Link {
        T val;
        Link* succ;
        Link* prev;
    };
    // ...
};
```

本例中，Link 是 List 的实现细节。因此，看起来这个例子完美地展示了类型最好定义在 List 作用域中，该类型还保持着 private 性质。这已经成为一种流行的设计技术，而且通常能很好地达成目的。但令人惊讶的是，与非局部 Link 类型相比，这种方法可能产生隐含的额外性能开销。假定没有 Link 的成员依赖于 Allocator 参数，而且我们需要使用 List<double,My_allocator> 和 List<double,Your_allocator>，则由于 List<double,My_allocator>::Link 和 List<double,Your_allocator>::Link 是不同类型，因此使用它们的代码不可能是等价的（如果没有聪明的优化器的话）。也就是说，将 Link 定义为成员，而它只使用了 List 的两个模板参

数其中之一的话，就会导致代码膨胀。于是我们考虑 Link 不是成员的设计：

```
template<typename T, typename Allocator>
class List;

template<typename T>
class Link {
    template<typename U, typename A>
        friend class List;
    T val;
    Link* succ;
    Link* prev;
};

template<typename T, typename Allocator>
class List {
    // ...
};
```

我将 Link 的所有成员都声明为 private 的，并授予 List 访问权限。除了将 Link 定义为非局部名字外，这个版本保持了 Link 作为 List 的实现细节的设计初衷。

但如果一个嵌入类的设计初衷并不是作为模板的实现细节呢？即，如果我们需要一个关联类型是为了应对各种各样的用户，这时情况又会是怎样呢？考虑下面的例子：

```
template<typename T, typename A>
class List {
public:
    class Iterator {
        Link<T>* current_position;
    public:
        // ... 常用的迭代器操作 ...
    };

    Iterator<T,A> begin();
    Iterator<T,A> end();
    // ...
};
```

在本例中，成员类型 List<T,A>::Iterator（显然）没有使用第二个模板参数 A。但是，由于 Iterator 是一个成员，它形式上依赖于 A（编译器所了解的就是如此），因此我们就不可能编写出一个处理 List 的函数，使它与如何用分配器构造 List 无关：

```
void fct(List<int>::Iterator b, List<int>::Iterator e)  // 错误：List 接受两个参数
{
    auto p = find(b,e,17);
    // ...
}

void user(List<int,My_allocator>& lm, List<int,Your_allocator>& ly)
{
    fct(lm.begin(),lm.end());
    fct(ly.begin(),ly.end());

}
```

相反，我们需要编写依赖于分配器实参的函数模板：

```
void fct(List<int,My_allocator>::Iterator b, List<int,My_allocator>::Iterator e)
{
    auto p = find(b,e,17);
    // ...
}
```

但这又破坏了我们的 user()：

```
void user(List<int,My_allocator>& lm, List<int,Your_allocator>& ly)
{
    fct(lm.begin(),lm.end());
    fct(ly.begin(),ly.end());          // 错误：fct 接受 List<int,My_allocator>::Iterator
}
```

对此，我们可以将 fct 定义为模板，为每个分配器生成不同的特例。但是，这样每次使用 Iterator 时都会生成一个新的特例，从而导致严重的代码膨胀［Tsafrir, 2009］。解决这个问题的方法是将 Iterator 移出类模板：

```
template<typename T>
struct Iterator {
    Link<T>* current_position;
};

template<typename T, typename A>
class List {
public:
    Iterator<T> begin();
    Iterator<T> end();
    // ...
};
```

这样，从类型角度，第一个模板参数相同的 List 的迭代器就可以相互代替使用了。而这样的结果正是我们所希望的。现在 user() 可以正常工作了，如果 fct() 被定义为一个函数模板，则调用 user() 只会为 fct() 定义生成一个拷贝（实例）。我的经验是“在模板中尽量避免嵌入类型，除非它们真正依赖于所有模板参数。”这条规则其实是规则“在代码中避免不必要的依赖关系”的一个特殊情况。

23.4.7 友元

如 23.4.6.3 所示，模板类可以将函数指定为 friend。考虑 19.4 节中的 Matrix 和 Vector 例子。通常，Matrix 和 Vector 都是模板：

```
template<typename T> class Matrix;

template<typename T>
class Vector {
    T v[4];
public:
    friend Vector operator*<>(const Matrix<T>&, const Vector&);
    // ...
};

template<typename T>
class Matrix {
    Vector<T> v[4];
```

```
public:
    friend Vector<T> operator*<>(const Matrix&, const Vector<T>&);
    // ...
};
```

友元函数名后面的 <> 是必需的，它清楚地指出友元函数是一个模板函数。如果没有 <>，友元函数将被假定是非模板函数。有了上述定义，乘法运算符即可直接访问 Matrix 和 Vector 中的数据：

```
template<typename T>
Vector<T> operator*(const Matrix<T>& m, const Vector<T>& v)
{
    Vector<T> r;
    // ... 使用 m.v[i] 和 v.v[i] 直接访问元素 ...
    return r;
}
```

友元不会影响所在模板类的作用域，也不会影响使用模板的代码所在的作用域。相反，友元函数和运算符是基于其实参类型查找到的（见 14.2.4 节、18.2.5 节和 iso.11.3 节）。与成员函数类似，友元函数只有使用时才会被实例化（见 26.2.1 节）。

类似普通类，类模板也可以指定其他类为 friend。例如：

```
class C;
using C2 = C;

template<typename T>
class My_class {
    friend C;           // 正确：C 是一个类
    friend C2;          // 正确：C2 是类的别名
    friend C3;          // 错误：在作用域中不存在一个类名为 C3
    friend class C4;    // 正确：引入了一个新的类 C4
};
```

友元依赖于模板实参的情况自然是很有趣的。例如：

```
template<typename T>
class my_other_class {
    friend T;                   // 我的参数是我的友元！
    friend My_class<T>;         // My_class 和我的参数一起构成我的友元
    friend class T;             // 错误："class" 是多余的
};
```

照例，友元关系既不能继承也不能传递（见 19.4 节）。例如，即使 My_class<int> 是 My_other_class<int> 的友元且 C 是 My_class<int> 的友元，C 也不会自然成为 My_other_class<int> 的友元。

我们不能直接将一个模板定义为一个类的友元，但我们可以将一个友元声明改为一个模板。例如：

```
template<typename T, typename A>
class List;

template<typename T>
class Link {
    template<typename U, typename A>
        friend class List;
    // ...
};
```

不幸的是，我们无法声明 Link<X> 为 List<X> 的友元。

友元类的设计目的是允许表达一小群紧密相关的概念。但如果友元关系的模式很复杂，几乎可以肯定是一个设计错误。

23.5 函数模板

很多人第一次使用模板是定义并使用诸如 vector（见 31.4 节）、list（见 31.4.2 节）和 map（见 31.4.3 节）这样的容器类，这也是模板最显而易见的用途。不久之后，就会有使用函数模板操作这些容器的需求了。对 vector 中的元素排序就是一个简单的例子：

```
template<typename T> void sort(vector<T>&);          // 声明

void f(vector<int>& vi, vector<string>& vs)
{
    sort(vi);   // sort(vector<int>&);
    sort(vs);   // sort(vector<string>&);
}
```

当调用一个函数模板时，函数实参类型决定了使用哪个模板版本；即，模板实参是从函数实参推断出来的（见 23.5.2 节）。

很自然，我们必须在某处给出函数模板的定义（见 23.7 节）：

```
template<typename T>
void sort(vector<T>& v)                    // 定义
    // 希尔排序（Knuth《计算机程序设计艺术 第3卷》，第84页）
{
    const size_t n = v.size();

    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap)
                if (v[j+gap]<v[j]) {    // 交换 v[j] 和 v[j+gap]
                    T temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
                }
}
```

请比较这个定义和 12.5 节中定义的 sort()。这个模板化的版本更清晰也更简短，因为它能依赖于所排序元素的更多信息。它通常也更快，因为它不依赖于一个比较函数指针。这意味着不需要间接的函数调用，内联一个简单的 < 也更容易。

进一步的简化是使用标准库模板 swap()（见 35.5.2 节），这能将交换操作约简为更自然的形式：

```
if (v[j+gap]<v[j])
    swap(v[j],v[j+gap]);
```

这不会引入任何新的开销，相反由于标准库 swap() 使用了移动语义，还可能有性能提升（见 35.5.2 节）。

在本例中，用运算符 < 进行比较操作。但并不是所有类型都有 < 运算符。这限制了这个 sort() 版本的适用范围，但我们可以通过添加一个参数很容易地消除这个限制（见 25.2.3 节）。例如：

```
template<typename T, typename Compare = std::less<T>>
void sort(vector<T>& v)                    // 定义
    // 希尔排序（Knuth《计算机程序设计艺术 第 3 卷》，第 84 页）
{
    Compare cmp;                           // 创建一个默认的比较对象
    const size_t n = v.size();

    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap)
                if (cmp(v[j+gap],v[j]))
                    swap(v[j],v[j+gap]);
}
```

我们现在既可以用默认的比较操作（<）进行排序，也可以提供自己的比较操作：

```
struct No_case {
    bool operator()(const string& a, const string& b) const;      // 大小写不敏感比较
};

void f(vector<int>& vi, vector<string>& vs)
{
    sort(vi);                              // sort(vector<int>&)
    sort<int,std::greater<int>>(vi);       // sort(vector<int>&) 使用 greater

    sort(vs);                              // sort(vector<string>&)
    sort<string,No_case>(vs);              // sort(vector<string>&) 使用 No_case
}
```

不幸的是，C++ 只允许指定末尾的模板实参，这一规则使得我们在指定比较操作时也必须同时指定元素类型（而不是通过推断获得）。

我将在 23.5.2 节中介绍函数模板实参的显式说明。

23.5.1 函数模板实参

对于编写可用于各种容器类型的通用算法（见 3.4.2 节和 32.2 节）来说，函数模板是必不可少的。而对于一次函数模板调用，从函数实参推断出模板实参的能力则是函数模板机制的关键。

编译器可以从一次调用中推断类型和非类型模板实参，前提是函数实参列表唯一标识出模板实参集合。例如：

```
template<typename T, int max>
struct Buffer {
    T buf[max];
public:
    // ...
};

template<typename T, int max>
T& lookup(Buffer<T,max>& b, const char* p);

Record& f(Buffer<string,128>& buf, const char* p)
{
    return lookup(buf,p); // 使用 lookup()，其中 T 是 string 类型，max 为 128
}
```

在本例中，lookup() 的 T 被推断为 string，max 被推断为 128。

注意，类模板参数并不是靠推断来确定的。原因在于一个类可以有多个构造函数，这种灵活性使得实参推断在很多情况下不可行，而在更多情况下会得到含混不清的结果。取而代之，类模板依赖特例化（见 25.3 节）机制在可选的定义中隐式地进行选择。如果我们需要基于推断出的类型创建一个对象，常用的方法是通过调用一个函数来进行推断（以及对象创建）。例如，考虑标准库 make_pair()（见 34.2.4.1 节）的一个简单变体：

```
template<typename T1, typename T2>
pair<T1,T2> make_pair(T1 a, T2 b)
{
    return {a,b};
}

auto x = make_pair(1,2);                    // x 是 pair<int,int>
auto y = make_pair(string("New York"),7.7); // y 是 pair<string,double>
```

如果不能从函数实参推断出一个模板实参（见 23.5.2 节），我们就必须显式指定它。这与模板类显式指定模板实参的方法一样（见 25.2 节和 25.3 节）。例如：

```
template<typename T>
T* create();                    // 创建一个 T，返回指向它的指针

void f()
{
    vector<int> v;              // 类模板，实参为 int
    int* p = create<int>();     // 函数模板，实参为 int
    int* q = create();          // 错误：无法推断模板实参
}
```

这种通过显式说明来确定函数模板返回类型的方法很常用。采用这种方法，我们可以定义一族对象创建函数（如 create()）或是一族类型转换函数（见 27.2.2 节）。这种显式限定函数模板的语法与 static_cast、dynamic_cast 等（见 11.5.2 节和 22.2.1 节）的语法相似。

在某些情况下，可以用默认模板实参简化显式限定（见 25.2.5.1 节）。

23.5.2　函数模板实参推断

如果一个模板函数实参的类型是下列结构的组合，则编译器可以从此函数实参推断出一个类型模板实参 T 或 TT，或是非类型模板实参 I（见 iso.14.8.2.1 节）。

T	const T	volatile T
T *	T&	T [constant_expression]
type [I]	class_template_name<T>	class_template_name<i>
TT<T>	T<i>	T<>
T type:: *	T T:: *	type T:: *
T (*)(args)	type(T:: *)(args)	T(type:: *)(args)
type (type:: *)(args_TI)	T (T:: *)(args_TI)	type(T:: *)(args_TI)
T (type:: *)(args_TI)	type (*)(args_TI)	

表中的 args_TI 是一个参数列表，对其递归地应用推断规则，可以推断出一个 T 或一个 I，而 args 则是不允许进行推断的参数列表。如果有参数不能用这种方法推断出来，则调用会有二义性。例如：

```
template<typename T, typename U>
void f(const T*, U(*)(U));

int g(int);

void h(const char* p)
{
    f(p,g);    // T 是 char，U 是 int
    f(p,h);    // 错误：不能推断 U
}
```

观察第一次调用 f() 的实参，我们可以很容易地推断出模板实参。但对于第二次 f() 调用，我们可以看到 h() 不匹配模式 U(*)(U)，因为它的参数类型和返回类型不同。

如果一个以上的函数实参都可以推断出一个模板参数，则多次推断的结果必须是一致的。否则，调用就是错误的。例如：

```
template<typename T>
void f(T i, T* p);

void g(int i)
{
    f(i,&i);              // 正确
    f(i,"Remember!");     // 错误，二义性：T 是 int 还是 const char*？
}
```

23.5.2.1 引用推断

对左值和右值采取不同的处理措施有时很有必要。考虑一个保存 {整数，指针} 对的类：

```
template<typename T>
class Xref {
public:
    Xref(int i, T* p)       // 保存一个指针：Xref 是所有者
        :index{i}, elem{p}, owner{true}
    { }

    Xref(int i, T& r)       // 保存一个指向 r 的指针，所有者是其他人
        :index{i}, elem{&r}, owner{false}
    { }

    Xref(int i, T&& r)      // 将 r 移入 Xref，Xref 变为所有者
        :index{i}, elem{new T{move(r)}}, owner{true}
    { }

    ~Xref()
    {
        if(owned) delete elem;
    }
    // ...
private:
    int index;
    T* elem;
    bool owned;
};
```

于是：

```
string x {"There and back again"};

Xref<string> r1 {7,"Here"};                    //r1 拥有字符串 {"Here"} 的一份拷贝
Xref<string> r2 {9,x};                         //r2 指向 x
Xref<string> r3 {3,new string{"There"}};       //r3 拥有字符串 {"There"}
```

在这段代码中，r1 的构造会使用构造函数 Xref(int,string&&)，因为 x 是一个右值。类似地，r2 的构造会选择 Xref(int,string&)，因为 x 是一个左值。

模板实参推断过程中是区分左值和右值的：X 类型的左值会被推断为一个 X&，而右值被推断为 X。这与非模板实参的处理不同：值会绑定到非模板实参右值引用（见 12.2.1 节）。但这一规则对实参转发（见 35.5.1 节）特别有用。考虑编写一个工厂函数，它在自由存储区创建 Xref 对象，并返回指向这些对象的 unique_ptr：

```
template<typename T>
    T&& std::forward(typename remove_reference<T>::type& t) noexcept; // 见 35.5.1 节
template<typename T>
    T&& std::forward(typename remove_reference<T>::type&& t) noexcept;
template<typename TT, typename A>
unique_ptr<TT> make_unique(int i, A&& a)    // make_shared（见 34.3.2 节）的简单变体
{
    return unique_ptr<TT>{new TT{i,forward<A>(a)}};
}
```

我们希望 make_unique<T>(arg) 从 arg 构造出一个 T，且不产生任何额外的拷贝。为了实现这一点，必须区分左值和右值。考虑下面这种用法：

```
auto p1 = make_unique<Xref<string>>(7,"Here");
```

"Here" 是一个右值，因此会调用 forward(string&&)，向下传递一个右值，于是会调用 Xref(int,string&&) 移动保存 "Here" 的 string。

最有趣的（也是最微妙的）情况是：

```
auto p2 = make_unique<Xref<string>>(9,x);
```

在这条语句中，x 是一个左值，因此会调用 forward(string&)，将一个左值传递下去：forward() 的 T 被推断为 string&，于是返回值变为 string& &&，也就是 string&（见 7.7.3 节）。因此，对左值 x 会调用 Xref(int,string&)，x 会被拷贝。

不幸的是，make_unique() 不是标准库的一部分，但这种方法得到了广泛应用。使用一个用于参数转发的可变参数模板（见 28.6.3 节）来定义一个接受任意实参的 make_unique() 相对来讲是比较容易的。

23.5.3 函数模板重载

我们可以声明多个同名的函数模板，甚至函数模板和普通函数也可以同名。当调用一个重载函数时，就必须利用重载解析机制找到正确的函数或函数模板进行调用。例如：

```
template<typename T>
    T sqrt(T);
template<typename T>
    complex<T> sqrt(complex<T>);
double sqrt(double);

void f(complex<double> z)
```

```
{
    sqrt(2);    // sqrt<int>(int)
    sqrt(2.0);  // sqrt(double)
    sqrt(z);    // sqrt<double>(complex<double>)
}
```

函数模板是函数概念的泛化，与此类似，函数模板的重载解析规则是普通函数重载解析规则的泛化。基本原则是：对每个模板，我们要找到对给定函数实参集合来说最佳的特例化版本。接下来，我们对这些特例化版本和所有普通函数应用普通函数重载解析规则（见iso.14.8.3节）：

[1] 找到参与重载解析的所有函数模板特例化版本（见23.2.2节）。具体方法是检查每个函数模板，确定如果作用域中没有其他同名函数模板或函数的话，哪些模板实参会被使用（如果有的话）。对于调用sqrt(z)，sqrt<double>(complex<double>)和sqrt<complex<double>>(complex<double>)将成为候选。见23.5.3.2节。

[2] 如果两个函数模板都可以调用，且其中一个比另一个更特殊化（见25.3.3节），则接下来的步骤只考虑最特殊化的版本。对于调用sqrt(z)，sqrt<double>(complex<double>)比sqrt<complex<double>>(complex<double>)更特殊化，因为任何匹配sqrt<T>(complex<T>)的调用也都能匹配sqrt<T>(T)。

[3] 对前两个步骤后还留在候选集中的函数模板和所有候选普通函数一起进行重载解析，方法与普通函数重载解析相同（见12.3节）。如果一个函数模板实参是通过模板实参推断（见23.5.2节）确定的，则不能再对它进行提升、标准类型转换或用户自定义类型转换。对sqrt(2)，sqrt<int>(int)是精确匹配，因此它优于sqrt(double)。

[4] 如果一个普通函数和一个特例化版本匹配得一样好，那么优先选择普通函数。因此，对sqrt(2.0)，sqrt(double)优于sqrt<double>(double)。

[5] 如果没发现任何匹配，则调用是错误的。如果我们最终得到多个一样好的匹配，则调用有二义性，这也是一个错误。

例如：

```
template<typename T>
T max(T,T);

const int s = 7;

void k()
{
    max(1,2);       // max<int>(1,2)
    max('a','b');   // max<char>('a','b')
    max(2.7,4.9);   // max<double>(2.7,4.9)
    max(s,7);       // max<int>(int(s),7)（使用了简单的类型转换）

    max('a',1);     // 错误，有二义性：max<char,char>() 还是 max<int,int>()？
    max(2.7,4);     // 错误，有二义性：max<double,double>() 还是 max<int,int>()？
}
```

最后两个调用的问题是，如果模板参数已经唯一确定，我们就不能对其应用提升和标准类型转换了。因此，在本例中没有任何一条规则能告诉编译器哪个版本更优。在大多数情况下，语言规则将一些微妙的问题交给程序员来做决定应该是好事。编译器可以不给出意料之外的

二义性错误，但取而代之的就是意料之外的解析会带来令人吃惊的结果。不同人对重载解析的“直觉”可能大相径庭，因此不可能设计出一组完美吻合直觉的重载解析规则。

23.5.3.1　二义性消解

可以通过显式限定来消解二义性：

```
void f()
{
    max<int>('a',1);          // max<int>(int('a'),1)
    max<double>(2.7,4);       // max<double>(2.7,double(4))
}
```

也可以通过添加适当的声明来消解二义性：

```
inline int max(int i, int j) { return max<int>(i,j); }
inline double max(int i, double d) { return max<double>(i,d); }
inline double max(double d, int i) { return max<double>(d,i); }
inline double max(double d1, double d2) { return max<double>(d1,d2); }

void g()
{
    max('a',1);       // max(int('a'),1)
    max(2.7,4);       // max(2.7,4)
}
```

对这些普通函数，编译器应用普通重载规则（见 12.3 节），而且使用 inline 确保没有额外开销。

这里 max() 的定义有些小儿科，我们本可以直接实现比较操作而不是调用模板 max() 的特例化版本。但是，使用模板的显式特例化是一种消解函数模板二义性的简单方法，而且可以避免在多个函数中出现几乎相同的代码，从而有利于代码维护。

23.5.3.2　实参代入失败

当对函数模板的一组实参查找最佳匹配时，编译器会检查实参的使用是否符合函数模板完整声明（包括返回类型）的要求。例如：

```
template<typename Iter>
typename Iter::value_type mean(Iter first, Iter last);

void f(vector<int>& v, int* p, int n)
{
    auto x = mean(v.begin(),v.end());    // 正确
    auto y = mean(p,p+n);                // 错误
}
```

在本例中，x 能够成功初始化，因为实参匹配 mean() 的声明，且 vector<int>::iterator 有一个名为 value_type 的成员。y 的初始化失败，因为虽然实参是匹配的，但 int* 没有名为 value_type 的成员。因此，我们不可能有这样的实例化版本：

```
int*::value_type mean(int*,int*);        // int* 没有名为 value_type 的成员
```

但是，如果还有另外一个 mean() 定义，又会怎样呢？

```
template<typename Iter>
typename Iter::value_type mean(Iter first, Iter last);      // 1 号

template<typename T>
T mean(T*,T*);                                              // 2 号

void f(vector<int>& v, int* p, int n)
```

```
{
    auto x = mean(v.begin(),v.end());    // 正确：调用 1 号
    auto y = mean(p,p+n);                // 正确：调用 2 号
}
```

这段代码中的两个初始化都能成功。但为什么当我们试图将 mean(p,p+n) 与第一个模板定义匹配时，没有得到编译错误呢？原因是，实参匹配是完美的，但当代入实际模板实参（int*）后，我们得到一个函数声明：

```
int*::value_type mean(int*,int*);    // int* 没有名为 value_type 的成员
```

这当然是一个无用的声明，因为一个指针不可能有名为 value_type 的成员。幸运的是，仅仅考虑一下这个可能的声明并不会构成一个错误。C++ 语言规定（见 iso.14.8.2 节），这种代入失败（substitution failure）并不是错误，它只会导致模板被忽略；即，这个模板的特例化版本不会作为候选。于是，mean(p,p+n) 将会匹配 2 号声明，这也是最终调用的版本。

如果没有“代入失败并不是错误”这条规则，即使存在无错的可选声明（如 2 号声明），我们也会得到编译错误。而且，这条规则提供了一种选择模板的通用工具。我将在 28.4 节介绍基于这条规则的技术。特别是，标准库提供了 enable_if 来简化模板的条件定义（见 35.4.2 节）。

这条规则有一个为人熟知的、无法发音的缩写 SFINAE（Substituition Failure Is Not An Error）。SFINAE 通常被用作动词，“F”发“v”的音：“我 SFINAE 掉了那个构造函数。”这听起来令人印象很深刻，但我倾向于避免使用这种术语。“这个构造函数由于代入失败被排除了”对大多数人来说更为清楚，而且更符合语法习惯。

因此，在生成一个候选函数来解析一个函数调用的过程中，如果编译器发现生成一个模板特例化是无意义的，就不会将它加入重载候选集中。如果一个模板特例化版本会导致类型错误，它就被认为是无意义的。在此，我们只考虑声明，模板函数定义和类成员函数的定义除非真被使用，否则不在考虑范围之内（也不会被生成）。例如：

```
template<typename Iter>
Iter mean(Iter first, Iter last)                    // 1 号
{
    typename Iter::value_type = *first;
    // ...
}

template<typename T>
T* mean(T*,T*);                                     // 2 号
void f(vector<int>& v, int* p, int n)
{
    auto x = mean(v.begin(),v.end());    // 正确：调用 1 号
    auto y = mean(p,p+n);                // 正确：调用 2 号
}
```

mean() 的 1 号声明与 mean(p,p+n) 也是匹配的。但由于类型错误，编译器并未实例化这个版本，而是将其删除了。

本例存在二义性错误。假如我们并未给出 2 号 mean()，则编译器会为 mean(p, p+n) 调用选择 1 号版本，我们就会得到一个实例化错误。因此，一个函数即使被选为最佳匹配，仍然可能编译失败。

23.5.3.3 重载和派生

重载解析规则保证函数模板能完美地和继承机制结合使用：

```
template<typename T>
    class B { /* ... */ };
template<typename T>
    class D : public B<T> { /* ... */ };

template<typename T> void f(B<T>*);

void g(B<int>* pb, D<int>* pd)
{
    f(pb);          // 当然选择 f<int>(pb)
    f(pd);          // f<int>(static_cast<B<int>*>(pd))
                    // 使用从 D<int>* 到 B<int>* 的类型转换
}
```

在本例中，对任意类型 T 函数模板 f() 接受一个参数 B<T>*。第二个调用的实参类型为 D<int>*，因此编译器很容易推断出 T 为 int，此调用唯一地解析为 f(B<int>*)。

23.5.3.4　重载和非推断的参数

对于在模板参数推断中未用到的函数实参，其处理方式与非模板函数实参完全相同。特别是，常用类型转换规则仍然有效。考虑下面的代码：

```
template<typename T, typename C>
T get_nth(C& p, int n);   // 获取第 n 个元素
```

此函数返回容器中第 n 个元素的值，元素类型为 C。由于 C 是在调用 get_nth() 时从实参推断出来的，因此对第一个参数不进行类型转换。但是，第二个参数是一个普通参数，因此会考虑全部对它可能的类型转换。例如：

```
struct Index {
    operator int();
    // ...
};

void f(vector<int>& v, short s, Index i)
{
    int i1 = get_nth<int>(v,2); // 严格匹配
    int i2 = get_nth<int>(v,s); // short 到 int 的标准类型转换
    int i3 = get_nth<int>(v,i); // 用户自定义类型转换：Index 到 int
}
```

这种语法有时被称为显式特例化（explicit specialization）（见 23.5.1 节）。

23.6　模板别名

我们可以用 using 语法或 typedef 语法为一个类型定义别名（见 6.5 节）。using 语法更常用，一个重要原因是它能用来为模板定义别名，模板的一些参数可以固定。考虑下面的代码：

```
template<typename T, typename Allocator = allocator<T>> vector;

using Cvec = vector<char>;                  // 两个参数都固定了

Cvec vc = {'a', 'b', 'c'};                  // vc 的类型为 vector<char,allocator<char>>

template<typename T>
using Vec = vector<T,My_alloc<T>>;          // vector 使用了我的分配器（第 2 个参数固定）

Vec<int> fib = {0, 1, 1, 2, 3, 5, 8, 13};   // fib 的类型为 vector<int,My_alloc<int>>
```

一般来说，如果绑定一个模板的所有参数，我们就会得到一个类型，但如果只绑定一部分，我们得到的还是一个模板。注意，我们在别名定义中从 using 得到的永远是一个别名。即，当使用别名时，与使用原始模板是完全一样的。例如：

```
vector<char,alloc<char>> vc2 = vc;          // vc2 和 vc 是相同类型
vector<int,My_alloc<int>> verbose = fib;    // verbose 和 fib 是相同类型
```

别名和原始模板的等价性暗示：当你在使用别名时，如果用到了模板特例化，就会（正确）得到特例化版本。例如：

```
template<int>
struct int_exact_traits { 系统：int_exact_traits<N>::type 是一个 N 位的类型
    using type = int;
};

template<>
struct int_exact_traits<8> {
    using type = char;
};
template<>
struct int_exact_traits<16> {
    using type = short;
};

template<int N>
using int_exact = typename int_exact_traits<N>::type; // 定义简便的别名

int_exact<8> a = 7; // int_exact<8> 是一个 8 位整型
```

如果在别名中并未用到特例化，我们就不能简单认为 int_exact 是 int_exact_traits<N>::type 的一个别名，此时两者的行为是不同的。另一方面，你不能定义别名的特例化版本。如果这样做了，代码的读者就很容易困惑到底特例化了什么，因此 C++ 不提供特例化别名的语法。

23.7 源码组织

用模板组织源码有三种很明显的方式：

[1] 在一个编译单元中，在使用模板前包含其定义。

[2] 在一个编译单元中，在使用模板前（只）包含其声明。在编译单元中稍后的位置包含模板定义（可能在使用之后）。

[3] 在一个编译单元中，在使用模板前（只）包含其声明。在其他编译单元定义模板。

由于技术和历史原因，C++ 并不支持第三种方法，即模板定义和使用的分离编译。目前最常用的方法是在每个用到模板的编译单元中都包含（通常使用 #include）模板定义，优化编译时间和消除目标代码冗余的任务就交给编译器了。例如，我可能在一个头文件 out.h 中提供模板 out()：

```
// 文件 out.h:

    #include<iostream>

    template<typename T>
    void out(const T& t)
```

```
{
    std::cerr << t;
}
```

在需要使用 out() 的地方，我们用 #include 包含头文件。例如：

```
// 文件 user1.cpp:

#include "out.h"
// 使用 out()
```

以及

```
// 文件 user2.cpp:

#include "out.h"
// 使用 out()
```

即，我们在多个不同的编译单元中都 #include out() 的定义和它所依赖的声明。（仅）在需要时生成代码以及优化读取冗余定义过程的任务则交由编译器负责。这种处理模板函数的策略与处理内联函数相同。

这种策略的一个明显问题是用户可能偶然依赖本来只是为 out() 的定义才包含的声明。我们可以采用方法［2］"稍后包含模板定义"、使用名字空间、避免使用宏以及更一般地通过减少包含的信息量等方式来限制这种危险。理想情况是最小化一个模板定义对环境的依赖。

为了对我们简单的 out() 例子使用"稍后包含模板定义"的方法，我们首先将 out.h 一分为二，将声明放到一个 .h 文件中：

```
// 文件 outdecl.h:

template<typename T>
void out(const T& t);
```

定义放到 .cpp 文件中：

```
// 文件 out.cpp:

#include<iostream>

template<typename T>
void out(const T& t)
{
    std::cerr << t;
}
```

用户现在就可以分别包含这两个文件了：

```
// 文件 user3.cpp:

#include "out.h"
// 使用 out()
#include "out.cpp"
```

这将模板实现对用户代码造成不良影响的机会降到了最低。不幸的是，它也增加了用户代码中的某些东西（比如宏）对模板定义造成不良影响的机会。

照例，非 inline、非模板函数和 static 成员（见 16.2.12 节）必须有唯一定义，位于某个编译单元中。这意味着这种成员最好不要用于那些需要包含在很多编译单元中的模板。如 out() 例子所示，一个模板函数的定义可能在不同编译单元中重复，因此要小心代码上下文

可能会微妙地改变模板定义的含义：

```
// 文件 user1.cpp:

    #include "out.h"
    // 使用 out()
```

以及

```
// 文件 user4.cpp:

    #define std MyLib
    #include "out.c"
    // 使用 out()
```

这段代码中的宏定义非常糟糕也容易出错，它改变了 out 的定义，因此 user4.cpp 中的 out 定义与 user1.cpp 中不同。这是一个错误，而且是编译器可能无法发现的错误。在一个大型程序中检查这种错误非常困难，因此，要小心地减少模板对上下文的依赖，并非常警惕宏（见 12.6 节）。

如果你需要对实例化代码的上下文有更多控制，可以使用显式实例化和 extern template（见 26.2.2 节）。

23.7.1 链接

模板链接规则实际上是生成的类和函数的链接规则（见 15.2 节和 15.2.3 节）。这意味着如果一个类模板的布局或是一个内联函数模板的定义发生了改变，所有使用该类或该函数的代码都要重新编译。

对定义在头文件中且被“到处”包含的模板来说，这意味着大量的重新编译工作，因为模板更趋向于在头文件中包含大量信息，比使用 .cpp 文件的非模板代码多得多。特别是，如果使用了动态链接库，就要特别小心所有使用模板的地方使用的应该是一致的定义。

有时，我们可以通过将模板的使用封装在非模板接口的函数中来降低复杂模板库中代码修改带来的风险。例如，我们可能要用支持多种类型的通用数值计算库（见第 29 章、40.4 节、40.5 节以及 40.6 节）实现一些计算。但实际上，我们通常预先知道是针对什么类型进行计算的。例如，在一个程序中我们可能只处理 double 类型数据并使用 vector<double>。在此情况下，我们可以定义：

```
double accum(const vector<double>& v)
{
    return accumulate(v.begin(),v.end(),0.0);
}
```

这样，我们就可以在代码中对 accum() 使用简单的非模板声明：

```
double accum(const vector<double>& v);
```

对 std::accumulate 的依赖已经消失，隐藏进一个 .cpp 文件中，我们的其他代码再也看不到这种依赖关系了。而且，#include<numeric> 所带来的编译时开销也只存在于这个 .cpp 文件中。

注意，我们抓住机会简化了 accum() 的接口（与 std::accumulate() 相比）。通用性是一个好的模板库的关键属性，但在一个特定应用中，它也可能被视为导致复杂性的罪魁祸首。

我很怀疑我会不会对标准模板库使用这个技术。标准模板库已经保持稳定很多年了，其

实现也为大家所熟知。特别是，我不会费心去做封装 vector<double> 这样的事。但是，对于更复杂难懂或是频繁改变的模板库，这种封装有时是有用的。

23.8　建议

[1] 使用模板表示用于很多实参类型的算法；23.1 节。
[2] 使用模板表示容器；23.2 节。
[3] 注意 template<class T> 和 template<typename T> 含义相同；23.2 节。
[4] 当设计一个模板时，首先设计并调试非模板版本；随后通过添加参数将其泛化；23.2.1 节。
[5] 模板是类型安全的，但类型检查的时机太晚了；23.3 节。
[6] 当设计一个模板时，仔细思考概念——它对模板实参的要求；23.3 节。
[7] 如果一个类模板必须是可拷贝的，为它定义一个非模板拷贝构造函数和一个非模板拷贝赋值运算符；23.4.6.1 节。
[8] 如果一个类模板必须是可移动的，为它定义一个非模板移动构造函数和一个非模板移动赋值运算符；23.4.6.1 节。
[9] 虚函数成员不能是模板成员函数；23.4.6.2 节。
[10] 只有当一个类型依赖类模板的所有实参时才将其定义为模板成员；23.4.6.3 节。
[11] 使用函数模板推断类模板实参类型；23.5.1 节。
[12] 对多种不同实参类型，重载函数模板来获得相同的语义；23.5.3 节。
[13] 借助实参代入失败机制为程序提供正确的候选函数集；23.5.3.2 节。
[14] 使用模板别名简化符号、隐藏实现细节；23.6 节。
[15] C++ 不支持模板分别编译；在每个用到模板的编译单元中都 #include 模板定义；23.7 节。
[16] 使用普通函数作为接口编写不能用模板处理的代码；23.7.1 节。
[17] 将大的模板和较严重依赖上下文的模板分开编译；23.7 节。

泛型程序设计

是时候将你的工作建立在坚实的理论基础之上了。

——乔舒·摩根

- 引言
- 算法和提升
- 概念
  发现概念；概念和约束
- 具体化概念
  公理；多实参概念；值概念；约束检查；模板定义检查
- 建议

24.1 引言

模板是什么？或者换句话说，当使用模板时，什么程序设计技术更有效？模板提供了：

- 传递类型参数（像传递值和模板一样）而不丢失信息的能力。这意味着有大好机会进行内联，当前的 C++ 实现的确充分利用了这一点。
- 推迟的类型检查（在实例化时进行）。这意味着有机会将来自不同上下文的信息编织在一起。
- 传递常量参数的能力。这意味着能进行编译时计算。

换句话说，模板为编译时计算和类型处理提供了一种强有力的机制，可以生成非常紧凑和高效的代码。记住：类型（类）既可以包含代码也可以包含值。

模板的首要用途，也是它最常见的用途，是支持泛型程序设计（generic programming），即关注通用算法设计、实现和使用的程序设计。在这里，“通用”的含义是算法可以接受各种各样的实参类型，只要这些类型满足算法对实参的要求即可。模板是 C++ 支持泛型程序设计的主要特性。它提供了（编译时）参数化多态。

人们对“泛型程序设计”有很多定义，造成了一定程度上的术语混淆。但是，在 C++ 的语境中，“泛型程序设计”强调用模板实现通用算法的设计。

更多地关注代码生成技术（将模板看作类型和函数的生成器），并依赖类型函数表示编译时计算的编程方式被称为模板元程序设计（template metaprogramming），将在第 28 章中进行介绍。

模板的类型检查是在模板定义时检查实参的使用，而不是（在模板声明中）检查显式的接口。这提供了通常所说的鸭子类型（duck typing，“如果它走路像鸭子，叫起来也像鸭子，那么它就是一只鸭子”）的一个编译时变体。或者用技术术语表达：我们对值进行操作，而操作的表示和含义仅依赖于要处理的值。这不同于另一种观点：对象具有类型，而类型决定了操作的表示和含义，而值是“活在”对象中的。这是 C++ 处理对象（如变量）的方式，只

有满足对象要求的值才能保存在对象中。而在编译时使用模板所做的事情并不涉及对象，只涉及值。特别是，在编译时是没有变量的。因此，模板程序设计很像在用动态类型语言编程，但它并没有运行时开销，而且在动态类型语言中呈现为运行时异常的错误在 C++ 中变成了编译时错误。

泛型程序设计和元程序设计的一个关键特征是内置类型和用户自定义类型的一致处理，这可能也是所有使用模板的编程风格的共同特征。例如，accumulate() 操作并不关心它累加的值的类型是 int、complex<double> 还是 Matrix。它关心的是它们能否用 + 运算符进行加法运算。将类型用作模板实参并不意味着或是要求使用类层次或是任何形式的运行时对象类型自识别。这在逻辑上很合理，会令程序员更加轻松愉快，对高性能应用来说也是必要的。

本节关注泛型程序设计的两个方面。

- 提升：泛化一个算法，使之能适用最大（但合理）范围的实参类型（见 24.2 节）。即，限制一个算法（或一个类）只依赖必要的属性。
- 概念：周密且严谨地说明一个算法（或一个类）对其实参的要求（见 24.3 节）。

24.2 算法和提升

函数模板就是普通函数的泛化：它能对多种数据类型执行动作，并且能用以参数方式传递来的各种操作实现要执行的动作。算法（algorithm）就是一个求解问题的过程或公式：通过一个有穷的计算序列生成结果。因此，函数模板通常也称为算法。

我们如何将一个在特定数据类型上执行特定操作的函数泛化为一个在多种数据类型上执行更通用操作的算法呢？最有效的方法是从一个（多个可能更好）具体实例来泛化出一个好的算法。这种泛化过程就称为提升（lifting）：即，从特殊函数提升为一个通用算法。在这样一个由具体到抽象的过程中，最重要的一点是保持性能并注意如何做才合理。过分聪明的程序员可能试图覆盖所有可能的类型和操作，这就把泛化推到一个不合理的程度了。因此，试图在缺乏具体实例的情况下直接从基本原理进行抽象，通常会使代码臃肿不堪，难以使用。

我将展示从一个具体实例提升出算法的过程。考虑下面的函数：

```
double add_all(double* array, int n)
    // 一个处理 double 数组的具体算法
{
    double s {0};
    for (int i = 0; i<n; ++i)
        s = s + array[i];
    return s;
}
```

显然，这个函数计算实参数组中的 double 值之和。再考虑下面的代码：

```
struct Node {
    Node* next;
    int data;
};

int sum_elements(Node* first, Node* last)
    // 另一个处理 int 链表的具体算法
{
    int s = 0;
    while (first!=last) {
        s += first->data;
```

```
        first = first->next;
    }
    return s;
}
```

这个函数计算单向链表中 int 值之和，链表是基于 node 类型实现的。

这两段代码在细节和风格上都很不同，但一个有经验的程序员会立刻说："这不过是累加算法的两个具体实现而已。"这是一个很流行的算法，和大多数流行算法一样，它有很多名字，包括归约、折叠、求和、累加，等等。让我们尝试以这两个具体算法为起点，逐步设计出一个通用算法，从而熟悉提升的流程。首先，我们尝试进行抽象，去掉数据类型，使得我们不必再明确说明

- double 还是 int；
- 数组还是链表。

为了实现这一点，我们编写如下的伪代码：

```
// 伪代码：

T sum(data)
    // 按某种方式用恰当容器类型进行参数化
{
    T s = 0
    while (not at end) {
            s = s + current value
            get next data element
    }
    return s
}
```

为了具体化这段代码，我们需要三个访问"容器"数据结构的操作：

- 未达末尾；
- 获取当前值；
- 获取下一个数据元素。

对于值类型，我们也需要三个操作：

- 初始化为 0；
- 加法运算；
- 返回结果。

显然，这种描述相当不严谨，但我们可以将其转换为如下的代码：

```
// 类 STL 的具体代码：

template<typename Iter, typename Val>
Val sum(Iter first, Iter last)
{
    Val s = 0;
    while (first!=last) {
        s = s + *first;
        ++first;
    }
    return s;
}
```

我了解 STL 表示值序列的常用方式（见 4.5 节），在这段代码中我利用了这一点，将序列表

示为支持以下三种操作的一对迭代器：

- *，用于访问当前值；
- ++，用于移动到下一个元素；
- != 比较迭代器，用来检查是否已到达序列末尾。

我们现在已经得到了一个算法（一个函数模板），它既可以用于数组也可以用于链表，数组或链表中保存的既可以是 int 也可以是 double。数组的例子可以立刻用此算法实现，因为 double* 就可作为迭代器使用：

