

C++语言导学

[美] 本贾尼·斯特劳斯特卢普 (Bjarne Stroustrup) 著
得克萨斯农工大学
杨巨峰 王刚 译
南开大学

A Tour of C++

A Tour of C++

Bjarne Stroustrup

C++ In-Depth Series • Bjarne Stroustrup



机械工业出版社
China Machine Press

C++语言导学

A Tour of C++

C++语言设计者全新力作，快速熟悉最新版本C++语言的最佳选择

C++11标准使得程序员能以更清晰、简明、直接的方式表达思想，从而编写出更快速和高效的代码。作为C++的设计者和最初的实现者，Bjarne Stroustrup在他的经典著作《C++程序设计语言》中详尽地介绍了C++语言的各种细节及其用法。

在这本书中，作者把C++完整参考手册的精华概述部分摘取出来，并且进行必要的扩充和提升，目的是让有经验的程序员在很短的时间内就能清晰地了解到构成现代C++语言的关键元素。本书虽薄，结构却清晰完整，作者在书中介绍了C++语言的绝大多数特性以及最重要的一些标准库组件。尤其难能可贵的是，虽然受篇幅所限无法对每个知识点展开深入讲解，但依然能够帮助程序员了解语言的全貌，并给出一些关键示例，便于他们更好地学习和使用C++语言。

作者并非孤立地介绍C++的特性，而是将其置于适当的程序设计风格之中，这些风格包括读者熟悉的面向对象程序设计和泛型编程等。本书内容丰富，涵盖的范围极其广泛。从基础知识开始逐渐延展到更多高级主题，并且包含了大量C++11的新语法点，比如移动语义、规范初始化、lambda表达式、高级容器、随机数和并行等。本书最后讨论了C++的设计、历史演变以及扩展。

本书的目的并非教会读者如何编程（学习编程可以参考作者的《C++程序设计原理与实践》），读者也不可能仅靠阅读本书就熟练掌握C++（为此读者可以参考作者的《C++程序设计语言》）。但是，如果你是一个C/C++程序员，并且想熟悉最新版本的C++语言，或者你精通其他高级语言，希望了解一下C++语言有何特性和长处，那么本书显然是最好的选择。

PEARSON

www.pearson.com

投稿热线: (010) 88379604
客服热线: (010) 88378991 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn



上架指导: 计算机程序设计

ISBN 978-7-111-49812-4



9 787111 498124 >

定价: 39.00元

计 算 机 科 学 丛

C++语言导学

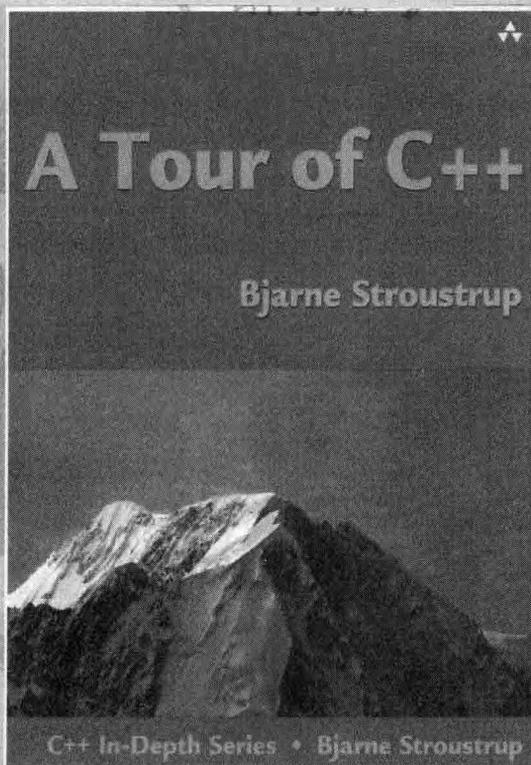
[美] 本贾尼·斯特劳斯特卢普 (Bjarne Stroustrup) 著

得克萨斯农工大学

杨巨峰 王刚 译

南开大学

A Tour of C++



★ 机械工业出版社

图书在版编目 (CIP) 数据

C++ 语言导学 / (美) 斯特劳斯特卢普 (Stroustrup, B.) 著; 杨巨峰, 王刚译. —北京: 机械工业出版社, 2015.3

(计算机科学丛书)

书名原文: A Tour of C++

ISBN 978-7-111-49812-4

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

中国版本图书馆 CIP 数据核字 (2015) 第 061340 号

本书版权登记号: 图字: 01-2013-9380

Authorized translation from the English language edition, entitled *A Tour of C++*, 9780321958310 by Bjarne Stroustrup, published by Pearson Education, Inc., Copyright © 2014.

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 © 2015.

本书中文简体字版由 Pearson Education (培生教育出版集团) 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。

本书作者是 C++ 语言的设计者和最初实现者, 写作本书的目的是让有经验的程序员快速了解 C++ 现代语言。书中几乎介绍了 C++ 语言的全部核心功能和重要的标准库组件, 以很短的篇幅将 C++ 语言的主要特性呈现给读者, 并给出一些关键示例, 让读者用很短的时间就能对现代 C++ 的概貌有一个清晰的了解, 尤其是关于面向对象编程和泛型编程的知识。

本书没有涉及太多 C++ 语言的细节, 非常适合想熟悉 C++ 语言最新特性的 C/C++ 程序设计人员以及精通其他高级语言而想了解 C++ 语言特性和优点的人员阅读。

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: 刘立卿

责任校对: 董纪丽

印刷: 北京瑞德印刷有限公司

版次: 2015 年 4 月第 1 版第 1 次印刷

开本: 185mm × 260mm 1/16

印张: 11

书号: ISBN 978-7-111-49812-4

定价: 39.00 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，不仅筹划了研究的范畴，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, Alfred 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

Pune, India, January 14, 2015

Dear Chinese reader,

C++ is a language that is used world-wide and for an astounding range of application areas. It is a language with a long and glorious history. Consequently, it is not a small language and can be a bit intimidating. I wrote this “tour” of C++ to make modern C++ (C++11) more accessible. The description of all major C++ language constructs and standard-library facilities is described in just 180 pages. It can be read in a day. It is written for people who are already programmers and who would like to know what today’s C++ is without drowning in details. It includes lists of practical advice for use of C++. If you want all the details, have a look at my *The C++ Programming Language (4th Edition)*. If you are not yet a programmer, consider my *Programming: Principles and Practice using C++*.

C++ offers a direct map of its language constructs into hardware facilities (like C) together with a set of mechanism (e.g., classes and templates) that can be used to define simple and efficient abstractions. It is aimed primarily at application areas where performance, reliability, and maintainability are essential. C++ is a language that will grow with your needs. To quote the opening line of the first edition of *The C++ Programming Language*: “C++ is a general-purpose programming language designed to make programming more enjoyable for the serious programmer.”

I am pleased that this thin book is now available to Chinese C++ programmers in their native language. I thank the translators and the translation review team for making this possible.

All the best with C++.

Bjarne Stroustrup

C++ 是一门经典的程序设计语言。

Bjarne Stroustrup 是 C++ 的设计者、最初的实现者和 C++ ISO 标准的主要制定者。

本书是 Bjarne Stroustrup 推出的一本介绍 C++ 的“有趣的新书”。与作者的其他著作相比，本书有三个特点。一是“新”：本书是初版，也是作者首次尝试以快速导览的新形式介绍 C++。从写作手法、章节组织到示例选取都力图推陈出新，一改语言类书籍教条枯燥的通病，文字间洋溢着新意。二是“薄”：本书篇幅短小，每个主题多则二三十页、少则十余页即叙述完成，不论随身携带或者置于案头，读者都可以在几天时间里读完本书并有所收获。三是“精”：本书的文字虽少，内容却不少，甚至可以说非常丰富。不但涉及 C++ 的绝大多数语言特性以及重要的标准库组件，而且涵盖了 C++11 标准几乎全部的新内容。

在翻译过程中我们有这样一个体会，与其说作者是在书中介绍一些语法和技术，不如说他是在传递思想——传递他在发明、设计和不断完善 C++ 语言过程中的所思和所虑。当思想和编程实践产生碰撞时，他又基于丰富的实践经验给出了非常中肯的建议。

很多学习者和程序员常常会有这样的疑问：C++ 是什么？读完本书，相信你会得到满意的答案。

作者 Bjarne 在 Morgan Stanley 的同事庄健刚、田敏、孙瀚峻、陈仇、宗栋、李阳、陈凯、傅光磊、吴妍洁、王立擘审校了本书译稿，并提出了宝贵的修改建议，在此表示衷心感谢！

由于时间紧迫且译者水平有限，书中的不当之处恳请广大读者批评指正。

2014 年冬日

于南开园

教而至简，不亦乐乎。

——西塞罗

现在的 C++ 仿佛进化成了一种新的语言。与 C++98 相比，C++11 更易于我们清晰、简洁、直观地表达思想。而且编译器可以更好地检查程序中的错误，程序的运行速度也提高了。

就像其他任何一种现代编程语言一样，C++ 的规模庞大且提供的库异常丰富，值得程序员认真学习以便高效地利用。这本小册子的目的是让有经验的程序员快速了解现代 C++ 语言，因此它几乎介绍了 C++ 的全部核心功能和重要的标准库组件。读者花费几个小时就能读完本书，但是想必所有人都清楚，要想写出漂亮的 C++ 程序绝非一日之功。好在本书的目的并非让读者熟练掌握一切，而只是介绍语言的概貌，给出一些经典的例子，然后帮助读者开始自己的 C++ 之旅。如果读者希望深入了解 C++ 语言，请阅读我的另一本著作《The C++ Programming Language, Fourth Edition》^①（简称 TC++PL4）。实际上，本书正是 TC++PL4 第 2 ~ 5 章的扩充版，只不过出于完整性和独立性的考虑我们稍微增加了一些内容。本书的篇章结构与 TC++PL4 保持一致，读者如果对细节感兴趣，可以在 TC++PL4 中进一步寻找答案。同样，在我的个人主页（www.stroustrup.com）上为 TC++PL4 编写的习题也可以用于本书。

我们假设读者已经拥有了一些编程经验。如果没有，建议你先找一本入门教材学习一下，比如《Programming: Principles and Practice Using C++》^② [Stroustrup, 2009]。即便你曾经编写过程序，所使用的语言或者编写的应用也可能在风格或形式上与本书相距甚远。

我们用城市观光的例子来比喻本书的作用，比方说参观哥本哈根或者纽约。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。仅靠这样一段旅程，你无法真正了解这座城市，对听到和看到的东西也是一知半解，更别提熟悉这座城市的生存法则。毕竟要想认识并融入一座城市，需要在其中生活很多年。不过幸运的是，此时你已经对城市的总体情况有了一些了解，知道了它的某些特殊之处，并且对有的方面产生了兴趣。接下来，你有机会开始真正的探索之旅了。

本书介绍 C++ 语言的主要功能，尤其是关于面向对象编程和泛型编程的知识。在写作时，我们没有涉及太多细节，更不想把本书写成参考手册。对于标准库也尽量去繁就简，用生动的例子进行讲解。本书没有介绍 ISO 标准之外的库，读者需要的话可以自行查阅有关资料。如果我们提到了某个标准库函数或类，读者很容易就能在头文件中找到它的定义，还可在互联网上搜集到更多与之有关的信息。

本书力求把 C++ 作为一个整体呈现在读者面前，而非像千层糕一样逐层地介绍。因此，

① 本书中文版《C++ 程序设计语言（原书第 4 版）》即将由机械工业出版社出版。——编辑注

② 本书中文版《C++ 程序设计原理与实践》已由机械工业出版社出版（ISBN 978-7-111-30322-0）。——编辑注

在这里我们不细分某项语言特性是归属于 C、C++98 还是 C++11，这些与语言沿革有关的信息在第 14 章（历史和兼容性）可以找到。

致谢

本书的大多数内容源自 TC++ PL4 [Stroustrup, 2012], 因此首先感谢协助我完成 TC++PL4 的所有同仁。还要感谢 Addison-Wesley 的编辑 Peter Gordon, 是他建议作者把 TC++PL4 的部分章节扩展成本书的。

Bjarne Stroustrup

目 录

A Tour of C++

出版者的话	
中文版序	
译者序	
前言	
第 1 章 基础知识	1
1.1 引言	1
1.2 程序	1
1.3 Hello, World!	2
1.4 函数	3
1.5 类型、变量和算术运算	5
1.6 作用域和生命周期	7
1.7 常量	8
1.8 指针、数组和引用	9
1.9 检验	11
1.10 建议	13
第 2 章 用户自定义类型	15
2.1 引言	15
2.2 结构	15
2.3 类	17
2.4 联合	18
2.5 枚举	19
2.6 建议	20
第 3 章 模块化	22
3.1 引言	22
3.2 分离编译	23
3.3 命名空间	24
3.4 错误处理	25
3.4.1 异常	26
3.4.2 不变式	27
3.4.3 静态断言	28
3.5 建议	29
第 4 章 类	30
4.1 引言	30
4.2 具体类型	31
4.2.1 一种算术类型	31
4.2.2 容器	33
4.2.3 初始化容器	34
4.3 抽象类型	36
4.4 虚函数	38
4.5 类层次结构	39
4.5.1 显式覆盖	40
4.5.2 层次结构的益处	41
4.5.3 层次结构漫游	42
4.5.4 避免资源泄漏	43
4.6 拷贝和移动	44
4.6.1 拷贝容器	44
4.6.2 移动容器	45
4.6.3 基本操作	47
4.6.4 资源管理	49
4.6.5 抑制操作	50
4.7 建议	51
第 5 章 模板	53
5.1 引言	53
5.2 参数化类型	53
5.3 函数模板	55
5.4 概念和泛型编程	56
5.5 函数对象	57
5.6 可变参数模板	59
5.7 别名	60
5.8 模板编译模型	61
5.9 建议	61
第 6 章 标准库概览	63
6.1 引言	63
6.2 标准库组件	63

6.3 标准库头文件和命名空间	64	10.5 谓词	102
6.4 建议	66	10.6 标准库算法概览	102
第7章 字符串和正则表达式	67	10.7 容器算法	103
7.1 引言	67	10.8 建议	104
7.2 字符串	67	第11章 实用工具	105
7.2.1 string 的实现	69	11.1 引言	105
7.3 正则表达式	69	11.2 资源管理	105
7.3.1 搜索	70	11.2.1 unique_ptr 和 shared_ptr	106
7.3.2 正则表达式符号表示	71	11.3 特殊容器	108
7.3.3 迭代器	75	11.3.1 array	109
7.4 建议	75	11.3.2 bitset	111
第8章 I/O 流	77	11.3.3 pair 和 tuple	111
8.1 引言	77	11.4 时间	113
8.2 输出	78	11.5 函数适配器	113
8.3 输入	79	11.5.1 bind()	113
8.4 I/O 状态	80	11.5.2 mem_fn()	114
8.5 用户自定义类型的 I/O	81	11.5.3 function	114
8.6 格式化	82	11.6 类型函数	115
8.7 文件流	83	11.6.1 iterator_traits	116
8.8 字符串流	83	11.6.2 类型谓词	117
8.9 建议	84	11.7 建议	118
第9章 容器	86	第12章 数值计算	119
9.1 引言	86	12.1 引言	119
9.2 vector	86	12.2 数学函数	119
9.2.1 元素	89	12.3 数值算法	120
9.2.2 范围检查	89	12.4 复数	121
9.3 list	90	12.5 随机数	121
9.4 map	91	12.6 向量算术	123
9.5 unordered_map	92	12.7 数值限制	124
9.6 容器概述	93	12.8 建议	124
9.7 建议	94	第13章 并发	125
第10章 算法	96	13.1 引言	125
10.1 引言	96	13.2 任务和 thread	126
10.2 使用迭代器	97	13.3 传递参数	126
10.3 迭代器类型	99	13.4 返回结果	127
10.4 流迭代器	100	13.5 共享数据	128

13.6 等待事件	129	14.2 C++11 扩展	140
13.7 任务通信	130	14.2.1 语言特性	140
13.7.1 future 和 promise	131	14.2.2 标准库组件	141
13.7.2 packaged_task	132	14.2.3 已弃用特性	142
13.7.3 async()	133	14.2.4 类型转换	143
13.8 建议	133	14.3 C/C++ 兼容性	143
第 14 章 历史和兼容性	135	14.3.1 C 和 C++ 是兄弟	144
14.1 历史	135	14.3.2 兼容性问题	145
14.1.1 大事年表	136	14.4 参考文献	147
14.1.2 早期的 C++	137	14.5 建议	149
14.1.3 ISO C++ 标准	139	索引	151

基础知识

首要任务，干掉所有语言专家。
——《亨利六世》(第二部分)

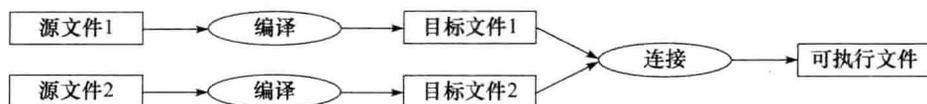
- 引言
- 程序
- Hello, World!
- 函数
- 类型、变量和算术运算
- 作用域和生命周期
- 常量
- 指针、数组和引用
- 检验
- 建议

1.1 引言

本章简要介绍 C++ 的符号系统、C++ 的存储和计算模型以及如何把代码组织成程序。这些语言特性支持 C 语言中绝大多数常见的编程模式，我们称之为面向过程的程序设计 (procedural programming)。

1.2 程序

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成目标文件，然后再用连接器把目标文件组合成可执行程序。一个 C++ 程序通常包含多个源代码文件，简称为源文件 (source file)。



一个可执行程序适用于一种特定的硬件 / 系统组合，不具可移植性。例如，Mac 上的可执行程序无法直接移植到 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：

- 核心语言特性 (core language feature)，比如内置类型 (如 char 和 int) 以及循环 (如 for 语句和 while 语句)；
- 标准库组件 (standard-library component)，比如容器 (如 vector 和 map) 以及 I/O 操作 (如 << 和 getline())。

每个 C++ 实现都提供标准库组件，它们其实也是非常普通的 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)。这就确保 C++ 在面对绝大多数要求较高的系统编程任务时既有丰富的表达力，同时也足够高效。

C++ 是一种静态类型语言，意味着在使用任何实体 (如对象、值、名称和表达式) 时，编译器都必须清楚该实体的类型。对象的类型决定了能在该对象上执行的操作。

1.3 Hello, World !

我们能接触到的最小的 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，该函数既不接受任何参数，也不做什么实际工作。

在 C++ 中，花括号 {} 表示成组的意思，上面的例子里它指出函数体的首尾边界。从双斜线 // 开始直到该行结束是注释，注释只供人阅读和参考，编译器会直接略过注释。

在每个 C++ 程序中有且仅有一个名为 main() 的全局函数，main() 函数是程序执行过程的起始点。如果 main() 返回一个 int 值，则这个值由程序返回给“系统”。如果 main() 不返回任何内容，则系统也会收到一个表示程序成功完成的值。来自 main() 的非零值表示程序失败。并非每一个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，程序会产生某些输出结果。例如，下面这个程序负责输出 Hello, World!：

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!\n";
}
```

2

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。如果没有这些声明，表达式

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

无法正确执行。运算符 << (“输出”) 把它的第二个参数写入第一个参数。在这个例子里，字符串字面值常量 "Hello, World!\n" 被写入标准输出流 std::cout。我们所说的字符

串面值常量是指一对儿双引号内的字符序列。在字符串面值常量中，反斜线 \ 紧跟一个其他字符组成一个“特殊字符”。在这个例子中，\n 是换行符，因此最终的输出结果是 Hello, World!, 后面紧跟一个换行。

符号 `std::` 负责指定名字 `cout` 所在的标准库命名空间（见 3.3 节）。本书在讨论标准特性时通常会省略掉 `std::`，在 3.3 节中我们将介绍在不使用显式限定符的情况下如何让命名空间中的名字可见。

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>           // 包含或引入关于 I/O 流标准库的声明

using namespace std;         // 使得无需使用 std:: 就能使 std 中的名字变得可见（见 3.3 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: the square of 1.234 is 1.52276 (1.234 的平方是 1.52276)
}
```

在上面的代码中，“返回类型” `void` 表示函数 `print_square` 不返回任何值。

1.4 函数

如果我们打算在 C++ 程序中完成某些任务，最好的方式就是调用函数。要想准确描述某项操作的细节，把它定义成函数是最优选择。需要注意的是：函数必须先声明后使用。

一条函数声明语句需要完成三项任务：指定函数的名字、函数的返回值类型（如果有的话）以及要想调用该函数必须提供的实参数量和类型。例如：

```
Elem* next_elem();           // 该函数无需实参，返回值是一个指向 Elem 的指针 (Elem*)
void exit(int);              // 该函数接受一个 int 实参，不返回任何内容
double sqrt(double);         // 该函数接受一个 double 实参，返回值也是一个 double 类型
```

3

在函数声明语句中，返回值类型位于函数的名字之前，实参类型则位于函数的名字之后，并且用括号括起来。

实参传递的过程与拷贝初始化非常类似，编译器负责检查实参的类型，并且在必要的时候执行隐式实参类型转换（见 1.5 节）。例如：

```
double s2 = sqrt(2);         // 调用 sqrt() 函数所用的实参是 double{2}
double s3 = sqrt("three");   // 错误: sqrt() 函数需要的实参类型是 double
```

对于发生在编译过程中的类型检查和转换，程序员需要给予足够的重视。

我们可以在函数的声明语句中写上实参的名字，这有助于程序的读者理解该函数的

含义。但事实上，除非该声明同时也是函数的定义，否则实参的名字不会影响编译过程。例如：

```
double sqrt(double d); // 返回 d 的平方根
double square(double); // 返回实参的平方
```

返回值类型和实参类型属于函数类型的一部分。对于类成员函数（见 2.3 节，4.2.1 节）来说，类名字本身也是函数类型的一部分。例如：

```
double get(const vector<double>& vec, int index); // 函数类型: double(const vector<double>&, int)
char& String::operator[](int index); // 函数类型: char& String::(int)
```

每个程序员都希望自己编写的代码易于理解，因为易于理解是提高代码可维护性的第一步。而要使得整个程序易于理解，首先要把复杂的计算任务分解到易于理解的若干个模块中（以函数和类的形式），并给这些模块起个通俗易懂的名字。函数组成了计算的基本词汇表，正如类型（包括内置类型和用户自定义类型）组成了数据的基本词汇表。C++ 标准算法（如 `find`、`sort` 和 `iota`）是程序函数化的良好开端（见第 10 章），接下来我们就能用这些表示通用任务或者特殊任务的函数组合出更复杂的计算模块了。

代码中错误的数量通常与代码的规模和复杂程度密切相关，多使用一些更短小的函数有助于降低代码的规模和复杂度。举例来说，通过把一项专门的任务定义成函数，我们就能在别的代码段内节省出空间，从而使程序的逻辑结构更加清晰易懂；同时我们也就不得不为这些任务命名并明确它们的依赖关系。

如果程序中存在名字相同但实参类型不同的函数，则编译器负责为每次调用选择匹配度最高的函数。例如：

```
void print(int); // 接受一个整型实参
void print(double); // 接受一个浮点型实参
void print(string); // 接受一个字符串类型的实参

void user()
{
    print(42); // 调用 print(int)
    print(9.65); // 调用 print(double)
    print("D is for Digital"); // 调用 print(string)
}
```

如果存在两个可供选择的函数并且它们难分优劣，则编译器认为此次调用具有二义性并

4 报错。例如：

```
void print(int,double);
void print(double,int);

void user2()
{
    print(0,0); // 错误：二义性调用
}
```

上述关于同名函数的规定就是我们所熟知的函数重载，函数重载是泛型编程（见 5.4 节）的一个关键问题。如果函数被重载了，则所有重载的同名函数应该实现相同的语义内容。`print()` 函数符合这一规定，每个 `print()` 函数都会把它的实参打印出来。

1.5 类型、变量和算术运算

每个名字和每个表达式都有自己的类型，类型决定了名字和表达式所能执行的操作。例如，下面的声明

```
int inch;
```

把 `inch` 的类型指定为 `int`，也就是说，`inch` 是一个整型变量。

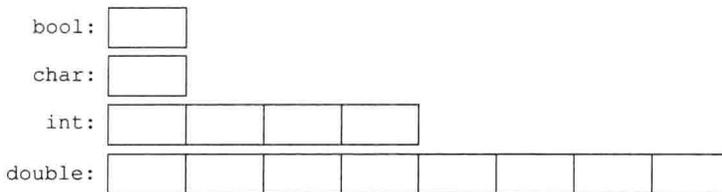
声明 (declaration) 是一条语句，负责为程序引入一个新的名字，并指定该命名实体的类型：

- 类型 (type) 定义了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 -273、42 和 1066
double    // 双精度浮点数，如 -273.15、3.14 和 299793.0
unsigned  // 非负整数，如 0、1 和 999
```

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内存空间 (通常是一个 8 位的字节)，其他类型的尺寸是 `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)，使用 `sizeof` 运算符可以得知该值。例如，`sizeof(char)` 等于 1，`sizeof(int)` 通常是 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

```

除此之外，C++ 还提供了一些逻辑运算符：

```

x&y    // 位与
x|y    // 位或
x^y    // 位异或
~x     // 按位求补
x&&y   // 逻辑与
x||y   // 逻辑或

```

位逻辑运算符是对它的运算对象逐位计算，所得的结果类型与运算对象的类型保持一致。一般逻辑运算符 && 和 || 返回的是 true 或者 false。

在赋值运算和算术运算中，C++ 编译器会在基本类型之间进行各种有意义的类型转换，以便它们能够自由地组合在一起，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;       // 将求和结果赋给 d
    i = d*i;       // 将乘积结果赋给 i (d*i 是一个 double 值，在这里被截成一个 int 值)
}

```

表达式中使用的类型转换称为常用算术类型转换 (usual arithmetic conversion)，它的目的是确保表达式以它的运算对象中最高的精度进行求值计算。例如，一个 double 值和一个 int 值求和，执行的是双精度浮点数的加法。

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，这种形式使用的是以花括号括起来的一个初始值列表：

```

double d1 = 2.3;           // 用 2.3 初始化 d1
double d2 {2.3};          // 用 2.3 初始化 d2
complex<double> z = 1;     // 数值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。抛开其他因素不谈，使用初始值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：

```

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)

```

不幸的是，像把 double 转换成 int 或者把 int 转换成 char 这样的窄化类型转换 (narrowing conversion) 即使会丢失一些信息，但 C++ 编译器不禁止这种转换，反而会隐式地自动执行。隐式窄化类型转换带来的问题是为了与 C 语言兼容而不得不付出的代价 (见 14.3 节)。

常量 (见 1.7 节) 在声明时必须进行初始化，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 4.2.1 节）。

在定义一个变量时，如果变量的类型可以由初始化符号推断得到，则无需显式指定其类型：

```
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
```

可以使用 `=` 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。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`）。

使用 `auto` 可以帮助我们避免冗余，并且无需再书写长类型名。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，类型的名字也可能相当长（见 10.2 节）。

除了传统的算术运算符和逻辑运算符之外，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，使用广泛。

7

1.6 作用域和生命周期

声明语句把一个名字引入它的作用域中：

- 局部作用域（local scope）：声明在函数（见 1.4 节）或者 `lambda`（见 5.5 节）内的名字称为局部名字（local name）。局部名字的作用域从声明它的地方开始，到声明语句所在的块的末尾为止。块（block）的边界用花括号 `{ }` 表示。函数参数的名字也属于局部名字。
- 类作用域（class scope）：如果一个名字定义在类（见 2.2 节，2.3 节，第 4 章）的内部，同时位于任何函数（见 1.4 节）、`lambda`（见 5.5 节）和 `enum class`（见 2.5 节）的外部，则我们把这个名字称为成员名字（member name）或者类成员名字（class member name）。成员名字的作用域从它的括起声明的左侧花括号 `{` 开始，到该声明结束为止。

- 命名空间作用域 (namespace scope)：如果一个名字定义在命名空间 (见 3.3 节) 的内部，同时位于任何函数、lambda (见 5.5 节)、类 (见 2.2 节, 2.3 节, 第 4 章) 和 enum class (见 2.5 节) 的外部，则我们把这个名字称为命名空间成员名字 (namespace member name)。它的作用域从声明它的地方开始，到命名空间结束为止。

声明在所有结构之外的名字称为全局名字 (global name)，我们说它位于全局作用域 (global namespace) 中。

某些对象也可以没有名字，比如临时对象或者用 new (见 4.2.2 节) 创建的对象。例如：

```
vector<int> vec;    // vec 是全局名字 (一个全局整型向量)

struct Record {
    string name; // name 是成员名字 (一个字符串类型的成员)
    // ...
};

void fct(int arg) // fct 是全局名字 (一个全局函数)
                // arg 是局部名字 (一个整型参数)
{
    string motto {"Who dares win"}; // motto 是局部名字
    auto p = new Record{"Hume"};    // p 指向一个未命名的 Record (用 new 创建的)
    // ...
}
```

我们必须先构建 (初始化) 对象，然后才能使用它，对象在作用域的末尾被销毁。对于命名空间对象来说，它的销毁点在整个程序的末尾。对于成员对象来说，它的销毁点依赖于它所属对象的销毁点。用 new 创建的对象一直“存活”到 delete (见 4.2.2 节) 销毁了它为止。

1.7 常量

C++ 支持两种不变性概念：

- `const`：大概的意思是“我承诺不改变这个值”。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`：大概的意思是“在编译时求值”。主要用于说明常量，作用是允许把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 不会更改其参数的值 (见 1.8 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
```

如果某个函数被用在常量表达式 (constant expression) 中，即该表达式在编译时求值，

则这个函数必须定义成 `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：函数中仅有一条计算某个值的 `return` 语句。`constexpr` 函数可以接受非常量实参，但此时其结果不再是一个常量表达式。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，常量表达式是语法规则所必需的（如数组的界（见 1.8 节）、`case` 标签（见 1.9 节）、某些模板参数（见 5.2 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。

1.8 指针、数组和引用

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：

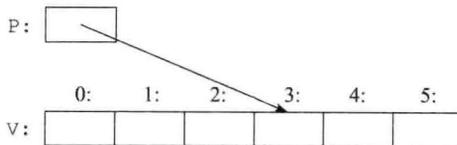
```
char* p;           // 该指针指向字符
```

在声明语句中，`[]` 表示“……的数组”，`*` 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 1.7 节）。指针变量中存放着一个指定类型对象的地址：

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

9

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”。可以用下面的图形来表示上述初始化定义的结果：



考虑将一个数组的 10 个元素拷贝给另一个数组的任务：

```
void copy_fct()
{
    int v1[10] = {0,1,2,3,4,5,6,7,8,9};
    int v2[10];           // v2 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：将 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 ++ 执行简单的加 1 操作。C++ 还提供了一种更简单的 for 语句，即范围 for (range-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)           // 遍历 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，将其从头到尾依次拷入 x 并打印”。注意，当我们使用一个列表初始化数组时无需指定其大小。范围 for 语句可用于任意的元素序列（见 10.1 节）。

如果我们不希望把 v 的值拷贝到变量 x 中，而只是令 x 引用一个元素，则可以编写如下代码：

```
void increment()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};
    for (auto& x : v)
        ++x;
    // ...
}
```

10

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需使用前置运算符 * 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。

当我们指定函数的参数类型时，引用特别有用。例如：

```
void sort(vector<double>& v);    // 排序 v
```

通过把参数类型定义成引用，我们在调用 sort(my_vec) 函数时就不必把实参拷贝给形参，而是直接在 my_vec 上执行操作，对它排序。

还有一种情况，我们既不想修改实参的内容，又希望节省参数拷贝的代价，此时可以使用 const 引用。例如：

```
double sum(const vector<double>&)
```

函数使用 const 引用类型的参数是一个非常普遍的现象。

当用于声明语句时，运算符（如 &、* 和 []）称为声明运算符（declarator operator）：

```
T a[n];    // T[n]: n 个 T 组成的数组
T* p;      // T*: 指向 T 的指针
T& r;      // T&: T 的引用
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
```

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)
// 统计在数组 p[] 中字符 x 出现的次数
// 我们假定 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!=nullptr; ++p)
        if (*p==x)
            ++count;
    return count;
}
```

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

11

`count_x()` 的定义假定 `char*` 是一个 C 风格字符串（C-style string），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。不过，使用 `nullptr` 能够避免混淆整数（如 0 或 `NULL`）和指针（如 `nullptr`）。

例子中用到的 `count_x()` 函数不必太复杂，我们只需对 `nullptr` 稍做检查即可。因为在上面的代码中 `for` 语句并没有执行初始化的部分，所以不妨把它改成更简单的 `while` 语句：

```
int count_x(char* p, char x)
// 统计在数组 p[] 中字符 x 出现的次数
// 我们假定 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
{
    int count = 0;
    while (p) {
        if (*p==x)
            ++count;
        ++p;
    }
    return count;
}
```

`while` 语句不断重复执行，直到它的条件部分变成 `false` 为止。

直接把指针作为条件检验（如 `while(p)`）的效果等同于比较该指针与空指针是否相等（如 `while(p!=nullptr)`）。

1.9 检验

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”）的含义和作用相对应，>> 运算符用于输入数据，cin 是标准输入流（见第 8 章）。>> 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 1.3 节）。

请注意，变量 answer 的定义语句出现在确实需要该变量的地方（不必提前），而声明语句则可以出现在任意位置。

12

我们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。

在使用 switch 语句的时候，如果我们想退出某个 case 分支，不一定要结束当前函数并返回结果，只需要添加一条 break 语句就能继续执行 switch 之后的语句。举个例子说明这点，下面的程序像是个指令式视频游戏，它虽然简单、初步，甚至有点枯燥无趣，但是足以说明问题：

```
void action()
{
    while (true) {
        cout << "enter action:\n";                // 提示用户输入指令
        string act;
        cin >> act;                                // 输入的所有字符存在一个字符串中
        Point delta (0,0);                        // Point 的内容是一个 {x,y} 坐标对
    }
}
```