```
double ad[] = {1,2,3,4};
double s = sum<double>(ad,ad+4);
```

为了让手工编写的单向链表类型也能用此算法处理，我们需要为其提供迭代器 Node*：

```
struct Node { Node* next; int data; };

Node* operator++(Node* p) { return p->next; }
int operator*(Node* p) { return p->data; }
Node* end(lst) { return nullptr; }

void test(Node* lst)
{
    int s = sum<int>(lst,end(lst));
}
```

我将 nullptr 作为尾迭代器。这里我使用了显式模板实参列表（<int>），允许调用者指定累加器变量的类型。

到目前为止我们得到的算法已经比现实世界中的很多代码都要更通用了。例如，sum() 可以用于浮点数链表（支持所有浮点数精度）、整数数组（支持所有整数宽度）以及其他很多类型，如 vector<char>。而且重要的是，sum() 与我们一开始手工编写的函数同样高效。我们不希望实现了通用性但牺牲了性能。

有经验的程序员会注意到还可以继续泛化 sum()。特别是，使用一个额外的模板参数显得有些笨拙，而且我们还需要将累加值初始化为 0。解决方法是让调用者提供初始值，然后从这个初始值推断 Val：

```
template<typename Iter, typename Val>
Val accumulate(Iter first, Iter last, Val s)
{
    while (first!=last) {
        s = s + *first;
        ++first;
    }
    return s;
}

double ad[] = {1,2,3,4};
double s1 = accumulate(ad,ad+4,0.0);    // 累加到一个 double 中
double s2 = accumulate(ad,ad+4,0);      // 累加到一个 int 中
```

但为什么要用 +？毕竟有时候我们希望进行累“乘”而不是累“加”。实际上，我们希望对序列中的元素进行的运算有很多种可能。这引出了进一步的泛化：

```
template<typename Iter, typename Val, typename Oper>
Val accumulate(Iter first, Iter last, Val s, Oper op)
```

```
{
    while (first!=last) {
        s = op(s,*first);
        ++first;
    }
    return s;
}
```

现在我们用参数 op 将元素值合并到累“加”器中。例如：

```
double ad[] = {1,2,3,4};
double s1 = accumulate(ad,ad+4,0.0,std::plus<double>);      // 加法
double s2 = accumulate(ad,ad+4,1.0,std::multiply<double>);
```

标准库对一些常见运算，如 plus 和 multiply，提供了对应的函数对象，可用作实参。这段代码体现了由调用者指定初始值的作用：固定的初始值 0 和 * 搭配进行累积运算是不正确的。标准库还为 accumulate() 提供了进一步泛化的版本，允许用户提供运算符来组合“求和”计算结果和累加器的值（见 40.6 节）。

提升技术要求程序员具备应用领域知识和一定的经验。设计算法最重要的指导原则是：在从具体实例提升算法的过程中，增加的特性（符号或运行时开销）不能损害算法的使用。标准库算法在设计过程中就非常注意性能问题。

24.3　概念

模板对实参的要求是什么？换句话说，模板代码对其实参类型有何假设？或者反过来，一个类型必须提供什么特性才能被接受为模板的实参？答案有无穷种可能，因为我们可以构建具有任意属性的类和模板，例如：

- 提供 – 但不提供 + 的类型；
- 能拷贝值但不能移动值的类型；
- 拷贝操作不进行拷贝的类型（见 17.5.1.3 节）；
- 用 == 比较是否相等的类型和其他用 compare() 比较相等性的类型；
- 将加法运算定义为成员函数 plus() 的类型和其他将加法运算定义为非成员函数 operator+() 的类型。

沿着这个方向继续下去会带来混乱。因为，如果每个类都有其独特的接口，那么编写能接受很多不同类型的模板就变得非常困难。反过来，如果每个模板对其实参的要求都是独特的，那么定义可用于很多模板的类型也同样变得非常困难。我们将不得不记住大量接口，并掌握它们的变化，这只在编写微型程序时才可能做到，在编写实际规模的库和程序时就完全不可控了。我们要做的是确定少量概念（要求的集合），使之能用于很多模板和很多实参类型。理想情况就如同我们在现实世界中所熟知的“插头兼容性”，通过设计少量标准插头使得插头和插座的兼容变得更容易。

24.3.1　发现概念

我以 23.2 节中的 String 类模板为例：

```
template<typename C>
class String {
    // ...
};
```

如果要将类型 X 作为 String 的实参，即 String<X>，那么 X 要满足什么要求呢？更一般的，在这样一个字符串类中，X 如果要承担字符类型的角色，它应具备什么性质呢？对此问题，一个有经验的设计者会有一些可能的答案，并从这些可能的答案开始着手设计。但是，让我们考虑如何从基本原理的层面回答这个问题。我们进行如下三阶段分析。

[1] 首先，我们考察模板的（最初）实现，确定它使用了参数类型的哪些属性（操作，函数，成员，等等），并确定这些操作的含义。得到的结果列表就是这个特定模板实现对其实参的最小要求。

[2] 接下来，我们考察其他合理的模板实现，列出它们对自己模板实参的要求。这样，我们就可以确定：为了适用于其他更多模板实现我们应该增加还是减少要求。或者，我们可以选择一个要求更少 / 更简单的实现。

[3] 最后，我们考察最终的结果列表（可能有多个），比较它与我们用过的其他模板的要求（概念）列表间的异同。我们尝试优选出一些简单的公共概念，使之能表示本来很长的要求列表。这么做的目的是让我们的设计能从这样的一般分类工作中受益。最终挑选出的概念应该容易被赋予有意义的名字，也容易记忆。它们还应该将概念的变化限制在必要的范围内，从而最大化模板和类型的互操作性。

前两个步骤与我们将具体算法泛化（"提升"）为通用算法的方法（见 24.2 节）非常相似，这是有其本质原因的。最后一步则是为了抵抗诱惑：试图为每个算法提供一组与其实现精确匹配的实参要求。因为这样的要求列表太特殊化了，而且不稳定：对实现的每个修改都意味着已成为算法接口一部分的要求也要相应改变。

对 String<C>，首先要考虑 String 实现（见 19.3 节）对参数 C 真正执行的操作。这就是此 String 实现的最小要求集合：

[1] 可以用拷贝赋值和拷贝初始化操作拷贝 C。

[2] String 可以用 == 和 != 比较 C。

[3] String 可以创建 C 的数组（意味着可以默认构造 C）。

[4] String 可以接受 C 的地址作为参数。

[5] 在销毁一个 String 时，它所保存的 C 也可以被销毁。

[6] String 的 >> 和 << 运算符可以用某种方式读写 C。

我们通常要求所有数据类型都要具有 [4] 和 [5] 两项技术特性，这里不会讨论不满足这两个要求的类型，这种类型几乎都是过分聪明的产物。少数重要的类型，如 std::unique_ptr，是不满足第一条要求"值可以拷贝"的，因为这些类型表示实际资源（见 5.2.1 节和 34.3.1 节）。但是，几乎所有"普通"类型都满足这条要求。执行拷贝操作能力的要求伴随着拷贝语义的要求——拷贝真正是原值的一个副本，即，两个拷贝在使用上完全一样（地址拷贝除外）。因此，拷贝能力通常伴随着另一个要求：提供具有常规语义的 ==（我们的 String 就是如此）。

我们要求模板实参具有赋值运算符，这意味着 const 类型不能作为模板实参。例如，String<const char> 就不能保证正常工作。与大多数类模板一样，对 String 而言这个问题还好。具备赋值运算符意味着算法可以使用此实参类型的临时变量，创建此实参类型对象的容器，等等。但并不意味着我们不能在接口说明中使用 const。例如：

```
template<typename T>
bool operator==(const String<T>& s1, const String<T>& s2)
```

```
{
    if (s1.size()!=s2.size()) return false;
    for (auto i = 0; i!=s1.size(); ++i)
        if (s1[i]!=s2[i]) return false;
    return true;
}
```

对 String<X>，我们要求类型 X 的对象可以拷贝。与此不冲突，operator==() 声明其实参类型是 const 的，从而承诺不会修改 String 中 X 类型的元素。

我们应该要求元素类型 C 具有移动操作吗？毕竟我们为 String<C> 设计了移动操作。答案是可以，但没有必要：我们要对 C 做的动作都可以通过拷贝操作很好地完成，如果某些拷贝被隐式转换为移动（例如，当我们返回一个 C 时），那当然就更好了。特别是，即使不要求移动操作，一些可能很重要的实例，例如 String<String<char>>，也会运转良好（正确且高效）。

到目前为止一切都好，但最后一条要求（我们可以用 >> 和 << 读写 C）似乎有些过分。我们真会读写每种字符串吗？也许改为“如果我们读写 String<X>，则 X 必须提供 >> 和 <<”更好？即，不是对所有 String 的 C 都设定这条要求，而是（只）对我们真正读写的 String（才）设定这条要求。

这是一条非常重要且基本的设计抉择：我们是为类模板实参设置要求（从而适用于所有类成员）还是仅为个别类函数成员设置实参要求。后者更灵活，但也更冗长（我们必须为每个有需求的函数描述要求），而且程序员也很难记忆。

考察到目前为止得到的要求列表，我注意到缺少了一些对“普通字符串”中的“普通字符”来说很常见的操作：

[1] 没有排序操作（如 <）；

[2] 没有转换为整数值的操作。

在这些初步分析后，我们可以思考我们的要求列表涉及哪些“众所周知的概念”（见 24.3.2 节）。对“普通类型”而言，核心概念是正规（regular），一个正规类型就是符合下列条件的类型：

- 你可以（通过赋值或初始化）用恰当的拷贝语义（见 17.5.1.3 节）拷贝它；
- 你可以默认构造它；
- 在一些小技术要求（如获取变量地址）上没有问题；
- 你可以（使用 == 和 !=）比较相等性。

对我们的 String 的模板实参而言，这些要求看起来是非常好的选择。我曾考虑去掉相等性比较操作，但发现没有相等性比较的话，拷贝操作基本是无用的。通常，选定 Regular 概念个安全赌注，而且思考 == 的含义可以帮助我们避免在定义拷贝操作时犯错。C++ 的所有内置类型都是正规的。

但对 String 而言，去掉排序操作（<）合理吗？请思考我们是如何使用字符串的。对一个模板（如 String）来说，使用它的需求决定了对其实参的要求。我们确实经常比较字符串，在进行字符串序列排序、字符串集合插入等操作时也会间接使用比较操作。而且，标准库 string 也确实提供了 <。在选定概念时，考察标准库以获得灵感通常是个好主意。因此，对 String 我们不仅要求 Regular，还要求排序操作，这就是概念 Ordered。

有趣的是，对于 Regular 是否应该包含 < 已有很多争论。看起来大多数数值类型都有

一个自然序。例如，字符被编码为位模式，可以解释为整数，而任何值序列都能按字典序排序。但是，也有很多类型的确没有自然序（如复数和图像），虽然我们可以为其定义一个。其他一些类型有多个自然序，但没有唯一的最佳选择（例如，记录可以按名字排序，也可以按地址排序）。最后，一些（合理的）类型根本就没有序。例如，考虑下面的枚举类型：

```
enum class rsp { rock, scissors, paper };
```

石头－剪子－布游戏完全依赖于下面的大小规则

- scissors<rock（剪刀比石头小）；
- rock<paper（石头比布小）；
- paper<scissors（布比剪刀小）。

但是，我们的 String 不应该接受任意类型作为其字符类型，而是应该接受支持字符串操作（如比较、排序和 I/O）的类型，因此我决定要求其实参类型支持排序操作。

如果要求 String 的模板实参增加默认构造函数及 == 和 < 运算符，我们就能为 String 提供几个有用的操作。实际上，我们对模板实参类型要求越多，模板实现者就越容易实现各种任务，模板也就能为其用户提供越多功能。但另一方面，避免让很少使用的要求和特定操作压垮模板也是很重要的：一条要求就意味着压在实参类型实现者身上的一份负担以及对可行实参类型的一层限制。因此，对 String<X> 我们要求：

- 满足 Ordered<X>；
- （仅）当我们使用 String<X> 的 >> 和 << 时，要求 X 具有 >> 和 <<；
- （仅）当我们定义并使用从 X 到整数的转换操作时，要求 X 具有转换为整数的能力。

目前，我们已经从语法特性角度表达了对 String 的字符类型的要求，诸如 X 必须提供拷贝操作、== 和 <。此外，我们还必须要求这些操作具有正确的语义：例如，拷贝操作确实进行复制、==（相等性）确实比较相等性以及 <（小于）确实提供排序。这种语义通常涉及操作之间的关系。例如，对于标准库，我们有（见 31.2.2.1 节）：

- 拷贝操作满足：任何与源对象相等的也都与副本相等（a==b 意味着 T(a)==T(b)），且副本独立于源对象（见 17.5.1.3 节）。
- 小于操作（如 <）提供了一个严格弱序（见 31.2.2.1 节）。

这些语义是用英语文本或（更好的是）数学描述定义的，但不幸的是我们无法用 C++ 本身表达语义要求（但请见 24.4.1 节）。对于标准库，你可以在 ISO 标准中找到用正规英语阐述的语义要求。

24.3.2　概念和约束

概念不是任意的属性集合。大多数类型（或一组类型）的属性列表并不能给出一个一致、有用的概念定义。要成为一个有用的概念，要求列表必须反映模板类的一组算法或一组操作的需求。在很多领域中，人们已经设计或发现了一些概念，能很好地描述领域的基本概念（C++ 所选择的术语“概念”就是源于人们头脑中的这种常见用法）。似乎有意义的概念出奇地少。例如，代数建立在单子、域和环这些概念之上；而 STL 则依赖于前向迭代器、双向迭代器和随机访问迭代器等概念。在某个领域中发现一个新概念当属重大成就，是可遇而不可求的。大多数情况下，你都是通过查阅某个研究或应用领域的基本文献来获得概念的。24.4.4 节会介绍本书所用的概念。

“概念”是一个非常一般性的思想，并非与模板存在固有联系。从某种意义上说，即使是 K&R C[Kernighan, 1978] 也有概念：带符号整型（signed integral type）可以认为是“内存中的整数”这一思想在 C 语言中的泛化。我们对模板实参的要求是概念（无论怎样表达），因此大多数与概念相关的有趣问题都产生于模板的语境中，仅此而已。

我将概念看作一个精心构造的实体，反映了一个应用领域的基本属性。因此，只应有少量概念，它们能起到指导算法和类型设计的作用。与之相似的是现实生活中的插头和插座，我们希望插头和插座的类型越少越好，这能让我们的使用更为简单，同时降低设计和制造成本。这种理想情况可能会与某些理念和观念冲突，包括：最小化每个泛型算法 / 参数化类的要求的理念（见 24.2 节）；为类提供绝对最小接口的理念（见 16.2.3 节）；以及一些程序员必须“严格按我意愿”编写代码的观念。但是，不付出一定努力，不制定某种形式的标准，就无法获得“插头兼容性”。

我为概念设定的标准非常高：我要求一个概念具有通用性、一定程度的稳定性、广泛的算法适用性、语义一致性以及其他很多性质。实际上，按照我的标准，很多常用的模板实参的简单约束都不够格称为概念。我认为这是不可避免的。特别是，我们编写过很多模板，它们并不能很好地反映通用算法或广泛应用的类型。相反，它们的重点是实现细节，它们的实参只反映了单一模板的必要细节，而这些模板只是为特定实现中的特定用途而设计的。我将这种模板实参的要求称为约束（constraint）或特殊概念（ad hoc concept，如果你必须这么叫的话）。一种看待约束的方式是将它们看作接口的不完全的（部分的）说明。通常，不完全的说明也是有用的，总比没有说明要好得多。

例如，我们考虑为平衡二叉树设计一个实验性的平衡策略库。树会接受一个平衡策略 Balancer 作为模板实参：

```
template<typename Node, typename Balance>
struct node_base {  // base of balanced tree
    // ...
};
```

一个平衡策略就是一个类，提供对树结点的三个操作。例如：

```
struct Red_black_balance {
    // ...
    template<typename Node> static void add_fixup(Node* x);
    template<typename Node> static void touch(Node* x);
    template<typename Node> static void detach(Node* x);
};
```

显然，我们希望说明对 node_base 的实参的要求，但平衡策略并非一个会广泛使用的接口，它也并不容易理解：它只是平衡树的一个特定实现的细节而已。平衡策略的设计思想（我犹豫是否使用术语“概念”）不太可能用在其他地方，甚至在对平衡树的实现进行重大修改时都可能会改变。而且，要确定一个平衡策略的确切语义是很困难的。首先，Balancer 的语义取决于 Node 的语义。从这个角度看，Balancer 不同于 Random_access_iterator 这种真正的概念。但是，我们仍然能将平衡策略的最小说明——“提供这三个对结点的操作”——用作 node_base 实参的一个约束。

注意“语义”在概念的讨论中频繁出现的方式。我发现当需要确定某些要求是一个概念还是仅仅是某个类型（或一组类型）上的一组特殊约束时，问题“我是否能写出半形式化的语义？”是最有用的判定标准。如果我能写出一个有意义的语义说明，这就是一个概念。否

则，得到的就只是一个约束，它可能很有用，但不要期望它是稳定的或是能广泛使用的。

24.4　具体化概念

不幸的是，C++ 没有提供专门的语言特性来直接表达概念。但是，将“概念”仅仅作为一个设计理念来处理，将它以注释的形式非正式地表达出来，也不是一种理想的方式。首先，编译器不会去理解注释，因此，只是表达为注释的要求必须由程序员来检查，而不能帮助编译器发现和报告错误。经验表明，即使语言不支持直接、完美地表达概念，我们也可以使用完成编译时模板实参属性检查的代码来近似表达概念。

一个概念就是一个谓词：即，我们将概念看作一个编译时函数，能检查一组模板实参，当实参满足概念要求时返回 true，否则返回 false。因此，我们可以将概念实现为一个 constexpr 函数。这里，我将使用术语约束检查（constraint check）表示对 constexpr 谓词的一次调用，它检查一组类型和值是否符合概念。与真正的概念相比，约束检查并不处理语义问题，它只是检查语法属性相关的假设。

考虑我们的 String：它的字符类型实参应该满足 Ordered：

```
template<typename C>
class String {
    static_assert(Ordered<C>(),"String's character type is not ordered");
    // ...
};
```

当用类型 X 实例化 String<X> 时，编译器将执行 static_assert。如果 Ordered<X> 返回 true，则编译继续，就像没有断言一样地生成代码。否则，会报告错误消息。

乍一看，这是一个很合理的解决方案。虽然我更喜欢下面这种形式：

```
template<Ordered C>
class String {
    // ...
};
```

但它还有待未来实现。因此我们现在还是考虑如何定义谓词 Ordered<T>()：

```
template<typename T>
constexpr bool Ordered()
{
    return Regular<T>() && Totally_ordered<T>();
}
```

即，如果一个类型 T 既满足 Regular 又满足 Totally_ordered，则它满足 Ordered。让我们继续“深挖”其中的含义：

```
template<typename T>
constexpr bool Totally_ordered()
{
    return Equality_comparable<T>()   // 有 == 和 !=
        && Has_less<T>()&& Boolean<Less_result<T>>()
        && Has_greater<T>() && Boolean<Greater_result<T>>()
        && Has_less_equal<T>() && Boolean<Less_equal_result<T>>()
        && Has_greater_equal<T>() && Boolean<Greater_equal_result<T>>();
}

template<typename T>
constexpr bool Equality_comparable()
```

```
{
    return Has_equal<T>() && Boolean<Equal_result<T>>()
        && Has_not_equal<T>() && Boolean<Not_equal_result<T>>();
}
```

也就是说，如果一个类型 T 是正规的且提供了六种常见的比较操作，则它是有序的。其中，比较操作得到的结果必须能转换为 bool 类型。而且每个比较运算符必须具有正确的数学含义。C++ 标准准确描述了这些含义（见 31.2.2.1 节和 iso.25.4 节）。

Has_equal 是用 enable_if 实现的，这种技术将在 28.4.4 节中介绍。

我将约束名字的首字母设定为大写（如 Regular），这么做违反了我自己的“独特风格”——类型名和模板名首字母大写，但函数名不这样做。但是，概念甚至比类型更为基础，因此我绝对有必要加以强调。我还将所有概念放在一个单独的名字空间中（Estd），期望它们（或是非常相似的名字）最终成为语言或标准库的一部分。

进一步探究有用的概念，我们可以定义 Regular：

```
template<typename T>
constexpr bool Regular()
{
    return Semiregular<T>() && Equality_comparable<T>();
}
```

Equality_comparable 确保类型具有 == 和 !=。而概念 Semiregular 则要求类型不能有不寻常的技术限制：

```
template<typename T>
constexpr bool Semiregular()
{
    return Destructible<T>()
        && Default_constructible<T>()
        && Move_constructible<T>()
        && Move_assignable<T>()
        && Copy_constructible<T>()
        && Copy_assignable<T>();
}
```

一个 Semiregular 类型同时具有移动和拷贝语义。大多数类型都是如此，不过也有某些类型不能拷贝，例如 unique_ptr。但我还不知道哪种有用的类型是可以拷贝但不能移动的。既不能移动也不能拷贝的类型非常罕见，如 type_info（见 22.5 节），这种类型通常用来表示系统属性。

我们也可以将约束检查用于函数中，例如：

```
template<typename C>
ostream& operator<<(ostream& out, String<C>& s)
{
    static_assert(Streamable<C>(),"String's character not streamable");
    out << "";
    for (int i=0; i!=s.size(); ++i) cout << s[i];
    out << "";
}
```

String 的输出运算符 << 需要用概念 Streamable 检查其实参 C 是否提供了输出运算符 <<：

```
template<typename T>
constexpr bool Streamable()
{
    return Input_streamable<T>() && Output_streamable<T>();
}
```

即，Streamable 检查对一个类型是否可以使用标准流 I/O（见 4.3 节和第 38 章）。

通过约束检查模板来实现概念检查有着明显的缺点：

- 约束检查被置于定义中，但实际上它们属于声明。即，概念是抽象接口的一部分，但约束检查只能用于抽象的实现中。
- 对约束的检查是实例化约束检查模板时发生的。因此，进行检查的时机可能比我们期望的时间晚。特别是，对于一个约束，我们更希望编译器保证在第一次调用时完成检查，但对当前的 C++ 来说这是不可能实现的。
- 我们可能忘记插入一个约束检查（特别是对函数模板）。
- 编译器不会检查一个模板实现是否只用到了其概念中说明的属性。因此，一个模板实现可能通过了约束检查，但仍然可能在类型检查时发现错误。
- 我们无法用编译器能够理解的方式（如使用注释）说明语义属性。

添加约束检查使得我们对模板实参的要求变为显式的说明，而且，一个设计良好的约束检查能令错误消息更容易理解。如果忘记插入约束检查，就会回到对实例化生成代码的普通类型检查。这可能有些遗憾，但并非灾难。如果使用约束检查技术，对基于概念的设计的检查就能更为健壮，而不再仅仅是类型系统的一部分。

如果需要，我们可以将约束检查放置在几乎任何地方。例如，为了检查一个特定类型是否满足一个特定概念的要求，我们可以将类型检查放在一个名字空间作用域中（如全局作用域）。例如：

```
static_assert(Ordered<std::string>,"std::string is not Ordered");        // 将会成功
static_assert(Ordered<String<char>>,"String<char> is not Ordered");      // 将会失败
```

第一个 static_assert 检查标准库 string 是否满足 Ordered（答案是肯定的，因为它提供了 ==、!= 和 <）。第二个 static_assert 检查我们设计的 String 是否满足 Ordered（答案是否定的，因为我们“忘记了”定义 <）。使用这样一个全局检查，能令约束检查的执行不再依赖于我们在程序中是否真正使用了这个特定的模板特例化版本。这可能是一个优点，但也可能是一个困扰，究竟如何取决于我们的目的。这种方式强制在程序中的特定点执行类型检查，通常这对错误隔离是有好处的。而且，这种方式也能帮助单元测试。但对于使用了很多库的程序来说，显式检查很快就会变得不可控。

对一个类型而言，满足 Regular 是很理想的情况。我们可以拷贝正规类型的对象，将它们存放于 vector 和数组中，比较它们，等等。如果一个类型满足 Ordered，我们还可以使用其对象的集合，排序其对象的序列，等等。因此，我们回过头来改进 String，使它满足 Ordered。特别是，我们为其增加 < 来提供字典序比较：

```
template<typename C>
bool operator<(const String<C>& s1, const String<C>& s2)
{
    static_assert(Ordered<C>(),"String's character type not ordered");
    bool eq = true;
    for (int i=0; i!=s1.size() && i!=s2.size(); ++i) {
        if (s2[i]<s1[i]) return false;
        if (s1[i]<s2[i]) eq = false;                    // 并非 s1==s2
    }
    if (s2.size()<s1.size()) return false;              // s2 并不比 s1 短
    if (s1.size()==s2.size() && eq) return false;       // s1==s2
    return true;
}
```

24.4.1　公理

与数学中一样，公理（axiom）就是我们认为正确但又无法证明的东西。在讨论模板实参要求时，我们用“公理”表示语义属性。我们使用公理描述一个类或算法对其输入集合有何假设。一个公理无论是如何表达的，都表示一个算法或类对其实参的期望（假设）。我们无法通过一般测试检查一个类型的值是否满足某个公理（这也是我们称之为公理的原因）。而且，只有算法真正使用的值才被要求满足公理。例如，一个算法可以小心地避免解引用空指针或拷贝非法浮点值 NaN。如果是这样，它就满足公理“指针必须可解引用且浮点值必须可拷贝”。还可以从相反的角度陈述公理——奇异值（如 NaN 和 nullptr）违反了某个前提条件，因此不予考虑。

C++（当前）还无法表达公理，但并非我们就只能用注释或设计文档来描述公理了，类似于概念，我们可以把公理表达得更具体一些。

考虑对一个正规类型，我们如何表达对它的一些关键语义要求：

```
template<typename T>
bool Copy_equality(T x)                          // 拷贝构造语义
{
    return T{x}==x;      // 副本应与源对象相等
}

template<typename T>
bool Copy_assign_equality(T x, T& y)             // 赋值语义
{
    return (y=x, y==x);  // 赋值结果应与源对象相等
}
```

换句话说，拷贝操作确实创建了副本：

```
template<typename T>
bool Move_effect(T x, T& y)                      // 移动语义
{
    return (x==y ? T{std::move(x)}==y) : true) && can_destroy(y);
}

template<typename T>
bool Move_assign_effect(T x, T& y, T& z)         // 移动赋值语义
{
    return (y==z ? (x=std::move(y), x==z)) : true) && can_destroy(y);
}
```

换句话说，移动操作生成的值与移动源应该相等，且移动源可以被销毁。

这些公理都可以用可执行代码来表达。我们可以用它们来进行检测，但最重要的是，比起简单地写一条注释，我们需要更加深入地思考如何表达它们。比起“普通英语”的表达方式，这种表达公理的方式更为准确。基本上，我们已经可用一阶谓词逻辑来表达这种伪公理了。

24.4.2　多实参概念

当我们考察一个单实参概念并将其用于一个类型时，看起来非常像在做传统的类型检查，概念就像是类型的类型。事实确实部分如此，但仅仅是部分而已。情况还可能更复杂，我们通常会发现实参类型间的关系对正确的说明和使用非常重要。考虑标准库 find() 算法：

```
template<typename Iter, typename Val>
Iter find(Iter b, Iter e, Val x);
```

Iter 模板实参必须是一个输入迭代器，而且为此概念定义一个约束检查模板（相对）容易。

到目前为止一切都好，但 find() 完全依赖于 x 与序列 [b:e) 中的元素进行的比较。我们需要说明类型应具有比较操作；即，需要指出 Val 和输入迭代器的值类型可以比较相等性。这需要一个双实参版本的 Equality_comparable：

```
template<typename A, typename B>
constexpr bool Equality_comparable(A a, B b)
{
    return Common<T, U>()
        && Totally_ordered<T>()
        && Totally_ordered<U>()
        && Totally_ordered<Common_type<T,U>>()
        && Has_less<T,U>() && Boolean<Less_result<T,U>>()
        && Has_less<U,T>() && Boolean<Less_result<U,T>>()
        && Has_greater<T,U>() && Boolean<Greater_result<T,U>>()
        && Has_greater<U,T>() && Boolean<Greater_result<U,T>>()
        && Has_less_equal<T,U>() && Boolean<Less_equal_result<T,U>>()
        && Has_less_equal<U,T>() && Boolean<Less_equal_result<U,T>>()
        && Has_greater_equal<T,U>() && Boolean<Greater_equal_result<T,U>>()
        && Has_greater_equal<U,T>() && Boolean<Greater_equal_result<U,T>>();
};
```

对单一概念而言，这个谓词有点儿过于冗长了。但是，我宁愿如此冗长地显式说明对所有运算符及其组合使用的要求，也不愿将此复杂性深埋于泛化之中。

有了这个谓词，我们就可以定义 find() 了：

```
template<typename Iter, typename Val>
Iter find(Iter b, Iter e, Val x)
{
    static_assert(Input_iterator<Iter>(),"find() requires an input iterator");
    static_assert(Equality_comparable<Value_type<Iter>,Val>(),
                  "find()'s iterator and value arguments must match");

    while (b!=e) {
        if (*b==x) return b;
        ++b;
    }
    return b;
}
```

在指定泛型算法时，多参数概念特别常见也特别有用。这也是你发现最多概念以及发现最多新概念需求（相对于从一个常见概念目录中挑选一个“标准概念”）的地方。定义良好的类型的变化要比算法对其实参要求的变化少得多。

24.4.3 值概念

概念可以表达对一组模板实参的任何（语法）要求。特别是，模板实参可以是一个整型值，因此概念也可以约束整数实参。例如，我们可以编写一个约束检查来检测一个值模板实参是否是小值：

```
template<int N>
constexpr bool Small_size()
```

```
{
    return N<=8;
}
```

一个更实际的例子是一个概念约束多个实参，其中包括数值实参。例如：

```
constexpr int stack_limit = 2048;

template<typename T,int N>
constexpr bool Stackable()    // T 是正则的且 N 个类型为 T 的元素可以存入一个小的栈中
{
    return Regular<T>() && sizeof(T)*N<=stack_limit;
}
```

这实现了一个“小到足以容纳于栈中”的概念。它可以这样使用：

```
template<typename T, int N>
struct Buffer {
    // ...
};

template<typename T, int N>
void fct()
{
    static_assert(Stackable<T,N>(),"fct() buffer won't fit on stack");
    Buffer<T,N> buf;
    // ...
}
```

与类型的基础概念相比，值的概念通常更小、更专用。

24.4.4　约束检查

本书所用的约束检查都可以在配套网站中找到。它们并不是标准的一部分，而且我希望将来能用适当的语言机制代替它们。但是，对思考模板和类型的设计而言，它们是很有用的，而且反映了标准库中事实上的概念。它们应该放在一个单独的名字空间中，避免与未来可能的语言特性或是概念的其他实现相互干扰。我将它们置于名字空间 Estd 中，但它可能是一个别名（见 14.4.2 节）。下列一些约束检查对读者来说可能很有用。

- Input_iterator<X>：X 是一个迭代器，我们可以用它一次性地遍历一个序列（使用 ++ 向前），每个元素只读取一次。
- Output_iterator<X>：X 是一个迭代器，我们可以用它一次性地遍历一个序列（使用 ++ 向前），每个元素只写入一次。
- Forward_iterator<X>：X 是一个迭代器，我们可以用它遍历一个序列（使用 ++ 向前）。单向链表（如 forward_list）都支持这种迭代器。
- Bidirectional_iterator<X>：X 是一个迭代器，我们可以用它在序列中向前（使用 ++）和向后（使用 --）移动。双向链表（如 forward_list）都支持这种迭代器。
- Random_access_iterator<X>：X 是一个迭代器，我们可以用它遍历一个序列（向前和向后）、使用下标操作随机访问元素以及用 += 和 -= 进行定位。数组支持这种迭代器。
- Equality_comparable<X,Y>：可以用 == 和 != 比较一个 X 和一个 Y。
- Totally_ordered<X,Y>：X 和 Y 满足 Equality_comparable，且可以用 <、<=、> 和 >= 比较一个 X 和一个 Y。

- Semiregular<X>：X 可以被拷贝、默认构造、在自由空间上分配而且没有恼人的小技术限制。
- Regular<X>：X 是 Semiregular 的，且可以比较相等性。标准库容器要求其元素是正规的。
- Ordered<X>：X 满足 Regular 和 Totally_ordered。标准库关联容器要求其元素是有序的，除非你显式提供了一个比较操作。
- Assignable<X,Y>：可以用 = 将一个 Y 赋予一个 X。
- Predicate<F,X>：可以用 X 调用 F，返回一个 bool 值。
- Streamable<X>：可以用 I/O 流读写 X。
- Movable<X>：X 可以移动；即，它具有一个移动构造函数和一个移动赋值函数。此外，X 还可以寻址及析构。
- Copyable<X>：X 满足 Movable 且可以拷贝。
- Convertible<X,Y>：一个 X 可以隐式转换为一个 Y。
- Common<X,Y>：一个 X 和一个 Y 可以无二义地转换为一个名为 Common_type<X,Y> 的公共类型。这是运算符 ?:(见 11.1.3 节）的运算对象兼容性规则的形式化描述。例如，Common_type<Base*,Derived*> 为 Base*，Comon_type<int,long> 为 long。
- Range<X>：X 可用于范围 for 语句（见 9.5 节），即，X 必须提供具有所需语义的成员 x.begin() 和 x.end()，或等价的非成员函数 begin(x) 和 end(x)。

显然，这些定义是非正式的。大多数情况下，这些概念基于标准库类型谓词（见 35.4.1 节），ISO C++ 标准库提供了正式定义（如 iso.17.6.3）。

24.4.5 模板定义检查

一个约束检查模板确保一个类型提供概念所要求的属性。如果模板实现实际上使用了比概念所保证的更多的属性，我们就可能得到类型错误。例如，标准库 find() 要求一对输入迭代器实参，但我们可能（不小心）像下面这样定义了它：

```
template<typename Iter, typename Val>
Iter find(Iter b, Iter e, Val x)
{
    static_assert(Input_iterator<Iter>(),"find(): Iter is not a Forward iterator");
    static_assert(Equality_comparable<Value_type<Iter>,Val>(),
                  "find(): value type doesn't match iterator");

    while (b!=e) {
        if (*b==x) return b;
        b = b+1;                    // 注意：不是 ++b
    }
    return b;
}
```

现在，b+1 是一个错误，除非 b 是一个随机访问迭代器（而不仅是约束检查所保证的前向迭代器）。但是，约束检查无法帮助我们检查出这个问题。例如：

```
void f(list<int>& lst, vector<string>& vs)
{
    auto p = find(lst.begin(),lst.end(),1209);          // 错误：list 并未提供 +
    auto q = find(vs.begin(),vs.end(),"Cambridge");     // 正确：vector 提供了 +
    // ...
}
```

第一个对 list 的 find() 调用会失败（因为 list 提供的前向迭代器并未定义 +），而对 vector 的调用则会成功（因为对 vector<string>::iterator 来说 b+1 没有问题）。

约束检查主要为模板用户提供这样一个服务：检查实际模板实参是否满足模板的要求。另一方面，约束检查无法帮助模板设计者确保模板实现未使用任何概念说明之外的属性。理想情况下，类型系统将保证这一点，但所需语言特性还有待未来实现。因此，如何检查一个参数化类或一个泛型算法的实现呢？

概念提供了一个很强的指导：模板实现不应使用概念未指定的实参性质，因此我们测试实现时使用的实参应该提供了概念所指定的属性，且只使用这类实参。这种类型有时被称为原型（archetype）。

因此，对于 find() 的例子，我们考察 Forward_iterator 和 Equality_comparable，或标准库定义的前向迭代器和相等性比较概念（见 iso.17.6.3.1 和 iso.24.2.5）。随后，我们确认需要一个 Iterator 类型，至少提供下列性质：

- 一个默认构造函数；
- 一个拷贝构造函数和一个拷贝赋值运算符；
- 运算符 == 和 !=；
- 一个前缀运算符 ++；
- 一个类型 Value_type<Iterator>；
- 一个前缀运算符 *；
- 将 * 的结果赋予一个 Value_type<Iterator> 的能力；
- 将一个 Value_type<Iterator> 赋予 * 的结果的能力。

这是标准库前向迭代器的一个稍简化的版本，但对于 find() 而言已经足够了。通过观察概念构造这样一个列表是很容易的。

有了这个列表，我们需要找到或定义一个只提供所需属性的类型。对于 find() 所需的前向迭代器来说，标准库 forward_list 恰好完美满足要求。这是因为“前向迭代器”正是为了表达遍历单向链表的能力。而实际上一个常用类型恰好是一个常用概念的原型的情况很少见。如果我们决定使用一个已有类型，就要小心避免选择一个比要求更灵活的类型。例如，测试算法（如 find()）常犯的一个错误是使用 vector。vector 具有非常高的通用性和灵活性，这使它应用广泛，但也令它不适合作为很多简单算法的原型。

如果找不到一个适合需求的现有类型，就必须自己定义一个。具体方法是查看要求列表，定义适合的成员：

```
template<typename Val>
struct Forward {          // 用来检查 find()
    Forward();
    Forward(const Forward&);
    Forward operator=(const Forward&);
    bool operator==(const Forward&);
    bool operator!=(const Forward&);
    void operator++();
    Val& operator*();     // 简化：不能处理 Val 的代理
};

template<typename Val>
using Value_type<Forward<Val>> = Val;      // 简化：见 28.2.4 节

void f()
```

```
{
    Forward<int> p = find(Forward<int>{},Forward<int>{},7);
}
```

在这种级别的测试中，我们不需要检查这些操作是否真正实现了正确的语义，只需检查模板实现不依赖于它不该依赖的属性就可以了。

在本例中，我已经简化了测试，没有引入实参 Val 的原型，而是简单地使用了 int。测试从 Val 的原型到 Iter 的原型的类型转换是更重要的工作，而且可能更通用。

编写一个测试工具检查 find() 的实现是否符合对 std::forward_list 或 X 的要求并不简单，但也并非泛型算法设计者会遇到的最困难的任务。使用相对较小且精心说明的概念集合会使测试工具的设计更可控。测试可以也应该在编译时完成。

请注意，这个简单的说明和检查策略导致 find() 要求其迭代器实参具有一个 Value_type 类型函数（见 28.2 节）。这令指针可用作迭代器。对很多模板参数而言，内置类型可与用户自定义类型一样使用（见 1.2.2 节和 25.2.1 节）是很重要的。

24.5　建议

［1］ 模板可传递实参类型而不丢失信息；24.1 节。
［2］ 模板提供了一种编译时编程的通用机制；24.1 节。
［3］ 模板提供了编译时“鸭子类型”；24.1 节。
［4］ 通过“提升”具体实例来设计泛型算法；24.2 节。
［5］ 用概念说明模板实参要求来泛化算法；24.3 节。
［6］ 不要赋予常规符号非常规含义；24.3 节。
［7］ 将概念用作设计工具；24.3 节。
［8］ 使用常用且规范的模板实参要求来追求算法和实参类型间的“插头兼容性”目标；24.3 节。
［9］ 发现概念的方法：最小化一个算法对其模板实参的要求，然后推广至更广用途；24.3.1 节。
［10］ 一个概念不仅是一个特定算法实现需求的描述；24.3.1 节。
［11］ 如可能，尽量从众所周知的概念列表中选择概念；24.3.1 节和 24.4.4 节。
［12］ 模板实参的默认概念是 Regular；24.3.1 节。
［13］ 并非所有模板实参类型都满足 Regular；24.3.1 节。
［14］ 一个概念不仅是一些语法上的要求，还有语义方面的要求；24.3.1 节，24.3.2 节和 24.4.1 节。
［15］ 用代码具体化概念；24.4 节。
［16］ 将概念表达为编译时谓词（constexpr 函数）并用 static_asser t() 或 enable_if<> 测试它们；24.4 节。
［17］ 将公理用作设计工具；24.4.1 节。
［18］ 将公理作为测试的指导；24.4.1 节。
［19］ 某些概念涉及两个或更多模板实参；24.4.2 节。
［20］ 概念不仅是类型的类型；24.4.2 节。
［21］ 概念可能涉及数值实参；24.4.3 节。
［22］ 将概念作为测试模板定义的指导；24.4.5 节。

特 例 化

让你陷入麻烦的并非你所不知的，
而是你所确信的并非如你所知。

——马克·吐温

- 引言
- 模板参数和实参
  类型作为实参；值作为实参；操作作为实参；模板作为实参；默认模板实参
- 特例化
  接口特例化；主模板；特例化顺序；函数模板特例化
- 建议

25.1 引言

在过去 20 多年，模板已经从一个相对简单的思想发展为大多数高级 C++ 程序设计的基础。特别是，模板是下列技术的关键：

- 提高类型安全（例如，通过杜绝使用类型转换；见 12.5 节）；
- 提升程序抽象水平（例如，通过使用标准容器和算法；见 4.4 节、4.5 节、7.4.3 节、第 31 章和第 32 章）；
- 提供更灵活、类型安全更佳且更高效的类型和算法参数化（见 25.2.3 节）。

这些技术都严重依赖于模板代码对模板实参的使用是否无额外开销且类型安全。大多数技术还依赖于模板提供的类型推断机制（有时称为编译时多态；见 27.2 节）。对于那些强调性能的应用领域，如高性能数值计算和嵌入式系统程序设计，这些技术是 C++ 能否成功使用的基础。成熟的例子请参考标准库（见第五部分）。

本章和接下来的两章将通过简单的例子介绍一些高级或特殊的语言特性，它们都以毫不妥协的灵活性和性能为目标。其中很多技术都是为了标准库的实现而开发的，主要用途也在于此。与大多数程序员一样，我绝大多数时间都宁愿忘记更高级的技术。只要可能，我会尽量保持代码简单，并使用库编写程序，以便从特定应用领域专家所掌握的高级特性中受益。

我在 3.4 节中已经介绍过模板了。本章是模板及其使用系列介绍中的一部分：

- 第 23 章更为详细地介绍模板。
- 第 24 章讨论泛型程序设计——模板最常见的用途。
- 第 25 章展示如何用一组实参特例化模板。
- 第 26 章关注与名字绑定相关的模板实现问题。
- 第 27 章讨论模板和类层次之间的关系。
- 第 28 章关注模板作为一种生成类和函数的语言。
- 第 29 章给出一个大型程序范例，展示了如何使用基于模板的程序设计技术。

25.2 模板参数和实参

模板可以接受参数：

- “类型类型”的类型参数；
- 内置类型值参数，如 int（见 25.2.2 节）和函数指针（见 25.2.3 节）；
- “模板类型”的模板参数（见 25.2.4 节）。

到目前为止最常用的是类型参数，但值参数也是很多重要技术的基础（见 25.2.2 节和 28.3 节）。

一个模板可以接受固定数量或可变数量的参数，对可变参数模板的讨论将推迟到 28.6 节。

注意，一种很常见的模板类型实参命名方法是采用首字母大写的短名字，如 T、C、Cont 和 Ptr。这是可以接受的，因为这种名字按惯例常用于相对较小的作用域（见 6.3.3 节）。但是，当使用 ALL_CAPS 这种名字时，就很可能同宏名冲突（见 12.6 节）。因此不要使用太长的名字，以免与宏名冲突。

25.2.1 类型作为实参

通过使用 typename 或 class 前缀，我们可以将一个模板实参定义为类型参数（type parameter）。两种前缀的效果是相同的。从语法角度来说，一个模板可接受任何（内置或用户自定义）类型作为其类型参数。例如：

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

template<typename T>
class X {
    // ...
};
f(1);                       // T 被推断为 int
f<double>(1);               // T 为 double
f<complex<double>>(1);      // T 为 complex<double>

X<double> x1;               // T 为 double
X<complex<double>> x2;      // T 为 complex<double>
```