```
    for (char ch : act) {
        switch (ch) {
            case 'u': // 向上
            case 'n': // 向北
                ++delta.y;
                break;
            case 'r': // 向右
            case 'e': // 向东
                ++delta.x;
                break;
            // ... 其他操作 ...
            default:
                cout << "I freeze!\n";
        }
        move(current+delta*scale);
        update_display();
    }
}
```

13

1.10 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 5 ~ 6 章、第 9 ~ 10 章和第 12 章有更加详细的描述。
- [2] 不必慌张，一切知识都会随着时间推移变得逐渐清晰；参见 1.1 节。
- [3] 要想写出漂亮的程序，你不需要知道 C++ 的所有细节。
- [4] 请关注编程技术，而非语言特性。
- [5] 关于语言定义的一切问题，尽在 ISO C++ 标准；参见 14.1.3 节。
- [6] 把有意义的一组操作“打包”成函数，然后给它起个好名字；参见 1.4 节。
- [7] 一个函数最好只处理一个明确的逻辑操作；参见 1.4 节。
- [8] 对于函数来说，愈简单愈好；参见 1.4 节。
- [9] 函数重载的适用情况是，几个函数的任务相同而处理的类型不同；参见 1.4 节。
- [10] 如果一个函数可能得在编译时求值，那么把它声明成 `constexpr`；参见 1.7 节。
- [11] 别使用“魔法常量”，尽量使用符号化的常量；参见 1.7 节。
- [12] 一条声明语句只声明一个名字。
- [13] 定义名字时，让普通的和局部的名字短一些，特殊的和非局部的名字则可以长一点。
- [14] 避免使用形似的名字。
- [15] 不要出现字母全是大写的名字。
- [16] 当指明了类型名字时，建议在声明语句中使用 `{}` 形式的初始值列表；参见 1.5 节。
- [17] 当使用 `auto` 关键字时，建议在声明语句中使用 `=` 进行初始化；参见 1.5 节。
- [18] 尽量避免使用未经初始化的变量；参见 1.5 节。
- [19] 作用域的范围不要过大；参见 1.6 节。

- [20] 使用指针时尽量简单、直接一些；参见 1.8 节。
- [21] 建议使用 `nullptr`，别再使用 `0` 和 `NULL` 了；参见 1.8 节。
- [22] 如果你还不打算初始化一个变量，那就先别声明它；参见 1.8 节，1.9 节。
- [23] 代码中一目了然的事情就不要加注释。
- [24] 注释是用来解释编程意图的。
- [25] 保持一致的缩进风格。
- [26] 尽量避免复杂的表达式。
- [14] [27] 尽量避免窄化类型转换；参见 1.5 节。

用户自定义类型

不必惊慌失措！

——道格拉斯·亚当斯

- 引言
- 结构
- 类
- 联合
- 枚举
- 建议

2.1 引言

我们把用基本类型（见 1.5 节）、`const` 修饰符（见 1.7 节）和声明运算符（见 1.8 节）构造出来的类型称为内置类型（`built-in type`）。C++ 语言的内置类型及其操作非常丰富，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高层特性。为此，C++ 语言在充分利用内置类型和操作的基础上，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高层特性。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，这些类型具有恰如其分的表现形式和操作，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 type`），诸如类和枚举等。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，本章的剩余部分将介绍其中最简单也是最基础的内容。第 4 ~ 5 章对抽象机制及其支持的编程风格进行了更加详细的描述。第 6 ~ 13 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，所以后面这些章节也从另一个角度阐述了我们到底能用第 1 ~ 5 章介绍的语言特性和编程技术完成什么样的任务。

15

2.2 结构

构建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*。

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, 它含有 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 const 引用 (见 1.8 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)) 的区域中分配内存。分配在自由存储中的对象独立于它所处的作用域, 它会一直“存活”到使用 delete 运算符 (见 4.2.2 节) 销毁它为止。

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, 在优雅程度和灵活性上我们的 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。本章余下部分以及接下来的两章将把 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 9 章介绍标

16 准库 vector, 在其中蕴含着很多漂亮的改进。

本书使用 vector 和其他标准库组件作为示例是为了达到以下两个目的:

- 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。

不要想着重写 `vector` 和 `string` 等标准库组件，直接使用它们更为明智。

访问 `struct` 的成员有两种方式：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 类

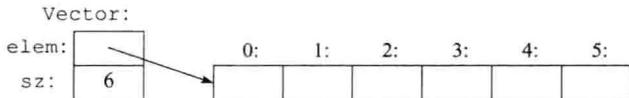
上面这种将数据与其操作分离的做法有其优势，比如我们可以非常自由地使用它的数据部分。不过对于用户自定义类型来说，为了将其所有属性捏合在一起，形成一个“真正的类型”，在其表示形式和操作之间建立紧密的联系还是很有必要的。特别是，我们常常希望自定义的类型易于使用和修改，希望数据具有一致性，并且希望表示形式最好对用户是不可见的。此时，最理想的做法是把类型的接口（所有代码都可使用的部分）与其实现（对外部不可访问的数据具有访问权限）分离开来。在 C++ 中，实现上述目的的语言机制称为类（`class`）。类含有一系列成员（`member`），可能是数据、函数或者类型。类的 `public` 成员定义了该类的接口，`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，我们可以定义一个 `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 4.2.3 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”（如通过 `new` 分配的自由存储，见 4.2.2 节）的一组可变数量的数据。第 4 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` 成员构成的，包括 `Vector()`、`operator[]()` 和 `size()`。2.2 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。因此构造函数 `Vector()` 替换了 2.2 节的 `vector_init()`。构造函数有一个特性与普通函数不同，它确保只用于初始化类的对象，因此定义一个构造函数可以解决类变量未初始化问题。

`Vector(int)` 规定了 `Vector` 对象的构造方式，此处意味着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值列表来初始化 `Vector` 的成员：

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

这条语句的含义是：首先从自由存储获取 `s` 个 `double` 类型的元素，用一个指向这些元素的指针初始化 `elem`；然后使用 `s` 初始化 `sz`。

访问元素的功能是由一个下标函数提供的，它叫做 `operator[]`，它的返回值是元素的引用（`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 3.4 节提及。同样我们也没有提供一种机制来“归还”通过 `new` 获取的 `double` 数组，4.2.2 节将介绍如何使用析构函数来完成这一任务。

我们常用的两个关键字 `struct` 和 `class` 没有本质区别，唯一的不同是 `struct` 的成员默认是 `public` 的。例如，我们也可以为 `struct` 定义构造函数和其他成员函数，这一点与 `class` 完全一致。

18

2.4 联合

`union`（联合）是一种特殊的结构（`struct`），它的所有成员被分配在同一块内存区域中，因此，`union` 实际占用的空间就是它最大的成员所占的空间。显然，同一时刻 `union` 中只能保存一个成员的值。例如，下面的程序实现了一个符号表的表项，它保存着一个名字

和一个值:

```
enum Type { str, num };

struct Entry {
    char* name;
    Type t;
    char* s; // 如果 t==str, 则使用 s
    int i;   // 如果 t==num, 则使用 i
};

void f(Entry* p)
{
    if (p->t == str)
        cout << p->s;
    // ...
}
```

在上面的程序中, 因为 `s` 和 `i` 永远不会同时用到, 所以无形中浪费了内存空间, 使用联合 (即把 `s` 和 `i` 定义为 `union` 的成员) 可以解决该问题。例如:

```
union Value {
    char* s;
    int i;
};
```

C++ 规定由程序员而非编译器负责跟踪 `union` 中实际存储的值:

```
struct Entry {
    char* name;
    Type t;
    Value v; // 如果 t==str, 则使用 s; 如果 t==num, 则使用 i
};

void f(Entry* p)
{
    if (p->t == str)
        cout << p->v.s;
    // ...
}
```

要想时刻保持类型域 (type field, 此例中是 `t`) 与 `union` 中所存类型的对应关系并不是一件容易的事情。为了避免潜在的错误, 我们可以把这些类型封装成一个 `union` 以便于确保其一致性。在应用程序的层面上, 基于这种标记联合 (tagged union) 的抽象概念比较普遍和实用, 相反“裸” `union` 很少会被用到。

19

2.5 枚举

除了类之外, C++ 还提供了另一种形式简单的用户自定义类型, 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 值，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（且易于记忆）的枚举值名字，可以提高代码的可读性，降低出错的风险。

enum 后面的 class 关键字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 不是一个 Color 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
```

默认情况下，enum class 只定义了赋值、初始化和比较（如 == 和 <，见 1.5 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
```

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（无需显式转换），则应该去掉 enum class 中的 class 而得到一个“普通”enum。“普通”enum 当中枚举值的作用域与其 enum 定义所处的作用域一致，并且会隐式地转换成整数值。例如：

```
enum Color { red, green, blue };
int col = green;
```

在这里，col 的值是 1。默认情况下枚举值对应的整数从 0 开始，依次加 1。“普通”enum 很早就用在 C++ 和 C 的程序中了，所以即使它的效果并非最优，时至今日仍被很多人使用。

2.6 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 8 章有更加详细的描述。
- [2] 把有关联的数据组织在一起（struct 或者 class）；参见 2.2 节。
- [3] 在 class 中区分接口部分和实现部分；参见 2.3 节。

- [4] 结构 `struct` 其实就是一个成员在默认情况下均为 `public` 的类 (`class`)；参见 2.3 节。
- [5] 构造函数负责执行和简化类的初始化过程；参见 2.3 节。
- [6] 避免使用“裸” `union`，把它置于类当中形成类型域；参见 2.4 节。
- [7] 用枚举类型来表示一组命名的常量；参见 2.5 节。
- [8] 与“普通” `enum` 相比，建议使用 `enum class`，这样可以省掉很多麻烦；参见 2.5 节。
- [9] 为了使得枚举类型安全易用，不妨为它定义一些操作；参见 2.5 节。

模 块 化

我打断你的时候不许打断我。

——温斯顿·丘吉尔

- 引言
- 分离编译
- 命名空间
- 错误处理
 - 异常；不变式；静态断言
- 建议

3.1 引言

一个 C++ 程序可能包含许多独立开发的部分，例如函数（见 1.3 节）、用户自定义类型（见第 2 章）、类层次（见 4.5 节）和模板（见第 5 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把某个部分的接口和实现分离开来。在语言的层面，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 型值，返回值也是一个 double 型

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 型元素
    int sz;
};
```

23

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 4.3 节）。`sqrt()` 的定义如下所示：

```
double sqrt(double d)    // sqrt() 的定义
{
    // ... 求解平方根的算法，与数学教科书中并无二致 ...
}
```

对于 `Vector` 来说，我们需要定义全部三个成员函数：

```

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，但不必定义 `sqrt()`，因为它是标准库的一部分。然而，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”，编写这些代码用到的语言特性就是我们正在使用的那些。

3.2 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）。库通常是一组分别编译的代码片段（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 型元素
    int sz;
};

```

24

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）到自己的程序当中以便访问接口。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 3.3 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其接口的 .h 文件：

```

// Vector.cpp:

#include "Vector.h" // 获得接口

Vector::Vector(int s)
    :elem(new double[s]), sz{s}    // 初始化成员
{
}

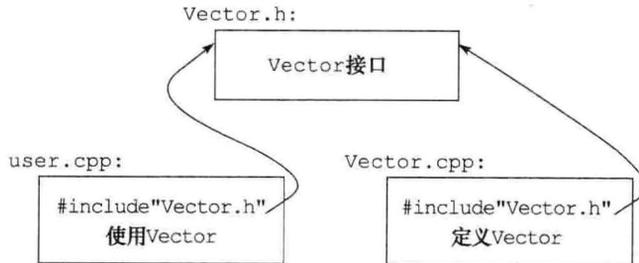
double& Vector::operator[](int i)
{
    return elem[i];
}

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

```

user.cpp 和 Vector.cpp 中的代码共享 Vector.h 中提供的 Vector 接口信息，但

25 这两个文件是相互独立的，可以被分别编译。该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；而是一个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，而后物理上通过文件划分高效地进行分离编译来充分利用模块化。

3.3 命名空间

除了函数（见 1.4 节）、类（见 2.3 节）和枚举（见 2.5 节）之外，C++ 还提供了一种称为命名空间（namespace）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命名空间中的名字冲突。例如，我们尝试利用自己定义的复数类型（见 4.2.1 节，12.4 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();
}
```

26

通过把代码放在命名空间 `My_code` 中，就可以确保我们的名字不会和命名空间 `std`（见 3.3 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 4.2.1 节，12.4 节），所以提前设置这样的预防措施显然是非常明智的。

要想访问其他命名空间中的某个名字，最简单的方法是在这个名字前加上命名空间的名字作为限定（例如 `std::cout` 和 `My_code::main`）。“真正的 `main()`” 定义在全局命名空间中，换句话说，它不属于任何自定义的命名空间、类或者函数。要想获取标准库命名空间中名字的攻击权，我们应该使用 `using` 指令：

```
using namespace std;
```

`using` 指令的作用是把一个指定命名空间中的名字在当前作用域中变得可见，就好像这些名字是当前作用域中的局部名字一样。因此上面这个 `std` 在用了 `using` 指令之后，就不需要再使用 `std::cout`，直接用 `cout` 就可以了。

命名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。通过使用命名空间，我们可以很容易地把若干独立开发的部件组织成一个程序。

3.4 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()`）。这些高级成分简化了程序设计，减少了产生错误的机会（例如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++ 的成分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。程序员应该在开始开发程序后,尽早地设计和描述错误处理的策略。

3.4.1 异常

让我们重新考虑 `Vector` 的例子。对 2.3 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被用在何种程序场景中)。
- `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并通知使用者,然后 `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抛出一个 `out_of_range` 异常:

[27]

```
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。为了实现这一目标,实现部分需要解开 (`unwind`) 函数调用栈以便返回主调函数的上下文。换句话说,异常处理机制把程序的控制权从当前作用域转移到处理该类型错误的代码,在必要的时候调用析构函数(见 4.2.2 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 ...
    }
    // ...
}
```

我们把可能发生异常的可疑程序放在一个 `try` 块当中。显然,对 `v[v.size()]` 的赋值操作将会出错。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,一些标准库容器访问函数也使用它。

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。但是也要注意不能过度使用 `try` 语句。4.2.2 节进一步介绍一些技术——称为资源请求即初始化 (Resource Acquisition Is Initialization)，这些技术使得错误处理简单易用，具有较好的系统性。

我们可以把一个永远不会抛出异常的函数声明成 `noexcept`。例如：

```
void user(int sz) noexcept
{
    Vector v(sz);
    iota(&v[0], &v[sz], 1); // 为 v 赋值 1, 2, 3, 4...
    // ...
}
```

一旦真的发生了错误，函数 `user()` 还是会抛出异常，此时标准库函数 `terminate()` 立即终止当前程序的执行。

28

3.4.2 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) 没有满足，所以函数将拒绝执行。在正式地说明 `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，这一规则在 `operator[]()` 内被检查。记号 `[a:b)` 指定了一个半开区间，其中 `a` 位于区间内而 `b` 不在。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。

然而上面的定义中，`operator[]()` 作用于 `Vector` 的对象并且只在 `Vector` 的成员有“合理”的值时才有意义。特别是，我们说过“`elem` 指向一个含有 `sz` 个 `double` 型元素的数组”，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;
}
```

本书使用标准库异常 `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，然后让它们把任意信息从检测异常的点传递到处理异常的点（见 3.4.1 节）。

29

通常情况下，当遭遇异常问题之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。要想在异常处理模块中抛出（重新抛出）异常，只需书写 `throw`；例如：

```

void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error) {
        cout << "test failed: length error\n";
        throw; // 重新抛出
    }
    catch (std::bad_alloc) {
        // 哎哟！test() 压根儿就没想要处理内存耗尽的问题
        std::terminate(); // 终止程序
    }
}

```

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类似的作用。不变式能够：

- 帮助我们准确地理解想要什么；
- 强制我们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++ 中由构造函数（见第 4 章）和析构函数（见 4.2.2 节，11.2 节）支撑的资源管理概念的基础。

3.4.3 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说明的主要目的。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<=sizeof(int)` 不满足，输出信息 `integers are too small`。也就是说，如果当前系统一个 `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（见 1.7 节）的东西。例如：

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

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

    // ...
}
```

30

通常情况下，`static_assert(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 节，11.6 节）。

对于运行时检查的断言，使用异常。

3.5 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 13 ~ 15 章有更加详细的描述。
- [2] 注意把声明（用作接口）和定义（用作实现）区别开来；参见 3.1 节。
- [3] 头文件的作用是描述接口和强调逻辑结构；参见 3.2 节。
- [4] 如果源文件实现了头文件当中的函数，则应该把头文件 `#include` 到源文件中；参见 3.2 节。
- [5] 不要在头文件中定义非内联函数；参见 3.2 节。
- [6] 用命名空间来表达逻辑结构；参见 3.3 节。
- [7] 用 `using` 指令来为基础库（如 `std`）或某个局部作用域进行（命名空间）转换；参见 3.3 节。
- [8] 不要在头文件中使用 `using` 指令；参见 3.3 节。
- [9] 当无法完成既定的任务时，记得抛出一个异常；参见 3.4 节。
- [10] 使用异常进行错误处理；参见 3.4 节。
- [11] 在设计阶段就想好错误处理的策略；参见 3.4 节。
- [12] 用专门设计的用户自定义类型作为异常类型（而非内置类型）；参见 3.4.1 节。
- [13] 别试图捕获每个函数中的每个错误；参见 3.4 节。
- [14] 如果你的函数不抛出异常，那么把它声明成 `noexcept`；参见 3.4 节。
- [15] 让构造函数建立不变式，不满足就抛出异常；参见 3.4.2 节。
- [16] 围绕不变式设计你的错误处理策略；参见 3.4.2 节。
- [17] 能在编译时检查的问题尽量在编译时检查（使用 `static_assert`）；参见 3.4.3 节。

31
32

类

那些类型一点儿都不“抽象”；它们如此真实，就像 `int` 和 `float` 一样。

——道格·麦克罗伊

- 引言
- 具体类型
 - 一种算术类型；容器；初始化容器
- 抽象类型
- 虚函数
- 类层次结构
 - 显式覆盖；层次结构的益处；层次结构漫游；避免资源泄漏
- 拷贝和移动
 - 拷贝容器；移动容器；基本操作；资源管理；抑制操作
- 建议

4.1 引言

第 4 ~ 5 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：

- 本章通俗地介绍定义和使用新类型（用户自定义类型）的方式，重点介绍与具体类（concrete class）、抽象类（abstract class）和类层次结构（class hierarchy）有关的基本属性、实现技术以及语言特性。
- 下一章介绍模板，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。函数用来表示基于用户自定义类型与内置类型的计算，它们有时泛化为模板函数（template function）和函数对象（function object）。

33

这些语言特性是用于支持所谓面向对象编程（object-oriented programming）和泛型编程（generic programming）等编程风格的。第 6 ~ 13 章将通过一些示例展示标准库功能及其用法。

C++ 最核心的语言特性就是类（class）。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为代码，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写的程序比直接搭建在内置类型之上的程序要容易理解得多。而且，往往库所提供的产品就是类。

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的作用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、更容易维护、更有效率、更优雅、更易用、更易读以及更易推断。大多数编程技术依赖于某些特定类的设计与实现。程序员的需求和偏好千差万别，因此对类的支持也应该是宽泛和丰富的。接下来，我们优先考虑对三种重要的类的基本支持：

- 具体类（见 4.2 节）；
- 抽象类（见 4.3 节）；
- 类层次结构中的类（见 4.5 节）。

很多有用的类都可以归到这三个类别当中，其他类也可以看成是这些类别的简单变形或是通过组合相关技术而实现的。

4.2 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 7.2 节，8.3 节，9.2 节）。

具体类型的典型定义特征是，它的成员变量是其定义的一部分。在很多重要例子中，如 `vector`，成员变量只不过是一个或几个指向保存在别处的数据的指针，但这种成员变量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 1.6 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 节）；
- 拷贝对象（见 4.6 节）。

类的成员变量可以被限定为私有的（就像 `vector` 一样，见 2.3 节），意味着这部分内容确实存在但是只能通过成员函数访问。因此，一旦成员变量发生了任何明显的改动，使用者就必须重新编译整个程序。这也是我们想让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必要的清晰性和效率的类型来说，这个代价是可以接受的，而且通常很理想。如果想提高灵活性，具体类型可以将其成员变量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成员访问它们。`vector` 和 `string` 的机理正是如此，我们可以把它们看成是带有精致接口的资源管理器。

34

4.2.1 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; // 成员变量：两个双精度浮点数
public:
    complex(double r, double i) :re{r}, im{i} {} // 用两个标量构建该复数
    complex(double r) :re{r}, im{0} {} // 用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 12.4 节）略作简化后的版本，类定义本身仅包含需要访问其成员变量的操作。它的成员变量非常简单，也是大家约定俗成的。出于编程实践的需要，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，否则仍旧没有实用价值。这意味着我们应该把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）不应该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，我们也可以在函数声明前加上关键字 `inline` 从而把它显式指定成内联的。一个工业级的 `complex`（就像标准库中的那个一样）必须精心实现，恰当地使用内联。

无需实参就可以调用的构造函数称为默认构造函数（default constructor），`complex()` 是 `complex` 类的默认构造函数。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` 修饰符表示这两个函数不会修改所调用的对象。

很多函数并不需要直接访问 `complex` 的成员变量，因此它们的定义可以与类的定义分

35 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使用。

`==` 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);    // 该函数的定义在其他地方

// ...

```

我们可以像下面这样使用 `complex` 类：

```

void f(complex z)
{
    complex a {2.3};        // 用 2.3 构建了 {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，编译器不允许我们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。

4.2.2 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 节中的定义所示，`Vector` 作为一种容器具有许多优点：它易于理解，建立了有用的不变式（见 3.4.2 节），提供了包含边界检查的访问功能（见 3.4.1 节），并且提供了 `size()` 以允许我们遍历它的元素。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 4.6.4 节），可将未使用的内存提供给新对象，但 C++ 并不保证垃圾回收器总是可用的。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，你更想使用精确的资源释放控制。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 指向一个含 sz 个 double 型元素的数组
    int sz;
public:
    Vector(int s):elem(new double[s]), sz(s)    // 构造函数：获取资源
    {
        for (int i=0; i!=s; ++i)    // 初始化元素
            elem[i]=0;
    }

    ~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在一个求补运算符~后跟上类的名字，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。这一切都无需Vector的使用者干预，他们只需要像对待普通的内置类型变量那样创建和使用Vector对象就可以了。例如：

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

    // ... 使用 v...

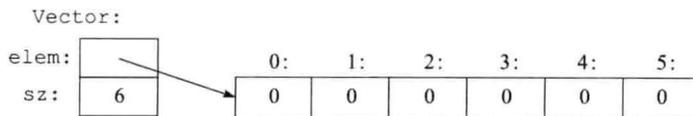
    {
        Vector v2(2*n);
        // ... 使用 v 和 v2...
    } // v2 在此处被销毁

    // ... 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（见1.6节）。出于简化的考虑，Vector没有涉及错误处理，相关内容可以参考3.4节。

构造函数/析构函数的机制是很多技术的基础，尤其是大多数C++通用资源管理技术（见11.2节）的基础。以下是一个Vector的图示：



构造函数负责为元素分配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。在构造函数中获取资源，然后在析构函数中释放它们，这种技术称为资源获取即初始化（Resource Acquisition Is Initialization, RAII），它使得我们避免使用“裸new操作”；换句话说，该技术可以防止在普通代码中分配内存，而是将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11.2节）。

4.2.3 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：

- 初始值列表构造函数（initializer-list constructor）：使用元素列表进行初始化；
- 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 double 值列表进行初始化
    // ...
    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 中
    return v;
}
```

上面的循环负责执行输入操作，它的终止条件是到达文件末尾或者遇到格式错误。在此之前，每个被读入的数依次添加到 `Vector` 的尾部，最后 `v` 的大小就是读入的元素数量。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。4.6.2 节将介绍移动构造函数，使用它我们就可以非常高效地从 `read()` 返回非常巨大的 `vector`。

38

用于定义初始值列表构造函数的 `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，我们可以这样书写：

```
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` 的初始值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{static_cast<int>(lst.size())}
{
    copy(lst.begin(),lst.end(),elem); // 从 lst 复制内容到 elem 中（见 10.6 节）
}
```

在上面的程序中，为了把初始值列表的大小转换成 `int` 类型，我们使用了糟糕的 `static_cast`（见 14.2.3 节）。这种写法比较呆板，因为一个手写的初始值列表的元素个数怎么也不可能超过整数所能表示的范围（16 位整数可以表示 32 767，32 位整数可以表示 2 147 483 647）。但是要记住类型系统是没有判断力的，它只知道变量的可能取值而非精确值，所以有时候它会无中生有地报告一些错误，然而对于程序员来说，这些报警信息迟早会发挥作用，它会防止程序发生特别严重的错误。

`static_cast` 本身并不负责检查要转换的值，它认为程序员自己知道应该如何正确地使用。这个假设显然不会一直成立，所以程序员如果不确定值是否合法，记得检查它。最好避免使用显式类型转换（通常称为强制类型转换（`cast`）以提醒人们为了实现转换所付出的努力），如果程序员学会善用类型系统和设计良好的标准库，他们就能在位于上层的软件系统中消除未经检查的类型转换。

4.3 抽象类型

`complex` 和 `Vector` 等类型之所以被称为具体类型 (concrete type), 是因为它们的实现属于定义的一部分。在这一点上, 它们与内置类型很相似。相反, 抽象类型 (abstract type) 则把使用者与类的实现细节完全隔离开来。为此, 我们分离接口与实现并且放弃了纯局部变量。因为我们对抽象类型的实现一无所知 (甚至对它的大小也不了解), 所以必须从自由存储 (见 4.2.2 节) 为对象分配空间, 然后通过引用或指针 (见 1.8 节, 11.2.1 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 4.2.1 节)
    virtual ~Container() {}               // 析构函数 (见 4.2.2 节)
};
```

对于后面定义的那些特定容器来说, 上面这个类纯粹是个接口。关键字 `virtual` 的意思是“可能在随后的派生类中被重新定义”。毫无疑问, 我们把这种用关键字 `virtual` 声明的函数称为虚函数 (virtual function)。Container 类的派生类负责为 Container 的接口提供具体实现。奇怪的 `=0` 语法说明该函数是纯虚函数 (pure virtual), 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()` 函数。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。一个常用来为其他类型提供接口的类, 我们把它称为多态类型 (polymorphic type)。

作为一个抽象类, 在 Container 中没有构造函数, 毕竟它不需要初始化数据。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 4.5 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` 类派生自 (derived) `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。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()` 被它所属类的析构函数 `~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 `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:

40

```

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`, 所以对于 `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 9.3 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, 类的实现是一个标准库 `list<double>`。一般情况下, 我们不会用 `list` 实现一个带下标操作的容器, 因为 `list` 取下标的性能很难与 `vector` 取下标相比。本例中我们只是用它完成了一个与之前完全不同的版本。

还可以通过一个函数创建 `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，也根本不需要知道，它可以使用任一种 `Container`。它只要了解 `Container` 定义的接口就可以了。因此，不论 `List_container` 的实现发生了改变或者我们使用了 `Container` 的一个全新派生类，都不需要重新编译 `use(Container&)`。

41 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 4.6 节，11.2.1 节）。

4.4 虚函数

我们进一步思考 `Container` 的用法：

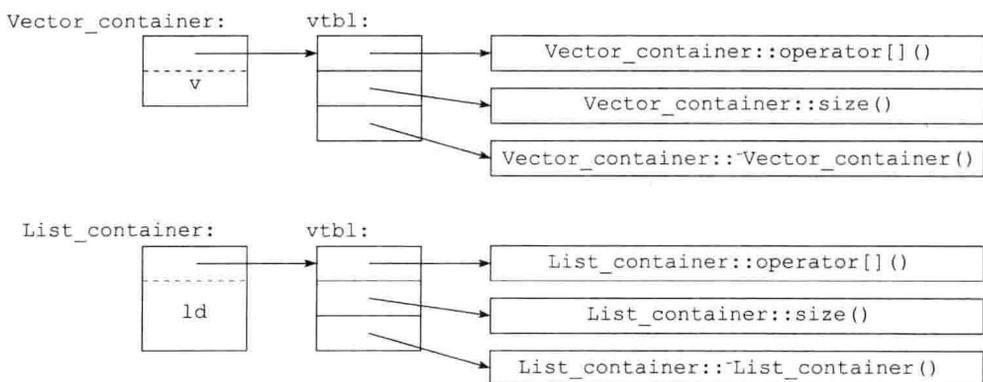
```

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

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

```

`use()` 中的 `c[i]` 是如何解析到正确的 `operator[]()` 的呢？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` 的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：

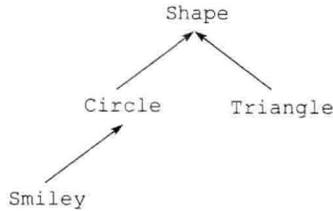


即使调用函数不清楚对象的大小和数据布局，`vtbl` 中的函数也能确保对象被正确使用。调用函数的实现只需要知道 `Container` 中 `vtbl` 指针的位置以及每个虚函数对应的索引就可以了。这种虚调用机制的效率非常接近“普通函数调用”机制（相差不超过 25%），而它的空间开销包括两部分：如果类包含虚函数，则该类的每个对象需要一个额外的指针；另外每个这样的类需要一个 `vtbl`。

4.5 类层次结构

Container 是一个非常简单的类层次结构的例子，所谓类层次结构（class hierarchy）是指通过派生（如 `:public`）创建的一组在框架中有序排列的类。我们使用类层次结构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“笑脸是一种圆，圆是一种形状”。在实际应用中，大的层次结构动辄包含上百个类，不论深度还是宽度都很大。不过本节中我们只考虑一个半真实半虚构的小例子，那就是屏幕上的形状：

42



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의 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，它们的实现基本上各不相同（除了 `vtbl` 指针的位置）。基于上面的定义，我们就能编写操纵一组形状指针的通用函数了：

```

void rotate_all(vector<Shape*> &v, int angle) // 把 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
};
  
```

43

到目前为止，Shape 和 Circle 示例涉及的语法知识并不比 Container 和 Vector_container 示例中涉及的多，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); // 眨眼次数为 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 的基类的 draw() 和 Smiley 的成员的 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 放在了标准库 vector 中，然后在析构函数里把它们释放。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了正确的析构函数，然后该析构函数再隐式调用其基类的析构函数和成员的析构函数。

在上面这个简单的例子中，程序员负责在表示笑脸的圆圈中恰当地放置眼睛和嘴。

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。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。

4.5.1 显式覆盖

如果派生类的某个函数与基类中虚函数的名字和类型都相同，则派生类的版本会覆盖基

类中的。在规模较大的层次结构中，有时候我们也搞不清到底该不该覆盖。派生类中会出现一些函数，它们的名字和类型与基类当中的差别微乎其微。有可能程序员本来希望它们覆盖的，只是不小心写错了；也有可能它们确实是不同的函数。为了避免造成这种混淆，程序员最好显式地指明一个函数是否覆盖另外的函数。举个例子来说，定义 `Smiley` 的语句可以等价地写成下面的形式：

```
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) override;

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

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

    // ...

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

在这段程序中，如果我们不小心把 `move` 写成了 `mov`，则程序将会报错，因为在 `Smiley` 的基类里根本不存在名为 `mov` 的虚函数。同样，如果我们不小心给 `wink()` 函数加上了 `override` 关键字，编译器也会报错。

45

4.5.2 层次结构的益处

类层次结构的益处主要体现在两个方面：

- 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。类层次结构中的类则有所区别：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 中读入形状描述信息
{
    // ... 从 is 中读取形状描述信息, 找到形状的种类 k...

    switch (k) {
    case Kind::circle:
        // 读取 circle 数据 {Point,int} 到 p 和 r
        return new Circle(p,r);
    case Kind::triangle:
        // 读取 triangle 数据 {Point,Point,Point} 到 p1, p2 和 p3
        return new Triangle(p1,p2,p3);
    case Kind::smiley:
        // 读取 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)      // 注意最后要删除掉元素
        delete p;
}

```

46

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`。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, 因此, `delete` 会调用最终的派生类的析构函数。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。

4.5.3 层次结构漫游

`read_shape()` 函数返回一个 `Shape*` 指针, 所以我们处理所有形状的方法都是类似的。如果我们想使用某个特定派生类的成员函数, 比如 `Smiley` 的 `wink()`, 则可以使用 `dynamic_cast` 运算符询问“这个 `Shape` 是一种 `Smiley` 吗?”:

```

Shape* ps {read_shape(cin)};

if (Smiley* p = dynamic_cast<Smiley*>(ps)) {
    // ... 指针 p 所指向的对象类型是 Smiley...
}
else {
    // ... 指针 p 所指向的对象类型不是 Smiley, 执行其他操作 ...
}

```

如果 `dynamic_cast` 的参数（此处是 `ps`）所指对象的类型与期望的类型（此处是 `Smiley`）或者期望类型的派生类不符，则 `dynamic_cast` 返回的结果是 `nullptr`。

当一个指向其他派生类的对象的指针是一个有效参数时，我们就能对指针类型使用 `dynamic_cast`。然后我们可以检验求值结果是否是 `nullptr`。这种用法常常用来在条件语句中初始化变量。

如果不能直接使用候选的派生类，可以用它的引用形式作为替代。当对象的类型与期望类型不符时，抛出 `bad_cast` 异常：

```
Shape* ps {read_shape(cin)};
Smiley& r {dynamic_cast<Smiley&>(*ps)}; // 有可能抛出 std::bad_cast 异常，需要在别的地方捕获并处理
```

适度使用 `dynamic_cast` 能让代码显得更简洁。如果我们能避免使用类型信息，就能写出更加简洁有效的代码。不过在某些情况下，缺失的类型信息必须被恢复出来，尤其是当我们把对象传递给某些系统，而这些系统只接受基类定义的接口时。当该系统稍后传回对象以供使用时，我们不得不恢复它原本的类型。与 `dynamic_cast` 类似的操作表达的含义有点像“是……的一种”或者“是……的一个实例”。

4.5.4 避免资源泄漏

有经验的程序员可能已经发现，上面的程序存在两处漏洞：

- 使用者可能不能用 `delete` 释放 `read_shape()` 返回的指针。
- `Shape` 指针容器的所有者可能无法用 `delete` 释放指针所指的对象。