类型实参是无约束的；即，接口中没有任何信息约束实参必须是一种特定类型或是一个类层次的一部分。一个实参类型是否有效完全依赖于模板是如何使用它的，这提供了鸭子类型的一种形式（见 24.1 节）。你可以将一般性的约束实现为概念（见 24.3 节）。

当用作模板实参时，用户自定义类型和内置类型是完全相同的。这一点很重要，它允许我们定义用于内置类型和用户自定义类型时完全一致的模板。例如：

```
vector<double> x1;              // double 的 vector
vector<complex<double>> x2;     // complex<double> 的 vector
```

特别是，无论使用内置类型还是用户自定义类型，都不会有额外的时空开销。

- 内置类型的值并未打上特殊容器对象的标签。
- 所有类型的值都可以从 vector 中直接获取，无须使用可能代价高昂的“get() 函数”（如虚函数）。
- 用户自定义类型的值并非隐含地通过引用访问。

一个类型必须在作用域内且可访问，才能作为模板实参。例如：

```
class X {
    class M { /* ... */ };
    // ...
    void mf();
};

void f()
{
    struct S { /* ... */ };
    vector<S> vs;           // 正确
    vector<X::M> vm;        // 错误：X::M 是私有的
    // ...
}

void M::mf()
{
    vector<S> vs;       // 错误：作用域中没有类型 S
    vector<M> vm;       // 正确
    // ...
};
```

25.2.2 值作为实参

非类型或模板的模板参数称为值参数（value parameter），传递给它的实参称为值实参（value argument）。例如，可以用整数实参提供大小和限制：

```
template<typename T, int max>
class Buffer {
    T v[max];
public:
    Buffer() { }
    // ...
};

Buffer<char,128> cbuf;
Buffer<int,5000> ibuf;
Buffer<Record,8> rbuf;
```

当运行时间和空间处于最重要的地位时，Buffer 这种简单受限的容器就显得很重要了。这种容器避免了更通用的 string 或 vector 在自由存储空间使用上的一些复杂问题，也不像内置数组那样存在转换为指针的问题（见 7.4 节）。标准库 array（见 34.2.1 节）的实现实际上就采用了这种思想。

传递给模板值参数的实参可以是（见 iso.14.3.2）：

- 整型常量表达式（见 10.4 节）；
- 外部链接的对象或函数的指针或引用（见 15.2 节）；
- 指向非重载成员的指针（见 20.6 节）；
- 空指针（见 7.2.2 节）。

一个指针必须具有 &of 或是 f 的形式，才能作为模板实参，其中 of 是对象或函数的名字，f 是函数名。指向成员的指针必须具有 &X::of 的形式，其中 of 是成员的名字。注意，字符串字面值常量不能作为模板实参：

```
template<typename T, char* label>
class X {
```

```
    // ...
};

X<int,"BMW323Ci"> x1;          // 错误：字符串字面值常量作为实参
char lx2[] = "BMW323Ci";
X<int,lx2> x2;                 // 正确：lx2 有外部链接
```

这一限制与不允许浮点数值实参一样，是为了简化分离编译的实现。不必将模板值实参想得太复杂，或是试图寻找“更聪明”的方法，理解它的最好方式是将其看作向函数传递整数和指针的一种机制。不幸的是（没什么重要理由），字面值常量（见 10.4.3 节）不能用作模板值实参。值模板实参机制是某些高级编译时计算技术（见第 28 章）的基础。

整数模板实参必须是一个常量。例如：

```
constexpr int max = 200;

void f(int i)
{
    Buffer<int,i> bx;          // 错误：需要常量表达式
    Buffer<int,max> bm;        // 正确：是常量表达式
    // ...
}
```

反过来，值模板参数在模板内部是一个常量，因而试图修改其值是错误的。例如：

```
template<typename T, int max>
class Buffer {
    T v[max];
public:
    Buffer(int i) { max = i; }  // 错误：试图为模板值参数赋值
    // ...
};
```

在模板参数列表中，一个类型模板参数出现后即可当作一个类型来使用。例如：

```
template<typename T, T default_value>
class Vec {
    // ...
};

Vec<int,42> c1;
Vec<string,""> c2;
```

这一特性与默认模板实参结合使用时特别有用（见 25.2.5 节），例如：

```
template<typename T, T default_value = T{}>
class Vec {
    // ...
};

Vec<int,42> c1;
Vec<int> c11;                  // 默认值是 int{}，即 0
Vec<string,"fortytwo"> c2;
Vec<string> c22;               // 默认值为 string{}，即 ""
```

25.2.3 操作作为实参

考虑标准库 map（见 31.4.3 节）的一个略微简化的版本：

```
template<typename Key, Class V>
```

```
class map {
    // ...
};
```

我们如何为 Key 设定比较准则？

- 我们不能在容器中硬编码比较准则，因为容器（通常）不能将自己的需求强加给元素类型。例如，map 默认使用 < 进行元素比较，但并不是所有 Key 都有一个我们想要使用的 <。
- 我们不能将排序准则硬编码进 Key 类型，因为依赖于某个关键字的元素（通常）有多种不同的排序方式。例如，一种最常见的 Key 类型是 string，而 string 可以用多种不同的准则进行排序（如大小写敏感和大小写不敏感）。

因此，不应将排序准则作为容器类型或元素类型的一部分。原则上，对 map 而言，排序准则概念可以表示为：

[1] 一个模板值实参（例如，一个指向比较函数的指针）

[2] map 模板的一个类型实参，确定一个比较对象的类型。

初看上去第一种方案（传递一个特定类型的比较对象）显得更简单。例如：

```
template<typename Key, typename V, bool(*cmp)(const Key&, const Key&)>
class map {
public:
    map();
    // ...
};
```

此 map 要求用户提供一个比较函数：

```
bool insensitive(const string& x, const string& y)
{
    // 大小写不敏感比较（如，"hello" 等于 "HellO"）
}

map<string,int,insensitive> m;          // 用 insensitive() 进行比较
```

但是，这种方式不是很灵活。特别是，map 的设计者必须决定是用一个函数指针比较（未知的）Key 类型，还是使用某种特定类型的函数对象进行比较。而且，由于比较对象的实参类型必须依赖 Key 类型，它很难提供默认的比较准则。

因此，第二种方案（将比较类型作为一个模板类型参数传递）更常用，也是标准库中所采用的方式。例如：

```
template<typename Key, Class V, typename Compare = std::less<Key>>
class map {
public:
    map() { /* ... */ }                     // 使用默认比较
    map(Compare c) :cmp{c} { /* ... */ }    // 覆盖默认构造函数
    // ...
    Compare cmp {};                         // 默认比较
};
```

最常见的情况是使用默认的小于操作进行比较。如果你希望使用不同的比较准则，可以提供一个函数对象（见 3.4.3 节）：

```
map<string,int> m1;                         // 使用默认比较操作（less<string>）

map<string,int,std::greater<string>> m2;    // 用 greater<string>() 进行比较
```

函数对象可携带状态。例如：

```
Complex_compare f3 {"French",3};                    // 创建一个比较对象（见 25.2.5 节）
map<string,int,Complex_compare> m3 {f3};            // 用 f3() 进行比较
```

我们也可以使用函数指针，包括可以转换为函数指针的 lambda 表达式（见 11.4.5 节）。例如：

```
using Cmp = bool(*)(const string&,const string&);
map<string,int,Cmp> m4 {insensitive};               // 用一个函数指针进行比较

map<string,int,Cmp> m4 {[](const string& a, const string b) { return a>b; } };
```

与传递函数指针相比，传递函数对象的方式有着明显的优点：

- 类内定义的简单类成员函数适合内联，而编译器必须特别关注才能内联通过函数指针完成的调用。
- 传递无数据成员的函数对象没有运行时开销。
- 很多操作都能以单一对象的形式传递且无额外运行时开销。

map 比较准则的传递只是一个例子而已。不过，其中用到的技术是通用的，被广泛用于类和函数的“策略”参数化。经典的例子包括算法的动作（见 4.5.4 节和 32.4 节）、容器的分配器（见 31.4 节和 34.4 节）和 unique_ptr 的删除器（见 34.3.1 节）。当我们需要为一个函数模板（如 sort）指定实参时，也有同样的两种选择，标准库对这种情况仍然选择了方案［2］（例如，见 32.4 节）。

如果在我们的程序中使用比较准则的地方只有一处，则使用 lambda 表达式更简洁地表达函数对象是有意义的：

```
map<string,int,Cmp> c3 {[](const string& x, const string& y) const { return x<y; }}; // 错误
```

但不幸的是，这是错误的，因为 lambda 不能转换为函数对象类型。我们可以命名 lambda，然后使用其名字：

```
auto cmp = [](const string& x, const string& y) const { return x<y; }
map<string,int,decltype(cmp)> c4 {cmp};
```

我发现为操作命名从设计和维护的角度是很有用的。而且，任何非局部命名和声明的实体都可能派上其他用场。

25.2.4　模板作为实参

有时传递模板（而不是类或值）作为模板实参很有用。例如：

```
template<typename T, template<typename> class C>
class Xrefd {
    C<T> mems;
    C<T*> refs;
    // ...
};

template<typename T>
    using My_vec = vector<T>;          // 使用默认分配器

Xrefd<Entry,My_vec> x1;      // 在 vector 中存储 Entry 的交叉引用

template<typename T>
class My_container {
```

```
    // ...
};

Xrefd<Record,My_container> x2;   // 在 My_container 中存储 Records 的交叉引用
```

为了将一个模板声明为模板参数，我们必须指定其所需的实参。例如，我们指定 Xrefd 的模板参数 C 是一个模板类，它接受单一类型实参。如果不这样做，我们就不能使用 C 的特例化版本了。我们用一个模板作为另一个模板的参数，通常是因为我们希望用多种实参类型（如上例中的 T 和 T*）对其进行实例化。即，我们希望用这个模板来声明另一个模板的成员，且希望这个模板是一个参数，从而可以让用户指定不同类型。

只有类模板可以作为模板实参。

对于模板需要一两个容器这种常见的情形，其实没有必要使用模板作为实参，传递容器类型是更好的方式（见 31.5.1 节）。例如：

```
template<typename C, typename C2>
class Xrefd2 {
    C mems;
    C2 refs;
    // ...
};

Xrefd2<vector<Entry>,set<Entry*>> x;
```

在本例中，C 和 C2 的值类型可以通过一个简单的类型函数（见 28.2 节）获得，例如，Value_type<C>，它可以获取一个容器的元素类型。这也是标准库容器适配器如 queue 所采用的技术（见 31.5.2 节）。

25.2.5 默认模板实参

每次使用 map 都要显式指定比较准则是很烦人的。特别是，less<Key> 通常就是最好的选择，还要反复说明就显得更加烦人。我们可以将 less<Key> 指定为模板实参 Compare 的默认类型，这样我们就只需指定那些特殊的比较准则了：

```
template<typename Key, Class V, typename Compare = std::less<Key>>
class map {
public:
    explicit map(const Compare& comp ={});
    // ...
};

map<string,int> m1;                 // 将使用 less<string> 进行比较
map<string,int,less<string>> m2;    // 与 m1 是相同类型

struct No_case {
    // 定义 operator()() 进行大小写敏感的字符串比较
};

map<string,int,No_case> m3;         // m3 是与 m1 和 m2 不同的类型
```

注意 map 的默认构造函数是如何创建一个默认比较对象 Compare{} 的，这是最常见的情况。如果我们希望更精细地控制对象构造，就必须显式指定。例如：

```
map<string,int,Complex_compare> m {Complex_compare{"French",3}};
```

对一个模板参数的默认实参，只有当我们真正使用它时编译器才会对其进行语义检查。特别是，只要我们不使用默认模板实参 less<Key>，即使 compare() 类型 X 的值，less<X> 也不会被编译。这一点对标准库容器的（如 std::map）设计非常重要，因为这些容器都依赖模板实参来指定元素默认值（见 31.4 节）。

类似于默认函数实参（见 12.2.5 节），我们也只能对尾部模板参数指定和提供默认实参：

```
void f1(int x = 0, int y);          // 错误：默认实参不在尾部
void f2(int x = 0, int y = 1);      // 正确

f2(,2);     // 语法错误
f2(2);      // 调用 f2(2,1);

template<typename T1 = int, typename T2>
class X1 {              // 错误：默认实参不在尾部
    // ...
};

template<typename T1 = int, typename T2 = double>
class X2 {      // 正确
    // ...
};

X2<,float> v1; // 语法错误
X2<float> v2;  // v2 为 X2<float,double>
```

C++ 不允许用“空”实参表示“使用默认实参”，这是经过深思熟虑的，也是在灵活性和潜在的模糊错误之间做出的权衡。

通过模板实参提供策略，进而通过默认实参提供最常用的策略，几乎是标准库中的通用技术（如 32.4 节）。但说来奇怪，basic_string（见 23.2 节和第 36 章）的比较并未使用这种技术，而是采用了 char_traits（见 36.2.2 节）。类似地，标准库算法依赖 iterator_traits（见 33.1.3 节），标准库容器依赖 allocators（见 34.4 节）。萃取的使用将在 28.2.4 节中介绍。

25.2.5.1　默认函数模板实参

很自然地，默认模板实参也可用于函数模板。例如：

```
template<typename Target =string, typename Source =string>
Target to(Source arg)           // 将 Source 转换为 Target
{
    stringstream interpreter;
    Target result;

    if (!(interpreter << arg)                   // 将 arg 写入流
        || !(interpreter >> result)             // 从流读取 result
        || !(interpreter >> std::ws).eof())     // 流中还有剩余内容?
        throw runtime_error{"to<>() failed"};

    return result;
}
```

只有当一个函数模板实参无法推断或没有默认实参时，我们才需要显式指定它，因此我们可以编写如下的代码：

```
auto x1 = to<string,double>(1.2);   // 太明确（也太繁琐了）
auto x2 = to<string>(1.2);          // Source 被推断为 double
auto x3 = to<>(1.2);                // Target 默认为 string；Source 被推断为 double
auto x4 = to(1.2);                  // <> 是冗余的
```

如果所有函数模板参数都有默认实参，则 <> 可以省略（与函数模板特例化中完全一样，见25.3.4.1节）。

to() 的这个实现对简单类型如 to<double>(int) 来说有点儿小题大作了，不过我们可以通过特例化来改进这个实现（见 25.3 节）。注意，to<char>(int) 是错误的，因为 char 和 int 没有共同的 string 表示。对于标量数值类型间的转换，我更愿意用 narrow_cast<>()（见 11.5 节）。

25.3 特例化

默认情况下，一个模板有单一的定义，可用于用户能想到的所有模板实参（或模板实参的组合）。但对某些模板设计者而言，并不希望总是这样。我们可能想表达“如果模板实参是一个指针，使用这个实现；否则，使用那个实现”或者是“若模板实参不是 My_base 的派生类的指针，给出一个错误”。很多类似的设计需求都可以通过这样一种技术来满足：为一个模板提供可选的多个定义，令编译器能根据用户使用模板时提供的实参来选择使用哪个定义。这种可选的模板定义称为用户自定义特例化（user-defined specialization），或简称用户特例化（user specialization）。我们可能像下面的代码这样使用 Vector：

```
template<typename T>
class Vector { // 通用向量类型
    T* v;
    int sz;
public:
    Vector();
    explicit Vector(int);

    T& elem(int i) { return v[i]; }
    T& operator[](int i);

    void swap(Vector&);
    // ...
};

Vector<int> vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;
```

在这段代码中，大多数 Vector 都是某种指针类型的 Vector。这种用法很常见，其原因有很多，但主要原因是我们必须使用指针才能实现运行时多态行为（见 3.2.2 节和 20.3.2 节）。也就是说，任何练习面向对象编程并使用类型安全容器（如标准库容器）的人最终都会大量使用保存指针的容器。

大多数 C++ 实现默认会复制模板函数的代码。对运行时性能而言这通常是好事，但对一些重要程序，如 Vector 的例子，除非特别小心，否则这种策略会导致代码膨胀。

幸运的是，有一个很明显的解决方案：指针的容器共享一个特殊的实现，这可以通过特例化表示。首先，我们为 void 指针的 Vector 定义一个特例化版本：

```
template<>
class Vector<void*> {          // 完整特例化
    void** p;
    // ...
    void*& operator[](int i);
};
```

这个特例化版本即可作为一个用于所有指针的 Vector 的公用实现。另一种用途是基于保存一个 void* 的单一共享实现类来实现 unique_ptr<T>。

前缀 template<> 表示这是一个不必指明模板参数的特例化版本。特例化版本所使用的模板实参由模板名后面括号 <> 中的内容指定。即，<void*> 指出这个定义将用于所有 T 为 void* 的 Vector。

Vector<void*> 是一个完整特例化（complete specialization）。即，在使用此特例化版本时不用再指定或推断任何模板参数。下面的 Vector 声明就会使用 Vector<void*> 特例化版本：

```
Vector<void*> vpv;
```

为了定义一个用于且只用于所有指针的 Vector 的特例化版本，我们可编写代码如下：

```
template<typename T>
class Vector<T*> : private Vector<void*> {    // 部分特例化
public:
    using Base = Vector<void*>;

    Vector() {}
    explicit Vector(int i) : Base(i) {}

    T*& elem(int i) { return reinterpret_cast<T*&>(Base::elem(i)); }
    T*& operator[](int i) { return reinterpret_cast<T*&>(Base::operator[](i)); }

    // ...
};
```

模板名后的特例化模式 <T*> 指出这个版本用于所有指针类型；即，它用于所有模板实参可表示为 T* 的 Vector。例如：

```
Vector<Shape*> vps;    // <T*> 为 <Shape*>，因此 T 是 Shape
Vector<int**> vppi;    // <T*> 为 <int**> 因此 T 是 int*
```

模式中包含模板参数的特例化版本称为部分特例化（partial specialization），与完整特例化（如 Vector<void*>）相对，其中“模式（pattern）”指一个特定类型。

注意，当使用部分特例化时，模板参数是从特例化模式推断出的；而它并非原模板的实际实参。例如，对 Vector<Shape*>，T 为 Shape 而非 Shape*。

Vector 的这个部分特例化版本为所有指针的 Vector 提供了共享的实现。Vector<T*> 类是 Vector<void*> 的一个接口，仅通过派生和内联扩展实现。

对 Vector 实现的这个改进并未影响呈现给用户的接口，这是很重要的。特例化的作用是为共同接口的不同使用提供可选的实现。很自然地，我们可以为通用 Vector 和指针的 Vector 设计不同的名字。但是，如果我们试图这样做，很多不了解详情的人就会忘记使用指针类，并发现代码比他们的预期长得多。在此情况下，将关键实现隐藏在公共接口之后就要好得多了。

在实际应用中，这种技术已被证明能有效抑制代码膨胀。不使用类似技术编程（用 C++ 或其他有类似类型参数化特性的语言）的人会发现，即使在中等规模的程序中，复制的代码也会占据数兆字节的内存空间。由于消除了编译这些额外 Vector 版本所需的时间，这种技术还能大幅度降低编译和链接时间。这种技术通过最大化共享代码量来最小化代码膨胀，用单一特例化版本实现所有指针链表就是一个很好的例子。

现在某些编译器在逐渐进化，能在没有程序员帮助的情况下完成这种特殊的优化，但这种技术本身仍然是一种很有用的通用技术。

有一类技术对多种类型的值使用单一的运行时表示，并依赖（静态）类型系统保证对这些值的使用仅依据它们声明的类型。这类技术曾被称为类型擦除（type erasure）。在C++领域，最早涉及这类技术的是最初的模板论文［Stroustrup, 1988］。

25.3.1 接口特例化

有时，特例化并非是为了优化算法，而是为了修改接口（乃至表示）。例如，标准库complex使用特例化来为重要版本（如complex<float>和complex<double>）调整构造函数和重要操作的实参类型。complex的通用（主）模板（见25.3.1.1节）就像下面这样：

```
template<typename T>
class complex {
public:
    complex(const T& re = T{}, const T& im = T{});
    complex(const complex&);                    // 类复制构造函数
    template<typename X>
        complex(const complex<X>&);             // 从 complex<X> 转换为 complex<T>

    complex& operator=(const complex&);
    complex<T>& operator=(const T&);
    complex<T>& operator+=(const T&);
    // ...
    template<typename X>
        complex<T>& operator=(const complex<X>&);
    template<typename X>
        complex<T>& operator+=(const complex<X>&);
    // ...
};
```

注意，标量赋值运算符接受引用实参。这对float来说效率不高，因此complex<float>采用传值参数：

```
template<>
class complex<float> {
public:
    // ...
    complex<float>& operator= (float);
    complex<float>& operator+=(float);
    // ...
    complex<float>& operator=(const complex<float>&);
    // ...
};
```

complex<double>也采用了类似的优化策略。此外，还提供了从complex<float>和complex<long double>到complex<double>的转换（如23.4.6节所述）

```
template<>
class complex<double> {
public:
    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(const complex<float>&);
    explicit constexpr complex(const complex<long double>&);
    // ...
};
```

注意，这些特例化版本的构造函数是 constexpr 的，这令 complex<double> 成为一个字面值类型。但我们不会对通用的 complex<T> 这样做。而且，这个定义利用了 complex<float> 转换为 complex<double> 是安全的（永远不会发生窄化）这一知识，因此可以定义接受 complex<float> 的隐式构造函数。但是，complex<long double> 的构造函数就被声明为显式的，以降低窄化转换的可能性。

25.3.1.1　实现特例化

特例化可用来为特定模板参数集合提供可选的类模板实现。在这种情况下，特例化版本甚至可以提供与通用模板不同的实现方式。例如：

```
template<typename T, int N>
class Matrix;          // T 的 N 维矩阵

template<typename T,0>
class Matrix {         // N==1 的特例化版本
    T val;
    // ...
};

template<typename T,1>
class Matrix {         // N=1 的特例化版本
    T* elem;
    int sz;    // 元素个数
    // ...
};

template<typename T,2>
class Matrix {         // N=2 的特例化版本
    T* elem;
    int dim1; // 行数
    int dim2; // 列数
    // ...
};
```

25.3.2　主模板

当同时拥有一个模板的通用定义及其针对特定模板实参集合定义的特例化实现版本时，我们称最通用的模板定义为主模板（primary template）。主模板为所有特例化版本定义了接口（见 iso.14.5.5）。即，主模板用来确定某个模板的使用是否合法，并参与重载解析。只有主模板被选择（匹配）后，才会考虑特例化版本。

主模板必须在任何特例化版本之前声明。例如：

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

template<typename T>
class List {                    // 错误：主模板声明在特例化版本之后
    // ...
};
```

主模板提供的关键信息是用户在使用模板或其特例化版本时应提供哪些模板参数。如果我们

为模板定义了约束检查（见24.4节），它本质上属于主模板的内容，因为概念是用户使用模板所关心且必须理解的内容。例如：

```
template<typename T>
class List {
    static_assert(Regular<T>(),"List<T>: T must be Regular");
    // ...
};
```

出于技术上的原因（C++语言不能识别约束检查究竟是什么），我们必须在每个特例化版本中都复制约束检查。

为了定义特例化版本，我们给出主模板的声明就足够了：

```
template<typename T>
class List;                    // 未定义

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

如果程序中用到主模板的话，我们需要在某处给出其定义（见23.7节）。但如果主模板在程序中从未实例化，则无须定义。基于此，我们可以定义只接受几种特定实参组合的模板。如果用户特例化了一个模板，则在该版本每次使用时（使用了特例化所用类型），其定义都必须在相同作用域中，即，在使用时定义必须是可见的。例如：

```
template<typename T>
class List {
    // ...
};
List<int*> li;

template<typename T>
class List<T*> {               // 错误：特例化未定义即使用
    // ...
};
```

在本例中，List对int*的特例化版本定义位于List<int*>使用之后。

对于一组特定的模板实参，每次使用都应基于相同的特例化实现，这点非常重要。如果不然，类型系统就崩溃了：在不同地方等价地使用同一个模板却会产生不同的结果，程序中不同部分创建的对象可能不兼容。这显然是灾难性的，因此程序员必须特别小心，保证显式特例化贯穿程序始终都是一致的。原则上，C++编译器应该有能力检测出不一致的特例化，但C++标准并不要求编译器做到这点，而确有编译器做不到这点。

模板的所有特例化版本都应声明在与主模板相同的名字空间中。如果某个特例化版本被使用，而且它是显式声明的（相对于从更通用的模板隐式生成），则它也必须显式定义（见23.7节）。换句话说，显式特例化一个模板就意味着编译器不再为此特例化版本生成（其他）定义。

25.3.3 特例化顺序

一个特例化版本比另一个版本更特殊（more specialized）是指与其特例化模式匹配的所有实参列表也都与另一个版本的特例化模式匹配，但反之不成立。例如：

```
template<typename T>
    class Vector;              // 通用的：主模板
template<typename T>
    class Vector<T*>;          // 任意指针的特例化版本
template<>
    class Vector<void*>;       // void* 的特例化版本
```

任何类型都可以作为最通用的 Vector 的模板实参，但只有指针能用作 Vector<T*> 的模板实参，而只有 void* 能用作 Vector<void*> 的模板实参。

在对象、指针等的声明中，最特例化的版本优先于其他版本被采纳。

特例化模式可用多个类型指定，这些类型由模板参数推断所允许的结构组合在一起（见 23.5.2 节）。

25.3.4　函数模板特例化

特例化对模板函数也很有用（见 25.2.5.1 节）。但是，我们可以重载函数，因此很少考虑特例化。而且，C++ 仅支持函数的完整特例化（见 iso.14.7），因此在可能需要尝试部分特例化的地方应使用重载。

25.3.4.1　特例化与重载

考虑 12.5 节和 23.5 节中的希尔排序算法。这些版本用 < 比较元素，用自定义的复杂代码交换元素。我们可以给出更好的定义：

```
template<typename T>
bool less(T a, T b)
{
    return a<b;
}

template<typename T>
void sort(Vector<T>& v)
{
    const size_t n = v.size();

    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i!=n; ++i)
            for (int j=i-gap; 0<=j; j-=gap)
                if (less(v[j+gap],v[j]))
                    swap(v[j],v[j+gap]);
}
```

这段代码并未改进算法本身，但它改进了实现。我们现在有了命名实体 less 和 swap，随后就可以为其提供改进版本。这种名字通常称为定制点（customization point）。

如你所见，sort() 不能正确排序 Vector<char*>，因为 < 比较的是两个 char* 而非两个字符串。即，它比较的是两个字符串的第一个 char 的地址，而我们希望它做的是比较指针指向的字符。less() 针对 const char* 的一个简单特例化版本可编写如下：

```
template<>
bool less<const char*>(const char* a, const char* b)
{
    return strcmp(a,b)<0;
}
```

类似于类模板特例化（见 25.3 节），前缀 template<> 表明这个特例化版本使用时可不必给出

模板参数。模板函数名之后的 <const char*> 表示当模板实参为 const char* 时使用此特例化版本。由于模板实参可以从函数实参列表推断出来，我们无须显式指定模板实参。因此，特例化版本的定义可简化如下：

```
template<>
bool less<>(const char* a, const char* b)
{
    return strcmp(a,b)<0;
}
```

给定前缀 template<>，第二个空 <> 就是冗余的了，因此我们通常可以像下面这样简单编写：

```
template<>
bool less(const char* a, const char* b)
{
    return strcmp(a,b)<0;
}
```

我更喜欢这种更简洁的形式。我们还可以继续简化。在最后这个版本中，特例化与重载的区别已经微乎其微了，而且这些区别很大程度上是无关紧要的，因此我们可以简单编写代码如下：

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

现在我们已经将 less() “特例化”为一个语义上正确的版本，接下来可以考虑如何处理 swap() 了。对我们的 sort() 来说，标准库 swap() 的语义是正确的，而且它已针对任何具有高效移动操作的类型进行了优化。因此，如果我们用 swap() 代替代价可能很高的一次拷贝操作，即可对很多实参类型提高性能。

当一个实参类型的非常规性导致通用算法得不到我们所希望的结果时（如 less() 之于 C 风格字符串），特例化就能派上用场了。这些“非常规类型”通常是内置指针和数组类型。

25.3.4.2 非重载的特例化

特例化和重载的区别是什么？从技术角度来说，它们的区别在于所有函数个体都可以参与重载，而只有主模板能参与特例化（见 25.3.1.1 节）。不过我实在想不出有什么实际例子体现出这种区别。

函数特例化的用途很有限。其中一个用途是在无实参的函数中进行选择：

```
template<typename T> T max_value();  // 无定义

template<> constexpr int max_value<int>() { return INT_MAX; }
template<> constexpr char max_value<char>() { return CHAR_MAX; }
//...

template<typename Iter>
Iter my_algo(Iter p)
{
    auto x = max_value<Value_type<Iter>>();      // max_value() 特例化版本所对类型
    // ...
}
```

在本例中，我使用类型函数 Value_type<> 获取 Iter 所指向的对象的类型。

为了获得与重载大致相同的效果，我们必须传递一个哑（无用的）实参。例如：

```
int max2(int) { return INT_MAX; }
char max2(char) { return INT_MAX; }

template<typename Iter>
Iter my_algo2(Iter p)
{
    auto x = max2(Value_type<Iter>{});      // 重载 max2() 所用类型
    // ...
}
```

25.4　建议

[1] 使用模板提高类型安全；25.1 节。
[2] 使用模板提高代码抽象水平；25.1 节。
[3] 使用模板提供灵活高效的类型和算法参数化；25.1 节。
[4] 记住，值模板实参必须是编译时常量；25.2.2 节。
[5] 使用函数对象作为类型实参，从而实现“策略化”的类型和算法参数化；25.2.3 节。
[6] 使用默认模板实参为简单使用提供简单符号表示；25.2.5 节。
[7] 为非常规类型（如数组）进行模板特例化；25.3 节。
[8] 利用模板特例化优化重要实例；25.3 节。
[9] 在任何特例化版本之前声明主模板；25.3.1.1 节。
[10] 特例化版本的定义必须位于对其所有使用都可见的作用域中；25.3.1.1 节。

实 例 化

每个复杂问题都有一个清晰、简单但错误的答案。

——亨利·路易斯·门肯

- 引言
- 模板实例化
  何时需要实例化；手工控制实例化
- 名字绑定
  依赖性名字；定义点绑定；实例化点绑定；多实例化点；模板和名字空间；过于激进的 ADL；来自基类的名字
- 建议

26.1 引言

模板提供了一种异常灵活的代码组织机制，这是它的一大优势。为了生成高质量代码，编译器从下列来源组合代码（信息）

- 模板定义及其词法环境；
- 模板实参及其词法环境；
- 模板使用环境。

决定最终代码性能的关键是编译器能同时从这些上下文环境中查看代码，并能依据所有可用信息将代码组织在一起。这其中存在的问题是模板定义代码的局部性不如我们所期望的那么高（模板实参和模板使用也是如此）。有时，我们可能会困惑于模板定义中使用的某个名字到底是什么：

- 它是一个局部名字？
- 它是与某个模板实参关联的名字？
- 它是类层次中一个基类中的名字？
- 它是一个具名的名字空间中的名字？
- 它是一个全局名字？

本章讨论这些与名字绑定（name binding）相关的问题以及它们所暗示的程序设计风格。

- 3.4.1 节和 3.4.2 节介绍了模板的基础知识。
- 第 23 章详细介绍了模板及模板实参的使用。
- 第 24 章讨论泛型程序设计及“概念”的关键思想。
- 第 25 章介绍了类模板和函数模板的技术细节，并介绍了特例化的概念。
- 第 27 章讨论模板和类层次之间的关系（支持泛型和面向对象程序设计）。
- 第 28 章关注模板作为一种生成类和函数的语言。
- 第 29 章给出一个大型程序范例，展示了如何组合使用语言特性和程序设计技术。

26.2　模板实例化

给定一个模板的定义及其使用，C++ 编译器应负责生成正确的代码。从一个类模板和一组模板实参，编译器应生成一个类定义并为程序中用到的成员函数生成定义（也仅为用到的成员函数生成定义；见 26.2.1 节）。从一个模板函数和一组模板实参，编译器应该生成一个函数。此过程通常称为模板实例化（template instantiation）。

生成的类和函数称为特例化（specializations）。当需要区分编译器生成的特例化版本和程序员显式编写的特例化版本（见 25.3 节）时，我们分别称它们为生成特例化（generated specialization）和显式特例化（explicit specialization）。显式特例化通常也被称为用户自定义特例化（user-define specialization）或简称用户特例化（user specialization）。

为了在重要程序中使用模板，程序员必须理解模板定义中的名字是如何绑定到声明的，以及源码可以如何组织（见 23.7 节）。

默认情况下，编译器依据名字绑定规则（见 26.3 节）为用到的模板生成类和函数。即，程序员无须显式指出需要为哪些模板的哪些版本生成代码。这是非常重要的，因为对程序员而言，确切了解需要模板的哪个版本并不简单。我们常会遇到程序员从未听说过的模板用于库的实现，或是从未听说过的模板实参类型用于模板使用这样的情况。例如，标准库 map（见 4.4.3 节和 31.4.3 节）是用红黑树模板实现的，只有那些最富好奇心的用户才了解它所使用的数据类型和操作。一般而言，只有递归地检查应用代码库中用到的模板，才可能获知需要生成的函数集合。这样的工作显然更适合计算机而不是人来做。

另一方面，程序员有时需要能特别指明应该从模板生成哪些代码（见 26.2.2 节），这样就能精细地控制实例化的上下文，这是很重要的。

26.2.1　何时需要实例化

只有在需要类定义时才必须生成类模板的特例化版本（见 iso.14.7.1）。特别是，如果只是为了声明类的指针，是不需要实际的类定义的。例如：

```
class X;
X* p;      // 正确：不需要 X 的定义
X a;       // 错误：需要 X 的定义
```

当定义模板类时，这个区别可能很重要。对一个模板类而言，除非程序中真正需要其定义，否则它是不会实例化的。例如：

```
template<typename T>
class Link {
    Link* suc;      // 正确：（还）不需要 Link 的定义
    // ...
};

Link<int>* pl;      // （还）不需要 Link<int> 的实例化

Link<int> lnk;      // 现在我们需要实例化 Link<int> 了
```

模板使用之处定义了一个实例化点（见 26.3.3 节）。

对于一个模板函数，只有当它真正被使用时，才需要一个函数实现来实例化它。“被使用”的含义是“被调用或被获取地址”。特别是，实例化一个类模板并不意味着要实例化它的所有成员函数。这使得程序员在定义类模板时获得了很重要的灵活度。考虑如下代码：

```
template<typename T>
class List {
    // ...
    void sort();
};

class Glob {
    // ... 无比较运算符 ...
};

void f(List<Glob>& lb, List<string>& ls)
{
    ls.sort();
    // ... 使用 lb 上的操作，但不能调 lb.sort() ...
}
```

在本例中，List<string>::sort() 被实例化了，但 List<Glob>::sort() 未被实例化。这既减少了生成的代码量也避免了重新设计程序的麻烦。假如为 List<Glob>::sort() 生成代码，我们就不得不为 Glob 增加一些 List::sort() 所需要的操作，并将 sort() 重新定义为非 List 的成员函数（无论如何是更好的方法）或是使用其他容器保存 Glob。

26.2.2 手工控制实例化

C++ 语言不要求用户做任何事来完成模板实例化。但它确实提供了两种途径帮助用户在需要时控制实例化。需要控制实例化的原因包括：

- 通过消除冗余的重复实例化代码来优化编译和链接过程；
- 准确掌握哪些实例化点被使用，从而消除复杂名字绑定上下文带来的意外。

一个显式实例化请求（通常简称为显式实例化，explicit instantiation）在语法上就是一个特例化声明加上关键字 template 前缀（template 后面没有 <）

```
template class vector<int>;                     // 类
template int& vector<int>::operator[](int);     // 成员函数
template int convert<int,double>(double);       // 非成员函数
```

模板声明以 template< 开始，而简单的 template 则表示一个实例化请求的开始。注意，template 后接一个完整的声明，仅有一个名字是不够的：

```
template vector<int>::operator[];   // 语法错误
template convert<int,double>;       // 语法错误
```

与模板函数调用类似，我们可以忽略可从函数实参推断出的模板实参（见 23.5.1 节）。例如：

```
template int convert<int,double>(double);   // 正确（冗余的）
template int convert<int>(double);          // 正确
```

当显式实例化一个类模板时，它的所有成员函数也同时被实例化。

实例化请求可能会对链接时间和重编译效率有很大影响。我曾经见过这类例子，将大部分模板实例化放在单一的编译单元中，从而将数小时的编译时间降为几分钟。

相同的特例化有两个定义是编译错误。如果这种多重特例化是用户自定义的（见 25.3 节）、隐式生成的（见 23.2.2 节）或是显式要求的，倒没太大问题。但是，C++ 标准不要求编译器检测分散在分离编译单元中的多重实例化。这允许聪明的编译器忽略冗余的实例化代码，从而避免使用显式实例化的麻烦所带来的问题。但是，C++ 又不要求编译器必须实现这种聪明的策略，因此“不那么聪明的”编译器的用户就必须自己避免多重实例化了。用户没有

检查多重实例化的最坏后果就是程序无法链接；而不会有语义悄然改变的情况。

作为显式实例化请求的补充，C++ 语言还提供了显式不实例化请求（通常称为外部模板，extern template）。其明显用途是：在某个编译单元中对一个特例化版本进行显式实例化，当在其他编译单元中使用此版本时，使用 extern template。这复制了经典的一次声明多次定义技术（见 15.2.3 节）。例如：

```
#include "MyVector.h"

extern template class MyVector<int>;    // 禁止隐式实例化
                                        // 在其他某处显式实例化
void foo(MyVector<int>& v)
{
    // ... 在这里使用 vector ...
}
```

“其他某处”的代码可能像下面这样：

```
#include "MyVector.h"

template class MyVector<int>;    // 在此编译单元中实例化；使用此实例化点
```

除了为一个类的所有成员生成特例化代码之外，显式实例化还确定了单一的实例化点，从而其他实例化点（见 26.3.3 节）可被忽略。我们可利用这一点将显式实例化置于共享库中。

26.3　名字绑定

我们在定义模板函数时应尽量降低对非局部信息的依赖，原因在于模板会在未知上下文中基于未知类型生成函数和类。每处微妙的上下文依赖都可能浮出水面，成为某人的烦恼，而这“某人”不太可能希望了解模板的实现细节。我们应遵循尽量避免全局名字这一基本原则，对模板尤其如此。因此，我们尽可能地令模板定义是自包含的，将本来是全局上下文中的实体以模板实参的形式提供（例如萃取，见 28.2.2 节和 33.1.3 节），并用概念记录模板实参依赖关系（见 24.3 节）。

但是，为了给模板一个最精练的实现，有时我们必须使用一些非局部名字。特别是，我们经常需要编写一组相互协作的模板函数，而非仅仅编写一个自包含的函数。这些函数可能是类成员，但并不总是这样，有时非局部函数可能是更好的选择。典型的例子是 sort() 调用 swap() 和 less()（见 25.3.4 节）。标准库算法是一个大规模的例子（见第 32 章）。当需要使用一些非局部实体时，应优选具名的名字空间而非全局作用域，这样做能保留一定的局部性。

具有常规名字和语义的操作，如，+、*、[] 和 sort()，是另外一类在模板定义中使用的非局部名字。考虑下面的代码：

```
bool tracing;

template<typename T>
T sum(std::vector<T>& v)
{
    T t {};
    if (tracing)
        cerr << "sum(" << &v << ")\n";
    for (int i = 0; i!=v.size(); i++)
        t = t + v[i];
```

```
    return t;
}
// ...

#include<quad.h>

void f(std::vector<Quad>& v)
{
    Quad c = sum(v);
}
```

模板函数 sum() 看起来无害，但它依赖的多个名字都不是在其定义中显式说明的，例如 tracing、cerr 和 + 运算符。在本例中，+ 定义在 <quad.h> 中：

```
Quad operator+(Quad,Quad);
```

重要的是，Quad 的相关内容都不在 sum() 的定义的作用域中，且不能假定 sum() 的编写者了解 Quad 类。特别是，+ 的定义在程序文本中的位置可能在 sum() 之后，甚至在时间上也是如此。

为模板显式或隐式使用的每个名字寻找其声明的过程称为*名字绑定*（name binding）。模板名字绑定的普遍问题是模板实例化涉及 3 个上下文，而它们又无法清晰地分开：

［1］ 模板定义的上下文；

［2］ 实参类型声明的上下文；

［3］ 模板使用的上下文。

当定义一个函数模板时，为了令其从实参的角度来讲有意义，我们希望保证有足够的上下文可用，而不必从使用点的环境中获取“意外内容”。为了有助于实现这一点，C++ 语言将模板定义中使用的名字分为以下两类。

［1］ *依赖性名字*：即，依赖于模板参数的名字。这类名字在实例化点完成绑定（见 26.3.3 节）。在 sum() 的例子中，+ 的定义可以在实例化上下文中找到，因为它接受模板实参类型作为运算对象。

［2］ *非依赖性名字*：即，不依赖于模板参数的名字。这类名字在模板的定义点完成绑定（见 26.3.2 节）。在 sum() 的例子中，模板 vector 定义在标准头文件 <vector> 中，而布尔变量 tracing 位于 sum() 定义点的作用域中。

无论是依赖性名字还是非依赖性名字，都必须位于其使用点的作用域中，或是在“实参依赖”查找中能找到（ADL；见 14.2.4 节）。

接下来的几节将深入讨论一些重要技术细节，解决对一个特例化版本如何绑定模板定义中的依赖性和非依赖性名字的问题。全部细节请参阅 iso.14.6。

26.3.1 依赖性名字

“N 依赖模板参数 T”的最简单的定义是“N 是 T 的成员”。但不幸的是，这并不很充分：Quad 的加法操作（见 26.3 节）就是一个反例。因此，我们称一个函数调用依赖一个模板参数当且仅当满足下列条件：

［1］ 根据类型推断规则（见 23.5.2 节），函数实参的类型依赖于一个模板参数 T，例如，f(T(1))、f(t)、f(g(t)) 及 f(&t)，假定 t 的类型是 T。

［2］ 根据类型推断规则（见 23.5.2 节），函数有一个参数依赖于 T，例如，f(T)、f(list<T>&)

及 f(const T*)。

大体上，如果被调用函数的实参或形参明显依赖于模板参数，则函数名是依赖性名字。例如：

```
template<typename T>
T f(T a)
{
    return g(a);    // 正确：a 是一个依赖性名字，因此 g 也是
}

class Quad { /* ... */ };
void g(Quad);

int z = f(Quad{2});        // f 的 g 绑定到 g(Quad)
```

如果一个函数调用碰巧有一个实参与实际的模板参数类型匹配，则它不是依赖性的。例如：

```
class Quad { /* ... */ };

template<typename T>
T ff(T a)
{
    return gg(Quad{1});      // 错误：作用域中没有 gg()，gg(Quad{1}) 并不依赖于 T
}

int gg(Quad);

int zz = ff(Quad{2});
```

假如 gg(Quad{1}) 被认为是依赖性的，那么对阅读模板定义代码的人来说，它的含义就太奇怪了。如果程序员希望 gg(Quad) 被调用，则 gg(Quad) 的定义应该放在 ff() 的定义之前，这样在分析 ff() 时，gg(Quad) 就在作用域中了。这与非模板函数定义的规则完全一样（见 26.3.2 节）。

默认情况下，编译器假定依赖性名字不是类型名。因此，为了使依赖性名字可以是一个类型，你必须用关键字 typename 显式说明，例如：

```
template<typename Container>
void fct(Container& c)
{
    Container::value_type v1 = c[7];   // 语法错误：编译器假定 value_type 不是类型名
    typename Container::value_type v2 = c[9];    // 正确：显式说明 value_type 是类型
    auto v3 = c[11];                             // 正确：让编译器推断
    // ...
}
```

我们可以引入类型别名（见 23.6 节）来避免使用 typename 的尴尬。例如：

```
template<typename T>
using Value_type<T> = typename T::value_type;

template<typename Container>
void fct2(Container& c)
{
    Value_type<Container> v1 = c[7];   // 正确
    // ...
}
```

类似地，命名 .（点）、–> 或 :: 后面的成员模板需要使用关键字 template。例如：

```
class Pool {    // 某个分配器
public:
    template<typename T> T* get();
    template<typename T> void release(T*);
    // ...
};

template<typename Alloc>
void f(Alloc& all)
{
    int* p1 = all.get<int>();                    // 语法错误：编译器 get 是非模板名
    int* p2 = all.template get<int>();           // 正确：编译器假定 get() 是一个模板
    // ...
}

void user(Pool& pool){
{
    f(pool);
    // ...
}
```

与使用 typename 显式声明一个名字是类型名相比，使用 template 显式声明一个名字是模板名很罕见。注意两个去歧义关键字的位置：typename 出现在被限定的名字之前，而 template 则紧挨在模板名之前。

26.3.2 定义点绑定

当编译器遇到一个模板定义时，它会判断哪些名字是依赖性的（见 26.3.1 节）。如果名字是依赖性的，编译器将查找其声明的工作推迟到实例化时（见 26.3.3 节）。

编译器将不依赖于模板实参的名字当作模板外的名字一样处理；因此，在定义点位置这种名字必须在作用域中（见 6.3.4 节）。例如：

```
int x;
template<typename T>
T f(T a)
{
    ++x;      // 正确：x 在作用域中
    ++y;      // 错误：作用域中没有 y，且 y 不依赖于 T
    return a; // 正确：a 依赖于 T
}

int y;

int z = f(2);
```

如果找到了名字的声明，则编译器就会使用这个声明，即使随后可能发现“更好的”声明也是如此。例如：

```
void g(double);
void g2(double);

template<typename T>
int ff(T a)
{
    g2(2);    // 调用 g2(double);
    g3(2);    // 错误：作用域中没有 g3() in scope
```

```
    g(2);       // 调用 g(double); g(int) 不在作用域中
    // ...
}

void g(int);
void g3(int);

int x = ff(a);
```

在本例中 ff() 会调用 g(double)。而 g(int) 出现得太晚了，编译器不会考虑它，就像 ff() 不是模板或 g 命名了一个变量一样。

26.3.3　实例化点绑定

确定依赖性名字（见 26.3.1 节）含义所需的上下文由模板的使用（给定一组实参）决定。这被称作此特例化版本的实例化点（point of instantiation）（见 iso.14.6.4.1）。模板对一组给定模板实参的每次使用都定义了一个实例化点。对一个函数模板而言，此位置位于包含模板使用的最近的全局作用域或名字空间作用域中，恰好在包含此次使用的声明之后。例如：

```
void g(int);

template<typename T>
void f(T a)
{
    g(a);       // g 在实例化点绑定
}
void h(int i)
{
    extern void g(double);
    f(i);
}
// f<int> 的实例化点
```

f<int> 的实例化点在 h() 之外。这是很重要的，它保证了 h() 中调用的 g() 是全局的 g(int) 而非局部的 g(double)。模板定义中一个未限定的名字永远也不应被绑定到一个局部名字上。忽略局部名字对阻止大量糟糕的类似宏的行为是很重要的。

为了允许递归调用，函数模板的实例化点位于实例化它的声明之后。例如：

```
void g(int);

template<typename T>
void f(T a)
{
    g(a);           // g 在实例化点绑定
    if (i) h(a-1);  // h 在实例化点绑定
}

void h(int i)
{
    extern void g(double);
    f(i);
}
// f<int> 的声明点
```

在本例中，实例化点必须位于 h() 的定义之后，否则（间接）递归调用 h(a-1) 就无法处理了。

对一个模板类或一个类成员而言，实例化点恰好位于包含其使用的声明之前。

```
template<typename T>
class Container {
    vector<T> v;    // 元素
    // ...
public:
    void sort();    // 排序元素
    // ...
};

// Container<int> 的实例化点
void f()
{
    Container<int> c;    // 使用点
    c.sort();
}
```

假如实例化点在 f() 之后，c.sort() 调用就无法找到 Container<int> 的定义了。

依靠模板实参来显式化依赖关系简化了我们对模板代码的思考，更使我们能访问局部信息。例如：

```
void fff()
{
    struct S { int a,b; };
    vector<S> vs;
    // ...
}
```

在本例中，S 有一个局部名字，但由于我们将它用作模板的显式实参，而不是试图将它的名字埋藏在 vector 的定义中，因而我们不会遇到出乎意料又难以捉摸的结果。

那么我们为什么不在模板定义中彻底避开局部名字呢？这当然能解决名字查找所遇到的技术问题，但类似普通函数和类定义，我们希望能在自己的代码中自由使用“其他函数和类型”。将所有依赖关系都转换为实参会导致非常凌乱的代码。例如：

```
template<typename T>
void print_sorted(vector<T>& v)
{
    sort(v.begin(),v.end());
    for (const auto T& x : v)
        cout << x << '\n';
}

void use(vector<string>& vec)
{
    // ...
    print_sorted(vec);    // 使用 std::sort 排序，之后使用 std::cout 打印
}
```

在本例中，我们使用了两个非局部名字（sort 和 cout，都来自标准库）。为了清除它们，我们需要增加函数参数：

```
template<typename T, typename S>
void print_sorted(vector<T>& v, S sort, ostream& os)
{
    sort(v.begin(),v.end());
    for (const auto T& x : v)
        os << x << '\n';
}
```

```
void fct(vector<string>& vec)
{
    // ...
    using Iter = decltype(vs.begin());    // vec 的迭代器类型
    print_sorted(some_vec,std::sort<Iter>,std::cout);
}
```

在这个简单的例子中，为了消除对全局名字 cout 的依赖而大费周章。但是，如 sort() 所示，通常增加参数会令代码过于冗长，从而变得难以理解。

而且，如果模板的名字绑定规则更激进些，比非模板代码名字绑定限制更多，那么编写模板代码就变成和编写非模板代码完全不同的工作了。模板代码和非模板代码再也不能简单自由地融合在一起了。

26.3.4　多实例化点

在以下位置，编译器会为模板生成特例化版本：

- 任何实例化点（见 26.3.3 节），
- 任何编译单元的末尾，
- 或是为生成特例化而特别创建的编译单元中。

这反映了编译器生成特例化版本可采用的三种明显策略：

［1］ 第一次遇到调用时生成一个特例化。

［2］ 在一个编译单元末尾，为其生成所有特例化。

［3］ 一旦编译器处理完程序的所有编译单元，为程序生成所有特例化。

三种策略各有优缺点，可以组合使用。

因此，如果一个程序用相同的模板实参组合多次使用一个模板，则模板有多个实例化点。如果选择不同的实例化点可能导致两种不同的含义，则程序是非法的。即，如果一个依赖性名字或一个非依赖性名字可能有不同的绑定，则程序是非法的。例如：

```
void f(int);                                    // 这里，我声明了 int 版本

namespace N {
    class X { };
    char g(X,int);
}

template<typename T>
void ff(T t, double d)
{
    f(d);           // f 绑定到 f(int)
    return g(t,d);  // g 可能绑定到 g(X,int)
}

auto x1 = ff(N::X{},1.1);           // ff<N::X,double>；可能将 g 绑定到 N::g(X,int)，1.1 窄化转换为 1


Namespace N {                                   // 重新打开 N 声明 double 版本
    double g(X,double);
}

auto x2 = ff(N::X,2.2);                         // ff<N::X,double>；将 g 绑定到 N::g(X,double)；最佳匹配
```

ff() 有两个实例化点。对第一次调用，我们可以在 x1 的实例化位置为其生成特例化版本 g(N::X,int)，进行调用。也可以等到编译单元末尾为其生成特例化版本 g(N::X,double)。这样，ff(N::X{},1.1) 调用就产生了一个二义性错误。

在一个重载函数的两个声明之间调用它是一种草率的编程风格。但是，在一个大型程序中，程序员可能没有理由怀疑这是一个问题。对此特殊情况，编译器能捕获二义性错误。但是，类似问题可能发生在分离的编译单元中，检测错误就变得非常困难了（无论对编译器还是程序员都是如此）。C++ 标准并不要求编译器实现捕获这种问题。

为了避免奇怪的名字绑定问题，应尽量限制模板中的上下文依赖。

26.3.5 模板和名字空间

当函数被调用时，即使其声明不在当前作用域中，只要它是在某个实参所在的名字空间中声明的（见 14.2.4 节），编译器就能找到它。这对模板定义中的函数调用尤为重要，因为有了这种机制，在实例化过程中才能找到依赖性函数。编译器完成依赖性名字的绑定（见 iso.14.6.4.2）是通过查看以下两条来实现的。

[1] 模板定义点所处作用域中的名字；

[2] 依赖性调用的一个实参的名字空间中的名字（见 14.2.4 节）。

例如：

```
namespace N {
    class A { /* ... */ };
    char f(A);
}

char f(int);

template<typename T>
char g(T t)
{
    return f(t);            // 总是依赖于 T 的实参的 f()
}

char f(double);

char c1 = g(N::A());        // 导致 N::f(N::A) 被调用
char c2 = g(2);             // 导致 f(int) 被调用
char c3 = g(2.1);           // 导致 f(int) 被调用；f(double) 不会被考虑
```

在本例中，f(t) 显然是依赖性的，因此不能在定义点绑定 f。为了为 g<N::A>(N::A) 生成特例化版本，编译器在名字空间 N 中查找名为 f() 的函数，最终找到了 N::f(N::A)。

编译器能找到 f(int) 是因为它位于模板定义点的作用域中。而 f(double) 不在此作用域中（见 iso.14.6.4.1），而且实参依赖查找过程（见 14.2.4 节）不会寻找那些只接受内置类型实参的全局函数，因此编译器找不到它。我发现这一规则很容易被遗忘。

26.3.6 过于激进的 ADL

实参依赖查找（argument-dependent lookup，通常简称为 ADL）对避免冗长代码很有用处（见 14.2.4 节）。例如：

```
#include <iostream>
```

```
int main()
{
    std::cout << "Hello, world" << endl;// 正确，因为使用了 ADL
}
```

没有实参依赖查找的话，编译器就无法找到 endl 运算符。有了 ADL 的帮助，编译器就能注意到 << 的第一个实参是定义在 STD 中的 ostream。因此，它在 STD 中查找 endl，并最终成功找到（在 <IOSTREAM> 中）。

但是，在与未受限模板组合使用时，ADL 可能显得“过于激进”了。考虑下面的代码：

```
#include<vector>
#include<algorithm>
// ...

namespace User {
    class Customer { /* ... */ };
    using Index = std::vector<Customer*>;

    void copy(const Index&, Index&, int deep);    // 依赖于 deep 的值进行深或浅拷贝

    void algo(Index& x, Index& y)
    {
        // ...
        copy(x,y,false);// 错误
    }
}
```

对 User::algo()，我们猜测 User 的作者是想调用 User::copy()，这是一个很合理的猜测。但是，结果并非如此。编译器注意到 Index 实际上是一个 vector，而 vector 定义在 std 中，因此就会查找 std 中是否有相关函数可用。最终在 <algorithm> 中，它找到了

```
template<typename In, typename Out>
Out copy(In,In,Out);
```

显然，这个通用模板完美匹配 copy(x,y,false)。另一方面，如果调用 User 中的 copy()，还必须进行一次 bool 到 int 的类型转换。对于本例，以及等价的例子而言，编译器的解析结果出乎程序员意料，而且是非常隐蔽的错误之源。因此，有人认为用 ADL 查找完全通用的模板是一个语言设计错误。毕竟 std::copy() 要求一对迭代器（而不仅仅是两个相同类型的实参，例如两个 Index）。C++ 标准是这样说的，但代码常常不是这样做的。很多这种问题可以通过使用概念来解决（见 24.3 节和 24.3.2 节）。例如，假如编译器了解 std::copy() 要求两个迭代器，就能产生一个容易发现的错误。

```
template<typename In, typename Out>
Out copy(In p1, In p2, Out q)
{
    static_assert(Input_iterator<In>(), "copy(): In is not an input iterator");
    static_assert(Output_iterator<Out>(), "copy(): Out is not an output iterator");
    static_assert(Assignable<Value_type<Out>,Value_type<In>>(), "copy(): value type mismatch");
    // ...
}
```

更好的情况是，编译器能注意到对这个调用而言 std::copy() 甚至不是一个合法的候选，从而选择调用 User::copy()。例如（见 28.4 节）：

```
template<typename In, typename Out,
        typename = enable_if(Input_iterator<In>()
                             && Output_iterator<Out>()
                             && Assignable<Value_type<Out>,Value_type<In>>())>
Out copy(In p1, In p2, Out q)
{
    // ...
}
```

不幸的是，很多这类模板都在程序库（例如标准库）中，我们无法像这样修改它们。

一个好的策略是，如果一个头文件包含类型定义，则避免再在其中放置完全通用（完全不受限）的函数模板。如果你确实需要放置这样的模板，用一个约束检查保护它通常是值得的。

如果一个库中包含会引起麻烦的未受限模板，用户应该如何做呢？我们通常知道函数来自于哪个名字空间，因此可以显式指定。例如：

```
void User::algo(Index& x, Index& y)
{
    User::copy(x,y,false);      // 正确
    // ...
    std::swap(x[i],x[j]);       // 正确：只会考虑 std::swap
}
```

如果我们不想指定使用哪个名字空间，但想让编译器在函数重载解析时考虑函数的特定版本，可以使用 using 声明（见 14.2.2 节）。例如：

```
template<typename Range, typename Op>
void apply(const Range& r, Op f)
{
    using std::begin;
    using std::end;
    for (auto& x : r)
        f(x);
}
```

这样，标准库 begin() 和 end() 就进入了重载集合，会被范围 for 语句用来遍历 Range（除非 Range 有 begin() 和 end() 成员；见 9.5.1 节）。

26.3.7 来自基类的名字

如果一个类模板有一个基类，则它可以访问来自基类的名字。与其他名字类似，有两种不同的可能：

- 基类依赖于一个模板实参。
- 基类不依赖于模板实参。

后一种情况很简单，像非模板类中的基类那样处理即可。例如：

```
void g(int);

struct B {
    void g(char);
    void h(char);
};

template<typename T>
```

```
class X : public B {
public:
    void h(int);
    void f()
    {
        g(2);       // 调用 B::g(char)
        h(2);       // 调用 X::h(int)
    }
    // ...
};
```

照例，局部名字会屏蔽其他名字，因此 h(2) 绑定到 X::h(int)，而 B::h(char) 不会被考虑。类似地，调用 g(2) 绑定到 B::g(char)，不会考虑 X 外声明的任何函数。即，不会考虑全局 g()。

对于依赖模板参数的基类，我们必须更加小心，有时需要显式指定我们需要什么。考虑下面的代码：

```
void g(int);

struct B {
    void g(char);
    void h(char);
};

template<typename T>
class X : public T {
public:
    void f()
    {
        g(2);       // 调用 ::g(int)
    }
    // ...
};
void h(X<B> x)
{
    x.f();
}
```

为什么 g(2) 不调用 B::g(char)（像上一个例子那样）呢？原因在于 g(2) 不依赖于模板参数 T。因此它在定义点进行绑定；来自于模板实参 T（恰好用作基类）的名字（还）未被编译器所知，因此不会被考虑。如果我们希望来自一个依赖性类的名字被考虑，就必须明确依赖关系。有三种方式实现这个目的：

- 用依赖性类型（例如 T::g）限定名字。
- 声明一个名字指向此类的一个对象（例如 this->g）。
- 用 using 声明将名字引入作用域（例如 using T::g）。

例如：

```
void g(int);
void g2(int);

struct B {
    using Type = int;
    void g(char);
    void g2(char)
};
```

```
template<typename T>
class X : public T {
public:
    typename T::Type m;     // 正确
    Type m2;                // 错误（Type 不在作用域中）

    using T::g2();          // 将 T::g2() 引入作用域

    void f()
    {
        this->g(2);         // 调用 T::g
        g(2);               // 调用 ::g(int)；是否出乎意料？
        g2(2);              // 调用 T::g2
    }
    // ...
};

void h(X<B> x)
{
    x.f();
}
```

只有在实例化点我们才能知道参数 T 的实参（这里是 B）是否具有所要求的名字。

我们很容易忘记限定来自基类的名字，而且即使没忘记的话，限定后的代码也显得有些冗长和杂乱。但是如果不这么做，模板类中的名字就会时而绑定到基类成员，时而绑定到全局实体，完全依赖于模板实参是什么。这不是理想情况，而且 C++ 语言的基本原则是模板定义应该尽可能地自包含（见 26.3 节）。

限定对模板的依赖性基类成员的访问可能有些麻烦。但是，显式限定对维护人员很有帮助，因此程序的原作者不能过多抱怨这种额外的输入负担。当整个类层次都被模板化时，常会发生这种问题。例如：

```
template<typename T>
class Matrix_base {         // 矩阵应用为空，对所有元素的操作
    // ...
    int size() const { return sz; }
protected:
    int sz;     // 元素数量
    T* elem;    // 矩阵元素
};

template<typename T, int N>
class Matrix : public Matrix_base<T> {    // N 维矩阵
    // ...
    T* data() // 返回指向元素存储空间的指针
    {
        return this->elem;
    }
};
```

在本例中，必须使用 this-> 限定。

26.4 建议

[1] 让编译器 / 实现在需要时生成特例化版本；26.2.1 节。

[2] 如果需要精确控制实例化环境，使用显式实例化；26.2.2 节。
[3] 如果需要优化生成特例化所需的时间，使用显式实例化；26.2.2 节。
[4] 在模板定义中避免微妙的上下文依赖；26.3 节。
[5] 名字必须在模板定义点的作用域中，或是可通过实参依赖查找（ADL）找到；26.3 节和 26.3.5 节。
[6] 在实例化点之间保持绑定上下文不变；26.3.4 节。
[7] 避免完全通用的模板可被 ADL 找到；26.3.6 节。
[8] 使用概念或 static_assert 避免选择不恰当的模板；26.3.6 节。
[9] 使用 using 声明限制 ADL 触及的范围；26.3.6 节。
[10] 恰当地使用 -> 或 T:: 限定来自模板基类的名字；26.3.7 节。

模板和类层次

欧几里得第五公设和贝多芬第五交响曲；
两者只知其一的话你只能算是半文盲。

——斯坦·凯利－布特尔

- 引言
- 参数化和类层次
  生成类型；模板类型转换
- 类模板层次
  模板作为接口
- 模板参数作为基类
  组合数据结构；线性化类层次
- 建议

27.1 引言

模板和派生机制的用途包括从已有类型构造新类型，说明接口，以及更一般的，利用各种共性编写有用的代码。

- 一个模板类定义一个接口。模板自己的实现及其特例化版本可通过此接口访问。实现模板的源码（在模板定义中）对所有参数类型都是相同的。而不同特例化版本的实现则可能大不相同，但它们都应实现主模板所指定的语义。一个特例化版本可以在主模板提供的功能之外增加新功能。
- 一个基类定义一个接口。类本身的实现及其派生类的实现可以（使用虚函数）通过此接口访问。不同派生类的实现可能大不相同，但它们都应实现基类所指定的语义。一个派生类可在基类提供的功能之外增加新功能。

从设计的角度来看，两种方法异常接近，应该赋予相同的名字。由于两者都允许一个算法只有唯一表达但用于多种类型，因此人们将它们都称为多态（polymorphic，源于希腊语“多种形状”）。为了区分它们，虚函数所提供的多态能力被称为运行时多态（run-time polymorphic），而模板所提供的被称为编译时多态（compile-time polymorphic）或参数多态（parametric polymorphic）。

泛型方法和面向对象方法之间的这种相似有一定的迷惑性。面向对象程序员更多关注类（类型）层次的设计，以单个类作为接口（见第 21 章）。泛型程序员更多关注算法设计，以模板实参的概念提供接口，适应许多类型（见第 24 章）。对程序员而言，最理想的当然是精通两种技术，对它们恰当使用达到随心所欲的境界。很多情况下，在一个最优设计中两种技术都要用到。例如 vector<Shape*> 是一个编译时多态（泛型）容器，保存来自运行时多态（面向对象）类层次的元素（见 3.2.4 节）。

一般而言，好的面向对象程序设计比好的泛型程序设计需要更多的预见性，因为层次中的所有类型必须显式共享基类所定义的接口。而一个模板可接受满足其概念要求的任意类型作为实参，即使这些类型间没有显式声明的共性也没有问题。例如 accumulate()（见 3.4.2、24.2 节和 40.6.1 节）既接受 int 的 vector，也接受 complex<double> 的 list，虽然两种元素类型之间和两种序列类型之间都不存在显式声明的关系。

27.2 参数化和类层次

如 4.4.1 节和 27.2.2 节所述，模板和类层次的组合是很多有用技术的基础。因此：

- 我们何时选择使用类模板？
- 我们何时需要依赖类层次？

让我们尝试从一个稍微简化的抽象视角考虑这些问题：

```
template<typename X>
class Ct {              // 用参数表达的接口
    X mem;
public:
    X f();
    int g();
    void h(X);
};

template<>
class Ct<A> {           // 特例化（针对 A）
    A* mem;             // 实现可与主模板不同
public:
    A f();
    int g();
    void h(A);
    void k(int);        // 新增功能
};
Ct<A> cta;              // 对 A 的特例化
Ct<B> ctb;              // 对 B 的特例化
```

基于这些定义，我们可以对变量 cta 和 ctb 使用 f()、g() 和 h()，使用的分别是 Ct<A> 和 Ct<B> 的实现。我使用了一个显式特例化（见 23.5.3.4 节）来说明特例化版本的实现可以与主模板不同，而且可以增加新功能。更简单的不增加新功能的情况到目前为止更为常见。

一个使用类层次的大致等价的代码如下所示：

```
class X {
    // ...
};

class Cx {                  // 用作用域中的类型表达的接口
    X mem;
public:
    virtual X& f();
    virtual int g();
    virtual void h(X&);
};

Class DA : public Cx {      // 派生类
public:
```

```
        X& f();
        int g();
        void h(X&);
};

Class DB : public Cx {      // 派生类
        DB* p;              // 表示可比基类提供的适配实现
public:
        X& f();
        int g();
        void h(X&);
        void k(int);        // 新增功能

};

Cx& cxa {*new DA};// cxa 是 DA 的接口
Cx& cxb {*new DB};// cxb 是 DB 的接口
```

基于这些定义，我们可以对变量 cxa 和 cxb 使用 f()、g() 和 h()，使用的分别是 DA 和 DB 的实现。我在类层次版本中使用了引用，来反映必须通过指针或引用操纵派生类对象的原则，这样才能保证运行时多态行为。

在两个例子中，我们操纵的都是共享一组公共操作的对象。从这个简化的抽象视角我们可以观察到：

- 如果生成类或派生类的接口需要适用不同的类型，模板有优势。而为了通过一个基类访问派生类的不同接口，我们必须使用某种形式的显式类型转换（见 22.2 节）。
- 如果生成类或派生类的实现仅有一个参数不同或是仅仅在少数特殊情况下不同，模板有优势。我们可以通过派生类或特例化来表达非常规实现。
- 如果在编译时无法获知对象的实际类型，就必须使用类层次。
- 如果生成类型或派生类型之间有层次关系，类层次有优势。基类提供公共接口。若使用模板，则程序员必须显式定义特例化版本之间的类型转换（见 27.2.2 节）。
- 如果不希望显式使用自由存储空间（见 11.2 节），模板有优势。
- 如果运行时效率非常重要，必须使用内联，则应该使用模板（因为类层次的有效性依赖于使用指针或引用，从而无法使用内联）。

保持基类最小化且类型安全可能很困难，用必须对派生类保持一致的已有类型来表达接口可能同样困难。最终结果常常是对基类接口过度限制（例如，我们设计了一个具有“富接口”的类 X，要求所有派生类“任何时候”都必须实现此接口）或限制不足（例如，使用 void* 或最小化的 Object*）。

模板和类层次的组合提供了更多的设计选择以及超出两者独自所能提供的灵活性。例如，一个基类指针可用作模板实参来提供运行时多态（见 3.2.4 节）；一个模板参数可用来指定一个基类接口从而提供类型安全性（见 26.3.7 节和 27.3.1 节）。

C++ 语言不支持 virtual 函数模板（见 23.4.6.2 节）。

27.2.1　生成类型

将类模板理解为特定类型的创建说明是很有用的。换句话说，模板实现是一种依据说明按需生成类型的机制。因此，类模板有时也被称为类型生成器（type generator）。

就 C++ 语言规则而言，一个类模板生成的两个类之间没有任何关系。例如：

```
class Shape {
    // ...
};

class Circle : public Shape {
{
    // ...
};
```

基于这些声明，人们有时会认为 set<Circle> 和 set<Shape> 之间或者至少是 set<Circle*> 和 set<Shape*> 之间必然存在内在联系。这是一个严重的逻辑错误，它基于一个有缺陷的论据："一个 Circle 是一个 Shape，因而一组 Circle 也是一组 Shape；因此，我应该可以将一组 Circle 作为一组 Shape 使用"。但"因此"这一部分是不成立的，因为一组 Circle 保证其中的成员是 Circle，但一组 Shape 不能提供同样的保证。例如：

```
class Triangle : public Shape {
    // ...
};

void f(set<Shape*>& s)
{
    // ...
    s.insert(new Triangle{p1,p2,p3});
}

void g(set<Circle*>& s)
{
    f(s); // 类型不匹配错误：s 是一个 set<Circle*>，而不是一个 set<Shape*>
}
```

这段代码会编译失败，因为不存在 set<Circle*>& 到 set<Shape*>& 的内置类型转换，也不应有这样的转换规则。set<Circle*> 的成员是 Circle 的保证令我们可以安全而高效地对集合成员使用 Circle 特有的操作，例如确定半径。如果我们允许一个 set<Circle*> 被当作一个 set<Shape*> 来处理，就不能维持这个保证了。例如，f() 将一个 Triangle* 插入到其实参 set<Shape*> 中。如果传递给它的实参是一个 set<Circle*>，则一个 set<Circle*> 只包含 Circle* 的保证就被破坏了。

逻辑上，我们可以将一个不变的 set<Circle*> 当作一个不变的 set<Shape*> 来处理——由于我们不能改变集合，向其中插入一个不当元素的问题不会再发生。即，我们可以提供一个从 const set<const Circle*> 到 const set<const Shape*> 的类型转换。C++ 语言默认不提供这样的转换，但 set 的设计者可以自行提供。

基类和数组的组合非常糟糕，因为内置数组并不像容器那样提供类型安全。例如：

```
void maul(Shape* p, int n)      // 危险！
{
    for (int i=0; i!=n; ++i)
        p[i].draw();            // 看起来无害，但其实很危险！
}

void user()
{
    Circle image[10];           // 一幅图像由 10 个圆组成
```

```
    // ...
    maul(image,10);          // "maul" 于 10 个页
    // ...
}
```

我们用 image 调用 maul() 会发生什么？首先，image 的类型从 Circle[] 转换（退化）为 Circle*，接下来，Circle* 被转换为 Shape*。这种数组名到数组首元素指针的隐式转换是 C 风格程序设计的基础。类似地，派生类指针向基类指针的隐式转换是面向对象程序设计的基础。两者结合，为灾难性后果提供了充足的机会。

在上例中，假定 Shape 是一个大小为 4 的抽象类，Circle 增加了一个圆心和一个半径，则 sizeof(Circle)>sizeof(Shape)。当我们查看 image 的内存布局时会发现：

user()的视图:	image[0]				image[1]	image[2]	image[3]
maul()的视图:	p[0]	p[1]	p[2]	p[3]			

当 maul() 试图对 p[1] 调用虚函数时，在它期望的地方并没有虚函数指针，调用会立即失败。

注意，这种灾难不需要进行显式类型转换就会发生，因此我们应：

- 优选容器而不是内置数组。
- 要对 void f(T* p, int count) 这样的接口保持高度警惕；当 T 可能是一个基类而 count 是一个元素数量时，麻烦就要来了。
- 当 .（点运算符）用于应该有运行时多态特性的实体时要保持警惕，除非它明显用于一个引用。

27.2.2 模板类型转换

由相同模板生成的类之间默认没有任何关系（见 27.2.1 节）。但是，对某些模板我们可能希望表达它们之间的关系。例如，当定义一个指针模板时，我们可能希望反映所指对象间的继承关系。成员模板（见 23.4.6 节）允许我们在需要时指定这种关系。考虑下面的代码：

```
template<typename T>
class Ptr {      // T 的指针
    T* p;
public:
    Ptr(T*);
    Ptr(const Ptr&);                        // 拷贝构造函数
    template<typename T2>
        explicit operator Ptr<T2>();        // 将 Ptr<T> 转换为 Ptr<T2>
    // ...
};
```

我们希望为这些用户自定义的 Ptr 定义类型转换操作来表达继承关系，这与我们所熟悉的内置指针间的继承关系相似。例如：

```
void f(Ptr<Circle> pc)
{
    Ptr<Shape> ps {pc};      // 正确
    Ptr<Circle> pc2 {ps};    // 错误
}
```

我们希望当且仅当 Shape 的确是 Circle 的一个直接或间接基类时，第一条初始化语句合法。

为此，我们一般需要定义一个类型转换操作，使得当且仅当一个 T* 可以赋予一个 T2* 时，Ptr<T> 到 Ptr<T2> 的转换合法。我们可以这样做：

```
template<typename T>
    template<typename T2>
        Ptr<T>::operator Ptr<T2>()
        {
            return Ptr<T2>{p};
        }
```

当且仅当 p（是一个 T*）可以作为构造函数 Ptr<T2>(T2*) 的参数时，return 会编译成功。因此，如果 T* 可以隐式转换为 T2*，则 Ptr<T> 可以转换为 Ptr<T2>。例如，现在可编写代码如下：

```
void f(Ptr<Circle> pc)
{
    Ptr<Shape> ps {pc};     // 正确：可以将 Circle* 转换为 Shape*
    Ptr<Circle> pc2 {ps};   // 错误：不能将 Shape* 转换为 Circle*
}
```

务必注意只定义逻辑上有意义的转换。如果存疑，应使用命名转换函数，而不是转换运算符。命名转换函数不容易出现二义性问题。

一个模板的模板参数列表和模板成员不能组合在一起。例如：

```
template<typename T, typename T2>    // 错误
Ptr<T>::operator Ptr<T2>()
{
    return Ptr<T2>(p);
}
```

一个变通方法是使用类型萃取和 enable_if()（见 28.4 节）。

27.3　类模板层次

使用面向对象技术，基类常用来为一组派生类提供一个公共接口。模板可用来参数化此接口，而此参数化过程实际上是试图用相同的模板参数对整个派生类层次进行参数化。例如，我们可能想参数化经典的形状类层次（见 3.2.4 节），使用的类型参数是目标输出“设备”的抽象：

```
template<typename Color_scheme, typename Canvas>      // 有问题的例子
class Shape {
    // ...
};

template<typename Color_scheme, typename Canvas>
class Circle : public Shape {
    // ...
};
template<typename Color_scheme, typename Canvas>
class Triangle : public Shape {
    // ...
};

void user()
{
    auto p = new Triangle<RGB,Bitmapped>{{0,0},{0,60},{30,sqrt(60*60-30*30)}};
```

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

一个熟悉面向对象编程的程序员对泛型编程的最初想法（在看过 vector<T> 之类的代码后）通常就是沿此思路而来。但是，我建议在混合使用面向对象和泛型技术时一定要小心。

从代码看，这个参数化的形状类层次太冗长了，不适合实际应用。这可通过默认模板实参机制（见 25.2.5 节）来解决。但是，冗长其实还不是主要问题。如果程序中只使用了一个 Color_scheme 和 Canvas 的组合，生成的代码量与非参数化版本几乎完全一致。这里强调“几乎”是因为编译器不会为虽有定义但未使用的类模板非虚成员函数生成代码。但是，如果程序中使用了 N 个 Color_scheme 和 Canvas 的组合，每个虚函数的代码会重复 N 次。由于一个图形类层次很可能包含很多派生类、很多成员函数以及很多复杂函数，最终很可能导致严重的代码膨胀。特别是，编译器无法获知一个虚函数是否被使用，因此它必须为所有虚函数以及所有被虚函数调用的函数生成代码。参数化一个巨大的、有很多虚成员函数的类层次通常是个坏主意。

对这个形状类层次例子，参数 Color_scheme 和 Canvas 不太可能对接口有很大影响：大多数成员函数不会将它们作为函数类型的一部分。这两个参数属于不小心跑到接口中的“实现细节”，可能对性能有严重影响。其实并非整个类层次都需要这些参数，仅仅是少数几个配置函数和（最可能是）底层绘图 / 渲染函数需要。“过度参数化”通常不是一个好主意（见 23.4.6.3 节），我们应该避免使用只影响少数成员的参数。如果一个参数只影响少数成员函数，尝试用这个参数将这些函数改为模板。例如：

```
class Shape {
    template<typename Color_scheme, typename Canvas>
        void configure(const Color_scheme&, const Canvas&);
    // ...
};
```

如何在不同类和不同对象间共享配置信息是另一个问题。显然，我们不能将 Color_scheme 和 Canvas 简单地保存在 Shape 中，而不用 Color_scheme 和 Canvas 参数化 Shape。一个解决方案是将 Color_scheme 和 Canvas 中的信息“翻译”为一组标准的配置参数（例如，一组整数）。另一个解决方案是为 Shape 添加一个 Configuration* 成员，Configuration 是一个基类，提供了配置信息的通用接口。

27.3.1　模板作为接口

模板类可用来为公共实现提供一个灵活且类型安全的接口。25.3 节中的向量就是这一思想的很好例子：

```
template<typename T>
class Vector<T*>
    : private Vector<void*>
{
    // ...
};
```

此技术通常可用来提供类型安全的接口，以及将类型转换的工作交给编译器，而不是强制用户编写转换操作。

27.4 模板参数作为基类

在使用类层次进行面向对象编程时，我们将因类而异的信息放置在基类中，并通过基类中的虚函数访问这些信息（见 3.2.4 节和 21.2.1 节）。这样，我们就能写出通用的代码，而无须担心实现中的各种变形。但是，这种技术不允许我们改变接口中的类型（见 27.2 节）。而且，与简单内联函数相比，这些虚函数的调用代价也很高。作为弥补，我们可以将专用信息和操作作为模板实参传递给基类。实际上，模板实参可以作为基类本身。

下面两小节解决一个一般性问题："我们如何将分离的专用信息紧凑地组合到具有良好接口的单一对象中？"这是一个基础性问题，其解决方案具有普遍意义。

27.4.1 组合数据结构

考虑编写一个平衡二叉树的库。由于我们是要设计一个供很多不同用户使用的库，因此不能将用户（应用领域）数据类型硬编码到树的代码中。可选的方案有很多：

- 我们可以将用户数据放置在一个派生类中，通过虚函数访问它。但虚函数调用（或等价的运行时解析和检查机制）代价相对较高，而且用户数据的接口并未以用户类型表达，因此访问数据时仍需要类型转换。
- 我们可以在树结点中保存一个 void*，并让用户使用它指向在结点外分配的数据。但这样一来，内存分配操作的数量就会加倍，并增加很多（可能代价很高的）指针解引用操作，而且每个结点需要增加空间保存这个指针。为使用正确类型访问用户数据，我们还需要一次类型转换，而此转换是不能进行类型检查的。
- 我们可以在结点中保存一个 Data*，其中 Data 是用户数据结构的"通用基类"。这能解决类型检查问题，但它兼有上述两种方法在代价和易用性上的缺点。

还有很多可选方案，但我们不再一一讨论了，而是考虑下面的方法：

```
template<typename N>
struct Node_base {        // 不了解 Val（用户数据）
    N* left_child;
    N* right_child;

    Node_base();

    void add_left(N* p)
    {
        if (left_child==nullptr)
            left_child = p;
        else
            // ...
    }
    // ...
};

template<typename Val>
struct Node : Node_base<Node<Val>> {        // 派生类作为其基类的一部分
    Val v;
    Node(Val vv);
    // ...
};
```

在这段代码中，我们将派生类 Node<Val> 作为模板实参传递给其基类（Node_base）。这允

许 Node_base 在其接口中使用 Node<Val>，即使它甚至不知道后者的真正名字！

注意，Node 的内存布局很紧凑。例如，一个 Node<double> 看起来与下面的结构大致等价：

```
struct Node_base_double {
    double val;
    Node_base_double* left_child;
    Node_base_double* right_child;
};
```

不幸的是，基于这个定义，用户必须了解 Node_base 的操作和最终树的结构。例如：

```
using My_node = Node<double>;

void user(const vector<double>& v)
{
    My_node root;
    int i = 0;

    for (auto x : v) {
        auto p = new My_node{x};
        if (i++%2)                    // 选择插入位置
            root.add_left(p);
        else
            root.add_right(p);
    }
}
```

但是，用户很难保持合理的树结构，我们通常希望树自身通过实现一个树平衡算法来保证这一点。但是，为了平衡一棵树来保证搜索效率，平衡算法需要了解用户数据值。

我们如何在前面的设计中增加一个平衡算法呢？当然可以将平衡策略硬编码到 Node_base 中，让它“偷看”用户数据。例如，像标准库 map 这种平衡树实现，（默认）要求值类型提供小于操作。这样，Node_base 的操作就可以简单使用 <：

```
template<typename N>
struct Node_base {
    static_assert(Totally_ordered<N>(), "Node_base: N must have a <");

    N* left_child;
    N* right_child;
    Balancing_info bal;

    Node_base();

    void insert(N& n)
    {
        if (n<left_child)
            // ... 进行某些操作 ...
        else
            // ... 进行其他某些操作 ...
    }
    // ...
};
```

这段代码能很好地完成工作。实际上，我们将越多信息植入 Node_base，实现就越简单。特别是，我们用来参数化 Node_base 的值类型可以不是一个结点类型（就像使用

std::map），而且可以将树放在单一的紧凑的程序包中。但是，这样做并未解决我们现在所关心的基本问题：如何组合来自多个特定的分离源的信息。将所有东西放在一个地方只是避开了这个问题。

因此让我们假定用户想要操纵 Node（例如，将结点从一棵树迁移到另一棵树），从而不能简单地将用户数据保存在匿名结点中。让我们进一步假定我们希望能使用不同的平衡算法，从而需要将平衡算法作为参数。这些假设迫使我们面临前文所述的基本问题。最简单的解决方案是令 Node 组合值类型和平衡算法类型。但是，Node 不需要使用平衡算法，因此它简单地将平衡算法传递给 Node_base：

```
template<typename Val, typename Balance>
struct Search_node : public Node_base<Search_node<Val, Balance>, Balance>
{
    Val val;          // 用户数据
    search_node(Val v): val(v) {}
};
```