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。

47

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 11.2.1 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:
        // 读取 circle 数据 {Point, int} 到 p 和 r
        return unique_ptr<Shape>{new Circle(p,r)}; // 见 11.2.1 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 5.5 节提供了另一种可选的方案。

4.6 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地复制，即依次复制每个成员。例如，使用 4.2.1 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，逐成员复制方式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，逐成员复制方式常常是不正确的，抽象类型更是如此。

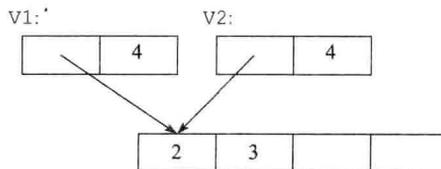
48

4.6.1 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。逐成员复制方式将违反资源句柄的不变式 (见 3.4.2 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 的成员变量复制给 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` 存在一个析构函数的事实强烈地预示着默认逐成员复制的语义是错误的，编译器至少应该给这个例子一个警告。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 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

    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：

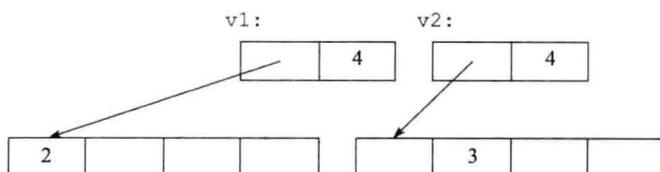
49

```

Vector::Vector(const Vector& a) // 拷贝构造函数
:elem(new double[a.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` 的结果现在可以表示成：



当然，除了拷贝构造函数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。

4.6.2 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。当我们给函数传递对象时，可通过使用引用类型来减少拷贝对象的代价，但是我们无法返回局部对象的引用（函数的调用者都没机会和返回结果碰面，局部对象就已经被销毁了）。以下的代码为例：

```

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;
}

```

50

要想从 + 运算符返回结果，需要把局部变量 res 的内容复制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 + 运算符一次）。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。最不合理的部分是 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：相比于拷贝（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。“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就是右值。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，所以我们可以安全地“窃取”它的值。Vector 的 operator+() 运算符的局部变量 res 就是一个例子。

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值或者赋值操作的右侧运算对象时，程序将使用移动操作。

51

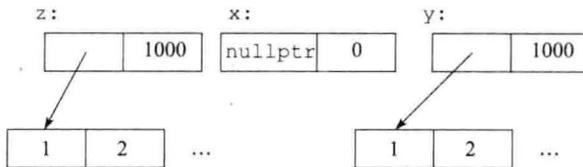
移动之后，源对象所进入的状态应该能允许运行析构函数。通常，我们也应该允许为一个移动操作后的源对象赋值。

程序员也许知道哪些地方不再使用某个值，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()` 不会真的移动什么，而是负责返回我们能移动的函数实参的引用——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）。

4.6.3 基本操作

在很多程序设计任务中，对象的构造方式都扮演着至关重要的角色，其用法的多样性完全依赖于语言对初始化过程支持的程度。

构造函数、析构函数、拷贝操作和移动操作在逻辑上有千丝万缕的联系，在定义这些函数时我们必须考虑这种内在的联系，否则就会遇到逻辑问题或者性能问题。如果类 `x` 的析构函数执行了某些特定的任务，比如释放自由存储空间或者释放锁，则该类也应该实现所有其他相关的函数：

```
class X {
public:
    X(Sometype);           // “普通的构造函数”：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：清空目标对象并拷贝
    X& operator=(X&&);    // 移动赋值运算符：清空目标对象并移动
    ~X();                 // 析构函数：清空资源
    // ...
};
```

52

在下面 5 种情况下，对象会被移动或拷贝：

- 被赋值给其他对象。
- 作为对象初始值。
- 作为函数的实参。
- 作为函数的返回值。
- 作为异常。

在上述所有情况下，将应用类的移动构造函数或拷贝构造函数（优化情形除外）。

除了用于命名对象和自由存储空间对象的初始化，构造函数还被用于初始化临时对象和实现显式类型转换。

编译器会根据需要生成上面这些成员函数，当然“普通的构造函数”除外。如果程序员希望显式地使用这些函数的默认实现，可以编写如下所示的代码：

```
class Y {
Public:
    Y(Sometype);
    Y(const Y&) = default; // 程序确实需要使用默认的拷贝构造函数
    Y(Y&&) = default;     // 程序确实需要使用默认的移动构造函数
    // ...
};
```

一旦显式地指定了某些函数的默认形式，编译器就不会再为函数生成其他默认定义了。

当类中含有指针或者引用类型的成员时，最好显式地指定拷贝操作和移动操作。如果不这样做，则当编译器生成的默认函数试图销毁指针或引用所指的對象时，系统将发生错误。即使有些对象我们不想销毁，也应该在函数当中指明这一点以便于读者理解。

接受单个参数的构造函数同时定义了从参数类型到类类型的转换。例如，`complex`（见 4.2.1 节）提供了一个接受 `double` 的构造函数：

```
complex z1 = 3.14; // z1 变成了 {3.14, 0.0}
complex z2 = z1*2; // z2 变成了 {6.28, 0.0}
```

显然这样的效果有时候合乎情理，有时候则不然。例如，`Vector`（见 4.2.2 节）提供了一个接受 `int` 的构造函数：

```
Vector v1 = 7; // OK: v1 含有 7 个元素
```

通常情况下该语句的实际执行结果并非我们的预期，标准库 `vector` 禁止这种 `int` 到 `vector` 的“转换”。

解决该问题的办法是只允许显式“类型转换”，也就是说我们把构造函数定义成下面的形式：

```
class Vector {
public:
    explicit Vector(int s); // 禁止 int 到 Vector 的隐式类型转换
    // ...
};
```

在修改了构造函数的定义后：

```
Vector v1(7); // OK: v1 含有 7 个元素
Vector v2 = 7; // 错误: 禁止 int 到 Vector 的隐式类型转换
```

关于类型转换的问题，大多数类的情况与 `Vector` 类似，`complex` 则只能代表一小部分。因此除非你有充分理由，否则最好把接受单个参数的构造函数声明成 `explicit` 的。

4.6.4 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）的生命周期进行完全控制。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进行简单高效的移动了。以表示并发活动的标准库 `thread`（见 13.2 节）和含有上百万个 `double` 的 `Vector` 为例，前者“不能”执行拷贝操作，而后者我们则“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(10000);             // 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 11.2.1 节）。

我们使用标准库 `vector` 存放 `thread`，因为在 5.2 节之前，我们还找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，我们也可以将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的风险。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`。

很多编程语言都把资源管理的任务委托给了垃圾回收器，C++ 同样提供了一个垃圾回收接口以供程序员使用。不过对于资源管理而言，建议优先使用更干净、通用也更好的局部化的处理措施，然后再考虑系统提供的垃圾回收机制。

本质上来说，垃圾回收是一种全局内存管理模式。适当使用当然没有问题，不过随着系统的分布式趋势（比如多核、缓存以及集群）日益明显，在局部范围内管理资源变得越来越重要了。

同时，内存也不是唯一一种资源。资源是指任何具有“获取→使用→（显式或隐式）释放”模式的东西，比如内存、锁、套接字、文件句柄和线程句柄等。一个好的资源管理系统

应该能够处理全部资源类型。任何长时间运行的系统都应该尽量避免资源泄漏，但从另一方面来说，过度占用资源和资源泄漏一样糟糕。例如，如果一个系统可能占用两倍的内存、锁、文件句柄等资源，则系统就必须储备两倍容量的资源以供使用。

在不得不求助于垃圾回收机制之前，优先使用资源句柄：让所有资源都在某个作用域内有所归属，并且在作用域结束的地方默认地释放资源。在 C++ 中，这被称为 RAII (Resource Acquisition Is Initialization, 资源获取即初始化)，它与错误处理一道组成了异常机制。我们使用移动构造函数或者“智能指针”把资源从一个作用域移动到另一个作用域，使用“共享指针”分享资源的所有权 (见 11.2.1 节)。

在 C++ 标准库中，RAII 无处不在：例如内存 (string、vector、map、unordered_map 等)、文件 (ifstream、ofstream 等)、线程 (thread)、锁 (lock_guard、unique_lock 等) 和通用对象 (通过 unique_ptr 和 shared_ptr 访问)。在日常应用中程序员也许无法察觉隐式资源管理在发挥作用，然而它确实使资源的实际占有时间被大大降低了。

4.6.5 抑制操作

对于处在层次结构中的类来说，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 4.3 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();
    // ...
};
```

编译器将捕获拷贝 Shape 的意图。如果你确实希望拷贝类层次结构中的某个对象，可以编写一个虚的 (virtual) 克隆函数。

在这个特殊的例子中，即使忘了删除 (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，此时试图移动 Shape 对象会造成编译器错误。在本例中拷贝操作的生成也被禁止了 (见 14.2.3 节)，因此试图拷贝 Shape 对象也会使得编译器报错。

在有些情况下我们不希望拷贝类的对象，类层次结构中的基类就是一个例子。通常，我们无法通过拷贝成员的方式来拷贝资源句柄 (见 4.6.1 节)。

55 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。

4.7 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 16 ~ 22 章有更加详细的描述。
- [2] 程序员应该直接用代码表达思想；参见 4.1 节。
- [3] 具体类是最简单的类。与复杂类或者普通数据结构相比，请优先选择使用具体类；参见 4.2 节。
- [4] 使用具体类表示简单的概念以及性能要求较高的组件；参见 4.2 节。
- [5] 定义一个构造函数来处理对象的初始化操作；参见 4.2.1 节，4.6.3 节。
- [6] 只有当函数确实需要直接访问类的成员变量部分时，才把它作为成员函数；参见 4.2.1 节。
- [7] 定义运算符的目的主要是模仿和借鉴它的经典用法；参见 4.2.1 节。
- [8] 把对称的运算符定义成非成员函数；参见 4.2.1 节。
- [9] 如果成员函数不会改变对象的状态，则应该把它声明成 `const` 的；参见 4.2.1 节。
- [10] 如果类的构造函数获取了资源，那么需要使用析构函数释放这些资源；参见 4.2.2 节。
- [11] 避免“裸” `new` 和“裸” `delete` 操作；参见 4.2.2 节。
- [12] 使用资源句柄和 RAII 管理资源；参见 4.2.2 节。
- [13] 如果类是一个容器，给它一个初始值列表构造函数；参见 4.2.3 节。
- [14] 如果需要把接口和实现完全分离开来，则使用抽象类作为接口；参见 4.3 节。
- [15] 使用指针和引用访问多态对象；参见 4.3 节。
- [16] 抽象类通常无需构造函数；参见 4.3 节。
- [17] 使用类的层次结构表示具有继承层次结构的一组概念；参见 4.5 节。
- [18] 含有虚函数的类应该同时包含一个虚的析构函数；参见 4.5 节。
- [19] 在规模较大的类层次结构中使用 `override` 显式地指明函数覆盖；参见 4.5.1 节。
- [20] 当设计类的层次结构时，注意区分实现继承和接口继承；参见 4.5.2 节。
- [21] 当类层次结构漫游不可避免时记得使用 `dynamic_cast`；参见 4.5.3 节。
- [22] 如果想在无法转换到目标类时报错，则令 `dynamic_cast` 作用于引用类型；参见 4.5.3 节。
- [23] 如果认为即使无法转换到目标类也可以接受，则令 `dynamic_cast` 作用于指针类型；参见 4.5.3 节。
- [24] 为了防止忘掉用 `delete` 销毁用 `new` 创建的对象，建议使用 `unique_ptr` 或者 `shared_ptr`；参见 4.5.4 节。
- [25] 如果默认的拷贝操作不适合当前类，记得重新定义一个或者干脆禁止使用它；参见 4.6.1 节，4.6.5 节。
- [26] 用传值的方式返回容器（移动而非拷贝容器以提高效率）；参见 4.6.2 节。
- [27] 对于容量较大的操作对象，使用 `const` 引用参数类型；参见 4.6.2 节。
- [28] 如果类含有析构函数，则该类很可能需要自定义或者删除移动和拷贝操作；参见 4.6.5 节。
- [29] 尽量让对象的构造、拷贝、移动和析构操作在掌控之中；参见 4.6.3 节。

- [30] 设计构造函数、赋值运算符和析构函数时应该全盘考虑，使之成为一体；参见 4.6.3 节。
- [31] 如果默认的构造函数、赋值运算符和析构函数符合要求，则让编译器负责生成它们，用户没必要再定义一遍；参见 4.6.3 节。
- [32] 默认情况下，把接受单参数的构造函数声明成 `explicit` 的；参见 4.6.3 节。
- [33] 如果类含有指针或引用类型的成员，则它需要一个析构函数以及非默认的拷贝操作；参见 4.6.3 节。
- [34] 确保提供强有力的资源安全保障，也就是说，永不泄漏任何资源；参见 4.6.4 节。
- [35] 如果一个类被用作资源句柄，则需要为它提供构造函数、析构函数和非默认的拷贝操作；参见 4.6.4 节。

模 板

这里是畅所欲言之地。
——本贾尼·斯特劳斯特卢普

- 引言
- 参数化类型
- 函数模板
- 概念和泛型编程
- 函数对象
- 可变参数模板
- 别名
- 模板编译模型
- 建议

5.1 引言

显然，人们在使用向量时不一定总是使用 `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。模板（`template`）是我们用一组类型或值对其进行参数化的一个类或一个函数。我们使用模板表示那些通用的概念，然后通过指定参数（比如指定元素的类型为 `double`）生成特定的类型或函数。

59

5.2 参数化类型

对于我们之前使用的 `double` 类型的向量，只要将其改为 `template` 并且用一个形参替换掉特定类型 `double`，就能泛化为任意类型的向量。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:
    explicit 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`”或更准确说是“对所有类型 `T`”的 C++ 表达。在引出类型参数时，使用 `class` 和使用 `typename` 是等价的，旧式代码当中经常出现 `template<class T>` 作为前缀。

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，我们可以按如下方式定义 `Vector`：

```

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，它并不是被放错了地方的输入运算符，不用像 C++98 中那样在两个 `>` 之间加个空格。

一个模板以及模板参数的集合被称为实例化。每一个程序中使用的实例化的代码在程序编译的后期，也就是实例化期被产生出来。这就是错误消息在很晚才产生出来并且难以理解的原因。

60

我们可以按如下方式使用 `Vector`：

```

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

```

为了让 `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.size() ? &x[0] : nullptr;      // 指针指向第一个元素或 nullptr
}

template<typename T>
T* end(Vector<T>& x)
{

```

```
    return begin(x)+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(Vector<string>& vs) // 字符串组成的 Vector
{
    for (auto& s : vs)
        cout << s << '\n';
}
```

类似地，我们也能将链表（list）、向量（vector）、映射（map）（也就是关联数组）、无序映射（也就是哈希表）等定义成模板（见第 9 章）。

模板是一种编译时的机制，因此与“手工编码”相比，并不会产生任何额外的运行时开销。事实上，`Vector<double>` 生成的代码和第 4 章的 `Vector` 生成的代码完全一致，而标准库 `vector<double>` 生成的代码则要更优（因为在它的实现过程中做了很多优化工作）。

除了类型参数外，模板也可以接受普通的值参数。例如：

```
template<typename T, int N>
struct Buffer {
    using value_type = T;
    constexpr int size() { return N; }
    T[N];
    // ...
};
```

其中的别名 `value_type` 和 `constexpr` 函数使得用户可以只读地访问模板参数。

值参数在很多情况下都非常有用。例如，我们可以用上面提供的 `Buffer` 创建任意大小的缓冲区，并且避免了使用自由存储（动态内存分配）所带来的开销：

```
Buffer<char,1024> glob; // 全局的字符缓冲区（静态分配）

void fct()
{
    Buffer<int,10> buf; // 局部的整数缓冲区（在栈上）
    // ...
}
```

只有常量表达式能用于模板的值参数。

5.3 函数模板

模板的用途远不只是用元素类型参数化容器，其中一项很重要的用途是参数化标准库中的类型和算法（见 9.6 节，10.6 节）。例如，下面这段程序可以用来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（累加双精度型浮点数）
    double dd = sum(ld,0.0);    // 求双精度型浮点数列表的和
    auto z = sum(vc,complex<double>{}); // 求 complex<double> 向量的和，初始值是 {0.0,0.0}
}

```

把一些 `int` 值累加到一个 `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。幸运的是我们无需显式地指定这些类型。

这里的 `sum()` 可以看作是标准库 `accumulate()`（见 12.3 节）的简化版本。

5.4 概念和泛型编程

应该把模板用在哪儿呢？换句话说，模板会让哪些程序设计技术更有效呢？模板提供了以下功能：

- 把类型（以及数值和模板）作为实参传递而不损失任何信息的能力。这为内联提供了很多便利，而现有的实现可以很好地利用这一点。
- 延迟的类型检查（在实例化时执行）。这意味着程序可以把多个上下文的有用信息捏合在一起。
- 把常量值作为实参传递的能力，也就是在编译时计算的能力。

总而言之，模板为编译时计算和类型控制提供了强有力的辅助支持，使得我们可以编写出更加简洁高效的代码。

模板最常见的应用是支持泛型编程（generic programming），泛型编程主要关注通用算法的设计、实现和使用。在这里，“通用”的含义是该算法可以支持很多种数据类型，只要这些类型符合算法逻辑的要求即可。模板是 C++ 支持泛型编程最重要的工具，提供了编译时的参数级多态性。

回顾 5.3 节的 `sum()` 函数。只要任意数据结构支持范围 `for` 循环所需的 `begin()` 和 `end()`，我们就能调用 `sum()`，显然标准库 `vector`、`list` 和 `map` 都在此列。进一步，对这些结构中元素类型的唯一要求是满足其用法：这种类型必须能被加到实参 `Value` 上，`int`、`double` 和 `Matrix`（任何合理的 `Matrix`）都满足该要求。我们认为 `sum()` 的泛化能力包含两个维度：存储元素的数据结构（容器）以及容器中元素的类型。

`sum()` 要求它的第一个模板实参是某种容器，第二个模板实参是某个数字，我们把这种要求称作概念（concept）。不幸的是，C++11 不提供对概念的直接支持，我们只能说 `sum()` 的模板实参必须是某种数据类型。另外两本著作 [Stroustrup, 2013] 和 [Sutton, 2012] 介绍了一些检查概念的技术以及关于 C++ 该如何为概念提供直接支持的建议，但本书的篇幅有限，无法详细介绍了。

毋庸讳言，好的、有用的概念是泛型编程的基础，而这些概念往往是被发现而不是被设计出来的。比如整数和浮点数（如经典 C 语言中的定义），以及更通用的数学概念如域、向量

空间，还有容器等等。它们表达了某个应用领域的基本概念。实际上，甄选和形式化概念以达到能够进行有效泛型编程的程度通常是一项很大的挑战。

让我们想一想“规范”（regular）这个概念。如果某种类型的特点和使用方式与 `int` 或 `vector` 非常像，我们就说这种类型是规范的。规范类型的对象应该：

- 能以默认的方式构造。
- 能以构造函数或赋值运算符的方式拷贝（当然要确保拷贝之后源对象和目标对象相互独立且等价）。
- 能用 `==` 和 `!=` 进行比较。
- 即使用在复杂的程序结构中也不会出错。

`string` 是另一种典型的规范类型，并且它和 `int` 一样是有序的（ordered），这意味着两个字符串可以用 `<`、`<=`、`>` 和 `>=` 等运算符进行合适的语义比较。概念不仅仅与语法有关，它还包含很多语义的要求。例如，永远别让符号 + 执行除法运算，这会造成意想不到的错误。 63

5.5 函数对象

模板的一个特殊用途是函数对象（function object，有时也被称为 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 `()`。

我们能为某些参数类型定义 `Less_than` 类型的命名变量：

```
Less_than<int> lti{42}; // lti(i) 将用 < 比较 i 和 42 (i<42)
Less_than<string> lts{"Backus"}; // lts(s) 将用 < 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参数出现。例如，我们可以像下面这样统计有多少个值令谓词返回 `true`：

```
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;
}

```

谓词 (predicate) 是调用时返回 true 或者 false 的对象。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';
}

```

64 其中, `Less_than<int>(x)` 构造的对象可以通过 `()` 操作符与名为 `x` 的 `int` 比较, 而 `Less_than<string>(s)` 构造的对象则可以通过同样的方式与名为 `s` 的 `string` 比较。函数对象的精妙之处在于它们附着着准备与之进行比较的值。我们无需为每个值 (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。可携带数据和高效这两个特性使得我们经常使用函数对象作为算法的参数。

用于指明通用算法的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), 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, 但也有可能显得晦涩难懂。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, 以便于更加清晰地表述它的目的以及在程序中随处使用它。

在 4.5.4 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<typename C, typename Oper>
void for_all(C& c, Oper op)      // 假定 C 是一个指针容器
{
    for (auto& x : c)
        op(x);                // 传给 op() 每个元素所指对象的引用
}
```

接下来，我们改写 4.5 节中的 `user()`，而无需编写一大堆 `_all()` 函数：

65

```
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)
}
```

我们给 `lambda` 传入 `Shape` 的引用，这样 `lambda` 就无需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。

5.6 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：

```
void f() {}      // 不执行任何操作

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
}
```

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，我们需要另外一个函数处理它。

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`）：

66

```
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，仅用三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。

因为模板具有很强的灵活性，所以它在标准库中被广泛使用。

5.7 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。例如，标准库头文件 `<cstdint>` 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`。使用别名 `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（见第 9 章），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; // C 的元素类型

template<typename Container>
void algo(Container& c)
{
    Vector<Element_type<Container>> vec; // 保存结果
    // ...
}
```

67

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了。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>
```

5.8 模板编译模型

为模板提供的类型检查机制负责检查模板定义中而非模板的显式接口中（在模板声明中）的参数使用情况。这种机制提供了被称为鸭子类型（duck typing，如果有种东西走路像鸭子、叫声像鸭子，那它就是鸭子）的编译时特性。它的含义也可以用比较技术性的词汇重新描述：当我们执行某种操作时，操作的效果和含义完全依赖于其操作对象的值。这一点与我们熟悉的“对象拥有某种数据类型”的视角完全不同，在那里是对象的类型决定操作的效果和含义。在 C++ 中，值“存在”于对象内，这是 C++ 对象（如变量）的工作机制。某个值只有符合对象的需求，才能放在对象当中。对于模板来说，编译时实际使用的是值，而非对象。

最直接的影响是：当我们使用模板时，模板的定义（不仅是声明）必须在作用域内。例如，标准库头文件 <vector> 含有 vector 的定义，这时会产生副作用。因为编译器需要综合程序中好几个地方的信息才能进行判断，所以编译过程可能很晚才报告一个类型错误并给出非常糟糕的错误信息。

5.9 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 20 ~ 29 章有更加详细的描述。
- [2] 用模板来表达那些可以作用于多种数据类型的算法；参见 5.1 节。
- [3] 用模板实现容器；参见 5.2 节。
- [4] 用模板提升代码的抽象水平；参见 5.2 节。
- [5] 定义模板时，最好先设计和调试出一个非模板版本，然后再通过添加参数进行泛化。
- [6] 模板是类型安全的，但是对类型的检查很晚才开始；参见 5.4 节。
- [7] 模板可以无损地传递参数类型。
- [8] 用函数模板推断类模板参数类型；参见 5.3 节。
- [9] 模板为编译时程序设计方法提供了通用机制；参见 5.4 节。
- [10] 设计模板时，需要谨慎考虑为模板参数设定的概念（必要的）；参见 5.4 节。
- [11] 把概念作为设计工具；参见 5.4 节。

- [12] 把函数对象作为算法的参数；参见 5.5 节。
- [13] 如果只在某处需要一个简单的函数对象，不妨使用 lambda 表达式；参见 5.5 节。
- [14] 不能把虚成员函数定义成模板成员函数。
- [15] 利用模板别名来简化表示方式并隐藏细节；参见 5.7 节。
- [16] 当函数参数的类型和数量都无法确定时，使用可变参数模板；参见 5.6 节。
- [17] 不要用可变参数模板处理同类型的参数（使用初始值列表）；参见 5.6 节。
- [18] 使用模板时要确保它的定义（不仅是声明）位于作用域内；参见 5.8 节。
- [19] 模板提供了编译时的“鸭子类型”；参见 5.8 节。
- [20] 模板不存在分离式编译：用到模板的地方都应该用 `#include` 包含模板的定义。

69
70

标准库概览

当无知只是瞬间，又何必浪费时间学习呢？

——霍布斯（漫画人物）

- 引言
- 标准库组件
- 标准库头文件和命名空间
- 建议

6.1 引言

没有任何一个重要的程序是仅仅用“裸语言”写成的。人们通常先开发出一系列库，随后将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，大多数情况下写起来非常乏味。而如果使用很好的程序库，几乎所有编程工作都会变得简单许多。

接着第 1 ~ 5 章、第 6 ~ 13 章将对重要的标准库工具和方法给出一个概要性的介绍。

我将简要介绍有用的标准库类型，如 `string`、`ostream`、`vector`、`map`、`unique_ptr`、`thread`、`regex` 和 `complex`，并介绍它们最常见的使用方法。如同在第 1 ~ 5 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。本章的目的是介绍那些最有用的标准库工具和方法相关的基本知识，而不是对它们进行详细介绍。

在 ISO C++ 标准中，标准库规范几乎占了三分之二篇幅。你应深入了解并尽量使用标准库来解决问题，而不是使用自制的替代品来编写程序。因为，标准库的设计中已经凝结了很多思想，还有很多思想体现在其实现中，未来，还会有大量的精力投入到其维护和扩展中。

本书介绍的标准库工具和方法，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、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”，以提供企业级或工业级的“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。

71

本书的目标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。当然，我们鼓励程序员去探索那些常见的非 C++ 标准的工具和方法。

6.2 标准库组件

标准库提供的工具和方法可以分为如下几类：

- 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）。
- C 标准库（进行了非常小的修改，以便尽量减少与类型系统的冲突）。
- 字符串（包括对国际字符和本地化的支持）；见 7.2 节。
- 对正则表达式匹配的支持；见 7.3 节。
- I/O 流，这是一个可扩展的输入输出框架，用户可向其中添加自己设计的类型、流、缓冲策略、区域设定和字符集。
- 容器（如 `vector` 和 `map`）和算法（如 `find()`、`sort()` 和 `merge()`）；见第 9 章和第 10 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 4.1.2 节和第 12 章。
- 对并发程序设计的支持，包括 `thread` 和锁机制；见第 13 章。在此基础上，用户就能够以库的形式添加新的并发模型。
- 支持模板元程序设计的工具（如类型特性；见 11.6 节）、STL 风格的泛型程序设计（如 `pair`；见 11.3.3 节）和通用程序设计（如 `clock`；见 11.4 节）。
- 用于资源管理的“智能指针”（如 `unique_ptr` 和 `shared_ptr`；见 11.2.1 节）和垃圾回收器接口（见 4.6.4 节）。
- 特殊用途容器，例如 `array`（见 11.3.1 节）、`bitset`（见 11.3.2 节）和 `tuple`（见 11.3.3 节）。

判断是否应该将一个类纳入标准库的主要标准包括：

- 它几乎对所有 C++ 程序员（包括初学者和专家）都有用。
- 它能以通用的形式提供给程序员，与简单版本相比，这种通用形式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，C++ 标准库提供了最常用的基本数据结构以及其上的基础算法。

6.3 标准库头文件和命名空间

每个标准库工具和方法都是通过若干标准库头文件提供的，例如：

```
#include<string>
#include<list>
```

72 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` 的命名空间中（见 3.3 节）。要使用标准库工具和方法，可以使用 `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，书中的例子很少显式使用 `std::` 前缀，也不会显式给出 `#include` 语句包含必要的头文件。要正确编译运行本书中的程序片段，必须补上 `#include` 语句包含恰当的头文件，并通过 `std::` 前缀等方式令标准库名字可用。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: string 是 std::string
```

这段代码将一个命名空间 (`std`) 中的所有名字都曝露到了全局命名空间中，一般来说这并不是一个好的编程习惯。但是，在本书中仅使用标准库，可以明确地知道它提供什么。

下表是一些挑选出的标准库头文件，其中的声明都放在命名空间 `std` 中：

挑选出的标准库头文件			
<code><algorithm></code>	<code>copy()</code> , <code>find()</code> , <code>sort()</code>	第 10 章	iso.25
<code><array></code>	<code>array</code>	11.3.1 节	iso.23.3.2
<code><chrono></code>	<code>duration</code> , <code>time_point</code>	11.4 节	iso.20.11.2
<code><cmath></code>	<code>sqrt()</code> , <code>pow()</code>	12.2 节	iso.26.8
<code><complex></code>	<code>complex</code> , <code>sqrt()</code> , <code>pow()</code>	12.4 节	iso.26.8
<code><forward_list></code>	<code>forward_list</code>	9.6 节	iso.23.3.4
<code><fstream></code>	<code>fstream</code> , <code>ifstream</code> , <code>ofstream</code>	8.7 节	iso.27.9.1
<code><future></code>	<code>future</code> , <code>promise</code>	13.7 节	iso.30.6
<code><ios></code>	<code>hex</code> , <code>dec</code> , <code>scientific</code> , <code>fixed</code> , <code>defaultfloat</code>	8.6 节	iso.27.5
<code><iostream></code>	<code>istream</code> , <code>ostream</code> , <code>cin</code> , <code>cout</code>	第 8 章	iso.25.7.4
<code><map></code>	<code>map</code> , <code>multimap</code>	9.5 节	iso.23.4.4
<code><memory></code>	<code>unique_ptr</code> , <code>shared_ptr</code> , <code>allocator</code>	11.2.1 节	iso.20.6
<code><random></code>	<code>default_random_engine</code> , <code>normal_distribution</code>	12.5 节	iso.26.5
<code><regex></code>	<code>regex</code> , <code>smatch</code>	7.3 节	iso.28.8
<code><string></code>	<code>string</code> , <code>basic_string</code>	7.2 节	iso.21.3
<code><set></code>	<code>set</code> , <code>multiset</code>	9.6 节	iso.23.4.6
<code><sstream></code>	<code>istrstream</code> , <code>ostrstream</code>	8.8 节	iso.27.8
<code><stdexcept></code>	<code>length_error</code> , <code>out_of_range</code> , <code>runtime_error</code>	3.4.1 节	iso.19.2
<code><thread></code>	<code>thread</code>	13.2 节	iso.30.3
<code><unordered_map></code>	<code>unordered_map</code> , <code>unordered_multimap</code>	9.5 节	iso.23.5.4
<code><utility></code>	<code>move()</code> , <code>swap()</code> , <code>pair</code>	第 11 章	iso.20.1
<code><vector></code>	<code>vector</code>	9.2 节	iso.23.3.6

此列表远未囊括所有标准库头文件。

C++ 标准库中也提供了来自 C 标准库的头文件，如 `<stdlib.h>`。这类头文件都有一个对应的版本，名字加上了前缀 `c` 并去掉了后缀 `.h`，如 `<cstdlib>`。这些对应版本中的声明都放在命名空间 `std` 中。

6.4 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 30 章中有更加详细的描述。
- [2] 不要重新发明轮子，应该使用库；参见 6.1 节。
- [3] 当有多种选择时，优先选择标准库而不是其他库；参见 6.1 节。
- [4] 不要认为标准库在任何情况下都是理想之选；参见 6.1 节。
- [5] 当使用标准库工具和方法时，记得用 `#include` 包含相应的头文件；参见 6.1 节。
- [74] [6] 记住，标准库工具和方法都定义在命名空间 `std` 中；参见 6.3 节。

字符串和正则表达式

优先选择标准而非另类。

——斯特伦克&怀特

- 引言
- 字符串
 - string 的实现
- 正则表达式
 - 搜索；正则表达式符号表示；迭代器
- 建议

7.1 引言

在大多数程序中，文本处理都是重要组成部分。C++ 标准库提供了 `string` 类型，使得程序员不必再使用 C 风格的文本处理方式——通过指针来处理字符数组。此外，C++ 标准库还提供了正则表达式匹配功能以查找文本中的特定模式。C++ 标准库中的正则表达式与大多数编程语言提供的正则表达式类似。`string` 和 `regex` 都支持多种字符类型（如 Unicode（万国码））。

7.2 字符串

标准库提供了 `string` 类型，具有比简单字符串字面值常量（见 1.3 节）更完整的字符串处理能力。`string` 类型提供了很多有用的字符串处理函数，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`、一个字符串字面值常量、一个 C 风格字符串或是一个字符连接到一个 `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 4.6.2 节）。

在很多应用中，连接操作最常见的用法是在一个 `string` 的末尾追加一些内容。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 对象是可修改的。除了 = 和 += 外，string 还支持下标操作（使用 []）和提取子串操作。在其他有用的操作中，string 还提供了直接处理子串的操作。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，保存其参数指定的子字符串的拷贝。第一个参数是指向 string 中某个位置的下标，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，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 可以相互比较，也可以与字符串字面值常量比较，例如：

```
string incantation;

void respond(const string& answer)
{
    if (answer == incantation) {
        // 执行一些操作
    }
    else if (answer == "yes") {
        // ...
    }
    // ...
}
```

76

string 提供了很多有用的操作，其中比较常用的有赋值（使用 =）、下标（使用 [] 或 at()，与 vector 类似；见 9.2.2 节）、迭代（像 vector 那样使用迭代器；见 10.2 节）、输入（见 8.3 节）、流操作（见 8.8 节）。

如果你需要一个 C 风格字符串（一个以 0 结尾的 char 数组），string 提供了对其包含的 C 风格字符串进行只读访问的接口，例如：

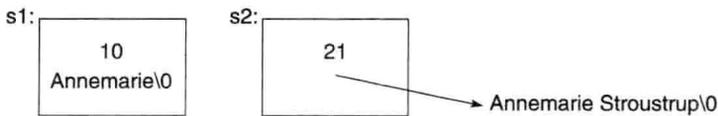
```
void print(const string& s)
{
    printf("For people who like printf: %s\n",s.c_str());
    cout << "For people who like streams: " << s << '\n';
}
```