Balance 两次被提及，一是因为它是结点类型的一部分，二是因为 Node_base 需要创建一个 Balance 类型的对象：

```
template<typename N, typename Balance>
struct Node_base : Balance {
    N* left_child;
    N* right_child;

    Node_base();

    void insert(N& n)
    {
        if (this->compare(n,left_child))      // 使用来自 Balance 的 compare()
            // ... 进行某些操作 ...
        else
            // ... 进行其他操作 ...
    }
    // ...
};
```

我本可以用 Balance 定义一个成员，而不是将它用作基类。但是，一些重要的平衡算法不需要每个结点的数据，因此我将 Balance 作为基类。这段代码受益于空基类优化（empty-base optimization）。C++ 语言保证，如果一个基类没有非 static 数据成员，在派生类对象中不会为此基类分配内存（见 iso.1.8）。而且，上面这个设计在形式上与实际的二叉树框架［Austern, 2003］差别很小。我们可以这样使用这些类：

```
struct Red_black_balance {
    // 实现红黑树所需数据和操作
};

template<typename T>
using Rbnode = Search_node<T,Red_black_balance>;  // 红黑树类型别名

Rbnode<double> my_root;          // double 数据的红黑树

using My_node = Rb_node<double>;

void user(const vector<double>& v)
```

```
{
    for (auto x : v)
        root.insert(new My_node{x});
}
```

结点的内存布局很紧凑，而且我们可以很容易地内联那些对性能要求很高的函数。我们通过这组稍微复杂的定义实现了类型安全且方便的类型组合。与那些将 void* 引入数据结构和函数接口的方法相比，这种复杂定义提供了明显的性能优势。使用 void* 会让我们无法使用有效的基于类型的优化技术。在平衡二叉树实现的关键部分选择使用低层（C 风格）编程技术意味着严重的运行时代价。

我们将平衡算法作为一个单独的模板实参：

```
template<typename N, typename Balance>
struct Node_base : Balance {
        // ...
};

template<typename Val, typename Balance>
struct Search_node
    : public Node_base<Search_node<Val, Balance>, Balance>
{
    // ...
};
```

一些人认为这种方式清晰、直白且通用；而其他一些人可能觉得这种方式冗长、令人困惑。一种替代方案是将平衡算法作为一个隐式实参，具体方法是将其声明为一个关联类型（Search_node 的成员类型）：

```
template<typename N>
struct Node_base : N::balance_type {    // 使用 N 的 balance_type
        // ...
};

template<typename Val, typename Balance>
struct Search_node
    : public Node_base<Search_node<Val,Balance>>
{
    using balance_type = Balance;
    // ...
};
```

这种技术在标准库中被频繁使用，来尽量减少显式模板实参。

这种从基类派生的技术已经存在很长时间了，最早在 ARM（1989）中就有提及，曾被用于一个数学软件中［Barton, 1994］，因此有时被称为巴顿 – 纳克曼技巧（Barton-Nackman trick）。吉姆·考普林称之为古怪的递归模板模式（curiously recurring template pattern, CRTP）［Coplien, 1995］。

27.4.2　线性化类层次

27.4.1 节的 Search_node 例子利用其模板来压缩表示形式并避免使用 void*。这是一个通用技术，也非常有用。特别是，很多处理树的程序依赖这种技术来实现类型安全和高性能。例如，“内部程序表示”（Internal Program Representation, IPR）［DosReis, 2011］是将 C++ 代码表达为抽象语法树的通用语义表示形式。它大量使用了模板参数作为基类的技术，

既作为一种实现辅助技术（实现继承），又提供了经典的面向对象方式的抽象接口（接口继承）。其设计解决了一系列难题，包括结点的紧致性（树中可能有数以百万计的结点）、优化的内存管理、访问速度（未引入不必要的间接访问或额外结点）、类型安全、多态接口以及通用性。

用户看到的是一个抽象类层次，提供了完美的封装和表达程序语义的清晰的功能接口。例如，变量是声明，声明是语句，语句是表达式，表达式是结点，会表达为：

```
Var -> Decl -> Stmt -> Expr -> Node
```

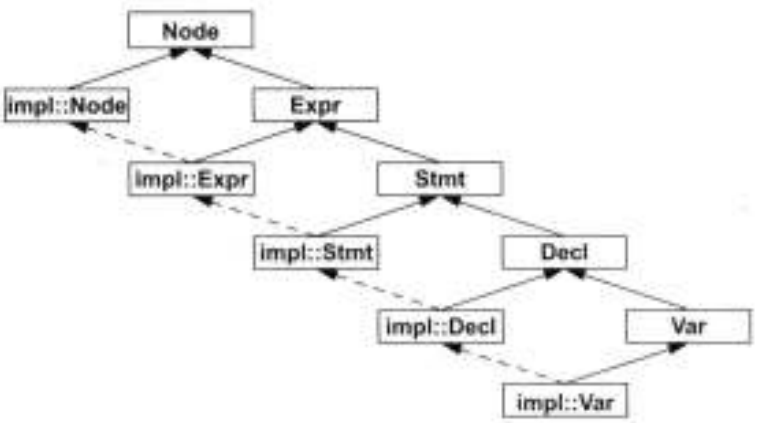
显然，IPR 的设计中进行了一些泛化，因为在 ISO C++ 中，语句不能当作表达式使用。

此外，IPR 的设计中还有一个平行的具体类层次，为接口层次中的类提供了紧凑而高效的实现：

```
impl::Var -> impl::Decl -> impl::Stmt -> impl::Expr -> impl::Node
```

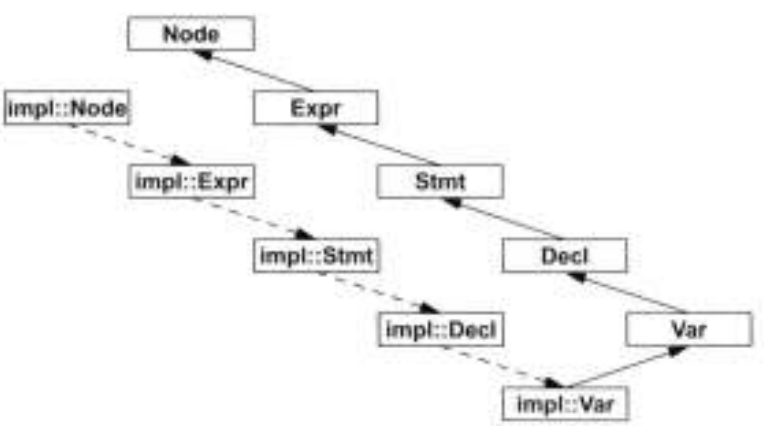
IPR 中共有大约 80 个叶结点类（如 Var、If_stmt 和 Multiply）和大约 20 个抽象概念（如 Decl、Unary 和 impl::Stmt）。

IPR 设计的第一次尝试是一个经典的多重继承“钻石”类层次（用实线箭头表示接口继承，用虚线箭头表示实现继承）：



这个设计能工作，但会导致严重的内存开销：由于需要很多数据导航至虚基类，结点尺寸很大。而且，很多虚基类的间接访问严重影响了程序性能。

最终方案是将此双层次结构线性化，从而无须使用虚基类：



全部类的派生链变为：

```
impl::Var ->
    impl::Decl<impl::Var> ->
        impl::Stmt<impl::Var> ->
            impl::Expr<impl::Var> ->
                impl::Node<impl::Var> ->
                    ipr::Var ->
                        ipr::Decl ->
                            ipr::Stmt ->
                                ipr::Expr ->
                                    ipr::Node
```

这样一个派生链表达为一个紧凑的对象，它只包含一个 vptr（见 3.2.3 节和 20.3.2 节），除此之外没有其他内部“管理数据”。

我将展示如何实现这个设计。首先介绍定义在名字空间 ipr 中的接口层次。位于最底层的是 Node，它保存用来优化遍历操作和 Node 类型识别的数据（code_category）和方便 IPR 图文件存储的数据（node_id）。这些都是很典型的应该对用户隐藏的“实现细节”。用户将会了解的是 IPR 图中的每个结点都有唯一的基类 Node，而这可以用来实现使用访客模式（visitor pattern）[Gamma, 1994]（见 22.3 节）的操作：

```
struct ipr::Node {
    const int node_id;
    const Category_code category;

    virtual void accept(Visitor&) const = 0;  // 供访客类使用的钩子
protected:
    Node(Category_code);
};
```

Node 的设计意图是只用作基类，因此其构造函数是 protected 的。它还有一个纯虚函数，因此不能被实例化（除非是其派生类实例化时它作为基类参与其中）。

表达式（Expr）是具有类型的 Node：

```
struct ipr::Expr : Node {
    virtual const Type& type() const = 0;
protected:
    Expr(Category_code c) : Node(c) { }
};
```

显然，这是 C++ 表达式的推广，因为语句和类型也是具有类型的，这体现了 IPR 的一个目标：表达所有 C++ 特性，但不实现 C++ 的所有非常规性和局限。

语句（Stmt）就是在源文件中有自己的位置并能添加各种信息注释的 Expr：

```
struct ipr::Stmt : Expr {
    virtual const Unit_location& unit_location() const = 0;          // 文件中位置
    virtual const Source_location& source_location() const = 0; // 文件

    virtual const Sequence<Annotation>& annotation() const = 0;
protected:
    Stmt(Category_code c) : Expr(c) { }
};
```

声明（Decl）是引入了新名字的 Stmt：

```
struct ipr::Decl : Stmt {
    enum Specifier { /* 存储类别、虚函数、访问控制等信息

    virtual Specifier specifiers() const = 0;
    virtual const Linkage& lang_linkage() const = 0;

    virtual const Name& name() const = 0;

    virtual const Region& home_region() const = 0;
    virtual const Region& lexical_region() const = 0;

    virtual bool has_initializer() const = 0;
    virtual const Expr& initializer() const = 0;

    // ...
protected:
    Decl(Category_code c) : Stmt(c) { }
};
```

如你所料，当涉及 C++ 代码表示时，Decl 是一个核心概念。在其中你可以找到作用域信息、存储类别、访问说明符、初始化器等内容。

最后，你可以定义一个类表示变量（Var），它是我们的接口层次中的叶结点类（最底层派生类）：

```
struct ipr::Var : Category<var_cat, Decl> {
};
```

基本上，Category 是一个辅助记号，作用是从 Decl 派生 Var，并提供 Category_code 用来优化 Node 类型识别：

```
template<Category_code Cat, typename T = Expr>
struct Category : T {
protected:
    Category() : T(Cat) { }
};
```

每个数据成员都是一个 Var，包括全局变量、名字空间变量、局部变量和类 static 变量以及常量。

与在编译器中见到的表示相比，这个接口非常小。除了 Node 中一些用于优化的数据之外，这个接口仅仅是一组具有纯虚函数的类而已。注意，它是一个没有虚基类的单一层次，这是一个很直接的面向对象设计。但是，实现这个简单、高效、可维护的接口并不容易，IPR 的解决方案显然并不是一个有经验的面向对象设计者一开始就能想到的。

每个 IPR 接口类（在 ipr 中）都有一个对应的实现类（在 impl 中）。例如：

```
template<typename T>
struct impl::Node : T {
    using Interface = T; // 使模板实参类型对用户可用
    void accept(ipr::Visitor& v) const override { v.visit(*this); }
};
```

这里的“小花招”是建立 ipr 结点与 impl 结点间的联系。特别是，impl 结点必须提供必要的数据成员并覆盖 ipr 结点中的抽象虚函数。对于 impl::Node，我们可以看到，如果 T 是一个 ipr::Node 或其派生类，则函数 accept() 被正确覆盖。

现在我们继续为 ipr 接口类的剩余部分提供实现类：

```
template<typename Interface>
struct impl::Expr : impl::Node<Interface> {
    const ipr::Type* constraint;        // constraint 是表达式的类型

    Expr() : constraint(0) { }

    const ipr::Type& type() const override { return *util::check(constraint); }
};
```

如果实参 Interface 是一个 ipr::Expr 或任何派生自 ipr::Expr 的类，则 impl::Expr 就是 ipr::Expr 的一个实现，我们可以保证这一点，因为 ipr::Expr 派生自 ipr::Node，这意味着 impl::Node 获得了基类 ipr::Node，这正是它所需要的。

换句话说，我们需要想办法为两组（不同）接口类提供实现。我们可以采用这种方式：

```
template<typename S>
struct impl::Stmt : S {
    ipr::Unit_location unit_locus;      // 编译单元中的逻辑位置
    ipr::Source_location src_locus;     // 源文件，行和列
    ref_sequence<ipr::Annotation> notes;

    const ipr::Unit_location& unit_location() const override { return unit_locus; }
    const ipr::Source_location& source_location() const override { return src_locus; }
    const ipr::Sequence<ipr::Annotation>& annotation() const override { return notes; }
};
```

即，impl::Stmt 提供了实现 ipr::Stmt 的接口所需的三个数据项，并覆盖了 ipr::Stmt 的三个虚函数来具体实现。

大体上，所有 impl 类的设计都遵循 Stmt 的模式：

```
template<typename D>
struct impl::Decl : Stmt<Node<D> > {
    basic_decl_data<D> decl_data;
    ipr::Named_map* pat;
    val_sequence<ipr::Substitution> args;

    Decl() : decl_data(0), pat(0) { }

    const ipr::Sequence<ipr::Substitution>& substitutions() const { return args; }
    const ipr::Named_map& generating_map() const override { return *util::check(pat); }
    const ipr::Linkage& lang_linkage() const override;
    const ipr::Region& home_region() const override;
};
```

最后，我们可以定义叶结点类 impl::Var：

```
struct Var : impl::Decl<ipr::Var> {
    const ipr::Expr* init;
    const ipr::Region* lexreg;

    Var();

    bool has_initializer() const override;
    const ipr::Expr& initializer() const override;
    const ipr::Region& lexical_region() const override;
};
```

注意，在我们的应用中，Var 不是一个模板，而是一个用户级抽象。Var 的实现提供了一个

泛型编程的范例，但其用途却是典型的面向对象程序设计。

继承和参数化的组合具有很强的表达力。但这种表达力可能会使初学者困惑，甚至偶尔会使面对新应用领域的有经验的程序员困惑。但是，组合带来的优点是毋庸置疑的：类型安全、性能以及源代码规模最小化。而类层次和虚函数所提供的扩展性也不会被妥协掉。

27.5　建议

[1] 当需要用代码表达一个通用概念时，仔细思考是将它表达为一个模板还是一个类层次；27.1 节。

[2] 一个模板通常为多种实参提供了公共代码；27.1 节。

[3] 抽象类能彻底隐藏实现细节，使其不为用户所见；27.1 节。

[4] 非常规实现通常最好表达为派生类；27.2 节。

[5] 如果不需要显式使用自由存储空间，模板比类层次更有优势；27.2 节。

[6] 如果内联很重要，则模板比抽象类有优势；27.2 节。

[7] 模板接口可以很容易地用模板参数类型表达；27.2 节。

[8] 如果需要进行运行时解析，则类层次是必需的；27.2 节。

[9] 模板和类层次的组合通常优于两者单独使用；27.2 节。

[10] 将模板理解为类型生成器（以及函数生成器）；27.2.1 节。

[11] 由相同模板生成的两个类之间没有必然联系；27.2.1 节。

[12] 不要混用类层次和数组；27.2.1 节。

[13] 不要简单模板化大的类层次；27.3 节。

[14] 一个模板可用来为一个单一（弱类型的）实现提供类型安全的接口；27.3.1 节。

[15] 模板可用来构成类型安全且紧凑的数据结构；27.4.1 节。

[16] 模板可用来线性化类层次（最小化空间和访问时间）；27.4.2 节。

元 编 程

探索未知之地应走两次：
一次用来犯错，一次用来纠正它们。

——约翰·斯坦贝克

- 引言
- 类型函数
  类型别名；类型谓词；选择函数；萃取
- 控制结构
  选择；迭代和递归；何时使用元编程
- 条件定义：Enable_if
  使用 Enable_if；实现 Enable_if；Enable_if 与概念；更多 Enable_if 例子
- 一个编译时列表：Tuple
  一个简单的输出函数；元素访问；make_tuple
- 可变参数模板
  一个类型安全的 printf()；技术细节；转发；标准库 tuple
- 国际标准单位例子
  Unit；Quantity；Unit 字面值常量；工具函数
- 建议

28.1 引言

操纵类和函数这种程序实体的编程通常称为元编程（metaprogramming）。我发现将模板视为生成器是很有用的：我们用模板来创建类和函数。这导致一个理念：模板程序设计用来编写特殊的程序，这种程序在编译时计算，并能生成代码。这一理念的变体也称为两级编程（two-level programming）、多级编程（multilevel programming）、生成式编程（generative programming）以及更常见的——模板元编程（template metaprogramming）。

使用元编程技术主要有两个目的：

- 提高类型安全。我们可以计算一个数据结构或算法所需的确切类型，从而不必直接操作低层数据结构（例如，我们可以消除很多显式类型转换）。
- 提高运行时性能。我们可以在编译时进行计算并选择在运行时要调用的函数。这样，我们就不必在运行时进行这些计算（例如，我们可以将很多多态行为解析为直接函数调用）。特别是，通过利用类型系统，可以显著提高内联的机会。而且，通过使用紧凑的数据结构（可能是生成的数据结构，见 27.4.2 节和 28.5 节），我们能更好地利用内存，既减少内存占用又提高运行速度。

模板的设计目标是具有很好的通用性以及能生成最优的代码［Stroustrup, 1994］。模

板提供了算术运算、选择以及递归的能力。实际上，模板构成了一个完整的编译时函数式程序设计语言［Veldhuizen, 2013］。即，模板及其实例化机制是图灵完备的。一个很好的范例是埃森耐克尔和恰尔内茨基仅花几页模板代码实现的一个 Lisp 解释器［Czarnecki, 2000］。C++ 编译时机制提供了一种纯函数式程序设计语言：你可以创建不同类型的值，但并不需要使用变量、赋值和递增运算符等特性。图灵完备性意味着可能会导致无穷编译时间，但通过编译限制可以很容易地解决（见 iso.B）。例如，无限递归会因某种编译时资源（如递归 constexpr 调用的次数、嵌套类数量或是递归嵌套的模板实例化的数量）耗尽而被捕获。

我们应该如何划定泛型编程和模板元编程的界限呢？两种极端情况是：

- 所有使用模板的编程都是模板元编程：毕竟任何一次编译时参数化都意味着一次生成“普通代码”的实例化过程。
- 所有都是泛型编程：毕竟我们仅仅是定义和使用通用类型及算法而已。

这两个极端都是没有意义的，因为它们其实都是将泛型编程和模板元编程视为等同。我认为对两者进行区分还是有意义的——可以帮助我们在问题求解方案中做出选择，并帮助我们聚焦给定问题的重点。当我编写一个通用类型或算法时，我不会感觉是在编写一个编译时程序。我并非在用所掌握的编程技巧编写程序的编译时部分，而是关注对实参要求的定义（见 24.3 节）。泛型程序设计本质上是一种设计哲学，如果你非要将它理解为编程技术的话，那它也是一种编程范型，而非普通编程技术（见 1.2.1 节）。

与此相反，元编程就是编程。重点是计算，通常还包括选择和某种形式的迭代。元编程本质上是一组实现技术。可以将实现复杂度分为四个等级：

［1］ 无计算（仅仅传递类型和值实参）。

［2］ 不使用编译时检测或迭代的（类型或值上的）简单计算，例如，布尔类型的 &&（见 24.4 节）或单位相加（见 28.7 节）。

［3］ 使用显式编译时检测的计算，例如，编译时 if（见 28.3 节）。

［4］ 使用编译时迭代（以递归的形式呈现；见 28.3.2 节）的计算

四个等级的顺序指出了复杂度的高低，暗示了任务的困难性、调试的困难性以及出现错误的可能性。

因此，元编程就是“元数据”和程序设计的组合：一个元程序就是一个编译时计算的代码，它生成运行时使用的类型或函数。注意，我没有说“模板元编程”，因为计算也可以使用函数 constexpr 完成。还要注意，你可以利用他人的元编程结果而不必真正自己进行元编程：调用一个背后是元程序的 constexpr 函数（见 28.2.2 节）或从一个模板类型函数抽取类型（见 28.2.4 节）本身并不是元编程，只是使用元程序而已。

泛型编程通常可以划归第一类“无计算”，但使用元编程技术支持泛型编程是完全可能的。当这样做时，就必须小心我们的接口说明要严谨定义并正确实现。一旦我们使用（元）编程作为接口的一部分，编程错误也可能悄然而至。而如果不使用编程，接口的含义就直接由语言规则定义。

泛型编程关注的焦点是接口说明，而元编程关注的则是编程——通常伴随着将类型作为值。

过度使用元编程会带来调试困难和过长的编译时间，从而导致实用性的降低。一如既往，我们必须遵循一些基本常识。有很多元编程的简单应用能产生高质量代码（更好的类型

安全、更低的内存占用以及更短的运行时间），同时也没有异常的编译时开销。很多标准库组件，例如 function（见 33.5.3 节）、thread（见 5.3.1 节和 42.2.2 节）和 tuple（34.2.4.2 节），是元编程技术简单应用的很好范例。

本章介绍基本的元编程技术，并展示元程序的基本构成。第 29 章将给出一个更复杂的示例。

28.2 类型函数

类型函数（type function）是这样一种函数：它接受至少一个类型参数或至少生成一个类型结果。例如，sizeof(T) 是一个内置类型函数，它返回给定类型参数 T 的对象大小（单位为 char 的数量；见 6.2.8 节）。

类型函数形式上不一定像常规函数一样，实际上大多数都不一样。例如，标准库的 is_polymorphic<T> 以模板参数的形式接受参数，而将名为 value 的成员作为返回结果：

```
if (is_polymorphic<int>::value) cout << "Big surprise!";
```

is_polymorphic 的成员 value 的值为 true 或 false。类似地，标准库类型转换也是类型函数，通过名为 type 的成员返回一个类型。例如：

```
enum class Axis : char { x, y, z };
enum flags { off, x=1, y=x<<1, z=x<<2, t=x<<3 };

typename std::underlying_type<Axis>::type x;    // x 是一个 char
typename std::underlying_type<Axis>::type y;    // y 可能是一个 int（见 8.4.2 节）
```

一个类型函数可以接受多个参数，返回多个结果值。例如：

```
template<typename T, int N>
struct Array_type {
    using type = T;
    static const int dim = N;
    // ...
};
```

这里的 Array_type 并不是一个标准库函数，甚至不是一个很有用的函数。我只是藉此展示如何编写一个简单的多参数、多返回值的类型函数。我们可以这样使用它：

```
using Array = Array_type<int,3>;

Array::type x;                  // x 是一个 int
constexpr int s = Array::dim;   // s 是 3
```

类型函数是编译时函数。即，类型函数只接受在编译时已知的参数（类型和值），并生成编译时可用的结果（类型和值）。

大多数类型函数接受至少一个类型参数，但也有一些很有用的类型函数并不是这样。例如，下面这个类型函数返回一个具有指定大小的整数类型：

```
template<int N>
struct Integer {
    using Error = void;
    using type = Select<N,Error,signed char,short,Error,int,Error,Error,Error,long>;
};

typename Integer<4>::type i4 = 8;   // 4 字节整数
typename Integer<1>::type i1 = 9;   // 1 字节整数
```

我将在 28.3.1.3 节中定义和介绍 Select。编写只接受值也只生成值的模板当然是可能的，但我不考虑这种类型函数。而且，表达这种编译时求值，constexpr 函数（见 12.1.6 节）通常是一种更好的方式。我可以用模板在编译时计算一个平方根，但既然能用 constexpr 函数更简洁地描述同一个算法（见 2.2.3 节、10.4 节和 28.3.2 节），我又为什么要用模板呢？

C++ 类型函数大多数都是模板，它们可以用类型和值完成非常通用的计算，是元编程的基础。例如，我们可能想要在栈中为小对象分配内存，而在自由存储空间中为大对象分配内存：

```
constexpr int on_stack_max = sizeof(std::string); // 我们希望在栈中分配的最大对象大小

template<typename T>
struct Obj_holder {
    using type = typename std::conditional<(sizeof(T)<=on_stack_max),
                                           Scoped<T>,        // 第一选择
                                           On_heap<T>        // 第二选择
                                          >::type;
};
```

标准库模板 conditional 是一个编译时选择器，可实现两种方案中的一个。如果它的第一个实参求值为 true，则结果（以成员 type 的方式给出）是第二个实参；否则，结果是第三个实参。28.3.1.1 节会展示如何实现 conditional。在本例中，如果对象 X 较小，则 Obj_holder<X> 的 type 被定义为 Scoped<X>；若 X 较大，则结果是 On_heap<X>。Obj_holder 可以这样使用：

```
void f()
{
    typename Obj_holder<double>::type v1;                 // 在栈中分配 double
    typename Obj_holder<array<double,200>>::type v2;      // 在自由存储中分配 array
    // ...
    *v1 = 7.7;      // Scoped 提供类指针的访问方式（* 和 []）
    v2[77] = 9.9;   // On_heap 提供类指针的访问方式（* 和 []）
    // ...
}
```

这个 Obj_holder 例子并非是假想的。例如，C++ 标准在其 function 类型（保存类似函数的实体，见 33.5.3 节）的定义中包含了如下注释："我们鼓励 C++ 编译器实现不对小的可调用对象使用动态内存分配，例如，当 f 的目标对象仅保存了一个对象指针或引用和一个成员函数指针时"（见 iso.20.8.11.2.1）。如果没有 Obj_holder 这样的模板作为支撑，我们很难遵循这条建议。

Scoped 和 On_heap 是如何实现的呢？它们的实现很普通，不涉及任何元编程，现在是：

```
template<typename T>
struct On_heap {
    On_heap() :p(new T) { }   // 分配
    ~On_heap() { delete p; }  // 释放

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

    On_heap(const On_heap&) = delete;          // 阻止拷贝
    On_heap operator=(const On_heap&) = delete;
private:
```

```
    T* p;       // 指向在自由存储空间中对象的指针
};

template<typename T>
struct Scoped {
    T& operator*() { return x; }
    T* operator->() { return &x; }

    Scoped(const Scoped&) = delete;              // 阻止拷贝
    Scoped operator=(const Scoped&) = delete;
private:
    T x;        // 对象本身
};
```

On_heap 和 Scoped 很好地展示了泛型编程和模板元编程是如何要求我们为一个通用构想（在本例中是对象分配的构想）的不同实现设计一致的接口的。

On_heap 和 Scoped 都既可以用作成员也可以用作局部变量。On_heap 总是将对象置于自由存储空间中，而 Scoped 则包含对象本身。

我们还可以实现 On_heap 和 Scoped 针对接受构造函数实参的类型的版本，28.6 节展示了如何来实现。

28.2.1 类型别名

注意，当我们使用 typename 和 ::type 来提取成员类型时，Obj_holder 的实现细节（类似 Int）是如何显现出来的。这个结果是语言的说明和使用方式所导致的，这是过去 15 年中程序员编写模板元程序的方式，也是 C++11 标准中所定义的方式。但我认为这种方式是不可忍受的。它时刻提醒我 C 语言那糟糕的往日岁月，那时每当使用用户自定义类型时都不得不加上 struct 关键字前缀。通过引入模板别名（见 23.6 节），我们可以隐藏 ::type 实现细节，并令一个类型函数看起来更像一个返回类型的函数（或更像一个类型）。例如：

```
template<typename T>
using Holder = typename Obj_holder<T>::type;

void f2()
{
    Holder<double> v1;
        // 在栈中分配 double
    Holder<array<double,200>> v2;     // 在自由存储中分配 array
    // ...
    *v1 = 7.7;      // Scoped 提供类指针的访问方式（* 和 []）
    v2[77] = 9.9;   // On_heap 提供类指针的访问方式（* 和 []）
    // ...
}
```

除非要解释一个实现或是解释 C++ 标准提供的特殊特性，否则我都会使用这种类型别名。当标准提供了一个类型函数（被称为“类型属性谓词”或“复合类型类别谓词”之类的东西），例如 conditional 时，我都会定义一个相应的类型别名（见 35.4.1 节）：

```
template<typename C, typename T, typename F>
using Conditional = typename std::conditional<C,T,F>::type;
```

请注意，很不幸的是，这些别名并非 C++ 标准的一部分。

28.2.1.1　何时不使用别名

有一种情况需要直接使用 ::type，而不是使用别名：当仅有一种候选可能是合法类型时，我们不应使用别名。先考虑一个类似的简单例子：

```
if (p) {
    p->f(7);
    // ...
}
```

当 p 的值为 nullptr 时，我们不能进入语句块。我们使用条件判断来检查 p 是否合法。类似地，我们可能希望检测一个类型是否合法。例如：

```
conditional<
  is_integral<T>::value,
  make_unsigned<T>,
  Error<T>
>::type
```

在本例中，我们检测 T 是否是一个整数类型（使用类型谓词 std::is_integral），若是，则返回其对应的无符号类型（使用类型函数 std::make_unsigned）。如果这条语句成功执行，则得到一个无符号类型；否则，我们需要处理 Error 指示符。

假如我们定义了 Make_unsigned<T> 来表示：

```
typename make_unsigned<T>::type
```

并试图将其用于一个非整数类型，例如 std::string，其实就是在尝试获取一个不存在的类型（make_unsigned<std::string>::type），结果就会得到一个编译错误。

在极少数不能采用别名方法隐藏的 ::type 情况中，我们可以回到显式的、面向实现的 ::type 风格。或者，我们可以引入一个类型函数 Delay，来将类型函数的求值延迟到使用时：

```
Conditional<
  is_integral<T>::value,
  Delay<Make_unsigned,T>,
  Error<T>
>
```

实现一个完美的 Delay 函数并不容易，但下面这个版本可适用于大多数情况：

```
template<template<typename...> class F, typename... Args>
using Delay = F<Args...>;
```

这个定义使用了一个模板模板参数（见 25.2.4 节）和可变参数模板（见 28.6 节）。

不管选择哪种方法避免不希望的实例化，都属于专家领域的内容，在涉及这些领域时我总是小心翼翼。

28.2.2　类型谓词

谓词是返回布尔值的函数。如果你想编写参数是类型的函数，显然你会希望询问有关参数类型的问题。例如，它是一个带符号类型吗？它是多态类型吗（即，它是否至少有一个虚函数）？它是从某个类型派生的吗？

很多这种问题的答案编译器都是了解的，并且通过一组标准库类型谓词（见 35.4.1 节）提供给了用户。例如：

```
template<typename T>
void copy(T* p, const T* q, int n)
{
    if (std::is_pod<T>::value)
        memcpy(p,q,n);          // 使用优化的内存拷贝
    else
        for (int i=0; i!=n; ++i)
            p[i] = q[i];        // 逐个值拷贝
}
```

在本例中，我们尝试用标准库函数 memcpy()（一般认为是最优的）来优化拷贝操作，前提是对象可以作为“普通旧数据”（POD；见 8.2.6 节）来处理。如果不满足此条件，我们逐个拷贝对象（可能使用的是对象的拷贝构造函数）。我们用标准库类型谓词 is_pod 来判断模板实参类型是不是一个 POD，结果是通过成员 value 给出的。这种标准库习惯用法与类型函数通过成员 type 给出结果的方式是相似的。

谓词 std::is_pod 是标准库中众多谓词之一（见 35.4.1 节）。由于 POD 判断规则较为复杂，is_pod 最有可能是一个编译器特性而不是实现为标准库 C++ 代码。

类似 ::type，使用 ::value 值会导致代码冗长且背离了屏蔽实现细节的习惯表示方式：一个返回 bool 值的函数应该通过 () 来调用。

```
template<typename T>
void copy(T* p, const T* q, int n)
{
    if (is_pod<T>())
    // ...
}
```

幸运的是，对所有标准库类型谓词，C++ 标准都支持这种使用方式。但不幸的是，出于语言技术方面的原因，这种方式不能用于模板实参。例如：

```
template<typename T>
void do_something()
{
    Conditional<is_pod<T>(),On_heap<T>,Scoped<Y>> x;   // 错误：is_pod<T>() 是一个类型
    // ...
}
```

这里，is_pod<T> 被解释为一个无参、返回 is_pod<T> 的函数类型（见 iso.14.3 [2]）。

对此问题我的解决方案是添加函数，提供在所有场景下都适用的习惯表示方式：

```
template<typename T>
constexpr bool Is_pod()
{
    return std::is_pod<T>::value;
}
```

我将这些类型函数的首字母大写，来避免与标准库版本的冲突。而且，我将它们置于一个独立的名字空间中（Estd）。

我们也可以自定义新的类型谓词。例如：

```
template<typename T>
constexpr bool Is_big()
{
    return 100<sizeof(T);
}
```

我们可以像下面这样使用这个（非常粗糙的）“大类型”的概念：

```
template<typename T>
using Obj_holder = Conditional<(Is_big<T>()), Scoped<T>, On_heap<T>>;
```

我们极少需要定义直接反映类型基本属性的谓词，因为标准库已经提供了很多这种谓词，例如 is_integral、is_pointer、is_empty、is_polymorphic 和 is_move_assignable（见 35.4.1 节）。当我们必须定义这种谓词时，有一些非常强大的技术可供使用。例如，我们可以定义一个类型函数来判断一个类是否有一个具有指定名字和类型的成员（见 28.4.4 节）。

当然，多参数的类型谓词也很有用。特别是，这提供了一种表达两种类型间关系的方法，例如 is_same、is_base_of 和 is_convertible，这几个谓词也都来自标准库。

对所有这些 is_* 函数，我用 is_* constexpr 函数来支持常用的 () 调用语法。

28.2.3 选择函数

函数对象仍然是某种类型的对象，因此选择类型和值的技术可用于选择函数。例如：

```
struct X { // 输出 X
    void operator()(int x) { cout <<"X" << x << "\n"; }
    // ...
};

struct Y { // 输出 Y
    void operator()(int y) { cout <<"Y" << y << "\n"; }
    // ...
};

void f()
{
    Conditional<(sizeof(int)>4),X,Y>{}(7);    // 创建一个 X 或一个 Y 并调用它

    using Z = Conditional<(Is_polymorphic<X>()),X,Y>;
    Z zz;      // 创建一个 X 或一个 Y
    zz(7);     // 调用一个 X 或一个 Y
}
```

如上所示，我们可以立即使用选出的函数对象类型，也可以“记住”它以备后用。使用包含求值成员函数的类，是在模板元编程中进行计算最通用也最灵活的机制。

Conditional 是一种编译时编程的机制，这也意味着条件必须是一个常量表达式。注意在 sizeof(int)>4 外围的括号；如果没有这对括号，我们将得到一个语法错误，因为编译器会将 > 解释为模板实参列表的结束标记。出于这个（以及其他）原因，我倾向于使用 <（小于号）而不是 >（大于号）。而且，我有时会为条件加上括号，以提高可读性。

28.2.4 萃取

标准库严重依赖于萃取（trait）技术。萃取被用来关联属性与类型。例如，一个迭代器的属性由其 iterator_traits 定义（见 33.1.3 节）：

```
template<typename Iterator>
struct iterator_traits {
    using difference_type = typename Iterator::difference_type;
    using value_type = typename Iterator::value_type;
    using pointer = typename Iterator::pointer;
```

```
    using reference = typename Iterator::reference;
    using iterator_category = typename Iterator::iterator_category;
};
```

你可以将萃取理解为返回很多结果的类型函数或是一组类型函数。

标准库提供了allocator_traits（见34.4.2节）、char_traits（见36.2.2节）、iterator_traits（见33.1.3节）、regex_traits（见37.5节）、pointer_traits（见34.4.3节）。标准库还提供了time_traits（见35.2.4节）和type_traits（见35.4.1节），它们其实是两个简单类型函数，容易给使用者造成困扰。

给定一个指针的iterator_traits，我们就可以讨论指针的value_type和difference_type，即使指针没有成员也是如此：

```
template<typename Iter>
Iter search(Iter p, Iter q, typename iterator_traits<Iter>::value_type val)
{
    typename iterator_traits<Iter>::difference_type m = q-p;
    // ...
}
```

这是一种非常有用也非常强大的技术，但：

- 它太冗长。
- 它通常要将一些弱相关的类型函数捆绑在一起。
- 它将实现细节暴露给用户。

而且，程序员有时为了“有备无患”会定义类型别名，这会导致不必要的复杂性。因此，我更倾向于使用简单的类型函数：

```
template<typename T>
    using Value_type = typename std::iterator_trait<T>::value_type;

template<typename T>
    using Difference_type = typename std::iterator_trait<T>::difference_type;
template<typename T>
    using Iterator_category= typename std::iterator_trait<T>::iterator_category;
```

这样，search的例子就可以变得更简洁：

```
template<typename Iter>
Iter search(Iter p, Iter q, Value_type<Iter> val)
{
    Difference_type<Iter> m = q-p;
    // ...
}
```

我觉得萃取现在被过度使用了。考虑如何在不了解萃取或任何类型函数的情况下编写这个例子：

```
template<typename Iter, typename Val>
Iter search(Iter p, Iter q, Val val)
{
    auto x = *p;                          // 如果我们不需要命名 p 的类型
    auto m = q-p;                         // 如果我们不需要命名 q-p 的类型

    using value_type = decltype(*p);      // 如果我们需要命名 p 的类型
    using difference_type = decltype(q-p); // 如果我们需要命名 q-p 的类型

    // ...
}
```

当然，decltype() 也是一种类型函数，因此我所做的只是去掉了用户自定义的和标准库中的类型函数。而且，auto 和 decltype 是 C++11 的新特性，因此旧代码是不可能这样编写的。

我们需要用萃取（或等价的诸如 decltype() 的特性）将一个类型与另一个类型关联起来，例如将 value_type 与 T* 关联。就此目的而言，萃取（或等价特性）是泛型编程或元编程中一种必不可少的非侵入式添加类型名的方法。当我们用萃取简单地为某个已有很好名字的实体提供一个新名字时，例如为 value_type* 命名 pointer，为 value_type& 命名 reference，其功用就不再那么清晰，潜在混乱的可能性也会增大。因此，不要只为“有备无患”而盲目地为所有实体定义萃取。

28.3 控制结构

为了在编译时实现通用计算，我们需要选择和递归机制。

28.3.1 选择

在前面几节中，除了使用普通常量表达式完成简单计算外（见 10.4 节），我还使用了：

- Conditional：在两种类型中进行选择的方法（std::onditional 的别名）
- Select：在多种类型中进行选择的方法（定义在 28.3.1.3 节中）

这两个类型函数返回类型。如果你希望在多个值中进行选择，?: 就足够了；Conditional 和 Select 是用来选择类型的。它们并非 if 和 switch 简单的编译时对应版本，虽然在用来选择函数对象时看起来有些像（见 3.4.3 节和 19.2.2 节）。

28.3.1.1 在两个类型中选择

如 28.2 节所示，Conditional 的实现异常简单。conditional 模板是标准库的一部分（定义在 <type_traits> 中），因此我们并不需要实现它，但其实现展示了一种重要的技术：

```
template<bool C, typename T, typename F>   // 通用模板
struct conditional {
    using type = T;
};

template<typename T, typename F>           // false 的特例化版本
struct conditional<false,T,F> {
    using type = F;
};
```

主模板（见 25.3.1.1 节）简单地将其 type 定义为 T（条件之后的第一个模板参数）。如果条件不为 true，就会选择 false 的特例化版本，type 将被定义为 F。例如：

```
typename conditional<(std::is_polymorphic<T>::value),X,Y>::type z;
```

显然，语法上还有改进余地（见 28.2.2 节），但基础逻辑是优美的。

特例化被用来分离一般情况和（一个或多个）特殊情况（见 25.3 节）。在本例中，主模板恰好实现一半功能，这个比例会从零（每个正确的情况都被一个特例化版本处理；见 25.3.3.1 节）到百分之百之间变化，但本章最后一个示例除外（见 28.5 节）。这种选择完全在编译时进行，不会消耗哪怕一个字节或一个时钟周期的运行时开销。

为了改进语法，我引入一个别名：

```
template<bool B, typename T, typename F>
using Conditional = typename std::conditional<B,T,F>::type;
```

基于此定义，我们可以编写代码如下：

```
Conditional<(is_polymorphic<T>()),X,Y> z;
```

我认为这是一个重要的改进。

28.3.1.2 编译时与运行时

来考察下面的代码：

```
Conditional<(std::is_polymorphic<T>::value),X,Y> z;
```

如果是第一次见到这样的代码，人们很容易会想“我们为什么不简单地用 if 语句呢？”考虑需要在两个类型中选择的情形，例如 Square 和 Cube：

```
struct Square {
    constexpr int operator()(int i) { return i*i; }
};

struct Cube {
    constexpr int operator()(int i) { return i*i*i; }
};
```

我们可能尝试熟悉的 if 语句：

```
if (My_cond<T>())
    using Type = Square;    // 错误：在 if 语句分支中声明
else
    using Type = Cube;      // 错误：在 if 语句分支中声明

Type x;   // 错误：Type 不在作用域中
```

声明不能作为 if 语句分支中的唯一语句（见 6.3.4 节和 9.4.1 节），因而即使 My_cond<T>() 是在编译时计算的，这段代码也不能正常工作。因此，普通 if 语句只对普通表达式有用，而对类型选择无效。

让我们尝试一个不涉及变量定义的例子：

```
Conditional<My_cond<T>(),Square,Cube>{}(99);   // 调用 Square{}(99) 或 Cube{}(99)
```

即，选择一个类型，构造一个该类型的默认对象，然后调用它。这样做是正确的。使用“常规控制结构”，这条语句可改写为：

```
((My_cond<T>())?Square:Cube){}(99);
```

这条语句不能正确实现目的，因为 Square{}(99) 和 Cube{}(99) 不产生类型，也不是相容类型的值，因此不能在条件表达式中进行比较（见 11.1.3 节）。我们可以尝试：

```
(My_cond<T>()?Square{}:Cube{})(99);   // 错误：?: 不允许运算对象不相容
```

不幸的是，这条语句面临同样问题：Square{} 和 Cube{} 是不相容的类型，因此不能用在 ?: 表达式中。而在元编程中通常不能有类型相容的限制，因为我们需要在并非显式相关的类型中进行选择。

最后，下面的语句是正确的：

```
My_cond<T>()?Square{}(99):Cube{}(99);
```

但它并不比最初的形式更易读：

```
Conditional<My_cond<T>(),Square,Cube>{}(99);
```

28.3.1.3　在多个类型中选择

多选一与二选一非常相似。下面的类型函数返回它的第 N 个实参类型：

```
class Nil {};

template<int I, typename T1 =Nil, typename T2 =Nil, typename T3 =Nil, typename T4 =Nil>
struct select;

template<int I, typename T1 =Nil, typename T2 =Nil, typename T3 =Nil, typename T4 =Nil>
using Select = typename select<I,T1,T2,T3,T4>::type;

// 针对 0-3 的特例化：

template<typename T1, typename T2, typename T3, typename T4>
struct select<0,T1,T2,T3,T4> { using type = T1; };   // N==0 的特例化

template<typename T1, typename T2, typename T3, typename T4>
struct select<1,T1,T2,T3,T4> { using type = T2; };   // N==1 的特例化

template<typename T1, typename T2, typename T3, typename T4>
struct select<2,T1,T2,T3,T4> { using type = T3; };   // N==2 的特例化

template<typename T1, typename T2, typename T3, typename T4>
struct select<3,T1,T2,T3,T4> { using type = T4; };   // N==3 的特例化
```

程序中不会用到 select 的通用版本，所以我没有给出其定义。我选择了从 0 开始的编号方式，以匹配 C++ 的风格。这种多选一的技术有很好的通用性：这些特例化版本呈现了模板参数的所有方面。我们可能需要超过四种选择，这可以通过可变参数模板来解决（见 28.6 节）。如想选择其他可能，可使用主（通用）模板。例如：

```
Select<5,int,double,char> x;
```

在本例中，这条语句会引起一个编译错误，因为通用版本的 Select 未定义。

Select 的一个可能的实际应用是选择一个函数的类型，它返回一个元组中的第 N 个元素：

```
template<int N, typename T1, typename T2, typename T3, typename T4>
Select<N,T1,T2,T3,T4> get(Tuple<T1,T2,T3,T4>& t);      // 见 28.5.2 节

auto x = get<2>(t);  // 假定 t 是一个 Tuple
```

在本例中，x 的类型是名为 t 的 Tuple 的第 3 个元素的类型 T3。元组中的元素是从 0 开始编号的。

使用可变参数模板（见 28.6 节），我们可以提供一个更简单也更通用的 select：

```
template<unsigned N, typename... Cases>    // 一般情况：不会被实例化
struct select;

template<unsigned N, typename T, typename... Cases>
struct select<N,T,Cases...> :select<N-1,Cases...> {
};
template<typename T, typename... Cases>    // 最终情况：N==0
struct select<0,T,Cases...> {
    using type = T;
};
```

```
template<unsigned N, typename... Cases>
using Select = typename select<N,Cases...>::type;
```

28.3.2 迭代和递归

下面这个阶乘函数模板可以很好地展示在编译时计算一个值的基本技术：

```
template<int N>
constexpr int fac()
{
    return N*fac<N-1>();
}

template<>
constexpr int fac<1>()
{
    return 1;
}

constexpr int x5 = fac<5>();
```

在本例中，阶乘是用递归而不是循环实现的。由于我们在编译时不能使用变量（见 10.4 节），因此这种方式是合理的。一般而言，我们使用递归实现编译时迭代。

注意，我们并未使用条件语句：上面代码中并没有 N==1 或 N<2 这样的检测。取而代之，当 fac() 调用选择了 N==1 的特例化版本时，递归停止。在模板元编程中（如同函数式编程），处理一系列值的常用方式是进行递归调用，直至到达一个对应终止条件的特例化版本。

对于本例，我们还可以使用一种更常见的方式完成阶乘计算：

```
constexpr int fac(int i)
{
    return (i<2)?1:fac(i-1);
}

constexpr int x6 = fac(6);
```

我觉得这种方式比函数模板的表达方式更为清晰，但每个人的偏好不同，而且对某些算法而言，将一般情况和终止情况分开来的表达方式更好。对编译器来说，非模板版本处理起来稍微容易一些。而两种方式的运行时性能当然是一样的。

constexpr 版本既可用于编译时求值，也可用于运行时求值。模板（元编程）版本则只能用于编译时。

28.3.2.1 使用类的递归

如果迭代过程中涉及更复杂的状态或更精细的参数化，我们可以使用类。例如，可以改写阶乘程序：

```
template<int N>
struct Fac {
    static const int value = N*Fac<N-1>::value;
};

template<>
struct Fac<1> {
    static const int value = 1;
};
```

```
constexpr int x7 = Fac<7>::value;
```

一个更实用的例子可在 28.5.2 节中找到。

28.3.3　何时使用元编程

使用上述控制结构，你能够在编译时进行任何计算（在编译限制许可范围内）。但仍有一个问题：你为什么要使用这些技术？当这些技术能比其他技术产生更整洁、性能更好且更易维护的代码时，我们就应该使用这些技术。元编程最明显的限制是：依赖于复杂模板的代码很难读也很难调试。大量使用复杂模板也会影响编译时间。如果你在理解一段使用了复杂实例化模式的代码时遇到了很大的困难，那么编译器理解它也会很困难。更糟的是，维护代码的程序员也很难理解它。

模板元编程吸引了很多聪明人：

- 部分是因为元编程允许我们表达一些不能简单实现运行时良好性能和类型安全的东西。若能显著改善这两点并能得到可维护性较好的代码，将是使用元编程的很好的（有时甚至是非常令人信服的）理由。
- 部分是因为元编程令我们能卖弄聪明，这显然是应该避免的。

那么你怎么知道是否过度使用了元编程呢？一个警告信号是你有冲动用宏（见 12.6 节）来隐藏一些过于丑陋难以直接使用的“细节”。考虑下面的例子：

```
#define IF(c,x,y) typename std::conditional<(c),x,y>::type
```

这是否走得太远了呢？它允许我们这样编写代码

```
IF(cond,Cube,Square) z;
```

而不必这样写

```
typename std::conditional<(cond),Cube,Square>::type z;
```

在本例中我使用了极短的名字 IF 和很长的形式 std::conditional，使问题的讨论有了一些偏向性。类似地，一个更复杂的条件几乎总是意味着更长的表示形式。两种形式的根本差别在于，若使用后者，我不得不用 typename 和 ::type 以便使用标准库表示方式，但这会暴露模板的实现技术。我希望隐藏这些细节，而宏能帮我做到。但是，如果需要很多人合作编程，而程序规模变得很大，则代码冗长总比表示上的不一致要好。

反对 IF 宏的另一个重要理由是其名字有误导性：conditional 并非是传统 if 的“插入式替换”。::type 体现了一个重要差异：conditional 在类型间进行选择；它并不直接改变控制流。有时它被用来选择函数，因此也能体现出计算上的分支；但有时并不是这样。IF 宏隐藏了其函数的一个重要方面。对其他很多“感性的”宏也有类似的反对理由：这些宏的命名是某个程序员根据自己对代码使用的理解而定的，而未反映基本功能。

在本例中，实现细节所导致的冗长问题，以及糟糕的命名问题，都可以很容易地使用类型别名解决（Conditional；见 28.2.1 节）。一般来说，我们应该努力寻找方法来清理呈献给用户的语法，但不要因此而发明一种私有语言。应优先选择系统化的技术，如特例化以及别名，而不是宏。为了实现编译时计算，应该优先选择 constexpr 函数而不是模板，并尽可能地隐藏 constexpr 函数中模板元编程的实现细节（见 28.2.2 节）。

或者，我们可以考察待完成工作的根本复杂性：

[1] 它需要语式的条件判断吗?

[2] 它需要递归吗?

[3] 我们能为模板实参写出概念(见24.3节)吗?

如果问题[1]或[2]的答案为"是",或者问题[3]的答案为"否",我们就应考虑是否存在维护问题了。也许可以采用某种形式的封装?记住,若实例化失败,则模板实现的复杂性就会被用户所见("泄露")。而且,很多程序员确实会查看头文件,这样,元程序的所有细节都会立刻暴露。

28.4 条件定义:Enable_if

当我们编写一个模板时,有时希望提供一个操作为某些模板实参所用,而不为其他实参所用。例如:

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*();     // 返回指向整个对象的引用
    T* operator->();    // 选择一个成员(仅用于类)
    // ...
};
```

若T是一个类,我们应该提供operator->(),但若T是一个内置类型,我们根本无法这样做(至少在通常语义下不行)。因此,我们希望有一种语言机制能表达"若此类型有此属性,则定义下面的内容。"显然,我们可以尝试这样:

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*();                          // 返回指向整个对象的引用
    if (Is_class<T>()) T* operator->();      // 语法错误
    // ...
};
```

但是,这段代码不正确。C++的if不能根据一个一般条件来选择定义。但是,类似Conditional和Select(见28.3.1节),有一种方法可以实现这个目的。我们可以编写一个有些古怪的类型函数,使operator->()的定义是有条件的。标准库提供了enable_if(在<type_traits>中)。Smart_pointer例子如下:

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*();                                  // 返回指向整个对象的引用
    Enable_if<Is_class<T>(),T>* operator->();        // 选择一个成员(仅用于类)
    // ...
};
```

与往常一样,我使用类型别名和constexpr函数来简化表示:

```
template<bool B, typename T>
using Enable_if = typename std::enable_if<B,T>::type;

template<typename T> bool Is_class()
{
    return std::is_class<T>::value;
}
```

如果 Enable_if 的条件求值为 true，其结果为第 2 个参数（在本例中是 T）。如果 Enable_if 的条件求值为 false，则其所在的函数声明会被完全忽略。在本例中，如果 T 是一个类，我们就得到一个返回 T* 的 operator->() 定义，否则就不会声明任何函数。

给定了使用 Enable_if 的 Smart_pointer 定义，我们就可这样编写代码：

```
void f(Smart_pointer<double> p, Smart_pointer<complex<double>> q)
{
    auto d0 = *p;          // 正确
    auto c0 = *q;          // 正确
    auto d1 = q->real();   // 正确
    auto d2 = p->real();   // 错误：p 不指向一个类对象
    // ...
}
```

你可能认为 Smart_pointer 和 operator->() 有些异乎寻常，但提供（定义）有条件的操作其实非常常见。标准库中有很多条件定义的例子，例如 Alloc::size_type（见 34.4.2 节）和 pair，若 pair 的两个元素都是可移动的则 pair 也是可移动的（见 34.2.4.1 节）。C++ 语言本身也有这样的例子，例如 -> 只能用于指向类对象的指针（见 8.2 节）。

在本例中，用 Enable_if 精心设计的 operator->() 声明改变了代码的错误类型，例如对 p->real()：

- 若 operator->() 不是条件声明的，我们在实例化时会得到一个 Smart_pointer<double> :: operator->() 定义错误“-> 用于一个非类指针”。
- 如果我们用 Enable_if 条件声明 operator->()，若在一个 Smart_pointer<double> 上使用 ->，则会在使用 Smart_pointer<double>::operator->() 时得到一个“ Smart_pointer<double>::operator->() 未定义”的错误。

无论哪种情况，都只有在对一个 Smart_pointer<T> 使用 -> 且 T 不是类时才会产生错误。

我们已经将错误检测和报告从 Smart_pointer<T>::operator->() 的实现处移到了其声明处。依赖于编译器具体实现，特别是依赖于错误发生于嵌套模板实例化的层次有多深，这一改变会有很大不同。一般而言，精确说明模板以便尽早检测到错误比寄希望于捕获到实例化错误更好。从这层意义上说，我们可以将 Enable_if 看作概念思想的变体（见 24.3 节）：它允许对模板要求进行更精确的说明。

28.4.1 使用 Enable_if

对很多使用场景而言，enable_if 的功能是非常理想的，但我们要使用的表示形式却常常很尴尬。例如：

```
Enable_if<Is_class<T>(),T>* operator->();
```

实现细节暴露无遗。但是，我们真正要表达的内容非常接近于下面的理想的极简表示：

```
declare_if (Is_class<T>()) T* operator->();    // 非 C++
```

但是，C++ 并未提供 declare_if 结构用于选择声明。

用 Enable_if 修饰返回类型，将其置于你能看到的最显眼的位置，也是它逻辑上应该在的位置，因为它会影响整个声明（而不仅仅是返回类型）。但是，某些声明没有返回类型。考虑 vector 的两个构造函数：

```
template<typename T>
```

```
class vector<T> {
public:
    vector(size_t n, const T& val); // n 个类型为 T 的元素，赋初值为 val

    template<typename Iter>
        vector(Iter b, Iter e); // 用 [b:e) 中的值进行初始化
    // ...
};
```

这段代码看起来完全无害，但接受元素数目参数的构造函数通常会有很大的破坏性。考虑下面的代码：

```
vector<int> v(10,20);
```

它是要初始化 10 个值为 20 的元素，还是用范围［10:20］中的元素进行初始化呢？C++ 标准要求选择前者，但上面的代码会选择后者，因为选择第一个构造函数的话，需要进行 int 到 size_t 的类型转换，而一对 int 能完美匹配模板构造函数。问题在于我“忘了”告诉编译器类型 Iter 必须是迭代器，不过这不难办到：

```
template<typename T>
class vector<T> {
public:
    vector(size_t n, const T& val);      // n 个类型为 T 的元素，赋初值为 val

    template<typename Iter, typename =Enable_if<Input_iterator<Iter>(),Iter>>
        vector(Iter b, Iter e);          // 用 [b:e) 中的值进行初始化
    // ...
};
```

这段代码中，（未用到的）默认模板实参会被实例化，因为我们当然不能推断未用的模板参数。这意味着只有当 Iter 为 Input_iterator 时（见 24.4.4. 节），声明 vector(Iter, Iter) 才会成功。

我引入 Enable_if 作为默认模板实参是因为这种方案最通用。它可以用于没有实参且（或）没有返回类型的模板。但是，在本例中，我们也可以将它用于构造函数的实参类型：

```
template<typename T>
class vector<T> {
public:
    vector(size_t n, const T& val);      // n 个类型为 T 的元素，赋初值为 val

    template<typename Iter>
        vector(Enable_if<Input_iterator<Iter>(),Iter>> b, Iter e);      // 用 [b:e) 中的值进行初始化
    // ...
};
```

这种 Enable_if 技术只适用于模板函数（包括类模板和特例化版本的成员函数）。Enable_if 的实现和使用依赖于函数模板重载规则细节（见 23.5.3.2 节）。因此，它不能用来控制类、变量或非模板函数的声明。例如：

```
Enable_if<(version2_2_3<config),M_struct>* make_default() // 错误：不是模板
{
    return new Mystruct{};
}

template<typename T>
```

```
void f(const T& x)
{
    Enable_if<(20<sizeof<T>),T> tmp = x;              // 错误：tmp 不是函数
    Enable_if<!(20<sizeof<T>),T&> tmp = *new T(x);    // 错误：tmp 不是函数
    // ...
}
```

对 tmp 来说，使用 Holder（见 28.2 节）几乎肯定可以得到更整洁的代码；如果你已设法在自由存储空间上构造一个对象，那你会如何 delete 它呢？

28.4.2 实现 Enable_if

Enable_if 的实现相当简单：

```
template<bool B, typename T = void>
struct std::enable_if {
    typedef T type;          返回指向整个对象的引用
};                           选择一个成员（仅用于类）

template<typename T>
struct std::enable_if<false, T> {};          // 若 B==false 则没有 ::type

template<bool B, typename T = void>
using Enable_if = typename std::enable_if<B,T>::type;
```

注意，我们可以忽略类型实参，将 void 作为默认实参。

这一简单的声明是如何成为一个很有用的基础结构的呢？在 23.5.3.2 节中对此给出了语言技术上的解释。

28.4.3 Enable_if 与概念

我们可以将 Enable_if 用于很多谓词，包括很多类型属性的检测（见 28.3.1.1 节）。概念是最通用也最有用的一些谓词。理想情况下，我们希望能依据概念进行重载，但 C++ 语言缺乏对概念的支持，我们所能做到的最好程度也就是使用 Enable_if 实现依据约束的选择了。例如：

```
template<typename T>
Enable_if<Ordered<T>()> fct(T*,T*);      // 优化实现

template<typename T>
Enable_if<!Ordered<T>()> fct(T*,T*);     // 非优化实现
```

注意 Enable_if 的默认结果为 void，因此 fct() 是一个 void 函数。我不确定使用这一默认类型是否能提高代码可读性，但我们可以像下面这样使用 fct()：

```
void f(vector<int>& vi, vector<complex<int>>& vc)
{
    if (vi.size()==0 || vc.size()==0) throw runtime_error("bad fct arg");
    fct(&vi.front(),&vi.back());          // 调用优化版本
    fct(&vc.front(),&vc.back());          // 调用非优化版本
}
```

两个调用的解析如注释所示，因为我们可以对 int 使用 <，但对 complex<int> 不能使用 <。如果我们不提供类型实参，Enable_if 会被解析为 void。

28.4.4 更多 Enable_if 例子

当使用 Enable_if 时，我们或早或晚都会询问一个类是否有指定名字和相应类型的成员。对很多标准操作来说，如构造函数和赋值运算符，标准库提供了相应的类型属性谓词，如 is_copy_assignable 和 is_default_constructible（见 35.4.1 节）。我们也可以构造新的谓词。考虑问题“若 x 的类型为 X，我是否能调用 f(x)？”我们可以定义 has_f 来回答这个问题，这其中能展示一些有用的技术以及很多模板元编程库（包括标准库的一部分）内部的骨架/样板代码。首先，我们定义通用的类以及表示可选方案的特例化版本：

```
struct substitution_failure { }; // 表示声明失败

template<typename T>
struct substitution_succeeded : std::true_type
{};

template<>
struct substitution_succeeded<substitution_failure> : std::false_type
{};
```

在本例中，substitution_failure 用来表示替换失败（见 23.5.3.2 节）。我们会从 std::true_type 派生一个类，但实参类型为 substitution_failure 的情况除外。显然，std::true_type 和 std::false_type 是分别表示值 true 和 false 的类型：

```
std::true_type::value == true
std::false_type::value == false
```

我们用 substitution_succeeded 来定义真正想要的类型函数。例如，我们可能想要一个函数 f，能用来调用 f(x)。为此，我们可以定义 has_f：

```
template<typename T>
struct has_f
   : substitution_succeeded<typename get_f_result<T>::type>
{};
```

这样，如果 get_f_result<T> 生成一个恰当的类型（应该是调用 f 的返回类型），则 has_f::value 为 true_type::value，即 true。若 get_f_result<T> 编译失败，它会返回 substitution_failure，因而 has_f::value 为 false。

到目前为止，一切都好，但当 f(x) 对类型为 X 的值 x 编译失败时，我们如何让 get_f_result<T> 返回 substitution_failure 呢？下面这个足够自白的定义可实现这一目的：

```
template<typename T>
struct get_f_result {
private:
    template<typename X>
        static auto check(X const& x) -> decltype(f(x));     // 可以调用 f(x)
    static substitution_failure check(...);                  // 不能调用 f(x)
public:
    using type = decltype(check(std::declval<T>()));
};
```

我们简单地声明了一个函数 check，使得 check(x) 的返回类型与 f(x) 一样。显然，只有当我们能调用 f(x) 时，check 的声明才能编译通过。如果我们不能调用 f(x)，check 的声明就会失败。在本例中，由于替换失败并不是一个错误（SFINAE；见 23.5.3.2 节），我们会得到 check() 的第二个定义，它的返回类型为 substitution_failure。当然，如果函数 f 本身就声

明为返回 substitution_failure，这个精心设计的小花招就失效了。

注意，decltype() 不对其运算对象进行求值。

在本例中，我们尝试将看起来像类型错误的东西转换为值 false。如果 C++ 语言能提供一个（内置）原语操作来实现这种转换，事情就简单多了。例如：

```
is_valid(f(x));     // f(x) 能编译通过吗？
```

但是，一种语言不可能将所有功能都纳入其中，作为其原语提供给程序员。有了上面的骨架代码后，我们只需为其提供常规语法：

```
template<typename T>
constexpr bool Has_f()
{
    return has_f<T>::value;
}
```

现在就可以这样使用它了：

```
template<typename T>
class X {
    // ...
    Enable_if<Has_f<T>()> use_f(const T&)
    {
        // ...
        f(t);
        // ...
    }
    // ...
};
```

X<T> 具有成员 use_f() 当且仅当对类型为 T 的 t 值可以调用 f(t)。

注意我们不能像下面这样简单使用 Has_f：

```
if (Has_f<decltype(t)>()) f(t);
```

即使 Has_f<decltype(t) 返回 false，编译器也会对 f(t) 调用进行类型检查（而且会失败）。

掌握了 Has_f 定义所使用的技术后，我们就可以为能想到的任何操作或成员 foo 定义 Has_foo 了。对每个 foo，骨架代码只有 14 行。对不同 foo 代码可能会有重复，但这不会给编程工作带来困难。

这意味着借助 Enable_if<>，我们可以依据几乎任何针对实参类型的逻辑标准来选择重载模板。例如，我们可以定义一个 Has_not_equals() 类型函数来检查类型是否定义了运算符 !=，并像下面这样使用它：

```
template<typename Iter, typename Val>
Enable_if<Has_not_equals<Iter>(),Iter> find(Iter first, Iter last, Val v)
{
    while (first!=last && !(*first==v))
        ++first;
    return first;
}
template<typename Iter, typename Val>
Enable_if<!Has_not_equals<Iter>(),Iter> find(Iter first, Iter last, Val v)
{
    while (!(first==last) && !(*first==v))
        ++first;
```

```
    return first;
}
```

这种特别的重载很容易变得混乱且不可控。例如，尝试添加一个新的版本，在可能的情况下使用 != 进行值的比较（即，*first!=v 而不是 !(*first==v)）。因此，我建议，尽可能优先使用更结构化的标准重载规则（见 12.3.1 节）和特例化规则（见 25.3 节）。例如：

```
template<typename T>
auto operator!=(const T& a, const T& b) -> decltype(!(a==b))
{
    return !(a==b);
}
```

这些规则确保当类型 T 已经定义了另一个专用的 != （作为一个模板函数或一个非模板函数）时，这个定义不会被实例化。我使用 decltype() 一是为了展示一般情况下如何从一个已定义的运算符派生出返回类型，二是为了处理很少见的情况——!= 返回非 bool 类型的值。

类似地，给定一个 <，我们就可以有条件地定义 >、<=、>=，等等。

28.5 一个编译时列表：Tuple

在本节中，我将通过一个简单但实际的例子展示元编程基本技术。我将定义一个 Tuple，它具有一个访问操作和一个输出操作。类似本节方法定义的 Tuple 在工业界已经使用超过 10 年了。我将在 28.6.4 节和 34.2.4.2 节中介绍更精致也更通用的 std::tuple。

我们的设计目标是允许像下面这样使用 Tuple：

```
Tuple<double, int, char> x {1.1, 42, 'a'};
cout << x << "\n";
cout << get<1>(x) << "\n";
```

输出结果应该是：

```
{ 1.1, 42, 'a'};
42
```

Tuple 的定义其实很简单：

```
template<typename T1=Nil, typename T2=Nil, typename T3=Nil, typename T4=Nil>
struct Tuple : Tuple<T2, T3, T4> {    // 布局：{T2,T3,T4} 在 T1 之前
    T1 x;

    using Base = Tuple<T2, T3, T4>;
    Base* base() { return static_cast<Base*>(this); }
    const Base* base() const { return static_cast<const Base*>(this); }

    Tuple(const T1& t1, const T2& t2, const T3& t3, const T4& t4) :Base(t2,t3,t4), x(t1) { }
};
```

因此，一个四个元素的 Tuple（通常被称为四元组，4-tuple）就是一个三个元素的 Tuple（三元组，3-tuple）后跟第四个元素。

我们用接受四个值（可能是四个不同类型）的构造函数来构造一个四元组 Tuple。它用后三个元素（尾）初始化基础的三元组，用第一个元素（头）初始化其成员 x。

Tuple 尾（Tuple 的基类）的处理在 Tuple 实现中是重要且反复出现的部分。因此，我定义了一个别名 Base 和一对成员函数 base() 来简化基类 / 尾部的处理。

显然，这个定义只能处理恰有四个元素的元组。而且，它将大部分工作交给三元组处

理。少于四个元素的元组被定义为特例化版本：

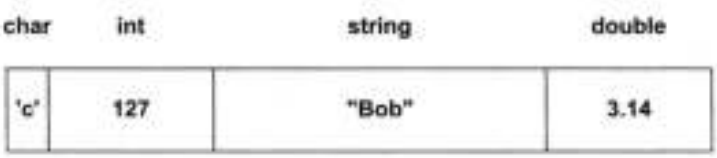
```
template<>
struct Tuple<> { Tuple() {} };                    // 零元组

template<typename T1>
struct Tuple<T1> : Tuple<> {                      // 一元组
    T1 x;

    using Base = Tuple<>;
    Base* base() { return static_cast<Base*>(this); }
    const Base* base() const { return static_cast<const Base*>(this); }

    Tuple(const T1& t1) :Base{}, x{t1} { }
};

template<typename T1, typename T2>
struct Tuple<T1, T2> : Tuple<T2> {                // 二元组，布局：T2 在 T1 之前
    T1 x;

    using Base = Tuple<T2>;
    Base* base() { return static_cast<Base*>(this); }
    const Base* base() const { return static_cast<const Base*>(this); }

    Tuple(const T1& t1, const T2& t2) :Base{t2}, x{t1} { }
};

template<typename T1, typename T2, typename T3>
struct Tuple<T1, T2, T3> : Tuple<T2, T3> {        // 三元组，布局：{T2,T3} 在 T1 之前
    T1 x;
    using Base = Tuple<T2, T3>;
    Base* base() { return static_cast<Base*>(this); }
    const Base* base() const { return static_cast<const Base*>(this); }

    Tuple(const T1& t1, const T2& t2, const T3& t3) :Base{t2, t3}, x{t1} { }
};
```

这些声明的重复性相当高，它们都遵循第一个 Tuple（四元组）的简单代码模式。四元组 Tuple 的定义是主模板，为所有大小（0、1、2、3 和 4）的 Tuple 提供了接口。这也是为什么我必须提供那些默认模板实参 Nil 的原因。实际上，它们永远也不会被用到。特例化会选择一个更简单的 Tuple，而不是使用 Nil。

我定义 Tuple 的方式是形成一个派生类的“栈”，这是一种很传统的方式（例如，std::tuple 就是用类似的方式定义的；见 28.5 节）。它带来一个奇怪的效果——一个 Tuple 的首元素位于最高地址，而尾元素与整个 Tuple 具有相同地址（采用常用实现技术的话）。例如：

```
tuple<double,string,int,char>{3.14,string{"Bob"},127,'c'}
```

此 Tuple 的内存布局可图示如下：



这种布局方式为一些有趣的优化方式提供了可能。考虑下面的代码：

```
class FO { /* 无数据成员的基类对象 */};

typedef Tuple<int*, int*> T0;
typedef Tuple<int*,FO> T1;
typedef Tuple<int*, FO, FO> T2;
```

在我的实现中，有 sizeof(T0)==8、sizeof(T1)==4 和 sizeof(T2)==4，因为空基类被优化掉了。这被称为空基类优化（empty-base optimization），是 C++ 语言所保证的（见 27.4.1 节）。

28.5.1 一个简单的输出函数

Tuple 的定义有一个非常规则的递归结构，我们可以利用它来定义一个显示元素列表的函数。例如：

```
template<typename T1, typename T2, typename T3, typename T4>
void print_elements(ostream& os, const Tuple<T1,T2,T3,T4>& t)
{
    os << t.x << ", ";                    // t 的 x
    print_elements(os,*t.base());
}
template<typename T1, typename T2, typename T3>
void print_elements(ostream& os, const Tuple<T1,T2,T3>& t)
{
    os << t.x << ", ";
    print_elements(os,*t.base());
}

template<typename T1, typename T2>
void print_elements(ostream& os, const Tuple<T1,T2>& t)
{
    os << t.x << ", ";
    print_elements(os,*t.base());
}

template<typename T1>
void print_elements(ostream& os, const Tuple<T1>& t)
{
    os << t.x;
}

template<>
void print_elements(ostream& os, const Tuple<>& t)
{
    os << " ";
}
```

四元组、三元组和二元组的 print_elements() 的相似性暗示着还有更好的解决方案（见 28.6.4 节），但现在我还是用这些 print_elements() 为 Tuple 定义一个 <<：

```
template<typename T1, typename T2, typename T3, typename T4>
ostream& operator<<(ostream& os, const Tuple<T1,T2,T3,T4>& t)
{
    os << "{ ";
    print_elements(os,t);
    os << " }";
```

```
    return os;
}
```

现在我们就可以编写下面这样的程序了：

```
Tuple<double, int, char> x {1.1, 42, 'a'};
cout << x << "\n";

cout << Tuple<double,int,int,int>{1.2,3,5,7} << "\n";
cout << Tuple<double,int,int>{1.2,3,5} << "\n";
cout << Tuple<double,int>{1.2,3} << "\n";
cout << Tuple<double>{1.2} << "\n";
cout << Tuple<>{} << "\n";
```

不出意料，输出结果是：

```
{ 1.1 42, a }
{ 1.2,3,5,7 }
{ 1.2,3,5 }
{ 1.2,3 }
{ 1.2 }
{ }
```

28.5.2　元素访问

如定义所示，Tuple 的元素数目是可变的，元素的类型也可能不同。我们希望高效地访问这些元素，并且不违反类型系统（即，不需要使用强制类型转换）。我们可以设想各种方法，例如为元素命名、为元素编号以及递归访问元素直至到达想要的元素为止。我们将用最后一种方法来实现最常见的访问策略：索引元素。特别是，我们希望为元组实现下标操作。不幸的是，我们无法实现一个适合的 operator[]，因此这里使用一个函数模板 get()：

```
Tuple<double, int, char> x {1.1, 42, 'a'};

cout << "{ "
     << get<0>(x) << ", "
     << get<1>(x) << ", "
     << get<2>(x) << " }\n";    // 输出 { 1.1, 42, a }

auto xx = get<0>(x); // xx 是一个 double
```

设计思想是从 0 开始索引元素，从而在编译时实现元素选择并保留所有类型信息。

函数 get() 构造一个类型为 getNth<T,int> 的对象。getNth<T,int> 的工作是返回一个指向第 N 个元素的引用，假定该元素的类型为 X。给定这样一个辅助类型，我们可以定义 get() 如下：

```
template<typename Ret, int N>
struct getNth {                 // getNth() 记住第 N 个元素的类型（Ret）
    template<typename T>
    static Ret& get(T& t)       // 从 t 的基类获得第 N 个元素的值
    {
        return getNth<Ret,N-1>::get(*t.base());
    }
};
template<typename Ret>
struct getNth<Ret,0> {
    template<typename T>
    static Ret& get(T& t)
```

```
    {
        return t.x;
    }
};
```

基本上，getNth 是一个特殊用途的 for 循环，通过 N-1 次递归实现。其成员函数都是 static 的，因为我们并不真的需要类 getNth 的任何对象。这个类只是作为一个保存 Ret 和 N 的场所，保存的方式使得编译器能使用它们。

作为索引 Tuple 的骨架代码，上面这个实现显得有点儿长，但至少它是类型安全的，也很高效。这里“高效”的意思是，如果有一个好的编译器（实际中很常见），那么访问 Tuple 成员不会带来额外的运行时开销。

为什么我们必须用 get<2>(x) 而不是呢 x[2]？我们可以尝试这样做：

```
template<typename T>
constexpr auto operator[](T t,int N)
{
    return get<N>(t);
}
```

不幸的是，这段代码是错误的：

- operator[]() 必须是成员函数，当然这很容易解决，我们将定义放在 Tuple 中即可。
- 在 operator[]() 中，并不知道实参 N 是否是一个常量表达式。
- 我“忘了”只有 lambda 才能从 return 语句推断返回类型（见 11.4.4 节），但这可以通过加上 ->decltype(get<N>(t)) 来解决。

为了能实现正确的 operator[]()，还需要 C++ 语言提供新的机制，目前我们还只能用 get<2>(x) 这样的语法来应付。

28.5.2.1 const 元组

如定义所示，get() 能用于非 const Tuple 元素，而且能用在赋值号左边。例如：

```
Tuple<double, int, char> x {1.1, 42, 'a'};

get<2>(x) = 'b';        // 正确
```

但它不能用于 const 元组：

```
const Tuple<double, int, char> xx {1.1, 42, 'a'};

get<2>(xx) = 'b';       // 错误：xx 是 const
char cc = get<2>(xx);   // 错误：xx 是 const（奇怪是吗？）
```

问题出在 get() 是通过非 const 引用接受参数的。但 xx 是一个 const，因此不是一个合法的实参。

我们自然也希望能使用 const Tuple。例如：

```
const Tuple<double, int, char> xx {1.1, 422, 'a'};
char cc = get<2>(xx);                   // 正确：从 const 读取值
cout << "xx: " << xx << "\n";
get<2>(xx) = 'x';                       // 错误：xx 是 const
```

为了处理 const Tuple，我们必须为 get() 和 getNth 的 get() 增加 const 版本。例如：

```
template<typename Ret, int N>
struct getNth {             // getNth() 记住第 N 个元素的类型（Ret）
    template<typename T>
```

```
        static Ret& get(T& t)      // 从 t 的基类获得第 N 个元素的值
        {
            return getNth<Ret,N-1>::get(*t.base());
        }

        template<typename T>
        static const Ret& get(const T& t)          // 从 t 的基类获得第 N 个元素的值
        {
            return getNth<Ret,N-1>::get(*t.base());
        }
    };

    template<typename Ret>
    struct getNth<Ret,0> {
        template<typename T> static Ret& get(T& t) { return t.x; }
        template<typename T> static const Ret& get(const T& t) { return t.x; }
    };

    template<int N, typename T1, typename T2, typename T3, typename T4>
    Select<N, T1, T2, T3, T4>& get(Tuple<T1, T2, T3, T4>& t)
    {
        return getNth<Select<N, T1, T2, T3, T4>,N>::get(t);
    }

    template<int N, typename T1, typename T2, typename T3>
    const Select<N, T1, T2, T3>& get(const Tuple<T1, T2, T3>& t)
    {
        return getNth<Select<N, T1, T2, T3>,N>::get(t);
    }
```

现在，我们既能处理 const 实参，也能处理非 const 实参了。

28.5.3　make_tuple

类模板不能推断其模板实参，但函数模板可通过其函数实参来推断模板实参。这意味着我们可以在代码中隐式创建一个 Tuple 类型——通过一个函数来构造：

```
template<typename T1, typename T2, typename T3, typename T4>
Tuple<T1, T2, T3, T4> make_tuple(const T1& t1, const T2& t2, const T3& t3, const T4& t4)
{
    return Tuple<T1, T2, T3, T4>{t1, t2, t3,t4};
}

// ... 其他 4 个 make_tuple ...
```

有了 make_tuple()，我们就可以这样编写程序：

```
auto xxx = make_Tuple(1.2,3,'x',1223);
cout << "xxx: " << xxx << "\n";
```

其他一些有用的函数，如 head() 和 tail()，也都很容易实现。标准库 tuple 提供了这样一些工具函数（见 28.6.4 节）。

28.6　可变参数模板

处理未知数目的元素是一个常见问题。例如，一个错误报告函数可能接受 0 ~ 10 个实参，一个矩阵可能会有 1 ~ 10 个维度以及一个元组可能有 0 ~ 10 个元素。注意，在第一

个和最后一个例子中，元素不必是相同类型。在大多数情况下，我们不希望分别处理每种情况。理想情况下，用一段代码就能处理一个元素、两个元素、三个元素等各种情况。而且，数字10是我凭空设定的；理想情况下，对元素数目不应有上限。

这么多年来，对此问题已经有了很多解决方案。例如，默认实参（见12.2.5节）可以允许一个函数接受可变数目的实参，函数重载（见12.3节）可为每个实参数目提供不同的函数版本。如果所有元素的类型都相同，则传递一个元素列表（见11.3节）是实现可变数目实参的一种可行方法。但是，为了优雅地处理实参数目未知、实参类型也未知（而且不同实参可能是不同类型）的情况，我们需要一些额外的语言支持，这就是被称为可变参数模板（variadic template）的语言特性。

28.6.1 一个类型安全的 printf()

考虑需要未知数目未知类型实参的函数的典型例子：printf()。C和C++标准库中都提供了printf()，它非常灵活，实际效果非常好（见43.3节）。但是，它不适用于用户自定义类型，而且不是类型安全的，此外还是黑客常常攻击之处。

printf()的第一个实参是一个C风格的字符串，被用作“格式字符串”。接下来的实参按照格式字符串的要求使用。格式字符串中是一些格式说明符，如表示浮点数的%g和表示零结尾字符数组的%s，它们控制如何解释接下来的实参。例如：

```
printf("The value of %s is %g\n","x",3.14);
string name = "target";
printf("The value of %s is %P\n",name,Point{34,200});

printf("The value of %s is %g\n",7);
```

第一个printf()调用如预期那样执行，但第二个调用有两个问题：格式说明符%s表示C风格字符串，printf()不能正确解释对应的std::string实参。此外，并不存在%P格式而且一般来说没有什么方法能直接打印用户自定义类型如Point的值。在第三个printf()调用中，我们提供了一个int作为%s的实参，而且我“忘了”为%g提供实参。一般来说，编译器不能将格式字符串所要求的实参数目和类型与程序员实际提供的实参数目和类型进行比较。最后一个调用的输出（如果有输出的话）会很糟糕。

使用可变参数模板，我们能实现一个可扩展且类型安全的printf()。与一般编译时编程一样，这个版本的实现也包括两部分：

[1] 处理只有一个实参（格式字符串）的情况。

[2] 处理至少有一个“额外”实参的情况，按格式字符串的要求将额外实参进行恰当格式化，在恰当位置输出。

最简单的情况是只有一个实参——格式字符串：

```
void printf(const char* s)
{
    if (s==nullptr) return;

    while (*s) {
        if (*s=='%' && *++s!='%') // 确认没有更多的实参
                                  // 在格式字符串中%%表示输出字面%字符
            throw runtime_error("invalid format: missing arguments");
```

```
            std::cout << *s++;
        }
    }
```

这段代码打印出格式字符串。如果没有发现格式说明符，这个 printf() 会抛出一个异常，原因是没有需要格式化的实参。格式说明符被定义为 % 后接 % 之外的其他字符（printf() 用 %% 表示一个普通的 % 字符，并非类型说明符的开始）。注意，即使 % 是字符串的最后一个字符，*++s 也不会越界，因为它指向结尾的零。

这段代码是正确的，我们还需让 printf() 能处理更多实参。这就是模板，特别是可变参数模板发挥作用的地方了：

```
template<typename T, typename... Args>      // 可变参数模板参数列表：一个或多个参数
void printf(const char* s, T value, Args... args)      // 函数参数列表：两个或多个参数
{
    while (s && *s) {
        if (*s=='%' && *++s!='%') {          // 一个格式说明符（忽略其具体是什么）
            std::cout << value;              // 使用第一个非格式实参
            return printf(++s, args...);     // 用实参列表尾作为参数进行递归调用
        }
        std::cout << *s++;
    }
    throw std::runtime_error("extra arguments provided to printf");
}
```

这个 printf() 查找并打印出第一个非格式实参，“剥离”该实参后递归调用自身。当没有更多非格式实参时，它会调用第一个（更简单的版本）printf()。普通字符（即非格式说明符 %）会简单打印出来。

重载 << 代替了使用格式说明符中的（易错的）“提示”。如果一个实参的类型定义了 <<，则该实参会被打印出来；否则，调用不会通过类型检查，程序也就不可能执行。% 之后的格式字符并未使用。我可以设想以类型安全的方式使用这些格式字符，但本例的目的不是设计一个完美的 printf()，而是解释可变参数模板。

Args... 定义了所谓的参数包（parameter pack）。一个参数包就是一个（类型 / 值）对序列，你可以从其中第一个实参开始逐个“剥离”每个实参。当用两个或多个实参调用 printf() 时，会选择下面这个版本

```
void printf(const char* s, T value, Args... args);
```

第一个实参作为 s，第二个实参作为 value，剩下的实参（如果有的话）捆绑为参数包 args 随后使用。在调用 printf(++s, args...) 中，参数包 args 被展开，其第一个元素被选为 value，参数包会比上一次调用少一个元素。这个过程重复执行，直至 args 变为空，这时会调用：

```
void printf(const char*);
```

如果我们真的希望检查 %s 这样的 printf() 格式指令，可以这样做：

```
template<typename T, typename... Args>      // 可变参数模板参数列表：一个或多个参数
void printf(const char* s, T value, Args... args)      // 函数参数列表：两个或多个参数
{
    while (s && *s) {
        if (*s=='%') {    // 一个格式说明符或 %%
            switch (*++s) {
            case '%':     // 不是格式说明符
```

```
                break;
            case 's':
                if (!is_C_style_string<T>() && !is_string<T>())
                    throw runtime_error("Bad printf() format");
                break;
            case 'd':
                if (!is_integral<T>()) throw runtime_error("Bad printf() format");
                break;
            case 'g':
                if (!is_floating_point<T>()) throw runtime_error("Bad printf() format");
                break;
            }
            std::cout << value;              // 使用第一个非格式实参
            return printf(++s, args...);     // 用实参列表尾作为参数进行递归调用
        }
        std::cout << *s++;
    }
    throw std::runtime_error("extra arguments provided to printf");
}
```

标准库提供了 std::is_integral 和 std::is_floating_point，但对于 C 风格字符串，你只能自己打造 Is_C_style_string。

28.6.2 技术细节

如果熟悉函数式程序设计，你应该发现了本节的 printf() 例子使用的并不是标准技术常用的表示方式。如果你不熟悉也没有关系，下面给出一些最简单的例子会对你有所帮助。首先，我们可以声明、使用一个简单的可变参数模板函数：