7.2.1 string 的实现

实现字符串类是一个常见的 C++ 编程练习，这个练习对提高编程能力是很有帮助的。但对一般用途而言，我们第一次尝试编写这样的类，即使精心构思，在易用性和性能上也很难与标准库 `string` 相比。在当前的 `string` 实现版本中，通常会使用短字符串优化（`short-string optimization`）技术。即短字符串会直接保存在 `string` 对象内部，而长字符串则保存在自由存储区中。考虑下面的例子：

```
string s1 {"Annemarie"};           // 短字符串
string s2 {"Annemarie Stroustrup"}; // 长字符串
```

内存布局可能像下面这样：



当一个 `string` 由短变长（或相反）时，它的表示（存储）形式会相应地调整。

`string` 的实际性能严重依赖运行时环境。特别是在多线程实现中，内存分配操作的代价相对较高；而且，当程序中使用大量长度不一的字符串时，会产生内存碎片问题。这些都是短字符串优化技术被普遍采用的主要原因。

为了处理多字符集，标准库定义了一个通用的字符串模板 `basic_string`，`string` 实际上是此模板用字符类型 `char` 实例化的一个别名：

```
template<typename Char>
class basic_string {
    // ... Char 类型的字符串
};

using string = basic_string<char>
```

用户可以定义任意字符类型的字符串。例如，假定我们有一个日文字符类型 `Jchar`，则可定义为：

```
using Jstring = basic_string<Jchar>;
```

77

现在，我们就可以在 `Jstring`——日文字符串上执行常见的字符串操作。与此类似，我们也可以处理 Unicode 字符串。

7.3 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、精炼的方法描述文本中的模式（如，形如 TX 77845 的美国邮政编码，或形如 2009-06-07 的 ISO 风格的日期），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：XXdddd-dddd 及其变形
```

在其他语言中使用过正则表达式的人会觉得 `\w{2}\s*d{5}(-d{4})?` 很熟悉。它指定了一个以两个字母开始 (`\w{2}`) 的模式，后面是任意个空白符 `\s*`，再接下来是五个数字 `\d{5}`，然后是可选的一个破折号和四个数字 `-d{4}`。如果你还不熟悉正则表达式，现在可能是一个学习它的好时机 ([Stroustrup, 2009], [Maddock, 2009], [Friedl, 1997])。

为了表达模式，我使用了一个原始字符串字面值常量 (raw string literal)，它以 `R"` (开始，以 `"`) 结束。原始字符串字面值常量的好处是可以直接包含反斜线和引号而无需转义，因此非常适合表示正则表达式——因为其中常常包含大量反斜线。如果使用常规字符串，模式定义需要写成这样：

```
regex pat {"\\w{2}\\s*d{5}(-d{4})?"; // 美国邮政编码模式
```

在 `<regex>` 中，标准库为正则表达式提供了如下支持：

- `regex_match()`：将正则表达式与一个 (已知长度的) 字符串进行匹配 (见 7.3.2 节)。
- `regex_search()`：在一个 (任意长的) 数据流中搜索与正则表达式匹配的字符串 (见 7.3.1 节)。
- `regex_replace()`：在一个 (任意长的) 数据流中搜索与正则表达式匹配的字符串并将其替换。
- `regex_iterator`：遍历匹配结果和子匹配 (见 7.3.3 节)。
- `regex_token_iterator`：遍历未匹配部分。

7.3.1 搜索

使用模式的最简单方式是在流中搜索它：

```
int lineno = 0;
for (string line; getline(cin,line);) { // 读入缓冲区 line 中
    ++lineno;
    smatch matches; // 匹配结果保存在这里
    if (regex_search(line,matches,pat)) // 在 line 中搜索 pat
        cout << lineno << " : " << matches[0] << '\n';
}
```

`regex_search(line,matches,pat)` 在 `line` 中搜索任何与正则表达式 `pat` 匹配的子串，如果找到匹配子串，就将其保存在 `matches` 中。如果未找到任何匹配，`regex_search(line,matches,pat)` 返回 `false`。变量 `matches` 的类型是 `smatch`。开头的 “s” 表示 “子” 或 “字符串” 的意思，一个 `smatch` 实质上是一个 `string` 的 `vector`，每个 `string` 保存的是一个子匹配。首元素 `matches[0]` 对应整个匹配。`regex_search()` 返回的是一组 (子) 匹配，通常表示为一个 `smatch` 对象。

```

void use()
{
    ifstream in("file.txt"); // 输入文件
    if (!in) // 检查文件是否正确打开
        cerr << "no file\n";

    regex pat {R"(w{2})s-d{5}(-d{4})?"}; // 美国邮政编码模式

    int lineno = 0;
    for (string line; getline(in,line); ) {
        ++lineno;
        smatch matches; // 匹配结果保存在这里
        if (regex_search(line, matches, pat)) {
            cout << lineno << ": " << matches[0] << '\n'; // 完整匹配
            if (1 < matches.size() && matches[1].matched)
                cout << "\t: " << matches[1] << '\n'; // 子匹配
        }
    }
}

```

此函数读取一个文件，在其中查找美国邮政编码，如 TX77845 和 DC 20500-0001。smatch 类型是一个保存 regex 匹配结果的容器。在本例中 matches[0] 对应整个模式而 matches[1] 对应可选的四个数字的子模式。

正则表达式的语法和语义的设计目标是使之能编译成可高效运行的自动机 [Cox, 2007]，这个编译过程是由 regex 类型在运行时完成的。

7.3.2 正则表达式符号表示

regex 库可以识别几种不同的正则表达式符号表示。本书中使用默认的符号表示，它是 ECMA 标准的一个变体，ECMA 标准被用于 ECMAScript 中（更为人们所熟知的名称是 JavaScript）。

正则表达式的语法基于下表所示的一些特殊意义的字符：

正则表达式的特殊字符			
.	任意单个字符（“通配符”）	\	下一个字符有特殊含义
[字符集开始	*	零或多次重复（后缀操作）
]	字符集结束	+	一或多次重复（后缀操作）
{	指定重复次数开始	?	可选（零或一次）（后缀操作）
}	指定重复次数结束		二选一（或）
(分组开始	^	行开始；非
)	分组结束	\$	行结束

例如，我们可以指定一个模式是以零个或多个 A 开头，后接一个或多个 B，最后是一个可选的 C：

```
^A*B+C?$
```

则下面这些字符串与此模式匹配：

```
AAAAAAAAAAAAABBBBBBBBBBC
BC
B
```

而下面这些字符串与此模式不匹配:

```
AAAAA      // 没有 B
AAAABC     // 多了前导空格
AABBCC     // 多于一个 C
```

模式的一个组成部分如果用括号围起, 则它构成一个子模式 (可从 `smatch` 中单独抽取出来)。例如:

```
\d+-\d+     // 没有子模式
\d+(-\d+)   // 一个子模式
(\d+)(-\d+) // 两个子模式
```

通过添加下表所示的后缀, 可以指定一个模式是可选的还是重复多次的 (如无后缀, 则只出现一次):

重 复	
{n}	严格重复 n 次
{n, }	重复 n 次或更多次
{n, m}	至少重复 n 次, 最多 m 次
*	零次或多次, 即, {0, }
+	一次或多次, 即, {1, }
?	可选的 (零次或一次), 即, {0, 1}

例如下面的正则表达式:

```
A{3}B{2,4}C*
```

则下面这些字符串与之匹配:

```
AAABBC
AAABBB
```

而下面这些字符串与之不匹配:

```
AABBC      // A 太少
AAABC      // B 太少
AAABBBBBCCC // B 太多
```

如果在任何重复符号 (? 、 * 、 + 及 { }) 之后放一个后缀 ? , 会使模式匹配器变得“懒惰”或者说“不贪心”。即当查找一个模式时, 匹配器会查找最短匹配而非最长匹配。而默认情况下, 模式匹配器总是查找最长匹配, 这就是所谓的最长匹配法则 (Max Munch rule)。考虑下面的字符串:

```
ababab
```

模式后缀 `(ab)*` 匹配整个字符串 `ababab`, 而 `(ab)*?` 只匹配第一个 `ab`。

下表列出了最常用的字符集:

字 符 集	
<code>alnum</code>	任意字母数字字符
<code>alpha</code>	任意字母

(续)

字符集	
blank	任意空白符，但不能是行分隔符
cntrl	任意控制字符
d	任意十进制数字
digit	任意十进制数字
graph	任意图形字符
lower	任意小写字符
print	任意可打印字符
punct	任意标点
s	任意空白符
space	任意空白符
upper	任意大写字符
w	任意单词字符（字母、数字字符再加下划线）
xdigit	任意十六进制数字字符

在正则表达式中，字符集名字必须用 `[::]` 包围起来。例如，`[[:digit:]]` 匹配一个十进制数字。而且，如果是定义一个字符集，外边还必须再包围一对方括号 `[]`。

下表的一些字符集还支持简写表示：

字符集简写		
<code>\d</code>	一个十进制数字	<code>[[:digit:]]</code>
<code>\s</code>	一个空白符（空格、制表符等等）	<code>[[:space:]]</code>
<code>\w</code>	一个字母（a ~ z）、数字（0 ~ 9）或下划线（_）	<code>[[:alnum:]]</code>
<code>\D</code>	非十进制数字	<code>[^[:digit:]]</code>
<code>\S</code>	非空白符	<code>[^[:space:]]</code>
<code>\W</code>	非字母、数字或下划线	<code>[^_[:alnum:]]</code>

此外，支持正则表达式的语言通常还提供如下表所示的字符集简写：

非标准（但常见的）字符集简写		
<code>\l</code>	一个小写字符	<code>[[:lower:]]</code>
<code>\u</code>	一个大写字符	<code>[[:upper:]]</code>
<code>\L</code>	非小写字符	<code>[^[:lower:]]</code>
<code>\U</code>	非大写字符	<code>[^[:upper:]]</code>

为了保证可移植性，应使用完整的字符集名字而不是简写。

考虑这样一个例子：编写一个模式，描述 C++ 标识符——以一个下划线或字母开头，后接一个由字母、数字或下划线组成的序列（可以是空序列）。为了展示其中的微妙之处，下面给出了一些错误的模式：

```

[:alpha:][:alnum:]*           // 错误：表示字符集应该在外边再加上一对括号
[[:alpha:]][[:alnum:]]*      // 错误：没有接受下划线（'_' 不是字母）
([[:alpha:]]_)[[:alnum:]]*   // 错误：下划线也不属于字母数字

([[:alpha:]]_)[[:alnum:]]*_   // 正确，但太笨拙了
[[:alpha:_]][[:alnum:]]*_    // 正确：在字符集中包含了下划线
[[:alpha:]][[:alnum:]]*      // 变换了顺序，也是正确的
[[:alpha:]]\w*              // \w 等价于 [[:alnum:]]

```

最后，下面的函数用最简单的 `regex_match()` 版本（见 7.3.1 节）来检查一个字符串是否是一个标识符：

```
bool is_identifier(const string& s)
{
    regex pat {"[_:alpha:]\w*"}; // 下划线或字母
                                // 后接零或多个下划线、字母或数字
    return regex_match(s,pat);
}
```

注意，要在一个普通字符串面值常量中包含一个反斜线，必须使用两个反斜线。使用原始字符串面值常量则可缓解这种特殊字符问题，例如：

```
bool is_identifier(const string& s)
{
    regex pat {R"([_:\alpha:]\w*)"};
    return regex_match(s,pat);
}
```

下面是一些模式的例子：

<code>Ax*</code>	// A, Ax, Axxxx	
<code>Ax+</code>	// Ax, Axxxx	A 不匹配
<code>\d-?\d</code>	// 1-2, 12	1--2 不匹配
<code>\w(2)-\d{4,5}</code>	// Ab-1234, XX-54321, 22-5432	数字也属于 \w
<code>(\d:)?(\d+)</code>	// 12:3, 1:23, 123, :123	123: 不匹配
<code>(bs BS)</code>	// bs, BS	bS 不匹配
<code>[aeiouy]</code>	// a, o, u	英语元音字母, x 不匹配
<code>[^aeiouy]</code>	// x, k	非元音字母, e 不匹配
<code>[a^eiouy]</code>	// a, ^, o, u	英语元音字母或 ^

在一个正则表达式中，被括号限定的部分形成一个 `group`（子模式），用 `sub_match` 来表示。如果你需要用括号但又不想定义一个子模式，则应使用 `(?)` 而不是 `()`。例如：

```
(\s|:|,)=(\d*) // 空白符、冒号和 / 或逗号，后接一个数
```

假设我们对数之前的字符不感兴趣（可能是分隔符），则可写成：

```
(?:\s|:|,)=(\d*) // 空白符、冒号和 / 或逗号，后接一个数
```

这样，正则表达式引擎就不必保存第一个字符：`(?)` 使得只有数字的部分才是子模式。下表是一些正则表达式分组的例子：

正则表达式分组例子

<code>\d*\s\w+</code>	无分组（子模式）
<code>(\d*)\s(\w+)</code>	两个分组
<code>(\d*)(\s(\w+))+</code>	两个分组（分组没有嵌套）
<code>(\s*\w*)+</code>	一个分组，但有一个或多个子模式；只有最后一个子模式保存为一个 <code>sub_match</code>
<code><(.*?)><(.*?)</\1></code>	三个分组； <code>\1</code> 表示“与分组 1 一样”

最后一个模式对于 XML 文件的解析很有用。它可以查找标签起始和结束的标记。注意，对标签起始和结束间的子模式，这里使用了非贪心匹配（懒惰匹配）`.*?`。假如使用普通的匹配策略 `.*`，下面这个输入就会导致问题：

```
Always look for the <b>bright</b> side of <b>life</b>.
```

如果对第一个子模式采用贪心匹配策略，则会将第一个 < 与最后一个 > 配对。这是正确的行为，但结果也许不是程序员所期望的。

有关正则表达式更为详尽的介绍，请参阅 [Friedl, 1997]。

7.3.3 迭代器

我们可以定义一个 `regex_iterator` 来遍历一个流（字符序列），在其中查找给定模式。例如，我们可以输出一个 `string` 中所有由空白符分隔的单词：

```
void test()
{
    string input = "aa as; asd ++e`asdf asdfg";
    regex pat {R"(\s+(\w+))"};
    for (sregex_iterator p(input.begin(),input.end(),pat); p!=sregex_iterator{}; ++p)
        cout << (*p)[1] << '\n';
}
```

它会输出：

```
as
asd
asdfg
```

我们漏掉了第一个单词 `aa`，因为它没有先导空格。如果我们将模式简化为 `R"((\ew+))"`，则会得到

```
aa
as
asd
e
asdf
asdfg
```

`regex_iterator` 是一种双向迭代器，因此我们不能直接遍历一个 `istream`。我们也不能通过 `regex_iterator` 写数据，默认的 `regex_iterator(regex_iterator{})` 也是表示序列结束的唯一方式。

83

7.4 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 36 ~ 37 章中有更加详细的描述。
- [2] 优先选择 `string` 操作而不是 C 风格的字符串函数；参见 7.1 节。
- [3] 使用 `string` 声明变量和成员而不是将它作为基类；参见 7.2 节。
- [4] 返回 `string` 应采用传值方式（依赖移动语义）；参见 7.2 节，7.2.1 节。
- [5] 直接或间接使用 `substr()` 读取子串，使用 `replace()` 写子串；参见 7.2 节。
- [6] 需要的话可以扩展或收缩一个 `string`；参见 7.2 节。
- [7] 当需要范围检查时，应使用 `at()` 而不是迭代器或 `[]`；参见 7.2 节。
- [8] 当需要优化性能时，应使用迭代器或 `[]` 而不是 `at()`；参见 7.2 节。
- [9] `string` 输入不会溢出；参见 7.2 节，8.3 节。

- [10] 只有迫不得已时，才使用 `c_str()` 获得一个 `string` 的 C 风格字符串表示；参见 7.2 节。
- [11] 使用 `stringstream` 或通用的值提取函数（如 `to<X>`）将字符串转换为数值；参见 8.8 节。
- [12] 可用 `basic_string` 构造任意类型字符的字符串；参见 7.2.1 节。
- [13] 将 `regex` 用于正则表达式的大部分常规用途；参见 7.3 节。
- [14] 除非是最简单的模式，否则应使用原始字符串字面值常量来表示；参见 7.3 节。
- [15] 使用 `regex_match()` 匹配整个输入；参见 7.3 节，7.3.2 节。
- [16] 使用 `regex_search()` 在一个输入流中搜索模式；参见 7.3.1 节。
- [17] 可以调整正则表达式符号表示来适应不同的标准；参见 7.3.2 节。
- [18] 默认的正则表达式符号表示是 ECMAScript 中所采用的表示法；参见 7.3.2 节。
- [19] 使用正则表达式要注意节制，它很容易变成一种难读的语言；参见 7.3.2 节。
- [20] 注意，`\i` 符号允许你用之前的子模式来描述子模式；参见 7.3.2 节。
- [21] 用 `?` 令模式的匹配采用“懒惰”策略；参见 7.3.2 节。
- [84] [22] 用 `regex_iterator` 来遍历一个流并查找给定模式；参见 7.3.3 节。

所见即所得。

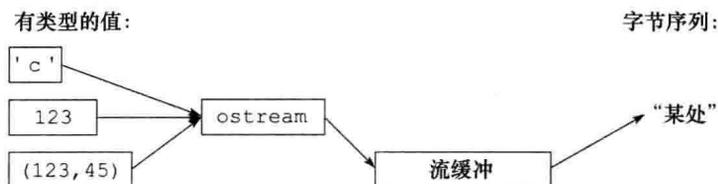
——布莱恩·克尼汉

- 引言
- 输出
- 输入
- I/O 状态
- 用户自定义类型的 I/O
- 格式化
- 文件流
- 字符串流
- 建议

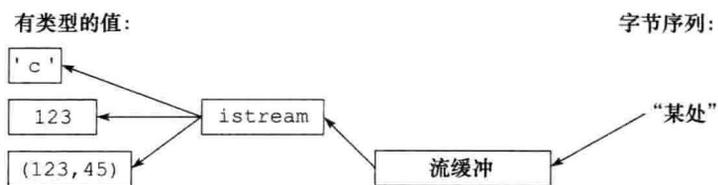
8.1 引言

I/O 流库提供了文本和数值的输入输出功能，这种输入输出是带缓冲的，可以是格式化的，也可以是未格式化的。

`ostream` 对象将有类型的对象转换为一个字符（字节）流：



`istream` 对象将一个字符（字节）流转换为有类型的对象：



`istream` 和 `ostream` 上的操作将在 8.3 节和 8.2 节介绍。这些操作都是类型安全且类型敏感的，都能扩展以处理用户自定义类型。

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。

标准库流可用于二进制 I/O、用于不同字符类型、用于不同区域设置，也可使用高级缓冲策略，但这些主题已经超出了本书的范围。

8.2 输出

在 `<ostream>` 中，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 8.5 节）。运算符 `<<`（“放到”）是输出运算符，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';
}
```

86 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就是其字符形式，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' 的整数值是 98 (C++ 实现中使用的 ASCII 编码值), 因此这个函数的输出结果为 a98c。

8.3 输入

在 <istream> 中, 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, 并能很容易地扩展到对用户自定义类型的支持。

运算符 >> (“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。

类似输出操作, 输入操作也可以链接起来, 所以也可以写成下面这样:

87

```
void f()
{
    int i;
    double d;
    cin >> i >> d;    // 读入到 i 和 d 中
}
```

两段代码执行的时候都是在读到非数字字符时终止整型数的读取。默认情况下, >> 会跳过起始的空白符, 因此一个恰当的完整输入序列可能是这样的:

```
1234
12.34e5
```

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";
}
```

如果你输入 Eric, 程序将回应:

```
Hello, Eric!
```

默认情况下，空白符（如空格或换行）会终止输入。因此，如果你输入 Eric Bloodaxe 冒充不幸的约克王，程序的回应仍会是：

```
Hello, Eric!
```

可以用函数 `getline()` 来读取一整行（包括结束的换行符），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!
```

行尾的换行符被丢掉了，因此接下来从 `cin` 输入会从下一行开始。

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，你就无需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。

88

8.4 I/O 状态

每个 `iostream` 都有状态，我们可以检查此状态来判断流操作是否成功。流状态最常见的应用是控制读取值序列：

```
vector<int> read_ints(istream& is)
{
    vector<int> res;
    int i;
    while (is>>i)
        res.push_back(i);
    return res;
}
```

这段代码从 `is` 读取整型值，直至遇到非整型值的内容（通常是输入结束）。这段代码的关键是 `is>>i` 操作返回一个指向 `is` 的引用，而检测一个 `iostream` 对象（如 `is`）的结果为 `true` 的话，表示流已经准备好进行下一个操作。

一般来说，I/O 状态包含了读写所需的所有信息，例如格式化信息（见 8.6 节）、错误状态（如输入是否已结束）以及使用了何种缓冲等等。特别是，一个用户可以设置状态来表示发生了错误（见 8.5 节），如果错误不严重也清除状态。例如，想象我们要读取一个整数序列，而其中可能包含嵌套：

```
while (cin) {
    for (int i; cin>>i; ) {
        // ... 使用读入的整数 ...
    }

    if (cin.eof()) {
```

```

    //... 一切顺利, 我们到达了文件尾 ...
}
else if (cin.fail()) {      // 一个也许能恢复的错误
    cin.clear();           // 将状态重置为 good()
    char ch;
    if (cin>>ch) {        // 查找形如 { ... } 的嵌套
        switch (ch) {
            case '{':
                // ... 嵌套结构开始 ...
                break;
            case '}':
                // ... 嵌套结构结束 ...
                break;
            default:
                cin.setstate(ios_base::failbit); // 将 cin 状态置为 fail()
        }
    }
}
// ...
}

```

8.5 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项:

```

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 (的引用) 作为第一个参数, 输出完毕后, 返回此流的引用。

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}; // 读入的值赋予 Entry 对象
                return is;
            }
        }
    }
}

```

```

    }
}
is.setstate (ios_base::failbit); // 将流状态置为 fail
return is;
}

```

输入操作返回它所操作的 `istream` 对象的引用，它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` 的操作是否成功了”。

`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：

90

```

{"John Marwood Cleese", 123456 }
{"Michael Edward Palin",987654}

```

我们可以用下面代码从输入流读取这样的值对，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}

```

请参考 7.3 节，其中介绍了在字符流中识别模式的更系统的方法（正则表达式）。

8.6 格式化

`iostream` 库提供了很多操作来控制输入输出的格式。最简单的格式化控制方式就是操纵符（manipulator），它们定义在 `<ios>`、`<istream>`、`<ostream>` 和 `<iomanip>`（接受参数的操纵符）中：例如，我们能以十进制（默认格式）、八进制或十六进制格式输出整数：

```

cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << '\n'; // 打印 1234, 4d2, 2322

```

我们还可以显式设置浮点数的输出格式：

```

constexpr double d = 123.456;

cout << d << "; " // 用默认格式输出 d
    << scientific << d << "; " // 用 1.123e2 风格输出 d
    << hexfloat << d << "; " // 用十六进制输出 d
    << fixed << d << "; " // 用 123.456 风格输出 d
    << defaultfloat << d << '\n'; // 用默认格式输出 d

```

这段代码会输出：

```

123.456; 1.234560e+002; 0x1.edd2f2p+6; 123.456000; 123.456

```

精度是在显示浮点数时用来确定数字位数的一个整数：

- 通用格式（`defaultfloat`）会根据可用空间的大小选择能最好地显示给定值的格式。精度指出最多显示多少位数字。
- 科学记数法（`scientific`）在小数点前显示一位数字，并显示一个指数。精度指出在小数点后最多显示多少位数字。

- 定点格式 (fixed) 显示整数部分、小数点和小数部分。精度指出在小数点后最多显示多少位数字。

浮点值在显示时会进行四舍五入而不是简单截断，而 `precision()` 不会影响整数输出。例如：

```
cout.precision(8);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';

cout.precision(4);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';
```

输出结果为：

```
1234.5679 1234.5679 123456
1235 1235 123456
```

这些操纵符都是“黏性的”即在后续的浮点值输出中会一直有效（直至新操纵符改变格式）。

91

8.7 文件流

在 `<fstream>` 中，标准库提供了从文件读取数据以及向文件写入数据的流：

- `ifstream` 用于从文件读取数据。
- `ofstream` 用于向文件写入数据。
- `fstream` 用于读写文件。

例如：

```
ofstream ofs("target");           // "o" 表示 "输出"
if (!ofs)
    error("couldn't open 'target' for writing");
```

通常通过检查流的状态来检测文件流是否正确打开。

```
ifstream ifs;                     // "i" 表示 "输入"
if (!ifs)
    error("couldn't open 'source' for reading");
```

假定检测成功，`ofs` 就可以像普通 `ostream` 一样使用（就像 `cout`），`ifs` 就可以像普通 `istream` 一样使用（就像 `cin`）。

我们可以在文件中进行定位，还可以对文件打开方式进行更细致的控制，但这些内容超出了本书范围。

8.8 字符串流

在 `<sstream>` 中，标准库提供了从 `string` 读取数据以及向 `string` 写入数据的流：

- `istringstream` 用于从 `string` 读取数据。
- `ostringstream` 用于向 `string` 写入数据。

- stringstream 用于读写 string。

例如：

```
void test()
{
    ostream oss;
    oss << "{temperature," << scientific << 123.4567890 << "}";
    cout << oss.str() << '\n';
}
```

92

istringstream 中的内容可以通过调用 str() 成员函数来获取。ostringstream 的一个最常见用途是先通过它对输出内容进行格式化，然后再将得到的字符串输出到 GUI。与此类似，我们可以将从 GUI 接收到的字符串放入 istringstream 中，然后通过它进行格式化输入（见 8.3 节）。

stringstream 既可用于读，也可用于写。例如，我们可以定义一个操作，在两种都有字符串表示的类型间进行转换：

```
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) // 从流读取结果
        ||!(interpreter >> std::ws).eof()) // 流中还有剩余内容?
        throw runtime_error{"to<>() failed"};

    return result;
}
```

只有当函数模板实参无法推断出来，或是没有默认值时，才需要显式指定它，因此我们可以编写下面的代码：

```
auto x1 = to<string,double>(1.2); // 完全显式的（但也是啰嗦的）
auto x2 = to<string>(1.2); // Source 被推断为 double
auto x3 = to<>(1.2); // Target 默认值为 string;Source 被推断为 double
auto x4 = to(1.2); // <> 是冗余的；
// Target 默认值为 string;Source 被推断为 double
```

如果所有函数模板实参都使用默认值，则 <> 可以省略。

我认为这是一个很好的例子，展示了通过组合语言特性和标准库工具和方法来实现代码的通用性和易用性。

8.9 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 38 章中有更加详细的描述。
- [2] istream 是类型安全、类型敏感且易扩展的；参见 8.1 节。
- [3] 如果一个类型的值存在有意义的文本表示形式，我们可以为它定义 << 和 >> 操作；参见 8.1 节，8.2 节，8.3 节。

- [4] `cout` 用于标准输出, `cerr` 用于报告错误; 参见 8.1 节。
- [5] 标准库提供了用于普通字符和宽字符的 `iostream`, 你可以为任何字符类型定义 `iostream`; 参见 8.1 节。
- [6] 标准库支持二进制 I/O; 参见 8.1 节。
- [7] 标准库提供了用于标准输入输出流、文件流和 `string` 流的标准 `iostream`; 参见 8.2 节, 8.3 节, 8.7 节, 8.8 节。
- [8] 将 `<<` 操作链接起来可以简化输出语句; 参见 8.2 节。
- [9] 将 `>>` 操作链接起来可以简化输入语句; 参见 8.3 节。
- [10] 不断读取输入来存入 `string` 中不会导致溢出; 参见 8.3 节。
- [11] 默认情况下 `>>` 会跳过起始空白符; 参见 8.3 节。
- [12] 使用流状态 `fail` 处理可恢复的 I/O 错误; 参见 8.4 节。
- [13] 可以为自己的类型定义 `<<` 和 `>>` 运算符; 参见 8.5 节。
- [14] 无需修改 `istream` 和 `ostream` 来添加新的 `<<` 和 `>>` 运算符; 参见 8.5 节。
- [15] 使用操纵符控制格式化; 参见 8.6 节。
- [16] `precision()` 对后续浮点输出操作一直有效; 参见 8.6 节。
- [17] 浮点格式 (如 `scientific`) 对后续浮点输出操作一直有效; 参见 8.6 节。
- [18] 当使用标准操纵符时使用 `#include <ios>`; 参见 8.6 节。
- [19] 当使用接受参数的标准操纵符时使用 `#include <iomanip>`; 参见 8.6 节。
- [20] 不要试图拷贝一个文件流。
- [21] 在使用一个文件流之前, 记得检查它是否正确绑定到了文件上; 参见 8.7 节。
- [22] 若在内存中进行 I/O 格式化, 使用 `stringstream`; 参见 8.8 节。
- [23] 对任意两种类型, 只要它们都有字符串表示形式, 就可以为它们定义类型转换操作; 参见 8.8 节。

容 器

它新颖、独一无二、简单，它必须成功！

——H. 尼尔森

- 引言
- vector
 - 元素；范围检查
- list
- map
- unordered_map
- 容器概述
- 建议

9.1 引言

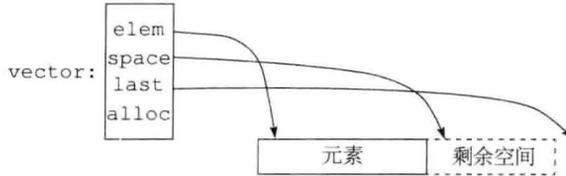
大多数计算任务都会涉及创建值的集合，然后对这些集合进行操作。一个简单的例子是读取字符存入 `string` 中，然后打印这个 `string`。如果一个类的主要目的是保存对象，那么我们通常称之为容器（`container`）。对给定的任务提供合适的容器及其上有用的基本操作，是构建任何程序的重要步骤。

我们通过一个简单示例程序来介绍标准库容器。这个程序保存名字和电话号码，这就是那种对不同背景的人都显得“简单而明显”程序。我们用 8.5 节中的 `Entry` 类来保存一个电话簿表项。在本例中，我们特意忽略很多现实世界中的复杂因素，例如，很多电话号码并不能用一个 32 位 `int` 来简单表示。

95

9.2 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。一个典型的 `vector` 实现（见 4.2.2 节，4.6 节）会包含一个句柄，保存指向首元素的指针，还会包含一个指向尾元素之后位置的指针以及一个指向所分配空间之后位置的指针（或者是等价的一个指针外加一个偏移量）（见 10.1 节）：



除了这些成员之外，`vector` 还会包含一个分配器（`alloc`），`vector` 通过它为自己的元素分配内存空间。默认的分配器使用 `new` 和 `delete` 分配和释放内存。

可以用一组值来初始化 `vector`，当然，值的类型必须与 `vector` 元素的类型吻合：

```
vector<Entry> phone_book = {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

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';
}
```

下标照样从 0 开始，因此 `book[0]` 保存的是 David Hume 的表项。`vector` 的成员函数 `size()` 返回元素的数目。

`vector` 的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 1.8 节）：

```
void print_book(const vector<Entry>& book)
{
    for (const auto& x : book) // 关于 "auto" 请查阅 1.5 节
        cout << x << '\n';
}
```

当我们定义一个 `vector` 时，就给定它一个初始大小（初始的元素数目）：

```
vector<int> v1 = {1, 2, 3, 4}; // 大小为 4
vector<string> v2; // 大小为 0
vector<Shape> v3(23); // 大小为 23; 元素初值: nullptr
vector<double> v4(32,9.9); // 大小为 32; 元素初值: 9.9
```

可以在一对圆括号中显式地给出 `vector` 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）。如果你不想要默认值，可以通过构造函数的第二个参数来指定一个值（例如，将 `v4` 的 32 个元素初始化为 9.9）。 96

我们定义 `vector` 时可以给定初始大小，随着程序的执行其大小可以被改变。`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。

标准库的 vector 经过了精心设计，通过不断调用 push_back() 来扩张 vector，这样会很高效率。为了说明如何做到这一点，考虑一个精心设计的简单 Vector 类（参见第4章和第5章），它使用了上图所示的存储方式：

```
template<typename T>
class Vector {
    T* elem;           // 指向首元素的指针
    T* space;         // 指向第一个未用（未初始化）位置的指针
    T* last;          // 指向最后一个存储位置的指针
public:
    // ...
    int size();       // 元素数目（space-elem）
    int capacity();  // Vector 的空间大小（last-elem）
    // ...
    void reserve(int newsz); // 将空间增大到 newsz
    // ...
    void push_back(const T& t); // 将 t 拷贝到 Vector
    void push_back(T&& t);     // 将 t 移动到 Vector
};
```

标准库 vector 有 capacity()、reserve() 和 push_back() 等几个成员。vector 的用户和它的其他成员都能使用 reserve() 来扩展空间以容纳更多元素。在此过程中，可能需要分配新的内存空间，并将现有元素拷贝到新空间中。

有了 capacity() 和 reserve()，实现 push_back() 就很简单了：

```
template<typename T>
void Vector<T>::push_back(const T& t)
{
    if (capacity()<size()+1) // 确保有空间能容纳 t
        reserve(size()+0?8:2*size()); // 将空间扩张一倍
    new(space){t};          // 将 *space 初始化为 t
    ++space;
}
```

这样，分配空间和迁移元素的频率就很低了。我原来习惯用 reserve() 来提高性能，但事实证明这是浪费精力：vector 所使用的启发式策略远好于我的估计。因此，我现在只有在元素指针时才用 reserve() 来避免迁移元素。

在赋值和初始化时，vector 可以被拷贝。例如：

```
vector<Entry> book2 = phone_book;
```

如 4.6 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 在赋值过程中会拷贝其中的元素。因此，在 book2 初始化完成后，它和 phone_book 各自保存每个 Entry 的一份拷贝。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。当拷贝并非必要时，应该使用引用或指针（见 1.8 节）或是移动操作（见 4.6.2 节）。

标准库 vector 很灵活、很高效，应将它作为默认容器，也就是说除非有充分的理由使用其他容器，否则应使用 vector。如果你的理由是“效率”，请进行性能测试——我们在容

97

器使用性能方面的直觉通常是很不可靠的。

9.2.1 元素

类似所有标准库容器，`vector` 是某种类型 `T` 的元素的容器，即 `vector<T>`。几乎任何类型都可以作为元素类型：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 *`）。当你插入一个新元素时，它的值被拷贝到容器中。例如，当你将一个整型值 7 存入容器，结果元素确实就是一个值为 7 的整型对象，而不是指向某个包含 7 的对象的引用或指针。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，这是非常关键的。

如果你有一个类层次结构（见 4.5 节）依赖 `virtual` 函数获得多态性，就不应在容器中直接保存对象，而应保存对象的指针（或智能指针；见 11.2.1 节）。例如：

```
vector<Shape> vs;           // 不正确——空间不足以容纳 Circle 或 Smiley
vector<Shape> vps;        // 好一些，但请参见 4.5.4 节
vector<unique_ptr<Shape>> vups; // 正确
```

9.2.2 范围检查

标准库 `vector` 并不进行范围检查。例如：

```
void silly(vector<Entry>& book)
{
    int i = book[book.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。这并不是我们所需要的，而这种越界错误又是常见的问题。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（但名字是 Vec）

    T& operator[](int i)                // 范围检查
    { return vector<T>::at(i); }

    const T& operator[](int i) const    // 常量版本；见 4.2.1 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()` 操作也完成下标操作，但它会在参数越界时抛出一个类型为 `out_of_range` 的异常（见 3.4.1 节）。

对一个 `Vec`，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 3.4.1 节）。如果用户不捕获异常，程序会以一种明确定义的方式退出，而不是继续执行或是以一种未定义的方式失败。一种尽量减少未捕获异常带来问题的方法是使用 `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";
}

```

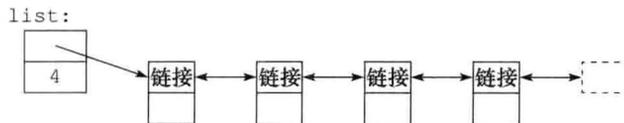
这段代码提供了默认的异常处理程序，这样，当我们未能成功捕获某个异常时，就会进入默认异常处理程序，在标准错误流 `cerr` 上打印一条错误信息（见 8.2 节）。

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），从而免去你定义 `Vec`（或等价的类）的麻烦。

99

9.3 list

标准库提供了一个名为 `list` 的双向链表：



如果希望在一个序列中添加、删除元素而无需移动其他元素，则应使用 `list`。对电话簿应用而言，插入、删除操作可能就很频繁，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}
};

```

当我们使用一个链表时，通常并不是想要像使用向量那样使用它，即不会用下标操作访问链表元素，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（如第 10 章所述）：

```

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。

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，可用来遍历 `list`（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（见第 10 章）。我们可以改写函数 `get_number()`，显式使用迭代器遍历 `list`（这个版本稍显繁琐）：

```
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”
}
```

范围 `for` 循环版本更简练，也更不容易出错。实际上，迭代器版本就是编译器实现范围 `for` 的大致方式。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，而当 `p` 指向一个类且该类有一个成员 `m` 时，`p->m` 等价于 `(*p).m`。

100

向 `list` 中添加元素以及从 `list` 中删除元素都很简单：

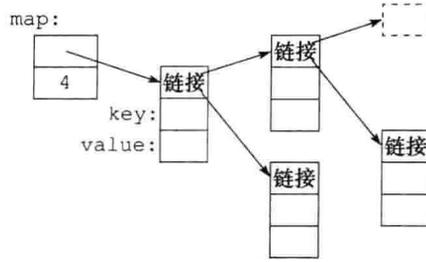
```
void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)
{
    phone_book.insert(p,ee); // 将 ee 添加到 p 指向的元素之前
    phone_book.erase(q);    // 删除 q 指向的元素
}
```

对于 `list` 来说，`insert(p,elem)` 将一个新元素插入到 `p` 指向的元素之前，新元素的值是 `elem` 的一份拷贝。类似地，`erase(p)` 从 `list` 中删除 `p` 指向的元素并销毁它。两个操作中，`p` 都可以是指向 `list` 尾后位置的迭代器。

上面这些 `list` 的例子都可以等价地写成使用 `vector` 的版本，而且令人惊讶的是（除非你了解机器的体系结构），当数据量较小时，`vector` 版本的性能会优于 `list` 版本。当我们想要的不过是一个元素序列时，就可以在 `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`。

9.4 map

编写程序在一个（名字，数值）对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）：



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 是键值对的容器，经过特殊优化来提高搜索性能。我们可以像初始化 vector 和 list 那样初始化 map（见 9.2 节，9.3 节）：

```
map<string,int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

101

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），得到的结果是与关键字关联的值（应该是 map 的第二个类型，称为值或映射类型）。例如：

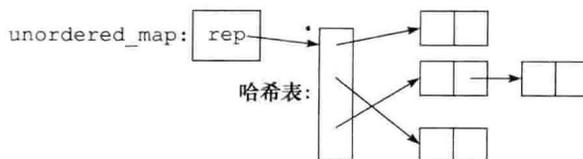
```
int get_number(const string& s)
{
    return phone_book[s];
}
```

换句话说，对 map 进行下标操作本质上是进行一次搜索。如果未找到 key，则向 map 插入一个新元素，它具有给定的 key，关联的值为 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 []。

9.5 unordered_map

搜索 map 的时间代价是 $O(\log(n))$ ， n 是 map 中的元素数目。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 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 <）。标准库哈希容器都被称为“无序”容器，因为它们不需要一个序函数：



例如，我们可以使用 <unordered_map> 中定义的 unordered_map 来表示电话簿：

```
unordered_map<string,int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

类似 map，我们也可以对 unordered_map 使用下标操作：

```
int get_number(const string& s)
{
    return phone_book[s];
}
```

标准库为 string 以及其他内置类型和标准库类型提供了默认的哈希函数。必要时，例如需要用无序容器保存自定义类型对象时，你可以定义自己的哈希函数。哈希函数通常以函数对象（见 5.5 节）的形式提供。例如：

102

```
struct Record {
    string name;
    int product_code;
    // ...
};

struct Rhash { // 为 Record 定义的哈希函数
    size_t operator()(const Record& r) const
    {
        return hash<string>()(r.name) ^ hash<int>()(r.product_code);
    }
};

unordered_set<Record,Rhash> my_set; // Record 的集合用 Rhash 进行搜索
```

如本例所示，组合已有哈希函数和异或运算 (^) 通常是一种很有效的构造新哈希函数的方式。

9.6 容器概述

标准库提供了一些最通用也最有用的容器类型（见下表），使得程序员能够根据应用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（见 9.2 节）
list<T>	双向链表（见 9.3 节）
forward_list<T>	单向链表
deque<T>	双端队列
set<T>	集合（只有关键字而没有值的 map）
multiset<T>	允许重复值的集合
map<K, V>	关联数组（见 9.4 节）
multimap<K, V>	允许重复关键字的 map
unordered_map<K, V>	采用哈希搜索的 map（见 9.5 节）
unordered_multimap<K, V>	采用哈希搜索的 multimap
unordered_set<T>	采用哈希搜索的 set
unordered_multiset<T>	采用哈希搜索的 multiset

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。

容器都定义在命名空间 `std` 中，通过 `<vector>`、`<list>`、`<map>` 等头文件提供（见 6.3 节）。此外，标准库还提供了容器适配器 `queue<T>`、`stack<T>`、`priority_queue<T>`。如果需要使用这些特性，请查阅相关资料。标准库还提供了一些更特殊化的类似容器的类型，如定长数组 `array<T,N>`（见 11.3.1 节）和 `bitset<N>`（见 11.3.2 节）。

103 从表达角度来看，标准库的各种容器和它们的基本操作被设计成相似的。而且，对于不同容器来说，操作的含义也是相同的。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，范围检查向量 `Vector`（见 3.4.2 节，第 4 章）就是一个例子。容器接口的一致性还使得我们可以设计与类型无关的算法。但是，凡事都有两面性，有优点就会有缺点。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，就需要移动元素，效率不佳。而 `list` 则恰好具有相反的特性。请注意，当序列较短、元素大小较小时，`vector` 通常比 `list` 更为高效（即便是 `insert()` 和 `erase()` 操作也是如此）。我推荐将标准库 `vector` 作为存储元素序列的默认类型：除非有充分的理由，否则不要选择其他容器。

标准库还提供了单向链表 `forward_list`，这是一种为空序列特别优化过的容器（只占用一个字的空间）。这种序列的元素数目为 0 或特别少，在实际中特别有用。

9.7 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 31 章中有更加详细的描述。
- [2] 一个标准库容器定义了一个序列；参见 9.2 节。
- [3] 标准库容器是资源句柄；参见 9.2 节，9.3 节，9.4 节，9.5 节。
- [4] 将 `vector` 作为你的默认容器；参见 9.2 节，9.6 节。
- [5] 对于简单的容器遍历，使用范围 `for` 循环或一对首尾迭代器；参见 9.2 节，9.3 节。
- [6] 使用 `reverse()` 避免指向元素的指针或迭代器失效；参见 9.2 节。
- [7] 在未经过测试的情况下，不要假定使用 `reverse()` 会带来性能收益，参见 9.2 节。
- [8] 使用容器及其 `push_back()` 和 `resize()` 操作，而不是使用数组和 `realloc()` 操作；参见 9.2 节。
- [9] 调整 `vector` 大小后，不要再使用旧迭代器；参见 9.2 节。
- [10] 不要假定 `[]` 有范围检查功能；参见 9.2 节。

- [11] 如果你需要确保进行范围检查, 应使用 `at()` 操作; 参见 9.2 节。
- [12] 向容器插入元素时, 元素是被拷贝进容器的; 参见 9.2.1 节。
- [13] 如要保持元素的多态行为, 在容器中保存指针而非对象; 参见 9.2.1 节。
- [14] 在 `vector` 上执行插入操作, 如 `insert()` 和 `push_back()`, 通常会异常高效; 参见 9.3 节。
- [15] 对通常为空的序列, 使用 `forward_list`; 参见 9.6 节。
- [16] 当事关性能, 不要相信你的直觉, 应进行性能测试; 参见 8.6 节。
- [17] `map` 通常用红黑树实现; 参见 9.4 节。
- [18] `unordered_map` 是哈希表; 参见 9.5 节。 104
- [19] 传递容器参数时, 应传递引用, 返回容器时, 应返回值; 参见 9.2 节。
- [20] 初始化一个容器时, 采用 `()` 初始化方式指定容器大小, 使用 `{}` 初始化方式给出元素列表; 参见 4.2.3 节, 9.2 节。
- [21] 优先选择紧凑的、连续存储的数据类型; 参见 9.3 节。
- [22] 遍历 `list` 的代价相对较高; 参见 9.3 节。
- [23] 如果需要在大量数据中进行搜索操作, 选择无序容器; 参见 9.5 节。
- [24] 如果需要按顺序遍历容器中的元素, 选择有序关联容器 (如 `map` 和 `set`); 参见 9.5 节。
- [25] 若元素类型没有自然的顺序 (如没有合理的 `<` 运算符), 选择无序容器; 参见 9.4 节。
- [26] 通过实验来检查你设计的哈希函数是否令人满意; 参见 9.5 节。
- [27] 通过组合标准哈希函数和异或运算设计出的哈希函数通常有较好效果; 参见 9.5 节。 105
↓
- [28] 了解标准库容器, 优先选择这些容器而不是自己实现的数据结构; 参见 9.6 节。 106

算 法

若无必要，勿增实体。

——奥卡姆的威廉^①

- 引言
- 使用迭代器
- 迭代器类型
- 流迭代器
- 谓词
- 标准库算法概览
- 容器算法
- 建议

10.1 引言

单纯一个数据结构是没太大用处的，比如一个孤立的链表或者数组。为了使用一个数据结构，我们还需要能对其进行基本访问的操作，如添加和删除元素的操作（就像为 `list` 和 `vector` 提供的那些操作）。而且，我们很少仅仅将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、搜索对象等更复杂的操作。因此，标准库除了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，我们可以简单而高效地排序一个 `Entry` 类型的 `vector`，或是将所有不重复的 `vector` 元素拷贝到一个 `list` 中：

```
void f(vector<Entry>& vec, list<Entry>& lst)
{
    sort(vec.begin(),vec.end());           // 用 < 确定元素顺序
    unique_copy(vec.begin(),vec.end(),lst.begin()); // 不拷贝相邻的重复元素
}
```

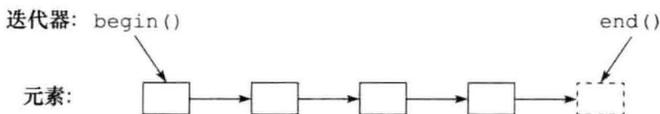
这段代码能正确执行有一个前提：`Entry` 必须定义了小于运算符（`<`）和相等运算符

[107] (`==`)。例如：

```
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
}
```

① 十四世纪前期，英国哲学家奥卡姆的威廉（William of Occam）提出著名的奥卡姆简约律（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

标准库算法都描述为元素（半开）序列上的操作。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` 中所有元素），`sort()` 对此序列进行排序操作。为了写（输出）数据，你只需指明要写的第一个元素。如果写了多个元素，则写入内容会覆盖起始元素之后的那些元素。因此，为了避免错误，`lst` 中的已有元素至少应与 `vec` 中的不重复元素一样多。

如果我们希望将不重复元素存入一个新容器中，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;
}
```

调用 `back_inserter(res)` 为 `res` 创建了一个迭代器，这种迭代器能将元素追加到容器末尾，在追加过程中可扩展容器空间来容纳新元素。这就使我们不必再预先分配一个足够容纳输出元素的空间然后进行填充。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()`）。标准库 `list` 具有移动构造函数（见 4.6.2 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）。

如果觉得 `sort(vec.begin(),vec.end())` 这种使用迭代器对的代码太冗长，可以定义容器版本的算法，代码就能简化为 `sort(vec)`（见 10.7 节）。

10.2 使用迭代器

当拿到一个容器，便可获得一些重要元素的迭代器：`begin()` 和 `end()` 就是最好的例子。此外，很多算法也都返回迭代器。例如，标准库算法 `find` 在一个序列中查找一个值并返回指向所找到元素的迭代器：

```
bool has_c(const string& s, char c)    //s 包含字符 c?
{
    auto p = find(s.begin(),s.end(),c);
    if (p!=s.end())
        return true;
    else
        return false;
}
```

类似很多标准库搜索算法，`find` 返回 `end()` 表示“未找到”。`has_c()` 还有一个更简

洁的版本：

```
bool has_c(const string& s, char c)    // s 包含字符 c?
{
    return find(s.begin(),s.end(),c)!=s.end();
}
```

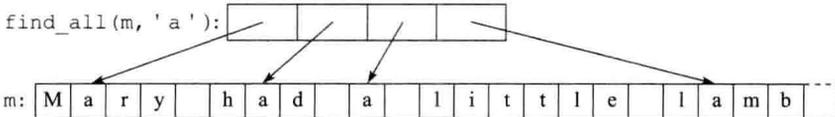
一个更有意思的练习是在字符串中查找一个字符出现的所有位置。我们可以返回一个 string 迭代器的 vector，其中保存“出现位置”的集合。返回一个 vector 是很高效的，因为 vector 提供了移动语义（见 4.6.1 节）。假定我们希望在找到的位置上做修改，就应传递一个非 const 字符串：

```
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() 的调用图示如下：



109

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，我们可以写个模板函数 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 typename 是必要的，它通知编译器：C 的 iterator 是一个类型，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 5.7 节）Iterator 来隐藏这些实现细节：

```

template<typename T>
using Iterator = 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这样的代码完成一些查找任务：

```

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,"red"))
        if (*p!="red")
            cerr << "vector bug!\n";

    for (auto p : find_all(vs,"green"))
        *p = "vert";
}

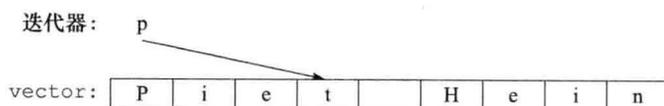
```

110

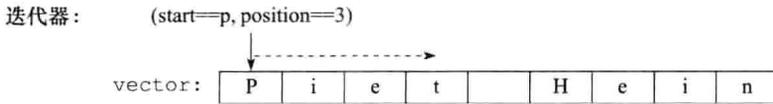
迭代器的重要作用是分离算法和容器（数据结构）。算法通过迭代器来处理数据，但它对存储元素的容器一无所知。反之亦然，容器也对处理其元素的算法一无所知，它所做的全部事情就是按需求提供迭代器（如 `begin()` 和 `end()`）。这种数据存储和算法分离的模型催生出非常通用和灵活的软件。

10.3 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，因为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信息，以便对容器完成某些任务。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，一个 `vector` 迭代器可能就是一个普通指针，因为指针是一种引用 `vector` 中元素的非常合理的方式：

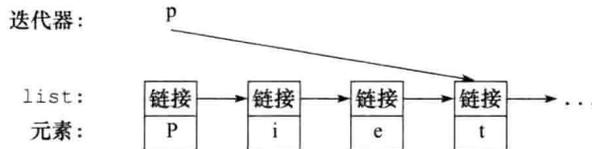


或者，一个 `vector` 迭代器也可以实现为一个指向 `vector`（存储空间起始地址）的指针再加上一个索引：



采用这种实现方式就能进行范围检查。

一个 `list` 迭代器必须是某种比简单指针更复杂的东西，因为一个 `list` 元素通常不知道它的下一个元素在哪里。因此，一个 `list` 迭代器可能是指向一个链接（链表指针）的指针：



111

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。类似地，`*` 运算符会得到迭代器所指向的元素。实际上，任何符合这些简单规则的对象都是一个迭代器——迭代器是一个概念（见 5.4 节）。而且，用户很少需要知道一个特定迭代器的类型，每个容器都“知道”自己对应的迭代器的类型，并以规范的名字 `iterator` 和 `const_iterator` 供用户使用。例如，`list<Entry>::iterator` 是 `list<Entry>` 的迭代器类型，我们很少需要担心“这个类型是如何被定义的”。

10.4 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，元素序列不仅仅出现在容器中。例如，一个输入流产生一个值序列，我们还可以将一个值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
```

这样，向 `*oo` 赋值就会将值打印到 `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信息的新方法。其中 `++oo` 是模仿“通过一个指针向数组中写入值”。

与此类似, `istream_iterator` 允许我们将一个输入流当作一个只读容器来处理。同样, 我们需要指明从哪个流读取数据以及数据类型是什么:

```
istream_iterator<string> ii (cin);
```

与其他迭代器类似, 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 {};
```

通常不直接使用 `istream_iterator` 和 `ostream_iterator`, 而是将它们作为参数传递给算法。例如, 我们可以写出一个简单的程序, 它从一个文件读取数据, 排序读入的单词, 去除重复单词, 最后将结果写入另一个文件中:

```
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 {}; // 输入结束
    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, 用输入初始化
    sort(b.begin(), b.end());    // 排序缓冲区中单词

    unique_copy(b.begin(), b.end(), oo); // 去重拷贝至输出

    return !is.eof() || !os;     // 返回错误状态 (见 1.3 节, 8.4 节)
}
```

112

一个 `ifstream` 就是一个可以绑定到文件的 `istream`, 一个 `ofstream` 就是一个可以绑定到文件的 `ostream` (见 8.7 节)。`ostream_iterator` 构造函数的第二个参数指出输出的间隔符。

实际上, 这个程序本不必这么长。这个版本读取字符串存入一个 `vector` 中, 然后执行 `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。可以将 `string` 保存在一个 `set` 中, 而 `set` 是不会保存重复元素, 而且能维护元素的顺序 (见 9.4 节)。这样, 我们就可以将使用 `vector` 的两行代码改为使用 `set` 的一行代码, 而且也不必再使用 `unique_copy()`, 使用更简单的 `copy()` 就可以了:

```
set<string> b {ii, eos};           // 从输入收集字符串
copy(b.begin(), b.end(), oo);    // 将缓冲区中单词拷贝到输出
```

`ii`、`eos` 和 `oo` 都只使用了一次, 因此我们可以继续精简程序:

```
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"});           // 拷贝到输出
return !is.eof() || !os;                                           // 返回错误状态 (见 1.3 节, 8.4 节)
}

```

至于最终的简化版本是否提高了可读性，就完全是个人偏好和经验的问题了。

10.5 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”的统一处理。但我们常常需要将针对每个元素的处理也作为算法的参数。例如，find 算法（见 10.2 节，10.6 节）提供了一种方便地查找给定值的方法。对于查找满足特定要求的元素这一问题，有一种更为通用的变量作为算法的参数，称为谓词（predicate）。例如，我们可能需要在 map 中搜索第一个大于 42 的值。我们在访问一个 map 的元素时，访问的其实是一个（关键字，值）对的序列。因此，我们可以从 map<string,int> 的元素序列中搜索一个 int 部分大于 42 的 pair<const string,int>:

113

```

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 5.5 节），保存了要比较的值（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 5.5 节）:

```

auto p = find_if(m.begin(), m.end(), [](const pair<string,int>& r) { return r.second>42; });

```

谓词不能改变它所施用的元素。

10.6 标准库算法概览

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] 具有五个重要特性：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，它们都定义在命名空间 std 中，通过头文件 <algorithm> 提供。这些标准库算法都以序列作为输入。一个从 b 到 e 的半开序列表示为 [b:e)。下表是一些算法的简介：

挑选出的一些标准库算法

<code>p=find(b, e, x)</code>	<code>p</code> 是 <code>[b:e)</code> 中第一个满足的 <code>*p==x</code> 迭代器
<code>p=find_if(b, e, f)</code>	<code>p</code> 是 <code>[b:e)</code> 中第一个满足的 <code>f(*p)==true</code> 迭代器
<code>n=count(b, e, x)</code>	<code>n</code> 是 <code>[b:e)</code> 中满足 <code>*q==x</code> 的元素 <code>*q</code> 的数目
<code>n=count_if(b, e, f)</code>	<code>n</code> 是 <code>[b:e)</code> 中满足 <code>f(*q)==true</code> 的元素 <code>*q</code> 的数目
<code>replace(b, e, v, v2)</code>	将 <code>[b:e)</code> 中满足 <code>*q==v</code> 的元素 <code>*q</code> 替换为 <code>v2</code>
<code>replace_if(b, e, f, v2)</code>	将 <code>[b:e)</code> 中满足 <code>f(*q)==true</code> 的元素 <code>*q</code> 替换为 <code>v2</code>
<code>p=copy(b, e, out)</code>	将 <code>[b:e)</code> 拷贝到 <code>[out:p)</code>
<code>p=copy_if(b, e, out, f)</code>	将 <code>[b:e)</code> 中满足 <code>f(*q)==true</code> 的元素 <code>*q</code> 拷贝到 <code>[out:p)</code>
<code>p=move(b, e, out)</code>	将 <code>[b:e)</code> 移动到 <code>[out:p)</code>
<code>p=unique_copy(b, e, out, f)</code>	将 <code>[b:e)</code> 拷贝到 <code>[out:p)</code> ，不拷贝连续的重复元素
<code>sort(b, e)</code>	排序 <code>[b:e)</code> 中的元素，用 <code><</code> 作为排序标准
<code>sort(b, e, f)</code>	排序 <code>[b:e)</code> 中的元素，用谓词 <code>f</code> 作为排序标准
<code>(p1, p2)=equal_range(b, e, v)</code>	<code>[p1:p2)</code> 是已排序序列 <code>[b:e)</code> 的子序列，其中元素的值都等于 <code>v</code> ，本质上等价于二分搜索 <code>v</code>
<code>p=merge(b, e, b2, e2, out)</code>	将两个序列 <code>[b:e)</code> 和 <code>[b2:e2)</code> 合并，结果保存到 <code>[out:p)</code>

114

这些算法以及其他很多算法（例如见 12.3 节）都可以用于容器、string 和内置数组。

一些算法如 `replace()` 和 `sort()` 会修改元素的值，但没有算法会在容器中添加或删除元素。原因在于序列中并不包含底层容器的信息。如果你需要添加元素，就需要使用了解容器信息的特性，如 `back_inserter()`（见 10.1 节），或是直接访问容器本身，如 `push_back()` 或 `erase()`（见 9.2 节）。

标准库算法在设计、规范和实现上通常比自己设计的使用循环的版本更好，因此应该了解标准库算法，使用它们编写程序，而不是从头另起炉灶。

10.7 容器算法

一个序列是通过一对迭代器 `[begin:end)` 定义的，这样的定义方式通用且灵活。但很多时候，算法所操作的序列代表了整个容器的内容。例如：

```
sort(v.begin(), v.end());
```

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<typename C>
    void sort(C& c)
    {
        sort(c.begin(), c.end());
    }

    template<typename C, typename Pred>
    void sort(C& c, Pred p)
    {
        sort(c.begin(), c.end(), p);
    }

    // ...
}
```

我将容器版本的 `sort()`（和其他容器版本的算法）放在它们自己的命名空间 `Estd` 中（“扩展的 `std`”），这样就可以避免与其他程序员使用命名空间 `std` 相互干扰。

10.8 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 32 章中有更加详细的描述。
- [115] [2] 一个标准库算法对一个或多个序列进行操作；参见 10.1 节。
- [3] 一个输入序列是一个半开序列，由一对迭代器所定义；参见 10.1 节。
- [4] 当进行搜索时，算法通常返回输入序列的末尾位置来指出“未找到”；参见 10.1 节。
- [5] 对所处理的序列，算法并不直接在其中添加或删除元素；参见 10.2 节，10.6 节。
- [6] 当编写循环代码时，思考它是否可以表达为一个通用算法；参见 10.2 节。
- [7] 使用谓词和其他函数对象可以使标准库算法有更宽泛的语义；参见 10.5 节，10.6 节。
- [8] 谓词不能修改其参数；参见 10.5 节。
- [9] 了解标准库算法，使用标准库算法而不是自己设计的循环版本编写程序；参见 10.6 节。
- [116] [10] 当“迭代器对”风格的代码变得繁琐时，引入容器版本的算法；参见 10.7 节。

实用工具

能在浪费时间中获得乐趣，就不是浪费时间。

——伯特兰·罗素

- 引言
- 资源管理
 - `unique_ptr` 和 `shared_ptr`
- 特殊容器
 - `array`; `bitset`; `pair` 和 `tuple`
- 时间
- 函数适配器
 - `bind()`; `mem_fn()`; `function`
- 类型函数
 - `iterator_traits`; 类型谓词
- 建议

11.1 引言

并非所有标准库组件都有一个“容器”或“I/O”这样响当当的名字，本章介绍几种不太显眼但是应用非常广泛的组件。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。

11.2 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（显式地或者隐式地）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，比如由于系统资源短缺，程序的运行时间会呈数量级增长。www.wzbook.net

标准库组件在设计时就避免了资源泄漏的问题，为此，组件依赖于成对的构造函数 / 析

构造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，4.2.2 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。例如标准库中的锁：

```
mutex m; // 用于确保共享数据被正确地访问
// ...
void f()
{
    unique_lock<mutex> lck {m}; // 获取互斥锁 m
    // ... 操作共享数据 ...
}
```

一个线程将无法操作共享数据直至 `lck` 的构造函数获取它的互斥锁 (`mutex`) `m` (见 13.5 节)，最后将 `lck` 对应的析构函数负责释放资源。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 `mutex`。

这是“资源获取即初始化”技术 (RAII；见 4.2.2 节) 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) 的方式也是类似的。

11.2.1 `unique_ptr` 和 `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资源。但是如果对象是在自由存储上分配的呢？在 `<memory>` 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。
2. `shared_ptr` 对应共享所有权的情况。

这些“智能指针”最基本的作用是防止因编程疏忽而造成内存泄漏。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{}; // 有可能会抛出异常
    if (j<77) return; // 有可能会“过早地”返回
    // ...
    p->do_something(); // 有可能会抛出异常
    sp->do_something(); // 有可能会抛出异常

    // ...
    delete p; // 销毁 *p
}
```

118

在这段代码中，如果 `i<99` 或者 `j<77`，则我们“忘了”释放指针 `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()` 都会释放它的对象。其实换个角度考虑一下，如果我们干脆不使用指

针，也不使用 new，那么上面的问题就不复存在了：

```
void f(int i, int j)    // 使用局部变量而非指针
{
    X x;
    // ...
}
```

遗憾的是，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，消耗的时空代价比前者小。通过使用 unique_ptr，我们还可以把自由存储上分配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)                // 检查文件是否正确打开
        throw No_file{};

    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指针的所有权，否则别轻易使用 shared_ptr。

之前介绍的创建智能指针的过程从逻辑上看有点儿古怪，代码也稍显冗长，毕竟我们需要先在自由存储上创建一个对象，然后再把指向该对象的指针赋给智能指针。为了解决这一问题，标准库在 <memory> 中提供了一个名为 make_shared() 的函数。其用法如下所示：

```

struct S {
    int i;
    string s;
    double d;
    // ...
};

shared_ptr<S> p1 {new S {1,"Ankh Morpork",4.65}};

auto p2 = make_shared<S>(2,"Oz",7.62);

```

这段代码的含义是：p2 是一个 `shared_ptr<S>`，它指向一个对象，该对象是在自由存储上分配的并且类型是 `S`，对象的内容是 `{1,string{"Ankh Morpork"},4.65}`。

目前在标准库中还没有与 `make_shared()` 和 `make_pair()`（见 11.3.3 节）类似的 `make_unique()`，但是要想定义它其实非常简单：

```

template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args)
{
    return std::unique_ptr<T>{new T{std::forward<Args>(args)...}};
}

```

标准库中没有这个定义，但不代表它没价值，相反，它不但有效而且应用广泛。其中的省略号...表示我们使用了可变参数模板（见 5.6 节）。我们可以编写下面的代码：

```
auto p2 = make_unique<S>(3,"Atlantis",11.3);
```

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 4.2.2 节）。不过，这些“智能指针”在概念上讲仍然是指针，因此我在管理资源时只把它们当成第二选择——容器和其他可以在一个更高的概念层次上管理资源的类型作为第一选择。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排除了资源管理的问题，但是数据竞争（见 13.7 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`。

120

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 4.6.2 节）。

11.3 特殊容器

标准库提供了几种容器（比如内置数组、`array` 和 `string`），它们与 STL 框架（见第

9 章, 第 10 章) 并非完全契合。我有时候会把这些容器叫做“拟容器”, 但是这么说似乎有些不公平。因为它们毕竟存放着元素, 所以确实是容器, 只不过它们各自都添加了一些约束或者额外的特性, 使得把它们放在 STL 的语境中显得有些异类。下表专门介绍这些特殊的容器, 这样有利于把它们的特性讲清楚, 也对后面专门介绍 STL 有益处。

“拟容器”	
<code>T[N]</code>	内置数组: 是一段固定尺寸且连续分配的序列, 包含 N 个 T 类型的元素; 隐式地转换成 T^*
<code>array<T,N></code>	是一段固定尺寸且连续分配的序列, 包含 N 个 T 类型的元素; 与内置数组类似, 但是解决了很多问题
<code>bitset<N></code>	是一段固定大小的序列, 包含 N 位
<code>vector<bool></code>	是一段位的序列, 紧密地存储在一个特殊的 <code>vector</code> 中
<code>pair<T,U></code>	两个元素, 类型分别是 T 和 U
<code>tuple<T...></code>	是一段序列, 存放着任意类型的任意个元素
<code>basic_string<C></code>	是一段字符的序列, 字符的类型是 C ; 提供字符串操作
<code>valarray<T></code>	是一个数组, 包含 T 类型的数值; 提供数值操作

标准库为什么要提供这么多容器类型呢? 因为它们能满足各种常见且功能各异 (时常会有交叠) 的需求。假设标准库没有提供这些容器, 那么程序员就不得不自己实现它们。例如:

- `pair` 和 `tuple` 是异构的; 而其他所有容器都是同构的 (所有元素的类型相同)。
- `array`、`vector` 和 `tuple` 的元素是连续分配的; 而 `forward_list` 和 `map` 是链接结构。
- `bitset` 和 `vector<bool>` 存放的是位 (bit), 并通过起代理作用的对象来访问; 其他所有容器存放任意类型的元素, 并且可以直接访问元素。
- `basic_string` 要求它的元素是某种类型的字符并且提供字符串操作, 比如连接操作和基于位置的操作等。
- `valarray` 要求它的元素是数字并且提供数字操作。

所有这些容器都提供了针对特定程序员群体的特殊服务。没有任何一个容器能满足所有需求, 因为有些需求本身就是矛盾的。例如: “可扩充性” 和 “确保所占空间固定” 是矛盾的; “执行加法运算时不移动元素” 和 “连续分配” 也是矛盾的。除此之外, 非常通用的容器常常意味着比特定的容器有更多开销, 而这些额外的开销根本是不必要的。

[121]

11.3.1 array

在 `<array>` 中定义的 `array` 表示一个尺寸固定的元素序列, 元素的数量在编译时指定。因此, `array` 的元素可以位于栈中或者对象内, 也可以位于静态存储空间。元素所属的作用域就是定义 `array` 的作用域。要想更好地理解 `array`, 我们不妨将其与内置数组比较。`array` 的特点是尺寸固定、不会隐式地转换成指针 (内置数组的这种自动转换有时候并非程序员所愿), 并且提供了一些便于操作的函数, 同时使用 `array` 的时空代价也并不比使用内置数组大。`array` 不承担 STL 容器作为 “元素句柄” 的作用, 相反, `array` 直接包含着元素。

我们可以用初始值列表初始化一个 `array`:

```
array<int,3> a1 = {1,2,3};
```

其中，初始值的数量不能多于 array 指定的元素数量。

编译器不允许我们省略 array 的元素数量：

```
array<int> ax = {1,2,3};    // 错误：没有指定 array 的元素数量
```

并且元素的数量必须是一个常量表达式：

```
void f(int n)
{
    array<string,n> aa = {"John's", "Queens' "};    // 错误：array 的元素数量必须是常量表达式
    //
}
```

如果你希望元素的数量可变，应该使用 vector。

我们也可以在必要的时候把 array 传递给一个需要指针的 C 风格函数。例如：