```
template<typename... Types>
void f(Types... args);      // 可变参数模板函数
```

即，f() 是一个可接受任意数目、任意类型实参的函数：

```
f();                    // 正确：args 不包含实参
f(1);                   // 正确：args 包含一个实参——int
f(2, 1.0);              // 正确：args 包含两个实参——int 和 double
f(2, 1.0, "Hello");     // 正确：args 包含三个实参——int、double 和 const char*
```

可变参数模板是通过 ... 表示方式定义的：

```
template<typename... Types>
void f(Types... args);      // 可变参数模板函数
```

声明 Types 中的 typename... 指明 Types 是一个模板参数包（template parameter pack）。args 的类型中的 ... 指明 args 是一个函数参数包（function parameter pack）。args 中每个函数实参的类型是 Types 中对应的模板实参。我们可以使用 class... 代替 typename...，含义是相同的。省略号（...）是一个独立的词法符号，因此你可以在它的前后放置空格。C++ 语法允许省略号出现在很多位置，它总是表示“零个或多个”的含义。你可以将参数包理解为一个值序列，编译器已经记住了这些值的类型。例如，我们可以将参数包 {'c',127,string{"Bob"},3.14} 图示如下：

char	int	string	double
'c'	127	"Bob"	3.14

这种结构通常被称为元组（tuple），C++ 标准未规定其内存布局。例如，其布局可能与上图相反（最后一个元素在最低内存地址；见 28.5 节）。但是，这是一种紧密的、非链接的表示方式。为了获得一个值，我们必须从起始地址开始遍历元素，直至到达想要的元素为止。Tuple 的实现展示了这种技术（见 28.5 节）。我们可以找到第一个元素的类型并据此访问它，然后（递归地）处理下一个实参。如果需要，我们可以定义类似 Tuple 的（以及 std::tuple 的；见 28.6.4 节）get<N> 的索引访问形式，但不幸的是 C++ 语言对此并不提供直接支持。

如果你有了一个参数包，可以通过在其后放置一个 ... 将其扩展为它所包含的元素序列。例如：

```
template<typename T, typename... Args>
void printf(const char* s, T value, Args... args)
{
    // ...
    return printf(++s, args...);      // 将 args 的元素作为实参进行递归调用
    // ...
}
```

将参数包扩展为其元素序列并不限于函数调用。例如：

```
template<typename... Bases>
class X : public Bases... {
public:
    X(const Bases&... b) : Bases(b)... { }
};

X<> x0;
X<Bx> x1(1);
X<Bx,By> x2(2,3);
X<Bx,By,Bz> x3(2,3,4);
```

在本例中，Bases... 指出 X 有零个或多个基类。当进行 X 的初始化时，构造函数要求零个或多个值，它们的类型是在可变参数模板实参 Bases 中指定的。这些值将被一个接一个地传递给对应的基类初始化器。

在大多数需要一个元素列表的地方，我们都可以用省略号表示“零个或多个”的含义（§ iso.14.5.3），例如在：

- 模板实参列表中；
- 函数实参列表中；
- 初始化器列表中；
- 基类说明符列表中；
- 基类或成员初始化器列表中；
- sizeof... 表达式中。

sizeof... 用来获得一个参数包中的元素数目。例如，我们可以为 tuple 定义一个构造函数，它接受一个 pair，初始化包含两个元素的 tuple：

```
template<typename... Types>
class tuple {
    // ...
    template<typename T, typename U, typename = Enable_if<sizeof...(Types)==2>
        tuple(const pair<T,U>>&);
};
```

28.6.3 转发

可变参数模板的一个重要用途是从一个函数向另一个函数转发参数。考虑如何编写这样一个函数，它接受的参数中，包含一个要被调用的对象以及一个要传递给该对象的实参列表（可能为空）：

```
template<typename F, typename... T>
void call(F&& f, T&&... t)
{
    f(forward<T>(t)...);
}
```

这个例子非常简单，但并不是一个虚假的例子。标准库 thread 就有使用这种技术的构造函数（见 5.3.1 节和 42.2.2 节）。一个可推断的模板实参类型的右值引用传参方式能正确区分右值和左值（见 23.5.2.1 节），本例中我利用了这一特点，而 std::forward() 也正是利用了这一点（见 35.5.1 节）。T&&... 中的 ... 是“接受零或多个 && 实参，类型均为 T”的含义。forward<T>(t)... 中的 ... 含义是“从 t 中转发零或多个实参”。

我使用了一个模板实参表示要调用对象的类型，使得 call() 可以接受函数、函数指针、函数对象以及 lambda。

我们可以这样测试 call()：

```
void g0()
{
    cout << "g0()\n";
}

template<typename T>
void g1(const T& t)
{
    cout << "g1(): " << t << '\n';
}

void g1d(double t)
{
    cout << "g1d(): " << t << '\n';
}

template<typename T, typename T2>
void g2(const T& t, T2&& t2)
{
    cout << "g2(): " << t << ' ' << t2 << '\n';
}

void test()
{
    call(g0);
    call(g1);                          // 错误：实参太少
    call(g1<int>,1);
    call(g1<const char*>,"hello");
    call(g1<double>,1.2);
    call(g1d,1.2);
    call(g1d,"No way!");               // 错误：提供给 g1d() 的实参类型错误
    call(g1d,1.2,"I can't count");     // 错误：提供给 g1d() 的实参太多
    call(g2<double,string>,1,"world!");
```

```
    int i = 99;                                    // 用左值进行测试
    const char* p = "Trying";
    call(g2<double,string>,i,p);

    call([](){ cout <<"l1()\n"; });
    call([](int i){ cout <<"l0(): " << i << "\n";},17);
    call([i](){ cout <<"l1(): " << i << "\n"; });
}
```

必须指明传递模板函数的哪个特例化版本，因为 call() 不能从其他实参的类型推断出使用哪一个版本。

28.6.4　标准库 tuple

28.5 节中的简单 Tuple 有一个明显缺点：它最多处理四个元素。本节介绍标准库 tuple（来自于 <tuple>；见 34.2.4.2 节）的定义，并解释其中用到的技术。std::tuple 和我们的简单 Tuple 的关键区别是，前者使用了可变参数模板来去除元素数目的限制。下面是关键定义：

```
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
    : private tuple<Tail...> {   // 递归
/*
    基本上，一个 tuple 保存其头（第一个 (type,value) 对）
    并从其尾导出其他元素（剩下的 (type/value) 对）
    注意，类型是编码在类型中的，并非存为数据
*/
    typedef tuple<Tail...> inherited;
public:
    constexpr tuple() { }// 默认：空 tuple

    // 用独立的实参构造 tuple:
    tuple(Add_const_reference<Head> v, Add_const_reference<Tail>... vtail)
        : m_head(v), inherited(vtail...) { }

    // 用另一个 tuple 构造 tuple:
    template<typename... VValues>
    tuple(const tuple<VValues...>& other)
        : m_head(other.head()), inherited(other.tail()) { }
    template<typename... VValues>
    tuple& operator=(const tuple<VValues...>& other)  // 赋值
    {
        m_head = other.head();
        tail() = other.tail();
        return *this;
    }
    // ...

protected:
    Head m_head;
private:
    Add_reference<Head> head() { return m_head; }
    Add_const_reference<const Head> head() const { return m_head; }

    inherited& tail() { return *this; }
    const inherited& tail() const { return *this; }
};
```

我并不能保证 std::tuple 一定是这样实现的。实际上，有多个流行的 C++ 实现中是通过一个辅助类（也是一个可变参数类模板）来导出剩余元素的，使得内存中的元素布局与具有相同元素类型的 struct 完全一样。

两个“添加引用”类型函数为非引用类型添加引用，我用它们来避免拷贝（见 35.4.1 节）。

奇怪的是，std::tuple 没有提供 head() 和 tail() 函数，因此我将它们声明为私有的。实际上，tuple 没有提供任何访问元素的成员函数。如果你希望访问 tuple 的元素，必须（直接或间接地）调用函数，将其分离为一个值和 ...。如果你希望标准库 tuple 提供 head() 和 tail()，可以这样编写：

```
template<typename Head, typename... Tail>
Head head(tuple<Head,Tail...>& t)
{
    return std::get<0>(t);      // 获得 t 的第一个元素（见 34.2.4.2 节）
}

template<typename Head, typename... Tail>
tuple<T&...> tail(tuple<Head, Tail...>& t)
{
   return /* 细节 */;
}
```

tail() 定义中的“细节”是一段复杂难看的代码。tuple 的设计者肯定并未打算让我们对 tuple 使用 tail()，因此他们才未提供相应的成员函数。

有了 tuple，我们就可以创建元组并对其进行拷贝等操作了：

```
tuple<string,vector,double> tt("hello",{1,2,3,4},1.2);
string h = head(tt.head);                          // "hello"
tuple<vector<int>,double> t2 = tail(tt.tail);      // {{1,2,3,4},1.2};
```

显式指出所有这些类型很乏味，我们可以从实参类型推断出它们，例如使用标准库的 make_tuple：

```
template<typename... Types>
tuple<Types...> make_tuple(Types&&... t)      // 简化版本（见 iso.20.4.2.4）
{
    return tuple<Types...>(t...);
}

string s = "Hello";
vector<int> v = {1,22,3,4,5};
auto x = make_tuple(s,v,1.2);
```

标准库 tuple 的成员比上面给出的实现要多得多（所以我标记 // ...）。而且，标准库版本还提供了一些辅助函数。例如，get() 用来访问元素（类似 28.5.2 节中的 get()），于是我们就可以编写这样的代码：

```
auto t = make_tuple("Hello tuple", 43, 3.15);
double d = get<2>(t);          // d 变为 3.15
```

也就是说，std::get() 为 std::tuple 提供了从零开始编号的编译时下标操作。

std::tuple 的每个成员都是对一部分人有用的，而大部分成员都是对很多人有用的。但这些成员的实现都未涉及更多的可变参数模板知识，因此我不再深入讨论。std::tuple 还有来自相同类型（拷贝和移动）、不同元组类型（拷贝和移动）以及 pair 类型的（拷贝和移动）构造函数和赋值运算符。接受 std::pair 实参的操作使用 sizeof...（见 28.6.2 节）确保其目标

tuple 恰有两个元素。std::tuple 共有九个构造函数和赋值运算符接受分配器（见 34.4 节）和 swap()（见 35.5.2 节）。

不幸的是，标准库没有为 tuple 提供 << 或 >>。更糟的是，为 std::tuple 编写 << 异常复杂，因为没有简单而通用的方法遍历标准库 tuple 中的元素。首先我们需要一个辅助函数；它是一个有两个 print() 函数的 struct。一个 print() 递归遍历列表打印元素，而另一个负责在没有元素可打印时停止递归：

```
template<size_t N> // 打印第 N 个及之后的元素
struct print_tuple {
    template<typename... T>
    typename enable_if<(N<sizeof...(T))>::type
    print(ostream& os, const tuple<T...>& t) const      // 非空元组
    {
        os << ", " << get<N>(t);        // 打印一个元素
        print_tuple<N+1>()(os,t);       // 打印剩余元素
    }

    template<typename... T>
    typename enable_if<!(N<sizeof...(T))>::type              // 空元组
    print(ostream&, const tuple<T...>&) const
    {
    }
};
```

这段代码采用的是一个递归函数结合一个停止递归的重载版本的模式（类似 28.6.1 节中的 printf()）。但是，注意它是如何很浪费地令 get<N>() 从 0 到 N 计数的。

现在我们可以为 tuple 编写 << 了：

```
std::ostream& operator << (ostream& os, const tuple<>&)    // 空元组
{
  return os << "{}";
}

template<typename T0, typename ...T>
ostream& operator<<(ostream& os, const tuple<T0, T...>& t) // 非空元组
{
  os << '{' << std::get<0>(t);  // 打印第一个元素
  print_tuple<1>::print(os,t);  // 打印剩余元素
  return os << '}';
}
```

现在就可以打印 tuple 了：

```
void user()
{
    cout << make_tuple() << '\n';
    cout << make_tuple("One meatball!") << '\n';
    cout << make_tuple(1,1.2,"Tail!") << '\n';
}
```

28.7 国际标准单位例子

使用 constexpr 和模板，我们几乎可以在编译时计算任何东西。计算的输入可能很复杂，但我们总是可以将数据 #include 到程序中。但是，我更倾向于只对简单计算这么做，因为以我的观点，这更有利于代码维护。在本节中，我将展示一个例子，它在实现复杂性和

实用性间做出了合理的权衡。编译额外开销很小，也没有运行时额外开销。这个例子提供了一个计算用单位（如米、千克和秒）的小程序库。这些米·千克·秒制（MKS）单位是科学中普遍使用的国际标准单位的一个子集。我选择这个例子是为了展示最简单的元编程技术是如何与其他语言特性和技术组合使用的。

我们希望将单位附加在值上，来避免无意义的计算。例如：

```
auto distance = 10_m;          // 10 米
auto time = 20_s;              // 20 秒
auto speed = distance/time;    // 5 米 / 秒

if (speed == 20)               // 错误：20 无量纲
// ...
if (speed == distance)         // 错误：不能比较米和米 / 秒
// ...
if (speed == 10_m/20_s)        // 正确：单位匹配
// ...
Quantity<MpS2> acceleration = distance/square(time); // MpS2 表示米 /(秒 * 秒)

cout << "speed==" << speed << " acceleration==" << acceleration << "\n";
```

单位为物理值提供了一个类型系统。如上所示，我们可以使用 auto 按需要隐藏类型（见 2.2.2 节），使用用户自定义字面值常量引入有类型的值（见 19.2.6 节），以及使用一个类型 Quantity 在需要时显式表示单位 Unit。一个 Quantity 就是一个带 Unit 的数值。

28.7.1 Unit

这里首先定义 Unit：

```
template<int M, int K, int S>
struct Unit {
    enum { m=M, kg=K, s=S };
};
```

Unit 的成员表示我们感兴趣的 3 个计量单位：

- 长度单位米；
- 质量单位千克；
- 时间单位秒。

注意，单位值并未编码在此类型中。此外，Unit 只在编译时使用。

我们可以为更常用的单位提供更常规的表示方式：

```
using M = Unit<1,0,0>;         // 米
using Kg = Unit<0,1,0>;        // 千克
using S = Unit<0,0,1>;         // 秒
using MpS = Unit<1,0,-1>;      // 米每秒（米 / 秒）
using MpS2 = Unit<1,0,-2>;     // 米每平方秒（米 /(秒 * 秒)）
```

负单位值表示除以带该单位的量。这种三值单位表示法非常灵活。我们可以表示任何涉及单位、质量和时间的计算所需要的恰当单位。我怀疑 Quantity<123,-15,1024> 没什么大用处，即乘以 123 份距离、除以 15 份质量再乘以 1024 份时间，但应知道这种计量单位系统是通用的。Unit<0,0,0> 表示一个无量纲实体——一个没有单位的值。

当我们将两个量相乘时，它们的单位会被相加。因此，Unit 的加法是有用的：

```
template<typename U1, typename U2>
```

```
struct Uplus {
    using type = Unit<U1::m+U2::m, U1::kg+U2::kg, U1::s+U2::s>;
};

template<typename U1, U2>
using Unit_plus = typename Uplus<U1,U2>::type;
```

类似地，当我们做两个量的除法时，它们的单位应相减：

```
template<typename U1, typename U2>
struct Uminus {
    using type = Unit<U1::m-U2::m, U1::kg-U2::kg, U1::s-U2::s>;
};

template<typename U1, U2>
using Unit_minus = typename Uminus<U1,U2>::type;
```

Unit_plus 和 Unit_minus 是 Unit 上的简单类型函数（见 28.2 节）。

28.7.2 Quantity

一个 Quantity 就是一个关联 Unit 的值：

```
template<typename U>
struct Quantity {
    double val;
    explicit Quantity(double d) : val{d} {}
};
```

进一步的改进可以将表示值的类型转换为模板参数，默认实参可以设置为 double。我们可以定义有不同单位的 Quantity：

```
Quantity<M> x {10.5};    // x 为 10.5 米
Quantity<S> y {2};       // y 为 2 秒
```

我将 Quantity 的构造函数设置为 explicit 的，这样就不可能将无量纲实体（如普通的 C++ 浮点字面值常量）隐式转换为 Quantity：

```
Quantity<MpS> s = 7;    // 错误：试图将一个 int 转换为米 / 秒

Quantity<M> comp(Quantity<M>);
// ...
Quantity<M> n = comp(7);    // 错误：comp() 要求一个距离
```

现在我们就可以开始考虑计算了。我们对物理度量可以进行哪些计算呢？这里不准备回顾整本物理课本的知识，但显然需要用到加、减、乘和除。你只能加减具有相同单位的值：

```
template<typename U>
Quantity<U> operator+(Quantity<U> x, Quantity<U> y) // 相同量纲
{
    return Quantity<U>{x.val+y.val};
}

template<typename U>
Quantity<U> operator-(Quantity<U> x, Quantity<U> y) // 相同量纲
{
    return Quantity<U>{x.val-y.val};
}
```

Quantity 的构造函数是 explicit 的，因此我们必须将结果的 double 值转换回 Quantity。

Quantity 乘法需要进行 Unit 加法。类似地，Quantity 除法需要 Unit 相减。例如：

```
template<typename U1, typename U2>
Quantity<Unit_plus<U1,U2>> operator*(Quantity<U1> x, Quantity<U2> y)
{
    return Quantity<Unit_plus<U1,U2>>{x.val*y.val};
}

template<typename U1, typename U2>
Quantity<Unit_minus<U1,U2>> operator/(Quantity<U1> x, Quantity<U2> y)
{
    return Quantity<Unit_minus<U1,U2>>{x.val/y.val};
}
```

定义了这些算术运算，我们就可以表达大多数计算了。但是，我们会发现现实世界中的计算包含相当多的缩放运算，即乘以或除以无量纲的值。我们可以使用 Unit<0,0,0> 但会有些冗长乏味：

```
Quantity<MpS> speed {10};
auto double_speed = Quantity<Unit<0,0,0>>{2}*speed;
```

为了使代码更为简洁，我们可以提供一个从 double 到 Quantity<Unit<0,0,0>> 的隐式类型转换，或是添加几个算术运算的变体。我选择了后者：

```
template<typename U>
Quantity<U> operator*(Quantity<U> x, double y)
{
    return Quantity<U>{x.val*y};
}

template<typename U>
Quantity<U> operator*(double x, Quantity<U> y)
{
    return Quantity<U>{x*y.val};
}
```

现在就可以这样编写代码了：

```
Quantity<MpS> speed {10};
auto double_speed = 2*speed;
```

我没有定义一个从 double 到 Quantity<Unit<0,0,0>> 的隐式类型转换，原因是我们不希望这种转换也能用于加法或减法：

```
Quantity<MpS> speed {10};
auto increased_speed = 2.3+speed;    // 错误：不能将一个无量纲标量值加到一个速度值上
```

我们最好依据应用领域来准确描述对代码的细节要求。

28.7.3 Unit 字面值常量

得益于大多数常见单元都定义了类型别名，我们可以这样编写代码：

```
auto distance = Quantity<M>{10};  // 10 米
auto time = Quantity<S>{20};      // 20 秒
auto speed = distance/time;       // 0.5 米/秒（米每秒）
```

看起来还不坏，但与传统上简单地将单位留给程序员处理的方式相比，这段代码还是有些冗

长了：

```
auto distance = 10.0;          // 10 米
double time = 20;              // 20 秒
auto speed = distance/time;    // 0.5 米 / 秒（米每秒）
```

我们需要用 .0 或显式的 double 来保证类型为 double（以及获得正确的除法结果）。

为两个例子生成的代码应该是相同的，在表示形式上我们还可以做得更好。我们可以为 Quantity 类型引入用户自定义字面值常量（UDL；见 19.2.6 节）：

```
constexpr Quantity<M> operator"" _m(double d) { return Quantity<M>{d}; }
constexpr Quantity<Kg> operator"" _kg(double d) { return Quantity<Kg>{d}; }
constexpr Quantity<S> operator"" _s(double d) { return Quantity<S>{d}; }
```

这样，我们就可以像本节最初的例子那样编写代码了：

```
auto distance = 10_m;          // 10 米
auto time = 20_s;              // 20 秒
auto speed = distance/time;    // 5 米 / 秒

if (speed == 20)               // 错误：20 是无量纲的
// ...
if (speed == distance)         // 错误：不能比较米和米 / 秒
// ...
if (speed == 10_m/20_s)        // 正确：单位匹配
```

我为 Quantity 和无量纲值的混合运算定义了 * 和 /，这样我们就可以用乘法或除法缩放单位。但是，我们还可以提供更多常用单位——以用户自定义字面值常量的形式：

```
constexpr Quantity<M> operator"" _km(double d) { return 1000*d; }
constexpr Quantity<Kg> operator"" _g(double d) { return d/1000; }
constexpr Quantity<Kg> operator"" _mg(double d) { return d/10000000; }      // 毫克
constexpr Quantity<S> operator"" _ms(double d) { return d/1000; }           // 毫秒
constexpr Quantity<S> operator"" _us(double d) { return d/1000; }           // 微秒
constexpr Quantity<S> operator"" _ns(double d) { return d/1000000000; }     // 纳秒
// ...
```

显然，这种非标准后缀的过度使用可能会变得不可控（例如，虽然因为 u 看起来有点儿像希腊字母 m，us 已经被广泛使用，但它还是有些可疑）。

我本可以定义更多类型来提供更多量级（如同 35.3 节中对 std::ratio 那样），但我认为还是保持 Unit 类型的简单性并关注如何完美完成主要任务为宜。

我在单位 _s 和 _m 中使用了下划线，以便与标准库提供的更短也更好的 s 和 m 后缀区分开来。

28.7.4　工具函数

为了完成完整的程序（本节最初的例子所要求的），我们还需要工具函数 square()、相等性判定运算符和输出运算符。定义 square() 很简单：

```
template<typename U>
Quantity<Unit_plus<U,U>> square(Quantity<U> x)
{
    return Quantity<Unit_plus<U,U>>(x.val*x.val);
}
```

这基本上展示了如何编写任意的计算函数。我本可以在返回值定义时构造 Unit 对象，但使用已有的类型函数更简单。我们也可以定义一个类型函数 Unit_double。

== 的定义看起来或多或少地像其他 ==，它只能用于比较 Unit 相同的值：

```
template<typename U>
bool operator==(Quantity<U> x, Quantity<U> y)
{
    return x.val==y.val;
}

template<typename U>
bool operator!=(Quantity<U> x, Quantity<U> y)
{
    return x.val!=y.val;
}
```

注意，我们用传值方式传递 Quantity。在运行时，它们表示为 double。

输出函数就是传统的字符处理：

```
string suffix(int u, const char* x)    // 组装后缀
{
    string suf;
    if (u) {
        suf += x;
        if (1<u) suf += '0'+u;

        if (u<0) {
            suf += '-';
            suf += '0'-u;
        }
    }
    return suf;
}
template<typename U>
ostream& operator<<(ostream& os, Quantity<U> v)
{
    return os << v.val << suffix(U::m,"m") << suffix(U::kg,"kg") << suffix(U::s,"s");
}
```

最终，本节最初的例子可以正确执行了：

```
auto distance = 10_m;           // 10 米
auto time = 20_s;               // 20 秒
auto speed = distance/time;     // .5 米 / 秒

if (speed == 20)                // 错误：20 是无量纲的
// ...
if (speed == distance)          // 错误：不能比较米和米 / 秒
// ...
if (speed == 10_m/20_s)         // 正确：单位匹配
// ...

Quantity<MpS2> acceleration = distance/square(time); // MpS2 表示米 / 秒 * 秒

cout << "speed==" << speed << " acceleration==" << acceleration << "\n";
```

使用一个不错的编译器，会将上面的程序编译为与直接使用 double 的版本完全一样的目标代码。但是，它会根据物理单位规则进行“类型检查”（在编译时）。这个例子很好地展示了如何用 C++ 设计一组全新的应用相关的类型，并能按应用领域中的规则进行类型检查。

28.8 建议

[1] 使用元编程提高类型安全；28.1 节。
[2] 使用元编程将计算移至编译时，从而提高性能；28.1 节。
[3] 避免过度使用元编程导致编译速度严重下降；28.1 节。
[4] 从编译时求值和类型函数的角度思考问题；28.2 节。
[5] 使用模板别名作为返回类型的类型函数的接口；28.2.1 节。
[6] 使用 constexpr 函数作为返回（非类型）值的类型函数的接口；28.2.2 节。
[7] 使用萃取非侵入式地关联类型与其属性；28.2.4 节。
[8] 使用 Conditional 在两个类型之间进行选择；28.3.1.1 节。
[9] 使用 Select 在多个类型之间进行选择；28.3.1.3 节。
[10] 使用递归表达编译时迭代；28.3.2 节。
[11] 对运行时无法很好完成的任务使用元编程；28.3.3 节。
[12] 使用 Enable_if 有条件地声明函数模板；28.4 节。
[13] Enable_if 可以用于很多谓词，概念就在其中那些最有用的谓词中；28.4.3 节。
[14] 当你需要一个接受实参数量、类型不定的函数时，使用可变参数模板；28.6 节。
[15] 不要对相同类型实参列表使用可变参数模板（更好的方式是初始化器列表）；28.6 节。
[16] 当需要参数转发时，使用可变参数模板和 std::move()；28.6.3 节。
[17] 使用简单元编程实现高效、优雅的单位系统（可进行细粒度的类型检查）；28.7 节。
[18] 使用用户自定义字面值常量简化单位的使用；28.7 节。

一个矩阵设计

绝不要说得比想得更清楚。

——尼尔斯·玻尔

- 引言
  Matrix 的基本使用；对 Matrix 的要求
- Matrix 模板
  构造和赋值；下标和切片
- Matrix 算术运算
  标量运算；加法；乘法
- Matrix 实现
  slice()；Matrix 切片；Matrix_ref；Matrix 列表初始化；Matrix 访问；零维 Matrix
- 求解线性方程组
  经典高斯消去法；旋转；测试；综合运算
- 建议

29.1 引言

孤立讨论语言特性乏味且无用。本章将展示如何组合使用语言特性来解决一个有挑战性的设计任务：一个通用的 N 维矩阵。

我从未见过一个完美的矩阵类。实际上，考虑到矩阵多种多样的用途，是否存在这样的完美矩阵类也值得怀疑。在本章中，我会介绍编写一个简单的 N 维稠密矩阵所需的编程和设计技术。如果不考虑更多需求，这个 Matrix 非常容易使用，而且足够紧凑和快速，即使程序员用 vector 或内置数组直接实现也很难做得更好了。Matrix 用到的设计和编程技术也适用于其他很多问题。

29.1.1 Matrix 的基本使用

Matrix<T,N> 是一个 N 维矩阵，元素类型为 T。我们可以这样使用它：

```
Matrix<double,0> m0 {1};          // 零维：标量
Matrix<double,1> m1 {1,2,3,4};    // 一维：向量（4 个元素）
Matrix<double,2> m2 {             // 二维（4*3 个元素）
    {00,01,02,03},      // 行 0
    {10,11,12,13},      // 行 1
    {20,21,22,23}       // 行 2
};
Matrix<double,3> m3{4,7,9};       // 三维（4*7*9 个元素，全体初始化为 0）
Matrix<complex<double>,17> m17;// 17 维（还未分配元素）
```

元素类型必须是我们可以存储的类型。对每种元素类型，我们并不要求具有上例中浮点数类型所具备的所有性质。例如：

```
Matrix<double,2> md;          // 正确
Matrix<string,2> ms;          // 正确：只是不要尝试算术运算

Matrix<Matrix<int,2>,2> mm { // 3*2 矩阵，每个元素是 2*2 矩阵
                              // 每个矩阵是一个似真"数"
    {// 行 0
        {{1, 2}, {3, 4}}, // 列 0
        {{4, 5}, {6, 7}}, // 列 1
    },
    {// 行 1
        {{8, 9}, {0, 1}}, // 列 0
        {{2, 3}, {4, 5}}, // 列 1
    },
    {// 行 2
        {{1, 2}, {3, 4}}, // 列 0
        {{4, 5}, {6, 7}}, // 列 1
    }
};
```

矩阵算术运算的数学性质与整数或浮点算术运算不完全一样（例如，矩阵乘法不满足交换律），因此我们必须小心使用矩阵。

类似于 vector，我们使用 () 指定大小，用 {} 指定元素值（见 17.3.2.1 节和 17.3.4.1 节）。指定大小时需给出各维大小，其数量必须与维数匹配，每个维度（每列）上的元素数也必须匹配。例如：

```
Matrix<char,2> mc1(2,3,4);          // 错误：多出一维
Matrix<char,2> mc2 {
    {'1','2','3'}                   // 错误：初始化器未给出第二维元素
};
Matrix<char,2> mc2 {
    {'1','2','3'},
    {'4','5'}                       // 错误：第三列缺少元素
};
```

Matrix<T,N> 用模板参数 N 指出了维数（order()）。每个维度都有固定的元素数（extent()），是从初始化器列表（Matrix 构造函数的实参，{} 形式）推断出的。元素总数可用 size() 获得。例如：

```
Matrix<double,1> m1(100);       // 一维：向量（100 个元素）
Matrix<double,2> m2(50,6000);   // 二维：50*6000 个元素

auto d1 = m1.order();           // 1
auto d2 = m2.order();           // 2

auto e1 = m1.extent(0);         // 100
auto e1 = m1.extent(1);         // 错误：m1 是一维的

auto e2 = m2.extent(0);         // 50
auto e2 = m2.extent(1);         // 6000

auto s1 = m1.size();            // 100
auto s2 = m2.size();            // 50*6000
```

我们可以通过多种形式的下标操作访问 Matrix 元素。例如：

```
Matrix<double,2> m {            // 二维（4*3 个元素）
    {00,01,02,03},  // 行 0
```

```
        {10,11,12,13},  // 行 1
        {20,21,22,23}   // 行 2
    };

    double d1 = m(1,2);                 // d==12
    double d2 = m[1][2];                // d==12
    Matrix<double,1> m1 = m[1];         // 行 1：{10,11,12,13}
    double d3 = m1[2];                  // d==12
```

我们可以定义一个输出函数，在调试时使用，如下所示：

```
template<typename M>
    Enable_if<Matrix_type<M>(),ostream&>
operator<<(ostream& os, const M& m)
{
    os << '{';
    for (size_t i = 0; i!=rows(m); ++i) {
        os << m[i];
        if (i+1!=rows(m)) os << ',';
    }
    return os << '}';
}
```

在本例中，Matrix_type 是一个概念（见 24.3 节）。Enable_if 是 enable_if 的类型的一个别名（见 28.4 节），因此这里 operator<<() 返回一个 ostream&。

这样，cout<<m 会打印出：{{0, 1,2,3},{10,11,12,13},{20,21,22,23}}。

29.1.2　对 Matrix 的要求

在继续讨论 Matrix 的实现之前，考虑我们可能想要哪些特性：

- N 维，N 是一个参数，其值可以从 0 到很大的值，无须为每一维特例化代码。
- N 维存储有很广泛的用途，因此元素类型应该是我们可以存储的任何类型（类似于 vector 元素）。
- 对任何可以合理地描述为数的类型都应该定义数学运算，包括 Matrix。
- 使用每个维度一个索引的 Fortran 风格下标操作，例如，对一个三维 Matrix，m(1,2,3) 得到一个元素。
- C 风格下标，例如，m[7] 得到一行（N 维 Matrix 的一行是其 N–1 维子 Matrix）。
- 下标操作应该快速并支持范围检查。
- 应该有移动赋值运算符和移动构造函数，以确保高效传递 Matrix 结果并消除代价昂贵的临时变量。

有一些数学矩阵运算，如 + 和 *=。

- 有读、写以及传递子矩阵引用 Matrix_ref 的方法，用来读写元素。
- 无资源泄漏应为基本保证（见 13.2 节）。
- 重要的熔合运算，例如，以单一函数调用完成 m*v+v2。

这是一个较长且要求较高的列表，但它并未令我到“每个人都要会能”的程度。例如，我并未列出：

- 更多数学上的矩阵运算。
- 特殊矩阵（如对角阵和三角阵）。
- 支持稀疏 Matrix。

- 支持 Matrix 运算的并行化。

虽然这些特性很有价值，但我们的目的是介绍基本编程技术，这些内容已经超出范围了。

为了提供前面列出的基本特性，我组合了多种语言特性和编程技术：

- 类（这是当然的）。
- 用数值和类型进行参数化。
- 移动构造函数和赋值运算符（最小化拷贝）。
- RAII（依赖于构造函数和析构函数）。
- 可变参数模板（用来指出每维的大小及进行索引）。
- 初始化器列表。
- 运算符重载（实现常规表示方式）。
- 函数对象（携带下标操作相关的信息）。
- 一些简单的模板元编程（例如，用于检查初始化器列表以及用于区分 Matrix_ref 的读写）。
- 实现继承，以最小化代码复制。

显然，这样一个 Matrix 可以作为内置类型（就像许多语言中那样）。但这里的关键点是 C++ 并未将它内置，而是提供给用户一些工具来实现自己的 Matrix。

29.2　Matrix 模板

下面是 Matrix 和它最重要的几个操作的声明，可一览 Matrix 概貌：

```
template<typename T, size_t N>
class Matrix {
public:
    static constexpr size_t order = N;
    using value_type = T;
    using iterator = typename std::vector<T>::iterator;
    using const_iterator = typename std::vector<T>::const_iterator;

    Matrix() = default;
    Matrix(Matrix&&) = default;                          // 移动构造函数
    Matrix& operator=(Matrix&&) = default;
    Matrix(Matrix const&) = default;                     // 拷贝构造函数
    Matrix& operator=(Matrix const&) = default;
    ~Matrix() = default;

    template<typename U>
        Matrix(const Matrix_ref<U,N>&);                  // 从 Matrix_ref 构造
    template<typename U>
        Matrix& operator=(const Matrix_ref<U,N>&);       // 从 Matrix_ref 赋值

    template<typename... Exts>                           // 指明每一维大小
        explicit Matrix(Exts... exts);

    Matrix(Matrix_initializer<T,N>);                     // 列表初始化
    Matrix& operator=(Matrix_initializer<T,N>);          // 列表赋值

    template<typename U>
        Matrix(initializer_list<U>) = delete;            // 除元素外不使用 {}
    template<typename U>
```

```
        Matrix& operator=(initializer_list<U>) = delete;

    static constexpr size_t order() { return N; }                          // 维数
    size_t extent(size_t n) const { return desc.extents[n]; }              // 第 n 维元素数
    size_t size() const { return elems.size(); }                           // 元素总数
    const Matrix_slice<N>& descriptor() const { return desc; }             // 定义下标操作的切片

    T* data() { return elems.data(); }                                     // "下边" 元素访问
    const T* data() const { return elems.data(); }

    // ...

private:
    Matrix_slice<N> desc;          // 定义 N 个维度大小的切片
    vector<T> elems;               // 元素
};
```

用 vector<T> 保存元素使我们不必再考虑内存管理和异常安全性。Matrix_slice 保存了必需的大小信息，用来将元素视为 N 维矩阵访问（见 29.4.2 节），可将它看作 gslice（见 40.5.6 节）针对我们的 Matrix 的专用版本。

Matrix_ref（见 29.4.3 节）的行为类似 Matrix，区别在于它是指向 Matrix 的引用，并不拥有自己的元素，指向的 Matrix 通常是一个子矩阵，例如一行或一列。

Matrix_initializer<T,N> 是 Matrix<T,N> 的初始化器列表，是一种恰当的嵌套结构（见 29.4.4 节）。

29.2.1　构造和赋值

Matrix 默认的拷贝和移动操作恰好具有正确的语义：逐成员拷贝 / 移动 desc（定义下标操作的切片描述符）和 elements。注意，在元素存储空间的管理方面，Matrix 完全借助于 vector。类似地，默认构造函数和析构函数也具有正确的语义。

接受维度大小参数的构造函数是使用可变参数模板（见 28.6 节）的一个很简单的例子：

```
template<typename T, size_t N>
    template<typename... Exts>
    Matrix<T,N>::Matrix(Exts... exts)
        :desc{exts...},        // 读入维度大小
         elems(desc.size)      // 分配 desc.size 个元素，并对它们进行默认初始化
    {}
```

接受初始化器列表的构造函数稍微复杂一些：

```
template<typename T, size_t N>
Matrix<T, N>::Matrix(Matrix_initializer<T,N> init)
{
    Matrix_impl::derive_extents(init,desc.extents);    // 从初始化器列表推断维度大小（见 29.4.4 节）
    elems.reserve(desc.size);                          // 为切片腾出空间
    Matrix_impl::insert_flat(init,elems);              // 用初始化器列表进行初始化（见 29.4.4 节）
    assert(elems.size() == desc.size);
}
```

Matrix_initializer 是恰当嵌套的 initializer_list（见 29.4.4 节）。Matrix_slice 的构造函数推断维度大小，经检查后保存在 desc 中。然后，用 insert_flat() 将元素保存在 elems 中，insert_flat() 定义在名字空间 Matrix_impl 中。

为了保证 {} 初始化方式只能用于元素列表，我删除（= delete）了简单 initializer_list 构造函数。这会导致对维度大小强制使用 () 初始化方式。例如：

```
enum class Piece { none, cross, naught };

Matrix<Piece,2> board1 {
    {Piece::none, Piece::none, Piece::none},
    {Piece::none, Piece::none, Piece::none},
    {Piece::none, Piece::none, Piece::cross}
};
Matrix<Piece,2> board2(3,3);  // 正确
Matrix<Piece,2> board3 {3,3}; // 错误：用 initializer_list<int> 进行初始化的构造函数被删除了
```

如果不删除这个构造函数，最后一个定义就是允许的了。

最后，我们必须能够用 Matrix_ref 构造 Matrix，即用一个 Matrix 或 Matrix 的一部分（子矩阵）的引用构造一个新的 Matrix：

```
template<typename T, size_t N>
    template<typename U>
    Matrix<T,N>::Matrix(const Matrix_ref<U,N>& x)
        :desc{x.desc}, elems{x.begin(),x.end()}  // 拷贝 desc 和元素
    {
        static_assert(Convertible<U,T>(),"Matrix constructor: incompatible element types");
    }
```

由于使用了模板，因此我们可以用保存相容元素类型的 Matrix 构造一个新的 Matrix。

一如往常，赋值操作的定义类似构造函数。例如：

```
template<typename T, size_t N>
    template<typename U>
    Matrix<T,N>& Matrix<T,N>::operator=(const Matrix_ref<U,N>& x)
    {
        static_assert(Convertible<U,T>(),"Matrix =: incompatible element types");

        desc = x.desc;
        elems.assign(x.begin(),x.end());
        return *this;
    }
```

即拷贝 Matrix 的成员。

29.2.2　下标和切片

我们可以通过下标（给出行或元素下标）、行和列或切片（行或列的一部分）等操作来访问 Matrix：

访问 Matrix<T,N>	
m.row(i)	m 的第 i 行，结果是一个 Matrix_ref<T, N-1>
m.column(i)	m 的第 i 列，结果是一个 Matrix_ref<T, N-1>
m[i]	C 风格下标操作，等价于 m.row(i)
m(i,j)	Fortran 风格元素访问，等价于 m[i][j]，类型是 T& 下标必须是 N 个
m(slice(i,n),slice(j))	用切片访问子矩阵，结果是一个 Matrix_ref<T, N>；slice(i, n) 是下标对应的维度上 [i:i+n) 范围内的元素；slice(j) 是下标对应的维度上 [j:max) 范围内的元素；max 是维度大小；下标必须是 N 个

下面是所有成员函数：

```
template<typename T, size_t N>
class Matrix {
public:
    // ...
    template<typename... Args>                                  // m(i,j,k) 用整数进行下标操作
        Enable_if<Matrix_impl::Requesting_element<Args...>(), T&>
        operator()(Args... args);
    template<typename... Args>
        Enable_if<Matrix_impl::Requesting_element<Args...>(), const T&>
        operator()(Args... args) const;
    template<typename... Args>                                  // m(s1,s2,s3) 用切片进行下标操作
        Enable_if<Matrix_impl::Requesting_slice<Args...>(), Matrix_ref<T, N>>
        operator()(const Args&... args);
    template<typename... Args>
        Enable_if<Matrix_impl::Requesting_slice<Args...>(), Matrix_ref<const T,N>>
        operator()(const Args&... args) const;

    Matrix_ref<T,N-1> operator[](size_t i) { return row(i); }   // m[i] 行访问
    Matrix_ref<const T,N-1> operator[](size_t i) const { return row(i); }

    Matrix_ref<T,N-1> row(size_t n);                            // 行访问
    Matrix_ref<const T,N-1> row(size_t n) const;

    Matrix_ref<T,N-1> col(size_t n);                            // 列访问
    Matrix_ref<const T,N-1> col(size_t n) const;

    // ...
};
```

C 风格下标操作 m[i] 取出并返回第 i 行：

```
template<typename T, size_t N>
Matrix_ref<T,N-1> Matrix<T,N>::operator[](size_t n)
{
    return row(n);    // 见 29.4.5 节
}
```

可以将 Matrix_ref（见 29.4.3 节）看作一个指向了 Matrix 的引用。

Matrix_ref<T,0> 是一个特例化版本，指向单个元素（见 29.4.6 节）。

这里通过列出每个维度上的索引实现 Fortran 风格的下标操作，例如，m(i,j,k) 得到一个标量：

```
Matrix<int,2> m2 {
    {01,02,03},
    {11,12,13}
};

m(1,2) = 99;          // 重写第 1 行第 2 列的元素 13
auto d1 = m(1);       // 错误：下标数目太少
auto d2 = m(1,2,3);   // 错误：下标数目太多
```

除了整数下标之外，我们还可以用 slice 进行下标操作。一个 slice 描述某个维度上的一个元素子集（见 40.5.4 节）。特别是，slice{i,n} 指向它所对应的维度上 [i:i+n) 范围内的元素。例如：

```
Matrix<int> m2 {
    {01,02,03},
```

```
        {11,12,13},
        {21,22,23}
    };

    auto m22 = m(slice{1,2},slice{0,3});
```

现在，m22 就是包含下列值的 Matrix<int,2>：

```
    {
        {11,12,13},
        {21,22,23}
    }
```

第一个下标（行下标）slice{1,2} 选取了后两行，第二个下标（列下标）slice{0,3} 选取了列中的所有元素。

使用 slice 的 () 运算符的返回值是一个 Matrix_ref，因此我们可以为其赋值。例如：