```
void f(int* p, int sz);    // C 风格的接口

void g()
{
    array<int,10> a;

    f(a.size());           // 错误：此处不存在期望的隐式类型转换
    f(&a[0],a.size());     // C 风格的用法
    f(a.data(),a.size());  // C 风格的用法

    auto p = find(a.begin(),a.end(),777);    // C++/STL 风格的用法
    // ...
}
```

有一个问题值得探讨：既然已经有了 vector，而且它的用法非常灵活，我们为什么还要使用 array 呢？答案显而易见：array 虽然不太灵活，但是它比 vector 简单。有时候，直接访问栈当中的元素比访问自由存储中的元素有非常明显的性能提升，因为我们需要通过 vector 句柄间接地访问后者，然后还得负责释放这些资源。不过 array 也有缺点，栈空间毕竟非常有限（尤其是在嵌入式系统中），一旦发生栈溢出，后果不堪设想。

[122]

另外一个问题是：array 与内置数组相比又有哪些优点呢？首先，array 的尺寸固定，因此易于使用标准库算法；其次，array 能够执行拷贝操作（赋值或初始化）；最重要的一点是 array 不会自动转换成指针，因为这种自动转换很多时候是程序员不希望看到的。下面的代码很好地说明了这一点：

```
void h()
{
    Circle a1[10];
    array<Circle,10> a2;
    // ...
    Shape* p1 = a1;    // 语法上正确，但是存在严重的隐患
    Shape* p2 = a2;    // 报告语法错误：禁止 <Circle,10> 自动转换成 Shape* 类型，从而避免了风险
    p1[3].draw();     // 程序错误
}
```

如果 `sizeof(Shape) < sizeof(Circle)`，则通过 `Shape*` 指针调用 `Circle[]` 的下标运算符会导致访问内存时产生错误的偏移量。因此，我们在注释中给出了“程序错误”

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` 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) // 输出所有相等的记录
        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。我们可以容易地用 `make_pair()` 函数创建一个 `pair`，而无需显式指定它的类型。例如：

```
void f(vector<string>& v)
{
    auto pp = make_pair(v.begin(), 2); // pp 的类型是 pair<vector<string>::iterator, int>
    // ...
}
```

124

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）。`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); // 获取 tuple 的第二个元素: 10
double d = get<2>(t); // 获取 tuple 的第三个元素: 1.23
```

`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` 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]`。

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坏的标志位），所以经常会用到 `pair`。而同时返回三个或者更多结果的时候并不太多见，因而 `tuple` 一般更多用于实现泛型算法。

11.4 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 3.3 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>` 的子空间 `std::chrono` 中。

判断程序“效率”的最有效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。

11.5 函数适配器

函数适配器接受一个函数作为它的参数，返回的结果是一个函数对象，我们可以使用这个函数对象调用原来的函数。标准库使用 `bind()` 和 `mem_fn()` 适配器绑定参数，这一过程也称为柯里化（Currying）或者偏函数评价（partial evaluation）。过去的代码习惯于使用绑定的方式，现在我们用 `lambda` 表达式（见 5.5 节）就可以满足大多数需求。

125

11.5.1 `bind()`

假设有一个函数和一组参数，则利用 `bind()` 可以生成一个函数对象，该对象用“剩余的”参数（如果有的话）调用函数。例如：

```
double cube(double);

auto cube2 = bind(cube,2);
```

当我们调用 `cube2()` 函数时，会用参数 2 调用 `cube` 函数，即 `cube(2)`。我们不必每次都绑定函数的所有参数，例如：

```
using namespace placeholders;

void f(int,const string&);
auto g = bind(f,2,_1); // 把 f() 的第一个参数绑定为 2
f(2,"hello");
g("hello"); // 等同于调用 f(2,"hello");
```

其中，参数 `_1` 表示一个占位符，它指定 `bind()` 所得函数对象的参数应该用在何处。在此例中，`g()` 的第一个参数被用作 `f()` 的第二个参数。

占位符定义在 `<functional>` 中，属于子命名空间 `std::placeholders`。
 如果想为重载函数绑定参数，必须显式地指定被绑定的是重载函数的哪个版本：

```
int pow(int,int);
double pow(double,double); // 重载了函数 pow()

auto pow2 = bind(pow, _1,2); // 错误：绑定的是哪个 pow()？
auto pow2 = bind((double(*))(double,double))pow, _1,2); // OK（但是形式上比较糟糕）
```

在之前的几段程序中，我们使用 `auto` 关键字声明变量，并用它存放 `bind()` 的结果，这样做的好处是我们不必费力思考 `bind()` 的结果到底是什么类型。函数本身的类型以及函数绑定的参数值都对最终所得的结果类型有影响。此外，由于返回的函数对象需要存储绑定的参数值，所以一般来说函数对象会比较大。当我们需要明确所需的参数类型以及返回结果的类型时，可以使用 `function`（见 11.5.3 节）。

11.5.2 mem_fn()

函数适配器 `mem_fn(mf)` 生成一个函数对象，我们能像调用非成员函数一样调用这个函数对象。例如：

```
void user(Shape p)
{
    p->draw();
    auto draw = mem_fn(&Shape::draw);
    draw(p);
}
```

某些算法需要它的操作以非成员函数的方式调用，`mem_fn()` 通常用在这些算法中。

126 例如：

```
void draw_all(vector<Shape>& v)
{
    for_each(v.begin(),v.end(),mem_fn(&Shape::draw));
}
```

所以，`mem_fn()` 可以被看作是一种从面向对象的调用到面向函数的调用的映射。

在很多情况下，`lambda` 表达式可以替代绑定，而且更简单也更通用。例如：

```
void draw_all(vector<Shape>& v)
{
    for_each(v.begin(),v.end(),[](Shape p){ p->draw(); });
}
```

11.5.3 function

`bind()` 能被直接使用，也能用来初始化一个 `auto` 变量，在这一点上 `bind()` 类似于 `lambda`。

如果我们想把 `bind()` 的结果赋给某个指定类型的变量，则应该使用标准库类型 `function`。我们通过指定返回类型和参数类型来指定一个 `function`。例如：

```
int f1(double);
function<int(double)> fct {f1}; // 初始化为 f1
```

```
int f2(int);

void user()
{
    fct = [](double d) { return round(d); }; // 把 lambda 赋给 fct
    fct = f1;                               // 把 function 赋给 fct
    fct = f2;                               // 错误: 参数类型不正确
}
```

标准库 `function` 是一种数据类型，它可以存放任意对象，只要该对象能被调用运算符 `()` 调用。也就是说，类型 `function` 的对象是一个函数对象（见 5.5 节）。例如：

```
int round(double x) { return static_cast<int>(floor(x+0.5)); } // 传统的四舍五入

function<int(double)> f; // f 能存放任意对象，只要该对象接受一个 double 且返回一个 int

enum class Round_style { truncate, round };

struct Round { // 存有状态信息的函数对象
    Round_style s;
    Round(Round_style ss) :s(ss) {}
    int operator()(double x) const { return static_cast<int>((s==Round_style::round) ? (x+0.5) : x); };
};
```

其中，我们使用 `static_cast`（见 14.2.3 节）显式地指定返回类型是 `int`。

127

```
void t1()
{
    f = round;
    cout << f(7.6) << '\n'; // 用 f 调用函数 round

    f = Round(Round_style::truncate);
    cout << f(7.6) << '\n'; // 调用函数对象

    Round_style style = Round_style::round;
    f = [style] (double x){ return static_cast<int>((style==Round_style::round) ? x+0.5 : x); };

    cout << f(7.6) << '\n'; // 调用 lambda

    vector<double> v {7.6};
    f = Round(Round_style::round);
    std::transform(v.begin(),v.end(),v.begin(),f); // 传递给算法

    cout << v[0] << '\n'; // 被 lambda 改变了
}
```

程序的运行结果是 8、7、8、8。

显然，`function` 可用于回调函数或者把函数作为实参传给算法等情形。

11.6 类型函数

类型函数（`type function`）是指在编译期求值的函数，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，这些类型函数帮助库实现者和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 12.7 节）。例如：

```
constexpr float min = numeric_limits<float>::min(); // 最小的正浮点数
```

与之类似，我们可以使用内置的 `sizeof` 运算符（见 1.5 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++ 编译期计算机制的一部分，它允许程序进行轻量级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）或者当含有模板时称为模板元编程（template metaprogramming）。接下来，我们介绍标准库提供的两种有用功能：`iterator_traits`（见 11.6.1 节）和类型谓词（见 11.6.2 节）。

11.6.1 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（见第 10 章）。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。`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 5.2 节）。

可以用标准库提供的 `iterator_traits` 检查当前容器支持哪种迭代器，这样我们就能够让 10.7 节介绍的 `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，则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函数。该函数接受一个额外的实参以指示它们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<Value_type<For>> 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
}
```

`Value_type<For>` 是 `For` 的元素类型，也称为它的值类型（value type）。每个标准库迭

代器都含有一个 `value_type` 成员，我们通过定义类型别名（见 5.7 节）得到了 `value_type<For>`：

```
template<typename C>
    using Value_type = typename C::value_type; // C 的值类型
```

这样 `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”发生在选择辅助函数的过程中：

```
template<typename 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： [129]

- 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，我们就能够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），它是一种在标准库和其他地方经常用到的提高程序灵活性和效率的方法。

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。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，你大可以忽略像 `iterator_traits` 这样的功能；不过与此同时，也就没办法利用它们来改进你的代码了。

11.6.2 类型谓词

标准库类型谓词是一种简单的类型函数，它负责回答关于类型的基本问题。例如：

```
bool b1 = is_arithmetic<int>(); // 是：int 是一种算术类型
bool b2 = is_arithmetic<string>(); // 否：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，它们包括 `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，我们定义了一个类型函数：

```
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。

[130]

11.7 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 33 ~ 35 章有更加详细的描述。
- [2] 对于库来说，大而全不如小而精；参见 11.1 节。
- [3] 所谓资源是指需要先获取后释放（显式或隐式）的东西；参见 11.2 节。
- [4] 用资源句柄来管理资源（RAII）；参见 11.2 节。
- [5] 用 `unique_ptr` 访问多态类型的对象；参见 11.2.1 节。
- [6] 用 `shared_ptr` 访问共享的对象；参见 11.2.1 节。
- [7] 与智能指针相比，优先选择含有特定语义的资源句柄；参见 11.2.1 节。
- [8] 与 `shared_ptr` 相比，优先选择 `unique_ptr`；参见 4.6.4 节，11.2.1 节。
- [9] 与普通的垃圾回收机制相比，智能指针更优；参见 4.6.4 节，11.2.1 节。
- [10] 在需要 `constexpr` 大小的序列的地方使用 `array`；参见 11.3.1 节。
- [11] `array` 比内置数组更好；参见 11.3.1 节。
- [12] 如果你需要使用 N 个二进制位，而 N 又并非恰好是某种内置整数类型的大小，则建议使用 `bitset`；参见 11.3.2 节。
- [13] 使用 `pair` 时，`make_pair()` 可以帮助我们进行类型推断；参见 11.3.3 节。
- [14] 使用 `tuple` 时，`make_tuple()` 可以帮助我们进行类型推断；参见 11.3.3 节。
- [15] 别轻易抱怨程序的效率低下，记得用事实说话；参见 11.4 节。
- [16] 在报告程序的执行时间时，用 `duration_cast` 把结果转换到一个适当的单位上；参见 11.4 节。
- [17] 通常情况下，`lambda` 可以起到与 `bind()` 和 `mem_fn()` 类似的作用；参见 11.5 节。
- [18] 用 `bind()` 创建函数和函数对象的变形；参见 11.5.1 节。
- [19] 用 `mem_fn()` 创建一个函数对象，通过它我们就能用传统的函数调用方式调用成员函数了；参见 11.5.2 节。
- [20] 用 `function` 存储某些能被调用的东西；参见 11.5.3 节。
- [21] 编写代码时可以显式地令其利用某些类型的属性；参见 11.6 节。

[131]
?
[132]

数值计算

计算的意义在于洞察力，而非数字本身。

——理查德·汉明

……但是对于学生而言，数字通常是培养洞察力最好的途径。

——A. 罗尔斯顿

- 引言
- 数学函数
- 数值算法
- 复数
- 随机数
- 向量算术
- 数值限制
- 建议

12.1 引言

当初人们设计 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，比如数据库访问、网络系统、设备控制、图形学、仿真和金融分析等，C++ 也逐渐成为了在大型系统中执行计算任务的一个有力竞争者。与此同时，数值计算远不止于用简单循环来处理浮点数序列。当计算规模越大、所需的数据结构越复杂，C++ 就越能发挥它的威力。实际情况是，C++ 被广泛应用于科学计算、工程计算、金融计算和其他含有复杂数值的计算任务中，而支持这类计算的功能和技术也逐渐发展起来。本章主要介绍标准库中支持数值计算的部分。

133

12.2 数学函数

在 `<cmath>` 中包含着很多标准数学函数 (standard mathematical function)，如参数类型为 `float`、`double` 和 `long double` 的 `sqrt()`、`log()` 和 `sin()` 函数：

标准数学函数

<code>abs(x)</code>	绝对值
<code>ceil(x)</code>	大于等于 x 的整数中最接近它的那个
<code>floor(x)</code>	小于等于 x 的整数中最接近它的那个
<code>sqrt(x)</code>	平方根, x 必须非负
<code>cos(x)</code>	余弦值
<code>sin(x)</code>	正弦值
<code>tan(x)</code>	正切值
<code>acos(x)</code>	反余弦值, 结果非负
<code>asin(x)</code>	反正弦值, 返回最接近 0 的结果
<code>atan(x)</code>	反正切值
<code>cosh(x)</code>	双曲余弦
<code>sinh(x)</code>	双曲正弦
<code>tanh(x)</code>	双曲正切
<code>exp(x)</code>	e 的指数
<code>log(x)</code>	自然对数, 以 e 为底, x 必须为正
<code>log10(x)</code>	以 10 为底的对数

与 `complex` (见 12.4 节) 有关的数学函数被定义在 `<complex>` 中。每个函数的返回类型与其实参的类型相同。

报告错误的方式是设置来自 `<cerrno>` 的 `errno`, 在出现定义域错误的地方将它设为 `EDOM`, 在值域错误的地方设为 `ERANGE`。例如:

```
void f()
{
    errno = 0; // 清除失效的错误状态
    sqrt(-1);
    if (errno == EDOM)
        cerr << "sqrt() not defined for negative argument";

    errno = 0; // 清除失效的错误状态
    pow(numeric_limits<double>::max(), 2);
    if (errno == ERANGE)
        cerr << "result of pow() too large to represent as a double";
}
```

在 `<cstdlib>` 中包含了其他几个数学函数。除此之外还有一个面向特殊数学函数 [134] (special mathematical function) 的专门的 ISO 标准 [C++Math, 2010]。

12.3 数值算法

在 `<numeric>` 中有一些泛化的数值算法, 比如 `accumulate()`。

数值算法 (iso.26.7)

<code>x = accumulate(b, e, i)</code>	x 是 i 以及 $[b: e)$ 范围内所有元素的和
<code>x = accumulate(b, e, i, f)</code>	用 f 代替 + 执行 <code>accumulate</code> 操作
<code>x = inner_product(b, e, b2, i)</code>	x 是 $[b: e)$ 和 $[b2: b2+(e-b))$ 的内积, 也就是 i 和所有 $(*p1) * (*p2)$ 的和, 其中 $p1$ 对应 $[b: e)$, $p2$ 对应 $[b2: b2+(e-b))$
<code>x = inner_product(b, e, b2, i, f, f2)</code>	用 f 和 $f2$ 代替 + 和 * 执行 <code>inner_product</code> 操作
<code>p = partial_sum(b, e, out)</code>	$[out: p)$ 的元素 i 是 $[b: b+i)$ 中元素的和

(续)

数值算法 (iso.26.7)

<code>p = partial_sum(b, e, out, f)</code>	用 <code>f</code> 代替 + 执行 <code>partial_sum</code> 操作
<code>p = adjacent_difference(b, e, out)</code>	[out: p) 的元素 <code>i</code> 是 $*(b+i) - *(b+i-1)$, 其中 $i > 0$; 如果 $e - b > 0$, 则 <code>*out</code> 是 <code>*b</code>
<code>p = adjacent_difference(b, e, out, f)</code>	用 <code>f</code> 代替 - 执行 <code>adjacent_difference</code> 操作
<code>iota(b, e, v)</code>	为 [b: e) 的每个元素依次赋值 <code>++v</code> , 赋值后的结果序列是 <code>v+1, v+2, …</code>

上述算法泛化了一些非常常见的操作 (比如求和操作), 使其可以作用于各种不同类型的序列, 其中, 在序列元素上执行的运算被作为参数传递给算法。对于每个算法来说, 在泛化的版本之外都有一个专门的应用了最常见运算的版本。例如:

```
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参数 (见 12.3 节)。

12.4 复数

标准库提供了一系列复数类型, 其形式与 4.2.1 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 = {});
    // ...
};
```

135

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>` (见 12.2 节) 定义了一些常见的数学函数, `sqrt()` 和 `pow()` (求幂指数) 即在其中。

12.5 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>` 中提供了多种不同的随机数发生器。随机数发生器包括两部分：

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。

一些典型的分布包括 `uniform_int_distribution`（生成的所有整数概率相等）、`normal_distribution`（正态分布，又名“铃铛曲线”）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);                   // 得到一个随机数发生器

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，它会把第二个参数（`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 11.5.1 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环境和读者自己的感觉。

对于初学者来说，完整版本的随机数发生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}; // 构建一个随机数发生器，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发生器呢？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()
{
    constexpr int max = 8;
```

```

Rand_int rnd (0,max);           // 创建一个随机数发生器

vector<int> histogram(max+1);    // 构建一个相应尺寸的 vector
for (int i=0; i!=200; ++i)
    ++histogram[rnd()];        // 用 [0:max] 之间每个数字出现的次数填充 vector

for (int i = 0; i!=histogram.size(); ++i) { // 输出柱状图
    cout << i << '\t';
    for (int j=0; j!=histogram[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  ████████████████████████████████████████
5  ████████████████████████████████████████
6  ████████████████████████████████████████
7  ████████████████████████████████████████
8  ████████████████████████████████████████
9  ████████████████████████████████████████

```

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，但是在本书中我们只用到 ISO 标准之内的功能。

137

12.6 向量算术

9.2 节介绍的 `vector` 被设计成一种通用机制，它可以存放值并且足够灵活，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];
    // ...
}

```

值得注意的是，`valarray` 为实现多维运算提供了足够的支持。

12.7 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。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!");
```

因为 `numeric_limits<int>::max()` 是一个 `constexpr` 函数（见 1.7 节），所以第二个断言是有效的。

12.8 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 40 章有更加详细的描述。
- [2] 数值问题非常微妙。如果你对于某个问题的数学含义不是 100% 肯定，一定要征询专家的建议或者做实验验证；参见 12.1 节。
- [3] 解决重要的数学计算问题时一定要充分利用库，而不仅仅是语言本身；参见 12.1 节。
- [4] 如果想从序列中计算某个结果，优先考虑使用 `accumulate()`、`inner_product()`、`partial_sum()` 或者 `adjacent_difference()`，实在不行再用循环；参见 12.3 节。
- [5] 用 `std::complex` 解决复数运算的问题；参见 12.4 节。
- [6] 把引擎绑定到某个分布上以得到一个随机数发生器；参见 12.5 节。
- [7] 确保你的随机数足够随机；参见 12.5 节。
- [8] 如果运行时效率比操作和元素类型的灵活性更重要的话，应该使用 `valarray`；参见 12.6 节。
- [9] 用 `numeric_limits` 可以访问数值类型的属性；参见 12.7 节。
- [10] 用 `numeric_limits` 来检查数值类型是否能够满足特定计算需求；参见 12.7 节。

并 发

保持简单：尽可能地简单，但不要过度简化。

——A. 爱因斯坦

- 引言
- 任务和 `thread`
- 传递参数
- 返回结果
- 共享数据
- 等待事件
- 任务通信

`future` 和 `promise`; `packaged_task`; `async()`

- 建议

13.1 引言

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而另一部分继续执行）。所有现代程序设计语言都对并发提供了支持。C++ 标准库并发特性的前身在 C++ 中已应用了 20 多年，经过对可移植性和类型安全的改进，成为了标准库的一部分，它几乎适用于所有现代硬件平台。标准库并发特性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。基于标准库并发特性，可以构建出这类高层并发模型，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和一套原子操作。原子操作可实现无锁的并发程序设计 [Dechev, 2010]。而内存模型则保证了：只要程序员避免了数据竞争（对可变数据的不受控制的并发访问），程序运行结果就是可预料的。但是，大多数用户眼中的并发就是标准库特性以及建立在其上的其他库。因此，本章简要介绍主要的标准库并发特性：`thread`、`mutex`、`lock()` 操作、`packaged_task` 和 `future`，并给出一些示例。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性能开销，当然也不保证有显著性能提升。

[141]

不要将并发看作灵丹妙药。如果串行执行已经能很好地完成任务，那么使用串行程序就好了，这通常更简单也更快。

13.2 任务和 thread

我们把可与其他计算并行执行的计算称为任务 (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，可构造一个 `std::thread` (可在 `<thread>` 中找到)，将任务作为它的参数。这里，任务是以函数或函数对象的形式实现的：

```
void f();           // 函数

struct F {         // 函数对象
    void operator(); // F 调用运算符 (见 5.5 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()`。“会合” (`join`) 一个 `thread` 表示“等待该线程结束”。

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 13.5 节) 相互通信。通常通过锁或其他机制防止数据竞争 (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` (一个函数) 和 `F` (一个函数对象) 可以这样实现：

```
void f() { cout << "Hello "; }

struct F {
    void operator()() { cout << "Parallel World!\n"; }
};
```

[142] 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`，而没有采取任何形式的同步。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，因为两个任务中的操作的执行顺序是不确定的。程序可能会产生下面这样“奇怪的”输出：

```
PaHeralllel o World!
```

当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，而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将它看作一个可以与调用者并发执行的函数。为此，我们只需传递参数、获取结果并保证两者不同时使用共享数据 (没有数据竞争)。

13.3 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 5.5 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,ref(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，并希望它在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, ref(some_vec)}` 初始化一个线程，这使用了 `thread` 的可变参数模板构造函数，它接受一个任意的参数序列（见 5.6 节）。`ref()` 是 `<functional>` 中定义的一个类型函数，很不幸，我们必须使用它来告诉可变参数模板将 `some_vec` 作为一个引用而不是一个对象来处理。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，传递给线程。因此，`F::operator()()` 与 `f()` 执行相同的算法，两个任务的处理是大致相同的：都为 `thread` 构造了一个函数对象来执行任务。

143

13.4 返回结果

在 13.3 节的例子中，我通过一个非 `const` 引用向线程传递参数。只有在希望任务修改引用所指向的数据时，我才会这么做（见 1.8 节）。这种返回结果的方法有些不正规，但并不罕见。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, cref(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';
}

```

这种技术很有效也很常见，但我不认为通过参数返回结果是一种很优雅的方法，因此我将在 13.7.1 节再次讨论这个问题。

13.5 共享数据

有时任务间需要共享数据。在此情况下，数据访问就必须进行同步，使得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很简单（例如，很多任务同时读取不变的数据是没有任何问题的），但请考虑如何确保在同一时刻至多有一个任务可以访问一组给定的对象。

此问题解决方法的基础是“互斥对象” mutex。一个 thread 使用 lock() 操作来获取一个互斥对象：

144

```

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()）。当一个 mutex 被释放，等待此 mutex 的 thread 会恢复执行（“被唤醒”）。互斥和锁机制在头文件 <mutex> 中提供。

共享数据和 mutex 间是一种常规的对应关系：程序员必须知道哪个 mutex 对应哪个数据。显然，这很容易出错，我们应努力借助多种语言特性来使这种对应关系更为清晰：

```

class Record {
public:
    mutex rm;
    // ...
};

```

对于一个名为 `rec` 的 `Record`，不难猜测：`rec.rm` 是一个 `mutex`，你在访问 `rec` 其他数据前应获取这个互斥对象。也许，加一条注释或更好的命名能让它更容易理解。

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);                // 获取所有三个锁
    //... 处理共享数据 ...
} // 隐式释放所有 mutex
```

`lock()` 调用只有在获取了所有 `mutex` 后才会继续执行，当它持有任意一个 `mutex` 时，绝不会阻塞（“睡眠”）（从而不会导致死锁）。每一个 `unique_lock` 的析构函数保证了当 `thread` 离开作用域时对应的 `mutex` 会被释放。

145

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各种任务已经执行哪些工作以及尚未执行哪些工作。在这方面，使用共享数据不如调用 - 返回模式。另一方面，一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，但加锁和解锁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“效率”而不经思考、不经测试就选择共享数据方式来进行线程间通信。

13.6 等待事件

有时，一个 `thread` 需要等待某种外部事件，如另一个 `thread` 完成了一个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。使用 `<chrono>` 中的时间相关的特性，可以写出如下代码：

```
using namespace std::chrono; // 见 11.4 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` 默认指向唯一的当前线程。

我使用 `duration_cast` 将时钟单位调整为期望的纳秒。

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。`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（通常称为一个事件）发生，这种条件通常是其他 thread 完成工作产生的结果。

`condition_variable` 支持很多种优雅而高效的共享方式，但也有可能变得相当复杂。考虑两个 thread 通过一个 queue 传递消息来通信的经典例子。简单起见，声明 queue 对象以及生产者、消费者共享 queue 而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
```

其中类型 `queue`、`condition_variable` 和 `mutex` 由标准库提供。

146 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...
    }
}
```

这里，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（在作用域结束处）
}
```

13.7 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。

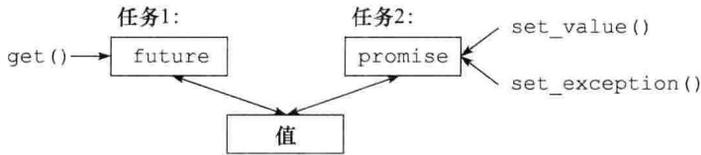
1. `future` 和 `promise` 用来从一个独立线程上创建出的任务返回值。
2. `package_task` 是帮助启动任务以及连接返回结果的机制。

3. `async()` 以类似调用函数的方式启动一个任务。

以上这些特性都定义在 `<future>` 中。

13.7.1 future 和 promise

`future` 和 `promise` 的关键点是它们允许在两个任务间传输一个值，而无需使用锁——“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一个值时，它将值放入一个 `promise` 中。具体 C++ 实现以某种方式令这个值出现在对应的 `future` 中，然后就可以从中读取这个值了（通常是任务的启动者读取此值）。这种模式如下图所示：[147]



如果我们有一个名为 `fx` 的 `future`，那么可以用 `get()` 从它获取一个类型为 `x` 的值。

```
X v = fx.get(); // 如必要，等待值被计算出来
```

如果值还未准备好，线程会阻塞直至值准备好。如果值不能正确准备好，`get()` 会抛出一个异常（可能是系统抛出的，或是从用 `get()` 获取值的任务传递来的）。

`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`）的命名是沿用了历史，现实中像这样的双关语有很多。

如果你有一个 `promise`，需要将一个类型为 `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
        px.set_exception(current_exception()); // 将异常传递给 future 的线程
    }
}
```

`current_exception()` 表示当前被捕获的异常。

为了处理经过 `future` 传递的异常，`get()` 的调用者必须准备好捕获异常。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
        // ... use v ...
    }
    catch (...) { // 糟糕：v 不能正确计算好
```

```

        // ... 处理错误 ...
    }
}

```

148

如果错误无需由 `g()` 自己处理，则代码可以最简化：

```

void g(future<X>& fx)          // 一个任务：从 fx 获取结果
{
    // ...
    X v = fx.get(); // 如必要，等待值准备好
    // ... 使用 v ...
}

```

13.7.2 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 `thread` 上的 `future` 和 `promise` 的相关设置。一个 `packaged_task` 提供了一层包装代码，实现将某个任务的返回值或异常放入一个 `promise` 中（如 13.7.1 节中代码所示）。如果你通过调用 `get_future()` 来向一个 `packaged_task` 发出请求，它会返回对应的 `promise` 的 `future`。例如，我们可以将两个任务连接起来，它们分别使用标准库 `accumulate()`（见 12.3 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();                                       // 获得结果
}

```

`packaged_task` 模板接受一个模板实参表示任务的类型（本例中为 `Task_type`，`double(double*, double*, double)` 的别名），并接受任务作为其构造函数的实参（本例中为 `accum`）。`move()` 操作是必需的，因为 `packaged_task` 不能被拷贝。原因在于 `packaged_task` 是一种资源句柄：它拥有一个 `promise` 且（间接）负责其任务所拥有的资源。

149

请注意这段代码没有显式使用锁：通过使用 `packaged_task`，我们可以将精力集中于要完成的任务，而不必操心用来管理其通信的机制。两个任务运行于两个独立的线程，因此可以并行执行。

13.7.3 `async()`

本章所遵循的思路是：将任务当作可以与其他任务并发执行的函数来处理，这也是各种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，可以根据需要使用。

如需启动可异步运行的任务，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) // 值得使用并发机制吗?
        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。使用 `async()`，你不必再操心线程和锁，只需考虑可能异步执行的任务。但这显然有一个限制：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，`async()` 根据调用发生时它所了解的系统可用资源量来确定使用多少个 `thread`。

使用猜测的方式确定计算量与 `thread` 启动开销的相对比例是一种很原始的方法，而且容易得到错误的结论（例如使用 `v.size() < 10000`）。但是，我们不可能在本节详细讨论如何管理 `thread`。因此，记住这种估计只不过是一个简单而且可能很糟糕的实现，不要在实际代码中使用它。

请注意，`async()` 并非一个专门为提高并行计算性能所设计的机制。例如，我们还可以用它来创建一个从用户获取信息的任务，而让“主程序”继续进行其他计算（见 13.7.3 节）。 150

13.8 建议

[1] 本章内容在 [Stroustrup, 2013] 的第 41 ~ 42 章中有更加详细的描述。

[2] 用并发提高响应速度或提高吞吐率；参见 13.1 节。

- [3] 只要性能可接受, 就应使用尽可能高层的抽象; 参见 13.1 节。
- [4] 将进程看作线程的一个可选的替代; 参见 13.1 节。
- [5] 标准库并发特性是类型安全的; 参见 13.1 节。
- [6] 内存模型可以免去大多数程序员在机器体系结构层面思考计算机的麻烦; 参见 13.1 节。
- [7] 内存模型使内存大致如程序员之朴素期望那样呈现; 参见 13.1 节。
- [8] 原子操作使程序员可以进行无锁的程序设计; 参见 13.1 节。
- [9] 无锁程序设计还是留给专家吧; 参见 13.1 节。
- [10] 有时串行版本比并发版本更简单也更快; 参见 13.1 节。
- [11] 避免数据竞争; 参见 13.1 节, 13.2 节。
- [12] `thread` 是系统线程的类型安全的接口; 参见 13.2 节。
- [13] 用 `join()` 等待 `thread` 结束; 参见 13.2 节。
- [14] 只要可能就避免显式数据共享; 参见 13.2 节。
- [15] 使用 `unique_lock` 管理互斥对象; 参见 13.5 节。
- [16] 使用 `lock()` 获取多个锁; 参见 13.5 节。
- [17] 使用 `condition_variable` 管理 `thread` 间通信; 参见 13.6 节。
- [18] 从并发执行的任务的角度思考并发程序设计, 而不是直接从 `thread` 角度思考; 参见 13.7 节。
- [19] 追求简洁; 参见 13.7 节。
- [20] 优先使用 `packaged_task` 和 `future`, 而不是直接使用 `thread` 和 `mutex`; 参见 13.7 节。
- [21] 使用 `promise` 返回结果, 从 `future` 获取结果; 参见 13.7.1 节。
- [22] 用 `packaged_task` 处理任务抛出的异常并管理返回值; 参见 13.7.2 节。
- [23] 使用 `packaged_task` 和 `future` 表达传递给外部服务的请求并等待应答; 参见 13.7.2 节。
- [24] 使用 `async()` 启动简单任务; 参见 13.7.3 节。

151
152

历史和兼容性

欲速则不达。

——屋大维，凯撒·奥古斯都

- 历史
 - 大事年表；早期的 C++；ISO C++ 标准
- C++11 扩展
 - 语言特性；标准库组件；已弃用特性；类型转换
- C/C++ 兼容性
 - C 和 C++ 是兄弟；兼容性问题
- 参考文献
- 建议

14.1 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]，同时为系统程序设计提供 C 的效率和灵活性 [Kernighan, 1978]。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 贝尔实验室的同事对 C++ 头十年的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》(Design and Evolution of C++) 一书（人们熟知的“D&E”）详细介绍了 C++ 的设计和演化以及 C++ 受其他程序设计语言影响的文档材料。

大多数作为 ISO C++ 标准的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, 我设法维护标准库特性与其提出者和改进者之间的关联 [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“独裁者”的作品, 而是千万个甘于奉献、有经验、辛勤工作的人的劳动结晶。

14.1.1 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++。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 (简称“TC++PL”, 10 月 14 日) [Stroustrup, 1986]。

1989 《C++ 参考手册批注版》出版 (简称“the ARM”) [Ellis, 1989]。

1991 《C++ 程序设计语言 (第二版)》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“资源获取即初始化”)。

1997 《C++ 程序设计语言 (第三版)》出版 [Stroustrup, 1997], 引入了 ISO C++ 标准, 包括命名空间、`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 [C++, 1998]。

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 [C++, 2004]。

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 [C++, 2011]。

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

2013 第一个完整的 C++11 实现出现。

2013 《C++ 程序设计语言 (第四版)》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。

14.1.2 早期的 C++

我最初设计和实现一种新语言的原因是希望能在多处理器间和局域网中（现在称为多核与集群）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（经典 C；见 14.3.1 节）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验。“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++ 这个名字是由 Rick Mascitti 在 1983 年夏天创造的，这个名字取代了我创造的“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。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，作为 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++ 的目的之一是让我和我的朋友们不必再用汇编语言、C 语言以及当时各种流行的语言编写程序。其主要目标是让程序员能更简单、更愉快地编写好程序。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，主导演化的主要是朋友、同事和我之间的讨论。

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。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：“一个构造函数为成员函数创建了执行环境，而析构函数则完成了相反的工作。”这是 C++ 资源管理策略的根源（导致了对异常的需求），也是许多技术使用户代码更简洁、更清晰的关键。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 1.4 节, 4.2.1 节）、`const`（见 1.7 节）、函数重载（见 1.4 节）、引用（见 1.8 节）、运算符重载（见 4.2.1 节）和虚函数（见 4.4 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当时很难习惯（现在很多人仍不习惯）这样一个理念：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，可能与另一个理念也遇到阻力相关：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仔细地人工打造程序的每个细节。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所能表达的边缘。

C++ 的很多设计都是我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。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 [DosReis, 2006] [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 4.2.2 节）等问题。

我推广了 C++ 的继承机制，使之支持多重基类 [Stroustrup, 1987a]。这种机制称为多重继承（multiple inheritance），人们认为它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。

156 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 Shapiro 和我开发的，这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（见第 13 章）。模板机制的发展受到了 vector、map、list 和 sort 等各种模板的影响，这些模板是由我、Andrew Koenig 和 Alex Stepanov 等人设计的。

1998 标准库中最重要的革新是 STL 的引入，这是标准库中一个算法和容器的框架（见第 9 章，第 10 章）。它是 Alex Stepanov（以及 Dave Musser 和 Meng Lee 等人）基于其几十年的泛型编程工作经验设计的。STL 已经在 C++ 社区和更大范围内产生了巨大影响。

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，我掌握了大约 25 种

语言，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。

14.1.3 ISO C++ 标准

C++ 的使用呈爆炸式增长，这导致了一些变化。1987 年，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 [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++ 大会和其他场合下的面对面接触实现的。

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，因此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了草案并提出了意见，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》[Ellis, 1989] 中找到。ANSI 的 X3J16 委员会于 1989 年 12 月筹建，这是由 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 C++ 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 票赞成 0 票反对。此标准的“错误修正版”于 2003 年发布，因此你有时

157

会听人提到 C++03，但它与 C++98 本质上是相同的语言。

当前的 C++ 标准是 C++11，它曾经多年被称为 C++0x，它是 WG21 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，不能发生由于不兼容而导致旧代码不能工作的问题。委员会不可以破坏数十亿行正在使用的 C++ 代码。

C++11 向标准库增加了很多工具和方法并推动了语言特性集合的完善，这都是为了满足

一种综合编程风格——在 C++98 中已被证明很成功的“范型”和风格的综合。C++11 标准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的语言。
- 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 13.1 节）和一组无锁编程特性，这些工作主要是由 Hans Boehm 和 Brian McKnight 等人完成的。在此基础上，我们添加了 thread 库。

14.2 C++11 扩展

在本节中，我列出 C++11 新增的语言特性和标准库组件。

14.2.1 语言特性

查看语言特性列表很容易让人感到困惑。你需要记住的是，语言特性不是单独使用的。特别是，大多数 C++11 新特性如果离开了旧特性提供的框架都毫无意义。

1. 用 { } 列表进行统一、通用的初始化（见 1.5 节，4.2.3 节）。
2. 从初始化器进行类型推断：auto（见 1.5 节）。
3. 防止类型窄化（见 1.5 节）。
4. 有保证的推广的常量表达式：constexpr（见 1.7 节）。
5. 范围 for 语句（见 1.8 节）。
6. 空指针关键字：nullptr（见 1.8 节）。
7. 有作用域的且强类型的 enum：enum class（见 2.5 节）。
8. 编译时断言：static_assert（见 3.4.3 节）。
9. { } 列表到 std::initializer_list 的语言层的映射（见 4.2.3 节）。
10. 右值引用（移动语义使能；见 4.6.2 节）。
11. 以 >>（两个大于号之间没有空格）结束的嵌套模板参数。
12. lambda（见 5.5 节）。
13. 可变参数模板（见 5.6 节）。
14. 类型和模板别名（见 5.7 节）。
15. 万国码（Unicode）字符。
16. long long 整数类型。
17. 对齐控制：alignas 和 alignof。
18. 在声明中将一个表达式的类型作为类型使用的能力：decltype。
19. 裸字符串字面值常量（见 7.3 节）。
20. 推广的 POD（Plain Old Data，简单旧数据）。

158

21. 推广的 union。
 22. 局部类作为模板参数。
 23. 后缀返回类型语法。
 24. 属性语法和两种标准属性: `[[carries_dependency]]` 和 `[[noreturn]]`。
 25. 防止异常传播: `noexcept` 说明符 (见 3.4.1 节)。
 26. 在表达式中检测 `throw` 的可能性: `noexcept` 运算符。
 27. C99 特性: 扩展的整型类型 (即可选的长整数类型的规则); 窄 / 宽字符串的连接; `__STDC_HOSTED__`; `_Pragma(X)`; 可变参数宏和空宏参数。
 28. 名为 `__func__` 的字符串保存当前函数的名字。
 29. `inline` 命名空间。
 30. 委托构造函数。
 31. 类内成员初始化器。
 32. 默认控制: `default` 和 `delete` (见 4.6.5 节)。
 33. 显式转换运算符。
 34. 用户自定义字面值常量。
 35. `template` 实例化的更显式的控制: `extern template`。
 36. 函数模板的默认模板参数。
 37. 继承构造函数。
 38. 覆盖控制: `override` 和 `final` (见 4.5.1 节)。
 39. 更简单、更通用的 SFINAE 规则。
 40. 内存模型 (见 13.1 节)。
 41. 线程局部存储: `thread_local`。
- 有关 C++98 到 C++11 变化的更完整的介绍, 请参阅 [Stroustrup, 2013]。

14.2.2 标准库组件

C++11 以两种形式向标准库添加新内容: 全新组件 (如正则表达式匹配库); 改进 C++98 组件 (如容器的移动语义)。

159

1. 容器的 `initializer_list` 构造函数 (见 4.2.3 节)。
2. 容器的移动语义 (见 4.6.2 节, 9.2 节)。
3. 单向链表: `forward_list` (见 9.6 节)。
4. 哈希容器: `unordered_map`、`unordered_multimap`、`unordered_set` 和 `unordered_multiset` (见 9.6 节, 9.5 节)。
5. 资源管理指针: `unique_ptr`、`shared_ptr` 和 `weak_ptr` (见 11.2.1 节)。
6. 并发支持: `thread` (见 13.2 节)、互斥对象 (见 13.5 节)、锁 (见 13.5 节) 和条件变量 (见 13.6 节)。
7. 高层并发支持: `packaged_thread`、`future`、`promise` 和 `async()` (见 13.7 节)。

8. tuple (见 11.3.3 节)。
9. 正则表达式: regex (见 7.3 节)。
10. 随机数: uniform_int_distribution、normal_distribution、random_engine 等 (见 12.5 节)。
11. 整数类型名, 如 int16_t、uint32_t 和 int_fast64_t。
12. 定长且连续存储的顺序容器: array (见 11.3.1 节)。
13. 拷贝和重抛出异常 (见 13.7.1 节)。
14. 用错误码报告错误: system_error。
15. 容器的 emplace() 操作。
16. constexpr 函数更广泛的应用。
17. noexcept 函数的系统使用。
18. 改进的函数适配器: function 和 bind() (见 11.5 节)。
19. string 到数值的转换。
20. 有作用域的分配器。
21. 类型萃取, 如 is_integral 和 is_base_of (见 11.6.2 节)。
22. 时间工具: duration 和 time_point (见 11.4 节)。
23. 编译时有理数运算: ratio。
24. 结束一个进程: quick_exit。
25. 更多算法, 如 move()、copy_if() 和 is_sorted() (见第 10 章)。
26. 垃圾回收 ABI (见 4.6.4 节)。
27. 底层并发支持: atomic。

14.2.3 已弃用特性

通过弃用一个特性, 标准委员会表达了希望程序员不再使用该特性的愿望。但是, 委员会没有权力立刻删除一个广泛使用的特性——即使该特性可能是冗余的或是危险的。因此, 委员会通过“弃用”这样一个强烈暗示, 提示程序员这个特性在将来的标准中可能消失, 应避免使用。如果程序员继续使用弃用的特性, 编译器可能给出警告。但是, 已弃用的特性仍是标准的一部分, 而且不幸的是, 历史表明, 出于兼容性考虑, 这些特性其实会“永远”保留。

- 如果一个类有析构函数, 为其生成拷贝构造函数和拷贝赋值运算符的特性被弃用了。
- 不再允许将字符串字面值常量赋予一个 char *。如果需要用字符串字面值常量赋值和初始化一个 char *, 应使用 const char * 或 auto。
- C++98 异常说明被弃用了:

```
void f() throw(X,Y); //C++98 异常说明, 现在已被弃用
```

支持异常说明的一些特性如 unexcepted_handler、set_unexcepted()、get_unexcepted() 和 unexcepted() 也被弃用了。应使用 noexcept (见 3.4.1 节)。

- 一些 C++ 标准库函数对象和相关函数被弃用了，其中大多数是与参数绑定相关的。应使用 `lambda`、`bind` 和 `function`（见 11.5 节）。
- `auto_ptr` 被弃用了。应使用 `unique_ptr`（见 11.2.1 节）。
- 存储说明符 `register` 被弃用了。
- `bool` 类型的 `++` 运算符被弃用了。

此外，委员会真正删除了确实无人使用的 `export` 特性，因为该特性太复杂，而且主流编译器提供商都没有支持它。

14.2.4 类型转换

为了支持命名转换（named cast），C 风格的类型转换已被弃用。命名转换包括：

- `static_cast`：对于合理的、行为良好的转换，例如将基类指针转换为派生类指针，使用此方式。
- `reinterpret_cast`：对于真正糟糕、不可移植的转换，例如将 `int` 转换为指针，使用此方式。
- `const_cast`：用来去掉 `const`。

例如：

```
Widget* pw = static_cast<Widget*>(pv);           //pv 是一个 void*，应指向一个 Widget
auto dd = reinterpret_cast<Device_driver*>(0xFF00); //0xFF 应是指向设备驱动程序的指针
char* pc = const_cast<char*>("Casts are inherently dangerous");
```

以 `0x` 开始的字面值常量表示十六进制整数（以 16 为底）。

程序员应该认真考虑在自己的程序中禁用 C 风格类型转换。当必须进行显式类型转换时，组合使用命名转换可以实现 C 风格转换能完成的任何任务。应该优先选择命名转换的原因是，它们在程序中更为显眼（因而更容易发现其中的错误）。

在大多数高层代码中，显式类型转换是可以完全避免的，因此仔细考虑你的设计中的类型转换是否有瑕疵（无论它们是如何表达的）。考虑定义一个函数 `narrow_cast<T>(v)` 来检查值 `v` 是否可以表示为一个类型为 `T` 的对象而不丢失信息（没有类型窄化），如果不行，抛出一个异常。对于类层次结构中的类型转换，优先选择带检查的 `dynamic_cast`（见 4.5.3 节）。

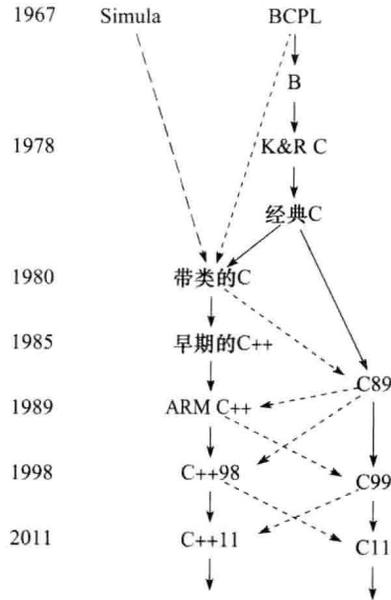
14.3 C/C++ 兼容性

除了少数例外，C++ 可以看作 C（这里指 C11 标准；见 [C11]）的超集。两者的不同大部分源于 C++ 极为强调类型检查。一个编写得很好的 C 程序往往也会是一个合法的 C++ 程序。主流编译器可以诊断出 C++ 和 C 之间的所有不同。ios.C 一节列出了 C99 和 C++ 之间的不兼容之处。在我撰写本书时，C11 标准还非常新，大多数 C 代码还是经典 C 代码或是 C99 代码 [C99]。

14.3.1 C 和 C++ 是兄弟

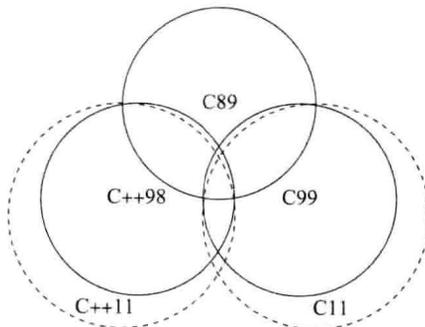
经典 C 有两个主要后代：ISO C 和 ISO C++。多年以来，两种语言以不同的步调，沿着不同的方向发展着。造成的一个结果就是它们都支持传统 C 风格编程，但支持的方式有着细微不同。所产生的不兼容会使某些人非常苦恼，这些人包括：同时使用 C 和 C++ 的人，使用一种语言编写程序但用到另一种语言编写的库的人，以及为 C 和 C++ 编写库和工具的人。

我为何会说 C 和 C++ 是兄弟呢？毕竟 C++ 很明显是 C 的后代。但是，请看下面简化后的家谱：



在此图中，实线表示大量特性的继承，长虚线表示主要特性的借用，而短虚线表示次要特性的借用。从中可以看出，ISO C 和 ISO C++ 是 K&R C [Kernighan, 1978] 的两个主要后代，因此它们是兄弟。两者发展过程中都从经典 C 继承了关键特性，但又都不是 100% 兼容经典 C。“经典 C”一词是我从丹尼斯·里奇的显示器上贴的便条中挑出来的。它大致相当于 K&R C 加上枚举和 struct 赋值两个特性。BCPL 是在 [Richards, 1980] 中定义的，C89 是在 [C90] 中定义的。

不兼容对程序员来说是噩梦，部分原因是造成了选择上的组合爆炸。考虑下面简单的维恩图解：



几个区域是不成比例的。C++11 和 C11 都包含了大部分 K&R C 特性，C++11 又包含了 C11 的大部分特性。但大多数特性都有明确的归属，例如：

C89 独有	调用未声明的函数
C99 独有	可变长度数组 (VLA)
C++ 独有	模板
C89 和 C99 拥有	Algol 风格函数定义
C89 和 C++ 拥有	将 C99 关键字 <code>restrict</code> 作为标识符
C++ 和 C99 拥有	// 注释
C89、C++ 和 C99 拥有	<code>struct</code>
C++11 独有	移动语义 (使用右值引用 <code>&&</code>)
C11 独有	类型通用表达式使用 <code>_Generic</code> 关键字
C++11 和 C11 拥有	原子操作

注意，C 和 C++ 的差别并不一定是 C++ 演化过程中对 C 特性做出改变的结果。有很多不兼容的例子是在将 C++ 中已存在很久的特性引入 C 时产生的。例如，`T*` 到 `void*` 的赋值以及全局 `const` 的连接 [Stroustrup, 2002]。有时一个特性都已经成为了 ISO C++ 标准的一部分，才被引入 C 并产生了不兼容，例如 `inline` 的具体含义。

14.3.2 兼容性问题

C 和 C++ 有很多小的不兼容之处。所有不兼容都能给程序员带来麻烦，但也都可以在 C++ 中解决。如果没有其他不可解决的不兼容问题，C 代码片段可以作为 C 程序编译并使用 `extern "C"` 机制与 C++ 程序链接到一起。

将一个 C 程序转换为 C++ 程序可能遇到的主要问题有：

- 次优的设计和编程风格。
- 将一个 `void*` 隐式转换为一个 `T*` (即没有使用显式类型转换)。
- 在 C 代码中将 C++ 关键字用作标识符。
- 作为 C 程序编译的代码片段和作为 C++ 程序编译的代码片段连接时不兼容。

163

14.3.2.1 风格问题

C 程序自然按 C 风格来编写，例如 K&R 风格 [Kernighan, 1988]。这意味着到处使用指针和数组，可能还有大量宏。用这些特性编写大型程序很难做到可靠。还有，资源管理和错误处理代码通常是特定程序专门编写的，需要文档说明 (而不是语言和工具支持)，而且文档往往不完整，代码的依附性也太强。将 C 程序简单地逐行转换为 C++ 程序时，最好全面检查得到的程序。实际上，我将 C 程序改写为 C++ 程序时从来没有不报错的。这种改写工作，如果不改变基础结构，那么根本的错误来源也就仍然存在。如果原始的 C 程序中就有不完整的错误处理、资源泄漏或是缓存溢出，那么在 C++ 版本中它们还会存在。为了获得大的收益，你必须改变代码的基础结构：

1. 不要将 C++ 看作增加了一些特性的 C。你可以这样使用 C++，但这将导致次优的结果。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. 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()`, 字符串比较用 `==` 而不是 `strcmp()`)。

3. C++ 几乎从不需要宏替换。作为替代, 使用 `const` (见 1.7 节)、`constexpr` (见 1.7 节)、`enum` 或 `enum class` (见 2.5 节) 来定义明示常量, 使用 `inline` (见 4.2.1 节) 来避免函数调用开销, 使用 `template` (见第 5 章) 来指明函数族或类型族, 使用 `namespace` (见 3.3 节) 来避免名字冲突。

4. 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出现的任何位置 (见 1.9 节), 包括 `for` 语句初始化部分 (见 1.8 节) 和条件中 (见 4.5.3 节)。

5. 不要使用 `malloc()`。`new` 运算符 (见 4.2.2 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 4.2.3 节, 10.1 节)。但注意不要简单地用“裸的”`new` 和 `delete` 来代替 `malloc()` 和 `free()` (见 4.2.2 节)。

6. 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。

7. 如果必须使用显式类型转换, 尝试使用命名转换 (如 `static_cast`; 见 14.2.3 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 7.2 节)、`array` (见 11.3.1 节) 和 `vector` (见 9.2 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。实际情况通常正好相反 (当然并不是绝对的)。

164

14.3.2.2 void*

在 C 中, `void*` 可用来为任何指针类型的变量赋值或初始化, 但在 C++ 中则是不行的。例如:

```
void f(int n)
{
    int* p = malloc(n*sizeof(int)); /* 不是 C++ 代码; 在 C++ 中, 用 "new" 分配 */
    // ...
}
```

这可能是最难处理的不兼容问题了。注意, 从 `void*` 到不同指针类型的转换并非总是无害的:

```
char ch;
void* pv = &ch;
int* pi = pv;      // 不是 C++ 改名
*pi = 666;        // 修改了 ch 和临近字节中的数据
```

如果你同时使用两种语言，应将 `malloc()` 的结果转换为正确类型。如果你只使用 C++，应避免使用 `malloc()`。

14.3.2.3 C++ 关键字

C++ 的关键字比 C 多得多。如果在 C 程序中将 C++ 关键字用作了标识符，则改写为 C++ 程序时必须修改这些名字（见下表）：

C++ 而非 C 的关键字					
<code>alignas</code>	<code>alignof</code>	<code>and</code>	<code>and_eq</code>	<code>asm</code>	<code>bitand</code>
<code>bitor</code>	<code>bool</code>	<code>catch</code>	<code>char16_t</code>	<code>char32_t</code>	<code>class</code>
<code>compl</code>	<code>const_cast</code>	<code>constexpr</code>	<code>decltype</code>	<code>delete</code>	<code>dynamic_cast</code>
<code>explicit</code>	<code>false</code>	<code>friend</code>	<code>inline</code>	<code>mutable</code>	<code>namespace</code>
<code>new</code>	<code>noexcept</code>	<code>not</code>	<code>not_eq</code>	<code>nullptr</code>	<code>operator</code>
<code>or_eq</code>	<code>private</code>	<code>protected</code>	<code>public</code>	<code>reinterpret_cast</code>	<code>static_assert</code>
<code>static_cast</code>	<code>template</code>	<code>this</code>	<code>thread_local</code>	<code>throw</code>	<code>true</code>
<code>try</code>	<code>typeid</code>	<code>typename</code>	<code>using</code>	<code>virtual</code>	<code>wchar_t</code>
<code>xor</code>	<code>xor_eq</code>				

此外，单词 `export` 是保留的，未来可能作为关键字。C99 采纳了关键字 `inline`。在 C 中，一些 C++ 关键字是定义在标准头文件中的宏，如下表所示：

作为 C 中宏的 C++ 关键字								
<code>and</code>	<code>and_eq</code>	<code>bitand</code>	<code>bitor</code>	<code>bool</code>	<code>compl</code>	<code>false</code>	<code>not</code>	<code>not_eq</code>
<code>or</code>	<code>or_eq</code>	<code>true</code>	<code>wchar_t</code>	<code>xor</code>	<code>xor_eq</code>			

这说明在 C 中它们能用 `#ifdef` 检查，也能重定义等。

14.3.2.4 连接

C 和 C++ 可以实现为使用不同的连接规范（通常很多实现也确实这么做）。其基本原因是 C++ 极为强调类型检查。还有一个实现上的原因是 C++ 支持重载，因此可能出现两个都叫 `open()` 的全局函数，连接器必须用某种办法解决这个问题。

为了让一个 C++ 函数使用 C 连接规范（从而使它可以被 C 程序片段所调用），或者反过来，让一个 C 函数能被 C++ 程序片段所调用，需要将其声明为 `extern "C"`。例如：

```
extern "C" double sqrt(double);
```

这样，`sqrt(double)` 就可以被 C 或 C++ 代码片段调用，而其定义既可以作为 C 函数编译，也可以作为 C++ 函数编译。

在一个作用域中，对于一个给定的名字，只允许一个具有该名字的函数使用 C 连接规范（因为 C 不允许函数重载）。连接说明不会影响类型检查，因此对一个声明为 `extern "C"` 的函数仍要应用 C++ 函数调用和参数检查规则。

14.4 参考文献

[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 (editor: Andrew Koenig): *International Standard – The C++ Language*. ISO/IEC 14882:1998.
- [C++,2004] ISO/IEC JTC1/SC22/WG21 (editor: Lois Goldtwaite): *Technical Report on C++ Performance*. ISO/IEC TR 18015:2004(E)
- [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 (editor: Pete Pecker): *International Standard – The C++ Language*. ISO/IEC 14882:2011.
- [Cox,2007] Russ Cox: *Regular Expression Matching Can Be Simple And Fast*. January 2007. swtch.com/~rsc/regex/regexpl.html.
- [Dahl,1970] O.-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language*. Norwegian Computing Center S-22. Oslo, Norway. 1970.
- [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.
- [Ellis,1989] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.
- [Friedl,1997]: Jeffrey E. F. Friedl: *Mastering Regular Expressions*. O'Reilly Media. Sebastopol, California. 1997. ISBN 978-1565922570.
- [Gregor,2006] Douglas Gregor et al.: *Concepts: Linguistic Support for Generic Programming in C++*. OOPSLA'06.
- [Ichbiah,1979] Jean D. Ichbiah et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices. Vol. 14, No. 6. June 1979.
- [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,1990] A. R. Koenig and B. Stroustrup: *Exception Handling for C++ (revised)*. Proc USENIX C++ Conference. April 1990.
- [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.
- [Richards,1980] Martin Richards and Colin Whitby-Stevens: *BCPL – The Language and Its Compiler*. Cambridge University Press. Cambridge. 1980. ISBN 0-521-21965-5.
- [Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Technical Report HPL-94-34 (R. 1). 1994.
- [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,2009] B. Stroustrup: *Programming – Principles and Practice Using C++*. Addison-Wesley. 2009. ISBN 0-321-54372-6.
- [Stroustrup,2010] B. Stroustrup: *The C++11 FAQ*. www.stroustrup.com/C++11FAQ.html.
- [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.
- [Stroustrup,2013] B. Stroustrup: *The C++ Programming Language (Fourth Edition)*. Addison-Wesley. 2013. ISBN 0-321-56384-0.
- [Sutton,2011] A. Sutton and B. Stroustrup: *Design of Concept Libraries for C++*. Proc. SLE 2011 (International Conference on Software Language Engineering). July 2011.
- [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.
- [Woodward,1974] P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide*. Her Majesty’s Stationery Office. London. 1974.

167

14.5 建议

- [1] 本章内容在 [Stroustrup, 2013] 的第 1 章和第 44 章中有更加详细的描述。
- [2] ISO C++ 标准 [C++, 2011] 定义了 C++。
- [3] 当学习 C++ 时，不要孤立地关注单个语言特性；参见 14.2.1 节。
- [4] 到目前为止，很多人已经使用 C++ 一二十年了。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++ 程序员多年间忽略的可能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。利用 C++11 新特性所提供的机会来更新你的设计和编程技术，使之更现代化：

168

1. 使用构造函数建立不变式（参见 3.4.2 节）。
 2. 使用构造 / 析构函数对简化资源管理（RAII；参见 4.2.2 节）。
 3. 避免“裸” new 和“裸” delete（参见 4.2.2 节）。
 4. 使用容器和算法而不是内置数组和专用代码（参见第 9 章，第 10 章）。
 5. 优先使用标准库特性而非自己开发的代码（参见第 6 章）。
 6. 使用异常而非错误代码来报告不能局部处理的错误（参见 3.4 节）。
 7. 使用移动语义来避免拷贝大对象（参见 4.6 节）。
 8. 使用 `unique_ptr` 来引用多态类型的对象（参见 11.2.1 节）。
 9. 使用 `shared_ptr` 来引用共享对象，即不止有一个所有者负责其析构的对象（参见 11.2.1 节）。
 10. 使用模板来保持静态类型安全（消除类型转换）并避免类层次结构的不必要使用（参见第 5 章）。
- [5] 在产品级代码中使用新特性之前，先进行试验，编写一些小程序，测试你计划使用的 C++ 实现与标准是否一致，性能是否满足要求。
- [6] 学习 C++ 时，使用你能得到的最新的、最完整的标准 C++ 实现。
- [7] C 和 C++ 的公共子集并非学习 C++ 的最好起点；参见 14.3.2.1 节。
- [8] 优先选择命名类型转换，如 `static_cast`，不要用 C 风格类型转换；参见 14.2.3 节。
- [9] 当你将一个 C 程序改写为 C++ 程序时，首先检查函数声明（原型）和标准头文件的使用是否一致；参见 14.3.2 节。
- [10] 当你将一个 C 程序改写为 C++ 程序时，重新命名与 C++ 关键字同名的标识符；参见 14.3.2.3 节。
- [11] 出于兼容性和类型安全的考虑，如果你必须使用 C，应使用 C 和 C++ 的公共子集编写程序；参见 14.3.2.1 节。
- [12] 当你将一个 C 程序改写为 C++ 程序时，将 `malloc()` 的结果转换为恰当的类型，或者索性将所有 `malloc()` 都改为 `new`；参见 14.3.2.2 节。
- [13] 当你用 `new` 和 `delete` 替换 `malloc()` 和 `free()` 进行内存管理时，考虑使用 `vector`、`push_back()` 和 `reserve()` 而不是 `realloc()`；参见 14.3.2.1 节。
- [14] C++ 不允许 `int` 到枚举类型的隐式类型转换，如果必须进行这种转换，使用显示类型转换。
- [15] 为使用 `std::string`，请包含头文件 `<string>`（`<string.h>` 提供的是 C 风格字符串函数）。
- [16] 每个标准 C 头文件 `<x.h>` 都将名字定义在全局命名空间中，对应的 C++ 头文件 `<cX>` 则将名字定义在命名空间 `std` 中。
- [17] 声明 C 函数时使用 `extern "C"`；参见 14.3.2.4 节。
- [18] 优先使用 `string` 而不是 C 风格字符串（C 风格字符串函数直接处理以 0 结尾的 `char` 数组）。
- [19] 优先使用 `iostream` 而不是 `stdio`。
- [20] 优先使用容器（如 `vector`）而不是内置数组。

索引

知识分为两种：一种是我们自己知道；另一种是我们知道能从哪里找到。

——萨缪尔·约翰逊

索引中的页码为英文原书页码，与书中页边标注的页码一致。

符号

- !=, not-equal operator (不等运算符), 6
- ", string literal (字符串面值常量), 3
- \$. regex (正则表达式的特殊字符), 79
- %
 - modulus operator (模运算符), 6
 - remainder operator (取余运算符), 6
- %=, operator (取模赋值复合运算符), 7
- &
 - address-of operator (取地址运算符), 10
 - reference to (引用), 10
- &&, rvalue reference (右值引用), 51
- (, regex (正则表达式的特殊字符), 79
- () , call operator (调用运算符), 64
- (?pattern (非子模式), 82
-) , regex (正则表达式的特殊字符), 79
- *
 - contents-of operator (取值运算符), 10
 - multiply operator (乘法运算符), 6
 - pointer to (指针), 9
 - regex (正则表达式), 79
- *=, contents-of operator (乘法赋值复合运算符), 7
- *, lazy (正则表达式的零或多次重复懒惰匹配), 80
- +
 - plus operator (加法运算符), 6
 - regex (正则表达式的特殊字符), 79
 - string concatenation (字符串连接), 75
- ++, increment operator (递增运算符), 7
- +=
 - increment operator (加法赋值复合运算符), 7
 - string append (字符串追加), 76
- +? lazy (正则表达式的一或多次重复懒惰匹配), 80
- , minus operator (减法运算符), 6
- , decrement operator (递减运算符), 7
- . , regex (任意字符 (通配符)), 79
- / , divide operator (除法运算符), 6
- // comment (注释), 2
- /= , scaling operator (除法赋值复合运算符), 7
- :public (公有继承), 40
- <<, output operator (输出运算符), 2
- <= , less-than-or-equal operator (小于等于运算符), 6
- < , less-than operator (小于运算符), 6
- =
 - 0 (纯虚函数), 39
 - and == (= 和 ==), 6
 - auto (通过初始值推断变量类型), 7
 - initializer (初始化符号), 6
 - string assignment (字符串赋值), 77
- ==
 - = and (= 和 ==), 6
 - equal operator (相等运算符), 6
 - string (字符串相等比较), 76
- > , greater-than operator (大于运算符), 6
- >= , greater-than-or-equal operator (大于等于运算符), 6
- >>template arguments (模板参数), 159
- ?, regex (正则表达式的特殊字符), 79

- ?? lazy (正则表达式的可选懒惰匹配), 80
 - [, regex (正则表达式的特殊字符), 79
 - []
 - array (array 下标操作), 122
 - array of (内置数组下标操作), 9
 - string (string 下标操作), 76
 - \, backslash (反斜线, 转义符), 3
 -], regex (正则表达式的特殊字符), 79
 - ^, regex (正则表达式的特殊字符), 79
 - _1, placeholders (占位符), 126
 - _2, placeholders (占位符), 126
 - {, regex (正则表达式的特殊字符), 79
 - { }
 - grouping (语句分组), 2
 - initializer (初始化符号), 6
 - { }? lazy (正则表达式的指定次数重复懒惰匹配), 80
 - |, regex (正则表达式的特殊字符), 79
 - }, regex (正则表达式的特殊字符), 79
 - ~, destructor (析构函数), 37
 - 0
 - = (纯虚函数), 39
 - nullptr NULL (空指针), 12
 - 0x hexadecimal literal (十六进制字面值常量), 161
- ## A
- abs() (绝对值函数), 134
 - abstract (抽象)
 - class (抽象类), 40
 - type (抽象类型), 39
 - accumulate() (求和算法), 135
 - acquisition RAIL, resource (资源获取即初始化), 118
 - adaptor, function (函数适配器), 125
 - address-of operator & (取地址运算符), 10
 - adjacent_difference() (相邻元素差算法), 135
 - aims, C++11 (C++11 的目标), 158
 - algorithm (算法), 107
 - container (容器算法), 108, 115
 - numerical (数值算法), 135
 - standard library (标准库算法), 114
 - <algorithm> (算法头文件), 73, 114
 - alias, using (类型别名), 67
 - alignas (对齐控制), 159
 - alignof (对齐控制), 159
 - allocation (内存分配), 37
 - almost container (拟容器), 121
 - alnum, regex (任意字母数字, 正则表达式), 81
 - alpha, regex (任意字母, 正则表达式), 81
 - [[:alpha:]] letter (字母字符集), 81
 - ANSI C++, 157
 - append +=, string (字符串追加), 76
 - argument (参数)
 - passing, function (函数参数传递), 52
 - type (模板类型参数), 61
 - value (模板值参数), 61
 - arithmetic (算术运算)
 - conversions, usual (常用算术类型转换), 6
 - operator (算术运算符), 6
 - vector (向量算术运算), 138
 - ARM (《C++ 参考手册批注版》), 157
 - array (数组)
 - array vs. (array 与内置数组), 123
 - of[] (内置数组下标操作), 9
 - array 122
 - [] (数组下标操作), 122
 - data() (获取起始地址), 122
 - initialize (初始化), 122
 - size() (获取大小), 122
 - vs. array (array 与内置数组), 123
 - vs. vector (array 与 vector), 122
 - <array> (array 头文件), 73
 - asin() (反正弦函数), 134
 - assembler (汇编器), 155
 - assertion static_assert (静态断言), 30
 - assignment (赋值)
 - =, string (字符串赋值), 77
 - copy (拷贝赋值), 49, 52
 - move (移动赋值), 51–52
 - associative array (关联数组), 参见 map
 - async() launch (启动异步任务), 150
 - at() (带范围检查的下标操作), 98
 - atan() (反正切函数), 134
 - atan2() (双参数反正切函数), 134
 - AT&T Bell Laboratories (AT&T 贝尔实验室), 157

auto = (通过初始值推断变量类型), 7
 auto_ptr, deprecated (已弃用特性), 161

B

back_inserter() (插入迭代器), 108
 backslash\ (反斜线, 转义符), 3
 base and derived class (基类和派生类), 40
 basic_string (字符串通用模板), 77
 BCPL (一种编程语言), 162
 begin() (获取容器首位置迭代器), 100, 108
 beginner, book for (入门书籍), 1
 Bell Laboratories, AT&T (AT&T 贝尔实验室), 157
 bibliography (参考文献), 166
 binary search (二分搜索), 114
 bind() (函数对象绑定), 126
 and overloading (绑定重载函数), 126
 binder (绑定器), 125
 bit-field, bitset and (bitset 和位域), 123
 bitset, 123
 and bit-field (bitset 和位域), 123
 and enum (bitset 和枚举), 123
 blank, regex (任意空白符(换行回车除外), 正则表达式), 81
 block (块)
 as function body, try (try 块作为函数体), 99
 try (try 块), 28
 body, function (函数体), 2
 book for beginner (入门书籍), 1
 bool (布尔类型), 5
 break (中断语句), 13

C

C, 155
 and C++ compatibility (C 和 C++ 的兼容性), 161
 Classic (经典 C), 162
 difference from (C++ 和 C 的差异), 161
 K&R (K&R C), 162
 macro, difference from (C++ 中一些关键字在 C 中是宏), 165

 programmer (C 程序员), 168
 void* assignment, difference from (C++ 中 void* 赋值与 C 不同), 165
 with Classes (带类的 C), 154
 with Classes language features (带类特性的 C), 155
 with Classes standard library (带类标准库的 C), 156
 C++
 ANSI (ANSI C++), 157
 compatibility, C and (C 和 C++ 的兼容性), 161
 core language (核心语言特性), 2
 history (C++ 历史), 153
 ISO (C++ ISO 标准), 157
 meaning (C++ 名称的含义), 155
 programmer (C++ 程序员), 168
 pronunciation (C++ 的发音), 155
 standard, ISO (C++ ISO 标准), 2
 standard library (C++ 标准库), 2
 standardization (C++ 的标准化进程), 157
 timeline (C++ 大事年表), 154
 C++03, 157
 C++0x, C++11, 155, 158
 C++11
 aims (目标), 158
 C++0x (也被称为 C++0x), 155, 158
 language features (语言特性), 158
 library components (标准库组件), 159
 C++98, 157
 standard library (标准库), 157
 C11, 161
 C89 and C99 (C89 和 C99), 161
 C99, C89 and (C89 和 C99), 161
 call operator () (调用运算符), 64
 callback (回调), 128
 capacity() (获取容器容量), 97
 capture list (捕获列表), 65
 carries_dependency (属性, 允许捕获函数调用间的依赖关系), 159
 cast (显式类型转换), 39
 deprecated C-style (C 风格显式类型转换已被弃用), 161
 named (命名转换), 161

- catch
 - clause (catch 子句), 28
 - every exception (捕获所有异常), 99
- catch(...) (捕获所有异常), 99
- ceil() (向上取整), 134
- char (字符类型), 5
- character sets, multiple (多字符集), 77
- chrono (处理时间工具的命名空间), 125
- <chrono> (时间处理头文件), 73, 125, 146
- class (类), 34
 - concrete (具体类), 34
 - scope (类作用域), 8
 - template (类模板), 59
- class
 - abstract (抽象类), 40
 - base and derived (基类和派生类), 40
 - hierarchy (类层次), 42
- Classic C (经典 C), 162
- C-library header (C 标准库头文件), 73
- clock timing (时钟, 计时用), 146
- <cmath> (数学函数头文件), 73, 134
- cntrl, regex (任意控制字符, 正则表达式), 81
- code complexity, function and (函数和代码的复杂度), 4
- comment, // (注释), 2
- communication, task (任务通信), 147
- comparison operator (比较运算符), 6
- compatibility, C and C++ (C 和 C++ 的兼容性), 161
- compilation (编译)
 - model, template (模板编译模型), 68
 - separate (分别编译), 24
- compiler (编译器), 2
- compile-time (编译时)
 - computation (编译时计算), 128
 - evaluation (编译时求值), 9
- complete encapsulation (完整封装), 52
- complex (复数类型), 35, 135
- <complex> (复数头文件), 73, 134-135
- complexity, function and code (函数和代码的复杂度), 4
- components, C++11 library (C++11 标准库组件), 159
- computation, compile-time (编译时计算), 128
- concatenation +, string (字符串连接运算), 75
- concept (概念), 63
- concrete (具体的)
 - class (具体类), 34
 - type (具体类型), 34
- concurrency (并发), 141
- condition, declaration in (在条件中声明), 47
- condition_variable (条件变量), 146
 - notify_one() (解锁一个等待条件的线程), 147
 - wait() (等待条件), 146
- <condition_variable> (条件变量头文件), 146
- const, immutability (常量的不变性), 8
- constant expression (常量表达式), 9
- const_cast (常量转换), 161
- constexpr
 - function (constexpr 函数), 9
 - immutability (constexpr 的不可变性), 8
- const_iterator (常量迭代器), 112
- constructor (构造函数)
 - and destructor (构造函数和析构函数), 155
 - copy (拷贝构造函数), 49, 52
 - default (默认构造函数), 35
 - delegating (委托构造函数), 159
 - explicit (显式构造函数), 53
 - inheriting (继承构造函数), 159
 - initializer-list (构造函数初始值列表), 38
 - invariant and (不变式和构造函数), 29
 - move (移动构造函数), 51-52
- container (容器), 36, 59, 95
 - algorithm (容器算法), 108, 115
 - almost (拟容器), 121
 - object in (容器中的对象), 98
 - overview (容器概览), 103
 - return (返回容器), 109
 - sort() (容器排序), 129
 - specialized (特殊容器), 121
 - standard library (标准库容器), 103
- contents-of operator * (取值运算符), 10
- conversion (转换)
 - explicit type (显式类型转换), 39, 161
 - narrowing (窄化转换), 7
- conversions, usual arithmetic (常用算术类型转

- 换), 6
 - copy (拷贝), 48
 - and hierarchy (拷贝和类层次), 55
 - assignment (拷贝赋值), 49, 52
 - constructor (拷贝构造函数), 49, 52
 - cost of (拷贝的代价), 50
 - memberwise (逐成员拷贝), 52
 - copy() (拷贝算法), 114
 - copyif() (条件拷贝算法), 114
 - core language, C++ (C++ 核心语言特性), 2
 - cos() (余弦函数), 134
 - cosh() (双曲余弦函数), 134
 - cost of copy (拷贝的代价), 50
 - count() (计数算法), 114
 - count_if() (条件计数算法), 113-114
 - cout, output (cout 输出流), 2
 - <cstdlib> (<stdlib.h> 的 C++ 版本), 73
 - C-style (C 风格)
 - cast, deprecated (C 风格显式类型转换, 已弃用), 161
 - error handling (C 风格错误处理), 134
 - string (C 风格字符串), 12
 - Currying (柯里化), 125
- ## D
- \d, regex (一个十进制数字, 正则表达式), 81
 - \D, regex (非十进制数字, 正则表达式), 81
 - d, regex (任意十进制数字, 正则表达式), 81
 - data race (数据竞争), 142
 - data(), array (获取起始地址), 122
 - D&E (《C++ 语言的设计和演化》), 154
 - deadlock (死锁), 145
 - deallocation (释放), 37
 - declaration (声明), 5
 - function (函数声明), 3
 - in condition (在条件中声明), 47
 - interface (接口声明), 23
 - declarator operator (声明运算符), 11
 - decltype (获取实体或表达式类型), 159
 - decrement operator-- (递减运算符), 7
 - default (默认)
 - constructor (默认构造函数), 35
 - operations (默认操作), 52
 - =default (默认拷贝 / 移动控制成员), 53
 - definition implementation (定义实现), 24
 - delegating constructor (委托构造函数), 159
 - =delete (禁止拷贝 / 移动控制成员), 55
 - delete
 - an operation (禁止一个操作), 55
 - naked (裸 delete), 38
 - operator (内存释放运算符), 37
 - deprecated (已弃用)
 - auto_ptr, 161
 - C-style cast (C 风格显式类型转换), 161
 - exception specification (异常说明), 161
 - feature (已弃用特性), 160
 - deque (双端队列), 103
 - derived class, base and (基类和派生类), 40
 - destructor (析构函数), 37, 52
 - ~ (析构函数名字前缀), 37
 - constructor and (构造函数和析构函数), 155
 - virtual (虚析构函数), 44
 - dictionary (字典), 参见 map
 - difference (差异)
 - from C (C++ 与 C 的差异), 161
 - from C macro (C++ 的一些关键字在 C 中是宏), 165
 - from C void* assignment (C++ 中 void* 赋值与 C 不同), 165
 - digit, [[:digit:]] (十进制数字字符集), 81
 - digit, regex (任意十进制数字, 正则表达式), 81
 - [[:digit:]]digit (十进制数字字符集), 81
 - dispatch, tag (标签分发), 129
 - distribution, random (随机分布), 136
 - divide operator / (除法运算符), 6
 - domain error (定义域错误), 134
 - double (双精度类型), 5
 - duck typing (鸭子类型), 68
 - duration (时间段), 125
 - duration_cast (时间段转换), 125
 - dynamic store (动态存储), 37
 - dynamic_cast (动态类型转换), 47
 - is instance of (是……的一个实例), 47
 - is kind of (是……的一种), 47

E

EDOM (定义域错误), 134
 element requirements (需要的元素数), 98
 encapsulation, complete (完整封装), 52
 end() (获取容器尾后迭代器), 100, 108
 engine, random (随机数引擎), 136
 enum, bitset and (bitset 和枚举), 123
 equal operator == (相等运算符), 6
 equal_range() (相等子序列算法), 114, 124
 ERANGE (值域错误), 134
 erase() (删除元素), 101
 errno (错误代码), 134
 error (错误)
 domain (定义域错误), 134
 handling (错误处理), 27
 handling, C-style (C 风格错误处理), 134
 range (值域错误), 134
 run-time (运行时错误), 27
 essential operation (基本操作), 52
 evaluation
 compile-time (编译时求值), 9
 partial (偏函数评价), 125
 example (例程)
 find_all() (查找所有出现位置), 109
 Hello, World!, 2
 Rand_int (随机整数), 137
 Vec (向量), 98
 exception (异常), 27
 and main() (异常和主函数), 99
 catch every (捕获所有异常), 99
 specification, deprecated (异常说明, 已弃用), 161
 explicit type conversion (显式类型转换), 39, 161
 explicit constructor (显式构造函数), 53
 exponential_distribution (指数分布), 136
 export removed (export, 已删除特性), 161
 exp() (指数函数), 134
 expression (表达式)
 constant (常量表达式), 9
 lambda (lambda 表达式), 65
 extern template (显式控制模板实例化), 159

F

fabs() (浮点绝对值函数), 134
 facility, standard library (标准库组件), 72
 feature, deprecated (已弃用特性), 160
 feature (特性)
 C with Classes language (带类特性的 C), 155
 C++11 language (C++11 语言特性), 158
 file, header (头文件), 25
 final (覆盖控制), 159
 find() (查找算法), 108, 114
 find_all() example(查找所有出现位置例程), 109
 find_if() (条件查找算法), 113-114
 first, pair member (pair 的 first 成员), 124
 floor() (向下取整函数), 134
 fmod() (浮点数模函数), 134
 for
 statement (for 语句), 10
 statement, range (范围 for 语句), 10
 forward_list (单向链表), 103
 <forward_list> (单向链表头文件), 73
 free store (自由存储区), 37
 frexp() (浮点数二进制分解函数), 134
 <fstream> (文件流头文件), 73
 __func__ (当前函数的名字), 159
 function (函数), 2
 adaptor (函数适配器), 125
 and code complexity (函数和代码的复杂度), 4
 argument passing (函数参数传递), 52
 body (函数体), 2
 body, try block as (try 块作为函数体), 99
 constexpr (constexpr 函数), 9
 declaration (函数声明), 3
 implementation of virtual (虚函数实现), 42
 mathematical (数学函数), 134
 object (函数对象), 64
 overloading (函数重载), 4
 template (函数模板), 62
 type (函数类型), 128
 value return (以传值方式返回结果), 52

function (标准库类型), 127
 and nullptr (function 类型和 nullptr), 127
fundamental type (基本类型), 5
future
 and promise (future 和 promise 用于任务通信), 147
 member get() (成员函数 get() 用来获取值), 147
<future> (任务通信头文件), 73, 147

G

garbage collection (垃圾回收), 54
generic programming (泛型编程), 62
get<>() (获取 tuple 成员), 125
get(), future member (future 成员, 获取传输的值), 147
graph, regex (任意图形字符, 正则表达式), 81
greater-than operator > (大于运算符), 6
greater-than-or-equal operator >= (大于等于运算符), 6
greedy match (贪心匹配), 80, 83
grouping, {} (代码分组), 2

H

half-open sequence (半开序列), 114
handle (句柄), 38
 resource (资源句柄), 49, 119
hash table (哈希表), 102
header (头文件)
 C-library (C 标准库头文件), 73
 file (头文件), 25
 standard library (标准库头文件), 73
heap (堆), 37
Hello, World! example (Hello, World! 例程), 2
hexadecimal literal, 0x (十六进制面值常量), 161
hierarchy (层次结构)
 class (类层次结构), 42
 copy and (拷贝和层次结构), 55
 navigation (层次结构漫游), 47
history, C++ (C++ 历史), 153

HOPL (ACM 程序设计语言历史大会), 154

I

if statement (if 语句), 12
immutability (不变性)
 const (const 不变性), 8
 constexpr (constexpr 不变性), 8
implementation (实现)
 definition (定义实现), 24
 inheritance (实现继承), 46
 iterator (迭代器实现), 111
 of virtual function (实现虚函数), 42
 string (string 实现), 77
in-class member initialization (类内成员初始化), 159
#include (包含头文件语句), 25
increment operator ++ (递增运算符), 7
inheritance (继承), 40
 implementation (实现继承), 46
 interface (接口继承), 46
 multiple (多重继承), 156
inheriting constructor (继承的构造函数), 159
initialization, in-class member (类内成员初始化), 159
initialize (初始化), 38
 array (array 初始化), 122
initializer (初始化符号)
 = (初始化符号), 6
 {} (初始值列表), 6
initializer-list constructor (初始值列表构造函数), 38
initializer_list (初始值列表类型), 38
inline (内联关键字), 35
 namespace (内联命名空间), 159
inlining (内联), 35
inner_product() (内积), 135
insert() (插入元素操作), 101
int (整型), 5
 output bits of (输出整型数的二进制表示), 123
interface (接口)
 declaration (接口声明), 23
 inheritance (接口继承), 46

- invariant (不变式), 29
 - and constructor (不变式和构造函数), 29
 - I/O, iterator and (迭代器和 I/O), 112
 - <ios> (I/O 流头文件), 73
 - <iostream> (I/O 流头文件), 2, 73
 - iota() (递增赋值算法), 135
 - is (是)
 - instance of, dynamic_cast (动态类型转换, 是……的一个实例), 47
 - kind of, dynamic_cast (动态类型转换, 是……的一种), 47
 - ISO (国际标准化组织)
 - C++, 157
 - C++ standard (ISO C++ 标准), 2
 - ISO-14882 (第一个 ISO C++ 标准), 157
 - istream_iterator (输入流迭代器), 112
 - iterator (迭代器), 108
 - and I/O (迭代器和 I/O), 112
 - implementation (迭代器实现), 111
 - iterator (迭代器类型), 100, 112
 - <iterator> (迭代器头文件), 130
 - iterator_category (迭代器类别), 129
 - iterator_traits (迭代器类型萃取), 128, 130
 - iterator_type (返回迭代器的类型), 129
- J**
- join(), thread (等待线程结束), 142
- K**
- key and value (关键字和值), 101
 - K&R C, 162
- L**
- \L, regex (非小写字符, 正则表达式), 81
 - \l, regex (小写字符, 正则表达式), 81
 - lambda expression (lambda 表达式), 65
 - language (语言)
 - and library (语言和库), 71
 - features, C with Classes (带类的 C 的特性), 155
 - features, C++11 (C++11 语言特性), 158
 - launch, async() (启动异步任务), 150
 - lazy (懒惰)
 - *? (正则表达式的零或多次重复懒惰匹配), 80
 - +? (正则表达式的一或多次重复懒惰匹配), 80
 - ?? (正则表达式的可选懒惰匹配), 80
 - ()? (正则表达式的指定重复次数懒惰匹配), 80
 - match (懒惰匹配), 80, 83
 - ldexp() (指数乘法函数), 134
 - leak, resource (资源泄漏), 47, 54, 118
 - less-than operator < (小于运算符), 6
 - less-than-or-equal operator <= (小于等于运算符), 6
 - letter, [[:alpha:]] (字母字符集), 81
 - library (库)
 - algorithm, standard (标准库算法), 114
 - C with Classes standard (带类标准库的 C), 156
 - C++98 standard (C++98 标准库), 157
 - components, C++11 (C++11 库组件), 159
 - container, standard (标准库容器), 103
 - facilities, standard (标准库组件), 72
 - language and (语言和库), 71
 - non-standard (非标准库), 71
 - standard (标准库), 71
 - lifetime, scope and (作用域和生命周期), 8
 - <limits> (数值类型信息头文件), 128, 138
 - linker (连接器), 2
 - list, capture (捕获列表), 65
 - list (链表容器), 100, 103
 - literal (字面值常量)
 - ", string (字符串字面值常量), 3
 - 0x hexadecimal (十六进制字面值常量), 161
 - raw string (裸字符串字面值常量), 78
 - user-defined (用户自定义字面值常量), 159
 - local scope (局部作用域), 8
 - lock() (加锁操作), 145
 - and RAII (加锁与资源获取即初始化), 145
 - log() (自然对数函数), 134
 - log10() (以 10 为底的对数函数), 134
 - long long (C++11 新整型类型), 159

lower, regex (任意小写字母, 正则表达式), 81

M

macro, difference from C (C++ 中一些关键字在 C 中是宏), 165

main() (主函数), 2

exception and (异常与主函数), 99

make_pair() (创建 pair), 124

make_shared() (创建共享指针), 120

make_tuple() (创建 tuple), 125

make_unique() (创建独占指针), 120

management, resource (资源管理), 54, 117

map (字典), 101, 103

<map> (字典头文件), 73

mapped type, value (值或映射类型), 101

match (匹配)

greedy (贪心匹配), 80, 83

lazy (懒惰匹配), 80, 83

mathematical (数学)

function (数学函数), 134

function, standard (标准数学函数), 134

<math.h> (数学头文件), 134

meaning, C++ (C++ 名字的含义), 155

member initialization, in-class (类内成员初始化), 159

memberwise copy (逐成员拷贝), 52

mem_fn() (构造非成员函数对象), 126

<memory> (内存头文件), 73, 118, 120

merge() (合并算法), 114

minus operator - (减法运算符), 6

model, template compilation (模板编译模型), 68

modf() (浮点取模运算), 134

modularity (模块化), 23

modulus operator % (模运算符), 6

move (移动), 51

assignment (移动赋值), 51-52

constructor (移动构造函数), 51-52

move() (返回右值引用), 52, 114

multimap (重复关键字字典), 103

multiple (多重)

character sets (多字符集), 77

inheritance (多重继承), 156

multiply operator * (乘法运算符), 6

multiset (重复关键字集合), 103

mutex (互斥对象), 144

<mutex> (互斥对象头文件), 144

N

\n (回车符), 3

naked (裸)

delete (裸 delete), 38

new (裸 new), 38

named cast (命名转换), 161

namespace scope (命名空间作用域), 8

namespace (命名空间), 26

inline (内联命名空间), 159

placeholders (命名空间 placeholders), 126

std (命名空间 std), 72

narrowing (窄化), 161

conversion (窄化转换), 7

navigation, hierarchy (类层次导航), 47

new

naked (裸 new), 38

operator (内存分配运算符), 37

noexcept (不抛出异常), 28

noexcept() (运算符), 159

non-standard library (非标准库), 71

noreturn (属性, 函数不返回), 159

normal_distribution (正态分布), 136

notation, regular expression (正则表达式符号), 79

not-equal operator != (不等运算符), 6

notify_one(), condition_variable, 147

NULL 0, nullptr (空指针), 12

nullptr (空指针), 11

function and (函数和空指针), 127

NULL 0 (空指针), 12

number, random (随机数), 136

<numeric> (数值算法头文件), 135

numerical algorithm (数值算法), 135

numeric_limits (数值范围), 138

O

object (对象), 5

- function (函数对象), 64
 - in container (容器中的对象), 98
 - object-oriented programming (面向对象程序设计), 42
 - operation, delete an (禁止一个操作), 55
 - operation (操作)
 - default (默认操作), 52
 - essential (基本操作), 52
 - operator (运算符)
 - %= (取模赋值复合运算符), 7
 - += (加法赋值复合运算符), 7
 - &, address-of (取地址运算符), 10
 - () , call (调用运算符), 64
 - *, contents-of (取值运算符), 10
 - , decrement (递减运算符), 7
 - /, divide (除法运算符), 6
 - ==, equal (相等运算符), 6
 - >, greater-than (大于运算符), 6
 - >=, greater-than-or-equal (大于等于运算符), 6
 - ++, increment (递增运算符), 7
 - <, less-than (小于运算符), 6
 - <=, less-than-or-equal (小于等于运算符), 6
 - , minus (减法运算符), 6
 - %, modulus (模运算符), 6
 - *, multiply (乘法运算符), 6
 - !=, not-equal (不等运算符), 6
 - <<, output (输出运算符), 2
 - +, plus (加法运算符), 6
 - %, remainder (取余运算符), 6
 - *=, scaling (乘法赋值复合运算符), 7
 - /=, scaling (除法赋值复合运算符), 7
 - arithmetic (算术运算符), 6
 - comparison (比较运算符), 6
 - declarator (声明运算符), 11
 - delete (内存释放运算符), 37
 - new (内存分配运算符), 37
 - overloaded (运算符重载), 36
 - user-defined (用户自定义运算符), 36
 - optimization, short-string (短字符串优化), 77
 - ostream_iterator (输出流迭代器), 112
 - out_of_range (越界异常), 98
 - output (输出)
 - bits of int (输出整型数的二进制表示), 123
 - cout (输出流), 2
 - operator << (输出运算符), 2
 - overloaded operator (重载的运算符), 36
 - overloading (重载)
 - bind() and (bind() 和重载), 126
 - function (重载函数), 4
 - override (覆盖), 40
 - override (关键字, 指出覆盖函数), 45
 - overview, container (容器概览), 103
 - ownership (所有权), 118
- P**
- packaged_task thread (打包任务), 149
 - pair (值对类型), 124
 - member first (first 成员), 124
 - member second (second 成员), 124
 - parameterized type (参数化类型), 59
 - partial evaluation (偏函数评价), 125
 - partial_sum() (前缀和算法), 135
 - passing data to task (向任务传递数据), 143
 - pattern, (? ((? 模式), 82
 - phone_book example (电话簿例程), 96
 - placeholders
 - _1 (占位符), 126
 - _2 (占位符), 126
 - namespace (命名空间 placeholders), 126
 - plus operator + (加法运算符), 6
 - pointer (指针)
 - smart (智能指针), 118
 - to * (指针类型), 9
 - polymorphic type (多态类型), 40
 - pow() (幂函数), 134
 - precondition (前置条件), 29
 - predicate (谓词), 64, 113
 - type (类型谓词), 130
 - print, regex (任意可打印字符, 正则表达式), 81
 - program (程序), 2
 - programmer (程序员)
 - C++ (C++ 程序员), 168
 - C (C 程序员), 168
 - programming (程序设计)
 - generic (泛型编程), 62
 - object-oriented (面向对象程序设计), 42

promise
 future and (future 和 promise 用于任务通信), 147
 member set_exception() (set_exception() 成员, 传递异常), 147
 member set_value() (set_value() 成员, 发送值), 147
 pronunciation, C++ (C++ 的读音), 155
 punct, regex (任意标点, 正则表达式), 81
 pure virtual (纯虚函数), 39
 purpose, template (模板的目的), 62
 push_back() (添加到队尾), 38, 97, 101
 push_front() (添加到队首), 101

R

R" (裸字符串), 78
 race, data (数据竞争), 142
 RAII (资源获取即初始化)
 lock() and (加锁和 RAII), 145
 resource acquisition (资源获取), 118
 RAII, 38
 Rand_int example (随机整数例程), 137
 random number (随机数), 136
 random (随机)
 distribution (随机分布), 136
 engine (随机数引擎), 136
 <random> (随机数头文件), 73, 136
 range (范围)
 checking Vec (Vec 的范围检查), 98
 error (范围错误), 134
 for statement (范围 for 语句), 10
 raw string literal (裸字符串字面值常量), 78
 reference (引用)
 &&, rvalue (右值引用), 51
 rvalue (右值), 52
 to & (引用类型), 10
 regex (正则表达式)
 | (或), 79
 § (行结束), 79
 ((分组开始), 79
) (分组结束), 79
 * (零或多次重复), 79
 + (一或多次重复), 79

. (任意单个字符 (通配符)), 79
 ? (可选), 79
 ^ (行开始; 非), 79
 [(字符集开始), 79
] (字符集结束), 79
 } (指定重复次数结束), 79
 { (指定重复次数开始), 79
 alnum (任意字母数字), 81
 alpha (任意字母), 81
 blank (任意空白符 (换行回车除外)), 81
 cntrl (任意控制字符), 81
 \d (一个十进制数字), 81
 d (任意十进制数字), 81
 \D (非十进制数字), 81
 digit (任意十进制数字), 81
 graph (任意图形字符), 81
 \l (小写字符), 81
 \L (非小写字符), 81
 lower (任意小写字符), 81
 print (任意可打印字符), 81
 punct (任意标点), 81
 regular expression (正则表达式), 78
 repetition (重复), 80
 \s (非空白符), 81
 \s (一个空白符), 81
 s (任意空白符), 81
 space (任意空白符), 81
 \U (非大写字符), 81
 \u (大写字符), 81
 upper (任意大写字符), 81
 w (任意字母、数字或下划线), 81
 \w (一个字母、数字或下划线), 81
 \W (非字母、数字或下划线), 81
 xdigit (任意十六进制数字), 81
 <regex> (正则表达式头文件), 73, 78
 regular expression (正则表达式), 78
 regex_iterator (正则匹配迭代器), 83
 regex_search (搜索匹配字符串), 78
 regular (正则)
 expression notation (正则表达式符号), 79
 expression <regex> (正则表达式头文件), 78
 expression regex (正则表达式类), 78
 reinterpret_cast (不可移植的类型转换), 161

- remainder operator % (余数运算符), 6
 - removed, export (export, 已删除特性), 161
 - repetition, regex (正则表达式, 重复), 80
 - replace() (替换算法), 114
 - string (字符串替换), 76
 - replace_if() (条件替换算法), 114
 - requirement, template (模板对参数的要求), 63
 - requirements, element (容器对元素的要求), 98
 - reserve() (容器预留空间), 97
 - resource (资源)
 - acquisition RAII (资源获取即初始化), 118
 - handle (资源管理), 49, 119
 - leak (资源泄漏), 47, 54, 118
 - management (资源管理), 54, 117
 - safety (资源安全), 54
 - rethrow (重抛出), 30
 - return (返回)
 - function value (函数返回值), 52
 - type, suffix (尾置返回类型), 159
 - return (返回语句)
 - container (返回容器), 109
 - type, void (void 返回类型), 3
 - returning results from task (从任务返回结果), 144
 - run-time error (运行时任务), 27
 - rvalue (右值)
 - reference (右值引用), 52
 - reference && (右值引用符号), 51
- S**
- \s, regex (一个空白符, 正则表达式), 81
 - s, regex (任意空白符, 正则表达式), 81
 - \S, regex (非空白符, 正则表达式), 81
 - safety, resource (资源安全), 54
 - scaling (缩放运算符)
 - operator *= (乘法赋值复合运算符), 7
 - operator /= (除法赋值复合运算符), 7
 - scope (作用域)
 - and lifetime (作用域和生命周期), 8
 - class (类作用域), 8
 - local (局部作用域), 8
 - namespace (命名空间作用域), 8
 - search, binary (二分搜索), 114
 - second, pair member (pair 成员, 第二个值), 124
 - separate compilation (分离编译), 24
 - sequence (序列), 108
 - half-open (半开序列), 114
 - set (集合容器), 103
 - <set> (集合头文件), 73
 - set_exception(), promise member (promise 成员, 传递异常), 147
 - set_value(), promise member (promise 成员, 传递值), 147
 - shared_ptr (共享指针), 118
 - sharing data task (任务共享数据), 144
 - short-string optimization (短字符串优化), 77
 - Simula (面向对象语言 Simula), 153
 - sin() (正弦函数), 134
 - sinh() (双曲正弦函数), 134
 - size of type (类型的大小), 5
 - size(), array (获取 array 的元素数), 122
 - sizeof (类型大小运算符), 5
 - sizeof() (类型大小函数), 128
 - size_t (保存大小的类型), 67
 - smart pointer (智能指针), 118
 - sort() (排序算法), 107, 114
 - container (容器排序算法), 129
 - space, regex (任意空白符, 正则表达式), 81
 - specialized container (特殊容器), 121
 - sqrt() (平方根函数), 134
 - <sstream> (字符串流头文件), 73
 - standard (标准)
 - ISO C++ (ISO C++ 标准), 2
 - library (标准库), 71
 - library algorithm (标准库算法), 114
 - library, C++ (C++ 标准库), 2
 - library, C with Classes (带类标准库的 C), 156
 - library, C++98 (C++98 标准库), 157
 - library container (标准库容器), 103
 - library facilities (标准库组件), 72
 - library header (标准库头文件), 73
 - library std (标准库命名空间 std), 72
 - mathematical functions (标准数学函数), 134
 - standardization, C++ (C++ 标准化), 157
 - statement (语句)

for (for 循环语句), 10
 if (if 条件分支语句), 12
 range for (范围 for 循环语句), 10
 switch (switch 多分支语句), 13
 while (while 循环语句), 12
 static_assert (静态断言), 138
 assertion (断言), 30
 static_cast (静态类型转换), 39, 161
 std (标准库命名空间), 2
 namespace (std 命名空间), 72
 standard library (标准库命名空间), 72
 <stdexcept> (异常头文件), 73
 STL (标准模板库), 157
 store (存储)
 dynamic (动态存储), 37
 free (自由存储区), 37
 string (字符串)
 C-style (C 风格字符串), 12
 literal " (字符串字面值常量), 3
 literal, raw (裸字符串字面值常量), 78
 Unicode (Unicode 字符串), 77
 string (字符串), 75
 [] (下标操作, 获取字符), 76
 == (字符串相等比较), 76
 append += (字符串追加操作), 76
 assignment = (字符串赋值运算符), 77
 concatenation + (字符串连接运算符), 75
 implementation (字符串实现), 77
 replace() (子串替换成员函数), 76
 substr() (获取子串成员函数), 76
 <string> (字符串头文件), 73, 75
 subclass, superclass and (超类和子类), 40
 substr(), string (获取子串成员函数), 76
 suffix return type (后缀返回类型), 159
 superclass and subclass (超类和子类), 40
 switch statement (switch 多分支语句), 13

T

table, hash (哈希表), 102
 tag dispatch (标签分发), 129
 tanh() (双曲正切函数), 134
 task (任务)
 and thread (任务和线程), 142

communication (任务通信), 147
 passing data to (向任务传递数据), 143
 returning results from (从任务返回结果), 144
 sharing data (任务共享数据), 144
 TC++PL (《C++ 程序设计语言》), 154
 template (模板)
 arguments, >> (模板参数), 159
 compilation model (模板编译模型), 68
 variadic (可变参数模板), 66
 template (模板关键字), 59
 class (类模板), 59
 extern (显式控制模板实例化), 159
 function (函数模板), 62
 purpose (模板的目的), 62
 requirement (模板对参数的要求), 63
 thread (线程类)
 join() (等待线程结束), 142
 packaged_task (打包任务), 149
 task and (任务和线程), 142
 <thread> (线程头文件), 73, 142
 thread_local (线程局部存储), 159
 time (标准库处理时间组件), 125
 timeline, C++ (C++ 大事年表), 154
 time_point (时间点类型), 125
 timing, clock (时钟, 计时用), 146
 try
 block (try 块), 28
 block as function body (try 块作为函数体), 99
 tuple (多值类型), 125
 type (类型), 5
 abstract (抽象类型), 39
 argument (模板类型参数), 61
 concrete (具体类型), 34
 conversion, explicit (显式类型转换), 39, 161
 function (类型函数), 128
 fundamental (基本类型), 5
 parameterized (参数化类型), 59
 polymorphic (多态类型), 40
 predicate (类型谓词), 130
 size of (类型大小), 5
 typename (模板类型参数), 59, 110
 <type_traits> (类型萃取), 130
 typing, duck (鸭子类型), 68

U

`\u, regex` (一个大写字符, 正则表达式), 81
`\U regex` (非大写字符, 正则表达式), 81
 Unicode string (Unicode 字符串), 77
`uniform_int_distribution` (整数均匀分布), 136
`uninitialized` (不初始化), 7
`unique_copy()` (去重拷贝), 107, 114
`unique_lock` (互斥锁), 144, 146
`unique_ptr` (独占指针), 47, 118
`unordered_map` (无序字典), 102–103
`<unordered_map>` (无序字典头文件), 73
`unordered_multimap` (重复关键字无序字典), 103
`unordered_multiset` (重复关键字无序集合), 103
`unordered_set` (无序集合), 103
`unsigned` (无符号), 5
`upper, regex` (任意大写字符, 正则表达式), 81
 user-defined (用户自定义)
 literal (用户自定义字面值常量), 159
 operator (用户自定义运算符), 36
`using alias` (类型别名), 67
 usual arithmetic conversions (常用算术类型转换), 6
`<utility>` (工具头文件), 73, 124–125

V

`valarray` (数值计算向量类型), 138
`<valarray>` (向量类型头文件), 138
`value` (值), 5
 argument (类型参数和值参数), 61
 key and (关键字和值), 101
 mapped type (映射类型), 101
 return, function (函数返回值), 52
`value_type` (值类型), 67
`variable` (变量), 5
`variadic template` (可变参数模板), 66

Vec

`example` (Vec 例程), 98
`range checking` (Vec 的范围检查), 98
`vector arithmetic` (向量算术运算), 138
`vector` (向量容器), 96, 103
 array vs. (array 容器和 vector 容器), 122
`<vector>` (向量头文件), 73
`vector<bool>` (位序列), 121
`virtual` (虚), 39
 destructor (虚析构函数), 44
 function, implementation of (虚函数的实现), 42
 function table vtbl (虚函数表), 42
 pure (纯虚函数), 39
`void` (无类型)
 * (无类型指针), 165
 * assignment, difference from C (C++ 中 void* 赋值与 C 不同), 165
 return type (无返回值), 3
`vtbl, virtual function table` (虚函数表), 42

W

`w, regex` (任意字母、数字或下划线, 正则表达式), 81
`\w, regex` (一个字母、数字或下划线, 正则表达式), 81
`\W, regex` (非字母、数字或下划线, 正则表达式), 81
`wait(), condition_variable` (等待条件变量), 146
 WG21 (ISO C++ 标准化工作的一部分), 154
`while statement` (while 循环语句), 12

X

X3J16 (C++ 标准化委员会), 157
`xdigit, regex` (任意十六进制数字字符, 正则表达式), 81