```
m(slice{1,2},slice{0,3}) = {
        {111,112,113},
        {121,122,123}
    }
```

现在 m 的值为

```
    {
        {01,02,03},
        {111,112,113},
        {121,122,123}
    }
```

选择某个位置之后的所有元素是非常常见的，因此我为其定义了简短表示：slice{i} 表示 slice{i,max}，其中 max 是比此维度上最大下标更大的值。这样，我们就可以将 m(slice{1,2}, slice{0,3}) 简化为等价的 m(slice{1,2},slice{0})。

另一种常见的简单情况是选择某一行或某一列的所有元素，因此在一组 slice 下标中使用一个普通的整数下标 i 会被解释为 slice{i,1}。例如：

```
Matrix<int> m3 {
        {01,02,03},
        {11,12,13},
        {21,22,23}
    };

    auto m31 = m(slice{1,2},1);        // m31 为 {{12},{22}}
    auto m32 = m(slice{1,2},0);        // m32 为 {{11},{21}}
    auto x = m(1,2);                   // x==13
```

基本上所有用于数值计算的编程语言都支持切片下标表示方式，因此希望它对你来说并不是那么陌生。

我将在 29.4.5 节中给出 row()、column() 和 operator()() 的实现。这些函数的 const 版本的实现与非 const 版本基本相同，关键差异就是 const 版本的返回结果是 const 元素。

29.3　Matrix 算术运算

现在我们就可以创建、拷贝 Matrix，也能访问矩阵元素和矩阵行了。但是，经常需要做的是矩阵数学运算，我们希望能不必以访问单个元素（标量）的方式来表达这些数学运算的算法。例如：

```
Matrix<int,2> mi {{1,2,3}, {4,5,6 }};          // 2*3 矩阵
Matrix<int,2> m2 {mi};                         // 拷贝
mi+=2;                                         // 数值相加：{{2,4,6},{8,10,12}}
Matrix<int,2> m3 = mi+m2;                      // 矩阵加：{{3,6,9},{12,15,18}}
Matrix<int,2> m4 {{1,2}, {3,4}, {5,6}};        // 3*2 矩阵
Matrix<int,1> v = mi*m4;                       // 矩阵乘：{{18,24,30},{38,52,66},{58,80,102}}
```

算术运算定义如下：

```
template<typename T, size_t N>
class Matrix {
    // ...

    template<typename F>
        Matrix& apply(F f);                    // 对每个元素 x 执行 f(x)

    template<typename M, typename F>
        Matrix& apply(const M& m, F f);        // 对给定元素执行 f(x,mx)

    Matrix& operator=(const T& value);         // 对标量赋值

    Matrix& operator+=(const T& value);        // 标量加
    Matrix& operator-=(const T& value);        // 标量减
    Matrix& operator*=(const T& value);        // 标量乘
    Matrix& operator/=(const T& value);        // 标量除
    Matrix& operator%=(const T& value);        // 标量模

    template<typename M>                       // 矩阵加
        Matrix& operator+=(const M& x);
    template<typename M>                       // 矩阵减
        Matrix& operator-=(const M& x);

    // ...
};

// 二元 +、-、* 运算定义为非成员函数
```

29.3.1　标量运算

标量算术运算简单地用右侧运算对象对每个元素执行相应的运算。例如：

```
template<typename T, size_t N>
Matrix<T,N>& Matrix<T,N>::operator+=(const T& val)
{
    return apply([&](T& a) { a+=val; } ); // 使用了 lambda（见 11.4 节）
}
```

这里用到了 apply()，它对 Matrix 的每个元素应用一个函数（或一个函数对象）：

```
template<typename T, size_t N>
    template<typename F>
    Matrix<T,N>& Matrix<T,N>::apply(F f)
    {
        for (auto& x : elems) f(x);        // 此循环使用基于范围的迭代器
        return *this;
    }
```

一如以往，返回 *this 使得链式语法成为可能。例如：

```
m.apply(abs).apply(sqrt);      // 对所有 i，m[i]=sqrt(abs(m[i]))
```

照例（见 3.2.1.1 节和 18.3 节），我们可以在类外用 += 这样的复合赋值运算符来定义 + 这样的“普通运算符”，例如：

```
template<typename T, size_t N>
Matrix<T,N> operator+(const Matrix<T,N>& m, const T& val)
{
    Matrix<T,N> res = m;
    res+=val;
    return res;
}
```

如果没有移动构造函数，这条返回语句将是一个很糟糕的性能缺陷。

29.3.2 加法

两个 Matrix 相加的运算与标量版本非常相似：

```
template<typename T, size_t N>
    template<typename M>
    Enable_if<Matrix_type<M>(),Matrix<T,N>&> Matrix<T,N>::operator+=(const M& m)
    {
        static_assert(m.order()==N,"+=: mismatched Matrix  dimensions");
        assert(same_extents(desc,m.descriptor()));          // 确保大小匹配

        return apply(m, [](T& a,Value_type<M>&b) { a+=b; });
    }
```

Matrix::apply(m,f) 是 Matrix::apply(f) 的双参数版本。它对两个 Matrix（m 和 *this）调用 f：

```
template<typename T, size_t N>
    template<typename M, typename F>
    Enable_if<Matrix_type<M>(),Matrix<T,N>&> Matrix<T,N>::apply(M& m, F f)
    {
        assert(same_extents(desc,m.descriptor()));          // 确保大小匹配
        for (auto i = begin(), j = m.begin(); i!=end(); ++i, ++j)
            f(*i,*j);
        return *this;
    }
```

现在可以很容易地定义 operator+() 了：

```
template<typename T, size_t N>
Matrix<T,N> operator+(const Matrix<T,N>& a, const Matrix<T,N>& b)
{
    Matrix<T,N> res = a;
    res+=b;
    return res;
}
```

我们定义了一个 + 运算，将两个相同类型的 Matrix 相加，得到同样类型的结果 Matrix。我们可以将它泛化：

```
template<typename T, typename T2, size_t N,
    typename RT = Matrix<Common_type<Value_type<T>,Value_type<T2>>,N>
Matrix<RT,N> operator+(const Matrix<T,N>& a, const Matrix<T2,N>& b)
{
    Matrix<RT,N> res = a;
    res+=b;
    return res;
}
```

常见情况下，T 和 T2 是相同的类型，Common_type 也是同样的类型。类型函数 Common_type 源自 std::common_type（见 35.4.2 节）。对两个内置类型，它类似 ?:，会给出一个能最好地保存算术运算结果的类型。如果 Common_type 的定义并不适用于我们想混合使用的一对类型，可以定义相应的版本。例如：

```
template<>
struct common_type<Quad,long double> {
    using type = Quad;
};
```

现在，Common_type<Quad,long double> 会得到 Quad。

我们还需要能用于 Matrix_ref（见 29.4.3 节）的运算。例如：

```
template<typename T, size_t N>
Matrix<T,N> operator+(const Matrix_ref<T,N>& x, const T& n)
{
    Matrix<T,N> res = x;
    res+=n;
    return res;
}
```

这类操作看起来就像对应的 Matrix 版本一样。因为访问 Matrix 和 Matrix_ref 元素并无二致：Matrix 和 Matrix_ref 的差别在于初始化和元素的所有权。

标量加法、乘法等运算的定义，以及 Matrix_ref 的处理，都是简单地重复加法运算定义中用到的技术。

29.3.3　乘法

矩阵乘法不像加法那么简单：一个 N*M 矩阵和一个 M*P 矩阵相乘得到一个 N*P 矩阵。M==1 的情形是两个向量相乘得到一个矩阵，P==1 的情形是一个矩阵乘以一个向量得到一个向量。我们可以将矩阵乘法推广到更高维度，但在这之前必须介绍张量［Kolecki, 2002］，而且我不希望这节的讨论从程序设计技术及如何使用语言特性转向物理和工程数学课程。因此，这里只讨论一维和二维。

将一个 Matrix<T,1> 作为一个 N*1 矩阵处理，并将另一个矩阵作为一个 1*M 矩阵处理，我们得到：

```
template<typename T>
Matrix<T,2> operator*(const Matrix<T,1>& u, const Matrix<T,1>& v)
{
    const size_t n = u.extent(0);
    const size_t m = v.extent(0);
    Matrix<T,2> res(n,m);                    // 一个 n*m 矩阵
    for (size_t i = 0; i!=n; ++i)
        for (size_t j = 0; j!=m; ++j)
            res(i,j) = u[i]*v[j];
    return res;
}
```

这是最简单的情况：矩阵元素 res(i,j) 的值就是 u[i]*v[j]。我并未试图将此定义推广到两个向量元素类型不同的情况。如必要，可以使用实现加法时讨论的技术。

注意，我向 res 的每个元素写入了两次值：一次是初始化为 T{}，一次是赋值 u[i]*v[j]。这大致将乘法的计算代价增加了一倍。如果你觉得这很严重，可以编写一个无此额外开销的

版本，然后观察你的程序的性能是否有明显差异。

接下来，我们可以实现一个 N*M 矩阵和一个向量（可视为一个 M*1 矩阵）的乘法，结果是一个 N*1 矩阵：

```
template<typename T>
Matrix<T,1> operator*(const Matrix<T,2>& m, const Matrix<T,1>& v)
{
    assert(m.extent(1)==v.extent(0));

    const size_t n = m.extent(0);
    Matrix<T,1> res(n);
    for (size_t i = 0; i!=n; ++i)
        for (size_t j = 0; j!=n; ++j)
            res(i) += m(i,j)*v(j);
    return res;
}
```

注意，res 的声明将其元素初始化为 T{}，即数值类型会被初始化为 0，这样 += 就从 0 开始。

N*M 矩阵乘以 M*P 矩阵的处理类似：

```
template<typename T>
Matrix<T,2> operator*(const Matrix<T,2>& m1, const Matrix<T,2>& m2)
{
    const size_t n = m1.extent(0);
    const size_t m = m1.extent(1);
    assert(m==m2.extent(0));        // 列数与行数必须匹配

    const size_t p = m2.extent(1);
    Matrix<T,2> res(n,p);
    for (size_t i = 0; i!=n; ++i)
        for (size_t j = 0; j!=m; ++j)
            for (size_t k = 0; k!=p; ++k)
                res(i,j) = m1(i,k)*m2(k,j);
    return res;
}
```

有很多方法优化这个重要的运算。

例如，最内层循环可以写成更简洁的形式：

```
res(i,j) = dot_product(m1[i],m2.column(j))
```

这里的 dot_product() 是标准库 inner_product()（见 40.6.2 节）的一个简单接口：

```
template<typename T>
T dot_product(const Matrix_ref<T,1>& a, const Matrix_ref<T,1>& b)
{
    return inner_product(a.begin(),a.end(),b.begin(),0.0);
}
```

29.4 Matrix 实现

到目前为止，我尚未给出 Matrix 实现中最复杂的（对某些程序员来说也是最有趣的）"机制"部分。例如：Matrix_ref 是什么？Matrix_slice 又是什么？如何用嵌套的 initializer_list 初始化 Matrix，又如何确保维度是正确的？如何保证不用不恰当的元素类型实例化 Matrix？

呈现这些代码最简单的方式是将 Matrix 的全部代码都放置在一个头文件中。这种情况下，对每个非成员函数的定义加上 inline 修饰。

非 Matrix、Matrix_ref 和 Matrix_slice 的成员的函数都定义在名字空间 Matrix_impl 中，不属于通用接口一部分的函数也是如此。

29.4.1　slice()

用作下标的简单 slice 用 3 个值描述了从整数（下标）到元素位置（索引）的映射：

```
struct slice {
    slice() :start(-1), length(-1), stride(1) { }
    explicit slice(size_t s) :start(s), length(-1), stride(1) { }
    slice(size_t s, size_t l, size_t n = 1) :start(s), length(l), stride(n) { }

    size_t operator()(size_t i) const { return start+i*stride; }

    static slice all;

    size_t start;       // 第一个索引
    size_t length;      // 包含的索引数目（可用于范围检查）
    size_t stride;      // 序列中元素间的距离
};
```

标准库也提供了一个 slice 版本，请查阅 40.5.4 节中更完整的讨论。本版本提供了便利的表示方式（例如，通过构造函数提供默认值）。

29.4.2　Matrix 切片

Matrix_slice 是 Matrix 实现的一部分，它将一组下标映射为一个元素的位置。它使用了广义的切片思想（见 40.5.6 节）：

```
template<size_t N>
struct Matrix_slice {
    Matrix_slice() = default;                                   // 空矩阵：无元素

    Matrix_slice(size_t s, initializer_list<size_t> exts); // 维度大小
    Matrix_slice(size_t s, initializer_list<size_t> exts, initializer_list<size_t> strs);// 维度大小和跨至

    template<typename... Dims>                                  // N 个维度大小
        Matrix_slice(Dims... dims);

    template<typename... Dims,
             typename = Enable_if<All(Convertible<Dims,size_t>()...)>>
        size_t operator()(Dims... dims) const;                  // 从一组下标计算索引

    size_t size;                    // 元素总数
    size_t start;                   // 起始偏移量
    array<size_t,N> extents;        // 每个维度大小
    array<size_t,N> strides;        // 每个维度上元素间的偏移量
};
```

换句话说，Matrix_slice 描述了一个内存区域中哪些部分可被认为是矩阵行和列。在通常的 C/C++ 矩阵行主布局中，一行元素是连续存储的，一列元素是固定间隔（跨距）存储的。Matrix_slice 就是一个函数对象，其 operator()() 实现跨距计算（见 40.5.6 节）：

```
template<size_t N>
```

```
template<typename... Dims>
size_t Matrix_slice<N>::operator()(Dims... dims) const
{
    static_assert(sizeof...(Dims) == N, "");

    size_t args[N] { size_t(dims)... };    // 将实参拷贝到一个数组中

    return inner_product(args,args+N,strides.begin(),size_t(0));
}
```

下标操作必须高效。而上面的代码是一个简化的算法，还需要进行优化。如果不考虑其他情况，我们可以用特例化去掉从可变参数模板包中拷贝出下标的操作。例如：

```
template<>
struct Matrix_slice<1> {

    // ...

    size_t operator()(size_t i) const
    {
        return i;
    }
};

template<>
struct Matrix_slice<2> {

    // ...

    size_t operator()(size_t i, size_t j) const
    {
        return i*stides[0]+j;
    }
};
```

Matrix_slice 对定义 Matrix 的形状（维度大小）以及实现 N 维下标操作非常重要。但其用途不止于此，它对定义子矩阵也很有用。

29.4.3　Matrix_ref

Matrix_ref 本质上就是 Matrix 类的一个克隆，用于表示子 Matrix。但是，Matrix_ref 并不拥有自己的元素，它从一个 Matrix_slice 和一个元素指针构造出来：

```
template<typename T, size_t N>
class Matrix_ref {
public:
    Matrix_ref(const Matrix_slice<N>& s, T* p) :desc{s}, ptr{p} {}
    // ... 很像 Matrix ...
private:
    Matrix_slice<N> desc;    // 矩阵形状
    T* ptr;                  // 指向矩阵的第一个元素
};
```

Matrix_ref 简单地指向“它的”Matrix 的元素。显然，Matrix_ref 的生命期不能超过其 Matrix。例如：

```
Matrix_ref<double,1> user()
```

```
{
    Matrix<double,2> m = {{1,2}, {3,4}, {5,6}};
    return m.row(1);
}

auto mr = user();    // 悬垂
```

Matrix 和 Matrix_ref 极大的相似性导致了代码的重复。如果这会带来困扰，我们可以从一个公共基类派生它们：

```
template<typename T, size_t N>
class Matrix_base {
    // ... 公共内容 ...
};

template<typename T, size_t N>
class Matrix : public Matrix_base<T,N> {
    // ... Matrix 特有内容 ...
private:
    Matrix_slice<N> desc;    // 矩阵形状
    vector<T> elements;
};

template<typename T, size_t N>
class Matrix_ref : public Matrix_base<T,N> {
    // ... Matrix_ref 特有内容 ...
private:
    Matrix_slice<N> desc;    // 矩阵形状
    T* ptr;
};
```

29.4.4　Matrix 列表初始化

从 initializer_list 构造 Matrix 的构造函数接受 Matrix_initializer 类型的参数：

```
template<typename T, size_t N>
using Matrix_initializer = typename Matrix_impl::Matrix_init<T, N>::type;
```

Matrix_init 描述了嵌套的 initializer_list 的结构。

Matrix_init<T,N> 只有一个成员类型 Matrix_init<T,N−1>：

```
template<typename T, size_t N>
struct Matrix_init {
    using type = initializer_list<typename Matrix_init<T,N-1>::type>;
};
```

N==1 是特殊情况，此时我们到达（嵌套最深的）initializer_list<T>：

```
template<typename T>
struct Matrix_init<T,1> {
    using type = initializer_list<T>;
};
```

为了避免意外，我们将 N==0 定义为错误：

```
template<typename T>
struct Matrix_init<T,0>;    // 故意设置为未定义的
```

现在可以完成接受 Matrix_initializer 的 Matrix 构造函数了：

```
template<typename T, size_t N>
Matrix<T, N>::Matrix(Matrix_initializer<T,N> init)
{
    Matrix_impl::derive_extents(init,desc.extents);    // 从初始化器列表推断维度大小（见 29.4.4 节）
    elems.reserve(desc.size);                          // 为切片留出空间
    Matrix_impl::insert_flat(init,elems);              // 从初始化器列表初始化（见 29.4.4 节）
    assert(elems.size() == desc.size);
}
```

为了实现上面的构造过程，我们还需要两个操作来递归下降地遍历 Matrix<T,N> 的 initializer_list 树。

- derive_extents() 确定 Matrix 的形状：
  - 检查树深度是否的确是 N；
  - 检查每一行（子 initializer_list）是否包含相同数目的元素；
  - 设定每一行的大小。
- insert_flat() 将 initializer_list(T) 树中的元素拷贝到 Matrix 的 elems 中。

derive_extents 被 Matrix 的构造函数所调用，它像下面这样初始化其 desc：

```
template<size_t N, typename List>
array<size_t, N> derive_extents(const List& list)
{
    array<size_t,N> a;
    auto f = a.begin();
    add_extents<N>(f,list);    // 将维度大小添加到 a 中
    return a;
}
```

调用者传递给它一个 initializer_list，它返回一个保存维度大小的 array。

递归从 N 一直执行到最后的 1，此时 initializer_list 变为一个 initializer_list<T>：

```
template<size_t N, typename I, typename List>
Enable_if<(N>1),void> add_extents(I& first, const List& list)
{
    assert(check_non_jagged(list));
    *first = list.size();
    add_extents<N-1>(++first,*list.begin());
}

template<size_t N, typename I, typename List>
Enable_if<(N==1),void> add_extents(I& first, const List& list)
{
    *first++ = list.size();        // 达到最深嵌套层
}
```

函数 check_non_jagged() 检查所有行是否都包含相同数目的元素：

```
template<typename List>
bool check_non_jagged(const List& list)
{
    auto i = list.begin();
    for (auto j = i+1; j!=list.end(); ++j)
        if (i->size()!=j->size())
            return false;
    return true;
}
```

我们需要定义 insert_flat()，它接受一个可能嵌套的初始化器列表并将其中的元素保存在一个 vector<T> 中呈现给 Matrix<T>。它接受一个以 Matrix_initializer 形式传递给 Matrix 的 initializer_list，将其 elements 作为目标返回：

```
template<typename T, typename Vec>
void insert_flat(initializer_list<T> list, Vec& vec)
{
    add_list(list.begin(),list.end(),vec);
}
```

不幸的是，我们不能假定元素在内存中连续保存，因此需要通过一组递归调用创建向量。如果我们有一个 initializer_list 列表，则递归地遍历它们：

```
template<typename T, typename Vec>   // 嵌套的 initializer_list
void add_list(const initializer_list<T>* first, const initializer_list<T>* last, Vec& vec)
{
    for (;first!=last;++first)
        add_list(first->begin(),first->end(),vec);
}
```

当到达一个列表，它包含的是非 initializer_list 类型的元素时，就将这些元素插入 vector 中：

```
template<typename T, typename Vec>
void add_list(const T* first, const T* last, Vec& vec)
{
    vec.insert(vec.end(),first,last);
}
```

这里我使用了 vec.insert(vec.end(),first,last)，因为不存在接受实参序列的 push_back()。

29.4.5　Matrix 访问

Matrix 提供了行、列、切片（见 29.4.1 节）和元素（见 29.4.3 节）访问。row() 和 column() 操作返回一个 Matrix_ref<T,N-1>，使用整数的 () 下标操作返回一个 T&，使用 slice 的 () 下标操作返回一个 Matrix<T,N>。

Matrix<T,N> 的每一行都是一个 Matrix_ref<T,N-1>，只要满足 1<N：

```
template<typename T, size_t N>
Matrix_ref<T,N-1> Matrix<T,N>::row(size_t n)
{
    assert(n<rows());
    Matrix_slice<N-1> row;
    Matrix_impl::slice_dim<0>(n,desc,row);
    return {row,data()};
}
```

我们还需要针对 N==1 和 N==0 的特例化版本：

```
template<typename T>
T& Matrix<T,1>::row(size_t i)
{
    return &elems[i];
}

template<typename T>
T& Matrix<T,0>::row(size_t n) = delete;
```

选择一列 column() 本质上与选择一行 row() 一样，仅在 Matrix_slice 的构造上有不同：

```
template<typename T, size_t N>
Matrix_ref<T,N-1> Matrix<T,N>::column(size_t n)
{
    assert(n<cols());
    Matrix_slice<N-1> col;
    Matrix_impl::slice_dim<1>(n,desc,col);
    return {col,data()};
}
```

const 版本与之等同。

Requesting_element() 和 Requesting_slice() 分别是针对整数下标和切片下标的概念。它们检查访问函数的实参类型是否适合作为下标。

针对整数下标的概念定义如下：

```
template<typename T, size_t N>          // 整数下标
    template<typename... Args>
    Enable_if<Matrix_impl::Requesting_element<Args...>(),T&>
    Matrix<T,N>::operator()(Args... args)
    {
        assert(Matrix_impl::check_bounds(desc, args...));
        return *(data() + desc(args...));
    }
```

谓词 check_bounds() 检查下标数目是否与维数相等，以及下标是否都在合法范围内：

```
template<size_t N, typename... Dims>
bool check_bounds(const Matrix_slice<N>& slice, Dims... dims)
{
    size_t indexes[N] {size_t(dims)...};
    return equal(indexes, indexes+N, slice.extents, less<size_t> {});
}
```

Matrix 中元素确切位置的计算是通过调用 Matrix 的 Matrix_slice 的广义切片计算（函数对象 desc(args...)）来完成的。将它添加到数据（data()）的开头，我们就得到了所需位置：

```
return *(data() + desc(args...));
```

这样，声明中最神秘的部分就留在了最后实现。operator()() 的返回类型的说明如下所示：

```
Enable_if<Matrix_impl::Requesting_element<Args...>(),T&>
```

这样，若下面的结果为 true 的话，返回类型就为 T&（见 28.4 节）。

```
Matrix_impl::Requesting_element<Args...>()
```

这个谓词简单地检查每个下标是否可以转换为要求的 size_t 类型，这是通过调用标准库谓词 is_convertible（见 35.4.1 节）的一个概念版本来实现的：

```
template<typename... Args>
constexpr bool Requesting_element()
{
    return All(Convertible<Args,size_t>()...);
}
```

函数 All() 简单地将谓词应用到可变参数模板的每个元素上：

```
constexpr bool All() { return true; }
```

```
template<typename... Args>
constexpr bool All(bool b, Args... args)
{
    return b && All(args...);
}
```

使用谓词 Requesting_element 以及使用“隐藏”在 Request 中的 Enable_if() 的原因是要在元素和 slice 下标运算符间进行选择。针对 slice 下标运算符的谓词如下所示：

```
template<typename... Args>
constexpr bool Requesting_slice()
{
    return All((Convertible<Argssize_t>() || Same<Args,slice>())...)
            && Some(Same<Args,slice>()...);
}
```

即如果至少有一个 slice 实参且所有实参都能转换为 slice 或 size_t，我们就能得到某些可以用来描述 Matrix<T,N> 的东西：

```
template<typename T, size_t N>    // 切片下标操作
    template<typename... Args>
        Enable_if<Matrix_impl::Requesting_slice<Args...>(), Matrix_ref<T,N>>
    Matrix<T,N>::operator()(const Args&... args)
    {
        matrix_slice<N> d;
        d.start = matrix_impl::do_slice(desc,d,args...);
        return {d,data()};
    }
```

我们可计算 slice 如下，slice 表示为 Matrix_slice 中的维度大小和跨距并用于切片下标操作：

```
template<size_t N, typename T, typename... Args>
size_t do_slice(const Matrix_slice<N>& os, Matrix_slice<N>& ns, const T& s, const Args&... args)
{
    size_t m = do_slice_dim<sizeof...(Args)+1>(os,ns,s);
    size_t n = do_slice(os,ns,args...);
    return m+n;
}
```

递归照例结束于一个简单函数：

```
template<size_t N>
size_t do_slice(const Matrix_slice<N>& os, Matrix_slice<N>& ns)
{
    return 0;
}
```

do_slice_dim() 复杂一些（计算正确的切片值），但它并未展示新的编程技术，因此不再介绍。

29.4.6　零维 Matrix

在 Matrix 代码中 N-1 出现了很多次，其中 N 是维数。因此，N==0 可能很容易成为一种糟糕的特殊情况（无论是从编程角度还是从数学角度）。在此，我们可以定义一个特例化版本来解决此问题：

```
template<typename T>
class Matrix<T,0> {
public:
```

```
        static constexpr size_t order = 0;
        using value_type = T;

        Matrix(const T& x) : elem(x) { }
        Matrix& operator=(const T& value) { elem = value; return *this; }

        T& operator()() { return elem; }
        const T& operator()() const { return elem; }

        operator T&() { return elem; }
        operator const T&() { return elem; }
    private:
        T elem;
};
```

Matrix<T,0> 不是一个真正的矩阵。它保存类型为 T 的单一元素，而且只能转换为该类型的一个引用。

29.5　求解线性方程组

只有当你理解了待求解的问题和用于表达求解方案的数学描述时，数值计算代码才是有意义的，否则这些代码只是一些废话而已。如果你学习过线性代数基本知识，那么本节的例子应该非常简单；否则，你可以将它简单地看作从数学课本上的求解方案到代码的一个转换，只做了很少的改变。

本节选择这个例子是为了展示 Matrix 一个非常实际且重要的应用。我们会求解具有如下形式的（任何）一组线性方程：

$$\begin{aligned} a_{1,1}x_1 + \cdots + a_{1,n}x_n &= b_1 \\ &\vdots \\ a_{n,1}x_1 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

在此线性方程组中，x 表示 n 个未知数；a 和 b 是给定的常数。简单起见，我们假设未知数和常数都是浮点数。我们的目标是找到同时满足 n 个方程的未知数值。这些方程可以紧凑地表示成一个矩阵和两个向量的运算：

```
Ax =b
```

此处，A 是一个由常数系数组成的 n*n 方阵：

$$A=\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$$

向量 x 和 b 分别是未知数和常数向量：

$$x=\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad b=\begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

这个系统可能有零个、一个或无穷多个解，这依赖于系数矩阵 A 和向量 b 的值。有很多方法可以用来求解线性系统。我们使用一个称为高斯消去法的经典方法［Freeman,1992］、［Stewart,1998］和［Wood,1999］。首先，我们对 A 和 b 做变换，使 A 变为上三角矩阵。“上三角”的意思是 A 的对角线之下的系数均为 0。换句话说，变换后的系统如下所示：

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ 0 & \ddots & \vdots \\ 0 & 0 & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

这个转换很容易实现。若想 a(i,j) 位置变为 0，可以将第 i 个方程乘以一个常数，使得 a(i,j) 等于第 j 列上的其他元素，比如 a(k,j)。然后将两个方程相减，就能将 a(i,j) 变为 0，第 i 行上的其他值也相应改变。

如果变换后对角线上所有系数都不为 0，则系统有唯一解，我们可使用“回代法”求出这个解。首先，最后一个方程可以很容易地解出：

$$a_{n,n}x_n = b_n$$

显然，x[n] 应该是 b[n]/a(n,n)。然后，将第 n 行从系统中删除，继续求 x[n−1] 的值，如此类推，直至求出 x[1] 的值。对每个 n，我们都要除以 a(n,n)，因此对角线上的值必须非零。如果此条件不满足，回代就会失败，意味着系统有零个或无穷多个解。

29.5.1　经典高斯消去法

现在我们来看一看如何用 C++ 代码表达高斯消去法。首先，我们简化符号表示，还是采用常用方法，为两个将要使用的 Matrix 类型起别名：

```
using Mat2d = Matrix<double,2>;
using Vec = Matrix<double,1>;
```

接下来，我们表达想要进行的计算：

```
Vec classical_gaussian_elimination(Mat2d A, Vec b)
{
    classical_elimination(A, b);
    return back_substitution(A, b);
}
```

即创建输入 A 和 b 的拷贝（采用传值参数），调用一个函数求解系统，然后用回代法计算结果并返回。关键点是我们对问题的分解和采用的符号表示都是源自数学教材的。为了完成完整程序，我们还需实现 classical_elimination() 和 back_substitution()。再次重申，求解方案来源于数学教材：

```
void classical_elimination(Mat2d& A, Vec& b)
{
    const size_t n = A.dim1();

    // 从第一列前进到最后一列之后的位置，将对角线之下的元素都置为 0:
    for (size_t j = 0; j!=n-1; ++j) {
        const double pivot = A[j, j];
        if (pivot==0) throw Elim_failure(j);
        // 将第 i 行对角线之下的元素都置为 0:
        for (size_t i = j+1; i!=n; ++i) {
            const double mult = A[i,j] / pivot;
            A[i](slice(j)) = scale_and_add(A[j](slice(j)), -mult,A[i](slice(j)));
            b[i] -= mult*b[j]; // b 也做相应改变
        }
    }
}
```

枢轴（pivot）是当前正在处理的行中恰好位于对角线上的元素。它必须是非零的，因为回代

过程中需要除以它；如果它为0，我们抛出一个异常，放弃求解。

```
Vec back_substitution(const Mat2d& A, const Vec& b)
{
    const size_t n = A.dim1();
    Vec x(n);

    for (size_t i = n-1; i>=0; --i) {
        double s = b[i]-dot_product(A[i](slice(i+1)),x(slice(i+1)));
        if (double m = A(i,i))
            x[i] = s/m;
        else
            throw Back_subst_failure(i);
    }
    return x;
}
```

29.5.2　旋转

我们可以通过对正在处理的行进行排序，将0和小值从对角线上移开，从而避免除零问题并实现一个更健壮的方案。“更健壮”的意思是对舍入误差更不敏感。但是，随着我们不断将0置于对角线之下，其他值也会随之改变，因此除了0之外，还需要通过重排顺序将小值也从对角线上移开（即不能简单地在一开始重排矩阵然后使用经典算法）：

```
void elim_with_partial_pivot(Mat2d& A, Vec& b)
{
    const size_t n = A.dim1();

    for (size_t j = 0; j!=n; ++j) {
        size_t pivot_row = j;
    // 寻找适合的枢轴：
        for (size_t k = j+1; k!=n; ++k)
            if (abs(A[k,j]) > abs(A[pivot_row,j]))
                pivot_row = k;

        // 如果找到了一个更好的枢轴，交换两行：
        if (pivot_row!=j) {
            A.swap_rows(j,pivot_row);
            std::swap(b[j],b[pivot_row]);
        }

        // 消去：
        for (size_t i = j+1; i!=n; ++i) {
            const double pivot = A[j,j];
            if (pivot==0) error("can't solve: pivot==0");
            const double mult = A[i,j]/pivot;
            A[i].slice(j) = scale_and_add(A[j].slice(j), -mult, A[i].slice(j));
            b[i] -= mult*b[j];
        }
    }
}
```

我们使用了 swap_rows() 和 scale_and_add()，目的是让代码更符合常规并避免自己编写显式循环代码。

29.5.3　测试

显然，我们需要测试代码。幸运的是，有一个简单的测试方法：

```
void solve_random_system(size_t n)
{
    Mat2d A = random_matrix(n); // 生成随机的 Mat2d
    Vec b = random_vector(n);   // 生成随机的 Vec

    cout << "A = " << A << endl;
    cout << "b = " << b << endl;

    try {
        Vec x = classical_gaussian_elimination(A, b);
        cout << "classical elim solution is x = " << x << endl;
            Vec v = A * x;
        cout << " A * x = " << v << endl;
    }
    catch(const exception& e) {
        cerr << e.what() << endl;
    }
}
```

有 3 种情况会进入 catch 子句：

- 代码错误（但作为乐观主义者，我们认为不会有错误）；
- 使 classical_elimination() 出错的输入（使用 elim_with_partial_pivot() 会降低这种可能）；
- 舍入错误。

但是，我们的测试不如希望的那么接近实际，因为完全随机的矩阵不太可能导致 classical_elimination() 出问题。

为了验证求得的解是否正确，我们打印了 A*x，它应该与 b 相等（或者在考虑舍入误差的情况下，就我们的目标而言足够接近）。由于可能产生舍入误差，我们没有像下面这样做：

```
if (A*x!=b) error("substitution failed");
```

由于浮点数只是实数的近似，我们必须接受近似正确的答案。一般而言，最好避免用 == 和 != 判断浮点计算结果是否正确，浮点数天然就是近似值。假如我觉得需要一个结果检查机制，可以定义一个允许一定程度误差的 equal() 函数，然后这样编写代码：

```
if (equal(A*x,b)) error("substitution failed");
```

random_matrix() 和 random_vector() 是随机数的简单应用，我将它们留给读者练习。

29.5.4　熔合运算

除了提供高效的基本运算之外，一个通用矩阵类还应处理三个相关的问题来满足重视性能的用户的需求：

［1］ 最小化临时变量数。

［2］ 最小化矩阵拷贝次数。

［3］ 最小化复合运算中对相同数据的多次循环访问。

考虑 U=M*V+W，其中 U、V 和 W 是向量（Matrix<T,1>），而 M 是一个二维矩阵 Matrix<T,2>。一个朴素的实现会引入临时向量保存 M*V 和 M*V+W 的结果，并会进行拷贝。而一个聪明

的实现会调用函数 mul_add_and_assign(&U,&M,&V,&W)，它不会引入任何临时变量，不会拷贝向量，并保证矩阵元素访问次数最少。

移动构造函数也对优化有帮助：用于 M*V 的临时变量也被用于 M*V+W。如果这样编写代码：

```
Matrix<double,1> U=M*V+W;
```

就会消除所有元素拷贝：为 M*V 中元素分配的局部变量就是最终 U 中保存元素的那些变量。

现在就剩下循环合并（loop fusion）问题了。但除了少量几种表达式，很少需要这种程度的优化，因此效率问题的一个简单解决方案是提供 mul_add_and_assign() 这样的函数，让用户在必要时选择使用。但是，可以设计一个 Matrix 实现对恰当形式的表达式自动应用这种优化，即我们将 U=M*V+W 作为具有四个运算对象的单一运算来处理。ostream 操纵符（见 38.4.5.2 节）也使用了这种基本技术。一般而言，可使用这种技术令 n 个二元运算符的组合表现得像一个 (n+1) 元运算符一样。处理 U=M*V+W 需要引入两个辅助类。然而，在某些系统上，通过使用更强大的优化技术，可以实现令人印象深刻的加速比（比如 30 倍加速）。首先，简单起见，我们将矩阵限定为双精度浮点数的二维矩阵：

```
using Mat2d = Matrix<double,2>;
using Vec = Matrix<double,1>;
```

我们定义 Mat2d 乘以 Vec 的结果：

```
struct MVmul {
    const Mat2d& m;
    const Vec& v;

    MVmul(const Mat2d& mm, const Vec &vv) :m{mm}, v{vv} { }

    operator Vec();// 求值并返回结果
};

inline MVmul operator*(const Mat2d& mm, const Vec& vv)
{
    return MVmul(mm,vv);
}
```

这里的“乘法”应代替 29.3 节中的版本，它除了保存运算对象的引用外什么都不做——M*V 的求值被推迟了。乘法生成的对象与很多技术社区中所说的闭包（closure）紧密相关。如果再增加一个加法运算，也可以类似处理：

```
struct MVmulVadd {
    const Mat2d& m;
    const Vec& v;
    const Vec& v2;

    MVmulVadd(const MVmul& mv, const Vec& vv) :m(mv.m), v(mv.v), v2(vv) { }

    operator Vec();// 求值并返回结果
};

inline MVmulVadd operator+(const MVmul& mv, const Vec& vv)
{
    return MVmulVadd(mv,vv);
}
```

M*V+W 的求值也被推迟了，我们现在必须保证当它被赋予一个 Vec 时能利用好的算法完成求值：

```
template<>
class Matrix<double,1> {        // 特例化（只用于本例）
    // ...
public:
    Matrix(const MVmulVadd& m)                    // 用 m 的结果进行初始化
    {
        // 为元素分配空间等工作
        mul_add_and_assign(this,&m.m,&m.v,&m.v2);
    }

    Matrix& operator=(const MVmulVadd& m)         // 将 m 的结果赋予 *this
    {
        mul_add_and_assign(this,&m.m,&m.v,&m.v2);
        return *this;
    }
    // ...
};
```

U=M*V+W 现在自动扩展为

```
U.operator=(MVmulVadd(MVmul(M,V),W))
```

由于函数采用的是内联方式，这个赋值又被解析为我们所希望的简单调用

```
mul_add_and_assign(&U,&M,&V,&W)
```

显然，拷贝和临时变量被消除了。而且，我们可以用一种优化的方式来编写 mul_add_and_assign()。如果我们只是用一种相对简单且非优化的方式来编写这个函数也没有关系，其表现形式也为编译器提供了很大的优化机会。

这种技术的重要性在于，可以用少量相当简单的语法形式完成大多数对性能确有要求的向量和矩阵运算。当然，对包含半打运算符的表达式进行优化通常不会有什么收益，对这种情况我们一般会编写一个函数。

这种技术基于这样的思想：利用编译时分析和闭包对象将一个子表达式的求值转换为一个表示复合运算的对象。它可用于很多具有相同特性的问题：需要在求值进行前将若干信息汇集到一个函数中。我将用于推迟求值的对象称为合成闭包对象（composition closure object），或简称合成器（compositor）。

如果这种合成技术被用来推迟所有运算的执行，则它被称为表达式模板（expression template）[Vandevoorde，2002][Veldhuizen，1995]。表达式模板系统地使用函数对象来将表达式描述为抽象语法树（Abstract Syntax Tree，AST）。

29.6　建议

[1]　列出基本使用情况；29.1.1 节。

[2]　始终提供输入输出操作以简化简单测试（如单元测试）；29.1.1 节。

[3]　小心列出程序、类或库在理想情况下应该具有哪些性质；29.1.2 节。

[4]　列出程序、类或库超出项目范围的性质；29.1.2 节。

[5]　当设计容器模板时，小心考虑对元素类型的要求；29.1.2 节。

[6]　考虑如何将运行时检查（如用于调试的检查）纳入设计中；29.1.2 节。

[7] 设计类时尽可能模仿已有的专业的符号表示方式和语义；29.1.2 节。
[8] 确保设计中不存在资源泄漏（例如，每个资源有唯一所有者并使用 RAII）；29.2 节。
[9] 考虑如何构造和拷贝类；29.1.1 节。
[10] 提供完整、灵活、高效且语义明确有效的元素访问操作；29.2.2 节和 29.3 节。
[11] 将实现细节放置在自己的 _impl 名字空间中；29.4 节。
[12] 将不要求直接访问类表示形式的常见操作实现为辅助函数；29.3.2 节和 29.3.3 节。
[13] 为了快速访问数据，保持数据紧凑存储并使用访问器对象来提供必要的复杂访问操作；29.4.1 节、29.4.2 节和 29.4.3 节。
[14] 数据结构通常可以用嵌套初始化器列表的形式表达；29.4.4 节。
[15] 当处理数值时，始终考虑“终止情况”，例如零和“很多”；29.4.6 节。
[16] 除了单元测试和检测代码是否符合要求之外，还要使用实际应用例子来对设计进行测试；29.5 节。
[17] 考虑如何将不常见的严格性能要求纳入设计中；29.5.4 节。

C++程序设计语言（第1~3部分）原书第4版

The C++ Programming Language Fourth Edition

C++语言之父的经典名著之最新版本，全面掌握标准C++11及其编程技术的权威指南！
第1版1985年，第2版1991年，第3版1997年，特别版2000年，第4版2013年，经典无限延伸……

本书是在C++语言和程序设计领域具有深远影响、畅销不衰的经典著作，由C++语言的设计者和最初的实现者Bjarne Stroustrup编写，对C++语言进行了最全面、最权威的论述，覆盖标准C++以及由C++所支持的关键编程技术和设计技术，本书英文原版一经面世，即引起业内人士的高度评价和热烈欢迎，先后被翻译成德、希、匈、西、俄、法、日、俄、中、韩等近20种语言，致以百万计的程序员从中获益，是无可取代的C++经典力作。

新的C++11标准使得程序员能以更清晰、更简明、更直接的方式表达思想，从而编写出更快速和高效的代码。在最新出版的第4版中，Stroustrup博士针对最新的C++11标准，为所有希望更有效使用C++语言编程的程序员重新组织、扩展和全面重写了该本C++语言的权威参考书和学习指南，细致、全面、综合地阐述了C++语言及其基本特性、抽象机制、标准库和关键设计技术。

新的C++11标准的内容包括

- 支持并发处理。
- 正则表达式、资源管理指针、随机数、改进的容器（包括哈希表）以及其他很多特性。
- 通用和一致的初始化机制、更简单的for语句、移动语义、基础的Unicode支持。
- lambda表达式、调用常量表达式、控制类默省定义的能力、可变参数模板、模板别名、用户定义的字面值常量。
- 并发性问题。

Pearson
www.pearson.com







定价：139.00元

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn