



计 算 机 科 学 丛 书

Pearson

原书第2版

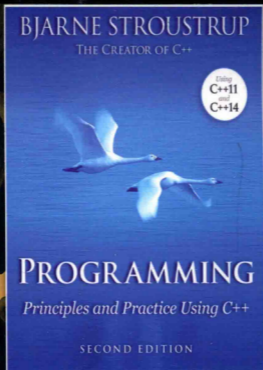
# C++程序设计

## 原理与实践

### (进阶篇)

[美] 本贾尼·斯特劳斯特鲁普 (Bjarne Stroustrup) 著 刘晓光 李忠伟 王刚 译

**Programming**  
Principles and Practice Using C++ Second Edition



机械工业出版社  
China Machine Press

# C++程序设计 原理与实践（进阶篇） 原书第2版

Programming Principles and Practice Using C++ Second Edition

《C++程序设计：原理与实践（原书第2版）》将经典程序设计思想与C++开发实践完美结合，全面地介绍了程序设计基本原理，包括基本概念、设计和编程技术、语言特性以及标准库等，教你学会如何编写具有输入、输出、计算以及简单图形显示等功能的程序。此外，本书通过对C++思想和历史的讨论、对经典实例（如矩阵运算、文本处理、测试以及嵌入式系统程序设计）的展示，以及对C语言的简单描述，为你呈现了一幅程序设计的全景图。

为方便读者循序渐进地学习，加上篇幅所限，《C++程序设计：原理与实践（原书第2版）》分为基础篇和进阶篇两册出版，基础篇包括第1~11章、第17~19章和附录A、C，进阶篇包括第12~16章、第20~27章和附录B、D、E。本书是进阶篇。

## 本书特点

- **C++初学者的权威指南。**无论你是从事软件开发还是其他领域的工作，本书都将为你打开C++程序设计之门。
- **中高级程序员的必备参考。**通过观察程序设计大师如何处理编程中的各种问题，你将获得新的领悟和指引。
- **全面阐释C++基本概念和技术。**与传统的C++教材相比，本书对基本概念和技术的介绍更为深入，为你编写实用、正确、易维护和有效的代码打下坚实的基础。
- **强调现代C++（C++11和C++14）编程风格。**本书从开篇就介绍现代C++程序设计技术，并揭示了大量关于如何使用C++标准库以及C++11和C++14的特性来简化程序设计的原理，使你快速掌握实用编程技巧。
- **配套教辅资源丰富。**本网站（[www.stroustrup.com/Programming](http://www.stroustrup.com/Programming)）提供了丰富的辅助资料，包括实例源码、PPT、勘误等。

## 作者简介

**本贾尼·斯特劳斯特鲁普**（Bjarne Stroustrup）英国剑桥大学计算机科学博士，C++的设计者和最初的实现者。他现在是德州农工大学计算机科学首席教授。1993年，由于在C++领域的重大贡献，他获得了ACM的 Grace Murray Hopper大奖并成为ACM院士。在进入学术界之前，他在AT&T贝尔实验室工作，是ISO C++标准委员会的创始人之一。



 Pearson

[www.pearson.com](http://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](http://www.hzbook.com)

网上购书：[www.china-pub.com](http://www.china-pub.com)

数字阅读：[www.hzmedia.com.cn](http://www.hzmedia.com.cn)

上架指导：计算机程序设计

ISBN 978-7-111-56252-8



9 787111 562528 >

定价：99.00元



计 算 机 科 学 丛 书

# C++程序设计

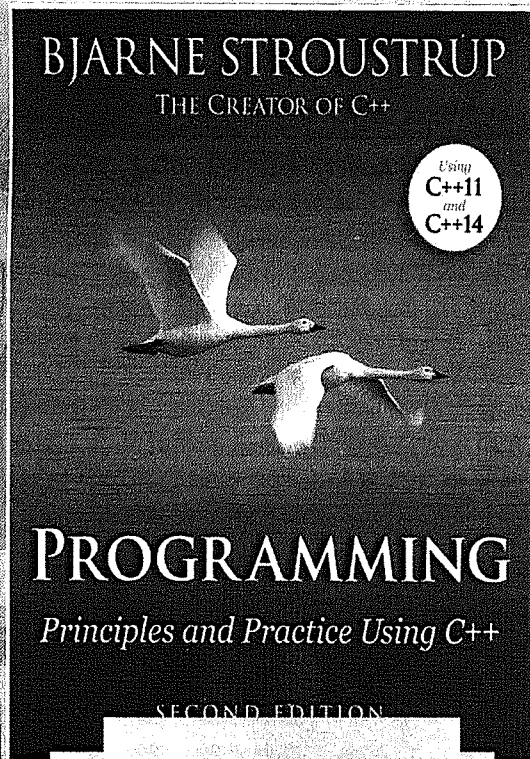
## 原理与实践

### (进阶篇)

[美] 本贾尼·斯特劳斯特鲁普 (Bjarne Stroustrup) 著 刘晓光 李忠伟 王刚 译

Programming

Principles and Practice Using C++ Second Edition



机械工业出版社  
China Machine Press

## 图书在版编目 (CIP) 数据

C++ 程序设计: 原理与实践 (进阶篇) (原书第 2 版) / (美) 本贾尼·斯特劳斯特鲁普 (Bjarne Stroustrup) 著; 刘晓光, 李忠伟, 王刚译. —北京: 机械工业出版社, 2017.3 (计算机科学丛书)

书名原文: Programming: Principles and Practice Using C++, Second Edition

ISBN 978-7-111-56252-8

I. C… II. ①本… ②刘… ③李… ④王… III. C 语言—程序设计—高等学校—教材  
IV. TP312.8

中国版本图书馆 CIP 数据核字 (2017) 第 039914 号

本书版权登记号: 图字: 01-2014-5459

Authorized translation from the English language edition, entitled *Programming: Principles and Practice Using C++, Second Edition* (978-0-321-99278-9) 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 © 2017.

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 (不包括香港、澳门特别行政区及台湾地区) 独家出版发行。未经出版者书面许可, 不得以任何方式抄袭、复制或节录本书中的任何部分。

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

C++ 之父 Bjarne Stroustrup 经典著作《C++ 程序设计: 原理与实践 (原书第 2 版)》基于最新的 C++ 11 和 C++ 14, 广泛地介绍了程序设计的基本概念和技术, 包括类型系统、算术运算、控制结构、错误处理等; 介绍了从键盘和文件获取数值和文本数据的方法以及以图形化方式表示数值数据、文本和几何图形; 介绍了 C++ 标准库中的容器 (如向量、列表、映射) 和算法 (如排序、查找和内积) 的设计和使用。同时还对 C++ 思想和历史进行了详细的讨论, 很好地拓宽了读者的视野。

为方便读者循序渐进地学习, 加上篇幅所限, 《C++ 程序设计: 原理与实践 (原书第 2 版)》分为基础篇和进阶篇两册出版, 基础篇包括第 1 ~ 11 章、第 17 ~ 19 章和附录 A、C, 进阶篇包括第 12 ~ 16 章、第 20 ~ 27 章和附录 B、D、E。本书是进阶篇。

本书通俗易懂、实例丰富, 可作为大学计算机、电子工程、信息科学等相关专业的教材, 也可供相关专业人员参考。

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: 和 静

责任校对: 李秋荣

印 刷: 北京诚信伟业印刷有限公司

版 次: 2017 年 4 月第 1 版第 1 次印刷

开 本: 185mm × 260mm 1/16

印 张: 29 (含 0.75 印张彩插)

书 号: ISBN 978-7-111-56252-8

定 价: 99.00 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

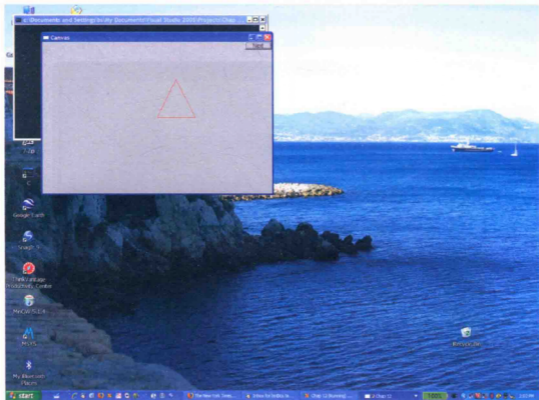
购书热线: (010) 68326294 88379649 68995259

读者信箱: hzjsj@hzbook.com

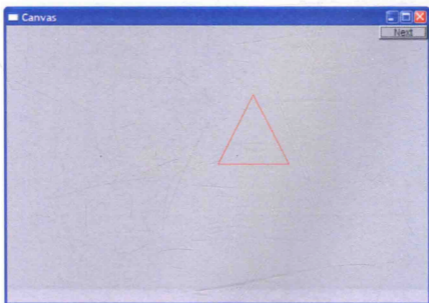
版权所有·侵权必究

封底无防伪标均为盗版

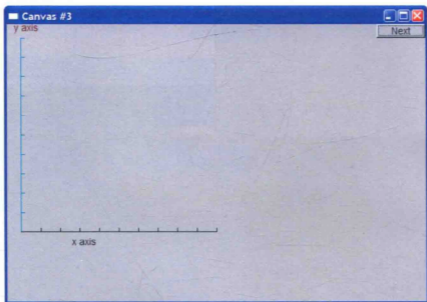
本书法律顾问: 北京大成律师事务所 韩光/邹晓东



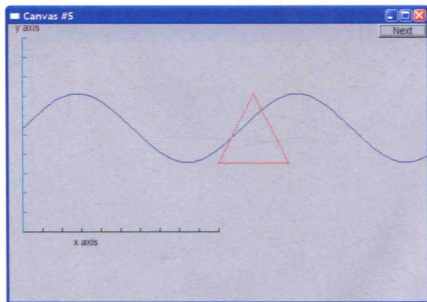
P66 插图



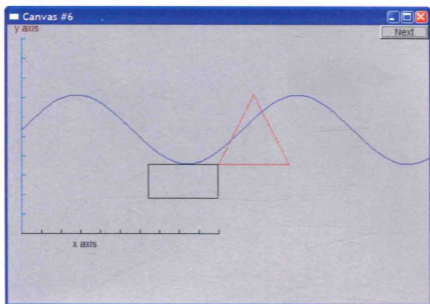
P68 插图



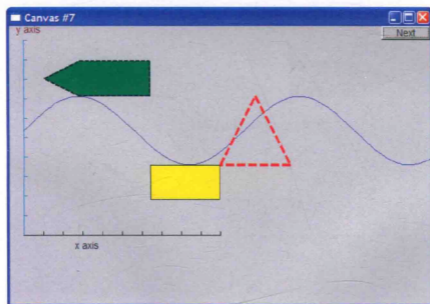
P74 插图



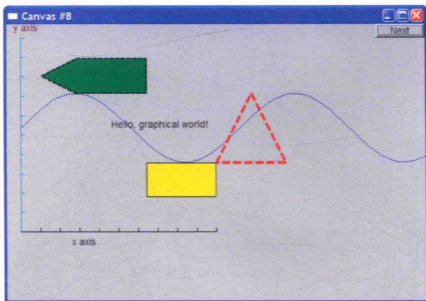
P76 上图



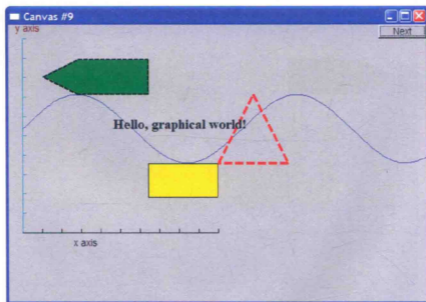
P76 下图



P78 插图

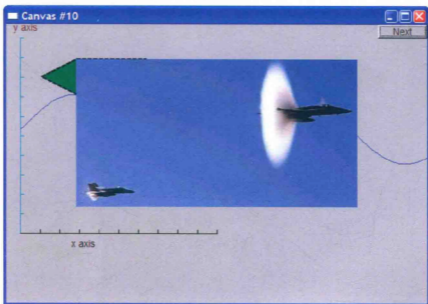


P79 上图

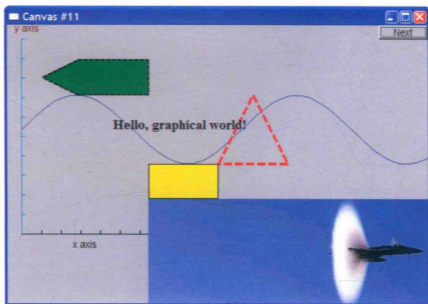


P79 下图

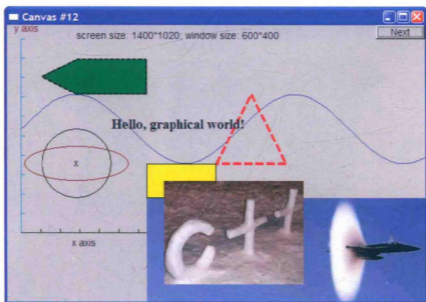




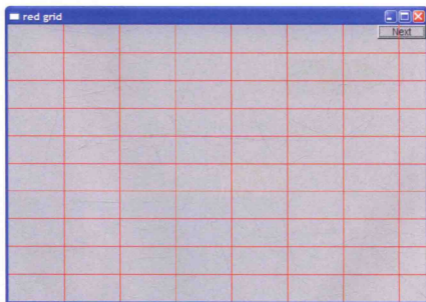
P80 上图



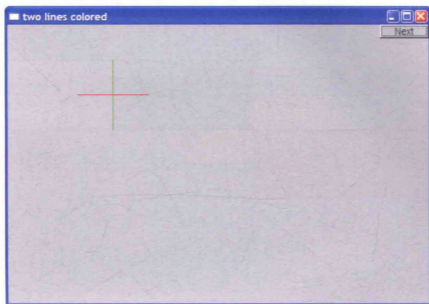
P80 下图



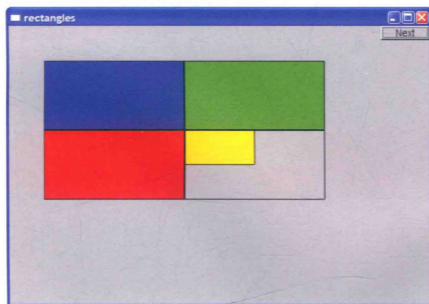
P81 插图



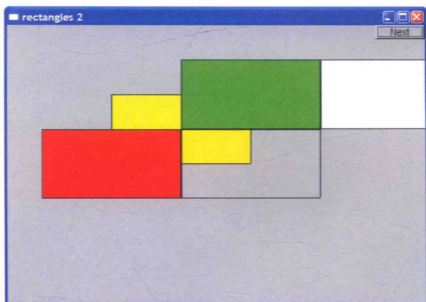
P91 插图



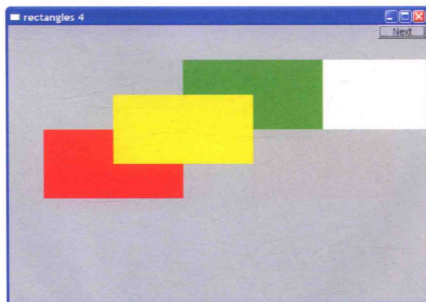
P95 上图



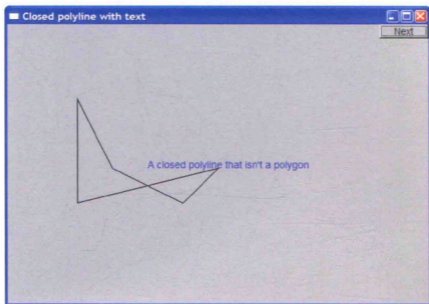
P100 上图



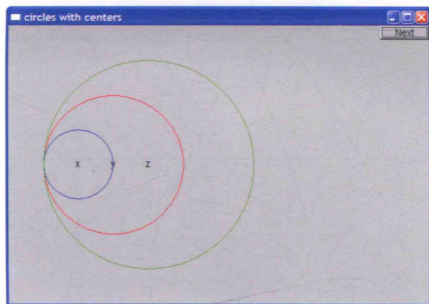
P100 下图



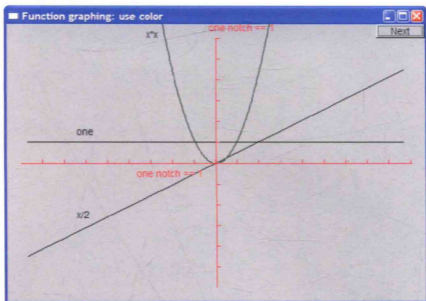
P101 下图



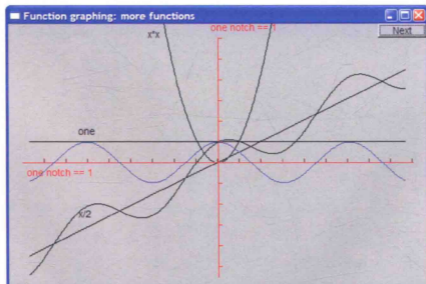
P104 插图



P111 插图

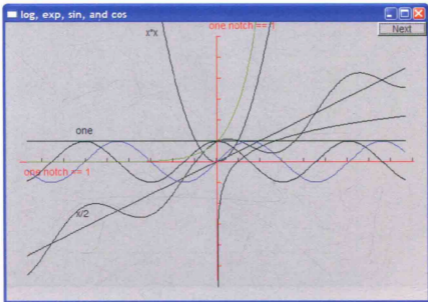


P142 下图

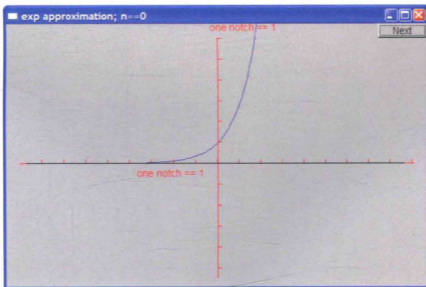


P145 上图

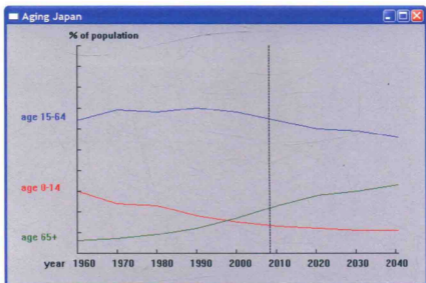




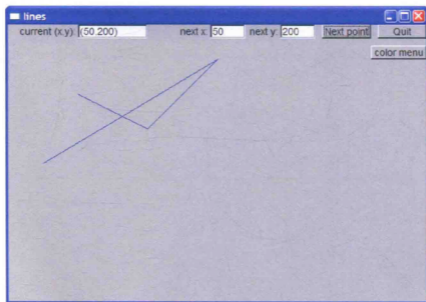
P145 下图



P149 插图



P158 插图



P176 下图

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，不仅肇划了研究的范畴，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，我们的联系方式如下：

华章网站：[www.hzbook.com](http://www.hzbook.com)

电子邮件：[hzsj@hzbook.com](mailto:hzsj@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



华章科技图书出版中心

程序设计是打开计算机世界大门的金钥匙，它使五花八门的软件对你来说不再是“魔法”。C++ 语言则是掌握这把金钥匙的有力武器，它优美、高效，从大洋深处到火星表面，从系统核心到高层应用，从掌中的手机到超级计算机，到处都有 C++ 程序的身影。本书的目标不是作为程序设计语言的简单入门教材，而是成为初学者学习基础实用编程技术的绝佳启蒙。如果你愿意努力学习，本书能帮助你理解使用 C++ 语言进行程序设计的基本原理及大量实践技巧，其中大多数可直接用于其他程序设计语言。基于这一目标，注重实践是本书的明显特点。它希望教会你编写真正能被他人所使用的“有用的程序”，而非“玩具程序”。因此，本书不是机械地介绍各种 C++ 特性，而是针对一些具体问题，不断精化其求解方案，在这个过程中自然地引出基本编程技术及相应的 C++ 程序特性。此外，本书还介绍了大量的求解实际问题的程序设计技术，如语法分析器的设计、图形化程序设计、利用正则表达式处理文本、数值计算程序设计以及嵌入式程序设计等。在其他大多数程序设计入门书籍中，是找不到这些内容的。像调试技术、测试技术等其他程序设计书籍着墨不多的话题，本书也有详细的介绍。程序设计远非遵循语法规则和阅读手册那么简单，而在于理解基本思想、原理和技术，并进行大量实践。本书阐述了这一理念，为如何才能达到编写有用的、优美的程序这一最终目标指引了明确的方向。

本书的作者 Bjarne Stroustrup 是 C++ 语言的设计者和最初的实现者，也是《The C++ Programming Language》(Addison-Wesley 出版社)一书的作者。 he 现在是摩根斯坦利技术部门的总经理和哥伦比亚大学的客座教授，美国国家工程院的院士，ACM 会士和 IEEE 会士。在进入学术界之前，他为 AT&T 贝尔实验室工作多年。他是 ISO 标准组织 C++ 委员会的创建者，现在是该委员会语言演化工作组的主席。本书第 1 版已成为程序设计领域的经典著作，第 2 版又进行了精心的修订，增加了一些新的内容，包括 C++14 的一些新特性。

虽然是面向初学者，但本书原版仍是大部头。为方便读者循序渐进地学习，我们重新组织了章节顺序，将原版书组织为基础篇和进阶篇两册。基础篇包括第 1 ~ 11 章、第 17 ~ 19 章和附录 A、C，进阶篇包括第 12 ~ 16 章、第 20 ~ 27 章和附录 B、D、E。

基础篇	原书	进阶篇	原书
第 1 章	第 1 章	第 15 章	第 20 章
第 2 章	第 2 章	第 16 章	第 21 章
第 3 章	第 3 章	第 17 章	第 12 章
第 4 章	第 4 章	第 18 章	第 13 章
第 5 章	第 5 章	第 19 章	第 14 章
第 6 章	第 6 章	第 20 章	第 15 章
第 7 章	第 7 章	第 21 章	第 16 章
第 8 章	第 8 章	第 22 章	第 22 章
第 9 章	第 9 章	第 23 章	第 23 章
第 10 章	第 10 章	第 24 章	第 24 章

(续)

基础篇	原书	进阶篇	原书
第 11 章	第 11 章	第 25 章	第 25 章
第 12 章	第 17 章	第 26 章	第 26 章
第 13 章	第 18 章	第 27 章	第 27 章
第 14 章	第 19 章	附录 C	附录 B
附录 A	附录 A	附录 D	附录 D
附录 B	附录 C	附录 E	附录 E

基础篇逻辑上分成四部分：第一部分介绍基本的 C++ 程序设计知识，包括第一个“Hello, World!”程序、对象、类型、值、计算、错误处理、函数、类等内容，以及一个计算器程序实例；第二部分介绍字符方式输入输出，包括输入输出流的基本概念和格式化输出方法；第三部分介绍数据结构的基本知识，重点介绍向量以及自由内存空间、数组、模板和异常；第四部分为附录，介绍了 C++ 语言概要和 Visual Studio 简要入门。通过基础篇的学习，读者可掌握 C++ 最基本的语言特性，以及运用这些特性编写高质量程序的基本技巧。

在此基础上，进阶篇希望帮助读者学习一些更高级的编程技术及相应的 C++ 语言特性，也逻辑上分成四部分：第一部分为数据结构和算法进阶知识，介绍容器和迭代器以及算法和映射；第二部分深入讨论输入输出，介绍图形 /GUI 类和图形化程序设计；第三部分希望拓宽读者的视野，介绍程序设计语言的理念和历史、文本处理技术、数值计算、嵌入式程序设计技术及测试技术，此外还较为详细地介绍了 C 语言与 C++ 的异同；第四部分为附录，包括标准库概要、FLTK 安装以及 GUI 实现等内容。

本书的引言、第 1 章以及第 2 ~ 9 章由任明明翻译，第 10、11 章和第 17 ~ 21 章由李忠伟翻译，第 22 ~ 27 章由刘晓光翻译，第 12 ~ 16 章以及前言、附录等由王刚翻译。翻译大师经典，难度超乎想象。接受任务之初，诚惶诚恐；翻译过程中，如履薄冰；完成后，忐忑不安。虽然竭尽全力，但肯定还有很多错漏之处，敬请读者批评指正。

感谢机械工业出版社华章公司的温莉芳总编辑将此重任交付译者，感谢朱劼等老师为本书所付出的心血，没有她们辛苦的编辑和审校，本书不可能完成。

译者

2016 年 11 月于南开大学

该死的鱼雷！全速前进。

——Admiral Farragut

程序设计是这样一门艺术，它将问题求解方案描述成计算机可以执行的形式。程序设计中很多工作都花费在寻找求解方案以及对其求精上。通常，只有在真正编写程序求解一个问题的过程中才会对问题本身理解透彻。

本书适合于那些从未有过编程经验但愿意努力学习程序设计技术的初学者，它能帮助读者理解使用 C++ 语言进行程序设计的基本原理并获得实践技巧。本书的目标是使你获得足够多的知识和经验，以便能使用最新、最好的技术进行简单有用的编程工作。达到这一目标需要多长时间呢？作为大学一年级课程的一部分，你可以在一个学期内完成这本书的学习（假定你有另外四门中等难度的课程）。如果你是自学的话，不要期望能花费更少的时间完成学习（一般来说，每周 15 个小时，14 周是合适的学时安排）。

三个月可能看起来是一段很长的时间，但要学习的内容很多。写第一个简单程序之前，就要花费大约一个小时。而且，所有学习过程都是渐进的：每一章都会介绍一些新的有用的概念，并通过真实应用中的例子来阐述这些概念。随着学习进程的推进，你通过程序代码表达思想的能力——让计算机按你的期望工作的能力，会逐渐稳步地提高。我绝不会说：“先学习一个月的理论知识，然后看看你是否能使用这些理论吧。”

为什么要学习程序设计呢？因为我们的文明是建立在软件之上的。如果不理解软件，那么你将退化到只能相信“魔术”的境地，并且将被排除在很多最为有趣、最具经济效益和社会效益的领域之外。当我谈论程序设计时，我所想到的是整个计算机程序家族，从带有 GUI（图形用户界面）的个人计算机程序，到工程计算和嵌入式系统控制程序（如数码相机、汽车和手机中的程序），以及文字处理程序等，在很多日常应用和商业应用中都能看到这些程序。程序设计与数学有些相似，认真去做的话，会是一种非常有用的智力训练，可以提高我们的思考能力。然而，由于计算机能做出反馈，程序设计不像大多数数学形式那么抽象，因而对多数人来说更易接受。可以说，程序设计是一条能够打开你的眼界，将世界变得更美好的途径。最后，程序设计可以是非常有趣的。

为什么学习 C++ 这门程序设计语言呢？学习程序设计是不可能不借助一门程序设计语言的，而 C++ 直接支持现实世界中的软件所使用的那些关键概念和技术。C++ 是使用最为广泛的程序设计语言之一，其应用领域几乎没有局限。从大洋深处到火星表面，到处都能发现 C++ 程序的身影。C++ 是由一个开放的国际标准组织全面考量、精心设计的。在任何一种计算机平台上都能找到高质量的、免费的 C++ 实现。而且，用 C++ 所学到的程序设计思想，大多数可直接用于其他程序设计语言，如 C、C#、Fortran 以及 Java。最后一个原因，我喜欢 C++ 适合编写优美、高效的代码这一特点。

本书不是初学程序设计的最简单入门教材，我写此书的用意也不在此。我为本书设定的目标是——这是一本能让你学到基本的实用编程技术的最简单书籍。这是一个非常雄心勃勃的目标，因为很多现代软件所依赖的技术，不过才出现短短几年时间而已。



我的基本假设是：你希望编写供他人使用的程序，并愿意认真负责地以较高质量完成这个工作，也就是说，假定你希望达到专业水准。因此，我为本书选择的主题覆盖了开始学习实用编程技术所需要的内容，而不只是那些容易讲授和容易学习的内容。如果某种技术是你做好基本编程工作所需要的，那么本书就会介绍它，同时展示用以支持这种技术的编程思想和语言工具，并提供相应的练习，期望你通过做这些练习来熟悉这种技术。但如果你只想了解“玩具程序”，那么你能学到的将远比我所提供的少得多。另一方面，我不会用一些实用性很低的内容来浪费你的时间，本书介绍的内容都是你在实践中几乎肯定会用到的。

如果你只是希望直接使用别人编写的程序，而不想了解其内部原理，也不想亲自向代码中加入重要的内容，那么本书不适合你，采用另一本书或另一种程序设计语言会更好些。如果这大概就是你对程序设计的看法，那么请同时考虑一下你从何得来的这种观点，它真的满足你的需求吗？人们常常低估程序设计的复杂程度和它的重要性。我不愿看到，你不喜欢程序设计是因为你的需求与我所描述的软件世界之间不匹配而造成的。信息技术世界中有很多地方是不要求程序设计知识的。本书面向的是那些确实希望编写和理解复杂计算机程序的人。

考虑到本书的结构和注重实践的特点，它也可以作为学习程序设计的第二本书，适合那些已经了解一点 C++ 的人，以及那些会用其他语言编程而现在想学习 C++ 的人。如果你属于其中一类，我不好估计你学习这本书要花费多长时间。但我可以给你的建议是，多做练习。因为你在学习中常见的一个问题是习惯用熟悉的、旧的方式编写程序，而不是在适当的地方采用新技术，多做练习会帮助你克服这个问题。如果你曾经按某种更为传统的方式学习过 C++，那么在在进行到第 7 章之前，你会发现一些令你惊奇的、有用的内容。除非你的名字是 Stroustrup，否则你会发现我在本书中所讨论的内容不是“你父辈的 C++”。

学习程序设计要靠编程实践。在这一点上，程序设计与其他需要实践学习的技艺是相似的。你不可能仅仅通过读书就学会游泳、演奏乐器或者开车，必须进行实践。同样，你也不可能不读写大量代码就学会程序设计。本书给出了大量代码实例，都配有说明文字和图表。你需要通过读这些代码来理解程序设计的思想、概念和原理，并掌握用来表达这些思想、概念和原理的程序设计语言的特性。但有一点很重要，仅仅读代码是学不会编程实践技巧的。为此，你必须进行编程练习，通过编程工具熟悉编写、编译和运行程序。你需要亲身体验编程中会出现的错误，学习如何修改它们。总之，在学习程序设计的过程中，编写代码的练习是不可替代的。而且，这也是乐趣所在！

另一方面，程序设计远非遵循一些语法规则和阅读手册那么简单。本书的重点不在于 C++ 的语法，而在于理解基础思想、原理和技术，这是一名好程序员所必备的。只有设计良好的代码才有机会成为一个正确、可靠和易维护的系统的一部分。而且，“基础”意味着延续性：当现在的程序设计语言和工具演变甚至被取代后，这些基础知识仍会保持其重要性。

那么计算机科学、软件工程、信息技术等又如何呢？它们都属于程序设计范畴吗？当然不是！但程序设计是一门基础性的学科，是所有计算机相关领域的基础，在计算机科学领域占有重要的地位。本书对算法、数据结构、用户接口、数据处理和软件工程等领域的重要概念和技术进行了简要介绍，但本书不能替代对这些领域的全面、均衡的学习。

代码可以很有用，同样可以很优美。本书会帮你了解这一点，同时理解优美的代码意味

着什么，并帮你掌握构造优美代码的原理和实践技巧。祝你学习程序设计顺利！

## 致学生

到目前为止，我在德州农工大学已经用本书教过几千名大一新生，其中 60% 曾经有过编程经历，而剩余 40% 从未见过哪怕一行代码。大多数学生的学习是成功的，所以你也可以成功。

你不一定是在某门课程中学习本书，本书也广泛用于自学。然而，不管你学习本书是作为课程的一部分还是自学，都要尽量与他人协作。程序设计有一个不好的名声——它是一种个人活动，这是不公正的。大多数人在作为一个有共同目标的团体的一份子时，工作效果更好，学习得更快。与朋友一起学习和讨论问题不是“作弊”，而是取得进步最有效同时也是最快乐的途径。如果没有特殊情况的话，与朋友一起工作会促使你表达出自己的思想，这正是测试你对问题理解和确认你的记忆的最有效方法。你没有必要独自解决所有编程语言和编程环境上的难题。但是，请不要自欺欺人——不去完成那些简单练习和大量的习题（即使没有老师督促你，你也不应这样做）。记住：程序设计（尤其）是一种实践技能，需要通过实践来掌握。如果你不编写代码（完成每章的若干习题），那么阅读本书就纯粹是一种无意义的理论学习。

大多数学生，特别是那些爱思考的好学生，有时会对自己努力工作是否值得产生疑问。当你产生这样的疑问时，休息一会儿，重新读一下前言，读一下第 1 章和第 22 章。在那里，我试图阐述我在程序设计中发现了哪些令人兴奋的东西，以及为什么我认为程序设计是能为世界带来积极贡献的重要工具。如果你对我的教学哲学和一般方法有疑问，请阅读引言。

你可能会对本书的厚度感到担心。本书如此之厚的一部分原因是，我宁愿反复重复一些解释说明或增加一些实例，而不是让你自己到处找这些内容，这应该令你安心。另外一个主要原因是，本书的后半部分是一些参考资料和补充资料，供你想要深入了解程序设计的某个特定领域（如嵌入式系统程序设计、文本分析或数值计算）时查阅。

还有，学习中请耐心些。学习任何一种重要的、有价值的新技能都要花费一些时间，而这是值得的。

## 致教师

本书不是传统的计算机科学导论书籍，而是一本关于如何构造能实际工作的软件的书。因此本书省略了很多计算机科学系学生按惯例要学习的内容（图灵完全、状态机、离散数学、乔姆斯基文法等）。硬件相关的内容也省略了，因为我假定学生从幼儿园开始就已经通过不同途径使用过计算机了。本书也不准备涉及一些计算机科学领域最重要的主题。本书是关于程序设计的（或者更一般地说，是关于如何开发软件的），因此关注的是少量主题的更深入的细节，而不是像传统计算机课程那样讨论很多主题。本书只试图做好一件事，而且计算机科学也不是一门课程可以囊括的。如果本书被计算机科学、计算机工程、电子工程（我们最早的很多学生都是电子工程专业的）、信息科学或者其他相关专业所采用，我希望这门课程能和其他一些课程一起进行，共同形成对计算机科学的完整介绍。

请阅读引言，那里有对我的教学哲学、一般方法等的介绍。请在教学过程中尝试将这些观点传达给你的学生。

## ISO 标准 C++

C++ 由一个 ISO 标准定义。第一个 ISO C++ 标准于 1998 年获得批准，所以那个版本的 C++ 被称为 C++98。写本书第 1 版时，我正从事 C++11 的设计工作。最令人沮丧的是，当时我还不能使用一些新语言特性（如统一初始化、范围 for 循环、move 语义、lambda 表达式、concept 等）来简化原理和技术的展示。不过，由于设计该书时考虑到了 C++11，所以很容易在合适的地方添加这些特性。在写作本版时，C++ 标准是 2011 年批准的 C++11，2014 ISO 标准 C++14 中的一些特性正在进入主流的 C++ 实现中。本书中使用的语言标准是 C++11，并涉及少量的 C++14 特性。例如，如果你的编译器不能编译下面的代码：

```
vector<int> v1;
vector<int> v2 {v1}; // C++14 风格的拷贝构造
```

可用如下代码替代：

```
vector<int> v1;
vector<int> v2 = v1; // C++98 风格的拷贝构造
```

若你的编译器不支持 C++11，请换一个的编译器。好的、现代的 C++ 编译器可从多处下载，见 [www.stroustrup.com/compilers.html](http://www.stroustrup.com/compilers.html)。使用较早且缺少支持的语言版本会引入不必要的困难。

## 资源

本书支持网站的网址为 [www.stroustrup.com/Programming](http://www.stroustrup.com/Programming)，其中包含了各种使用本书讲授和学习程序设计所需的辅助资料。这些资料可能会随着时间的推移不断改进，但对于初学者，现在可以找到这些资料：

- 基于本书的讲义幻灯片；
- 一本教师指南；
- 本书中使用的库的头文件和实现；
- 本书中实例的代码；
- 某些习题的解答；
- 可能会有用处的一些链接；
- 勘误表。

欢迎随时提出对这些资料的改进意见。

## 致谢

我要特别感谢已故的同事和联合导师 Lawrence “Pete” Petersen，很久以前，在我还未感受到教授初学者的惬意时，是他鼓励我承担这项工作，并向我传授了很多能令课程成功的教学经验。没有他，这门课程的首次尝试就会失败。他参与了这门课程最初的建设，本书就是为这门课程所著。他还和我一起反复讲授这门课程，汲取经验，不断改进课程和本书。在本书中我使用的“我们”这个字眼，最初的意思就是指“我和 Pete”。

我要感谢那些直接或间接帮助过我撰写本书的学生、助教和德州农工大学讲授 ENGR 112、113 及 CSCE 121 课程的教师，以及 Walter Daugherty、Hyunyoung Lee、Teresa Leyk、Ronnie Ward、Jennifer Welch，他们也讲授过这门课程。还要感谢 Damian Dechev、Tracy

Hammond、Arne Tolstrup Madsen、Gabriel Dos Reis、Nicholas Stroustrup、J. C. van Winkel、Greg Versoonder、Ronnie Ward 和 Leor Zolman 对本书初稿提出的建设性意见。感谢 Mogens Hansen 为我解释引擎控制软件。感谢 Al Aho、Stephen Edwards、Brian Kernighan 和 Daisy Nguyen 帮助我在夏天躲开那些分心的事，完成本书。

感谢 Art Werschulz，他在纽约福特汉姆大学的课程中使用了本书第 1 版，并据此提出了很多建设性的意见。还要感谢 Nick Maclaren，他在剑桥大学使用了本书的第 1 版，并对本书的习题提出了非常详尽的建议。他的学生在知识背景和专业需求上与德州农工大学大一学生有巨大的差异。

感谢 Addison-Wesley 公司为我安排的审阅者 Richard Enbody、David Gustafson、Ron McCarty 和 K. Narayanaswamy，他们基于自身讲授 C++ 课程或大学计算机科学导论课程的经验，对本书提出了宝贵的意见。还要感谢我的编辑 Peter Gordon 为本书提出的很多有价值的意见以及极大的耐心。我非常感谢 Addison-Wesley 公司的制作团队，他们为本书的高质量出版做出了很多贡献，他们是：Linda Begley（校对），Kim Arney（排版），Rob Mauhar（插图），Julie Nahil（制作编辑），Barbara Wood（文字编辑）。

感谢本书第 1 版的译者，他们发现并帮助澄清了很多问题。特别是，Loïc Joly 和 Michel Michaud 在法语翻译版中做了全面的技术检查，修改了很多问题。

我还要感谢 Brian Kernighan 和 Doug McIlroy 为撰写程序设计类书籍定下了一个非常高的标杆，以及 Dennis Ritchie 和 Kristen Nygaard 为实用编程语言设计提供的非常有价值的经验。

当实际地形与地图不符时，相信实际地形。

——瑞士军队谚语

## 讲授和学习本书的方法

我们是如何帮助你学习的？又是如何安排学习进程的？我们的做法是，尽力为你提供编写高效的实用程序所需的最基本的概念、技术和工具，包括程序组织、调试和测试、类设计、计算、函数和算法设计、绘图方法（仅介绍二维图形）、图形用户界面（GUI）、文本处理、正则表达式匹配、文件和流输入输出（I/O）、内存管理、科学/数值/工程计算、设计和编程思想、C++ 标准库、软件开发策略、C 语言程序设计技术。认真完成这些内容的学习，我们会学到如下程序设计技术：过程式程序设计（C 语言程序设计风格）、数据抽象、面向对象程序设计和泛型程序设计。本书的主题是程序设计，也就是表达代码意图所需的思想、技术和工具。C++ 语言是我们的主要工具，因此我们比较详细地描述了很多 C++ 语言的特性。但请记住，C++ 只是一种工具，而不是本书的主题。本书是“用 C++ 语言进行程序设计”，而不是“C++ 和一点程序设计理论”。

我们介绍的每个主题都至少出于两个目的：提出一种技术、概念或原理，介绍一个实用的语言特性或库特性。例如，我们用一个二维图形绘制系统的接口展示如何使用类和继承。这使我们节省了篇幅（也节省了你的时间），并且还强调了程序设计不只是简单地将代码拼装起来以尽快地得到一个结果。C++ 标准库是这种“双重作用”例子的主要来源，其中很多主题甚至具有三重作用。例如，我们会介绍标准库中的向量类 `vector`，用它来展示一些广泛使用的设计技术，并展示很多用来实现 `vector` 的程序设计技术。我们的一个目标是向你展示一些主要的标准库功能是如何实现的，以及它们如何与硬件相配合。我们坚持认为一个工匠必须了解他的工具，而不是仅仅把工具当作“有魔力的东西”。

对于一个程序员来说，总是会对某些主题比其他主题更感兴趣。但是，我们建议你不要预先判断你需要什么（你怎么知道你将来会需要什么呢？），至少每一章都要浏览一下。如果你学习本书是作为一门课程的一部分，你的老师会指导你如何选择学习内容。

我们的教学方法可以描述为“深度优先”，同时也是“具体优先”和“基于概念”。首先，我们快速地（好吧，是相对快速地，从第 1 章到第 11 章）将一些编写小的实用程序所需的技巧提供给你。在这期间，我们还简明扼要地提出很多工具和技术。我们着重于简单具体的代码实例，因为相对于抽象概念，人们能更快领会具体实例，这就是多数人的学习方法。在最初阶段，你不应期望理解每个小的细节。特别是，你会发现对刚刚还工作得好好的程序稍加改动，便会呈现出“神秘”的效果。尽管如此，你还是要尝试一下！还有，请完成我们提供的简单练习和习题。请记住，在学习初期你只是没有掌握足够的概念和技巧来准确判断什么是简单的，什么是复杂的。请等待一些惊奇的事情发生，并从中学习吧。

我们会快速通过这样一个初始阶段——我们想尽可能快地带你进入编写有趣程序的阶段。有些人可能会质疑，“我们的进展应该慢些、谨慎些，我们应该先学会走，再学跑！”但

是你见过小孩学习走路吗？实际上小孩在学会平稳地慢慢走路之前就开始尝试跑了。与之相似，你可以先勇猛向前，偶尔摔一跤，从中获得编程的感觉，然后再慢下来，获得必要的精确控制能力和准确的理解。你必须在学会走之前就开始跑！

⚠ 你不要投入大量精力试图学习一些语言或技术细节的所有相关内容。例如，你可以熟记所有 C++ 的内置类型及其使用规则。你当然可以这么做，而且这么做会使你觉得自己很博学。但是，这不会使你成为一名程序员。如果你学习中略过一些细节，将来可能偶尔会因为缺少相关知识而被“灼伤”，但这是获取编写好程序所需的完整知识结构的最快途径。注意，我们的这种方法本质上就是小孩学习其母语的方法，也是教授外语的最有效方法。有时你不可避免地会被难题困住，我们鼓励你向授课老师、朋友、同事、指导教师等寻求帮助。请放心，在前面这些章节中，所有内容本质上都不困难。但是，很多内容是你所不熟悉的，因此最初可能会感觉有点难。

随后，我们介绍一些入门技巧来拓宽你的知识。我们通过实例和习题来强化你的理解，为你提供一个程序设计的概念基础。

☞ 我们非常强调思想和原理。思想能指导你求解实际问题——可以帮助你知在什么情况下问题求解方案是好的、合理的。你还应该理解这些思想背后的原理，从而理解为什么要接受这些思想，为什么遵循这些思想会对你和使用你的代码的用户有帮助。没有人会满意“因为事情就是如此”这样的解释。更为重要的是，如果真正理解了思想和原理，你就能将自己已知的知识推广到新的情况；就能用新的方法将思想和工具结合起来解决新的问题。知其所以然是学会程序设计技巧所必需的。相反，仅仅不求甚解地记住大量规则和语言特性有很大局限，是错误之源，是在浪费时间。我们认为你的时间很珍贵，尽量不要浪费它。

我们把很多 C++ 语言层面的技术细节放在了附录和手册中，你可以随时按需查找。我们假定你有能力查找到需要的信息，你可以借助目录来查找信息。不要忘了编译器和互联网的在线功能。但要记住，要对所有互联网资源保持足够的怀疑，直至你有足够的理由相信它们。因为很多看起来很权威的网站实际上是由程序设计新手或者想要出售什么东西的人建立的。而另外一些网站，其内容都是过时的。我们在支持网站 [www.stroustrup.com/Programming](http://www.stroustrup.com/Programming) 上列出了一些有用的网站链接和信息。

请不要过于急切地期盼“实际的”例子。我们理想的实例都是能直接说明一种语言特性、一个概念或者一种技术的简短代码。很多现实世界中的实例比我们给出的实例要凌乱很多，而且所能展示的知识也不比我们的实例更多。包含数十万行代码的成功商业程序中所采用的技术，我们用几个 50 行规模的程序就能展示出来。理解现实世界程序的最快途径是好好研究一些基础的小程序。

另一方面，我们不会用“聪明可爱的风格”来阐述我们的观点。我们假定你的目标是编写供他人使用的实用程序，因此书中给出的实例要么是用来说明语言特性，要么是从实际应用中提取出来的。我们的叙述风格都是用专业人员对（将来的）专业人员的那种口气。

## 一般方法

☞ 本书的内容组织适合从头到尾一章一章地阅读，当然，你也常常要回过头来对某些内容读上第二遍、第三遍。实际上，这是一种明智的方法，因为当遇到还看不出什么门道的地方时，你通常会快速掠过。对于这种情况，你最终还是再次回到这个地方。然而，这么做要适度，因为除了交叉引用之外，对本书其他部分，你随便翻开一页，就从那里开始学习，



并希望成功，是不可能的。本书每一节、每一章的内容安排，都假定你已经理解了之前的内容。

本书的每一章都是一个合理的自包含单元，这意味着应一口气读完（当然这只是理论上，实际上由于学生紧密的学习计划，不总是可行）。这是将内容划分为章的主要标准。其他标准包括：从简单练习和习题的角度，每章是一个合适的单元；每一章提出一些特定的概念、思想或技术。这种标准的多样性使得少数章过长，所以不要教条地遵循“一口气读完”的准则。特别是当你已经考虑了思考题，做了简单练习，并做了一些习题时，你通常会发现你需要回过头去重读一些小节和几天前读过的内容。

“它回答了我想到的所有问题”是对一本教材常见的称赞，这对细节技术问题是理想的，而早期的读者也发现本书有这样的特性。但是，这不是全部的理想，我们希望提出初学者可能想不到的问题。我们的目标是，回答那些你在编写供他人使用的高质量软件时需要考虑的问题。学习回答好的（通常也是困难的）问题是学习如何像一个程序员那样思考所必需的。只回答那些简单的、浅显的问题会使你感觉良好，但无助于你成长为一名程序员。

我们努力尊重你的智力，珍惜你的时间。在本书中，我们以专业性而不是精明伶俐为目标，宁可节制地表达一个观点而不大肆渲染它。我们尽力不夸大一种程序设计技术或一个语言特性的重要性，但请不要因此低估“这通常是有用的”这种简单陈述的重要程度。如果我们平静地强调某些内容是重要的，意思是你如果不掌握它，或早或晚都会因此而浪费时间。

我们不会伪称本书中的思想和工具是完美的。实际上没有任何一种工具、库、语言或者技术能够解决程序员所面临的所有难题，至多能帮助你开发、表达你的问题求解方案而已。我们尽量避免“无害的谎言”，也就是说，我们会尽力避免过于简单的解释，虽然这些解释清晰且易理解，但在实际编程和问题求解时却容易弄错。另一方面，本书不是一本参考手册，如果需要 C++ 详细完整的描述，请参考 Bjarne Stroustrup 的《The C++ Programming Language》第 4 版（Addison-Wesley 出版社，2013 年）和 ISO 的 C++ 标准。

## 简单练习和习题等

程序设计不仅仅是一种脑力活动，实际动手编写程序是掌握程序设计技巧必不可少的一环。本书提供两个层次的程序设计练习：

- **简单练习：**简单练习是一种非常简单的习题，其目的是帮助学生掌握一些相对死板的实际编程技巧。一个简单练习通常由一系列的单个程序修改练习组成。你应该完成所有简单练习。完成简单练习不需要很强的理解能力、很聪明或者很有创造性。简单练习是本书的基本组成部分，如果你没有完成简单练习，就不能说完成了本书的学习。
- **习题：**有些习题比较简单，有些则很难，但多数习题都是想给学生留下一定的创造和想象空间。如果时间紧张，你可以做少量习题，但题量至少应该能使你弄清楚哪些内容对你来说比较困难，在此基础上应该再多做一些，这是你的成功之道。我们希望本书的习题都是学生能够做出来的，而不是需要超乎常人的智力才能解答的复杂难题。但是，我们还是期望本书习题能给你足够多的挑战，能用光甚至是最好的学生的所有时间。我们不期待你能完成所有习题，但请尽情尝试。

另外，我们建议每个学生都能参与到一个小的项目中去（如果时间允许，能参与更多项

目当然就更好了)。一个项目的目的就是要编写一个完整的有用程序。理想情况下,一个项目由一个多人小组(比如三个人)共同完成。大多数人会发现做项目非常有趣,并在这个过程中学会如何把很多事情组织在一起。

一些人喜欢在读完一章之前就把书扔到一边,开始尝试做一些实例程序;另一些人则喜欢把一章读完后,再开始编码。为了帮助前一种读者,我们用“试一试”板块给出了对于编程实践的一些简单建议。一个“试一试”通常来说就是一个简单练习,而且只着眼于前面刚刚介绍的主题。如果你略过了一个“试一试”而没有去尝试它,那么最好在做这一章的简单练习时做一下这个题目。“试一试”要么是该章简单练习的补充,要么干脆就是其中的一部分。

在每章末尾你都会看到一些思考题,我们设置这些思考题是想为你指出这一章中的重点内容。一种学习思考题的方法是把它们作为习题的补充:习题关注程序设计的实践层面,而思考题则试图帮你强化思想和概念。因此,思考题有点像面试题。

每章最后都有“术语”一节,给出本章中提出的程序设计或 C++ 方面的基本词汇表。如果你希望理解别人关于程序设计的陈述,或者想明确表达出自己的思想,就应该首先弄清术语表中每个术语的含义。

—重复是学习的有效手段,我们希望每个重要的知识点都在书中至少出现两次,并通过习题再次强调。

## 进阶学习

当你完成本书的学习时,是否能成为一名程序设计和 C++ 方面的专家呢?答案当然是否定的!如果做得好的话,程序设计会是一门建立在多种专业技能上的精妙的、深刻的、需要高度技巧的艺术。你不能期望花四个月时间就成为一名程序设计专家,这与其他学科一样:你不能期望花四个月、半年或一年时间就成为一名生物学专家、一名数学家、一名自然语言(如中文、英文或丹麦文)方面的专家,或是一名小提琴演奏家。但如果你认真地学完了这本书,你可以期待也应该期待的是:你已经在程序设计领域有了一个很好的开始,已经可以写相对简单的、有用的程序,能读更复杂的程序,而且已经为进一步的学习打下了良好的理论和实践基础。

学习完这门入门课程后,进一步学习的最好方法是开发一个真正能被别人使用的程序。在完成这个项目之后或者同时(同时可能更好)学习一本专业水平的教材(如 Stroustrup 的《The C++ Programming Language》),学习一本与你做的项目相关的更专门的书(比如,你如果在做 GUI 相关项目的话,可选择关于 Qt 的书,如果在做分布式程序的话,可选择关于 ACE 的书),或者学习一本专注于 C++ 某个特定方面的书(如 Koenig 和 Moo 的《Accelerated C++》、Sutter 的《Exceptional C++》或 Gamma 等人的《Design Patterns》)。完整的参考书目参见本引言或本书最后的参考文献。

最后,你应该学习另一门程序设计语言。我们认为,如果只懂一门语言,你是不可能成为软件领域的专家的(即使你并不是想做一名程序员)。

## 本书内容顺序的安排

讲授程序设计有很多方法。很明显,我们不赞同“我学习程序设计的方法就是最好的学习方法”这种流行的看法。为了方便学习,我们较早地提出一些仅仅几年前还是先进技术的

内容。我们的设想是，本书内容的顺序完全由你学习程序设计过程中遇到的问题来决定，随着你对程序设计的理解和实际动手能力的提高，一个主题一个主题地平滑向前推进。本书的叙述顺序更像一部小说，而不是一部字典或者一种层次化的顺序。

一次性地学习所有程序设计原理、技术和语言功能是不可能的。因此，你需要选择其中一个子集作为起点。更一般地，一本教材或一门课程应该通过一系列的主题子集来引导学生。我们认为，选择适当的主题并给出重点是我们的责任。我们不能简单地罗列出所有内容，必须做出取舍；在每个学习阶段，我们选择省略的内容与选择保留的内容至少同样重要。

作为对照，这里列出我们决定不采用的教学方法（仅仅是一个缩略列表），对你可能有用：

- C 优先：用这种方法学习 C++ 完全是浪费学生的时间，学生能用来求解问题的语言功能、技术和库比所需的要少得多，这样的程序设计实践很糟糕。与 C 相比，C++ 能提供更强的类型检查、对新手来说更好的标准库以及用于错误处理的异常机制。
- 自底向上：学生本该学习好的、有效的程序设计技巧，但这种方法分散了学生的注意力。学生在求解问题过程中所能依靠的编程语言和库方面的支持明显不足，这样的编程实践质量很低、毫无用处。
- 如果你介绍某些内容，就必须介绍它的全部：这实际上意味着自底向上方法（一头扎进涉及的每个主题，越陷越深）。这种方法硬塞给初学者很多他们并不感兴趣而且可能很长时间内都用不上的技术细节，令他们厌烦。这样做毫无必要，因为一旦学会了编程，你完全可以自己到手册中查找技术细节。这是手册擅长的方面，如果用来学习基本概念就太可怕了。
- 自顶向下：这种方法对一个主题从基本原理到细节逐步介绍，倾向于把读者的注意力从程序设计的实践层面上转移开，迫使读者一直专注于上层概念，而没有任何机会实际体会这些概念的重要性。这是错误的，例如，如果你没有实际体会到编写程序是那么容易出错，而修正一个错误是那么困难，你就无法体会到正确的软件开发原理。
- 抽象优先：这种方法专注于一般原理，保护学生不受讨厌的现实问题限制条件的困扰，这会导致学生轻视实际问题、语言、工具和硬件限制。通常，这种方法基于“教学用语言”——一种将来不可能实际应用，有意将学生与实际的硬件和系统问题隔绝开的语言。
- 软件工程理论优先：这种方法和抽象优先的方法具有与自顶向下方法一样的缺点：没有具体实例和实践体验，你无法体会到抽象理论的价值和正确的软件开发实践技巧。
- 面向对象先行：面向对象程序设计是一种组织代码和开发工作的很好方法，但并不是唯一有效的方法。特别是，以我们的体会，在类型系统和算法式编程方面打下良好的基础，是学习类和类层次设计的前提条件。本书确实从一开始就使用了用户自定义类型（一些人称之为“对象”），但我们直到第 6 章才展示如何设计一个类，而直到第 17 章才展示了类层次。
- 相信魔法：这种方法只是向初学者展示强有力的工具和技术，而不介绍其下蕴含的技术和特性。这让学生只能去猜这些工具和技术为什么会有这样的表现，使用它们会付出多大代价，以及它们恰当的应用范围，而通常学生会猜错！这会导致学生过分刻板地遵循相似的工作模式，成为进一步学习的障碍。

自然，我们不会断言这些我们没有采用的方法毫无用处。实际上，在介绍一些特定的内容时，我们使用了其中一些方法，学生能体会到这些方法在这些特殊情况下的优点。但是，当学习程序设计是以实用为目标时，我们不把这些方法作为一般的教学方法，而是采用其他方法：主要是具体优先和深度优先方法，并对重点概念和技术加以强调。

## 程序设计和程序设计语言

✂ 我们首先介绍程序设计，把程序设计语言放在第二位。我们介绍的程序设计方法适用于任何通用的程序设计语言。我们的首要目的是帮助你学习一般概念、理论和技术，但是这些内容不能孤立地学习。例如，不同程序设计语言在语法细节、编程思想的表达以及工具等方面各不相同。但对于编写无错代码的很多基本技术，如编写逻辑简单的代码（第 5 章和第 6 章），构造不变式（9.4.3 节），以及接口和实现细节分离（9.7 节和 19.1 ~ 19.2 节）等，不同程序设计语言则差别很小。

程序设计技术的学习必须借助于一门程序设计语言，代码设计、组织和调试等技巧是不可能从抽象理论中学到的。你必须用某种程序设计语言编写代码，从中获取实践经验。这意味着你必须学习一门程序设计语言的基本知识。这里说“基本知识”，是因为花几个星期就能掌握一门主流实用编程语言全部内容的日子已经一去不复返了。本书中 C++ 语言相关的内容只是我们选出的它的一个子集，是与编写高质量代码关系最紧密的那部分内容。而且，我们所介绍的 C++ 特性都是你肯定会用到的，因为这些特性要么是出于逻辑完整性的要求，要么是 C++ 社区中最常见的。

## 可移植性

✂ 编写运行于多种平台的 C++ 程序是很常见的情况。一些重要的 C++ 应用甚至运行于我们闻所未闻的平台！我们认为可移植性和对多种平台架构 / 操作系统的利用是非常重要的特性。本质上，本书的每个例子都不仅是 ISO 标准 C++ 程序，还是可移植的。除非特别指出，本书的代码都能运行于任何一种 C++ 实现，并且确实已经在多种计算机平台和操作系统上测试通过了。

不同系统编译、链接和运行 C++ 程序的细节各不相同，如果每当提及一个实现问题时就介绍所有系统和所有编译器的细节，是非常单调乏味的。我们在附录 B 中给出了 Windows 平台 Visual Studio 和 Microsoft C++ 入门的大部分基本知识。

如果你在使用任何一种流行的但相对复杂的 IDE（集成开发环境，Integrated Development Environment）时遇到了困难，我们建议你尝试命令行工作方式，它极其简单。例如，下面给出的是在 Unix 或 Linux 平台用 GNU C++ 编译器编译、链接和运行一个包含两个源文件 `my_file1.cpp` 和 `my_file2.cpp` 的简单程序所需的全部命令：

```
c++ -o my_program my_file1.cpp my_file2.cpp
./my_program
```

是的，这真的就是全部。

## 提示标记

✂ 为了方便读者回顾本书，以及帮读者发现第一次阅读时遗漏的关键内容，我们在页边空白处放置三种“提示标记”：

- ✂: 概念和技术。
- 📌: 建议。
- ⚠: 警告。

## 附言

很多章最后都提供了一个简短的“附言”，试图给出本章所介绍内容的全景描述。我们这样做是因为意识到，知识可能是（而且通常就是）令人畏缩的，只有当完成了习题、学习了进一步的章节（应用了本章中提出的思想）并进行了复习之后才能完全理解。不要恐慌，放轻松，这是很自然的，可以预料到的。你不可能一天之内就成为专家，但可以通过学习本书逐步成为一名合格的程序员。学习过程中，你会遇到很多知识、实例和技术，很多程序员已经从中发现了令人激动的和有趣的东西。

## 程序设计和计算机科学

程序设计就是计算机科学的全部吗？答案当然是否定的！我们提出这一问题的唯一原因就是确实曾有人将其混淆。本书会简单涉及计算机科学的一些主题，如算法和数据结构，但我们的目标还是讲授程序设计：设计和实现程序。这比广泛接受的计算机科学的概念更宽，但也更窄：

- 更宽，因为程序包含很多专业技巧，通常不能归类于任何一种科学。
- 更窄，因为就涉及的计算机科学的内容而言，我们没有系统地给出其基础。

本书的目标是作为一门计算机科学课程的一部分（如果成为一个计算机科学家是你的目标的话），成为软件构造和维护领域第一门课程的基础（如果你希望成为一个程序员或者软件工程师的话），总之是更大的完整系统的一部分。

本书自始至终都依赖计算机科学，我们也强调基本原理，但我们是理论和经验为基础来讲授程序设计，是把它作为一种实践技能，而不是一门科学。

## 创造性和问题求解

本书的首要目标是帮助你学会用代码表达自己的思想，而不是教你如何获得这些思想。沿着这样一个思路，我们给出很多实例，展示如何求解问题。每个实例通常先分析问题，随后对求解方案逐步求精。我们认为程序设计本身是问题求解的一种描述形式：只有完全理解了一个问题及其求解方案，你才能用程序来正确表达它；而只有通过构造和测试一个程序，你才能确定你对问题和求解方案的理解是完整、正确的。因此，程序设计本质上是理解问题和求解方案工作的一部分。但是，我们的目标是通过实例而不是通过“布道”或是问题求解详细“处方”的展示来说明这一切。

## 反馈方法

我们不认为存在完美的教材；个人的需求总是差别很大的。但是，我们愿意尽力使本书和支持材料更接近完美。为此，我们需要大家的反馈，脱离读者是不可能写出好教材的。请大家给我们发送反馈报告，包括内容错误、排版错误、含混的文字、缺失的解释等。我们也感谢有关更好的习题、更好的实例、增加内容、删除内容等的建议。大家提出的建设性意见会帮助将来的读者，我们会将勘误表张贴在支持网站：[www.stroustrup.com/Programming](http://www.stroustrup.com/Programming)。

## 参考文献

下面列出了前面提及的参考文献，以及可能对你有用的一些文献。

Becker, Pete, ed. *The C++ Standard*. ISO/IEC 14882:2011.

Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4, Second Edition*. Prentice Hall, 2008. ISBN 0132354160.

Koenig, Andrew, and Barbara E. Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 020170353X.

Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*. Addison-Wesley, 2005. ISBN 0321334876.

Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2001. ISBN 0201604647.

Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2002. ISBN 0201795256.

Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.

Stroustrup, Bjarne. "Learning Standard C++ as a New Language." *C/C++ Users Journal*, May 1999.

Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 0321563840.

Stroustrup, Bjarne. *A Tour of C++*. Addison-Wesley, 2013. ISBN 0321958314.

Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 1999. ISBN 0201615622.

更全面的参考文献列表可以在本书最后找到。

你也许有理由问：“是一些什么人想要教我程序设计？”那么，下面给出作者的一些生平信息。Bjarne Stroustrup 和 Lawrence “Pete” Petersen 合著了本书。Stroustrup 还设计并讲授了面向大学一年级学生的课程，这门课程是与本书同步发展起来的，以本书的初稿作为教材。

## Bjarne Stroustrup

我是 C++ 语言的设计者和最初的实现者。在过去大约 40 年间，我使用 C++ 和许多其他程序设计语言进行过各种各样的编程工作。我喜欢那些用在富有挑战性的应用（如机器人控制、绘图、游戏、文本分析以及网络应用）中的优美而又高效的代码。我教过能力和兴趣各异的人设计、编程和 C++ 语言。我是 ISO 标准组织 C++ 委员会的创建者，现在是该委员会语言演化工作组的主席。

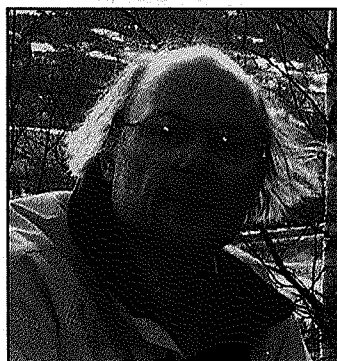
这是我第一本人门性的书。我编著的其他书籍如《The C++ Programming Language》和《The Design and Evolution of C++》都是面向有经验的程序员的。

我生于丹麦奥尔胡斯一个蓝领（工人阶级）家庭，在家乡的大学获得了数学与计算机科学硕士学位。我的计算机科学博士学位是在英国剑桥大学获得的。我为 AT&T 工作了大约 25 年，最初在著名的贝尔实验室的计算机科学研究中心——Unix、C、C++ 及其他很多东西的发明地，后来在 AT&T 实验室研究中心。

我现在的美国国家工程院的院士，ACM 会士（Fellow）和 IEEE 会士。我获得了 2005 年度 Sigma Xi（科学研究协会）的科学成就 William Procter 奖，我是首位获得此奖的计算机科学家。2010 年，我获得了丹麦奥尔胡斯大学最古老也最富声望的奖项 Rigmor og Carl Holst-Knudsens Videnskapspris，该奖项颁发给为科学做出贡献的与该校有关的人士。2013 年，我被位于俄罗斯圣彼得堡的信息技术、力学和光学（ITMO）国立研究大学授予计算机科学荣誉博士学位。

至于工作之外的生活，我已婚，有两个孩子，一个是医学博士，另一个在进行博士后研究。我喜欢阅读（包括历史、科幻、犯罪及时事等各类书籍），还喜欢各种音乐（包括古典音乐、摇滚、蓝调和乡村音乐）。和朋友一起享受美食是我生活中必不可少的一部分，我还喜欢参观世界各地有趣的地方。为了能够享受美食，我还坚持跑步。

关于我的更多信息，请见我的网站 [www.stroustrup.com](http://www.stroustrup.com)。特别是，你可以在那里找到我名字的正确发音。



## Lawrence “Pete” Petersen

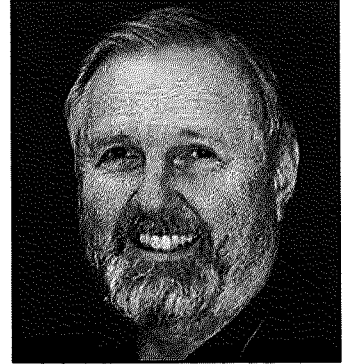
2006年年末，Pete 如此介绍他自己：“我是一名教师。近20年来，我一直在德州农工大学讲授程序设计语言。我已5次被学生选为优秀教师，并于1996年被工程学院的校友会选为杰出教师。我是 Wakonse 优秀教师计划的委员和教师发展研究院院士。

作为一名陆军军官的儿子，我的童年是在不断迁移中度过。在华盛顿大学获得哲学学位后，我作为野战炮兵官员和操作测试研究分析员在军队服役了22年。1971年至1973年期间，我在俄克拉荷马希尔堡讲授野战炮兵军官的高级课程。1979年，我帮助创建了测试军官的训练课程，并在1978年至1981年及1985年至1989年期间在跨越美国的九个不同地方以首席教官的身份讲授这门课程。

1991年我组建了一个小型的软件公司，生产供大学院系使用的管理软件，直至1999年。我的兴趣在于讲授、设计和实现供人们使用的实用软件。我在乔治亚理工大学获得了工业管理学硕士学位，在德州农工大学获得了教育管理学硕士学位。我还从 NTS 获得了微型计算机硕士学位。我在德州农工大学获得了信息与运营管理学博士学位。

我和我的妻子 Barbara 都生于德州的布莱恩。我们喜欢旅行、园艺和招待朋友；我们花尽可能多的时间陪我们的儿子和他们的家庭，特别是我们的孙子和孙女 Angelina、Carlos、Tess、Avery、Nicholas 和 Jordan。”

令人悲伤的是，Pete 于2007年死于肺癌。如果没有他，这门课程绝对不会取得成功。





出版者的话	
译者序	
前言	
引言	
作者简介	
<b>第 15 章 容器和迭代器</b> .....	<b>1</b>
15.1 存储和处理数据 .....	1
15.1.1 处理数据 .....	1
15.1.2 泛化代码 .....	2
15.2 STL 理念 .....	4
15.3 序列和迭代器 .....	7
15.3.1 回到实例 .....	8
15.4 链表 .....	9
15.4.1 链表操作 .....	11
15.4.2 遍历 .....	12
15.5 再次泛化 vector .....	13
15.5.1 遍历容器 .....	15
15.5.2 auto .....	15
15.6 实例：一个简单的文本编辑器 .....	16
15.6.1 处理行 .....	18
15.6.2 遍历 .....	18
15.7 vector、list 和 string .....	21
15.7.1 insert 和 erase .....	22
15.8 调整 vector 类达到 STL 版本的 功能 .....	24
15.9 调整内置数组达到 STL 版本的 功能 .....	26
15.10 容器概览 .....	27
15.10.1 迭代器类别 .....	28
简单练习 .....	29
思考题 .....	30
术语 .....	30
习题 .....	31
附言 .....	32
<b>第 16 章 算法和映射</b> .....	<b>33</b>
16.1 标准库算法 .....	33
16.2 最简单的算法 find() .....	34
16.2.1 一些一般的应用 .....	35
16.3 通用搜索算法 find_if() .....	36
16.4 函数对象 .....	38
16.4.1 函数对象的抽象视图 .....	39
16.4.2 类成员上的断言 .....	39
16.4.3 lambda 表达式 .....	40
16.5 数值算法 .....	41
16.5.1 累积 .....	42
16.5.2 泛化 accumulate() .....	43
16.5.3 内积 .....	44
16.5.4 泛化 inner_product() .....	45
16.6 关联容器 .....	45
16.6.1 map .....	46
16.6.2 map 概览 .....	47
16.6.3 另一个 map 实例 .....	50
16.6.4 unordered_map .....	51
16.6.5 set .....	53
16.7 拷贝 .....	54
16.7.1 基本拷贝算法 .....	55
16.7.2 流迭代器 .....	55
16.7.3 使用 set 保持顺序 .....	57
16.7.4 copy_if .....	57
16.8 排序和搜索 .....	58
16.9 容器算法 .....	60
简单练习 .....	60
思考题 .....	61
术语 .....	62
习题 .....	62
附言 .....	63
<b>第 17 章 一个显示模型</b> .....	<b>64</b>
17.1 为什么要使用图形 .....	64

17.2 一个基本显示模型	65	18.16 Mark	111
17.3 第一个例子	66	18.17 Image	112
17.4 使用 GUI 库	68	简单练习	114
17.5 坐标系	69	思考题	115
17.6 Shape	70	术语	115
17.7 使用 Shape 类	70	习题	116
17.7.1 图形头文件和主函数	70	附言	116
17.7.2 一个几乎空白的窗口	71		
17.7.3 坐标轴	73	<b>第 19 章 设计图形类</b>	117
17.7.4 绘制函数图	74	19.1 设计原则	117
17.7.5 Polygon	75	19.1.1 类型	117
17.7.6 Rectangle	76	19.1.2 操作	118
17.7.7 填充	78	19.1.3 命名	119
17.7.8 Text	78	19.1.4 可变性	120
17.7.9 Image	80	19.2 Shape	121
17.7.10 更多未讨论的内容	81	19.2.1 一个抽象类	122
17.8 让图形程序运行起来	81	19.2.2 访问控制	123
17.8.1 源文件	82	19.2.3 绘制形状	125
简单练习	83	19.2.4 拷贝和可变性	127
思考题	83	19.3 基类和派生类	128
术语	83	19.3.1 对象布局	130
习题	84	19.3.2 类的派生和虚函数的定义	131
附言	84	19.3.3 覆盖	131
		19.3.4 访问	133
<b>第 18 章 图形类</b>	85	19.3.5 纯虚函数	134
18.1 图形类概览	85	19.4 面向对象程序设计的好处	135
18.2 Point 和 Line	87	简单练习	136
18.3 Lines	88	思考题	136
18.4 Color	91	术语	137
18.5 Line_style	93	习题	137
18.6 Open_polyline	95	附言	138
18.7 Closed_polyline	96		
18.8 Polygon	97	<b>第 20 章 绘制函数图和数据图</b>	139
18.9 Rectangle	99	20.1 简介	139
18.10 管理未命名对象	102	20.2 绘制简单函数图	139
18.11 Text	104	20.3 Function	143
18.12 Circle	105	20.3.1 默认参数	143
18.13 Ellipse	107	20.3.2 更多例子	144
18.14 Marked_polyline	108	20.3.3 lambda 表达式	146
18.15 Marks	110	20.4 Axis	146

20.5 近似	148	22.1.3 风格 / 范型	188
20.6 绘制数据图	152	22.2 程序设计语言历史概览	190
20.6.1 读取文件	153	22.2.1 最早的程序设计语言	191
20.6.2 一般布局	154	22.2.2 现代程序设计语言的起源	193
20.6.3 数据比例	155	22.2.3 Algol 家族	197
20.6.4 构造数据图	156	22.2.4 Simula	203
简单练习	158	22.2.5 C	204
思考题	159	22.2.6 C++	207
术语	159	22.2.7 今天	209
习题	159	22.2.8 参考资料	210
附言	160	思考题	211
<b>第 21 章 图形用户界面</b>	<b>161</b>	术语	212
21.1 用户界面的选择	161	习题	212
21.2 “Next” 按钮	162	附言	213
21.3 一个简单的窗口	163	<b>第 23 章 文本处理</b>	<b>214</b>
21.3.1 回调函数	164	23.1 文本	214
21.3.2 等待循环	166	23.2 字符串	214
21.3.3 lambda 表达式作为回调 函数	166	23.3 I/O 流	217
21.4 Button 和其他 Widget	167	23.4 映射	218
21.4.1 Widget	167	23.4.1 实现细节	222
21.4.2 Button	168	23.5 一个问题	224
21.4.3 In_box 和 Out_box	169	23.6 正则表达式的思想	225
21.4.4 Menu	170	23.6.1 原始字符串常量	227
21.5 一个实例	170	23.7 用正则表达式进行搜索	228
21.6 控制流反转	173	23.8 正则表达式语法	229
21.7 添加菜单	174	23.8.1 字符和特殊字符	230
21.8 调试 GUI 代码	178	23.8.2 字符集	230
简单练习	179	23.8.3 重复	231
思考题	179	23.8.4 子模式	232
术语	180	23.8.5 可选项	232
习题	180	23.8.6 字符集和范围	233
附言	181	23.8.7 正则表达式错误	234
<b>第 22 章 理念和历史</b>	<b>182</b>	23.9 使用正则表达式进行模式匹配	235
22.1 历史、理念和专业水平	182	23.10 参考文献	239
22.1.1 程序设计语言的目标和 哲学	182	简单练习	239
22.1.2 编程理念	183	思考题	239
		术语	240
		习题	240
		附言	241

<b>第 24 章 数值计算</b> .....	242	25.4.2 一个问题: 不正常的接口	280
24.1 简介 .....	242	25.4.3 解决方案: 接口类	282
24.2 大小、精度和溢出 .....	242	25.4.4 继承和容器	285
24.2.1 数值限制 .....	245	<b>25.5 位、字节和字</b> .....	287
24.3 数组 .....	245	25.5.1 位和位运算 .....	287
24.4 C 风格的多维数组 .....	246	25.5.2 bitset .....	290
24.5 Matrix 库 .....	247	25.5.3 有符号数和无符号数 .....	292
24.5.1 矩阵的维和矩阵访问 .....	248	25.5.4 位运算 .....	295
24.5.2 一维矩阵 .....	250	25.5.5 位域 .....	296
24.5.3 二维矩阵 .....	252	25.5.6 实例: 简单加密 .....	297
24.5.4 矩阵 I/O .....	253	<b>25.6 编码规范</b> .....	301
24.5.5 三维矩阵 .....	254	25.6.1 编码规范应该是怎样的 .....	302
24.6 实例: 求解线性方程组 .....	255	25.6.2 编码原则实例 .....	303
24.6.1 经典的高斯消去法 .....	256	25.6.3 实际编码规范 .....	307
24.6.2 选取主元 .....	257	<b>简单练习</b> .....	308
24.6.3 测试 .....	257	<b>思考题</b> .....	308
24.7 随机数 .....	258	<b>术语</b> .....	310
24.8 标准数学函数 .....	261	<b>习题</b> .....	310
24.9 复数 .....	262	<b>附言</b> .....	311
24.10 参考文献 .....	263	<b>第 26 章 测试</b> .....	312
简单练习 .....	264	26.1 我们想要什么 .....	312
思考题 .....	264	26.1.1 警告 .....	313
术语 .....	265	26.2 程序正确性证明 .....	313
习题 .....	265	26.3 测试 .....	313
附言 .....	266	26.3.1 回归测试 .....	314
<b>第 25 章 嵌入式系统程序设计</b> .....	267	26.3.2 单元测试 .....	315
25.1 嵌入式系统 .....	267	26.3.3 算法和非算法 .....	320
25.2 基本概念 .....	269	26.3.4 系统测试 .....	325
25.2.1 可预测性 .....	271	26.3.5 寻找不成立的假设 .....	326
25.2.2 理想 .....	272	26.4 测试方案设计 .....	327
25.2.3 生活在故障中 .....	272	26.5 调试 .....	328
25.3 内存管理 .....	274	26.6 性能 .....	328
25.3.1 动态内存分配存在的问题 .....	274	26.6.1 计时 .....	329
25.3.2 动态内存分配的替代方法 .....	276	26.7 参考文献 .....	331
25.3.3 存储池实例 .....	277	<b>简单练习</b> .....	331
25.3.4 栈实例 .....	278	<b>思考题</b> .....	331
25.4 地址、指针和数组 .....	279	<b>术语</b> .....	332
25.4.1 未经检查的类型转换 .....	279	<b>习题</b> .....	332
		<b>附言</b> .....	333

第 27 章 C 语言 .....	334	27.5.4 一个风格问题 .....	353
27.1 C 和 C++: 兄弟 .....	334	27.6 输入/输出: stdio .....	354
27.1.1 C/C++ 兼容性 .....	335	27.6.1 输出 .....	354
27.1.2 C 不支持的 C++ 特性 .....	336	27.6.2 输入 .....	355
27.1.3 C 标准库 .....	338	27.6.3 文件 .....	356
27.2 函数 .....	338	27.7 常量和宏 .....	356
27.2.1 不支持函数名重载 .....	338	27.8 宏 .....	357
27.2.2 函数参数类型检查 .....	339	27.8.1 类函数宏 .....	358
27.2.3 函数定义 .....	340	27.8.2 语法宏 .....	359
27.2.4 C++ 调用 C 和 C 调用 C++ .....	341	27.8.3 条件编译 .....	360
27.2.5 函数指针 .....	343	27.9 实例: 侵入式容器 .....	360
27.3 小的语言差异 .....	344	简单练习 .....	365
27.3.1 struct 标签名字空间 .....	344	思考题 .....	365
27.3.2 关键字 .....	345	术语 .....	366
27.3.3 定义 .....	345	习题 .....	366
27.3.4 C 风格类型转换 .....	347	附言 .....	367
27.3.5 无类型指针的转换 .....	347	附录 C 标准库概要 .....	368
27.3.6 枚举 .....	348	附录 D 安装 FLTK .....	410
27.3.7 名字空间 .....	348	附录 E GUI 实现 .....	413
27.4 自由存储空间 .....	349	术语表 .....	419
27.5 C 风格字符串 .....	350	参考文献 .....	423
27.5.1 C 风格字符串和 const .....	352		
27.5.2 字节操作 .....	352		
27.5.3 实例: strcpy() .....	353		

# 容器和迭代器

只做一件事，并把它做好。多个程序协同工作。

——Doug McIlroy

本章和下一章将分别介绍 C++ 标准库 (STL) 中的容器和算法部分。STL 是一个用于处理 C++ 程序中数据的可扩展框架。我们首先通过一个简单的例子来说明 STL 的设计理念和基本概念，然后详细讨论迭代器、链表和 STL 中的容器。STL 通过序列 (sequence) 和迭代器 (iterator) 的概念将容器 (数据) 和算法 (处理) 关联起来。本章的内容为下一章介绍通用和高效的算法奠定了基础。作为示例，本章实现了一个文字编辑器的基本框架。

## 15.1 存储和处理数据

在处理数据量很大的问题之前，我们先来看一个简单的例子，它说明了解决一般数据处理问题的基本方法。Jack 和 Jill 分别负责测量来往车辆的速度，结果用浮点数来表示。Jack 是一个 C 语言的程序员，所以将测量值保存到一个数组中，而 Jill 将测量值保存到一个 vector 对象中。如果我们要在程序中使用他们的数据，该如何操作呢？

我们可以让 Jack 和 Jill 的程序将结果分别写到某个文件中，然后再从文件中读入数据。使用这种方法，我们的程序将与 Jack 和 Jill 所选用的数据结构和接口彻底无关。通常，这种程序之间的独立性是一种很好的特性，此时我们可以采用第 10 和 11 章中介绍的方法来获得输入数据，并利用 `vector<double>` 对象来进行计算。

但是，如果我们的任务不适合使用文件呢？假设我们必须每秒钟调用一次数据生成函数来获得一组新的数据。例如，下面的程序每秒都会调用 Jack 和 Jill 的函数来获得将要处理的数据：

```
double* get_from_jack(int* count); // Jack 将 double 值存入一个数组并将元素个数藉由 *count 返回
vector<double>* get_from_jill();   // Jill 填充 vector
```

```
void fct()
{
    int jack_count = 0;
    double* jack_data = get_from_jack(&jack_count);
    vector<double>* jill_data = get_from_jill();
    // ... 处理 ...
    delete[] jack_data;
    delete jill_data;
}
```

上面这段代码假设我们要自己安排存储数据的空间，而且在用完这些数据之后要自己负责删除。另一个假设是我们不能重写 Jack 和 Jill 的代码，而且通常我们也不想这样做。

### 15.1.1 处理数据

显然，这个例子过于简单，但是它与很多实际问题并没有本质区别。如果我们能够很好

地解决这个例子，就能够处理一大类通用的编程问题。问题的关键在于我们无法控制提供数据的程序以什么形式来存储数据。我们可以自由决定是沿用原有的数据格式，还是转换为另一种形式来进行存储和处理。

我们想要如何处理数据？排序？找出最大值？找出平均值？找出大于 65 的值？比较 Jill 和 Jack 的数据？处理需求多种多样，我们只能根据具体任务来编写处理程序。这里，我们主要是学习怎样处理数据，完成大量数据的计算。首先从简单的处理开始：找到数据集中的最大值。我们可以将 `fct()` 函数中内容为“…处理…”的注释行替换为下面这段代码：

```
// ...
double h = -1;
double* jack_high; // jack_high 将指向值最大的元素
double* jill_high; // jill_high 将指向值最大的元素
for (int i=0; i<jack_count; ++i)
    if (h<jack_data[i]) {
        jack_high = &jack_data[i]; // 保存最大元素的地址
        h = jack_data[i];         // 更新“最大元素”
    }

h = -1;
for (int i=0; i<jill_data->size(); ++i)
    if (h<(*jill_data)[i]) {
        jill_high = &(*jill_data)[i]; // 保存最大元素的地址
        h = (*jill_data)[i];         // 更新“最大元素”
    }
cout << "Jill's max: " << *jill_high
    << "; Jack's max: " << *jack_high;

// ...
```

注意访问 Jill 数据时使用的语法 `(*jill_data)[i]`。`get_from_jill()` 函数返回一个指向 `vector` 对象的指针，即 `vector<double>*`。为了获得数据内容，我们首先要解引用指针以获得 `vector`——`*jill_data`，然后对其使用下标操作。然而，`*jill_data[i]` 并不是我们想要的结果，因为运算符 `[]` 的优先级要高于运算符 `*`，所以这个表达式的含义是 `*(jill_data[i])`，必须在 `*jill_data` 外使用括号，结果即为 `(*jill_data)[i]`。



### 试一试

如果可以修改 Jill 的代码，应该如何修改代码的接口来避免复杂的数据访问方法？

## 15.1.2 泛化代码



我们希望使用统一的方法来访问和处理数据，这样可以避免因为每次获得的数据格式不同而编写不同的处理代码。下面我们以 Jack 和 Jill 的代码为例，讨论如何让我们的代码更通用、更统一。

显然，我们对 Jack 和 Jill 的数据的处理方法很相似。但是两段代码有一些恼人的差异：`jack_count` 和 `jill_data->size()`，`jack_data[i]` 和 `(*jill_data)[i]`。我们可以通过使用引用来避免第二个不同之处：

```
vector<double>& v = *jill_data;
for (int i=0; i<v.size(); ++i)
```

```
    if (h<v[i]) {
        jill_high = &v[i];
        h = v[i];
    }
```

这段代码已经非常接近处理 Jack 数据的代码了。接下来如何编写一个可以同时处理 Jack 和 Jill 数据的函数呢？方法有很多（参考习题 3），出于通用性的考虑（这一点在接下来的两章中十分明显），我们选择下面这种基于指针的方法：

```
double* high(double* first, double* last)
// 返回一个指针，指向 [first,last) 中值最大的元素
{
    double h = -1;
    double* high;
    for(double* p = first; p!=last; ++p)
        if (h<*p) { high = p; h = *p; }
    return high;
}
```

使用这个函数，数据处理代码可以改写为：

```
double* jack_high = high(jack_data,jack_data+jack_count);
vector<double>& v = *jill_data;
double* jill_high = high(&v[0],&v[0]+v.size());
```

这段代码更加简洁：不仅省去了很多变量的定义，并且只出现了一段循环代码（在 high() 中）。如果我们想要得到最大值，只需查看 \*jack\_high 和 \*jill\_high，例如：

```
cout << "Jill's max: " << *jill_high
    << "; Jack's max: " << *jack_high;
```

注意，high() 函数要求所处理的数据保存在一个数组中，所以“找出最大值”的算法返回的是指向数组元素的指针。

## 试一试

这段程序中有两个潜在的严重错误。其中一个会导致程序崩溃，另一个会导致 high() 函数返回错误的结果。下面将要介绍的通用技术会充分暴露出这两个错误，并给出系统的避免方法。现在我们只需要找出这两个错误，并提出修改意见。

high() 函数的局限性在于只能处理某个特定的问题：

- 只能处理数组。vector 的元素必须保存在数组中，但实际上数据的存储方式还有可能是 list 和 map（见 15.4 节和 15.6.1 节）。
- 可以处理 double 类型的 vector 或数组，但是无法处理其他类型的元素，例如 vector<double\*> 或 char[10]。
- 只能找出最大值，无法完成其他简单的数据计算功能。

下面，我们探讨如何在更通用的数据集合上进行计算。

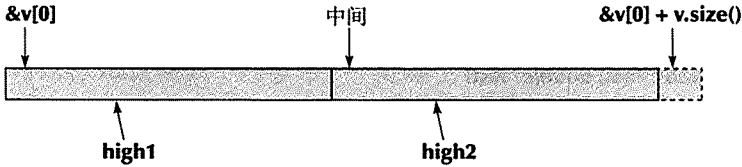
通过指针的方式来实现“找出最大值”的算法会带来一个意想不到的通用性：我们不仅可以找出整个数组或 vector 中的最大值，还可以找出数组或 vector 的某个部分的最大值，例如：

```
// ...
vector<double>& v = *jill_data;
double* middle = &v[0]+v.size()/2;
```



```
double* high1 = high(&v[0], middle); // 前一半的最大值
double* high2 = high(middle, &v[0]+v.size()); // 后一半的最大值
// ...
```

这里 `high1` 指向 `vecotr` 中前半部分的最大值，`high2` 指向 `vecotr` 中后半部分的最大值。下面是这个结果的图示：



`high()` 函数的参数是指针，这样的代码偏于底层，更容易引起错误。我们怀疑对于大多数程序员来说，找出 `vector` 中最大值的代码显然应像下面这样：

```
double* find_highest(vector<double>& v)
{
    double h = -1;
    double* high = 0;
    for (int i=0; i<v.size(); ++i)
        if (h<v[i]) { high = &v[i]; h = v[i]; }
    return high;
}
```

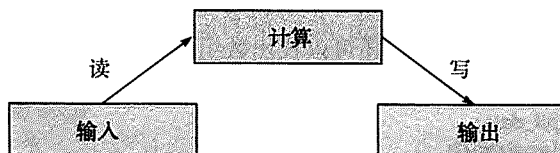
然而，这段代码失去了我们“偶然”从 `high()` 所获得的灵活性——我们不能用 `find_highest()` 来查找 `vector` 某一部分中的最大值。我们实际上只是为了同时处理数组和 `vector` 才决定“摆弄指针”，但却意外地获得了某种灵活性。应该记住：代码泛化可以获得适用于多个问题的通用函数。

## 15.2 STL 理念

C++ 标准库为处理数据序列提供了一个专门的框架，称为 STL。STL 是标准模板库 (Standard Template Library) 的简称。STL 是 ISO C++ 标准库的部分，它提供了容器（例如 `vector`、`list` 和 `map`）和通用算法（例如 `sort`、`find` 和 `accumulate`）。因此我们可以称 `vector` 这类对象为 STL 或标准库的一部分。标准库的其他部分，例如 `ostream`（第 10 章）和 C 风格的字符串处理函数（附录 C.10.3），并不属于 STL。为了更好地理解 STL，我们首先考虑在处理数据时必须解决的问题和对求解方案的理念。



计算过程包含两个主要方面：计算和数据。有时我们只关注计算方面，谈论 `if` 语句、循环、函数和错误处理等。有时我们关注的是数据，谈论数组、向量、字符串和文件等。然而，要完成真正的工作我们需要同时考虑计算和数据。如果对大量数据不进行分析、可视化和查找“感兴趣的部分”，则数据是无法理解的。相反，我们可以随心所欲地进行计算，但是只有与实际数据关联之后，才能避免枯燥乏味的无意义计算。而且，“计算部分”要优雅地与“数据部分”进行交互。



当谈起数据时，我们会想到很多类型的数据：数十个形状、数百个温度值、数千个日志记录、数百万个点、数十亿个网页等，即我们在讨论数据的容器、数据流等。特别地，我们并不是要讨论如何为一个对象（例如，一个复数、一个温度读数或一个圆形等）选择最恰当的数值。对于这些类型，可以参看第 9、11 和 19 章。

考虑我们需要对“大量数据”进行的一些简单操作：

- 按照字典序排序。
- 根据姓名在电话本中找到对应的电话号码。
- 找出温度的最高值。
- 找出所有大于 8800 的值。
- 找出第一个值为 17 的元素。
- 根据单元编号对遥测记录进行排序。
- 根据时间戳对遥测记录进行排序。
- 找出第一个值大于“Petersen”的元素。
- 找出最大值。
- 找出两个序列的第一个不同之处。
- 计算两个序列中对位元素两两之积。
- 找出一个月中每天的最高气温。
- 在销售记录中找出最畅销的 10 件商品。
- 统计“Stroustrup”在网页中出现的次数。
- 计算各个元素之和。

注意，我们在讨论上述数据处理任务时，并没有提到数据如何存储。很显然，我们必须使用链表、向量、文件和输入流等来完成这些任务，但是我们不必了解这些数据存储（或收集）的细节，即可讨论如何对它们进行处理。重要的是这些值或对象的类型（元素类型），我们如何访问这些值或对象，以及要对它们进行什么操作。

这些任务非常常见，我们自然希望能编写代码简单高效地完成这些任务。与之相对，我们需要考虑的问题是：

- 数据类型（“数据种类”）变化万千。
- 数据集存储方法多到让人眼花缭乱。
- 我们想对数据集执行的任务也数量繁多。

为了尽可能降低这些问题的影响，我们希望编写的代码能够处理各种数据类型，处理各种数据存储方法，适用于各种处理任务。换句话说，我们想通过泛化代码来适应各种变化。我们要避免对每一个问题都从头开始寻找处理方法，那样会浪费大量的时间。

为了编写能够达到上述目的的代码，首先用更加抽象的方式来看待我们要对数据进行的工作：

- 收集数据并装入容器。
  - 例如 `vector`、`list` 和数组。
- 组织数据。
  - 用于打印；
  - 用于快速访问。
- 提取数据。

- 通过索引 (例如, 第 42 个元素);
- 通过值 (例如, 年龄字段是 7 的第一条记录);
- 根据属性 (例如, 所有温度字段大于 32 小于 100 的记录)。
- 修改容器。
  - 增加数据;
  - 减少数据;
  - 排序 (根据某种标准)。
- 进行简单的数值运算 (例如, 将每一个元素乘以 1.7)。

我们在完成上述任务时要避免陷入各种细节之中: 各种容器间的差别、各种数据元素访问方法间的差别以及各种数据类型间的差别。如果我们能够做到这一点, 就等于向编写简单高效的通用代码的目标迈出了一大步。

回顾前面几章介绍的编程工具和技术, 我们 (已经) 可以编写功能相似但与数据类型无关的代码:

- 使用 `int` 与使用 `double` 基本没有差异。
- 使用 `vector<int>` 与使用 `vector<string>` 基本没有差异。
- 使用 `double` 类型的数组与使用 `vector<double>` 基本没有差异。



我们希望通过合理组织代码, 实现只有当我们想要完成一些全新的任务时才需要编写新的代码。特别是, 我们希望编写一些完成基本任务的代码, 使得我们不必每次发现一种新的数据存储方式或数据解释方式时都重写整个程序。

- 在 `vector` 中查找一个值与在数组中查找一个值差异不大。
- 查找 `string` 时不区分大小写与区分大小写的差异不大。
- 绘制实验数据图时使用准确值与使用四舍五入值的差异不大。
- 拷贝文件与拷贝 `vector` 对象的差异不大。

根据上面的发现, 我们希望编写的代码具有以下特点:

- 容易阅读。
- 容易修改。
- 规范。
- 简短。
- 快速。

为了简化编程工作, 我们会:

- 使用统一的方式访问数据。
  - 与数据存储方法无关;
  - 与数据类型无关。
- 使用类型安全的方式访问数据。
- 便于遍历数据。
- 紧凑存储数据。
- 快速
  - 读取数据;
  - 增加数据;
  - 删除数据。



- 对通用算法提供标准实现。
  - 例如拷贝、查找、搜索、排序、求和等。

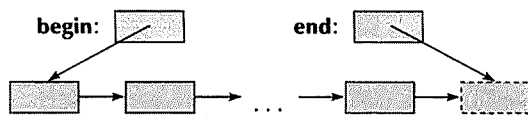
STL 提供了上述功能以及更多。我们不应仅仅将它看作一个强大的功能库，更应看作一个兼顾灵活性与性能的函数库设计的典范。STL 由 Alex Stepanov 设计，提供了一个作用于数据结构之上的通用、正确和高效的算法框架。其设计理念满足简单性、通用性和数学上的优雅。

除了使用设计理念和原则清晰的框架来处理数据之外，另一种策略是让程序员使用最基本的语言特性从头开始编写每个程序，并采用一些当时看起来不错的思想。这种方式费时费力，而且得到的程序代码往往非常糟糕，毫无章法可言，除作者之外的人很难理解，而且在其他场合能够复用的可能性微乎其微。 ⚠

在介绍完 STL 的设计初衷和理念之后，我们会给出 STL 的一些基本定义，最后通过例子展示如何在实际应用中运用这些基本理念：编写更好的处理数据的代码，而且让编写过程更简单。

### 15.3 序列和迭代器

序列是 STL 中的核心概念。从 STL 的角度来看，数据集合就是一个序列。序列具有头部和尾部。我们可以对一个序列从头到尾进行遍历，对序列中的元素进行有选择的读写操作。我们利用一对迭代器来表示序列头部和尾部。迭代器 (iterator) 是一种可以标识序列中元素的对象。我们可以按照如下方式来看待一个序列： ✂



这里的 `begin` 与 `end` 就是迭代器，它们标识了序列的头部和尾部。我们通常称 STL 的序列是“半开”的，因为由 `begin` 所标识的元素是序列的一部分，而迭代器 `end` 通常指向序列尾部之后的一个位置。在数学中，这种序列 (区间) 可以表示为 `[begin: end)`。两个元素间的箭头表示如果有一个指向第一个元素的迭代器，那么我们就可以得到一个指向第二个元素的迭代器。

那么究竟什么是迭代器呢？迭代器是一个相当抽象的概念：

- 迭代器指向序列中的某个元素 (或者序列末端元素之后)。
- 可以使用 `==` 和 `!=` 来对两个迭代器进行比较。
- 可以使用单目运算符 `*` 来访问迭代器所指向的元素。
- 可以利用操作符 `++` 来令迭代器指向下一个元素。


例如，如果 `p` 和 `q` 是两个指向同一个序列的迭代器： ✂

#### 标准迭代器的基本操作

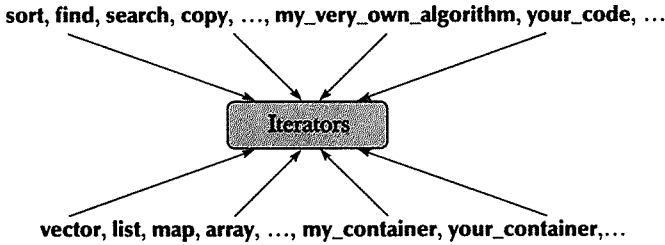
<code>p==q</code>	当且仅当 <code>p</code> 和 <code>q</code> 指向序列中的同一个元素或都指向序列末端元素之后为真
<code>p!=q</code>	<code>!(p==q)</code>
<code>*p</code>	表示 <code>p</code> 所指向的元素
<code>*p=val</code>	对 <code>p</code> 所指向的元素进行写操作
<code>val=*p</code>	对 <code>p</code> 所指向的元素进行读操作
<code>++p</code>	使 <code>p</code> 指向序列中的下一个元素或序列末端元素之后

很明显，迭代器的概念与指针（见 12.4 节）相关。实际上，指向数组中某一元素的指针就是一个迭代器。但是，许多迭代器不仅仅是指针。比如说，我们可以定义一个边界检查的迭代器，当试图使它指向 `[begin: end)` 之外时抛出一个异常。把迭代器作为一种抽象的概念而不是类型可以给我们带来很大的灵活性和通用性。本章和下一章将会举例说明这一点。

### 试一试

 编写一个函数 `void copy(int* f1, int* e1, int* f2)`，该函数把 `[f1: e1)` 定义的 `int` 型数组复制到数组 `[f2: f2+(e1-f1))` 中。注意只能使用上面所提到的迭代器操作（不能用索引）。

我们可以利用迭代器来实现代码（算法）与数据的连接。程序编写人员了解迭代器的使用方法（并不需要了解迭代器实际是如何访问数据的），数据提供者向用户提供相应的迭代器而不是数据存储的细节信息。这样得到的是一个简单的程序结构，而且使算法和容器之间保持了很好的独立性。正如 Alex Stepanov 所说：“STL 算法和容器可以共同完成很强大的功能，而这却是因为它们根本不知道对方的存在。”但是，它们都知道由一对迭代器所定义的序列。



换句话说，我们的代码不再需要知道存储和访问数据的不同方法，它只需要对迭代器有一定的了解。另一方面，作为数据提供方，我们不再需要为不同的用户分别编写代码，而只需要为我们的数据配备好合适的迭代器。实际上，最简单的迭代器只是由 `*`、`++`、`==`、`!=` 等操作所定义的，这无疑会令它既方便又快速。

STL 框架包含大概 10 种容器和 60 种由迭代器相连接的算法（参见第 16 章）。另外，许多组织和个人都在开发符合 STL 风格的容器和算法。STL 可能是目前最著名的泛型编程的例子（见 14.3.2 节）。只要你了解了它的基本概念和一些简单的例子，就可以很容易掌握其他相关的内容。

### 15.3.1 回到实例

下面我们来看看如何使用 STL 来描述问题“查找序列中的最大元素”：

```
template<typename Iterator>
Iterator high(Iterator first, Iterator last)
    // 返回指向 [first:last) 中最大元素的迭代器
{
    Iterator high = first;
    for (Iterator p = first; p!=last; ++p)
        if (*high<*p) high = p;
    return high;
}
```

注意我们去掉了用来存储当前最大元素的变量 `h`。当我们不能确定序列中元素的类型时，使用 `-1` 来完成初始化看起来非常随意和奇怪。这是因为它确实非常随意和奇怪！其中还隐藏着潜在错误：在我们的例子中 `-1` 能奏效仅仅是因为恰好不会存在负的速度。我们要记住像 `-1` 这样的“魔数”是非常不利于程序的维护的（见 4.3.1 节、7.6.1 节、10.11.1 节等）。在本例中，我们可以看到它还会限制函数的用途，并意味着我们对问题求解方案还没有形成一个比较全面的认识，也就是说，“魔数”是一种偷懒的表现。

注意这里的 `high()` 可以被用于所有可以使用 `<` 进行比较的元素类型。比如说，我们可以利用 `high()` 来查找 `vector<string>` 中按字典序最靠后的字符串（见习题 7）。

`high()` 模板函数可以用于任何由一对迭代器定义的序列。举例来说，我们可以严格复制例程：

```
double* get_from_jack(int* count); // Jack 将 double 值保存在数组中，并藉由 *count 返回元素个数
vector<double>* get_from_jill(); // Jill 填充 vector

void fct()
{
    int jack_count = 0;
    double* jack_data = get_from_jack(&jack_count);
    vector<double>* jill_data = get_from_jill();

    double* jack_high = high(jack_data, jack_data + jack_count);
    vector<double>& v = *jill_data;
    double* jill_high = high(&v[0], &v[0] + v.size());

    cout << "Jill's high " << *jill_high << "; Jack's high " << *jack_high;
    // ...
    delete[] jack_data;
    delete jill_data;
}
```

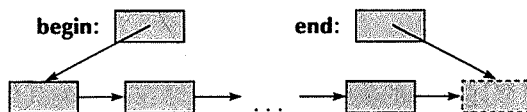
这里的两个调用中，`high()` 的 `Iterator` 模板参数的类型为 `double*`。除了（最终）确保 `high()` 被正确实现外，这和我们之前的例子没有任何区别。更准确地说，所运行的代码并没有什么不同，但在代码的通用性上却有很大的区别。`high()` 的模板版本适用于任何由一对迭代器所定义的序列。在进一步了解 STL 规范细节和所提供的能免除我们编写常见繁琐代码之苦的算法之前，我们先来集中了解数据元素集合的存储方法。

### 🔧 试一试

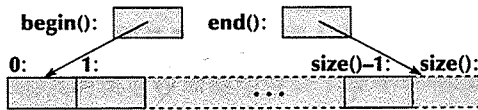
在我们的程序中有一个严重的错误。请找到并修改它，提出一种针对这种问题的通用解决方法。

## 15.4 链表

下面让我们再回顾一下序列概念的图形表示：



将它与我们描绘 `vector` 内存结构的示意图相比较：



下标 0 本质上与迭代器 `v.begin()` 一样都指向同一个元素，并且下标 `v.size()` 与 `v.end()` 一样都指向最后一个元素之后的位置。

`vector` 的元素在内存中是连续存储的。这并非 STL 序列概念所要求的特性，因此在 STL 中，很多算法在将一个元素插入两个已有元素的中间时都不需要移动已有的元素。上面序列抽象概念的图意味着，在不移动其他元素的前提下进行元素插入（或元素删除）操作是可能的。STL 迭代器概念支持上述操作。

直接体现上述 STL 序列概念的数据结构是链表（linked list）。在抽象模型中的箭头通常由指针实现。链表中的一个元素是“链接”的一部分，一个“链接”由这一元素以及一个或多个指针组成。如果链表的一个链接只包含一个指针（指向下一个链接），我们称这样的链表为单向链表，如果一个链接包含一个指向前驱链接的指针以及一个指向后继链接的指针，则这样的链表为双向链表。在后续小节中，我们将勾勒一个双向链表的实现，且该实现与 C++ 标准库 `list` 的实现相同。双向链表的概念可图示如下：

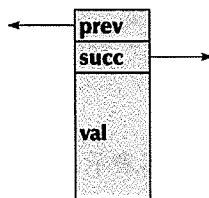


上述概念可由代码实现：

```
template<typename Elem>
struct Link {
    Link* prev;    // 前趋链接
    Link* succ;    // 后继（下一个）链接
    Elem val;      // 值
};

template<typename Elem> struct list {
    Link<Elem>* first;
    Link<Elem>* last; // 最后一个链接之后的位置
};
```

一个 `Link` 的内存布局如下所示：



链表的实现方法和呈现给用户的方法有多种。附录 C 中给出了标准库所采用的一种方法。在本节中，我们将只概述链表的关键属性——能够在不影响其他已有元素的前提下插入

和删除元素，展示如何遍历一个链表，以及给出链表使用的一个示例。

当你考虑使用链表时，我们强烈建议你将自己所考虑的链表操作绘图表示。图形是描绘链表操作的一种十分有效的方法。

### 15.4.1 链表操作

对于链表，我们需要使用哪些操作呢？

- 对 `vector` 所实现的操作（构造函数、大小等），除了下标。
- 插入（添加一个元素）和删除（移除一个元素）。
- 访问元素以及遍历链表：迭代器。

在 STL 中，上述的迭代器类型是 `list` 类的一个成员，因此我们也会这样设计：

```
template<typename Elem>
class list {
    // 表示和实现细节
public:
    class iterator;           // 成员类型：迭代器

    iterator begin();        // 指向首元素的迭代器
    iterator end();          // 指向尾元素之后的迭代器

    iterator insert(iterator p, const Elem& v); // 将 v 插入链表中 p 之后的位置
    iterator erase(iterator p);                // 从链表中删除 p

    void push_back(const Elem& v);             // 将 v 插入链表末尾
    void push_front(const Elem& v);           // 将 v 插入链表头
    void pop_front();                          // 删除首元素
    void pop_back();                           // 删除尾元素

    Elem& front();                             // 获取首元素
    Elem& back();                              // 获取尾元素

    // ...
};
```

就像“我们的” `vector` 并没有完全实现标准库 `vector` 一样，上述 `list` 定义与标准库 `list` 定义也不完全相同。但上述 `list` 中没有任何错误，它仅仅是不完全而已。“我们的” `list` 的目的在于加深你对链表的理解：链表是什么，`list` 应该如何实现，以及如何使用 `list` 的关键特性。如果读者想获得更多的信息，请参考附录 C 或其他专家级别的 C++ 书籍。

迭代器是 STL `list` 定义中的核心部分。迭代器被用于标示元素插入的位置以及待删除（擦除）的元素。它们也可被用于在链表中进行“导航”。迭代器的这一用途与我们在 15.1 节和 15.3.1 节中使用指针遍历数组和向量十分相似。迭代器的这一风格对于标准库算法而言十分关键（见 16.1 ~ 16.3 节）。

为什么不在 `list` 中使用下标操作呢？我们可以为 `list` 实现下标操作，但它会是一种极为缓慢的操作：`list[1000]` 操作将会从第一个元素开始访问每个元素，直到访问到第 1000 个元素为止。如果我们希望这么做，那么可以自己实现这一操作（或使用 `advance()`，参见 15.6.2 节）。因此，标准库 `list` 并没有提供下标语法。

我们将迭代器的类型作为 `list` 的成员（一个嵌套类）的原因在于，我们没有任何理由将迭代器的类型实现为全局类。这一迭代器的类型将只会由 `list` 类使用。另外，这也使



得我们能够将每一容器的迭代器都命名为 `iterator`。在标准库中存在着 `list<T>::iterator`、`vector<T>::iterator`、`map<K,V>::iterator` 等迭代器类型。

## 15.4.2 遍历

`list` 迭代器必须提供 `*`、`++`、`==` 和 `!=` 操作。因为标准库中的链表为双向链表，因此该链表还提供了 `--` 操作，以实现链表的“从后”向前的遍历操作：

```
template<typename Elem> // 要求 Element<Elem>() (见 14.3.3 节)
class list<Elem>::iterator {
    Link<Elem>* curr; // 当前链接
public:
    iterator(Link<Elem>* p) : curr(p) {}

    iterator& operator++() { curr = curr->succ; return *this; } // 前向
    iterator& operator--() { curr = curr->prev; return *this; } // 反向
    Elem& operator*() { return curr->val; } // 获得值 (解引用)

    bool operator==(const iterator& b) const { return curr==b.curr; }
    bool operator!=(const iterator& b) const { return curr!=b.curr; }
};
```

这些函数十分简明且极具效率：函数实现中不存在循环，不存在复杂的表达式，不存在“可疑的”函数调用。如果你还不清楚这些实现的意义，请再快速回顾一下前面的示意图。这一 `list` 迭代器只是一个指向链接的指针。注意，即使 `list<Elem>::iterator` 的实现（代码）与我们在 `vector` 和数组中用作迭代器的简单指针的实现极为不同，两者操作的意义（语义）是相同的。基本上，`list` 迭代器提供了对 `Link` 指针的 `++`、`--`、`*`、`==` 和 `!=` 操作。

现在让我们再次回顾 `high()` 的实现：

```
template<typename Iter> // 要求 Input_iterator<Iter>() (见 14.3.3 节)
Iter high(Iter first, Iter last)
// 返回一个迭代器，指向 [first, last] 中的最大值元素
{
    Iter high = first;
    for (Iter p = first; p!=last; ++p)
        if (*high<*p) high = p;
    return high;
}
```

我们可以将其用于 `list`：

```
void f()
{
    list<int> lst; for (int x; cin >> x; ) lst.push_front(x);

    list<int>::iterator p = high(lst.begin(), lst.end());
    cout << "the highest value was " << *p << "\n";
}
```

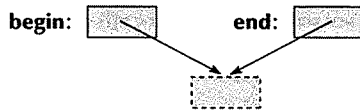
在上述代码中，`Iter` 参数的“取值”为 `list<int>::iterator`，并且 `++`、`*` 和 `!=` 操作的实现都与数组的代码有很大不同，但操作的意义是相同的。模板函数 `high()` 仍然遍历数据（在这里是 `list`）和查找最大取值。我们可以在 `list` 的任何位置插入一个元素，因此使用了 `push_front()` 在链表首部添加元素，而这一操作的目的是为了显示我们确实能够这么做。当然，我们也可以像对 `vector` 一样对 `list` 使用 `push_back()` 函数。

## 👉 试一试

标准库 `vector` 不提供 `push_front()`。为什么？为 `vector` 实现 `push_front()` 并将其与 `push_back()` 进行比较。

现在，是时候提出这样的问题了：“如果 `list` 为空会怎样？”换句话说，“如果 `lst.begin() == lst.end()` 会怎样？”在这种情况下，`*p` 将会试图对最后一个元素 `lst.end()` 之后的位置进行解引用，这是一个灾难！或者——可能更糟地——结果可能是一个错误的随机值。

此问题的最后一种描述形式给我们带来了一个提示：可以通过比较 `begin()` 和 `end()` 测试一个链表是否为空——实际上，可以通过比较序列的开始和结束判断任何 STL 序列是否为空：



这是令序列的 `end` 指向最后一个元素之后的位置而不是指向最后一个元素的一个更深层次的原因：空序列不再是一种特殊情况。我们不喜欢特殊情况，因为——根据定义——我们不得不为这些特殊情况编写特殊的代码。

在我们的例子中，可以按如下方式对 `list` 进行测试：

```
list<int>::iterator p = high(lst.begin(), lst.end());
if (p==lst.end()) // 我们到达链表尾了吗？
    cout << "The list is empty";
else
    cout << "the highest value is " << *p << '\n';
```

我们采用这种形式的测试方法系统地测试 STL 算法。

因为标准库提供了链表，我们在这里不再继续深入探讨它的具体实现。取而代之的是，我们将简要讨论链表适用的场合（如果你对链表的实现细节感兴趣，参考习题 12 ~ 14）。

## 15.5 再次泛化 `vector`

显然，通过 15.3 ~ 15.4 节的例子我们发现，标准库 `vector` 包含一个 `iterator` 成员类型，以及 `begin()` 和 `end()` 成员函数（与 `std::list` 类似）。然而，我们并没有在第 14 章中为 `vector` 类提供这些成员。那么，对于不同类型的容器而言，它们究竟采用了什么方法，以使它们或多或少地能够在 15.3 节所介绍的 STL 泛型编程风格中相互替换使用？首先，我们将简要介绍一种解决方案（简单起见，我们忽略了分配器），然后再对解决方案进行解释：

```
template<typename T> // 要求 Element<T>() (见 14.3.3 节)
class vector {
public:
    using size_type = unsigned long;
    using value_type = T;
    using iterator = T*;
    using const_iterator = const T*;

    // ...
```

```

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

size_type size();

// ...
};

```



using 声明为一个类型创建别名，即对于我们的 vector，iterator 是我们用作迭代器类型 T\* 的一个同义词，它的另一个名字。现在，对于 vector 对象 v，我们可以编写如下代码：

```
vector<int>::iterator p = find(v.begin(), v.end(), 32);
```

以及

```
for (vector<int>::size_type i = 0; i < v.size(); ++i) cout << v[i] << '\n';
```

通过别名的方式，我们事实上不需要知道 iterator 和 size\_type 的实际类型。特别地，由于使用了 iterator 和 size\_type，上述代码也可以用于那些 size\_type 不是 unsigned long 类型（在很多嵌入式系统中，size\_type 为其他类型）并且 iterator 为类而不是简单指针（这种情况在 C++ 实现中很普遍）的 vector。

标准库以相似的方式定义了 list 和其他标准容器。例如：

```

template<typename T>          // 要求 Element<T>() (见 14.3.3 节)
class list {
public:
    class Link;
    using size_type = unsigned long;
    using value_type = T;
    class iterator;          // 参见 15.4.2 节
    class const_iterator;    // 类似迭代器，但不允许向元素写入值

    // ...

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    size_type size();

    // ...
};

```

这样，我们编写代码时就不必操心使用的是 list 还是 vector。所有标准库算法中都使用了上述这些容器的成员类型名，例如 iterator 和 size\_type，所以，算法的实现不依赖于容器的实现或者具体操作的是什么容器（参见第 16 章）。

还有一种方法可以替代对容器 C 使用 C::iterator——Iterator<C>，这也是我们通常更倾向使用的方法。通过一个简单的模板别名即可实现这种用法：

```

template<typename C>
using Iterator = typename C::iterator;    // Iterator<C> 表示类型名 C::iterator

```

出于语言技术层面的原因，我们需要为 C::iterator 加上 typename 前缀，以表明 iterator

是一个类型，这也是我们更倾向于使用 `Iterator<C>` 的原因之一。类似地，我们可以定义

```
template<typename C>
using Value_type = typename C::value_type;
```

这样，我们就可以在代码中使用 `Value_type<C>` 了。这些类型别名不包含在标准库中，但你可以从 `std_lib_facilities.h` 中找到它们。

`using` 声明是 C++11 的新特性，与 C 和 C++ 中为人熟知的 `typedef`（见附录 A.16）相似，可以看作后者的泛化。

### 15.5.1 遍历容器


使用 `size()`，我们可以从头到尾遍历一个 `vector`。例如：

```
void print1(const vector<double>& v)
{
    for (int i = 0; i < v.size(); ++i)
        cout << v[i] << '\n';
}
```

这段代码不适用于链表，因为 `list` 不提供下标操作。但是，我们可以用一个简单的范围 `for` 循环（见 4.6.1 节）来遍历标准库 `vector` 和 `list`。例如：

```
void print2(const vector<double>& v, const list<double>& lst)
{
    for (double x : v)
        cout << x << '\n';

    for (double x : lst)
        cout << x << '\n';
}
```

这段代码既适用于标准库容器，也适用于“我们的”`vector` 和 `list`。它是如何办到的？“窍门” 在于范围 `for` 循环是基于 `begin()` 和 `end()` 函数的，前者返回指向我们的 `vector` 的首元素的迭代器，后者返回指向尾元素之后位置的迭代器。范围 `for` 循环其实不过是使用迭代器遍历序列的循环的一种“语法糖衣”而已。如果我们为自己的 `vector` 和 `list` 定义了 `begin()` 和 `end()`，就“偶然地”提供了范围 `-for` 所需要的东西。

### 15.5.2 auto

当我们不得不编写循环遍历一个通用结构时，命名迭代器是很令人厌烦的事情。考虑下面代码：

```
template<typename T>    // 要求 Element<T>()
void user(vector<T>& v, list<T>& lst)
{
    for (vector<T>::iterator p = v.begin(); p != v.end(); ++p) cout << *p << '\n';

    list<T>::iterator q = find(lst.begin(), lst.end(), T{42});
}
```

这里最恼人的地方是编译器显然已经知道了 `list` 的 `iterator` 类型和 `vector` 的 `size_type`。我们为什么还必须告诉编译器它已经知道的事情呢？这样做徒增我们当中不擅打字的人的烦恼，并增加出错的机会。幸运的是，我们无须这样做：可以将变量声明为 `auto` 的，表示使用 `iterator` 类型作为变量的类型：

```

template<typename T>    // 要求 Element<T>()
void user(vector<T>& v, list<T>& lst)
{
    for (auto p = v.begin(); p!=v.end(); ++p) cout << *p << '\n';

    auto q = find(lst.begin(), lst.end(), T{42});
}

```

这里，`p` 是一个 `vector<T>::iterator`，`q` 是一个 `list<T>::iterator`。我们几乎可以在任何包含初始化的定义中使用 `auto`。例如：

```

auto x = 123; // x 是一个 int
auto c = 'y'; // c 是一个 char
auto& r = x; // r 是一个 int&
auto y = r; // y 是一个 int (引用被隐式解引用了)

```

注意，字符串面值常量的类型为 `const char*`，因此对字符串面值常量使用 `auto` 可能导致令人不快的意外：

```

auto s1 = "San Antonio"; // s1 是一个 const char* (意外! ?)
string s2 = "Fredericksburg"; // s2 是一个 string

```

当我们确切知道想要的类型时，通常指明类型和使用 `auto` 一样容易。

`auto` 的常见用途是在范围 `for` 循环中指明循环变量。考虑下面代码：

```

template<typename C>    // 要求 Container<T>
void print3(const C& cont)
{
    for (const auto& x : cont)
        cout << x << '\n';
}

```

在这段代码中，我们使用 `auto` 的原因是给出容器 `cont` 的元素类型不是那么容易。我们使用 `const` 的原因是并不写入容器元素，而我们使用 `&`(引用) 的原因是为了避免元素过大拷贝代价太高。

## 15.6 实例：一个简单的文本编辑器

✘ 列表最重要的性质就是可以在不移动元素的情况下对其进行插入或删除操作。下面我们通过一个例子来说明这一点。考虑应该如何能在文本编辑器中表示一个文本文件中的字符。所选用的表示方式应当能够使对文本文件进行的操作简单而高效。

那么具体会涉及哪些操作呢？假设文件能存储在计算机的内存中。也就是说，我们可以选择任何一种适合的表示方式，当需要保存到文件中时，只要把它转换成一个字节流就可以了。相应地，我们也可以把一个文件中的字符转成字节流，从而把它读入内存中。这说明我们只需要选择一种合适的内存中的表示方式就可以了。所选择的表示方式需要能够很好地支持以下 5 种操作：

- 从输入的字节流创建该表示方式。
- 插入一个或多个字符。
- 删除一个或多个字符。
- 在其中查找一个 `string`。
- 产生一个字节流从而输出到文件或屏幕中。

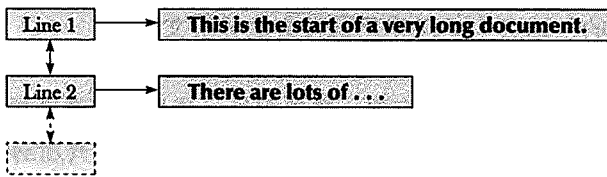
最简单的表示方式就是 `vector<char>`。但是，在 `vector` 中，每加入或删除一个元素时我们就需要移动后面所有的元素。例如：

This is he start of a very long document.  
There are lots of . . .

我们希望加入字符 t:

This is the start of a very long document.  
There are lots of . . .

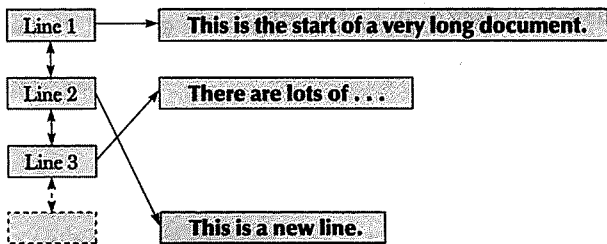
但是，如果用 `vector<char>` 来存储这些字符，那么我们就需要把从 h 开始的所有字符都向右移动。这有可能会大量的复制操作。实际上，如果要在一篇 70 000 字的文本（差不多是英文版本章的长度，计空格）中插入或删除一个字符，那么我们平均需要移动 35 000 个字符。它所需要的执行时间会很长，使我们难以接受。因此，我们不妨把表示方式分成几块，这样不需要移动很多元素就可以对文本的某一部分进行修改。我们把文本文件看成由一系列“行”组成，并用 `list<Line>` 进行表示，其中 `Line` 是一个 `vector<char>`。例如：



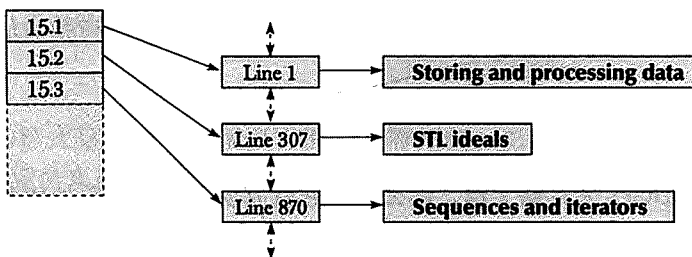
现在，当我们需要插入字符 t 时，只需要移动相应一行中的元素就可以了。而且，如果需要的话，我们可以插入新的一行而不需要移动任何元素。例如，我们可以在“document.”后面加入字符串“This is a new line.”。

This is the start of a very long document.  
This is a new line.  
There are lots of . . .

而我们需要做的只是在它们中间加入一行：



能够在不移动现有链接的情况下在链表中加入新的链接是非常重要的，其逻辑上的原因是我们可能正用迭代器指向这些现有链接，或用指针（以及引用）指向这些链接中的对象。这样，这些迭代器或指针就不会受到插入行或删除行操作的影响。例如，文字处理器利用 `vector<list<Line>::iterator>` 来存储指向当前 Document 的每个标题和子标题的迭代器：



我们可以在不影响指向 15.3 节的迭代器的情况下向 15.2 节加入新的行。

也就是说，出于性能和逻辑上的考虑，我们采用的是一个“行”的 `list`，而不是“行”的 `vector` 或一个存储所有字符的 `vector`。请注意，由于这种情况很少出现，我们前面提到的“尽量使用 `vector`”的准则仍然适用。如果要用 `list` 替代 `vector`，那么你最好有充分的理由说服你自己（见 15.7 节）。不管（链接的）`list` 还是 `vector` 都可以表示逻辑上的“列表”结构。STL 中和我们日常所说的列表（如待办事项、杂货店列表或日程表）最为接近的是序列，而大多数序列最好的表示方式是 `vector`。

### 15.6.1 处理行

我们应该如何判定文档中什么是“一行”呢？有三种显然的选择：

1. 靠换行符（例如 `\n`）来判断。
2. 用某种“自然的”标点（如 `.`）采用某种方法分析文档。
3. 把所有超过给定长度（如 50 个字符）的行都分成两行。

毫无疑问还有其他不那么直观的方法。简单起见，这里我们选择第一种方法。

我们把编辑器中的文档表示成 `Document` 类的一个对象。抛开所有优化，我们的文档类型如下所示：

```
using Line = vector<char>;      // 一行就是一个字符 vector

struct Document {
    list<Line> line;           // 一个文档就是一个行的 list
    Document() { line.push_back(Line{}); }
};
```

每个 `Document` 对象都以一个空行开始：`Document` 的构造函数会创建一个空行并把它加入行的链表中。

读取文档并分行的操作可以按照如下方式完成：

```
istream& operator>>(istream& is, Document& d)
{
    for (char ch; is.get(ch); ) {
        d.line.back().push_back(ch);    // 添加字符
        if (ch=='\n')
            d.line.push_back(Line{});  // 添加一行
    }
    if (d.line.back().size()) d.line.push_back(Line{}); // 添加最后的空行
    return is;
}
```

`vector` 和 `list` 都有一个 `back()` 成员函数，返回指向尾元素的引用。但使用 `back()` 时一定要确定确实存在尾元素：不要在一个空容器上使用它。这也是我们定义的 `Document` 都以一个空 `Line` 结尾的原因。注意我们会保存输入的每个字符，包括换行符（`\n`）。保存这些换行符极大地简化了输出，但你必须小心如何定义字符数（简单统计字符数得到的结果会包含空格和换行）。

### 15.6.2 遍历

如果用 `vector<char>` 来存储文档，那么遍历它就方便多了。那要如何遍历一个行的链表呢？我们当然可以使用 `list<Line>::iterator` 遍历链表，但如果希望可以逐个处理字符而不必

考虑换行呢？为此，我们为 Document 类专门定义一个迭代器：

```
class Text_iterator { // 在行内跟踪行和字符位置
    list<Line>::iterator ln;
    Line::iterator pos;
public:
    // 迭代器从行 ll 中位置 pp 处的字符开始
    Text_iterator(list<Line>::iterator ll, Line::iterator pp)
        :ln(ll), pos(pp) {}

    char& operator*() { return *pos; }
    Text_iterator& operator++();
    bool operator==(const Text_iterator& other) const
        { return ln==other.ln && pos==other.pos; }
    bool operator!=(const Text_iterator& other) const
        { return !(*this==other); }
};

Text_iterator& Text_iterator::operator++()
{
    ++pos; // 前进到下一个字符
    if (pos==(*ln).end()) {
        ++ln; // 前进到下一行
        pos = (*ln).begin(); // 若 ln==line.end() 则错误，因此要确保避免这种情况
    }
    return *this;
}
```

为了发挥 Text\_iterator 的作用，我们为 Document 定义常规的 begin() 和 end() 操作：

```
struct Document {
    list<Line> line;

    Text_iterator begin() // 第一行的首字符
        { return Text_iterator(line.begin(), (*line.begin()).begin()); }
    Text_iterator end() // 最后一行尾字符之后的位置
    {
        auto last = line.end();
        --last; // 我们知道文档不为空
        return Text_iterator(last, (*last).end());
    }
};
```

我们需要奇怪的 (\*line.begin()).begin() 语法，这是因为我们想获得 line.begin() 所指向的数据的开始位置；由于标准库迭代器支持 -> 运算符，我们也可以使用替代语法 line.begin()->begin()。

现在我们可以按照如下方式遍历文档的字符了：

```
void print(Document& d)
{
    for (auto p : d) cout << *p;
}

print(my_doc);
```

以字符序列的形式呈现文档在很多方面是很有用的，但通常我们遍历文档不是查找字符，而是查找更特定的东西。例如，下面的代码删除第 n 行：

```
void erase_line(Document& d, int n)
{
```



```

    if (n<0 || d.line.size()-1<=n) return;
    auto p = d.line.begin();
    advance(p,n);
    d.line.erase(p);
}

```

调用 `advance(p,n)` 将迭代器 `p` 向前移动 `n` 个元素；`advance()` 是标准库函数，不过我们也可以像下面这样实现自己的版本：

```

template<typename Iter> // 要求 Forward_iterator<Iter>
void advance(Iter& p, int n)
{
    while (0<n) { ++p; --n; }
}

```

注意，`advance()` 可用来模拟下标操作。实际上，对于一个 `vector v`，`p=v.begin()`；`advance(p,n)*p=x` 和 `v[n]=x` 大体上是一样的。之所以说是“大体一样”，是因为 `advance()` 一个元素一个元素费力地移动过前 `n-1` 个元素，直到下标指向第 `n` 个元素为止。对于一个 `list`，我们不得不采用这种费力的方法，这是为更灵活的元素布局所付出的代价。

如果迭代器既支持向前移动又支持向后移动，比如 `list` 的迭代器，那么向标准库 `advance()` 函数传递负的参数会使迭代器向后移动。对能处理下标操作的迭代器，例如 `vector` 的迭代器，标准库 `advance()` 会直接移动到指定的元素，而不会用 `++` 运算符缓慢地移动。显然，标准库 `advance()` 确实比我们自己定义的版本要聪明一些。这点很值得注意：标准库特性通常都是花费了很多时间精心设计的，我们很难付出同样精力，所以还是优先使用标准库特性而不是“家庭制作”吧。

### 试一试

重写 `advance()` 函数，使它接受负的参数时向后移动。

对用户来讲，查找可能是最直观的一种迭代了。我们会查找某一个单词（例如 `milkshake` 或 `Gavin`），查找某一不能被看作词组的字符序列（例如 `secret\nhomestead`，也就是说，前一行以 `secret` 结尾，而后一行以 `homestead` 开始），或查找正则表达式（例如 `[bB]w*ne`，表示大写或小写字母 `B` 后接 `0` 或多个字母再接 `ne`，详见第 23 章）。下面我们看看如何处理第二种情况：在我们的 `Document` 中查找某一给定的字符串。我们采用一种简单的非最优算法：

- 在文档中查找搜索串的第一个字符。
- 判断该字符及其后的字符是否与我们的搜索串匹配。
- 如果是，则结束；否则，继续查找搜索串首字符下一次出现的位置。

出于通用性的考虑，我们采用 STL 中常用的方式——将待搜索的文本定义为由一对迭代器所表示的序列。这样我们就可以像搜索完整文档一样对文档的任意部分应用我们的搜索函数。如果在文档中找到了我们所要的字符串，就返回一个指向其首字符的迭代器；否则返回一个指向序列末端的迭代器。

```

Text_iterator find_txt(Text_iterator first, Text_iterator last, const string& s)
{
    if (s.size()==0) return last; // 不能查找一个空字符串
    char first_char = s[0];

```

```

while (true) {
    auto p = find(first,last,first_char);
    if (p==last || match(p,last,s)) return p;
    first = ++p;        // 查找下一个字符
}
}

```

返回序列尾来表示“未找到”是一个非常重要的 STL 规范。match() 函数非常简单，它只是对两个字符序列进行了比较。请尝试自己编写出这个函数。用来在字符序列中查找某一给定字符的 find() 函数可能是标准库中最简单的算法了（见 16.2 节）。我们可以像下面这样使用 find\_txt() 函数：

```

auto p = find_txt(my_doc.begin(), my_doc.end(), "secret\nhomestead");
if (p==my_doc.end())
    cout << "not found";
else {
    // 做一些操作
}

```

我们的“文本处理器”非常简单。显然，我们的目的是简洁高效，而不是提供一个“特性丰富”的编辑器。但不要因此就傻傻地认为提供高效的插入、删除或查找任意序列的操作是一件非常简单的事情。我们选择这个例子来展示序列、迭代器和容器（如 list 和 vector）等 STL 概念结合某些 STL 规范（或者说技术，如返回序列尾来表示失败）是多么强大和通用。注意，如果我们想要的话，也可以把 Document 定义成一个 STL 容器——实际上通过提供 Text\_iterator，我们已经完成了将一个 Document 表示为一个值序列的关键工作了。

## 15.7 vector、list 和 string

为什么我们对行用 list 而对字符用 vector 呢？更准确地说，我们为什么要用 list 保存行的序列而用 vector 保存字符序列呢？再有，为什么不用 string 来存储一行呢？

我们可以把这些问题再一般化一些。到现在为止，我们知道了四种存储字符序列的方法：

- char[] (字符数组)；
- vector<char>;
- string;
- list<char>。

那么对于一个给定问题应该采取哪种存储方式呢？当问题非常简单时，选择哪种方式都无所谓，因为它们都有非常相似的接口。例如，给定一个 iterator，我们可以使用 ++ 遍历所有字符，用 \* 访问字符。在与 Document 相关的代码示例中，我们的确可以将 vector<char> 换成 list<char> 或 string，而不会引起任何逻辑上的问题。这是非常好的特性，因为这令我们只需从性能角度选择存储方式。但是，在考虑性能之前，我们先来看看这些存储方式的逻辑特性：有什么是它能做而其他方式所不能的？

- Elem[]: 不知道它自己的大小。没有 begin()、end() 或任何其他有用的容器成员函数。⚠️ 不能系统地实现边界检查。可以作为参数传递给用 C 或 C++ 编写的函数。其中的元素在内存中连续存储。数组的大小在编译时就确定了。比较 (== 和 !=) 和输出 (<<) 操作使用的是指向数组第一个元素的指针，而不是元素。
- vector[Elem]: 基本上可以做所有事，包括 insert() 和 erase()。支持下标操作。在其上的列表操作，例如 insert() 和 erase()，通常需要移动字符（当元素很大或元素数目

很多时效率会比较低)。可实现边界检查。元素在内存中连续存储。`vector` 可以扩展 (例如使用 `push_back()`)。向量的元素 (连续) 存储在数组中。比较运算符 (`==`、`!=`、`<`、`<=`、`>`、`>=`) 对元素进行比较。

- `string`：提供了所有常见的有用操作，还提供了特殊的文本处理操作，例如字符串的连接 (`+` 和 `+=`)。其元素保证在内存中连续存储。`string` 可以扩展。比较运算符 (`==`、`!=`、`<`、`<=`、`>`、`>=`) 对元素进行比较。
- `list[Elem]`：提供了除下标外所有常见的有用操作。我们在进行 `insert()` 或 `erase()` 操作时不必移动其他元素。每个元素需要两个额外的字 (用来存储链接指针)。`list` 可以扩展。比较运算符 (`==`、`!=`、`<`、`<=`、`>`、`>=`) 对元素进行比较。

正如我们之前提到的 (见 12.2 节和 13.6 节)，当我们需要在底层和内存打交道或需要和 C 程序交互时数组是非常有用且必需的 (见 27.1.2 节和 27.5 节)。在其他情况下，`vector` 由于更方便、灵活且安全，应是首选。

### 试一试


上述的区别在实际的代码中意味着什么？分别定义一个保存值 "Hello" 的 `char` 数组、`vector<char>`、`list<char>` 和 `string`，并把它们作为参数传递给一个函数。该函数首先输出传来的字符串中的字符数目，并将其与函数内定义的 "Hello" 相比较 (以判断你是否真的传递了 "Hello")，然后再与 "Howdy" 比较，看看它们在字典中谁更靠前。把参数复制到另一个相同类型的变量中。

### 试一试

重复上面的“试一试”，这次测试 `int` 数组、`vector<int>` 和 `list<int>`，都保存数值 (1,2,3,4,5)。

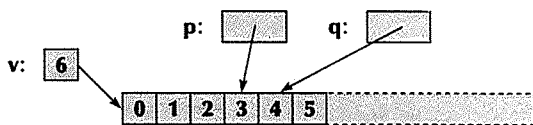
## 15.7.1 insert 和 erase

标准库 `vector` 是我们使用容器时的首选。它几乎具有所有所需特性，所以我们只有在没有办法时才会使用其他的替代品。`vector` 主要的问题在于每当我们执行列表操作 (`insert()` 和 `erase()`) 时，都需要对元素进行移动。当 `vector` 中的元素很多或元素很大时，移动元素会产生很高的代价。但也不必太担心这一点。我们可以放心地用 `push_back()` 读取 50 万个浮点数存入一个 `vector` 中，实验证明相对于预分配所有内存的方法，`push_back()` 并无明显的性能劣势。在为了性能而做出重大改变前一定要进行性能测试，即使是专家也很难猜测性能。

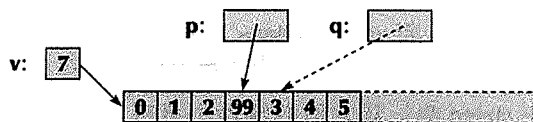
 正如在 15.6 节中所提到的，移动元素的特性还意味着一个逻辑限制：当你对于一个 `vector` 执行列表操作 (如 `insert()`、`erase()` 和 `push_back()`) 时一定不要保留指向其元素的迭代器或指针：若元素移动，你的迭代器或指针将会指向错误的元素，甚至根本不指向任何元素。这也正是 `list` 相对于 `vector` (以及 `map`，参见 16.6 节) 的根本优势。如果你在程序中需要使用很多大对象，而且会在很多地方 (用迭代器或指针) 指向它们，则应考虑使用 `list`。

我们来比较一下 `list` 和 `vector` 的 `insert()` 和 `erase()` 操作。首先看一个仅用来展示关键点的例子：

```
vector<int>::iterator p = v.begin(); // 获取一个 vector
++p; ++p; ++p; // 指向其第 4 个元素
auto q = p; // 指向其第 5 个元素
++q;
```

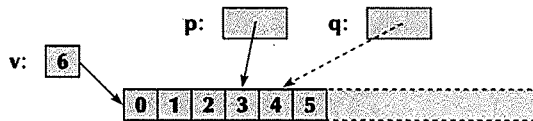


```
p = v.insert(p,99); // p 指向插入的元素
```



现在  $q$  是无效的。随着向量的大小增长，可能会为其元素分配新的内存。如果  $v$  有空闲空间，则它会原地增长， $q$  很可能指向的是值为 3 的元素而不是值为 4 的元素，但千万不要认为一定会是这样。

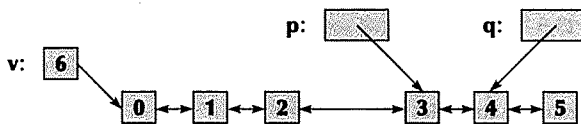
```
p = v.erase(p); // p 指向被删除的元素之后的位置
```



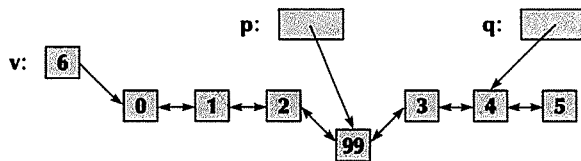
也就是说，如果在  $\text{insert}()$  操作后执行一次  $\text{erase}()$  操作删除刚刚插入的元素，那么我们就回到了起始状态，只是  $q$  变成无效了。但是，在这两次操作之间，我们移动了插入点之后的所有元素，随着  $v$  增长所有元素可能都被重新分配空间了。

作为对比，我们使用  $\text{list}$  来完成相同的操作：

```
list<int>::iterator p = v.begin(); // 获取一个 list
++p; ++p; ++p; // 指向第 4 个元素
auto q = p; // 指向第 5 个元素
++q;
```

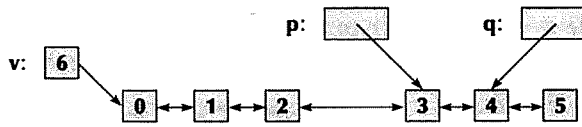


```
p = v.insert(p,99); // p 指向插入的元素
```



注意，`q` 仍然指向值为 4 的元素。

```
p = v.erase(p); // p 指向被删除元素之后的位置
```



我们又一次发现回到了起始状态。但是，与 `vector` 的不同之处在于，我们没有移动任何元素，`q` 始终是有效的。

`list<char>` 与其他三种容器相比需要至少 3 倍的存储空间——在 PC 上，一个 `list<char>` 需要 12 个字节来保存每个元素，而 `vector<char>` 只需要 1 个字节。当字符数很多时，这种差距可能很重要。

那 `vector` 哪方面优于 `string` 呢？从它们的特性中可以发现，`string` 似乎能完成比 `vector` 更多的功能。但这也正是部分问题的所在：由于 `string` 必须完成更多功能，对它进行优化也就更困难了。实际上，`vector` 设计思想之一就是对其 `push_back()` 这样的“内存操作”进行优化，而 `string` 并没有。取而代之，`string` 对拷贝操作、短字符串处理以及与 C 风格字符串的交互进行了优化。在文本编辑器的例子中，我们选择 `vector` 是因为需要使用 `insert()` 和 `delete()`，另一方面也是出于性能的考虑。`vector` 和 `string` 逻辑的主要差异在于 `vector` 几乎可以用于任何元素类型，而只有在处理字符时才需要考虑 `string`。总之，只有当需要进行字符串操作（例如字符串连接或读取空白符间隔的单词）时才考虑使用 `string`，其他情况下，就用 `vector` 好了。

## 15.8 调整 `vector` 类达到 STL 版本的功能

在 15.5 节中为 `vector` 增加了 `begin()`、`end()` 和类型别名后，现在只差 `insert()` 和 `erase()` 就接近我们设计一个 `std::vector` 的近似版本的目标了：

```
template<typename T, typename A = allocator<T>>
// 要求 Element<T>() && Allocator<A>() (见 14.3.3 节)
class vector {
    int sz; // 大小
    T* elem; // 指向元素的指针
    int space; // 元素数加上空闲“槽”数
    A alloc; // 用来分配元素内存
public:
    // ...与第 14 章和 15.5 节代码相同的内容...
    using iterator = T*; // T* 是最简单的迭代器

    iterator insert(iterator p, const T& val);
    iterator erase(iterator p);
};
```

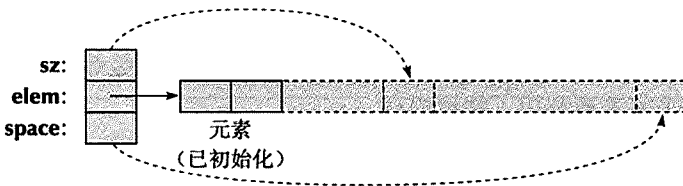
我们还是使用指向元素类型的指针 `T*` 作为迭代器的类型，这是最简单的方法。我们将边界检查迭代器的实现留作练习（习题 18）。

人们通常不会为元素连续存储的数据类型（如 `vector`）提供列表操作，如 `insert()` 或 `erase()`。但 `insert()` 和 `erase()` 这样的列表操作对短 `vector` 或少量元素极为有用也极为高效。我们已经反复看到了 `push_back()` 的作用，它是另一个常见的列表操作。

基本上，我们可以通过拷贝所有位于所删除元素之后的元素来实现 `vector<T,A>::erase()`。利用 14.3.6 节中定义的 `vector` 再加上上述内容，我们得到：

```
template<typename T, typename A>    // 要求 Element<T>() &&
                                   // Allocator<A>() (见 14.3.3 节)
vector<T,A>::iterator vector<T,A>::erase(iterator p)
{
    if (p==end()) return p;
    for (auto pos = p+1; pos!=end(); ++pos)
        *(pos-1) = *pos;           // 将元素拷贝到“左边一个位置”
    alloc.destroy(&*(end()-1));    // 销毁最后一个元素的多余拷贝
    --sz;
    return p;
}
```

借助下面的图示，你可以更容易理解上面代码：



`erase()` 的代码非常简单，但在纸上试着画几个例子可能是个好主意。有没有正确地处理空 `vector`？为什么要判断 `p==end()`？如果删除 `vector` 的最后一个元素会怎么样？如果使用下标语法的话代码的可读性会不会更好？

相对而言，`vector<T,A>::insert()` 就有一点复杂了：

```
template<typename T, typename A>    // 要求 Element<T>() &&
                                   // Allocator<A>() (见 14.3.3 节)
vector<T,A>::iterator vector<T,A>::insert(iterator p, const T& val)
{
    int index = p-begin();
    if (size()==capacity())
        reserve(size()+0?8:2*size()); // 确保我们有空间

    // 首先将最后一个元素拷贝到未初始化的空间
    alloc.construct(elem+sz, *back());
    ++sz;
    iterator pp = begin()+index;    // 存放 val 的位置
    for (auto pos = end()-1; pos!=pp; --pos)
        *pos = *(pos-1);           // 将元素拷贝到右边一个位置
    *(begin()+index) = val;        // “插入” val
    return pp;
}
```

请注意：

- 由于迭代器不能指向序列之外，所以我们使用指针来完成，比如 `elem+sz`。这就是为什么分配器用指针而不是迭代器来定义。
- 当我们使用 `reserve()` 时，元素会被移动到一块新的内存中。因此，我们必须记住插入元素位置的索引，而不是指向它的迭代器。当 `vector` 为其元素重新分配内存时，指向 `vector` 内部的迭代器会失效——可以理解为它们指向的是旧的内存。

- 我们使用分配器参数 **A** 的方式很直观，但不准确。当你需要实现一个容器时，最好还是仔细读一下相关的标准。
- 由于这些微妙的细节，我们尽量避免处理底层内存问题。自然地，标准库 `vector`（以及所有其他标准库容器）能正确实现这些重要的语义细节。这也是优先使用标准库而不是“家庭制作”的原因之一。

出于性能原因，你不应在一个含有 100 000 个元素的 `vector` 内部使用 `insert()` 或 `erase()`；对这种操作，使用 `list`（或 `map`，参见 16.6 节）更为适合。但是，`vector` 确实提供了 `insert()` 和 `erase()` 操作，而且如果我们只是移动几个或几十个数据的话，其性能是没有问题的——毕竟现代计算机已非常擅长这种拷贝操作（见习题 20）。注意不要用 `list`（链表）表示少量元素的列表。

## 15.9 调整内置数组达到 STL 版本的功能

我们之前反复指出内置数组的不足之处：它们动不动就会隐式转换成指针，它们不能通过赋值操作进行拷贝，它们不知道自己的大小（见 13.6.2 节），等等。我们也指出了它们最大的优点：它们近乎完美地利用了物理内存。

为了综合二者之长，我们可以创建一个具有数组优点而没有其不足的 `array` 容器。`array` 的一个版本已经作为技术报告的一部分引入 C++ 标准中。由于技术报告中的特性并不要求所有 C++ 实现都必须包含，因此你所使用的实现可能并不包含 `array`。但其思路是简单有用的：



```
template <typename T, int N>           // 要求 Element<T>()
struct array {                         // 不完全是标准库 array
    using value_type = T;
    using iterator = T*;
    using const_iterator = const T*;
    using size_type = unsigned int;    // 下标类型

    T elems[N];
    // 不需要显式构造函数 / 拷贝操作 / 析构函数

    iterator begin() { return elems; }
    const_iterator begin() const { return elems; }
    iterator end() { return elems+N; }
    const_iterator end() const { return elems+N; }

    size_type size() const;

    T& operator[](int n) { return elems[n]; }
    const T& operator[](int n) const { return elems[n]; }

    const T& at(int n) const;          // 范围检查访问
    T& at(int n);                     // 范围检查访问

    T * data() { return elems; }
    const T * data() const { return elems; }
};
```

这个定义并不完整，也不完全符合 C++ 标准，但可展示设计思想。如果你使用的 C++ 实现并未提供标准 `array`，它也可提供一个有用的定义。如果 C++ 实现提供了标准 `array`，它应该在 `<array>` 中。注意由于 `array<T, N>` “知道”它的大小为 `N`，我们可以提供（我们也

确实提供了) 赋值 ==、!= 等操作, 就像 `vector` 一样。

作为一个示例, 我们把 `array` 和 15.4.2 节中 STL 版 `high()` 结合起来使用:

```
void f()
{
    array<double,6> a = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5 };
    array<double,6>::iterator p = high(a.begin(), a.end());
    cout << "the highest value was " << *p << '\n';
}
```

注意当我们编写 `high()` 时, 并没有考虑 `array`。之所以 `high()` 可以与 `array` 一起使用是因为这二者都是按照标准的习惯进行定义的。

## 15.10 容器概览

STL 提供了一些容器:

标准容器	
<code>vector</code>	连续存储的元素序列; 应用作默认容器
<code>list</code>	双向链表; 当你希望在不移动现有元素的情况下完成对元素的插入和删除时使用
<code>deque</code>	<code>list</code> 和 <code>vector</code> 的交叉; 除非你对算法和计算机体系结构知识非常精通, 否则不要使用它
<code>map</code>	平衡有序树; 当你需要按值访问元素时使用它 (参见 16.6.1 ~ 16.6.3 节)
<code>multimap</code>	平衡有序树, 可以包含同一个 <code>key</code> 的多个拷贝; 当你需要按值访问元素时使用它 (参见 16.6.1 ~ 16.6.3 节)
<code>unordered_map</code>	哈希表; 一种优化的 <code>map</code> ; 当映射规模很大、对性能要求很高且可以设计出好的哈希函数时使用它 (参见 16.6.4 节)
<code>unordered_multimap</code>	可以包含同一个 <code>key</code> 的多个拷贝的哈希表; 一种优化的 <code>multimap</code> ; 当映射规模很大、对性能要求很高且可以设计出好的哈希函数时使用它 (参见 16.6.4 节)
<code>set</code>	平衡有序树; 当你需要追踪单个值时使用它 (参见 16.6.5 节)
<code>multiset</code>	可以包含同一个 <code>key</code> 的多个拷贝的平衡有序树; 当你需要追踪单个值时使用它 (参见 16.6.5 节)
<code>unordered_set</code>	类似 <code>unordered_map</code> , 但只保持值而非 (键, 值) 对
<code>unordered_multiset</code>	类似 <code>unordered_multimap</code> , 但只保持值而非 (键, 值) 对
<code>array</code>	大小固定的数组, 不存在内置数组所存在的大部分问题 (参见 15.9 节)

你能在书籍或在线文档中找到关于这些容器及其使用的大量信息。下面是一些高质量的参考资料:

Josuttis, Nicholai M. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 2012. ISBN 978-0321623218. Use only the 2nd edition.

Lippman, Stanley B., Jose Lajoie, and Barbara E. Moo. *The C++ Primer*. Addison-Wesley, 2005. ISBN 0201721481. Use only the 5th edition.

Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, 2012. ISBN 978-0321714114. Use only the 4th edition.

The documentation for the SGI implementation of the STL and the `iostream` library: [www.sgi.com/tech/stl](http://www.sgi.com/tech/stl). Note that they also provide complete code.



你觉得被骗了吗？你觉得我们应该向你介绍所有容器及其应用吗？这是不可能的。相关的标准特性、技术以及库实在是太多了，你不可能一下就把它们全部掌握。程序设计领域是如此丰富，任何人都难以掌握所有特性和技术，它同时也是一门高雅的艺术。作为一名程序员，你必须养成查找语言特性、库和相关技术新信息的习惯。程序设计是一个时刻快速变化的领域，所以安于现状是走向落后的“秘诀”。“查一查”对很多问题都是一个很好的答案，而随着你的技能逐渐增长、成熟，这一答案会变得越来越重要。

另一方面，你会发现当你了解了 `vector`、`list`、`map` 以及第 16 章中介绍的标准算法后，其他 STL 容器或类 STL 容器的使用会变得非常容易，你还会发现非 STL 容器及其应用也变得容易理解了。

那么什么是容器呢？在上述所有资源中你都可以找到 STL 容器的定义。这里我们只给出一个非正式的定义。一个 STL 容器

- 是一个元素序列 [`begin():end()`]。
- 提供拷贝元素的拷贝操作。拷贝可以通过赋值操作或拷贝构造函数来实现。
- 其元素类型命名为 `value_type`。
- 具有名为 `iterator` 和 `const_iterator` 的迭代器类型。迭代器提供具有恰当语义的 `*`、`++`（前缀和后缀）、`==` 以及 `!=` 运算符。`list` 的迭代器还提供可以在序列中向后移动的 `--` 操作，它也被称为双向迭代器（`bidirectional iterator`）。`vector` 的迭代器还提供 `--`、`[]`、`+` 以及 `-` 运算符，它也被称为随机访问迭代器（`random-access iterator`）（参见 15.10.1 节）。
- 提供 `insert()`、`erase()`、`front()`、`back()`、`push_back()`、`pop_back()` 以及 `size()` 等操作，`vector` 和 `map` 还提供下标操作（例如运算符 `[]`）。
- 提供比较运算符（`==`、`!=`、`<`、`<=`、`>` 以及 `>=`）进行元素比较。容器对 `<`、`<=`、`>`、`>=` 采用字典顺序，也就是说，它们按字典中开始单词排在首位的顺序比较元素。

上面这个列表只是给你一个关于容器的大概印象。更详细的内容请参见附录 C。更准确的说明和更完整的特性列表可以参考《The C++ Programming Language》或 C++ 标准。

一些数据类型提供了标准容器所要求的大部分特性，但又不是全部。我们称之为“拟容器”。最常见的如下表所示。

“拟容器”	
<code>T[n]</code> 内置数组	没有 <code>size()</code> 或其他成员函数；当可以使用 <code>vector</code> 、 <code>string</code> 或 <code>array</code> 等容器时，尽量不要选择内置数组
<code>string</code>	只存储字符，但对文本处理提供了许多有用的操作，例如连接（ <code>+</code> 和 <code>+=</code> ）；相对于其他字符串，应优先选用标准字符串
<code>valarray</code>	具有向量操作的数值向量，但有许多限制制约了高性能实现；只有当你需要进行大量向量计算时使用

另外，许多个人与组织都致力于开发符合（或接近符合）标准容器要求的容器。

如存疑，选用 `vector`。除非有充分的理由，否则选用 `vector`。

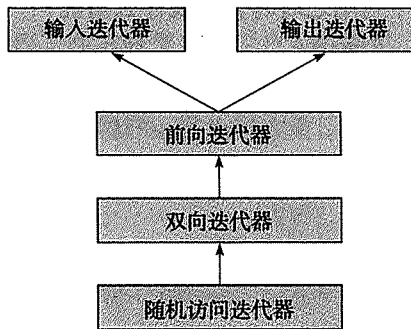
### 15.10.1 迭代器类别

在我们之前的讨论中，似乎所有迭代器都是可以通用的。其实，只有在进行简单的操作

时它们才是通用的，比如对某个序列进行一次遍历并读取每个值一次。如果你希望完成更为复杂的操作，例如向后遍历或下标操作，就会需要一些更为高级的迭代器了。

迭代器类别	
输入迭代器	我们可以用 ++ 向前移动，用 * 读取元素值。istream 提供的就是这类迭代器，参见 16.7.2 节。如果 (*p).m 有效，则可使用简写形式 p->m
输出迭代器	我们可以用 ++ 向前移动，用 * 写入元素值。ostream 提供的就是这类迭代器，参见 16.7.2 节
前向迭代器	我们可以反复用 ++ 向前移动，用 * 读写元素（当然，元素不能是 const 的）。如果 (*p).m 有效，则可使用简写形式 p->m
双向迭代器	我们可以用 ++ 向前移动，用 -- 向后移动，用 * 读写元素（除非元素是 const 的）。list、map 和 set 所提供的就是这类迭代器。如果 (*p).m 有效，则可使用简写形式 p->m
随机访问迭代器	我们可以用 ++ 向前移动，用 -- 向后移动，用 * 或 [] 读写元素（除非元素是 const 的）。我们可以对随机访问迭代器进行下标操作，用 + 向迭代器加上一个整数，用 - 向迭代器减去一个整数。我们可以通过对指向同一序列的两个迭代器进行减法操作来得到它们的距离。vector 提供的就是这类迭代器。如果 (*p).m 有效，则可使用简写形式 p->m

从这些提供的操作我们可以看出，当可以使用输出或输入迭代器时，也可以使用前向迭代器完成相同的功能。双向迭代器也是一种前向迭代器，而随机访问迭代器也是一种双向迭代器。我们可以把迭代器类别图示如下：



注意，由于迭代器种类并不是类，因此这个层次结构并非是用派生实现的类层次。

## 简单练习

1. 定义一个含有 10 个元素 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} 的 int 型数组。
2. 定义一个含有 10 个同样元素的 vector<int>。
3. 定义一个含有 10 个同样元素的 list<int>。
4. 另外再定义一个数组、一个向量、一个链表，并分别把它们初始化为上面所定义的数组、向量和链表的拷贝。
5. 把数组中的每个元素值加 2，把向量中的每个元素值加 3，把链表中的每个元素值加 5。
6. 编写一个简单的 copy() 操作。

```

template<typename Iter1, typename Iter2>
// 要求 Input_iterator<Iter1>() && Output_iterator<Iter2>()
Iter2 copy(Iter1 f1, Iter1 e1, Iter2 f2);
  
```

该操作像标准库 `copy` 函数一样把 `[f1,e1)` 复制到 `[f2, f2+(e1-f1))` 并返回 `f2+(e1-f1)`。注意如果 `f1==e1`，那么该序列为空，此时不需要复制任何内容。

7. 使用你的 `copy` 把数组中的内容拷贝到向量中，再把链表中的内容拷贝到数组中。
8. 用标准库 `find()` 来判断向量中是否含有值 3，如果有则输出它的位置；用 `find()` 来判断链表中是否含有值 27，如果有则输出它的位置。第一个元素的位置为 0，第二个元素的位置为 1，以此类推。注意，如果 `find()` 返回的是序列尾，则说明没有找到所查的元素。记住，每一步后都要进行测试。

## 思考题

1. 为什么不同人编写的代码看起来会不一样？请举例说明。
2. 会有哪些关于数据的简单问题？
3. 存储数据有哪些不同的方式？
4. 可以对一组数据做哪些基本操作？
5. 理想的数据存储方式应该是怎样的？
6. 什么是 STL 序列？
7. 什么是 STL 迭代器？它支持哪些操作？
8. 如何把迭代器移到下一个元素？
9. 如何把迭代器移到上一个元素？
10. 当你试图把迭代器移动到序列尾之后时会出现什么情况？
11. 哪些迭代器可以移动到上一个元素？
12. 为什么要把数据与算法分离开？
13. 什么是 STL？
14. 什么是链表？它和向量本质上的区别是什么？
15. 链表中的链接是什么？
16. `insert()` 的功能是什么？`erase()` 呢？
17. 如何判断一个序列是否为空？
18. `list` 中的迭代器提供了哪些操作？
19. 如何使用 STL 遍历一个容器？
20. 什么时候应该使用 `string` 而不是 `vector`？
21. 什么时候应该使用 `list` 而不是 `vector`？
22. 什么是容器？
23. 容器的 `begin()` 和 `end()` 应该实现什么功能？
24. STL 提供了哪些容器？
25. 什么是迭代器类别？STL 提供了哪几类迭代器？
26. 哪些操作是随机访问迭代器提供了而双向迭代器没有提供的？

## 术语

algorithm (算法)	<code>begin()</code>	doubly-linked list(双向链表)
array container (array 容器)	container (容器)	element (元素)
auto	contiguous (连续的)	empty sequence (空序列)

end()	linked list (链表)	traversal (遍历)
erase()	sequence (序列)	using
insert()	singly-linked list (单向链表)	type alias (类型别名)
iteration (迭代)	size_type	value_type
iterator (迭代器)	STL (标准模板库)	

## 习题

1. 如果还未完成本章所有“试一试”，请完成。
2. 编译运行 15.1.2 节中的例子 Jack-and-Jill。利用几个小文件作为输入测试它。
3. 回顾回文的例子（见 13.7 节），用不同技术重写 15.1.2 节中的 Jack-and-Jill 程序。
4. 用 STL 技术找到并修改 15.3.1 节 Jack-and-Jill 例子中的错误。
5. 为 `vector` 定义输入和输出运算符 (`>>` 和 `<<`)。
6. 基于 15.6.2 节中的内容，为 `Document` 编写一个“查找并替换”操作。
7. 在一个未排序的 `vector<string>` 中查找按字典顺序排在最后的字符串。
8. 编写一个可以统计 `Document` 中字符总数的函数。
9. 编写一个可以统计 `Document` 中单词数的程序。该程序有两个版本：一种将单词定义为“以空白符分隔的字符序列”，另一种将单词定义为“一个连续的字母序列”。例如，在第一种定义下，`alpha.numeric` 和 `as12b` 都是一个单词，而在第二种定义下它们则都为两个单词。
10. 编写另一个统计单词的程序。在该程序中，用户可以指定空白符集合。
11. 以 `list<int>` 为（引用）参数，创建一个 `vector<double>`，并将链表中的元素拷贝到向量中。验证拷贝操作的完整性与正确性。然后将元素按照值升序排序并打印。
12. 完成 15.4.1 ~ 15.4.2 节中 `list` 的定义，并让 `high()` 能正确运行。分配一个 `Link` 表示链表尾之后一个位置。
13. 实际上，在 `list` 中我们并不需要一个“真的”尾后位置 `Link`。改写上面的程序，用 0 表示指向（不存在的）尾后位置 `Link (list<Elem>::end())` 的指针，这样可以使空列表的大小和一个指针的大小相同。
14. 定义一个单向链表 `slist`，风格类似 `std::list`。由于 `slist` 没有后向指针，`list` 中的哪些操作可以合理地去掉？
15. 定义一个与指针 `vector` 很相似的 `pvector`，不同之处在于 `pvector` 中的指针指向对象，而且其析构函数会对每个对象进行 `delete` 操作。
16. 定义一个与 `pvector` 很相似的 `ovector`，不同之处在于 `[]` 和 `*` 运算符返回元素所指向的对象的引用，而不是返回指针。
17. 定义一个 `ownership_vector`，像 `pvector` 一样保存对象指针，但为用户提供了一种机制，可以决定哪些对象为向量所有（即哪些对象由析构函数进行 `delete` 操作）。
18. 为 `vector` 定义一个范围检查迭代器（随机访问迭代器）。
19. 为 `list` 定义一个范围检查迭代器（双向迭代器）。
20. 运行一个小的计时实验来比较 `vector` 和 `list` 的使用代价。有关如何对程序进行计时的内容可以在 26.6.1 节中找到。生成  $N$  个属于区间  $[0, N)$  的 `int` 型随机数。每生成一个随机数就把它加入 `vector<int>` 中（该 `vector` 大小每次增加 1）。保持该 `vector` 为有序的，也就

是说，新元素插入位置之前的所有元素都小于等于新元素的值，而新元素之后的所有元素都大于新元素的值。用 `list<int>` 保存 `int` 完成相同的实验。在  $N$  取什么样的值时 `list` 会比 `vector` 快？请解释实验结果。该实验由 John Bentley 最先提出。

## 附言



如果我们有  $N$  种数据容器和  $M$  种想要对其执行的操作，那么结果很容易变成我们要编写  $N \times M$  段代码。如果数据有  $K$  种不同的类型，那么代码的总数甚至会达到  $N \times M \times K$ 。STL 通过将元素类型作为参数（解决了因子  $K$ ）以及将算法与数据访问分离解决了这一问题。通过利用迭代器实现任意算法对任意容器中数据的访问，我们只需编写  $N+M$  种算法。这大大简化了我们的工作。例如，如果我们有 12 种容器和 60 种算法，暴力方法需要编写 720 个函数；而 STL 的策略只需要编写 60 个函数和定义 12 种迭代器，这可以省去我们 90% 的工作。实际上，这还低估了节省的工作量，因为很多算法接受一对迭代器参数，而两个迭代器可以是不同类型（例如，参见习题 6）。而且，STL 还给出了算法定义规范，可以简化编写正确的、可组合的代码的工作，因此工作量的节省实际上会更大。

## 算法和映射

理论上，实践是简单的。

——Trygve Reenskaug

本章将完成我们对 STL 基本思想的介绍以及对 STL 所提供工具的纵览。在本章中，我们主要关注算法。我们的主要目的是给你介绍一些最有用的算法，它们能够节省你大量时间，即使达不到以月计，也能达到以天计。每个算法都将通过其使用示例和支持的编程技术来介绍。本章的另一个目的是提供足够的工具，令你在需要标准库和其他库之外的特性时有能力自己编写优雅高效的算法。另外，本章还将介绍三种容器：map、set 和 unordered\_map。

## 16.1 标准库算法

标准库大约提供了 80 种有用的算法。所有算法都至少在某些场景下对某些人是有用的；我们将在本章着重介绍其中一些对很多人通常都有用的算法以及一些对某些人极为有用的算法：

### 挑选的标准算法

<code>r = find(b,e,v)</code>	<code>r</code> 指向 <code>[b:e)</code> 中 <code>v</code> 首次出现的位置
<code>r = find_if(b,e,p)</code>	<code>r</code> 指向 <code>[b:e)</code> 中令 <code>p(x)</code> 为 <code>true</code> 的第一个元素 <code>x</code>
<code>x = count(b,e,v)</code>	<code>x</code> 为 <code>v</code> 在 <code>[b:e)</code> 中出现的次数
<code>x = count_if(b,e,p)</code>	<code>x</code> 为 <code>[b:e)</code> 中满足 <code>p(x)</code> 为 <code>true</code> 的元素的个数
<code>sort(b,e)</code>	用 <code>&lt;</code> 运算符对 <code>[b:e)</code> 排序
<code>sort(b,e,p)</code>	用谓词 <code>p</code> 对 <code>[b:e)</code> 排序
<code>copy(b,e,b2)</code>	将 <code>[b:e)</code> 拷贝至 <code>[b2:b2+(e-b))</code> ； <code>b2</code> 之后应有足够的空间用于存储元素
<code>unique_copy(b,e,b2)</code>	将 <code>[b:e)</code> 拷贝至 <code>[b2:b2+(e-b))</code> ；不拷贝相邻的重复元素
<code>merge(b,e,b2,e2,r)</code>	将有序序列 <code>[b2:e2)</code> 和 <code>[b:e)</code> 合并，并放入 <code>[r:r+(e-b)+(e2-b2))</code> 之中
<code>r = equal_range(b,e,v)</code>	<code>r</code> 是有序范围 <code>[b:e)</code> 的一个子序列，且其中所有元素值均为 <code>v</code> ，本质上是通过二分搜索查找 <code>v</code>
<code>equal(b,e,b2)</code>	<code>[b:e)</code> 和 <code>[b2:b2+(e-b))</code> 的所有元素对应相等？
<code>x = accumulate(b,e,i)</code>	<code>x</code> 是将 <code>i</code> 与 <code>[b:e)</code> 中所有元素进行累加的结果
<code>x = accumulate(b,e,i,op)</code>	与 <code>accumulate</code> 类似，但用 <code>op</code> 进行“求和”运算
<code>x = inner_product(b,e,b2,i)</code>	<code>x</code> 是 <code>[b:e)</code> 与 <code>[b2:b2+(e-b))</code> 的内积
<code>x = inner_product(b,e,b2,i,op,op2)</code>	与 <code>inner_product</code> 类似，但用 <code>op</code> 和 <code>op2</code> 取代内积的 <code>+</code> 和 <code>*</code>

默认情况下，相等比较用 `==` 进行，而序则是基于 `<`（小于）的。标准库算法可在 `<algorithm>` 找到。如果想获得更多信息，请参考附录 C.5 和 16.2 ~ 16.5 节中列出的资源。

这些算法接受一个或几个序列。一个输入序列由一对迭代器定义，一个输出序列由一个指向首元素的迭代器定义。通常，一种算法可以由一个或多个操作参数化，这些操作可以定义为函数对象或函数。这些算法通常会通过返回输入序列尾来报告“失败”。例如，如果 `find(b,e,v)` 未找到 `v`，则返回 `e`。

## 16.2 最简单的算法 `find()`

`find()` 可能是最简单但又很有用的算法，它在一个序列中查找一个给定值：

```
template<typename In, typename T>
// 要求 Input_iterator<In>()
//    && Equality_comparable<Value_type<T>>() (见 14.3.3 节)
In find(In first, In last, const T& val)
// 在 [first,last) 中查找等于 val 的第一个元素
{
    while (first!=last && *first != val) ++first;
    return first;
}
```

让我们看看 `find()` 的定义。你自然可以无须了解 `find()` 的确切实现细节就使用它——实际上，我们已经在前面的章节中使用过 `find()` 了（例如 15.6.2 节）。但是，`find()` 的定义展示了很多有用的设计思想，因此了解其实现是有价值的。

✘ 首先，`find()` 对一个序列进行操作，这个序列由一对迭代器定义。它在一个半开区间 `[first: last)` 中查找给定值 `val`。返回结果是一个迭代器，要么指向 `val` 在序列中首次出现的位置，要么就是 `last`。在 STL 中，返回指向尾后位置的迭代器 (`last`) 是最常用的报告“未找到”的方法。因此，我们可以像下面这样使用 `find()`：

```
void f(vector<int>& v, int x)
{
    auto p = find(v.begin(),v.end(),x);
    if (p!=v.end()) {
        // 我们在 v 中找到了 x
    }
    else {
        // v 中没有 x
    }
    // ...
}
```

在本例中，序列照例由一个容器（STL `vector`）中所有元素组成。我们检查返回的迭代器是否指向序列的结束，来判断是否找到了想要的值。返回值类型与迭代器参数的类型是一致的。

为了避免明确指明返回类型，我们使用了 `auto`。若对象的“类型”定义为 `auto`，则从其初始化器获取类型。例如：

```
auto ch = 'c'; // ch 是一个 char
auto d = 2.1; // d 是一个 double
```

类型说明符 `auto` 在泛型编程中特别有用，例如在 `find()` 中明确指明真实类型（本例中是 `vector<int>::iterator`）是很烦人的。

我们现在已经知道如何使用 `find()` 了，从而也了解了如何使用那些采用相同规范的算法。在学习更多用法和更多算法之前，让我们再进一步观察 `find()` 的定义：

```

template<typename In, typename T>
    // 要求 Input_iterator<In>()
    //    && Equality_comparable<Value_type<T>>() (见 14.3.3 节)
In find(In first, In last, const T& val)
    // 在 [first,last) 中查找等于 val 的第一个元素
{
    while (first!=last && *first != val) ++first;
    return first;
}

```

当你第一次看这段代码时，你会注意代码中的循环吗？我们不会。这个循环真的非常简洁、高效，并且直接表达了基础算法。然而，可能在看了若干例子之后，你才会注意到这个循环。让我们用比较常见的方式重写 find()，并比较两个版本：

```

template<typename In, typename T>
    // 要求 Input_iterator<In>()
    //    && Equality_comparable<Value_type<T>>() (见 14.3.3 节)
In find(In first, In last, const T& val)
    // 在 [first,last) 中查找等于 val 的第一个元素
{
    for (In p = first; p!=last; ++p)
        if (*p == val) return p;
    return last;
}

```

这两个定义在逻辑上是等价的，而且一个真正优秀的编译器能够为它们生成相同的代码。然而，现实中很多编译器都没有那么好，它们无法消除额外的变量 (p)，也不能重排代码以使所有条件测试都在同一位置执行。我们为什么要为此担忧并加以解释呢？一部分原因在于 find() 的第一个版本（也是应该优选的版本）的风格已变得十分流行，我们必须学会它以阅读他人编写的代码；另一部分原因是，对于用来处理大量数据且被频繁使用的小函数而言，性能是十分重要的。

### 试一试

你确定这两个定义在逻辑上是等价的吗？你是如何确定的？试着给出两者等价的论证。然后，用一些数据测试这两个定义。著名的计算机科学家 Don Knuth 曾经说，“我只是证明了算法的正确性，并没有对它进行测试”。即使是数学证明也可能包含错误。为了证明你的观点，推理和测试缺一不可。

## 16.2.1 一些一般的应用

find() 算法是通用的。这意味着它能用于不同的数据类型。实际上，find() 算法的通用性包括两个方面；它能用于：

- 任何 STL 风格的序列；
- 任何元素类型。

下面是一些例子（如果你感到困惑，请参考 15.4 节中的图表）：

```

void f(vector<int>& v, int x)           // 适用于 int 的 vector
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* 我们找到了 x */}
}

```



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

✂ 在此例中，`find()` 使用了 `vector<int>::iterator` 实现遍历操作；即，`++first` 中的 `++` 只是将指针移动到内存中的下一个位置（存储了 `vector` 的下一个元素），而 `*first` 中的 `*` 对该指针进行解引用。迭代器比较（`first != last`）就是指针的比较，而值的比较（`*first != val`）就是比较两个整数。

让我们再尝试 `list`：

```
void f(list<string>& v, string x)    // 适用于 string 的 list
{
    list<string>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* 我们找到了 x */}
    //...
}
```

✂ 在本例中，`find()` 使用了 `list<string>::iterator` 实现遍历操作。用到的运算符都具有符合要求的喻意，因此代码逻辑与 `vector<int>` 的例子相同。但实现是很不同的；即，`++first` 中的 `++` 简单地顺着元素的 `Link` 部分中的指针指向 `list` 下一元素的存储位置，而 `*first` 中的 `*` 将获得 `Link` 的值部分。迭代器比较（`first != last`）是 `Link * 指针` 的比较，而值的比较（`*first != val`）则是使用 `string` 的 `!=` 运算符比较两个 `string`。

因此，`find()` 是极为灵活的：只要我们遵循了迭代器的简单规则，就可以用 `find()` 在我们能想象的任何序列和能定义的任何容器中查找元素。例如，我们可以使用 `find()` 在 15.6 节中定义的 `Document` 中查找一个字符：

```
void f(Document& v, char x)    // 适用于 char 的 Document
{
    Text_iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* 我们找到了 x */}
    //...
}
```

这种灵活性是 STL 算法的标志性特点，也使得它们远比初次接触的人所能想象的更为有用。

### 16.3 通用搜索算法 `find_if()`

其实我们并没有那么经常地需要查找一个特定值。我们通常更感兴趣的是在序列中查找符合某种标准的值。如果能够允许我们自己定义查找标准，这样的 `find` 操作就更为有用。例如，我们也许希望查找大于 42 的值，也许希望在不考虑大小写的情况下比较字符串，也许希望找到第一个奇数值，也许希望查找一个地址域值为“17 Cherry Tree Lane”的记录。

根据用户提供的标准进行查找的标准算法是 `find_if()`：

```
template<typename In, typename Pred>
    // 要求 Input_iterator<In>() && Predicate<Pred, Value_type<In>>()
    In find_if(In first, In last, Pred pred)
{
    while (first!=last && !pred(*first)) ++first;
    return first;
}
```

显然（当你比较源码时），`find_if()` 的实现与 `find()` 的实现很相似，除了前者使用 `!pred(*first)`

而非 `*first != val`；即，一旦断言 `pred()` 成立，`find_if()` 就立即停止搜索，而不是当一个元素等于给定值时停止。

断言是一种返回 `true` 或 `false` 的函数。显然，`find_if()` 要求断言接受一个参数，这样就可以说 `pred(*first)`。我们可以容易地编写一个断言来检查值的某种属性，例如“字符串是否包含字母 `x`？”“值是否大于 42？”“数是否是奇数？”例如，我们可以通过如下方式在 `int` 的向量中查找第一个奇数：

```
bool odd(int x) { return x%2; }    // % 是模运算符

void f(vector<int>& v)
{
    auto p = find_if(v.begin(), v.end(), odd);
    if (p!=v.end()) { /* 我们找到了一个奇数 */
        // ...
    }
}
```

在这个 `find_if()` 调用中，`find_if()` 会对每一元素调用 `odd()` 直至它找到了第一个奇数。注意，当你将一个函数作为参数传递时，你不应在它的名字后面加上 `()`，因为这样做会调用它。

类似地，我们可以查找一个链表中第一个大于 42 的元素：

```
bool larger_than_42(double x) { return x>42; }

void f(list<double>& v)
{
    auto p = find_if(v.begin(), v.end(), larger_than_42);
    if (p!=v.end()) { /* 我们找到了一个 >42 的值 */
        // ...
    }
}
```

最后这个例子并不是十分令人满意。如果我们下次想找出大于 41 的元素该怎么办呢？我们不得不编写一个新的函数。那查找大于 19 的元素又如何？又要编写另一个函数。应该有更好的方法！

如果我们想要与任意的值 `v` 进行比较，我们需要某种方法令 `v` 成为 `find_if()` 的断言的一个隐含参数。我们可以尝试如下（选择 `v_val` 作为变量名以避免与其他名称发生冲突）：

```
double v_val;    // larger_than_v() 将此值与自己的实参进行比较
bool larger_than_v(double x) { return x>v_val; }

void f(list<double>& v, int x)
{
    v_val = 31;    // 将 v_val 设置为 31，用于下一次 larger_than_v 调用
    auto p = find_if(v.begin(), v.end(), larger_than_v);
    if (p!=v.end()) { /* 我们找到了一个 >31 的值 */

        v_val = x;    // 将 v_val 设置为 x，用于下一次 larger_than_v 调用
        auto q = find_if(v.begin(), v.end(), larger_than_v);
        if (q!=v.end()) { /* 我们找到了一个 >x 的值 */

            // ...
        }
    }
}
```

哟！我们相信编写这段代码的人最终能得到想要的结果，但我们很同情代码的用户和维护者。再次强调：应该还有更好的方法！

## 试一试

为什么我们讨厌这样使用 `v` 呢？请给出这种编程方式可能导致的三种隐晦错误。列出三个应用，你特别讨厌发现这种代码。

## 16.4 函数对象

因此，我们希望向 `find_if()` 传递断言，同时希望断言能够将元素与以参数形式传递的值进行比较。特别地，我们希望能编写如下形式的代码：


```
void f(list<double>& v, int x)
{
    auto p = find_if(v.begin(), v.end(), Larger_than(31));
    if (p!=v.end()) { /* 我们找到了一个 >31 的值 */}

    auto q = find_if(v.begin(), v.end(), Larger_than(x));
    if (q!=v.end()) { /* 我们找到了一个 >x 的值 */}

    // ...
}
```

显然，`Larger_than` 必须满足如下条件：

- 能作为断言被调用，例如，`pred(*first)`；
- 能够存储一个数值，例如 `31` 或 `x`，以备调用时使用。

 为了满足这些条件，我们需要“函数对象”，即一种能够实现函数行为的对象。我们需要对象的原因是对象能够存储数据，例如待比较的值。举例来说：

```
class Larger_than {
    int v;
public:
    Larger_than(int vv) : v(vv) {}           // 保存参数
    bool operator()(int x) const { return x>v; } // 比较
};
```


有趣的是，此定义就能使前面的例子正常工作了。现在，我们需要弄明白它为何能奏效。当我们调用 `Larger_than(31)` 时，（显然）我们创建了一个 `Larger_than` 类对象，其数据成员 `v` 保存了值 `31`。例如：

```
find_if(v.begin(),v.end(),Larger_than(31))
```

在这里，我们将对象 `Larger_than(31)` 作为参数 `pred` 的实参传递给 `find_if()`。对 `v` 的每个元素，`find_if()` 会调用：

```
pred(*first)
```

这对我们的函数对象调用名为 `operator()` 的调用运算符，传递给它的参数是 `*first`。结果将是元素值 `*first` 和 `31` 的比较结果。

 我们在这里看到的是：一个函数调用可被视为一个运算符——`()` 运算符。“`()` 运算符”也被称为函数调用运算符（function call operator）和应用运算符（application operator）。因此，`pred(*first)` 中的 `()` 由 `Larger_than::operator()` 赋予含义，就像 `v[i]` 中的下标操作由 `vector::operator[]` 赋予含义一样。

### 16.4.1 函数对象的抽象视图

我们已经学习了一种机制，允许一个“函数”“随身携带”它所要的数据。显然，函数对象为我们提供了一种非常通用、强大且便利的机制。下面的例子展示了函数对象的更一般性的概念：

```
class F {           // 函数对象的抽象例子
    S s;           // 状态
public:
    F(const S& ss) :s(ss) { /* 建立初始状态 */ }
    T operator() (const S& ss) const
    {
        // 用 ss 对 s 进行某些操作
        // 返回一个类型为 T 的值 (T 通常为 void、bool 或 S)
    }

    const S& state() const { return s; } // 暴露状态
    void reset(const S& ss) { s = ss; } // 重置状态
};
```

类 F 的对象用其成员 s 存储数据。如果需要，一个函数对象可以拥有很多数据成员。某个对象保存数据的另一种表达方式是称其“具有状态”。当我们创建一个 F 时，可以初始化其状态。当需要时，我们可以读取其状态。对于 F，我们提供了一个操作 state() 来读取状态，还提供了另一个操作 reset() 来设置状态。不过，我们设计一个函数对象时可以根据需要提供任何访问状态的方法。我们当然也可以直接或间接地通过普通函数调用语法来调用函数对象。在上面代码中，我们定义 F 在被调用时只接收一个参数，但可以根据需要定义接受多个参数的函数对象。

函数对象的使用是 STL 中最主要的参数化方法。我们通过函数对象指定需要查找的数据（见 16.3 节），定义排序标准（见 16.4.2 节），在数值算法中指定算术运算（见 16.5 节），定义值相等的含义（见 16.8 节）以及其他很多事情。函数对象的使用是灵活性和通用性的主要源泉。

函数对象通常是十分高效的。特别地，向一个模板函数以传值的方式传递一个小的函数对象通常能够带来优化的性能。原因很简单，但对于熟悉将函数作为参数传递的人来说可能是奇怪的：传递函数对象所产生的代码通常远比传递函数所产生的代码更小、更快。但这一结论仅当函数对象较小（如只占 0、1 或 2 个字）或者采用的是引用方式传递，并且函数调用运算符比较简单（如简单的比较操作 <）且定义为内联方式（例如定义在类内）时才是正确的。本章中——以及本书中——的大多数例子都满足这些条件。小且简单的函数对象能够带来高性能的基本原因在于它们保留了足够的类型信息供编译器产生优化代码。甚至老旧的没有复杂优化器的编译器都能够为 Larger\_than 中的比较操作生成一条简单的“大于”机器指令而不是生成一个函数调用。一次函数调用所需花费的时间通常是执行一条简单比较操作所花费时间的 10 到 50 倍。另外，函数调用所产生的代码通常是简单比较操作所产生代码的数倍之大。

### 16.4.2 类成员上的断言

我们已经看到，标准算法能够正确处理由基本类型（如 int 和 double）元素组成的序列。但是，在一些应用领域，类对象的容器更为常见。下面这个例子是很多领域中应用的关键操

作——根据多个标准对记录进行排序：

```
struct Record {
    string name;      // 标准 string 易于使用
    char addr[24];   // 旧风格，以匹配数据库布局
    // ...
};

vector<Record> vr;
```

我们有时希望根据名字对 `vr` 进行排序，有时又希望根据地址进行排序。除非我们能同时优雅、高效地实现这两种排序标准，否则我们的技术的实用价值就会受到局限。幸运的是，同时实现两种排序标准并不难。我们可以编写如下代码：

```
// ...
sort(vr.begin(), vr.end(), Cmp_by_name()); // 按名字排序
// ...
sort(vr.begin(), vr.end(), Cmp_by_addr()); // 按地址排序
// ...
```



`Cmp_by_name` 函数对象通过比较 `name` 成员来比较两个 `Record`。`Cmp_by_addr` 函数对象则通过比较 `addr` 成员来比较两个 `Record`。为了允许用户指定比较标准，标准库算法 `sort` 接受可选的第三参数用以指定比较标准。`Cmp_by_name()` 为 `sort()` 构造了一个 `Cmp_by_name` 对象，用来比较 `Record`。这看起来不错——意思是我们不介意维护这样的代码。现在，我们所要做的就是定义 `Cmp_by_name` 和 `Cmp_by_addr`：

```
// Record 对象的不同比较方法

struct Cmp_by_name {
    bool operator()(const Record& a, const Record& b) const
    { return a.name < b.name; }
};

struct Cmp_by_addr {
    bool operator()(const Record& a, const Record& b) const
    { return strncmp(a.addr, b.addr, 24) < 0; } // !!!
};
```

`Cmp_by_name` 类的实现十分简单。函数调用运算符 `operator ()()` 简单地用标准 `string` 的 `<` 运算符对 `name` 字符串进行比较。但 `Cmp_by_addr` 中的比较操作很丑陋。这是因为我们采用了一种丑陋的方式表示地址：24 个字符的数组（非 0 结尾）。之所以采用这种方式，一部分原因是为了展示函数对象是如何用于掩盖丑陋且容易产生错误的代码的，另一部分原因是这种特别的表示方式曾被作为一个挑战呈现给我：“一个 STL 不能处理的丑陋而又重要的现实问题。”实际上，STL 能够处理。比较函数使用了标准 C（和 C++）库函数 `strncmp()`，该函数能够比较固定长度的字符数组，当第二个“字符串”在字典序中排在第一个“字符串”之后时它返回一个负数。假如你需要进行这种晦涩的比较操作，可以查阅此参数（如附录 C.11.3）。

### 16.4.3 lambda 表达式

我们通常在程序中某处定义一个函数对象（或一个函数），然后在其他地方使用它，这有些令人厌烦。如果想要执行的操作很容易说明、很容易理解且之后再不会用到的话，还必须这么做就更令人生厌了。这种情况下，我们可以使用 `lambda` 表达式（见 20.3.3 节）。可能

思考 lambda 表达式的最好方式是将它看作定义一个函数对象（具有 () 运算符的类）然后立即创建其对象的一种简写语法。例如，我们可以像下面这样编写代码：

```
// ...
sort(vr.begin(), vr.end(),          // 按名字排序
     [](const Record& a, const Record& b)
     { return a.name < b.name; }
);
// ...
sort(vr.begin(), vr.end(),          // 按地址排序
     [](const Record& a, const Record& b)
     { return strcmp(a.addr, b.addr, 24) < 0; }
);
// ...
```

对于此例，我们怀疑一个命名的函数对象是否会增加代码维护的负担，而且也许 Cmp\_by\_name 和 Cmp\_by\_addr 还有其他用途。

但是，考虑 16.4 节的 find\_if() 例子。在那个例子中，我们需要将操作作为参数传递，且此操作需要携带数据：

```
void f(list<double>& v, int x)
{
    auto p = find_if(v.begin(), v.end(), Larger_than(31));
    if (p!=v.end()) { /* 我们找到了一个 >31 的值 */}

    auto q = find_if(v.begin(), v.end(), Larger_than(x));
    if (q!=v.end()) { /* 我们找到了一个 >x 的值 */}

    // ...
}
```

还有一种等价的替代方法：

```
void f(list<double>& v, int x)
{
    auto p = find_if(v.begin(), v.end(), [](double a) { return a>31; });
    if (p!=v.end()) { /* 我们找到了一个 >31 的值 */}
    auto q = find_if(v.begin(), v.end(), [&](double a) { return a>x; });
    if (q!=v.end()) { /* 我们找到了一个 >x 的值 */}

    // ...
}
```

lambda 版本可以与局部变量 x 进行比较，这令它更具吸引力。

## 16.5 数值算法

大多数的标准库算法都涉及处理数据管理问题：它们需要对数据进行拷贝、排序、查找等。但是，只有少数算法涉及数值计算。当我们需要进行计算时，这些数值算法就变得十分重要了，并且这些算法为我们在 STL 框架中编写数值算法提供了范例。

在 STL 标准库中只有四种数值算法：

---

### 数值算法

`x = accumulate(b,e,i)`

累加序列中的值；例如，对 {a, b, c, d} 计算 i+a+b+c+d。结果 x 的类型与初始值 i 的类型一致

---

(续)

数值算法	
<code>x = inner_product(b,e,b2,i)</code>	将两个序列的对应元素相乘并将结果累加。例如，对 {a, b, c, d} 和 {e, f, g, h} 计算 $i+a \cdot e+b \cdot f+c \cdot g+d \cdot h$ 。结果 <code>x</code> 的类型与初始值 <code>i</code> 的类型一致
<code>r = partial_sum(b,e,r)</code>	对一个序列的前 <code>n</code> 个元素进行累加，并将累加结果生成一个序列。例如，对 {a, b, c, d} 将生成 {a, a+b, a+b+c, a+b+c+d}
<code>r = adjacent_difference(b,e,b2,r)</code>	对一个序列的相邻元素进行减操作，并将得到的差生成一个序列。例如，对 {a, b, c, d} 将生成 {a, b-a, c-b, d-c}

这些算法可以在 `<numeric>` 中找到。我们将介绍前两个，如果你觉得有需要的话，可以自己查阅其他两个的详细情况。

### 16.5.1 累积

`accumulate()` 是最简单但最有用的数值算法。在其最简单的形式中，该算法将一个序列中的值进行累加：

```
template<typename In, typename T>
// 要求 Input_iterator<T>() && Number<T>()
T accumulate(In first, In last, T init)
{
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

给定初始值 `init`，该算法将序列 `[first:last)` 中的每个值加到 `init` 上，并将和返回。`init` 通常被称为累加器。例如：

```
int a[] = { 1, 2, 3, 4, 5 };
cout << accumulate(a, a+sizeof(a)/sizeof(int), 0);
```

这段代码将打印 15，即  $0+1+2+3+4+5$ （0 是初始值）。显然，`accumulate()` 能够被用于所有类型的序列：

```
void f(vector<double>& vd, int* p, int n)
{
    double sum = accumulate(vd.begin(), vd.end(), 0.0);
    int sum2 = accumulate(p,p+n,0);
}
```

结果（和）的类型与 `accumulate()` 用来保存累加器的变量的类型一致。这带来了一定的灵活性，这可能是十分重要的，例如：

```
void g(int* p, int n)
{
    int s1 = accumulate(p, p+n, 0);           // 累加到一个 int
    long s1 = accumulate(p, p+n, long{0});   // 累加 int 到 long
    double s2 = accumulate(p, p+n, 0.0);     // 累加 int 到 double
}
```

在一些计算机上 `long` 的有效位数要比 `int` 更多。与 `int` 型相比，`double` 能够表示更大范围的数，但可能精度更差。我们将在第 24 章中再讨论范围和精度在数值计算中所起的作用。

将保存结果的变量用作初始值是一种常见的指明累加器类型的方法：

```
void f(vector<double>& vd, int* p, int n)
{
    double s1 = 0;
    s1 = accumulate(vd.begin(), vd.end(), s1);
    int s2 = accumulate(vd.begin(), vd.end(), s2);    // 糟糕!
    float s3 = 0;
    accumulate(vd.begin(), vd.end(), s3);          // 糟糕!
}
```

记住：要初始化累加器并将 `accumulate()` 的结果保存在这个变量中。在本例中，`s2` 在初始化之前就用作了初始化器，因此算法的结果是未定义的。我们将 `s3` 传递给 `accumulate()`（传值方式，参见 8.5.3 节），但算法结果并未被保存，这只是浪费时间。

## 16.5.2 泛化 `accumulate()`

基本的三参数 `accumulate()` 版本执行累加运算。但是，我们可能还想在序列上执行很多其他有用的运算，例如乘法和减法。为此，STL 提供了另一个四参数的 `accumulate()` 版本，允许我们指定要执行的运算：

```
template<typename In, typename T, typename BinOp>
// 要求 Input_iterator<In>() && Number<T>()
//      && Binary_operator<BinOp, Value_type<In>, T>()
T accumulate(In first, In last, T init, BinOp op)
{
    while (first!=last) {
        init = op(init, *first);
        ++first;
    }
    return init;
}
```

任何接受两个累加器类型实参的操作均能用于这一版本的 `accumulate()`。例如：

```
vector<double> a = { 1.1, 2.2, 3.3, 4.4 };
cout << accumulate(a.begin(), a.end(), 1.0, multiplies<double>());
```

这段代码将打印 35.1384，即  $1.0 \times 1.1 \times 2.2 \times 3.3 \times 4.4$ （1.0 为初始值）。这里提供的二元运算符 `multiplies<double>()` 是一个实现乘法运算的标准库函数对象；`multiplies<double>` 实现 `double` 的乘法；`multiplies<int>` 实现 `int` 的乘法，等等。还有一些其他的二元函数对象：`plus`（加法），`minus`（减法），`divides`，`modulus`（取余）。这些对象均在 `<functional>` 中定义（见附录 C.6.2）。

注意，为了计算浮点数的积，初始值显然应设为 1.0。

如 `sort()` 例子（见 16.4.2 节）中所示，我们常常对类对象中包含的数据更感兴趣，而不仅仅是普通内置类型。例如，给定物品的单位价格和单位数，我们可能想要计算所有物品的价值总和：

```
struct Record {
    double unit_price;
    int units;        // 销售的单位数
    // ...
};
```

我们可以让 `accumulate` 的运算符从一个 `Record` 元素中抽取 `units`，将其与单位价格相乘



并加到累加器中：

```
double price(double v, const Record& r)
{
    return v + r.unit_price * r.units; // 计算价格并累加
}

void f(const vector<Record>& vr)
{
    double total = accumulate(vr.begin(), vr.end(), 0.0, price);
    // ...
}
```

我们在这里很“懒惰”，使用了函数而不是函数对象——这仅仅是为了展示可以这么做。

我们倾向于优先选用函数对象：

- 如果需要在调用之间保存值；
- 或者，如果代码很短以致内联化会带来很大不同（至多是几个原语操作）。

基于第二个原因，我们应该在本例中选用函数对象。



### 试一试

定义一个 `vector<Record>`，用你所选择物品的四个记录将其初始化，并用上面的函数计算物品的总价值。

## 16.5.3 内积

给定两个向量，将它们对应位置的元素相乘并将结果累加，这一运算称为向量的内积（inner product），内积在很多领域都十分有用（例如物理和线性代数，参见 24.6 节）。如果你更喜欢代码而不是文字，下面就是 STL 版本：

```
template<typename In, typename In2, typename T>
// 要求 Input_iterator<In> && Input_iterator<In2>
//    && Number<T> (参见 14.3.3 节)
T inner_product(In first, In last, In2 first2, T init)
// 注意：这就是我们将两个向量相乘（生成一个标量）的方法
{
    while(first!=last) {
        init = init + (*first) * (*first2); // 元素对相乘
        ++first;
        ++first2;
    }
    return init;
}
```


这段代码将内积的概念推广到任意元素类型的任何序列。以股票市场指数为例。在股票市场中，每一上市公司都会被分配一个“权重”。例如，在道琼斯工业指数中，我们看到 Alcoa 公司的最新权重为 2.4808。为了获得当前的指数值，我们将每个公司的股票价格与其权重相乘，并将所得所有加权价格相加。显然，这就是价值和权重的内积。例如：


```
// 计算道琼斯工业指数
vector<double> dow_price = { // 每个公司的股票价格
    81.86, 34.69, 54.45,
    // ...
};
```

```
list<double> dow_weight = {           // 每个公司在指数中的权重
    5.8549, 2.4808, 3.8940,
    // ...
};

double dji_index = inner_product( // (权重, 值) 对相乘并累加
    dow_price.begin(), dow_price.end(),
    dow_weight.begin(),
    0.0);

cout << "DJI value " << dji_index << '\n';
```

注意，`inner_product()` 处理两个序列。但它只接受三个参数，对于第二个序列只描述了  开始位置。算法假设第二个序列包含的元素个数要等于或多于第一个序列。如果这一假设不成立，将产生运行时错误。对于 `inner_product()` 而言，第二个序列包含更多元素是没问题的，那些“多余的元素”将简单地不予处理。

两个序列不需要具有相同的类型，元素类型也不必相同。为了展示这一点，我们使用了  `vector` 存储价格、用 `list` 存储权重。


#### 16.5.4 泛化 `inner_product()`

`inner_product()` 可以像 `accumulate()` 那样泛化，但它需要两个额外参数：一个用于将累加器与新值组合起来（与 `accumulate()` 完全一样），另一个用于组合元素值对：

```
template<typename In, typename In2, typename T, typename BinOp,
typename BinOp2>
// 要求 Input_iterator<In> && Input_iterator<In2> && Number<T>
// && Binary_operation<BinOp, T, Value_type<In>()
// && Binary_operation<BinOp2, T, Value_type<In2>()
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
    while(first!=last) {
        init = op(init, op2(*first, *first2));
        ++first;
        ++first2;
    }
    return init;
}
```

在 16.6.3 节中，我们将回到道琼斯的例子，给出一个更优雅的解决方案，其中就使用了这个泛化的 `inner_product()`。

## 16.6 关联容器

除了 `vector` 之外，最有用的标准库容器恐怕就是 `map` 了。一个 `map` 就是一个（键， 值）对的有序序列，你可以基于一个关键字在其中查找对应的值；例如 `my_phone_book["Nicholas"]` 应该是 Nicholas 的电话号码。在流行度的竞争中，`map` 唯一的潜在竞争对手是 `unordered_map`（见 16.6.4 节），它是一种针对字符串关键字优化过的 `map`。类似 `map` 和 `unordered_map` 的数据结构有很多名字，例如关联数组（associative array）、哈希表（hash table）和红黑树（red-black tree）等。流行的和有用的概念似乎总是有很多名称。在标准库中，我们将这类数据结构统称为关联容器（associative container）。

标准库提供了 8 个关联容器：

关联容器	
map	(键, 值) 对的有序容器
set	关键字的有序容器
unordered_map	(键, 值) 对的无序容器
unordered_set	关键字的无序容器
multimap	关键字可以出现多次的 map
multiset	关键字可以出现多次的 set
unordered_multimap	关键字可以出现多次的 unordered_map
unordered_multiset	关键字可以出现多次的 unordered_set

这些容器可以在 `<map>`、`<set>`、`<unordered_map>` 和 `<unordered_set>` 中找到。

### 16.6.1 map

思考一个概念上简单的例子：建立一个单词在文本中出现次数的列表。最明显的方式是维护一个我们看到单词的列表并维护每个单词遇到的次数。当我们读入一个新单词时，首先查看是否曾经见到过它；如果见过，将其计数器加一；否则，将它插入列表并赋值为 1。我们可以使用 `list` 或 `vector` 来完成它，但是我们不得不为读取的每个单词进行一次查找。这可能很慢。`map` 存储关键字的方式令判断关键字是否存在变得很容易，这使得搜索部分在我们的任务中变得微不足道：

```
int main()
{
    map<string,int> words;    // 维护(单词,频率)对

    for (string s; cin>>s; )
        ++words[s];        // 注意:用 string 作为 words 的下标

    for (const auto& p : words)
        cout << p.first << ": " << p.second << '\n';
}
```

这个程序中真正有趣的部分是 `++words[s]`。正如我们在 `main()` 第一行中看到的，`words` 是一个 `(string, int)` 对的 `map`；也就是说，`words` 将 `string` 映射到 `int`。换句话说，给定一个 `string`，`words` 可以令我们访问对应的 `int`。当我们用 `string`（保存输入的单词）来对 `words` 进行下标操作时，`words[s]` 得到对应 `s` 的 `int` 的引用。让我们来看一个具体的例子：

```
words["sultan"]
```

✂ 如果我们没见到过字符串 `"sultan"`，`"sultan"` 将会被插入 `words`，伴随着 `int` 的默认值 0。现在，`words` 会有一项 `( "sultan", 0 )`。因此结果就是，如果我们之前未见到过 `"sultan"`，则 `++words["sultan"]` 会将值 1 与字符串 `"sultan"` 相关联。详细来说：`map` 会发现 `"sultan"` 不在其中，它插入一个 `( "sultan", 0 )` 对，然后 `++` 会将该值加一，得到 1。

我们现在回过头来再看这个程序：`++words[s]` 得到我们输入的每个单词，并将其对应的值加一。当新单词第一次出现时，它会得到值 1。现在，这个循环的含义就清晰了：

```
for (string s; cin>>s; )
    ++words[s];        // 注意:用 string 作为 words 的下标
```

这个循环读取输入的每个单词（用空格分隔），并计算每个单词的出现次数。现在我

们要做的就是生成输出了。我们可以遍历一个映射，就像遍历其他 STL 容器那样。一个 `map<string,int>` 的元素是 `pair<string,int>`。每个 `pair` 的第一个元素名为 `first`，第二个元素名为 `second`，因此输出循环为

```
for (const auto& p : words)
    cout << p.first << ": " << p.second << "\n";
```

作为测试，我们可以将第 1 版《The C++ Programming Language》的开篇句子输入程序：


C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types.

我们得到输出

```
C: 1
C++: 3
C,: 1
Except: 1
In: 1
a: 2
addition: 1
and: 1
by: 1
defining: 1
designed: 1
details,: 1
efficient: 1
enjoyable: 1
facilities: 2
flexible: 1
for: 3
general: 1
is: 2
language: 1
language.: 1
make: 1
minor: 1
more: 1
new: 1
of: 1
programmer.: 1
programming: 3
provided: 1
provides: 1
purpose: 1
serious: 1
superset: 1
the: 3
to: 2
types.: 1
```

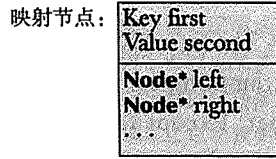
如果我们不想区分大小写字母或者希望去掉标点符号，我们可以这样做：参见习题 13。

## 16.6.2 map 概览

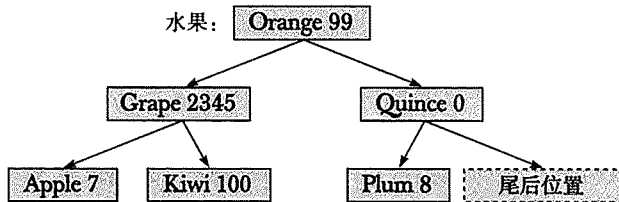
那么，映射是什么呢？映射的实现有很多种方式，但是 STL 实现映射通常采用平衡二 

叉搜索树，更具体一些——红黑树。我们将不会探究其细节，但是现在你知道了技术术语，这样，如果你想了解更多知识，就可以通过书籍或互联网来查找。

一棵树由多个节点构成（与链表由链接构成相似，参见 15.4 节）。一个 Node 保存一个关键字和对应的值，并且指向两个子节点。



这就是 `map<Fruit,int>` 在内存中的样子，假设我们插入了 (Kiwi, 100)、(Quince, 0)、(Plum, 8)、(Apple, 7)、(Grape, 2345) 和 (Orange, 99)：



若保存关键字值的 Node 成员的名字为 first，二叉搜索树的基本规则是：

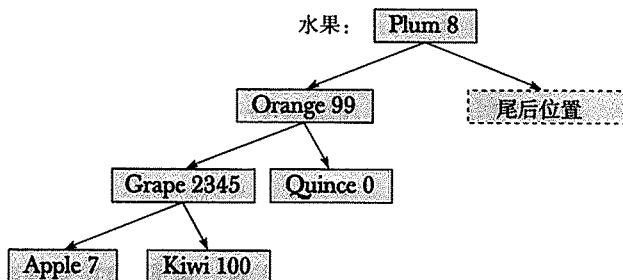
`left->first<first && first<right->first`

即，对每个节点，

- 它的左子节点的关键字小于本节点的关键字；
- 而且，本节点的关键字小于它的右子节点的关键字。

你可以对树中的每个节点验证这个规则是成立的。这允许我们“从根向下”搜索树。非常奇怪的是，在计算机科学文献中，树是从根向下生长的。在本例中，根节点是 (Orange, 99)。我们沿着树向下比较，直到发现要查找的值或它应该处于的位置。若一棵树的与根等距离的所有子树的节点数大致相等（如本例），那么这棵树被称为平衡的 (balanced)。平衡树最小化了一次搜索平均要访问的节点数。

一个 Node 可能还保存更多的数据，映射可以用之来保持树中节点的平衡。当一棵树中的每个节点的左、右子树点数大致相同时，这棵树就是平衡的。如果一棵有  $N$  个节点的树是平衡的，我们找到每个节点最多需要查找  $\log_2(N)$  个节点。这比我们在链表中从开始位置查找一个关键字，平均要查找  $N/2$  个节点的情况（这种线性查找的最坏情况是  $N$ ）要好得多。（参见 16.6.4 节。）例如，我们看一棵非平衡的树：



这棵树仍然遵守每个节点的关键字大于它的左子节点、小于它的右子节点的规则：

**left->first<first && first<right->first**

但是，这个版本的树是非平衡的，因此我们现在要经过三“跳”到达 Apple 和 Kiwi，而在平衡树中只需要两“跳”。对于有很多节点的树来说，这个差别可能非常巨大，因此用于实现 map 的树是平衡的。

使用 map 时并不需要理解树。这里只是做个合理的假设——专业人员至少了解所用工具的基础知识。我们必须了解的是由标准库提供的 map 的接口。下面是一个稍微简化过的版本：

```
template<typename Key, typename Value, typename Cmp = less<Key>>
    // 要求 Binary_opeartion<Cmp, Value>() (参见 14.3.3 节)
class map {
    // ...
    using value_type = pair<Key, Value>; // 管理 (Key, Value) 对的 map

    using iterator = sometype1;        // 类似指向树节点的指针
    using const_iterator = sometype2;

    iterator begin();                   // 指向首元素
    iterator end();                     // 指向尾后位置

    Value& operator[](const Key& k);    // 用 k 进行下标操作

    iterator find(const Key& k);        // 存在关键字为 k 的项吗?

    void erase(iterator p);             // 删除 p 指向的元素
    pair<iterator, bool> insert(const value_type&); // 插入一个 (key, value) 对
    // ...
};
```

你可以在 `<map>` 中找到真实的版本。你可以将迭代器想象成一个 `Node*`，但是你不能 ✕ 依赖使用这种特殊类型的自己的实现版本来实现迭代器。

显然，map 的接口与 vector 和 list (见 15.5 节和附录 C.4) 是很相似的。最大的不同是在遍历时，map 的元素类型为 `pair<Key, Value>`。这个类型是另一个有用的 STL 类型：

```
template<typename T1, typename T2>
struct pair {                               // std::pair 的简化版本
    using first_type = T1;
    using second_type = T2;

    T1 first;
    T2 second;
    // ...
};

template<typename T1, typename T2>
pair<T1, T2> make_pair(T1 x, T2 y)
{
    return {x, y};
}
```

我们从标准库复制了 pair 的完整定义及其有用的辅助函数 make\_pair()。

注意，当你对一个 map 进行遍历时，将按关键字定义的序访问元素。例如，如果我们 ✕ 对例子中的水果进行遍历，我们将得到：

(Apple,7) (Grape,2345) (Kiwi,100) (Orange,99) (Plum,8) (Quince,0)

我们插入水果的顺序无关紧要。

`insert()` 操作有一个奇怪的返回值，我们经常在简单的程序中忽略它。返回值包含一对迭代器，指向 (key, value) 元素，还包含一个 `bool` 值，如果这次 `insert()` 调用成功地插入了 (key, value) 对，则其值为 `true`。如果关键字已在映射中，则插入失败且返回的 `bool` 值为 `false`。



注意，通过提供第三个参数（映射声明中的 `Cmp`），你可以定义映射使用的序的含义。例如：

```
map<string, double, No_case> m;
```

`No_case` 定义不区分大小写的比较，参见 16.8 节。默认的序是由 `less<Key>` 定义的，表示“小于”。

### 16.6.3 另一个 `map` 实例

为了更好地体会 `map` 的用途，我们回到 16.5.3 节中的道琼斯工业指数的例子。只有当所有的权重与它们对应的名字出现在 `vector` 中相同位置时，这段代码才是正确的。这个前提是隐式的，很容易成为隐蔽错误的来源。有很多方法可以解决这个问题，但一个有吸引力的方法是将权重与其公司的股票代码保存在一起，例如 (“AA”, 2.4808)。“股票代码”是公司名称的缩写，用在需要简洁表示的地方。与此相似，我们可以将公司股票代码与其股票价格保存在一起，例如 (“AA”, 34.69)。最后，对于那些不经常与美国股票市场打交道的人，我们可以将公司股票代码与公司名称保存在一起，例如 (“AA”, “Alcoa Inc.”)。也就是说，我们维护三个相关值的映射。

首先，我们实现（代码，价格）映射：

```
map<string,double> dow_price = { // 道琼斯工业指数（代码，价格）；
                                // 最新报价请看
                                // www.djindexes.com
                                {"MMM",81.86},
                                {"AA",34.69},
                                {"MO",54.45},
                                // ...
                                };
```

接下来是（代码，权重）映射：

```
map<string,double> dow_weight = { // 道琼斯工业指数（代码，权重）
                                   {"MMM", 5.8549},
                                   {"AA",2.4808},
                                   {"MO",3.8940},
                                   // ...
                                   };
```

最后是（代码，名称）映射：

```
map<string,string> dow_name = { // 道琼斯工业指数（代码，名称）
                                   {"MMM","3M Co."},
                                   {"AA"] = "Alcoa Inc."},
                                   {"MO"] = "Altria Group Inc."},
                                   // ...
                                   };
```

通过这些映射，我们可以方便地提取各种信息。例如：

```
double alcoa_price = dow_price["AAA"]; // 从 map 读取值
double boeing_price = dow_price["BA"];

if (dow_price.find("INTC") != dow_price.end()) // 在 map 中查找表项
    cout << "Intel is in the Dow\n";
```

遍历映射是很容易的。我们只需记住关键字称为 `first`，而值称为 `second`：

```
// 对道琼斯工业指数中每家公司写入其股票价格
for (const auto& p : dow_price) {
    const string& symbol = p.first; // “股票代码”
    cout << symbol << '\t'
         << p.second << '\t'
         << dow_name[symbol] << '\n';
}
```

我们甚至可以直接使用映射来完成某些计算。特别是，我们可以计算出指数，就像我们在 16.5.3 节中所做的那样。我们可以从各自的映射中提取出股票价格和权重，并将它们相乘。我们可以很容易地编写一个函数，可对任意两个 `map<string,double>` 完成这个操作：

```
double weighted_value(
    const pair<string,double>& a,
    const pair<string,double>& b
) // 提取值并相乘
{
    return a.second * b.second;
}
```

现在，我们将这个函数加入 `inner_product()` 的泛化版本，并得到指数值：


```
double dji_index =
    inner_product(dow_price.begin(), dow_price.end(), // 所有公司
                 dow_weight.begin(), // 它们的权重
                 0.0, // 初始值
                 plus<double>(), // 加分（与往常一样）
                 weighted_value); // 提取股票价格和权重并相乘
```

为什么有人将这类数据保存在 `map` 中，而不是 `vector` 中呢？我们使用 `map` 的目的是令不同的值之间的联系显式表现出来。这是一个常见的原因。另一个原因是 `map` 会按关键字定义的序来保存它的元素。当我们遍历上面的 `dow` 时，我们按字母顺序输出股票代码；假如我们使用 `vector`，则需要自己进行排序。使用 `map` 的最常见的原因不过是我们希望基于关键字查找值。对于大的序列，使用 `find()` 来查找某些东西的速度远比在一个排序的结构（例如 `map`）中查找慢得多。

### 试一试

编译运行这个小例子。然后，添加几个你自己选择的公司，以及你自己选择的权重。

## 16.6.4 unordered\_map

为了在一个 `vector` 中找到一个元素，`find()` 需要检查所有的元素，从首元素到正确的 



元素或一直到末尾。其平均代价与  $\text{vector}(N)$  的长度成比例，我们称这个代价为  $O(N)$ 。

为了在一个  $\text{map}$  中找到一个元素，下标操作需要在树中从根节点开始到正确值的元素或一直到叶节点检查路径上所有元素。其平均代价与树的深度成比例。一棵有  $N$  个节点的平衡二叉树的最大深度为  $\log_2(N)$ ，代价为  $O(\log_2(N))$ 。 $O(\log_2(N))$ ——即与  $\log_2(N)$  成比例的代价——与  $O(N)$  相比实际上是非常好的：

$N$	15	128	1023	16 383
$\log_2(N)$	4	7	10	14

实际的代价将会依赖于我们多快查找到所要的值以及比较和迭代操作的代价有多大。通常，追踪指针（在  $\text{map}$  中查找所做的）的代价比递增一个指针（ $\text{find}()$  在  $\text{vector}$  中所做的）大得多。

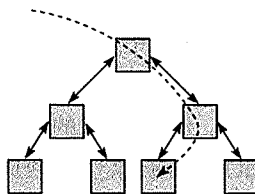
✂ 对于有些类型，特别是整数和字符串，我们甚至可以做得比  $\text{map}$  的树搜索更好。我们这里不深入细节，但其思路是给定一个关键字，我们可以计算其在  $\text{vector}$  中的索引。这个索引被称为一个哈希值（hash value），而使用这种技术的容器通常被称为哈希表（hash table）。应用中可能出现的关键字数量远大于哈希表中的位置数。例如，我们经常用一个哈希函数将数十亿个可能的字符串映射成 1000 个元素的  $\text{vector}$  中的索引。这可能有些棘手，但我们可以很好地处理它，这对实现大的映射特别有用。哈希表的主要优点是查找的平均代价接近常数且与表中的元素数量无关，即  $O(1)$ 。很明显，这对于大的映射来说是一个显著的优点，例如一个有 500 000 个 web 地址的映射。如果想获得有关哈希查找的更多知识，你可以查阅有关  $\text{unordered\_map}$  的文档（可在互联网中找到），或者有关数据结构的基础教材（查找“哈希表”和“哈希”）。

在一个（未排序）向量、一棵平衡二叉树和一个哈希表中的查找过程图示如下：

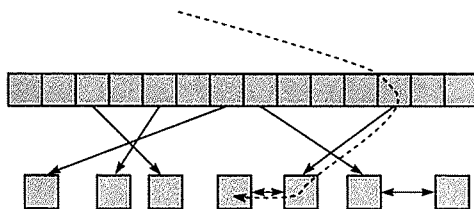
- 在未排序  $\text{vector}$  中查找：



- 在  $\text{map}$ （平衡二叉树）中查找：



- 在  $\text{unordered\_map}$ （哈希表）中查找：



STL `unordered_map` 是使用一个哈希表来实现的，正如 STL `map` 是使用一个平衡二叉树，STL `vector` 是使用一个矩阵来实现的一样。STL 的部分功用就是将这些数据存储和访问方法与算法一起纳入一个通用框架中。相应的经验法则是：

- 除非你有好的理由，否则使用 `vector`。
- 如果你需要基于值来进行查找（而且你的关键字类型有合理而高效的小于操作），这时使用 `map`。
- 如果你需要在一个大的映射中进行大量查找，并且你不需要有序的遍历（而且可以为你的关键字类型找到一个好的哈希函数），这时使用 `unordered_map`。

在这里，我们不会描述 `unordered_map` 的细节。我们可以将 `unordered_map` 与 `string` 或 `int` 类型的关键字共同使用，这方面与 `map` 完全一样，差别是当你遍历元素时，并不是有序访问元素。例如，我们可以重写 16.6.3 节中的道琼斯工业指数例子如下：

```
unordered_map<string,double> dow_price;

for (const auto& p : dow_price) {
    const string& symbol = p.first;    // “股票代码”
    cout << symbol << '\t'
         << p.second << '\t'
         << dow_name[symbol] << '\n';
}
```

现在，在 `dow` 中查找的速度可能更快。但是，这个变化并不会很显著，这是因为在指数中只有 30 个公司。假如我们保存了纽约证券交易所中所有公司的股票价格，性能差异就可能显现出来了。但是，我们需要注意一个逻辑上的不同：遍历得到的输出将不会按字母顺序排列。

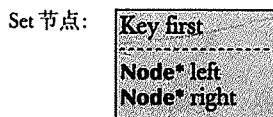
未排序的映射在 C++ 标准中是新内容，而且还远不是“一等成员”，因为它们是在技术报告而非标准中定义的。但现有编译器已广泛支持它们，即便不支持，通常也能看到它们的前身——名为 `hash_map` 之类的东西。

### 试一试

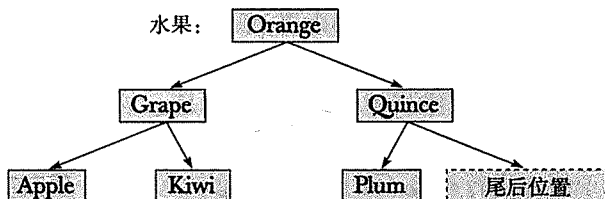
编写一个使用 `#include<unordered_map>` 的小程序。如果它不能正常工作，说明你的 C++ 实现未包含 `unordered_map`。如果你的 C++ 实现未提供 `unordered_map`，你需要下载一个可用的实现（例如参见 [www.boost.org](http://www.boost.org)）。

## 16.6.5 set

我们可以将 `set`（集合）看作一个对其值不感兴趣的 `map`，或干脆看作一个没有值的 `map`。我们可以图示一个 `set` 如下：



我们可以将 `map` 的例子（见 16.6.2 节）中的水果用 `set` 表示，如下图所示：



集合有什么用？如果我们看见一个值，碰巧有很多问题需要我们记住。跟踪哪种水果有货（与价格无关）就是一个例子，构造一个字典是另一个例子。一个稍微不同的使用风格是用集合保存“记录”；即，元素是可能包含“大量”信息的对象——我们只需使用一个成员作为关键字。例如：

```

struct Fruit {
    string name;
    int count;
    double unit_price;
    Date last_sale_date;
    // ...
};

struct Fruit_order {
    bool operator()(const Fruit& a, const Fruit& b) const
    {
        return a.name < b.name;
    }
};

set<Fruit, Fruit_order> inventory; // 用 Fruit_order(x,y) 比较水果
  
```

这里，我们再次看到使用函数对象是如何显著扩大 STL 组件的应用范围的。

✘ 由于 set 没有值类型，因此它也不支持下标操作（operator[]()）。我们必须使用“链表操作”，例如 insert() 和 erase()。不幸的是，map 和 set 都不支持 push\_back()——原因很明显：set 不是由程序员决定在哪里插入新值，取而代之使用 insert()。例如：

```

inventory.insert(Fruit("quince",5));
inventory.insert(Fruit("apple",200,0.37));
  
```

set 优于 map 的一点是你可以直接使用从迭代器得到的值。由于不像 map（见 16.6.3 节）那样有（键，值）对，解引用操作直接得到一个元素类型的值：

```

for (auto p = inventory.begin(), p!=inventory.end(); ++p)
    cout << *p << '\n';
  
```

当然，假设你已经为 Fruit 定义了 <<。或者我们可以写出如下等价代码：

```

for (const auto& x : inventory)
    cout << x << '\n';
  
```

## 16.7 拷贝

在 16.2 节中，我们认为 find() 是“最简单的有用算法”。当然，这一点可以讨论。很多简单算法都是有用的——甚至其中有些编写起来有些过于简单了。当你可以使用其他人编写和调试好的代码时，为什么要费力编写新的代码？当谈及简单性和有效性时，copy() 可以与

find() 媲美。STL 提供了三个版本的拷贝：

拷贝操作	
copy(b,e,b2)	将 [b:e) 拷贝到 [b2:b2+(e-b))
unique_copy(b,e,b2)	将 [b:e) 拷贝到 [b2:b2+(e-b))，禁止拷贝相邻的相同元素
copy_if(b,e,b2,p)	将 [b:e) 拷贝到 [b2:b2+(e-b))，但是仅拷贝满足谓词 p 的元素

### 16.7.1 基本拷贝算法

基本拷贝算法的定义如下：

```
template<typename In, typename Out>
// 要求 Input_iterator<In>() && Output_iterator<Out>()
Out copy(In first, In last, Out res)
{
    while (first!=last) {
        *res = *first;    // 拷贝元素
        ++res;
        ++first;
    }
    return res;
}
```

给定一对迭代器，copy() 将一个序列拷贝到另一个序列，目的序列用一个迭代器指明首元素。例如：

```
void f(vector<double>& vd, list<int>& li)
// 将一个 int 的 list 拷贝到一个 double 的 vector
{
    if (vd.size() < li.size()) error("target container too small");
    copy(li.begin(), li.end(), vd.begin());
    // ...
}
```

注意，copy() 的输入序列类型可以与输出序列类型不同。这是 STL 算法的一种有用泛化：它们可用于各种序列，而无须对其实现做不必要的假设。我们要记得检查在输出序列中是否有足够的空间以保存拷贝来的元素。检查空间的大小是程序员的责任。STL 算法的设计目标是最大的通用性和最佳的性能，它们（默认）没有做范围检查和其他代价昂贵的测试来保护用户。有时候，你可能希望它们做这些检查，但是当你想进行检查时，你可以像上面代码那样自己来完成。

### 16.7.2 流迭代器

你可能听到过短语“拷贝到输出”和“从输入拷贝”。这种思考方式对某种形式的 I/O 来说是很好、很有用的，我们确实可以用 copy() 做这些事情。

记住，一个序列是这样的东西：

- 它有开始和结尾；
- 我们可以用 ++ 移动到下一个元素；
- 我们可以用 \* 得到当前元素的值。

我们可以很容易地用这种方式表示输入和输出流。例如：

```
ostream_iterator<string> oo(cout); // 向 *oo 赋值就是写到 cout

*oo = "Hello, "; // 表示 cout << "Hello,"
++oo; // "准备好下一个输出操作"
*oo = "World!\n"; // 表示 cout << "World!\n"
```

你可以想象如何来实现它。标准库提供了一个 `ostream_iterator` 类型，就可以这样工作：`ostream_iterator<T>` 是一个迭代器，你可以用它写入类型为 `T` 的值。

类似地，标准库提供了 `istream_iterator<T>` 类型用于读取类型为 `T` 的值：

```
istream_iterator<string> ii(cin); // 读取 *ii 就是从 cin 读取一个 string

string s1 = *ii; // 表示 cin>>s1
++ii; // "准备好下一个输入操作"
string s2 = *ii; // 表示 cin>>s2
```

通过 `ostream_iterator` 和 `istream_iterator`，我们可以对自己的 I/O 使用 `copy()`。例如，我们可以实现一个“快速和混乱的”字典，如下所示：

```
int main()
{
    string from, to;
    cin >> from >> to; // 获取源文件和目标文件名

    ifstream is (from); // 打开输入流
    ofstream os (to); // 打开输出流

    istream_iterator<string> ii (is); // 创建输入流迭代器
    istream_iterator<string> eos; // 输入哨兵
    ostream_iterator<string> oo {os, "\n"}; // 创建输出流迭代器

    vector<string> b (ii, eos); // b 是一个 vector，初始化用于输入
    sort(b.begin(), b.end()); // 对缓冲区进行排序
    copy(b.begin(), b.end(), oo); // 拷贝缓冲区到输出
}
```

迭代器 `eos` 是表示“输入结束”的流迭代器。当一个 `istream` 到达输入结束（经常被表示为 `eof`），它的 `istream_iterator` 将等于默认的 `istream_iterator`（这里称为 `eos`）。

注意，我们使用一对迭代器来初始化 `vector`。作为一个容器的初始化器，一对迭代器 (`a`, `b`) 表示“将序列 [`a:b`] 读取到容器”。自然地，我们使用的一对迭代器是 (`ii, eos`)——输入的开始与结束。这令我们不必使用 `>>` 和 `push_back()`。我们强烈建议不要使用下面的替代方案。

```
vector<string> b(max_size); // 不要猜测输入的大小！
copy(ii, eos, b.begin());
```

那些试图猜测输入的最大规模的人，通常会发现他们低估了输入规模，从而遇到严重的问题——缓冲区溢出，无论对于他们自己还是他们的用户都是很严重的问题。这种溢出也是安全问题的一个来源。

### 试一试

首先，编译上面的程序令其正确运行，用一个小文件来测试它，例如一个包含几百个单词的文件。然后，尝试我们着重强调不推荐的猜测输入规模的版本，观察当输入缓冲区 `b` 溢出时发生什么。注意，最坏的情况是在特定例子中溢出没有导致任何错误，这样你就可能试图将它交付给用户。

在这个小程序中，我们读取单词然后进行排序。这在当时看来是一个明显的解决方案，但是我们为什么要将单词放在“错误的位置”，以至于随后我们不得不进行排序？更糟糕的是，我们发现一个单词在输入中出现几次，我们就会保存和打印它几次。

我们可以用 `unique_copy()` 代替 `copy()` 来解决后一个问题。`unique_copy()` 不会重复拷贝相同的值。例如，如果使用普通的 `copy()`，输入

```
the man bit the dog
```

程序会生成

```
bit
dog
man
the
the
```

如果我们使用 `unique_copy()`，程序将会输出

```
bit
dog
man
the
```

这些换行是从哪里来的？带有分隔符的输出是很常见的，`ostream_iterator` 的构造函数允许你（可选的）指定在每个值之后打印一个字符串：

```
ostream_iterator<string> oo {os, "\n"}; // 创建输出流迭代器
```

很明显，对于供人类阅读的输出来说，换行分隔符是很常见的选择，但是也许我们喜欢使用空格作为分隔符呢？可以编写代码如下：

```
ostream_iterator<string> oo {os, " "}; // 创建输出流迭代器
```

这将会生成输出

```
bit dog man the
```

### 16.7.3 使用 `set` 保持顺序

有一个更容易的方式来得到上面那样的输出，使用 `set` 而不是 `vector`：

```
int main()
{
    string from, to;
    cin >> from >> to; // 获取源文件和目标文件名

    ifstream is {from}; // 创建输入流
    ofstream os {to}; // 创建输出流

    set<string> b (istream_iterator<string>(is), istream_iterator<string>{});
    copy(b.begin(), b.end(), ostream_iterator<string>(os, " ")); // 拷贝缓冲区至输出
}
```

当我们将值插入一个 `set` 时，重复的值被忽略掉。而且，`set` 中的元素是按顺序保存的，因此不需要进行排序。通过使用正确的工具，大多数任务很容易完成。

### 16.7.4 `copy_if`

`copy()` 算法进行无条件拷贝。`unique_copy()` 算法禁止拷贝相同的相邻元素。第三种拷贝

算法只拷贝谓词为真的元素：

```
template<typename In, typename Out, typename Pred>
// 要求 Input_iterator<In>() && Output_iterator<Out>() &&
// Predicate<Pred, Value_type<In>>()
Out copy_if(In first, In last, Out res, Pred p)
// 拷贝满足谓词的元素
{
    while (first!=last) {
        if (p(*first)) *res++ = *first;
        ++first;
    }
    return res;
}
```

使用 16.4 节中的 `Larger_than` 函数对象，我们可以找到一个序列中大于 6 的所有元素，如下所示：

```
void f(const vector<int>& v)
// 拷贝值大于 6 的所有元素
{
    vector<int> v2(v.size());
    copy_if(v.begin(), v.end(), v2.begin(), Larger_than(6));
    // ...
}
```

⚠ 由于我犯的一个错误，这个算法错失进入 1998 ISO 标准的机会。这个错误现在已经被补救了，但是你仍然可以找到没有 `copy_if` 的 C++ 实现。如果是这样，请使用本节中的定义。

## 16.8 排序和搜索

✂ 我们经常希望自己的数据是有序的。为达到这个目的，我们可以使用一个能维护顺序的数据结构，例如 `map` 或 `set`，或进行排序。在 STL 中，最常见和有用的排序操作是 `sort()`，我们已经使用过多次了。在默认情况下，`sort()` 使用 `<` 作为排序标准，但是我们可以提供自己的标准：

```
template<typename Ran>
// 要求 Random_access_iterator<Ran>()
void sort(Ran first, Ran last);

template<typename Ran, typename Cmp>
// 要求 Random_access_iterator<Ran>()
// && Less_than_comparable<Cmp, Value_type<Ran>>()
void sort(Ran first, Ran last, Cmp cmp);
```

作为一个基于用户指定规则进行排序的例子，我们将介绍如何进行不考虑大小写的字符串排序：

```
struct No_case { // lowercase(x) < lowercase(y)?
    bool operator()(const string& x, const string& y) const
    {
        for (int i = 0; i<x.length(); ++i) {
            if (i == y.length()) return false; // y<x
            char xx = tolower(x[i]);
            char yy = tolower(y[i]);
            if (xx<yy) return true; // x<y
        }
    }
};
```


```

        if (yy<xx) return false;           // y<x
    }
    if (x.length()==y.length()) return false; // x==y
    return true;                          // x<y (x 中字符数更少)
}
};

void sort_and_print(vector<string>& vc)
{
    sort(vc.begin(),vc.end(),No_case());


    for (const auto& s : vc)
        cout << s << '\n';
}

```

一旦一个序列已排序，我们不再需要用 `find()` 从开始位置进行搜索；我们可以利用这种  序进行二分搜索。二分搜索的基本工作原理如下：

假设我们在查找值  $x$ ，查看中间元素：

- 如果元素的值等于  $x$ ，我们已经找到它！
- 如果元素的值小于  $x$ ，则值等于  $x$  的任何元素必然位于右边，因此我们查找右半部分（在这半部分进行二分查找）。
- 如果元素的值大于  $x$ ，则值等于  $x$  的任何元素必然位于左边，因此我们查找左半部分（在这半部分进行二分查找）。
- 如果我们已经到达最后一个元素（向左或向右），也没有找到  $x$ ，那么没有等于值  $x$  的元素。

对于更长的序列，二分搜索比 `find()`（线性搜索）的速度更快。二分搜索的标准库算法  是 `binary_search()` 和 `equal_range()`。我们说的“更长”的含义是什么呢？这要视具体情况而定，但即使序列中只有 10 个元素，也足以体现出 `binary_search()` 相对于 `find()` 的优势。对于一个有 1000 个元素的序列，`binary_search()` 可能要比 `find()` 快 200 倍，因为其代价为  $O(\log_2(N))$ ，参见 16.6.4 节。


`binary_search` 算法有两种变形：

```

template<typename Ran, typename T>
bool binary_search(Ran first, Ran last, const T& val);

template<typename Ran, typename T, typename Cmp>
bool binary_search(Ran first, Ran last, const T& val, Cmp cmp);

```


这些算法要求和假设输入序列是已排序的。如果未排序，“有趣的事情”如无限循环就  可能会发生。`binary_search()` 简单地告诉我们一个值是否存在：

```

void f(vector<string>& vs)    // vs 已排序
{
    if (binary_search(vs.begin(),vs.end(),"starfruit")) {
        // vector 中有杨桃
    }

    // ...
}

```

因此，如果我们只关心一个值是否在序列中，`binary_search()` 是理想的。如果我们还关  心找到的元素，可以使用 `low_bound()`、`upper_bound()` 或 `equal_range()`（参见附录 C.5.4 和



23.4 节)。有些情况下我们关心找到的是哪个元素，原因通常是对象包含更多信息而不仅仅是一个关键字，而很多元素可能具有相同的关键字，或者是我们希望找到符合搜索标准的元素。

## 16.9 容器算法

到目前为止，我们都是用元素序列来定义标准库算法。序列用迭代器指明：一个输入序列定义为一对迭代器 [b:e)，其中 b 指向序列首元素，e 指向序列尾元素之后位置（见 15.3 节）。一个输出序列简单地用一个迭代器指定，该迭代器指向序列的首元素。例如：

```
void test(vector<int> &v)
{
    sort(v.begin(),v.end()); // 对从 v.begin() 到 v.end() 的 v 的元素进行排序
}
```

这种方式很好、也很通用。例如，我们可以排序 vector 的一半内容：

```
void test(vector<int> &v)
{
    sort(v.begin(),v.begin()+v.size()); // 排序 v 的前一半元素
    sort(v.begin()+v.size(),v.end()); // 排序 v 的后一半元素
}
```

但是，指明元素范围有些啰嗦，而大多数情况下，我们需要排序整个 vector 而不是一半。因此，大多数情况下，我们希望这样编写代码：

```
void test(vector<int> &v)
{
    sort(v); // 排序 v
}
```

标准库未提供 sort() 的这种变形，但我们可以自己定义：

```
template<typename C> // 要求 Container<C>()
void sort(C& c)
{
    std::sort(c.begin(),c.end());
}
```

实际上，我们发现这个版本如此有用，因此将其加入到了 std\_lib\_facilities.h 中。

像这样可以很容易地处理输入序列，但为了保持简单性，我们倾向于还是保持返回类型为迭代器。例如：

```
template<typename C, typename V> // 要求 Container<C>()
Iterator<C> find(C& c, Val v)
{
    return std::find(c.begin(),c.end(),v);
}
```

Iterator<C> 自然是 C 的迭代器类型。

### 简单练习

在每一步操作（每个练习）之后打印 vector。

1. 定义一个 struct Item{string name; int iid; double value; /\*...\*/};，创建一个 vector<item> 类型的对象 vi，读取来自一个文件中的 10 个 Item 填入 vi。

- 按 name 对 vi 排序。
- 按 iid 对 vi 排序。
- 按 value 对 vi 排序，按 value 的降序打印（即先打印最大的值）。
- 插入 Item("horse shoe", 99, 12.34) 和 Item("Canon S400", 9988, 499.95)。
- 按 name 指定两个 Item，从 vi 中删除（擦除）它们。
- 按 iid 指定两个 Item，从 vi 中删除（擦除）它们。
- 采用 list<Item> 而不是 vector<Item> 重复上述练习。

现在尝试 map：

- 定义一个 map<string, int> 类型的对象 msi。
- 插入 10 个（名字，值）对，例如，msi["lecture"] = 21。
- 输出（名字，值）对到 cout，输出的格式由你自行定义。
- 删除 msi 中的（名字，值）对。
- 编写一个函数，该函数能够从 cin 中读取值对并将其存入 msi 之中。
- 从输入读入 10 个值对，并将它们存入 msi 中。
- 将 msi 的元素写入 cout。
- 输出 msi 中（整型）数值的总和。
- 定义一个 map<int, string> 类型的对象 mis。
- 将 msi 中的值存入 mis；即，如果 msi 的元素为 ("lecture", 21)，则 mis 应具有元素 (21, "lecture")。
- 输出 mis 的元素到 cout。

更多 vector 练习：

- 从一个文件中读入一些浮点值（至少 16 个），并将其存入一个 vector<double> 类型的对象 vd 之中。
- 输出 vd 到 cout。
- 定义一个 vector<int> 类型的对象 vi，且 vi 具有的元素数量与 vd 相同；将 vd 的元素拷贝至 vi 之中。
- 输出 (vd[i], vi[i]) 值对到 cout，每行输出一个值对。
- 输出 vd 元素的总和。
- 输出 vd 元素总和与 vi 元素总和的差值。
- 标准库中有一个称为 reserve 的算法，接受一个序列（由一对迭代器定义）作为参数；反转 vd，并输出 vd 到 cout。
- 计算 vd 中元素的平均值，并将结果输出。
- 定义一个 vector<double> 类型的对象 vd2，并将 vd 中所有值低于（小于）平均值的元素拷贝至 vd2 之中。
- 对 vd 进行排序，并输出 vd。

## 思考题

- 有用的 STL 算法的例子有哪些？
- find() 有什么用途？至少给出五个例子。
- count\_if() 有什么用途？

4. `sort(b,e)` 的排序标准是什么?
5. STL 算法如何将一个容器作为其输入参数?
6. STL 算法如何将一个容器作为其输出参数?
7. STL 算法通常如何表示“未找到”或“失败”?
8. 什么是函数对象?
9. 函数对象与函数之间有哪些区别?
10. 什么是断言?
11. `accumulate()` 有什么用途?
12. `inner_product()` 有什么用途?
13. 什么是关联容器? 至少给出五个例子。
14. `list` 是一个关联容器吗? 为什么不是?
15. 二叉树的基本序性质是什么?
16. 对于一棵树而言, 对其进行平衡意味着什么?
17. `map` 的每一元素占用了多少空间?
18. `vector` 的每一元素占用了多少空间?
19. 当可用一个 (有序的) `map` 时, 为什么我们还会使用 `unordered_map`?
20. `set` 与 `map` 有何区别?
21. `multi_map` 与 `map` 有何区别?
22. 当我们能够“仅仅编写一个简单的循环”时, 为什么还应使用 `copy()` 算法?
23. 什么是二分搜索?

## 术语

<code>accumulate()</code>	<code>find_if()</code>	searching (搜索)
<code>algorithm</code> (算法)	function object (函数对象)	sequence (序列)
application: () (应用: ())	generic (泛型)	set
associative container (关联容器)	hash function (哈希函数)	<code>sort()</code>
balanced tree (平衡树)	<code>inner_product()</code>	sorting (排序)
<code>binary_search()</code>	lambda	stream iterator (流迭代器)
<code>copy()</code>	<code>lower_bound()</code>	<code>unique_copy()</code>
<code>copy_if()</code>	<code>map</code>	<code>unordered_map</code>
<code>equal_range()</code>	predicate (断言)	<code>upper_bound()</code>
<code>find()</code>		

## 习题

1. 浏览本章所有内容, 并完成所有你未完成的“试一试”练习。
2. 找到一个可靠的 STL 文档资源, 列举所有标准库算法。
3. 实现 `count()` 并对其进行测试。
4. 实现 `count_if()` 并对其进行测试。
5. 如果我们不能通过返回 `end()` 表示“未找到”, 那应该怎么办? 重新设计并实现 `find()` 和 `count()`, 它们接受指向第一个和最后一个元素的迭代器。将新实现与标准版本进行比较。

6. 在 16.6.5 节的水果例子中，我们将 Fruit 对象拷贝至 set 中。如果我们不希望拷贝 Fruit 对象呢？我们可以用 `set<Fruit *>` 作为替代。然而，为了这么做，我们还需要为这个集合定义一个比较操作。通过 `set<Fruit *, Fruit_comparison>` 实现水果例子，并讨论两种实现之间的差别。
7. 为 `vector<int>` 编写一个二分搜索函数（不使用标准函数）。你可以选择任何你喜欢的接口。对该函数进行测试。你如何确认你的二分搜索是正确的？现在为 `list<string>` 编写一个二分搜索函数。对该函数进行测试。这两个二分搜索函数的相似程度如何？如果你不了解 STL，你觉得这两个二分搜索函数的相似程度会如何？
8. 修改 16.6.1 节中词频的例子，使它按频率顺序输出（而不是按字典序）。一个例子是，应该输出行 3: C++ 而不是 C++:3。
9. 定义一个 Order 类，该类包含（顾客）姓名、地址、数据与 `vector<Purchase>` 等成员。Purchase 是一个包含（产品）name、unit\_price 和 count 等成员类。定义一种将 Order 内容写入文件以及从文件中读入 Order 内容的机制。构建一个至少包含 10 个 Order 的文件，将该文件内容读入一个 `vector<Order>` 中，按（顾客）姓名进行排序，然后写回文件。构建另一个至少包含 10 个 Order 的文件，其中大约 1/3 内容与第一个文件相同，将文件内容读入一个 `list<Order>` 中，按（顾客）地址进行排序，然后写回文件。用 `std::merge()` 将两个文件合并，写入另一个文件。
10. 计算上一题中两个文件中订单的总价值。一个 Purchase 的价值为 `unit_price*count`。
11. 设计一个 GUI 接口能输入 Order 信息写入文件。
12. 设计一个 GUI 接口能查询 Order 文件；例如，“查找 Joe 的所有订单”，“查询文件 Hardware 中订单的总价值”以及“列出文件 Clothing 中所有订单”。提示：首先设计一个非 GUI 接口；然后，在此基础上实现 GUI 接口。
13. 编写一个程序，该程序能够对文本文件进行“清理”以便结果能用于一种单词查询程序；具体来说，用空白符替换标点符号，将单词转换为小写形式，用 do not（等等）替换 don't，以及去除复数形式（例如，ships 变为 ship）。不要野心太大。例如，确定复数形式通常而言是很困难的，因此如果你同时找到了单词 ship 和 ships，那么你简单删除 s 就可以了。将程序用于一个真实的至少包含 5000 个单词的文本文件（例如一篇研究论文）。
14. 编写一个程序（使用上一题程序的输出作为输入），该程序能够回答诸如“文件中单词 ship 出现了多少次？”“哪一个单词出现最频繁？”“文件中最长的单词是什么？”“哪个单词最短？”“列出所有以 s 开头的单词”“列出所有包含四个字符的单词”之类的问题。
15. 为上一题的程序设计一个 GUI 接口。

## 附言

STL 是 ISO C++ 标准库中关于容器和算法的部分。它提供了非常通用、灵活和有用的基本工具。它能够节省我们的很多工作：重新发明轮子可能是有趣的，但毫无效率可言。我们应该优先选用 STL 容器和基本算法，除非我们有充分的理由不这样做。而且，STL 是泛型编程的一个例子，它展示了具体问题及具体解决方案是如何构成一个有用且通用的工具集的。如果你需要处理数据——大部分程序的确需要——STL 提供了一个例子、一些思想以及一种有用的方法。

# 一个显示模型

直到 20 世纪 30 年代，世界才从黑白变成彩色的。

——Calvin's dad

本章描述了一个显示模型（GUI 的输出部分），并给出了使用方法和一些基本概念，如屏幕坐标、线和颜色等。Line、Lines、Polygon、Axis 和 Text 都是 Shape 的实例。Shape 是内存中的一个对象，我们可以将其显示在屏幕上并进行适当的操作。后面两章将进一步探讨这些类，第 18 章侧重类的实现，第 19 章侧重设计问题。

## 17.1 为什么要使用图形

为什么我们要用四章篇幅介绍图形，一章篇幅介绍 GUI（图形用户界面）？毕竟，本书主要讲述程序设计而不是图形学。实际上有很多非常有趣的话题，我们不可能一一讨论，在这里只是介绍与图形有关的内容。那么，为什么要使用图形呢？从本质上讲，图形学是一个学科，在其学习过程中，我们可以对软件设计、程序设计及其语言特性等重要的领域进行深入的探讨。

- 图形很有用。虽然在使用 GUI 时，更多工作仍在于程序设计而非图形，更多在于软件设计而非编写代码。但是，图形在很多领域都非常重要，例如，对于科学计算、数据分析或者任何一个量化问题，没有数据图形化功能是不可想象的。第 20 章给出了一个简单（但通用）的数据图形化工具。
- 图形很有趣。在多数计算领域中，一段代码的效果是很难立刻呈现出来的，即便最后代码没有 bug 了，也无法立刻看出来。这时，即使图形没有用，我们也会倾向于使用图形界面来获得即时效果。
- 图形提供了许多有趣的代码。通过阅读大量程序代码，可以找到一种对代码好坏的判断标准，就像要成为好的作家必须阅读大量书籍、文章和高质量的报纸一样。由于程序代码与屏幕显示有某种直接关系，图形代码比复杂程度接近的其他类别的代码更易于阅读。经过本章几分钟的介绍你就能够阅读图形代码，再经过第 18 章的学习就能编写图形代码。
- 图形设计实例非常丰富。实际上，设计和实现一个好的图形和 GUI 库是很困难的。图形领域中有非常丰富的具体、贴近实际的例子，可供学习设计策略和设计技术。通过相对较少的图形和 GUI 代码，就能展示包括类的设计、函数的设计、软件分层（抽象）、库的创建等在内的许多技术。
- 图形有利于解释面向对象程序设计的概念及其语言特征。与传闻相反，面向对象程序最初并不是为了图形化应用所设计的（参见第 22 章），但它确实很快就应用到图形领域中，而且图形化应用提供了一些非常易于理解的面向对象设计实例。
- 一些关键的图形学概念不是那么简单直白的。因此应该在教学中进行讲授，而不是

留待你靠自己的主动性（以及耐心）去学习理解。如果我们不展示图形和 GUI 程序的实现方法，你可能会认为它们是不可思议的魔法，这显然偏离了本书的一个基本目标。

## 17.2 一个基本显示模型

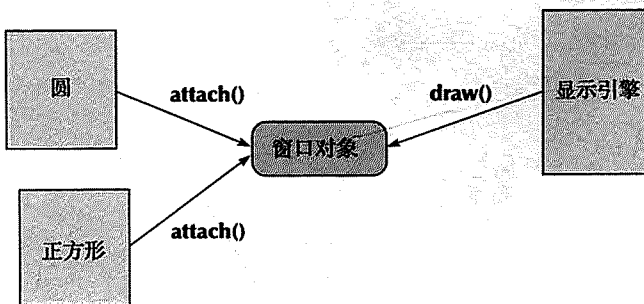
iostream 库是面向字符的输入输出流，用于处理数值序列或者书籍文本最为适合。其中，直接支持图形位置概念的仅有 `newline` 和 `tab` 控制字符。版面设计（排版、“标注”）语言，如 Troff、TeX、Word、HTTP、XML（及其配套的图形包），允许在一维字符流中嵌入颜色和二维位置等概念。例如：

```
<hr>
<h2>
Organization
</h2>
This list is organized in three parts:
<ul>
  <li><b>Proposals</b>, numbered EPddd, ...</li>
  <li><b>Issues</b>, numbered EIddd, ...</li>
  <li><b>Suggestions</b>, numbered ESddd, ...</li>
</ul>
<p>We try to ...
</p>
```

这段 HTML 代码指定了一个文档头（`<h2>...</h2>`）、一个包含若干列表项（`<li>...</li>`）的列表（`<ul>...</ul>`）和一个段落（`<p>`）。这里，我们省略了很多无关的代码。这类语言的关键点是，你可以在普通文本中表示版面的概念，但代码与屏幕上的显示内容之间不是直接关联的，而是由解释这些“标注”命令的程序来控制屏幕上的显示内容。这种技术极为简单，又极为有效（现在你所阅读的所有文档等基本都是这样生成的），但也有其缺点。

本章和之后四章介绍另外一种技术：一种直接在屏幕显示的图形及图形用户界面的概念。其基本概念先天就是图形化的（而且都是二维的，适应计算机屏幕的矩形区域），这些基本概念包括坐标、线、矩形和圆等。从编程的角度看，其目的是建立内存中的对象和屏幕图像的直接对应关系。

其基本模型如下：我们利用图形系统提供的基本对象（如线）组合出更复杂的对象；然后将这些对象“添加”到一个表示物理屏幕的窗口对象中；最后，用一个程序将我们添加到窗口上的对象显示在屏幕上。我们可以将这个程序看作屏幕显示本身，或者是一个“显示引擎”，或者是“我们的图形库”，或是“GUI 库”，甚至（幽默地）将其看作“在屏幕背后进行画图工作的小矮人”。



“显示引擎”负责在屏幕上绘制线，将文本串放置在屏幕上，为屏幕区域着色，等等。简单起见，我们将使用“我们的 GUI 库”甚至“系统”来表示显示引擎，虽然 GUI 库的功能不只是绘制对象。与我们的代码调用 GUI 库实现大部分图形功能一样，GUI 库将它的很多工作交由操作系统来完成。

## 17.3 第一个例子

我们的目标是定义一些类，能够用来创建可以在屏幕上显示的对象。例如，我们希望绘制一个由一系列相连的线构成的图形，下面程序给出了一个非常简单的版本：

```
#include "Simple_window.h" // 访问 window 库
#include "Graph.h"         // 访问图形库工具

int main()
{
    using namespace Graph_lib; // 图形库工具在 Graph_lib 中

    Point tl {100,100};        // 将起点置于窗口左上角

    Simple_window win {tl,600,400,"Canvas"}; // 生成一个简单的窗口

    Polygon poly;              // 生成一个 shape (一个 polygon)

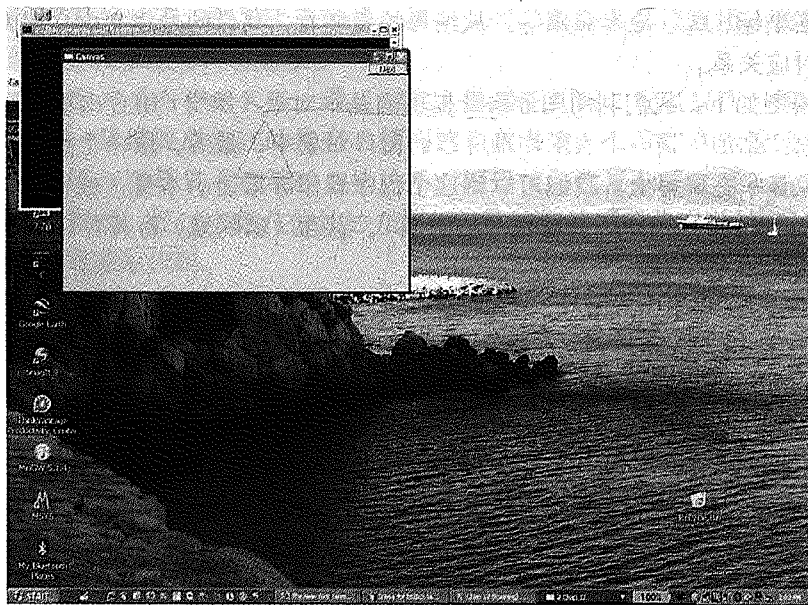
    poly.add(Point{300,200});  // 加入一个点
    poly.add(Point{350,100});  // 加入另一个点
    poly.add(Point{400,200});  // 加入第三个点

    poly.set_color(Color::red); // 调整 ploy 的属性

    win.attach (poly);         // 将 ploy 关联到窗口

    win.wait_for_button();     // 将控制权交给显示引擎
}
```

运行该程序，屏幕显示如下。



我们来逐行分析这个程序，看看它做了什么。它首先包含图形接口库的头文件：

```
#include "Simple_window.h" // 访问 window 库
#include "Graph.h"        // 访问 图形库工具
```

接着，在 main() 函数的开始处告知编译器在 Graph\_lib 中查找图形工具：

```
using namespace Graph_lib; // 图形库工具在 Graph_lib 中
```

然后，定义一个点作为窗口的左上角：

```
Point tl {100,100}; // 将起点置于窗口左上角
```

接下来在屏幕上创建一个窗口：

```
Simple_window win {tl,600,400,"Canvas"}; // 生成一个简单的窗口
```

我们使用 Graph\_lib 接口库中一个名为 Simple\_window 的类表示窗口，此处定义了一个名为 win 的 Simple\_window 对象，即 win 是 Simple\_window 类的变量。初始化列表中的值将窗口 win 的左上角设置为 tl，宽度和高度分别设置为 600 像素和 400 像素。我们随后会介绍更多细节，但此处的关键点就是通过给定宽度和高度来定义一个矩形。字符串 Canvas 用于标识该窗口，你可以在窗口框左上角的位置看到 Canvas 字样。

接下来，我们在窗口中放置一个对象：

```
Polygon poly; // 生成一个 shape( 一个 polygon)

poly.add(Point{300,200}); // 加入一个点
poly.add(Point{350,100}); // 加入另一个点
poly.add(Point{400,200}); // 加入第三个点
```

我们定义了一个多边形对象 poly，并向其添加顶点。在我们的图形库中，一个 Polygon 对象开始为空，可以向其中添加任意多个顶点。由于我们添加了三个顶点，因此得到了一个三角形。一个点是一个值对，给出了点在窗口内的  $x$  和  $y$ （水平和垂直）坐标。

纯粹是为了展示图形库的功能，我们接下来将多边形的边染为红色：

```
poly.set_color(Color::red); // 调整 poly 的属性
```

最后，我们将 poly 添加到窗口 win：

```
win.attach(poly); // 将 poly 关联到窗口
```

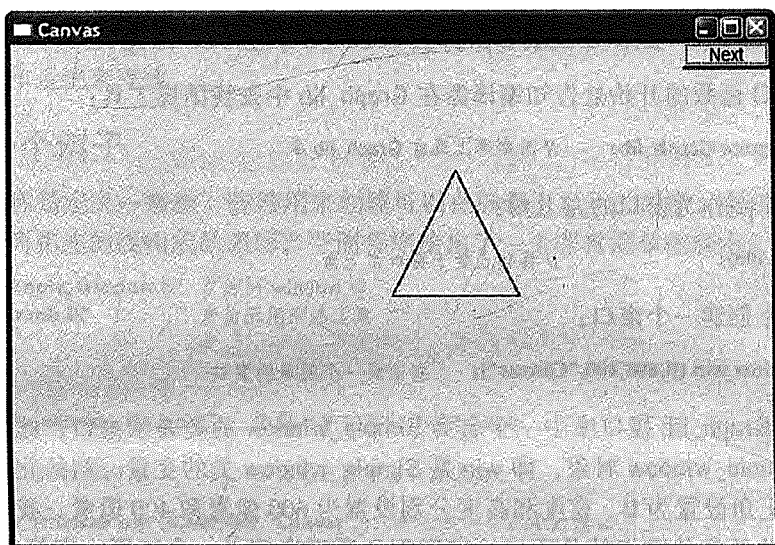
如果程序执行得不是那么快的话，你会注意到，到现在为止，屏幕上没有任何显示，是的，什么都没有。我们创建了一个窗口（确切地说，是 Simple\_window 类的一个对象），创建了一个多边形（名为 poly）并将其染为红色（Color::red），最后将其添加到窗口（名为 win），但我们还没有要求在屏幕上显示此窗口。显示操作由程序的最后一行代码来完成：

```
win.wait_for_button(); // 将控制权交给显示引擎
```

为了让 GUI 系统在屏幕上显示一个对象，你必须将控制权交给“系统”。wait\_for\_button() 就是完成这个功能，另外，它还等待用户按下（点击）窗口中的“Next”按钮，以便执行下面的程序。这样，在程序结束和窗口消失之前，你就有机会看到窗口中的内容。当你按下按钮后，程序会结束，关闭该窗口。



单独地看这个窗口，效果如下图所示：



你可能注意到，我们小小地“作弊”了一下。标记为“Next”的按钮是从哪里来的？实际上它是 `Simple_window` 类内置的。在第 21 章中，我们将会从 `Simple_window` 类过渡到“普通”的 `Window` 类，它不包含任何可能造成混淆的内置功能。那时，我们还会介绍如何编写代码来控制与窗口的交互。

在接下来的三章里，当希望逐阶段（一帧一帧）地显示信息时，我们将简单地使用“Next”按钮来实现显示画面的转换。

对于操作系统为每个窗口添加窗口框（frame），你应该非常熟悉了，但可能没有特别留意过。不过，本章和后面章节中的图片都是在微软 Windows 系统下生成的，因此你“免费”得到了窗口框右上角的三个常用按钮。这些按钮是很有用的：如果你的程序变得杂乱无章（在调试过程中确实有可能出现这种情况），可以点击 × 按钮结束程序。当你在其他系统上运行程序时，根据系统惯例的不同，添加的窗口框也可能有所不同。在上面的示例程序中，我们对窗口框所做的仅仅是设置了一个标签（即 `Canvas`）。

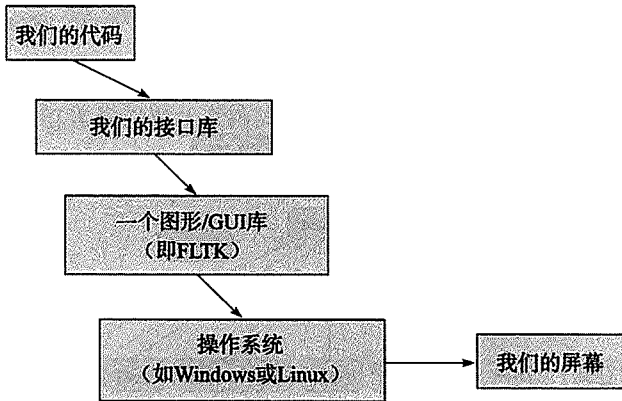
## 17.4 使用 GUI 库

✘ 在本书中，我们不直接采用操作系统的图形和 GUI（图形用户界面）工具，否则会将程序限制在一种特定的操作系统上，而且需要处理很多复杂的细节问题。与处理文本 I/O 一样，我们将使用一个函数库来消除操作系统间的差异、I/O 设备的变化等问题，并简化程序代码。不幸的是，C++ 并没有提供一个像标准流 I/O 库一样的标准 GUI 库，于是我们从很多可用的 C++ GUI 库选择了一个。为了不局限于这种 GUI 库，并且避免一开始就接触其复杂功能，我们只使用一组在任何 GUI 库中都只需几百行程序就能实现的接口类。

我们使用的（目前还只是间接使用）GUI 工具包名为 `FLTK`（Fast Light Tool Kit，读作“full tick”），该工具包源自 [www.fltk.org](http://www.fltk.org)。我们的代码可以移植到任何使用 `FLTK` 的平台（Windows、Unix、Mac、Linux 等）。我们的接口类也可以使用其他的图形工具包重新实现，因此基于它的代码的移植性实际上还要更好一些。

接口类实现的编程模型比通常的工具包提供的更简单。例如，我们整个图形和 GUI 接口库的 C++ 代码大约为 600 行，而最简单的 FLTK 文档也达 370 页。你可以从 [www.fltk.org](http://www.fltk.org) 下载，但我们并不推荐你阅读，目前还不需要那些细节。第 17 ~ 21 章给出的概念可用于任何一个流行的 GUI 工具包，当然我们也会解释接口类是如何映射到 FLTK 的，以便在必要的时候能够直接使用其他的工具包。

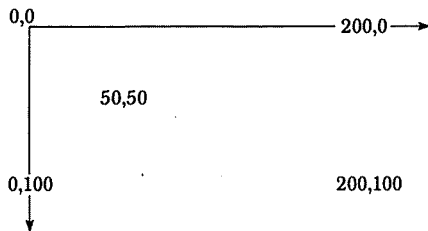
我们实现的“图形世界”的部分结构如下：



接口类为二维形状提供了简单、用户可扩展的基本框架，并支持简单的颜色。为了实现这些功能，我们给出了基于“回调函数”的 GUI 概念，这些函数由屏幕上的用户自定义按钮等组件触发（参见第 21 章）。

## 17.5 坐标系

计算机屏幕是一个像素组成的矩形区域，像素是一个可以设置为某种颜色的点。在程序中，最常见的方式就是将屏幕建模为像素组成的矩形区域，每个像素由  $x$ （水平）坐标和  $y$ （垂直）坐标确定。最左端的像素的  $x$  坐标为 0，向右逐步递增，直到最右端的像素为止；最顶端的像素的  $y$  坐标为 0，向下逐步递增，直到最底端的像素为止。



注意， $y$  坐标是“向下增长”的。这可能有点奇怪，特别是对数学家而言。但是，屏幕（窗口）大小各异，左上角可能是不同屏幕的唯一共同之处了，因此将其设定为原点。

不同屏幕的像素数可能各不相同，常见的尺寸有：1024 × 768、1280 × 1024、1400 × 1050 和 1600 × 1200。

在使用屏幕与计算机进行交互时，通常从屏幕上划分出特定用途的、由程序控制的矩形区域——窗口。对窗口的操作与屏幕完全一致。基本上，我们将窗口看作一个小屏幕。

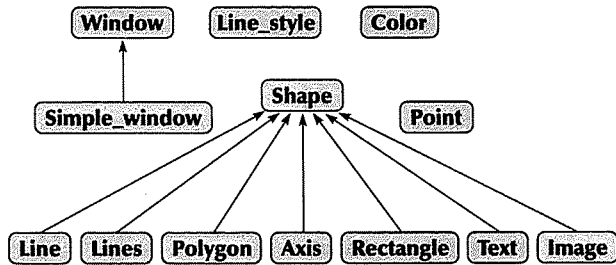
例如：

```
Simple_window win {tl,600,400,"Canvas"};
```

该语句定义了宽度为 600 像素、高度为 400 像素的矩形区域， $x$  坐标从左到右为 0 ~ 599， $y$  坐标从上到下为 0 ~ 399。能够进行绘制的窗口区域通常被称为画布 (canvas)。我们指定的  $600 \times 400$  像素指的就是“内部大小”，即位于系统提供的窗口框内部的大小，不包括标题栏、退出按钮等占用的空间。

## 17.6 Shape

我们提供的基本绘图工具包由 12 个类构成：



箭头表示：当需要箭头头部的类时，可以使用尾部的类。例如：当需要一个 Shape 时，我们可以提供一个 Polygon，也就是说，Polygon 是一种 Shape。

我们将从以下类开始进行介绍：

- Simple\_window、Window。
- Shape、Text、Polygon、Line、Lines、Rectangle、Function 等。
- Color、Line\_style、Point。
- Axis。

第 21 章将引入 GUI (用户交互) 类：

- Button、In\_box、Menu 等。

我们可以很容易地添加更多的类 (当然取决于你对“容易”的定义)，例如：

- Spline、Grid、Block\_chart、Pie\_chart 等。

不过，定义或描述一个完整的 GUI 框架及其所有功能已经超出了本书的范围。

## 17.7 使用 Shape 类

本节介绍图形库的一些基本工具：Simple\_window、Window、Shape、Text、Polygon、Line、Lines、Rectangle、Function、Color、Line\_style、Point、Axis。目的是让你知道这些工具能够实现什么功能，而并非详细理解某个类。下一章将会介绍每个类的设计与实现。

下面来学习一个简单的程序，我们将逐行解释代码，并给出每一行代码在屏幕上的显示效果。在程序运行时，你会看到当我们向窗口添加形状以及改变已有形状时，屏幕上图像的变化情况。大体上，我们是通过分析程序的执行情况来“动画演示”代码的流程。

### 17.7.1 图形头文件和主函数

首先，我们包含定义了图形和 GUI 工具接口的头文件：

```
#include "Window.h"           // 一个普通窗口
#include "Graph.h"
```

或者

```
#include "Simple_window.h"    // 如果我们希望有“Next”按钮
#include "Graph.h"
```

你可能已经猜到, `Window.h` 包含与窗口有关的工具, `Graph.h` 包含在窗口上绘制形状(包括文本)的有关工具, 这些工具都定义在 `Graph_lib` 名字空间中。为简化起见, 我们使用名字空间指令, 使得 `Graph_lib` 中的名字可以直接在程序中使用。

```
using namespace Graph_lib;
```

照例, `main()` 函数包含我们要(直接或间接)执行的代码及例外处理:

```
int main ()
try
{
    // 这里是我们的代码
}
catch(exception& e) {
    // 报告一些错误
    return 1;
}
catch(...) {
    // 更多的错误报告
    return 2;
}
```

`main()` 函数进行编译时, 必须已定义了 `exception`。如果我们照例包含了 `std_lib_facilities.h`, 就会得到 `exception`, 否则我们会从标准头文件处开始直接处理, 此时需要包含 `<stdexcept>`。

### 17.7.2 一个几乎空白的窗口

在这里, 我们不讨论错误处理(参见第 5 章, 特别是 5.6.3 节), 直接进入 `main()` 函数中的图形代码:

```
Point tl {100,100};           // 我们窗口的左上角

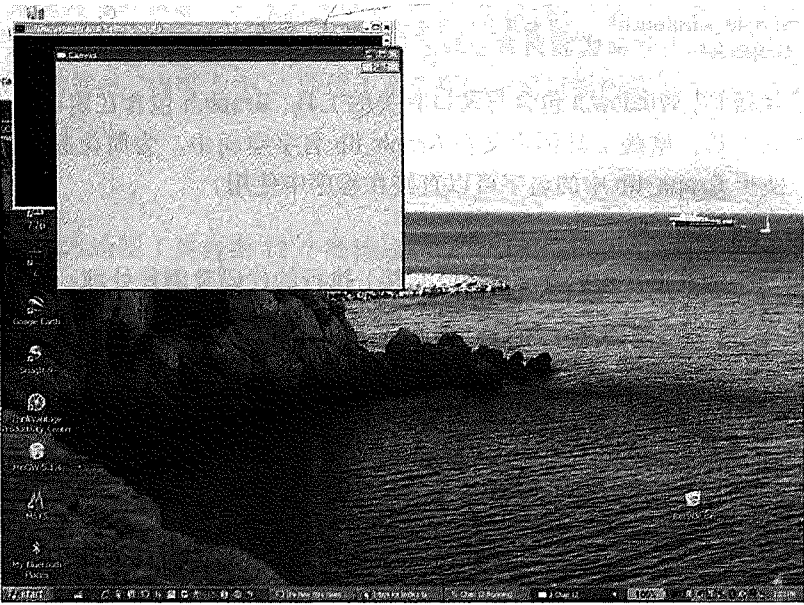
Simple_window win {tl,600,400,"Canvas"};
    // 屏幕坐标点 tl 对应左上角
    // 窗口大小 (600*400)
    // 标题: Canvas
win.wait_for_button();       // 显示!
```

这段代码首先创建一个 `Simple_window`, 即一个有“Next”按钮的窗口, 并将它显示在屏幕上。显然, 为了得到 `Simple_window` 对象, 我们应该包含头文件 `Simple_window.h` 而不是 `Window.h`。在这里, 我们明确给出了窗口在屏幕上的显示位置: 其左上角位于 `Point{100, 100}`。这个位置很接近屏幕的左上角, 但没有过于靠近。很显然, `Point` 是一个类, 其构造函数有两个整型参数, 表示点在屏幕上的  $(x, y)$  坐标。我们可以将代码写为:

```
Simple_window win {Point{100,100},600,400,"Canvas"};
```

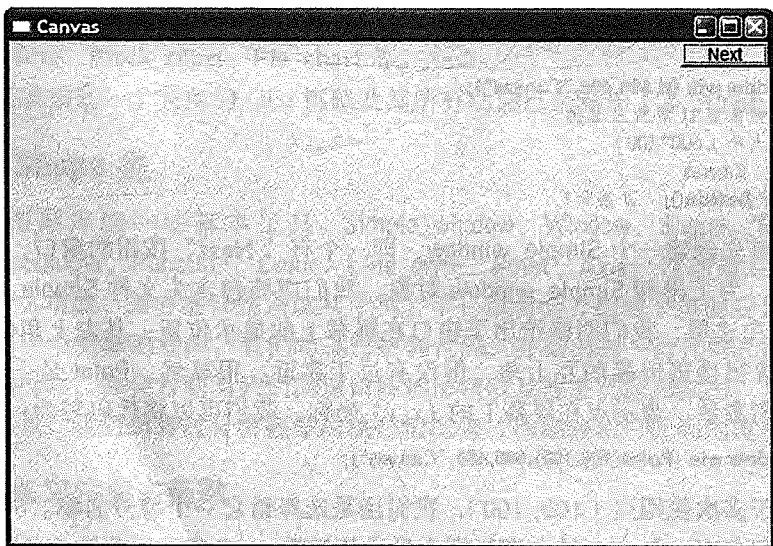
然而, 为了便于多次使用点  $(100, 100)$ , 我们还是选择给它一个符号名称。600 和 400 分别是窗口的宽度和高度, `Canvas` 是在窗口框上显示的标签。

为了真正将窗口绘制在屏幕上，我们必须将控制权交给 GUI 系统。我们通过调用 `win.wait_for_button()` 来达到这一目的，结果如下：



在窗口的背景中，我们看到了一个笔记本电脑的桌面（已经临时清理过了）。如果你对桌面背景这种不相关的事情感到好奇，我可以告诉你，我拍摄照片时正站在安提布的毕加索资料馆附近俯瞰尼斯湾。隐藏在程序窗口之后的黑色控制台窗口是用来运行我们的程序的。控制台窗口不太美观，而且也不是必需的，但当一个尚未调试通过的程序陷入无限循环或无法继续执行时，我们可以通过它来终止程序。如果你仔细观察，会发现我们使用的是微软 C++ 编译器，当然你可以使用其他的编译器（如 Borland 或者 GNU）。

在之后的介绍中，我们将去掉程序窗口周围分散注意力的内容，仅仅给出窗口本身：



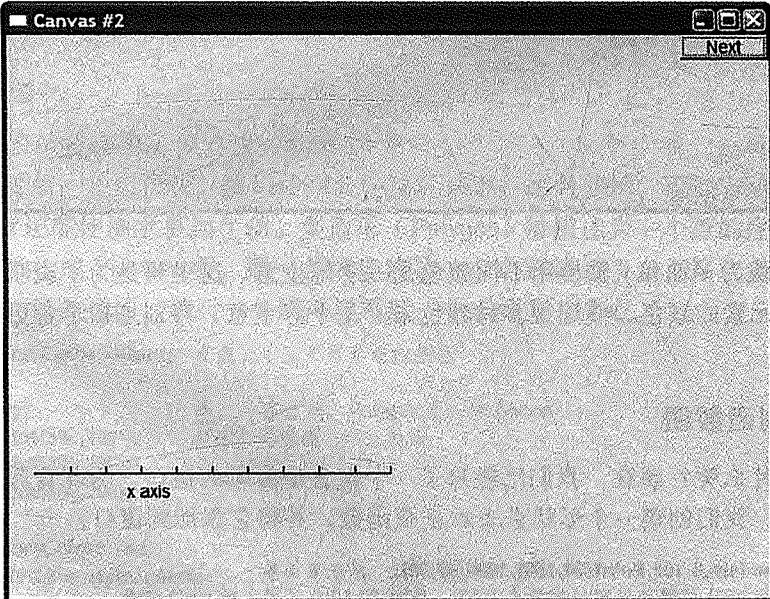
窗口的实际尺寸（以像素计算）依赖于屏幕的分辨率。某些屏幕的像素要比其他屏幕更大。

### 17.7.3 坐标轴

一个几乎空白的窗口没有什么意思，最好给它添加一些信息。希望添加些什么内容呢？注意：并不是所有的图形都是有趣的或者是关于游戏的，我们将从坐标轴——一种严肃的、有点复杂的图形开始。一个没有坐标轴的图形通常是很难看的。没有坐标轴的帮助，我们通常难以弄清数据的含义。或许你可以借助伴随的文字来解释，但使用坐标轴要保险得多；人们通常不会阅读文字描述，而且好的图形表示通常与其语境是分离的。因此，图形需要坐标轴：

```
Axis xa {Axis::x, Point(20,300), 280, 10, "x axis"}; // 生成一个坐标轴
    // Axis 是一种 Shape
    // Axis::x 表示是水平的
    // 从点 (20,300) 处开始
    // 280 个像素长
    // 10 个“刻度”
    // 坐标轴标签为“x axis”
win.attach(xa); // 将 xa 添加到窗口 win
win.set_label("Canvas #2"); // 重新给窗口添加标签
win.wait_for_button(); // 显示！
```

操作步骤为：创建坐标轴对象，将其添加到窗口，最后进行显示：



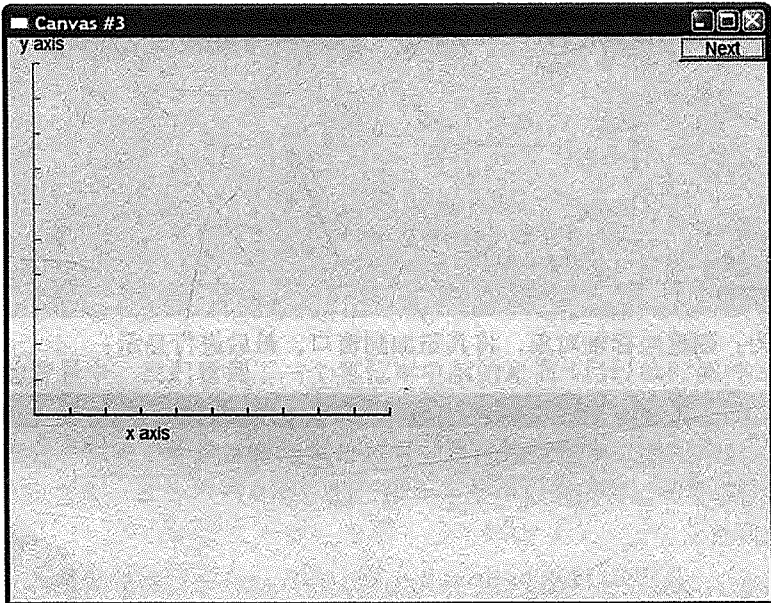
可以看到，`Axis::x` 是一条水平线，其上有指定数量的“刻度”（10 个）和一个标签“`x axis`”。通常，标签用于解释坐标轴和刻度的含义。我们通常把 `x` 轴放在窗口底端附近。在实际应用中，我们更喜欢用符号常量来表示高度和宽度，这样“在底端上方附近”就可以用 `y_max-bottom_margin` 这样的符号表示，而不是用 `300` 这样的“魔数”（参见 4.3.1 节和 20.6.2 节）。

为了帮助识别程序的输出，我们用 Window 的成员函数 `set_label()` 将该窗口的标签重新设置为“Canvas #2”。

现在，添加一个  $y$  坐标轴：

```
Axis ya (Axis::y, Point{20,300}, 280, 10, "y axis");
ya.set_color(Color::cyan);           // 选择一种颜色
ya.label.set_color(Color::dark_red); // 选择一种文本的颜色
win.attach(ya);
win.set_label("Canvas #3");
win.wait_for_button();               // 显示！
```

我们将  $y$  轴和标签的颜色分别设置为 `cyan` 和 `dark_red`（只是为了展示一些工具的使用）。



我们并不认为  $x$  轴和  $y$  轴使用不同颜色是一个好主意。这里只是为了说明如何设置形状或者其中某个元素的颜色。使用很多种颜色未必是个好主意，特别是初学者更容易热衷使用很多颜色。

#### 17.7.4 绘制函数图

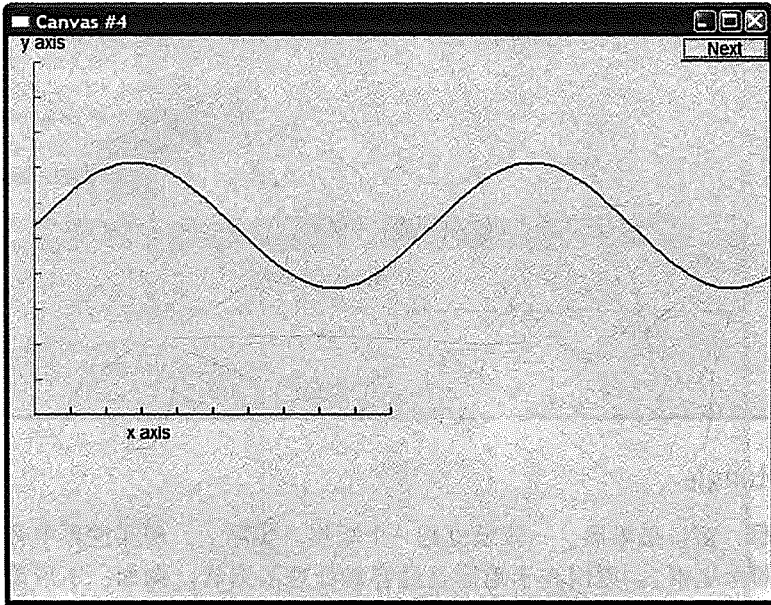
接下来做什么呢？现在，我们已经有了一个包含坐标轴的窗口，因此看起来画出一个函数是个好主意。我们创建一个形状来表示正弦函数，并将它添加到窗口：

```
Function sine {sin,0,100,Point{20,150},1000,50,50}; // 正弦曲线
// 在 (20,150) 位置处设置坐标原点 (0,0)，绘制 [0,100] 区间上的 sin() 函数
// 使用 1000 个点，x 值的比例乘 50，y 值的比例乘 50

win.attach(sine);
win.set_label("Canvas #4");
win.wait_for_button();
```

在这段代码中，名为 `sine` 的 `Function` 对象使用标准库函数 `sin()` 产生的值绘制一条正弦曲线。我们将在 20.3 节详细讨论如何绘制函数图。现在，你只需知道，绘制函数图时必须给出起

始点的位置和输入值集合（值域），并且还需给出一些信息来说明如何将这些内容塞入窗口（缩放）。



请注意在到达窗口右边界时曲线是如何停止的。当我们绘制的点超出窗口矩形区域时，将被 GUI 系统简单忽略掉，永远不会真正显示出来。

### 17.7.5 Polygon

函数图是表示数据的一种方法，在第 20 章将会看到更多实例。我们还可以在窗口中绘制不同类型的对象：几何形状。我们使用几何形状来进行图形演示，可以表示用户交互组件（如按钮），通常还能使演示更加生动。多边形（Polygon）被描述为一个点的序列，这些点通过线连接起来就构成 Polygon 类。第一条线连接第一个点到第二个点，第二条线连接第二个点到第三个点，以此类推，最后一条线连接最后一个点到第一个点。

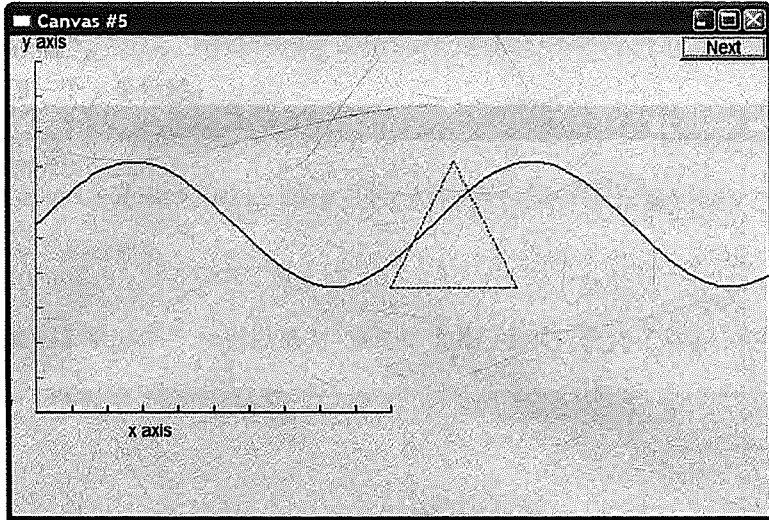
```
sine.set_color(Color::blue); // 我们改变正弦曲线的颜色

Polygon poly;                // 一个多边形, Polygon 是一种 Shape
poly.add(Point{300,200});    // 3 个点构成一个三角形
poly.add(Point{350,100});
poly.add(Point{400,200});

poly.set_color(Color::red);
poly.set_style(Line_style::dash);
win.attach(poly);
win.set_label("Canvas #5");
win.wait_for_button();
```

这段代码首先展示了如何改变正弦曲线的颜色。然后，与 17.3 节的例子一样，我们添加了一个三角形，作为一个多边形的例子。然后我们再次设置了颜色，最后设置了线型。Polygon 的线都有“线型”，默认线型为实线，但我们也可根据需要改为虚线、点状线等（参见 18.5 节）。这段程序显示如下图形：





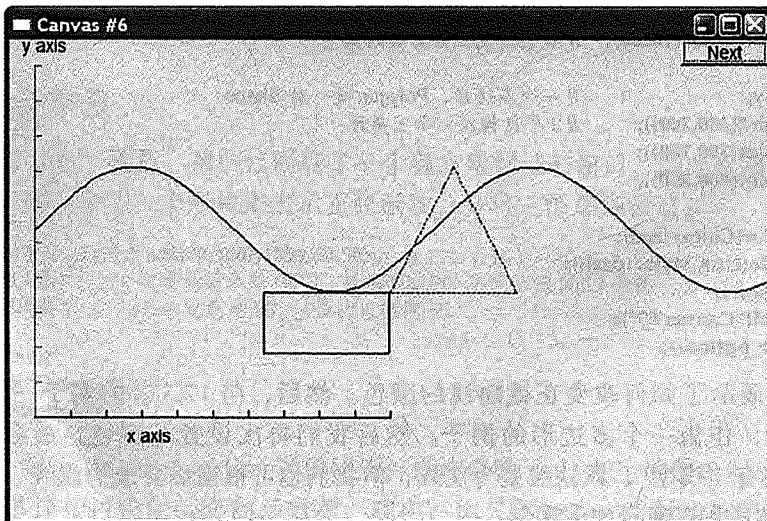
### 17.7.6 Rectangle

✂ 屏幕是矩形，窗口是矩形，一张纸也是一个矩形。实际上，现实世界中有很多形状都是矩形（至少是圆角矩形）。原因在于矩形是最容易处理的形状。例如：矩形易于描述（左上角和宽度、高度，或者左上角和右下角，诸如此类），易于判断一个点在矩形之内还是之外，易于用硬件快速绘制像素构成的矩形。

与其他封闭的形状相比，大多数高级图形库能够更好地处理矩形。因此，我们将矩形类 `Rectangle` 从多边形类 `Polygon` 中独立出来。一个 `Rectangle` 可以用左上角坐标、宽度和高度来描述：

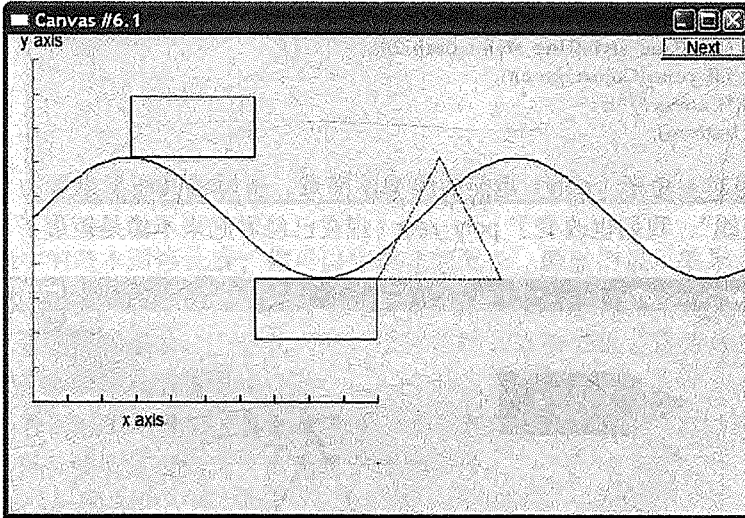
```
Rectangle r(Point(200,200), 100, 50); // 左上角，宽度，高度
win.attach(r);
win.set_label("Canvas #6");
win.wait_for_button();
```

由此可得：



请注意，将位置正确的四个点连接起来并不一定得到一个 **Rectangle**。当然，在屏幕上创建一个看起来像 **Rectangle** 的 **Closed\_polyline** 是很简单的（你甚至可以创建一个看起来像是 **Rectangle** 的 **Open\_polyline**），例如：

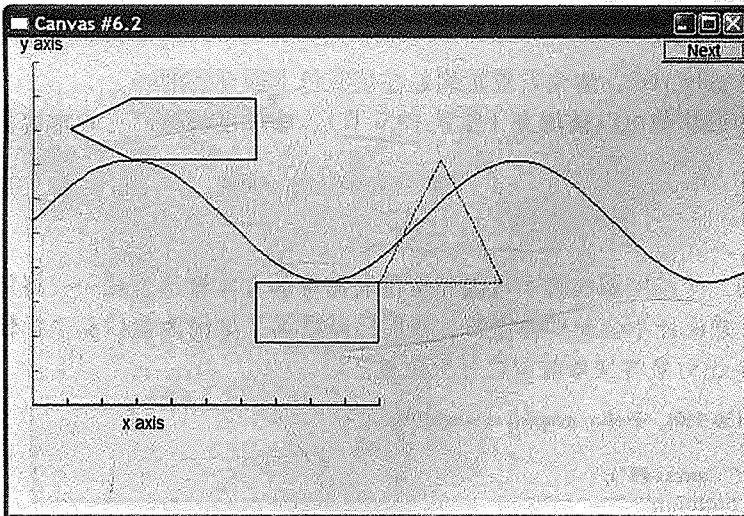
```
Closed_polyline poly_rect;  
poly_rect.add(Point{100,50});  
poly_rect.add(Point{200,50});  
poly_rect.add(Point{200,100});  
poly_rect.add(Point{100,100});  
win.attach(poly_rect);
```



实际上，**poly\_rect** 对应的屏幕图像（**image**）是一个矩形。但内存中的 **poly\_rect** 对象并不是一个 **Rectangle** 对象，而且它也不“知道”有关矩形的任何内容。验证这一点的最简单方法是再添加一个点：

```
poly_rect.add(Point{50,75});
```

矩形是不会有 5 个点的：



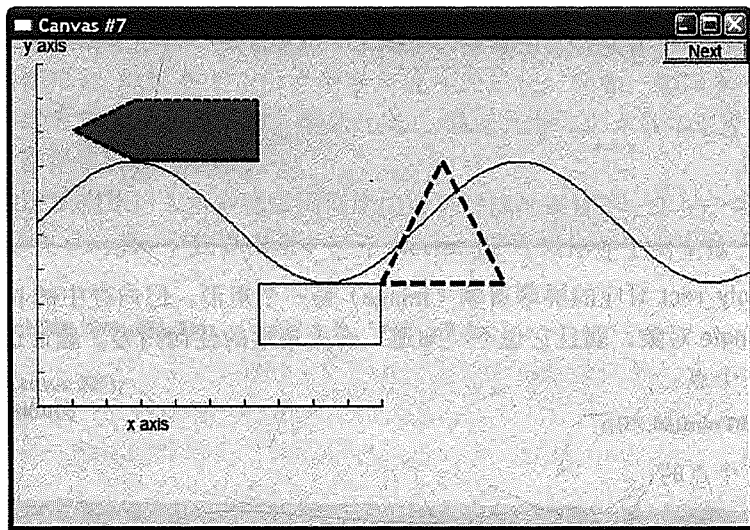
✂ 对于我们分析程序非常重要的一点是：**Rectangle** 不仅仅是在屏幕上看起来像是一个矩形而已，它还应该从根本上保证此形状（几何意义上）始终是一个矩形。这样，我们编写代码时就可以信赖 **Rectangle**——它确实表示屏幕上的一个矩形，而且保证不会改变为其他形状。

### 17.7.7 填充

前面绘制形状都是绘制轮廓，我们也可以使用某种颜色“填充”一个矩形：

```
r.set_fill_color(Color::yellow); // 为矩形内部填充颜色
poly.set_style(Line_style(Line_style::dash,4));
poly_rect.set_style(Line_style(Line_style::dash,2));
poly_rect.set_fill_color(Color::green);
win.set_label("Canvas #7");
win.wait_for_button();
```

我们还觉得对三角形（**poly**）当前的线型不满意，所以将其线型设置为“粗（正常线型的 4 倍粗细）虚线”，我们也改变了 **poly\_rect**（现在已经看起来不像是矩形了）的线型。



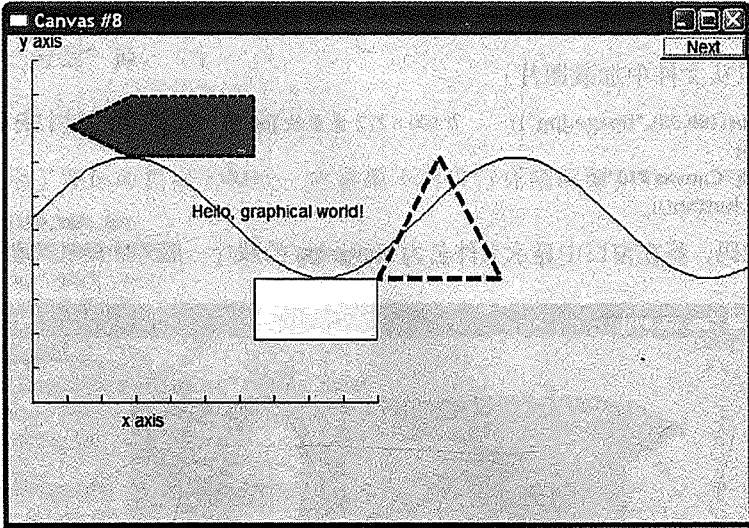
如果你仔细观察 **poly\_rect**，你会发现轮廓是在填充色上层显示的。

任何封闭的形状都可以被填充（参见 18.9 节）。矩形很特殊，它非常容易填充（填充速度也非常快）。

### 17.7.8 Text

✂ 最后，任何一个绘图系统都不可能完全没有简单的文本输出方式——将每个字符看作线的集合来绘制，并保证不会剪切掉字符。我们已经展示了如何为窗口和坐标轴设置标签，但我们也能使用 **Text** 对象将文本放置在任何位置。

```
Text t {Point(150,150), "Hello, graphical world!";}
win.attach(t);
win.set_label("Canvas #8");
win.wait_for_button();
```

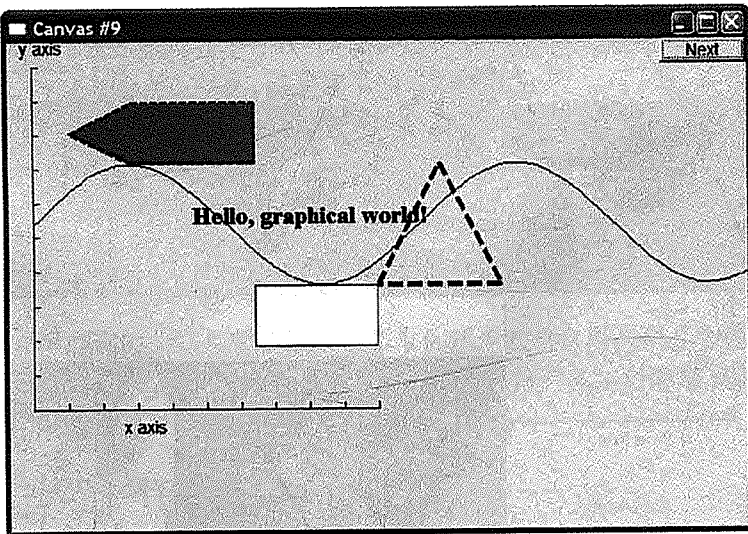


利用此例中的基本图形元素，你可以生成任何复杂、微妙的显示效果。请注意本章所有代码的一个共同特点：没有循环和选择语句，而且所有数据都是“硬编码的”。输出内容只是基本图形元素的简单组合。一旦我们开始使用数据和算法来组合这些基本图形，就可以得到更复杂、更有趣的输出效果了。

我们已经看到过如何改变文本的颜色了：坐标轴的标签（参见 17.7.3 节）本身就是一个 Text 对象。此外，我们还可以为文本选择字体和字号：

```
t.set_font(Font::times_bold);  
t.set_font_size(20);  
win.set_label("Canvas #9");  
win.wait_for_button();
```

这段代码将 Text 的字符串“Hello, graphical world!”中的字符放大到 20 号字，字体设置为粗体 Times。

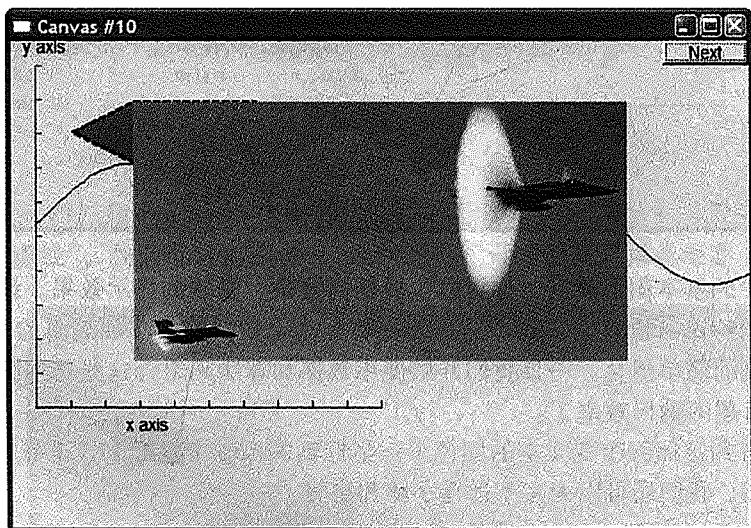


### 17.7.9 Image

我们还可以从文件中加载图片：

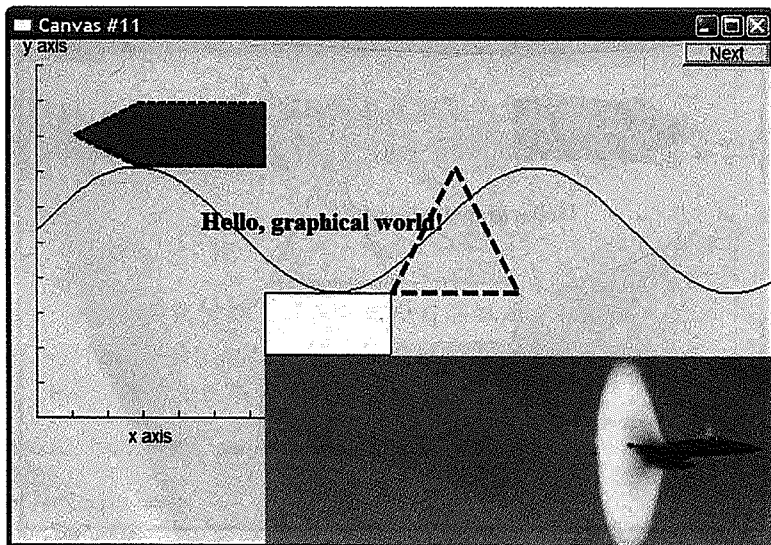
```
Image ii {Point{100,50},"image.jpg"}; // 400×212 像素的 jpg 图
win.attach(ii);
win.set_label("Canvas #10");
win.wait_for_button();
```

执行上述代码，将在窗口中显示文件名为 image.jpg 的照片，照片中两架飞机正在突破音障：



这幅照片比较大，我们刚好把它放在了文本和图形上层。因此，为了清理窗口，我们将它稍微移开一点：

```
ii.move(100,200);
win.set_label("Canvas #11");
win.wait_for_button();
```



请注意不在窗口区域之内的部分图片是如何被简单忽略掉的。超出窗口区域的内容都会这样被图形系统“剪裁”掉。

### 17.7.10 更多未讨论的内容

下面代码展示了图形库更多的特性，在这里不再进行详细解释：

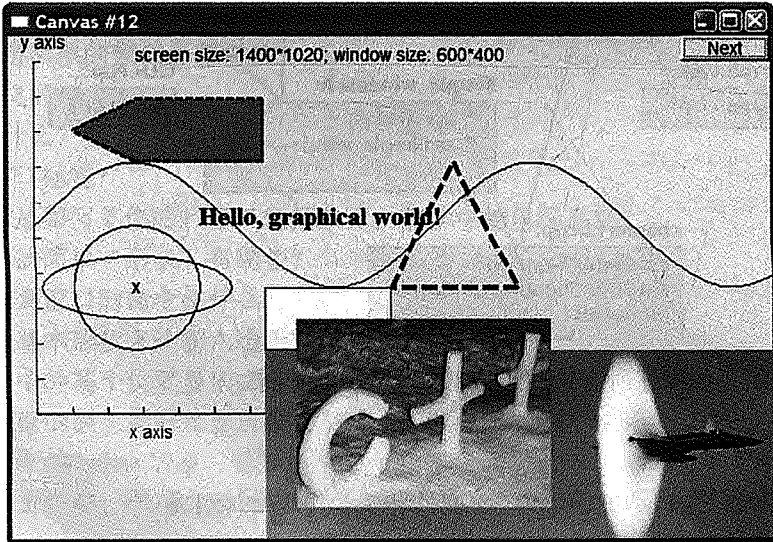
```
Circle c {Point{100,200},50};
Ellipse e {Point{100,200}, 75,25};
e.set_color(Color::dark_red);
Mark m {Point{100,200},'x'};

ostringstream oss;
oss << "screen size: " << x_max() << "*" << y_max()
    << "; window size: " << win.x_max() << "*" << win.y_max();
Text sizes {Point{100,20},oss.str()};

Image cal {Point{225,225},"snow_cpp.gif"}; // 320×240 像素的 gif 图
cal.set_mask(Point{40,40},200,150); // 显示图片的中间部分
win.attach(c);
win.attach(m);
win.attach(e);

win.attach(sizes);
win.attach(cal);
win.set_label("Canvas #12");
win.wait_for_button();
```

你能猜出这段代码显示什么内容吗？是不是很容易猜？



代码与屏幕显示内容的关联是很直接的。如果你还未看出这段代码是如何产生这样的输出的，请继续学习后续章节，很快就会搞清楚的。请注意我们是如何使用 `ostringstream`（参见 11.4 节）来格式化输出尺寸的文本对象的。

## 17.8 让图形程序运行起来

我们已经看到了如何创建窗口以及如何窗口中绘制各种各样的形状。在后续章节中，



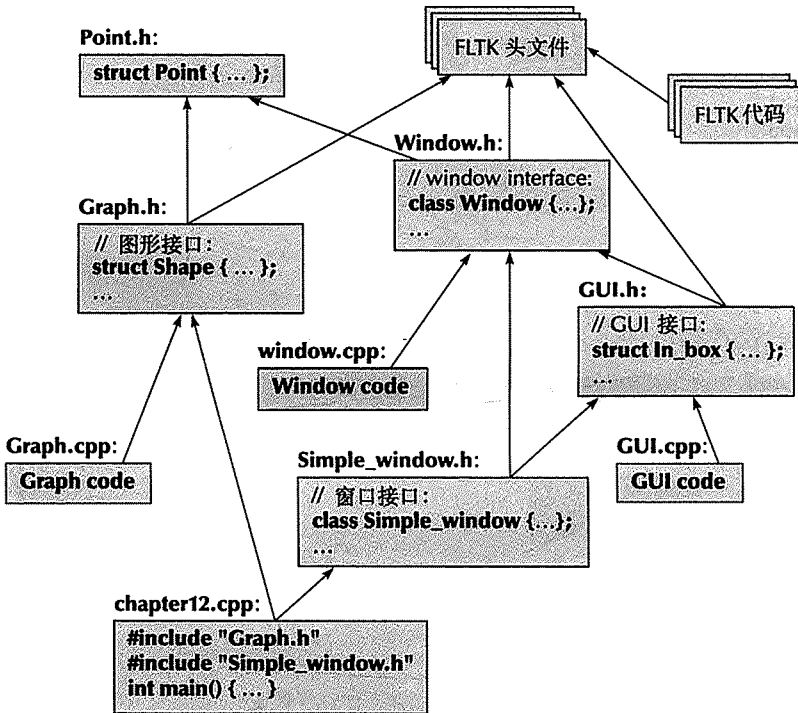
我们将会看到这些 Shape 类是如何定义的，以及它们更多的使用方法。

为了使这个图形程序运行起来，还需要其他程序的帮助。除了主函数中已有的代码外，我们还需要编译接口库代码，安装 FLTK 库（或者所使用的任何 GUI 系统），并将它与我们的代码正确地链接在一起，才能让这个图形程序运行起来。

我们可以将这个程序看作由 4 个不同的部分构成：

- 我们编写的代码（main() 函数等）；
- 接口库（Window、Shape、Polygon 等）；
- FLTK 库；
- C++ 标准库。

另外，程序还间接用到了操作系统。忽略了操作系统和标准库之后，我们的图形代码的组织结构可描述如下：



附录 D 说明了如何让所有这些组成部分一起运转起来。

### 17.8.1 源文件

我们的图形和 GUI 接口库由 5 个头文件和 3 个代码文件组成：

- 头文件：
  - Point.h;
  - Window.h;
  - Simple\_window.h;
  - Graph.h;
  - GUI.h.
- 代码文件：

- Window.cpp;
- Graph.cpp;
- GUI.cpp。

在学习第 21 章之前，你可以忽略 GUI 文件。

## 简单练习

本练习是与“Hello, World!”程序有相同功能的一个图形程序，目的是让你熟悉最简单的图形工具。

1. 创建一个空窗口 `Simple_window`，尺寸为  $600 \times 400$ ，标签为 `My window`，编译这个程序，然后链接并运行。注意：必须以附录 D 描述的方法链接 FLTK 库，在代码中包含头文件 `Graph.h` 和 `Simple_window.h`，并将 `Graph.cpp` 和 `Window.cpp` 加入你的项目中。
2. 逐个添加 17.7 节中的例子，每添加一个就测试一下。
3. 检查并简单修改每个例子（例如颜色、位置、点的数量等）。

## 思考题

1. 我们为什么要使用图形？
2. 什么时候我们尽量不使用图形？
3. 为什么图形对程序员来说是有趣的？
4. 什么是窗口？
5. 我们的图形接口类（图形库）在哪个名字空间中？
6. 使用我们的图形库实现基本的图形功能，需要哪些头文件？
7. 我们使用的最简单的窗口是怎样的？
8. 什么是最小化的窗口？
9. 窗口标签是什么？
10. 如何设置窗口标签？
11. 屏幕坐标系是如何工作的？窗口坐标系呢？数学中的坐标系呢？
12. 我们能显示的简单“形状”有哪些？
13. 将形状添加到窗口的命令是什么？
14. 你使用哪种基本形状来绘制六边形？
15. 如何在窗口中的某个位置显示文本？
16. 如何将你最好的朋友的照片显示在窗口中（使用你自己编写的程序）？
17. 你创建了一个 `Window` 对象，但屏幕上没有显示任何内容，可能的原因有哪些？
18. 你创建了一个形状，但窗口中没有显示任何内容，可能的原因有哪些？

## 术语

color (颜色)	graphics (图形)	JPEG
coordinates (坐标)	GUI	line style (线型)
display (显示)	GUI library (GUI 库)	software layer (软件层)
fill color (填充颜色)	HTML	window (窗口)
FLTK	image (图像)	XML



## 习题

我们建议使用 Simple\_window 完成下面的练习。

1. 分别用 Rectangle 和 Polygon 绘制矩形，并将 Polygon 的边设置为红色，Rectangle 的边设置为蓝色。
2. 绘制一个  $100 \times 30$  的 Rectangle，并在其内显示文本“*Howdy!*”。
3. 绘制你名字的首字母，高度为 150 个像素，使用粗线，每个字母使用不同的颜色。
4. 绘制一个  $3 \times 3$  的红白交替的井字棋棋盘。
5. 在矩形周围绘制一个 1/4 英寸宽的红色框，矩形的高度为屏幕高度的 3/4，宽度为屏幕宽度的 2/3。
6. 当绘制的 Shape 不能完全放在窗口内时会发生什么现象？当绘制的窗口不能完全置于屏幕内时又会发生什么现象？编写两个程序说明这两种现象。
7. 绘制一个二维的房屋正视图，包括一个门、两个窗户、带烟囱的屋顶，可以随意添加其他的细节，比如从烟囱“冒烟”等。
8. 绘制奥林匹克五环旗。如果你记不得颜色了，请查找相关资料。
9. 在屏幕上显示一个图像，例如一个朋友的照片，并使用窗口标题和窗口内的描述文字进行说明。
10. 绘制 17.8 节中的文件结构图。
11. 由外向内绘制一系列正多边形，最里面的的是一个等边三角形，外边是一个正方形，再外边是一个正五边形，依此类推。数学专业的人士请注意：让  $N$ - 边形的所有顶点都落在  $(N+1)$ - 边形的边上。提示：三角函数被包含在 `<cmath>` 中（参见 24.8 节和附录 C9.2）。
12. 超椭圆是一个由下面的方程定义的二维形状：

$$\left| \frac{x}{a} \right|^m + \left| \frac{y}{b} \right|^n = 1; \quad m, n > 0$$

请在互联网上查找超椭圆，以便对这种形状有更感性的认识。编写程序，通过连接超椭圆上的点构成“星状”模式。程序接受参数  $a$ 、 $b$ 、 $m$ 、 $n$  和  $N$ ，在由  $a$ 、 $b$ 、 $m$  和  $n$  定义的超椭圆上选择  $N$  个等间隔（按某种“等间隔”的定义）的点，然后将每个点连接到其他一个或多个点（可以用一个参数  $k$  之处连接的点数，或者直接使用  $N-1$ ，即连接其他所有点）。

13. 设计一种为上一题中的超椭圆形状添加颜色的方法。可以为某些线指定一种颜色，其他线使用另一种或另几种颜色。

## 附言

理想的程序设计是将我们的概念直接表示为程序中的实体。因此，我们通常使用类表示思想，使用类对象表示现实世界的实体，使用函数表示行为和计算。这种思路显然可以用于图形领域。当我们有了概念，例如圆和多边形，就可以在程序中用 Circle 类和 Polygon 类来表示。图形应用的不同寻常之处在于，当我们编写图形程序时，还有机会在屏幕上看到那些类对象。也就是说，程序状态可以直接呈现出来，供我们观察，而在大多数应用中我们是没那么幸运的。思想、代码和输出的直接对应是图形程序设计如此有吸引力的重要原因。不过，请记住，图形只是体现了在代码中使用类直接表达概念的思想而已。这种思想才是最重要的，它非常有效、通用：我们想到的任何东西都可以在代码中用类、类对象或者一组类来描述。

## 图 形 类

不能改变你的思考方式的语言是不值得学习的。

——习语

第 17 章介绍了使用一组简单的接口类可以做哪些图形应用，以及如何做。本章将介绍更多的类。本章的重点是 `Point`、`Color`、`Polygon`、`Open_polyline` 等单个接口类及其实例的设计、使用和实现。后续章节将介绍设计一组相关的类的方法及其更多的实现技术。

## 18.1 图形类概览

图形和 GUI 库提供了大量的特性。“大量”的意思是数百个类，每个类一般都有数十个处理函数。阅读它的说明书、手册或者文档有点像阅读一本老式的植物学教材，其中列出了数千种植物的详细信息，而这些植物的分类模糊不清。真是令人沮丧！但也有令人兴奋的一面，浏览最新的图形 /GUI 库特性会使你感觉像是一个孩子进入了糖果店，但搞清楚应该从哪里开始，哪些是真正对你有用的，仍旧十分困难。

我们设计这个接口库的目标之一就是减小成熟的图形 /GUI 库的复杂性带来的学习难度。我们只提出了 24 个几乎没有任何操作的类，但它们仍能够产生有用的图形输出。另一个紧密相关的目标是通过这些类引入重要的图形和 GUI 概念。现在，你已经能编写程序，将结果显示为简单的图形了。经过本章的学习，你的图形程序设计能力将会超出大多数人最初的期待。经过第 19 章的学习，你将会理解大多数设计技术及其思想，从而能够加深理解，并能够根据需要扩展图形表达能力。为了实现扩展，你既可以向我们的图形 /GUI 库中添加新的特性，也可以采用其他 C++ 图形 /GUI 库。

重要的接口类如下表所示：

图形接口类	
<code>Color</code>	用于设置线、文本及形状填充的颜色
<code>Line_style</code>	用于设置线形
<code>Point</code>	表示屏幕上和 Window 内的位置
<code>Line</code>	对应屏幕上的一个线段，用两个 <code>Point</code> 对象（端点）来描述
<code>Open_polyline</code>	相连的线段序列，用一个 <code>Point</code> 对象序列来描述
<code>Closed_polyline</code>	与 <code>Open_polyline</code> 类似，唯一差别是必须有一个线段连接最后一个 <code>Point</code> 和第一个 <code>Point</code>
<code>Polygon</code>	所有线段均不相交的 <code>Closed_polyline</code>
<code>Text</code>	字符串文本
<code>Lines</code>	线段集合，用 <code>Point</code> 对的集合来描述
<code>Rectangle</code>	矩形，经过优化，显示快速、便捷

(续)

图形接口类	
Circle	圆, 用圆心和半径定义
Ellipse	椭圆, 用圆心和两个轴来描述
Function	一个函数, 画出其一段值域内的图形
Aixs	带标签的坐标轴
Mark	用一个字符 (如 x 或 o) 标记的点
Marks	带标记 (如 x 和 o) 的点的序列
Marked_polyline	点被标记的 Open_polyline
Image	图像文件的内容

第 20 章将介绍 **Function** 和 **Axis**, 第 21 章介绍主要的 GUI 接口类:

GUI 接口类	
Window	屏幕的一个区域, 我们用来显示图形对象
Simple_window	带有 “Next” 按钮的窗口
Button	窗口中的一个矩形, 通常带有标签, 我们可以通过点按它来执行对应函数
In_box	窗口中的一个框, 通常带标签, 用户可在其中输入文本字符串
Out_box	窗口中的一个框, 通常带标签, 用户程序可在其中输出字符串
Menu	Button 向量

源文件组织如下:

图形接口源文件	
Point.h	Point
Graph.h	所有其他接口类
Window.h	Window
Simple_window.h	Simple_window
GUI.h	Button 和其他 GUI 类
Graph.cpp	Graph.h 中函数的定义
Window.cpp	Window.h 中函数的定义
GUI.cpp	GUI.h 中函数的定义

除图形类以外, 我们还设计了一个用于保存 **Shape** 或者 **Widget** 的容器类:

Shape 或 Widget 的容器	
Vector_ref	向量, 其接口便于保存未命名的元素

当你阅读下面几节时，请不要太快。虽然并没有什么困难的内容，但本章的目的不是向你展示一些漂亮的图片——你每天都会在自己的计算机屏幕上或者电视上看到更漂亮的图片，本章的重点在于：

- 展示代码和它生成的图片之间的关系。
- 让你习惯阅读代码，思考代码是如何工作的。
- 让你考虑代码的设计，特别是如何在代码中用类来表示各种概念。为什么这样设计那些类？还能怎样设计？在设计中我们做出了很多决定，其中大多数都可以给出同样合理的但不同的决定，在某些情况下甚至可以彻底不同。

因此，请不要着急，否则你将会漏掉一些重要的内容，因而可能觉得习题很困难。

## 18.2 Point 和 Line

在任何图形系统中，点（point）都是最基本的组成部分。定义点就是定义我们如何来组织几何空间。在这里，我们使用人们习惯的面向计算机的二维布局的点的定义：整数坐标  $(x, y)$ 。如 17.5 节的描述， $x$  坐标从 0（屏幕的左边界）到  $\text{max}_x()$ （屏幕的右边界）， $y$  坐标从 0（屏幕的上边界）到  $\text{max}_y()$ （屏幕的下边界）。

如头文件 Point.h 中定义，Point 用一对整数（坐标值）表示：

```
struct Point {
    int x, y;
};

bool operator==(Point a, Point b) { return a.x==b.x && a.y==b.y; }
bool operator!=(Point a, Point b) { return !(a==b); }
```

Graph.h 中还定义了 Shape（将在第 19 章详细介绍）和 Line：

```
struct Line : Shape {           // 一个 Line 是由两个 Point 定义的一种 Shape
    Line(Point p1, Point p2);   // 由两个 Point 创建一个 Line
};
```

“: Shape”表示 Line 是一种 Shape，Shape 称为 Line 的基类（base class），或简称为基（base）。基本上，Shape 提供了一些能使 Line 的定义更为简单的特性。当我们觉得需要特殊形状，如 Line 和 Open\_polyline 时，我们将会对此进行解释（见第 19 章）。

Line 由两个 Point 定义。下面代码创建了 Line 对象，并将其绘制出来，我们省略了“基本框架”（#include 等语句，见 17.3 节）：

```
// 绘制两条线

constexpr Point x {100,100};

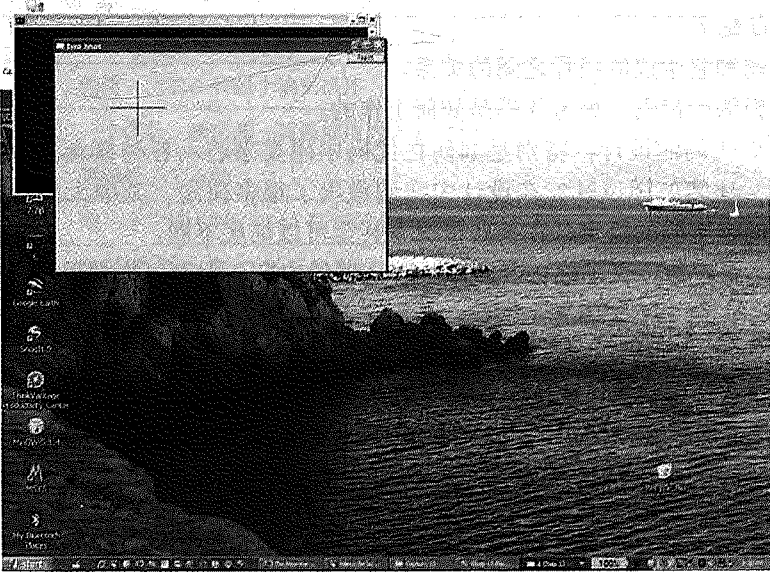
Simple_window win1 {x,600,400,"two lines"};

Line horizontal {x,Point{200,100}};           // 创建一条水平的线
Line vertical {Point{150,50},Point{150,150}}; // 创建一条垂直的线

win1.attach(horizontal);                     // 将线添加到窗口
win1.attach(vertical);

win1.wait_for_button();                       // 显示！
```

执行结果如下：



`Line` 的设计目标就是尽量简单，因此可以工作得很好。你不必像爱因斯坦那么聪明，就可以猜出下面语句：

```
Line vertical {Point{150,50},Point{150,150}};
```

语句创建一条从点 (150, 50) 到点 (150, 150) 的垂直线。当然，我们还不知道 `Line` 的实现细节，但创建 `Line` 对象时不必了解这些信息。实际上，`Line` 的构造函数的实现非常简单：

```
Line::Line(Point p1, Point p2) // 通过两个点创建一条线
{
    add(p1); // 将 p1 添加到该形状
    add(p2); // 将 p2 添加到该形状
}
```

构造函数只是简单地“添加”了两个点。但是，添加到哪里？`Line` 又是如何在窗口中绘制出来的？答案就在 `Shape` 类中。我们将在第 19 章对此进行介绍，现在你只需要了解，`Shape` 能够保存定义线的点，知道如何绘制点对构成的线，而且提供了函数 `add()` 允许一个对象将 `Point` 添加到 `Shape` 中。此处的关键点是，`Line` 的定义非常简单。大多数实现细节都已由“系统”完成了，因此我们可以将精力集中于编写易于使用的类。

从现在开始我们将忽略 `Simple_window` 的定义和 `attach()` 调用。虽然它们对于完整的程序来说是必不可少的框架，但对具体 `Shape` 的讨论却没有什么意思。

## 18.3 Lines

事实上，我们很少仅仅绘制一条线。我们通常思考问题都是针对包含很多线的对象，如三角形、多边形、路径、迷宫、网格、柱状图、数学函数、数据图等。最简单的“复合图形对象类”是 `Lines`：

```

struct Lines : Shape {
    Lines() {} // 相关的多条线 // 空对象
    Lines(initializer_list<Point> lst); // 通过一个点的列表进行初始化

    void draw_lines() const;
    void add(Point p1, Point p2); // 加入一条由两个点定义的线
};

```

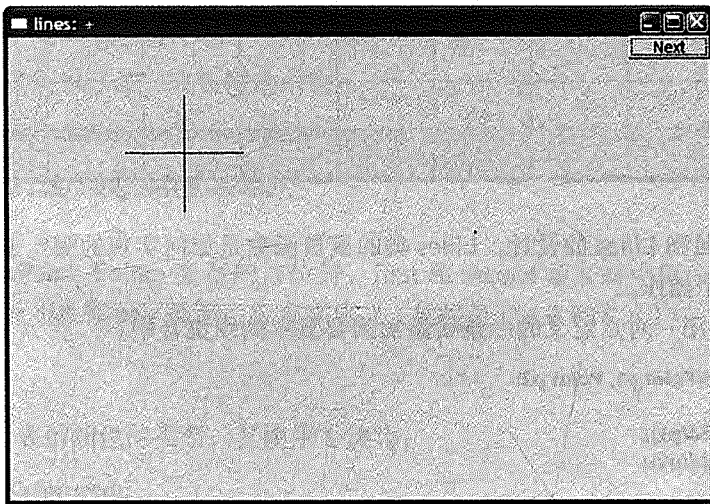
Lines 对象是简单的线的集合，每条线由一对 Point 定义。例如，将 18.2 节的 Line 的例子中的两条线作为单个图形对象的组成部分，我们可以这样定义它们：

```

Lines x;
x.add(Point{100,100}, Point{200,100}); // 第一条线：水平的
x.add(Point{150,50}, Point{150,150}); // 第二条线：垂直的

```

其输出结果很难与 Line 的例子区分开来。



区分此窗口与 18.2 节中窗口的唯一途径是给予两者不同的标签。

一组 Line 对象和 Lines 对象中的一组线的区别完全是我们看问题视角的不同。使用 Lines，我们是想表达两条线是联系在一起的，必须一起处理。例如，我们使用单个命令就可以改变 Lines 对象中所有线的颜色。而另一方面，我们却可以为不同的 Line 对象设置不同的颜色。一个更实际的例子是如何定义网格。网格是由许多等间隔的水平线和垂直线构成的。但是，我们将网格视为一个整体，于是将这些水平线和垂直线定义为一个名为 grid 的 Lines 对象的组成部分：

```

int x_size = win3.x_max(); // 获取窗口的大小
int y_size = win3.y_max();
int x_grid = 80;
int y_grid = 40;

Lines grid;
for (int x=x_grid; x<x_size; x+=x_grid)
    grid.add(Point{x,0},Point{x,y_size}); // 垂直线
for (int y = y_grid; y<y_size; y+=y_grid)
    grid.add(Point{0,y},Point{x_size,y}); // 水平线

```

注意：这里使用 `x_max()` 和 `y_max()` 获得窗口的尺寸，这也是我们第一个计算显示对象的代码。对于本例，使用一组 `Line` 对象的方式，为每条网格线定义一个命名变量，显然是无法忍受的，使用一个 `Lines` 对象是更合适的方式。这段代码执行结果如下：



让我们重新回到 `Lines` 的设计。`Lines` 类的成员函数是如何实现的呢？`Lines` 只提供了两个构造函数和两个操作。

`add()` 函数将用一对点定义的一条线添加到要显示的线集合中：

```
void Lines::add(Point p1, Point p2)
{
    Shape::add(p1);
    Shape::add(p2);
}
```

`add(p1)` 前需要加 `Shape::` 限定符，否则编译器将会调用 `Lines` 类的 `add()` 函数（非法的）而不是 `Shape` 类的 `add()` 函数。

`draw_lines()` 函数绘制 `add()` 定义的线：

```
void Lines::draw_lines() const
{
    if (color().visibility())
        for (int i=1; i<number_of_points(); i+=2)
            fl_line(point(i-1).x,point(i-1).y,point(i).x,point(i).y);
}
```

`Lines::draw_lines()` 一次获取两个点（从点 0 和 1 开始），并使用底层的画线库函数（`fl_draw()`）绘制两点之间的线。可见性是 `Lines` 类的 `Color` 对象的一个属性（见 18.4 节），所以我们在画线之前检查它是否可见。

如第 19 章所述，`draw_lines()` 是被“系统”调用的。我们不需要检查点的数目是否为偶数，因为 `Lines` 类的 `add()` 函数每次严格添加两个点。`Shape` 类定义了 `number_of_points()` 函数和 `point()` 函数（见 19.2 节），它们以只读方式访问 `Shape` 的点。因为成员函数 `draw_lines()` 不修改形状，因此将其定义为 `const` 类型（见 9.7.4 节）。

Lines 的默认构造函数只是简单地创建一个空对象（不包括任何线）：这种从没有任何点的空形状开始，根据需要逐步添加点的模型比使用其他构造函数更加灵活。我们当然也可以增加一个带初始化器列表的构造函数，参数中每个 Point 类都定义一条线。对于这样的带参数的构造函数（见 13.2 节），我们能够很简单地定义包含 1 条线、2 条线、3 条线等的 Lines。例如，可以用如下代码来描述第一个 Lines 的例子：

```
Lines x = {
    {Point{100,100}, Point{200,100}}, // 第一条线：水平的
    {Point{150,50}, Point{150,150}} // 第二条线：垂直的
};
```

或者，甚至是这样：

```
Lines x = {
    {{100,100}, {200,100}}, // 第一条线：水平的
    {{150,50}, {150,150}} // 第二条线：垂直的
};
```

很容易定义带初始化器列表的构造函数：

```
void Lines::Lines(initializer_list<pair<Point,Point>> lst)
{
    for (auto p : lst) add(p.first,p.second);
}
```

auto 是 pair<Point, Point> 类型的占位符，first 和 second 是点对中第一个和第二个成员的名称。initializer\_list 和 pair 类型由标准库定义（见附录 C.6.4 和 C.6.3）。

## 18.4 Color

Color 是用于表示颜色的类型，其使用方法为：

```
grid.set_color(Color::red);
```

该语句将 grid 中的线设置为红色的，于是我们得到





Color 定义了颜色的概念，并给出了一些常用颜色的符号名称：

```
struct Color {
    enum Color_type {
        red=FL_RED,
        blue=FL_BLUE,
        green=FL_GREEN,
        yellow=FL_YELLOW,
        white=FL_WHITE,
        black=FL_BLACK,
        magenta=FL_MAGENTA,
        cyan=FL_CYAN,
        dark_red=FL_DARK_RED,
        dark_green=FL_DARK_GREEN,
        dark_yellow=FL_DARK_YELLOW,
        dark_blue=FL_DARK_BLUE,
        dark_magenta=FL_DARK_MAGENTA,
        dark_cyan=FL_DARK_CYAN
    };

    enum Transparency { invisible = 0, visible=255 };

    Color(Color_type cc) :c{FL_Color(cc)}, v{visible} { }
    Color(Color_type cc, Transparency vv) :c{FL_Color(cc)}, v{vv} { }
    Color(int cc) :c{FL_Color(cc)}, v{visible} { }
    Color(Transparency vv) :c{FL_Color()}, v{vv} { } // 默认颜色

    int as_int() const { return c; }

    char visibility() const { return v; }
    void set_visibility(Transparency vv) { v=vv; }
private:
    char v; // 当前不可见或者可见
    FL_Color c;
};
```

Color 的目标是：

- 隐藏颜色的实现方式：FLTK 的 FL\_Color 类型；
- 将 FL\_Color 映射到 Color\_type 值；
- 给定颜色常量的范围；
- 提供一个简单的透明性机制（可见的或者不可见的）。

你可以按照以下方式选择颜色：

- 从命名颜色列表中选择，例如 Color::dark\_blue。
- 从一个小的“调色板”中选择，通过指定 0 ~ 255 之间的一个值，大多数颜色都能很好地在屏幕上显示。例如 Color(99) 为暗绿色，代码实例见 18.9 节。
- 从 RGB（红、绿、蓝）系统中选择一个值，我们不对这种方法进行详细介绍，若需要可以查阅相关资料。特别是，在互联网中搜索“RGB color”会找到很多资源，比如 [http://en.wikipedia.org/wiki/RGB\\_color\\_model](http://en.wikipedia.org/wiki/RGB_color_model) 和 [www.rapidtables.com/web/color/RGB\\_Color.htm](http://www.rapidtables.com/web/color/RGB_Color.htm)。相关内容见习题 13 和 14。

注意：构造函数可以利用 Color\_type 或者普通的 int 来创建 Color。所有版本的构造函数都将初始化成员变量 c。你可能认为 c 这个名字太短、太模糊，但由于它只在 Color 内部

很小的作用域内使用，不会用于更一般的用途，所以应该没有问题。在我们的设计中，数据成员 `c` 被声明为私有的，以避免用户直接使用它；它的类型被声明为 FLTK 中的 `FL_Color` 类型——我们不希望将 `FL_Color` 呈现给用户。但是，将颜色看作 `int` 值来表示其 RGB 或其他值也是很常见的，所以我们提供了 `as_int()` 函数。因为 `as_int()` 函数不会真正改变 `Color` 对象，故将其定义为 `const` 类型成员函数。

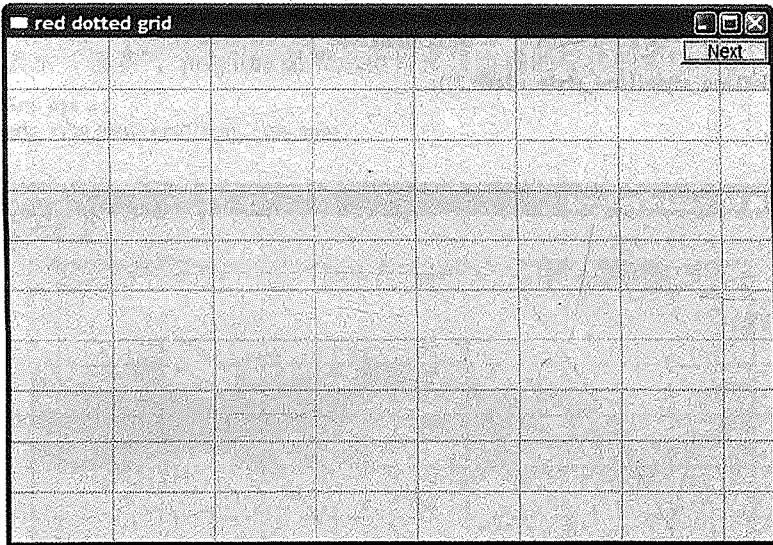
成员变量 `v` 的取值为 `Color::visible` 或 `Color::invisible`，表示颜色的透明性（可见的或者不可见的）。你可能觉得“不可见的颜色”很奇怪，但需要将复合形状的某一部分置为不可见时，这种方式很有效。

## 18.5 Line\_style

在一个窗口中绘制多条线时，可以通过颜色或线型，或者两者结合将它们区分开来。线型是用来描述线的外形的一种模式，可以按照如下方法使用 `Line_style`：

```
grid.set_style(Line_style::dot);
```

这条语句将 `grid` 中的线显示为点状线而非实线：



这使得网格看起来变“稀疏”了，更离散了。我们还可以调整线宽（粗细），以使网格线达到我们的喜好和要求。

`Line_style` 类型的定义如下：

```
struct Line_style {
    enum Line_style_type {
        solid=FL_SOLID,           // -----
        dash=FL_DASH,            // - - - -
        dot=FL_DOT,              // .....
        dashdot=FL_DASHDOT,      // - . - .
        dashdotdot=FL_DASHDOTDOT, // - . . .
    };

    Line_style(Line_style_type ss) :s{ss}, w{0} {}
};
```

```

Line_style(Line_style_type lst, int ww) :s{lst}, w{ww} { }
Line_style(int ss) :s{ss}, w{0} { }

    int width() const { return w; }
    int style() const { return s; }
private:
    int s;
    int w;
};

```

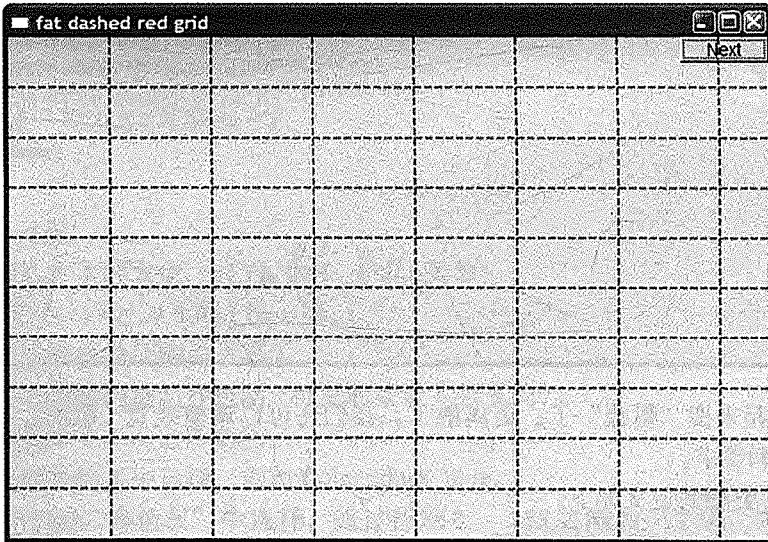
定义 `Line_style` 所使用的程序设计技术与 `Color` 的定义完全一样。我们隐藏了 FLTK 使用普通 `int` 型值表示线型的细节，为什么这样做呢？因为这些实现细节很可能随着库的升级而发生变化。下一个 FLTK 版本完全可能有专门的 `Fl_linestyle` 类型，我们也完全有可能使用其他 GUI 库来设计接口类。无论哪种情况，我们都不希望我们的代码或者用户的代码充斥着使用普通 `int` 值表示线型的片段。

☞ 大多数情况下，我们完全无须担心线型，使用默认线型就可以了（默认宽度和实线）。如果我们没有显式指定线宽，构造函数会设定默认线宽。设定默认值是构造函数所擅长的工作，而恰当的默认值对用户会有很大帮助。

注意：`Line_style` 有两个“分量”：模式（如虚线或实线）和宽度（线的粗细）。宽度用整数度量，默认值为 1。我们可以这样来设置粗虚线：

```
grid.set_style(Line_style(Line_style::dash,2));
```

运行结果如下：



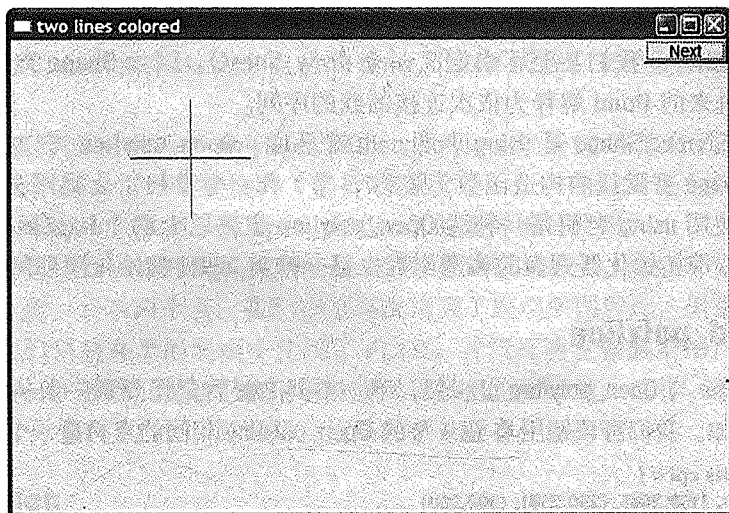
注意，颜色和线型设定会对形状中所有线起作用，这是将许多线组合为单个图形对象（例如 `Lines`、`Open_polyline`、`Polygon` 等）的好处之一。如果我们想分别控制线的颜色或线型，必须将它们定义为独立的 `Line` 对象。例如：

```

horizontal.set_color(Color::red);
vertical.set_color(Color::green);

```

运行结果如下：

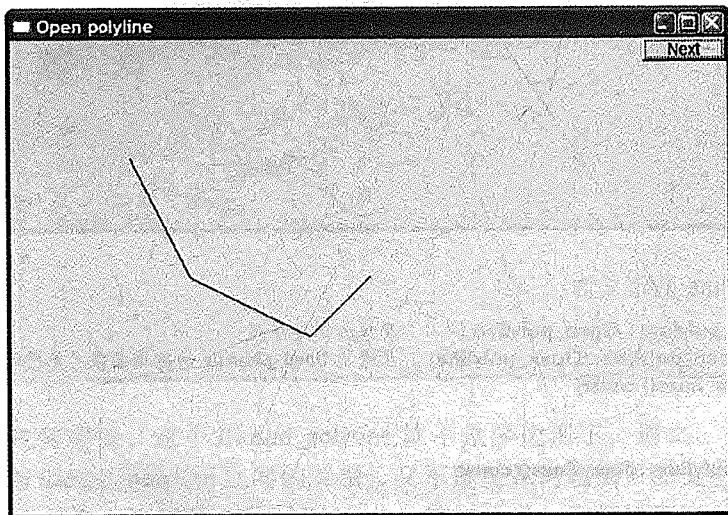


## 18.6 Open\_polyline

Open\_polyline 是由依次相连的线的序列组成的形状，由一个点的序列定义。poly 一词来源于希腊语，表示“很多”，polyline 是表示由许多线组成的形状的常用术语。例如：

```
Open_polyline opl = {
    {100,100}, {150,200}, {250,250}, {300,200}
};
```

连接所有的点可得如下形状：



本质上，Open\_polyline 不过是我们 在幼儿园时遇到的“点连线”游戏的好听一点的说法罢了。

Open\_polyline 类的定义如下：

```
struct Open_polyline : Shape {    // 线的开放序列
    using Shape::Shape;          // 使用 Shape 的构造函数（见附录 A.16）
    void add(Point p) { Shape::add(p); }
};
```

`Open_polyline` 继承了 `Shape`。`Open_polyline` 的 `add()` 函数允许用户访问 `Shape` 的 `add()` 函数 (即 `Shape::add()`)。我们甚至不必定义一个 `draw_lines()`，因为 `Shape` 类默认情况下会将 `add()` 函数添加进来的 `Point` 解释为依次连接的线的序列。

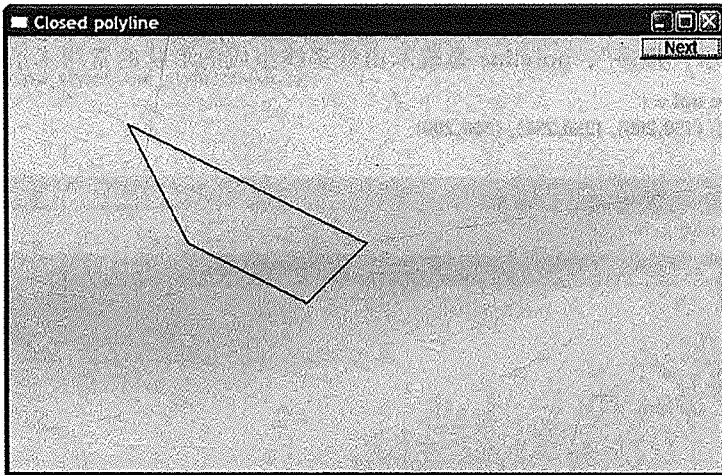
语句 `using Shape::Shape` 是 `using` 声明。也就是说，`Open_polyline` 可以使用 `Shape` 定义的构造函数。`Shape` 有默认的构造函数 (见 9.7.3 节) 和一个带初始化器列表的构造函数 (见 13.2 节)，所以使用 `using` 声明是一种为 `Open_polyline` 定义以上两个构造函数的简单速写方法。对于 `Lines`，带初始化器列表的构造函数也是一种对 `add()` 初始化序列的速写方法。

## 18.7 Closed\_polyline

`Closed_polyline` 与 `Open_polyline` 很相似，唯一不同之处就是还要画一条从最后一个点到第一个点的线。例如，我们可以使用与 18.6 节的 `Open_polyline` 相同的点构造一个 `Closed_polyline`：

```
Closed_polyline cpl = {
    {100,100}, {150,200}, {250,250}, {300,200}
};
```

除了最后的闭合线条之外，结果 (当然) 与 18.6 节的例子完全相同：



`Closed_polyline` 的定义为：

```
struct Closed_polyline : Open_polyline { // 线的闭合序列
    using Open_polyline::Open_polyline; // 使用 Open_polyline 的构造函数 (见附录 A.16)
    void draw_lines() const;
};

void Closed_polyline::draw_lines() const
{
    Open_polyline::draw_lines(); // 先画“开放的多线条部分”

    // 然后画闭合线
    if (2 < number_of_points() && color().visibility())
        fl_line(point(number_of_points()-1).x,
                point(number_of_points()-1).y,
                point(0).x,
                point(0).y);
}
```

using 声明语句（见附录 A.16）说明 `Closed_polyline` 的构造函数是和 `Open_polyline` 一样的。我们需要为 `Closed_polyline` 定义自己的 `draw_lines()` 函数，来绘制最后一个点到第一个点之间的闭合线。

我们只要编写完成 `Closed_polyline` 与 `Open_polyline` 差异的那部分代码即可。这是一种重要的程序设计技术，有时被称为“差异程序设计”——我们只需为派生类（本例中的 `Closed_polyline`）和基类（本例中的 `Open_polyline`）的差异编写代码。

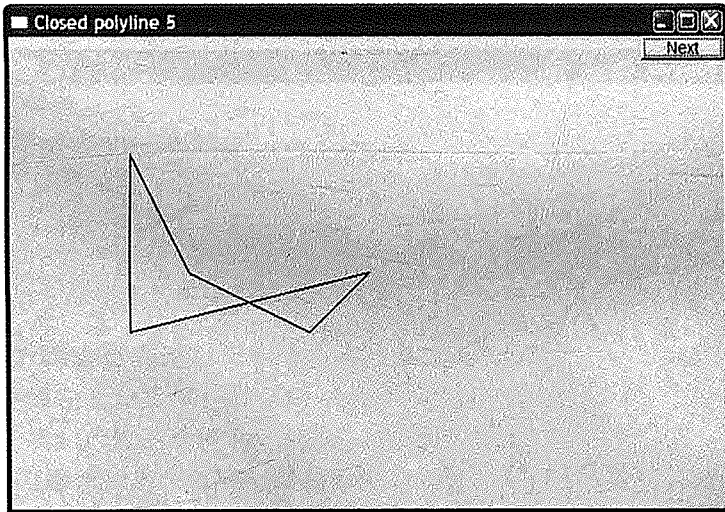
那么我们如何绘制闭合线呢？我们使用 FLTK 的画线函数 `fl_line()` 来完成这一工作。它接受 4 个整型参数，表示两个点。我们这里再次用到了底层的图形库。但是，请注意，与其他例子一样，我们只是在类的实现中使用了 FLTK，并没有将它暴露给用户。任何用户代码都无须引用 `fl_line()` 函数，或是了解用整数隐式表示点这类细节。这样，当我们需要时，就可以用其他 GUI 库替代 FLTK，而对用户代码几乎不会有任何影响。

## 18.8 Polygon

`Polygon` 与 `Closed_polyline` 非常相似，唯一的区别是 `Polygon` 不允许出现交叉的线。例如：18.7 节中的 `Closed_polyline` 是一个多边形，但如果再添加一个点：

```
cpl.add(Point{100,250});
```

运行结果为：



根据经典几何定义，这个 `Closed_polyline` 就不是多边形了。那么，如何定义 `Polygon` 才能正确利用与 `Closed_polyline` 之间的关系，又不违反几何规则？前文中有一个明显的暗示：`Polygon` 是不存在交叉线的 `Closed_polygon`。换句话说，我们就可以强调由点建立形状的过程，如果新添加的 `Point` 定义的线段不与 `Polygon` 任何现有的线相交时，这样的 `Closed_polyline` 就是 `Polygon`。

由此可定义 `Polygon` 如下：

```
struct Polygon : Closed_polyline { // 非交叉线的闭合序列
    using Closed_polyline::Closed_polyline; // 使用 Closed_polyline 的构造函数
    void add(Point p);
```

```

    void draw_lines() const;
};

void Polygon::add(Point p)
{
    // 检查新的线没有和现有的线交叉（这里没有给出代码）
    Closed_polyline::add(p);
}

```

我们在这里继承了 `Closed_polyline` 中 `draw_lines()` 函数的定义，这不但节省了工作量，还避免了重复代码。不幸的是，每次调用 `add()` 时，我们都需要检查是否有线段交叉。这导致一个低效的（平方阶的）算法——定义一个  $N$  个点构成的 `Polygon` 时，需调用  $N(N-1)/2$  次 `intersect()` 函数，因此算法的时间复杂度为  $N$  的平方阶。在实际应用中，我们假设 `Polygon` 类只用于顶点数比较少的多边形。例如，创建一个 24 个 `Point` 的 `Polygon` 需要调用 `intersect()` 函数  $24 \times (24-1)/2 = 276$  次，这还是可以接受的，但如果创建 2000 个顶点构成的多边形则需要大约 2 000 000 次函数调用，这时可能需要寻找更优的算法，接口可能也需要相应修改。

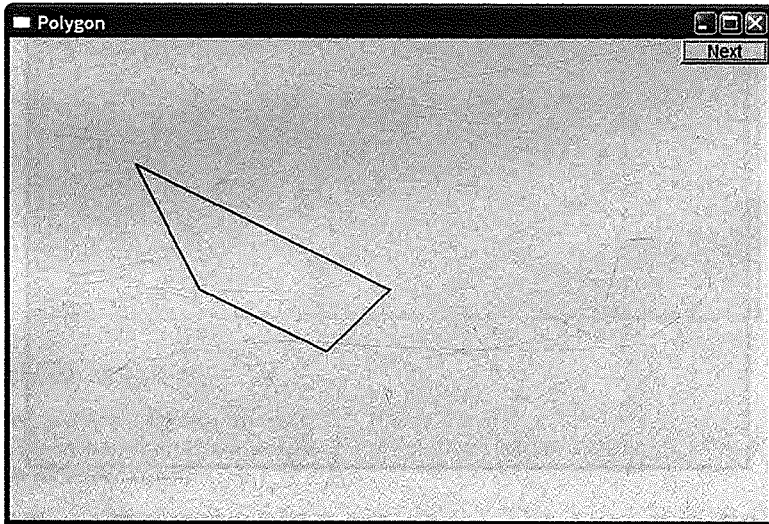
使用带初始化器序列的构造函数，我们可以这样创建多边形：

```

Polygon poly = {
    {100,100}, {150,200}, {250,250}, {300,200}
};

```

显然，该代码创建了一个与 18.7 节的 `Closed_polyline` 等价的 `Polygon`：



确保 `Polygon` 真正表示多边形是极其棘手的，`Polygon::add()` 函数中省略的相交检查是整个图形库中最复杂的部分。如果你对高精度几何坐标计算感兴趣，可以研究一下它的代码。

棘手的是，只有定义了所有点之后才能验证 `Polygon` 的不变式“这些点表示一个多边形”。也就是说，我们不能在构造函数中建立 `Polygon` 的不变式——虽然这是我们强烈建议的方式。我们考虑去掉 `add()` 函数，要求 `Polygon` 由至少包含 3 个点的初始化器列表完整定义，但在一个程序产生一系列的点时，这可能会导致使用比较复杂。

## 18.9 Rectangle

在屏幕上最常见的形状是矩形，部分是因为文化（大多数门、窗、照片、墙、书柜、页等都是矩形的），部分是因为技术（保证坐标位于矩形空间内，比任何其他形状都简单）。无论如何，矩形是如此常见，因而 GUI 系统直接支持矩形，而不是把它们当作四个角都是直角的凸多边形。

```
struct Rectangle : Shape {
    Rectangle(Point xy, int ww, int hh);
    Rectangle(Point x, Point y);
    void draw_lines() const;

    int height() const { return h; }
    int width() const { return w; }
private:
    int h; // 高度
    int w; // 宽度
};
```

使用两个顶点（左上角和右下角），或者一个顶点（左上角）和宽度、高度就可以定义矩形。构造函数可以定义如下：

```
Rectangle::Rectangle(Point xy, int ww, int hh)
    : w(ww), h(hh)
{
    if (h<=0 || w<=0)
        error("Bad rectangle: non-positive side");
    add(xy);
}

Rectangle::Rectangle(Point x, Point y)
    : w(y.x-x.x), h(y.y-x.y)
{
    if (h<=0 || w<=0)
        error("Bad rectangle: first point is not top left");
    add(x);
}
```

每个构造函数都会恰当地对成员变量 `h` 和 `w` 进行初始化（使用成员初始化语法，见 9.4.4 节），并将矩形左上角点保存在 `Rectangle` 的基类 `Shape` 中（使用 `add()`）。此外，还进行了完整性检查：我们当然不希望 `Rectangle` 的高度和宽度是负数。

一些图形 /GUI 系统对矩形特殊对待的原因之一是，判断哪些像素位于矩形内部的算法要比 `Polygon` 和 `Circle` 等其他形状简单得多，因而也快得多。因此，对于矩形，“填充”操作——也就是将区域内的像素设置为指定颜色的操作很常用，而其他形状则较少应用这个操作。我们可以在构造函数中设定填充颜色，或者用 `set_fill_color()` 函数进行设定（`Shape` 类提供的颜色相关的操作之一）：

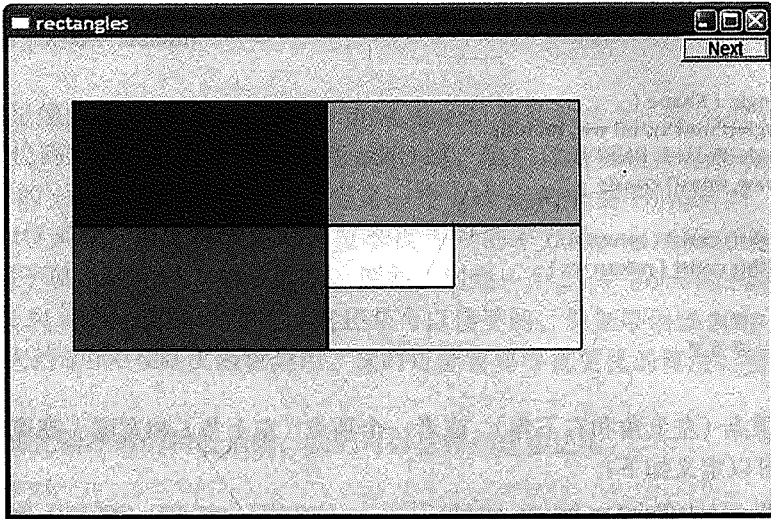
```
Rectangle rect00 {Point{150,100},200,100};
Rectangle rect11 {Point{50,50},Point{250,150}};
Rectangle rect12 {Point{50,150},Point{250,250}}; // 位于 rect11 下方
Rectangle rect21 {Point{250,50},200,100}; // 位于 rect11 右侧
Rectangle rect22 {Point{250,150},200,100}; // 位于 rect21 下方

rect00.set_fill_color(Color::yellow);
rect11.set_fill_color(Color::blue);
```



```
rect12.set_fill_color(Color::red);  
rect21.set_fill_color(Color::green);
```

运行结果为：

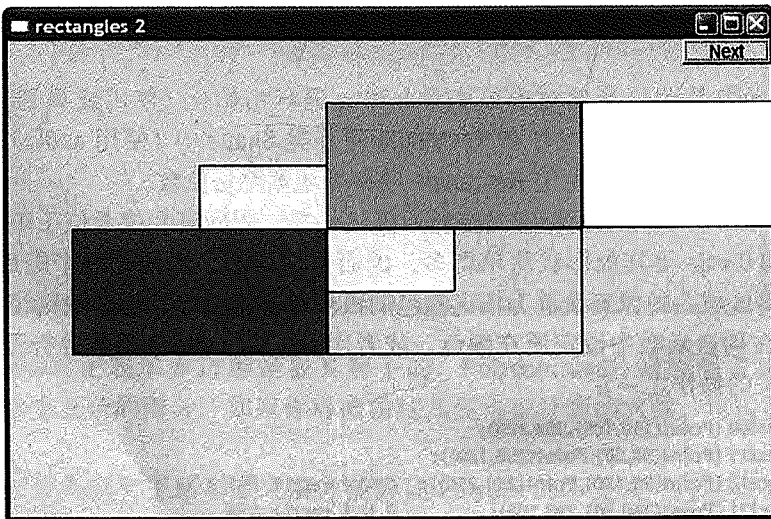


当不设定填充颜色时，矩形是透明的，这也是为什么在上图中你可以看到黄色矩形 rect00 的一角。

我们可以在窗口内部随意移动形状（见 19.2.3 节），例如：

```
rect11.move(400,0);    // 移动到 rect21 右侧  
rect11.set_fill_color(Color::white);  
win12.set_label("rectangles 2");
```

运行结果为：

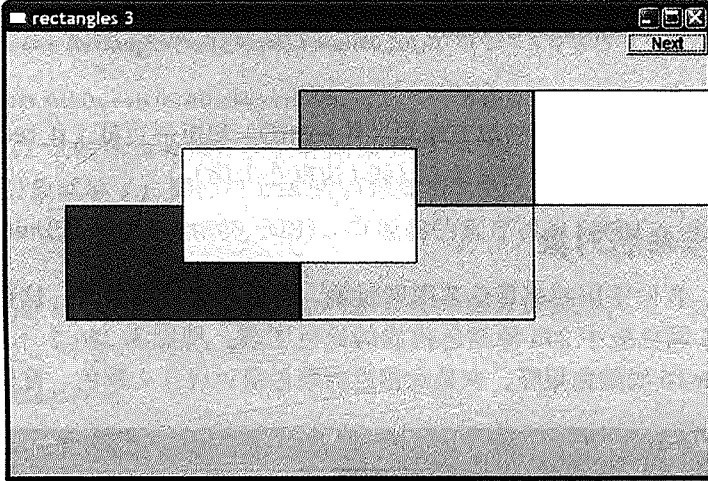


注意，白色矩形 rect11 只有一部分位于窗口之内，位于窗口之外的部分被“剪裁”掉了，不会在屏幕上显示。

另外请注意形状的层次，一个形状是如何放置在另一个的上层的。这与你将几张纸放在桌子上是一样的，最先放下的那张纸位于最底层。我们的 Window 类（见附录 E.3）提供了一种重排形状次序的简单方法，可以使用 `Window::put_on_top()` 函数通知窗口将一个形状放在最顶层。例如：

```
win12.put_on_top(rect00);  
win12.set_label("rectangles 3");
```

运行结果为：



注意，尽管矩形被填充了某种颜色（除了一个之外），但仍然可以看到它们的边框。如果不需要，可以将其去掉：

```
rect00.set_color(Color::invisible);  
rect11.set_color(Color::invisible);  
rect12.set_color(Color::invisible);  
rect21.set_color(Color::invisible);  
rect22.set_color(Color::invisible);
```

运行结果为：



注意，在填充颜色和线的颜色都被设置为 `invisible` 后，矩形 `rect22` 就看不到了。

由于 `Rectangle` 的 `draw_lines()` 函数必须处理线的颜色和填充颜色，因此实现变复杂了：

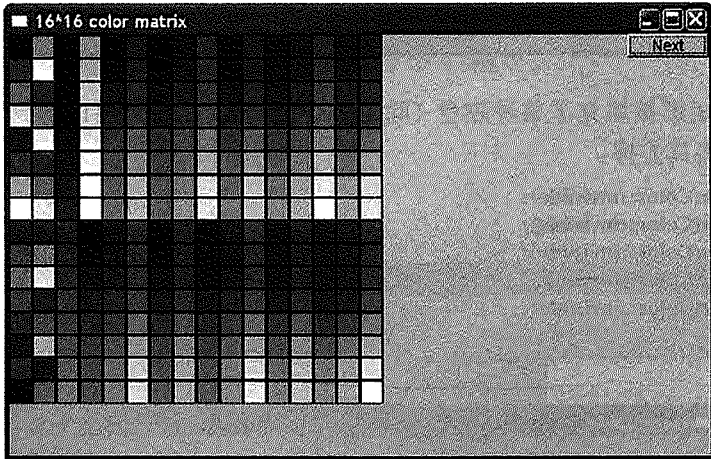
```
void Rectangle::draw_lines() const
{
    if (fill_color().visibility()) { // 填充
        fl_color(fill_color().as_int());
        fl_rectf(point(0).x,point(0).y,w,h);
    }

    if (color().visibility()) { // 填充区域上层的线
        fl_color(color().as_int());
        fl_rect(point(0).x,point(0).y,w,h);
    }
}
```

如你所见，FLTK 提供了绘制填充矩形 (`fl_rectf()`) 和矩形边框 (`fl_rect()`) 的功能。在我们的代码中，默认情况下两者均被绘制（线 / 边框在上层）。

## 18.10 管理未命名对象

到目前为止，我们使用的都是命名图形对象。当处理大量对象时，这种方法不再可行。例如，绘制 FLTK 调色板中 256 种颜色构成的比色图表，即绘制 256 个不同颜色填充的格子，构成一个  $16 \times 16$  的颜色矩阵，来显示相近的颜色值对应什么颜色。首先给出结果：



命名 256 个格子不但繁琐，而且相当不明智。左上角格子的一个显然的“命名”方式是它在矩阵中的位置  $(0, 0)$ ，其他任何一个格子都可以由其坐标  $(i, j)$  进行标识（“命名”）。我们在本例中需要做的是找到一种表示对象矩阵的方法。我们考虑过使用 `vector<Rectangle>`，但实践证明不够灵活。例如，不同类型未命名对象（元素）的集合应该是一种很有用的功能。我们将在 19.3 节讨论灵活性问题，这里只给出本例解决方案：采用能够保存已经命名和未命名对象的向量类型。

```
template<class T> class Vector_ref {
public:
    // ...
    void push_back(T&); // 加入一个已命名对象
    void push_back(T*); // 加入一个未命名对象
```

```
T& operator[](int i);    // 加下标: 便于读写访问
const T& operator[](int i) const;
```

```
int size() const;
```

```
};
```

它与标准库 `vector` 的使用方法非常类似:

```
Vector_ref<Rectangle> rect;
```

```
Rectangle x {Point{100,200},Point{200,300}};
rect.push_back(x);    // 加入已命名的
```

```
rect.push_back(new Rectangle(Point{50,60},Point{80,90})); // 加入未命名的
```

```
for (int i=0; i<rect.size(); ++i) rect[i].move(10,10);    // 使用 rect
```

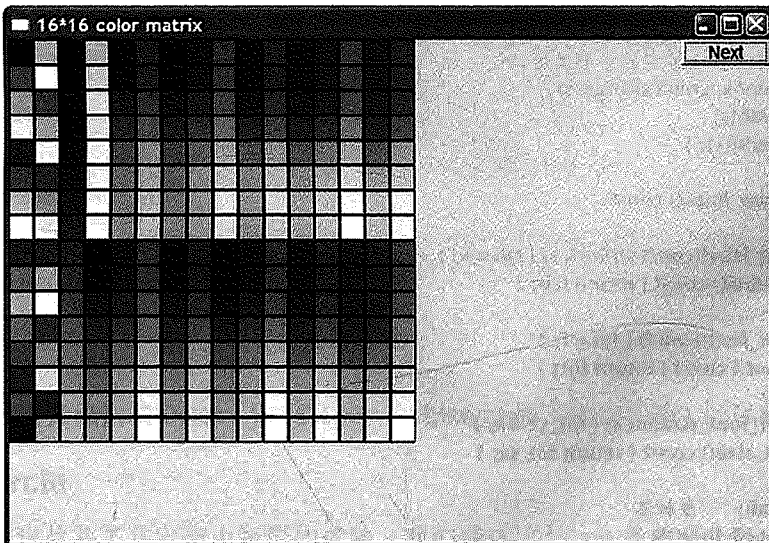
我们已在第 12 章解释 `new` 操作符, 在附录 E 给出 `Vector_ref` 的实现。现在, 知道可以用它保存未命名对象就够了。操作符 `new` 后面是类型名称 (如 `Rectangle`), 然后是可选初始化器列表 (如 `{Point{50, 60}, Point{80, 90}}`)。有经验的程序员可以放心, 我们并没有引入内存泄漏问题。

有了 `Rectangle` 和 `Vector_ref` 以后, 我们接下来就可以处理颜色了。例如, 我们可以这样实现上述具有 256 种颜色的比色图表:

```
Vector_ref<Rectangle> vr;
```

```
for (int i = 0; i<16; ++i)
    for (int j = 0; j<16; ++j) {
        vr.push_back(new Rectangle(Point{i*20,j*20},20,20));
        vr[vr.size()-1].set_fill_color(Color{i*16+j});
        win20.attach(vr[vr.size()-1]);
    }
```

这段代码创建了一个保存 256 个 `Rectangle` 的 `Vector_ref`, 这些矩形在 Window 中排列为  $16 \times 16$  的矩阵。我们将矩形的颜色设定为 0, 1, 2, 3, 4, ... 创建一个矩形后, 就将其添加到窗口中, 显示结果如下:

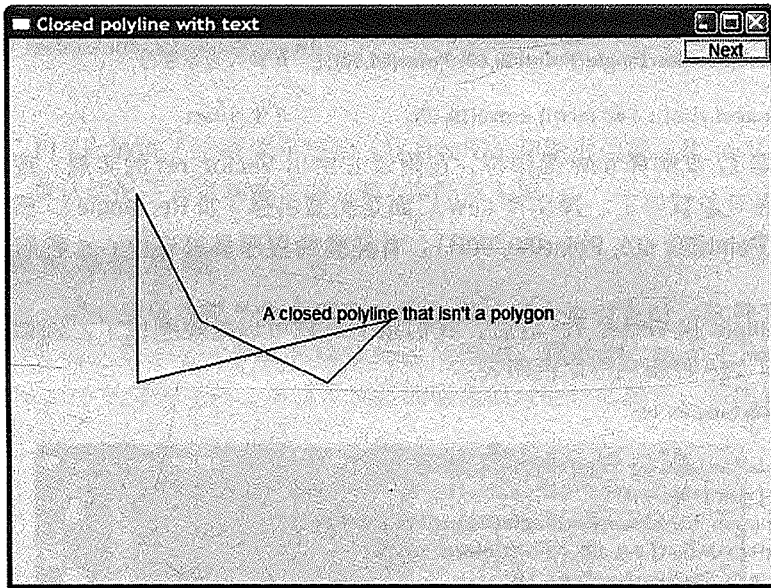


## 18.11 Text

很明显，我们需要在图形显示中添加文本的功能。例如，为 18.8 节中“奇怪”的 `Closed_polyline` 添加标签：

```
Text t (Point{200,200},"A closed polyline that isn't a polygon");
t.set_color(Color::blue);
```

运行结果为：



一个 `Text` 对象定义了起始位置为 `Point` 的一行文本，其中 `Point` 为文本行的左下角。限制为一行文本的原因是要保证跨系统的可移植性。不要尝试在字符串中放入换行符，它在窗口中不一定会产生换行效果。字符串流（见 11.4 节）对于构造 `Text` 对象中的 `string` 是很有用的（例子见 17.7.7 和 17.7.8 节）。`Text` 的定义如下：

```
struct Text : Shape {
    // 该点在第一个字符的左下角处
    Text(Point x, const string& s)
        : lab{s}
        { add(x); }

    void draw_lines() const;

    void set_label(const string& s) { lab = s; }
    string label() const { return lab; }

    void set_font(Font f) { fnt = f; }
    Font font() const { return fnt; }

    void set_font_size(int s) { fnt_sz = s; }
    int font_size() const { return fnt_sz; }
private:
    string lab; // 标签
    Font fnt {fl_font()};
```

```
int fnt_sz {(fl_size()<14)?14:fl_size()};
};
```

如果想把字符的字体大小调整到小于 14 或者大于 FLTK 的默认值，那么你必须进行显式设定。这个例子展示了用条件测试避免用户改变底层库的行为。在此例中，对 FLTK 的更新改变了它的默认设置，使得字符显示过小，可能令现有的程序不能正常工作。所以我们决定阻止这样的问题发生。

我们提供了初始化器作为成员初始化器，而不是作为构造函数初始化器列表的一部分，因为初始化器不依赖于构造函数的参数。

因为只有 Text 类知道其字符串是如何存储的，所以 Text 必须有自己的 draw\_lines() 函数：

```
void Text::draw_lines() const
{
    fl_draw(lab.c_str(),point(0).x,point(0).y);
}
```

字符颜色的设置类似于形状（如 Open\_polyline 和 Circle 等）中线的颜色的设置方法，你可以用 set\_color() 函数选择一种颜色，用 color() 函数获取当前使用的颜色。字号和字体的处理相似，下面列出了一小部分预定义字体：

```
class Font { // 字符的字体
public:
    enum Font_type {
        helvetica=FL_HELVETICA,
        helvetica_bold=FL_HELVETICA_BOLD,
        helvetica_italic=FL_HELVETICA_ITALIC,
        helvetica_bold_italic=FL_HELVETICA_BOLD_ITALIC,
        courier=FL_COURIER,
        courier_bold=FL_COURIER_BOLD,
        courier_italic=FL_COURIER_ITALIC,
        courier_bold_italic=FL_COURIER_BOLD_ITALIC,
        times=FL_TIMES,
        times_bold=FL_TIMES_BOLD,
        times_italic=FL_TIMES_ITALIC,
        times_bold_italic=FL_TIMES_BOLD_ITALIC,
        symbol=FL_SYMBOL,
        screen=FL_SCREEN,
        screen_bold=FL_SCREEN_BOLD,
        zapf_dingbats=FL_ZAPF_DINGBATS
    };

    Font(Font_type ff) :f{ff} {}
    Font(int ff) :f{ff} {}

    int as_int() const { return f; }
private:
    int f;
};
```

Font 类的定义风格与 Color（见 18.4 节）和 Line\_style（见 18.5 节）的风格是一样的。

## 18.12 Circle

为了说明世界并不是完全由矩形构成的，我们设计了 Circle 类和 Ellipse 类。Circle 是由

圆心和半径定义的：

```
struct Circle : Shape {
    Circle(Point p, int rr); // 圆心和半径

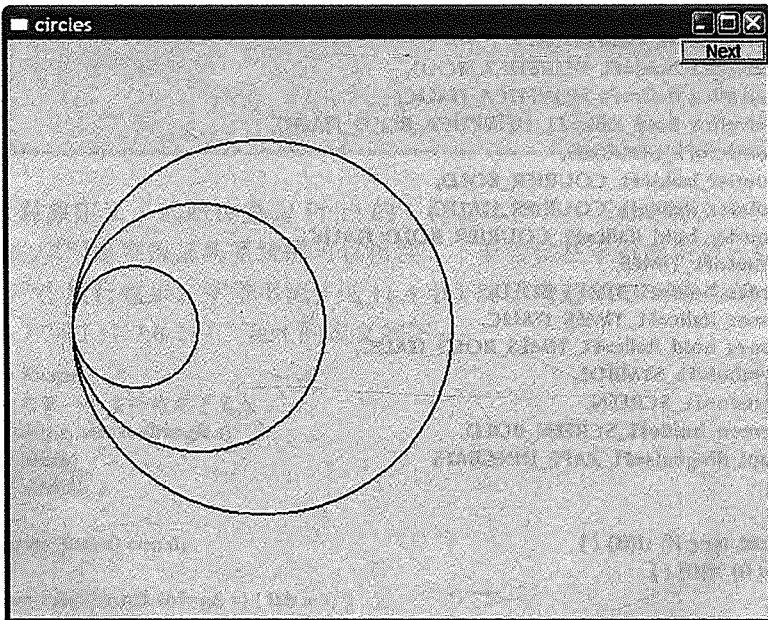
    void draw_lines() const;

    Point center() const ;
    int radius() const { return r; }
    void set_radius(int rr)
    {
        set_point(0,Point{center().x-rr,center().y-rr}); // 保持圆心
        r = rr;
    }
private:
    int r;
};
```

Circle 类的使用方法为：

```
Circle c1 {Point{100,200},50};
Circle c2 {Point{150,200},100};
Circle c3 {Point{200,200},150};
```

上述语句产生圆心在同一水平线上的三个不同半径的圆，运行结果为：



Circle 类实现的一个特别之处是，它所存储的点不是圆心，而是圆的正方边界的左上角。我们本来可以将两个点都存储起来，但最终选择存储了 FLTK 最优画圆函数所使用的那个。Circle 提供了一个很好的例子，展示了对于同一个概念，一个类如何用来呈现与其实现不同的（可能更好的）视角。

```
Circle::Circle(Point p, int rr) // 圆心和半径
    :r{rr}
```

```

{
    add(Point{p.x-r,p.y-r}); // 保存左上角的点
}

Point Circle::center() const
{
    return {point(0).x+r, point(0).y+r};
}

void Circle::draw_lines() const
{
    if (color().visibility())
        fl_arc(point(0).x,point(0).y,r+r,r+r,0,360);
}

```

注意如何使用 `fl_arc()` 函数绘制圆，其中头两个参数表示左上角，接下来两个参数表示外接矩形的宽度和高度，最后两个参数表示绘制的起止角度。环绕 360 度才绘制出一个圆，但我们也可以用 `fl_arc()` 函数绘制部分圆（椭圆），见习题 1。

## 18.13 Ellipse

椭圆与 `Circle` 类似，但通过长轴和短轴定义，而不是半径。也就是说，定义椭圆需要给出圆心坐标以及从圆心到  $x$  轴和  $y$  轴与椭圆交点的距离。

```

struct Ellipse : Shape {
    Ellipse(Point p, int w, int h); // 圆心，到圆心的最大和最小距离

    void draw_lines() const;

    Point center() const;
    Point focus1() const;
    Point focus2() const;

    void set_major(int ww)
    {
        set_point(0,Point{center().x-ww,center().y-h}); // 保持圆心
        w = ww;
    }
    int major() const { return w; }

    void set_minor(int hh)
    {
        set_point(0,Point{center().x-w,center().y-hh}); // 保持圆心
        h = hh;
    }
    int minor() const { return h; }
private:
    int w;
    int h;
};

```

可以这样使用 `Ellipse` 类：

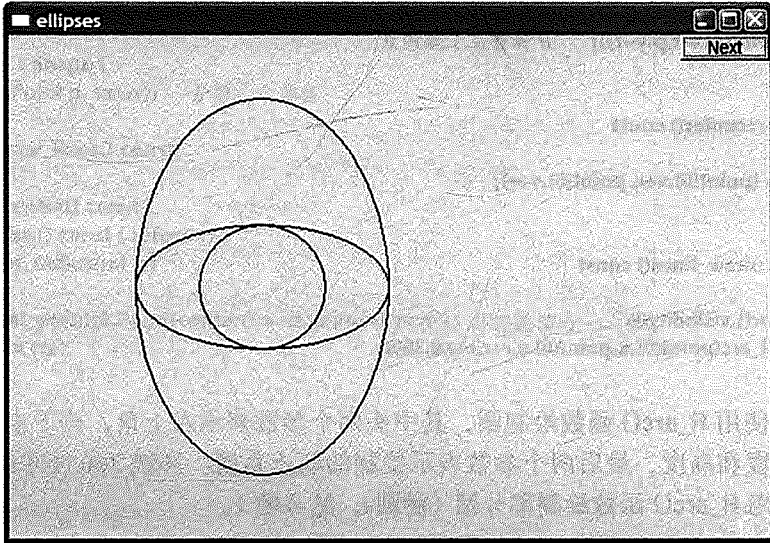
```

Ellipse e1 {Point{200,200},50,50};
Ellipse e2 {Point{200,200},100,50};
Ellipse e3 {Point{200,200},100,150};

```

上述语句产生圆心相同、长轴和短轴不同的三个椭圆，运行结果为：





注意，长轴与短轴相等 (`major()==minor()`) 的椭圆看起来就是一个圆。

另一种表示椭圆的常用方法是指定两个焦点和从一个点到两个焦点的距离之和。给定一个 `Ellipse`，可以计算出一个焦点。例如：

```
Point focus1() const
{
    if (h<=w) // 焦点在 x 轴上
        return {center().x+int(sqrt(double(w*w-h*h))),center().y};
    else // 焦点在 y 轴上
        return {center().x,center().y+int(sqrt(double(h*h-w*w))});
}
```

✂ 为什么一个 `Circle` 不是一个 `Ellipse`？从几何角度看，圆都是椭圆，但椭圆不一定是圆。特别地，圆是两个焦点相同的椭圆。假定把 `Circle` 定义为 `Ellipse`，我们就会在表示时付出额外的空间开销（圆由圆心和半径定义，椭圆由圆心和两个轴定义）。我们不喜欢在不需要额外空间的地方有额外的空间开销。更主要的原因是，如果不禁止 `set_major()` 和 `set_minor()` 函数，就无法将 `Circle` 定义为 `Ellipse`。毕竟，如果我们使用 `set_major()` 将长轴设置得与短轴不相等 (`major()!=minor()`)，就不是一个圆了（从数学家的角度），至少在设置之后就不再是圆了。我们不允许对象的类型发生变化，即一个对象不能在 `major()!=minor()` 时是椭圆，而在 `major()==minor()` 时又是圆。但是，能够允许的是一个 `Ellipse` 对象有时看起来像是圆。另一方面，`Circle` 永远不会变成两个轴不相等的椭圆。

☞ 在设计类时，我们应该小心不要自作聪明，也不要被“直觉”所欺骗，以至于设计出一些毫无意义的“类”来。相反地，我们应该注意如何用类表达某些关系密切的概念，不能设计成简单的数据和函数成员的集合。不思考要表达的思想 / 概念，只是将代码简单地堆积在一起，会制造出我们难以解释且其他程序员难以维护的“黑客代码”。如果你不是利他主义者，请记住“其他程序员”可能就包括几个月之后的你。另外，这种代码也很难调试。

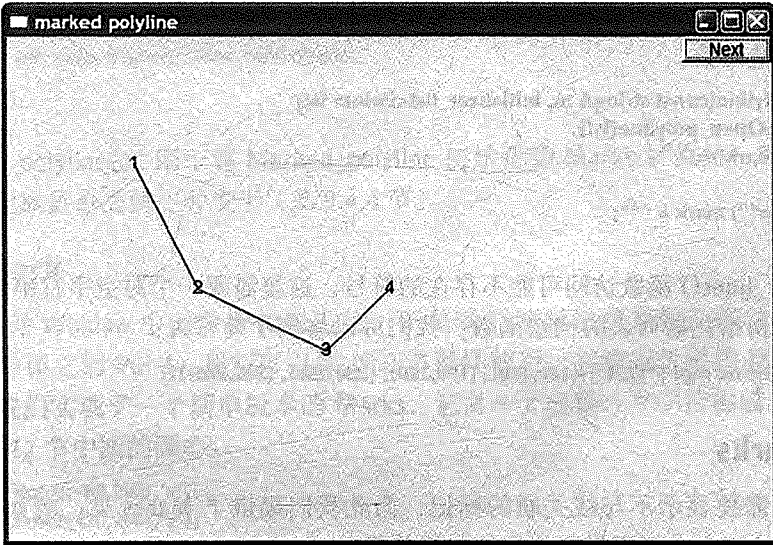
## 18.14 Marked\_polyline

我们常常需要对图中的点做“标记”。画图的一种方式开放的线段，因此我们只需

“标记”开放多段线的每个点即可。Marked\_polyline 就能实现这一目的，例如：

```
Marked_polyline mpl {"1234"};
mpl.add(Point{100,100});
mpl.add(Point{150,200});
mpl.add(Point{250,250});
mpl.add(Point{300,200});
```

运行结果为：



Marked\_polyline 的定义为：

```
struct Marked_polyline : Open_polyline {
    Marked_polyline(const string& m) : mark(m) { if (m=="") mark = ""; }
    Marked_polyline(const string& m, initializer_list<Point> lst);
    void draw_lines() const;
private:
    string mark;
};
```

它继承了 Open\_polyline 类，因此“免费”实现了对 Point 的处理，我们只需添加处理标记的代码。特别是，draw\_lines() 函数应修改为：

```
void Marked_polyline::draw_lines() const
{
    Open_polyline::draw_lines();
    for (int i=0; i<number_of_points(); ++i)
        draw_mark(point(i), mark[i%mark.size()]);
}
```

调用 Open\_polyline::draw\_lines() 负责划线操作，因此我们只需处理标记。我们将标记存储为一个字符串，按顺序选取其中字符：在创建 Marked\_polyline 时，使用 mark[i%mark.size()] 选择下一个显示的标记字符。其中 % 是模（取余）运算符，也就是说，我们循环使用数组 mark 来选取标记。这个版本的 draw\_lines() 函数使用一个小的辅助函数 draw\_mark()，完成在给定点实际输出一个字符：

```

void draw_mark(Point xy, char c)
{
    constexpr int dx = 4;
    constexpr int dy = 4;

    string m {1,c}; // 含有单个字符 c 的字符串
    fl_draw(m.c_str(),xy.x-dx,xy.y+dy);
}

```

其中，常量 `dx` 和 `dy` 用来使字符居中显示，字符串 `m` 被初始化为单个字符 `c`。

接受一个初始化器列表参数的构造函数简单地将列表转发给 `Open_polyline` 的接受初始化器列表的构造函数。

```

Marked_polyline(const string& m, initializer_list<Point> lst)
    :Open_polyline(lst),
    mark(m)
{
    if (m=="") mark = "*";
}

```

为了避免 `draw_lines()` 函数访问可能不存在的符号，这里需要一个对空字符串的检测。

有了接受初始化器列表的构造函数，我们可以将例子简写为：

```

Marked_polyline mpl {"1234",{{100,100}, {150,200}, {250,250}, {300,200}}};

```

## 18.15 Marks

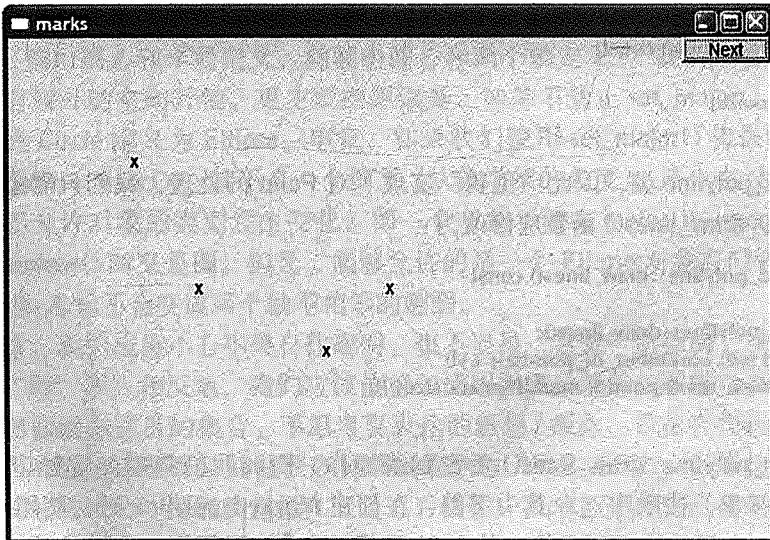
我们有时需要显示不与线关联的标记，为此我们提供了 `Marks` 类。例如，我们可以标记上例中用到的四个点而不需要将它们与线关联：

```

Marks pp {"x",{{100,100}, {150,200}, {250,250}, {300,200}}};

```

运行结果为：



`Marks` 的一个明显用途是显示表示离散事件的数据，对这类应用，用线连接各个数据点显然不合理。一个例子是一群人的（身高和体重）数据。

Marks 实际上是简单地将 `Marked_polyline` 的线设置为不可见 (invisible) 而实现的:

```
struct Marks : Marked_polyline {
    Marks(const string& m)
        :Marked_polyline(m)
    {
        set_color(Color{Color::invisible});
    }

    Marked_polyline(const string& m, initializer_list<Point> lst)
        : Marked_polyline(m, lst)
    {
        set_color(Color{Color::invisible});
    }
};
```

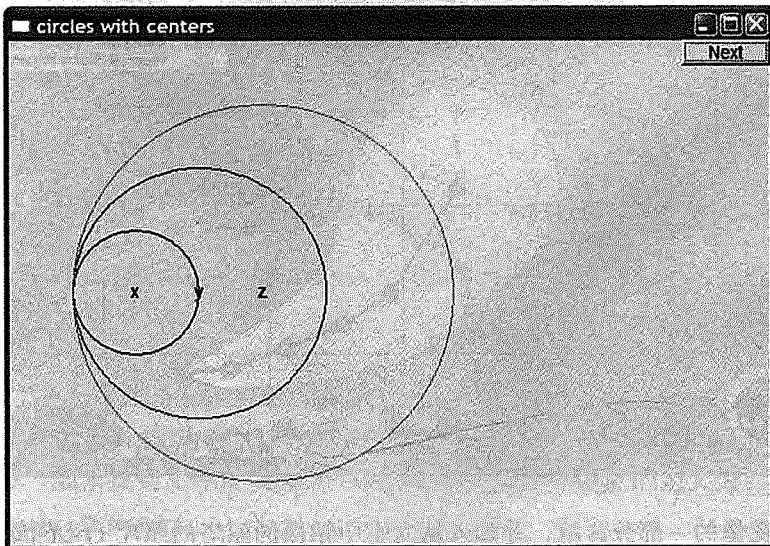
`:Marked_polyline(m)` 用于将 `Marked_polyline` 初始化为 `Marks` 对象的一部分。这种用法是用于初始化成员语法的一种变形 (见 9.4.4 节)。

## 18.16 Mark

`Point` 只是 `Window` 中的一个位置而已, 不是我们绘制的或是我们能看到的某种东西。若想标记一个孤立的 `Point`, 我们可以像 18.2 节那样使用一对线或者使用 `Marks`。但这有些繁琐, 因此我们实现了一个简单版本的 `Marks`, 它由一个点和一个字符构成。例如, 可以用它来标记 18.12 节中圆的圆心:

```
Mark m1 {Point{100,200}, 'x'};
Mark m2 {Point{150,200}, 'y'};
Mark m3 {Point{200,200}, 'z'};
c1.set_color(Color::blue);
c2.set_color(Color::red);
c3.set_color(Color::green);
```

运行结果为:



`Mark` 不过是一个初始化时立刻给定起始点 (通常也只有这一个点) 的 `Marks`:

```

struct Mark : Marks {
    Mark(Point xy, char c) : Marks(string{1,c})
    {
        add(xy);
    }
};

```

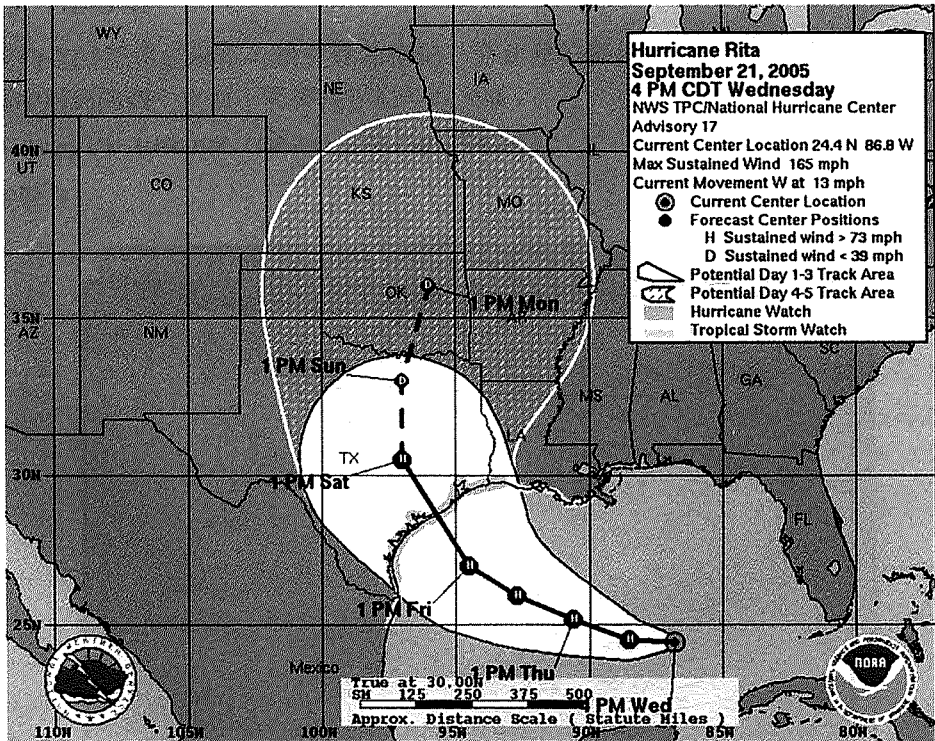
`string{1,c}` 是字符串 `string` 类的一个构造函数，初始化一个仅包含单个字符 `c` 的 `string` 对象。

`Mark` 的全部作用只是为标记为单个字符的、单个点的 `Marks` 对象提供了一种简化表示。对于“是否值得花力气定义这样一个类，它是否毫无意义，只是增加了复杂性和混乱”，还没有一个明确、合理的答案。我们反复推敲过这个问题，最后还是认为它对用户是有用的，而实现它的代价很小。

为什么使用字符作为“标记”？我们本可以使用任何小形状，但字符集合简单实用，很适合作为标记。能使用大量不同“标记”来区分不同点通常是很有用的。另外，像 `x`、`o`、`+` 和 `*` 等字符都具有中心对称性。

## 18.17 Image

平均每台 PC 机保存着数千个图像文件，而在网络上能找到的图像则数以百万计。我们自然希望即使是在很简单的程序中也能显示这些图像。例如，下图 (`rita_path.gif`) 是丽塔飓风到达德克萨斯墨西哥湾的路线图：



我们可以选择图像的一部分区域，并加入从太空中拍摄的丽塔的照片 (`rita.jpg`):

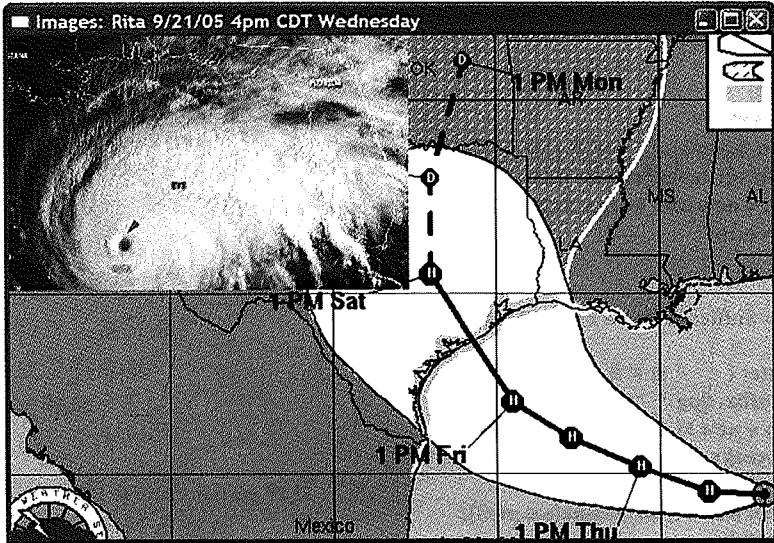
```

Image rita (Point{0,0}, "rita.jpg");
Image path {Point{0,0}, "rita_path.gif"};

```

```
path.set_mask(Point{50,250},600,400); // 选择可能是飓风着陆的位置

win.attach(path);
win.attach(rita);
```



set\_mask() 函数选择要显示图像的某个子图像。在本例中，它从 rita\_path.gif (载入到变量 path) 选择一个 600 × 400 像素大小的图像，其左上角在 path 中的坐标为 (50, 250)。这种操作非常常见，所以我们实现了 set\_mask 来直接支持它。

多个形状是按照它们添加的顺序确定层次次序的，就像将纸放在桌子上一样。由于 path 先于 rita 添加到窗口，所以它在 rita “下层”。

图像编码方式非常多，我们只处理最常用的两种 JPEG 和 GIF：

```
enum class Suffix { none, jpg, gif};
```

在我们的图形接口库中，图像在内存中用 Image 类对象表示：

```
struct Image : Shape {
    Image(Point xy, string file_name, Suffix e = Suffix::none);
    ~Image() { delete p; }
    void draw_lines() const;
    void set_mask(Point xy, int ww, int hh)
        { w=ww; h=hh; cx=xy.x; cy=xy.y; }
private:
    int w,h; // 为关联到 (cx,cy) 处的图像定义 “masking box”
    int cx,cy;
    FL_Image* p;
    Text fn;
};
```

Image 构造函数打开指定文件，然后按参数或者文件后缀名指定的编码格式创建图像。若图像无法显示（如未找到文件），则显示 Bad\_image。Bad\_image 的定义为：

```
struct Bad_image : FL_Image {
    Bad_image(int h, int w) : FL_Image(h,w,0) {}
    void draw(int x,int y, int, int, int) { draw_empty(x,y); }
};
```

在图形库中，图像处理是非常复杂的。但我们的图形接口库中的 `Image` 类的复杂性主要来自构造函数中的文件处理：

```
// 略微烦琐的构造函数
// 因为和图像文件有关的错误调试起来会比较痛苦
Image::Image(Point xy, string s, Suffix e)
    :w{0}, h{0}, fn{xy, ""}
{
    add(xy);

    if (!can_open(s)) { // 我们能打开 s 吗?
        fn.set_label("cannot open \""+s+"");
        p = new Bad_image(30,20); // 显示 "error image"
        return;
    }

    if (e == Suffix::none) e = get_encoding(s);

    switch(e) { // 检查是否为已知编码格式
    case Suffix::jpg:
        p = new FL_JPEG_Image(s.c_str());
        break;
    case Suffix::gif:
        p = new FL_GIF_Image(s.c_str());
        break;
    default: // 未知的编码格式
        fn.set_label("unsupported file type \""+s+"");
        p = new Bad_image(30,20); // 显示 "error image"
    }
}
```

我们通过文件名后缀来选择图像对象类型 (`FL_JPEG_Image` 或者 `FL_GIF_Image`)。使用 `new` 创建对象，并将地址赋予一个指针。这是一个与 `FLTK` 结构有关的实现细节，这里不详细讨论（见第 12 章对 `new` 操作符和指针的讨论）。`FLTK` 使用 C 语言风格的字符串，所以我们需要使用 `s.c_str()` 而不是 `s`。

现在，我们只需实现 `can_open()` 函数，来测试是否可以打开指定的文件进行读操作：

```
bool can_open(const string& s)
    // 检查名为 s 的文件是否存在以及是否可打开进行读操作
{
    ifstream ff(s);
    return ff;
}
```

打开一个文件然后再关闭的方法虽然比较笨拙，但对于区分“不能打开文件”错误和文件中数据格式错误却很有效，也具有很好的可移植性。

如果需要，你可以查阅 `get_encoding()` 函数。该函数可以抽取给定文件名的后缀，并在已知后缀名表中查找。后缀表是用标准库 `map` 容器实现的（见 16.6 节）。

## 简单练习

1. 创建一个  $800 \times 1000$  大小的 `Simple_window`。
2. 将窗口左侧的  $800 \times 800$  区域绘制为  $8 \times 8$  的网格（因此每个格子的大小为  $100 \times 100$ ）。
3. 将主对角线上的 8 个格子填充为红色（使用 `Rectangle`）。
4. 找一个  $200 \times 200$  像素大小的图像（`JPEG` 或 `GIF` 格式），在网格中放置它的 3 份拷贝（每

个图像占 4 个格子)。如果不能找到恰好为  $200 \times 200$  像素大小的图像, 使用 `set_mask()` 函数从大图像中选择一个  $200 \times 200$  的区域。注意, 不要挡住红色的格子。

5. 添加一个  $100 \times 100$  像素大小的图像, 当点击 “Next” 按钮时, 将它从一个格子移动到另一个格子。将 `wait_for_button()` 放在循环中, 并编写代码为图像选择下一个格子。

## 思考题

1. 为什么我们不直接使用商业的或者开源的图形库?
2. 为了实现简单的图形显示, 你大约需要使用我们的图形接口库中多少个类?
3. 为了使用图形接口库, 需要哪些头文件?
4. 哪些类定义了闭合形状?
5. 我们为什么不简单地使用 `Line` 表示所有形状?
6. `Point` 的参数的含义是什么?
7. `Line_style` 的成员有哪些?
8. `Color` 的成员有哪些?
9. 什么是 RGB?
10. 两条 `Line` 和包含两条线的 `Lines` 有什么区别?
11. 每种 `Shape` 都有的属性有哪些?
12. 由 5 个 `Point` (顶点) 定义的 `Closed_polyline` 有多少条边?
13. 如果你定义了 `Shape` 但没有添加到 `Window` 中, 你会看到什么?
14. `Rectangle` 与包含 4 个 `Point` (4 个顶点) 的 `Polygon` 有什么区别?
15. `Polygon` 与 `Closed_polygon` 有什么区别?
16. 填充 (`fill`) 和轮廓 (`outline`) 哪个在更上层?
17. 为什么我们没有定义一个 `Triangle` 类 (毕竟我们定义了 `Rectangle`)?
18. 在 `Window` 中怎样移动 `Shape`?
19. 怎样为 `Shape` 设置一行文本的标签?
20. 能够为 `Text` 对象中的文本串设置哪些属性?
21. 什么是字体? 我们为什么要关心字体?
22. `Vector_ref` 的作用是什么? 如何使用?
23. `Circle` 和 `Ellipse` 的区别是什么?
24. 如果指定的文件不包含图像, 当用该文件显示一个 `Image` 时会发生什么现象?
25. 如何显示图像的一部分?

## 术语

closed shape (闭合形状)	image (图像)	point (点)
color (颜色)	image encoding (图像编码)	polygon (多边形)
ellipse (椭圆)	invisible (不可见)	polyline (多段线)
fill (填充)	JPEG	unnamed object (未命名对象)
font (字体)	line (线)	<code>Vector_ref</code>
font size (字号)	line style (线型)	visible (可见的)
GIF	open shape (开放形状)	



## 习题

对每个“定义类”的练习，显示几个对象来验证其正确性。

1. 定义 `Arc` 类，绘制部分椭圆。提示：使用 `fl_arc()` 函数。
2. 定义由 4 条线和 4 个圆弧组成的 `Box` 类，绘制一个圆角矩形。
3. 定义 `Arrow` 类，绘制带有箭头的线。
4. 定义函数 `n()`、`s()`、`e()`、`w()`、`center()`、`ne()`、`se()`、`sw()` 和 `nw()`。每个函数接受一个 `Rectangle` 参数，返回一个 `Point`。它们定义了位于矩形的边上和内部的“连接点”。例如，`nw(r)` 是名为 `r` 的 `Rectangle` 的西北（左上）角。
5. 分别为 `Circle` 和 `Ellipse` 定义练习 4 给出的函数，使“连接点”位于图形轮廓上或外部，但不超出外接矩形。
6. 编写程序绘制一个类似于 17.6 节的类结构图，如果你先定义一个 `Box` 类表示带有文本标签的矩形，那么这个问题就简单了。
7. 创建一个 RGB 比色图表（在网络上搜索“RGB 比色图表”）。
8. 定义 `Regular_hexagon` 类（`regular hexagon` 为正六边形），构造函数的参数为中心和从中心到每个角的距离。
9. 用 `Regular_hexagon` 铺贴窗口的一部分区域（至少使用 8 个六边形）。
10. 定义 `Regular_polygon` 类，构造函数的参数为中心、边数（>2）和从中心到每个角的距离。
11. 绘制一个  $300 \times 200$  像素大小的椭圆，然后以圆心为原点，绘制长度分别为 400 和 300 像素的  $x$  轴和  $y$  轴。标记椭圆的两个焦点；标记椭圆边上不在坐标轴上的一个点，并绘制连接焦点到该点的两条线。
12. 绘制一个圆，然后沿圆周移动一个标记（每按一次“Next”按钮，标记移动一段距离）。
13. 绘制 18.10 节的颜色矩阵，但不显示每种颜色的边界线。
14. 定义直角三角形类，并使用 8 个不同颜色的直角三角形绘制一个八边形。
15. 用一些小的直角三角形铺贴窗口。
16. 用六边形重做上题。
17. 用一些不同颜色的六边重做上题。
18. 定义 `Poly` 类表示多边形，并在构造函数中判断给定点是否真的构成一个多边形。提示：需给定点作为构造函数的参数。
19. 定义 `Star` 类。其中一个参数为点的数目。使用不同数量的点、不同颜色的边、不同的填充颜色绘制一些星形图。

## 附言

第 17 章讲解了如何使用图形库的类。本章使我们上升到程序员“食物链”的更上一层：除了使用工具以外，还能设计工具。

# 设计图形类

实用的，持久的，优美的。

——维特鲁威

图形相关的这些章节有两个目的：我们希望为信息显示提供有用的工具，同时我们还希望通过一系列图形接口类来说明一般的设计与实现技术。特别地，本章介绍接口设计的思想和继承的概念。为此，我们需要先介绍一些和面向对象程序设计直接相关的语言特性：类派生、虚函数和访问控制。我们不认为能孤立于使用和实现来讨论设计，所以我们关于图形类设计的讨论是相当具体化的，或许你应该把这章看作“图形类的设计与实现”。

## 19.1 设计原则

我们的图形接口类的设计原则是什么？首先，这是一个什么类别的问题？什么是“设计原则”？我们为什么要考虑这些设计原则，而不是直接继续考虑如何生成图形这类重要的问题呢？

### 19.1.1 类型

图形是一个很好的应用领域的例子。因此，我们所关注的是如何为（像我们一样的）程序员提供一组基本的应用程序概念和工具，本章就给出了这样一个例子。如果我们的代码以混乱、不一致、不完整或者其他不好的方式呈现这些概念，生成图形输出的难度就会增大。我们希望我们的图形类能够降低程序员学习和使用的难度。

我们的程序设计理想是用代码直接描述应用领域概念。这样，如果你理解应用领域，你就能理解代码，反之亦然。例如：

- **Window**——一个由操作系统负责管理的窗口。
- **Line**——一条线，就如你在屏幕上所见。
- **Point**——一个坐标点。
- **Color**——颜色，就如你在屏幕上所见。
- **Shape**——所有形状统称（以我们的图形 /GUI 视角来看待世界时）。

最后一个例子 **Shape** 与其他例子不同，它是一个通用的、纯抽象的概念。我们永远无法在屏幕上看到一个“一般形状”；我们只能看到线、六边形这样的具体形状。这一点已经反映在我们的类型定义中：你可以尝试创建一个 **Shape** 变量，编译器将会阻止你。

我们的图形接口类构成一个库，这些类经常被组合在一起使用。它们给出了一个示例，当你定义描述其他图形形状类时，可以作为参考。这些类也可以作为基本组件，供你来构造描述其他形状的复杂的类。我们并不是仅仅定义了一些无关的类的集合，所以不能孤立地为每个类进行设计。这些类一起提供了一个如何生成图形的视图，我们必须确保这个视图是相当优雅和一致的。考虑到我们的库的规模，以及图形应用领域的庞大，我们显然不能对它

的完整性有什么期望。相反，我们的目标是简洁性和可扩展性。

事实上，没有类库能直接对其应用领域的各个方面进行建模。这不仅仅是不可能的，而且是毫无意义的。考虑编写一个用于显示地理信息的库，你希望显示植被吗？国家、州或者其他的行政边界呢？道路系统呢？铁路呢？河流呢？突出显示社会和经济数据吗？温度和湿度的季节性变化呢？大气层中风的模式呢？航空线路呢？需要标记学校的地点吗？快餐店的位置呢？地方景点呢？对于一个全面的地理应用来说，“这些都要！”可能是一个很好的答案。但对于简单的图形显示程序，显然不是。对于一个支持这类地理应用的库来说，包含所有上述功能可能是一个不错的方案。但是这样的库不太可能涵盖其他图形应用，例如徒手绘图、编辑照片图像、科学计算可视化以及航空器控制显示等。

✂ 所以，如往常一样，我们必须确定对我们来说什么是最重要的。对于图形库设计，就是要决定我们希望做好哪种图形 /GUI。试图做好所有事情通常会走向失败。好的库会从一个特定的角度直接、清晰地对其应用领域进行建模，强调应用的某些方面，对其他方面则不太关注。

我们提供的类都是用于简单的图形和简单的图形用户界面，它们主要针对那些需要表现数值和图形化输出的数值计算 / 科学计算 / 工程计算等应用领域的用户。你可以在这些类的基础之上创建自己的类。如果这还不够，我们已经在实现中提供了足够多的 FLTK 细节，如果你需要，可以从中找到如何更直接地使用它（或者是一个类似的完善的图形 /GUI 库）的方法。不过，如果你决定按照这样一条路线来做的话，请先熟练掌握第 12 章和第 13 章的内容。这两章包括一些指针和内存管理的相关内容，这都是直接使用大多数图形 /GUI 库所必需的。

☞ 我们的图形 /GUI 库的一个关键设计决策是提供大量的“小”类和较少的操作。例如，我们提供了 `Open_polyline`、`Closed_polyline`、`Polygon`、`Rectangle`、`Marked_polyline`、`Marks` 和 `Mark`，而不是带有很多参数和操作的单一类（可能命名为“`polyline`”），能通过这些参数和操作指定一个对象是哪一种多边形，甚至可能将一种多边形变化为另一种多边形。这种思路的极致就是只提供一个类 `Shape`，所有形状都归为 `Shape` 的一种情况。我们认为使用很多小类能够更加直接、有效地建模图形领域。一个提供“所有形状”的单一类，不具备一个能帮助理解、调试以及提高性能的框架，会使用户对数据和操作选项感到混乱。

### 19.1.2 操作

✂ 我们为每个类提供了最少的操作。我们的目标是用最小的接口来实现想做的事情。当我们需要更大的便利性时，可以通过增加非成员函数或新的类来实现。

☞ 我们希望所有类的接口有一致的风格。例如，在不同的类中，执行相似操作的所有函数有相同的函数名，接受相同类型的参数，还可能要求这些参数的顺序也相同。考虑设计这样一个构造函数：如果一个形状需要一个位置，该构造函数接受一个 `Point` 作为第一个参数。

```
Line ln {Point{100,200},Point{300,400}};
Mark m {Point{100,200},'x'}; // 显示单个点，标记为 'x'
Circle c {Point{200,200},250};
```

所有处理点的函数都使用 `Point` 类来表示点，这看起来是很显然的方式，但是很多类库都采用了多种风格的混合。例如，想象一个简单的画线函数，我们可以使用下面两种不同风

格中任何一种：

```
void draw_line(Point p1, Point p2); // 从 p1 到 p2 (我们的风格)
void draw_line(int x1, int y1, int x2, int y2); // 从 (x1, y1) 到 (x2, y2)
```

我们甚至可以同时允许这两种风格，但是出于一致性以及改进类型检查和可读性的考虑，我们选择第一种方式。一致地使用 `Point` 类还会避免将坐标和其他一般整数对（如宽度和高度）混淆。例如，考虑下面代码：

```
draw_rectangle(Point{100,200}, 300, 400); // 我们的风格
draw_rectangle(100,200,300,400); // 另一种方式
```

第一个调用利用一个 `Point`、一个宽度和一个高度绘制了一个矩形，我们可以很容易地推断出这些参数的含义。但是第二个调用呢？矩形是由 (100, 200) 和 (300, 400) 这两个点定义的吗？还是由一个点 (100, 200) 和宽度 300 以及高度 400 定义的呢？或者是完全不同的其他东西（对某些人可能是合理的）？而一致地使用 `Point` 类可以避免这种混淆。

顺便说一下，如果一个函数需要一个宽度值和一个高度值，那么实参总是按照这样的顺序给出（就像我们总是先给出  $x$  坐标，再给出  $y$  坐标一样）。在这种微小细节上保持一致性，会极大地方便使用，减少运行时错误。

逻辑上，等价的操作应该有相同的名字。例如，对任何类型的形状，所有添加点、线等的函数都叫作 `add()`，所有画线函数叫作 `draw_lines()`。这种一致性能帮助我们记忆（只需要记住更少的细节），同时在设计新类的时候也能给予我们帮助（按常规进行即可）。有时候，这种一致性甚至允许我们编写能用于很多不同类型的代码，因为这些类型上的操作有着同样的模式。这种代码被称为泛型程序，参见第 14 ~ 16 章。

### 19.1.3 命名

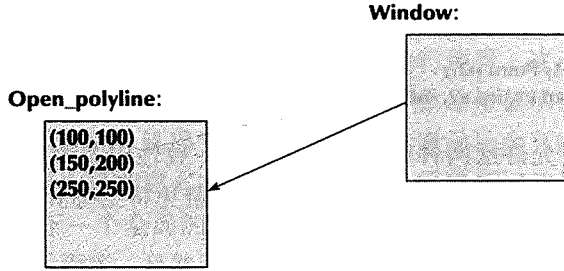
逻辑上，不同的操作应该有不同的名字。这看起来似乎是很显然的，但是请考虑：为什么我们将一个 `Shape` “添加” (`attach`) 到一个 `Window` 中，但却将一个 `Line` “加入” (`add`) 一个 `Shape` 呢？两个操作都是“将某物放到某物之中”，那么这种相似性是不是应该反映为相同的名字呢？不是！这种相似性之后隐藏了一个根本的不同点。考虑如下代码：

```
Open_polyline opl;
opl.add(Point{100,100});
opl.add(Point{150,200});
opl.add(Point{250,250});
```

这里，我们将 3 个点加入 `opl` 中。在 `add()` 调用完成之后，形状 `opl` 就不再关心“我们的”点了，而是自己为每个点维护一份副本。事实上，我们很少保存点的副本——我们将这个工作交给形状。而另一方面，考虑下面代码：

```
win.attach(opl);
```

这里，我们将窗口 `win` 和我们的形状 `opl` 关联起来；`win` 不会为 `opl` 生成一个备份——它仅仅是保存 `opl` 的一个引用。因此，在 `win` 使用 `opl` 期间，保证 `opl` 可用是我们的责任而不是 `win` 的责任。也就是说，在 `win` 使用 `opl` 的时候，我们不能让 `opl` 离开其作用域。我们可以更新 `opl`，`win` 下一次绘制 `opl` 时，所做的更改就会显示在屏幕上。`attach()` 和 `add()` 的区别如下图所示：



本质上，`add()` 的参数传递采用传值方式（拷贝副本）而 `attach()` 采用传引用方式（共享单一对象）。我们可以选择将图形对象拷贝到 `Window` 中，但那将是一个完全不同的程序设计模型，在那种模型中确实应该使用 `add()` 而不是 `attach()`。但在当前的模型中，我们只是将图形对象“添加”到 `Window` 中。这种模型有一些重要的暗示。例如，我们不能这样做：建立一个对象，将其添加到窗口中，接着将对象销毁，然后还希望程序能继续正常工作。

```

void f(Simple_window& w)
{
    Rectangle r {Point{100,200},50,30};
    w.attach(r);
} // r 的生命期在这里结束了

int main()
{
    Simple_window win {Point{100,100},600,400,"My window"};
    // ...
    f(win); // 这里会出现问题
    // ...
    win.wait_for_button();
}
  
```

⚠ 当我们已经退出 `f()` 函数，运行到 `wait_for_button()` 的时候，`win` 所引用和显示的对象 `r` 已经不存在了。在第 12 章中，我们将展示如何使函数内创建的对象在函数返回后还继续存在。但现在，我们必须避免将那些生命期在 `wait_for_button()` 之前就结束的对象添加到窗口。`Vector_ref`（参见 18.10 节和附录 E.4）可以帮助我们解决这个问题。

注意，如果我们将 `f()` 的 `Window` 参数声明为 `const` 引用类型（如 8.5.6 节推荐的那样），编译器会阻止我们犯这类错误：我们不能 `attach(r)` 到一个 `const Window`，因为 `attach()` 需要修改 `Window` 对象，以便记录 `r`。

#### 19.1.4 可变性

✂ 当我们设计一个类时，“谁可以修改其数据（描述）？”以及“如何修改？”是我们必须回答的关键问题。我们试图保证只有类自身能够修改其对象的状态。`public/private` 间的区别是实现这一效果的关键，但我们将给出使用更加灵活 / 微妙的机制（`protected`）的例子。这意味着我们不仅仅是为类提供一个数据成员，比如一个名为 `label` 的 `string` 对象；我们还必须考虑在构造之后是否允许修改它，以及如果允许的话，如何修改。我们还必须决定非成员函数是否需要读取 `label` 的值，以及如果需要的话，如何读取。例如：

```

struct Circle {
    // ...
private:
  
```

```

    int r;    // 半径
};

Circle c (Point{100,200},50);
c.r=-9;     // 可以吗？不——编译时错误：Circle::r 是私有的

```

正像你可能在第 18 章已经注意到的那样，我们决定阻止对大部分数据成员的直接访问。不直接暴露数据成员，使我们有检查那些“愚蠢”的数据，比如一个半径为负数的 Circle 对象。出于实现简单的考虑，我们只是有限地利用了这一机会，所以还是要小心处理你的数据。我们决定不进行一致的、全面的检查，一方面希望保持代码的简洁，另一方面是因为用户（你、我）提供的“愚蠢”数据只会在屏幕上绘制出乱七八糟的图像，而不会破坏珍贵的数据。

我们将屏幕（可看作一组 Window）当作一个纯粹的输出设备。我们可以在屏幕上显示新对象以及移除旧的对象，但不会向“系统”请求他人绘制的图像的信息，我们能获取的信息只来自自己创建的表示图像的数据结构。

## 19.2 Shape

Shape 类是一个一般概念，表示可显示在屏幕上 Window 中的对象：

- Shape 是一个概念，将图形对象与 Window 抽象关联起来，而 Window 提供了操作系统和物理屏幕之间的联系。
- Shape 是一个类，可以处理画线所用的颜色和线型。为了实现这一功能，Shape 中保存了一个 Line\_style 和一个 Color（用于线型和填充）。
- Shape 可以包含一个 Point 序列，以及绘制这些点的基本方法。

经验丰富的设计者会意识到，一个处理三方面工作的类很可能出现问题。但是，我们这里需要比一般解决方案更简单的方式，因此还是选用了这种设计策略。

我们首先给出完整的类，然后再讨论它的实现细节：

```

class Shape {    // 处理颜色和线型，包含一组线条
public:
    void draw() const;           // 处理颜色，画线
    virtual void move(int dx, int dy); // 将形状移动到 +dx 和 +dy 的位置

    void set_color(Color col);
    Color color() const;

    void set_style(Line_style sty);
    Line_style style() const;

    void set_fill_color(Color col);
    Color fill_color() const;

    Point point(int i) const;    // 只读方式访问所有的点
    int number_of_points() const;

    Shape(const Shape&) = delete; // 防止拷贝
    Shape& operator=(const Shape&) = delete;

    virtual ~Shape() {}
protected:
    Shape() {}

```

```

Shape(initializer_list<Point> lst); // add() 操作将 Point 加入到 Shape 中

virtual void draw_lines() const; // 绘制合适的线条
void add(Point p); // 将 p 加入到点集中
void set_point(int i, Point p); // points[i]=p;
private:
vector<Point> points; // 不是所有的形状都使用
Color lcolor {fl_color()}; // 线条使用的颜色和字符 (默认)
Line_style ls {0};
Color fcolor {Color::invisible}; // 填充颜色
};

```

这是一个相对复杂的类，用以支持各种各样的图形类，及表示屏幕上形状的一般概念。然而，它仍然只有 4 个数据成员和 15 个成员函数。而且，这些函数都较为简单，因此我们可以将注意力集中在设计方面。在本节的剩余部分中，我们将逐个研究这些类成员，并解释它们在设计中的作用。

### 19.2.1 一个抽象类

考虑 Shape 类的第一个构造函数：

```

protected:
Shape() {}
Shape(initializer_list<Point> lst); // add() 操作将 Point 加入到 Shape 中

```

构造函数是 protected 的，这意味着只有 Shape 类的派生类可以直接使用它（使用 :Shape 符号）。换句话说，Shape 只能用作其他类的基类，如 Line 和 Open\_polyline。“protected:”用于构造函数的目的是：保证我们不直接创建 Shape 对象。例如：

```
Shape ss; // 错误：不能创建 Shape
```



Shape 被设计为只能当作一个基类。在这种情况下，如果我们允许直接创建 Shape 对象，不会发生什么特别不好的事情；但由于可以限制性使用，我们仍然保留了修改 Shape 对象的权限，这使得 Shape 类不适于直接使用。同样，通过禁止直接创建 Shape 对象，我们直接实现了这样一种思想：不能创建 / 显示一般性的形状，而只能创建 / 显示特定的形状，例如 Circle 或者 Closed\_polyline。仔细思考一下这一思想！一个形状看起来是什么样子？唯一合理的回答是反问“形状是什么？”。我们通过 Shape 类所描述的形状概念是一个抽象的概念。这是一种重要的、很常用也很有用的设计思想，因此我们不希望在程序中实践这一思想时打折扣。允许用户直接创建 Shape 对象会违背我们用类直接表示概念这一目标。

默认的构造函数将成员设置为各自的默认值。这里要再次强调，实现中使用的底层库 FLTK 完成了实际工作。然而，这里对 FLTK 的使用并没有直接提及 FLTK 的颜色和风格的概念，它们只是作为 Shape、Color 和 Line\_style 类实现的一部分。vector<Points> 的默认值为空向量。

初始化器列表构造函数也可以使用默认的初始化程序，然后使用 add() 操作将参数列表中的元素加入到 Shape：

```

Shape::Shape(initializer_list<Point> lst)
{
    for (Point p : list) add(p);
}

```



如果一个类只能被用作基类，它就是一个抽象（abstract）类。另一种更常用的定义抽

象类的方法称为纯虚函数 (pure virtual function), 参见 19.3.5 节。与抽象类相对的是具体 (concrete) 类, 即可以创建对象的类。注意, 抽象和具体是一对非常简单的技术词汇, 我们可能每天都会用到它们来表示区别。我们可能去商店买一台照相机, 但是不会只向售货员要一台“照相机”带回家。照相机是什么牌子的? 具体型号是什么? 单词“照相机”是一个通称, 它代表一个抽象的概念。而 Olympus E-M5 代表具体的一类照相机, 而我们 (花费一大笔钱) 可以获得它的一个特定实例: 一个具有唯一序列号的特定的照相机。所以说, “照相机”更像一个抽象类 (基类), “Olympus E-M5”更像一个具体类 (派生类), 而我手中的真实的照相机 (如果我买了它) 则更像一个对象。

声明

```
virtual ~Shape() {}
```

定义了一个虚析构造函数, 参见 12.5.2 节。

## 19.2.2 访问控制

Shape 类将所有数据成员均声明为 private:

private:

```
vector<Point> points;
Color lcolor {fl_color()}; // 线条使用的颜色和字符 (默认)
Line_style ls {0};
Color fcolor {Color::invisible}; // 填充颜色
```

数据成员的初始化程序不依赖于构造函数的参数, 所以我们在数据成员声明中具体确定它们。像以往一样, 向量的默认值是“空”, 所以不需要对其进行明确。构造函数将使用这些默认值。

因为 Shape 类的数据成员被声明为 private, 因此我们需要为它们提供访问函数。访问函数的设计有多种风格, 我们选择了一种较为简单、方便、易读的方式。如果有一个成员代表一个属性 X, 我们可以提供一对函数 X() 和 set\_X() 分别用于该成员 (属性) 的读和写。例如:

```
void Shape::set_color(Color col)
{
    lcolor = col;
}

Color Shape::color() const
{
    return lcolor;
}
```

这种风格最主要的不便之处在于不能将成员变量和读取函数设定为相同的名字。像以往一样, 我们将最方便的名字赋予函数, 因为它们是公共接口的一部分, 而 private 类型的变量命名就不那么重要了。注意, 我们用 const 指出读取函数不能修改 Shape 对象 (参见 9.7.4 节)。

Shape 类保存了一个名为 points 的 Point 向量, Shape 负责它的维护, 用来支持其派生类。我们提供了将 Point 对象添加到 points 中的函数 add():

```
void Shape::add(Point p) // 保护的
{
    points.push_back(p);
}
```



`points` 初始时当然应该是空的。我们决定为 `Shape` 提供一个完整的功能接口，而不是让用户（即使是 `Shape` 的派生类的成员函数）直接访问数据成员。对于某些人来说，提供功能接口是非常正常的，因为他们觉得将类的成员设计为公有（`public`）是不好的设计。而对于另一些人，我们的设计看起来过于严格了，因为我们甚至不允许派生类的成员函数直接对数据成员进行访问。

一个派生自 `Shape` 的形状（比如 `Circle` 和 `Polygon`）是了解点（`point`）的含义的。基类 `Shape` 则并不“理解”这些点，它只是存储它们。因此，派生类需要控制如何添加点。例如：

- `Circle` 和 `Rectangle` 不允许用户添加点，因为添加点没有任何意义。一个矩形加一个额外的点又是什么呢？（参见 17.7.6 节。）
- `Lines` 只允许添加成对的点（而不是一个单独的点，参见 18.3 节）。
- `Open_polyline` 和 `Marks` 允许添加任意多个点。
- `Polygon` 只允许通过具有相交性检查功能的 `add()` 函数来添加点（参见 18.8 节）。

我们将 `add()` 设计为 `protected`（即只能从派生类进行访问），保证由派生类来控制如何添加这些点。如果 `add()` 函数为 `public`（任何人都可以添加点）或者 `private`（只有 `Shape` 可以添加点），就会使得实际功能无法符合我们对形状的理想。

同样，我们将 `set_point()` 设计为 `protected`。即，只有派生类能知道点的含义是什么以及是否可以在不违反不变式的前提下修改它。例如，如果我们有一个 `Regular_hexagon` 类，定义为六个点的集合，即使只改变一个点也有可能使图形不再是一个“正六边形”。而另一方面，如果改变四边形的一个点，其结果仍然会是一个四边形。事实上，在示例类和代码中，我们并没有发现有对 `set_point()` 函数的需求，因此 `set_point()` 在这里只是为了保证我们能够读取和设置 `Shape` 每个属性的设计原则仍旧成立。例如，如果来实现一个 `Mutable_rectangle` 类，我们可以从 `Rectangle` 类派生，并且提供更改变点的操作。

我们将存放 `Point` 的向量 `points` 设计为 `private`，以保护它不会被意外地修改。为了能使它有用，我们还需要提供成员函数实现对它的访问：

```
void Shape::set_point(int i, Point p) // 不会被用到，目前为止也不需要
{
    points[i] = p;
}

Point Shape::point(int i) const
{
    return points[i];
}

int Shape::number_of_points() const
{
    return points.size();
}
```

在派生类的成员函数中，这些函数的使用方法如下：

```
void Lines::draw_lines() const
// 连接点对进行画线
{
    for (int i=1; i<number_of_points(); i+=2)
        fl_line(point(i-1).x,point(i-1).y,point(i).x,point(i).y);
}
```

你可能会担心那些琐碎的访问函数。它们是不是很低效？会不会使程序变慢？会不会增加程序生成代码的大小？不会的，它们都会被编译器以“内联”（`inlined`）方式进行编译。实际上，调用 `number_of_points()` 跟直接调用 `points.size()` 使用一样多的内存，执行一样多的指令。

这些访问控制的考虑和决定是非常重要的，接近于最小版本的 `Shape` 类可以定义如下：

```
struct Shape { // 接近最小定义——太简单——不会被用到
    Shape();
    Shape(initializer_list<Point>);
    void draw() const; // 处理颜色并调用 draw_lines
    virtual void draw_lines() const; // 绘制合适的线
    virtual void move(int dx, int dy); // 将形状移动到 +dx 和 +dy 的位置
    virtual ~Shape();

    vector<Point> points; // 不会被所有的形状使用
    Color lcolor;
    Line_style ls;
    Color fcolor;
};
```

我们增加的 12 个成员函数和两行访问控制说明（`private:` 和 `protected:`）有何价值呢？其基本作用是保护类的描述不会被设计者以不可预见的方式更改，从而使我们能用更少的精力写出更好的类。这就是所谓的“不变式”（见 9.4.3 节）。下面，我们将通过定义 `Shape` 类的派生类来说明这一优点。一个简单的例子是 `Shape` 类的早期版本用到了下面两个成员：

```
FL_Color lcolor;
int line_style;
```

这种实现方式被证明局限性太大（线型为 `int` 类型不能完美地表示线宽，而 `FL_Color` 不能表示不可见方式），并且使得代码凌乱。如果这两个变量是公有的（`public`），并被用户代码所使用，那么改进接口库就只能以重写这些用户代码为代价（因为在用户代码中使用了名字 `lcolor` 和 `line_style`）。

另外，访问函数在符号表示方面更为方便。例如，`s.add(p)` 比 `s.points.push_back(p)` 更易读、易写。

### 19.2.3 绘制形状

我们现在已经介绍了除 `Shape` 类核心之外的所有内容：

```
void draw() const; // 处理颜色并调用 draw_lines
virtual void draw_lines() const; // 绘制合适的线
```

`Shape` 最基本的功能是绘制形状。但我们不可能将其他所有功能、数据都去掉，而又不 对 `Shape` 造成损害（参见 19.4 节）。绘制是 `Shape` 的本职工作，它借助 `FLTK` 和操作系统的 基本机制来完成这一工作。但是，从用户的观点来看，它只是提供了两个函数：

- `draw()` 函数首先应用线型和颜色设置，然后调用 `draw_lines()`。
- `draw_lines()` 在屏幕上绘制像素。

函数 `draw()` 并没有使用任何新奇的技术，只是简单地调用 `FLTK` 函数来设置 `Shape` 中 指定的颜色和线型，接着调用 `draw_lines()` 函数在屏幕上进行实际的绘制，最后将颜色和线 型恢复到调用之前的情况：

```

void Shape::draw() const
{
    Fl_Color oldc = fl_color();
    // 没有好的可移植方式来获得当前的线型
    fl_color(lcolor.as_int());           // 设置颜色
    fl_line_style(ls.style(),ls.width()); // 设置线型
    draw_lines();
    fl_color(oldc);                     // 重置颜色 (改为绘制之前颜色)
    fl_line_style(0);                   // 重置线型为默认值
}

```

⚠ 不幸的是，FLTK 并没有提供获得当前线型的方法，所以线型只是设置为默认值。这就是有些时候为了简单性和可移植性而不得不接受的妥协。我们认为，试图在接口库中实现这一功能是不值得的。

注意，`Shape::draw()` 函数并不处理填充颜色或者线的可见性，这些都由单独的函数 `draw_lines()` 来完成，它对如何解释这些设置有更好的了解。原则上，所有颜色和线型都可以交给 `draw_lines()` 函数来处理，但是重复性会相当高。

✂ 现在考虑我们应该如何处理 `draw_lines()` 函数。如果你稍微想一下，就会明白让 `Shape` 类的一个函数来完成多种不同形状的绘制是非常困难的。那样的话，可能需要在 `Shape` 对象中保存每个形状的最后像素。如果我们继续使用 `vector<Point>` 模型，将会存储非常多的点。更糟的是，“屏幕”（即图形硬件）已经做了这些，而且做得更好。

✂ 为了避免额外的工作和存储空间，`Shape` 类采用了另外一种方式：它为每种 `Shape`（即每个 `Shape` 的派生类）都提供了定义自己的绘制函数的机会。`Text`、`Rectangle` 或 `Circle` 类都可能适合自己的更好的绘制方法。事实上，大部分形状类都是这样。毕竟，这些类确切地“知道”它们所要绘制的内容。例如，将 `Circle` 类定义为一个点和一个半径，远好于许多线段。在需要的时候，通过一个点和一个半径生成要绘制的像素实际上并不像想象的那么困难。因此 `Circle` 类定义了自己的 `draw_lines()` 函数，我们更希望调用这个函数而不是 `Shape` 类的 `draw_lines()` 函数。这就是将 `Shape::draw_lines()` 声明为 `virtual` 的意义所在：

```

struct Shape {
    // ...
    virtual void draw_lines() const; // 让每个派生类定义自己的 draw_lines(), 如果它们选择这么做的话
    // ...
};

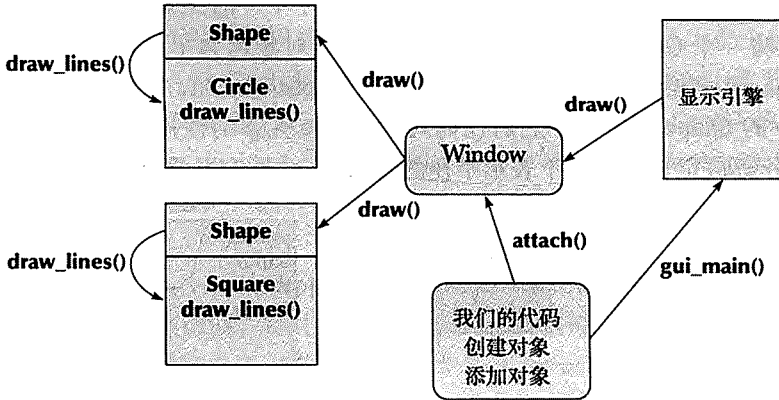
struct Circle : Shape {
    // ...
    void draw_lines() const;        // “覆盖” Shape::draw_lines()
    // ...
};

```

所以，如果某个 `Shape` 是一个 `Circle` 对象的话，调用它的 `draw_lines()` 就会以某种方式调用 `Circle` 的一个版本；同样，如果它是一个 `Rectangle` 对象的话，就会调用 `Rectangle` 的一个版本。这正是 `draw_lines()` 声明中关键字 `virtual` 的含义：如果一个派生自 `Shape` 的类定义了自己的 `draw_lines()` 函数（和 `Shape` 类的 `draw_lines()` 函数有相同的类型），那么此 `draw_lines()` 将被调用，而不是 `Shape` 类的 `draw_lines()`。第 18 章显示了这一机制是如何在 `Text`、`Circle`、`Closed_polyline` 等类中起作用的。在派生类中定义一个函数，使之可以通过基类提供的接口进行调用，这种技术称为“覆盖”（overriding）。

注意，尽管在 Shape 类中处于核心地位，draw\_lines() 还是被定义成 protected，这意味着它不能被“一般用户”调用——这是 draw() 的目的而不是 draw\_lines() 的，draw\_lines() 只是作为一个“实现细节”被 draw() 函数及 Shape 的派生类使用。

这样就完成了 17.2 节中的显示模型。驱动屏幕的系统了解 Window 类，Window 类了解 Shape 类并可以调用它的 draw() 函数。最后，draw() 函数调用特定形状类的 draw\_lines() 函数。我们代码对 gui\_main() 函数的调用会启动这个显示引擎。



gui\_main() 函数是什么？到目前为止还没有在我们的代码中实际看到过它。相反我们使用了 wait\_for\_button()，它用更单纯的方式调用显示引擎。

Shape 类的 move() 函数简单地将保存的每个点相对于当前位置移动一个偏移量：

```
void Shape::move(int dx, int dy) // 将形状移动到 +dx 和 +dy 的位置
{
    for (int i = 0; i < points.size(); ++i) {
        points[i].x += dx;
        points[i].y += dy;
    }
}
```

像 draw\_lines() 一样，move() 也是虚函数，因为派生类可能包含所要移动的数据，而 Shape 对此并不了解。例如，参考 Axis (17.7.3 节和 20.4 节)。

逻辑上，move() 函数对于 Shape 类并不是必需的，提供它只是为了方便，同时也是为了提供另一个虚函数的例子。只要形状类包含了不在 Shape 类中存储的点，就应该定义自己的 move() 函数。

#### 19.2.4 拷贝和可变性

Shape 类将拷贝构造函数和拷贝赋值运算符声明为 delete:

```
Shape(const Shape&) = delete; // 防止拷贝
Shape& operator=(const Shape&) = delete;
```

这样做的效果是禁止了默认的拷贝操例，例如：

```
void my_fct(Open_polyline& op, const Circle& c)
{
    Open_polyline op2 = op; // 错误：Shape 的拷贝构造函数被删除
    vector<Shape> v;
```

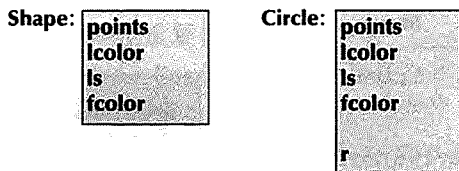


```

v.push_back(c);           // 错误: Shape 的拷贝构造函数被删除
// ...
op = op2;                // 错误: Shape 的赋值被删除
}

```

但是拷贝在很多地方都非常有用！你只要看一下 `push_back()` 函数，没用拷贝功能，我们甚至无法使用 `vector`（`push_back()` 将参数的一份拷贝放在向量中）。为什么防止拷贝会给程序员带来麻烦呢？对一个类型而言，如果默认的拷贝操作可能引起麻烦，你可以将其禁止。关于“麻烦”的一个很好的例子是 `my_fct()`，由于 `v` 中的元素“槽”大小为一个 `Shape`，所以我们不能将一个 `Circle` 对象拷贝到其中。`Circle` 对象包含半径数据而 `Shape` 对象没有，所以 `sizeof(Shape) < sizeof(Circle)`。如果允许执行 `v.push_back(c)`，则这个 `Circle` 对象将被“切断”存入 `v` 的 `Shape` 元素中，之后任何对此 `Shape` 元素的使用都可能会引起崩溃，因为对 `Circle` 的操作都假定它包含一个表示半径的成员 (`r`)，而它并没有拷贝过来：



`op2` 的拷贝构造函数和向 `op` 的赋值操作也面临着完全一样的问题。考虑如下情况：

```

Marked_polyline mp {"x"};
Circle c(p,10);
my_fct(mp,c);    // Open_polyline 参数指向一个 marked_polyline

```

现在 `Open_polyline` 的拷贝操作会将对象 `mp` 的 `string` 成员 `mark` “切掉”。

⚠ 本质上，类层次结合参数引用传递方式与默认拷贝是不能混合的。当你设计一个将要作为基类的类时，应使用 `=delete` 禁用它的拷贝构造函数和拷贝赋值操作，就像我们对 `Shape` 所做的那样。

切断（是的，这确实是一个技术术语）并不是我们禁止拷贝的唯一原因。如果没有拷贝操作的话，有很多思想可以更好地实现。回忆一下，图形系统不得不记住 `Shape` 对象的存储位置，以便将它显示在屏幕上。这就是为什么我们要将 `Shape` “添加”（`attach`）而不是拷贝到 `Window`。例如，如果 `Window` 只是拥有 `Shape` 的副本，而不是 `Shape` 的引用，对原始版本的修改不会影响到副本。所以如果我们改变了 `Shape` 的颜色，`Window` 将不会注意这种变动，还是显示没有修改的颜色。因此，拷贝一个副本在实际中并不如使用原始版本好。

✎ 如果我们希望拷贝不同类型的对象，而默认拷贝操作被禁用了，这时可以实现一个显式函数来完成这个工作。这种拷贝函数通常被称为 `clone()`。很显然，只有当成员读取函数能充分表达构造副本需要什么内容时，我们才能编写出 `clone()` 函数，而所有的 `Shape` 类恰好都是这种情况。

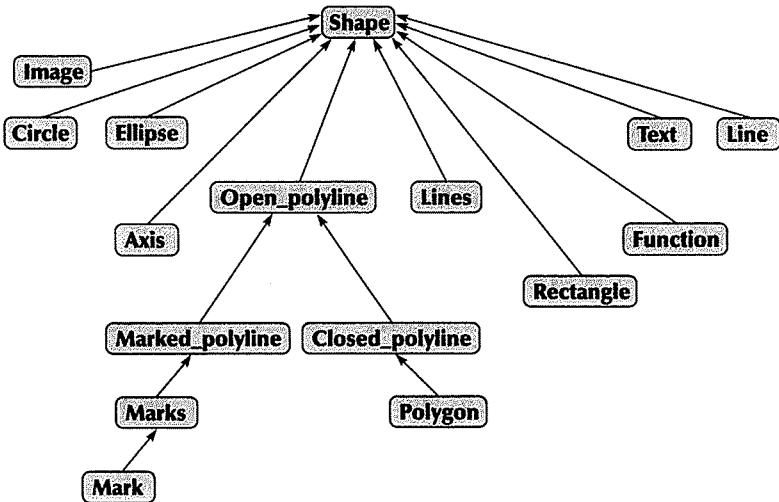
### 19.3 基类和派生类

让我们从一个更为技术性的角度来观察基类和派生类，也就是说，在本节中（只在本节）我们将讨论的焦点从程序设计、应用设计和图形转移到程序设计语言的特性上来。当设计一个图形接口库时，我们依赖于三个关键的语言机制：

- 派生 (derivation): 从一个类构造另一个类的方法, 使新构造的类可以替换原来的类。例如, Circle 类派生自 Shape 类, 或者换句话说, “Circle 是某种 Shape” 或者 “Shape 是 Circle 的基类”。派生类 (这里是 Circle) 除了自己的成员以外, 还包括基类 (这里是 Shape) 的所有成员。这通常被称为继承 (inheritance), 因为派生类 “继承” 了其基类的所有成员。在某些上下文环境中, 派生类被称为子类 (subclass), 而基类被称为超类 (Superclass)。
- 虚函数 (virtual function): 在基类中定义一个函数, 在派生类中有一个类型和名称完全一样的函数, 当用户调用基类函数时, 实际上调用的是派生类中的函数。例如, 当 Window 对 Circle (添加到 Window 的 Shape) 调用 draw\_lines() 函数时, Circle 类的 draw\_lines() 函数得到执行, 而不是 Shape 类本身的 draw\_lines() 函数。这通常被称为运行时多态 (run-time polymorphism)、动态分派 (dynamic dispatch) 或者运行时分派 (run-time dispatch), 因为具体调用哪个函数是根据运行时实际使用的对象类型来确定的。
- 私有和保护成员 (private and protected member): 我们保持类的实现细节为私有的, 以保护它们不被直接访问, 简化维护操作, 这通常被称为封装 (encapsulation)。

继承、运行时多态和封装是面向对象程序设计 (object-oriented programming) 最常用的概念。因此, 除了其他的程序设计风格之外, C++ 还直接支持面向对象程序设计。例如, 在第 15 和 16 章中, 我们将看到 C++ 如何支持泛型编程。C++ 借用了 Simula67 语言 (给予了明确的致谢) 的核心机制, Simula67 是第一个直接支持面向对象程序设计的语言 (参见第 22 章)。

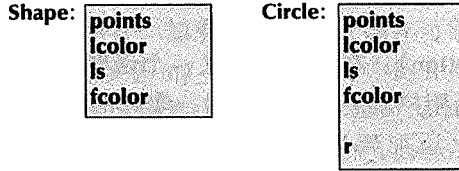
这里有很多技术术语! 但是它们代表什么意思? 同时它们在计算机中实际是如何工作的? 我们首先为图形接口类画一个简单的继承关系图:



箭头从派生类指向它的基类。这种图示可以帮助我们看到类之间的关系, 因此经常会出现在程序员的黑板上。与一个商业框架相比, 这是一个非常小的 “类层次”, 仅仅包含 16 个类, 而且只有 Open\_polyline 类的后代才会有多于一层的情况。很明显, 虽然公共基类 (Shape) 代表的是一个抽象概念, 我们永远不能直接创建其对象, 但它仍是最重要的一个类。

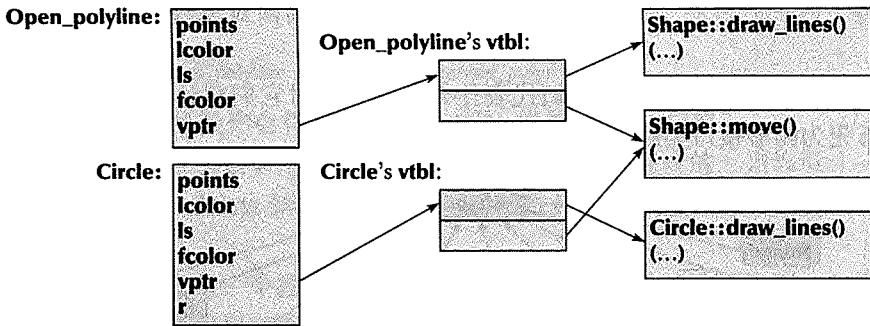
### 19.3.1 对象布局

对象在内存中是如何布局的呢？就像我们在 9.4.1 节中所看到的，一个类的成员定义了对象的布局：数据成员在内存中一个接一个地存储。当使用继承时，派生类的数据成员被简单地放在基类的成员之后。例如：



一个 Circle 对象包含 Shape 类的数据成员（毕竟它也是一种 Shape），并且可以当作 Shape 对象使用。此外，Circle 对象还有它自己的数据成员 r，存放在继承的数据成员之后。

为了处理一个虚函数调用，我们需要（并且必须）在 Shape 对象中存储更多的信息：当我们调用 Shape 的 draw\_lines() 函数时，可以借助这些信息分辨出实际应该调用哪个函数。常用的方法是增加一个函数列表的地址，这个表通常被称为 vtbl（即“virtual table”或者“virtual function table”，虚函数表），它的地址通常被称为 vptr（即“virtual pointer”，虚指针）。我们已经在第 12 ~ 13 章中讨论过指针，在这里，它们的作用类似于引用。对于 vtbl 和 vptr，一个给定的实现可能使用不同的名字。将 vptr 和 vtbl 加入布局图中可得：



因为 draw\_lines() 函数是第一个虚函数，所以它占据了 vtbl 中的第一个位置，紧接着是第二个虚函数 move()。只要你需要，一个类可以有任意多个虚函数，其 vtbl 的规模则视需要而定（一个位置对应一个虚函数）。当我们调用 x.draw\_lines() 的时候，编译器查找 x 的 vtbl 中 draw\_lines() 对应位置，调用找到的函数。本质上，代码只不过是按照图中箭头寻找对应的函数而已。因此，如果 x 是一个 Circle，Circle::draw\_lines() 将会被调用。如果 x 是另一个类型，比如 Open\_polyline，而它的 vtbl 与 Shape 类一样，则 Shape::draw\_lines() 将会被调用。类似地，由于 Circle 没有定义它自己的 move() 函数，所以如果 x 是一个 Circle，则 x.move() 将调用 Shape::move()。本质上，虚函数调用产生的目标代码首先简单地寻找 vptr，通过它找到对应的 vtbl，然后调用其中正确的函数。其代价大约是两次内存访问加上一次普通函数调用，既简单又快速。

Shape 是一个抽象类，所以我们不能实际拥有一个 Shape 对象。但是一个 Open\_polyline 对象拥有和“平凡形状”一样的布局，因为它并没有增加数据成员，也没有定义虚函数。对于每个具有虚函数的类，只有一个全局的 vtbl，而不是每个对象都有自己的 vtbl，所以 vtbl

并不会明显地增加程序目标代码的大小。

注意，在上面的布局图中，我们没有画出任何非虚函数。我们不需要那样做，因为那些函数的调用方式没有任何特别之处，所以它们不会增加对象的大小。

定义一个和基类中虚函数的名称和类型都相同的函数（比如 `Circle::draw_lines()`），以使派生类的函数代替基类中的版本被放入 vtbl 中的技术称为覆盖（overriding）。例如，`Circle::draw_lines()` 覆盖了 `Shape::draw_lines()`。

为什么我们要告诉你这些关于 vtbl 和内存布局的内容呢？为了进行面向对象程序设计，你需要了解这些内容吗？实际上并不需要，但很多人非常想知道事情是如何实现的（我们也是如此），而当人们不理解事情的时候，荒诞的说法就会产生。我们遇到过一些讨厌虚函数的人，“因为它们代价很高”。为什么？如何得出代价高的结论？和什么相比？什么情况下这些代价会产生问题？我们解释了虚函数的实现模型，你就不会再有这些恐惧了。如果你需要一个虚函数调用（在运行时选择被调用函数），你不可能使用其他语言特性编写出速度更快或者使用更少内存的代码。这一点很容易理解。

### 19.3.2 类的派生和虚函数的定义

我们通过在类名后给出一个基类来指定一个类为派生类。例如：

```
struct Circle : Shape { /* ... */};
```

默认情况下，结构（struct）的成员都是公有的（参见 9.3 节），基类中的公有成员也会成为结构的公有成员。另一种等价的定义方式如下：

```
class Circle : public Shape { public: /* ... */};
```

这两种 `Circle` 的声明是完全等价的，至于哪一种方式更好，你可能和其他人争论很长时间也没有结论。我们的意见是，不如把时间花在其他问题上，可能更有价值。

注意不要忘记了 `public` 关键字。例如：

```
class Circle : Shape { public: /* ... */}; // 可能是个错误
```

这将使 `Shape` 成为 `Circle` 的一个私有基类，`Circle` 将不能访问 `Shape` 的公有函数。这很可能不是你想要的，一个好的编译器会给出警告，提示这可能是个错误。当然也有私有基类正确使用的例子，不过那不在本书讨论范围之内。

一个虚函数必须在类的声明中被声明为 `virtual`，但是如果你把函数定义放在类外，关键字 `virtual` 就不必也不能出现在那里了。例如：

```
struct Shape {
    // ...
    virtual void draw_lines() const;
    virtual void move();
    // ...
};

virtual void Shape::draw_lines() const { /* ... */} // 错误
void Shape::move() { /* ... */} // OK
```

### 19.3.3 覆盖

当你希望覆盖一个虚函数时，必须使用与基类中完全相同的名字和类型。例如：





```

struct Circle : Shape {
    void draw_lines(int) const;    // 可能是个错误 (int 参数?)
    void drawlines() const;     // 可能是个错误 (名字拼写错误?)
    void draw_lines();           // 可能是个错误 (缺少 const?)
    // ...
};

```

这里，编译器会看到 3 个与 `Shape::draw_lines()` 无关的函数（因为它们有不同的名字或者不同的类型），这些函数没有覆盖它。一个好的编译器会给出警告，提示这些可能是错误。你不能也不必在覆盖函数中加入一些内容来保证它确实覆盖了一个基类的函数。

`draw_lines()` 是一个真实的例子，我们难以模仿其所有细节来学习覆盖技术。因此，下面给出一个纯技术性的例子来说明覆盖：

```

struct B {
    virtual void f() const { cout << "B::f "; }
    void g() const { cout << "B::g "; }    // 不是虚函数
};

struct D : B {
    void f() const { cout << "D::f "; }    // 覆盖 B::f
    void g() { cout << "D::g "; }
};

struct DD : D {
    void f() { cout << "DD::f "; }        // 不覆盖 D::f (不是 const)
    void g() const { cout << "DD::g "; }
};

```

这段代码给出了一个简单的类层次关系，仅仅包含一个虚函数 `f()`。我们可以试着使用它。特别地，我们可以试着调用 `f()` 和非虚函数 `g()`。除非要处理的对象的类型是 `B`（或者是 `B` 的派生类），否则 `g()` 并不知道类型是什么。

```

void call(const B& b)
    // D 是一种 B，所以 call() 可以接受一个 D
    // DD 是一种 D，D 是一种 B，所以 call() 可以接受一个 DD
{
    b.f();
    b.g();
}

int main()
{
    B b;
    D d;
    DD dd;
    call(b);
    call(d);
    call(dd);

    b.f();
    b.g();

    d.f();
    d.g();

    dd.f();
    dd.g();
}

```

你将得到

```
B::f B::g D::f B::g D::f B::g B::f B::g D::f D::g DD::f DD::g
```

当你理解了为什么是这样的输出结果以后，你就会清楚继承和虚函数机制了。

显然，很难去追踪哪个派生类函数覆盖了哪个基类函数。幸运的是，编译器可以帮助我们进行检查。我们可以显式地声明一个函数是覆盖函数。假设派生类函数都是想进行覆盖的，那么我们可以通过添加 `override` 关键字来表示，上面的例子就变成

```
struct B {
    virtual void f() const { cout << "B::f "; }
    void g() const { cout << "B::g "; } // 不是虚函数
};

struct D : B {
    void f() const override { cout << "D::f "; } // 覆盖 B::f
    void g() override { cout << "D::g "; } // 错误：没有可以覆盖的虚函数 B::g
};

struct DD : D {
    void f() override { cout << "DD::f "; } // 错误：不覆盖 D::f (不是 const)
    void g() const override { cout << "DD::g "; } // 错误：没有可以覆盖的虚函数 D::g
};
```

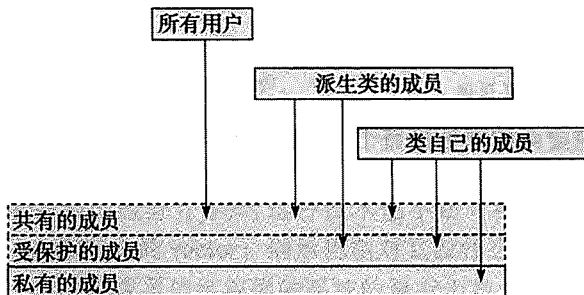
在大规模、复杂的类层次结构中显式地使用 `override` 关键字是特别有用的。

#### 19.3.4 访问

C++ 为类成员访问提供了一个简单的模型。类的成员可以是：

- 私有的 (`private`)：如果一个成员是私有的，它的名字只能被其所属类的成员使用。
- 保护的 (`protected`)：如果一个成员是保护的，它的名字只能被其所属类及其派生类的成员使用。
- 公有的 (`public`)：如果一个成员是公有的，它的名字可以被所有函数使用。

这一模型也可以图形化表示如下：



基类可以是 `private`、`protected` 或者 `public`：

- 如果类 D 的一个基类是 `private`，它的 `public` 和 `protected` 成员的名字只能被类 D 的成员使用。
- 如果类 D 的一个基类是 `protected`，它的 `public` 和 `protected` 成员的名字只能被类 D 及其派生类的成员使用。
- 如果类的一个基类是 `public`，它的名字可以被所有函数使用。

这些定义忽略了“友元”(friend)的概念和一些次要的细节,那不在本书讨论范围之内。如果你想成为语言专家,你需要学习 Stroustrup 的《The Design and Evolution of C++》《The C++ Programming language》和 ISO C++ 标准。但我们并不推荐你成为语言专家(知道语言定义的每一个微小细节),作为一个程序员(一个软件开发者、工程师、用户以及所有实际使用语言的人)乐趣更多,对社会也更有用。

### 19.3.5 纯虚函数

✂ 一个抽象类是一个只能作为基类的类。我们使用抽象类来表示那些抽象的概念,即相关实体共性的泛化所对应的那些概念。人们曾写过很厚的哲学书试图精确地定义抽象概念(abstract concept)(或抽象(abstraction),或泛化(generalization)等)。无论其哲学定义如何,抽象概念的思想是极其有用的。例如,“动物”(相对于任何特定种类的动物)、“设备驱动程序”(相对于某种特定设备的驱动程序)和“出版物”(相对于任何特定种类的书或杂志)。在程序中,抽象类通常定义了一组相关类(类层次)的接口。

✂ 在 19.2.1 节中,我们看到了如何通过声明 `protected` 构造函数来定义一个抽象类。下面是另一种更常用的方法:声明一个或者多个需要在派生类中被覆盖的虚函数。例如:

```
class B {                // 抽象基类
public:
    virtual void f()=0;   // 纯虚函数
    virtual void g()=0;
};

B b;                    // 错误: B 是抽象类
```

这个奇怪的语法“=0”指出 `B::f()` 和 `B::g()` 是“纯”虚函数,即它们必须在派生类中被覆盖。因为 `B` 有纯虚函数,所以我们不能创建一个 `B` 的对象。覆盖纯虚函数可解决这个“问题”:

```
class D1 : public B {
public:
    void f() override;
    void g() override;
};

D1 d1;                  // 正确
```

注意,除非所有纯虚函数都被覆盖了,否则该派生类也是抽象的:

```
class D2 : public B {
public:
    void f() override;
    // 没有 g()
};

D2 d2;                 // 错误: D2 仍然是抽象的

class D3 : public D2 {
public:
    void g() override;
};

D3 d3;                 // 正确
```

通常，带有纯虚函数的类的目标是提供纯粹的接口，即它们倾向于不包含任何数据成员（数据成员在派生类中定义），因此没有任何构造函数（如果没有任何数据成员需要初始化，那么就不需要构造函数）。

## 19.4 面向对象程序设计的好处

当我们说 Circle 派生自 Shape，或者 Circle 是一种 Shape 的时候，实际上获得了如下好处（其中之一或者两者皆有）：

- 接口继承（interface inheritance）：需要 Shape 对象参数（通常作为一个引用参数）的函数可以接受 Circle 对象参数（并且可以通过 Shape 提供的接口使用 Circle）。
- 实现继承（implementation inheritance）：当我们定义 Circle 及其成员函数时，我们可以使用 Shape 提供的功能（如数据成员和成员函数）。

一个不能提供接口继承的设计（即一个派生类的对象不能当作其公有基类的对象使用）是一个拙劣且易出错的设计。例如，我们可能定义一个以 Shape 为公有基类的类 Never\_do\_this，然后定义函数覆盖 Shape::draw\_lines()，它不绘制任何形状，相反，将自己的中心向左移动 100 个像素。这个“设计”是有致命缺陷的，因为尽管 Never\_do\_this 提供了一个 Shape 的接口，但其实现没有保持 Shape 所要求的语义（意义或行为）。永远不要这样做！

接口继承之所以得名，是因为其优点：代码使用基类（“接口”，这里是 Shape）提供的接口，而无须知道具体的派生类（“实现”，这里是 Shape 的派生类）。

实现继承之所以得名，是因为其优点：通过使用基类（这里是 Shape）提供的功能，简化了派生类（例如 Circle）的实现。

注意，我们的图形库设计严重依赖接口继承：“图形引擎”调用 Shape::draw()，接着 Shape::draw() 将调用 Shape 的虚函数 draw\_lines() 完成实际的图形显示工作。无论“图形引擎”还是实际 Shape 类都不知道有哪些具体形状。特别地，我们的“图形引擎”（FLTK 加上操作系统的图形功能）是在我们的图形类几年之前就编写、编译好的！我们只是定义了特定的形状并且将它们当作 Shape 对象添加到了 Window 中（Window::attach() 接受一个 Shape& 类型的参数，参见附录 E.3）。而且，由于 Shape 类不知道你的图形类，当你每次定义一个新的图形接口类时，不需要重新编译 Shape 类。

换句话说，我们可以向程序中加入新形状，而不用修改已有的代码。这是一个软件设计 / 开发 / 维护的圣杯：扩展一个系统而不用修改它。哪些改进不必修改已有类还是有一定限制的（如 Shape 提供了非常有限的服务），同时这种技术也不是对所有的程序设计问题都能很好地应用（例如第 12 ~ 14 章定义的 vector，继承机制对其没什么用处）。然而无论如何，接口继承是设计和实现对于改进需求鲁棒性很强的系统的最有力技术之一。

同样，实现继承也能带来很多好处，但是世界上没有万能灵药。通过在 Shape 中放入有用的服务，我们避免了在派生类中一遍又一遍进行重复性工作的烦恼。这对现实世界中的程序设计尤为重要。然而，它带来了一个额外代价，任何对于 Shape 接口或者对于 Shape 数据成员布局的更改都必须重新编译所有的派生类及其用户代码。对于一个广泛使用的库来说，这种重新编译是绝对行不通的。当然，有一些方法可以在得到大多数好处的同时避免大多数的问题，参见 19.3.5 节。

## 简单练习

不幸的是，我们无法构造一个能帮助理解一般设计原则的简单练习，所以在本练习中我们把注意力集中在支持面向对象程序设计的语言特性上。

1. 定义带有一个虚函数 `vf()` 和一个非虚函数 `f()` 的类 `B1`。在 `B1` 内定义这两个函数，实现这两个函数使它们都输出自己的名字（例如 “`B1::vf()`”）。将这两个函数定义为公有的。建立一个 `B1` 对象并且调用每个函数。
2. 从 `B1` 类派生一个 `D1` 类，并且覆盖 `vf()`。建立一个 `D1` 对象，并调用 `vf()` 和 `f()`。
3. 定义一个 `B1` 的引用 (`B1&`) 并且初始化为一个 `D1` 对象，并调用 `vf()` 和 `f()`。
4. 为 `D1` 定义一个 `f()` 函数，重复 1 ~ 3 小题，并解释其结果。
5. 在 `B1` 中定义一个纯虚函数 `pvf()`，重复 1 ~ 4 小题，并解释其结果。
6. 定义一个派生自 `D1` 的 `D2` 类，并且在 `D2` 中覆盖 `pvf()`。建立一个 `D2` 类的对象并且调用 `f()`、`vf()`、`pvf()` 函数。
7. 定义带有一个纯虚函数 `pvf()` 的 `B2` 类。定义 `D21` 类，使其包含一个 `string` 数据成员和一个覆盖 `pvf()` 的成员函数，`D21::pvf()` 输出 `string` 数据成员的值。定义 `D22` 类，它与 `D21` 类一样，只是数据成员为 `int` 类型。定义函数 `f()`，接受一个 `B2&` 参数，并对此参数调用 `pvf()` 函数。使用 `D21` 对象和 `D22` 对象调用 `f()`。

## 思考题

1. 什么是应用领域？
2. 什么是理想的命名？
3. 我们可以命名哪些东西？
4. `Shape` 类提供了哪些功能？
5. 如何区别抽象类和非抽象类？
6. 如何将类设计为抽象类？
7. 访问控制能够控制什么？
8. 私有 (`private`) 数据成员有什么好处？
9. 虚函数是什么？如何区别于一个非虚函数？
10. 什么是基类？
11. 如何定义一个派生类？
12. 对象的布局意味着什么？
13. 使一个类更易于测试，应该做哪些工作？
14. 继承关系图是什么？
15. 保护 (`protected`) 对象和私有 (`private`) 对象有什么区别？
16. 类中的哪些成员可以被它的派生类访问？
17. 如何区别纯虚函数和其他虚函数？
18. 为什么将一个成员函数设计为虚函数？
19. 为什么将一个虚函数设计为纯虚函数？
20. 覆盖的含义是什么？
21. 接口继承和实现继承有什么区别？

## 22. 什么是面向对象程序设计?

### 术语

abstract class (抽象类)	polymorphism (多态)
access control (访问控制)	private (私有)
base class (基类)	protected (保护)
derived class (派生类)	public (公有)
dispatch (分派)	pure virtual function (纯虚函数)
encapsulation (封装)	subclass (子类)
inheritance (继承)	superclass (超类)
mutability (可变性)	virtual function (虚函数)
object layout (对象布局)	virtual function call (虚函数调用)
object-oriented override (面向对象覆盖)	virtual function table (虚函数表)

### 习题

1. 定义两个类 **Smiley** 和 **Frowny**，它们都派生自 **Circle** 类，并且都有两只眼睛和一张嘴。接下来，分别从 **Smiley** 和 **Frowny** 类派生类，为其添加一个适当的帽子。
2. 尝试拷贝一个 **Shape** 对象，会发生什么？
3. 定义一个抽象类并且尝试定义一个该类型的对象，会发生什么？
4. 定义一个类似于 **Circle** 的 **Immobile\_Circle** 类，只是它不能移动。
5. 定义 **Striped\_rectangle** 类，不采用标准的填充方式，而是用一个像素宽的水平线“填充”该矩形的内部（比如每隔一个像素画一条线）。你可能需要通过设置线宽和线间距来获得喜欢的图案。
6. 使用 **Striped\_rectangle** 中的技术定义 **Striped\_circle** 类。
7. 使用 **Striped\_rectangle** 中的技术定义 **Striped\_closed\_polyline** 类（需要一些算法上的创新）。
8. 定义 **Octagon** 类，表示正八边形。编写测试程序，测试它的所有成员函数（包括你自己定义的和继承自 **Shape** 类的）。
9. 定义 **Group** 类，表示 **Shape** 的容器，为其设计适合的操作，能恰当处理类的不同成员。提示：使用 **Vector\_ref**。利用 **Group** 定义一个国际跳棋棋盘，棋子可以在程序的控制下移动。
10. 定义一个非常像 **Window** 的 **Pseudo\_window** 类（尽你所能，但不必花费太大精力）。它应该是圆角的，应带有标签和控制图标。也许你可以添加一些假的“内容”，如一幅图像。它不必做任何实质性工作。一种可接受的方法（实际上我们建议这样做）是将其显示在一个 **Simple\_window** 中。
11. 定义一个 **Binary\_tree** 类，它派生自 **Shape** 类。层数作为一个参数（**levels==0** 表示没有节点，**levels==1** 表示有一个节点，**levels==2** 表示有一个顶层节点和两个子节点，**levels==3** 表示有一个顶层节点、两个子节点以及这两个子节点的各自两个子节点，依此类推）。使用小圆圈表示一个节点，并用线连接这些节点。注：在计算机科学中，树是从一个顶层节点（有趣且合乎逻辑的是它经常被称为根）向下生长的。
12. 修改 **Binary\_tree**，使用虚函数来绘制它的节点。然后，从 **Binary\_tree** 派生一个新类，对节点使用一个不同的表示（比如一个三角形）来覆盖此虚函数。

13. 修改 `Binary_tree`，使其接受一个参数（或者多个）来指出用什么类型的线连接这些节点（例如一个向下箭头或者一个红色的向下箭头）。注意，本题和前一题是如何使用两种不同的方式使得类的层次结构更加灵活和有用的。
14. 为 `Binary_tree` 类增加一个操作，将文本添加到节点上。你可能必须修改 `Binary_tree` 的设计来实现这个功能。选择一种方式来标识节点，例如，你可以用字符串“`lrrlr`”表示向下遍历二叉树的左、右、右、左和右会到达当前节点（以 `l` 或者 `r` 开头都可与根节点匹配）。
15. 大多类层次是与图形无关的。定义 `Iterator` 类，它包含一个返回值为 `double*` 类型的纯虚函数 `next()`（参见第 12 章）。基于 `Iterator` 类派生 `Vector_iterator` 和 `List_iterator` 类，使 `Vector_iterator` 的 `next()` 函数生成指向 `vector<double>` 中下一个元素的指针，而 `List_iterator` 对于 `list<double>` 类型实现相同的操作。`Vector_iterator` 对象通过一个 `vector<double>` 初始化，对于 `next()` 的第一次调用应得到指向第一个元素的指针（如果向量不空的话）。如果没有下一个元素的话，`next()` 应返回 0。编写函数 `void print(Iterator&)`，打印 `vector<double>` 和 `list<double>` 中的元素，从而实现测试。
16. 定义 `Controller` 类，它包含 4 个虚函数 `on()`、`off()`、`set_level(int)` 和 `show()`。至少从 `Controller` 派生出两个类，第一个派生类是一个简单的测试类，它的 `show()` 函数打印出这个类是设置为开还是关，以及当前的级别；第二个派生类需要以某种方法控制一个 `Shape` 对象的线颜色，“级别”的确切含义由你自己决定。试着找到 `Controller` 类可以控制的第三种“东西”。
17. 在 C++ 标准库中定义的异常，例如 `exception`、`runtime_exception` 和 `out_of_range`（参见 5.6.3 节），被组织为一个类层次（使用一个很有用的虚函数 `what()`，它返回一个字符串来解释发生了什么错误）。查找 C++ 标准异常类的层次结构，绘制它的类层次图。

## 附言

软件设计的理想不是构造一个可以做任何事情的程序，而是构造很多类，这些类可以准确反映我们的思想，可以组合在一起工作，允许我们用来构造漂亮的应用程序，并且具有最小的工作量（相对于任务的复杂度而言）、足够高的性能以及保证产生正确的结果等优点。这样的程序易于理解、易于维护，而简单地将一些代码快速拼凑在一起来完成某个特定工作则不可能具有这样的优点。类、封装（由 `private` 和 `protected` 支持）、继承（由类的派生支持）和运行时多态（由虚函数支持）都是我们构建这样系统的最有力的工具。

# 绘制函数图和数据图

至善者善之敌。

——伏尔泰

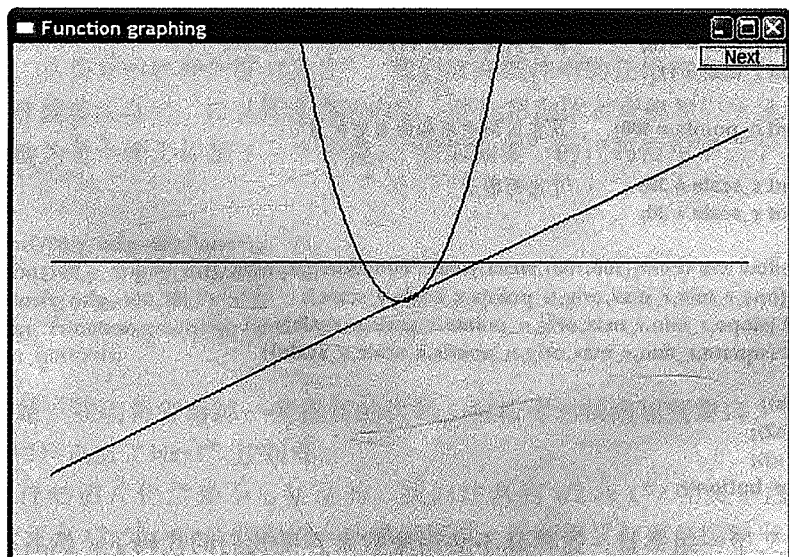
如果是从事实验领域，你需要用图表示数据；如果是从事自然现象的数学建模领域，你需要用图表示函数。本章将讨论这类图形的基本机制。跟往常一样，我们将介绍这些机制的使用方法以及它们的设计理念。本章给出的关键实例是绘制一个带有单参数的函数图，及显示从文件中读取的值。

## 20.1 简介

与可视化领域的专业软件系统相比，本章介绍的工具比较原始。我们的主要目标不是输出的美观性，而是理解如何生成这样的图形输出以及其中所使用的编程技术。你会发现，本章使用的设计方法、编程技术和基本数学工具比提出的图形工具有更长久的价值。因此，请不要快速掠过那些代码段，代码比它们计算和绘制出来的形状更重要。

## 20.2 绘制简单函数图

让我们开始吧。首先看一些例子，它们展示了我们能绘制什么图形以及使用什么样的代码来绘制。特别地，请仔细阅读所使用的图形接口类。此处，我们首先绘制了一条抛物线、一条水平线和一条斜线。





实际上，因为本章介绍函数的图形化，所以这条水平线并不仅仅是一条水平线，它是我们将下面的函数图形化得到的。

```
double one(double) { return 1; }
```

这大概是我们能想到的最简单的函数：该函数只有一个参数，而且对任何参数都返回 1。因为我们不需要用这个参数来计算结果，所以不需要为它命名。对于每一个传递给 one() 函数的参数  $x$ ，我们得到  $y$  的值都是 1；即，对所有的  $x$  而言，这条线由  $(x, y) == (x, 1)$  定义。

像所有初级的数学参数一样，本例有些过于简单和学究气，所以让我们看一个稍微复杂些的函数：

```
double slope(double x) { return x/2; }
```

这是生成斜线的函数。对每一个  $x$ ，我们得到的  $y$  值为  $x/2$ 。换句话说， $(x, y) == (x, x/2)$ 。两条线的交点是  $(2, 1)$ 。

现在我们可以试验一些更有趣的函数，平方函数似乎是有规律地在本书中重复出现：

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

如果你还记得中学的几何课程（即使你不记得了也不要紧），这个函数定义了一个最低点在  $(0, 0)$  且以  $y$  轴对称的抛物线。换句话说， $(x, y) == (x, x*x)$ 。所以，抛物线的最低点和斜线相交于点  $(0, 0)$ 。

下面是绘制这三个函数的代码：

```
constexpr int xmax = 600;           // 窗口大小
constexpr int ymax = 400;

constexpr int x_orig = xmax/2;      // (0, 0) 坐标位置是窗口的中心
constexpr int y_orig = ymax/2;
constexpr Point orig (x_orig,y_orig);

constexpr int r_min = -10;          // 区间为 [-10: 11]
constexpr int r_max = 11;

constexpr int n_points = 400;      // 在区间中用到的点的数量

constexpr int x_scale = 30;         // 比例因子
constexpr int y_scale = 30;

Simple_window win (Point{100,100},xmax,ymax,"Function graphing");
Function s {one,r_min,r_max,orig,n_points,x_scale,y_scale};
Function s2 {slope,r_min,r_max,orig,n_points,x_scale,y_scale};
Function s3 {square,r_min,r_max,orig,n_points,x_scale,y_scale};

win.attach(s);
win.attach(s2);
win.attach(s3);
win.wait_for_button();
```


首先，我们定义了一组常量，这样就不必用魔数来弄乱我们的代码了。然后，我们创建了一个窗口，定义这些函数，将它们添加到窗口上，最后将控制权交给图形系统进行实际的

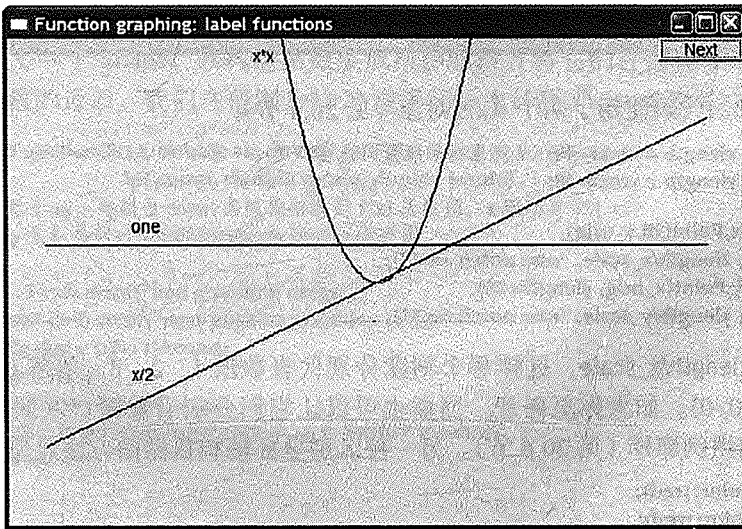
绘制。


除了 `s`、`s2` 和 `s3` 这三个 Function 的定义外，其余的都是重复性的和“样板”代码：

```
Function s {one,r_min,r_max,orig,n_points,x_scale,y_scale};
Function s2 {slope,r_min,r_max,orig,n_points,x_scale,y_scale};
Function s3 {square,r_min,r_max,orig,n_points,x_scale,y_scale};
```

每个 Function 都指出了如何在窗口中绘制其第一个参数（接受一个 `double` 类型参数并返回一个 `double` 类型值的函数），第二个和第三个参数给出 `x` 的取值范围（传递给要绘制的函数的参数），第四个参数（此处是 `orig`）告知 Function 原点  $(0,0)$  在窗口中的位置。


如果你认为参数过多，容易混淆，我们同意这一点。我们的理想方案是使用尽可能少的  参数，因为参数过多会造成混淆且容易出错。但是，本例确实需要这么多参数。我们将在稍后（20.3 节）解释后三个参数。无论如何，我们先为图形添加标签：

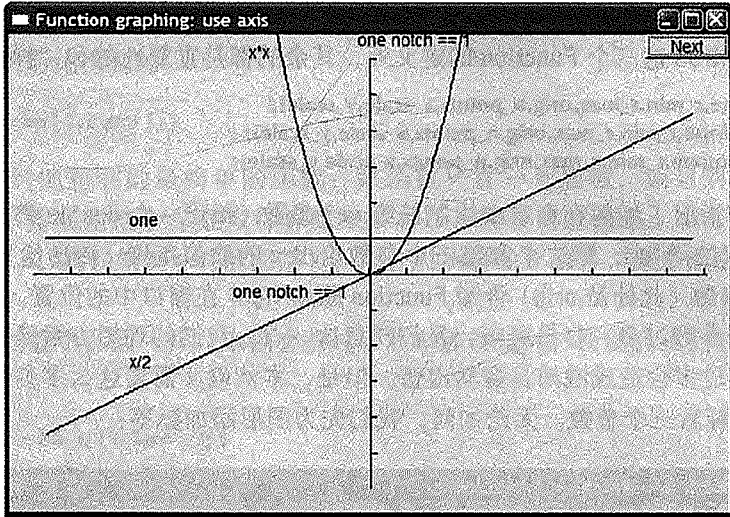


我们总是尝试使图形能自我解释。人们不会总是去阅读图形周围的解释文字，而且图像  可能会被移动，从而导致解释文字“丢失”。我们放在图片中的任何内容都会成为图片的一部分，更容易被读者注意到。如果是合理的内容，还能帮助读者理解我们所显示的图形。此处，我们简单地每个图形添加了一个标签。“添加标签”的代码使用了三个 `Text` 对象（见 18.11 节）：

```
Text ts {Point{100,y_orig-40},"one"};
Text ts2 {Point{100,y_orig+y_orig/2-20},"x/2"};
Text ts3 {Point{x_orig-100,20},"x*x"};
win.set_label("Function graphing: label functions");
win.wait_for_button();
```

从现在开始，我们将省略掉一些重复的代码，包括将形状添加到窗口，为窗口添加标签，以及等待用户点击“Next”按钮等。

但是，这样的图片仍然是不可接受的。我们注意到 `x/2` 与 `x*x` 相交于点  $(0, 0)$ ，同时 `one` 与 `x/2` 相交于点  $(2, 1)$ ，但这些现象有些难以察觉，我们需要使用坐标轴来清楚地展现给  读者：



实现坐标轴的代码使用了两个 `Axis` 对象（见 20.4 节）：

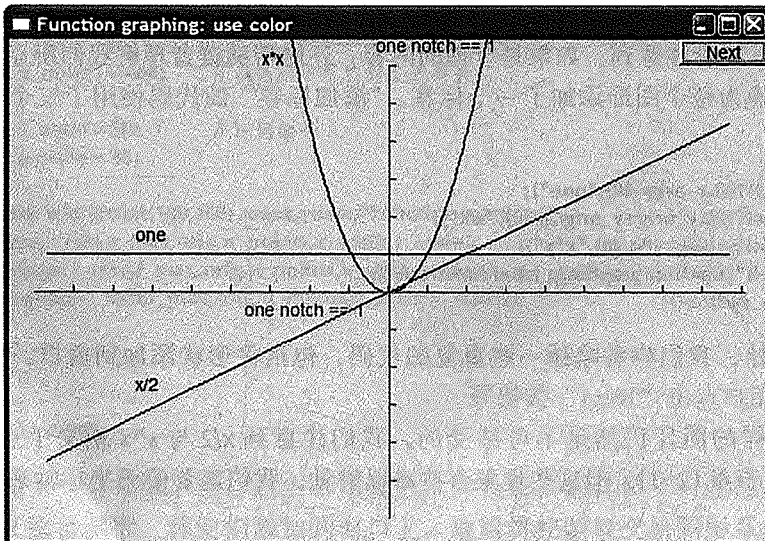
```
constexpr int xlength = xmax-40; // 把坐标轴设置得比窗口略小一些
constexpr int ylength = ymax-40;
```

```
Axis x {Axis::x,Point{20,y_orig},
        xlength,xlength/x_scale,"one notch == 1"};
Axis y {Axis::y,Point{x_orig,ylength+20},
        ylength,ylength/y_scale,"one notch == 1"};
```

刻度的数量为  $xlength/x\_scale$ ，这样每个刻度分别代表数值 1、2、3，等等。按照惯例，坐标轴相交于点  $(0, 0)$ 。如果你更愿意，当然也可以让它们分别沿着窗口左侧和底部的边缘，就像显示数据的惯例那样（见 20.6 节）。另一种区别坐标轴和数据的方法是使用不同颜色：

```
x.set_color(Color::red);
y.set_color(Color::red);
```

于是可得：



这是一个可以接受的输出结果了，虽然出于美学的原因，我们可能想要在顶端留出一些空白以便和底部和两边对称。将  $x$  轴的标签放到更远的左侧也是一个更好的想法。我们留下这些缺陷，这样就可以时常提及它们——总是会有很多美学细节需要我们继续完善。程序设计艺术的一个重要部分就是知道什么时候停止，将节省出的时间用于更有意义的事情上（比如学习新的技术或者睡觉）。记住：“至善者善之敌。”

## 20.3 Function

Function 图形接口类的定义如下：

```
struct Function : Shape {
    // 不存储函数参数
    Function(Fct f, double r1, double r2, Point orig,
            int count = 100, double xscale = 25, double yscale = 25);
};
```

Function 是一个 Shape，其构造函数生成很多线段并将它们存储在 Shape 中。这些线段是对函数  $f$  的值的近似。我们在范围  $[r1:r2)$  内等间隔地计算了  $\text{count}$  次  $f$  的值：

```
Function::Function(Fct f, double r1, double r2, Point xy,
                  int count, double xscale, double yscale)
// 将 (0,0) 置于 xy，绘制由 count 条线段组成的 f(x) 函数图，x 取值为 [r1, r2)
// x 坐标和 y 坐标分别由 xscale 和 yscale 进行比例设定
{
    if (r2-r1<=0) error("bad graphing range");
    if (count <=0) error("non-positive graphing count");
    double dist = (r2-r1)/count;
    double r = r1;
    for (int i = 0; i<count; ++i) {
        add(Point{xy.x+int(r*xscale),xy.y-int(f(r)*yscale)});
        r += dist;
    }
}
```

$\text{xscale}$  和  $\text{yscale}$  的值被分别用于设定  $x$  坐标和  $y$  坐标的比例。我们需要设置待显示的数值的比例，使其能适合于窗口的绘制区域。

注意，Function 对象并不存储传递给它的构造函数的值，因此，我们随后无法查询函数的原点，以及使用不同比例重新绘制函数，等等。它实现的功能就是（在它的 Shape 中）存储点并将自身绘制到屏幕上。如果我们需要在构造之后还能灵活地改变 Function，就必须存储那些我们希望修改的值（参见习题 2）。

我们用于表示一个函数参数的 Fct 是什么类型？它是一个叫作 `std::function` 的标准库类型的变量，这个标准库可以“记下”后面要调用的函数。Fct 要求的参数是 `double` 类型，并且返回一个 `double` 类型值。

### 20.3.1 默认参数

注意我们赋予 Function 构造函数参数  $\text{xscale}$  和  $\text{yscale}$  初始值的方法。这种初始化值称为默认参数（default argument），如果调用者没有给出参数值，将使用此默认值。例如：

```
Function s(one, r_min, r_max,orig, n_points, x_scale, y_scale);
Function s2(slope, r_min, r_max, orig, n_points, x_scale); // 没有 yscale
```

```
Function s3 {square, r_min, r_max, orig, n_points}; // 没有 xscale, 没有 yscale
Function s4 {sqrt, r_min, r_max, orig}; // 没有 count, 没有 xscale, 没有 yscale
```

上面的代码等价于:

```
Function s {one, r_min, r_max, orig, n_points, x_scale, y_scale};
Function s2 {slope, r_min, r_max, orig, n_points, x_scale, 25};
Function s3 {square, r_min, r_max, orig, n_points, 25, 25};
Function s4 {sqrt, r_min, r_max, orig, 100, 25, 25};
```

另一种替代方法是提供几个重载函数。我们可以定义四个构造函数，而不是定义一个有三个默认参数的构造函数:

```
struct Function : Shape { // 替代方法, 不使用默认参数
    Function(Fct f, double r1, double r2, Point orig,
            int count, double xscale, double yscale);
    // y 的默认比例
    Function(Fct f, double r1, double r2, Point orig,
            int count, double xscale);
    // x 和 y 的默认比例
    Function(Fct f, double r1, double r2, Point orig, int count);
    // count 的默认值以及 x 和 y 的默认比例
    Function(Fct f, double r1, double r2, Point orig);
};
```



定义四个构造函数的工作量更多一些，同时对于这个含有四个构造函数的版本，默认参数的特性仍然存在，只不过它隐藏在构造函数的定义中，而不是在声明中显式地给出。默认参数经常被用于构造函数中，但是它对其他类型的函数也适用。注意：只能将末尾的参数定义为默认参数。例如：

```
struct Function : Shape {
    Function(Fct f, double r1, double r2, Point orig,
            int count = 100, double xscale, double yscale); // 错误
};
```

如果一个参数有一个默认参数值，那么其后的所有参数都必须有一个默认参数值：

```
struct Function : Shape {
    Function(Fct f, double r1, double r2, Point orig,
            int count = 100, double xscale=25, double yscale=25);
};
```

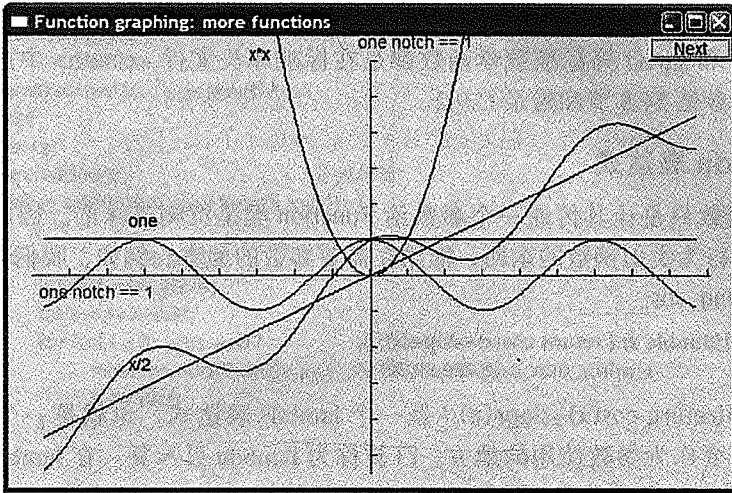
有时候，选择好的默认参数值比较容易。这样的例子包括字符串的默认值（空字符串）和 `vector` 的默认值（空 `vector`）。对于其他情况，如 `Function`，选择一个默认值却不是那么简单；我们通过一些实验和一次失败的尝试之后，找到了现在所使用的这组默认值。记住，你不是必须要提供默认参数，而且如果你发现很难给出一个默认值，简单地将它留给用户来指定即可。

### 20.3.2 更多例子

我们增加了一对函数：一个简单的余弦函数（`cos`），它来自标准库，以及一个用来说明如何组合函数的例子——沿着斜率  $x/2$  的倾斜余弦函数：

```
double sloping_cos(double x) { return cos(x)+slope(x); }
```

结果如图所示：



对应的代码为：

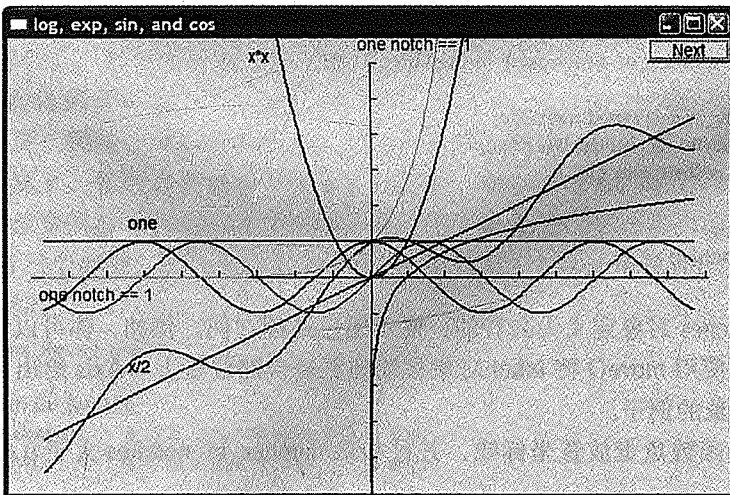
```
Function s4 {cos,r_min,r_max,orig,400,30,30};
s4.set_color(Color::blue);
Function s5 {sloping_cos, r_min,r_max,orig,400,30,30};
x.label.move(-160,0);
x.notches.set_color(Color::dark_red);
```

除了增加这两个函数外，我们还移动了  $x$  轴的标签并稍微改变了它的刻度颜色（只是为了说明如何实现）。

最后，我们图形显示一个对数函数、一个指数函数、一个正弦函数和一个余弦函数：

```
Function f1 {log,0.000001,r_max,orig,200,30,30}; // log() 算法，底数为 e
Function f2 {sin,r_min,r_max,orig,200,30,30}; // sin()
f2.set_color(Color::blue);
Function f3 {cos,r_min,r_max,orig,200,30,30}; // cos()
Function f4 {exp,r_min,r_max,orig,200,30,30}; // exp(), 指数函数 e^x
```

因为  $\log(0)$  是没有定义的（数学上是负无穷大），所以  $\log$  的范围从一个小的正数开始。得到的结果如下：



本例中我们用不同颜色区分这些函数而不是为它们添加标签。

`cos()`、`sin()` 和 `sqrt()` 等标准数学函数都是在标准库头文件 `<cmath>` 中声明的。标准数学函数的列表请参见 24.8 节和附录 C.9.2。

### 20.3.3 lambda 表达式

定义一个函数只是让其传递一个参数给 `Function` 类是沉闷乏味的。因此，C++ 提供了一种方法来定义行为类似函数的东西，能用在需要参数的地方。例如，我们可以按照下面的方法来定义 `sloping_cos`。

```
Function s5 {[](double x) { return cos(x)+slope(x); },
            r_min,r_max,orig,400,30,30);
```

`[](double x){return cos(x)+slope(x);}` 是一个 lambda 表达式，也就是一个没有命名的函数，可定义于需要作为参数使用的地方。`[]` 被称为 lambda 引入符。在 lambda 引入符之后，lambda 表达式指定了所需参数（参数列表）和需要执行的动作（函数体）。返回类型可以由 lambda 表达式内容得出。这里，从 `cos(x)+slope(x)` 的类型可以推断出返回类型是 `double`。如果我们想要明确指出返回类型，可以用下面的方法：

```
Function s5 {[](double x) -> double { return cos(x)+slope(x); },
            r_min,r_max,orig,400,30,30);
```

很少需要为一个 lambda 表达式指定其返回类型。主要原因是保持 lambda 表达式的简单，以免成为一系列错误或混乱的源头。如果一段代码确实非常重要，那么它应该被命名，还可能需要注释，使编写程序的本人之外的其他人也易于理解。我们推荐使用命名函数来编写不适合在一两行内完成的代码。

lambda 表达式可以使用 lambda 引入符来访问局部变量，参见 20.5 节与 16.4.3 节。

## 20.4 Axis

当我们显示数据时，就需要使用 `Axis`（如 20.6.4 节），因为一个没有比例信息的图形常常令人怀疑。一个 `Axis` 由一条线、在这条线上的一系列“刻度”和一个文本标签组成。`Axis` 的构造函数计算坐标线和（可选的）这条线上作为刻度的一些线：

```
struct Axis : Shape {
    enum Orientation { x, y, z };
    Axis(Orientation d, Point xy, int length,
        int number_of_notches=0, string label = "");

    void draw_lines() const override;
    void move(int dx, int dy) override;
    void set_color(Color c);

    Text label;
    Lines notches;
};
```

其中 `label` 和 `notches` 对象定义为公有的，便于用户处理它们。例如，你可以为刻度设置一个不同的颜色或者使用 `move()` 将 `label` 对象移动到更合适的位置上。`Axis` 给出了一个由若干半独立对象组合对象的例子。

`Axis` 的构造函数负责放置坐标线，并且如果 `number_of_notches` 大于 0 的话，它还负责添加“刻度”。

```

Axis::Axis(Orientation d, Point xy, int length, int n, string lab)
:label(Point(0,0),lab)
{
    if (length<0) error("bad axis length");
    switch (d){
    case Axis::x:
    { Shape::add(xy);           // 坐标线
      Shape::add(Point(xy.x+length,xy.y));

      if (0<n) {           // 添加刻度
          int dist = length/n;
          int x = xy.x+dist;
          for (int i = 0; i<n; ++i) {
              notches.add(Point(x,xy.y),Point(x,xy.y-5));
              x += dist;
          }
      }

      label.move(length/3,xy.y+20); // 在线的下方放置标签
      break;
    }
    case Axis::y:
    { Shape::add(xy);           // 生成一个 y 轴
      Shape::add(Point(xy.x,xy.y-length));

      if (0<n) {           // 添加刻度
          int dist = length/n;
          int y = xy.y-dist;
          for (int i = 0; i<n; ++i) {
              notches.add(Point(xy.x,y),Point(xy.x+5,y));
              y -= dist;
          }
      }
      label.move(xy.x-10,xy.y-length-10); // 在顶部放置标签
      break;
    }
    case Axis::z:
      error("z axis not implemented");
    }
}

```

与很多实际的代码相比，这个构造函数非常简单，但是请仔细阅读，因为它并不是十分简单，并且还展示了一些有用的技术。注意我们如何将线存储在 `Axis` 的 `Shape` 部分（使用 `Shape::add()`），而将刻度存储在一个独立的对象（`notches`）中。通过这种方式，我们可以独立地处理线和刻度；例如，我们可以将它们设置为不同的颜色。类似地，标签放置在相对于坐标轴的固定位置上，但因为它也是一个独立的对象，我们总是可以将它移动到一个更好的位置。我们使用枚举类型 `Orientation` 来为用户提供一个方便的并且不易出错的符号。

因为 `Axis` 有三个部分，所以当我们希望把 `Axis` 作为一个整体来操作的时候，必须提供相应的函数。例如：

```

void Axis::draw_lines() const
{
    Shape::draw_lines();
    notches.draw(); // 刻度可能和线具有不同的颜色
    label.draw(); // 标签可能和线具有不同的颜色
}

```



我们使用 `draw()` 而不是 `draw_lines()` 函数来绘制 `notches` 和 `label`，这样我们可以使用它们各自的颜色。线被存储在 `Axis::Shape` 中，并且使用存储在相同位置的颜色。

我们可以分别为线、刻度和标签设置颜色，但是从风格上来说，一般最好不要这样做，因此我们提供了一个函数将这三部分设置为相同的颜色：

```
void Axis::set_color(Color c)
{
    Shape::set_color(c);
    notches.set_color(c);
    label.set_color(c);
}
```

类似地，`Axis::move()` 同时移动 `Axis` 的所有部分：

```
void Axis::move(int dx, int dy)
{
    Shape::move(dx,dy);
    notches.move(dx,dy);
    label.move(dx,dy);
}
```

## 20.5 近似

本节中我们给出图形化函数的另一个例子：“动态地展示”一个指数函数的计算过程。其目的是帮助你获得数学函数的感性认识（如果你还没有），说明使用图形来显示计算的方式，给出一些供你阅读的代码，最后对计算中的常见问题给出警告。

计算指数函数的一种方法是计算如下级数：

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \dots$$

这个级数的项越多，得到的  $e^x$  的值就越精确；也就是说，我们计算的项越多，得到的结果就有更多的正确位数。我们要做的就是计算这个级数，同时每计算一项就图形显示其结果。这里的感叹号代表其通常的数学意义阶乘；也就是说，我们要按顺序图形显示如下函数：

```
exp0(x) = 0           // 没有任何项
exp1(x) = 1           // 一项
exp2(x) = 1+x         // 两项；pow(x,1)/fac(1)==x
exp3(x) = 1+x+pow(x,2)/fac(2)
exp4(x) = 1+x+pow(x,2)/fac(2)+pow(x,3)/fac(3)
exp5(x) = 1+x+pow(x,2)/fac(2)+pow(x,3)/fac(3)+pow(x,4)/fac(4)
...
```

每个函数都比前一个更接近于  $e^x$ 。此处，`pow(x, n)` 是返回  $x^n$  值的标准库函数。标准库中没有阶乘函数，所以我们必须自己定义：

```
int fac(int n)        // factorial(n); n!
{
    int r = 1;
    while (n>1) {
        r*=n;
        --n;
    }
    return r;
}
```

`fac()` 的另一种实现方法参见习题 1。给定 `fac()`，我们就可以按如下方式计算出第  $n$  项：

```
double term(double x, int n) { return pow(x,n)/fac(n); } // 级数的第 n 项
```

给定 `term()`，计算指数函数的  $n$  项级数精度就很简单了：

```
double expe(double x, int n)    // 对 x 进行 n 项求和
{
    double sum = 0;
    for (int i=0; i<n; ++i) sum+=term(x,i);
    return sum;
}
```

让我们使用这个函数来生成一些图形。首先，我们提供一些坐标轴和“实际”指数函数——标准库函数 `exp()`，这样就可以看到我们使用 `expe()` 得到的近似值与真实值的接近程度。

```
Function real_exp {exp,r_min,r_max,orig,200,x_scale,y_scale};
real_exp.set_color(Color::blue);
```

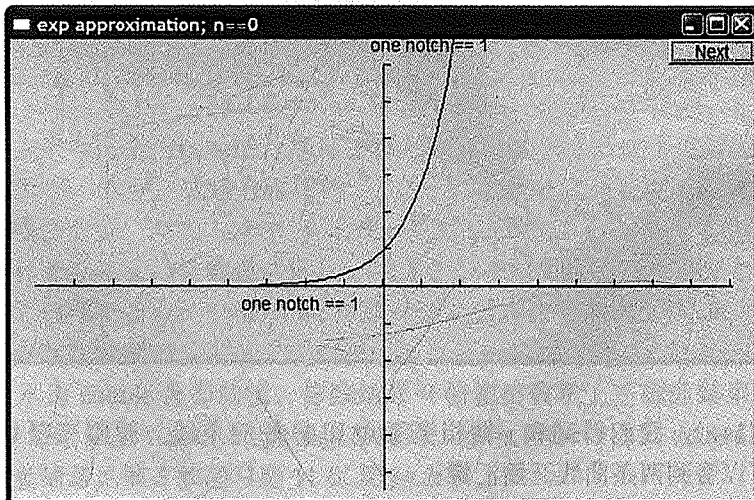
我们如何使用 `expe()` 呢？从程序设计的角度看，难点在于我们的图形类 `Function` 使用的是带有一个参数的函数，而 `expe()` 有两个参数。在 C++ 中，到目前为止还没有此问题的完美解决方法。不过，`lambda` 表达式提供了一种解决方法（见 20.3.3 节）。考虑下面的代码：

```
for (int n = 0; n<50; ++n) {
    ostream ss;
    ss << "exp approximation; n==" << n ;
    win.set_label(ss.str());
    // 得到下一个近似值
    Function e {[n](double x) { return expe(x,n); }},
               r_min,r_max,orig,200,x_scale,y_scale);
    win.attach(e);
    win.wait_for_button();
    win.detach(e);
}
```

`Lambda` 引入符 `[n]` 说明 `lambda` 表达式可以访问局部变量 `n`，通过这种方式，对 `expe(x, n)` 的调用就可以在创建它的 `Function` 时得到 `n` 的值，而在每一次 `Function` 调用中得到 `x` 的值。

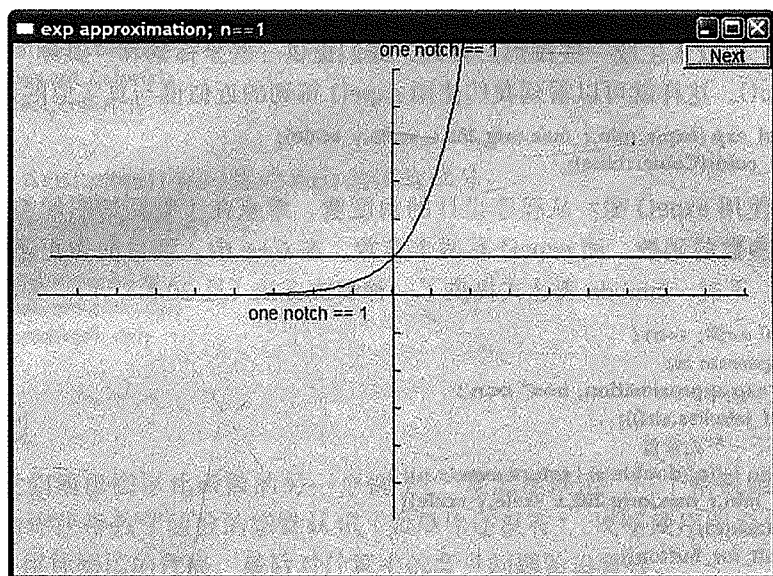
注意这个循环最后的 `detach(e)`。`Function` 对象 `e` 的作用域是在 `for` 循环体内。因此每执行一次循环步我们便得到一个名字为 `e` 的新 `Function` 对象，而每个循环步结束这个 `e` 就消亡了，下一次会被新的 `e` 代替。因为 `e` 将会销毁，所以窗口必须丢弃那个旧的 `e`。这样，`detach(e)` 就保证了窗口不会试图绘制一个已经被销毁的对象。

此代码首先显示一个窗口，其中只显示了坐标轴和蓝色的“实际”指数函数：



我们知道  $\exp(0)$  的值为 1，所以这条蓝色的“实际指数”与  $y$  轴相交于点  $(0,1)$ 。

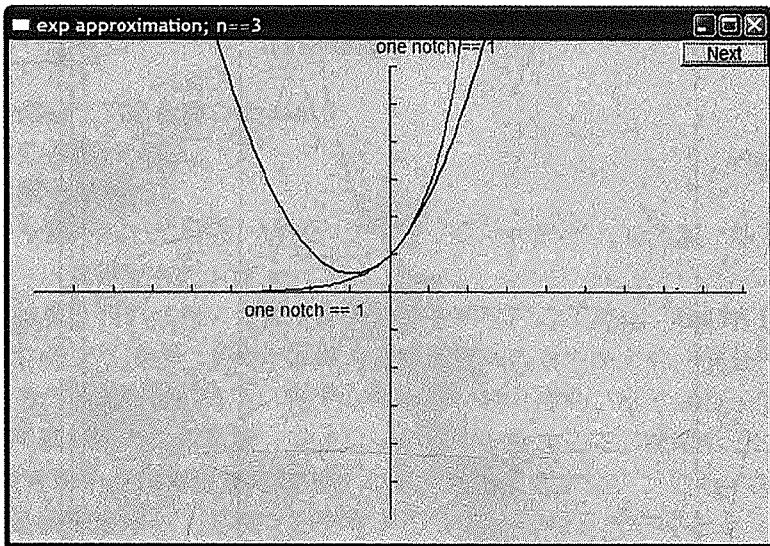
如果仔细观察，你会发现我们实际用一条黑色的线在  $x$  轴正上方绘制了零个项的近似值 ( $\exp_0(x) \equiv 1$ )。点击“Next”，我们得到只使用一项的近似值。注意我们将项数显示在窗口标题栏中：



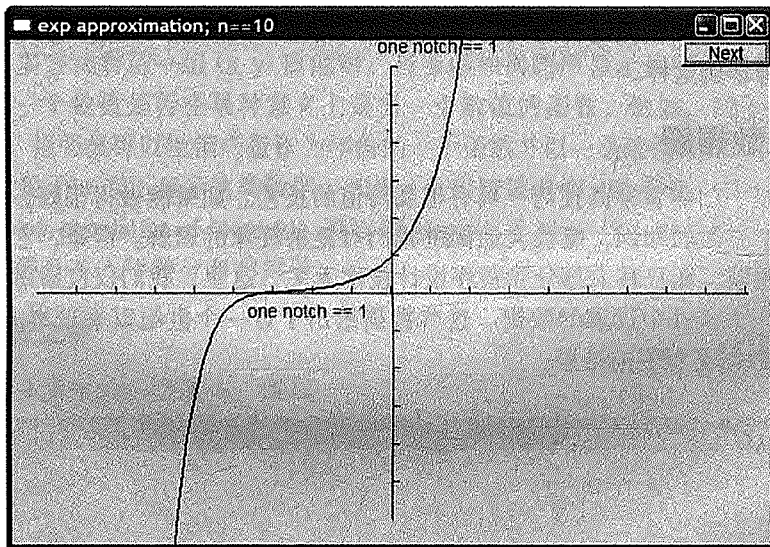
这里的函数是  $\exp_1(x) \equiv 1+x$ ，该近似值只使用了级数中一项。它准确地和指数函数相交于  $(0, 1)$ ，但我们还可以做得更好：



使用两项  $(1+x+x^2/2)$ ，我们得到和  $y$  轴相交于点  $(0,1)$  的对角线。使用三项  $(1+x+x^2/2+x^3/6)$ ，我们可以看到两条曲线开始汇聚：

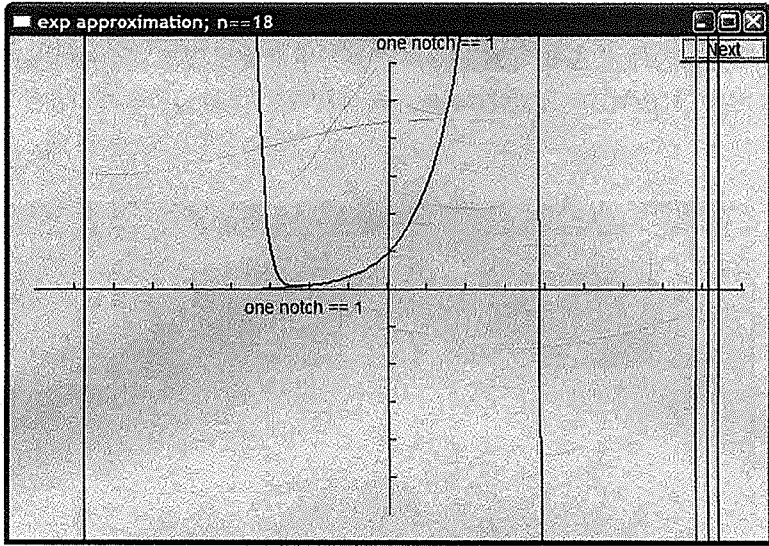


使用 10 项可以得到更好结果，特别是对于大于 -3 的值：



如果你不仔细思考这个问题的话，你可能会认为可以简单地通过使用越来越多的项来得到越来越好的近似值。然而，这是有极限的，当超过 13 项之后会发生一些奇怪的事情。首先，近似值开始慢慢变差，而在 18 项时会出现一些竖直线：

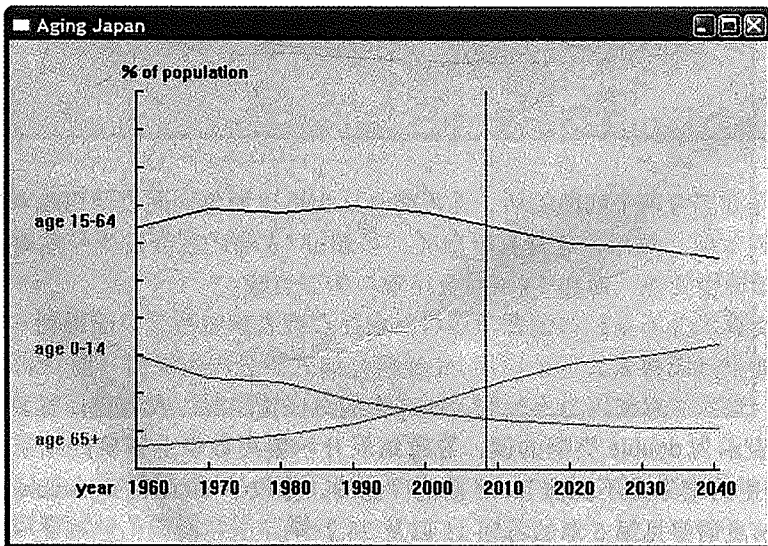
记住，浮点算术并不是纯粹的数学运算。用浮点数表示实数，只能得到和固定位数一样的近似结果。如果试图将太大的整数用 `int` 类型存储的话会产生溢出，而 `double` 类型存储的是一个近似值。当我注意到因为项数变大而产生的奇怪输出时，我首先怀疑我们的运算开始产生一些不能表示为 `double` 类型的值，导致结果开始偏离数学上正确的结果。后来，我才意识到 `fac()` 产生的结果是不能由 `int` 类型值存储的。修改了 `fac()` 得到 `double` 类型的值才解决问题。更多信息请参见第 5 章的习题 11 以及 24.2 节。 ⚠



最后这幅图片是“看起来正确”不等同于“通过测试”这一原则的一个很好的例子。在把程序交给他人使用之前，一定要进行测试，即便是那些起初看似合理的东西。除非你对程序有更深入的理解，否则稍微延长运行时间或者给出稍微不同的输入数据，就可能导致程序陷入混乱——就像本例这样。

## 20.6 绘制数据图

显示数据是一门需要很高技巧、具有很高价值的技艺。如果能做得很好，通常是结合了技术和艺术两个方面的知识，能极大地促进我们对复杂现象的理解。但是，它也使图形化变成一个巨大的领域，而且其大部分和程序设计技术无关。这里，我们仅仅展示一个简单的例子，它显示从一个文件中读取的数据。这些数据给出了近一个世纪以来日本人的年龄构成。2008 年以后的数据是预测的结果：



我们将使用这个例子来讨论显示这种数据涉及的程序设计问题：

- 读取文件；
- 调整数据的比例适合窗口的大小；
- 显示这些数据；
- 给图形加上标签。

我们不会涉及艺术上的细节。这基本上属于“简单的图形”，而不是“图形艺术”。当然，如果需要，你可以做得更有艺术性。

给定一组数据，我们必须考虑如何能最好地显示它。为了简化，我们将只处理那些方便用二维显示的数据，不过这也正是大部分人需要处理的数据中最大的一部分。注意，那些柱状图、饼图和类似的流行显示方式，其实也都是用一种有趣的二维方式显示的。三维数据的处理通常也是生成一系列二维图像，或者在一个窗口中叠加几张二维图形（如“日本人年龄”的例子），或者对单个点添加信息标签等。如果要超出这些方法，我们就必须编写新的图形类或者采用其他的图形库。

所以，我们的数据是基本的数值对，例如 (year, number of children)。如果有更多的数据，例如 (year, number of children, number of adults, number of elderly)，我们只是需要确定哪一对或者哪几对数据是需要绘制的。在我们的例子中，我们只是图形显示 (year, number of children)，(year, number of adults) 和 (year, number of elderly)。

我们有很多方式看待一组 (x, y) 数值对。当考虑如何图形显示这样一组数据的时候，重要的是要考虑一个数值是否在某种意义上是另一个数值的函数。例如，对于一个 (year, steel production) 对，很明显可以把钢产量作为年份的一个函数并以一条连续的线显示数据。可以用 `Open_polyline`（见 18.6 节）显示这类数据。如果 y 不应该被看作 x 的函数，例如 (gross domestic product per person, population of country)，可以用 `Marks`（见 18.15 节）绘制相互独立的点。

现在，回到日本人年龄分布的例子。

### 20.6.1 读取文件

年龄分布文件由很多行组成，例如：

```
(1960 : 30 64 6)
(1970 : 24 69 7)
(1980 : 23 68 9)
```

冒号后面的第一个数字是儿童（0 ~ 14 岁）在总人数中的百分比，第二个是成年人（15 ~ 64 岁）的百分比，第三个是老年人（65 岁以上）的百分比。我们的目标就是读出这些数据。注意，数据的格式有点不规则。与往常一样，我们必须处理这些细节。

为了简化这个任务，我们首先定义一个保存数据项的 `Distribution` 类型和一个读取这些数据项的输入操作符。

```
struct Distribution {
    int year, young, middle, old;
};

istream& operator>>(istream& is, Distribution& d)
// 假定格式为 (year: young middle old)
{
    char ch1 = 0;
    char ch2 = 0;
```

```

char ch3 = 0;
Distribution dd;

if (is >> ch1 >> dd.year
    >> ch2 >> dd.young >> dd.middle >> dd.old
    >> ch3) {
    if (ch1 != '(' || ch2 != ':' || ch3 != ')') {
        is.clear(ios_base::failbit);
        return is;
    }
}
else
    return is;
d = dd;
return is;
}

```

这是第 10 章中概念的一个直接应用。如果你不熟悉这段代码，请复习第 10 章。我们不是必须要定义一个 `Distribution` 类型和一个 `>>` 操作符。然而，相比于“只是读取数字并图形显示它们”的蛮力方法，这样做可以简化代码。我们使用 `Distribution` 将代码划分为有助于理解和调试的几个逻辑部分。不要觉得引入新类型“只是为了使代码清晰”。类的定义和使用能使代码更加直接地对应我们对概念的思考。即使对那些仅仅在代码局部区域中使用的“小”概念也应这么做，例如一行数据表示一年的年龄分布，这会很有帮助。

给定 `Distribution`，读取循环可这样设计：

```

string file_name = "japanese-age-data.txt";
ifstream ifs (file_name);
if (!ifs) error("can't open ",file_name);

// ...

for (Distribution d; ifs>>d; ) {
    if (d.year<base_year || end_year<d.year)
        error("year out of range");
    if (d.young+d.middle+d.old != 100)
        error("percentages don't add up");
    // ...
}

```

即，我们试图打开文件“japanese-age-data.txt”，如果找不到这个文件就退出程序。不将文件名“硬编码”到源代码中会更好一些，但我们考虑到这只是一个“一次性”的小程序，所以没有采用这种更好的方法。不将文件名“硬编码”的便利方式适合于长期使用的应用程序，但会增加这个小程序的负担。另一方面，我们确实是把 `japanese-age-data.txt` 存在一个命名的 `string` 变量中，如果我们将此程序或它的一部分用作它用，程序也很容易修改。

读取循环检查所读年份是否在预期的范围内，同时检查百分比之和是否为 100。这是一个基本的数据检查，因为 `>>` 检查了每个单独的数据项的格式，我们不用在主循环中再做更多的检查。

## 20.6.2 一般布局

那么我们想在屏幕上显示什么呢？你可以从 20.6 节的开始看到答案。这些数据看起来需要三个 `Open_polyline`，分别对应三个年龄组。因为这些图形需要添加标签，所以我们决定为每条线在窗口的左侧写一个“标题”。在这种情况下，这种方式看起来比将标签放在线上



某个位置的方式更为清晰。另外，我们使用颜色来区分图形并与标签关联。

我们想用年份来标注  $x$  轴。2008 年处的那条竖直线表明后面的图形是根据预测数据绘制的。

我们决定使用窗口标签作为图形的标题。

让图形化的代码既正确又美观是非常棘手的。主要原因是我们需要做很多有关尺寸和偏移量的高精度计算。为了简化，我们开始先定义一组符号常量来表示对屏幕空间的使用方式：

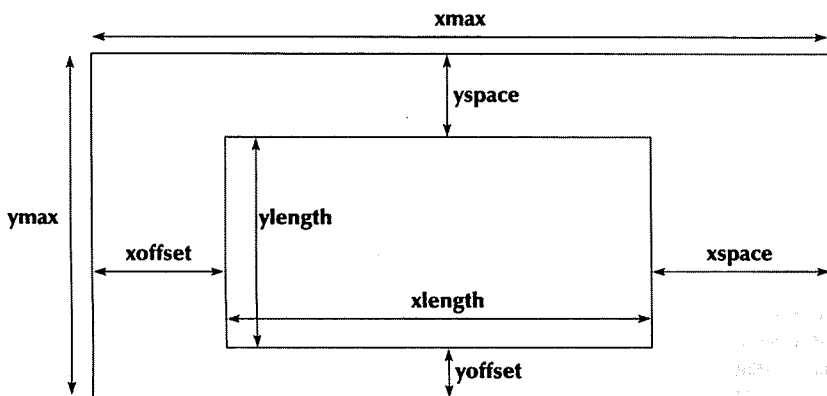
```
constexpr int xmax = 600; // 窗口大小
constexpr int ymax = 400;

constexpr int xoffset = 100; // 窗口左边到 y 坐标轴的距离
constexpr int yoffset = 60; // 窗口底部到 x 坐标轴的距离

constexpr int xspace = 40; // 坐标轴上方的空间
constexpr int yspace = 40;

constexpr int xlength = xmax - xoffset - xspace; // 坐标轴长度
constexpr int ylength = ymax - yoffset - yspace;
```

本质上这里定义了一个矩形空间（窗口）及其内部的另一个矩形（通过坐标轴定义）。



如果没有这样一个表明窗口中的事物位置的“示意图”和用于定义位置的符号常量，当程序输出不能反映所要求的结果时，我们将会迷失并且感到无助。

### 20.6.3 数据比例

接下来，需要定义怎样才能让数据适合这个空间。我们通过按比例缩放数据来达到这个目的，使数据适合于由坐标轴定义的空间。因此，我们需要使用比例因子，即数据范围和坐标轴范围之间的比值。

```
constexpr int base_year = 1960;
constexpr int end_year = 2040;

constexpr double xscale = double(xlength)/(end_year - base_year);
constexpr double yscale = double(ylength)/100;
```

比例因子（ $xscale$  和  $yscale$ ）应该是浮点数——否则我们的运算将会面对严重的舍入误差。为了避免整数除法，在做除法之前先把长度数据转换成 `double` 类型（见 4.3.3 节）。



现在，我们可以通过减去基准值（1960），使用 `xscale` 进行比例缩放，并加上 `xoffset`，将一个数据点放到  $x$  轴上。按同样的方式可以处理  $y$  轴上的数据。不过，当试图重复做这件事情的时候，我们发现很难记得非常清楚。这可能是一个不太重要的运算，但它是需要技巧和时间的。为了简化代码并减小出错的机会（尽量减少令人沮丧的调试工作），我们定义了一个很小的类来完成这个运算。

```
class Scale {           // 数据值转化为坐标值
    int cbase;         // 坐标基准
    int vbase;         // 基准值
    double scale;

public:
    Scale(int b, int vb, double s) : cbase(b), vbase(vb), scale(s) {}
    int operator()(int v) const { return cbase + (v-vbase)*scale; } // 参见 16.4 节
};
```

我们需要一个类，因为这个运算依赖于三个常量值，而我们又愿意总是不必要地重复它们。给定这个类，我们可定义：

```
Scale xs (xoffset,base_year,xscale);
Scale ys (ymax-yoffset,0,-yscale);
```

注意我们是如何使 `ys` 的比例因子变为负值，以反映  $y$  坐标向下增长的——我们通常用图形上更高的点表示更大的数值。现在，我们可以使用 `xs` 将一个年份转换为  $x$  坐标。类似地，可以使用 `ys` 将一个百分数转换为  $y$  坐标。

#### 20.6.4 构造数据图

最后，我们具备了采用合理方式来编写图形化代码的所有先决条件。首先我们创建窗口并放置坐标轴：

```
Window win (Point{100,100},xmax,ymax,"Aging Japan");

Axis x (Axis::x, Point{xoffset,ymax-yoffset}, xlength,
        (end_year-base_year)/10,
        "year 1960 1970 1980 1990 "
        "2000 2010 2020 2030 2040");
x.label.move(-100,0);

Axis y (Axis::y, Point{xoffset,ymax-yoffset}, ylength, 10,"% of population");

Line current_year (Point{xs(2008),ys(0)},Point{xs(2008),ys(100)});
current_year.set_style(Line_style::dash);
```

坐标轴的交点 `Point{xoffset, ymax-yoffset}` 表示 (1960, 0)。注意这里是如何放置刻度来反映数据的。在  $y$  轴上有 10 个刻度，每一个刻度代表 10% 的人口。在  $x$  轴上，每个刻度代表 10 年，而具体的刻度数值是通过 `base_year` 和 `end_year` 计算得来的，因此，如果我们改变这个范围，该坐标轴也会自动地重新计算。这是在代码中避免使用“魔数”的一个优点。但是，在  $x$  轴上的标签违反了 this 规则：它只是用手动调整标签字符串的方式得到的结果，直到数字正好在对应的刻度下面。更好的方法是针对一组单独的“刻度”给出一组对应的标签。

请注意标签字符串的格式，我们使用了两个相邻的文字常量字符串：

```
"year 1960 1970 1980 1990 "
"2000 2010 2020 2030 2040"
```

相邻的文字常量字符串会被编译器连接起来，相当于：

```
"year 1960 1970 1980 1990 2000 2010 2020 2030 2040"
```

这是一个用来布局长字符串从而使代码更可读的有用“技巧”。

`current_year` 对象是一条分割已知数据和预测数据的垂直线。注意如何使用 `xs` 和 `ys` 来正确地布局和按比例缩放这条线。

给定坐标轴，我们就可以处理数据了。定义 3 个 `Open_polyline` 对象，在读取循环中向它们添加数据。

```
Open_polyline children;
Open_polyline adults;
Open_polyline aged;

for (Distribution d; ifs>>d; ) {
    if (d.year<base_year || end_year<d.year) error("year out of range");
    if (d.young+d.middle+d.old != 100)
        error("percentages don't add up");
    const int x = xs(d.year);
    children.add(Point{x,ys(d.young)});
    adults.add(Point{x,ys(d.middle)});
    aged.add(Point{x,ys(d.old)});
}
```

`xs` 和 `ys` 的使用可以让数据的放置和按比例缩放变得容易。像 `Scale` 这种“很小的类”对于简化符号和避免不必要的重复是非常重要的，能够提高程序的可读性和正确性。

为了使图形更具有可读性，我们为每一个图形添加标签并设置颜色：

```
Text children_label {Point{20,children.point(0).y},"age 0-14"};
children.set_color(Color::red);
children_label.set_color(Color::red);

Text adults_label {Point{20,adults.point(0).y},"age 15-64"};
adults.set_color(Color::blue);
adults_label.set_color(Color::blue);

Text aged_label {Point{20,aged.point(0).y},"age 65+"};
aged.set_color(Color::dark_green);
aged_label.set_color(Color::dark_green);
```

最后，我们需要把不同的 `Shape` 对象添加到 `Window` 对象中，然后启动 GUI 系统（见 19.2.3 节）：

```
win.attach(children);
win.attach(adults);
win.attach(aged);

win.attach(children_label);
win.attach(adults_label);
win.attach(aged_label);

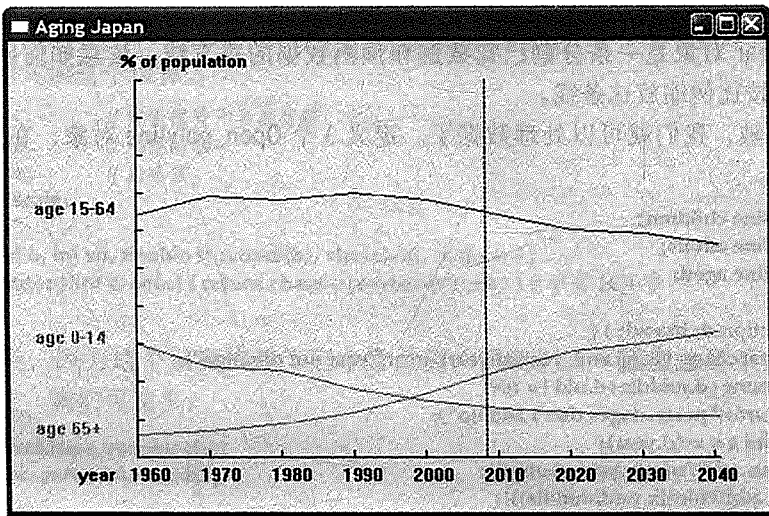
win.attach(x);
win.attach(y);
win.attach(current_year);

gui_main();
```

所有代码都可以放在 `main()` 中，但我们认为将辅助类 `Scale` 和 `Distribution` 与 `Distribution` 的

输入操作符一起放在 `main()` 之外更好。

假如你已经忘记我们正在生成什么图形，我们再次给出输出结果：



## 简单练习

函数图形显示练习：

1. 创建一个标签为“Function graphs”的空的大小为  $600 \times 600$  的 Window。
2. 注意，你可能需要参照“installation of FLTK”（可从课程网站下载）中的说明创建一个项目，并设置项目属性。
3. 将 `Graph.cpp` 和 `Window.cpp` 移入你的项目中。
4. 向窗口中加入一条  $x$  坐标轴和一条  $y$  坐标轴，长度均为 400，标签为“1=20 pixels”，每隔 20 个像素画一个刻度。两条坐标轴相交于  $(300, 300)$ 。
5. 将两条坐标轴都设置为红色。

在下面的练习中，对每个图形显示的函数都使用一个独立的 Shape。

1. 图形显示函数 `double one(double x) { return 1; }`，参数范围  $[-10, 11]$ ，原点  $(0, 0)$  位于坐标点  $(300, 300)$ ，输出 400 个点，（在窗口中）不缩放。
2. 将  $x$  轴和  $y$  轴缩放比例均改为 20。
3. 之后所有练习均使用当前的参数范围和缩放比例等设置。
4. 将 `double slope(double x) { return x/2; }` 的图形显示加到窗口中。
5. 用 Text 对象“ $x/2$ ”为斜线添加标签，添加位置为斜线的左下角。
6. 将 `double square(double x) { return x*x; }` 的图形显示加入窗口中。
7. 向窗口中加入余弦曲线（不要编写一个新函数）。
8. 将余弦曲线设置为蓝色。
9. 编写函数 `sloping_cos()`，将余弦曲线添加到 `slope()`（如上定义）上，并将此函数的图形显示加入窗口。

类定义练习：

1. 定义一个 struct `Person`，包含一个类型为 `string` 的名字 (`name`) 和类型为 `int` 的年龄 (`age`)。

2. 定义一个类型为 `Person` 的变量，用 “Goofy” 和 63 对其初始化，并将其输出到屏幕 (`cout`)。
3. 为 `Person` 定义输入操纵符 (`>>`) 和输出操纵符 (`<<`)，从键盘 (`cin`) 读入一个 `Person` 值，并将其写到屏幕 (`cout`)。
4. 为 `Person` 定义一个构造函数，初始化 `name` 和 `age`。
5. 将 `Person` 的描述改为 `private`，并提供 `const` 成员函数 `name()` 和 `age()` 实现名字和年龄的读取。
6. 修改 `>>` 和 `<<`，使之适应重新定义过的 `Person`。
7. 修改构造函数，检查 `age`，保证它在范围 `[0:150)` 之内，检查 `name`，保证它不包含 `;"'[]* & ^ % $ # @ !`。如果发生错误，使用 `error()`。测试这个构造函数。
8. 从标准输入 (`cin`) 读取一组 `Person`，存入一个 `vector<Person>` 中，将它们写到屏幕 (`cout`)。用正确和错误的输入数据测试这个程序。
9. 修改 `Person` 的描述，用 `first_name` 和 `second_name` 代替 `name`。如果用户未提供 `first_name` 或 `second_name`，则给出一个错误。相应修改 `>>` 和 `<<`，并进行测试。

## 思考题

1. 接受一个参数的函数是怎样的？
2. 什么情况下你会使用（连续的）线表示数据？什么情况下你会使用点？
3. 什么函数（数学公式）定义一条斜线？
4. 什么是抛物线？
5. 如何生成  $x$  轴和  $y$  轴？
6. 什么是默认参数？什么时候使用默认参数？
7. 如何把函数叠加在一起？
8. 如何给一个图形化的函数加上颜色和标签？
9. 我们说一个级数近似一个函数，这是什么意思？
10. 为什么在编写代码绘制图形之前需要画出它的布局草图？
11. 如何按比例缩放图形使得输入恰好适合显示区域？
12. 如何按比例缩放输入可以避免试验和误差？
13. 为什么要格式化输入，而不只是让文件包含那些“数字”？
14. 如何设计图形的总体布局？如何在代码中反映出来？

## 术语

approximation (近似)	function (函数)	scaling (缩放比例)
default argument (默认参数)	lambda	screen layout (屏幕布局)

## 习题

1. 下面是定义阶乘函数的另一种方式：

```
int fac(int n) { return n>1 ? n*fac(n-1) : 1; } // 阶乘 n!
```

对于 `fac(4)`，因为 `4>1`，所以第一次调用会执行 `4*fac(3)`，接下来是 `4*3*fac(2)`，然后是

- $4*3*2*fac(1)$ , 即  $4*3*2*1$ 。试着体会它是怎么执行的。一个函数调用它自身称为递归 (recursive)。在 20.5 节中给出的另一种实现方式称为迭代 (iterative), 因为它对一系列数值反复进行计算 (使用 while)。验证递归函数  $fac()$  的执行过程, 并通过计算 0, 1, 2, 3, 4, ..., 20 的阶乘, 验证是否与迭代函数  $fac()$  有相同的执行结果。你更倾向于  $fac()$  的哪种实现? 为什么?
2. 定义一个  $Fct$  类, 除了存储构造函数的参数之外, 它与  $Function$  类一样。给  $Fct$  类提供“复位”(reset) 操作, 从而可以重复利用它对不同的范围、不同的函数等生成图形。
  3. 修改上面的  $Fct$  类, 使其带有一个额外的参数来控制精度或者其他内容。为了更加灵活, 可将该参数的类型设置为模板参数。
  4. 在一个图上绘制正弦 ( $\sin()$ )、余弦 ( $\cos()$ )、正弦与余弦的和 ( $\sin(x)+\cos(x)$ ) 以及正弦平方与余弦平方的和 ( $\sin(x)*\sin(x)+\cos(x)*\cos(x)$ )。注意要提供坐标轴和标签。
  5. “动态显示”(像 20.5 节中那样) 级数  $1-1/3+1/5-1/7+1/9-1/11+\dots$ 。它是著名的莱布尼兹级数, 收敛到  $\pi/4$ 。
  6. 设计并实现一个柱状图类。它的基本数据是一个保存  $N$  个数值的  $vector<double>$ , 每个数值用一个矩形形状的“柱”(bar) 表示, 其高度代表对应的数值。
  7. 细化该柱状图类, 实现对图本身和每一个单独的柱添加标签的功能, 并允许使用颜色。
  8. 有一个以厘米为单位的身高 (取整到最接近的 5cm 的整数倍值) 和身高为对应值的人数构成的集合: (170, 7), (175, 9), (180, 23), (185, 17), (190, 6), (195, 1)。如何图形化这些数据? 如果你想不到更好的方法, 做一个柱状图。记得提供坐标轴和标签。把数据放在一个文件中, 并从该文件读取这些数据。
  9. 找另一个身高的数据集合 (英制, 1 英寸大约 2.54 厘米), 然后用前面练习中的程序图形化这些数据。例如, 从网上搜索“身高分布”或者“美国人的身高”, 并忽略没用的内容, 或者向你的朋友询问他们的身高。理想情况下, 你不需要对新的数据集做任何改变。关键思路是计算出数据的缩放比例。从输入中读取标签也有助于代码重用尽量少地修改代码。
  10. 什么样的数据不适合用线图或者柱状图表示? 寻找一个例子, 给出显示它的一种方法 (例如一个标记点的集合)。
  11. 计算两个或者更多地点 (例如, 英格兰的剑桥和马萨诸塞州的剑桥; 有很多城镇叫作“剑桥”) 一年中每个月的平均最高气温, 并将它们显示在一张图上。注意坐标轴、标签、颜色的使用等。

## 附言

数据的图形表示是重要的。与一组数据相比, 我们更容易理解由数据制作而来的图。当需要绘图形的时候, 大部分人都会使用其他人的代码——函数库。这些库是如何构造的呢? 而如果你手头没有这样一个库, 又该如何做呢? “一个普通的绘图工具”背后的基本思想是什么呢? 现在你已经知道: 它不是神秘的魔术或者脑外科手术。我们只讨论了二维图形; 而三维图形化表示在科学、工程、市场等领域同样非常有用, 甚至更加有趣。将来如有恰当的时机, 请仔细研究它们!

# 图形用户界面

计算不再是指计算机，而是生活。

——尼古拉斯·尼葛洛庞帝

图形用户界面（GUI）允许用户通过点击按钮、选择菜单、以多种方式输入数据以及在屏幕上显示文本和图形等方式与程序进行交互。这正是我们与电脑以及网站交互时经常采用的方式。在本章中，我们将介绍编写代码来定义和控制 GUI 应用的基本方法。特别地，我们将介绍如何编写代码，通过回调函数与屏幕上的实体进行交互。我们的 GUI 工具是建立在系统工具之上的。附录 E 给出了更低层次的特性和接口，这些特性和接口使用了第 12 章和 13 章中介绍的特性和技术。在本章中，我们将注意力集中在使用方法上。

## 21.1 用户界面的选择

每个程序都有用户接口。在小装置上运行的程序接口可能局限于从几个按钮输入和用一个闪光信号灯输出。而其他的计算机仅仅通过一条线就可以连接到外面的世界。这里，我们将考虑一般的情况，即程序是和一个看着屏幕、使用键盘和定点设备（如鼠标）的用户进行交互。在这种情况下，程序员有三种主要的选择：

- 使用控制台输入 / 输出：对于专业技术工作而言，这是一种强有力的方式，输入是简单的、文本式的，由一些命令和短数据项（比如文件名或者简单的数值）组成。如果输出也是文本式的，我们可以将它显示在屏幕上或者存储在文件中。C++ 的标准库 `iostream`（参见第 10 ~ 11 章）为这种方式提供了适合、方便的机制。如果需要图形输出，我们可以使用一个图形显示库（如第 17 ~ 20 章），不需要对我们的程序设计风格进行明显的修改。
- 使用图形用户界面（GUI）库：用户的交互基于操纵屏幕对象的方式（定点、点击、拖放、悬停等）。通常（但不总是），这种方式总是伴随着信息的高度图形化显示。任何用过现代计算机的人都能举出一些体现这种方式便利性的例子。任何希望感受 Windows/Mac 应用的人都需要使用 GUI 交互方式。
- 使用网络浏览器界面：对于这种方式，我们需要使用一种标记（布局）语言，例如 HTML 或者 XML，通常还会使用一种脚本语言。阐述如何实现这种方式已经超出了本书的讨论范围，但一般来说，它是有远程访问需求的应用程序的理想模式。在这种方式下，程序和屏幕之间的通信还是文本方式的（使用字符流）。浏览器是一个 GUI 应用程序，它负责将其中一些文本信息转换成图形元素，并将鼠标点击等转换成文本数据传递回程序。

对于很多人来说，GUI 的使用就是现代程序设计的本质，而且有时与屏幕对象的交互被认为是程序设计的核心概念。我们并不同意这种观点：GUI 是一种 I/O 形式，应用程序的主要逻辑和 I/O 相互分离是软件设计的主要观点之一。无论是否可能，我们更愿意在主要程序

逻辑和用来进行输入 / 输出的部分之间建立起一个清晰的接口。这种分离机制允许我们通过改变程序呈现给用户的方式，来将程序移植到不同的 I/O 系统中，更重要的是，这种机制允许我们将程序逻辑和用户交互分开来考虑。

也就是说，从多个角度来看，GUI 都是非常重要和有趣的。本章既研究如何把图形元素集成到我们的应用程序中，同时也探索如何防止对界面的过度关注而影响我们的思维。

## 21.2 “Next” 按钮

如何提供一个像第 17 ~ 20 章例子中用于驱动图形显示的“Next”按钮呢？在那里，我们使用窗口中的一个按钮绘制图形。很明显，这是一种简单的 GUI 程序设计模式。实际上，因为它过于简单，以至于有些人可能会争论它是不是一个“真正的 GUI”。无论如何，让我们看看它是怎样实现的，因为它将引领我们直接进入所有人都认可的真正的 GUI 程序设计。

第 17 ~ 20 章中代码的典型结构如下：

```
// 创建对象并且 / 或者操纵对象，显示在窗口 win 中
win.wait_for_button();
```

```
// 创建对象并且 / 或者操纵对象，显示在窗口 win 中
win.wait_for_button();
```

```
// 创建对象并且 / 或者操纵对象，显示在窗口 win 中
win.wait_for_button();
```

每当运行到 `wait_for_button()`，就可以在屏幕上看到要显示的对象，直到我们点击按钮来得到程序下一部分的输出。从程序逻辑的观点来看，这种方式与逐行输出到屏幕（控制台窗口），在某处停下来，然后从键盘接收输入的程序没有区别。例如：

```
// 定义变量并且 / 或者计算，产生输出
cin >> var; // 等待后续输入
```

```
// 定义变量并且 / 或者计算，产生输出
cin >> var; // 等待后续输入
```

```
// 定义变量并且 / 或者计算，产生输出
cin >> var; // 等待后续输入
```

但是从实现的观点来看，这两种程序截然不同。当你的程序执行到 `cin>>var` 时，它停下来并且等待“系统”读取你输入的字符。然而，监视屏幕并且跟踪鼠标的系统（图形用户界面系统）运行在一种截然不同的模式下：GUI 跟踪鼠标的位置和用户对鼠标所做的操作（点击等）。当你的程序需要一个动作时，它必须：

- 告诉 GUI 关心哪些事情（例如，“有人点击了‘Next’按钮”）；
- 告诉 GUI 当有人做这些事的时候，应该如何处理；
- 在 GUI 检测到程序感兴趣的动作之前一直处于等待状态。

与控制台程序的不同之处在于，GUI 并不是简单地返回我们程序；它的设计目标是对很多不同的用户动作给出不同的响应，例如点击很多按钮中的某一个、改变窗口的尺寸、当窗口被其他内容挡住之后重新绘制窗口以及弹出“弹出式”菜单等。

首先，我们只是想说，“当有人按下我的按钮时请将我唤醒”；也就是说，“当有人点击鼠标按钮，且光标位置在我的按钮图形显示的矩形区域内时，请继续执行我的程序”。这是

我们能够想象到的最简单的动作。然而，这样的操作并不是由“系统”提供的，所以我们必须自己编写代码。观察它的实现方法是理解 GUI 程序设计的第一步。

### 21.3 一个简单的窗口

实际上，“系统”（GUI 库和操作系统的组合）不断跟踪鼠标的位置及其按钮是否被按下。✂  
一个程序可以关注屏幕的某个区域，并请求“系统”在某些被关注的事件发生时调用某个函数。在本例中，我们请求系统当用户在“我们的按钮”上点击鼠标时，调用我们的一个函数（“回调函数”）。要完成这些，必须：

- 定义一个按钮；
- 显示按钮；
- 定义一个 GUI 可以调用的函数；
- 将定义的按钮和函数告知 GUI；
- 等待 GUI 调用我们的函数。

下面进行具体实现。按钮是 Window 的一部分，所以我们（在 Simple\_window.h 中）定义了类 Simple\_window，这个类包括数据成员 next\_button：

```
struct Simple_window : Graph_lib::Window {
    Simple_window(Point xy, int w, int h, const string& title);

    void wait_for_button(); // 简单的事件循环
private:
    Button next_button;     // “Next” 按钮
    bool button_pushed;    // 实现细节

    static void cb_next(Address, Address); // next_button 的回调
    void next();           // next_button 被按下时将执行相关动作
};
```

很显然，类 Simple\_window 派生自 Graph\_lib 库的 Window 类。所有的窗口都必须直接或者间接地派生自 Graph\_lib::Window 类，因为它将我们对窗口的设想和系统的窗口实现（通过 FLTK）连接起来的类。对于 Window 类实现的细节，可参考附录 E.3。

我们的按钮在 Simple\_window 的构造函数中被初始化：

```
Simple_window::Simple_window(Point xy, int w, int h, const string& title)
    :Window(xy,w,h,title),
    next_button(Point{x_max()-70,0}, 70, 20, "Next", cb_next),
    button_pushed(false)
{
    attach(next_button);
}
```

不出意外地，Simple\_window 将自己的位置 (x, y)、尺寸 (w, h) 和标题 (title) 传递给 Graph\_lib 的 Window 类来处理。接下来，构造函数使用位置 (Point{x\_max()-70, 0})，大致位于右上角)、尺寸 (70, 20)、标签 (“Next”) 和一个“回调”函数 (cb\_next) 初始化 next\_button。前四个参数的作用与我们对 Window 所做的相同：将一个矩形形状放在屏幕上并且为它添加标签。

最后，我们通过 attach() 将 next\_button 按钮添加到 Simple\_window 中；也就是说，告知窗口必须要将这个按钮显示在它的位置上，并且保证 GUI 系统知道它的存在。



`button_pushed` 成员是一个相当隐晦的实现细节，我们用它来跟踪自从上次执行 `next()` 起到现在按钮是否被按下。事实上，这里做的每件事基本上都属于实现细节，因此将其声明为 `private`。忽略实现细节后的代码为：

```
struct Simple_window : Graph_lib::Window {
    Simple_window(Point xy, int w, int h, const string& title);

    void wait_for_button(); // 简单的事件循环

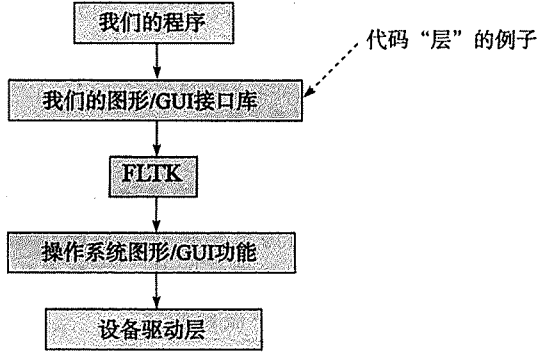
    // ...
};
```

也就是说，用户可以创建一个窗口并等待按钮被按下。

### 21.3.1 回调函数

✂ 此处，函数 `cb_next()` 是一个新的而且很有趣的代码。它就是当 GUI 系统检测到按钮被按下时，我们希望 GUI 系统调用的那个函数。由于我们将该函数交给 GUI，以便 GUI 能“回过头来调用我们”，所以它通常被称为回调函数（callback function）。我们通过它的前缀 `cb_` 代表“回调”来指出 `cb_next()` 的用途。前缀的作用只是帮助我们理解——没有任何语言或者库有这样的命名要求。很明显，我们选择名字 `cb_next` 是因为它是“Next”按钮的回调函数。`cb_next` 的定义是一段比较丑陋的“样本”代码。

在给出代码之前，先让我们看看这里发生的事情：



✂ 我们的程序是运行在多“层”代码之上的。它使用我们的图形库，而图形库是利用 FLTK 库实现的，FLTK 库又是使用操作系统功能实现的。在一个系统中，可能还会有更多的层和子层。无论以什么方式，当鼠标设备驱动检测到点击操作时，我们的函数 `cb_next()` 必须被调用。我们将 `cb_next()` 的地址和 `Simple_window` 对象的地址经过软件层次向下传递；当“Next”按钮被按下时，一些“下层”的代码就会调用 `cb_next()` 函数。

GUI 系统（和操作系统）可以被不同语言编写的程序使用，所以它不能向所有用户强加一些好的 C++ 风格。特别地，它并不知道我们的 `Simple_window` 类或者 `Button` 类。事实上，它根本就不知道类和成员函数的概念。回调函数的类型是经过小心选择的，以便可被低层的程序设计（包括 C 和汇编）所用。回调函数是没有返回值的，接受两个地址参数。我们可以遵从这些规则来声明一个 C++ 成员函数如下：

```
static void cb_next(Address, Address); // next_button 的回调
```

关键字 `static` 用于保证 `cb_next()` 可以作为一个普通函数被调用；也就是说，不是作为针对某个特定对象的 C++ 成员函数被调用。令系统能够正确调用一个 C++ 成员函数当然很好，但回调接口必须能被多种语言所用，所以我们只能将其定义为 `static` 类型的成员函数。`Address` 参数指明 `cb_next()` 的参数是“内存中某些内容”的地址。大部分语言都不支持 C++ 语言的引用，所以这里不能使用引用。编译器并不告诉你“那些内容”是什么类型的。在本例中，我们非常接近底层硬件，不能像往常那样从语言中得到帮助。“系统”激活一个回调函数时，传递给它的第一个参数是触发回调的 GUI 实体（Widget）的地址。本例中不需要使用第一个参数，所以我们不关心它的命名。第二个参数包含这个 Widget 的窗口的地址；对于 `cb_next()` 来说，它是我们的 `Simple_window` 对象。我们可以按照如下方式使用这个信息：

```
void Simple_window::cb_next(Address, Address pw)
// 为位于 pw 处的窗口调用 Simple_window::next()
{
    reference_to<Simple_window>(pw).next();
}
```

`reference_to<Simple_window>(pw)` 告诉编译器 `pw` 中的地址可以认为是 `Simple_window` 对象的地址；也就是说，我们可以将 `reference_to<Simple_window>(pw)` 当作 `Simple_window` 对象的引用来使用。在附录 E.1 中，我们给出了 `reference_to` 的定义。现在，我们只是乐于看到最后获得了一个指向 `Simple_window` 对象的引用，这样就可以像以往那样访问我们的数据和函数。最后，通过调用我们的成员函数 `next()`，尽可能快地脱离和系统有关的代码。

我们本来可以将所有想要执行的代码都放在 `cb_next()` 中，但是像大多数好的 GUI 程序设计者一样，我们更愿意把这些麻烦的低层内容和精巧的用户代码相分离，所以我们使用两个函数来处理回调：

- `cb_next()` 简单地将回调函数的系统约定映射到一个普通的成员函数 (`next()`)。
- `next()` 实现我们实际要做的事情（不需要知道回调函数的系统约定）。

使用两个函数的本质原因是一个通用设计原则，即“一个函数应该执行单一的逻辑行为”：`cb_next()` 使我们脱离了低层系统相关的部分，`next()` 执行我们期望的动作。任何时候，当我们需要一个（来自“系统”的）对我们窗口的回调时，就可以定义这样一对函数；例如，可以参考 21.5 ~ 21.6 节。继续下一步之前，让我们先重复一下到目前为止我们做了什么：

- 定义了我们的 `Simple_window` 类。
- `Simple_window` 的构造函数将 `next_button` 对象注册到 GUI 系统。
- 当我们点击屏幕上的 `next_button` 时，GUI 调用 `cb_next()` 函数。
- `cb_next()` 将低层的系统信息转换成对我们的窗口成员函数 `next()` 的调用。
- `next()` 执行我们想要做的事情以响应按钮点击的动作。

这是进行函数调用的一个相当完美的方法。不过要记住，我们是在处理鼠标（或者其他硬件设备）动作和程序之间的基本通信机制。特别地：

- 通常有很多程序同时在运行。
- 这些程序都是在操作系统之后很久才编写的。
- 这些程序都是在 GUI 库之后很久才编写的。
- 这些程序所用的语言可能和实现操作系统的语言完全不同。
- 这种技术处理所有类型的交互（不仅仅是按下按钮这样的小例子）。
- 一个窗口可以有多个按钮，一个程序可以有多个窗口。

不过，一旦理解了 `next()` 是如何被调用的，我们就从本质上理解了如何使用 GUI 接口来处理程序中所有动作。

### 21.3.2 等待循环

那么，在这种最简单的情况下，我们希望 `Simple_window` 的 `next()` 函数在每次按下按钮的时候做些什么呢？本质上，我们需要一个能将我们的程序停止在某个点上的操作，以便让我们有机会观察到目前为止已经完成了哪些工作。同时还希望 `next()` 函数能够从停止点重新启动程序：

```
// 创建对象并且 / 或者操纵对象，将其显示在窗口中
win.wait_for_button(); // next() 使得程序从这里开始
// 创建对象并且 / 或者操纵对象
```

实际上，这很容易做到。首先定义 `wait_for_button()`：

```
void Simple_window::wait_for_button()
    // 修改事件循环
    // 处理所有事件（按每个事件的默认方式），当 button_pushed 为 true 时退出
    // 这条语句实现绘图而不会有控制流反转
{
    while (!button_pushed) Fl::wait();
    button_pushed = false;
    Fl::redraw();
}
```



像大多数 GUI 系统一样，FLTK 提供了一个暂停程序运行的函数，直到某个事件发生程序才恢复运行。这个 FLTK 版本的函数称为 `wait()`。实际上，`wait()` 需要关注很多事情，因为发生任何影响我们程序的事件都会将程序唤醒。例如，如果程序运行在 Microsoft Windows 环境下，当窗口被移动或者被其他窗口遮挡时，窗口的重新绘制工作是由程序来完成的。`Window` 类还需要处理调整窗口尺寸的工作。在默认情况下，`Fl::wait()` 会处理所有这些工作。每当 `wait()` 函数处理完成某个事件后，它总会返回，使我们的代码有机会处理一些事情。

因此，当有人点击我们的“Next”按钮时，`wait()` 就会调用 `cb_next()` 并且返回（到我们的“等待循环”）。为了使 `wait_for_button()` 继续运行，`next()` 只需将布尔变量 `button_pushed` 设置为 `true`。这很容易：

```
void Simple_window::next()
{
    button_pushed = true;
}
```

当然，还需要在某个合适的地方预先定义 `button_pushed`：

```
bool button_pushed; // 在构造函数中初始化为 false
```

等待之后，`wait_for_button()` 需要将 `button_pushed` 复位并且重绘（`redraw()`）窗口以保证我们所做的任何改变都能够在屏幕上表现出来。这就是它所做的全部。

### 21.3.3 lambda 表达式作为回调函数

对于 `Widget` 上的每一个动作，我们都应该定义两个函数：一个来映射回调函数的系统

约定，一个来做我们实际要做的动作。考虑下面的代码：

```
struct Simple_window : Graph_lib::Window {
    Simple_window(Point xy, int w, int h, const string& title);

    void wait_for_button(); // 简单的事件循环
private:
    Button next_button;    // “Next” 按钮
    bool button_pushed;   // 实现细节

    static void cb_next(Address, Address); // next_button 的回调
    void next();          // 当 next_button 被按下时要做的动作
};
```

通过使用 lambda 表达式（见 20.3.3 节），我们可以不需要显式声明映射函数 cb\_next()。相反，我们在 Simple\_window 的构造函数中这样来定义映射关系：

```
Simple_window::Simple_window(Point xy, int w, int h, const string& title)
    :Window(xy,w,h,title),
    next_button(Point{x_max()-70,0}, 70, 20, "Next",
        [](Address, Address pw) { reference_to<Simple_window>
            (pw).next(); }
    ),
    button_pushed(false)
{
    attach(next_button);
}
```

## 21.4 Button 和其他 Widget

我们按照下面的方式来定义按钮：

```
struct Button : Widget {
    Button(Point xy, int w, int h, const string& label, Callback cb);
    void attach(Window&);
};
```

由此可知，按钮（Button）是一个具有位置（xy）、尺寸（w、h）、文本标签（label）和回调函数（cb）的 Widget。基本上，任何出现在屏幕上带有关联动作（例如回调函数）的东西都是 Widget。

### 21.4.1 Widget

是的，构件（widget）是一个技术术语。构件还有另一个名字——控件（control），虽然更具描述性，但不够形象。我们使用构件来定义通过 GUI（图形用户界面）与程序进行交互的形式。我们的 Widget 接口类如下：

```
class Widget {
    // Widget 是一个 FI-widget 的句柄——不是 FI_widget
    // 我们尽量使接口类独立于 FLTK
public:
    Widget(Point xy, int w, int h, const string& s, Callback cb);

    virtual void move(int dx,int dy);
    virtual void hide();
    virtual void show();
    virtual void attach(Window&) = 0;
```

```

    Point loc;
    int width;
    int height;
    string label;
    Callback do_it;
protected:
    Window* own; // 每个 Widget 都属于一个 Window
    Fl_Widget* pw; // 连接到 FLTK Widget
};

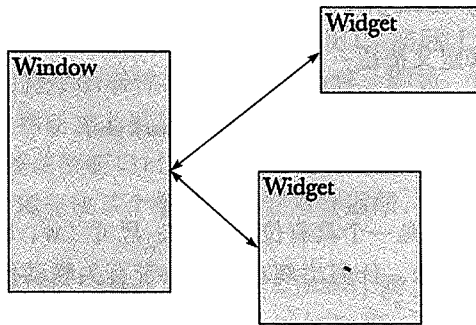
```

Widget 有两个有趣的函数，我们可以将它们用在 Button（以及很多其他从 Widget 派生出来的类，例如 Menu，参见 21.7 节）上：

- hide() 使该 Widget 对象不可见。
- show() 使该 Widget 对象再次可见。

一个 Widget 对象开始是可见的。

如同 Shape 对象一样，我们可以在 Window 中移动（move()）一个 Widget 对象，不过在进行该操作之前必须把它添加（attach()）到 Window 对象中。注意，我们将 attach() 声明为一个纯虚函数（见 19.3.5 节）：每一个派生自 Widget 的类必须定义它自己的 attach() 函数。事实上，系统级构件是在 attach() 函数中创建的。作为 Window 自己的 attach() 函数实现的一部分，构件的 attach() 函数是由 Window 对象调用的。实际上，连接窗口和构件如同跳双人舞蹈，各自必须完成自己的那部分工作。最终结果就是一个窗口知道它所包含的构件，而每个构件也知道它所属的窗口：



注意，一个 Window 对象并不知道它所处理的 Widget 的类型。如 19.4 节中描述的那样，我们使用面向对象程序设计来保证 Window 可以处理每种类型的 Widget。同样，一个 Widget 对象也不知道它所属的 Window 的类型。

这个实现还不够完美，因为我们将数据成员定义为外部可访问的。而成员 own 和 pw 是严格用于派生类实现的，所以我们将它们声明为 protected。

在 GUI.h 中可以找到 Widget 类以及我们所使用的具体构件类（Button、Menu 等）的定义。

## 21.4.2 Button

Button 是我们处理的最简单的 Widget，当我们点击按钮的时候，其功能就是调用一个回调函数：

```

class Button : public Widget {
public:
    Button(Point xy, int ww, int hh, const string& s, Callback cb)
        :Widget(xy,ww,hh,s,cb) {}

    void attach(Window& win);
};

```

这就是全部代码。attach() 函数包含所有（相对地）繁琐的 FLTK 代码。我们把相关解释留在附录 E 中（在读完第 12、13 章之前不要去阅读）。现在，你只需要知道定义一个 Widget 并不是特别困难。

我们并不处理按钮（或其他 Widget）的外观，这个问题有些复杂和棘手。问题在于，对于外观风格我们几乎有无限多种选择，而有些风格是系统强制要求的。而且，从程序设计技术的角度来看，呈现不同的按钮外观并不需要任何新知识。如果这令你失望的话，你可以注意这样一个事实，将一个 Shape 对象放置在一个按钮上面，并不会对按钮的功能造成任何影响，而且你已经知道了如何生成任何需要的形状。

### 21.4.3 In\_box 和 Out\_box

我们提供了两种 Widget 用于文本输入 / 输出：

```

struct In_box : Widget {
    In_box(Point xy, int w, int h, const string& s)
        :Widget(xy,w,h,s,0) {}
    int get_int();
    string get_string();

    void attach(Window& win);
};
struct Out_box : Widget {
    Out_box(Point xy, int w, int h, const string& s)
        :Widget(xy,w,h,s,0) {}
    void put(int);
    void put(const string&);

    void attach(Window& win);
};

```

In\_box 能够接受输入给它的文本，我们可以使用 get\_string() 将文本作为字符串读出或者使用 get\_int() 将其作为整数读出。如果你想知道是否已经有文本输入，可以使用 get\_string() 读取并检查是否得到了空字符串：

```

string s = some_inbox.get_string();
if (s == "") {
    // 处理缺失的输入
}

```

Out\_box 用来向用户呈现信息。与 In\_box 类似，我们可以使用 put() 输出字符串或者整数。21.5 节给出了使用 In\_box 和 Out\_box 的例子。

我们并没有提供 get\_floating\_point(), get\_complex() 等函数。但不必为此担心，因为你可以获得字符串，将它放入一个 stringstream 中，就可以按你喜欢的方式做任意的格式化输入了（见 11.4 节）。

### 21.4.4 Menu

我们提供了一个非常简单的“菜单”的概念：

```
struct Menu : Widget {
    enum Kind { horizontal, vertical };
    Menu(Point xy, int w, int h, Kind kk, const string& label);
    Vector_ref<Button> selection;
    Kind k;
    int offset;
    int attach(Button& b);           // 将 Button 添加到 Menu
    int attach(Button* p);           // 添加新的 Button 到 Menu

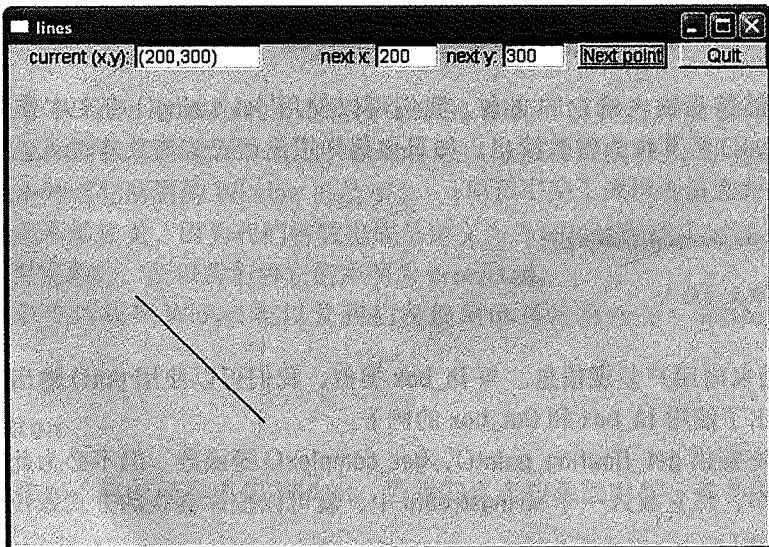
    void show()                       // 显示所有的 Button
    {
        for (Button& b : selection) b.show();
    }
    void hide();                       // 隐藏所有的 Button
    void move(int dx, int dy);        // 移动所有的 Button

    void attach(Window& win);        // 将所有的 Button 添加到窗口 win 中
};
```

Menu 本质上是一个按钮向量。跟以前一样，Point 对象 xy 指出左上角的位置。宽度和高度的作用是，当添加按钮至菜单的时候，用来重设按钮大小。参见 21.5 节和 21.7 节的例子。每一个菜单按钮（“菜单项”）是一个独立的 Widget，作为 attach() 的参数提供给 Menu。接着，Menu 提供一个 attach() 操作将所有 Button 添加到 Window 对象。Menu 对象使用 Vector\_ref（见 18.10 节和附录 E.4）跟踪它的所有 Button。如果想要一个“弹出式”菜单，你只能自己创建一个，参见 21.7 节。

### 21.5 一个实例

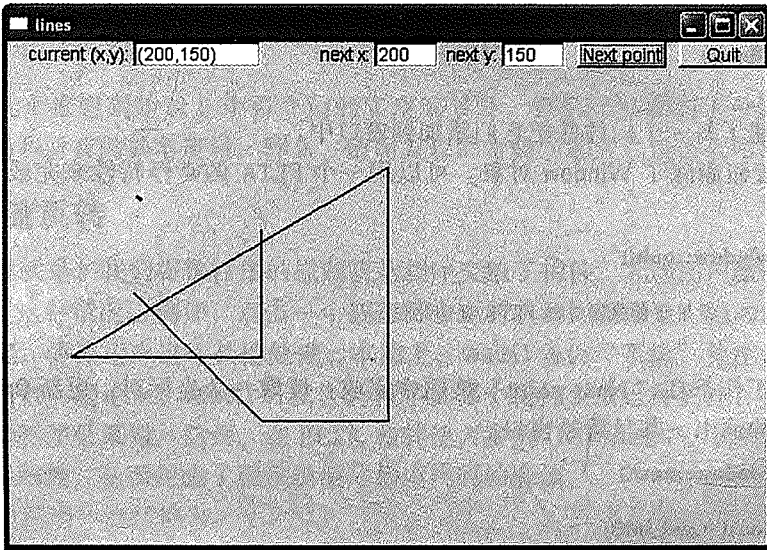
为了更好地感受基本的 GUI 工具，我们给出了一个简单的应用程序，它是一个包含输入、输出和一些图形的窗口：



这个程序允许用户绘制一系列由坐标对指定的线段（开放多线段，参见 18.6 节）。使用方法是用户反复在“next x”和“next y”框中输入  $(x, y)$  坐标对；每输入一个坐标对就点击一次“Next point”按钮。

开始时，“current(x, y)”框是空的，程序等待用户输入第一个坐标对。用户输入后，起点出现在“current(x, y)”框中，每次输入的新坐标都会用来绘制一条线：一条从当前点（显示在“current(x, y)”框中的坐标）到新输入点  $(x, y)$  之间的线，然后  $(x, y)$  就成为新的当前点。

这样就能够绘制一条开放多线段，完成之后用户可以通过“Quit”按钮退出。整个过程非常直截了当，同时该程序还展示了几个有用的 GUI 特性：文本输入 / 输出、线的绘制和多按钮。上面的窗口显示了输入两个坐标对之后的结果，输入 7 个坐标对则可得：



让我们用下面的代码来定义一个表示这种窗口的类。代码很直接：

```
struct Lines_window : Window {
    Lines_window(Point xy, int w, int h, const string& title);
    Open_polyline lines;
private:
    Button next_button;    // 将 (next_x, next_y) 添加到线
    Button quit_button;
    In_box next_x;
    In_box next_y;
    Out_box xy_out;

    void next();
    void quit();
};
```

代码中使用 `Open_polyline` 对象表示线。代码还声明了按钮和文本框（分别名为 `Button`、`In_box` 和 `Out_box`），并且为每个按钮声明了一个实现其功能的成员函数。我们决定不用“样板式的”回调函数，而是使用 `lambda` 表达式。

`Line_window` 的构造函数负责初始化所有成员：



`Lines_window`'s constructor initializes everything:

```
Lines_window::Lines_window(Point xy, int w, int h, const string& title)
:Window(xy,w,h,title),
next_button(Point{x_max()-150,0}, 70, 20, "Next point",
[] (Address, Address pw) {reference_to<Lines_window>(pw).next();},
quit_button(Point{x_max()-70,0}, 70, 20, "Quit",
[] (Address, Address pw) {reference_to<Lines_window>(pw).quit();},
next_x(Point{x_max()-310,0}, 50, 20, "next x:"),
next_y(Point{x_max()-210,0}, 50, 20, "next y:"),
xy_out(Point{100,0}, 100, 20, "current (x,y):")
{
    attach(next_button);
    attach(quit_button);
    attach(next_x);
    attach(next_y);
    attach(xy_out);
    attach(lines);
}
```

也就是说，创建了每一个构件并把它们添加到窗口中。

“Quit”按钮删除了 Window 对象。可以用一个 FLTK 的奇怪特性来完成——简单地隐藏窗口。

```
void Lines_window::quit()
{
    hide();    // 用于删除窗口的 FLTK 奇怪特性
}
```

所有实际工作都在“Next point”按钮中完成：读取一个坐标对，更新 `Open_polyline` 对象，更新位置的输出，并且重绘窗口。

```
void Lines_window::next()
{
    int x = next_x.get_int();
    int y = next_y.get_int();
    lines.add(Point(x,y));

    // 更新当前的位置读数
    ostringstream ss;
    ss << '(' << x << ', ' << y << ')';
    xy_out.put(ss.str());

    redraw();
}
```

这段代码很容易理解。我们用 `get_int()` 从 `In_box` 得到整数坐标值；用一个 `ostringstream` 对象来格式化要输出到 `Out_box` 的字符串；`str()` 成员函数负责从 `ostringstream` 对象中读取字符串；`redraw()` 函数将最终结果显示给用户；直到 `Window` 的 `redraw()` 函数被调用之前，屏幕上一直显示的是旧图像。

那么，这个程序有什么奇怪和不同之处呢？让我们看看它的 `main()` 函数：

```
#include "GUI.h"

int main()
try {
    Lines_window win (Point{100,100},600,400,"lines");
```

```

    return gui_main();
}
catch(exception& e) {
    cerr << "exception: " << e.what() << '\n';
    return 1;
}
catch (...) {
    cerr << "Some exception\n";
    return 2;
}

```

这里基本没有做任何事情！`main()` 函数体只是定义了窗口 `win` 并调用函数 `gui_main()`。这里并没有其他的函数，没有任何我们在第 6、7 章中见过的 `if`、`switch` 或者循环等代码，仅仅是一个变量的定义和对函数 `gui_main()` 的调用，而 `gui_main()` 本身也只是调用 FLTK 的 `run()` 函数。更进一步，我们会发现 `run()` 函数也只是一个简单的无限循环：

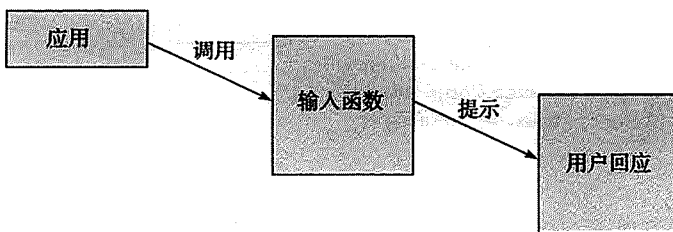
```
while(wait());
```

除了少数几个推迟到附录 E 中再介绍的实现细节外，我们已经看到了令画线程序运行起来的所有代码和所有的基本逻辑。那么，前面发现的奇怪问题到底是怎么回事呢？

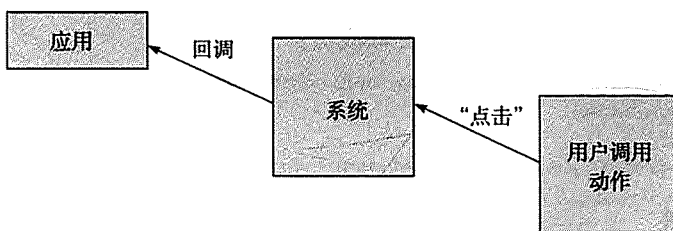
## 21.6 控制流反转


这里的关键就是，我们将执行序的控制权从程序交给了构件：无论用户激活、运行哪个构件，都会发生这种情况。例如，点击一个按钮就会运行它的回调函数。当回调函数返回以后，程序就挂起，等待用户进行其他处理。本质上，`wait()` 告诉“系统”关注这些构件并调用对应的回调函数。理论上，`wait()` 可以告诉程序员哪个构件要求这种关注，并将调用恰当函数的任务留给程序员来做。然而，在 FLTK 和其他大多数 GUI 系统中，`wait()` 只是简单地调用适当的回调函数，从而免去了程序员编写相应代码的麻烦。

一个“常规程序”的结构为：



一个“GUI 程序”的结构为：



“控制流反转”的一个含义是程序的执行顺序完全由用户的行为决定。这使得程序的组 

织和调试都更加复杂。很难猜想用户想要做什么，也很难想象一个随机回调序列可能产生的所有影响。这使得系统测试非常困难（参见第 26 章）。处理这些问题的技术已经超出了本书的讨论范围，但是我们建议你格外小心那些由用户通过回调来驱动的代码。除了明显的控制流问题之外，还有关于可见性的问题，以及跟踪构件与数据关联的困难。为了尽量减少麻烦，关键是要保证 GUI 部分的程序简洁性，同时逐步增量式构建一个 GUI 程序，并且在每一个阶段都要测试。当你编写一个 GUI 程序的时候，绘制对象及它们之间交互的图表也是很关键的。

被不同回调函数触发的代码是如何相互通信的呢？最简单的方法是处理保存在窗口中的数据，就像 21.5 节中的例子那样。在那个例子中，`Line_window` 的 `next()` 函数通过点击“Next point”按钮被调用，它从 `In_box` 读取数据 (`next_x` 和 `next_y`)，并且更新 `lines` 成员变量和 `Out_box` (`xy_out`)。显然，由回调调用的函数可以做任何事情：打开文件，连接网络等。但是，目前我们只考虑简单的情况：将数据保存在窗口中。

## 21.7 添加菜单

下面我们通过为画线程序添加菜单，来进一步研究“控制流反转”带来的控制和通信问题。首先，我们提供一个简单的菜单，允许用户改变 `lines` 成员变量中所有线的颜色。下面我们添加 `color_menu` 菜单及其回调函数：

```
struct Lines_window : Window {
    Lines_window(Point xy, int w, int h, const string& title);

    Open_polyline lines;
    Menu color_menu;

    static void cb_red(Address, Address); // 红色按钮的回调
    static void cb_blue(Address, Address); // 蓝色按钮的回调
    static void cb_black(Address, Address); // 黑色按钮的回调

    // 动作
    void red_pressed() { change(Color::red); }
    void blue_pressed() { change(Color::blue); }
    void black_pressed() { change(Color::black); }
    void change(Color c) { lines.set_color(c); }

    // 和之前一样
};
```

重复写出这些几乎相同的回调函数和“动作”函数非常乏味。但是，这在概念上比较简单，而且那些在输入方面更加简单的方式也超出了本书的讨论范围。如果你愿意，可以使用 `lambda` 表达式代替 `cb_` 函数。当一个菜单按钮被按下时，它会将线改为所要求的颜色。

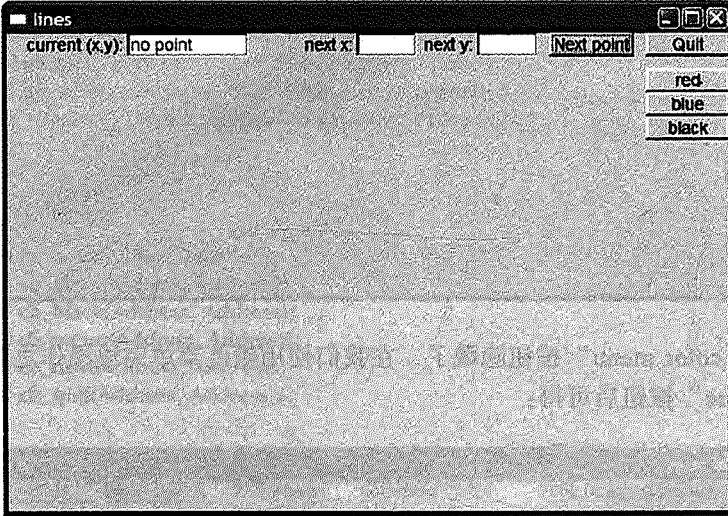
我们需要初始化已经定义的 `color_menu` 成员：

```
Lines_window::Lines_window(Point xy, int w, int h, const string& title)
    :Window(xy,w,h,title),
    // 和之前一样
    color_menu(Point(x_max()-70,40),70,20,Menu::vertical,"color")
{
    // 和之前一样
    color_menu.attach(new Button(Point{0,0},0,0,"red",cb_red));
    color_menu.attach(new Button(Point{0,0},0,0,"blue",cb_blue));
    color_menu.attach(new Button(Point{0,0},0,0,"black",cb_black));
```

```
attach(color_menu);
```

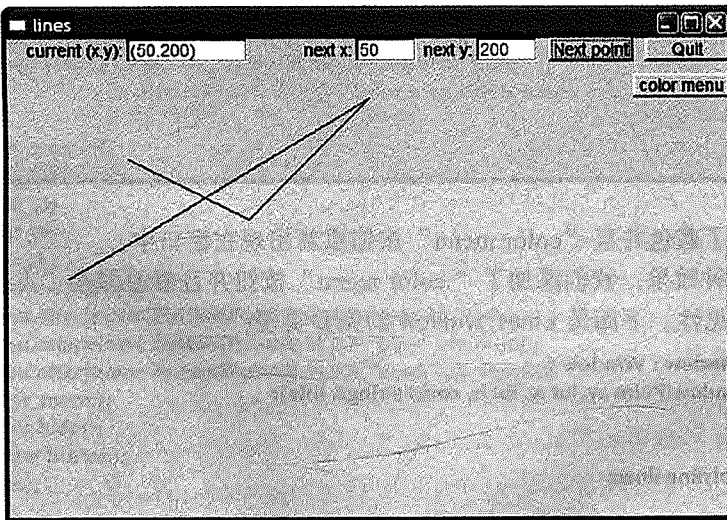
```
}
```

这些按钮是动态添加到菜单上的（使用 `attach()`），并且可以根据需要移除和替换它们。`Menu::attach()` 调整按钮的尺寸和位置，并将它们添加到窗口中。这就是这段代码的全部工作，结果如下：

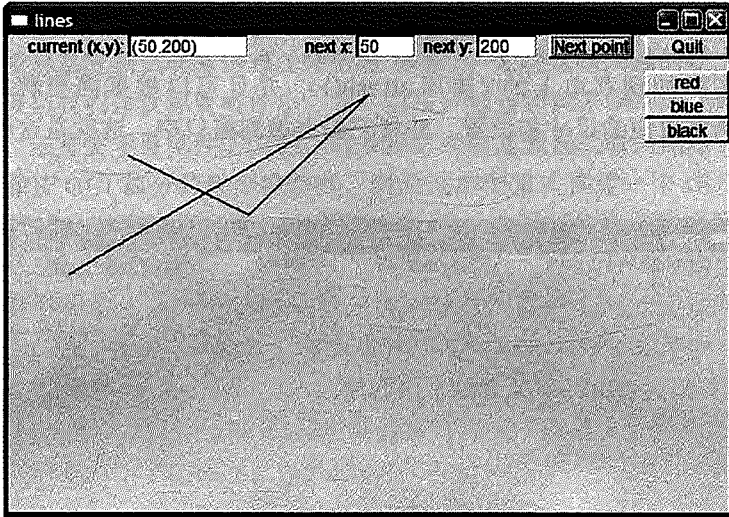


程序运行了一段时间之后，我们发现真正需要的是一个“弹出式”菜单；也就是说，我们不希望把稀有的屏幕空间用在一个菜单上，除非我们正在使用它。因此，我们添加一个“color menu”按钮。当我们按下它的时候，弹出颜色菜单，并且在完成一个选择操作之后，菜单重新隐藏起来，而“color menu”按钮再次出现在窗口中。

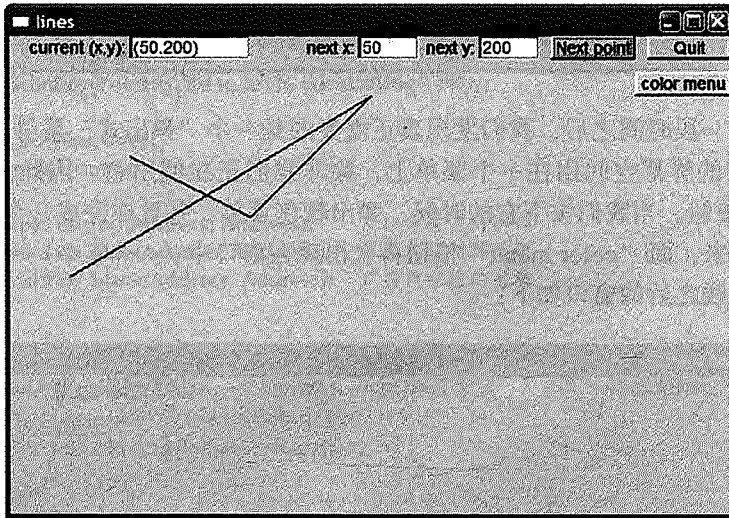
添加了几条线之后的窗口如下：



我们看到了新的“color menu”按钮和一些（黑色的）线。点击“color menu”按钮会出现颜色菜单：



注意，这时候“color menu”按钮隐藏了，在我们使用颜色菜单完成操作之前并不需要这个按钮。点击“blue”按钮后可得：



现在，线都变成了蓝色并且“color menu”按钮重新出现在窗口中。

为了达到这种效果，我们添加了“color menu”按钮并且修改那些“点击”函数来调整菜单和按钮的可见性。下面是 Lines\_window 的完整实现：

```

struct Lines_window : Window {
    Lines_window(Point xy, int w, int h, const string& title);
private:
    // 数据：
    Open_polyline lines;

    // 构件：
    Button next_button;    // 将点 (next_x, next_y) 添加到线
    Button quit_button;   // 结束程序
    In_box next_x;

```

```

In_box next_y;
Out_box xy_out;
Menu color_menu;
Button menu_button;

void change(Color c) { lines.set_color(c); }

void hide_menu() { color_menu.hide(); menu_button.show(); }
// 由回调函数激活的动作
void red_pressed() { change(Color::red); hide_menu(); }
void blue_pressed() { change(Color::blue); hide_menu(); }
void black_pressed() { change(Color::black); hide_menu(); }
void menu_pressed() { menu_button.hide(); color_menu.show(); }
void next();
void quit();

// 回调函数
static void cb_red(Address, Address);
static void cb_blue(Address, Address);
static void cb_black(Address, Address);
static void cb_menu(Address, Address);
static void cb_next(Address, Address);
static void cb_quit(Address, Address);
};

```

注意，除了构造函数以外，其他成员都是私有的。本质上，这个窗口类就是完整程序。所有工作都是通过它的回调函数完成的，因此不需要窗口之外的任何代码。我们将声明进行了简单排列，以使类更具有可读性。构造函数提供了所有子对象的参数并将这些对象添加到窗口中：

```

Lines_window::Lines_window(Point xy, int w, int h, const string& title)
:Window(xy,w,h,title),
next_button(Point{x_max()-150,0}, 70, 20, "Next point", cb_next),
quit_button(Point{x_max()-70,0}, 70, 20, "Quit", cb_quit),
next_x(Point{x_max()-310,0}), 50, 20, "next x:"),
next_y(Point{x_max()-210,0}), 50, 20, "next y:"),
xy_out(Point{100,0}, 100, 20, "current (x,y):"),
color_menu(Point{x_max()-70,30},70,20,Menu::vertical,"color"),
menu_button(Point{x_max()-80,30}, 80, 20, "color menu", cb_menu)
{
attach(next_button);
attach(quit_button);
attach(next_x);
attach(next_y);
attach(xy_out);
xy_out.put("no point");
color_menu.attach(new Button(Point{0,0},0,0,"red",cb_red));
color_menu.attach(new Button(Point{0,0},0,0,"blue",cb_blue));
color_menu.attach(new Button(Point{0,0},0,0,"black",cb_black));
attach(color_menu);
color_menu.hide();
attach(menu_button);
attach(lines);
}

```

注意，初始化的顺序和数据成员的声明顺序是一致的，这是书写初始化代码的正确顺序。事实上，成员初始化的执行顺序总是按照成员声明的顺序。如果一个基类或者成员的构

造函数的次序不对，一些编译器会给出警告信息。

## 21.8 调试 GUI 代码

一旦 GUI 程序开始工作，通常就很容易调试了：因为你看到的就是你得到的。然而，在第一个形状和构件开始出现在窗口中，甚至是窗口出现在屏幕上之前，通常会是一个充满挫折的阶段。试试下面的 `main()` 函数：

```
int main()
{
    Lines_window {Point{100,100},600,400,"lines"};
    return gui_main();
}
```

你看到错误了吗？无论你是否看到了，你都应该试一试；程序可以编译通过并运行，但是 `Line_window` 并未给你划线的机会，你能得到的最多是屏幕上的一闪而已。如何从这样的程序中找到错误呢？

- 小心使用经过严格验证的程序组件（类、函数、库）。
- 简化所有的新代码，降低程序从最简版本“增长”的速度，仔细逐行检查代码。
- 检查所有的链接设置。
- 与已经正常运行的程序比较。
- 向朋友解释代码。

⚠ 你会发现很难跟踪代码的执行过程。如果你已经学会了使用调试器，你可能还有机会，但在这种情况下，只是加入“输出语句”将不再有效——因为根本看不到任何输出。即使是使用调试器也有问题，因为有很多程序在同时运行（“多线程”）——你的代码并不是唯一试图和屏幕交互的代码。代码简化和理解代码的系统方法是关键。

那么前面 `main()` 函数的问题出在哪儿呢？下面给出了正确的版本（来自 21.5 节）：

```
int main()
{
    Lines_window win{Point{100,100},600,400,"lines"};
    return gui_main();
}
```

我们“忘记”了 `Lines_window` 的名字 `win`。因为我们实际上并不需要这个名字，这看起来是合理的，但是编译器会认为既然我们不再使用这个窗口，那就要立即销毁它。天啊！这个窗口的生命期仅仅是毫秒级。出现前面的结果就不足为奇了。

⚠ 另一个常见的问题是将一个窗口恰好放在了另一个窗口上面，明显（或者不是非常明显）看起来就像只有一个窗口。另一个窗口到哪儿去了呢？我们很可能会寻找这种代码中并不存在的错误，而浪费宝贵的时间。如果我们把一个形状放在了另一个形状上面，也可能会出现同样的问题。

⚠ 还有，可能会让事情更糟的是，当我们使用 GUI 库的时候，异常并不总是如我们希望的那样正常工作。因为我们的代码由 GUI 库控制，我们抛出的异常可能永远不会到达处理程序，GUI 库或者操作系统可能会“吃掉”它（也就是说，它们可能依赖于不同于 C++ 异常的错误处理机制，可能完全忽略了 C++）。

☛ 在调试中经常发现的问题包括：由于 `Shape` 和 `Widget` 对象没有被添加到窗口中而没有显示；由于超出对象的作用域导致出错。考虑程序员如何在菜单中创建并添加按钮：

```

// 将按钮装载到一个菜单所需的辅助函数
void load_disaster_menu(Menu& m)
{
    Point orig {0,0};
    Button b1 {orig,0,0,"flood",cb_flood};
    Button b2 {orig,0,0,"fire",cb_fire};
    // ...
    m.attach(b1);
    m.attach(b2);
    // ...
}
int main()
{
    // ...
    Menu disasters {Point{100,100},60,20,Menu::horizontal,"disasters"};
    load_disaster_menu(disasters);
    win.attach(disasters);
    // ...
}

```

上面的代码不能正常运行。所有按钮都是 `load_disaster_menu()` 函数内的局部变量，将它们添加到菜单上并不会改变这一点。在 13.6.4 节中可以找到这一问题的解释（不要返回指向局部变量的指针），而 8.5.8 节说明了局部变量的内存布局情况。这里的关键是，在 `load_disaster_menu()` 函数返回之后，那些局部对象就已经被销毁了，`disasters` 菜单将指向不存在（销毁了）的对象。结果肯定是错误并且让人吃惊的。解决方法是使用 `new` 创建的未命名对象而不是命名的局部对象：

```

// 将按钮装载到一个菜单所需的辅助函数
void load_disaster_menu(Menu& m)
{
    Point orig {0,0};
    m.attach(new Button{orig,0,0,"flood",cb_flood});
    m.attach(new Button{orig,0,0,"fire",cb_fire});
    // ...
}

```

正确方法甚至比（非常常见的）错误方法更加简单。

## 简单练习

1. 使用 FLTK 的链接程序设置（如附录 D 中的描述）建立一个新项目。
2. 使用 `Graph_lib` 的工具，输入 21.5 节的画线程序并运行它。
3. 使用 21.7 节中描述的那种“弹出式”菜单修改程序并运行它。
4. 再次修改这个程序，添加第二个菜单用于选择线型，并运行它。

## 思考题

1. 你为什么需要图形用户界面？
2. 什么时候你需要非图形用户界面？
3. 什么是软件的层次结构？
4. 为什么需要对软件分层？
5. C++ 程序与操作系统通信时的基本问题是什么？
6. 什么是回调函数？



7. 什么是构件?
8. 构件的另一个名称是什么?
9. 首字母缩写词 FLTK 是什么意思?
10. FLTK 如何发音?
11. 你还听说过其他的 GUI 工具集吗?
12. 哪些系统使用术语构件? 哪些系统更喜欢使用控件一词?
13. 构件的例子有哪些?
14. 什么时候需要使用输入框?
15. 输入框中的值是什么类型的?
16. 什么时候需要使用按钮?
17. 什么时候需要使用菜单?
18. 什么是控制流反转?
19. 调试 GUI 程序的基本策略是什么?
20. 为什么调试 GUI 程序比调试“普通的流式输入输出程序”更难?

## 术语

button (按钮)	dialog box (对话框)	visible/hidden (可见 / 隐藏)
callback (回调)	GUI	waiting for input (等待输入)
console I/O (控制台 I/O)	menu (菜单)	wait loop (等待循环)
control (控件)	software layer (软件层次)	widget (构件)
control inversion (控制流反转)	user interface (用户接口)	

## 习题

1. 创建一个 `My_window` 类, 它和 `Simple_window` 有些相似, 除了多两个按钮——`next` 和 `quit`。
2. 创建一个带有  $4 \times 4$  的正方形按钮方阵的窗口 (基于 `My_window`)。当按下按钮时执行一个简单的动作, 比如在一个输出框中打印它的坐标, 或者变换为一个稍微不同的颜色 (直到按下另一个按钮)。
3. 在按钮 (`Button`) 上放置一个图像 (`Image`), 当按下按钮时移动按钮和图像。使用下面的随机数发生器为“图像按钮”选择新位置:

```
#include<random>

inline int rand_int(int min, int max)
{
    static default_random_engine ran;
    return uniform_int_distribution<>(min,max)(ran);
}
```

它返回一个  $[min, max)$  内的随机整数 (`int`)。

4. 创建一个菜单, 包括绘制圆、绘制正方形、绘制等边三角形、绘制六边形等菜单项。创建一个 (或两个) 输入框用于输入坐标对, 将点击某个菜单项后绘制的形状放置在这个坐标指定的位置上。抱歉, 没有拖放功能。

5. 编写程序绘制一个你选择的形状，并在每次点击“Next”时将其移动到一个新的位置。这个新位置根据从输入流读取的一个坐标对决定。
6. 编写一个“模拟时钟”，即一个带有转动指针的时钟。通过一个库函数调用从操作系统获取时间值。这个练习的主要工作是找到能够获取时间的函数，找到等待一小段时间（比如1秒）的方法，以及根据你找到的文档学会使用它们。提示：`clock()`、`sleep()`。
7. 使用前面练习中的技术，创建一个飞机图像并让它在窗口中“飞来飞去”。窗口有一个“Start”按钮和一个“Stop”按钮。
8. 编写一个货币换算器。在启动时从一个文件读取汇率。在输入窗口中输入数额，然后提供一种选择需换算的两种货币类型的方式（例如使用两个菜单）。
9. 修改第7章的计算器，令它从输入框获取输入，并将结果返回到输出框。
10. 编写一个程序，可以让用户从一组数学函数（例如 `sin()` 和 `log()`）中进行选择，并为这些函数提供参数，然后将它们图形化显示出来。

## 附言

GUI 是一个庞大的主题，它的很多内容与已有系统的风格和兼容性有关。而且，我们必须处理种类过于繁多的构件（例如 GUI 库提供许多不同的按钮风格），可能植物学家对此会感到更亲切些。然而，其中只有很少一部分与重要的程序设计技术有关，所以我们不需要在这方面进行研究。其他的主题如按比例缩放、旋转、变形、三维对象、阴影等，涉及很多图形学和数学方面的话题，而我们在本章中并未讨论这些内容。

你必须要知道的一件事情是，多数 GUI 系统都提供了一个“GUI 程序生成工具”，你可以在图形方式下设计你的窗口布局，将回调和动作函数与按钮、菜单等关联起来。对许多应用程序来说，这样一个 GUI 生成工具有助于减少那些编写“框架式代码”（例如我们的回调函数）的乏味工作。然而，我们应该尝试理解程序是如何运行的。有时，这类工具生成的代码与你本章中看到的代码相同。有时，这类代码可能会使用一些更加精致和 / 或代价更高的机制。

# 理念和历史

如果某人说，“我想要这样一种程序设计语言，我只需说出我希望做什么，它就能帮我完成。”那么就给他一个棒棒糖吧。

——Alan Perlis

本章非常简要、有选择性地介绍了程序设计语言的历史及其设计理念。这种理念和表达它的语言是达到专业水平的基础。由于本书使用 C++ 语言，因此我们主要关注 C++ 以及影响 C++ 的其他语言。本章的目的是，对本书所介绍的设计理念给出其背景和发展前景。对每种语言，我们会介绍其设计者：一种语言不仅仅是一种抽象的创造，还是一个具体的解决方案——是对实践中所遇到的问题的回应。

## 22.1 历史、理念和专业水平

✘ “历史是一堆废话”这是亨利·福特的名言。然而在很久以前，一个相反的观点就被广泛引用了：“不能记住历史的人注定要重复历史。”这里的关键问题是，我们应该选择了解哪一部分历史，又该摒弃哪一部分。“95% 的事情都是无用的”是另一个相关的论调（虽然我们认为 95% 可能是一个低估的数字）。对于历史和当前实践的关系，我们的观点是，如果对历史没有一定的理解，就不可能达到专业水平。如果你几乎不了解你所在领域的背景，你就很容易被蒙蔽，历史中有太多这样的例子，任何领域都充满了似是而非而又没有实际作用的内容。历史的真正意义在于那些已经在实践中证明自身价值的思想和理念。

△ 我们乐于探讨很多程序设计语言和软件（如操作系统、数据库、图形软件、互联网软件、脚本等等）中关键性思想的起源，但你不得不在其他地方查找这些重要且有用的软件和程序设计领域。我们的篇幅甚至不够（仅仅是）揭开程序设计语言历史和理念的面纱。

☛ 程序设计的最终目标一定是生成有用的系统，人们在热烈讨论程序设计技术和语言时，常常会忘记这一点。千万不要忘记它！如果你需要提醒，那么请重新阅读第 1 章。

### 22.1.1 程序设计语言的目标和哲学

✘ 程序设计语言是什么？程序设计语言应该为我们做什么？“程序设计语言是什么”的常见答案包括：

- 指示机器操作的一种工具；
- 算法的符号表示法；
- 与其他程序员交流的工具；
- 进行实验的工具；
- 控制电脑设备的一种手段；
- 表示各种概念关系的一种方法；
- 表达高层设计的一种方法。

而我们的答案是“以上答案都对，而且还有其他的答案！”显然，我们首先要考虑那些

应用于常见领域的程序设计语言，这是贯穿本章的内容。此外，还有一些专用语言和应用于特定领域的语言，这些语言应用面比较窄并且通常有着更准确定义的目的。

我们希望程序设计语言具有哪些特性呢？

- 可移植性。
- 类型安全。
- 定义准确。
- 高性能。
- 简明表达思想的能力。
- 易调试。
- 易测试。
- 能访问所有系统资源。
- 平台独立性。
- 可运行在所有平台上（例如 Linux、Windows、智能手机、嵌入式系统等）。
- 长期稳定性。
- 能针对应用领域的变化做适当的改变。
- 易于学习。
- 轻量级。
- 支持流行的程序设计模式（例如面向对象程序设计和泛型程序设计）。
- 有利于程序分析。
- 提供大量工具。
- 大规模社群的支持。
- 适合初学者（如学生、自学者）学习。
- 为专业人员（如建筑工程师）提供全面的工具。
- 有大量软件开发工具可选用。
- 有大量软件组件（如库）可选用。
- 被一个开放的软件社群所支持。
- 被主要的平台厂商所支持（微软、IBM 等）。

不幸的是，我们不能同时拥有所有这些特性。这非常令人失望，因为客观地说每一种特性都很好：它们都能为程序设计提供帮助，没有提供这些特性的程序语言会给程序员带来额外的工作量和复杂性。我们不能同时拥有这些特性的原因很简单：有一些特性是相互排斥的。例如，你不可能在拥有 100% 的平台独立性的同时，还能访问到系统的所有资源；如果一个程序访问的某种资源并不是每个平台都会提供，那么这个程序就不可能在所有平台都能运行。与之类似，我们非常希望一种语言（包括它的工具和库）是轻量级的并且容易学习，但是这样就不可能为所有系统和应用领域都提供全面的支持。

这就是设计理念的重要之处。对语言、库、工具以及程序的设计者，这些理念指导他们对技术进行选择和取舍。没错，当你编写程序的时候，你就是一个设计者，必然要做出设计上的选择和取舍。

## 22.1.2 编程理念

《C++ Programming Language》的序言中提到，“C++ 语言是一种通用程序设计语言，

它的一个主要设计目的就是让那些认真严肃的程序员也能体验到程序设计的乐趣”。这是什么意思？程序设计不就是生产产品么？不就是正确性、质量和可维护性吗？不就是上市日期吗？不就是效率么？就是对软件工程的支持吗？当然，这些说法都没错，但是我们不能忘记程序员——也就是人。考虑另外一个例子，Don Knuth 说过，“Alto 最好的特性就是它不会在晚上运行得更快”。Alto 是来自施乐 Palo Alto 研究中心（PARC）的一台计算机，是最早的“个人计算机”之一。而当时的主流计算机是与之相对的“分时共享计算机”，在白天会有大量用户竞争访问计算机（因而晚上会运行更快）。

✂ 程序设计工具和技术存在的意义是为了让一个程序员——一个人，能更好地工作并创造出更好的成果。请不要忘记这一点。那么，什么样的指导方针可以帮助程序员以最小的代价设计出最好的软件呢？本书自始至终都在阐述我们对此的理念，因此本节只是对这些内容做一个总结。

✂ 我们希望自己的代码有良好架构的主要原因是，在良好架构下，我们可以不必花费很大力气就能修改程序。架构越好，修改程序、寻找和修正错误、增加新特性、移植到新的体系结构中以及优化性能等等工作就更容易。这就是我们所说的“良好”的准确含义。

在本节的剩余部分，我们将：

- 重新审视我们尝试达到的目标，也就是我们想从代码中得到什么。
- 提出两种一般性的软件开发方法，并说明两者的结合比单独使用其中任何一种方法都要更好。
- 思考用代码表达程序结构的关键问题：
  - 直接表达思想；
  - 抽象层次；
  - 模块化；
  - 一致性和简约主义。

✂ 理念是要拿来用的。它是思考的工具，而不仅仅是用来取悦管理人员和考核人员的华丽词汇。我们所编写的程序应该尽可能接近我们的设计理念。当陷入程序泥潭的时候，我们最好回过头来看一看，问题是否出在违背了设计理念。有时这是很有帮助的。当评估一个程序时（最好是在交付用户之前），我们应该寻找那些违背设计理念的部分，这些部分是将来最有可能出问题的地方。应该尽可能广泛地应用设计理念，但也要考虑实践相关问题（例如性能和简单性）和语言的弱点（不存在完美的语言），这些因素会阻碍我们得到符合设计理念的完美结果。

☛ 设计理念可以指导我们做出具体的技术决策。例如，我们不能孤立地对一个库的每个接口都做出决策（见 19.1 节），否则，得到的结果将是个灾难，正确的方法是：回到我们的基本原则，首先确定对于这个特定的库来说什么是最重要的，然后设计一套一致的接口集合。理想情况下，我们应该在文档和代码注释中清楚地描述出这个特定设计方案所遵循的设计原则及其中的折中选择。

☛ 在一个项目的开始，首先应该回顾设计理念，找出它与待解决问题及其解决方案最初思路的相关之处。这是获得和优化设计思路的好方法。随后在设计 and 开发过程中，当你陷入困境时，回过头来查看一下程序中哪个部分偏离设计理念最远——这些就是最有可能隐藏错误、出现设计缺陷的地方。与“在相同的地方反复查看、用相同技术反复寻找错误”的基本调试技术相比，这种方法提供了另一种调试途径。“错误总是存在于你没有查看到的地方——否则你早就找到它了。”

### 22.1.2.1 我们需要的是什么

典型情况下，我们需要：

- 正确性：是的，定义什么是“正确的”非常困难，但这却是完成工作的重要一步。通常，对于一个给定项目，别人会为我们给出正确性的定义，但是接下来我们还必须要理解其含义。
- 可维护性：每个成功的程序都会随着时间的推移而修改；它可能会被移植到新的硬件或软件平台上，可能添加一些新的功能，或者需要修改新发现的错误。下面一节关于程序结构理念的内容就讨论了可维护性。
- 性能：性能（“效率”）是一个相对的概念。性能必须与程序的用途相适应。有一种常见的观点：高效的代码必然是低层的，结构良好的高层代码会导致低效。而我们的经验恰恰相反，达到满意性能的途径通常是遵循我们所推荐的理念和方法。例如，STL 就是一个同时兼顾抽象和高效的代码。执迷于低层细节与不屑于低层细节一样容易导致糟糕的性能。
- 按时交付：交付给用户一个完美的程序，但时间上却延期了一年，这一般是无法接受的。显然，人们的期望常常不切实际，但是，我们必须在合理的时间内交付高质量的软件。一种观点认为“按时完成”就意味着粗制滥造，这并不是事实。相反，我们发现重视良好的结构（例如资源管理、不变式和接口设计）、设计时考虑测试、恰当地使用库（经常是为特定应用或特定领域而设计的库）是如期完工的有效方法。

上述目标要求我们关注代码结构：

- 如果程序中有错误（每一个大型程序都有错误），那么清晰的结构有助于发现错误。
- 如果需要让初学者理解程序或者需要修改程序，清晰的结构会比一大堆细节更容易理解。
- 如果程序遇到了性能问题，高层程序（更接近设计理念，并有良好的结构）比低层程序或凌乱的程序更易于性能调整。首先，高层程序更易于理解。其次，相对于低层程序，高层程序在设计早期就已经考虑测试和性能调整因素了。

请注意程序的可理解性。任何能帮助我们理解、分析程序的方法都是有益的。基本上，规律性总比不规律要好，只要这种规律性不是因为过度简化而形成的。

### 22.1.2.2 一般性的方法

编写正确的软件，有两种方法：

- 自底向上 (bottom-up)：只用已证明正确性的组件来构建系统。
- 自顶向下 (top-down)：用可能包含错误，但是能捕获所有错误的组件来构建系统。

有趣的是，大部分可靠系统都是组合这两种截然相反的方法来构造的。原因很简单：对于一个真实的大型系统而言，任何一种方法都无法提供所需的正确性、适应性和可维护性：

- 我们无法构造并“证明”足够多的基本组件来消除所有错误源。
- 当组合有错误的基本组件（库、子系统、类层次等）来构建最终系统时，我们无法完全弥补组件的缺陷。

两种方法的结合比任何一种方法单独使用都要好：我们可以实现（或借用或购买）足够好的组件，其遗留的错误可以通过错误处理机制和系统化的测试来弥补。而且，如果我们坚持构造更好的组件，就可以用它们构造出更大的组件，从而减少对“凌乱的专用代码”的需求。

测试是软件开发的重要一环，我们将在第 26 章中详细介绍。测试是一种系统化地寻找

错误的方法。“尽早测试和日常化测试”是一个流行的观点。我们在程序设计时就应考虑测试问题，努力使测试更简单，并使错误在杂乱的代码中更难以“藏身”。

### 22.1.2.3 思想的直接表达



当我们表达某事物时——不管它是高层的还是低层的——理想的情况是直接代码来表达，而不是用其他辅助方式。这一基本理念有几种不同形式：

- 用代码直接表达思想。例如，用特殊类型（如 `Month` 或 `Color`）表示参数，比一般类型（如 `int`）更好。
- 用代码独立地表达相互独立的思想。例如，除少数情况外，标准 `sort()` 算法可以对任意元素类型的任意标准容器进行排序；排序、排序标准、容器和元素类型的概念是独立的。而假如我们构造了一个“`vector`，其对象在自由空间上分配，元素类型是 `Object` 的派生类，此类定义了一个 `before()` 成员函数，供 `vector::sort()` 使用”，那么就得到了一个远不如标准 `sort()` 那么通用的 `sort()`，因为我们对存储、类层次、可用的成员函数、序等等做出了假设。
- 用代码直接表达思想之间的关系。最常见的可以直接表达的关系是继承（例如 `Circle` 是一种 `Shape`）和参数化（例如，`vector<T>` 表示所有向量都具有的共性，与特定的元素类型无关）。
- 自由组合代码表达的思想——当且仅当这种组合有意义时。例如，`sort()` 允许我们使用各种不同的元素类型和各种容器，但元素必须支持 `<`（如果不支持，我们在使用 `sort()` 时就要用一个额外的参数指定比较操作），且容器必须支持随机访问迭代器。
- 简单地表达简单的思想。遵循上述理念，会导致过度通用的代码。例如，我们可能得出超出任何人需求的过于复杂的类层次（继承结构），或者每个（明显）简单的类都设置了七个参数。为了避免每个用户都不得不面对各种可能的复杂情况，我们应尽力提供处理最普遍或者最重要的情形的简单版本。例如，除了使用 `op` 的通用排序函数 `sort(b, e, op)` 外，我们还提供隐含使用“`<`”作为比较操作的版本 `sort(b, e)`。我们还可提供使用“`<`”对于标准容器进行排序的 `sort(c)` 和使用 `op` 对标准容器进行排序的 `sort(c, op)`。

### 22.1.2.4 抽象层次



我们更愿意在尽可能高的抽象层上工作，即，我们的理念就是以尽可能一般化的方式来表达解决方案。

例如，考虑如何表示电话本的条目（就像我们在 PDA 或手机上保存它那样）。我们可以用 `vector<pair<string, Value_type>>` 来表达（姓名，值）对的集合。然而，如果我们实际上总是用姓名来访问该集合的话，`map<string, Value_type>` 是一种更高层的抽象，它可以使我们免去编写（和调试）访问函数的麻烦。另一方面，`vector<pair<string, Value_type>>` 又比使用两个数组 `string[max]` 和 `Value_type[max]` 的表示方式抽象程度更高。因为在两个数组的表示方式中，字符串和它的值的关系是隐含的。最低层次的抽象可能是一个 `int`（元素的个数）加上两个 `void*`（指向某种程序员了解却不为编译器所知的表示形式）。在我们的例子中，到目前为止介绍的每种表示方式都可认为是非常低层的，因为它们更关注值对的表示形式，而不是其功能。为了更为接近实际应用，我们可以定义一个直接反映使用方式的类。例如，我们可以设计一个 `Phonebook` 类，其接口方便使用，然后以它为基础来编写程序。这个 `Phonebook` 类可以用上述任何一种表示方法来实现。

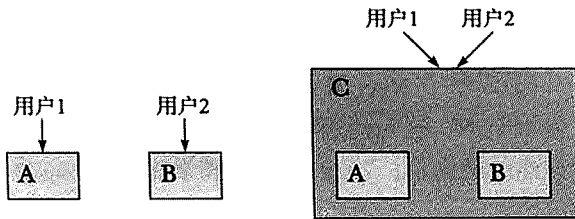
我们更喜欢较高层次的抽象（如果我们有一个适合的抽象机制，并且我们的语言对这种机制的支持效率较高的话）的原因是，比起在计算机硬件层次表达的解决方案，它更接近于我们思考问题和解决方案的方式。

对于低层抽象，人们使用它的理由往往是“效率”。但要注意，你应该仅在真正需要提高效率时才使用低层抽象（见 25.2.2 节）。而且，使用低级（更原始）语言特性不一定能获得更好的性能。相反，有时还会失去进行优化的机会，而高层程序设计则可提供优化可能。例如，使用 Phonebook 类，实现方式可以在 string[max] 加上 Value\_type[max] 和 map<string, Value\_type> 之间进行选择。对于有些应用，前者更有效，而对另一些应用则是后者更有效。当然，如果应用程序仅包含你的个人通讯录，性能不是主要考虑因素。然而，当我们必须记录和为数百万个条目的时候，这种权衡就变得很有意义了。更重要的是，如果使用低层特性，一段时间后，处理低层特性就会占用程序员绝大部分时间，以至于没有时间对程序进行改进（在性能方面或其他方面）。

22.1.2.5 模块化

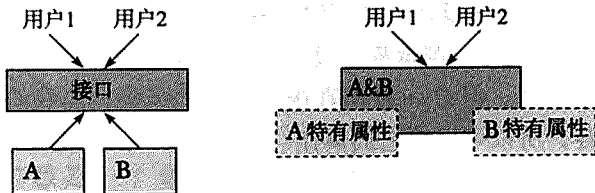
模块化是一种理念。我们希望能用“组件”（函数、类、类层次、库等等）来构建系统，这些组件可以独立构造、理解和测试。理想情况下，我们也希望每个组件都可以用于很多程序中（“重用”）。所谓重用 (reuse)，就是用以前测试过并且在其他地方已经使用过的组件构建系统，也包括组件的设计和使用等工作。在前文讨论类、类层次、接口设计和泛型程序设计时，我们已经接触过重用的概念。我们所讨论的大部分“程序设计风格”（见 22.1.3 节）都与设计、实现和使用潜在的“可重用”组件有关。请注意，并不是每个组件都能用于很多程序；一些代码过于专用，难以改进以用于其他地方。

代码中的模块化应该能反映出应用中重要的逻辑差异。我们不是简单地把两个完全无关的类 A 和 B 放在一个“可重用组件”C 中来“提高重用性”。由于需将 A 和 B 的接口合并为 C 的接口，这会使代码复杂化：



如上图所示，用户 1 和用户 2 都使用 C。除非你查看了 C 的内部，否则你可能认为两个用户从组件共享中受益了。从共享（“重用”）中获得的益处（在本例中，实际并未受益）应该包括更好的测试、更少的代码总量、更大的用户基础等。不幸的是，虽然本例有一些过度简化，但所呈现的问题并不是一个特别罕见的现象。

如何做才能解决这个问题呢？也许应该提供一个 A 和 B 的公共接口：





两个图旨在表示继承和参数化。在两种情况下，为了使重用更有价值，所提供的接口必须要比 A 和 B 的接口的简单合并更小。换句话说，A 和 B 必须要有有一个能使用户受益的最小的基本共性集合。请注意，我们又回到了接口问题（见 9.7 节和 25.4.2 节）和不变式问题（见 9.4.3 节）。

### 22.1.2.6 一致性和简约主义

一致性和简约主义是我们表达思想的最基本的理念。所以我们可能因表象而忽略它们。不过，如果一个设计已经非常杂乱，想要洗练地重新表达它确实很困难。因此，一致性和简约主义应该作为设计标准，在设计过程中就应遵循，并应该影响到哪怕最微小的程序细节：

- 如果你怀疑一个特性的功用，那么不要添加这个特性。
- 为相似的特性设计相似的接口（和名字），但有一个前提——这种相似性是根本性的。
- 为不同的特性设计不同的名字（或许接口风格也应不同），但前提是这种差异性是本性的。

一致的命名方式、接口风格和实现风格都对维护工作有帮助。当代码一致时，新程序员不需要对庞大系统的每个部分都学习一系列新的规范。STL 就是一个例子（见第 15 ~ 16 章、附录 C.4 ~ C.6）。如果不可能实现一致性（例如，程序包含古老的代码或其他语言编写的代码），一种解决方法是为这些代码设计一个与程序其他部分风格相吻合的接口。与之相对的是，不做任何特殊处理，让外来的（“陌生的”、“糟糕的”）风格“感染”要访问这些入侵代码的每个程序部分。

一种保持简约主义和一致性的方法是：仔细地（并且一贯地）为每个接口做好文档。这样，我们就有更大机会发现不一致的地方和重复的内容。做好前置条件、后置条件和不变式的文档，与仔细留意资源管理和错误报告一样，都是非常有用的。一致的错误处理和资源管理策略，对实现简洁的程序是非常必要的（见 14.5 节）。

- 对某些程序员来说，关键的设计原则是 KISS（“Keep It Simple, Stupid”，简单的才是最好的）。我们甚至听到有人声称 KISS 是唯一有价值的设计原则。然而，我们更倾向于一些不那么广为人知的原则，例如“保持事情的简单性”（Keep simple things simple）和“尽可能保持简洁，但不要过分简单化”（Keep it simple: as simple as possible, but no simpler）。后一句话引自阿尔伯特·爱因斯坦，这句话表明，超出了一定界限的过分简化是危险的，因而对设计是有害的。一个显然的疑问是：“为谁简化，和谁比较？”

### 22.1.3 风格 / 范型

- 当我们设计并实现一个程序时，应该保持统一的风格。C++ 支持四种基本的风格：

- 过程式程序设计；
- 数据抽象；
- 面向对象程序设计；
- 泛型程序设计。

这些风格有时被称作“程序设计范型”（某种程度上有些自夸）。除此之外，还有很多其他“范型”，如函数式程序设计、逻辑程序设计、基于规则的程序设计、基于约束的程序设计以及面向方面的程序设计。但 C++ 并不直接支持这些风格，而我们也无法在一本人门书籍中涵盖所有这些内容，所以可以将它们留作“未来工作”。即使我们介绍的几种范型 / 风格，也有大量细节不得不略去，都留作将来进一步的学习：

- 过程式程序设计 (procedural programming): 一种利用函数 (对参数进行操作) 构造程序的思想。例如数学库函数 `sqrt()` 和 `cos()`。C++ 通过函数的概念来支持这种风格 (见第 8 章)。这种风格最有价值的地方在于可以选择多种方式传递参数: 传值、传引用或者常量引用。在这种程序设计风格中, 数据通常被组织为数据结构 (`struct`), 而不使用显式的抽象机制 (如类的私有数据成员或成员函数)。注意, 这种程序设计风格以及函数都是其他风格不可或缺的一部分。
- 数据抽象 (data abstraction): 其思想为——首先为应用领域提供一组适合的数据类型, 然后使用这些数据类型编写程序。矩阵就是一个经典的例子 (见 24.3 ~ 24.6 节)。这种风格非常倚重显式数据隐藏 (如使用类的私有数据成员)。标准的 `string` 和 `vector` 都是典型的例子, 它们显示出了数据抽象与泛型程序设计的参数化之间的紧密联系。这种风格之所以被称作“抽象”, 是因为我们通过接口来访问数据类型, 而不是直接访问其实现。
- 面向对象程序设计 (object-oriented programming): 其思想为——将类型组织为层次结构, 以使用代码直接表达它们之间的关系。一个经典的例子是第 19 章的 `Shape` 类。如果各类型间有固有的层次关系的话, 这种风格显然是很有价值的。但它也有被滥用的趋势, 即, 人们设计类型的层次结构, 并不是基于其内在的关系。因此, 当你设计派生类的时候, 一定要问一下为什么? 你想要表达的是什么? 在你的问题中, 基类 / 派生类的差异会对你有什么帮助?
- 泛型程序设计 (generic programming): 其思想为——对于具体算法, 通过添加参数来描述算法哪些部分可以变化而不必改变其他部分, 从而将算法“提升”到更高的抽象层。第 15 章的 `high()` 是一个简单的算法提升的例子。STL 中的 `find()` 和 `sort()` 算法也是体现了泛型程序设计思想的经典算法。详细情况请参考第 15 ~ 16 章以及下面的例子。

现在把这些风格揉合在一起感受一下吧! 通常人们一提到程序设计风格 (“范型”), 都是将它们看作毫无关联的: 你要么使用泛型程序设计, 要么使用面向对象程序设计。但如果你的目标是尽可能好地表达解决方案, 就需要组合多种风格了。这里的“好”是指代码易读、易编写、易于维护以及足够高效。考虑这个源于 Simula (见 22.2.4 节) 的经典的 “Shape 例子”, 它通常被看作是面向对象程序设计的例子。第一个解决方案可能是这样的:

```
void draw_all(vector<Shape*>& v)
{
    for(int i = 0; i<v.size(); ++i) v[i]->draw();
}
```

它看起来的确是“当然的面向对象程序设计”。它主要依赖类的层次和虚函数调用为每个给定的 `Shape` 找到正确的 `draw()` 函数; 即, 对一个 `Circle`, 它调用的是 `Circle::draw()`, 而对于 `Open_polyline`, 它调用的是 `Open_polyline::draw()`。但 `vector<Shape*>` 本质上是一个泛型程序设计结构: 它依赖于编译时解析的参数 (元素类型)。为了强调这点, 我们再来看一个例子: 用简单的标准库算法来重写上面的循环。

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

`for_each()` 的第三个参数是一个函数, `for_each()` 会对序列 (由前两个参数指出, 见附录

C.5.1) 中每个元素调用该函数。现在，第三个函数调用被假定为一个使用  $f(x)$  语法调用的普通函数（或是一个函数对象），而不是一个使用  $p \rightarrow f(x)$  语法调用的成员函数。因此，我们使用标准库函数 `mem_fun()`（见附录 C.6.2）表明我们实际是希望调用一个成员函数（虚函数 `Shape::draw()`）。这里的关键点在于 `for_each()` 和 `mem_fun()` 实际上都是模板，一点也不“面向对象”，它们明显属于我们通常所说的泛型程序设计。这里更为有趣的是，`mem_fun()` 是一个返回类对象的独立（模板）函数。换句话说，它也可以轻易地被归为普通的数据抽象风格（非继承性的）甚至是过程化程序设计风格（非数据隐藏）。所以，我们可以说这行代码使用了 C++ 支持的所有四种基本风格的主要特点。

但为什么要编写第 2 个版本的“draw all Shapes”呢？它的功能与第 1 个版本基本相同，代码反而更长一些！我们可以给出的一个理由是：与 `for` 相比，用 `for_each()` 表达循环“更显然而且更不容易出错”。但对于许多人来讲，这并不那么有说服力。还有一种更好的理由：`for_each()` 表示的是要做什么（遍历序列），而不是怎样去做。但是，对大多数开发者来说，“有用”才更具说服力：第 2 个版本为我们指出了一种可以用来解决更多问题的泛化方法（泛型程序设计的优良传统）。为什么用 `vector` 而不是 `list` 或一般的序列来保存形状呢？因此，我们可以给出第 3 个版本（也是更一般的版本）：

```
template<class Iter> void draw_all(Iter b, Iter e)
{
    for_each(b,e,mem_fun(&Shape::draw));
}
```

这个版本适用于所有的形状序列。特别是，它甚至可以用于 `Shape` 数组：

```
Point p {0,100};
Point p2 {50,50};
Shape* a[] = { new Circle(p,50), new Triangle(p,p2,Point(25,25)) };
draw_all(a,a+2);
```

通过限制只能用于容器，我们还能够给出一个更简单的版本：

```
template<class Cont> void draw_all(Cont& c)
{
    for (auto& p : c) p->draw();
}
```

或者，使用 C++ 14 的一些概念（见 14.3.3 节）：

```
void draw_all(Container& c)
{
    for (auto& p : c) p->draw();
}
```

显然，这段代码仍是面向对象、泛型和过程程序设计结合的。它依赖于类层次中的数据抽象和个体容器的实现。由于没有更合适的术语，我们称这种最恰当地混合多种风格的程序设计方式为多范式程序设计（multi-paradigm programming）。但是，我更愿意将其认为是简单的程序设计：范式（paradigm）主要反映了问题求解上的限制和我们用来表达解决方案的编程语言的弱点。我预测程序设计会有一个光明的未来，在技术、编程语言和支持工具方面会有巨大进展。

## 22.2 程序设计语言历史概览

最初的程序设计就是程序员用手将 0 和 1 刻在石头上！好吧，事情并不是这样的，但也

差不了太多。在本节中，我们将从程序设计的（几乎）最初阶段讲起，快速介绍一下程序设计语言发展历史中与 C++ 程序设计相关的一些主要进展。

已有的程序设计语言实在太多了。语言以至少每十年 2000 种的速度不断被发明出来，而语言“死亡”的速度也差不多。本节主要介绍过去 60 年中出现的 10 种语言，更多信息请参考 <http://research.ihost.com/hopl/HOPL.html>。在这个网站上，你可以找到三个 AGM SIGPLAN HOPL (History Of Programming Language, 程序设计语言历史) 会议的全部论文的连接。这些论文都是经过全面的同行评阅的，比一般的网络资源更加完整和可信。本节讨论的语言都是曾在 HOPL 上进行过报告的。注意，如果你在搜索引擎中输入一篇著名论文的完整标题，你有很大机会找到文章的全文。而且，大多数计算机科学家都有自己的主页，你可以在那里找到他们的研究工作的更多相关信息。

本章对每一种语言的介绍都很简短，实际上每种语言（包括本章未提及的数百种语言）都值得用一整本书来介绍。每一种语言的内容都经过了精挑细选。我们希望你能接受这样一个挑战：对每种语言努力学习更多知识，而不是简单地认为“X 语言的全部内容不过如此而已！”请记住，本章介绍的每一种语言都是一个了不起的成就，都曾为我们的世界做出过巨大的贡献。由于篇幅所限，我们无法更全面地介绍这些语言——但总比完全不介绍要好。我们本打算为每一种语言都提供一小段代码，但很遗憾，在本章中并不适合这样做（见习题 5、6）。

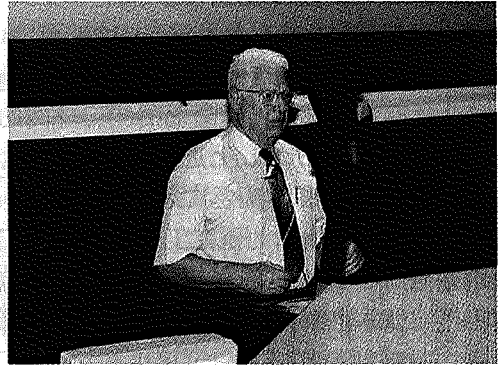
我们见过太多这样的情况：人们在介绍某种人造产品（例如一种程序设计语言）时，只是简单地介绍它是什么，或者介绍成某个匿名“开发过程”的产物。这样的介绍歪曲了历史：通常（特别是在形成早期），一种程序设计语言是理想、职业、个人偏好以及外部限制条件作用在一个或（通常是）多个人身上的结果。因此，我们强调与语言相关联的关键人物。并不是 IBM、贝尔实验室、剑桥大学等机构设计了程序语言，而是来自这些机构中的人设计了语言（通常与朋友或同事合作完成）。

请注意，有一种奇怪的现象常常扭曲我们对历史的观点。我们为那些著名的科学家或工程师树碑立传之时，都是他们已经功成名就很久之后——已经成为国家科学院院士、皇家学会院士、圣约翰爵士、图灵奖获得者等。换句话说，离他们取得最重要的成就的时间已经过去几十年了。当然，几乎所有著名的科学家和工程师都是在一生中不断地创造出专业成就。但是，当你回过头去审视你所喜欢的程序设计语言和程序设计技巧是如何产生的时候，你可以试着想象一下：一个年轻人（即使是现在，科学和工程领域中的女性仍是太少了，因此假定是一位男性）正在试图计算他是否有足够的钞票请女朋友去一个体面的餐厅吃饭；或者是一位父亲正在考虑该将一篇重要的论文提交到哪个会议上，以便这个年轻的能够顺便度个假。至于灰白的胡须、秃顶和过时的服装，那都是很久以后的事情了。

### 22.2.1 最早的程序设计语言

从 1949 年开始，当第一代“现代”贮存程序式电子计算机出现之时，它们就都具有自己的程序设计语言。当时，算法（例如行星轨道的计算）的表达和特定机器的指令间是一一对应的。显然，科学家（当时的用户大部分都是科学家）将数学公式记在笔记上，但程序只是一串机器指令的列表。最初的程序列表是十进制或八进制数——与计算机内存中的表示形式完全匹配。后来，汇编器和“自动编码”出现了，即，人们发明了用符号名

称表示机器指令和机器特性（如寄存器）的语言。这样，程序员写出“LD R0 123”就可以将内存地址 123 中的内容读取到 0 号寄存器中。但是，每台机器都有自己的指令集和语言。



如果要选出那个时代有代表性的程序语言设计者，剑桥大学计算机实验室的 David Wheeler 无疑是当然的候选人。1949 年，他编写了运行于贮存程序式计算机上的第一个真正的程序（我们在 4.4.2.1 节提到的“平方表”程序）。大约有十个人都声称自己实现了最早的编译器（用于编译机器相关的“自动编码”），David Wheeler 是其中之一。他发明了函数调用（是的，即使是如此显而易见的简单事情，也还是需要某人在某时将它发明出来）。在 1951 年，他写了一篇杰出的论文来介绍如何设计库，这篇论文的内容超前了那个时代至少 20 年！他与 Maurice Wilkes（在互联网上搜索一下他！）和 D. J. Jill 合作完成了第一本关于程序设计的书。他是第一位计算机专业博士学位获得者（1951 年在剑桥大学），后来他的主要贡献在硬件领域（cache 体系结构、早期的局域网）和算法方面（如，TEA 加密算法（见 25.5.6 节），“Burrows-Wheeler 转换”（用于 bzip2 中的压缩算法））。David Wheeler 碰巧还是 Bjarne Stroustrup 的博士论文导师——计算机科学还是一门年轻的学科。David Wheeler 的一些最重要的成就都是在研究生阶段取得的。他后来继续在剑桥大学工作，成为剑桥大学教授和皇家学会院士。

### 参考文献

Burrows, M., and David Wheeler. “A Block Sorting Lossless Data Compression Algorithm.”

Technical Report 124, Digital Equipment Corporation, 1994.

Bzip2 link: [www.bzip.org/](http://www.bzip.org/).

Cambridge Ring website: <http://koo.corpus.cam.ac.uk/projects/earlyatm/cr82>.

Campbell-Kelly, Martin. “David John Wheeler.” *Biographical Memorials of Fellow of the Royal Society*, Vol. 52, 2006. (David Wheeler 的传记。)

EDSAC: <http://en.wikipedia.org/wiki/EDSAC>.

Knuth, Donald. *The Art of Computer Programming*. Addison-Wesley, 1968, 还有后来的许多版本。请在每一卷的索引中查找“David Wheeler”。

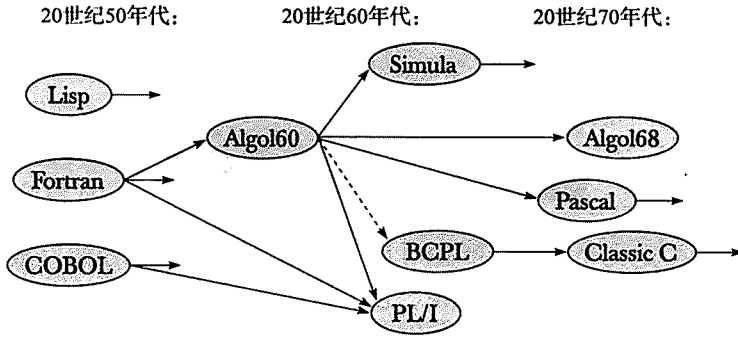
TEA link: [http://en.wikipedia.org/wiki/Tiny\\_Encryption\\_Algorithm](http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm).

Wheeler, D.J. “The Use of Sub-routines in Programmes.” *Proceedings of the 1952 ACM National Meeting*. (这就是 1951 年时所写的库的设计的论文。)

Wilkes. M.V., D. Wheeler, and D.J. Gil. *Preparation of Programs for an Electronic Digital Computer*. Addison-Wesley, 1951; 2nd edition, 1957. 第一本程序设计书籍。

## 22.2.2 现代程序设计语言的起源

下面是重要的早期程序设计语言的发展历程：



这些程序设计语言的重要性部分是因为它们曾经被广泛使用（目前在某些情况下仍在被广泛使用），另一个原因是，它们是重要的现代程序设计语言的祖先——而且通常还是直接祖先，具有相同的名字。在本节中，我们介绍三种早期程序设计语言——Fortran、COBOL 和 Lisp，大多数现代程序语言的祖先都可以追溯到这三种语言。

### 22.2.2.1 Fortran

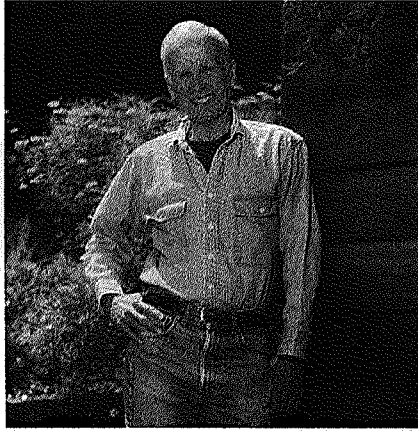
1956 年 Fortran 的发明可能是程序设计语言发展历史中最重要的一步。“Fortran”表示“公式转换”（Formula Translation），其基本思想是将人类（而不是机器）习惯的符号表示转换为高效的机器代码。Fortran 的符号表示法是一种适合于科学家和工程师描述问题数学求解方案的模型，而不是由（最新的）电子计算机所提供的机器指令。

以现代观点来看，Fortran 可以看作对“用代码直接描述应用领域”的首次尝试。它允许程序员像课本中那样书写线性代数公式。Fortran 提供了数组、循环和标准的数学函数（使用标准数学符号，如  $x+y$  和  $\sin(x)$ ）。它有一个数学函数的标准库，也提供了 I/O 机制，用户还可以自己定义函数和库。

Fortran 所使用的符号表示大都是机器无关的，因此 Fortran 代码通常只需很少改动就可以从一台计算机移植到另一台计算机上。这在当时的发展水平上，是一个巨大的进步。因此，Fortran 被认为是第一个高级程序设计语言。

Fortran 有一个非常重要的优点：由 Fortran 源码生成的机器码可以达到几乎最优的效率。要知道当时的计算机有几个房间那么大，并且极其昂贵（是一个优秀的程序员团队的年薪总和的许多倍），（按现代的标准）它们还慢得出奇（例如 100 000 条指令 / 秒），内存小得可怜（例如 8K 字节）。不过，人们还是能将有用的程序塞到这些机器中，因此，像 Fortran 这种符号描述方法上的改进，如果不能保持高效率，即便能大大提高程序员的生产率和程序的可移植性，也不会取得成功。

Fortran 在科学与工程计算这一目标领域取得了巨大的成功，并且一直在不断改进、完善。Fortran 语言的主要版本包括 II、IV、77、90、95、03。目前，关于 Fortran77 和 Fortran90 谁应用更广泛的争论仍然在继续。



第一个 Fortran 的定义和实现是由 IBM 的 John Backus 领导的小组完成的：“我们不知道需要什么以及如何做，某种程度上来说，它就是自然而然地发展起来了。”的确，他又如何能知道呢？之前从没有人做过类似的事情！但是一路走来，他们开发或者说发明了编译器的基本结构：词法分析、语法分析、语义分析和优化。时至今日，在数值计算优化领域，Fortran 仍然处于领导地位。此外，（在最初的 Fortran 之后）还出现了一种专门用于表示文法的符号系统：Backus-Naur 范式 (BNF)。这种方法在 Algol60 (见 22.2.3.1 节) 中首次被使用，现在已经用于大部分现代程序设计语言。在第 6、7 章中，我们也使用了某个版本的 BNF 来描述文法。



很久之后，John Backus 开辟了一个全新的程序设计语言分支（“函数式程序设计”）。与基于读写内存位置的从机器出发的方式相反，这种程序设计风格主张用数学方式来编写程序。需要注意的是，纯数学是没有赋值的概念的，甚至连操作的概念也没有。纯数学只是在一组给定的条件下，“简单地”声明什么肯定是真的。函数式程序设计的思想部分源于 Lisp (见 22.2.2.3 节)，一些函数式程序设计的思想也反映在 STL 中 (见第 16 章)。

### 参考文献

Backus, John. “Can Programming Be Liberated from the von Neumann Style?” *Communications of the ACM*, 1977. (Backus John 的图灵奖演说。)

Backus, John. “The History of FORTRAN I, II, and III.” *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

Hutton, Graham, *Programming in Haskell*. Cambridge University Press, 2007. ISBN 0521692695.

ISO/IEC 1539. *Programming Languages-Fortran*. (“Fortran 95” 标准。)

Paulson, L.C. *ML for the Working Programmer*. Cambridge University Press, 1991. ISBN 0521390222.

### 22.2.2.2 COBOL

COBOL (Common Business-Oriented Language, 通用面向商业语言) 曾是面向商业应用程序员的主要语言 (现在某些情况下仍然是)，就像 Fortran 曾是面向科学应用程序员的主要语言 (现在某些情况下仍然是) 一样。COBOL 主要用于数据处理：

- 数据复制；

- 数据存储和检索（记账）；
- 打印（报表）。

计算被看作小事（在 COBOL 的核心应用领域通常是正确的）。人们期望 / 宣称 COBOL 是如此接近“商务英语”，连管理人员都能用它来编程，从而很快就会使程序员变得多余。这是那些热衷于削减程序设计开支的经理的良好愿望，但从来没有实现，哪怕接近实现。

COBOL 最初是由一个委员会（CODASYL）在 1959 ~ 1960 年设计的，这个委员会是由美国国防部和一些主要的计算机制造商发起的，其目的是解决商业计算的需求。COBOL 的设计直接以 Grace Hopper 发明的 FLOW-MATIC 语言为基础。她的贡献之一就是使用了一种与英语十分接近的语法（与之相对的是由 Fortran 开创的使用数学符号的语法，目前仍然占据主导地位）。与 Fortran 以及其他所有成功的语言一样，COBOL 也历经不断的演化和发展。主要的版本包括 60、61、65、68、70、80、90 和 04。

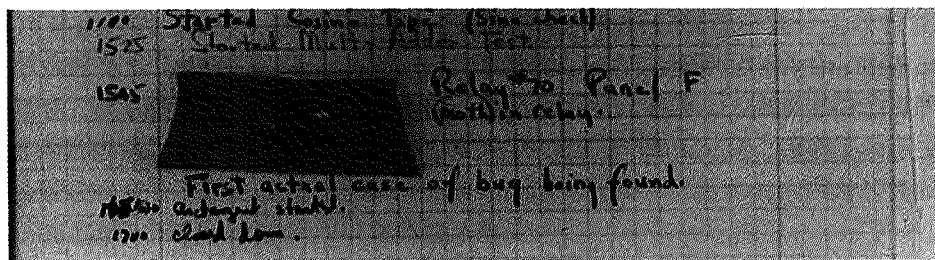
Grace Murray Hopper 拥有耶鲁大学的数学博士学位。在二战期间她为美国海军工作，研究最早期的计算机。在（早期的）计算机工业界工作了几年后，她又回到了海军。



“海军少将 Grace Murray Hopper 博士（美国海军）在早期的计算机程序设计领域做出了杰出的贡献。她将软件开发思想的研究作为一生的事业，作为这个领域的领路人，是她引领了从原始的程序设计技术到使用复杂编译器的转变。她坚信‘我们原来就是这么做的’不是继续这么做的必然理由。”

——Anita Borg 在 1994 年 “Grace Hopper Celebration of Women in Computing” 会议上的发言

Grace Murray Hopper 一直被认为是第一个将计算机中的错误称为“bug”的人。她无疑是最早使用这一术语并且在文档中对此进行了记载的人：





我们可以看到，bug 是一只真正的虫子（一只飞蛾），它直接引起了一个硬件故障。但是现代计算机故障多为软件故障，很少能如此生动地呈现出来。

### 参考文献

G. M. Hopper 传记: <http://tergestesoft.com/~eddysworld/hopper.htm>.

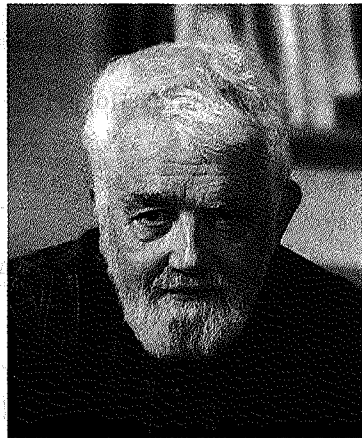
ISO/IEC 1989:2002. *Information Technology-Programming Languages-COBOL*.

Sammet, Jean E. "The Early History of COBOL." *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

### 22.2.2.3 Lisp

Lisp 最初是 John McCarthy 1958 年于麻省理工学院设计的一种语言，主要用于链表和符号处理（也因此而得名“LISt Processing”）。与编译型语言不同，最初的 Lisp 是解释型语言（现在通常仍是）。List 有几十种（可能更多，有几百种）变种。实际上，人们经常说“Lisp 默认就是复数”。目前最流行的版本是 Common Lisp 和 Scheme。这种语言曾经是（现在也是）人工智能领域研究的支柱（虽然发布的产品通常是用 C 或者 C++ 实现的）。Lisp 最主要的灵感源泉是 1 演算（的数学思想）。

在各自的应用领域中，Fortran 和 COBOL 的设计目标都是为了解决现实世界中的问题。而 Lisp 社群则更加关注程序设计本身和程序的优雅性。通常这些努力都很成功。Lisp 是第一种将自身定义与硬件分离的语言，也是第一种将语义建立在某种数学形式之上的语言。如果说 Lisp 有一个特定应用领域的话，也很难给出其准确定义：“人工智能”或者“符号计算”都不像“商业处理”和“科学计算”那样能清楚地对应到某种普通日常工作。在很多现代语言，尤其是函数式语言中都能发现来自 Lisp（或来自 Lisp 社群）的设计理念。



John McCarthy 在加州理工学院获得数学学士学位，在普林斯顿大学获得数学博士学位。你可能已经注意到了，很多程序语言的设计者都来自数学专业。在麻省理工学院完成了他载入史册的工作后，McCarthy 于 1962 年来到斯坦福大学，参与建立了斯坦福人工智能实验室。他被公认为人工智能（artificial intelligence）一词的发明人，并在这一领域做出了许多贡献。

## 参考文献

- Abelson, Harold, and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996, ISBN 0262011530.
- ANSI INCITS 226-1994 (formerly ANSI X3.226:1994). *American National Standard for Programming Language-Common LISP*.
- McCarthy, John. "History of LISP." *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.
- Steele, Guy L., Jr. *Common Lisp: The Language*. Digital Press, 1990. ISBN 1555580416.
- Steele, Guy L., Jr., and Richard Gabriel. "The Evolution of Lisp." Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

## 22.2.3 Algol 家族

在 20 世纪 50 年代后期, 许多人认为程序设计变得过于复杂、专用、不科学。人们还觉得程序设计语言的种类过于繁多了, 而且这些语言的组合既没有充分考虑通用性, 也没有坚实的理论基础。从那时起, 这种质疑的观点多次被提及, 但真正的改变来自 IFIP (International Federation of Information Processing, 国际信息处理联合会) 支持的一个工作组。在短短几年时间内, 他们创立了一种崭新的程序设计语言。这种语言颠覆了我们对于程序设计语言及其定义的认识。多数的现代程序设计语言 (包括 C++) 都曾从中受益。

### 22.2.3.1 Algol60

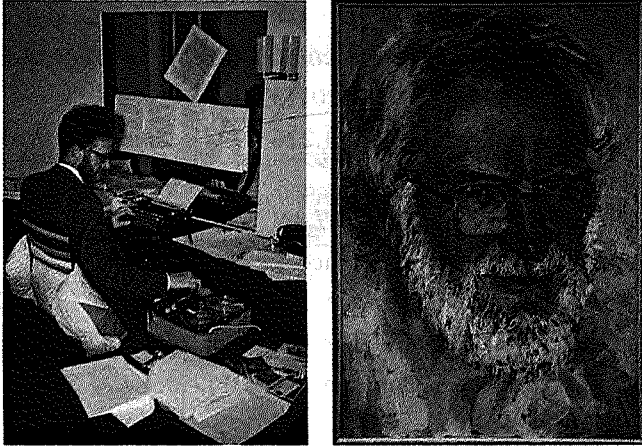
Algol (ALGOrithmic Language) 是由 IFIP 2.1 工作组设计的, 是对现代程序设计语言概念的重要突破: ✂

- 词法作用域;
- 使用文法定义语言;
- 语法和语义规则明确分离;
- 语言定义和实现明确分离;
- 系统化地使用 (静态即编译时) 类型;
- 直接支持结构化编程。

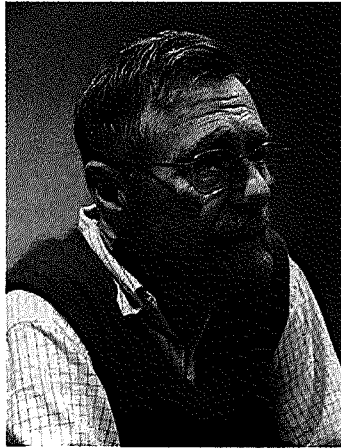
“通用编程语言”的理念就源于 Algol。在它之前, 程序设计语言都是专门服务于科学 (如 Fortran)、商业 (如 COBOL)、表处理 (如 Lisp) 或仿真等。在这些语言中, 与 Algol60 最接近的是 Fortran。

不幸的是, Algol60 从未在非学术领域中广泛使用。因为很多工业界人士认为它“过于古怪”, Fortran 程序员认为它“太慢”, COBOL 程序员认为它“对商业处理的支持不足”, Lisp 程序员认为它“不够灵活”, 大多数工业界人士 (包括控制程序设计工具投资的经理) 认为它“太学院派”, 很多美国人认为它“太欧洲”。多数的批评是正确的。例如, Algol60 报告中没有定义任何 I/O 机制! 但是, 同时代的其他语言也存在类似的问题, 不能因此而否定 Algol 语言的重要地位, Algol 为很多领域定下了新的标准。

Algol60 的一个问题是没人知道如何实现它。这一问题最后由 Peter Naur (Algol60 报告的编者) 和 Edsger Dijkstra 领导的程序员团队解决:



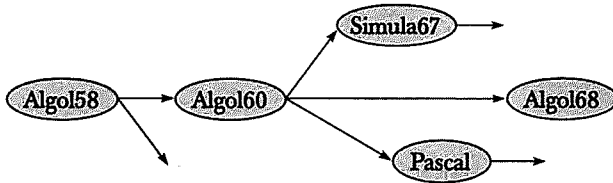
Peter Naur 就读于哥本哈根大学（学习天文），随后在哥本哈根理工大学（DTH）工作，并为丹麦计算机制造商 Regnecentralen 工作。他最早接触程序设计是（1950 ~ 1951）在英国剑桥大学计算机实验室（当时丹麦还没有计算机），之后他在这个领域的杰出贡献跨越了学术界和工业界。他是 Backus-Naur 范式（Backus-Naur Form, BNF，用于描述文法）的共同发明人，也是最早提议对程序进行形式化推理的人（大约在 1971 年，Bjarne Stroustrup 从 Peter Naur 的学术论文中第一次接触到了不变式的使用）。Naur 从未停止对计算科学前景的思考，一直在关注程序设计中的人的因素。事实上，他后期的研究工作已经可以归为哲学范畴了（除了他认为传统的学院派哲学毫无意义）。他是哥本哈根大学第一位 Datalogi 教授（datalogi 是丹麦语，最好翻译为“informatics”（信息学）；Peter Naur 非常不喜欢“计算机科学”（computer science）一词，认为这是彻底的用词不当，因为他并不认为“计算”指的就是“计算机”）。



Edsger Dijkstra 是另一位史上最伟大的计算机科学家。他在莱顿学习物理，但早期的计算相关的工作是在阿姆斯特丹数学中心进行的。他后来又在很多地方工作过，包括埃因霍温理工大学、宝来公司和德州大学奥斯汀分校。除了 Algol 语言方面的杰出工作外，他还是利用数学逻辑研究程序设计和算法的先驱和积极倡导者，此外还是 THE 操作系统的设计

者和实现者之一。THE 是最早的具有系统化处理并发操作能力的操作系统之一。THE 表示 “Technische Hogeschool Eindhoven” ——Edsger Dijkstra 当时工作的大学。他的最著名的论文 “Go-To Statement Considered Harmful”，令人信服地阐述了非结构化控制流所存在的问题。

Algol 家族树如下：



注意 Simula67 和 Pascal，这两种语言是很多（几乎是所有的）现代语言的祖先。

### 参考文献

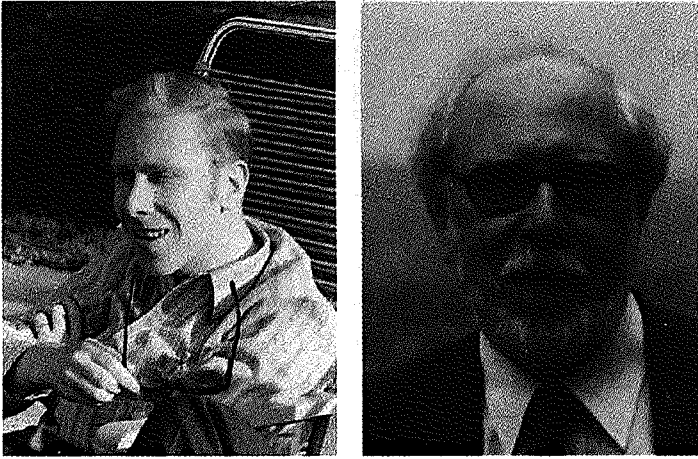
- Dijkstra, Edsger W. “Algol 60 Translation: An Algol 60 Translator for the x1 and Making a Translator for Algol 60.” Report MR 35/61. Mathematisch Centrum (Amsterdam), 1961.
- Dijkstra, Edsger. “Go-To Statement Considered Harmful.” *Communications of the ACM*, Vol. 11 No. 3. 1968.
- Lindsey, C. H. “The History of Algol68.” Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
- Naur, Peter, ed. “Revised Report on the Algorithmic Language Algol 60.” A/S Regnecentralen (Copenhagen), 1964.
- Naur, Peter, “Proof of Algorithms by General Snapshots.” *BIT*, VOL. 6, 1966, pp. 310-16. (可能是第一篇有关程序正确性证明的论文。)
- Naur, Peter. “The European Side of the Last Phase of the Development of ALGOL 60.” *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.
- Perlis, Alan J. “The American Side of the Development of Algol.” *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.
- Van Wijngaarden, A., B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, eds. *Revised Report on the Algorithmic Language Algol 68* (Sept. 1973). Springer-Verlag, 1976.

### 22.2.3.2 Pascal

在 Algol 家族树中，Algol68 语言是一个巨大且雄心勃勃的项目。像 Algol60 一样，它也是由 “Algol 委员会” (IFIP 工作组 2.1) 负责。但是看上去它 “永远” 也不能完成，以至于很多人失去了耐心，并怀疑这样一个项目所产生的成果是否真的有用。Algol 委员会的一个成员 Niklaus Wirth，决定设计、实现自己的 Algol 语言。这就是 Pascal，它也是源于 Algol，但与 Algol68 不同，它是 Algol60 的简化。

Pascal 于 1970 年完成，它确实很简单，带来的一个后果就是不够灵活。人们一般认为它只适合于教学，但早期的相关论文都把它描述为 Fortran 的替代品，用于当时的超级计算

机。Pascal 确实非常容易学习，并且随着一个可移植性极好的版本的实现，它逐渐成为一种十分流行的教学语言，但实践证明它没有对 Fortran 造成任何威胁。



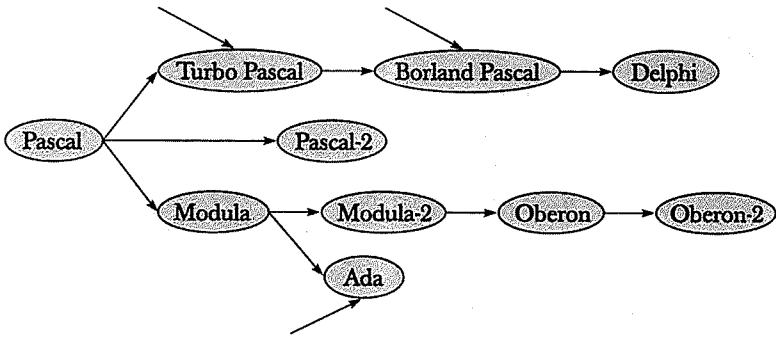
Pascal 是瑞士苏黎世联邦理工学院 (ETH) 的 Niklaus Wirth 教授 (上面的照片分别是 1969 年和 2004 年拍摄的) 的杰作。他在加州大学伯克利分校获得了电子工程和计算机科学博士学位，并终生和加州有着不解之缘。如果说要推选一位程序语言设计终极专家的话，Wirth 教授是这个世界上最配得上这个称号的人。在 25 年时间里，他设计和实现了下列语言：

- Algol W;
- PL/360;
- Euler;
- Pascal;
- Modula-2;
- Oberon;
- Oberon-2;
- Lola (一种硬件描述语言)。

Niklaus Wirth 把这些工作当做对简洁性的无止境的追求。他的工作对这个领域产生了巨大的影响。学习这一系列语言是一种非常有趣的练习。Wirth 教授是唯一在 HOPL 会议上提出过两种语言的人。

最终，纯粹的 Pascal 被证明对于工业界来说太简单、太严格了。20 世纪 80 年代，主要是在 Anders Hejlsberg 的努力下，将 Pascal 从消亡的边缘拉了回来。Anders Hejlsberg 是 Borland 的三位创始人之一。最初他设计并实现了 Turbo Pascal (与其他实现相比，有着更加灵活的参数传递机制)，后来又设计了类似 C++ 的对象模型 (但是仅有单一继承，并有更好的模块机制)。他就读于哥本哈根科技大学——Peter Naur 曾工作过的地方——世界有时真的是很小啊。Anders Hejlsberg 后来为 Borland 设计了 Delphi 语言，为微软设计了 C# 语言。

Pascal 家族树 (经过必要的简化后) 如下所示：



### 参考文献

Borland/Turbo Pascal. [http://en.wikipedia.org/wiki/Turbo\\_Pascal](http://en.wikipedia.org/wiki/Turbo_Pascal).

Hejlsberg, Anders, Scott Wiltamuth, and Peter Golde. *The C# Programming Language, Second Edition*. Microsoft. NET Development Series. ISBN 0321334434.

Wirth, Niklaus. "The Programming Language Pascal." *Acta Informatics*, Vol.1 Fasc 1, 1971.

Wirth, Niklaus. "Design and Implementation of Modula." *Software-Practice and Experience*, Vol.7 No.1, 1997.

Wirth, Niklaus. "Recollections about the Development of Pascal." Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol.28 No.3, 1993.

Wirth, Niklaus. *Modula-2 and Oberon*. Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III). San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.

#### 22.2.3.3 Ada

Ada 语言是为美国国防部专门设计的。特别是，它是一种适合于为嵌入式系统编写可靠、可维护程序的语言。它的最明显的祖先是 Pascal 和 Simula (见 22.2.3.2 和 22.2.4 节)。Ada 设计小组的领导人是 Jean Ichbiah——他曾经是 Simula 用户组的主席。Ada 的设计强调：

- 数据抽象 (但直到 1995 年才引入继承机制)；
- 强大的静态类型检查；
- 语言直接支持并发性。

Ada 的设计目标是希望在程序设计语言中体现软件工程思想。其结果就是，美国国防部设计出的不是一门语言，而是设计语言的一套详细流程。众多的人和组织都对此做出了贡献，整个项目是通过一系列的竞赛来推进的，首先是评选出最佳的语言规范，随后就是设计能体现出最佳规范思想的最佳语言。这个长达 20 多年的巨大项目 (1975 ~ 1998) 从 1980 年开始由 AJPO (Ada Joint Program Office, Ada 联合计划组织) 负责管理。

在 1979 年，语言设计完成，并以 Augusta Ada Lovelace 女士 (著名诗人拜伦勋爵的女儿) 的名字命名。Lovelace 女士被认为是第一位现代程序员 (当然，这里“现代”的定义很宽泛)，因为她在 19 世纪 40 年代曾和 Charles Babbage (剑桥大学的卢卡斯数学教授——牛顿曾经担任过这一职位!) 一起工作，研究一种革命性的机械式计算机。不幸的是，Babbage 的机器没能成功地成为一个实用工具。



正是因为其设计遵循了详尽的流程，Ada 被认为是终极的“委员会设计”语言。作为最终赢得胜利的设计团队的领导者，来自法国 Honeywell Bull 公司的 Jean Ichbiah 却强烈地否认这一点。然而，我猜测（基于和他的讨论），如果没有这个详细流程的束缚，他本可以设计出更好的语言。

Ada 被美国国防部确定为军事应用程序强制使用语言已经有很多年历史了，以至于有这样的说法：“Ada 不仅仅是一个好的思想，更是一项法律！”最初，Ada 只是“强制”使用而已，但随着许多项目获得“豁免权”来使用其他语言（通常是 C++），美国国会通过了一项法律，要求大多数军事应用程序必须使用 Ada。后来，这一法律由于所面对的商业和技术上的现实问题，不得不被废除了。Bjarne Stroustrup 是极少数工作成果曾被美国国会禁止的人之一。

我们坚信，与它获得的声誉相比，Ada 实际上要好得多。假如美国国防部不那么强硬地推广 Ada，而且能够正确应用它的话（用做开发过程、软件开发工具、文档等的标准），它也许会成功得多。直到现在，Ada 仍是航空应用程序和类似的高级嵌入式系统应用程序的重要编程语言。

Ada 在 1980 年成为军事标准，1983 年成为 ANSI 标准（第一个 Ada 实现在 1983 年完成——在第一个标准完成之后三年！），1987 年成为 ISO 标准。这个 ISO 标准在 1995 年被全面地（当然在保证兼容性的前提下）进行了修订。重要的改进包括更灵活的并发机制以及支持继承。

### 参考文献

Barnes, John. *Programming in Ada 2005*. Addison-Wesley, 2006. ISBN 0321340787.

整理过的 Ada 参考手册，包括国际标准（ISO/IEC 8652:1995）。*Information Technology-Programming Languages-Ada*，由 *Technical Corrigendum 1*（ISO/IEC 8652:1995:TC1:2000）修改后更新而来。

Ada information page: [www.adaic.org/](http://www.adaic.org/).

Whitaker, William A. *ADA-The Project: The DoD High Order Language Working Group. Proceedings of the ACM History of Programming Languages Conference (HOPL-2)*. *ACM SIGPLAN Notices*, Vol. 28 No.3, 1993.

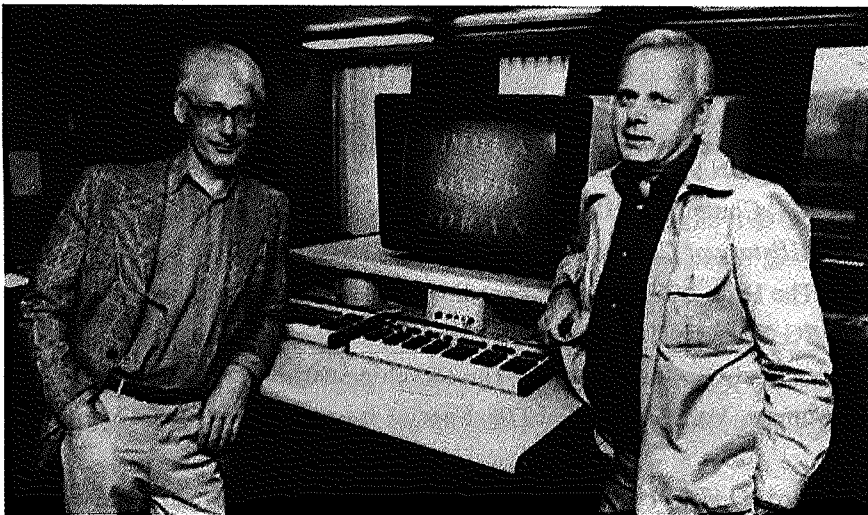
## 22.2.4 Simula

20 世纪 60 年代中期，Kristen Nygaard 和 Ole-Johan Dahl 设计了 Simula，当时他们在挪威计算中心和奥斯陆大学工作。Simula 毫无疑问是 Algol 语言家族的成员。实际上，Simula 几乎就是 Algol60 的超集。但是，我们选择单独对 Simula 进行介绍，这是因为现在人们所说的“面向对象程序设计”的大多数基本思想都来自 Simula。它是第一种提供继承和虚函数机制的语言。术语类（class）——表示“用户自定义类型”和虚函数（virtual function）——表示可以覆盖并可通过基类接口调用的函数，都来源于 Simula。

Simula 的贡献并不局限于语言方面，它更重要的贡献是提出了明确的面向对象设计的概念，这基于用代码来建模现实世界现象的思路：

- 用类和类对象表示思想。
- 用类层次（继承）表示层次关系。

因此，程序变成一组相互作用的对象，而不是单个的庞然大物。

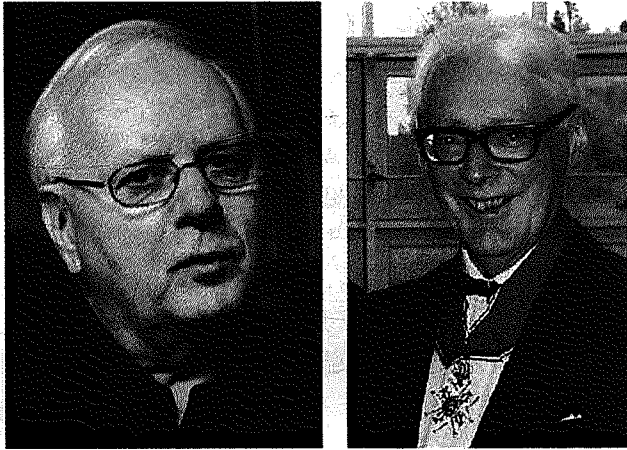


Kristen Nygaard 是 Simula 的共同发明人（另一位是 Ole-Johan Dahl，照片中左侧戴眼镜者），他在很多方面都堪称巨人（包括身高），他的热情和慷慨也完全配得上这样的声誉。他构思了面向对象编程和设计（特别是继承）的基本思想，并在此后的几十年间不断探索这些思想的含义。他从不满足于简单、短期和短视的答案。他还数十年如一日积极投身到社会活动中。他本应获得更高的声望，如果挪威不是置身于欧盟之外的话。但他认为欧盟有可能成为一个中央集权和官僚主义的噩梦般的机构，它不会关心挪威这样的边缘小国的需求。20 世纪 70 年代中期，Kristen Nygaard 花了大量时间在丹麦奥尔胡斯大学的计算机科学系上（当时，Bjarne Stroustrup 在那里攻读硕士学位）。

Kristen Nygaard 在奥斯陆大学获得数学硕士学位。他在 2002 年去世，就在他即将（与他终生的朋友 Ole-Johan Dahl 一起）获得 ACM 图灵奖之前的一个月，这个奖项对计算机科学家来说是最高的荣誉。

Ole-Johan Dahl 是一位更传统的学者。他的研究兴趣主要集中在语言规范和形式化方法方面。1968 年，他成为奥斯陆大学第一位信息学（计算机科学）正教授。





2000 年 8 月, Dahl 和 Nygaard 被挪威国王授予圣奥拉夫高级骑士勋章。在他们的家乡, 纯粹的计算机技术人员也能获得如此高的荣誉!

#### 参考文献

- Birtwistle, G., O-J. Dahl, B. Myrhaug, and K. Nygaard. *SIMULA Begin*. Student-litteratur (Lund. Sweden), 1979. ISBN 9144062125.
- Holemevik, J. R. "Compiling SIMULA: A Historical Study of Technological Genesis." *IEEE Annals of the History of Computing*, Vol. 16 No. 4, 1994, pp.25-37.
- Krogdahl, S. "The Birth of Simula." Proceeding of the HiNC 1 Conference in Trondheim, June 2003 (IFIP WG 9.7, 与 IFIP TC 3 合办).
- Nygaard, Kristen, and Ole-Johan Dahl. "The Development of the SIMULA Languages." *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Lanuages Conference.
- SIMULA Standard. *DATA processing-Programming Languages-SIMULA*. Swedish Standard, Stockholm, Sweden (1987). ISBN 9171622349.

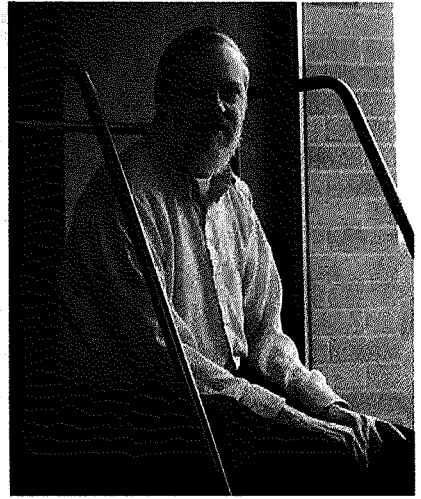
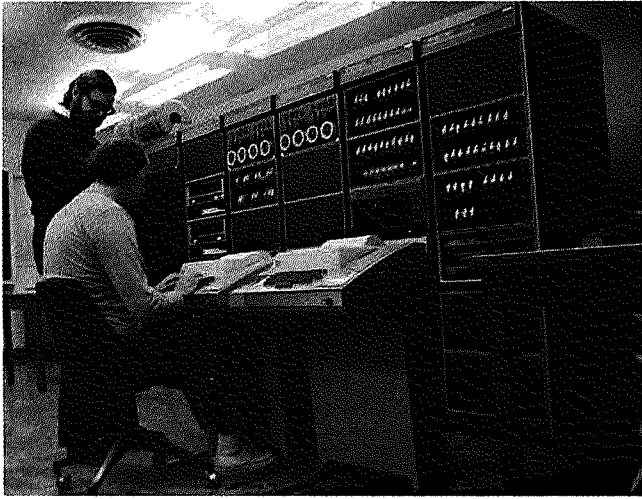
#### 22.2.5 C



在 1970 年, 一件“众所周知”的事情是, 重要系统的程序设计——特别是操作系统的实现, 必须使用汇编语言, 从而不具备可移植性。这与 Fortran 出现之前科学计算程序设计的处境很相像。一些个人和组织开始着手挑战这个成见, 最终, C 语言(见第 27 章)成为这些工作中的最成功者。

Dennis Ritchie 在位于新泽西茉莉山丘的贝尔电话实验室计算科学研究中心设计并实现了 C 语言。C 的魅力在于它是一种简单的编程语言, 这种简单性是慎重规划后的结果, 而且 C 语言与硬件的基本特性关联非常紧密。目前 C 语言版本的复杂性(其中大多数出于兼容性考虑也出现在了 C++ 中)大多数是在 Dennis Ritchie 的最初设计之后添加进来的, 并且某些情况并不符合他的原意。C 的成功部分是因为很早就被广泛使用, 但它真正强大之处在于从语言特性到硬件设施的直接映射(见 25.4 ~ 25.5 节)。Dennis Ritchie 曾简单地将 C 描述为“一种强类型、弱检查的语言”; 也就是说, C 是一种静态(编译时)类型的系统, 在程

序中不按对象定义的方式使用它是非法的——但 C 编译器又不会检查这种问题。但当资源有限时，这样做还是有意义的，例如内存只有 48K 字节时，这样做可以得到更简短的代码。在 C 投入应用后不久，人们设计了一种称为 lint 的程序，它将类型系统验证从编译器中分离出来。

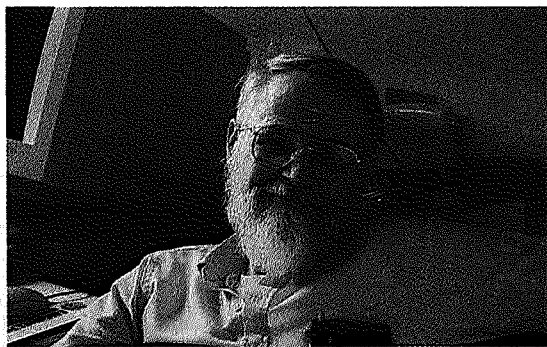


Dennis Ritchie 与 Ken Thompson 一起发明了 Unix，它无疑是最有影响力的操作系统。C 一直都与 Unix 操作系统紧密联系在一起，近年来，又与 Linux 和开源项目的发展紧密相连。

在为贝尔实验室计算机科学研究中心工作了 40 年之后，Dennis Ritchie 现在已经从朗讯公司贝尔实验室退休了。他从哈佛大学获得物理学学士学位，从哈佛大学获得应用数学博士学位。但或者因为忘记或者因为拒绝支付 60 美元的注册费，他没有获得博士学位证书。



在 1974 至 1979 年间，贝尔实验室中的很多人都对 C 的设计和应用产生过影响。Doug McIlroy 是所有人都喜欢的评论家、讨论伙伴和创意丰富的人。他影响了 C、C++、Unix 和其他很多研究工作。



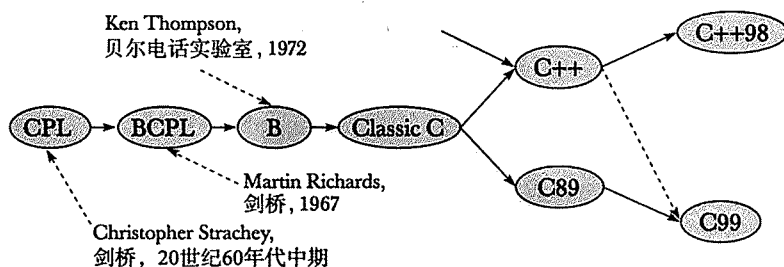
Brian Kernighan 是一位杰出的程序员和作家。他的代码和文章都是清晰性的典范。本书的风格就部分来源于他的杰作《The C Programming Language》(Brian Kernighan 和 Dennis Ritchie 合著, 因而被称为“K&R”)中的指南部分。

仅有好的思想是不够的; 为了能被更大范围的人们所用, 应该将这些思想归约到最简单的形式, 并以目标读者群中更多人能够接受的方式阐述清楚。在表达思想时, 冗长啰嗦是最可怕的敌人, 同样可怕的还有模糊和过度简化。纯粹主义者经常嘲笑这种大众化的努力方向, 他们喜欢将“原始结果”以一种只有专家才能理解的方式表达出来。我们的理念与他们不同: 将不平凡的、有价值的思想灌输到初学者的头脑中, 是一件很困难的事情, 对于提高我们的专业水准是很有帮助的, 也能最大程度为社会做出贡献。

多年以来, Brian Kernighan 参与了很多有影响的程序设计和出版项目。两个典型的例子是 AWK——一种早期的脚本语言, 用作者名字的首字母命名 (Aho、Weiberger 和 Kernighan), 以及 AMPL (A Mathematical Programming Language, 数学编程语言)。

目前, Brian Kernighan 是普林斯顿大学的教授; 他是一位优秀的教师, 擅长于将复杂问题讲得清晰易懂。他为贝尔实验室计算机科学研究中心工作了超过 30 年。贝尔实验室后来更名为 AT&T 贝尔实验室, 然后又被拆分为 AT&T 实验室和朗讯贝尔实验室。他从多伦多大学获得了物理学学士学位, 从普林斯顿大学获得电子工程博士学位。

C 语言的家族树结构如下图:



C 源于三种语言: 英国的一直未完成的项目 CPL; Martin Richards 离开剑桥大学访问麻省理工学院时设计的 BCPL (Basic CPL) 语言; 以及由 Ken Thompson 设计的称为 B 的解释型语言。后来, C 制订了 ANSI 和 ISO 标准, 并有很多特性受到了 C++ 的影响 (如函数参数检查和 const)。

CPL 是剑桥大学和伦敦的帝国理工学院的合作项目。这个项目最初是在剑桥大学进行的, 因此当初“C”的正式含义是“剑桥”(Cambridge)。当帝国理工学院参与进来后,

“C”的正式含义就变为“联合”(Combined)。实际上(我们常常听说的)它一直就表示“Christopher”，即CPL的主要设计者Christopher Strachey。

#### 参考文献

Brian Kernighan 的主页：<http://cm.bell-labs.com/cm/cs/who/bwk> 和 [www.cs.princeton.edu/~bwk/](http://www.cs.princeton.edu/~bwk/)。

Dennis Ritchie 的主页：<http://cm.bell-labs.com/cm/cs/who/dmr>。

ISO/IEC 9899:1999. *Programming Language - C*. (C 标准。)

Kernighan, Brian, and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 1978. Second Edition, 1988. ISBN 0131103628.

贝尔实验室计算机科学研究中心成员列表：<http://cm.bell-labs.com/cm/cs/alumni.html>。

Ritchards, Martin. *BCPL - The Language and Its Compiler*. Cambridge University Press, 1980. ISBN 0521219655.

Ritchie, Dennis. “*The Development of the C Programming Language*.” Proceedings of the ACM History of Programming Language Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

Salus, Peter. *A Quarter Century of UNIX*. Addison-Wesley, 1994. ISBN 0201547775.

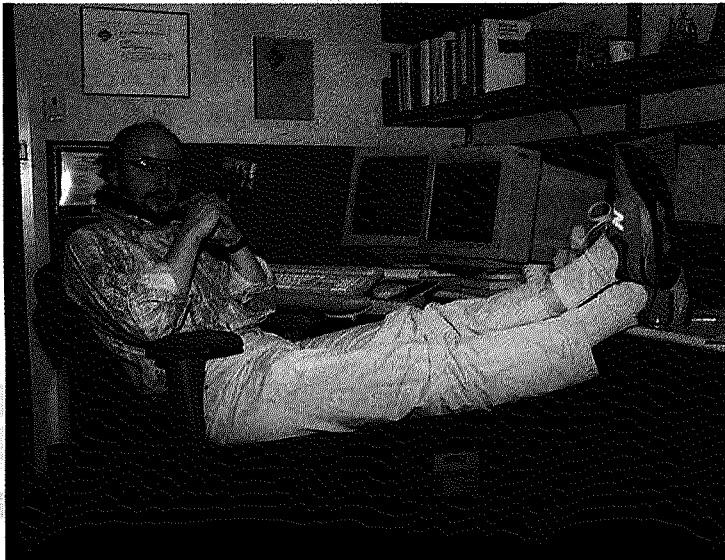
### 22.2.6 C++

C++ 是一种偏向于系统编程的通用程序设计语言，它的特点是：



- 可以看做是更好的 C；
- 支持数据抽象；
- 支持面向对象程序设计；
- 支持泛型程序设计。

C++ 最初是由 Bjarne Stroustrup 在新泽西茉莉山丘的贝尔电话实验室计算科学研究中心设计并实现的。他的办公室与 Dennis Ritchie、Brian Kernighan、Ken Thompson、Doug McIlroy 以及其他 Unix 巨人们相邻。



Bjarne Stroustrup 在家乡的丹麦奥胡斯大学数学专业获得了计算机科学硕士学位。然后，他来到剑桥大学，在 David Wheeler 指导下获得了计算机科学博士学位。C++ 的主要贡献包括：



- 使抽象技术对于主流项目来说，其应用代价可以承受，易于管理。
- 将面向对象和泛型程序设计技术用于对性能有较高要求的应用领域的先驱。

在 C++ 出现之前，这些技术（通常一起混杂在“面向对象程序设计”的标签之下）并不为工业界所熟知。与 Fortran 之前的科学计算程序设计和 C 之前的系统程序设计所处的情况类似，人们“公认”这些技术对于实际应用来说代价太高，对于“普通程序员”来说太难以掌握。

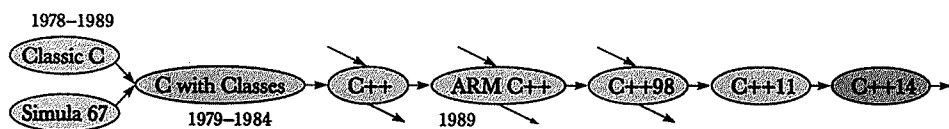
C++ 的研究工作始于 1979 年，在 1985 年发布了第一个商用版本。在最初的设计和实现之后，Bjarne Stroustrup 与贝尔实验室和其他地方的朋友们进一步对其进行完善，直到 1990 年 C++ 标准化进程正式开始。从那时起，C++ 的定义由 ANSI（美国国家标准化组织）负责制订，从 1991 年起改由 ISO（国际标准化组织）负责。Bjarne Stroustrup 在这一进程中承担了主要工作，他是负责语言新特性的关键小组的主席。第一个国际标准（C++98）在 1998 年批准通过，第二个版本在 2011 年通过（C++11）。下一个 ISO 标准版本将是 C++14（已于 2014 年通过），接下来会是 C++1y，最可能是 C++17。

经过最初十年的成长，C++ 最重要的发展就是 STL——容器和算法的标准库。它主要是 Alexander Stepanov 几十年努力的成果，其目标是生成更通用、更有效的软件，它从数学之美、数学之实用中受到了很多启发。



Alex Stepanov 是 STL 的发明者和泛型程序设计的先驱。他毕业于莫斯科大学，研究工作包括机器人、算法及其他领域，他在这些工作中使用过多种语言（包括 Ada、Scheme 和 C++）。从 1979 年开始，他在美国学术和工业界工作，曾为通用电气实验室、AT&T 贝尔实验室、惠普、Silicon Graphics 和 Adobe 等工作。

C++ 语言家族树如下图所示：



Bjarne Stroustrup 最初的设想是“支持类的 C”——综合 C 和 Simula 的思想。但随着其后继者 C++ 的实现，这一设想就消失了。

人们讨论程序设计语言时，常常集中在优雅的设计和高级特性上。但是，以这两方面评判，C 和 C++ 并不是计算领域中历史上最成功的两种语言。它们的强大在于灵活性、性能和稳定性。重要的软件系统的生命期会超过几十年，它们通常会耗尽硬件资源，并且还常常遇到完全无法预料的修改需求。C 和 C++ 已经证明了它们能在这种环境中茁壮成长。我们喜欢引用 Dennis Ritchie 的名言：“有些语言是为了证明一个观点而设计，而其他一些语言则是为了解决问题。”这里的“其他”主要就是就是指 C。Bjarne Stroustrup 喜欢说：“其实我知道如何设计一种比 C++ 更漂亮的语言。”C++ 与 C 一样，其目标不是抽象的美（尽管我们很赞成在可能的情况下追求美），而是实用。

### 参考文献

Alexander Stepanov 的文章：[www.stepanovpepers.com](http://www.stepanovpepers.com)。

Bjarne Stroustrup 的主页：[www.stroustrup.com](http://www.stroustrup.com)。

ISO/IEC 14882:2011. *Programming Language-C++*. (C++ 标准。)

Stroustrup, Bjarne. “A History of C++:1979—1991.” *Proceedings of the ACM History of Programming Languages Conference (HOPL-2)*. *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.

Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 978-0321563842.

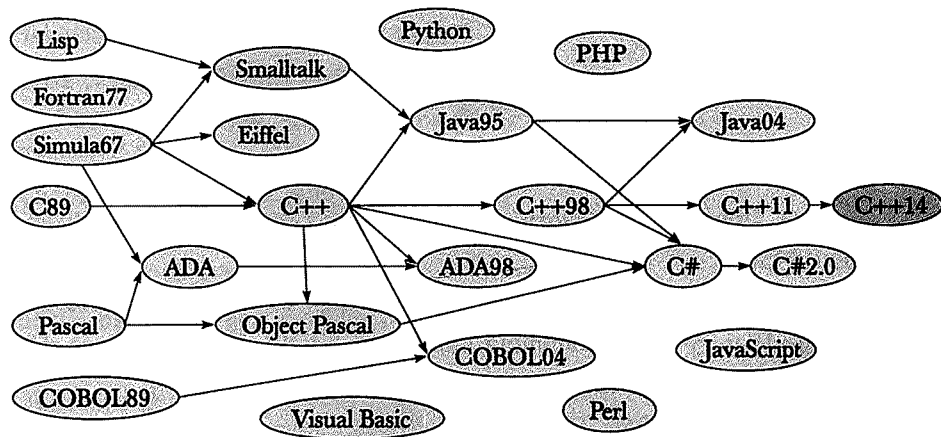
Stroustrup, Bjarne. *A Tour of C++*, Addison-Wesley, 2013. ISBN 978-0321958310.

Stroustrup, Bjarne. “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, Aug., and Sept. 2002

Stroustrup, Bjarne. “Evolving a Language in and for the Real World: C++ 1991—2006.” *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*. San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.

### 22.2.7 今天

目前还在使用的程序设计语言有哪些，用于什么领域？这确实是一个难以回答的问题。当前语言的家族树即便是以最简化的形式呈现，也很拥挤、杂乱：



⚠ 实际上，我们在互联网上（或其他地方）找到的大多数统计数据都比谣言强不了多少，因为它们所选取的都是一些与语言的使用关联度很差的指标。例如，不少网站张贴的内容包括程序语言的名称、编译器出货量、学术论文量和书籍销售量等。但所有这些指标都更倾向于新语言而不是已成熟的语言。我们再换个问题，什么人可以称作程序员呢？每天都使用程序设计语言的人？只是在学习时写些小程序的学生算是程序员吗？讲授程序设计的教授呢？几乎每年都只写一个程序的物理学家呢？如果一个专业程序员每周都会使用几种不同的语言，那么他应该被统计多次还是一次呢？我们可以看到，对这些问题的每个答案都会导致不同的统计方式和结果。

但是，我们认为有必要给你一个具体意见：2014 年全世界大约有 1000 万专业程序员。我们是从 IDC（一个数据收集公司）的数据、与出版商和编译器提供商进行的讨论以及各种互联网资源得出这个结论的。这个数量可能不准确，但我们确定，无论“程序员”定义如何，只要大致合理，实际值肯定在 100 万到 1 亿之间。程序员们使用哪种语言呢？Ada、C、C++、C#、COBOL、Fortran、Java、PERL、PHP、Python 以及 Visual Basic 可能（仅仅是可能）占据了超过 90% 的份额。

✎ 除了本章提到的语言之外，我们还可以列出几十甚至上百种语言。但除了对有趣的和重要的语言公平些以外，这没有任何意义。如果需要的话，你可以自行查找相关信息。一个专业人员通常都懂几种语言，并且有能力在必要时学习新的语言。世界上并不存在适用于所有人和所有应用的“真正的语言”。实际上，我们所能想到的所有重要系统，都使用了不止一种语言来实现。

## 22.2.8 参考资料

上面提到的每种语言都有一个参考资料列表。下面是涵盖几种语言的一些参考资料：

更多的语言设计者的网页 / 图片

[www.angelfire.com/tx4/cus/people/](http://www.angelfire.com/tx4/cus/people/).

几个语言例子

<http://dmoz.org/Computers/Programming/Languages/>.

教科书

Scott, Michael L. *Programming Language Pragmatics*. Morgan Kaufmann, 2000. ISBN 1558604421.

Sebesta, Robert W. *Concepts of Programming Languages*. Addison-Wesley, 2003. ISBN 0321193628.

历史书籍

Bergin, T.J., and R.G. Gibson, eds. *History of Programming Languages – II*. Addison-Wesley, 1996. ISBN 0201895021.

Hailpern, Brent, and Barbara G. Ryder, eds. *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*. San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.

Lohr, Steve. *Go To: The Story of the Math Majors, Brideg Players, Engineers, Chess Wizards*,

- Maverick Scientists and Iconoclasts-The Programmers Who Created the Software Revolution.* Basic Books, 2002. ISBN 9780465042265.
- Sammet, Jean. *Programming Languages: History and Fundamentals.* Prentice-Hall, 1969. ISBN 0137299885.
- Wexelblat, Richard L., ed. *History of Programming Languages.* Academic Press, 1981, ISBN 0127450408.

## 思考题

1. 历史有什么用处？
2. 程序设计语言有什么用处？请给出几个例子。
3. 请给出几种客观上可以认为是优点的程序设计语言的基本特性。
4. 抽象的含义是什么？更高层的抽象呢？
5. 我们对代码的四个高层理念是什么？
6. 列出高层程序设计的潜在优点。
7. 重用是什么？它可以带来什么好处？
8. 什么是过程式程序设计？给出一个具体的例子。
9. 什么是数据抽象？给出一个具体的例子。
10. 什么是面向对象程序设计？给出一个具体的例子。
11. 什么是泛型程序设计？给出一个具体的例子。
12. 什么是多范式程序设计？给出一个具体的例子。
13. 第一个运行在贮存程序式计算机上的程序出现在什么时候？
14. David Wheeler 以什么工作知名？
15. John Backus 设计的第一种语言的主要贡献是什么？
16. Grace Murray Hopper 设计的第一种语言是什么？
17. John McCarthy 的主要研究领域是什么？
18. Peter Naur 对 Algo60 有哪些贡献？
19. Edsger Dijkstra 以什么工作知名？
20. Niklaus Wirth 设计并实现的是哪种语言？
21. Anders Hejlsberg 设计的是哪种语言？
22. Jean Ichbiah 在 Ada 项目中扮演什么角色？
23. Simula 开创了什么程序设计风格？
24. Kristen Nygaard 曾在哪里（奥斯陆之外）教书？
25. Ole-Johan Dahl 以什么工作知名？
26. Ken Thompson 是哪个操作系统的主要设计者？
27. Doug McIlroy 以什么工作知名？
28. Brian Kernighan 最著名的著作是什么？
29. Dennis Ritchie 曾在哪里工作？
30. Bjarne Stroustrup 以什么工作知名？
31. Alex Stepanov 使用哪种语言设计了 STL？
32. 列出 10 种 22.2 节中没有介绍的语言。



33. Scheme 是哪种语言的变种?
34. C++ 最主要的两个起源是什么?
35. C++ 语言中 C 的表示什么?
36. Fortran 是一个缩写吗? 如果是, 全称是什么?
37. COBOL 是一个缩写吗? 如果是, 全称是什么?
38. Lisp 是一个缩写吗? 如果是, 全称是什么?
39. Pascal 是一个缩写吗? 如果是, 全称是什么?
40. Ada 是一个缩写吗? 如果是, 全称是什么?
41. 哪种编程语言最好?

## 术语

在本章中, “术语” 是真实的语言、人和组织。

### • 语言

Ada	Algol	BCPL	C	C++	COBOL
Fortran	Lisp	Pasca	Scheme	Simula	

### • 人

Charles Babbage	John Backus	Ole-Johan Dah	Edsger Dijkstra
Anders Hejlsberg	Grace Murray Hopper	Jean Ichbiah	Brian Kernighan
John McCarthy	Doug McIlroy	Peter Naur	Kristen Nygaard
Dennis Ritchie	Alex Stepanov	Bjarne Stroustrup	Ken Thompson
David Wheeler	Niklaus Wirth		

### • 组织

贝尔实验室	Borland 公司	剑桥大学 (英国)
ETH (苏黎世联邦理工学院)	IBM 公司	麻省理工大学 (MIT)
挪威计算机中心	普林斯顿大学	斯坦福大学
哥本哈根理工大学	美国国防部	美国海军

## 习题

1. 请给出程序设计的定义。
2. 请给出程序设计语言的定义。
3. 浏览本书, 注意章节中的插图。哪些插图是计算机科学家? 撰写一段文字, 总结每位科学家的贡献。
4. 浏览本书, 注意章节中的插图。哪些插图不是计算机科学家? 指出每幅插图出自哪个国家、哪个领域。
5. 用本章提到的每种语言各编写一个 “Hello, World” 程序。
6. 对于本章提到的每种语言, 找到一本流行的教科书, 查找其中使用的第一个完整程序。用所有其他语言编写这个程序。警告: 这个练习的规模很容易达到 100 个程序。
7. 我们明显 “遗漏” 了很多重要的语言。特别是, 我们基本没有提及 C++ 之后程序设计语言的发展。列出你认为应该介绍的五种现代语言, 然后沿着本章语言部分的路线, 用一半的篇幅介绍其中三种。

8. C++ 有什么用处？为什么？撰写一份 10 至 20 页的报告。
9. C 有什么用处？为什么？撰写一份 10 至 20 页的报告。
10. 针对一种语言（不包括 C 和 C++），撰写一份 10 至 20 页的报告，描述这种语言的起源、目标和功能。给出足够的具体例子。介绍谁在使用这种语言，以及用它做什么。
11. 谁是剑桥大学现任的卢卡斯教授？
12. 本章提到的语言设计者中，谁拥有数学学位？谁没有？
13. 本章提到的语言设计者中，谁拥有博士学位？在哪个领域？谁没有博士学位？
14. 本章提到的语言设计者中，谁获得过图灵奖？图灵奖是什么？查找这些人是因何贡献而获奖。
15. 编写一个程序，输入一个由 (name, year) 值对构成的文件，例如 (Algol, 1960) 和 (C, 1974)，程序在时间轴上画出这些名字。
16. 修改上一个练习的程序，改为读取由 (name, year, (ancestors)) 三元组组成的文件，例如 (Fortran, 1956, ())、(Algol, 1960, (Fortran)) 和 (C++, 1985, (C, Simula))，在时间轴中画出这些语言，并在每个祖先和后代之间画一个箭头。用这个程序画出 22.2.2 和 22.2.7 节中的图表的改进版本。

## 附言

很明显，我们只是泛泛地介绍了程序设计语言的历史和如何开发更好软件的理念。我们认为历史和理念非常重要，能令你体会到什么是错误的。我们希望本章的内容已经传达出了一些令我们激动的东西，传达出了我们的观点——好的软件 / 好的程序设计方法这一问题浩瀚无边，程序设计语言的设计和实现已经充分表明了这一点。请记住，程序设计（开发高质量的软件）才是最基础、最重要的，程序设计语言只是工具。

# 文本处理

所谓显然的事情通常并非真的那么显然……  
使用“显然”这个词往往意味着缺乏逻辑论证。

——Errol Morris

本章主要介绍如何从文本中提取信息。我们将大量知识以单词的形式保存在文档中，例如书籍、电子邮件或者“打印”的表格，以便将来能从中提取出某种更易于计算的信息格式。在本章中，我们首先回顾标准库中最常用的文本处理功能：`string`、`iostream` 以及 `map`。然后，我们将介绍正则表达式（`regex`），它可以用来描述文本中的模式。最后，我们将展示如何使用正则表达式从文本中寻找和提取特定的数据元素（如邮政编码），以及如何验证文本文件的格式。

## 23.1 文本

从本质上来说，我们无时无刻不在处理文本。我们阅读的书籍中全都是文本，我们在电脑屏幕上看到的很多内容都是文本，我们处理的源代码也是文本。我们使用的（所有）通信信道充斥着文本。两个人之间的所有交流内容也都可以表示为文本。但我们不要走极端，图像和声音通常还是表示为二进制格式（即比特包）更好些。不过，对于几乎所有其他信息，都适合于使用计算机程序进行文本分析和转换。

从第 3 章开始，我们就已经看到了 `iostream` 和 `string` 的使用。因此，在本章中，我们只是简单回顾一下这些标准库中的语言特性。标准库中的“映射”（`map`，23.4 节）是一种非常实用的文本处理工具，我们将以电子邮件分析问题为例展示 `map` 的使用。在回顾了这些标准库特性后，我们将重点介绍如何使用正则表达式搜索文本中的模式（23.5 ~ 23.10 节）。

## 23.2 字符串

一个 `string` 包含一个字符序列，并提供了一些有用的操作，如：向 `string` 添加一个字符、获得 `string` 的长度以及 `string` 连接等。实际上，标准库中的 `string` 提供了很多操作，但其中大部分只用于相当复杂的低层方式的文本处理。在本章中，我们只简单回顾少数最常用的操作。如果需要的话，你可以在参考手册或者专业级的教材中查阅这些操作（以及全部 `string` 操作）的细节。这些操作的定义可以在 `<string>` 中找到（注意：不是 `<string.h>`）：

---

### 挑选出的一些字符串操作

---

<code>s1 = s2</code>	将 <code>s2</code> 的内容赋予 <code>s1</code> ， <code>s2</code> 可以是一个 <code>string</code> 或者 C 风格字符串
<code>s += x</code>	将 <code>x</code> 添加到 <code>s</code> 的末尾， <code>x</code> 可以是一个字符、一个 <code>string</code> 或者一个 C 风格字符串

---

(续)

## 挑选出的一些字符串操作

<code>s[i]</code>	下标
<code>s1+s2</code>	连接运算, 结果字符串的开始部分拷贝自 <code>s1</code> , 后接 <code>s2</code> 的拷贝
<code>s1==s2</code>	比较字符串的值, <code>s1</code> 或 <code>s2</code> 可以是 C 风格字符串, 但不允许两者皆是。!= 类似
<code>s1&lt;s2</code>	按字典序比较两个字符串的先后, 两者之一可以是 C 风格字符串, 但不允许两者皆是。 <=、> 和 >= 类似
<code>s.size()</code>	<code>s</code> 中字符的数目
<code>s.length()</code>	<code>s</code> 中字符的数目
<code>s.c_str()</code>	返回 <code>s</code> 中字符构成的 C 风格字符串
<code>s.begin()</code>	指向第一个字符的迭代器
<code>s.end()</code>	指向 <code>s</code> 末尾之后一个位置的迭代器
<code>s.insert(pos, x)</code>	将 <code>x</code> 插入到 <code>s[pos]</code> 之前的位置。 <code>x</code> 可以是一个 <code>string</code> 或者是 C 风格字符串。必要时会扩展 <code>s</code> 的存储空间来容纳 <code>x</code>
<code>s.append(x)</code>	将 <code>x</code> 插入到 <code>s</code> 之后的位置。 <code>x</code> 可以是一个 <code>string</code> 或者是 C 风格字符串。必要时会扩展 <code>s</code> 的存储空间来容纳 <code>x</code>
<code>s.erase(pos)</code>	删除 <code>s</code> 中从 <code>s[pos]</code> 开始的尾部字符。 <code>s</code> 的大小变为 <code>pos</code>
<code>s.erase(pos,n)</code>	删除 <code>s</code> 中从 <code>s[pos]</code> 开始的 <code>n</code> 个字符。 <code>s</code> 的大小变为 <code>max(pos,size-n)</code>
<code>pos=s.find(x)</code>	在 <code>s</code> 中查找 <code>x</code> 。 <code>x</code> 可以是一个字符、一个 <code>string</code> 或者 C 风格字符串。若找到, <code>pos</code> 的值为找到的子串的第一个字符的下标, 否则为 <code>string::npos</code> ( <code>s</code> 末尾之后的位置)
<code>in&gt;&gt;s</code>	从流 <code>in</code> 中读取一个词存入 <code>s</code> 中, 词以空白符间隔
<code>getline(in, s)</code>	从流 <code>in</code> 中读取一行存入 <code>s</code>
<code>out&lt;&lt;s</code>	将 <code>s</code> 的内容输出到 <code>out</code>

我们已经在第 10 章和第 11 章中介绍了 I/O 操作, 在 23.3 节中将进行小结。注意输入操作在必要时会扩展字符串的存储空间, 因此不可能发生溢出。

`insert()` 和 `append()` 操作会移动已有字符, 为新字符留出空间。`erase()` 操作则“向前”移动字符, 避免删除字符后在字符串中留下空洞。

标准库 `string` 实际上是一个称为 `basic_string` 的模板, 它支持多种字符集, 如 Unicode (在“普通字符”之外还提供了几千个特殊字符, 如 £、Ω、μ、δ、☺ 以及 ♪)。例如, 如果你需要声明一个保存 Unicode 字符的类型, 其名字就叫作 `Unicode`, 可以使用如下代码:

```
basic_string<Unicode> a_unicode_string;
```

我们之前所使用的标准字符串类 `string`, 实际上就是保存普通字符的 `basic_string`:

```
using string = basic_string<char>; // string 即 basic_string<char> (见 15.5 节)
```

本章不会介绍 Unicode 字符或者是 Unicode 字符串, 如果你需要使用这些功能的话, 可以查阅相关资料, 你会发现其使用方法(从语言的角度, 以及从 `string`、`iostream` 和正则表达式的角度)与普通字符和普通字符串是一致的。如果你需要使用 Unicode 字符, 最好求教于有经验的人。编写这类程序, 除了程序设计语言方面的考虑外, 还要遵循系统方面的惯例。



在文本处理语境中，几乎所有内容都可以表示为字符串，这是很重要的。例如，在本页中，数字 12.333 表示为 6 个字符的字符串（前后都有空白符）。如果我们要读取这个数，就必须将这 6 个字符转换为一个浮点型数，然后才能进行算术运算。这就要求提供两个方向的转换功能：值转换为 string 和 string 转换为值。在 11.4 节中，我们已经看到了如何利用 `ostringstream` 将一个整数转换为一个 string。这种技术可以推广到任何具有 << 操作符的类型：

```
template<typename T> string to_string(const T& t)
{
    ostringstream os;
    os << t;
    return os.str();
}
```

其使用方式如下例：

```
string s1 = to_string(12.333);
string s2 = to_string(1+5*6-99/7);
```

这两行代码执行后，s1 的值变为 "12.333"，s2 的值变为 "17"。实际上，`to_string()` 不仅可以用于数值类型，还可用于其他任何具有 << 操作符的类型 T。与之相反的转换，也就是从 string 到数值的转换，也同样简单、有用：

```
struct bad_from_string : std::bad_cast { // 报告字符串转换错误的类
    const char* what() const override
    {
        return "bad cast from string";
    }
};

template<typename T> T from_string(const string& s)
{
    istringstream is {s};
    T t;
    if (!(is >> t)) throw bad_from_string{};
    return t;
}
```

使用示例如下：

```
double d = from_string<double>("12.333");

void do_something(const string& s)
try
{
    int i = from_string<int>(s);
    // ...
}
catch (bad_from_string e) {
    error("bad input string",s);
}
```

与 `to_string()` 相比，`from_string()` 要稍微复杂一些：同一个字符串所表示的值，可以解释为很多类型。这也意味着，我们所看到的字符串内容，未必表示我们所期待的类型的值。例如：

```
int d = from_string<int>("Mary had a little lamb");    // 糟糕！
```

这样就可能引起错误，我们用异常 `bad_from_string` 来表示这类错误。在 23.9 节中，我们将说明为什么 `from_string()`（或等价的函数）对文本处理那么重要，其原因就在于我们需要从文本域中提取数值。在 21.4.3 节中，我们已经看到一个类似的函数 `get_int()` 在 GUI 代码中的应用。

注意，`to_string()` 和 `from_string()` 这两个函数是非常相似的。实际上，它们大致互为逆操作。这样，对于所有“适当的类型 T”，我们有如下结论（忽略空白符、舍入等细节）：

```
s==to_string(from_string<T>(s))    // 对所有 s
```

以及

```
t==from_string<T>(to_string(t))    // 对所有 t
```

这里，“适当”的意思是指 T 应该具有默认构造函数、`>>` 操作符以及一个匹配的 `<<` 操作符。

请注意，`to_string()` 和 `from_string()` 的实现中都使用了一个 `stringstream` 来完成所有困难的工作。实际上，对于任何具有匹配的 `<<` 和 `>>` 操作符的类型，我们都可以利用这种技术手段来实现类型之间的转换操作。

```
template<typename Target, typename Source>
Target to(Source arg)
{
    stringstream interpreter;
    Target result;

    if (!(interpreter << arg)                // 将 arg 写入到流
        || !(interpreter >> result)         // 从流读取 result
        || !(interpreter >> std::ws).eof())  // 流中还有内容?
        throw runtime_error{"to<>() failed"};

    return result;
}
```

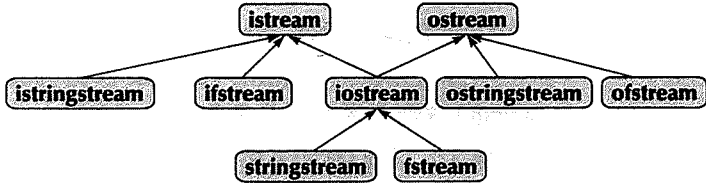
`!(interpreter >> std::ws).eof()` 看起来有些怪，但实际上是很巧妙的，它将提取数据后可能遗留在 `stringstream` 中的空白符读取出来。空白符是允许出现的，但在它之后不允许再有任何字符，我们可以通过查看是否到达“文件尾”来检测这一情况。这样，当我们试图从一个 `string` 对象中读取一个 `int` 值时，`to<int>("123")` 和 `to<int>("123 ")` 都会成功读取数值，而 `to<int>("123.5")` 会失败，因为最后有一个 `.5`。

## 23.3 I/O 流

如上节所示，我们利用 I/O 流在字符串和其他类型间建立起联系。I/O 流库不仅仅可以实现输入和输出，它还可以实现字符串格式和内存中类型之间的转换。标准库 I/O 流提供了对字符串进行读、写和格式化的功能。我们已经在第 10 章和第 11 章介绍了 `iostream` 库，因此现在只是简单小结一下：

流 I/O	
<code>in &gt;&gt; x</code>	根据 x 的类型从 in 读取数据存入 x
<code>out &lt;&lt; x</code>	根据 x 的类型将其内容写入 out
<code>in.get(c)</code>	从 in 读取一个字符存入 c
<code>getline(in, s)</code>	从 in 读取一行内容存入字符串 s

标准流的类层次如下图所示（见 19.3 节）：



这些类一起构成了标准流库，使我们可以对文件和字符串进行 I/O 操作（还可对其他任何可以看作文件或字符串的对象如键盘、屏幕等进行 I/O 操作，参见第 10 章）。而且，如第 10 章和第 11 章所述，`iostream` 还提供了精心设计的格式化机制。上图中，箭头表示继承关系（见 19.3 节），因而，一个 `stringstream` 对象可以作为一个 `iostream` 或者 `istream` 或者 `ostream` 来使用。

与 `string` 类似，`iostream` 可以使用 Unicode 这样的大字符集，用法与普通字符集一致。我们再次提醒，如果你需要使用 Unicode I/O，最好求助于有经验的人。为了正确使用 Unicode，你的程序除了要考虑程序设计语言方面的问题，还要遵循系统方面的惯例。

## 23.4 映射

关联数组（映射、哈希表）是很多文本处理的关键（`key` 为双关语，有“关键”之意，也有“关键字”之意）。原因很简单：当我们处理文本时，主要工作就是收集信息，而信息通常与文本字符串（如名字、地址、邮政编码、社会保险号、职位等）相关联。即使某些文本字符串可以转换为数值，但通常更方便也更简单的方式还是将它们按文本来处理，并使用此文本作为信息的标识。16.6 节中的单词计数的例子是一个很好的简单示例。如果你还不太习惯使用 `map`，请重新阅读 16.6 节，然后再继续学习本节的内容。

下面我们以电子邮件问题为例来讨论 `map` 在文本处理中的应用。我们常常需要搜索和分析电子邮件消息和邮件日志，这一般要借助一些专门程序（如 Thunderbird 或者 Outlook）来完成。利用这些程序，我们不必查看海量的完整邮件内容，就能找到所需的信息。然而，通常我们要查找的信息，如发件人、收件人、发送路径以及其他更多的内容，都是以邮件头中文本的形式呈现给邮件程序的。邮件头已是一条完整的消息，有成千上万的工具用来分析它。这些工具中的大多数都利用正则表达式（将在 23.5 ~ 23.9 节中进行详细介绍）提取信息，并用某种形式的关联数组关联相关的邮件。例如，我们常常需要查找某个人发送来的所有邮件，或者同一主题的所有邮件，或者内容与特定主题相关的所有邮件。

在这里，我们使用一个非常简单的邮件文件来展示一些从文本文件中提取数据的技术。这个邮件文件的头是来自于 [www.faqs.org/rfcs/rfc2822.html](http://www.faqs.org/rfcs/rfc2822.html) 的实际的 RFC2822 头。如下所示：

```

xxx
xxx
-----
From: John Doe <jdoe@machine.example>
To: Mary Smith <mary@example.net>
Subject: Saying Hello
Date: Fri, 21 Nov 1997 09:55:06 -0600
Message-ID: <1234@local.machine.example>
  
```

This is a message just to say hello.  
 So, "Hello".

-----  
**From:** Joe Q. Public <john.q.public@example.com>  
**To:** Mary Smith <@machine.tld:mary@example.net>, , jdoe@test .example  
**Date:** Tue, 1 Jul 2003 10:52:37 +0200  
**Message-ID:** <5678.21-Nov-1997@example.com>

Hi everyone.

-----  
**To:** "Mary Smith: Personal Account" <smith@home.example>  
**From:** John Doe <jdoe@machine.example>  
**Subject:** Re: Saying Hello  
**Date:** Fri, 21 Nov 1997 11:00:00 -0600  
**Message-ID:** <abcd.1234@local.machine.tld>  
**In-Reply-To:** <3456@example.net>  
**References:** <1234@local.machine.example> <3456@example.net>

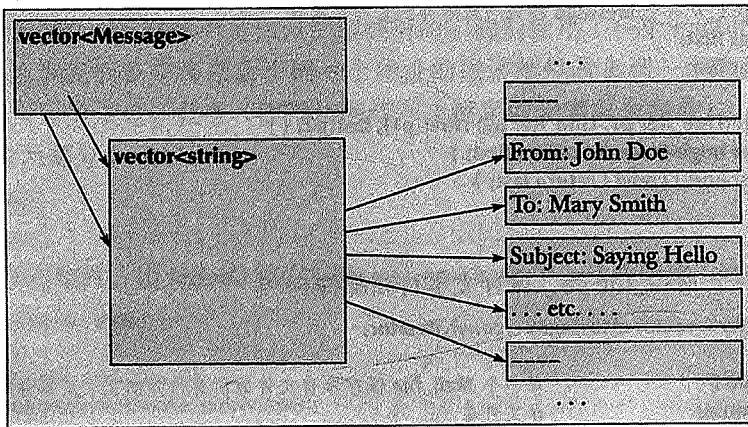
This is a reply to your reply.

为简单起见，我们已经丢弃了邮件文件中的大部分内容。我们还在每个邮件之后添加了一行“-----”来标识邮件的结束，这进一步简化了邮件的分析。我们将会写一个简短的“玩具程序”，它查找所有“John Doe”发送的邮件，并输出这些邮件的主题。这个程序虽然简短，但其中的技术可以用来完成很多更复杂、更有趣的任务。

首先，我们要确定是要随机访问数据，还是通过一个输入流以流方式分析数据。我们选择第一种方式，因为在一个实际程序中，我们可能只关心某几个发件人，或者来自于一个发件人的某些信息。而且，相对于流式访问，随机访问确实更困难些，因此我们可以学习更多技术。特别地，我们将再次使用迭代器。

我们的基本思路是读取一个完整的邮件文件，将其内容保存到一个称为 Mail\_file 的数据结构中。这个数据结构用一个向量 `vector<string>` 保存邮件文件的所有文本行，并通过另一个向量 `vector<Message>` 指明每个邮件的起止位置。

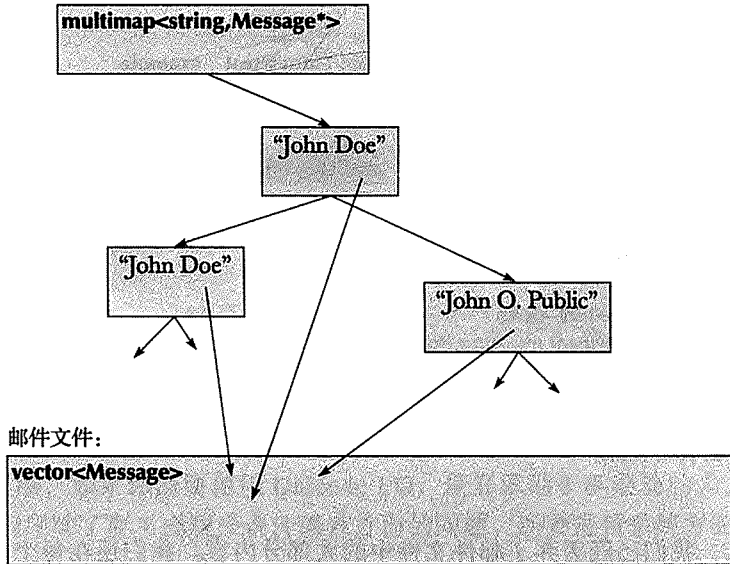
邮件文件:



为此，我们将加入迭代器和 `begin()`、`end()` 函数，以便能有一种一致的方法遍历所有行和所有邮件。通过这个“样板代码”，我们可以方便地访问邮件。完成它之后，我们就可以



编写“玩具程序”了，这个程序将每个发件人的所有邮件收集在一起，以方便访问。



最终，我们输出所有发自“John Doe”的邮件的主题内容，以此来展示如何使用这套玩具程序来处理邮件。

编写这个程序需要使用很多基础的标准库功能：

```

#include<string>
#include<vector>
#include<map>
#include<fstream>
#include<iostream>
using namespace std;

```

我们将 Message 定义为 vector<string>（保存文本行的向量）的一对迭代器。

```
typedef vector<string>::const_iterator Line_iter;
```

```

class Message { // 一个 Message 指向一封邮件的首行和末行
    Line_iter first;
    Line_iter last;
public:
    Message(Line_iter p1, Line_iter p2) :first{p1}, last{p2} {}
    Line_iter begin() const { return first; }
    Line_iter end() const { return last; }
    // ...
};

```

我们定义一个 Mail\_file 类，保存文本行和邮件：

```
using Mess_iter = vector<Message>::const_iterator;
```

```

struct Mail_file {
    string name; // 一个 Mail_file 保存来自文件的所有行并简化了对邮件的访问
                // 文件名
    vector<string> lines; // 按顺序保存的行
    vector<Message> m; // 按顺序保存的邮件

    Mail_file(const string& n); // 读取文件 n，保存到 lines 中

```

```

    Mess_iter begin() const { return m.begin(); }
    Mess_iter end() const { return m.end(); }
};

```

注意我们是如何将迭代器加入数据结构中的，以便能更容易地、更有条理地访问数据结构的内容。我们这里并没有真正使用标准库算法，但由于使用了迭代器，程序很容易改为使用标准库算法的版本。

为了在邮件中查找和提取信息，我们需要两个辅助函数：

```

// 在一个 Message 中查找发件人姓名
// 若找到返回 true
// 若找到，将发件人姓名放入 s 中
bool find_from_addr(const Message* m, string& s);

// 如果有的话，返回 Message 的主题，否则返回 ""
string find_subject(const Message* m);

```

最后，我们就可以编写从文件提取信息的代码了：

```

int main()
{
    Mail_file mfile ("my-mail-file.txt");    // 从一个文件读取数据初始化 mfile

    // 首先将来自每个发件人的邮件收集在一起，保存在一个 multimap 中

    multimap<string, const Message*> sender;

    for (const auto& m : mfile) {
        string s;
        if (find_from_addr(&m,s))
            sender.insert(make_pair(s,&m));
    }

    // 现在遍历 multimap
    // 并提取 John Doe 的邮件的主题
    auto pp = sender.equal_range("John Doe <jdoe@machine.example>");
    for(auto p = pp.first; p!=pp.second; ++p)
        cout << find_subject(p->second) << '\n';
}

```

我们来仔细分析一下程序中是如何使用映射的。我们使用了一个 multimap（见 15.10 节和附录 C.4），因为我们希望将来自于同一个地址的很多邮件收集到一个地方存储。标准库的 multimap 就可以完成这一任务（使得访问相同关键字的元素更为容易）。显然，我们的工作（通常）分为两部分：

- 创建 map。
- 使用 map。

我们遍历所有邮件，用 insert() 将它们插入到 multimap 中，从而创建了 multimap 的内容：

```

for (const auto& m : mfile) {
    string s;
    if (find_from_addr(&m,s))
        sender.insert(make_pair(s,&m));
}

```

插入到 map 中的内容都是我们用 make\_pair() 构造的（关键字，值）对。我们使用“自制的” find\_from\_addr() 函数查找发件人的姓名。

在程序中，为什么我们首先将 Message 对象存于一个 vector 中，然后又通过这个 vector 创建了一个 multimap 呢？为什么我们不直接将 Message 对象存入 multimap 中呢？原因很简单，这也是一个数据处理的基本原则：

- 首先，我们创建一个通用的数据结构，可利用它来完成很多工作。
- 然后，我们转为特定的应用。

通过这种方法，我们可以创建一些或多或少可重用的组件。反之，如果我们直接在 Mail\_file 中创建一个 map 的话，在希望完成其他任务时，可能就需要对其进行重新定义。特别是，我们的 multimap 对象（称为 sender）是按照邮件的地址域进行排序的。而对于其他很多应用来说，这种顺序可能毫无意义，它们可能关心返回地址、收件人、抄送地址、主题、时间戳等。

这种逐步（或者说逐层（layer））创建应用程序的方法，可以极大地简化设计、实现、文档和维护工作。关键点在于每个部分都只做一件事，而且是以一种简单直接的方法去做。与之相对的方法是，将所有事情都放在一起做，这种方法需要极高的聪明才智。显然，我们这个“从一个电子邮件头中提取信息”的小程序，只是逐层构造方法的一个很简单的应用。逐层构造方法的保持独立组件相分离、模块化以及渐进构造的思想，会随着问题规模增大体现出更大的价值。

为了提取所需信息，我们简单地使用 equal\_range() 函数查找所有包含关键字“John Doe”的条目（见附录 C.4.10）。然后遍历它返回的序列 [first, second) 中的所有元素，利用 find\_subject() 提取主题域：

```
auto pp = sender.equal_range("John Doe <jdoe@machine.example>");

for (auto p = pp.first; p!=pp.second; ++p)
    cout << find_subject(p->second) << '\n';
```

当我们遍历一个 map 的元素时，我们得到一个（关键字，值）对的序列。对于每个对，第一个元素（本例中是 string 类型的关键字）称为 first，第二个元素（本例中是 Message 类型的值）称为 second（见 16.6 节）。

### 23.4.1 实现细节

显然，我们需要实现前面程序中所使用的函数。我们可以将这个工作留作练习，以节省版面。但我们决定给出这些实现细节，以使这个例子更加完整。Mail\_file 的构造函数打开邮件文件，并构造 lines 和 m 两个向量：

```
Mail_file::Mail_file(const string& n)
    // 打开名为 n 的文件
    // 从 n 读取文本行存入 lines
    // 在 lines 中查找邮件，将它们组合起来存入 m
    // 简单起见，假定每个邮件以一行 --- 结束
{
    ifstream in {n};           // 打开文件
    if (!in) {
        cerr << "no " << n << '\n';
        exit(1);              // 终止程序
    }

    for (string s; getline(in,s);) // 创建文本行的 vector
        lines.push_back(s);
```

```

auto first = lines.begin();    // 创建 Message 的 vector
for (auto p = lines.begin(); p!=lines.end(); ++p) {
    if (*p == "-----") {    // 邮件结束
        m.push_back(Message(first,p));
        first = p+1;        // -----不是邮件的一部分
    }
}
}

```

这段程序中的错误处理代码是不完整的。如果我们希望这个程序真正被他人使用，还需要进一步完善。

### 试一试

我们真正在意的是：真正运行这个例子，并且确保你理解了运行结果。什么样的错误处理代码才算是“更好的”？修改 `Mail_file` 的构造函数，处理与“-----”相关的可能的格式错误。

下面是 `find_from_addr()` 和 `find_subject()` 的实现，当前的版本只是简单地占据位置而已，当我们能更好地从文件中识别信息时（使用正则表达式，见 23.6 ~ 23.10 节），将给出更好的实现方式。

```

int is_prefix(const string& s, const string& p)
    // p 是 s 的第一部分？
{
    int n = p.size();
    if (string(s,0,n)==p) return n;
    return 0;
}

bool find_from_addr(const Message* m, string& s)
{
    for (const auto& x : m)
        if (int n = is_prefix(x, "From: ")) {
            s = string(x,n);
            return true;
        }
    return false;
}

string find_subject(const Message* m)
{
    for (const auto& x : m)
        if (int n = is_prefix(x, "Subject: ")) return string(x,n);
    return "";
}

```

请注意我们使用子串的方式：`string(s, n)` 构造了一个包含 `s[n]..s[s.size()-1]` 之间字符的子串，而 `string(s, 0, n)` 则构造了一个包含 `s[0]..s[n-1]` 之间字符的子串。由于这些操作会创建新的字符串并进行字符拷贝，因此在使用上必须注意性能问题。✘

为什么 `find_from_addr()` 和 `find_subject()` 如此不同？比如，一个返回 `bool` 值，而另一个返回一个 `string`。原因在于我们想说明：✘

- `find_from_addr()` 应该区分有地址行但内容为空（""）和无地址行两种不同的情况。对于第一种情况，`find_from_addr()` 返回 `true`（因为找到了地址行）并将 `s` 置为空字符串

"" (因为地址为空)。而对于第二种情况, 应该返回 `false` (因为没有地址行)。

- 对于主题为空或者没有主题行的情况, `find_subject()` 都返回 ""。

`find_from_addr()` 将两种情况区分开来, 这是否有意义呢? 是否必要呢? 我们认为是有意义的, 而且绝对是必要的。当在数据文件中查找信息时, 会频繁出现这种不同情况间的细微差别: 我们是否找到了希望查找的域? 这个域中的内容是否有用? 在一个实际的程序中, `find_from_addr()` 和 `find_subject()` 都应该按照现在的 `find_from_addr()` 的风格来设计, 以使用户能区分这种差别。

现在这个版本的程序还没有进行过性能优化, 但对于大多数应用场景来说, 应该已经足够快了。特别是它对输入文件只读取一次, 而且对于文件中的文本也不会保存多份拷贝。对于大文件, 用 `unordered_multimap` 替换 `multimap` 可能会提高性能, 但这需要经过验证, 否则不能下定论。

如果你对标准库中的关联容器 (`map`、`multimap`、`set`、`unordered_map` 和 `unordered_multimap`) 还不太熟悉的话, 请参考 16.6 节。

## 23.5 一个问题

I/O 流和 `string` 可以帮助我们读写、存储字符序列, 并对其进行一些基本操作。但是, 在应用中一个非常常见的问题是: 对于要处理的文本, 我们需要考虑其中包含一个特定字符串或多个相似字符串的情况。我们来看一个很简单的例子: 查找一个电子邮件 (一个单词序列) 中是否包含美国某些州的邮政编码 (州名的缩写加上编码——两个字母后接五个数字)

```
for (string s; cin>>s; ) {
    if (s.size()==7
        && isalpha(s[0]) && isalpha(s[1])
        && isdigit(s[2]) && isdigit(s[3]) && isdigit(s[4])
        && isdigit(s[5]) && isdigit(s[6]))
        cout << "found " << s << "\n";
}
```

其中 `isalpha(x)` 判断 `x` 是否是字母, `isdigit(x)` 判断 `x` 是否是数字 (见 11.6 节)。

这个简单 (过于简单了) 的解决方案存在若干问题:

- 它太冗长了 (4 行代码, 8 个函数调用)。
- 我们忽略了 (有意的?) 所有没有用空白符将自身与上下文分开的邮政编码 (如 "TX77845", TX77845-1234 以及 ATX77845)。
- 我们忽略了 (有意的?) 所有用空白符分隔州名缩写和数字的邮政编码 (如 TX 77845)。
- 我们将州名缩写为小写的字符串也识别为合法的邮政编码 (有意的?) (例如 tx77845)。
- 如果我们希望查找完全不同格式的邮政编码 (例如 CB30FD), 不得不重写全部代码。应该还有更好的解决方案! 但在设计新的方法之前, 我们来分析一下, 当处理更复杂的情况时, 如果还使用“简单有效的旧方法”来编写代码, 会遇到哪些问题:
  - 如果希望处理多种格式, 我们不得不增加 `if` 语句和 `switch` 语句。
  - 如果希望既能处理大写, 又能处理小写, 我们不得不进行显式的大小写转换 (通常转换为小写) 或者再加入 `if` 语句。

- 我们希望能以某种形式(何种形式?)来描述我们要查找的上下文。这意味着我们必须处理单个字符而不是字符串,而且失去了 `istream` 提供的很多优点(见 7.8.2 节)。

如果你愿意,可以使用旧方法实现上述功能。但显然,沿着这条路走下去,程序中会充斥着大量 `if` 语句,来处理大量的特殊情况。即便是前面那个简单的例子,我们也需要处理一些不同选项(例如,同时支持五位和九位数字的编码)。对于其他很多实例,我们都需要处理上下文中的重复(例如,任意多个数字后接一个感叹号,如 123! 和 123456!)。另外,我们还必须处理前缀和后缀。就像 11.1 ~ 11.2 节中讨论的那样,人们对输出格式的偏好是不会被程序员对规律性和简洁性的追求所局限的。你只要思考一下人们书写日期的五花八门的格式,就会赞同这一观点:

```
2007-06-05
June 5, 2007
jun 5, 2007
5 June 2007
6/5/2007
5/6/07
...
```

此时,甚至更早,一个有经验的程序员就会下结论“一定有更好的解决方法!”,然后就去寻找新的方法了(而不会再用旧方法编写代码)。解决文本上下文搜索问题的一个最简单而且最通行的方法是使用所谓的正则表达式(regular expression)。正则表达式是大量文本处理的基础,Unix 的 `grep` 命令就是基于正则表达式的(见习题 8),很多文本处理语言(如 AWK、PERL 以及 PHP)也都将支持正则表达式作为必备的功能。

我们借助一个库来实现基于正则表达式的搜索。这个库是 C++ 标准库的一部分。它与 PERL 中的正则表达式处理程序是兼容的。这样,PERL 正则表达式处理的大量文档、教程和手册就都可以拿来作为参考。例如,可以参考 C++ 标准委员会的工作文档(在互联网上搜索“WG21”即可找到),或者是 John Maddock 写的 `boost::regex` 的文档,以及大多数 PERL 的教程。在本章中,我们会介绍正则表达式的一些基本概念和基本使用方法。

## 试一试

前面两个段落“粗心地”使用了几个你未见过的名词和缩写,而未加解释。请在互联网上搜索这些名词,了解它们的含义。

## 23.6 正则表达式的思想

正则表达式的基本思想是:定义一个模式,在文本中搜索这个模式。我们以邮政编码问题为例,看看对于 TX77845 这样的邮政编码,如何定义模式。下面是第一个尝试:

```
wwdddd
```

其中 `w` 表示“任意字母”,`d` 表示“任意数字”。这里使用 `w` (“word”之意)而不是 `l` (“letter”),是因为 `l` 太容易与数字 1 混淆了。对于本例,这种符号表示是没有问题的,但我们还是来看一看它对于更复杂的情况,如 9 位数字的邮政编码格式(TX77845-5629),是否还有效:

```
wwdddd-dddd
```

✂ 这个模式看起来没有什么问题，但 `d` 如何就能表示“任意数字”，而“-”就表示它的字面意思——“普通破折号”呢？不管怎样，我们确实应该指出 `w` 和 `d` 具有特殊含义：它们表示字符集而非字符的字面含义（`w` 表示“一个 `a` 或 `b` 或 `c` 或…”），`d` 表示“一个 `1` 或 `2` 或 `3` 或…”）。这有些过于微妙了，更好的表示方式是在这种表示一类单词的字母之前加上一个反斜线符号，以此来指出这是一个特殊符号，而不是字面含义，这与 C++ 语言中区分特殊符号的方式是相同的（例如，`\n` 表示换行）。于是模式变为：

```
\w\d\d\d\d\d-\d\d\d\d
```

新的模式看起来有些丑陋，但至少不会引起歧义，而且反斜线符号显然具有一种“这是不寻常内容”的意味。在这里，我们表示一个字符多次重复的方式就是简单地重复几次。这样做令人厌烦，而且容易出错。我们真的在破折号之前获取了 5 个数字，而在其后获取了 4 个数字吗？答案是肯定的。但我们自始至终没有明确地表达 5 和 4，而是需要通过手工计数来保证数量正确。更好的方法是在字符之后用一个数值表示重复的次数，例如：

```
\w2\d5-\d4
```

我们同样应该定义某种语法来说明 2、5 和 4 是计数值，而不是表示文本中应该出现这几个数字。我们将计数值放在花括号中来表示这种含义：

```
\w{2}\d{5}-\d{4}
```

这种语法使得 `{` 成为与 `\` 一样的特殊字符，但这是不可避免的。而且，当我们需要表示文本中出现花括号和反斜线符号的时候，也有办法处理。

到目前为止，看起来还不错，但我们还需要解决两个更为棘手的细节：最后 4 个数字是可选的。我们设计的模式应该既能接受 `TX77845`，也能接受 `TX77854-5629`。解决这一问题有两种方式，一种是：

```
\w{2}\d{5} 或 \w{2}\d{5}-\d{4}
```

另一种是：

```
\w{2}\d{5} 和 可选的 -\d{4}
```

✂ 为了准确地描述这两种方式，我们首先需要有一种能表达分组（子模式）的语法，用来描述 `\w{2}\d{5}` 和 `-\d{4}` 是 `\w{2}\d{5}-\d{4}` 的两个组成部分。习惯上，我们用括号来表示分组的概念：

```
(\w{2}\d{5})(-\d{4})
```

我们现在已经将模式划分为两个子模式了，接下来就可以描述我们最初的意图了——第二部分是可选的。与前面一样，新功能的引入伴随着新的特殊符号的引入，描述子模式的“`(`”现在与 `\` 和 `{` 一样“特殊”了。我们引入两个新的特殊符号：“`|`”用来表达“或”（两个子模式二选一）的概念，“`?`”用来表达某个子模式可选（有或无）的概念。于是，邮政编码后 4 位数字可选的第一种表示方式为：

```
(\w{2}\d{5})|(\w{2}\d{5}-\d{4})
```

第二种表示方式为：

```
(\w{2}\d{5})(-\d{4})?
```

与花括号表示计数一样（如 `\w{2}`），我们用问号（`?`）做后缀来表示可选的概念。例如

`(-\d{4})?` 表示 “`-\d{4}` 是可选的”。也就是说，可以接受以一个破折号接 4 个数字为后缀的邮政编码；当然，没有这样后缀的编码也是接受的。实际上，5 位数字邮政编码 (`\w{2}\d{5}`) 两边的括号没有任何作用，可以将其去掉：

```
\w{2}\d{5}(-\d{4})?
```

为了与 23.5 节中提出的需求完全吻合，我们在开始的两个字母之后加上一个可选的空格：

```
\w{2} ?\d{5}(-\d{4})?
```

“?” 看起来有点怪，它实际就是一个空格后接一个 ?，表示空格是可选的。如果你不想用这么突兀的形式，可以把空格放在括号中：

```
\w{2}( ?)\d{5}(-\d{4})?
```

如果还是觉得比较含糊，可以引入一个新的特殊符号来表示空格符，如 `\s` (`s` 表示空格 (space))。于是模式变为：

```
\w{2}\s?\d{5}(-\d{4})?
```

看起来已经达到最初的要求了，但是，如果有人在开头的两个字母之后写了两个空格，会发生什么情况呢？按现在的定义，模式会接受 `TX77845` 和 `TX 77845`，但不接受 `TX 77845`，这显然不合要求。我们需要一种语法来描述 “0 或多个空格符”，我们引入特殊符号 “\*”，以它作为后缀就表示 “0 或多个” 的含义。模式可以写为：

```
\w{2}\s*\d{5}(-\d{4})?
```

如果你随着本节的讲述一步步地走过来，会觉得这个最终的正则表达式很合理。这种表示模式的方法符合逻辑而且非常简洁。而且，我们并非随意地选择了一些描述方法，本节所介绍的符号表示都是非常普遍和通行的。对于很多文本处理工作，你必须编写、阅读这种符号表示。当然，这种描述方法看起来有些古怪，好像是你家的小猫在键盘上散步所产生的符号串。而且，采用这样的描述方法，敲错任何一个符号（甚至是一个空格）都可能完全改变其含义。但是，请尽量熟悉它。我们无法找到任何一种其他描述方法，能明显优于正则表达式。而且，这种描述风格自 30 年前由 Unix 的 `grep` 命令引入后，一直流行到现在，甚至从未进行过大的修改。

### 23.6.1 原始字符串常量

注意所有正则表达式模式中的反斜杠。要在 C++ 的字符串常量中加入一个反斜杠 (`\`)，我们需要在前面再加一个反斜杠，以上述邮编为例：

```
\\w{2}\\s*\\d{5}(-\\d{4})?
```

要把这一表达式表示成为一个字符串常量，我们需要写成如下形式：

```
"\\w{2}\\s*\\d{5}(-\\d{4})?"
```

再多考虑一点，很多我们使用的表达式都可能包括双引号 (`"`)。要在一个字符串常量中包含双引号，我们也需要在其前面加上反斜杠。这很快就变成不可控了。事实上，“特殊字符问题”在 C++ 和其他编程语言中都是令人厌烦的问题。因此，我们引入这原始字符串常量来处理正则表达式。在原始字符串常量中，反斜杠就是一个反斜杠（而不是转义符），双引号就是双引号（而不表示字符串的结束）。使用原始字符串常量的邮编例子如下：



```
R"(\w{2}\s*\d{5})(-\d{4})?"
```

这里 R" 表示字符串的开始，)" 表示字符串结束。这个字符串的 22 个字符如下：

```
\w{2}\s*\d{5})(-\d{4})?
```

不包括结束字符 0。

## 23.7 用正则表达式进行搜索

现在，我们可以使用上节定义的邮政编码的模式来搜索文件中的邮政编码了。程序先定义模式，然后逐行读取文件，在其中搜索模式。如果在某行中找到了模式，则输出行号：

```
#include <regex>
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int main()
{
    ifstream in {"file.txt"};           // 输入文件
    if (!in) cerr << "no file\n";

    regex pat {R"(\w{2}\s*\d{5})(-\d{4})?"}; // 邮政编码模式
    int lineno = 0;
    for (string line; getline(in,line);) { // 从输入读取文本行存入输入缓冲区
        ++lineno;
        smatch matches;                  // 匹配的字符串会保存在这里
        if (regex_search(line, matches, pat))
            cout << lineno << ": " << matches[0] << "\n";
    }
}
```

程序的一些细节需要解释一下。我们使用了标准正则表达式库 <regex>，利用它，我们可以定义一个模式 pat：

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

✂ 一个 regex 模式实际上也是一个 string，因此我们可以用字符串来初始化它。在这里，我们使用的是原始字符串常量。但是，一个 regex 不仅仅是一个字符串，还是一种复杂的模式匹配机制。当初始化一个 regex 变量时，这个机制就建立起来了。这种复杂机制已经超出了本书的讨论范围，但我们无须了解这些，我们只要知道，一旦用上节定义的模式初始化了一个 regex 变量，我们就可以用它在文件的每一行中搜索邮政编码了：

```
smatch matches;
if (regex_search(line, matches, pat))
    cout << lineno << ": " << matches[0] << "\n";
```

regex\_search(line, matches, pat) 搜索 line 中与正则表达式 pat 匹配的内容，如果找到，则将结果保存在 matches 中。如果未找到匹配内容，返回 false。

✂ 变量 matches 的类型是 smatch，前缀 s 表示“子（匹配）”（sub）或字符串（string）。一个 smatch 本质上是一个子匹配（类型为 string）的向量。第一个元素（本例中为 matches[0]）是完整匹配。如果 i < matches.size()，我们可以将 matches[i] 当作一个字符串。对于一个正则表达式，如果最多有 N 个子模式，则 matches.size() == N + 1。

✂ 那什么是子模式呢？一个较好的初步的回答是：“模式中任何放在括号中的内容都是一

个子模式。”如模式“`\w{2}s*d{5}(-d{4})?`”中，我们唯一能看到的子模式是四位扩展数字，因为它是在括号中的，因此我们猜测（实际就是这样）`matches.size()==2`。这样，我们猜测可以通过 `matches` 很容易地访问后四位数字，如下面代码所示：

```
for (string line; getline(in,line); ){
    smatch matches;
    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
    }
}
```

严格来说，我们不必测试 `1<matches.size()`，因为我们已经知道模式的详细结构。但最好还是加上这个检测（因为我们已经试验过 `pat` 中多种不同的模式，并非所有模式都恰好有一个子模式）。我们可以通过 `matches` 中的对位元素，来判断一个子模式是否匹配成功。在本例中，是通过 `matches[1].matched` 来判断的。当 `matches[i].matched` 为假时，即子模式未匹配时，`matches[i]` 的内容会是一个空字符串。类似地，不存在的子模式（如对本例的模式访问 `matches[17]`）会按未匹配来处理。

对包含如下内容的文件测试我们的程序：

```
address TX77845
ffff tx 77843 asasasaa
ggg TX3456-23456
howdy
zzz TX23456-3456sss ggg TX33456-1234
cvzcv TX77845-1234 sdsas
xxxTx77845xxx
TX12345-123456
```

得到如下输出结果：

```
pattern: "\w{2}s*d{5}(-d{4})?"
1: TX77845
2: tx 77843
5: TX23456-3456
   : -3456
6: TX77845-1234
   : -1234
7: Tx77845
8: TX12345-1234
   : -1234
```

注意：

- 我们未被 `ggg` 那行中错误的格式所欺骗（它错在哪里？）。
- 在 `zzz` 那行中，我们只找到了第一个邮政编码（本来就是要求每行只找一个）。
- 在第 5 行和第 6 行中我们正确地找到了后缀形式的编码。
- 我们找到了第 7 行中“隐藏”在 `xxx` 中的编码。
- 我们找到了隐藏在 `TX12345-123456` 中的编码（这样做是否不正确？）。

## 23.8 正则表达式语法

上一节介绍了一个较为简单的正则表达式匹配的例子。下面我们更为系统、完整地介绍一下正则表达式（以 `regex` 库为线索来介绍）。

✂ 正则表达式 (regular expression, 简称正则式, “regexp” 或 “regex”) 实际上是一种表达字符模式的语言, 只不过这种语言的规模很小。它是一种强大 (表达能力强) 而简洁的语言, 而又有有些神秘。经过几十年的使用, 产生了很多微妙的特性和“方言”。在本章中, 我们只介绍它的一个子集, 这也是使用最为广泛的一支方言 (PERL)。如果你希望了解更多的特性以便表达更复杂的模式, 或者你希望了解其他方言, 请搜索互联网。网络上相关的学习指南 (质量差异很大) 俯拾皆是。

☛ regex 库还支持 ECMAScript、POSIX、awk、grep 和 egrep 表示法和许多搜索选项。这是非常有用的, 特别是当你需要使用的正则式是用其他语言设计的时候。如果你需要了解这些额外的特性, 可以查阅相关资料。不过, 请记住, “使用最多的特性” 不是一个好的程序设计风格。无论什么时候, 都请替可怜的程序维护人员着想 (很有可能就是你自己), 他需要阅读并理解你的代码: 因此编写代码时不要炫耀你的聪明, 并且避免使用那些晦涩难懂的特性。

### 23.8.1 字符和特殊字符

一个正则式描述了一个模式, 用来在字符串中查找匹配的字符。默认情况下, 模式中的一个字符在字符串中就匹配它自身。例如, 正则式 “abc” 就匹配 “Is there an abc here?” 中的 abc。

正则式的强大来自于具有特殊含义的“特殊字符”以及字符组合:

特殊含义的字符	
.	任意单个字符 (“通配符”)
[	字符集
{	计数
(	子模式开始
)	子模式结束
\	下一个字符具有特殊含义
*	0 个或多个
+	一个或多个
?	可选 (0 个或一个)
	二选一 (或)
^	行的开始; 否定
\$	行的结束

例如, `x.y` 匹配任何以 `x` 开头以 `y` 结束的、长度为 3 的字符串, 例如 `xxy`、`x3y` 和 `xay`, 但不匹配 `xyx`、`3xy` 及 `xy`。

注意, `{...}`、`*`、`+` 及 `?` 是后缀运算符。例如, `\d+` 表示 “一个或多个十进制数字”。

如果你想在模式中使用这些特殊符号的普通字符含义, 需要利用反斜线进行“转义”。例如, `+` 表示 “一个或多个” 运算符, 而 `\+` 表示加号。

### 23.8.2 字符集

最常用的字符组合也有 “特殊字符” 的简洁形式表示:

表示字符集的特殊字符		
\d	一个十进制数字	[[digit:]]
\l	一个小写字母	[[lower:]]
\s	一个空白符(空格符、制表符等)	[[space:]]
\u	一个大写字母	[[upper:]]
\w	一个字母(a ~ z 或 A ~ Z)或数字(0 ~ 9)或下划线(_)	[[alnum:]]
\D	\d 之外的字符	[^digit:]]
\L	\l 之外的字符	[^lower:]]
\S	\s 之外的字符	[^space:]]
\U	\u 之外的字符	[^upper:]]
\W	\w 之外的字符	[^alnum:]]

注意，大写形式的特殊字符表示“对应的小写形式特殊字符所表示的字符之外的所有字符”。特别地，\W 表示“不是一个字母”而非“一个大写字母”。

第三列的内容(如 [[digit:]]) 是表示相同含义的另一种较长的表示方式。

与 string 和 istream 库类似，regex 库可以处理大字符集，如 Unicode。与前文一样，我们不对此进行详细介绍，如需要，你可以查找相关资料或求助于有经验的人。Unicode 文本处理已经超出了本书的讨论范围。

### 23.8.3 重复

模式的重复可以通过一些后缀运算符来实现：

重复	
{n}	严格重复 <i>n</i> 次
{n,}	重复 <i>n</i> 次或更多次
{n,m}	重复至少 <i>n</i> 次至多 <i>m</i> 次
*	重复 0 次或多次，即 {0,}
+	重复一次或多次，即 {1,}
?	可选(0 次或一次)，即 {0,1}

例如：

**Ax\***

与任何以 A 开始，后接 0 或多个 x 的字符串匹配，如

```
A
Ax
Axx
Axxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

如果希望字符至少出现一次，则用 + 替换 \*。例如：

**Ax+**

匹配那些以 A 开始，后接一个或多个 x 的字符串，如

```
Ax
Axx
Axxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

但不匹配

**A**

常用的出现 0 次或一次（“可选”）的概念用问号表示。例如，

`\d-?d`

匹配以破折号分隔的两个数字和连续两个数字，如

`1-2`

`12`

但不匹配

`1--2`

如需指定特定的重复次数，或者一定范围内的重复次数，可用花括号。例如：

`\w{2}-\d{4,5}`

匹配以两个字母（或数字、下划线）和一个破折号开始，后接四个或五个数字的字符串，如

`Ab-1234`

`XX-54321`

`22-54321`

但不匹配

`Ab-123`

`?b-1234`

注意，数字也属于字符集 `\w`。

### 23.8.4 子模式

为了指定模式中的子模式，用括号将其括起来。例如：

`(d*?)`

它定义了一个子模式，表示 0 或多个数字后接一个冒号。一个复杂的模式可用多个子模式组成。例如：

`(d*?)?(d+)`

它表示字符串前半部分是任意长度的数字序列（可以为空）后接一个冒号，也可以为空，后一部分是一个或多个数字的序列。多么冗长的叙述！难怪人们发明正则表达式这样一种简洁、准确的方法来描述这些模式。

### 23.8.5 可选项

“或”运算符 (`|`) 表示二选一的概念。例如：

`Subject: (FW:|Re:)?(.*)`

匹配主题行，其中包含可选的 `FW:` 或者 `Re:`，后接 0 个或多个任意字符。例如，它匹配如下字符串：

**Subject: FW: Hello, world!**

**Subject: Re:**

**Subject: Norwegian Blue**

但不匹配：

**SUBJECT: Re: Parrots**

**Subject FW: No subject!**

注意，或运算的两个子正则式均不能为空：

```
(def) // 错误
```

多个连续的和运算是允许的：

```
(bs|Bs|bS|BS)
```

### 23.8.6 字符集和范围

前文已经介绍了一些表示字符集的特殊字符，如表示数字的 `\d`，表示字母、数字或下划线的 `\w` 等（见 23.7.2 节）。不过，有时我们还需要定义其他的字符集，这也很容易：

<code>[\w @]</code>	字母、数字、下划线、空格或 @
<code>[a-z]</code>	小写字母
<code>[a-zA-Z]</code>	大写或小写字母
<code>[Pp]</code>	大写或小写的 p
<code>[\w\~]</code>	字母、数字、下划线或破折号（破折号表示范围）
<code>[asdfghjkl;']</code>	美式 QWERTY 键盘中间一行上的所有字符
<code>[.]</code>	句点
<code>[{[(\*+?^\$]</code>	所有特殊字符（这里表示字符本身，不是特殊字符的含义）

在字符集中，`-`（破折号）表示范围，如 `[1-3]` 表示 1、2 或 3，`[w~z]` 表示 w、x、y 或 z。范围的使用一定要很小心：并非所有语言都具有相同的字母，而且并非所有字符集中字符顺序都一致。如果你觉得要使用的范围不是最常见的英语字母表中的字母或者数字范围，请查阅相关资料。

注意，我们可以在字符集中使用特殊字符，如 `\w`（表示任意单词字符）。于是产生一个问题，如何表示反斜杠符号呢？与往常一样，对其进行转义即可：“`\\`”。

如果字符集的第一个字符是 `^`，则表示“非”的概念。例如：

<code>[^aeiouy]</code>	非英语元音
<code>[^\d]</code>	非数字
<code>[ ^aeiouy]</code>	空格、^ 或者英语元音

在最后一个正则式中，`^` 不是字符集中的第一个字符，因此它只是一个普通字符，而不是非运算符——正则表达式就是如此微妙。

`regex` 的一些实现中还提供了一组命名字符集用于匹配。例如，如果希望匹配字母数字符号（即，匹配一个字母或者一个数字：`a-z` 或 `A-Z` 或 `0-9`），可以使用 `[:alnum:]`。在这里，`alnum` 是字符集的名称（字母数字字符集）。于是，加引号的非空字母数字串对应的正则式为 `"[:alnum:]"`。为了将此正则式放在程序中的字符串内，需要将引号转义：

```
string s {"\" [:alnum:]"+"^"}
```

而且，在 `regex` 中，引号也是特殊符号。因此为了将字符串转换为 `regex` 对象，我们还需再产生一个反斜线符号，使得在 `regex` 对象中，引号被转义，而不是表示特殊符号。

```
regex s {"\\\" [:alnum:]"+"\\\""};
```

使用原始字符串常量更简单：

```
regex s2 {R{"\" [:alnum:]"+"^"}}
```

对于反斜杠和双引号，我们更倾向于使用原始字符串常量。在许多应用程序中都是这么使用的。

使用正则表达式带来了许多符号表示惯例，下面列出了一些标准的命名字符集。

字符集	
alnum	任意字母数字
alpha	任意字母
blank	任意空白符，不包括换行
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	任意小写字母
print	任何可打印字符
punct	任意标点字符
s	任意空白符
space	任意空白符
upper	任意大写字母
w	任意单词字符（字母、数字及下划线）
xdigit	任意十六进制数字字符

特定的 regex 实现可能提供更多的命名字符集，但如果你决定使用的字符集不在上表内，一定要检查可移植性是否足够好，是否能满足你最初的需求。

### 23.8.7 正则表达式错误

如果你指定了一个错误的正则表达式，会产生什么后果？如：

```
regex pat1 {"(ghi)"};    // 可选项缺失
regex pat2 {"[c-a]"};    // 不是一个范围
```



当我们将一个模式赋予 regex 时，它会对模式进行检查，如果正则表达式匹配时发现模式不合法或者过于复杂，无法用于匹配时，它会抛出一个 bad\_expression 异常。

下面这段程序对体会正则表达式匹配很有帮助：

```
#include <regex>
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
using namespace std;

// 从输入中读取一个模式和一组文本行
// 检查模式并在文本行中搜索此模式

int main()
{
    regex pattern;

    string pat;
    cout << "enter pattern: ";
    getline(cin, pat);    // 读取模式

    try {
        pattern = pat;    // 这条语句会检查 pat
        cout << "pattern: " << pat << "\n";
    }
}
```


```

catch (bad_expression) {
    cout << pat << " is not a valid regular expression\n";
    exit(1);
}

cout << "now enter lines:\n";
int lineno = 0;

for (string line; getline(cin,line); ) {
    ++lineno;
    smatch matches;
    if (regex_search(line, matches, pattern)) {
        cout << "line " << lineno << ": " << line << '\n';
        for (int i = 0; i<matches.size(); ++i)
            cout << "\tmatches[" << i << "]: "
                << matches[i] << '\n';
    }
    else
        cout << "didn't match\n";
}
}

```

 试一试

编译、运行这个程序，尝试一些模式，如 `abc`、`x.*x`、`(.*)`、`\{([^\}]*\)}` 以及 `\w+ \w+ (Jr\.)?`。

## 23.9 使用正则表达式进行模式匹配

正则表达式有两种主要用途：

- 搜索：在（任意长的）数据流中搜索与正则式匹配的字符串——`regex_search()` 就实现此功能，它在数据流中搜索与正则式匹配的子串。
- 匹配：判断一个字符串（已知长度）是否与模式匹配——`regex_match()` 检查模式和给定字符串是否完全匹配。

23.6 节中给出的搜索邮政编码的程序是正则式搜索功能的很好示例。下面，我们介绍一个匹配操作的例子。例如，我们需要从下表这样的结构中提取数据：

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER IALT
0A	12	11	23
1A	7	8	15
1B	4	11	15
2A	10	13	23
3A	10	12	22
4A	7	7	14
4B	10	5	15
5A	19	8	27
6A	10	9	19
6B	9	10	19
7A	7	19	26



(续)

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER IALT
7G	3	5	8
7I	7	3	10
8A	10	16	26
9A	12	15	27
0MO	3	2	5
0P1	1	1	2
0P2	0	5	5
10B	4	4	8
10CE	0	1	1
1MO	8	5	13
2CE	8	5	13
3DCE	3	3	6
4MO	4	1	5
6CE	3	4	7
8CE	4	4	8
9CE	4	9	13
REST	5	6	11
Alle klasser	184	202	386

这个表记录的是 Bjarne Stroustrup 的母校在 2007 年的学生数，它实际上是从互联网上提取出来的，其原始格式看起来很整洁，而且正是我们进行数据分析时常见的那种典型格式：

- 它包含数值域。
- 它包含字符域，其中字符串只有了解上下文的人才知道其含义。（在本例中，这种情况更为明显，因为文字都是丹麦文。）
- 字符串中包含空格。
- 数据“域”用“分隔符”分开，在本例中，分隔符为制表符。

⚠ 我们选择的这个例子“相当典型”而且“不是很困难”，但有一点比较微妙：人眼是无法看出空格和制表符之间的区别的，这只能在程序中进行区分。

我们将展示正则表达式的如下用途：

- 验证表格布局是否正确（即，是否每行包含的域的数目都正确）。
- 验证合计值是否正确（每列最后一行上的数值为其上所有数值之和）。

☛ 如果我们可以完成这些任务，那么我们就几乎能做任何事！例如，由原表格创建出一个新的表格：将具有相同起始数字的行（表示年级：一年级用 1 表示，依此类推）合并在一起，或者分析学生数是逐年增长还是减少（参考习题 10 ~ 11）。

为了对表格进行分析，我们需要两个模式：一个用于分析表头行，另一个用于分析剩余行：

```
regex header {R"^([w ]+( [w ]+)*$)";
regex row {R"^([w ]+( \d+)( \d+)( \d+)$)";
```

⚠ 请记住，我们一直在称赞正则表达式简洁、功能强大，但我们从未称赞它易于被初学者理解。实际上，“只写语言”的名声对正则式来说是恰如其分的。我们从表头开始，由于它

(第一行)不包含任何数值,将其直接丢弃即可。但是,为了多做一些练习,我们还是对其进行分析。它包含四个由制表符分隔的“单词域”(“字母数字域”)。这些域中可以包含空格,因此,我们不能简单地用 `\w` 来匹配其中的字符,而应该用 `[\w ]`,即,一个字母、数字、下划线或者一个空格。因此,匹配第一个域的正则式为 `[\w ]+`。我们希望第一个域在行首,因此可用 `^[ \w ]+` 匹配,符号“`^`”表示“行首”的含义。行中剩余域可描述为一个制表符后接多个单词: `( [ \w ]+)`。现在,我们先给出匹配任意多个这种域,最后是行尾的正则式: `( [ \w ]+)*$`。美元符号(`$`)表示“行尾”。

注意,人眼是看不出制表符和空格之间的区别的,但在本例中,排版时已经将制表符展开了,因此可以明确地区分开来。

现在来看更有趣的部分:如何为数值行设计模式。如表头行一样,数值行的行首也是单词域,因此子正则式为 `^[ \w ]+`。后面是三个数值域,每个域之前是一个制表符(`\d+`),因此,完整的正则式为:

```
^[ \w ]+( \d+)( \d+)( \d+)$
```

将其放入原始字符串常量,是:

```
R"^[ \w ]+( \d+)( \d+)( \d+)$"
```

现在,模式已经设计完毕,下面所要做的就是使用它们分析表格。首先验证表格布局:

```
int main()
{
    ifstream in {"table.txt"}; // 输入文件
    if (!in) error("no input file\n");

    string line; // 输入缓冲区
    int lineno = 0;

    regex header {R"^[ \w ]+( [ \w ]+)*$"}; // 文件头行
    regex row {R"^[ \w ]+( \d+)( \d+)( \d+)$"}; // 数据行

    if (getline(in,line)) { // 检查文件头行
        smatch matches;
        if (!regex_match(line, matches, header))
            error("no header");
    }
    while (getline(in,line)) { // 检查数据行
        ++lineno;
        smatch matches;
        if (!regex_match(line, matches, row))
            error("bad line",to_string(lineno));
    }
}
```

简洁起见,我们省略了 `#include`。我们的目的是检查每行中的所有字符,因此使用 `regex_match` 而不是 `regex_search`。两者的区别在于, `regex_match` 需匹配输入中所有字符才能判断匹配成功,而 `regex_search` 只要在输入中找到匹配的字符串即可。如果你想用的是 `regex_search`,但误输入了 `regex_match` (或反之),这种错误很难查找出来。两个函数对“匹配”参数的使用是相同的。

接下来我们对表中的数据进行验证。我们对男孩(“drenge”)和女孩(“piger”)两列保存其学生数之和。对每一行,我们检查最后一个域(“ELEVER IALT”)是否等于前两个

域之和。最后一行（“Alle klasser”）的内容是同列中其他数据的合计值。为了进行这些检查，我们修改了模式 row，将文本域设计为子模式，这样就可以识别“Alle klasser”了。

```
int main()
{
    ifstream in {"table.txt"};    // 输入文件
    if (!in) error("no input file");

    string line;                  // 输入缓冲区
    int lineno = 0;

    regex header {R"^(^[\w ]+(\ [\w ]+)*$)"};    // 文件头行
    regex row {R"^(^[\w ]+(\ \d+)( \d+)( \d+)$)"}; // 数据行

    if (getline(in,line)) {      // 检查文件头行
        smatch matches;
        if (regex_match(line, matches, header)) {
            error("no header");
        }
    }

    // 列合计:
    int boys = 0;
    int girls = 0;

    while (getline(in,line)) {
        ++lineno;
        smatch matches;
        if (!regex_match(line, matches, row))
            cerr << "bad line: " << lineno << "\n";

        if (in.eof()) cout << "at eof\n";

        // 检查行:
        int curr_boy = from_string<int>(matches[2]);
        int curr_girl = from_string<int>(matches[3]);
        int curr_total = from_string<int>(matches[4]);
        if (curr_boy+curr_girl != curr_total) error("bad row sum \n");

        if (matches[1]== "Alle klasser") {    // 最后一行
            if (curr_boy != boys) error("boys don't add up\n");
            if (curr_girl != girls) error("girls don't add up\n");
            if (!(in>>ws).eof()) error("characters after total line");
            return 0;
        }

        // 更新合计:
        boys += curr_boy;
        girls += curr_girl;
    }

    error("didn't find total line");
}
```

最后一行在语义上与其他行是不同的——它是其他行之和，我们通过标签“Alle klasser”来识别它。在这一行之后，我们不再接受任何非空白字符（使用来自 to<> 的技术，见 23.2 节），如果未找到这一行（合计值），则输出一个错误信息。

我们使用 23.2 节中的 from\_string 函数从数据域中提取整型值。我们已经确认这些域中

只包含数字，因此无须检查这次字符串到整数的转换是否成功。

## 23.10 参考文献

正则表达式是一种很流行，也很有用的工具。很多程序设计语言都支持正则表达式，其格式也各种各样。其理论基础是一种优美的形式语言理论，其高效的实现技术则基于状态机。正则表达式的全部概念、基础理论、实现以及状态机的一般用法已经超出了本书的讨论范围。不过，由于这些主题都是计算机科学课程中重要的内容，而正则式又如此流行，因此如果你需要学习这些内容或者仅仅是感兴趣的话，很容易找到相关资料。一些参考文献罗列如下。

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition* (通常被称为“龙书”) . Addison-Wesley, 2007. ISBN 0321547985.

Cox, Russ. “Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, ... ).” <http://swtch.com/~rsc/regex/regexp1.html>.

Maddock, J. boost::regex documentation. [www.boost.org/](http://www.boost.org/).

Schwartz, Randal L., Tom Phoenix, and Brian D. Foy. *Learning Perl, Fourth Edition*. O'Reilly, 2005. ISBN 0596101058.

## 简单练习

1. 确认你的机器上安装的标准库是否包含 `regex`。提示：尝试使用 `std::regex` 和 `tr1::regex`。
2. 编译、运行 23.7 节中的小程序，并弄清如何通过设置工程属性或命令行选项来使用 `regex` 头文件及链接 `regex` 库。
3. 使用上一小题中的程序测试 23.7 节中的模式。

## 思考题

1. 我们在哪里查找“文本”？
2. 标准库中哪些功能对于文本分析非常有用？
3. `insert()` 的插入位置是其位置（或迭代器）之前还是之后？
4. Unicode 是什么？
5. 如何将字符串转换为其他类型？反过来呢？
6. 假定 `s` 是一个字符串，`cin>>s` 和 `getline(cin,s)` 的区别在哪里？
7. 列出标准流。
8. 一个 `map` 对象中的关键字是什么？给出一些关键字类型的例子。
9. 如何遍历 `map` 的元素？
10. `map` 和 `multimap` 的差别在哪？哪种有用的 `map` 的操作在 `multimap` 中不存在，这样设计的原因是什么？
11. 向前迭代器需要哪些操作？
12. 空域和域不存在有什么区别？给出两个例子。
13. 正则表达式中为什么需要使用转义符？

14. 如何将正则表达式存入 `regex` 变量?
15. `\w+\s\d{4}` 与什么样的字符串匹配? 给出三个例子。如果将此模式转换为 `regex` 变量, 需要用什么样的字符串初始化 `regex` 变量?
16. 在程序中, 如何确定一个字符串是否是合法的正则表达式?
17. `regex_search()` 的功能是什么?
18. `regex_match()` 的功能是什么?
19. 如何在正则表达式中表示句点符号 (`.`)?
20. 如何在正则表达式中表示“至少三个”的概念?
21. 字符 `7` 与 `\w` 匹配吗? 下划线符号 `_` 呢?
22. 如何在正则表达式中表示大写字母?
23. 如何自定义字符集?
24. 如何从整数域中提取数值?
25. 如何用正则表达式表示浮点数?
26. 如何从匹配结果中提取浮点值?
27. 子匹配是什么? 如何访问子匹配结果?

## 术语

<code>match</code> (匹配)	<code>regex_match()</code>	<code>search</code> (搜索)
<code>multimap</code>	<code>regex_search()</code>	<code>smatch</code>
<code>pattern</code> (模式)	<code>regular expression</code> (正则表达式)	<code>sub-pattern</code> (子模式)

## 习题

1. 编译、运行邮件文件分析程序, 创建一个更大的邮件文件来测试它。一定要加入一些可能触发错误的邮件消息, 例如有两个地址行的邮件、多个邮件具有相同的地址和 / 或相同的主题、空邮件等。另外, 用一些显然不符合程序定义的邮件形式的内容进行测试, 例如, 一个不包含“-----”行的大文件。
2. 添加一个 `multimap` 对象, 用来保存主题。修改程序, 令其从键盘接收一个字符串, 输出所有主题与此字符串匹配的邮件。
3. 修改 23.4 节中的邮件分析程序, 使用正则表达式查找主题和发件人。
4. 找一个真正的邮件文件 (包含真实邮件消息), 修改邮件分析程序, 使其能提取指定发件人的邮件的主题行。
5. 找一个大的邮件文件 (包含几千个邮件消息), 测试用 `multimap` 输出所有邮件消息所花费的时间, 测试改用 `unordered_multimap` 后的时间。注意, 我们的应用并未利用 `multimap` 的优点。
6. 编写一个程序, 从一个文本文件中查找日期。输出包含日期的行, 格式为“行号: 行内容”。以一个简单的日期格式为起点, 如 `12/24/2000`, 设计、测试程序。随后再加入更多的格式。
7. 编写程序 (与上题类似的程序), 在文件中查找信用卡号码。上网搜索一下真实的信用卡号码是什么格式。
8. 修改 23.8.7 节中的程序, 使其接受一个模式和一个文件名作为输入, 输出文件中匹配模

- 式的行，输出格式为“行号：行内容”。如果未找到匹配行，则不输出任何内容。
9. 使用 `eof()` (见附录 C.7.2) 来检测某行是否是表格的最后一行。采用这种方法简化 23.9 节中的程序。用表格后接空行的文件和不以换行结束的文件测试程序。
  10. 修改 23.9 节中的表格验证程序，用原表格中的数据创建并输出一个新表格，其中所有首数字相同（同一年级）的行被合并在一起。
  11. 修改 23.9 节中的表格验证程序，检查学生数是逐年增加还是减少。
  12. 基于习题 6 中程序，编写一个新程序，查找所有日期并将格式改为 ISO 标准格式 `yyyy-mm-dd`。程序读入输入文件，转换日期格式后将结果写入输出文件。两个文件内容完全一致，只是日期格式可能不同。
  13. 句点 (.) 与 `\n` 匹配吗？编写一个程序验证之。
  14. 编写一个类似 23.8.7 中的程序，可以输入模式，进行匹配。但是，它从文件（以 `\n` 作为行的分隔）读取输入，因此可以测试跨行的模式。测试这个程序，并记录至少一打以上的测试结果。
  15. 给出一个不能用正则表达式描述的模式。
  16. 本题不适合初学者：证明上题中的模式确实不是正则表达式。

## 附言

我们很容易陷入这样一个观点：计算机和计算都是面对数字的，计算就是数学的一种形式。这显然是不正确的。只要看看你的计算机屏幕就很清楚了，上面充满了文本和图片。甚至说不定它正在演奏音乐呢。对于不同应用，使用适当的工具是非常重要的——从 C++ 的角度，就是要使用适合的库。对于文本处理，正则表达式库通常是关键工具——另外不要忘了 `map` 和标准库算法。

# 数值计算

每个复杂问题都存在一个清晰、简洁但是错误的解答。

——H. L. Mencken

本章介绍用于数值计算的一些基本语言特性和标准库功能。我们提出大小、精度以及截断等一些基本问题。本章的核心部分是关于多维数组的讨论，既讨论 C 风格的多维数组，也介绍  $N$  维矩阵库。我们还将介绍随机数，它被广泛用于测试、仿真以及电脑游戏中。最后，我们介绍标准库数学函数，并简要介绍标准库对复数的支持。

## 24.1 简介

对某些人来说，数字、数值计算就是一切，比如很多科学家、工程师以及统计学家等。对更多的人来说，数值计算在某些时候是必要的。例如，一个计算机科学家偶尔与一个物理学家合作时，就属于这种情况。而对于大多数人来说，很少会用到数值计算（不是整数和浮点数的简单算术运算，而是更复杂的计算）。本章的目的是介绍一些用于处理简单数值计算问题的程序设计语言技术细节。我们不会介绍数值分析或者浮点数运算的微妙难懂之处，这些内容已经远远超出了本书的讨论范围，而且与应用中领域相关的问题是紧密融合的。本章主要讨论如下问题：

- 一些内置类型是有固定大小的，由此引发的精度、溢出等问题。
- 数组：内置的多维数组类型和更适于数值计算的 `Matrix` 库。
- 随机数的最基本的概念。
- 标准库中的数学函数。
- 复数。

其中 `Matrix` 库是重点，它使矩阵（多维数组）的处理变得简单。

## 24.2 大小、精度和溢出

当我们使用内置类型和普通计算技术时，数值会占用固定大小的内存。也就是说，整数类型（`int`、`long` 等等）只是数学上的整数的近似，同样地，浮点数据类型（`float`、`double` 等等）也只是数学上的实数的近似。这意味着，从数学的角度看，计算机中的某些计算是不精确的，甚至是错的。例如：

```
float x = 1.0/333;
float sum = 0;
for (int i=0; i<333; ++i) sum+=x;
cout << setprecision(15) << sum << "\n";
```

执行这段代码，一些人很天真地期望得到 1，但实际并不是这样：

```
0.999999463558197
```

这就是我们所期待的结果，因为我们了解计算机数值计算——这就是一个截断错误的例子。

在计算机中，一个浮点数只占用固定数目的二进制位，因此我们总是可以“愚弄”计算机：让它做一个运算，其结果需要更多的二进制位来保存。例如，有理数  $1/3$  是无法用十进制数精确表示的（无论我们使用多少位十进制数字都不能）。 $1/333$  也是如此，于是，当我们将  $x$ （计算机中与  $1/333$  最为接近的浮点数值）累加 333 次时，我们得到是与 1 有微小差距的一个值。每当我们进行大量浮点数运算时，就会产生截断误差，唯一的问题是误差对结果的影响是否严重。

要时时检查计算结果是否合理。当进行计算时，你必须清楚什么是“合理的结果”，否则就很容易被“愚蠢的错误”或者计算误差所愚弄。要保持对截断误差的警惕，如果有疑问，一定要请教专家或者仔细研究数值计算的相关资料。

### 试一试

将上例中的 333 改为 10，重新运行程序。你预计会得到什么结果？实际得到了什么结果？我们早已警告过你了！

相对于实数，用固定位数表示整数所引起的问题更为引人注目。原因在于，浮点数被定义为实数的近似，因此后果是丢失精度（即丢失最低有效位），而整数则是引起溢出（即丢失最高有效位）。因此，浮点数运算的误差总是比较细微，不易被初学者察觉，而整数的误差则往往非常惊人，很难不被注意。请记住，我们宁愿错误更早地、更突出地显现出来，以便能及时修正。

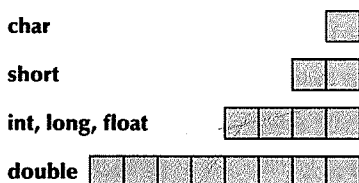
考察下面的程序：

```
short int y = 40000;
int i = 1000000;
cout << y << " " << i * i << "\n";
```

其输出结果为：

```
-25536 -727379968
```

这就是典型的溢出现象。我们当然希望能表示任意整型值，以获得准确的计算结果。但计算机中的整型只能表示（相对）较小的整数，其宽度不足以精确表示所有整数。在本例中，一个两字节的 `short` 类型不能表示 40 000，而一个四字节的 `int` 类型不能表示 1 000 000 000 000。C++ 内置类型的准确宽度依赖于硬件平台和编译器（见附录 A.8）。我们可以使用 `sizeof(x)` 来获得 `x` 的宽度（以字节为单位），`x` 可以是一个变量或者一个类型。由定义，`sizeof(char)==1`。一些常见类型的大小如下所示：



这是在 Windows 平台上，使用微软编译器时类型的宽度。对于整数和浮点数，C++ 都提供了不同宽度的类型。但除非有很好的理由，否则最好只使用 `char`、`int` 和 `double` 这几个



标准宽度的类型。在大多数程序中（当然不是所有），其他整数和浮点数类型所带来的麻烦比带来的好处更多。

你可以将一个整数赋予一个浮点数。如果整型值超出了浮点类型的表示范围，则会丢失精度。例如：

```
cout << "sizes: " << sizeof(int) << ' ' << sizeof(float) << '\n';
int x = 2100000009; // 大 int
float f = x;
cout << x << ' ' << f << '\n';
cout << setprecision(15) << x << ' ' << f << '\n';
```

在我们的计算机上，输出结果为：

```
Sizes: 4 4
2100000009 2.1e+009
2100000009 2100000000
```

float 类型和 int 类型占用相同大小的内存空间（4 个字节）。一个 float 值由一个“尾数” $a$ （通常是 0 和 1 之间的一个数）和一个指数  $b$  组成（ $a \times 10^b$ ），因此无法准确表示最大的 int 值（如果我们想把最大 int 的准确值存入一个 float 值中，尾数已经占用了所有空间，指数根本没有位置存放了）。因此在上例中，f 只能保存尽量接近 2100000009 的值，对于最后一个 9 它已经无能为力了，这就是输出结果是 2100000000 的原因。

⚠ 另一方面，如果你将一个浮点数赋予一个整数，会导致截断，即，小数部分（小数点之后的数字）被简单丢弃。例如：

```
float f = 2.8;
int x = f;
cout << x << ' ' << f << '\n';
```

x 的值会是 2，而不是 3——这里并不是进行“四舍五入”。C++ 中 float 转换为 int 采用的是截断而非舍入。

⚠ 当进行计算时，你必须清楚可能会发生的溢出和截断。C++ 不会捕获这些问题，考虑下面的程序：

```
void f(int i, double fpd)
{
    char c = i; // 是的：char 的确是非常小的整数
    short s = i; // 小心：一个 int 可能放不进一个 short int 中
    i = i+1; // i 如果是最大的 int 会怎样？
    long lg = i*i; // 小心：一个 long 可能不会比一个 int 占用更大空间
    float fps = fpd; // 小心：一个 double 可能放不进一个 float 中
    i = fpd; // 截断：如 5.7→5
    fps = i; // 你可能丢失精度（对非常大的 int 值）
}

void g()
{
    char ch = 0;
    for (int i = 0; i < 500; ++i)
        cout << int(ch++) << '\t';
}
```

如果对这段程序有疑问，尝试运行它！对这类问题，垂头丧气或者仅仅依赖文档都是不可取的，实验是最好的方法！

## 试一试

运行 `g()`。修改 `f()`，打印 `c`、`s`、`i` 等等。用不同的类型来测试这个函数。

我们在 25.5.3 节中还会更详细地介绍各种整数类型的表示及它们之间的转换。如有可能，应该使用尽可能少的类型，这会减少混乱。例如，如果在程序中只使用 `double`，而不用 `float`，就减少了可能的 `double` 和 `float` 的转换问题。实际上，我们倾向于只使用 `int`、`double` 和 `complex`（见 24.8 节）进行算术计算，只使用 `char` 用于字符处理，而 `bool` 用于逻辑运算，除非迫不得已，否则不使用其他类型。

### 24.2.1 数值限制

每种 C++ 的实现都在 `<limits>`、`<climits>`、`<limits.h>` 和 `<float.h>` 中指明了内置类型的属性，因此程序员可以利用这些属性来检查数值限制、设置哨兵机制等等。附录 C.9.1 中列出了这些值，它们对于开发低层程序是非常重要的。如果你觉得需要这些属性值，表明你的工作很可能比较靠近硬件。但这些属性还有其他用途，例如，对语言实现细节感到好奇是很正常的：“一个 `int` 有多大？”，“`char` 是有符号的吗？”等等。希望从系统文档中找到这些问题的正确答案是很困难的，而 C++ 标准对这类问题大多没有明确规定。较好的办法是写一个简短的小程序来获得这些问题的答案：

```
cout << "number of bytes in an int: " << sizeof(int) << '\n';
cout << "largest int: " << INT_MAX << '\n';
cout << "smallest int value: " << numeric_limits<int>::min() << '\n';

if (numeric_limits<char>::is_signed)
    cout << "char is signed\n";
else
    cout << "char is unsigned\n";

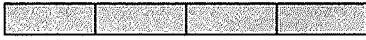
char ch = numeric_limits<char>::min(); // 最小的正数
cout << "the char with the smallest positive value: " << ch << '\n';
cout << "the int value of the char with the smallest positive value: "
    << int(ch) << '\n';
```

如果你编写的程序将来要用在多种硬件平台上，那么能在程序中获取上面这些信息就非常有价值了。另一种方法是将这些信息硬编码到程序中，但这对维护人员来说是灾难性的。这些属性值对溢出检测也是很有用的。

## 24.3 数组

数组（array）就是一个元素序列，我们可以通过下标（位置）来访问元素。我们通常也把这种数据结构称为向量（vector）。我们特别关注的一种数组是：每个元素本身也是一个数组，即多维数组，通常也被称为矩阵（matrix）。术语的多样性是一个概念的流行程度和使用广泛程度的标志。标准库中的 `vector`（见附录 C.4）、`array`（见 15.9 节）和内置数组类型（见附录 A.8.2）都是一维的。那么，如果我们需要二维数组（比如矩阵）的话，应该怎么办？如果我们需要七维数组呢？

我们可以将一维和二维数组想象为如下结构：



一个向量（如`Matrix<int>v(4)`），也被称为一个一维数组，或者一个 $1 \times N$ 矩阵

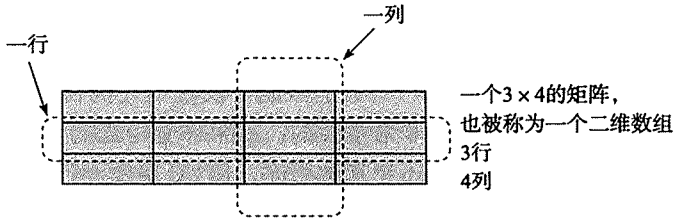


一个 $3 \times 4$ 矩阵（如`Matrix<int, 2>m(3,4)`），也被称为一个二维数组

数组对于大多数计算问题（“数值运算”）来说都是非常重要的数据结构，很多有趣的科学计算、工程计算、统计运算以及金融计算都极大地依赖于数组。



我们通常把数组看作行和列组成的结构：



一列就是一个  $x$  坐标相同的元素的序列，一行就是一个  $y$  坐标相同的元素的序列。

## 24.4 C 风格的多维数组

利用 C++ 内置的数组类型也可创建多维数组，方法是 multidimensional array 简单地看作数组的数组，即，数组的元素也是数组。例如：

```
int ai[4];           // 一维数组
double ad[3][4];    // 二维数组
char ac[3][4][5];   // 三维数组
ai[1] = 7;
ad[2][3] = 7.2;
ac[2][3][4] = 'c';
```



这种方法继承了一维数组的优点和缺点：

- 优点
  - 直接映射到硬件。
  - 低层操作效率高。
  - 语言直接支持。
- 缺点
  - C 风格的多维数组是数组的数组（见下文）。
  - 大小是固定的（即在编译时就固定下来）。如果希望在运行时再确定大小，就必须使用动态内存分配。
  - 不能干净地传递数组参数，只能转换为指向其首元素的指针。
  - 没有越界检查。通常，数组不知道它自己的大小。
  - 没有数组的整体运算，甚至没有赋值（拷贝）。

内置数组类型被广泛用于数值计算，但同时也是造成程序错误和程序过于复杂的主要原因。对于大多数人来说，编写和调试使用内置数组的程序都是很痛苦的。如果你不得不使用

内置数组，请查找相关资料（如《The C++ Programming Language》）。不幸的是，C++ 使用与 C 相同的内置多维数组，因此还有很多“在别处”的代码在使用这种数组。

内置数组最大的问题是不能干净地传递多维数组参数，必须转而使用指针，并显式计算数组元素位置。例如：▲

```
void f1(int a[3][5]);           // 只对 [3][5] 矩阵有用

void f2(int [ ][5], int dim1); // 第一维是可变的

void f3(int [5][ ], int dim2); // 错误：第二维不能是可变的

void f4(int[ ][ ], int dim1, int dim2); // 错误（而且无论如何也不可行）

void f5(int* m, int dim1, int dim2) // 奇怪，但可行
{
    for (int i=0; i<dim1; ++i)
        for (int j = 0; j<dim2; ++j) m[i*dim2+j] = 0;
}
```

在这段程序中，虽然  $m$  是一个二维数组，但我们只能将它作为  $\text{int}^*$  类型的参数来传递。只要数组的第二维大小是可变的（作为一个参数），就无法告知编译器参数  $m$  是一个  $(\text{dim1}, \text{dim2})$  数组，而只能传递指向其起始地址的指针。表达式  $m[i*\text{dim2}+j]$  实际就表示  $m[i,j]$ ，但由于编译器不知道  $m$  是一个二维数组，我们不得不显式计算  $m[i,j]$  在内存中的位置。

以我们的观点来看，这太麻烦、太原始，也太容易出错了。而且运行速度也可能会很慢，因为显式计算元素地址会使代码优化更为复杂。因此，我们不再介绍内置多维数组，而是重点讨论 Matrix 库中的多维数组机制，它没有上述缺点。

## 24.5 Matrix 库

如果以数值计算为目标的话，我们到底希望从数组 / 矩阵库中获得什么呢？

- “程序中使用数组的方式应该与数学 / 工程教科书上对数组的使用方式相近”。
  - 向量、矩阵、张量等等类似。
- 具备编译时和运行时检查功能。
  - 支持任意维数组。
  - 支持每一维任意多个元素。
- 数组是真正的变量 / 对象。
  - 可以作为参数传递。
- 支持常见的数组运算：
  - 下标：()
  - 切片：[]
  - 赋值：=
  - 标量运算（+=、-=、\*=、%= 等等）
  - 融合的向量运算（如  $\text{res}[i]=\text{a}[i]*\text{c}+\text{b}[2]$ ）
  - 点积（ $\text{res}=\text{a}[i]*\text{b}[i]$  的和，也被称为内积）
- 将传统的数组 / 向量的概念转换为代码，这些代码要是你自己来写的话，会花费极大的精力，而且效率也不会比现在的更好。
- 如果需要，你可以扩展它（也就是说，库的实现没有使用什么“魔法”）。

Matrix 实现了上述功能，也只实现了这些。如果你需要更多功能，例如高级的数组函数、稀疏数组、控制内存布局等等，可以自己编写相应的程序或者选用一个更接近你要求的库（第二种方式更好些）。但是，很多这些“高级”功能可以通过在 Matrix 之上构造算法和数据结构来实现。Matrix 库不是 ISO C++ 标准库的一部分。你可以在课程网站上找到 Matrix.h，整个库定义在名字空间 Numeric\_lib 中。我们选择“矩阵”作为库的名字，是因为“向量”和“数组”在 C++ 标准库中已经用得太多了。在英语中，矩阵 matrix 的复数形式是 matrices，matrixes 也是正确的，但很少使用。由于“Matrix”指的是一个 C++ 语言实体，因此在本书的英文原版中使用 Matrixes，以避免混淆。Matrix 库的实现使用了一些高级技术，因此我们不会对此进行介绍。

### 24.5.1 矩阵的维和矩阵访问

考察下面的简单例程：

```
#include "Matrix.h"
using namespace Numeric_lib;

void f(int n1, int n2, int n3)
{
    Matrix<double,1> ad1(n1); // 元素类型为 double；一维
    Matrix<int,1> ai1(n1);   // 元素类型为 int；一维
    ad1(7) = 0;            // 下标用 ()——Fortran 风格
    ad1[7] = 8;            // [] 也可以——C 风格
    Matrix<double,2> ad2(n1,n2); // 二维
    Matrix<double,3> ad3(n1,n2,n3); // 三维
    ad2(3,4) = 7.5;        // 真正的多维下标
    ad3(3,4,5) = 9.2;
}
```

✂ 可以看到，当你定义一个 Matrix 对象时，你指定了元素类型以及维数。显然，Matrix 是一个模板，元素类型和维数是模板参数。给定 Matrix 两个模板参数（如，Matrix<double,2>）后，就得到一个具体类型（类），你可以利用它来定义对象（如，Matrix<double,2> ad2(n1,n2)），其中的参数指定了矩阵的维。这样就定义了一个二维数组 ad2，两个维度的大小分别为 n1 和 n2。我们可以使用下标操作从 Matrix 中获取元素，对于一维 Matrix，指定一个下标即可；对于二维 Matrix，需指定两个下标；依此类推。

与内置数组类型和 vector 相似，Matrix 的下标是从 0 开始的（Fortran 语言从 1 开始）。也就是说，Matrix 元素的下标范围是 [0, max)，其中 max 是元素总数。

✂ 这种方式很简单，而且“完全出自于教科书”。如果你对这点有疑问，可以查阅适合的数学教科书，而不是程序设计手册。这里唯一的“小聪明”是，你可以省略维数：默认值是一维数组。注意，下标操作既可以使用 []（C 和 C++ 风格），也可以使用 ()（Fortran 风格），这使我们能更好地处理多维数组。[x] 下标运算符总是接受单一下标，得到矩阵对应的一行；如果 a 是 N 维矩阵，则 a[x] 为 N-1 维矩阵。(x,y,z) 下标运算符接受一个或多个下标，得到矩阵的一个元素，下标的数目必须与维数相等。

下面程序给出了 Matrix 的一些错误用法：

```
void f(int n1, int n2, int n3)
{
    Matrix<int,0> ai0; // 错误：无 0 维矩阵
```

```

Matrix<double,1> ad1(5);
Matrix<int,1> ai(5);
Matrix<double,1> ad11(7);

ad1(7) = 0;           // Matrix_error 异常 (7 越界)
ad1 = ai;             // 错误: 不同的元素类型
ad1 = ad11;          // Matrix_error 异常 (维数不同)
Matrix<double,2> ad2(n1); // 错误: 缺少第二维长度
ad2(3) = 7.5;         // 错误: 下标数目不对
ad2(1,2,3) = 7.5;    // 错误: 下标数目不对

Matrix<double,3> ad3(n1,n2,n3);
Matrix<double,3> ad33(n1,n2,n3);
ad3 = ad33;          // 正确: 相同的元素类型, 相同的维数
}

```

一种错误是声明的维数与使用的维数不符, 这种错误会在编译时被捕获。而越界错误则在运行时被捕获, 程序会抛出一个 `Matrix_error` 异常。

二维矩阵的第一维是行, 第二维是列, 因此可用 (row, column) 来索引二维矩阵 (二维数组)。也可以使用 `[row][column]`, 因为对于二维矩阵, 使用单个下标会得到一个一维矩阵 (行), 如下图所示:

				$a[1][2]$	
$a[0]:$	00	01	02	03	$a(1,2)$
$a[1]:$	10	11	12	13	
$a[2]:$	20	21	22	23	

此矩阵在内存中以“行主次序”存放:

00	01	02	03	10	11	12	13	20	21	22	23
----	----	----	----	----	----	----	----	----	----	----	----

一个 `Matrix` 对象是“知道”自己的维数和每维大小的, 因此, 将 `Matrix` 对象作为参数传递是很简单的:

```

void init(Matrix<int,2>& a) // 将每个元素初始化为一个字符值
{
    for (int i=0; i<a.dim1(); ++i)
        for (int j = 0; j<a.dim2(); ++j)
            a(i,j) = 10*i+j;
}

void print(const Matrix<int,2>& a) // 逐行打印元素
{
    for (int i=0; i<a.dim1(); ++i) {
        for (int j = 0; j<a.dim2(); ++j)
            cout << a(i,j) << '\t';
        cout << '\n';
    }
}

```

可以看到, `dim1()` 返回第一维的元素数目, `dim2()` 为第二维的元素数目, 依此类推。元素类型和维数是 `Matrix` 类型的一部分, 因此函数参数不能是任意 `Matrix` (但模板参数可以):

```

void init(Matrix& a); // 错误: 缺少元素类型和维数

```

注意，Matrix 库不支持矩阵整体运算，如将两个四维矩阵相加，或者将一个二维矩阵和一个一维矩阵相乘。为这些运算设计优美而高效的算法超出了当前这个库的范围，但我们可以在 Matrix 库之上设计这些算法（见习题 12）。

## 24.5.2 一维矩阵

我们可以对最简单的 Matrix——一维 Matrix 做什么操作呢？

如前所述，声明时可以省略维数，因为默认是一维：

```
Matrix<int,1> a1(8); // a1 是一个一维的 int 矩阵
Matrix<int> a(8);   // 表示 Matrix<int,1> a(8);
```

因此，a 和 a1 是相同的类型（Matrix<int,1>）。我们可以获取矩阵的大小（元素总数）和每一维的大小（这一维中的元素数目），对于一维矩阵，这两个值显然是相同的：

```
a.size();           // Matrix 中元素数目
a.dim1();           // 第一维中元素数目
```

我们可以按内存中的实际布局获取元素，即，获得指向第一个元素的指针：

```
int* p = a.data(); // 作为执行数组的指针提取数据
```

如果希望将 Matrix 对象传递给只接受指针参数的 C 风格的函数，这个操作是很有用的。我们可以像下面代码这样对矩阵进行下标操作：

```
a(i); // 第 i 个元素 (Fortran 风格)，但进行范围检查
a[i]; // 第 i 个元素 (C 风格)，进行范围检查
a(1,2); // 错误：a 是一个一维 Matrix
```



一些算法常常需要访问 Matrix 的一部分，这种“部分”被称为一个 slice()（一个子 Matrix 或一个元素范围），它有两种形式：

```
a.slice(i); // 从 a[i] 到矩阵末尾的元素
a.slice(i,n); // 从 a[i] 到 a[i+n-1] 的 n 个元素
```

下标和切片操作既可以作为右值，也可以作为左值，因为它们直接指向 Matrix 的元素，而不是创建拷贝。例如：

```
a.slice(4,4) = a.slice(0,4); // 将 a 的前一半赋予后一半
```

如果 a 的初值为

```
{1 2 3 4 5 6 7 8}
```

则执行这条语句后，a 变为：

```
{1 2 3 4 1 2 3 4}
```

注意，最常用的子矩阵是“开始元素段”和“末尾元素段”，即 a.slice(0,j)——范围 [0:j)，和 a.slice(j)——范围 [j:a.size())。特别地，上面那条语句可以写为：

```
a.slice(4) = a.slice(0,4); // 将 a 的前一半赋予后一半
```

也就是说，语法的设计上更倾向于常用情况。你可以指定超出 a 的范围的 i 和 n 的值，但最终的结果只取 a 的有效范围内的那段。例如，a.slice(i,a.size()) 实际得到范围是 [i:a.size())，而 a.slice(a.size()) 和 a.slice(a.size(),2) 得到的则是空矩阵。对于很多算法来说，这一特性在某些时候是很有用的。这个特性实际上是从数学领域借鉴来的。显然，a.slice(i,0) 是空的，我们不会故意写出这样的代码，但算法中使用 a.slice(i,n) 而 n 恰巧为 0 的情况是很

可能出现的。如果此时能得到空矩阵（而不是产生一个错误）的话，算法可以更为简洁。

Matrix 也支持（对 C++ 对象来说）常见的拷贝操作，实现所有元素的复制：

```
Matrix<int> a2 = a;    // 拷贝初始化
a = a2;              // 拷贝赋值
```

我们可以对矩阵中每个元素进行相同的内置运算（标量运算）：

```
a *= 7;              // 标量计算：对每个 i 执行 a[i]*=7（也支持 +=、-=、/= 等）
a = 7;              // 对每个 i 执行 a[i]=7
```

只要元素类型支持，其他赋值运算符和组合赋值运算符（=、+=、-=、/=、\*=、%=、^=、&=、|=、>>=、<<=）也都可以这样使用。我们还可以对矩阵每个元素执行相同的函数：

```
a.apply(f);         // 对每个元素 a[i] 执行 a[i]=f(a[i])
a.apply(f,7);       // 对每个元素 a[i] 执行 a[i]=f(a[i],7)
```

组合赋值运算符和函数 apply() 都修改了 Matrix 中的元素，如果你不希望这样，而是希望创建一个新的 Matrix 对象保存运算结果，可以这样做：

```
b = apply(abs,a);   // 创建一个新 Matrix，其 b(i)==abs(a[i])
```

其中的 abs 是标准库中的绝对值函数（见 24.8 节）。本质上，apply(f,x) 与 x.apply(f) 相对应，就像 + 与 += 对应一样。例如：

```
b = a*7;            // 对每个 i, b[i]=a[i]*7
a *= 7;            // 对每个 i, a[i]=a[i]*7
y = apply(f,x);    // 对每个 i, y[i]=f(x[i])
x.apply(f);        // 对每个 i, x[i]=f(x[i])
```

其运行结果 a==b 且 x==y。

在 Fortran 中，第二个版本的 apply 被称为“广播”函数，通常写作 f(x) 而不是 apply(f,x)。为了使这一特性对任意函数 f 都适用（而不是像 Fortran 中那样，只对少数函数适用），我们需要为“广播”操作定义一个名字，因此（再次）使用了 apply。

另外，为了匹配接受两个参数的成员函数 apply——a.apply(f,x)，我们提供了：

```
b = apply(f,a,x);   // 对每个 i, b[i]=f(a[i],x)
```

例如：

```
double scale(double d, double s) { return d*s; }
b = apply(scale,a,7); // 对每个 i, b[i]= a[i]*7
```

注意，“独立式”apply() 接受一个函数作为参数，该函数通过其参数生成运算结果，然后 apply() 利用运算结果初始化结果矩阵。它通常不修改参数中的 Matrix 对象。成员函数 apply 的不同之处在于，会修改原矩阵中元素。例如：

```
void scale_in_place(double& d, double s) { d *= s; }
b.apply(scale_in_place,7); // 对每个 i, b[i] *= 7
```

Matrix 库还支持传统数值计算库中一些最常用的函数：

```
Matrix<int> a3 = scale_and_add(a,8,a2); // 融合乘-加运算
int r = dot_product(a3,a);           // 点积
```

函数 scale\_and\_add() 通常被称为融合乘-加运算（fused multiply-add，简称 fma），它对矩阵中每个元素 i 执行 result(i)=arg1(i)\*arg2+arg3(i)。点积运算也被称为内积 inner\_product，我们已经在 16.5.3 节中对这种运算进行了介绍，它对矩阵中每个元素执行 result+=



`arg1(i)*arg2(i)`, `result` 的初值为 0。

一维数组是非常常用的，可以用内置数组类型、`vector` 或者 `Matrix` 来实现。我们之所以使用 `Matrix` 库，更多的是因为需要进行矩阵的整体运算，如 `*=`，或者需要使用多维矩阵。

像 `Matrix` 库这类的工具，可以被描述为“与数学描述很好匹配”或者“令程序员不必编写循环来处理矩阵中每个元素”。总之，利用这些库编写程序，代码会非常简洁，而且不容易出错。`Matrix` 库提供的那些操作，如拷贝、为所有元素赋值以及对所有元素执行相同运算等等，都使我们不必编写、维护循环代码（也不必为写出的循环是否正确而烦恼）。

`Matrix` 提供了两个构造函数，将数据从内置数组复制到 `Matrix` 对象中：

```
void some_function(double* p, int n)
{
    double val[] = { 1.2, 2.3, 3.4, 4.5 };
    Matrix<double> data(p,n);
    Matrix<double> constants(val);
    //...
}
```

如果数据是来自于某个未使用 `Matrix` 库的程序片段，是以数组或 `vector` 的形式提供的，这两个构造函数就非常有用。

注意，编译器能够推断出已经初始化的数组的规模，因此上面这段程序在定义 `constants` 时无须给出元素个数，即 4。另一方面，如果只是给出一个指针的话，编译器是无法获得元素数目的，因此定义 `data` 时，必须给出指针 `p` 指向的数组的规模 (`n`)。

### 24.5.3 二维矩阵

`Matrix` 库的设计思想是：不同维度的矩阵除了维数之外，其他方面实际上是非常相似的，因此，很多一维数组的概念都可用于二维数组：

```
Matrix<int,2> a(3,4);

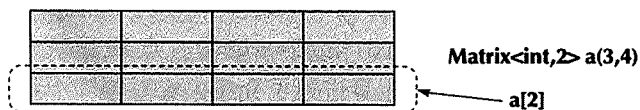
int s = a.size();           // 元素数
int d1 = a.dim1();         // 一行中的元素数
int d2 = a.dim2();         // 一列中的元素数
int* p = a.data();         // 提取数据——C 风格数组指针的形式
```

可以看到，我们能够获取元素总数和每一维的元素数目，可以获取矩阵在内存中存储区域的指针。

我们可以使用下标操作：

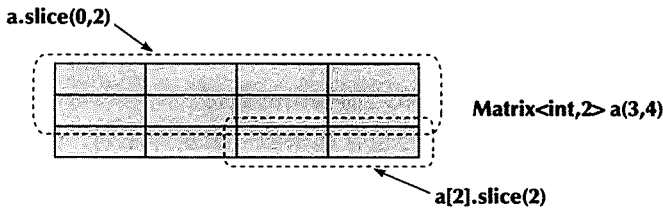
```
a(i,j);                    // 第 (i,j) 个元素 (Fortran 风格)，但进行范围检查
a[i];                      // 第 i 行 (C 风格)，进行范围检查
a[i][j];                   // 第 (i,j) 个元素 (C 风格)
```

对于一个二维矩阵，下标操作 `[i]` 获得它的第 `i` 行，即一个一维矩阵。这意味着，我们可以利用这一特性从二维矩阵中提取行，传递给那些需要一维矩阵甚至内置数组 (`a[i].data()`) 参数的操作或者函数。注意，`a(i,j)` 可能比 `a[i][j]` 更快，虽然这完全由编译器和优化器所决定。



我们可以进行切片操作：

```
a.slice(i);           // 从 a[i] 到矩阵末尾的那些行
a.slice(i,n);        // 从 a[i] 到 a[i+n-1] 的那些行
```



注意，一个二维矩阵的切片仍是二维矩阵（可能行数更少）。

二维矩阵的标量操作与一维矩阵类似，这些操作并不关心元素是如何组织的，只是简单地按元素在内存中存放的次序对它们进行处理而已。

```
Matrix<int,2> a2 = a;    // 拷贝初始化
a = a2;                // 拷贝赋值
a *= 7;                // 标量运算（以及 +=、-=、/= 等）
a.apply(f);            // 对每个元素 a(i,j), a(i,j)=f(a(i,j))
a.apply(f,7);          // 对每个元素 a(i,j), a(i,j)=f(a(i,j),7)
b=apply(f,a);          // 创建一个新 Matrix, b(i,j)=f(a(i,j))
b=apply(f,a,7);        // 创建一个新 Matrix, b(i,j)=f(a(i,j),7)
```

行交换常常是很有用的，Matrix 库也支持这种操作：

```
a.swap_rows(1,2);      // 交换行 a[1]↔a[2]
```

Matrix 并没有提供 swap\_columns() 操作，你可以自己实现它（见习题 11）。原因在于  $\triangleleft$  元素是按行优先次序存储的，行和列并不是完全对称的。这种不对称性也体现在  $[]$  得到矩阵的一行，但 Matrix 并未提供提取列的运算符。在下标操作 (i,j) 中，第一个下标 i 对应第 i 行。这种不对称性也反映了深层次的数学性质。

在现实世界中，有很多事物都是二维结构的，因此显然可以用二维矩阵来描述：

```
enum Piece { none, pawn, knight, queen, king, bishop, rook };
Matrix<Piece,2> board(8,8); // 一个国际象棋棋盘
```

```
const int white_start_row = 0;
const int black_start_row = 7;
```

```
Matrix<Piece> start_row
= {rook, knight, bishop, queen, king, bishop, knight, rook};
```

```
Matrix<Piece> clear_row(8); // 8 个元素，具有默认值
```

对 clear\_row 的初始化利用了两个事实：none==0；元素缺省情况下被初始化为 0。

对于 start\_row 和 clear\_row，我们可能希望写成这样：

```
board[white_start_row] = start_row; // 重置白方
for (int i = 1; i<7; ++i) board[i] = clear_row; // 清除棋盘中央
board[black_start_row] = start_row; // 重置黑方
```

注意，当我们使用  $[]$  提取一行时，我们得到一个左值（见 4.3 节），即，我们可以对其赋值。

## 24.5.4 矩阵 I/O

Matrix 库为一维和二维矩阵提供了非常简单的 I/O 功能：

```
Matrix<double> a(4);
cin >> a;
cout << a;
```

这段代码会读取以空白符间隔的 4 个 double 值，以花括号起止，例如：

```
{ 1.2 3.4 5.6 7.8 }
```

输出与输入内容很相似，因此你可以读取之前输出的数据。

二维矩阵的 I/O 操作简单读写花括号包含起来的一维矩阵序列，例如：

```
Matrix<int,2> m(2,2);
cin >> m;
cout << m;
```

这段代码读取类似下面所示的内容：

```
{
{ 1 2 }
{ 3 4 }
}
```

输出操作与输入操作非常接近。

矩阵的 << 和 >> 操作符主要是为了方便编写简单程序。如果你有更高的要求，就只能自己实现了。另外，<< 和 >> 的定义是在 MatrixIO.h 中（而不是 Matrix.h），因此，只是使用矩阵基本功能（而不使用 I/O 功能）的话，就不需要包含此头文件了。

### 24.5.5 三维矩阵

三维（以及更高维）矩阵与二维矩阵相比，除了维数更多外，其他方面非常相似。考察下面代码：

```
Matrix<int,3> a(10,20,30);

a.size();           // 元素数
a.dim1();           // 第一维元素数
a.dim2();           // 第二维元素数
a.dim3();           // 第三维元素数
int* p = a.data();  // 提取数据，以 C 风格数组指针的形式
a(i,j,k);           // 第 (i,j,k) 个元素 (Fortran 风格)，但进行范围检查
a[i];               // 第 i 行 (C 风格)，进行范围检查
a[i][j][k];         // 第 (i,j,k) 个元素 (C 风格)
a.slice(i);         // 从第 i 行到最后一行
a.slice(i,j);       // 从第 i 行到第 j 行
Matrix<int,3> a2 = a; // 拷贝初始化
a = a2;             // 拷贝赋值
a *= 7;             // 标量运算（以及 +=、-=、/= 等）
a.apply(f);         // 对每个元素 a(i,j,k), a(i,j,k)=f(a(i,j,k))
a.apply(f,7);       // 对每个元素 a(i,j,k), a(i,j,k)=f(a(i,j,k),7)
b=apply(f,a);       // 创建一个新 Matrix, b(i,j,k)=f(a(i,j,k))
b=apply(f,a,7);     // 创建一个新 Matrix, b(i,j,k)=f(a(i,j,k),7)
a.swap_rows(7,9);   // 交换行 a[7]↔a[9]
```

如果你理解二维矩阵的相关概念，那么也就能理解三维矩阵了。例如，在这段程序中，a 是一个三维矩阵，那么 a[i] 就是一个二维矩阵（如果 i 是合法下标），a[i][j] 就是一个一维矩阵（如果 j 是合法下标），而 a[i][j][k] 就是一个整型元素（如果 k 是合法下标）。

我们倾向于把现实世界看成三维的，因此三维矩阵显然可以用于现实世界的建模（例

如，使用笛卡尔坐标系进行物理仿真)：

```
int grid_nx;           // 网格分辨率；启动时设置
int grid_ny;
int grid_nz;
Matrix<double,3> cube(grid_nx, grid_ny, grid_nz);
```

如果我们将时间加入，作为第四维，那么就得到了一个四维空间，可用一个四维 Matrix 描述。依此类推。

作为 Matrix 的高级功能，它还支持  $N$  维矩阵，详见《The C++ Programming Language》第 29 章。

## 24.6 实例：求解线性方程组

对于一个数值计算程序，如果你能理解代码背后的数学含义，它对你来说就是有意义的，否则，它就是废话一堆。本节给出的例子是求解线性方程组问题，如果你学习过基本的线性代数知识，会觉得它很简单；否则，你就把它看作求解方案到代码的简单转换就可以了。

求解线性方程组是矩阵的一个相当实际也非常重要的应用，其目标是求解下面这种形式的线性方程组：

$$\begin{aligned} a_{1,1}x_1 + \cdots + a_{1,n}x_n &= b_1 \\ &\vdots \\ a_{n,1}x_1 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

其中  $x_1, \dots, x_n$  表示  $n$  个未知数， $a_{1,1}, \dots, a_{n,n}$  和  $b_1, \dots, b_n$  是给定的常量。简单起见，我们假定未知数和常量都是浮点值。问题的目标是找到能同时满足  $n$  个方程的未知数值。方程组可以更简洁地表示为矩阵和向量乘法的形式：

$$Ax = b$$

其中， $A$  是一个  $n \times n$  的系数方阵：

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \vdots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$$

向量  $x$  和  $b$  分别是未知数和常量向量：

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad \text{且} \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

这个系统可能有 0 个、1 个或者无穷多个解，这取决于系数矩阵  $A$  和向量  $b$ 。求解线性系统的方法有很多，本节使用一种经典的方法——高斯消去法（参见 Freeman 和 Phillips 的《Parallel Numerical Algorithms》；Stewart 的《Matrix Algorithms》(第一卷) 以及 Wood 的《Introduction to Numerical Analysis》)。首先，我们对  $A$  和  $b$  进行变换，使得  $A$  变为一个上三角矩阵。所谓上三角矩阵，就是对角线之下的所有元素均为 0。换句话说，系统转换为如下形式：

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ 0 & \ddots & \vdots \\ 0 & 0 & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

实现这个目标是很容易的。为了使  $a(i, j)$  变为 0，我们先将它乘以一个常量，使它等于第  $j$  列上的另一个元素，比如说等于  $a(k, j)$ 。然后，用第  $i$  个方程减去第  $k$  个方程， $a(i, j)$  即变为 0，矩阵第  $i$  行其他元素的值也相应发生改变。

如果这样一个变换最终使得对角线上所有元素都非 0，方程组就有唯一解，此解可以通过“回代”(back substitution) 求得。其过程是这样的，首先通过最后一个方程求得  $x_n$ ：

$$a_{n,n} x_n = b_n$$

显然， $x_n = b_n / a_{n,n}$ 。随后，将第  $n$  行从系统中消去，继续求解  $x_{n-1}$ ，依此类推，直至求解出  $x_1$  的值。在每个步骤中，都是除以  $a_{i,i}$ ，因此对角线上的元素必须非 0。否则，回代方法就失败了，意味着方程组有 0 个或无穷多个解。

### 24.6.1 经典的高斯消去法

我们现在来看看如何用 C++ 程序来表示上述计算方法。首先，定义两个要使用的具体 Matrix 类型，以简化程序：

```
typedef Numeric_lib::Matrix<double, 2> Matrix;
typedef Numeric_lib::Matrix<double, 1> Vector;
```

接下来我们将高斯消去法计算过程描述为程序：

```
Vector classical_gaussian_elimination(Matrix A, Vector b)
{
    classical_elimination(A, b);
    return back_substitution(A, b);
}
```

即，先为两个输入 A 和 b 创建拷贝（使用赋值参数），然后调用一个函数求解方程组，最后调用回代函数计算结果并将结果返回。关键之处在于，我们分解问题的方式和符号表示都完全来自于原始的数学描述。下面所要做的就是实现 classical\_elimination() 和 back\_substitution() 了，解决方案同样完全来自于数学教科书：

```
void classical_elimination(Matrix& A, Vector& b)
{
    const Index n = A.dim1();

    // 从第 1 列一直遍历到倒数第二列
    // 将对角线之下所有元素都填充 0
    for (Index j = 0; j < n - 1; ++j) {
        const double pivot = A(j, j);
        if (pivot == 0) throw Elim_failure(j);

        // 将第 i 行在对角线之下的元素都填充 0
        for (Index i = j + 1; i < n; ++i) {
            const double mult = A(i, j) / pivot;
            A[i].slice(j) = scale_and_add(A[j].slice(j), -mult, A[i].slice(j));
            b(i) -= mult * b(j); // 对 b 做相应变化
        }
    }
}
```

“pivot”表示当前行位于对角线上的元素，它必须是非 0。因为需要用它作为除数；如果它为 0，我们将放弃计算，抛出一个异常：

```
Vector back_substitution(const Matrix& A, const Vector& b)
```

```

{
    const Index n = A.dim1();
    Vector x(n);

    for (Index i = n-1; i >= 0; --i) {
        double s = b(i)-dot_product(A[i].slice(i+1),x.slice(i+1));

        if (double m = A(i,i))
            x(i) = s/m;
        else
            throw Back_subst_failure(i);
    }

    return x;
}

```

## 24.6.2 选取主元

pivot 为 0 的问题是可以避免的，我们可以对行进行排序，从而将 0 和较小的值从对角线上移开，这样就得到了一个更鲁棒的方案。“更鲁棒”是指对于舍入误差更不敏感。但是，随着我们将 0 置于对角线之下，元素值也会发生改变。因此，我们必须反复进行重排序，以将较小的值从对角线上移开（即，不能一次重排矩阵后就直接使用经典算法）：

```

void elim_with_partial_pivot(Matrix& A, Vector& b)
{
    const Index n = A.dim1();

    for (Index j = 0; j < n; ++j) {
        Index pivot_row = j;

        // 查找一个合适的主元：
        for (Index k = j+1; k < n; ++k)
            if (abs(A(k,j)) > abs(A(pivot_row,j))) pivot_row = k;

        // 如果我们找到了一个更好的主元，交换两行：
        if (pivot_row != j) {
            A.swap_rows(j,pivot_row);
            std::swap(b(j), b(pivot_row));
        }

        // 消去：
        for (Index i = j+1; i < n; ++i) {
            const double pivot = A(j,j);
            if (pivot==0) error("can't solve: pivot==0");
            const double mult = A(i,j)/pivot;
            A[i].slice(j) = scale_and_add(A[j].slice(j), -mult, A[i].slice(j));
            b(i) -= mult*b(j);
        }
    }
}

```

在这里我们使用了 `swap_rows()` 和 `scale_and_multiply()`，这样程序更符合习惯，我们也不必显式编写循环代码了。

## 24.6.3 测试

显然，我们下面应该对代码进行测试。幸运的是，我们有一种简单的测试方法：

```

void solve_random_system(Index n)
{
    Matrix A = random_matrix(n);      // 见 24.7 节
    Vector b = random_vector(n);
    cout << "A = " << A << "\n";
    cout << "b = " << b << "\n";

    try {
        Vector x = classical_gaussian_elimination(A, b);
        cout << "classical elim solution is x = " << x << "\n";
        Vector v = A*x;
        cout << "A*x = " << v << "\n";
    }
    catch(const exception& e) {
        cerr << e.what() << "\n";
    }
}

```

程序在三种情况下会进入 catch 子句：

- 代码中有 bug（但是，作为乐观主义者，我们不认为现在的代码中有 bug）。
- 输入内容使 classical\_elimination 出现错误（elim\_with\_partial\_pivot 在很多情况下可以做得更好）。
- 舍入误差导致问题。

但是，这种测试方法不像我们期望的那样接近实际，因为真正的随机矩阵不太可能导致 classical\_elimination 出现错误。

为了测试我们的程序，我们输出  $A*x$ ，其值应该与  $b$  相当。但考虑到存在舍入误差，若其值与  $b$  足够接近就应认为结果正确，这也是为什么测试程序中没有采用下面语句来判断结果是否正确的原因：

```
if (A*x!=b) error("substitution failed");
```

在计算机中，浮点数只是实数的近似，因此我们必须接受近似的计算结果。一般来说，应该避免使用 `==` 和 `!=` 来判断结果是否正确。

Matrix 库并没有定义矩阵与向量的乘法运算，因此我们为测试程序定义这个运算：

```

Vector operator*(const Matrix& m, const Vector& u)
{
    const Index n = m.dim1();
    Vector v(n);
    for (Index i = 0; i<n; ++i) v(i) = dot_product(m[i],u);
    return v;
}

```

我们再次看到，一个简单 Matrix 操作能帮助我们完成大部分工作。Matrix 的输出操作是在 MatrixIO.h 中定义的，我们在 24.5.4 节中已经对此进行了介绍。random\_matrix() 和 random\_vector() 是随机数的简单应用（见 24.7 节），这两个函数的实现留作练习。Index 是索引类型，它是用 typedef 定义的（见附录 A.16）。我们使用 using 将 Index 引入当前作用域：

```
using Numeric_lib::Index;
```

## 24.7 随机数

如果你要求人们说出一个随机数，大多数人会回答 7 或者 17，这表明人们认为这两个

数是“最随机的”。几乎不会有人回答 0，因为 0 被视为一个非常完美的数，没人认为它是“随机的”，因此被看作“最不随机”的数。从数学的观点来看，这绝对是错误的。随机数并不是指单个的数。我们常用的、常说的随机数，是指一个服从某种分布的序列，其特点是你无法很容易地从序列前一部分的内容预测出下一个数是什么。随机数的应用领域非常广，包括程序测试（用于生成大量测试用例）、游戏（确保游戏的下一步与之前的步骤不同）以及仿真（令仿真对象在参数限定范围内“随机地”运行）等等。

随机数既是一个实用工具，也是一个数学问题，它高度复杂，这与它在现实世界中的重要性是相匹配的。在本节中，我们只讨论随机数的最基本的内容，这些内容可用于简单的测试和仿真。在 `<random>` 中，标准库提供了复杂的方法来产生适应不同数学分布的随机数。这一随机数标准库基于下面两个基础概念：

- 发生器 (engine, 随机数发生器): 发生器是一个可以产生均匀分布整型值序列的函数对象。
- 分布 (distribution): 分布是一个函数对象，给定一个发生器产生的序列作为输入，分布可以按照相应数学公式产生一个值的序列。

例如，考虑 24.6.3 节中用到的 `random_vector()` 函数。调用 `random_vector(n)` 就会生成一个 `Matrix<double,1>` 类型的矩阵对象，它包含 `n` 个元素，元素值都是 `[0:n)` 之间的随机数。

```
Vector random_vector(Index n)
{
    Vector v(n);
    default_random_engine ran{};           // 生成整数
    uniform_real_distribution<> ureal(0,max); // 将 int 映射为 [0:max) 中的 double
    for (Index i = 0; i < n; ++i)
        v(i) = ureal(ran);

    return v;
}
```

默认发生器 (`default_random_engine`) 简单、代价低、容易运行。对日常应用，它已经足够了。对于更专业的应用，标准库提供了其他发生器，它们有着更好的随机性和不同的执行代价。例如，`linear_congruential_engine`, `mersenne_twister_engine` 和 `random_device` 等。如果你希望使用它们，或者希望获得比 `default_random_engine` 更好的性能，你需要查阅相关资料。请完成习题 10，你会对你所用系统的随机数发生器的质量有所体会。

`std_lib_facilities.h` 中的两个随机数发生器定义如下

```
int randint(int min, int max)
{
    static default_random_engine ran;
    return uniform_int_distribution<>(min,max)(ran);
}

int randint(int max)
{
    return randint(0,max);
}
```

这些函数经常会被用到，当然还有其他的，让我们看看正态分布如何产生：

```
auto gen = bind(normal_distribution<double>(15,4.0),
               default_random_engine{});
```



<functional> 中的标准库函数 `bind()` 构造了一个函数对象，当调用它时，它会调用它的一个参数，并将第二个参数作为这次调用的参数。因而在本例中，`gen()` 返回一个正态分布序列，其均值为 15，方差 4.0，使用的是 `default_random_engine`。我们可以这样使用这一函数：

```
vector<int> hist(2*15);

for (int i = 0; i < 500; ++i)           // 产生 500 个值的柱状图
    ++hist[int(round(gen()))];
for (int i = 0; i != hist.size(); ++i) { // 输出柱状图
    cout << i << '\t';
    for (int j = 0; j != hist[i]; ++j)
        cout << '*';
    cout << '\n';
}
```

我们得到：

```
0
1
2
3  **
4  *
5  *****
6  ****
7  ****
8  *****
9  *****
10 *****
11 *****
12 *****
13 *****
14 *****
15 *****
16 *****
17 *****
18 *****
19 *****
20 *****
21 *****
22 *****
23 *****
24 *****
25 *
26 *
27
28
29
```

正态分布经常被用到，它也被称为高斯分布或者（显然的原因）“钟形曲线”。其他分布包括 `bernoulli_distribution`，`exponential_distribution` 和 `chi_squared_distribution`。在《The C++ Programming Language》中你能找到详细介绍。整数分布的返回值是闭区间 `[a:b]`，而实数（浮点）分布的返回值是开区间 `[a:b)`。

默认情况下，程序的每次运行中，发生器（除了 `random_device`）产生同样的序列。这非常有利于程序调试。如果希望同一发生器产生不同的序列，我们需要设定不同的初值。这一初始化过程被称为“种子”。例如：

```

auto gen1 = bind(uniform_int_distribution<>(0,9),
    default_random_engine{});
auto gen2 = bind(uniform_int_distribution<>(0,9),
    default_random_engine{10});
auto gen3 = bind(uniform_int_distribution<>(0,9),
    default_random_engine{5});

```

为了获得不可预测的序列，我们经常使用当前时间（以纳秒为单位，见 26.6.1 节）或其他类似的事物作为种子。

## 24.8 标准数学函数

标准库中也提供了常用的标准数学函数（cos、sin、log 等等），这些函数的定义都在 `<cmath>` 中：

标准数学函数	
<code>abs(x)</code>	绝对值
<code>ceil(x)</code>	向上取整—— $\lceil x \rceil$ 的最大整数
<code>floor(x)</code>	向下取整—— $\lfloor x \rfloor$ 的最小整数
<code>sqrt(x)</code>	平方根， $x$ 必须是非负数
<code>cos(x)</code>	余弦
<code>sin(x)</code>	正弦
<code>tan(x)</code>	正切
<code>acos(x)</code>	反余弦，结果取为非负数
<code>asin(x)</code>	反正弦，结果取最接近 0 的值
<code>atan(x)</code>	反正切
<code>sinh(x)</code>	双曲正弦
<code>cosh(x)</code>	双曲余弦
<code>tanh(x)</code>	双曲正切
<code>exp(x)</code>	$e$ 的指数
<code>log(x)</code>	自然对数，即以 $e$ 为底， $x$ 必须是正数
<code>log10(x)</code>	以 10 为底的对数

标准数学函数的参数可以是如下类型：`float`、`double`、`long double` 和 `complex`（见 24.9 节）。如果你需要做浮点计算，会发现这些函数非常有用。你需要了解更多细节的话，相关文档非常多，标准库的联机文档就可以作为一个很好的入门材料。

如果一个标准数学函数无法计算出有效结果，它会设置变量 `errno`。例如：

```

errno = 0;
double s2 = sqrt(-1);
if (errno) cerr << "something went wrong with something somewhere";
if (errno == EDOM) // 定义域错误
    cerr << "sqrt() not defined for negative argument";
pow(very_large, 2); // 不是一个好主意
if (errno == ERANGE) // 范围错误
    cerr << "pow(" << very_large << ", 2) too large for a double";

```

如果你要做一些重要的计算，在计算完成之后应该检查 `errno` 的值，确保它仍为 0。如果 `errno` 变为非 0，一定是出现错误了。请查阅手册或者联机文档，检查哪些数学函数可以设置 `errno`，以及 `errno` 的不同的值分别代表什么错误类型。

如上例所示，`errno` 非 0 仅仅表示“什么地方发生错误了”，获得具体错误类型和错误位置还要正确检测 `errno` 的值。标准库之外的函数在发生错误时也可能设置 `errno` 的值，因此

在检查 `errno` 值的时候一定要小心，以确保找到正确的错误位置。正确的方法是在调用标准库函数前确认 `errno==0`，在函数返回后立刻检查 `errno` 的值，这样就可以保证 `errno` 的值反映了函数的错误类型。就像上例中那样，我们可以检测 `errno` 是否等于 `EDOM` 和 `ERANGE` 来判断错误类型。其中 `EDOM` 表示定义域错误（即参数错误），而 `ERANGE` 表示越界错误（即参数导致的问题）。

基于 `errno` 的错误处理技术已经有很长历史了，它的产生可以追溯到第一个 C 语言数学函数出现的年代（1975 年）。

## 24.9 复数

复数在科学和工程计算中被广泛使用。我们假定你已经了解了相关的数学知识，因此本节只介绍如何用 ISO C++ 标准库来表达复数运算。复数及其标准数学函数的定义都在 `<complex>` 中：

```
template<class Scalar> class complex {
    // 一个复数是一对标量值，本质上是一个坐标对
    Scalar re, im;
public:
    constexpr complex(const Scalar & r, const Scalar & i) :re(r), im(i) {}
    constexpr complex(const Scalar & r) :re(r),im(Scalar ()) {}
    complex() :re(Scalar ()), im(Scalar ()) {}

    constexpr Scalar real() { return re; } // 实部
    constexpr Scalar imag() { return im; } // 虚部

    // 运算符： += -= *= /=
};
```

标准库支持标量类型 `float`、`double` 和 `long double` 构成的复数，除了 `complex` 的成员函数和标准数学函数外（见 24.8 节），`<complex>` 还提供了大量有用的运算：

复数运算	
<code>z1+z2</code>	加法
<code>z1-z2</code>	减法
<code>z1*z2</code>	乘法
<code>z1/z2</code>	除法
<code>z1==z2</code>	相等比较
<code>z1!=z2</code>	不等比较
<code>norm(z)</code>	<code>abs(z)</code> 的平方
<code>conj(z)</code>	共轭：如果 $z$ 是 $(re,im)$ ，则 <code>conj(z)</code> 为 $(re,-im)$
<code>polar(rho,theta)</code>	用给定的极坐标 $(rho,theta)$ 构造一个复数
<code>real(z)</code>	实部
<code>imag(z)</code>	虚部
<code>abs(z)</code>	模，也称为 $rho$
<code>arg(z)</code>	辐角，也称为 $theta$
<code>out&lt;&lt;z</code>	输出
<code>in&gt;&gt;z</code>	输入

注意：`complex` 未提供 `<` 或者 `%`。

`complex<T>` 的使用与 `double` 这样的内置类型完全一样。例如：

```

Using cmplx = complex<double>; // 有时 complex<double> 太烦琐了

void f(cmplx z, vector<cmplx>& vc)
{
    cmplx z2 = pow(z,2);
    cmplx z3 = z2*9.3+vc[3];
    cmplx sum = accumulate(vc.begin(), vc.end(), cmplx{});
    // ...
}

```

请记住，并不是对所有的 int 和 double 运算，complex 都定义了相应的复数运算。例如：

```
if (z2<z3) // 错误：复数没有 < 运算符
```

注意，C++ 标准库中复数的表示方式（布局）与 C 和 Fortran 中的对应类型是兼容的。

## 24.10 参考文献

实际上，本章所讨论的话题，如舍入误差、矩阵运算以及复数运算等等，如果孤立地看，没有任何意义。本章只是面向那些具备一定数学基础，需要进行数值计算的人，简单地介绍 C++ 中的一些相关特性。

假若你在这些领域上已经有些荒废了，或者仅仅是有些好奇，我们向你推荐以下资源。

MacTutor 数学史档案，<http://www-gap.dcs.st-and.ac.uk/~history>：

- 对所有喜欢数学或者仅仅是需要使用数学的人都是一个极好的网站。
- 对一些希望了解数学的人性化一面的人（例如，想知道谁是唯一一个获得过奥运会金牌的知名数学家），这是一个极好的网站。在网站上可以找到：
  - 著名数学家的传略和成就。
  - 一些“古董”。
- 著名的函数曲线。
- 著名数学问题。
- 数学领域的各种主题。
  - 代数
  - 分析
  - 数论
  - 几何和拓扑学
  - 数学物理学
  - 数理天文学
    - ◆ 数学历史
    - ◆ ...

Freeman, T. L., and Chris Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.

Gullberg, Jan. *Mathematics-From the Birth of Numbers*. W W. Norton, 1996. ISBN 039304002X.

最有趣的基础数学书籍之一。少有的既能愉快阅读，又能查找到特定主题（如矩阵）的数学书籍。

Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1998. ISBN: 0201896842.

Stewart, G. W. *Matrix Algorithms, Volume I: Basic Decompositions*. SIAM, 1998. ISBN

0898714141.

Wood, Alistair. *Introduction to Numerical Analysis*. Addison-Wesley, 1999. ISBN 020194291X.

## 简单练习

1. 打印 char、short、int、long、float、double、int\* 和 double\* 的宽度（利用 sizeof，而不是 <limits>）。
2. 用 sizeof 打印 Matrix<int> a(10)、Matrix<int> b(10)、Matrix<double> c(10)、Matrix<int,2> d(10,10)、Matrix<int,3> e(10,10,10) 的宽度。
3. 打印第 2 题中每个矩阵的元素数目。
4. 编写程序，从 cin 读入 int 型整数，输出每个整数的平方根 sqrt()，或者 “no square root”（检查 sqrt() 的返回值，判断运算是否非法）。
5. 从输入读取 10 个浮点值，将它们存入一个 Matrix<double> 对象。Matrix 没有定义 push\_back() 操作，所以小心处理错误的 double 值的情况。最后输出矩阵。
6. 计算 [0,n]\*[0,m] 乘法表，将它保存在一个二维矩阵中。n 和 m 的值从 cin 读入。最后，以漂亮的格式输出乘法表（假定 m 足够小，屏幕的一行能容纳乘法表的一行）。
7. 从 cin 读入 10 个复数 complex<double>（cin 支持 complex 的 >> 操作），将它们保存在一个矩阵中。计算并输出这 10 个复数的和。
8. 读入 6 个 int 型整数，存入一个矩阵对象 Matrix<int,2> m(2,3) 中，并输出这些整数。

## 思考题

1. 哪些人使用数值计算？
2. 精度是什么？
3. 什么是溢出？
4. 一般来说 double 的宽度是多少？int 呢？
5. 你如何检测溢出？
6. 在哪里能找到数值限制？如，最大的 int 值是多大？
7. 数组是什么？行和列呢？
8. C 风格的多维数组是什么？
9. 程序设计语言中支持矩阵运算的部分（如矩阵库）必备的特性是什么？
10. 矩阵的维是什么？
11. 一个矩阵可以有多少维（从理论上、数学上看）？
12. 子矩阵是什么？
13. 什么是“广播”运算？请举出一些例子。
14. Fortran 风格的下标和 C 风格的下标有什么区别？
15. 如何对矩阵中每个元素都进行一个相同的操作？请举例。
16. 融合运算是什么？
17. 请定义点积运算。
18. 什么是线性代数？
19. 什么高斯消去法？
20. (线性代数中的、“现实生活中”的) 主元 (pivot) 是什么？

21. 什么令一个数随机?
22. 什么是均匀分布?
23. 从哪里可以找到标准数学函数? 它们支持哪些类型的参数?
24. 复数的虚部是什么?
25.  $-1$  的平方根是什么?

## 术语

array (数组)	Matrix
C	multidimensional (多维)
column (列)	random number (随机数)
complex number (复数)	real (实数)
dimension (维)	row (行)
dot product (点积)	scaling (标量的)
element-wise operation (逐元素运算)	size (大小)
errno	sizeof
Fortra	slicing (子矩阵)
fused operation (融合运算)	subscripting (下标)
imaginary (虚部)	uniform distribution (均匀分布)

## 习题

1. `a.apply(f)` 和 `apply(f,a)` 所使用的函数参数 `f` 是不同的。为两个 `apply()` 分别编写一个 `triple()` 函数, 完成的功能都是将数组 `{ 1 2 3 4 5 }` 中元素值乘三倍。定义一个统一的 `triple()` 函数, 既能用于 `a.apply(triple)`, 又能用于 `apply(triple,a)`。解释一下, 为什么采取这种方式编写 `apply()` 所使用的函数并不是一种好的方法。
2. 重做习题 1, 但实现的是函数对象而不是函数。提示: `Matrix.h` 中有示例。
3. 本题不适合初学者 (利用本书中介绍的工具无法完成此题): 编写一个 `apply(f,a)`, 对于函数参数 `f`, `apply` 可以接受的类型有 `void (T&)`、`T (const T&)` 以及对应的函数对象。提示: 参考 `Boost::bind`。
4. 编译、测试高斯消去法程序。
5. 用 `A=={{0 1} {1 0}}` 和 `b=={5 6}` 测试高斯消去法程序, 观察错误。然后测试 `elim_with_partial_pivot()`。
6. 将高斯消去法程序中的向量运算 `dot_product()` 和 `scale_and_add()` 替换为循环。测试这个程序, 并添加必要的注释, 提高代码的可读性。
7. 重写高斯消去法程序, 不使用 `Matrix` 库, 只使用内置数组或 `vector`。
8. 以动画形式演示高斯消去法。
9. 重写非成员函数 `apply()`, 返回所应用函数返回类型的数组, 即, 如果 `f` 的返回类型为 `R`, 则 `apply(f,a)` 返回一个 `Matrix<R>` 对象。注意: 本题的求解要用到本书未涉及的一些模板方面的知识。
10. 你所使用的 `default_random_engine` 到底有多随机? 编写程序, 读入两个整数 `n` 和 `d`, 调用 `randint(n)` `d` 次, 记录生成的随机数。输出随机数的分布, 即 `[0:n]` 间每个值出现的次

数，观察计数的相似程度。测试较小的  $n$  和  $d$ ，观察生成较少的随机数是否会导致明显的分布不均。

11. 编写与 24.5.3 节中 `swap_rows()` 对应的 `swap_columns()`。显然，你必须阅读、理解 `Matrix` 库中的一些代码，才能完成这个函数的设计。不必太在意效率：让 `swap_columns()` 与 `swap_rows()` 一样快是不可能的。
12. 实现

```
Matrix<double> operator*(Matrix<double,2>&,Matrix<double>&);
```

和

```
Matrix<double,N>operator+(Matrix<double,N>&,Matrix<double,N>&);
```

如果需要，请在教科书中寻找相关的数学定义。

## 附言

如果你不太喜欢数学，那么你可能也不太喜欢这一章，你可能应该选择一些不需要本章知识的工作。另一方面，如果你喜欢数学，我们希望你能仔细体会基本的数学概念和代码实现之间是如此接近。

# 嵌入式系统程序设计

“不安全”就意味着“有人可能付出生命代价”。

——安全官员

本章介绍嵌入式程序设计：介绍为“小设备”编写程序的基本知识，而不是为那些配置有屏幕和键盘的传统计算机编写程序。我们重点讨论编写“更接近硬件”的程序所需的基本原理、程序设计技术、语言特性和编码规范。语言方面主要包括资源管理、内存管理、指针和数组的使用以及位运算等问题，重点是底层特性的使用和替代方法。我们不会介绍特殊的机器架构或者直接访问硬件设备的方法，这些应该是专门文档和手册介绍的内容。本章最后会给出一个加密 / 解密算法的实现作为示例。

## 25.1 嵌入式系统

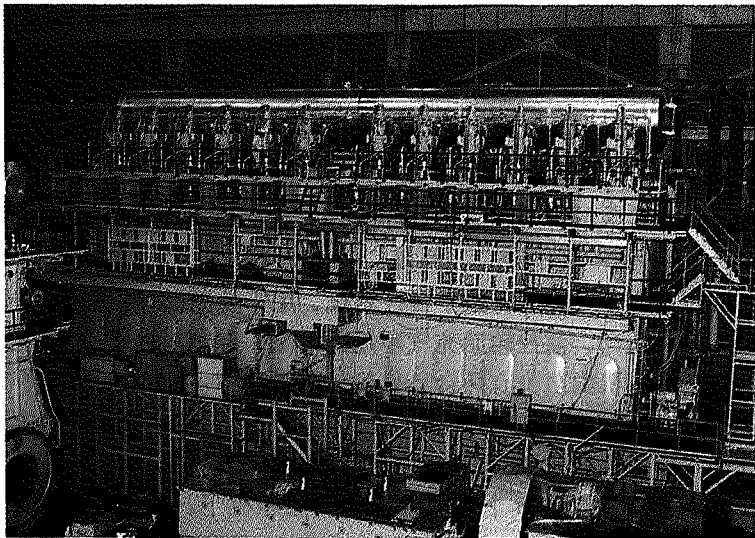
实际上，世界上大多数的计算机系统都不太像“计算机”。它们可能是一个大型系统的组成部分，或者仅仅是一个“小设备”。例如：

- 汽车：一台新式汽车可能配有数十台计算机，用于控制燃油喷射、监控引擎性能、调节收音机、控制刹车、监控轮胎充气不足的情况、控制风挡雨刷等。
- 电话：一部新式手机内至少有两台计算机，通常其中一台专门用于信号处理。
- 飞机：一架现代飞机内也有多台计算机，完成从运行乘客娱乐系统到摆动翼端优化飞行特性等各种各样的任务。
- 照相机：现在已经有配置 5 个以上处理器的照相机了，甚至每个镜头都有独立的处理器。
- 信用卡（“智能卡”的一种）。
- 医疗设备监测器和控制器（例如 CAT 扫描仪）。
- 电梯（升降机）。
- PDA（Personal Digital Assistant，个人数字助理）。
- 打印机控制器。
- 音响系统。
- MP3 播放器。
- 厨房用具（如电饭煲和烤面包机）。
- 电话交换设备（通常包含数千个专用计算机）。
- 水泵控制器（抽水或者抽油等）。
- 焊接机器人：在人类焊工无法进入的狭小或者危险的环境中完成焊接任务。
- 风力涡轮机：一些风力涡轮机高达 200 米（650 英尺），能产生数兆瓦电能。
- 防潮闸控制器。
- 装配线质量监控器。



- 条码阅读器。
- 汽车组装机器人。
- 离心机控制器（很多医学分析过程中要用到）。
- 磁盘驱动器控制器。

这些计算机都是大型系统的一部分。这些“大型系统”通常看起来不像一台计算机，我们通常也不把它们看作计算机。当看到一辆小轿车沿着大街驶来，我们绝不会说：“快看！那儿有一个分布式计算机系统！”是的，小轿车也是一个分布式计算机系统，但其运行已经与机械系统、电子系统以及电气系统非常紧密地结合在一起了，我们实际上无法孤立地考察计算机系统。它在计算上（时间上和空间上）的限制和程序正确性的定义上都已经不能与整个系统分开了。通常，一台嵌入式计算机控制某个物理设备，计算机的正确行为被定义为物理设备的正常操作。我们来看一台大型的船用柴油机：



注意位于 5 号汽缸前端的人。这是一台巨大的引擎，这种引擎为大型船舶提供动力。如果一台这样的引擎发生故障，你就会在早报的头版上看到相关的新闻。在这个引擎上，每个汽缸前端都有一个由三台计算机组成的汽缸控制系统。每个汽缸控制系统都通过两个独立的网络和引擎控制系统（由另外三台计算机组成）相连。引擎控制系统又连接到控制室，在那里，机械工程师可以通过一个专门的 GUI 系统与引擎控制系统交互。在航线控制中心，可以使用无线电系统（通过卫星）对整个系统进行远程监控。更多的实例可参考第 1 章。

那么，从一个程序员的观点来看，运行在这样一台引擎内的计算机之上的程序有什么特殊之处呢？更一般地，为各种各样的嵌入式系统编写程序时，有哪些问题是原来编写“普通程序”时不必过于担心，但现在需要特别关注的呢？

- 通常，嵌入式系统中，可靠性（reliability）是至关重要的：因为故障可能是突如其来的、损失巨大的（可能“数以亿计”）而且可能是致命的（如船舶失事时船上的人员或者类似环境下的动物）。
- 通常，嵌入式系统中，资源（内存、处理器、能源等）是有限的：对于引擎中的计算机，这可能不是一个问题。但请考虑手机、传感器、航天探测器等系统，其中的资

源问题就很严重了。在我们的日常应用中，配置主频 2GHz 的双核 CPU 和 8GB 内存的笔记本很常见，但飞机上或航天探测器中的关键计算机系统可能只配备 60MHz 的处理器和 256KB 的内存，而一个小型装置中的计算机系统可能只有主频低于 1MHz 的处理器和几百个字的内存。能够抵抗环境灾难（如振动、碰撞、不稳定的电力供应、温度过高过低、湿度过高、人为破坏等等）的计算机通常比普通的笔记本慢很多。

- 通常，在嵌入式系统中，实时响应（real-time response）是必需的：如果燃油喷射器错过了一个喷射周期，就意味着一个能输出十万马力的非常复杂的系统会发生糟糕的事情。错过几个周期，即不能正常工作一秒钟左右，推进器就会产生奇怪的行为——可能产生 33 英尺（10 米）的距离偏差和 130 吨的动力偏差。你显然不希望发生这样的事情。
- 通常，嵌入式系统需要一年到头不间断地正常运行：或许计算机系统是工作在绕地球轨道运行的通信卫星上；又或许系统非常便宜，因而制造量极为巨大，较高的返修率会给厂商带来极大损失（如 MP3 播放器、带嵌入式芯片的信用卡以及汽车的燃油喷射器）。在美国，电话骨干网交换机的强制可靠性标准是 20 年中停机时间在 20 分钟以内（这甚至没有减去更新交换机程序所需的停机时间）。
- 通常，对于嵌入式系统，内行维护（hands-on maintenance）是不可行的或者非常少见：对于一艘大型船舶，大概每两年左右进港一次进行整体维护，这时你就可进行计算机系统的维护了，但前提是计算机专家此时恰好有时间，又恰好在船舶停靠地。计划外的内行维护是不可行的，例如，当船舶在太平洋中央遇到大风暴时，是不允许出现任何故障的。再如，你是不可能派人去维修绕火星飞行的航天探测器的。

很少有系统面临所有这些问题，但即使仅仅面临其中一个问题，也需要领域专家来解决。本章的目标不是使你立刻成为专家，设定这样的目标是很愚蠢也是很不负责任的。我们的目标是使你了解基本问题和解决问题的基本概念，使你对构造这类系统的基本技术有所体会。也许经过学习后，你就会对这些重要的技术产生兴趣，产生深入学习的愿望。设计和实现嵌入式系统的人，对我们科技文明的很多方面都相当重要。

那么本章内容与初学者有关吗？与 C++ 程序员有关吗？回答是肯定的。现实生活中，嵌入式系统的数量远远多于传统 PC。我们遇到的程序设计中，有一大部分与嵌入式系统有关，因而你的第一个实际工作就可能涉及嵌入式系统程序设计。而且，本节开头列出的嵌入式系统的例子，都是我本人亲眼所见的使用 C++ 进行程序设计的实际例子。

## 25.2 基本概念

嵌入式系统程序设计工作中的很大一部分与普通程序设计没有太大区别，因此本书介绍的大部分概念和技术仍然适用。不过，本章的重点是描述两者之间的不同之处：我们必须调整程序设计语言工具的使用方式，以适应嵌入式系统的一些限制，而且通常我们需要以底层方式访问硬件。

- 正确性（correctness）：对于嵌入式系统，正确性比普通系统更为重要。“正确性”不只是一个抽象概念。在嵌入式系统中，一个程序的正确性不仅仅是一个产生正确结果的问题，还意味着要在正确的时间得到正确的结果以及只使用允许范围内的资源。理想情况下，我们要小心、仔细地定义正确性包含哪些内容，但通常，完整的定义


只有在实验之后才能得到。一般来说，整个系统都实现完毕后（不仅包括计算机系统的实现，还包含物理设备等其他部分）才能进行关键性的实验。完整的正确性定义对于嵌入式系统来说既非常困难又非常重要。“非常困难”是指“给定的时间和可用的资源不足以完成”，我们必须竭尽所能使用所有可用的工具和技术。幸运的是，在某个特定领域，可运用的规范、仿真方法、测试技术和其他技术可能超乎我们的想象，有效使用的话可能帮助我们完成目标。“非常重要”是指“故障会导致极大的伤害甚至毁灭性的后果”。

- ✘ ● 容错 (fault tolerance)：我们必须仔细定义程序应该处理哪些情况。例如，对于一个普通的学生练习程序，如果我们在程序演示时踢掉电线，还要求它能继续正常工作，显然是不公平的。对于一个普通的 PC 应用程序，不应该要求它能处理电源故障。但是，对于嵌入式系统，电源故障并不罕见，在某些情况下程序应该有能力对此进行处理。例如，系统的关键部件可能配备双电源、备用电池等等。“我假定硬件正常工作”不应成为应用程序不进行错误处理的借口。因为，经过很长时间运行，面对各种各样的工作条件，硬件发生故障是很正常的。例如，一些电话交换机程序和—些航天器程序会假设计算机内存中的值迟早会发生位偏转（例如，从 0 变成 1）。或者，可能内存中某个位一直保持为 1，将其改变为 0 的操作都会被忽略掉。如果内存足够大，而且使用时间较长的话，这类错误总是会发生。如果内存暴露于强辐射中，例如系统运行于地球大气层之外，这种错误就会很快发生。当我们设计一个系统时（无论是不是嵌入式系统），应该明确系统应提供的容错能力。通常的默认情况是假定硬件会按规定方式工作，对于更重要的系统，这个假设就需要调整了。
- ✘ ● 不停机 (no downtime)：嵌入式系统一般需要长时间运行，其间不进行软件更新，也无须有经验的了解系统实现的操作员人工干预。“长时间”可以是几天、几个月、几年甚至硬件的整个生命周期。其他类型的系统可能也有这样的需求，但嵌入式系统与大量“普通应用”和本书中的示例程序、系统程序有着非常大的不同。这种“必须永远运行”的需求意味着对错误处理和资源管理的极高要求。那“资源”又是什么呢？所谓资源就是机器只能提供有限数量的那些东西。在程序中，你需要显式地获取资源（“获取资源” (acquire the resource)， “分配” (allocate)），使用完毕后还应该将其归还系统（“释放” (release、free、deallocate)，可以显式或隐式归还）。资源的例子很多，如内存、文件句柄、网络连接（套接字）以及锁等等。对于长期运行的程序，除了一些需要一直使用的资源之外，其他资源在使用完毕后都必须释放。例如，如果一个程序每天都忘记关闭一个文件，会使大多数系统在大约一个月后崩溃。如果一个程序每天都忘记释放 100 字节的内存，那么一年中就会浪费大约 32KB 内存——这足以令一个小型设备在几个月后崩溃。这种资源“泄漏”问题最令人讨厌的是，程序会良好运行几个月，然后突然崩溃。如果程序必然崩溃，那么我们宁愿它尽可能早地崩溃，以便我们及时发现、修正错误。我们希望系统能在提交用户之前就早早地暴露问题，而不是在用户使用过程中出现错误。
- ✘ ● 实时性限制 (real-time constraint)：对于一个嵌入式系统，如果每个操作都严格要求在一个时限内完成，那么我们称它是硬实时的 (hard real time)。如果大多数时间要求操作在时限内完成，但对于偶尔的超时能够忍受，那么就称这种系统是软实时的 (soft real time)。汽车车窗控制器和立体声音响放大器都属于软实时系统：人们不会

注意到车窗的移动延迟了零点几秒钟；只有受过训练的人才会察觉到音高变化时几毫秒的延迟。一个硬实时系统的例子是燃油喷射器，喷射燃油的时间必须严格地与活塞运动同步。如果时间偏差哪怕零点几毫秒，引擎性能就会受影响，磨损也会更严重。如果时间偏差更大的话，甚至会使引擎停止工作，从而导致一起事故甚至灾难。

- 可预测性 (predictability)：对于嵌入式系统程序来说，可预测性是一个关键概念。显然，这个术语有很多直观的含义，但对于嵌入式系统程序设计，它有一个专门的定义：如果一个操作在一台给定的计算机上每次的执行时间总是相同的，而同类操作的执行时间也都是相同的，那么我们就称这个操作是可预测的。如，若  $x$  和  $y$  是整数， $x+y$  的执行时间是不变的，而  $xx+yy$  ( $xx$  和  $yy$  也是整数) 的执行时间与  $x+y$  也总是相同的。通常，我们可以忽略由系统架构造成的细微的运行时间差异（如，高速缓存和流水线造成的运行时间差异），操作的运行时间就取其上限。在硬实时系统中，绝对不能使用不可预测的操作，在软实时系统中可以使用，但也必须非常小心。一个经典的不可预测操作的例子是列表的顺序搜索 (`find()`)，列表的元素数目是未知的，也很难给出其上限。只有当我们能确切地预测列表元素数目或者是能预测最大元素数目时，顺序搜索才能用于硬实时系统。也就是说，为了保证请求在给定时限内被响应，我们必须能够（也许需要借助于代码分析工具）计算所有在时限之前可能执行的代码流的执行时间。
- 并发性 (concurrency)：一个嵌入式系统通常需要响应来自于外部的的事件。因而程序可能需要同时处理很多事情，因为有可能很多外部事件同时发生。程序同时处理多个动作，称为并发 (concurrent) 或者并行 (parallel)。不幸的是，那些吸引人的、有难度的、重要的并行程序设计知识已经超出了本书的讨论范围。

## 25.2.1 可预测性

C++ 的可预测性相当好，但还不够完美。除了以下几个语言特性外，所有其他 C++ 语言特性（包括虚函数调用）都是可预测的： 

- 动态内存空间分配 `new` 和 `delete`（见 25.3 节）。
- 异常（见 14.5 节）。
- 动态类型转换 `dynamic_cast`（见附录 A.5.7）。

应该避免在硬实时系统中使用这些语言特性。我们将在 25.3 节详细讨论 `new` 和 `delete` 所存在的问题，这是一个根本性的问题，任何语言实现都会存在。注意，标准库中的 `string` 和标准容器（`vector`、`map` 等等）间接地使用了动态内存分配，因此它们也是不可预测的。`dynamic_cast` 的问题则是当前实现所导致的，而非根本性问题。

异常的问题在于，对每个 `throw`，如果不考察更大范围的代码，程序员无法知道需要花费多长时间才能找到与之匹配的 `catch`，甚至是否存在这样一个 `catch` 都无法获知。在一个嵌入式系统程序中，最好期盼确实存在这样一个 `catch`，而且在抛出异常后能及时执行到这个 `catch`，因为期望只会依赖调试工具的 C++ 程序员发现这类问题，实在是不可靠。如果有这么一个工具，它能找到每个 `throw` 所匹配的 `catch`，并能计算出多长时间能到达 `catch`，就能解决异常所面临的这个问题了。但到目前为止，这样的工具还处于研究阶段，离实用还很遥远。因此，如果程序必须是可预测的，你就需要使用返回代码等老式技术来编写错误处理程序，虽然这类技术可能冗长乏味，但它们是可预测的。

## 25.2.2 理想

⚠ 编写嵌入式系统程序的过程中存在这样一种危险：对性能和可靠性的追求导致程序员倒退到只使用低级语言特性的地步。如果是编写一小段程序，这样做还是可行的。但是，一般情况下，它容易使整体设计陷入混乱，使程序的正确性难以验证，还会大大增加系统开发的成本和时间。

✎ 与以往一样，我们的理想是尽量使用较高层次的抽象，以便能够很好地描述要解决的问题。不要退回到编写汇编代码的地步！与以往一样，尽可能直接用代码表达你的思想（已给定所有条件）。与以往一样，努力编写最清晰、最干净、最易维护的代码。除非真的需要，否则不要执着于代码优化。性能（时间、空间）对一个嵌入式系统通常很重要，但是试图“榨干”每一小段代码的性能极限就是误入歧途了。而且，对于很多嵌入式系统而言，重要的是正确性和“足够快”。超越“足够快”就没有意义了，系统只能空闲下来，等待进行下一个操作。在编写每一小段代码时都试图达到最高效率，既花费大量时间，又会导致大量 bug，而且通常还会导致失去优化的机会，因为这样实现的算法和数据结构难以理解、难以修改。例如，这种“低层优化”编程方式通常会导致内存优化难以进行，因为大量相似的代码片段出现在很多地方，但又无法共享这些代码，因为它们都有细微的差异。

John Bentley（以设计高效代码著称）提出了两条“优化法则”：

- 第一：不要做优化。
- 第二（仅对行家里手）：还是不要做优化。

在进行优化之前，必须确认你完全理解了系统。唯有这样，你才能确信优化是正确而又可靠的。在程序设计过程中，应该把精力集中在算法和数据结构上。当系统的早期版本可以运行起来后，再视需要仔细测试、调节系统。幸运的是，这种朴素的程序设计策略也可能带来惊喜：简洁的代码有时会足够快而且不会占用太多的内存。即便有这种可能，也不要抱太大期望，相反的情况也是很常见的，还是要进行仔细的测试。

## 25.2.3 生活在故障中

设想我们准备设计并实现一个不会失效的系统。这里“不会失效”的意思是“可以在没有人工干预的情况下正常工作一个月”。那么我们必须防御哪些类型的故障呢？我们当然可以排除太阳向新星演变的情况，系统被大象踩踏的情况应该也无须考虑。但是，一般来说我们很难估计会发生什么样的故障。对于一个特定系统，我们可以也应该假定哪些故障更容易发生，例如：

- 功率骤变 / 电源故障。
- 插头从插座上脱落。
- 系统被落下的碎片击中，处理器被损坏。
- 系统坠落（硬盘可能会因为冲击而损坏）。
- X 射线导致内存中某些位的值不按程序语言的定义而改变。

⚠ 瞬时故障通常是最难查找的，所谓瞬时故障（transient error），就是指在“某些时候”会发生，但不会在程序每次运行时都发生的故障。例如，我们听说过处理器只有在温度超过 130 °F（54 °C）时才会行为异常，正常情况下是不会达到这么高的温度的。但是，如果系统（偶然地、不小心地）堆积在工厂车间的角落里，被厚厚地覆盖着，散热不好的话，还是有

可能达到这个温度的，当然系统在实验室中进行测试时不会达到这个温度。

在实验室之外发生的故障是很难修复的。你很难想象，为了让 JPL 的工程师能够监测火星巡游者号上的软件和硬件故障（仅仅信号传输就需要 20 分钟），并能在弄清问题后通过软件更新方式来修复故障，在设计和实现系统时会花费多么大的努力。

为了设计和实现一个具有容错能力的系统，领域知识（即关于系统本身、它的工作环境及其使用方式的知识）是必需的。在本章中，我们只能涉及一些一般原则。注意，这里讨论的每条“一般原则”都是一个庞大的主题，都有几十年研究和开发历史，相关的文献都数以千计。



- **避免资源泄漏 (resource leak)**：绝不能发生泄漏。要明确程序使用哪些资源，要确保你（完全）拥有这些资源。任何泄漏最终都会令系统或者子系统崩溃。最重要的资源是 CPU 时间和内存。通常，程序还会使用其他资源，如锁、通信信道、文件等等。
- **副本 (replicate)**：如果某个硬件资源（如一台计算机、一个输出设备、一个轮子等等）的正常运转对系统至关重要，那么设计者就面临这样一个基本的选择：是否应该为关键资源配置副本？对于硬件故障，我们要么简单地承受故障，要么配置热备设备，在故障时通过软件切换到热备设备。例如，船用柴油机燃油喷射器的控制器有三重副本，各副本之间通过一个双重备份的网络链接。注意，“热备”设备不需要与原设备完全一样（例如，可能航天探测器的主天线接收能力很强，而备份天线较弱）。而且，在系统无故障时，“热备”设备通常也可以投入工作，以提升系统性能。
- **自检 (self-check)**：要了解掌握程序（或硬件）什么时候出现故障。硬件设备（如存储设备）通常都能监测自身的运行状况，对小故障进行修复，将无法处理的严重故障报告给用户，这对于我们进行故障检测非常有帮助。软件则可以检查数据结构的一致性，检查不变量（见 9.4.3 节），以及依赖内部的“完整性检查”（断言）进行故障检测。不幸的是，自检机制本身也有可能是不可靠的，报告错误的过程本身可能导致一个新的错误，对这种情况必须加以小心——对错误检测模块本身的完全彻底的检测是非常困难的。
- **能够迅速排除有错误的代码**：解决的策略就是系统的模块化。每个模块都完成一项特定的工作，在此之上完成基于模块的错误处理。如果一个模块无法完成自己的工作，它可以将这一情况报告给其他模块。保持模块内的错误处理尽量简单（这样，故障恢复的可能性就更高，修复效率也更高），由其他模块负责更严重的错误。一个高可靠的系统一定是模块化的和层次化的。在每个层次中，严重错误都报告给下一层次来处理。最终的层次可能是由操作人员来处理。一个模块收到一个严重错误（另一个模块无法自己处理的错误）的通知后，可以采取适当的措施，可以重启错误模块，或者启动一个更简单（但也更可靠）的“备份”模块。对于一个给定系统，准确定义什么是“模块”，应该是系统整体设计的一部分，但你可以把模块看作一个类、一个库、一个程序或者一台计算机上的所有程序。
- **对子系统进行监控**——如果子系统自身不能或没有监测自身故障的话。在一个多层系统中，上层模块可以监控下层模块。很多不允许失效的系统（如船用引擎或者空间站的控制器的控制器）对关键子系统都配置三重备份。这种三重备份不仅仅是为了设置两个热备设备，还有一个很重要的目的：在设备行为不一致时，通过投票，采用少数服从多数的策略（一个服从两个）来确定正确的结果。在多层结构很难实施的地方（即，系统的最高层，或者不允许失效的子系统），三重备份就显得非常有用。

我们可以设计更多这样的原则，并在实现中小心保证，但系统仍然会出现不可预知的问题。因此，在交付用户使用之前，还是需要进行系统的、全面的测试，参见第 26 章。

## 25.3 内存管理

计算机中两种最重要的资源是时间（执行指令）和空间（保存数据和程序的内存）。在 C++ 中，有三种分配内存的方法（见 12.4 节和附录 A.4.2）：

- 静态内存 (static memory)：是由链接器分配的，其生命期为整个程序的运行期间。
- 栈内存 (stack memory，也称为自动内存)：在调用函数时分配，当函数返回时释放。
- 动态内存 (dynamic memory，也称为堆 (heap))：用 `new` 操作分配，`delete` 操作释放。

下面，我们从嵌入式程序设计的角度来考察这几种内存分配方式。特别地，我们将把可预测性（见 25.2.1 节）作为必备的性质考虑在内，也就是说，我们针对的是硬实时系统程序设计和安全系统程序设计。

在嵌入式系统程序设计中，静态内存分配不会引起任何特殊的问题：因为所有的内存分配工作都在程序开始运行之前就已经完成了，也远在系统部署之前。

⚠ 栈内存如果分配过多，就可能导致一些问题，但这并不难处理。系统设计者须确保没有任何程序在执行时会使栈溢出，这通常意味着函数调用的最深层次不能超过限制，即，我们必须保证调用链（如，`f1` 调用 `f2`，`f2` 调用 `f3`，…，调用 `fn`）不会太长。在某些系统中，可能就需要禁止使用递归函数了。这对某些系统和某些递归函数是合理的，但并不是对所有系统和递归函数都如此。例如，我们知道 `factorial(10)` 最多产生对 `factorial` 的 10 层调用，因此可以很容易确保栈不会溢出。但是，嵌入式系统程序员可能更愿意使用循环来实现 `factorial`（见 20.5 节），以避免任何疑问或意外。

在嵌入式程序中，动态内存分配通常是被禁止或者受到严格限制的。即，`new` 或者被禁止，或者只在启动时使用，而 `delete` 则被严格禁止。基本的原因是：

- 可预测性：动态内存分配是不可预测的，也就是说，它不能保证在固定时间内完成。实际上，不能按时完成的情况还不是少数，因为许多 `new` 的实现都有这样一个特点：在已经分配和释放了很多对象后，再分配新的对象，所花费的时间会呈上升趋势。
- 碎片 (fragmentation)：动态内存分配会造成碎片问题，即，在分配和释放了大量对象后，剩余的内存会“碎片化”——空闲内存被分割成大量小“空洞”，每个空洞都很小，无法容纳程序所需对象，从而使这些空闲内存毫无用处。因此，可用空闲内存量远远小于初始内存总量减去已分配的内存量。

下一节会解释为什么会出这种不可接受的情况。重要的是在硬实时程序设计和安全程序设计中，我们必须避免使用 `new` 和 `delete`。下面几节会介绍一些方法，可以使用栈和存储池技术系统地避免动态内存分配带来的问题。

### 25.3.1 动态内存分配存在的问题

`new` 的问题究竟在哪里呢？实际上问题是出在 `new` 和 `delete` 的结合使用上。考察下面程序中内存分配和释放的过程：

```
Message* get_input(Device&); // 在自由空间创建一个 Message
while(* ... *) {
```

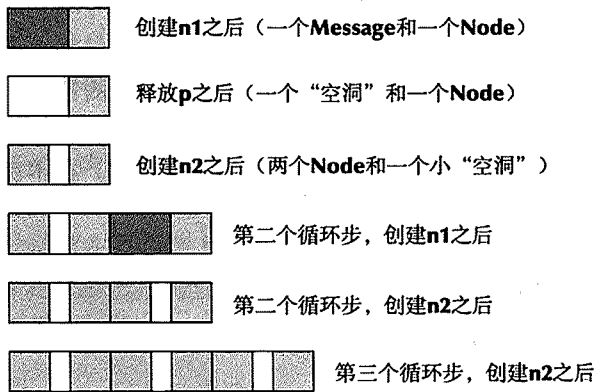
```

Message* p = get_input(dev);
//...
Node* n1 = new Node(arg1,arg2);
//...
delete p;
Node* n2 = new Node (arg3,arg4);
//...
}

```

在每个循环步中，我们创建了两个 Node，在此期间，我们还分配了一个 Message，然后又释放了它。当我们需要用从某个“设备”而来的输入创建一个数据结构时，常常会使用这样的代码。考察这段代码，每执行一个循环步，我们可能期望“消耗” $2 * \text{sizeof}(\text{Node})$ 个字节的内存（再加上动态内存分配的额外开销）。但不幸的是，真正的内存“消耗”并不一定如我们所愿。实际上，每个循环步总是会消耗掉更多的内存。

我们假定系统使用一个简单（但并非与实际不符）的内存管理程序。另外假定一个 Message 比一个 Node 稍大。下图展示了动态内存的使用情况，其中 Message 用深灰表示，Node 用浅灰表示，而白色表示“空洞”（即“未使用空间”）：

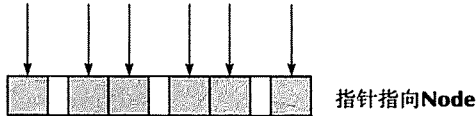


可以看到，每执行一个循环步，我们就会在动态内存中留下一些未用空间（“空洞”）。这些空洞可能只有几个字节大小，但如果我们不能加以有效利用，其危害与内存泄漏是一样的——即使是微小的泄漏，在长时间运行后也会导致系统崩溃。在内存中，空闲空间分散，形成很多小“空洞”，无法满足新的内存需求的情况，就称为内存碎片（memory fragmentation）。内存管理程序最终会把足够大的“空洞”用尽，只留下无法使用的小空洞。这是任何频繁使用 new 和 delete 的系统长期运行后都会遇到的一个严重问题，最终，内存中布满了无法使用的碎片。此时再执行 new 操作，就需要在大量对象和碎片中搜索足够大的区域，所花费的时间就会急剧增加。显然，这对于嵌入式系统来说是不可接受的。对于非嵌入式系统，这也是一个严重问题。

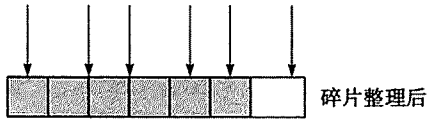
为什么不让“语言”或“系统”来处理这个问题呢？或者，我们为什么不能编写不会形成“空洞”的程序呢？我们首先看一下消除“空洞”的最直接的方法：移动 Node，将空闲空间压缩成一片连续的区域，这样就可以用来存储新的对象了。

不幸的是，“系统”无法完成这样的任务。原因在于 C++ 是直接内存地址来访问对象的。例如，指针 n1 和 n2 中都是对象的实际内存地址。因此，如果我们移动了对象，这些指针就不再指向正确的对象，其中的地址就变为无效了。下图给出了指针保存对象地址的示意：





现在，我们如果移动对象，整理碎片，就会出现下面的情况：



不幸的是，由于移动对象时没有相应地修改指针，现在的指针已经乱七八糟了。那么我们为什么不在移动对象的同时修改指针呢？我们可以编写一个程序来完成这个工作，但是前提是必须知道数据结构的细节。一般情况下，“系统”（C++ 运行时支持系统）是无法知道指针在哪里的，也就是说，给定一个对象，无法回答“程序中的哪些指针现在指向这个对象？”。即使可以回答这个问题，这种方法（称为压缩垃圾收集，compacting garbage collection）通常也不是最好的方法。例如，为了完成收集任务，除了程序所使用的内存外，还需要两倍于此的空间来跟踪指针以及移动对象。在嵌入式系统中，是不会有这么多额外内存空间的。另外，一个高效的垃圾内存收集程序很难达到可预测性。

我们自己当然可以回答“指针在哪？”的问题，因此可以自己编写程序来进行空间压缩。这是可行的，但更简单的方法是从根本上避免碎片的出现。在本例中，我们可以在分配 Message 之前为两个 Node 分配空间：

```
while(...){
    Node* n1 = new Node;
    Node* n2 = new Node;
    Message* p = get_input(dev);
    // 在节点中保存数据
    delete p;
    // ...
}
```

但是，用重整代码的方法来避免碎片问题通常比较困难。既避免碎片，同时又要保证代码仍旧可靠，最乐观地估计，也是一项非常困难的工作。而且，代码的重整可能与其他编码基本原则冲突。因此，我们倾向于限制动态内存分配的使用，这样就从根本上消除了碎片问题。通常，预防比治疗更有效。

### 试一试

将上面的程序补充完整，输出创建的对象地址和大小，观察是否会出现“空洞”，“空洞”又是如何分布的。如果有时间的话，试着画一下内存布局（就像上面那几个图一样），这能帮你更好地了解这个问题。

## 25.3.2 动态内存分配的替代方法

我们已经决定从根本上避免碎片，但如何做到呢？首先，一个简单的事实是：单独使用 new 操作不会导致碎片，用 delete 操作释放内存时才会产生空洞。因此，我们第一步先禁止

delete。这意味着，一旦为对象分配了内存空间，那么其生命周期就会持续到程序结束。

如果不再使用 delete 了，new 就是可预测的了吗？也就是说，所有 new 操作都会花费相同的时间了吗？对于一般的实现，确实是这样，但并不是所有系统都保证如此。通常，嵌入式系统中都有一段初始化代码，在加电或重启后完成初始化任务。在初始化期间，我们可以任意分配内存空间，只要不超过限额即可。分配方式可以使用 new，也可以使用全局（静态）内存。从程序结构的角度来说，应该尽量避免使用全局数据，但全局内存分配方式可以实现内存空间的预分配。这方面更准确的规则应作为系统编程规范的一部分（见 25.6 节）。

有两种数据结构在实现可预测内存分配时非常有用：

- 栈：在栈中，你可以分配任意大小的内存空间（分配的总量不超过预设的最大值），最后分配的空间总是最先被释放。也就是说，栈只在栈顶一端增长和缩小。因而也就不存在碎片问题，因为内存空间的分配和释放是不会交叉的。
- 存储池（pool）：所谓存储池，就是一组相同大小的对象的集合。只要需分配的对象数未超过存储池的容量，就可以在其中任意分配和释放对象。由于所有对象都是相同大小，因此也不会产生碎片。

采用栈和存储池，分配和释放都是可预测的，速度也很快。

这样，对于硬实时系统或者关键系统，我们可以视需要自己定义栈和存储池。但最好能使用现成的、已经过测试的栈和存储池代码（只要其定义符合我们的需要就可以使用）。

注意，我们不能使用 C++ 标准库中的容器（vector、map 等）和 string，因为它们间接使用了 new。你可以创建（购买或借用）可预测的“类标准”容器，当然这些代码的用途不仅仅局限于嵌入式系统。

注意，嵌入式系统通常在可靠性上要求非常严格，因此，无论选择怎样的解决方案，我们都不能倒退到直接使用大量低层特性的程序设计风格。充斥着指针、显式类型转换等特性的代码，是极难保证正确性的。

### 25.3.3 存储池实例

存储池是这样一种数据结构，我们可从中分配指定类型的对象，随后可将这些对象释放。下图说明了存储池的工作原理，其中深灰表示“已分配的对象”，而浅灰表示“空闲空间”：



我们可以定义 Pool 如下：

```
template<typename T, int N>
class Pool { // N 个 T 类型对象的存储池
public:
    Pool(); // 创建 N 个 T 的存储池
    T* get(); // 从存储池获取一个 T；若无空闲 T 返回 0
    void free(T*); // 将 get() 获得的一个 T 归还存储池
    int available() const; // 空闲 T 的数目
private:
    // T[N] 所需空间以及跟踪哪些 T 被分配、哪些未分配所需的数据（如一个空闲对象链表）
};
```

每个 Pool 对象都包含类型相同的一组对象，对象数目有上限值。Pool 的使用方法如下

面代码所示：

```
Pool<Small_buffer,10> sb_pool;
Pool<Status_indicator,200> indicator_pool;

Small_buffer* p = sb_pool.get();
// ...
sb_pool.free(p);
```

程序员应该确存储池不会被耗尽，“确保”的准确含义依赖于具体应用。对于某些系统，程序员应该保证只有在 pool 有空闲空间的情况下才调用 get()。在另外一些系统中，程序员可以检测 get() 的返回值，在返回值为 0 的情况下进行一些补救措施。第二种策略的一个典型例子是电话系统，假定它最多同时处理 100 000 个呼叫。每个呼叫都需要一些资源，比如一个拨号缓冲区。如果系统中的拨号缓冲区都已耗尽（即，dial\_buffer\_pool.get() 返回 0），系统可以拒绝建立新的通话连接（还可能“杀掉”一些已有的通话连接来释放一些空间）。拨打电话的人则可以稍后再拨。

当然，我们的 Pool 模板只是存储池一般思想的一种实现，还可以根据需要进行选择其他实现方式。例如，对于内存分配限制不那么苛刻的系统，我们可以修改存储池的定义，在构造函数中指定元素数目，甚至在需要时改变元素数目。

### 25.3.4 栈实例

栈是这样一种数据结构，我们可以从中分配内存空间，而最后分配的区域被最先释放。下图说明了栈的工作方式，其中深灰表示“已分配内存”，浅灰表示“空闲空间”：



如图所示，栈向右“生长”。

定一个对象栈，与定义一个对象存储池类似：

```
template<typename T, int N>
class Stack {
    // N 个 T 类型对象的栈
    // ...
};
```

然而，大多数系统都需要为不同大小的对象分配内存。存储池无法满足这种需求，但栈却可以。下面我们就展示如何定义一个能从中分配大小不同的“原始”内存空间，而非固定大小对象的栈。

```
template<int N>
class Stack {
    // N 个字节的栈
public:
    Stack(); // 创建一个 N 个字节的栈
    void* get(int n); // 从栈中分配 n 个字节
    // 若无空闲空间返回 0
    void free(); // 将 get() 返回的最后一个值归还给栈
    int available() const; // 可用字节数
private:
    // char[N] 所需空间以及跟踪哪些已分配、哪些未分配所需数据（如一个栈顶指针）
};
```

get() 从栈中分配指定大小的内存空间，返回指向起始地址的 void\* 指针，因此，我们需要显式将其转换为所需的类型。这种栈的使用方式如下：


```
Stack<50*1024> my_free_store; // 50KB 空间用于构建一个栈

void* pv1 = my_free_store.get(1024);
int* buffer = static_cast<int*>(pv1);
void* pv2 = my_free_store.get(sizeof(Connection));
Connection* pconn = new(pv2) Connection(incoming,outgoing,buffer);
```

static\_cast 的使用已经在 12.8 节中介绍过了。语法 new(pv2) 表示“定址 new”，即“在 pv2 指向的内存空间中创建一个对象”。也就是说，它并不分配新的内存空间。这段代码假定 Connection 有一个构造函数，接受参数 (incoming, outgoing, buffer)。如果没有定义这样的构造函数，编译会失败。

同样，Stack 模板也只是栈的一般思想的一种实现而已，还可以有其他的实现方式。例如，如果内存分配的限制不那么苛刻，我们可以修改栈的定义，实现在构造函数中指定预分配的空间大小。

## 25.4 地址、指针和数组

可预测性只是某些嵌入式系统的需求，而可靠性则是所有嵌入式系统都需要的。因此， 应该避免使用那些已被证明容易出错的语言特性和程序设计技术（这里是指在嵌入式系统中容易出错，在其他环境中并不一定）。指针就是这样一种语言特性，使用不慎很容易导致错误，有两个问题最为突出：

- (未经检查的和不安全的) 显式类型转换。
- 将指向数组元素的指针作为参数传递。

前一个问题通常可以简单地通过严格禁止使用显式类型转换来解决。指针 / 数组问题则更微妙，理解起来更有难度，解决方法可以使用 (简单的) 类或者标准库功能 (如 array, 参见 15.9 节)。因此，本节主要讨论如何解决指针 / 数组问题。

### 25.4.1 未经检查的类型转换

在低层系统中，物理资源 (如外部设备的控制寄存器) 及其基础软件通常位于特定的地址。我们不得不在程序中直接使用这些地址，并将它们转换为所需类型：

```
Device_driver* p = reinterpret_cast<Device_driver*>(0xffb8);
```

请参考 12.8 节。这种语法很不常用，你可能需要借助手册和联机帮助才不会写错。硬件资源 (资源的寄存器的地址——通常表示为十六进制整数) 和指向硬件资源控制软件的指针之间的对应关系是脆弱的。你需要保证其正确性，但又得不到编译器的帮助 (因为这本来就不是程序设计语言方面的问题)。通常，int 类型到指针类型的简单转换 (reinterpret\_cast)，是连接一个应用程序和它的重要硬件资源所必需的。但这样的转换是完全未经检查的，因此很容易出错。

只要显式类型转换 (reinterpret\_cast, static\_cast 等，见附录 A.5.7) 并非必需，就应该避免使用。通常，一些先前使用 C 或者 C 风格 C++ 的程序员喜欢使用这种类型转换，但实际上很多情况下是不必要的。

## 25.4.2 一个问题：不正常的接口

如上所述（13.6.1 节），一个数组常常作为参数，以指针的形式传递给函数（指针通常指向数组的第一个元素）。这样，数组大小就“丢失”了，从而导致接受参数的函数无法判断数组中共有多少个元素。这个问题是很多微妙而难以修正的 bug 的根源。下面，我们考察一些数组 / 指针问题的例子，并给出一个替代方法。我们以一个非常差的接口程序（但不幸，这个例子在实际程序中并不罕见）作为开始，然后尝试改进它：

```
void poor(Shape* p, int sz)           // 糟糕的接口设计
{
    for (int i = 0; i < sz; ++i) p[i].draw();
}

void f(Shape* q, vector<Circle>& s0) // 非常糟糕的代码
{
    Polygon s1[10];
    Shape s2[10];
    // 初始化
    Shape* p1 = new Rectangle(Point{0,0},Point{10,20});
    poor(&s0[0],s0.size());           // #1 (传递来自 vector 的数组)
    poor(s1,10);                     // #2
    poor(s2,20);                      // #3
    poor(p1,1);                       // #4
    delete p1;
    p1 = 0;
    poor(p1,1);                       // #5
    poor(q,max);                      // #6
}
```

⚠ 函数 `poor()` 是一个设计得很差的接口：它使调用者极易出错，又几乎没有给实现者预防错误的机会。

### 试一试

在继续阅读之前，尝试找出 `f()` 中的错误。特别是，对 `poor()` 的哪次调用会导致程序崩溃？

乍一看，这些对 `poor()` 的调用没有什么问题，但这些代码正是那种会花费程序员整夜时间来除错的程序，对高水平工程师来说也会是一场噩梦。

1. 元素类型传递错误，如 `poor(&s0[0], s0.size())`。而且 `s0` 还可能是空的，此时 `&s0[0]` 本身就是错的。
2. 使用了“魔数”：`poor(s1,10)`（此处是正确的）。这里，元素类型也是错误的。
3. 使用了错误的“魔数”：`poor(s2, 20)`。
4. 正确的调用：第一个 `poor(p1, 1)`。
5. 传递了一个空指针：第二个 `poor(p1, 1)`。
6. 可能是正确的：`poor(q, max)`。仅看这个代码片段，不能判断这个调用是否正确。为了判断 `q` 指向的数组是否包含至少 `max` 个元素，必须找到 `q` 和 `max` 的定义并获得程序运行到此处时它们的值。

上述这些错误都很简单，我们并未涉及微妙的算法或数据结构问题。所有问题都出在

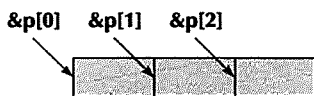
poor() 的接口上，它包含一个以指针方式传递的数组，这导致了一系列的问题。你可以体会一下，我们所使用的 p1 和 s0 这种无意义的名字是如何使问题更加模糊不清的。这些名字虽然有助记忆，但容易造成混淆，使得这些错误更难以查找。

理论上，编译器可以找到其中一些错误（如第二个 poor(p1,1) 中 p1==0 的情形）。但在现实中，我们之所以能免受这些错误的困扰，主要还是因为编译器发现了程序试图定义抽象类 Shape 的对象。但是，这并未解决 poor() 接口方面的问题，因此我们无法松口气。接下来，我们将使用一个非抽象的 Shape，这样就能专注于接口问题了。

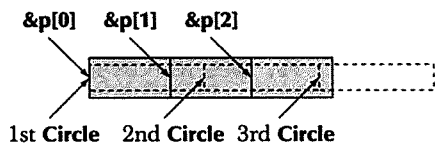
poor(&s0[0], s0.size()) 到底错在哪里呢？&s0[0] 指向一个 Circle 数组的首元素，因此它是一个 Circle\* 指针。poor 期待的是一个 Shape\* 指针，而我们传递给它的的是一个 Shape 派生类对象指针（Circle\*）。这显然是允许的：我们需要这种类型转换，因为面向对象程序设计中常常需要用同一段代码对源于同一基类（本例中是 Shape）的不同派生类的对象（见 19.2 节）进行处理。但是，poor() 不仅仅把 Shape\* 作为一个指针来使用，还将它作为数组使用，通过下标访问其元素：

```
for (int i = 0; i < sz; ++i) p[i].draw();
```

这段代码顺序访问内存地址 &p[0], &p[1], &p[2], ... 上的对象：



就内存地址而言，这些指针的间距为 sizeof(Shape)（见 12.3.1 节）。但不幸的是，对于此次 poor() 的调用，sizeof(Circle) 大于 sizeof(Shape)，因此内存布局如下所示：



也就是说，poor() 中调用 draw() 时，指针实际指向一个 Circle 对象的中间！这很可能立即导致程序崩溃。

poor(s1, 10) 的问题更为隐蔽。其中使用了“魔数”10，因此我们很容易立刻怀疑它是问题的根源，但实际上这条语句中还隐藏着一个更深层次的问题。我们使用 Polygon 数组作为 poor 的参数，不会遇到 Circle 数组所面临的那些问题，原因是 Polygon 没有在基类 Shape 的基础上增加数据成员（而 Circle 加入了新的数据成员，参见 18.8 和 18.12 节）。也就是说，sizeof(Shape)==sizeof(Polygon)，更一般地讲，Polygon 和 Shape 具有相同的内存布局。换句话说，我们真的很“幸运”，对 Polygon 定义的任何微小修改都会导致程序崩溃。因此，当前的 poor(s1,10) 可以正常工作，但它是一个 bug，迟早会引起程序错误。这条语句毫无疑问是低质量的代码。

上述这些问题都是程序设计法则“‘D 是 B’并不意味着‘D 的容器是 B 的容器’”在实际代码中的体现（见 14.3.3 节）。请看下例：

```
class Circle : public Shape { /* ... */};
```

```
void fv(vector<Shape>&);
```

```
void f(Shape &);
```

```

void g(vector<Circle>& vd, Circle & d)
{
    f(d);    // 正确: 从 Circle 到 Shape 的隐式转换
    fv(vd); // 错误: 不存在 vector<Circle> 到 vector<Shape> 的转换
}

```

好了，我们已经知道上述 `poor()` 的调用是非常糟糕的代码了，但这种代码会出现在嵌入式程序中吗？也就是说，在安全些和性能要求都很高的领域中，我们会遇到这类问题吗？我们是否可以简单地把这种代码作为错误的根源，告诉“普通程序”的程序员“不要在嵌入式程序中使用这种代码”呢？恐怕还不能这么简单地处理，因为很多现代的嵌入式系统都严重依赖 GUI，而 GUI 程序通常采用面向对象程序设计方法开发，其代码组织很像前面给出的例子。这方面的例子有很多，如 iPod 的用户界面、一些手机的用户界面以及“小设备”（包括飞机）上操作人员使用的显示界面等等。另外一个例子是很多相似设备（例如很多电动机）的控制器可以构成一个典型的类层次。换句话说，这种代码，特别是这种函数声明方式，确实会在嵌入式程序中出现，我们必须加以考虑。我们需要一种更安全的方法来传递一组数据，不会引起上述严重的问题。

因此，我们不希望数组参数以“指针 + 大小”的方式传递。那么有什么替代方法吗？最简单的方法是传递容器（如 `vector`）的引用，例如：

```
void poor(Shape* p, int sz);
```

就不存在先前的函数接口

```
void general(vector<Shape>&);
```

所存在的那些问题。如果在你的开发环境中，`std::vector`（或者等价的工具）是可用的，那么在函数接口中就一直使用它，而不要以指针加大小的方式传递内置数组。

如果你无法使用 `vector` 或等价的工具，就会陷入困境，虽然可以直接使用我们定义的接口类 `Array_ref`，但仍旧需要一些复杂的语言特性和技术来编写程序。

### 25.4.3 解决方案：接口类

不幸的是，在很多嵌入式系统中我们都不能使用 `std::vector`，因为它依赖动态内存分配。一种解决方法是实现一个特殊的 `vector`，更简单的方法是定义一个与 `vector` 功能相似但不使用动态内存分配的容器。在给出这个容器的定义之前，先思考一下我们希望这个容器具有什么功能：

- 它只是内存中对象的一个引用（它不拥有对象，也不分配、释放对象）。
- 它“知道”自己的大小（这样就有可能实现范围检查）。
- 它“知道”元素的确切类型（这样它就不会成为类型错误的根源）。
- 传递代价（拷贝）低，传递方式可以是一个（指针，数量）对。
- 它不能显式转换为一个指针。
- 通过接口对象，能容易地描述元素范围的子区域。
- 它和内置数组一样容易使用。

我们只是尽可能地接近“和内置数组一样容易使用”这一目标，实际上也不应完全“一样容易使用”，因为那样就意味着“一样容易引起错误”。

下面给出了接口类的一个定义：

```

template<typename T>
class Array_ref {
public:
    Array_ref(T* pp, int s) : p(pp), sz(s) {}

    T& operator[ ](int n) { return p[n]; }
    const T& operator[ ](int n) const { return p[n]; }

    bool assign(Array_ref a)
    {
        if (a.sz!=sz) return false;
        for (int i=0; i<sz; ++i) { p[i]=a.p[i]; }
        return true;
    }

    void reset(Array_ref a) { reset(a.p,a.sz); }
    void reset(T* pp, int s) { p=pp; sz=s; }

    int size() const { return sz; }

    // 默认拷贝操作:
    // Array_ref 不拥有任何资源
    // Array_ref 具有引用语义
private:
    T* p;
    int sz;
};

```

这个接口类 `Array_ref` 已经尽可能地简化了：

- 没有定义 `push_back()` (因为可能需要动态内存分配)，也没有定义 `at()` (可能需要使用异常机制)。
- `Array_ref` 本质是一种引用，因此复制操作只拷贝 `(p, sz)`，而不会拷贝引用的对象。
- 不同的 `Array_ref` 可以用不同的数组进行初始化，这样它们具有相同的类型，但大小不一样。
- 我们可以使用 `reset()` 来更新 `(p, sz)` 的值，这样就可以改变 `Array_ref` 的大小 (很多算法要求指定子范围)。
- 没有定义迭代器接口 (如果需要的话，加入迭代器功能很容易)。实际上，`Array_ref` 本质上很接近由两个迭代器形成的一个范围。

`Array_ref` 并不拥有元素，也不进行内存管理，它只不过是一种访问及传递元素序列的机制。在这一点上，它与标准库的 `array` (见 15.9 节) 是不同的。

为了简化 `Array_ref` 的初始化，我们设计了一些有用的辅助函数：

```

template<typename T> Array_ref<T> make_ref(T* pp, int s)
{
    return (pp) ? Array_ref<T>(pp,s) : Array_ref<T>({nullptr,0});
}

```

如果我们用一个指针来初始化 `Array_ref`，那么就必须显式提供数组的大小。这显然是 `Array_ref` 的一个弱点，因为调用者有可能提供错误的大小。而且，如果调用者传递来的指针是从一个派生类指针隐式转换为基类指针的，如，将 `Polygon[10]` 传递给 `Shape*`，那么我们在 25.4.2 中讨论的那个棘手的问题就又出现了。但是，只要保留这种初始化方式，这个问题就很难解决，我们有时只能相信程序员。



前一段代码中我们对空指针进行了检查（因为它通常是错误之源），我们同样也应小心空 vector：

```
template<typename T> Array_ref<T> make_ref(vector<T>& v)
{
    return (v.size()) ? Array_ref<T>(&v[0],v.size()) : Array_ref<T>(nullptr,0);
}
```

这段代码的思想是传递 vector 中的元素数组。虽然在很多 Array\_ref 的应用场合（嵌入式系统）中并不适宜使用 vector。不过，与适合在嵌入式系统中使用的容器（如，基于存储池的容器，参见 25.3.3 节）相比，vector 具有很多相似的特点。

最后的一个辅助函数利用内置数组（编译器知道其大小）来初始化 Array\_ref：

```
template<typename T, int s> Array_ref<T> make_ref(T (&pp)[s])
{
    return Array_ref<T>(pp,s);
}
```

T(&pp)[s] 的语法有些奇怪，它声明了一个引用类型参数 pp，pp 引用的是一个元素类型为 T、元素个数为 s 的数组。这样，就可以使用数组来初始化 Array\_ref 了（数组大小是已知的）。由于 C++ 不允许声明空数组，所以这里不必对此进行检测：

```
Polygon ar[0]; // 错误：无元素
```

有了 Array\_ref 后，我们就可以重写例程了：

```
void better(Array_ref<Shape> a)
{
    for (int i = 0; i<a.size(); ++i) a[i].draw();
}

void f(Shape* q, vector<Circle>& s0)
{
    Polygon s1[10];
    Shape s2[20];
    // 初始化
    Shape* p1 = new Rectangle(Point{0,0},Point{10,20});
    better(make_ref(s0)); // 错误：需要 Array_ref<Shape>
    better(make_ref(s1)); // 错误：需要 Array_ref<Shape>
    better(make_ref(s2)); // 正确（无须转换）
    better(make_ref(p1,1)); // 正确：一个元素
    delete p1;
    p1 = 0;
    better(make_ref(p1,1)); // 正确：无元素
    better(make_ref(q,max)); // 正确（若 max 合法）
}
```

新的程序有如下改进：

- 代码更简洁。程序员大多数情况下无须考虑大小，即便某些时候需要考虑，也仅仅局限于 Array\_ref 初始化的部分，而不会出现在代码其他位置。
- 解决了 Circle[] 转换为 Shape[]、Polygon[] 转换为 Shape[] 所存在的类型问题。
- 隐含地解决了 s1、s2 所存在的错误的元素数目问题。
- max 的潜在问题（以及其他的指针指向的元素数目问题）变得更为明显了，这里是我们唯一需要显式处理大小的地方。
- 我们系统地、隐式地解决了空指针和空 vector 问题。

### 25.4.4 继承和容器

但是，如果我们需要将 `Circle` 对象序列作为 `Shape` 对象序列来处理，也就是说，我们确实需要 `better()`（实际上是 `draw_all()` 的变形，参见 14.3.2 节和 22.1.3 节）来处理多态，又该怎么办呢？基本上，这是办不到的。在 14.3.3 节和 25.4.2 节中，我们已经看到，类型系统有很充分的理由拒绝将 `vector<Circle>` 作为 `vector<Shape>` 来处理。基于同样理由，`Array_ref<Circle>` 也不能作为 `Array_ref<Shape>`。如果你忘记了这部分内容，最好重新阅读 14.3.3 节，因为这是一个非常基础的程序设计原则，虽然它有些不方便。

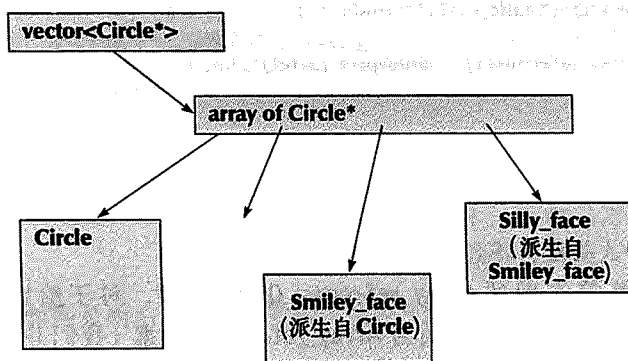
而且，为了防止运行时的多态行为，我们必须通过指针（或者引用）来访问多态对象：`better()` 中是不该使用 `afil.draw()` 的。当我们一看到对多态对象使用点操作符而不是箭头 (`->`) 时，就该想到可能要出问题了。

那么我们应该怎么做呢？首先，必须使用指针（或引用）来访问对象，因此，在例子程序中应该使用 `Array_ref<Circle*>`、`Array_ref<Shape*>` 等等，而不是 `Array_ref<Circle>`、`Array_ref<Shape>`。

但是，我们又不能将 `Array_ref<Circle*>` 转换为 `Array_ref<Shape*>`，否则接下来的代码就可能将一些不是 `Circle*` 的元素放入 `Array_ref<Shape*>` 中。不过，我们可以钻个空子：

- 在这个例子中，我们并不想修改 `Array_ref<Shape*>`，而只是想将形状画出来！这是一个有趣而且有用的特例：由于我们不修改 `Array_ref<Shape*>`，上述不该将 `Array_ref<Circle*>` 转换为 `Array_ref<Shape*>` 的理由也就不成立了。
- 所有指针数组都具有相同的内存布局（不管指针指向的是什么类型的对象），因此我们不会陷入 25.4.2 节所述的布局问题。

也就是说，将 `Array_ref<Circle*>` 作为不可变的（immutable）`Array_ref<Shape*>` 来处理，不存在任何问题。接下来，我们只要找到这样一种转换方法就可以了，请看下图：



将这样一个 `Circle*` 数组作为一个不可变的 `Shape*` 来处理（利用 `Array_ref`），在逻辑上是没有任何问题的。

看起来我们已经闯入专家领域了。事实上，这个问题确实非常棘手，用现有的工具是很难解决的。但是，我们还是先来看一下如何为这个有问题但又很常见的接口模式（指针问题和元素数量问题，参见 25.4.2 节）找到一种接近完美的解决方案吧。请记住：不要为了显示自己的聪明而进入“专家领域”。通常来说，最好的开发策略是从库中找到专家们已经设计实现好并经过测试的工具，直接使用它们。

首先，我们重写 `better()`，对多态对象的访问全部改用指针，以确保我们不会“弄乱”给定的容器：

```
void better2(const Array_ref<Shape*const> a)
{
    for (int i = 0; i < a.size(); ++i)
        if (a[i])
            a[i]->draw();
}
```

由于改用了指针，所以必须检测指针是否为空。为了确保 `better2()` 不会通过 `Array_ref` 修改数组或向量的内容，我们在两处使用了 `const`。第一个 `const` 保证我们不会对 `Array_ref` 使用修改（更新）操作，如 `assign()` 和 `reset()`。第二个 `const` 放在 `*` 之后，表示这是一个常量指针（不是指向常量内容的指针），即，我们不希望修改指针本身（`Array_ref` 的元素）。

接下来，我们需要解决核心问题：如何表达如下意图？

- `Array_ref<Circle*>` 可以转换为类似 `Array_ref<Shape*>` 的东西（这样就能在 `better2()` 中使用）。
- 但是只能转换为不可变的 `Array_ref<Shape*>`。

我们可以通过定义一个转换运算符来实现上述目标：

```
template<typename T>
class Array_ref {
public:
    // 与以往相同

    template<typename Q>
    operator const Array_ref<const Q>()
    {
        // 检查元素的隐式转换
        static_cast<Q>(*static_cast<T*>(nullptr)); // 检查元素
                                                    // 转换
        return Array_ref<const Q>{reinterpret_cast<Q*>(p),sz}; // 转换
                                                                    // Array_ref
    }

    // 与以往相同
};
```

这段代码有点令人头疼，不过基本要点如下：

- 类型转换运算符实现到 `Array_ref<const Q>` 的转换，对于给定的类型 `Q`，它先将 `Array_ref<T>` 的一个元素转换为 `Array_ref<Q>` 的元素（我们并不使用转换的结果，只是检验一下转换是否可行）。
- 接下来，转换运算符使用强制类型转换（`reinterpret_cast`）获得一个指定元素类型的指针，来构造新的 `Array_ref<const Q>`。强制转换通常会有额外开销，因此，不要对多重继承的类进行 `Array_ref` 类型转换（见附录 A.12.4）。
- 请注意 `Array_ref<const Q>` 中的 `const`，它的作用就是保证不会将 `Array_ref<const Q>` 复制到老版本的可变的 `Array_ref<Q>` 中。

我们已经警告过你，你已经进入了“令人头疼”的“专家领域”。不过，这个版本的 `Array_ref` 还是比较容易使用的（令人头疼的只是定义和实现，而非应用）：

```

void f(Shape* q, vector<Circle*>& s0)
{
    Polygon* s1[10];
    Shape* s2[20];
    // 初始化
    Shape* p1 = new Rectangle(Point{0,0},10);
    better2(make_ref(s0)); // 正确: 转换为 Array_ref<Shape*const>
    better2(make_ref(s1)); // 正确: 转换为 Array_ref<Shape*const>
    better2(make_ref(s2)); // 正确 (无须转换)
    better2(make_ref(p1,1)); // 错误
    better2(make_ref(q,max)); // 错误
}

```

最后两条语句对指针的使用是错误的，因为两个指针是 `Shape*` 类型，而 `better2()` 需要的是一个 `Array_ref<Shape*>` 类型的参数。也就是说，`better2()` 需要的是包含指针的容器，而非指针本身。如果我们希望将指针传递给 `better2()`，就必须将指针置于容器中（如内置数组或者 `vector`）传递。对于一个单独的指针，我们可以使用 `make_ref(&p1,1)`，虽然看起来有些笨拙，但能够达到目的。但是，对于数组（包含多于一个元素），如果不创建指向元素的指针，再置于一个容器中，是没有办法处理的。

总结一下，为了弥补数组的弱点，我们可以建立简单、安全、易用和高效的接口。这也是这一节的目的。“每一个问题都会有一个迂回的解决办法”（David Wheeler），这也是常说的“计算机科学第一定律”。我们正式采用这一思想解决接口问题。

## 25.5 位、字节和字

在本书前面的章节中，已经讨论过内存硬件层次的一些概念，如位、字节和字。但在普通程序设计中，我们不会过多考虑这些概念，我们思考问题的方式是将数据看作特定类型的对象，如 `double`、`string`、`Matrix` 以及 `Sample_Window`。在嵌入式程序设计中，我们必须对内存的低层组织方式有更多的了解，在本节中，我们会对此进行讨论。

如果你对整数的二进制和十六进制表示的相关知识不太了解，请参考附录 A.2.1.1。

### 25.5.1 位和位运算

一个字节可以看作 8 个位的序列：

7:	6:	5:	4:	3:	2:	1:	0:
1	0	1	0	0	1	1	1

注意，位编号的习惯顺序是由右（最低有效位）至左（最高有效位）。类似地，一个字也可看作 4 个字节的序列：

3:	2:	1:	0:
0xff	0x10	0xde	0xad

编号顺序同样是由右至左，即从最低有效位到最高有效位。这两个图过分简化了现实世界中的情况：曾经存在一个字节有 9 位的计算机（虽然没有一台的寿命超过十年），而一个字包含两个字节的计算机就更常见了。不过，只要你记得在使用“8 位”和“4 字节”这两个特性之前查阅一下系统手册，就不会出现问题了。

如果希望程序是可移植的，那么请在程序中使用 `<limits>`（见 24.2.1 节）以确保类型大

小不会弄错。为了方便编译器的检查，可以在程序中加入断言：

```
static_assert(4<=sizeof(int),"ints are too small");
static_assert(!numeric_limits<char>::is_signed,"char is signed");
```

一个 `static_assert` 的第一个参数是一个常量表达式，它应该为真。如果它不为真，即断言失败，编译器会将第二个参数（一个字符串）作为错误消息的一部分输出。

✘ 在 C++ 中我们如何来表示一组二进制位呢？答案取决于我们要处理多少位，以及希望哪些操作更方便和高效。我们可以将整型值当作一组二进制位来使用：

- `bool`——1 位，但占用整个字节的空间。
- `char`——8 位。
- `short`——16 位。
- `int`——通常是 32 位，但在很多嵌入式系统中是 16 位。
- `long int`——32 位或 64 位（至少与 `int` 一样）。
- `long long int`——32 位或 64 位（至少与 `long` 一样）。

上面列出的都是典型的类型大小，但在不同的实现中可能有所不同。因此，最稳妥的方法是实际测试一下。另外，标准库中也提供了处理位的方法：

- `std::vector<bool>`——当我们需要超过  $8 * \text{sizeof}(\text{long})$  个二进制位时使用。
- `std::bitset`——当需要超过  $8 * \text{sizeof}(\text{long})$  个位时使用。
- `std::set`——无序的、命名的二进制位集合（见 16.6.5 节）。
- 文件：海量的二进制位（见 25.5.6 节）。

而且，我们还可以使用如下两个语言特性来表示二进制位：

- 枚举 (`enum`)，参见 9.5 节。
- 位域，参见 25.5.5 节。

这么多表示“位”的方法，从一个侧面反映了：在计算机内存中，实际上任何数据最终都表示为一组二进制位，因此人们迫切地需要提供很多方法来查看位、命名位以及完成位运算。注意，所有内置语言特性都是处理固定数量的二进制位（如 8、16、32 和 64），因此可以直接使用硬件提供的指令以最佳性能进行运算。与之相对，标准库特性都能处理任意数量的位。这可能会影响性能，但不要忙着下结论：如果你能将一组二进制位很好地映射到下层硬件，这些库特性通常都有很好的性能。

我们先来考察用整数表示二进制位的方式。C++ 提供了硬件直接支持的位运算，这些运算都是对运算对象逐位进行操作：

位运算		
	或	如果 x 的第 n 位为 1 或 y 的第 n 位为 1，则 x y 的第 n 位为 1
&	与	如果 x 的第 n 位为 1 且 y 的第 n 位为 1，则 x&y 的第 n 位为 1
^	异或	如果 x 的第 n 位为 1 或 y 的第 n 位为 1 且不同时为 1，则 x^y 的第 n 位为 1
<<	左移位	x<<s 的第 n 位是 x 的第 n+s 位
>>	右移位	x>>s 的第 n 位是 x 的第 n-s 位
~	补	~x 的第 n 位是 x 的第 n 位的反

你可能觉得将“异或”（`^`，有时被称为“xor”）作为一个基本运算有些奇怪，但在很多图形和加密程序中，异或是一个基本运算。

编译器不会把移位运算符 << 误认为是一个输出操作符，但人有可能犯这样的错误。为了避免混淆，请记住输出操作符的左操作对象是一个 `ostream` 流，而移位运算符的左运算对象是一个整数。

注意，& 与 && 是不同的，| 与 || 也是不同的，& 和 | 会对运算对象的每一位独立进行计算（见附录 A.5.5），计算结果的位数与运算对象相同。与之相反，&& 和 || 只是返回 true 或者 false。

我们来尝试一些例子。我们常常用十六进制表示位的模式，下表列出了半字节值（4 位）的十六进制和二进制表示：

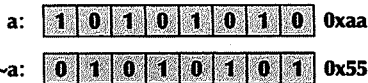
十六进制	位模式	十六进制	位模式
0x0	0000	0x8	1000
0x1	0001	0x9	1001
0x2	0010	0xa	1010
0x3	0011	0xb	1011
0x4	0100	0xc	1100
0x5	0101	0xd	1101
0x6	0110	0xe	1110
0x7	0111	0xf	1111

当数值小于 9 时，我们可以使用十进制，但使用十六进制可以提醒我们现在是在思考位模式。对于字节和字，十六进制非常有用。一个字节中的二进制位，可以表示为两个十六进制数字，例如：

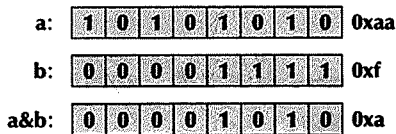
十六进制字节	位模式
0x00	0000 0000
0x0f	0000 1111
0xf0	1111 0000
0xff	1111 1111
0xaa	1010 1010
0x55	0101 0101

在进行位运算时，使用 `unsigned`（见 25.5.3 节）可以令情况更简单，避免一些不必要的问题。例如：

```
unsigned char a = 0xaa;
unsigned char x0 = ~a;    // a 的补集
```



```
unsigned char b = 0x0f;
unsigned char x1 = a&b;  // a 与 b
```



```
unsigned char x2 = a^b; // a 异或 b
```

```
a:  1 0 1 0 1 0 1 0  0xaa
b:  0 0 0 0 1 1 1 1  0xf
a^b: 1 0 1 0 0 1 0 1 0xaa5
```

```
unsigned char x3 = a<<1; // 左移 1 位
```

```
a:  1 0 1 0 1 0 1 0  0xaa
a<<1: 0 1 0 1 0 1 0 0  0x54
```

注意，最低位（第 0 位）填入了一个 0，可以看作是从第 7 位左边“移来”的。而原来的最高位（第 7 位）被简单丢弃。

```
unsigned char x4 == a>>2; // 右移 2 位
```

```
a:  1 0 1 0 1 0 1 0  0xaa
a>>2: 0 0 1 0 1 0 1 0  0x2a
```

注意，最高两位（第 6 位和第 7 位）都填入了 0，可以看作是从第 0 位右边“移来”的，最低两位（第 1 位和第 0 位）被简单丢弃。

在处理位运算时，就可以像这样画出位模式，这样的图示能使我们对比模式有一个很好的直观感觉。不过，对于更复杂的例子，手工画出位模式就太烦琐了。下面的这个小程序能将整数转换为二进制位描述形式：

```
int main()
{
    for (int i; cin>>i; )
        cout << dec << i << "=="
            << hex << "0x" << i << "=="
            << bitset<8*sizeof(int)>(i) << '\n';
}
```

其中使用了标准库中的 `bitset` 来打印整数的某个位：

```
bitset<8*sizeof(int)>(i)
```

一个 `bitset` 是一组固定数量的二进制位。在本例中，我们使用一个整数中所能容纳的二进制位数，也就是 `8*sizeof(int)`，并用整数 `i` 来初始化 `bitset`。

### 试一试

编译、运行这个例子程序，试着输入一些整数，体会二进制和十六进制表示形式。如果你对负数的表示形式感到迷惑，请在阅读 25.5.3 节后再重试。

## 25.5.2 bitset

标准库模板类 `bitset` 是在 `<bitset>` 中定义的，它用于描述和处理二进制位集合。每个 `bitset` 的大小是固定的，在创建时指定：

```
bitset<4> flags;
bitset<128> dword_bits;
bitset<12345> lots;
```

默认情况下，bitset 被初始化为全 0，但通常我们都会给它一个初始值，可以是一个无符号的整数或者“0”和“1”组成的字符串。例如：

```
bitset<4> flags = 0xb;
bitset<128> dword_bits {string{"10101010101010"}};
bitset<12345> lots;
```

这两段代码中，lots 被初始化为全 0，dword\_bits 的前 112 位被初始化为全 0，后 16 位由程序显式指定。如果你给出的初始化字符串中包含 0 和 1 之外的符号，bitset 会抛出一个 `std::invalid_argument` 异常：

```
string s;
cin>>s;
bitset<12345> my_bits(s); // 可能抛出 std::invalid_argument
```

常用的位运算符都可用于 bitset。例如，假定 b1、b2 和 b3 都是 bitset：

```
b1 = b2&b3;           // 与
b1 = b2|b3;          // 或
b1 = b2^b3;          // 异或
b1 = ~b2;             // 补
b1 = b2<<2;           // 左移
b1 = b2>>3;           // 右移
```

大致来说，对于位运算而言，bitset 就像 unsigned int（见 25.5.3 节）一样，只不过其大小任意，由用户指定。你能对 unsigned int 做什么（除了算术运算之外），就能对 bitset 做什么。特别地，bitset 对 I/O 也很有用：

```
cin>>b;                // 从输入读取一个 bitset
cout<<bitset<8>('c'); // 输出字符 'c' 的位模式
```

当读入 bitset 时，输入流会寻找 0 和 1，例如，如果输入下面内容：

```
10121
```

输入流会读入 101，21 会被留下。

对于字节和字，bitset 中的位是由右至左编号的（从最低有效位到最高有效位）。这样，第 7 位的值就是  $2^7$ ：

```
7: 6: 5: 4: 3: 2: 1: 0:
1 0 1 0 0 1 1 1
```

对于 bitset 而言，编号顺序不仅仅是遵循惯例的问题，还起到二进制位的索引下标的作用。例如：

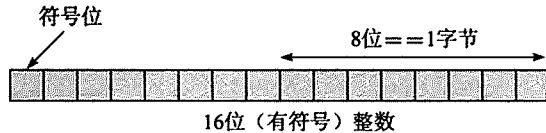
```
int main()
{
    constexpr int max = 10;
    for (bitset<max> b; cin>>b; ) {
        cout << b << '\n';
        for (int i=0; i<max; ++i) cout << b[i]; // 反转顺序
        cout << '\n';
    }
}
```



如果你希望了解 `bitset` 的更多内容，请参考联机帮助、手册或者专业级的教材。

### 25.5.3 有符号数和无符号数

与大多数语言一样，C++ 同时支持有符号数和无符号数。无符号数在内存中的描述是很简单的：第 0 位表示 1、第 1 位表示 2，第 2 位表示 4，依此类推。但是，有符号数就引出一个问题：我们如何区分正数和负数？对此，C++ 给了硬件设计者一定的自由选择的余地，不过几乎所有实现都使用了二进制补码表示法。最靠左的二进制位（最高有效位）被用来作为“符号位”：



如果符号位为 1，就表示负数。二进制补码表示法已经成为事实上的标准方法。为了节约篇幅，我们只讨论如何在 4 位二进制整数中表示有符号数值：

正数：	0	1	2	4	7
	0000	0001	0010	0100	0111
负数：	1111	1110	1101	1011	1000
	-1	-2	-3	-5	-8

基本思想就是：用  $x$  的位模式的补码 ( $\sim x$ ，参见 25.5.1 节) 来表示  $-(x+1)$  的位模式。

到目前为止，我们一直在使用有符号整数（如 `int`）。更好的程序设计原则是：

- 当需要表示数值时，使用有符号数（如 `int`）。
- 当需要表示位集合时，使用无符号数（如 `unsigned int`）。

这是一个很好的程序设计原则，但很难严格遵循，因为一些人更喜欢用无符号数进行某些算术运算，而我们有时需要用这类代码。特别是还有一些历史遗留问题，例如，在 C 语言历史的早期，`int` 还是 16 位大小，每一位都很重要，而一个 `vector` 的大小 `v.size()` 返回的是一个无符号数。例如：



```
vector<int> v;
// ...
for (int i = 0; i < v.size(); ++i) cout << v[i] << '\n';
```

好的编译器会给出一个警告，指出存在有符号数（即 `i`）和无符号数（即 `v.size()`）混合运算的情况。有符号数和无符号数混合运算有可能会带来灾难性的后果。例如，循环变量 `i` 可能会溢出，即，`v.size()` 有可能比最大的有符号 `int` 值还要大。当 `i` 的值增大到有符号 `int` 所能表示的最大正数（2 的幂减 1，幂次等于 `int` 的二进制位数减 1，如，`int` 为 16 位宽度，此值为  $2^{15}-1$ ）时，下一次增 1 运算不会得到更大的整型值，而会得到一个负数。因此循环永远也不会停止！每当我们到达最大整数时，接着就会从最小负 `int` 值重新开始。因此，如果 `v.size()` 的值为  $32 \times 1024$  或者更大，循环变量为 16 位 `int` 型的话，这个循环就是一个（可能非常严重的）bug。如果循环变量是 32 位 `int` 型的话，当 `v.size()` 的值大于等于  $2 \times 1024 \times 1024 \times 1024$  时就会出现同样的问题。



因此，严格来说，本书中的大部分循环都是有问题的。换句话说，对于嵌入式系统，我们要么证实循环不会达到临界点，要么将循环代码改写为另外一种形式。为了避免这个问

题，我们可以使用 `vector` 提供的 `size_type` 或者是迭代器：

```
for (vector<int>::size_type i = 0; i<v.size(); ++i) cout << v[i] << '\n';

for (vector<int>::iterator p = v.begin(); p!=v.end(); ++p) cout << *p << '\n';

for (int x : v) cout << x << '\n';
```

`size_type` 确保是无符号的，因此，第一种形式（使用无符号数）与 `int` 型循环变量的版本相比，多出一个二进制位来表示循环变量的数值（而不是符号）。这个改进很重要，但终究只是多出一位来表示循环的范围（循环次数多出一倍）。而使用迭代器的版本就不存在这个限制。

### 试一试

下面这个例子看起来没什么问题，但它实际上是个死循环：

```
void infinite()
{
    unsigned char max = 160;    // 非常大
    for (signed char i=0; i<max; ++i) cout << int(i) << '\n';
}
```

运行这个程序，解释为什么会形成死循环。

大致来说，我们有两个原因将无符号数当作整数来使用，而不是简单作为一组二进制位（即，不使用 `+`、`-`、`*` 和 `/`）：

- 有更多的二进制位来表示数值，从而获得更高的精度。
- 用来表示逻辑属性，其值不能是负数。

前者就是我们刚刚看到的，使用无符号循环变量带来的效果。

混合使用有符号数和无符号数的问题在于，在 C++ 中（C 中也一样），两者转换的方式很奇怪，而且难以记忆。例如：

```
unsigned int ui = -1;

int si = ui;
int si2 = ui+2;
unsigned ui2 = ui+2;
```

奇怪的是，第一个初始化操作能够成功完成，`ui` 被赋予 4294967295，这个 32 位无符号整数的位模式恰好与有符号数 `-1` 相同（二进制表示为“全 1”）。一些人认为这样编写代码很简洁，因此喜欢用 `-1` 表示“全 1”。而另外一些人则认为这是一个不好的特性。从无符号数到有符号数的转换规则与此类似，因此第二个初始化操作将 `si` 的初值设置为 `-1`。如我们所料，`si2` 被赋予 `1` (`-1+2==1`)，`ui2` 也被赋予同样的值。`ui2` 的计算结果应该会让你惊讶一会儿：为什么 `4294967295+2` 会得到 `1`？如果我们将 `4294967295` 表示为十六进制 `0xffffffff`，就比较清楚了：`4294967295` 是最大的 32 位无符号整数，因此 `4294967297` 无法用 32 位整数表示，无论是无符号或是有符号都不行。我们可以说 `4294967295+2` 产生了溢出，或者更准确地说，这里使用了模运算。也就是说，32 位整数运算都要对  $2^{32}$  取模。

现在所有事情都清楚了吗？即使你已经弄清了这些奇怪的规则，我们也希望上述讨论能让你信服这样一个观点：用无符号数表示数值，以获得一个额外的二进制位的精度，无异于玩火。这样做会导致混乱，而且是潜在的错误之源。

如果发生整数溢出，会有什么后果呢？考虑下面代码：

```
int i = 0;
while (++i) print(i); // 按整数打印 i，后接一个空格
```

这段程序会打印出什么样的数值序列呢？显然，这取决于 `Int` 是如何定义的（注意，这里大写的 `I` 并不是打字错误）。对任何一种大小有限制的整数类型，最终都会出现溢出的情况。如果 `Int` 是无符号类型（如 `unsigned char`、`unsigned int` 或 `unsigned long long`），由于 `++` 运算会进行模运算，循环变量 `i` 达到最大值后会变为 0（循环从而终止）。如果 `Int` 是有符号类型（如 `signed char`），`i` 达到最大值后会突然变为最小的负数然后逐渐增大为 0（循环终止）。例如，如果 `Int` 是 `signed char`，输出的序列为 `1 2 ... 126 127 -128 -127 ... -2 -1`。

再次提出那个问题：如果发生整数溢出，会有什么后果？答案是程序还会继续执行，就好像有更多二进制位保存结果一样，但实际上一些无法容纳的二进制被丢弃了。一般的策略是丢弃最靠左的位（最高有效位）。如果在赋值语句中赋予变量一个超出其表示范围的值，也会看到类似的效果：

```
int si = 257; // 不能放入一个 char 中
char c = si; // 隐式转换为 char
unsigned char uc = si;
signed char sc = si;
print(si); print(c); print(uc); print(sc); cout << '\n';
```

```
si = 129; // doesn't fit into a signed char
c = si;
uc = si;
sc = si;
print(si); print(c); print(uc); print(sc);
```

输出结果为

```
257  1      1      1
129 -127  129  -127
```

产生这样的结果的原因是，257 比 8 个二进制位所能表示的最大值（255，即“8 个 1”）大 2；129 比 7 个二进制位所能表示的最大值（127，即“7 个 1”）大 2，因而符号位被置位，有符号变量的值变为负数。注意，程序的运行结果表明：在我们的计算机上，`char` 是有符号的（`c` 的行为与 `sc` 一致，而与 `uc` 不同）。

### 试一试

在纸上画出上面程序中涉及的位模式，算出若 `si=128`，输出结果是什么。运行程序，检验你的手算结果是否正确。

旁注：我们为什么要引入 `print()` 函数？为什么不用：

```
cout << i << ' ';
```

原因很简单，如果 `i` 是 `char` 型，这条语句就会输出一个字符，而不是其整型值。因此，我们引入 `print` 函数，对所有整数类型进行一致处理，定义如下：

```
template<typename T> void print(T i) { cout << i << '\t'; }

void print(char i) { cout << int(i) << '\t'; }
```

```
void print(signed char i) { cout << int(i) << '\t'; }
```

```
void print(unsigned char i) { cout << int(i) << '\t'; }
```

总结一下：无符号数的使用可以和有符号数完全一样（包括普通的算术运算），但是要  
避免这样使用，因为这样做会使问题变得非常复杂，程序也容易出错。

- 永远不要为了多出一个二进制位的精度而用无符号数表示数值。
- 如果你需要一个额外的二进制位，很快就会再需要一个。

不幸的是，无符号数的算术运算是很难完全避免的：

- 标准库容器的下标都是无符号数。
- 一些人就是喜欢无符号数的算术运算。



## 25.5.4 位运算

我们为什么需要位运算呢？好吧，实际上大多数人并不喜欢位运算。位处理靠近下层而且容易出错，有可能的话就应该尽量使用替代方法。但是，位描述和位运算是非常基础的，也是非常有用的，我们不能假装它不存在。这听起来有些消极，有些令人气馁，但值得仔细思考。一些人确实喜欢摆弄位和字节，因此应当记住：位处理在某些时候是不可避免的（虽然开始时有些不情愿，但在过程中你很可能获得不少乐趣），但不要在程序中到处使用位运算。John Bentley 的两句话用在这里很适合：“喜弄位者易被位反噬 (bitten)” “喜弄字节者易被字节反噬 (byttten)”。

那么，什么时候应该使用位运算呢？有些情况下，应用程序要处理的对象就是位的形式，那么使用位运算就是顺理成章的了。这方面的例子包括硬件指示器（“标识位”）、低层通信（需要从字节流中提取不同类型的值）、图形应用（需要用多个层次的图像组成图片）以及加密（参见下一节）。

例如，考虑如何从一个整数中提取（低层）信息（可能我们想将它按字节传输，就像二进制 I/O 的处理方式）：

```
void f(short val)                // 假设 16 位，2 字节短整数
{
    unsigned char right = val&0xff; // 最右（最低有效）字节
    unsigned char left = val>>8;   // 最左（最高有效）字节
    // ...
    bool negative = val&0x8000;    // 符号位
    // ...
}
```

这种运算是很常见的，通常被称为“移位和掩码”运算。“移位”运算（使用 << 或 >>）将二进制位移动到我们所期望的位置（本例中是移动到字的最低有效部分），以方便处理。“掩码”运算是将运算对象与一个特殊的位模式（本例中是 0xff）进行“位与”（&）运算，目的是去掉那些我们不需要的位。

如果希望对二进制位命名，我们通常使用枚举类型，例如：

```
enum Printer_flags {
    acknowledge=1,
    paper_empty=1<<1,
    busy=1<<2,
    out_of_black=1<<3,
```

```

    out_of_color=1<<4,
    //...
};

```

每个枚举常量被赋予的值与名字的含义是完全吻合的：

```

out_of_color    16    0x10    0001 0000
out_of_black    8     0x8     0000 1000
busy            4     0x4     0000 0100
paper_empty     2     0x2     0000 0010
acknowledge     1     0x1     0000 0001

```

这种常量值在某些情况下是很有用的，因为它们可以任意组合：

```

unsigned char x = out_of_color | out_of_black; // x 变为 24 (16+8)
x |= paper_empty; // x 变为 26 (24+2)

```

注意，这里的 `|=` 起到了“置位”的作用。类似地，`&` 可以起到“检测位”的作用：

```

if (x & out_of_color) { // out_of_color 被置位吗？(是的，置位了)
    //...
}

```

命名二进制位同样可以用来进行掩码运算：

```

unsigned char y = x & (out_of_color | out_of_black); // y 变为 24

```

此时，`y` 的内容就是 `x` 的第 3 位和第 4 位 (`out_of_color` 和 `out_of_black` 对应第 3、4 位)。

将 `enum` 作为二进制位集合来使用，是一种非常常用的方法。此时，我们就需要一种方法将位运算的计算结果再“转换回”`enum`：

```

Flags z = Printer_flags(out_of_color | out_of_black); // 类型转换是必要的

```

之所以需要这样的类型转换，是因为编译器不知道 `out_of_color | out_of_black` 的值是否是合法的 `Flags` 值。编译器的怀疑是合理的：毕竟，没有任何一个枚举常量的值为 24 (`out_of_color | out_of_black` 的计算结果)。当然，在本例中，我们知道赋值语句是合理的（但编译器并不知道）。

### 25.5.5 位域

如前所述，硬件接口是使用位运算最多的地方。通常，接口就是一组二进制位和不同大小的数。“二进制位和数”通常是命名的，出现在字的不同位置，我们称之为“设备寄存器”（device register）。C++ 提供了一种特殊的语言特性来处理这种固定的数据布局：位域（bitfield）。我们来考察这样一个例子：操作系统中页管理程序所使用的页号，其结构为：

位置:	31:	9:	6:	3:	2:	1:	0:
PPN:	22	3	3	1	1	1	1
名字:	PFN	未用	CCA	↑ 脏页 不可达	↑ 全局 有效		

可以看到，一个 32 位的字被分为两个数值域（一个占用 22 位，一个占用 3 位）和四个标识位（每个 1 位）。这些数据的大小和位置是固定的。在字的中间还有一个“未用”域。此数据布局可用如下 `struct` 类型来描述：

```

struct PPN {
    unsigned int PFN : 22; // R6000 物理页号
    int : 3; // 页帧号
    unsigned int CCA : 3; // 未用
    bool nonreachable : 1; // 协同缓存算法
    bool dirty : 1;
    bool valid : 1;
    bool global : 1;
};

```

我们必须查阅参考手册才知道 PFN 和 CCA 应该定义为无符号整数，但如果没有手册的帮助，我们可以直接利用位模式图来设计 struct。位域在字中是由左至右排列的。每个域的位宽用一个整数指定，名字和位宽之间用冒号隔开。不允许为位域指定绝对的位置（如第 8 位）。如果总位宽超出了一个字的容纳能力，则超出部分被置于下一个字中。希望这种方式能满足你的需求。定义完成后，位域的使用就与其他变量没什么差别了：

```

void part_of_VM_system(PPN * p)
{
    // ...
    if (p->dirty) { // 内容改变
        // 拷贝到磁盘
        p->dirty = 0;
    }
    // ...
}

```

如果没有位域，要想获得一个字中间区域的信息，就必须使用复杂的移位和掩码运算，位域使这一切变得简单。例如，对于一个 PPN 类型的对象 pn，可以这样提取 CCA：

```
unsigned int x = pn.CCA; // 提取 CCA
```

如果是用一个 int 型变量 pni 表示页号，则必须这样来提取 CCA：

```
unsigned int y = (pni>>4)&0x7; // 提取 CCA
```


也就是说，先将 CCA 右移到最低有效位，然后与 0x7（即最右 3 位置位）进行掩码运算来去掉所有其他位。你可以查看一下编译得到的机器码，多半会发现生成的代码就是这两个指令。

CCA、PPN 和 PFN 这种字头缩写形式是常见的低层程序设计风格，显然，在设计普通程序时，这并不是一个好的风格。

### 25.5.6 实例：简单加密

接下来，我们实现一个简单的加密算法：微型加密算法（Tiny Encryption Algorithm, TEA），作为位/字节级别数据处理的一个实例。这个算法最初是由剑桥大学的 David Wheeler 设计的（见 22.2.1 节）。它很简单，但应付一般攻击还是绰绰有余的。

你不必过于仔细地阅读加密程序（除非你真的需要理解算法，而且对困难有心理准备）。我们给出这个加密程序只是为了让你体会一下如何编写实用的位处理代码。如果你希望学习加密的知识，请查阅专门的教材。至于用其他语言实现 TEA 算法的相关内容，请参考 [http://en.wikipedia.org/wiki/Tiny\\_Encryption\\_Algorithm](http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm) 以及英国布拉福德大学 Simon Shepherd 教授关于 TEA 的网站。

加密/解密的基本思想是很简单的。我想发送给你一些文本，但我不想让其他人看懂发 

送的内容。因此，我先把要发送的文本进行转换，然后再发送，使得不知道确切转换方式的人就无法看懂转换后的内容；而你是知道转换方法的，可以通过逆变换得到原始文本。这个转换过程就称为加密。进行加密需要一个算法（我们必须假定所有人都能获得这个算法）和一个称为“密钥”的字符串。你和我都知道密钥（我们希望窃听者不知道）。当你获得密文时，可以使用“密钥”对其解密，即，重新构造出我要发送的“明文”。

TEA 算法接受三个参数， $v$  是包含两个无符号 long ( $v[0]$ ,  $v[1]$ ) 的数组，表示要加密的 8 个字符， $w$  是用来保存密文的，也是包含两个无符号 long ( $w[0]$ ,  $w[1]$ ) 的数组，而  $k$  是密钥，是包含 4 个无符号 long ( $k[0]$ ... $k[3]$ ) 的数组：

```
void encipher(
    const unsigned long *const v,
    unsigned long *const w,
    const unsigned long * const k)
{
    static_assert(sizeof(long)==4,"size of long wrong for TEA");

    unsigned long y = v[0];
    unsigned long z = v[1];
    unsigned long sum = 0;
    const unsigned long delta = 0x9E3779B9;

    for (unsigned long n = 32; n-->0; ) {
        y += (z<<4 ^ z>>5) + z^sum + k[sum&3];
        sum += delta;
        z += (y<<4 ^ y>>5) + y^sum + k[sum>>11 & 3];
    }
    w[0]=y;
    w[1]=z;
}
```

注意，所有数组都是无符号类型，这样我们就可以放心地进行位运算，而不必担心突然出现负数。移位 (<< 和 >>)、异或 (^) 以及位与 (&) 这些位运算结合无符号数加法运算形成了完整的加密算法。这段代码只能用于 long 的大小是 4 字节的机器。也就是说，代码中散布着“魔数”（即 `sizeof(long)==4`）。如前所述，这通常不是一种好的程序设计策略，但这样写程序，代码能放在一页纸内。而相应的数学公式也能写在一个信封的背面，或者，按照最初的设想，牢牢地记在程序员的头脑中。David Wheeler 最初设计算法时，就希望加密算法如此简单：在他外出旅行忘带笔记和便携式电脑的情况下，仅凭头脑也能回忆起算法，完成加密操作。除了简洁，这段代码还很快。变量  $n$  的值决定了循环次数：循环次数越多，加密强度越高。据我所知，当  $n==32$  时，TEA 尚未被攻破过。

下面是解密函数：

```
void decipher(
    const unsigned long *const v,
    unsigned long *const w,
    const unsigned long * const k)
{
    static_assert(sizeof(long)==4,"size of long wrong for TEA");

    unsigned long y = v[0];
    unsigned long z = v[1];
    unsigned long sum = 0xC6EF3720;
    const unsigned long delta = 0x9E3779B9;
```

```

// sum=delta<<5, 一般而言 sum=delta*n
for (unsigned long n = 32; n-- > 0; ) {
    z -= (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
    sum -= delta;
    y -= (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
}
w[0]=y;
w[1]=z;
}

```

如果文件需要在不安全的信道上传输，我们可以像下面代码这样对其用 TEA 加密：

```

int main()    // 发送者
{
    const int nchar = 2*sizeof(long);    // 64 位
    const int kchar = 2*nchar;          // 128 位

    string op;
    string key;
    string infile;
    string outfile;
    cout << "please enter input file name, output file name, and key:\n";
    cin >> infile >> outfile >> key;
    while (key.size()<kchar) key += '0'; // 补齐密钥
    ifstream inf(infile);
    ofstream outf(outfile);
    if (!inf || !outf) error("bad file name");

    const unsigned long* k =
        reinterpret_cast<const unsigned long*>(key.data());

    unsigned long outptr[2];
    char inbuf[nchar];
    unsigned long* inptr = reinterpret_cast<unsigned long*>(inbuf);
    int count = 0;

    while (inf.get(inbuf[count])) {
        outf << hex;                // 使用十六进制输出
        if (++count == nchar) {
            encipher(inptr,outptr,k);
            // 用前导 0 补齐
            outf << setw(8) << setfill('0') << outptr[0] << ' '
                << setw(8) << setfill('0') << outptr[1] << ' ';
            count = 0;
        }
    }

    if (count) { // 补齐
        while(count != nchar) inbuf[count++] = '0';
        encipher(inptr,outptr,k);
        outf << outptr[0] << ' ' << outptr[1] << ' ';
    }
}

```

程序最重要的部分是 while 循环，剩余部分只是起辅助作用。while 循环读入字符存到输入缓冲区 inbuf 中，每次都 8 个字符传递给 encipher() 进行加密。TEA 不关心传递给它的字符，实际上，它并不知道自己加密的是什么。例如，你可能在加密一幅照片或者一次电话通话。TEA 所关心的只是接受 64 位（两个无符号 long）明文，生成 64 位密文。因此，我



们用一个指针指向 inbuf，将其转换为 unsigned long\* 类型并传递给 TEA。对密钥也是相同的处理方式。由于 TEA 使用 128 位的密钥（4 个 unsigned long），因此我们对用户输入“打补丁”，将其补齐为 128 位。最后一条语句将 0 补在文本末尾，使其位数变为 TEA 所要求的 64 的整数倍（8 个字节）。

密文如何传输呢？我们可以自由选择传输方法，但要注意的是，由于密文是二进制位序列，而不是 ASCII 或 Unicode 字符，因此不能像普通文本一样传输。可以选择二进制 I/O 方法（见 11.3.2 节），不过在本例中我们用十六进制数的形式输出密文。

```

5b8fb57c 806fbcce 2db72335 23989d1d 991206bc 0363a308
8f8111ac 38f3f2f3 9110a4bb c5e1389f 64d7efe8 ba133559
4cc00fa0 6f77e537 bde7925f f87045f0 472bad6e dd228bc3
a5686903 51cc9a61 fc19144e d3bcde62 4fdb7dc8 43d565e5
fld3f026 b2887412 97580690 d2ea4f8b 2d8fb3b7 936cfa6d
6a13ef90 fd036721 b80035e1 7467d8d8 d32bb67e 29923fde
197d4cd6 76874951 418e8a43 e9644c2a eb10e848 ba67dcd8
7115211f dbe32069 e4e92f87 8bf3e33e b18f942c c965b87a
44489114 18d4f2bc 256da1bf c57b1788 9113c372 12662c23
eeb63c45 82499657 a8265f44 7c866aae 7c80a631 e91475e1
5991ab8b 6aedbb73 71b642c4 8d78f68b d602bfe4 d1eadde7
55f20835 1a6d3a4b 202c36b8 66a1e0f2 771993f3 11d1d0ab
74a8cfd4 4ce54f5a e5fda09d acbdf110 259a1a19 b964a3a9
456fd8a3 1e78591b 07c8f5a2 101641ec d0c9d7e1 60dbeb11
b9ad8e72 ad30b839 201fc553 a34a79c4 217ca84d 30f666c6
d018e61c d1c94ea6 6ca73314 cd60def1 6e16870e 45b94dc0
d7b44fcd 96e0425a 72839f71 d5b6427c 214340f9 8745882f
0602c1a2 b437c759 ca0e3903 bd4d8460 edd0551e 31d34dd3
c3f943ed d2cae477 4d9d0b61 f647c377 0d9d303a ce1de974
f9449784 df460350 5d42b06c d4dedb54 17811b5f 4f723692
14d67edb 11da5447 67bc059a 4600f047 63e439e3 2e9d15f7
4f21bbbe 3d7c5e9b 433564f5 c3ff2597 3a1ealdf 305e2713
9421d209 2b52384f f78fbae7 d03c1f58 6832680a 207609f3
9f2c5a59 ee31f147 2ebc3651 e017d9d6 d6d60ce2 2be1f2f9
eb9de5a8 95657e30 cad37fda 7bce06f4 457daf44 eb257206
418c24a5 de687477 5c1b3155 f744fbff 26800820 92224e9d
43c03a51 d168f2d1 624c54fe 73c99473 1bce8fbb 62452495
5de382c1 1a789445 aa00178a 3e583446 dcbd64c5 ddda1e73
fa168da2 60bc109e 7102ce40 9fed3a0b 44245e5d f612ed4c
b5c161f8 97ff2fc0 1dbf5674 45965600 b04c0afa b537a770
9ab9bee7 1624516c 0d3e556b 6de6eda7 d159b10e 71d5c1a6
b8bb87de 316a0fc9 62c01a3d 0a24a51f 86365842 52dabf4d
372ac18b 9a5df281 35c9f8d7 07c8f9b4 36b6d9a5 a08ae934
239efba5 5fe3fa6f 659df805 faf4c378 4c2048d6 e8bf4939
31167a93 43d17818 998ba244 55dba8ee 799e07e7 43d26aef
d5682864 05e641dc b5948ec8 03457e3f 80c934fe cc5ad4f9
0dc16bb2 a50aa1ef d62ef1cd f8fbbf67 30c17f12 718f4d9a
43295fed 561de2a0

```



### 试一试

如果密钥是 bs，明文是什么？



这个程序还不是很完美，任何安全专家都会告诉你，将明文和密文保存在一起是个笨主

意，对于打补丁、密钥长度为 2 等问题也会提出看法。不过，本书是一本程序设计书籍，而非计算机安全书籍。

我们测试这个程序的方法是：读入密文，进行解密，与明文比较。当编写程序时，能进行简单的正确性测试总是好的。

下面是解密程序的核心部分：

```
unsigned long inptr[2];
char outbuf[nchar+1];
outbuf[nchar]=0; // 终止符
unsigned long* outptr = reinterpret_cast<unsigned long*>(outbuf);
inf.setf(ios_base::hex ,ios_base::basefield); // 使用十六进制输入

while (inf>>inptr[0]>>inptr[1]) {
    decipher(inptr,outptr,k);
    outf<<outbuf;
}
```

注意这条语句的使用：

```
inf.setf(ios_base::hex ,ios_base::basefield);
```

它读入十六进制数。对于解密程序而言，这些数据保存在输出缓冲区 outbuf 中，进行类型转换后作为二进制位进行处理。

TEA 这个例子属于嵌入式系统程序设计范畴吗？这不太明确，但你可以想象它被用于安全系统或者金融业务系统，这些应用都是由很多“小设备”组成的。无论如何，TEA 程序展示了很多好的嵌入式程序设计方法：它基于一个清晰的（数学）模型，能确保其正确性，它很简洁、很快速而且直接依赖于硬件特性。encipher() 和 decipher() 的接口风格并不符合我们的习惯。但是，它们不仅用 C++ 实现，还用 C 实现，因此不能使用 C 语言不支持的 C++ 特性。另外，程序中出现的很多“魔数”都直接来自于数学模型。

## 25.6 编码规范

错误的源头是多种多样的。最严重、最难以修正的错误都是与上层设计决策相关的，这些设计决策包括总体的错误处理策略、遵循（或不遵循）特定的标准、算法设计、数据表示方式等等。这些问题已经超出了本书的讨论范围，我们讨论的重点是那些由于糟糕的程序设计方式而导致的错误。“糟糕的程序设计方式”主要是指使用语言特性的方式容易引起错误，或者表达思想的方式模糊不清。

编码规范试图解决后一类问题，它定义一个“排版风格”来为程序员指引方向：对于特定应用，使用哪些 C++ 语言特性比较恰当。例如，嵌入式程序设计编码规范可能会禁止使用 new。通常，编码规范还会尽力令不同程序员编写出的代码更为相似，虽然他们可以自由选择程序设计风格。例如，某个编码规范可能会要求循环结构都用 for 语句实现（即禁止 while 语句）。这能使代码更一致，在开发大型项目时，这对代码维护有着重要作用。请注意，一种编码规范只针对一类特定的程序设计，只为某些特定的程序员提供帮助。不存在一种适合于所有 C++ 应用和所有 C++ 程序员的编码规范。

编码规范试图解决的是解决方案表达方式方面的问题，而不是应用的复杂性方面的问题。因此，我们可以说，编码规范试图解决偶然复杂性，而不是必然复杂性。

引起偶然复杂性的主要原因包括：



- 过于聪明的程序员，他们在表达复杂解决方案时试图使用那些并不理解或并不喜欢的语言特性。
- 未经良好培训的程序员，不会使用最适合的语言特性和库功能。
- 不必要的程序设计风格变化，这会导致完成相似工作的代码在形式上差异很大，给代码维护人员带来困扰。
- 不恰当的程序设计语言，这会导致所使用的语言特性非常不适合于某些应用领域或某些程序员。
- 没有有效利用库，导致程序中存在大量专门处理低层资源的代码。
- 不恰当的编码规范，导致解决某些问题时，付出额外的工作量或者无法采用最优的解决方案，从而引起一些棘手的问题。

### 25.6.1 编码规范应该是怎样的



一个好的编码规范应该能帮助程序员写出好的代码，即，对于一些小的程序设计问题能够直接给出答案，而无须程序员花费时间逐个解决。有一句在工程师间流传很久的格言：“形式即解放。”理想情况下，编码规范应该是指示性的，指出应该做什么。看起来显然应该这样，但很多编码规范只是简单罗列了不能做什么，而没有指明应该做什么。仅仅告诉程序员什么不能做不会对编程有什么帮助，而且常常会令人很恼火。



好的编码规范中的原则应该都是可验证的，最好可以通过程序来验证。也就是说，当我们写完程序后，查看一下代码就可以很容易地回答这个问题：“我是否违反了编码原则？”

对于列出的编码原则，一个好的编码规范应该解释清楚其理论依据。不能只是对程序员说“我们就是这样做的！”，这只会增加程序员的厌恶感。更糟的是，如果程序员觉得规范的某些部分毫无益处，甚至妨碍他们写出高质量的程序，就会不停地尝试推翻它。不要期待编码规范的全部内容都受到欢迎。即使是最好的编码规范也都是一定程度上的折中，而且大多数“不该做”都会引起问题，即便你自己没碰到。例如，不一致的命名规范是混乱之源，但不同人都有自己特别偏好的命名规范，而强烈抵触其他规范。例如，我个人认为首字母大写的标识符命名法（如 `CamelCodingStyle`）“非常丑陋”，强烈倾向于“下划线风格”（如 `underscore_style`），认为这种风格更清晰、本质上更易读，很多人也赞同我的观点。但另一方面，也有很多人并不赞同这一观点。显然，没有任何一种命名规范能满足所有人，但多数情况下，一个一致风格绝对比没有规范更好。

关于编码规范应该是怎样的，我们总结如下：

- 一个好的编码规范应该针对特定应用领域和特定程序员来设计。
- 一个好的编码规范应该既有指示性，又有限制性。
  - 推荐一些“基础的”库功能作为指示性原则，通常是最有效的方式。
- 一个编码规范就是一个编码原则集合，指明了程序风格
  - 通常应该指定命名和缩进原则：如“使用‘Stroustrup 布局风格’”。
  - 通常应该指定允许使用的语言子集：如“不要使用 `new` 或 `throw`”。
  - 通常应该指定注释原则：如“每个函数应该用一段注释描述其功能”。
  - 通常应该指明使用哪些库：如“使用 `<iostream>` 而不是 `<stdio.h>`”或者“使用 `vector` 和 `string` 而不是内置数组和 C 风格字符串”。
- 大多数编码规范的共同目标是提高程序的

- 可靠性
  - 可移植性
  - 可维护性
  - 可测试性
  - 重用性
  - 可扩展性
  - 可读性
- 一个好的编码规范要比没有规范更好。如果没有编码规范，就不应该启动一个大型（很多人，多年才能完成）的工业项目。
  - 一个糟糕的编码规范甚至比没有规范更糟。例如，一个限制使用 C 语言子集的 C++ 编程规范是有害的。但不幸的是，糟糕的编码规范并不罕见。
  - 任何一个编码规范，即便是好的编码规范，也都会有程序员不喜欢。大多数程序员希望按自己喜欢的方式编写代码。

## 25.6.2 编码原则实例

下面，我们会给你列出一些编码规范中的编码原则。自然，我们之所以选择这些原则，就是希望它们能对你有所帮助。但是，我们所见过的实际的编码规范还没有少于 35 页的，大多数还要长得更多。所以，在本节中我们不会给出一个完整的编码规范。而且，如前所述，任何好的编码规范都是为特定应用领域和特定程序员所设计的。因此，我们不会伪称这些编码原则是通用的。

我们为编码原则编了号，并为每条原则给出了简短的设计依据。为了帮助理解，很多原则都附带了一些例子。我们将编码原则分为推荐规则（recommendation）和严格规则（firm rule）两类，对于前者，程序员偶尔可以不遵守，而后者则必须严格遵守。对于现实中的编码规范，只有管理者才能授权修改严格规则。如果你在程序中违反了推荐规则或者严格规则，应该通过注释来说明原因。在规范中，原则的例外情况也可与原则一并列出。本节中给出的每条严格原则都用一个大写字母 R 及其编号标识，而推荐原则都用小写字母 r 及其编号标识。

我们将编码原则分类如下：

- 一般原则
- 预处理原则
- 命名和布局原则
- 类原则
- 函数和表达式原则
- 硬实时原则
- 关键系统原则

“硬实时”和“关键系统”原则仅用于硬实时和关键系统程序设计。

与一个好的实际编码规范相比，我们使用的有些术语并不明确（如，“关键”的确切含义是什么？），而列出的原则也有些过于简单。你会发现它们与 JSF++ 原则（见 25.6.3 节）有相似之处，这并不是偶然的，我本人参与了 JSF++ 原则的规划。不过，本书中的代码实例并未遵循本节中给出的编码原则，毕竟，本书中的程序并不是为关键的嵌入式系统所编写的。

### 1. 一般原则

**R100:** 任何函数和类的代码规模都不应超过 200 行 (不包括注释)。

原因: 长的函数和类会更复杂, 因而难于理解和测试。

**r101:** 任何函数和类都应该能完全显示在一屏上, 并完成单一的逻辑功能。

原因: 如果程序员只能看到函数或类的一部分, 就很可能漏掉有错误的部分。如果一个函数试图完成多个功能, 与单功能的函数相比, 其规模就可能很大, 而且会更复杂。

**R102:** 所有代码都应该遵循 ISO/IEC 14882:2011(E) C++ 标准。

原因: 在 ISO/IEC 14882 标准之上的扩展和变形可能会不稳定, 定义不明确, 而且可能影响可移植性。

### 2. 预处理原则

**R200:** 除了用于源码控制的 `#ifdef` 和 `#ifndef` 之外, 不要使用宏。

原因: 宏不遵守定义域和类型规则, 而且使代码变得更不清晰、不易读。

**R201:** `#include` 只能用于包含头文件 (`*.h`)。

原因: `#include` 用于访问接口的声明而非实现细节。

**R202:** 所有 `#include` 语句都应位于任何非预处理声明之前。

原因: 如果 `#include` 语句位于程序中间, 就很可能被阅读程序的人忽略, 而且容易导致程序不同部分对名字的解析不一致。

**R203:** 头文件 (`*.h`) 不应包含非常量变量的定义或非内联、非模板函数定义。

原因: 头文件应该包含接口声明而非实现细节。但是, 常量常被看作接口的一部分; 出于性能的考虑, 一些非常简单的函数应该作为内联函数 (因此应该放在头文件中); 而当前的编译器要求完整的模板定义都放在头文件中。

### 3. 命名和布局原则

**R300:** 应该使用缩进, 并且在一个源码文件中缩进风格应该一致。

原因: 可读性和代码风格。

**R301:** 每条新语句都另起一行。

原因: 可读性。

例子:

```
int a = 7; x = a+7; f(x,9); // 违反
int a = 7;           // 正确
x = a+7;           // 正确
f(x,9);           // 正确
```

例子:

```
if (p<q) cout << *p; // 违反
```

例子:

```
if (p<q)
    cout << *p; // 正确
```

**R302:** 标识符的名字应该都具有描述性。

标识符可以包含常见的缩写和字头缩略。

如果 *x*、*y*、*i*、*j* 是按习惯方式使用，可以认为是具有描述性的。

使用下划线风格 (`number_of_elements`) 而不是字头缩略风格 (`numberOfElement`)。

不要用匈牙利命名法。

类型、模板和名字空间的命名都以大写字母开头。

避免过长的名字。

例子: `Device_driver` 和 `Buffer_pool`。

原因: 可读性。

注意: C++ 标准规定, 以下划线开头的标识符留作语言实现所用, 因此在用户程序中应被禁止。

例外: 调用经过认证的库, 来自库中名字是可以使用的。

**R303:** 标识符不能只在以下方面不同:

- 大小写不同;
- 只相差下划线;
- 只是字母 *O*、数字 0 或字母 *D* 间的替换;
- 只是字母 *I*、数字 1 或字母 *l* 之间的替换;
- 只是字母 *S* 和数字 5 之间的替换;
- 只是字母 *Z* 和数字 2 之间的替换;
- 只是字母 *n* 和字母 *h* 之间的替换。

例子: `Head` 和 `head` // 违反

原因: 可读性。

**R304:** 标识符不能只包含大写字母和下划线。

例子: `BLUE` 和 `BLUE_CHEESE` // 违反

原因: 全部大写字母的标识符被广泛用于宏名, 可能用于经过认证的库中的 `#include` 文件, 而不应该用于用户程序。

例外: 宏名用于保护 `#include` 不被重复包含。

#### 4. 函数和表达式原则

**r400:** 内层循环的标识符和外层循环的标识符不应重名。

原因: 可读性和代码风格。

例子:

```
int var = 9; { int var = 7; ++var; } // 违反: var 重名
```

**R401:** 声明的作用域应该尽量小。

原因: 保持变量的初始化和使用尽量靠近, 以降低混乱的可能性; 令离开作用域的变量释放其资源。

**R402:** 所有变量都要初始化。

例子:

```
int var;                // 违反: var 没有初始化
```

原因: 未初始化的变量通常是错误之源。

例外: 如果数组或容器会立即从输入接收数据, 则不必初始化。

注意: 许多类型, 例如 `vector` 和 `string`, 有默认的构造函数来完成初始化。

**R403:** 不应使用类型转换。

原因: 类型转换是错误之源。

例外: `dynamic_cast` 可以使用。

例外: 新风格的类型转换可以使用, 用来将硬件地址转换为指针, 或者将从程序外部 (如 GUI 库) 获取的 `void*` 转换为恰当类型的指针。

**R404:** 函数接口中不应使用内置数组类型, 即, 如果一个函数参数是指针, 那么它必须指向单个元素。如果希望传递数组, 应使用 `Array_ref`。

原因: 数组只能以指针方式传递, 而元素数目无法附着其上, 只能分开传递。而且, 隐式的数组到指针的转换和派生类到基类的转换会引起内存错误。

## 5. 类原则

**R500:** 对于没有共有数据成员的类, 用 `class` 声明。对没有私有数据成员的类, 用 `struct` 声明。不要定义既有共有数据成员, 又有私有数据成员的类。

原因: 清晰性。

**r501:** 如果类包含析构函数或者指针 / 引用类型的成员, 必须为其定义恰当的或禁止 (即, 不能使用默认版本) 拷贝构造函数和拷贝赋值运算符。

原因: 析构函数通常会释放资源。对于具有析构函数或者指针和引用类型的类, 默认拷贝语义几乎不可能“做正确的事”。

**R502:** 如果类包含虚函数, 那么它必须具有虚析构函数。

原因: 虚函数可以通过基类接口来使用, 通过基类接口访问对象的函数可能会删除对象, 派生类必须有某种机制 (析构函数) 来进行清理工作。

**R503:** 接受单一参数的构造函数必须显式声明。

原因: 避免奇怪的隐式类型转换。

## 6. 硬实时原则

**R800:** 不应使用异常。

原因: 异常不可预测。

**R801:** `new` 只能在初始化时使用。

原因: 不可预测。

例外: 可以用定址的 `new` 从栈中分配内存。

**R802:** 不应使用 `delete`。

原因: 不可预测, 可能会引起碎片问题。

R803: 不应使用 `dynamic_cast`。

原因: 不可预测 (假定是用普通方法实现的)。

R804: 不应使用标准库容器, `std::array` 除外。

原因: 不可预测 (假定是用普通方法实现的)。

## 7. 关键系统原则

R900: 递增和递减运算不能作为子表达式。

例子:

```
int x = v[++i]; // 违反
```

例子:

```
++i;  
int x = v[i]; // 正确
```

原因: 可能会被漏掉。

R901: 代码不应依赖于算术表达式优先级之下的优先级规则。

例子:

```
x = a*b+c; // 正确
```

例子:

```
if (a<b || c<=d) // 违反: 应加上括号 (a<b) 和 (c<=d)
```

原因: C/C++ 基础较差的程序员写出的代码中常常会有优先级混乱的情况。

上面列出的编码原则并未顺序编号, 这样可以随时加入新的原则, 而不必更改已有的编号, 也不会破坏分类。编码原则常常以编号被人熟知, 改变这些编号会受到用户的反对。

### 25.6.3 实际编码规范

C++ 编码规范已有很多, 大多数是公司所有, 并未广泛使用。在很多情况下, 这对程序员来说可能是好事, 也许只有这些公司的程序员除外。下面列出了一些对程序设计有帮助的编码规范, 当然前提是将它们应用在恰当的领域:

Google C++ Style Guide: <http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml>. 一个传统风格而且限制严格并且还在不断更新的编码规范。

Lockheed Martin Corporation. *Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program*. 文档编号 2RDU000001 Rev C. 2005 年 12 月. 俗称“JSF++”, 这是洛克希德-马丁航空公司为飞机软件所编制的规范。编制和使用这个规范的人, 是那些正在编写人类生活必不可少的软件的程序员。请参考 [www.stroustrup.com/JSF-AV-rules.pdf](http://www.stroustrup.com/JSF-AV-rules.pdf)。

Programming Research. *High-integrity C++ Coding Standard Manual 2.4 版*. 请参考 [www.programmingresearch.com](http://www.programmingresearch.com)。

Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guide-lines, and Best Practices*. Addison-Wesley, 2004. ISBN 0321113586. 本书很大程度上可以看作“元编



码规范”，即，它并不是定义特定的编码原则，而是为设计原则指出方向：什么是好的原则及为什么。

☛ 注意，并不是说阅读了以上书籍，你就不必再去了解实际的应用领域、程序设计语言以及相关的程序设计技术了。对于大多数应用领域，当然也包括嵌入式程序设计，你还是需要了解操作系统和硬件体系结构。如果你需要使用 C++ 进行低层程序设计，请查阅 ISO C++ 委员会关于性能的报告（ISO/IEC TR 18015，[www.stroustrup.com/performanceTR.pdf](http://www.stroustrup.com/performanceTR.pdf)）。提及“性能”，他们/我们主要是指“嵌入式程序设计”。

☛ 语言方言和专有语言在嵌入式领域是很常见的，但只要条件允许，请使用标准语言（如 ISO C++）、标准工具和标准库。这会使你的学习曲线更短，而且可能延长你的工作成果的寿命。

## 简单练习

1. 编译、运行下面的程序：

```
int v = 1; for (int i = 0; i < sizeof(v) * 8; ++i) { cout << v << ' '; v <<= 1; }
```

2. 将 `v` 改为 `unsigned int` 类型，重新编译、运行程序。
3. 使用十六进制常量定义 `short unsigned int` 变量，使其值：
  - a) 所有位都置位（置为 1）。
  - b) 最低有效位置位。
  - c) 最高有效位置位。
  - d) 最低字节所有位都置位。
  - e) 最高字节所有位都置位。
  - f) 每隔一位置位（最低有效位置位）。
  - g) 每隔一位置位（最低有效位置为 0）。
4. 将上题中每个数以十进制形式和十六进制形式输出。
5. 用位运算（`|`、`&`、`<<`），只用常量 1 和 0，重做第 3、4 题。

## 思考题

1. 什么是嵌入式系统？给出十个例子，至少有三个是本章未提及的。
2. 嵌入式系统的特殊之处在哪里？给出常见的五点。
3. 给出嵌入式系统中可预测性的定义。
4. 为什么嵌入式系统的维修很困难？
5. 为什么出于性能的考虑进行系统优化是个糟糕的主意？
6. 为什么我们更倾向于使用高层抽象而不是低层代码？
7. 什么是瞬时错误？为什么我们特别害怕这种错误？
8. 如何设计具有故障恢复能力的系统？
9. 为什么我们无法防止所有故障？
10. 什么是领域知识？给出一些应用领域的例子。
11. 为什么对于嵌入式系统程序设计来说领域知识是必要的？
12. 什么是子系统？给出一些例子。

13. 从 C++ 语言的角度, 存储可以分为哪三类?
14. 什么情况下你会使用动态内存分配?
15. 为什么在嵌入式系统中使用动态内存分配通常是不可行的?
16. 什么情况下在嵌入式系统中使用 `new` 是安全的?
17. 在嵌入式系统中使用 `std::vector` 的潜在问题是什么?
18. 在嵌入式系统中使用异常的潜在问题是什么?
19. 什么是递归函数调用? 为什么一些嵌入式系统程序员要避免它? 替代方法是什么?
20. 什么是内存碎片?
21. 什么是垃圾收集器 (在程序设计中)?
22. 什么是内存泄漏? 它为什么会发生?
23. 什么是资源? 请举例。
24. 什么是资源泄漏? 如何系统地预防?
25. 为什么不能将对象从一个内存位置简单地移动到另一个位置?
26. 什么是栈?
27. 什么是存储池?
28. 为什么栈和存储池不会导致内存碎片?
29. 为什么 `reinterpret_cast` 是必要的? 它又会导致什么问题?
30. 指针作为函数参数有什么危险? 请举例。
31. 指针和数组可能引起什么问题? 请举例。
32. 在函数接口中, 可以用什么机制替代 (指向数组的) 指针参数?
33. “计算机科学第一定律” 是什么?
34. 位是什么?
35. 字节是什么?
36. 通常一个字节有多少位?
37. 位运算有哪些?
38. 什么是“异或”运算? 它有什么用处?
39. 如何描述位序列?
40. 字中位的习惯编号次序是怎样的?
41. 字中字节的习惯编号次序是怎样的?
42. 什么是字?
43. 通常一个字中有多少位?
44. `0xf7` 的十进制值是多少?
45. `0xab` 的位序列是什么?
46. `bitset` 是什么? 你什么情况下需要使用它?
47. `unsigned int` 和 `signed int` 的区别是什么?
48. 什么时候使用 `unsigned int` 比 `signed int` 更好?
49. 如果需要处理的元素数目非常巨大, 如何设计一个循环?
50. 如果将 `-3` 赋予一个 `unsigned int` 变量, 它的值会是什么?
51. 我们为什么要直接处理位和字节 (而不是处理上层数据类型)?
52. 什么是位域?

53. 位域的用途是什么?
54. 加密是什么? 为什么要加密?
55. 能对照片进行加密吗?
56. TEA 表示什么?
57. 如何以十六进制输出一个数?
58. 编码规范的目的是什么? 列出几条需要编码规范的原因。
59. 为什么没有一个普适的编码规范?
60. 列出一些好的编码规范应该具备的特点。
61. 编码规范如何会起到不好的效果?
62. 列出至少十条你认可 (发现有用) 的编码规范。解释它们为什么有用。
63. 我们为什么要避免使用字母全部大写的标识符?

## 术语

address (地址)	encryption (加密)	pool (存储池)
bit (位)	exclusive or (异或)	predictability(可预测性)
bitfield (位域)	gadget (小设备)	real time (实时)
bitset	garbage collector (垃圾收集程序)	resource (资源)
coding standard (编码规范)	hard real time (硬实时)	soft real time (软实时)
embedded system (嵌入式系统)	leak (泄漏)	unsigned

## 习题

1. 如果你还没有做本章中的“试一试”练习, 现在做一下。
2. 为 0 ~ 9 的数字创建一个对应单词表, 使得所有十六进制数字都能用英文字母 (串) 表示。如, 用 o 表示 0, 用 l 表示 1, 用 to 表示 2, 等等。这样, 十六进制数能够拼成像单词的形式, 如 0xF001 拼写为 Fool, 0xBEEF 拼写为 Beef。在提交之前, 小心地去除单词表中的粗话。
3. 用以下位模式初始化一个 32 位有符号整数, 并打印出结果: 全 0、全 1、1 和 0 交替 (最高有效位为 1)、0 和 1 交替 (最高有效位为 0)、两个 1 和两个 0 交替 (110011001100…), 两个 0 和两个 1 交替 (001100110011…), 全 1 字节和全 0 字节交替 (最高字节为全 1)、全 0 字节和全 1 字节交替 (最高字节为全 0)。对 32 位无符号数重做本题。
4. 为第 7 章的计算器程序添加位运算 &、|、^ 和 ~。
5. 编写一个无限循环, 观察执行效果。
6. 编写一个不易察觉的无限循环。如果设计出的循环未能无限执行下去只是因为耗尽了某种系统资源, 也可认为达到了本题的要求。
7. 输出 0 ~ 400 这些数值的十六进制形式, 输出 -200 到 200 这些数值的十六进制形式。
8. 输出你的键盘上的每个字符的数值。
9. 不使用任何标准头文件 (如 <limits>), 也不借助任何文档, 计算在你的系统中, 一个 int 包含多少位, 并确定 char 是有符号的还是无符号的。
10. 仔细分析 25.5.5 节中关于位域的例子程序。编写程序, 初始化一个 PPN, 然后读取并输出每个位域的值; 接着改变每个位域的值 (可以向每个位域赋值) 并输出结果。改用一

个 32 位无符号数存储 PPN，并使用位运算（见 25.5.4 节）访问每个域，重做此题。

11. 重做上题，将二进制位保存在 `bitset<32>` 中。
12. 对 25.5.6 节中的例子，解密密文，输出明文。
13. 利用 TEA（见 25.5.6 节）实现两台计算机之间的“保密”通信。最低限度要实现安全的 Email。
14. 实现一个简单 `vector`，能保存至多  $N$  个元素，存储空间从一个存储池中分配。测试  $N=1000$ ，元素类型为整数的情况。
15. 测试用 `new` 分配 10 000 个对象所花费的时间（见 26.6.1 节），对象大小为 1 字节到 1000 字节之间的随机值，然后测试用 `delete` 释放这些对象所花费的时间。测试两次，第一次按分配的逆序进行释放，第二次按随机顺序进行释放。然后，测试从存储池中分配 10 000 个大小固定为 500 字节的对象的时间和释放它们的时间。接着测试从栈中分配 10 000 个大小为 1 字节到 1000 字节之间随机数的对象的时间和逆序释放它们的时间。比较测试结果。每个测试至少重复 3 次以确保结果是一致的。
16. 给出 20 条编码风格方面的原则（不要简单地拷贝 25.6 节中的内容）。将这些原则应用到你最近编写的一个 300 行以上的程序中。每应用一条原则，就为其编写一个简短（一或两行）的注释。在这个过程中，你是否发现代码中有错误？代码是否变得更清晰了？还是有些部分更不清晰了？根据这些结果修改编码原则。
17. 在 25.4.3 和 25.4.4 节中我们提供了一个 `Array_ref` 类，宣称能以简单、安全的方式访问数组元素。特别是我们宣称它能正确处理继承。尝试用 `Array_ref<Shape*>` 以不同方式将一个 `Rectangle*` 放入 `vector<Circle*>` 中，不引起任何类型转换或其他行为不确定的操作。这应该是不可能办到的。

## 附言

那么，嵌入式程序设计基本上就是“摆弄位”吗？完全不是这样，特别是当你有意减少位运算，以免它成为影响正确性的潜在问题时。但是，在系统某些地方，我们不得不处理位和字节，问题只是在哪里使用以及如何使用而已。在大多数系统中，低层代码可以也应该局部化，不应使位运算遍布整个程序。我们接触过的最有意思的系统中有许多都是嵌入式系统，而一些最有意思、最有挑战性的程序设计工作也是属于这个领域。

# 测 试

我只证明代码的正确性，不做测试。

——Donald Knuth

本章介绍正确性相关的测试及设计技术。这是一个非常庞大的主题，我们在这一章中只能窥其冰山一角。本章重点介绍一些单元测试的思想和技术，所谓单元测试，就是针对函数和类等程序单元进行测试。我们会讨论如何使用接口，以及如何选择测试。我们将着重介绍通过系统设计来简化测试工作，及在软件开发的早期就开展测试工作的重要性。我们还会简单介绍程序的正确性和处理性能问题方面的内容。

## 26.1 我们想要什么

让我们做一个简单的实验：写一个二分搜索程序。现在就写，不要等到本章的结束，不要等到下一节。亲自动手是非常重要的，就是现在！二分搜索是一种用于有序序列搜索的方法，开始时先访问序列中间元素：

- 如果中间元素等于我们要搜索的元素，结束搜索。
- 如果中间元素小于我们要搜索的元素，我们继续使用二分方法搜索右半部分。
- 如果中间元素大于我们要搜索的元素，我们继续使用二分方法搜索左半部分。
- 结果显示了搜索是否成功，这里可以使用一些允许我们修改元素的工具，例如，下标、指针或迭代因子。

可以使用小于运算 (<) 进行比较（排序）操作。按照你的喜好，可以使用任何一种数据结构，任何一种函数调用规范，以及任何一种返回结果的方法，但是一定要亲手编写这个程序。在本例中，使用他人的代码是会起反作用的，即使你列明了出处。特别要注意的是，不要使用标准库算法（`binary_search` 或 `equal_range`）。虽然在大多数情况下，它是你的首选。多花点时间在这上面吧。

现在你已经完成了你自己的二分搜索函数了。如果没有，请返回上一段。如何确定你的搜索函数是正确的呢？如果还无法确定的话，你可以先写下你认为代码正确的理由。如何相信你的理由呢？是否有部分参数可能会有问题？

✘ 这是一段非常简单的代码，它实现了一个很常规但也最经典的算法。你的编译器用了 20 万行代码，你的操作系统包含了 1000 万到 5000 万行代码，你下一次度假或开会时搭乘的飞机上的安全系统包含了 50 万到 200 万行代码。这会让你感到舒服一点么？你在二分搜索函数中使用的技术是如何应用到这类大型实际软件中去的呢？

令人好奇的是，虽然包含如此复杂的代码，但大多数大型软件绝大部分时间都能正常工作。当然，我们也不会把以游戏为主的消费型 PC 上运行的软件当作“关键系统”。对我们来说更为重要的是，对安全性要求严格的软件几乎需要在任何时刻都能正常运行。我们想不起过去十年中发生过因软件问题造成飞机或汽车事故的案例。至于金额为 0.00 美元的支票

导致银行软件严重混乱的故事，也已经非常久远了，这些问题基本上不会再发生了。但是，软件毕竟是像你我这样的人编写的。你很清楚自己是会犯错的，事实上我们都会犯错误。那么如何让软件不会出错呢？

最基本的答案是：“我们”已经知道如何用不可靠的组件构建可靠的系统。我们尽力让每一个程序，每一个类，每一个函数都正确，但在最初总是会出错的。于是我们调试、测试、重新设计程序，找出并排除尽可能多的错误。然而，在任何复杂系统中，还是会有错误隐藏其中。我们知道错误存在，但我们无法找到它们。或者说，在有限的时间内，在可以投入的人力、物力条件下，我们无法找到这些错误。于是，我们继续重新设计系统，来解决这些意想不到的“不可能的”错误。最终得到的可能是一个非常可靠的系统。但请注意，即使是这些非常可靠的系统，其中也隐藏着错误，只不过大多数情况下系统都能工作正常，只是偶尔运行状况不如我们预期。但是，这类系统不会崩溃，并且提供的服务总能满足最低需求。例如，在通话请求异常高的时候，电话系统可能不能满足所有通话请求，但它仍然可以提供许多连接服务。

现在，我们可以上升到哲学层面，讨论一下我们所猜测和顾忌的意外错误是否是真正的错误，但我们先不考虑这个问题。对系统程序员而言，弄明白如何使我们的系统更加稳定才是更有助于生产的。

### 26.1.1 警告

测试是一个庞大的主题。很多人都在研究测试应该怎样做，不同的工业和应用领域有着不同的习惯和标准。这很自然，对于视频游戏软件和航空系统软件，你所需要的可靠性标准显然是不一样的。但这种情况会导致术语和工具的混乱。本章介绍的内容可以作为你个人项目进行测试的思想源泉，如果你参与大型系统的测试，也可从本章汲取思想。大型系统的测试包括了各种测试工具和组织结构的组合，这些内容不在本章的讨论范围之内。

## 26.2 程序正确性证明

等一下！为什么我们要为测试问题忙得团团转，为什么不能直接证明程序的正确性呢？正如 Edsger Dijkstra 一针见血地指出的那样，“测试只能揭示错误的存在，而不能证明不存在错误”。这引出了对程序正确性证明的明显需求：“像数学家证明数学定理那样。”

不幸的是，证明一个普通程序的正确性是现有研究所不能解决的（一些非常特殊的应用领域除外），而且证明本身也会包含错误（就像数学家也会出错一样），整个程序证明领域也是一个非常艰深的研究方向。因此，我们要尽可能使程序更加结构化，以便能对其进行推理，使我们能确信它是正确的。但是，我们还是要进行测试（参见 26.3 节），并更好地组织代码，使之具备错误恢复能力，能应对剩余未发现的错误（参见 26.4 节）。

## 26.3 测试

在 5.11 节中，我们说过“测试是一种系统地查找错误的方法”。下面，让我们来看一下相关的技术。

一般来说，人们把测试分为单元测试（unit testing）和系统测试（system testing）两种。“单元”是指函数和类等程序的组成部分。当我们对单元进行独立测试的时候，一旦错误发生，我们就能确定在哪里寻找错误——错误一定是在我们所测试的单元中（或者用来引导测

试的代码)。与之相对，系统测试只能告诉我们系统中存在错误。一种典型情况是，当我们完成单元测试后，在系统测试中发现了错误，那么可能是单元之间的交互出现了问题。与独立单元内的错误相比，这类错误更难查找，也更难修复。

显然，一个单元（例如一个类）可以包含多个小单元（例如函数或其他类）。一个系统（例如电子商务系统）也可以包含多个子系统（例如数据库、GUI、网络系统、订单确认系统）。因此，单元测试和系统测试的区别不像你想象得那么清晰，但有一个一般性的策略：做好自己编写的单元的测试，这样既节省了自己的时间，也减少了最终用户的烦恼。

关于测试的一种看法是：任何复杂系统都是由许多单元组成的，这些单元又是由更小的单元组成的。因此，我们从最小的单元开始测试，然后再依次测试更大的单元，直到整个系统测试完毕为止。也就是说，“系统”是最大的单元（除非我们用它来构建更大的系统）。

因此，我们首先考虑的是如何测试一个单元（例如函数、类、嵌套类或者模板）。测试有白盒测试（你可以看到被测单元的实现细节）和黑盒测试（你只能看到被测单元的接口）之分。我们不会花很多精力区分这两种方式，你应该尽一切办法了解被测单元的细节。但要注意的，可能有人随后修改被测单元，因此，不要依赖任何在单元接口中无法确定的信息。实际上，测试的基本思想是向接口发送被测单元可以接受的任何输入，然后观察其反应是否正确。

✂ 需要注意的是，因为被测单元的代码可能会被修改（你或其他人），这就需要进行回归测试。基本上，只要你修改了代码，就必须重新进行测试，以保证你的修改没有造成破坏。因此，在代码升级后，必须重新进行单元测试，在整个系统提交前（或者你自己使用前），也要重新进行完整的系统测试。

这种对系统的完整测试通常被称为回归测试（regression testing），因为这种测试通常包括对以前发现的错误的再次检查，以避免代码修改将错误重新引入。如果旧的错误仍然存在，系统就“回归”了，需要再次修正错误。

### 26.3.1 回归测试

👉 收集在过去测试中有助于发现错误的测试用例，是为系统建立有效测试集的主要工作。如果有用户在使用系统的话，他们会将错误信息发送给你。千万不要将这些错误报告丢弃，要用错误跟踪系统来确保这一点。因为，一个错误报告表明：或者存在一个系统错误，或者存在一个用户使用问题，两者都是有用的。

通常，错误报告会包括许多无关的信息，我们的第一项任务就是将所报告的问题局限在程序的最小范围内。这经常要去掉大部分提交的代码：特别是，我们会尝试去掉库的使用和不会导致错误的应用程序代码。找出最小被测对象通常有助于我们在系统代码中确定错误区域，这一最小化的程序将被提交做回归测试。找出最小测试对象的方法是不断去除无关代码直到错误消失为止，然后将最后一次去除的代码重新加入。不断持续这一过程直到候选代码可删为止。

仅仅运行上百个（或上万个）来自错误报告的测试用例可能看起来不那么具有系统性，但在这个过程中我们真正所做的是系统地利用用户和开发人员的经验。回归测试用例集就体现了开发者的群体记忆。对一个大系统来说，我们不能简单地依靠开发人员来了解设计和实现的细节。回归测试用例集就可以确保系统的变化不会脱开发者和用户所认可的范围。

## 26.3.2 单元测试

好吧，已经说得够多了！让我们来尝试一个实际的例子吧：测试一个二分搜索。下面的描述来自 ISO 标准（见 25.3.3.4 节）：

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value);
```

```
template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value, Compare comp);
```

**要求：** [first, last) 之间所有元素  $e$  按表达式  $e < \text{value}$  和  $!(\text{value} < e)$  或  $\text{comp}(e, \text{value})$  和  $!\text{comp}(\text{value}, e)$  划分。并且，对 [first, last) 之间所有元素  $e$  来说， $e < \text{value}$  意味着  $!(\text{value} < e)$ ， $\text{comp}(e, \text{value})$  意味着  $!\text{comp}(\text{value}, e)$ 。

**返回：** 如果位于区间 [first, last) 内的一个迭代器  $i$  满足如下条件： $!(*i < \text{value}) \&\& !(\text{value} < *i)$  或  $\text{comp}(*i, \text{value}) == \text{false} \&\& \text{comp}(\text{value}, *i) == \text{false}$ ，函数返回真。

**复杂度：** 最多  $\log(\text{last} - \text{first}) + 2$  次比较。

没有经验的人很难读懂上述形式化定义（好吧，只是半形式化的）。但是如果你已经真正完成了本章开头我们所强烈建议的二分搜索编程练习的话，你就会对如何实现二分搜索并测试它有一些很好的想法。这个（标准）版本接受三个参数：两个前向迭代器（见 15.10.1 节）和一个数值。如果数值出现在两个迭代器限定的范围内，函数就返回真。两个迭代器必须对应一个有序序列。比较（排序）操作通过运算符  $<$  完成。我们可以为 `binary_search` 增加一个参数——比较操作函数，这样就可以用用户指定的任意比较操作进行二分搜索了，我们将此作为练习。

在这里，我们只处理编译器不能发现的错误。因此，类似下面的问题不再考虑：

```
binary_search(1,4,5);           // 错误：整型数值不能作为迭代器
vector<int> v(10);
binary_search(v.begin(),v.end(),"7"); // 错误：不能在整型向量中搜索字符串
binary_search(v.begin(),v.end());   // 错误：丢失参数
```

我们应该如何系统地（systematically）测试 `binary_search()` 呢？显然，我们不可能测试所有可能的参数。因为可能的参数值和类型的组合的数目会是一个非常巨大的数字！因此，我们必须精挑细选测试用例。我们需要一些选择标准：

- 很可能导致错误的测试（找出大多数错误）。
- 可能导致严重错误的测试（找出可能导致最坏结果的错误）。

这里的“严重错误”是指那些会导致最坏结果的错误。通常，这是一个模糊的概念。但对特定程序来说，就可能非常明确。例如，孤立考虑二分搜索的话，所有错误的严重程度都差不多。但是，如果 `binary_search` 是用于一个大程序，而该程序会将所有计算结果都仔细检查两遍的话，“返回一个错误结果”要比“陷入死循环什么也不返回”好得多。当查找这类错误时，将 `binary_search` “骗”入一个死循环（或者是非常长的循环），要比“骗”它返回一个错误结果花费多得多的力气。注意，我们使用了“欺骗”一词。与其他工作相比，测试就是一种发挥我们的创造性思维，来达到“如何让这个程序运行不正常”目的的活动。因此，最好的测试人员不但要有系统性思维，还要非常“狡猾”（当然，要有正当的理由）。



### 26.3.2.1 测试策略

我们应该如何破坏 `binary_search` 的正常运行呢？首先，我们要检查 `binary_search` 的要求，即，它对于输入的假设是怎样的。不幸的是，从测试者的观点来看，`binary_search` 明确声明 `[first,last)` 必须是有序序列，也就是说，确保这一点是调用者的责任。因此，我们不能通过输入无序序列，或者 `last<first` 来破坏 `binary_search`，那是不公平的。注意，在 `binary_search` 的要求中并没有说明，如果我们给定的输入不满足条件的话，它会如何做。在 ISO 标准的其他地方，`binary_search` 声明：对于不满足要求的输入，它可能会抛出一个异常，但这不是强制的。我们要记住这些事实，当测试其他程序对 `binary_search` 的使用方式时，这些事实是很有用的：调用者给出不满足函数要求的输入，显然可能成为错误之源。

我们设想 `binary_search` 可能出现下列错误：

- 不返回（例如无限循环）。
- 崩溃（例如错误的引用，无限递归）。
- 未找到值，即使该值确实在序列中。
- 找到了值，但该值并不在序列中。

此外，我们还要记住下列序列会给用户错误“可乘之机”：

- 序列未排序（例如，`{2,1,5,-7,2,10}`）。
- 序列不合法（例如，`binary_search(&a[100], &a[50], 77)`）。

我们只不过是简单调用 `binary_search(p1, p2, v)` 而已，为什么函数还会出现错误呢（测试者发现的错误）？一般来说，通常是“特殊情况”导致错误发生。特别是，当测试对象是处理序列的程序时，我们可以从序列开始和末尾入手构造“特殊序列”。另外，我们总是应该对空序列进行测试。让我们先来看一些有序的整型数组：

```
{ 1,2,3,5,8,13,21 } // 一个“普通序列”
{ } // 空序列
{ 1 } // 仅仅一个元素
{ 1,2,3,4 } // 偶数个元素
{ 1,2,3,4,5 } // 奇数个元素
{ 1, 1, 1, 1, 1, 1 } // 所有元素相同
{ 0,1,1,1,1,1,1,1,1,1 } // 差异元素在开始
{ 0,0,0,0,0,0,0,0,0,0,1 } // 差异元素在末尾
```

也可以用程序来生成一些测试序列：

- `vector<int> v1;`  
`for (int i=0; i<100000000; ++i) v.push_back(i);` // 一个非常大的序列
- 一些元素个数随机的序列。
- 一些由随机数组成的序列（仍是有序的序列）。

这不是我们预期的那种系统测试。毕竟，我们只是“挑拣”了一些序列。但是，这里我们用到了处理数据集时很有用的一般性原则，包括：

- 空集。
- 小数据集。
- 大数据集。
- 极限分布的数据集。
- 序列末尾处可能发生问题的数据集。

- 包含重复元素的数据集。
- 包含奇数和偶数个元素的数据集。
- 由随机数组成的数据集。

使用随机数序列的目的是看看是否能幸运地发现一些我们没有考虑到的错误。这是一种蛮力技术，但是能节省我们的时间。

为什么要使用“奇数 / 偶数”个元素的序列呢？这是因为很多算法都会用划分的方法把输入序列分为两部分，而程序员可能只考虑了奇数情况或偶数情况。更一般的问题是，当我们划分一个序列的时候，划分点会成为一个子序列的末尾。正如我们所知，序列的末尾往往是错误容易发生的地方。

一般来说，在设计测试用例时我们应该重点考虑：

- 极限情况（大的、小的、奇异分布的输入，等）。
- 边界条件（边界附近的任何情况）。

“极限情况”、“边界条件”具体是什么含义，取决于我们所测试的实际程序。

### 26.3.2.2 一个简单的测试

我们可以进行两类测试：应该搜索成功的测试（例如，搜索在序列确实存在的值）；应该搜索失败的测试（例如，搜索一个空集）。对每一组输入序列，我们都会构造应该成功和应该失败的测试用例。我们将从最简单和最明显的用例开始，然后逐步改进，直至找到对 `binary_search` 来说足够好的测试用例：

```
vector<int> v { 1,2,3,5,8,13,21 };
if (binary_search(v.begin(),v.end(),1) == false) cout << "failed";
if (binary_search(v.begin(),v.end(),5) == false) cout << "failed";
if (binary_search(v.begin(),v.end(),8) == false) cout << "failed";
if (binary_search(v.begin(),v.end(),21) == false) cout << "failed";
if (binary_search(v.begin(),v.end(),-7) == true) cout << "failed";
if (binary_search(v.begin(),v.end(),4) == true) cout << "failed";
if (binary_search(v.begin(),v.end(),22) == true) cout << "failed";
```

这个测试虽然有些重复和繁琐，但却是一个不错的开端。实际上，许多简单的测试集与这个例子一样，就是一个长长的调用序列。这种方法最大的优点就是简单。即使是一个新手，也能够测试集中加入新的测试用例。但是，我们通常还可以做得更好。例如，当某个测试失败时，这个程序没有告诉我们哪个测试用例失败了。这是不可接受的。而且，写测试代码不应该是“剪切和粘贴”式编程，我们应该像写其他代码一样编写测试程序。改进如下：

```
vector<int> v { 1,2,3,5,8,13,21 };
for (int x : {1,5,8,21,-7,2,44})
    if (binary_search(v.begin(),v.end(),x) == false) cout << x << " failed";
```

假定最终我们会进行数十个测试，有没有这种改进就会大不一样了。对于真实系统，我们经常要进行几千个测试。因此，准确定位哪个测试出错是非常必要的。

在继续讨论之前，请注意上面例子中所体现出的（半系统化）测试技术：我们从序列的末尾和“中部”取一些值作为要搜索的值，这些测试用例应导致搜索成功。我们当然可以将该序列的所有值逐一作为输入，但这显然是不现实的。对于导致搜索失败的测试用例，我们从序列两端和中部各选择一个值（不在序列中的值）。当然，这也不是一种系统的测试，但这种测试用例构造模式是一种非常常用的技术，它在测试数值序列或数值范围类程序时非常有用。

这些初步的测试有什么问题么？

- 我们（最初时）重复编写相同的代码。
- 我们（最初时）手工设定测试编号。
- 输出信息很少（用处也不大）。

经过仔细考虑后，我们决定把测试用例保存在文件中。每个测试都包含一个唯一的标签、一个要搜索的值、数值序列以及期望的计算结果。例如，

```
{ 27 7 { 1 2 3 5 8 13 21 } 0 }
```

这个测试的编号是 27。它在序列 {1 2 3 5 8 13 21} 中查找 7，期望运行结果是 0（即失败）。我们为什么要把测试用例保存文件中，而不是把它们硬编码到程序中呢？是的，对于这个例子来说，我们可以直接输入测试用例，放在程序中。但是，在程序中放入大量数据，无疑会使程序变得杂乱无章。而且，我们经常会用程序生成测试用例，而不是手工编写。计算机生成的测试用例一般都保存在数据文件中，无法直接放在程序之中。现在，我们可以编写一个使用各种不同测试用例文件的测试程序了：

```
struct Test {
    string label;
    int val;
    vector<int> seq;
    bool res;
};

istream& operator>>(istream& is, Test& t); // 使用定义的格式

int test_all(istream& is)
{
    int error_count = 0;
    for (Test t; is>>t; ) {
        bool r = binary_search(t.seq.begin(), t.seq.end(), t.val);
        if (r != t.res) {
            cout << "failure: test " << t.label
                 << " binary_search: "
                 << t.seq.size() << " elements, val==" << t.val
                 << " ->" << t.res << "\n";
            ++error_count;
        }
    }
    return error_count;
}

int main()
{
    int errors = test_all(ifstream("my_tests.txt"));
    cout << "number of errors: " << errors << "\n";
}
```

下面是我们所使用的部分输入序列：

```
{ 1.1 1 { 1 2 3 5 8 13 21 } 1 }
{ 1.2 5 { 1 2 3 5 8 13 21 } 1 }
{ 1.3 8 { 1 2 3 5 8 13 21 } 1 }
{ 1.4 21 { 1 2 3 5 8 13 21 } 1 }
{ 1.5 -7 { 1 2 3 5 8 13 21 } 0 }
{ 1.6 4 { 1 2 3 5 8 13 21 } 0 }
```

```
{1.722{123581321}0}
{21{}0}
{3.11{1}1}
{3.20{1}0}
{3.32{1}0}
```

在这里可以看出我们将标签定义为字符串类型而不是数值类型的原因：可以更灵活地为测试“编号”——本例中使用了带小数的标签，来区分同一个序列之上的不同测试。我们还可以使用更复杂的格式，来避免在测试数据文件中反复给出同一个序列。

### 26.3.2.3 随机序列

在选择测试数据的时候，我们会尽力击败程序编写者（常常就是我们自己！），会重点在那些可能隐藏着错误的区域选择数据（例如，复杂的条件序列、序列末尾处、循环等等）。但是，由于通常我们就是程序的编写者，我们在编写和调试代码时已经考虑过这些因素了。因此，我们在设计测试方案时很可能重复编写程序时所犯的逻辑错误，这导致一些重要问题被忽略。这也是为什么要让与开发人员无关的另一些人参与到测试方案设计中的原因之一。有一种技术有时会对解决这一问题有所帮助：简单地生成（许多）随机值。下面这个函数使用 `randint()`（参见 24.7 节和 `std_lib-facilities.h`）生成一个二分搜索测试用例，并输出到 `cout`：

```
void make_test(const string& lab, int n, int base, int spread)
// 向 cout 输出一个测试描述，标签为 lab
// 以 base 为起始值，随机生成 n 个元素的序列
// 元素之间的平均距离是 [0:spread] 之间的均匀分布
{
    cout << "{" << lab << " " << n << " {";
    vector<int> v;
    int elem = base;
    for (int i = 0; i < n; ++i) { // 生成元素
        elem += randint(spread);
        v.push_back(elem);
    }

    int val = base + randint(elem - base); // 搜索
    bool found = false;
    for (int i = 0; i < n; ++i) { // 输出元素并查看是否找到相应值
        if (v[i] == val) found = true;
        cout << v[i] << " ";
    }
    cout << "}" << found << " }\n";
}
```

请注意我们没有用 `binary_search` 来检验随机数 `val` 是否在随机序列中。我们不能用待测程序来为测试用例确定正确的值。

实际上，`binary_search` 并不特别适合用随机数序列进行蛮力测试。虽然我们对这些测试用例能否发现我们“手工打造”的测试用例未发现的错误持怀疑态度，但这些技术通常还是很管用的。不管怎么样，让我们先动手构造一些基于随机数的测试：

```
int no_of_tests = randint(100); // 做 50 次测试
for (int i = 0; i < no_of_tests; ++i) {
    string lab = "rand_test_";
    make_test(lab + to_string(i), // 使用来自 23.2 节的 to_string
```

```

        randint(500),           // 元素个数
        0,                     // base 值
        randint(50));         // spread 值
    }

```

如果我们需要测试很多操作的累积效应，而一个操作的结果取决于之前的操作是如何处理的话，即系统是有状态的（见 5.2 节），系统运行就是状态间的迁移。对于这种情况，基于随机数的测试用例特别有用。

基于随机数的测试用例对 `binary_search` 效果不是很明显，原因在于，对于同一个序列，不同搜索之间都是独立的。当然，这个结论的前提是假定 `binary_search` 的实现中没有犯致命的愚蠢错误，例如，对序列进行了修改。对此，我们可以设计一个更好的测试用例（见习题 5）。

### 26.3.3 算法和非算法

上文以 `binary_search()` 为例介绍了一些简单的测试技术，`binary_search()` 是一个恰当的算法，它具有下列特点：



- 对输入数据有明确的要求。
- 明确描述了算法运行后对输入数据有什么影响（在本例中，没有影响）。
- 算法不依赖于显式输入之外的数据。
- 对于外部环境没有严格限制（例如，没有特殊的时间、空间和资源共享要求）。

它还包含了明显的前置和后置条件（见 5.10 节）。换句话说，它是测试人员梦寐以求的。但是，我们不可能总是这么幸运：很多时候我们不得不测试用糟糕的英语和很多图表表示的混乱代码（这还是乐观估计）。

等一下！我们是否对混乱的程序逻辑有些放任了呢？在不能准确描述代码要做什么的情况下，我们如何能够讨论正确性和测试呢？但问题是，在软件开发、测试中，很多内容都很难用非常清晰的数学形式来描述。而且，很多时候虽然在理论上能够给出严谨的数学描述，但所需数学知识超出了编写和测试代码的程序员的能力。因此，在现实世界的实际条件以及时间的双重压力下，我们只好放弃准确描述被测程序的理想目标，而要面对现实：只要被测程序在（测试人员，很多时候就是我们自己）掌握之中就可以了。

假设你要测试一个混乱的函数代码，这里“混乱”是指：

- 输入：它对输入（隐式或显式的）的要求没有像我们希望的那样进行准确定义。
- 输出：它对输出（隐式或显式的）的要求没有像我们希望的那样进行准确定义。
- 资源：它所使用的资源（时间、内存、文件等）没有像我们希望的那样准确定义。

这里的“隐式或显式”表示我们不但要检查正式的参数和返回值，还要检查函数对全局变量、`iostream`、文件、空闲内存空间的分配等的影响。那么，我们要怎么做呢？首先，这类函数一般都很长，或者我们不能把它的需求和影响描述清楚。也许我们讨论的是一个有 5 页长的函数，或者它使用“辅助函数”的方式复杂、烦琐。你可能会认为 5 页很长了。是的，这确实很长，但我们还会遇到更长的函数。而且不幸的是，这种事情经常会发生。



如果这是我们写的代码并且我们有时间的话，我们首先要做的是把这些“混乱的函数”分割成为许多小函数。每个小函数都符合我们理想中的严格定义函数，然后首先测试这些小函数。但是，我们的目标是测试软件——即系统地找出尽可能多的错误——而不是修正找到的错误。

那么，我们要找什么呢？作为测试人员，我们的任务是找出错误。错误可能隐藏在哪里呢？包含错误的代码有什么特点呢？

- 与“其他代码”微妙的相关性：因此我们可以检查全局变量、非常量引用参数、指针等的使用。
- 资源管理：查找内存管理（new 和 delete 操作）、文件使用、锁等。
- 查找循环：检查终止条件（例如 `binary_search()`）。
- if 和 switch 语句（也称为“分支语句”）：查找这些程序中的逻辑错误。

让我们逐个举例说明。

### 26.3.3.1 相关性

考虑下面这个无意义的函数：

```
int do_dependent(int a, int& b)    // 混乱的函数
    // 不可控的相关性
{
    int val;
    cin>>val;
    vec[val] += 10;
    cout << a;
    b++;
    return b;
}
```

在测试 `do_dependent()` 的时候，我们不能仅仅检查参数合法性，以及函数对参数做了什么运算。我们还要考虑函数所使用的全局变量 `cin`、`cout` 和 `vec`。在这个小函数中，对这些全局变量的使用方式很容易看清，但在实际程序中，这些细节往往会隐藏在大量代码中间。幸运的是，一些软件可以帮助我们找出这种相关性。不幸的是，这一办法并不总是可行，也很难推广。假定没有分析软件能帮助我们，我们只能逐行检查代码，找出其中所有的相关性。

在测试 `do_dependent()` 的时候，我们需要考虑：

- 它的输入：
  - a 的值。
  - b 的值以及 b 所引用的整型值。
  - cin 的输入值（存入 `val`）和 cin 的状态。
  - cout 的状态。
  - vec 的值以及 `vec[val]` 的值。
- 它的输出：
  - 返回值。
  - b 所引用的整型值（我们对它做了增量操作）。
  - cin 的状态（包括流状态和格式状态）。
  - cout 的状态（包括流状态和格式状态）。
  - vec 的状态（我们对 `vec[val]` 做了赋值操作）。
  - 任何 vec 可能抛出的异常（`vec[val]` 可能越界）。

这是一个很长的列表，实际上，它比函数本身都要长。这也可以解释为什么我们不喜欢全局变量，并且非常关注非常量引用（以及指针）。最理想的情况莫过于一个函数仅读入它的参数，计算结果只以返回值的形式给出：这样我们就能很容易地理解并测试这样的函数。

一旦确定了输入和输出，我们就可以回过头来看看 `binary_search()` 的例子。我们测试的方式是给出输入值（隐式或显式的输入），检查函数是否输出期望的结果（隐式或显式的）。对于 `do_dependent()`，我们需要从一个非常大的 `val` 值和负的 `val` 值开始，来看看它会输出什么。而且，看上去 `vec` 最好具备边界检查机制（否则我们可以很容易地构造出非常严重的错误）。当然，我们还要按照文档的说明检查所有的输入和输出。但对于这种混乱的函数来说，我们不要期望它会有清晰、完整和准确的说明。因此，我们只需考虑如何击破这个函数（即找到错误），然后开始询问什么是正确的。通常情况下，这样的测试和询问会导致代码的重新设计。

### 26.3.3.2 资源管理

考虑下面这个无意义的函数：

```
void do_resources1(int a, int b, const char* s) // 混乱的函数
    // 不可控的相关性
{
    FILE* f = fopen(s, "r");           // 打开文件 (C 风格)
    int* p = new int[a];               // 分配内存
    if (b <= 0) throw Bad_arg();       // 可能抛出异常
    int* q = new int[b];               // 分配更多内存
    delete[] p;                        // 释放由指针 p 指向的内存
}
```

为测试 `do_resource1()`，我们必须考虑每个申请到的资源是否都被妥善处理了，即，是否每一个资源都被释放或者转交给了其他函数了。

在本例中，显然存在这些问题：

- 名为 `s` 的文件没有关闭。
- 如果 `b <= 0` 或者第二个 `new` 发生异常的话，指针 `p` 指向的内存会发生泄漏。
- 如果 `0 < b` 的话，指针 `q` 指向的内存会发生泄漏。

此外，我们还要考虑打开文件操作可能会失败。为了显示这种糟糕的结果，我们故意使用了一种非常古老的编程风格（`fopen()` 是 C 语言中的打开文件的标准方法）。为了使测试人员的工作更为简单，我们可以将代码改写如下：

```
void do_resources2(int a, int b, const char* s) // 混乱较少的函数
{
    ifstream is(s);                    // 打开文件
    vector<int> v1(a);                  // 创建向量 (本身拥有内存空间)
    if (b <= 0) throw Bad_arg();       // 可能抛出异常
    vector<int> v2(b);                  // 创建另一向量 (本身拥有内存空间)
}
```

现在每一块内存空间都被一个能够自己释放内存的对象所拥有。考虑如何才能写出更简洁（更清晰）的函数，有时是思考测试方法的很好途径。14.5.2 节中的“资源分配即初始化”（RAII）技术提供了一种解决资源管理问题的一般策略。

⚠ 请注意，资源管理不仅仅是检查每一块内存是否被释放。有时，我们会从其他地方接收到资源（例如，作为参数），有时我们也会将资源传出函数（例如，作为返回值）。在这种情况下，很难确定什么是正确的。考虑下面的函数：

```
FILE* do_resources3(int a, int* p, const char* s) // 混乱的函数
    // 不可控的资源传递
{
    FILE* f = fopen(s, "r");
```

```

delete p;
delete var;
var = new int[27];
return f;
}

```

`do_resource3()` 将打开的文件（推测已打开）作为返回值是否正确呢？将通过参数 `p` 传递给它的内存释放是否正确呢？此外，我们还偷偷改变了全局变量 `var`（显然它是一个指针）。基本上，将资源传进 / 传出是很常见也很有用的，但要知道资源传递是否正确，还要掌握一些资源管理策略方面的知识。谁拥有这些资源？谁将删除 / 释放这些资源？程序文档应该清楚、简洁地回答这些问题（通常只是我们的美好愿望）。不管怎样，资源的传递都是孕育错误的温床，当然这也是测试的重点之一。

需要注意的是，在上面的例子中，我们是如何通过使用全局变量（故意地）使资源管理复杂化的。当我们把各种可能的错误来源混杂在一起的时候，一切都会变得糟糕。作为程序员，我们要避免这一点。作为测试人员，我们要重点检查这种情况。 ⚠

### 26.3.3.3 循环

当我们讨论 `binary_search()` 的时候，对循环结构进行了检查。基本上，大部分错误都是在循环结构中发生的：

- 在开始循环的时候，是否所有数据都正确地初始化了？
- 循环是否正确地终止了呢（经常是在最后一个元素出问题）？

下面是一个循环结构出错的例子：

```

int do_loop(const vector<int>& v)    // 混乱的函数
    // 不可控的循环
{
    int i;
    int sum;
    while(i<=vec.size()) sum+=v[i];
    return sum;
}

```

这个例子有三处明显的错误（哪三处？）。此外，好的测试人员应该马上意识到对 `sum` 的加法运算可能会导致溢出问题。

- 很多循环都包含数据处理，因此当有大量输入数据时，可能会产生某种溢出错误。 ⚠
- 一个臭名昭著的循环错误是缓冲区溢出，我们可以通过系统地询问两个关于循环的关键问题，来捕获这类错误：

```

char buf[MAX];    // 固定大小的缓冲区

char* read_line() // 危险的做法
{
    int i = 0;
    char ch;
    while(cin.get(ch) && ch!='\n') buf[i++] = ch;
    buf[i+1] = 0;
    return buf;
}

```

当然，你不会如此编写代码！（为什么不？`read_line()` 有什么问题吗？）但糟糕的是，这种代码编写方法很常见，而且还有许多变化形式，例如：



```
// 危险的做法
gets(buf);           // 读一行到缓冲区
scanf("%s",buf);    // 读一行到缓冲区
```

⚠ 请在文档中查阅 `gets()` 和 `scanf()` 的相关内容，要像躲避瘟疫一样躲开这两个函数。这里“危险”的含义是：这种缓冲区溢出是“黑客攻击”（即非法闯入计算机系统）的主要手段。现在很多编译器都会对 `gets()` 及其近亲给出警告信息，原因就在于此。

#### 26.3.3.4 分支

显然，当必须做出选择的时候，我们可能会做出错误的抉择。这就使得 `if` 和 `switch` 语句成为测试人员的好目标。有两个主要问题需要检查：

- 所有的可能性都被覆盖了么？
- 操作是否与分支正确联系起来？

考虑下面这个函数：

```
void do_branch1(int x, int y)    // 混乱的函数
// 不可控的 if 使用
{
    if (x<0) {
        if (y<0)
            cout << "very negative\n";
        else
            cout << "somewhat negative\n";
    }
    else if (x>0) {
        if (y<0)
            cout << "very positive\n";
        else
            cout << "somewhat positive\n";
    }
}
```

其中最明显的错误是我们“忘记”了  $x$  是 0 的情况。当测试非零值的时候（或是测试正值和负值的时候），零经常被忘记或者错误地与其他情况混在一起（例如考虑负数的情况）。此外，这个程序中还隐藏着一个很微妙（但不常见）的错误： $(x>0 \ \&\& \ y<0)$  和  $(x>0 \ \&\& \ y>=0)$ ，从某种角度看，它们是被颠倒了。这通常是在编辑程序时使用剪切 - 粘贴操作所导致的。

`if` 语句的使用方式越复杂，就越可能出现这种错误。从测试人员的角度出发，我们应该检查代码并确保所有分支都已被测试。对于 `do_branch1()` 来说，一个明显的测试集是

```
do_branch1(-1,-1);
do_branch1(-1, 1);
do_branch1(1,-1);
do_branch1(1,1);
do_branch1(-1,0);
do_branch1(0,-1);
do_branch1(1,0);
do_branch1(0,1);
do_branch1(0,0);
```

基本上，这是一种蛮力测试方法，通过“遍历所有可能”来查找错误。由于我们已经注意到 `do_branch1()` 使用 `<` 与 `>` 检测非 0 值，因此采用了这一方法。此外，为了检查  $x$  为正值时的可能错误，我们还需要把每个调用与期望的正确结果结合起来。

处理 `switch` 语句的方法与 `if` 语句基本上是一样的。

```
void do_branch1(int x, int y) // 混乱的函数
// 不可控的 switch
{
    if (y<0 && y<=3)
        switch (x) {
            case 1:
                cout << "one\n";
                break;
            case 2:
                cout << "two\n";
            case 3:
                cout << "three\n";
        }
}
```

这里，我们犯了四个错误：

- 我们对错误的变量进行了范围检查（应是  $y$  而不是  $x$ ）。
- 对  $x=2$ ，我们忘记了 `break` 语句，这会导致错误的操作。
- 我们忘记了默认情况（认为在 `if` 语句中已经考虑了这种情况）。
- 我们使用了  $y<0$  而实际上我们的意思是  $0<y$ 。

作为测试人员，我们一定要检查这些未处理的情况。请注意，仅仅“修复错误”是不够的。如果我们不检查所有可能的情况，错误还可能再次出现。作为测试人员，我们希望能够系统地捕捉到所有可能的错误。这样简单的代码，如果只是修正错误，我们很可能在修正过程中再犯错，不仅不能解决问题，甚至还可能引入新的不同的错误。检查代码的目的并不是要找到错误（虽然这很重要），而是要设计出能够捕获所有错误的测试集（或者，现实一点，捕获尽可能多的错误）。

需要注意的是：循环有一个隐式的“`if`”：它用于检测是否达到循环终止条件。因此，循环也包含分支语句。当我们看到代码中的分支语句，首先要考虑的问题是：“我们是否已经覆盖（测试）了所有分支？”令人惊讶的是，对于实际代码，覆盖所有分支并不总是可行的（因为在实际代码中，根据需要，一个函数可能被其他函数调用，但调用并不是所有情况下都必然进行的）。因此，对于测试人员来说，一个更一般的问题是：“你所要求的代码覆盖率是多少？”答案最好是“我们测试大部分分支”，然后解释为什么余下的分支很难测试。100%的分支覆盖只是理想的情况。

### 26.3.4 系统测试

重要系统的测试是一项技术性工作。例如，对电话系统的计算机控制子系统进行测试，就需要在放置了许多机架式计算机的专门机房中进行，通过模拟数万人的通话来测试控制系统。这种测试系统本身的价值就达到数百万美元，而且需要非常熟练的工程师团队来实施测试。而电话系统一旦投入使用，其主要的电话交换机需要在持续运行 20 年的时间内最多停机 20 分钟（包括各种原因，例如停电、水灾、地震等）。我们不会深入讨论这一问题，比起解决这个问题，教会一个物理系新生计算火星探测器的方向修正量会更容易些。但我们会给你介绍一些思想，这些思想对于测试一个较小的系统或者理解更大系统的测试会有所帮助。

首先，请记住测试的目的是发现错误，特别是那些可能频繁发生和很严重的错误。编写和运行大量的测试不是一件简单的工作。它要求测试人员要对待测系统有一定的理解。与单元测试相比，系统测试的有效性更依赖于应用程序的相关知识（领域知识）。开发一个系

统不仅仅要用到编程语言和计算机科学知识，还需要对应用领域和用户的了解。这也是激励我们从事编程工作的重要原因之一：我们可以接触到许多有趣的应用程序，认识很多有趣的人。

一个完整的待测系统应该是由许多组成部分（单元）构成，其构造可能会花费很长时间。因此，一个可行的策略是在完成所有单元测试之后，对于大量系统测试每天只做一次（通常是在开发人员晚上睡觉的时候）。在这个过程中，回归测试是一项关键工作。最可能发生错误的地方是新加入的代码和以前发现过错误的代码。因此，重新运行旧的测试集（回归测试）是非常必要的。如果不这样做，一个大系统永远也不会达到稳定状态。因为在我们消除旧错误的同时，也可能引入新错误。

⚠ 注意，我们认为：当修正错误的时候，意外地引入一些新的错误是很正常的。我们希望新错误的数量低于已排除的错误的数量，而且新错误的严重程度也低于老的错误。然而，至少在重新进行回归测试，并对新代码进行测试之前，我们必须假定系统是有问题的（我们的错误修正工作引起了问题）。

### 26.3.5 寻找不成立的假设

`binary_search` 的规范明确要求输入序列必须是有序的。这个条件让我们不能利用许多单元测试的技巧。但这显然为编写糟糕代码提供了机会，我们已经设计的测试（系统测试除外）不能发现其中的错误。我们是否能利用对系统“单元”（函数、类等）的理解来设计更好的测试呢？

✗ 不幸的是，最简单的答案是否定的。作为纯粹的测试者，我们不能改变代码，而只能检查是否违反了接口的要求（前置条件），必须有人在每次调用前进行前置条件检查，或者实现为函数的一部分（见 5.5 节）。但是，只有待测代码是我们自己编写的，我们才可以插入这样的测试代码。如果我们是测试人员，并且代码的编写者会听从我们的要求（这并不总能实现），我们可以告诉他们有要求未检查的情况，并要求他们确保要求都被检查。

回到 `binary_search` 的例子：我们不能检测输入序列 `[first,last)` 是否是一个合法序列以及是否有序（见 26.3.2.2 节）。但是，我们可以用下面的函数来检查：

```
template<class Iter, class T>
bool b2(Iter first, Iter last, const T& value)
{
    // 检查 [first,last) 是否是合法序列
    if (last<first) throw Bad_sequence();

    // 检查序列是否有序
    if (2<=last-first)
        for (Iter p = first+1; p<last; ++p)
            if (*p<*(p-1)) throw Not_ordered();

    // 检查都通过，调用 binary_search
    return binary_search(first,last,value);
}
```

现在，`binary_search` 中没有包含这些测试代码，原因包括：

- 比较运算 `last<first` 不能用于前向迭代器；例如，`std::list` 的迭代器就没有 `<`（参见 C.3.2 节）。通常，没有一个真正好的办法来测试有一对迭代器定义了一个合法的序列（可以从 `first` 开始迭代，期待最终能遇到 `last`，但这不是一个好的检测方法）。

- 通过扫描整个序列来确定序列是否有序的代价远超过执行 `binary_search` 本身 (`binary_search` 的目的不是盲目地遍历整个序列去查找某个值, 这是 `std::find` 的做法)。

那我们能做什么呢? 我们可以在测试中用 `b2` 来替代 `binary_search` (只在用随机访问迭代器调用 `binary_search` 时进行替换)。此外, 如果允许的话, 我们可以要求 `binary_search` 的编写者插入能打开要求检测的代码:

```
template<class Iter, class T>    // 警告: 包含伪代码
bool binary_search (Iter first, Iter last, const T& value)
{
    if (test enabled) {
        if (Iter is a random access iterator) {
            // 检查 [first,last) 是否是合法序列:
            if (last<first) throw Bad_sequence();
        }

        // 检查序列是否有序:
        if (first!=last) {
            Iter prev = first;
            for (Iter p = ++first; p!=last; ++p, ++ prev)
                if (*p<*prev) throw Not_ordered();
        }
    }

    // 现在开始执行 binary_search
}
```

其中, `test enabled` 的含义依赖于测试是如何 (为特定方式组织的专用系统) 安排的, 因此我们把它设为伪代码: 在测试你自己的代码时, 你就可以用一个 `test_enabled` 变量来代替。我们也把 `Iter is a random access iterator` 检测作为伪代码, 因为我们没有解释“迭代器特征” (`iterator trait`) 是什么。如果你真的需要做这个检测, 你可以在高阶 C++ 书籍中查找迭代器特征的相关内容。

## 26.4 测试方案设计

在开始编写程序的时候, 我们当然希望它最终是完整的、正确的。我们知道, 为了达成这一目标, 必须进行测试。因此, 从编写程序的第一天起, 我们在设计中就要考虑正确性和测试。实际上, 许多优秀的程序员都有个口号“早测试, 经常测试”, 而且, 他们不会在考虑好代码将来如何测试之前, 就急于动手编写。及早考虑测试问题有助于避免发生在早期的错误 (也有助于以后发现错误)。我们赞成这种程序设计哲学。一些程序员甚至在编写程序单元之前就编写好了单元测试。

26.3.2.1 节和 26.3.3 节中的例子解释了这些关键的思想:

- 使用定义良好的接口, 这样你可以测试这些接口的使用。
- 设计用文字描述各种操作的方式, 这样它们就可以被存储、分析和重放。这也包括输出操作。
- 在调用代码中嵌入对未检查假设 (判定) 的检测, 以便在系统测试前捕获错误参数。
- 最小化依赖性, 并且保持各种依赖关系清晰可见。
- 有一个清晰的资源管理策略。

从哲学上讲, 这些思想可以看作单元测试技术能很好地应用于子系统和整个系统的保证。

如果不考虑性能的话，我们可以将未检查假设（要求、前置条件等）的检测代码一直打开。但是，程序通常不包含这部分代码，这是有原因的。例如，我们看到检查输入序列是否有序比 `binary_search` 本身还要复杂，并且代价更高。因此，设计一个系统能允许我们根据需要选择性打开 / 关闭这种检测，是一个好想法。对大多数系统来说，在最终（发布）版本中留下相当多代价比较低的检查代码是个好主意：因为在一些“不可能”的情况发生的时候，我们更希望通过一个明确的错误信息来了解情况，而不是一个简单的系统崩溃。

## 26.5 调试

调试是一种技术，也是一种态度。显然，态度更重要。请回顾一下第 5 章，注意一下调试与测试的不同。这两者都捕获错误，但是调试远比测试专用，重点关注排除已知错误和实现新特性。任何一种能让调试更像测试的方法我们都会去尝试。要说我们喜欢测试确实有些夸张，但是我们确实讨厌调试。及早进行单元测试，在设计时考虑测试，都有助于最小化调试工作量。

## 26.6 性能

对一个有用的程序来说，仅仅满足正确性是不够的。即使假定有足够的工具来编写出有用的程序，程序也必须拥有一定的性能。一个好的程序应该是“效率足够高的”，即它能够在给定的资源条件下，在可接受的时间内得到结果。但要注意的是绝对的效率并不是我们的首要目标。一种固有的认识是运行更快的程序可能会给开发工作带来麻烦，因为它会导致复杂的代码（可能包含更多的错误，调试工作量等大），使维护（包括移植和性能调优）工作更困难、代价更高。

那么，我们怎么才能知道一个程序（或程序单元）是“效率足够高的”呢？理论上讲，我们是不可能知道的。而且很多程序运行的硬件都非常快，使得这个问题不那么关键。我们曾经看到过这种情况：为了更好地检测系统发布后所发生的错误（即使是最好的代码，当它与其他代码一起工作时，错误也会发生），有的正式产品会以调试模式编译（虽然这可能导致运行速度比发布模式慢 25 倍）。

因此，对问题“效率是否足够高”的答案是：“测量感兴趣的测试用例花费多长时间。”在这里，你显然要充分了解最终用户“感兴趣”的是什麼，以及他们对于这些“感兴趣”的测试用例可接受的最长运行时间是多长。逻辑上，我们可以用秒表进行计时，并检查所花费的时间是否合理。在实际中，我们可以使用一些函数，例如 `system_clock`（见 26.6.1 节）来完成计时功能。而且，我们还可以自动比较测试所花费的时间与预先估计的合理时间。一种替代方法（或者与前一种比较同时做）是，记下测试花费的时间，与以前的测试进行比较。这是一种性能回归测试。

一些最糟糕的性能 bug 是由糟糕的算法引起的，这种问题也可以通过测试发现。使用大数据集进行测试的原因之一就是要暴露低效的算法。例如，假设有一个应用程序求矩阵每行元素之和（使用第 24 章中的 `Matrix` 库），下面是某人提供的相应函数：

```
double row_sum(Matrix<double,2> m, int n); // 求 m[n] 中的元素之和
```

现在，有人使用这个函数生成一个 `vector`，`v[n]` 保存前 `n` 行元素之和：

```
double row_accum(Matrix<double,2> m, int n) // 计算 m[0:n] 中元素之和
{
```

```
double s = 0;
for (int i=0; i<n; ++i) s+=row_sum(m,i);
return s;
}
```

```
// 计算 m 中每行的累加和
vector<double> v;
for (int i = 0; i<m.dim1(); ++i) v.push_back(row_accum(m,i+1));
```

设想这个函数是单元测试的一部分，或者是系统测试所测试的应用程序的一部分。不管在哪种情况下，只要矩阵足够大，你都会发现一些奇怪的现象：基本上，程序所需要的时间与  $m$  的元素数目的平方成正比。为什么呢？因为函数先是求第一行所有元素之和，然后再求第二行元素之和（再次访问了第一行所有元素），接着求第三行所有元素之和（再次访问了前两行的所有元素），依此类推。

如果你认为这个例子不够好，那么想象一下如果 `row_sum()` 需要通过访问数据库读取数据的话，会发生什么吧。读硬盘会比读取主存慢上千倍。

现在，你可能会抱怨“没人会写出这么愚蠢的代码！”。抱歉，我们见过更糟的。当隐藏在应用程序代码之中时，糟糕的算法（从性能的角度来看）是很难被发现的。当你第一次看这段代码的时候，你注意到性能问题了吗？除非你专门关注于这类问题，否则问题是很难以被发现的。下面是一个来自真实的服务程序的例子：

```
for (int i=0; i<strlen(s); ++i) {
    // 对 s[i] 的一些处理
}
```

一般情况下，`s` 是一个包含 20K 个字符的字符串。

并不是所有的性能问题都是由糟糕的算法导致的。实际上（正如我们在 26.3.3 节中指出的），我们所编写的大部分代码不能归类为特定的算法。一般来说，这些“非算法”的性能问题都被归类为“糟糕的设计”。具体包括：

- 信息的重复计算（例如，上面的矩阵行求和问题）。
- 重复检查（例如，在循环中每次都检查数据的索引，或者在函数间传递参数时，即便参数没有改变，也重复检查它）。
- 重复访问硬盘（或网络）。

注意“重复”一词。显然，我们是指“不必要的重复”，但是，除非你重复很多次，这类操作不会对性能产生显著影响。对于函数参数和循环变量，我们当然要仔细检查。但是，如果对同一个值进行一百万次检查的话，这种冗余检查可能会伤害性能。如果通过测试，我们发现性能受到了影响，我们就要看看是否可以消除这些重复操作。然而，除非你认为性能是个关键问题，否则不要轻易删除代码。过早的优化反而可能浪费时间或引入更多错误。

### 26.6.1 计时

你怎么才能知道一段代码是否足够快呢？你如何确定一个操作的执行时间呢？在大部分情况下，你可以简单地通过看表来达到目的（秒表、挂钟或闹钟）。虽然这种方法不科学，也不准确，但是如果这种方法不可行的话（意味着你没有来得及看清花费了多少时间），通常你就可以下结论：程序足够快。但纠缠于性能问题并不是一个好的思路。

如果你希望获得精确时间，或者你不能坐在那里看秒表的话，你就需要计算机来帮助



你。计算机可以获得准确的运行时间。例如，在 Unix 系统中，只需要在所运行命令前加上 `time` 前缀，系统就会显示所花费的时间。你可以用 `time` 来查看编译一个 C++ 源文件 `x.cpp` 需要多长时间。通常，你的编译指令如下：

```
g++ x.cpp
```

如果要查看编译时间，可以加上 `time`：

```
time g++ x.cpp
```

上面的指令将编译 `x.cpp` 并将所花费的时间输出到屏幕上。对于小程序来说，这是一种简单、有效的办法。需要记住的是这种方法需要多运行几次，因为你的计算机上的“其他活动”可能影响计时的准确性。如果连续三次得到大致相同的结果，通常你就可以信任这一结果了。

但是，如果待测程序的运行时间是毫秒级的，应该怎么办呢？如果你希望更准确地测量程序的某一部分花费的时间，应该怎么办呢？你可以使用标准库函数 `system_clock` 来计时，下面这个例子中就是采用这种方法测量函数 `do_something()` 所花费的时间：

```
#include <chrono>
#include <iostream>
using namespace std;

int main()
{
    int n = 1000000;    // 重复 do_something() n 次

    auto t1 = system_clock::now();    // 开始计时

    for (int i = 0; i < n; i++) do_something();    // 循环计时
    auto t2 = system_clock::now();    // 结束计时

    cout << "do_something() " << n << " times took "
         << duration_cast<milliseconds>(t2-t1).count() << "milliseconds\n";
}
```

`system_clock` 是一个标准计时器。`system_clock::now()` 返回调用时的时间点 (`time_point`)。两个时间点之差 (这里是 `t2-t1`) 就是所用时长 (`duration`)。为了回避 `duration` 和 `time_point` 的具体类型细节，这里我们使用了 `auto` 类型。用手表看时间非常简单，但是使用标准库函数复杂很多。实际上，标准库的时间工具是为高级物理应用设计的，它的灵活和普适性远超普通用户的需要。

时长 `duration` 可以用指定的计量单位表示，例如秒 (`seconds`)、毫秒 (`milliseconds`) 或纳秒 (`nanoseconds`) 等。我们使用 `duration_cast` 将时间强制转化为指定的计量单位。这一转换是必要的，因为不同的系统、不同的计时方法使用的计量单位都是不同的。不要忘了 `.count()`，它实际上是从 `duration` 中抽取出来的计数单位个数 (时钟计数)，`duration` 包括时钟计数和计量单位。

`system_clock` 是为测量秒级时间设计的。不要用它来测量小时级别的时间。

⚠ 再次强调，对于任一个测试对象，如果你不能连续三次得到大致相同的结果，那么这个测试是不可信的。什么是“大致相同的结果”呢？合理的答案是“误差在 10% 以内”。现代计算机是非常快的：每秒执行 10 亿条指令是很普通的。这意味着很多程序的运行时间很难被测量，除非你把某个程序重复运行上万次或者它本身确实就很慢，例如有写硬盘或访问互

联网的情况。对于后者，可能我们只需要运行重复几百次就可以了。但是对于如何正确理解这些实验结果，可能会有一些困难。

## 26.7 参考文献

- Stone, Debbie, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. *User Interface Design and Evaluation*. Morgan Kaufmann, 2005. ISBN 0120884364.
- Whittaker, James A. *How to Break Software: A practical Guide to Testing*. AddisonWesley, 2003. ISBN 0321194330.

## 简单练习

运行下列 `binary_search` 程序的测试：

- 实现 26.3.2.2 节中 `Test` 的输入操作符。
- 对来自 26.3 节的序列，将测试描述补充完整。
  - `{123581321}` // 一个“平凡序列”
  - `{}`
  - `{1}`
  - `{1234}` // 偶数个元素
  - `{12345}` // 奇数个元素
  - `{11111111}` // 所有元素相等
  - `{01111111111111}` // 开头的元素不同
  - `{000000000000001}` // 结束的元素不同
- 基于 26.3.1.3 节的内容，编写一个程序，它可以生成：
  - 一个非常大的序列（你认为什么是大，为什么？）。
  - 十个序列，每个序列的元素个数是随机的。
  - 十个序列，每个序列分别包含 0, 1, 2, ..., 9 个随机数作为元素（但仍然有序）。
- 重复上述测试，但序列中元素为字符串，例如 `{Bohr Darwin Einstein Lavoisier Newton Turing}`。

## 思考题

- 制作一个应用程序的列表，对每个程序都给出可能导致最严重后果的 `bug` 的简单说明。例如：航班控制程序——坠机：231 人死亡；损失价值 5 亿美元的设备。
- 为什么我们不直接证明程序的正确性呢？
- 单元测试与系统测试有什么不同？
- 什么是回归测试，它为什么很重要？
- 测试的目的是什么？
- 为什么 `binary_search` 不检查它的要求？
- 如果我们不能检查到所有错误，那么我们应该主要查找哪类错误？
- 在处理数据序列时，错误最可能出现在代码的哪部分？
- 为什么在测试时使用大数值是个好注意？
- 为什么测试用例经常以数据而不是代码形式出现？



11. 为什么我们要使用大量基于随机数的测试用例？什么时候使用？
12. 为什么测试带有 GUI 的程序很困难？
13. 为什么需要独立测试一个“单元”？
14. 可测试性和可移植性的联系是什么？
15. 为什么测试一个类要比测试一个函数困难？
16. 测试的可重复性为什么很重要？
17. 当发现一个“单元”依赖于未检查的假设（前置条件）时，测试者应该怎么办？
18. 程序设计者 / 实现者应该如何做，才能改进测试？
19. 测试与调试有什么不同？
20. 什么时候性能是我们要考虑的因素？
21. 对于如何（容易地）制造低性能问题，给出两个（或更多）例子。

## 术语

assumptions (假设)	pre-condition (前置条件)	test coverage (测试覆盖)
black-box testing (黑盒测试)	proof (证明)	test harness (测试工具)
branching (分支)	regression (回归)	testing (测试)
design for testing (测试设计)	resource usage (资源使用)	timing (时间)
inputs (输入)	state (状态)	unit test (单元测试)
outputs (输出)	system_clock	white-box testing (白盒测试)
post-condition (后置条件)	system test (系统测试)	

## 习题

1. 使用 26.3.2.1 节中的测试用例测试 26.1 节中的 `binary_search` 算法程序。
2. 修改 `binary_search` 的测试，使它能够处理任意数据类型，然后测试 `string` 序列和浮点序列。
3. 使用接受比较操作参数的 `binary_search` 版本，重复习题 1 中的练习。列出引入额外的参数后，可能出现的新错误。
4. 设计一种测试数据的格式，可以让你只需定义一次数据序列，但可以在多个测试中使用。
5. 在 `binary_search` 的测试集中加入一个测试，它能够捕获 `binary_search` 修改数据序列这种（不太可能的）错误。
6. 对第 7 章的计算器程序做一些尽可能小的改动，使它能够从文件输入数据，并可以将输出存入文件中（或者使用你的操作系统的 I/O 重定向功能）。然后为它设计一套可行的综合测试。
7. 测试 15.6 节中的“简单文本编辑器”程序。
8. 为第 17 ~ 20 章中的图形界面库增加一个文本界面。例如，字符串“`Circle(Point(0,1),15)`”应该生成一个调用 `Circle(Point(0,1),15)`。使用这个文本界面生成一个“儿童图画”：一个带屋顶的二维房子，两个窗户和一个门。
9. 为图形界面库增加一个基于文本的输出格式。例如，当调用 `Circle(Point(0,1),15)` 时，一个字符串“`Circle(Point(0,1),15)`”也应被发送到输出流中。
10. 使用习题 9 中的基于文本的界面为图形界面库写一个更好的测试。

11. 测量 26.6 节中的矩阵求和例子的时间，其中矩阵是方阵，维度分别是 100、10 000、1 000 000 和 10 000 000。元素值是  $[-10:10]$  之间的随机数。使用一个更有效的算法（非  $O(N^2)$ ）重写程序，并比较所花费的时间。
12. 写一个程序，它能生成随机浮点数并用 `std::sort()` 对这些数进行排序。比较 500 000 个和 5 000 000 个 `double` 值进行排序所花费的时间。
13. 重复上一题中的实验，但使用的是随机字符串，长度范围是  $[0:100)$ 。
14. 重复上一题，但是使用 `map` 而不是 `vector`，这样我们就不需要进行排序了。

## 附言

作为程序员，我们梦想能够编写出第一次运行就通过的完美程序。但是现实是残酷的：保证程序的正确性是很困难的，而且当我们（和我们的同事）改进代码时，很难使程序保持在正确的状态。测试（包括在设计时考虑测试问题）是保证我们提交的系统真正正常运行的主要方法。无论何时，当我们在这个高技术世界中结束一天工作的时候，我们真的应该对（经常被忘记的）测试人员报以善意。

# C 语言

C 是一种强类型、弱检查的程序设计语言。

——Dennis Ritchie

本章简要概述 C 语言及其标准库，假定读者已经掌握了 C++ 语言。我们列出 C 不支持的 C++ 特性，并通过例子程序展示 C 语言如何应对这些特性缺失。我们还会讨论 C 和 C++ 不兼容的地方，以及 C 和 C++ 的互操作方式。我们会通过举例来说明 I/O、列表操作、内存管理和字符串操作方面的 C 特性。

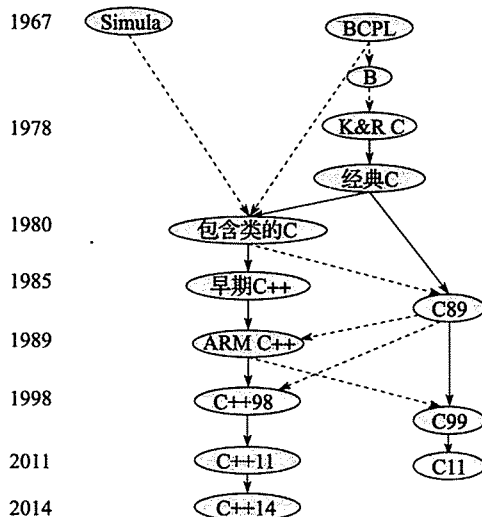
## 27.1 C 和 C++：兄弟

C 语言是由贝尔实验室的 Dennis Ritchie 设计和开发的，Brian Kernighan 和 Dennis Ritchie 合著的《The C Programming Language》一书（俗称“K&R”）使它迅速普及。这本书可能是迄今为止最好的 C 语言入门书籍和最好的程序设计书籍之一（见 22.2.5 节）。C++ 最初的定义文本，是在 Dennis Ritchie 的 1980 版 C 定义的基础上修改而来的。在此之后，两种语言都有了进一步的发展。与 C++ 一样，现在 C 语言的标准制定也是由 ISO 标准组织负责的。

我们大致上可以将 C 看作 C++ 的子集。因此，从 C++ 的角度看，介绍 C 语言就归结为两个问题：

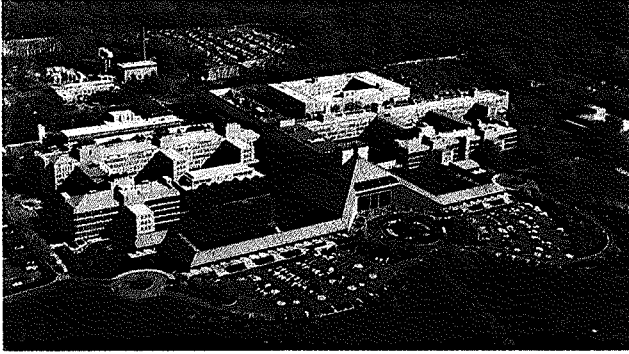
- C 的哪些特性并非 C++ 的子集。
- C++ 的哪些特性 C 并不支持，以及用什么样的 C 特性和技术可以弥补。

历史上，现代 C 语言和现代 C++ 语言是兄弟关系，两者都是“经典 C”的直系后裔。这里的“经典 C”是指 Brian Kernighan 和 Dennis Ritchie 的《The C Programming Language》第 1 版中介绍的 C 再加上结构赋值和枚举类型两个特性：



在所有的 C 版本中，C89（见 K&R 第 2 版）目前占据统治地位，本章介绍的就是 C89。还有其他一些经典 C 在使用，C99 也有一些应用，但只要你掌握了 C++ 和 C89，使用这些不同的“方言”都不成问题。

C 和 C++ 都“出生”在新泽西州茉莉山贝尔实验室的计算机科学研究中心（有段时间，我的办公室和 Brian Kernighan、Dennis Ritchie 的办公室就隔着一个走廊和几道门）：



C 和 C++ 的标准制定目前都由 ISO 标准委员会负责。两种语言都有大量的实现在使用中。现在的编译系统通常都同时支持 C 和 C++，通过编译选项或源程序后缀选择是按哪种语言编译。两种语言比其他任何语言都要更普及。两者最初的设计目的和当前最重要的应用都是系统级程序设计，如：

- 操作系统内核。
- 设备驱动程序。
- 嵌入式系统。
- 编译器。
- 通信系统。

等价的 C 和 C++ 程序在性能上没有什么差异。

与 C++ 一样，C 的应用非常广泛。两者合并计算的话，其开发社群应该是地球上最大的软件开发社群。

### 27.1.1 C/C++ 兼容性

我们经常会听到“C/C++”这种提法。但是，并不存在这种语言，这种提法是无知的表现。我们只在讨论 C/C++ 兼容性问题，以及论及大的 C/C++ 共享技术社群时才会用“C/C++”的说法。

C++ 大体上是（但不完全是）C 的一个超集。大部分 C 和 C++ 都有的特性，其语义在两种语言中也是相同的，例外情况很少。C++ 的设计目标之一就是“尽可能接近 C，直到不能再近”，目的在于：

- 易于两种语言间的转换；
- 两种语言的共存。

两者的不兼容之处大多与 C++ 更严格的类型检查有关。

下面是一个合法的 C 语言程序，但它不是合法的 C++ 程序，原因在于其中一个标识符（class）是 C++ 的关键字，但不是 C 的关键字（见 27.3.2 节）：

```
int class(int new, int bool); /* C 合法, C++ 非法 */
```

下面是一个在两种语言中都合法但语义不同的例子:

```
int s = sizeof('a'); /* 在 C 中表示 sizeof(int), 一般是 4, 而在 C++ 中通常是 1 */
```

在 C 语言中, 字符常量 (如 'a') 的类型是 int, 而在 C++ 中是 char。但是, 对于一个 char 型变量 ch, 两种语言中都有 sizeof(ch)==1。

关于兼容性和语言差异的话题总是不那么令人兴奋, 因为里面没有什么新的程序设计技术可学。你可能会对 printf() (见 27.6 节) 感兴趣, 但除此之外 (以及一些无用的工程师间的幽默), 本章显得有些干巴巴的。本章的目的很简单: 使你能读写 C 程序 (如果你有这种需求的话)。本章内容主要是指出一些潜在的危险: 一些有经验的 C 程序员认为很显然, 但对于 C++ 程序员通常很意外的东西。我们希望你能以最小的代价学会规避这些危险, 而不必在实际应用中撞得头破血流。

大多数 C++ 程序员迟早都会遇到必须处理 C 代码的情况, 就像大多数 C 程序员必须处理 C++ 代码一样。本章介绍的很多内容对于大部分 C 程序员来说都是很熟悉的内容, 但也有些内容属于“专家级知识”。原因很简单: 人们对什么是“专家级”很难达成共识, 我们只是介绍那些实际程序中很常见的问题。也许理解兼容性问题是赢得“C 专家”赞誉的捷径, 但请记住: 真正的专家水平是指使用语言 (本章中是 C) 的水平高超, 而不是指理解了一些深奥的语言规则 (如一些兼容性问题)。

### 参考文献

ISO/IEC 9899:1999. *Programming Languages-C*. 此标准定义了 C99, 目前大多数编译器实现的是 C89 (通常增加了一些扩展特性)。

ISO/IEC 9899:2011. *Programming Languages — C*. 这定义了 C11。

ISO/IEC 14882:2011. *Programming Languages — C++*.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. ISBN 0131103628.

Stroustrup, Bjarne. “Learning Standard C++ as a New Language.” *C/C++ Users Journal*, May 1999.

Stroustrup, Bjarne. “C and C++: Siblings”; “C and C++: A Case for Compatibility”; “C and C++: Case Studies in Compatibility.” *The C/C++ Users Journal*, July, Aug., and Sept. 2002.

我的论文在我的主页上很容易找到。

### 27.1.2 C 不支持的 C++ 特性

从 C++ 的角度, C (即 C89) 缺少很多特性, 如:

- 类和成员函数
  - 解决方法: 使用 struct 和全局函数。
- 派生类和虚函数
  - 解决方法: 使用 struct、全局函数和函数指针 (见 27.2.3 节)。
- 模板和内联函数
  - 解决方法: 使用宏 (见 27.8 节)。

- 异常
  - 解决方法：使用错误代码、错误返回值等。
- 函数重载
  - 解决方法：不同函数使用不同名字。
- new/delete:
  - 解决方法：使用 malloc()/free() 和分离的初始化 / 结束处理代码。
- 引用
  - 解决方法：使用指针。
- const、constexpr 或者常量表达式形式的函数
  - 解决方法：使用宏。
- bool 类型
  - 解决方法：使用 int。
- static\_cast、reinterpret\_cast 和 const\_cast
  - 解决方法：使用 C 风格的类型转换如 (int)a 来替代 C++ 风格的 static<int>(a)。

很多有用的代码都是用 C 写的，因此上面这个列表实际上在提醒我们：没有什么语言特性是绝对必要的。很多语言特性，甚至是大多数 C 语言特性，只是为了方便程序员编写程序而设计的。毕竟，如果你足够聪明、足够有耐心，而且给你足够时间的话，任何程序都可以用汇编语言写出来。注意，由于 C 和 C++ 都使用相同的机器模型，而且这个模型非常接近实际计算机，因此它们都非常适合于模拟很多不同的程序设计风格。

本章剩余部分将介绍在没有这些特性的情况下如何编写有用的程序。对于使用 C 语言，我们的基本建议如下：

- 用 C 语言特性来模拟 C++ 特性所支持的程序设计技术。
- 当编写 C 程序时，使用 C++ 中的 C 子集。
- 调整编译器的警告级别，确保进行函数参数检查。
- 对于大型程序，使用 lint（见 27.2.2 节）。

很多 C/C++ 不兼容之处的细节相当晦涩难懂。但是，如果只是读写 C 程序，大多数细节你都不必记忆，因为：

- 当你使用 C 不支持的 C++ 特性时，编译器会提醒你。
- 如果遵循上述原则，你几乎不会遇到相同语句在 C 和 C++ 中语义不同的情况。

虽然缺少上述 C++ 特性，但也有一些特性在 C 中更为重要：

- 数组和指针。
- 宏。
- typedef（C 和 C++ 98 对于简单的 using 声明是等价的，见 15.5 节和附录 A.16）。
- sizeof。
- 类型转换。

本章中也会给出这些特性的一些例子。

我将 C 的前身 BCPL 中的 // 注释引入了 C++，因为我真的厌烦了输入 /\* ... \*/ 方式的注释。很多 C 的方言包括 C99 和 C11 都支持 //，因此它很多情况下是安全的，尽管使用就是。但在本章中，对于 C 语言的例子程序，我们只使用 /\* ... \*/ 注释。C99 和 C11 吸纳了一些 C++ 的特性（以及一些和 C++ 兼容的特性），但在本章中我们使用 C89，因为 C89 要普及得多。

### 27.1.3 C 标准库



很自然，C++ 中与类和模板相关的特性 C 是不支持的，包括：

- vector
- map
- set
- string
- STL 算法：如 `sort()`、`find()` 和 `copy()`
- `iostream`
- `regex`

对于这些特性，我们通常可以用基于数组、指针和函数的 C 标准库特性来完成类似功能。C 标准库主要包括如下部分：

- `<stdlib.h>`：一般工具（如 `malloc()` 和 `free()`，参见 27.4 节）。
- `<stdio.h>`：标准 I/O，参见 27.6 节。
- `<string.h>`：C 风格字符串处理和内存管理，参见 27.5 节。
- `<math.h>`：标准浮点数学函数，参见 24.8 节。
- `<errno.h>`：`<math.h>` 所涉及的错误码，参见 24.8 节。
- `<limits.h>`：整数类型的大小，参见 24.2 节。
- `<time.h>`：日期和时间，参见 26.6.1 节。
- `<assert.h>`：调试用的断言，参见 27.9 节。
- `<ctype.h>`：字符集，参见 11.6 节。
- `<stdbool.h>`：布尔宏。

这些标准库特性的完整说明，请查阅好的 C 语言教材，如 K&R。所有这些库（和头文件）也存在于 C++ 中。

## 27.2 函数

在 C 中：

- 函数不能重名。
- 函数参数类型检查是可选的，不是强制的。
- 没有引用类型（因而参数传递也没有传引用方式）。
- 没有成员函数。
- 没有内联函数（C99 除外）。
- 有可选的函数定义语法。

除此之外，C 的函数与 C++ 很相似。下面我们逐个考察这些差别。

### 27.2.1 不支持函数名重载

考虑下面代码：

```
void print(int);           /* 输出一个 int */
void print(const char*);  /* 输出一个 string **/* 错误！**/
```

第二个声明是错误的，因为两个函数不能同名。所以你必须设计两个恰当的名字：


```
void print_int(int);          /* 输出一个 int */
void print_string(const char*); /* 输出一个 string */
```

不支持函数重载有时还会被认为是一个好的特性：不会发生使用错误的函数输出整数的意外情况！显然，我们并不接受这种说法，不支持函数重载使得泛型程序设计思想变得很尴尬，因为泛型程序设计的重要特点就是用相同的名字表示语义相似的函数。

## 27.2.2 函数参数类型检查

考虑下面代码：

```
int main()
{
    f(2);
}
```


它在 C 编译器中会顺利编译通过，也就是说，你不必在使用函数之前声明函数（虽然可以这么  做，而且也应该这么做）。f() 可能定义在程序某个地方，也可能在其他文件中，但如果并未定义的话，连接程序就会报错。

不幸的是，即使 f() 定义在其他文件中，它也有可能是这样的：

```
/* 其他文件：*/

int f(char* p)
{
    int r = 0;
    while (*p++) r++;
    return r;
}
```

连接程序不会报告这个错误。你只会得到一个运行时错误，或者是奇怪的运行结果。

我们如何应对这类问题呢？头文件的一致使用是一个实用的方法。如果你调用或定义  的每个函数都在一个特定的头文件中声明，而且无论是否用到，每个程序文件都包含此头文件，那么我们就可以确保函数声明检查。然而，在大型程序中，这很难做到。因此，大多数 C 编译器都具备对应的编译选项，以选择是否对未定义函数的调用给出警告，我们可以利用这一特性。而且，从 C 语言出现的早期开始，就已经有了能检查代码一致性问题的程序。这些程序通常被称为 lint。在编译每个大型程序之前，我们都应使用 lint 来检查程序。你会发现 lint 会推动你以非常类似 C++ 子集的方式来使用 C。因为 C++ 最初的一个设计目标就是使编译器能容易地检查更多（而不是所有）lint 能检查的问题。

可以让 C 进行函数参数检查。这很简单，只要在函数声明时指出参数类型即可（如同 C++ 那样）。这种函数声明称为函数原型（function prototype）。但是，要小心那些未指定参数的函数声明，这些声明不是函数原型，并不保证进行函数参数检查：

```
int g(double);          /* 函数原型——类似 C++ 的函数声明 */
int h();                /* 不是函数原型——参数类型未定义 */

void my_fct()
{
    g();                /* 错误：缺少参数 */
    g("asdf");        /* 错误：类型不匹配 */
    g(2);               /* 正确：2 被转换为 2.0 */
    g(2,3);            /* 错误：参数过多 */
}
```



```

h();          /* 编译通过, 但可能有意料之外的结果 */
h("asdf");   /* 编译通过, 但可能有意料之外的结果 */
h(2);        /* 编译通过, 但可能有意料之外的结果 */
h(2,3);      /* 编译通过, 但可能有意料之外的结果 */
}

```

其中, `h()` 的声明没有指定参数类型。这并不意味着 `h()` 不接受参数, 而是意味着“接受任何参数集合, 期望它们与被调函数匹配”。再次申明, 好的编译器会对这类问题给出警告, 而 `lint` 可以检查出这类问题。

C++	C 中等价语法
<code>void f(); // 首选方式</code>	<code>void f(void);</code>
<code>void f(void);</code>	<code>void f(void);</code>
<code>void f(...); // 接受任何参数</code>	<code>void f(); /* 接受任何参数 */</code>

对于作用域中找不到函数原型的的情况, C 定义了一组特殊的规则来转换参数。例如, `char` 和 `short` 会转换为 `int`, 而 `float` 会转换为 `double`。如果需要了解相关内容, 比如说 `long` 如何处理, 请查阅好的 C 教材。我们的建议很简单: 所有函数都应该给出函数原型。

注意, 即使错误类型的参数通过了编译器的检查, 例如将一个 `char*` 传递给了 `int` 型的参数, 在其使用中也会出错。如 Dennis Ritchie 所说: “C 是一个强类型、弱检查的程序设计语言。”

### 27.2.3 函数定义

你可以像在 C++ 中一样定义函数, 定义本身就是函数原型:

```

double square(double d)
{
    return d*d;
}

void ff()
{
    double x = square(2);          /* 正确: 2 被转换为 2.0 并调用 */
    double y = square();          /* 丢失参数 */
    double y = square("Hello");  /* 错误: 类型不匹配 */
    double y = square(2,3);       /* 错误: 参数过多 */
}

```

没有参数的函数定义不是函数原型:

```

void f() { /* 完成某个任务 */ }

void g()
{
    f(2); /* C 语言可运行; C++ 出错 */
}

```

其中

```
void f(); /* 未指定参数类型 */
```

意为“`f()` 可以接受任意数目、任意类型的参数”, 这看起来真的很奇怪。为此, 我发明了一种新的语法, 用关键字 `void` 显式表示“什么都没有”的含义 (英文单词 `void` 的意思就是“什

么都没有”):

```
void f(void); /* 没有可接收的参数 */
```

我很快就后悔了，因为这种方式看起来很奇怪，而且如果能够保证进行参数类型检查的话，这种方式完全是多余的。更糟的是，Dennis Ritchie（C 语言之父）和 Doug McIlroy（在贝尔实验室计算机科学研究中心内部，在审美方面他是最终裁决者，参见 22.2.5 节）都称 void 参数是“讨厌的”。不幸的是，这个讨厌的东西在 C 社群内已经非常流行了。在 C++ 中不要使用它，因为在 C++ 中，它不仅丑陋，而且逻辑上是多余的。

C 还提供了另一种 Algol60 风格的函数定义方式——参数类型（可选的）与参数名分离：

```
int old_style(p,b,x) char* p; char b;
{
    /* ... */
}
```

这种“旧式定义”比 C++ 的历史还早，它也不是函数原型。默认情况下、未指定类型的参数被当作 int 型。因此，x 是函数 old\_style() 的一个整型参数。我们可以这样调用 old\_style():

```
old_style();           /* 正确：所有参数都缺失 */
old_style("hello", 'a', 17); /* 正确：所有参数都严格匹配 */
old_style(12, 13, 14); /* 正确：12 类型不匹配 */
                       /* 但可能 old_style() 不会用到 p */
```

编译器应该会接受这些调用（但我们期望它对第一个和第三个调用能给出警告）。

对于函数参数检查问题，我们的建议是：

- 坚持使用函数原型（使用头文件）。
- 设置编译器的警告级别，使它能捕获参数类型错误。
- 使用 lint。

遵循这些原则的结果就是，写出的 C 代码也是正确的 C++ 代码。

## 27.2.4 C++ 调用 C 和 C 调用 C++

如果编译器支持的话，可以将 C 编译器编译出的目标文件和 C++ 编译出的目标文件连接在一起。例如，可以将 GNU C/C++ 编译器（GCC）编译出的 C 和 C++ 目标文件连接在一起。微软 C/C++ 编译器（MSC++）生成的 C 和 C++ 目标文件也可以连接在一起。这种开发方式很常见也很有用，你可以使用的库的范围大大拓宽，不必受限于一语言的库。

C++ 的类型检查比 C 严格。特别是，C++ 的编译器和连接器会检查两个函数 f(int) 和 f(double) 的定义和使用是否一致，即使定义和使用在不同的源文件中。而 C 连接器不会做这种检查。为了从 C++ 中调用 C 函数，以及从 C 中调用 C++ 函数，应该通知编译器我们的意图：

```
// 在 C++ 中调用 C 函数：
extern "C" double sqrt(double); // 以 C 函数连接

void my_c_plus_plus_fct()
{
    double sr = sqrt(2);
}
```

`extern "C"` 告知编译器使用 C 连接器约定。除此之外，与普通 C++ 函数没有什么区别。实际上，C++ 标准库函数 `sqrt(double)` 通常就是 C 标准库中的 `sqrt(double)`。采用这种方法，无须对 C 程序进行任何特殊处理，其中的函数就能被 C++ 程序调用。我们要做的全部事情只是让 C++ 采用 C 的连接约定。

`extern "C"` 也可用来使 C++ 函数能被 C 调用：

// 在 C 中调用 C++ 函数：

```
extern "C" int call_f(S* p, int i)
{
    return p->f(i);
}
```

这样，我们就可以在 C 程序中间接调用成员函数 `f()` 了：

/\* 在 C 中调用 C++ 函数：\*/

```
int call_f(S* p, int i);
struct S* make_S(int, const char*);

void my_c_fct(int i)
{
    /* ... */
    struct S* p = make_S(x, "foo");
    int x = call_f(p, i);
    /* ... */
}
```

在 C 中无须（或不可能）声明这是一个 C++ 函数。

这种互操作性的好处是显而易见的：可以混合使用 C 和 C++ 编写程序。特别是，C++ 程序可以使用 C 库，而 C 程序也可以使用 C++ 库。而且，很多语言（如 Fortran）都提供和 C 的调用接口。

在上例中，我们假定 C 和 C++ 可以共享 `p` 指向的类对象。对于大多数类对象来说，这是没有问题的。特别是，如果你定义了下面这样的类：

```
// 在 C++ 中：
class complex {
    double re, im;
public:
    // 所有常用操作
};
```

你可以在 C++ 程序和 C 程序之间传递对象指针，甚至可以在 C 程序中访问 `re` 和 `im`，只需定义如下结构类型：

```
/* 在 C 中：*/
struct complex {
    double re, im;
    /* 没有操作 */
};
```

任何程序设计语言中的内存布局规则都可能很复杂，不同语言混合编程中的内存布局规则就更加难以说清。但对于 C 和 C++ 来说，内置类型和无虚函数的类（`struct`）对象可以直接传递。如果类具有虚函数，你只能传递对象的指针，而且实际的处理工作应该交给 C++ 代码。`call_f()` 就是一个例子：`f()` 可能是虚函数，这个例子展示了如何在 C 程序中调用虚

函数。

除了内置类型外，最简单也最安全的类型共享方式就是在一个公共头文件中定义 `struct`。但是，这种方法严重限制了 C++ 的编程模式，因此我们不会局限于这种方式。

### 27.2.5 函数指针

如果希望在 C 中使用面向对象技术（见 19.2 ~ 19.4 节），应该怎么做呢？最重要的是要找到虚函数的替代技术。大多数人会马上想到一个主意：在 `struct` 中添加一个“类型域”来指明对象是基类还是某个派生类，例如，对于 `Shape` 及其派生类，指明给定对象是哪种形状。如下面程序所示：

```
struct Shape1 {
    enum Kind { circle, rectangle } kind;
    /* ... */
};

void draw(struct Shape1* p)
{
    switch (p->kind) {
        case circle:
            /* 画一个圆 */
            break;
        case rectangle:
            /* 画一个矩形 */
            break;
    }
}

int f(struct Shape1* pp)
{
    draw(pp);
    /* ... */
}
```

这段程序可以正常工作，但存在两处隐患：

- 我们必须为每个“伪虚函数”（如 `draw()`）编写一个新的 `switch` 语句。
- 每当加入一个新的形状，我们就必须修改每个“伪虚函数”（如 `draw()`），为 `switch` 语句增加一个 `case` 分支。

第二个问题非常讨厌，它意味着我们不可能将“伪虚函数”放在库中，因为用户不得不频繁修改这些函数。虚函数最有效的替代技术之一是函数指针：

```
typedef void (*Pfct0)(struct Shape2*);
typedef void (*Pfct1int)(struct Shape2*,int);

struct Shape2 {
    Pfct0 draw;
    Pfct1int rotate;
    /* ... */
};

void draw(struct Shape2* p)
{
    (p->draw)(p);
}

void rotate(struct Shape2* p, int d)
```

```
{
    (p->rotate)(p,d);
}
```

Shape2 的使用与 Shape1 一样:

```
int f(struct Shape2* pp)
{
    draw(pp);
    /* ... */
}
```

稍微做一点改动, 对象就不必携带每个伪虚函数的指针, 而是保存一个指向函数指针数组 (对于 C++ 中虚函数的所有实现) 的指针即可。在实际程序设计中, 这种方法的主要问题是正确初始化所有函数指针。

## 27.3 小的语言差异

本节介绍 C 和 C++ 之间的一些小的差异, 如果你从未听说过这些差异的话, 就容易犯错。一些差异还会严重影响与之明显相关的程序设计工作。

### 27.3.1 struct 标签名字空间

在 C 中, struct 的名字 (C 中没有类) 与其他标识符位于不同的名字空间。因此, 每个 struct 的名字 (称为一个结构标签 (structure tag)) 必须以关键字 struct 为前缀。例如:

```
struct pair { int x,y; };
pair p1;          /* 错误: pair 没有关键字 struct */
struct pair p2;   /* 正确 */
int pair = 7;     /* 正确: 结构标签没有在此作用域 */
struct pair p3;   /* 正确: 结构标签没有被 int 屏蔽 */
pair = 8;         /* 正确: pair 指的是 int 类型 */
```

令人惊讶的是, 归功于一个不很正当的兼容性漏洞, 这段代码在 C++ 中也是正确的。变量 (或者函数) 与 struct 同名是一种常见的 C 语言用法, 但我们并不推荐这样做。

如果你不希望在每个结构名之前写上 struct 前缀, 可以使用 typedef (见 15.5 节)。下面的程序写法很常见:

```
typedef struct { int x,y; } pair;
pair p1 = { 1, 2};
```

一般而言, typedef 在 C 中更为常见, 也更为有用, 因为在 C 语言中你没有办法定义附带操作的新类型。

在 C 中, 嵌套的 struct 的名字与包含它的 struct 位于同一个作用域中, 例如:

```
struct S {
    struct T { /* ... */ };
    /* ... */
};

struct T x;    /* C 语言支持 (C++ 不支持) */
```

而在 C++ 中, 必须这样写:

```
S::T x;    // C++ 支持 (C 不支持)
```

只要可能, 不要使用嵌入的 struct: 其作用域规则与大多数人的自然的 (也是合理的)

想法不同。

### 27.3.2 关键字

很多 C++ 关键字不是 C 关键字（因为 C 不支持对应的功能），因此可以用作 C 标识符：

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

不要将这些名字用作 C 标识符，否则你的程序将无法移植为 C++ 程序。如果你在头文件中使用了这些名字，头文件将无法被 C++ 使用。 ⚠

一些 C++ 关键字是 C 中的宏：

C++ 关键字，C 中的宏				
and	bitor	false	or	wchar_t
and_eq	bool	not	or_eq	xor
bitand	compl	not_eq	true	xor_eq

在 C 中，这些宏是在 `<iso646.h>` 和 `<stdbool.h>`（`bool`、`true`、`false`）中定义的。不要利用它们是 C 的宏这一特性。

### 27.3.3 定义

与 C89 相比，C++ 允许将定义放置在程序更多的地方。例如：

```
for (int i = 0; i < max; ++i) x[i] = y[i];    // C 中，这种 i 的定义是不允许的

while (struct S* p = next(q)) {            // C 中，这种 p 的定义是不允许的
    /* ... */
}

void f(int i) {
    if (i < 0 || max <= i) error("range error");
    int a[max];    // 错误：C 中语句之后的声明是不允许的
    /* ... */
}
```

C (C89) 不允许在 `for` 语句的初始化部分、条件判断或者语句块中语句之后放置声明。

这段代码在 C 中必须这样写：

```
int i;
for (i = 0; i < max; ++i) x[i] = y[i];

struct S* p;
while (p = next(q)) {
    /* ... */
}

void f(int i)
{
    if (i < 0 || max <= i) error("range error");
    {
        int a[max];
        /* ... */
    }
}
```

在 C++ 中，未初始化的声明被视为一个定义，但在 C 中，仍被视为一个声明，因此可以重复多次：

```
int x;
int x;    /* 定义或声明一个名为 x 的整型变量，在 C 中合法，在 C++ 中错误 */
```

在 C++ 中，每个实体只能定义一次。这引出一个有趣的问题：如果不同源文件中有两个同样的定义，会发生什么情况？

```
/* 文件 x.c: */
int x;

/* 文件 y.c: */
int x;
```

单独编译 x.c 和 y.c 的话，C 和 C++ 编译器都不会发现错误。但是，如果在 C++ 中将 x.c 和 y.c 一起编译，连接程序会报告“重复定义”错误。如果两者是在 C 中一起编译，连接会顺利通过，因为（根据 C 的规则）连接程序认为 x.c 和 y.c 共享一个 x。但最好不要用这种方式，如果你确实希望不同源文件共享一个全局变量 x，应该显式声明：

```
/* 文件 x.c: */
int x = 0;    /* 定义 */

/* 文件 y.c: */
extern int x;    /* 声明，不是定义 */
```

更好的方式是使用头文件：

```
/* 文件 x.h: */
extern int x;    /* 声明，不是定义 */

/* 文件 x.c: */
#include "x.h"
int x = 0;    /* 定义 */


/* 文件 y.c: */
#include "x.h"
/* 声明在头文件中 */
```

当然，最好的方式是避免使用全局变量。

### 27.3.4 C 风格类型转换

在 C 中 (C++ 中也可以), 可以用下面这样的简单语法来实现值  $v$  向类型  $T$  的转换:

$(T)v$

这种“C 风格类型转换”, 或称为“老式类型转换”, 受到打字不熟练或者马虎的程序员  的欢迎, 因为它非常简单, 而且你不必了解  $v$  到底是如何转换为类型  $T$  的。但代码维护人员却很害怕这种形式, 因为它几乎是隐形的, 而且对于程序作者的意图没有给出任何线索。C++ 风格的类型转换 (或称为新式类型转换 (new-style cast) 或模板方式转换 (template-style cast), 参见附录 A.5.7) 是显式的类型转换, 因此易于发现、意义明确。但在 C 中, 我们只能使用老式的类型转换:

```
int* p = (int*)7;      /* 重解释二进制位模式: reinterpret_cast<int*>(7) */
int x = (int)7.5;     /* 截断 double: static_cast<int>(7.5) */

typedef struct S1 { /* ... */ } S1;
typedef struct S2 { /* ... */ } S2;
S2 a;
const S2 b;          /* 未初始化 const 在 C 中是允许的 */

S1* p = (S1*)&a;      /* 重解释二进制位模式: reinterpret_cast<S1*>(&a) */
S2* q = (S2*)&b;      /* 强制转换常量: const_cast<S2*>(&b) */
S1* r = (S1*)&b;      /* 取出常量并改变类型; 可能存在错误 */
```

即使在 C 语言中, 我们也很犹豫是否推荐使用宏 (见 27.8 节), 但宏又确实可以表达某些思想:

```
#define REINTERPRET_CAST(T,v) ((T)(v))
#define CONST_CAST(T,v) ((T)(v))

S1* p = REINTERPRET_CAST (S1*,&a);
S2* q = CONST_CAST(S2*,&b);
```

这种方法并不具备 `reinterpret_cast` 和 `const_cast` 的类型检查能力, 而且很丑陋, 但它至少易于发现, 也使程序员的意图更为明显。

### 27.3.5 无类型指针的转换

在 C 中, 可以将 `void*` 赋予任何指针类型的变量, 或用它来初始化指针变量, 在 C++ 中这是不可以的。例如:

```
void* alloc(size_t x);          /* 分配 x 个字节 */

void f (int n)
{
    int* p = alloc(n*sizeof(int)); /* C 中合法, C++ 中错误 */
    /* ... */
}
```

在这段代码中, `alloc()` 返回的 `void*` 值被隐式地转换为 `int*` 类型。而在 C++ 中, 我们必须这样写:

```
int* p = (int*)alloc(n*sizeof(int)); /* C 和 C++ 都合法 */
```

这条语句使用了 C 风格的类型转换 (参见 27.3.4 节), 因此在 C 和 C++ 中都是正确的。



⚠ 为什么在 C++ 中 `void*` 到 `T*` 的隐式类型转换是非法的呢？因为这种转换可能是不安全的：

```
void f()
{
    char i = 0;
    char j = 0;
    char* p = &i;
    void* q = p;
    int* pp = q;      /* 不安全；C 中合法，C++ 中错误 */
    *pp = -1;        /* 对从 &i 开始的地址空间进行覆盖 */
}
```

在这段代码中，我们甚至不能确定哪个内存区域被更改了。也许是 `j` 和 `p` 的一部分？也许是用于 `f()` 的调用管理的内存区域（`f` 的调用栈）？无论如何，对 `f()` 的调用都是一件糟糕的事情。

注意，从 `T*` 到 `void*` 的（反向）转换是百分之百安全的，这样的转换不会形成像上面代码一样糟糕的程序，因此在 C 和 C++ 中都允许这样的转换。

不幸的是，隐式的 `void*` 到 `T*` 的类型转换在 C 程序中随处可见，这也许是实际程序中最主要的 C/C++ 兼容性问题。

### 27.3.6 枚举

在 C 中，可以将一个 `int` 值赋予一个 `enum` 变量，而无须类型转换。例如：

```
enum color { red, blue, green };
int x = green;      /* C 和 C++ 都合法 */
enum color col = 7; /* C 中合法；C++ 中错误 */
```

这个特性意味着，在 C 中我们可以对枚举变量进行增 1（++）和减 1（--）运算。这可能很方便，但会导致灾难性的后果：

```
enum color x = blue;
++x;      /* x 变为 green；C++ 中错误 */
++x;      /* x 变为 3；C++ 中错误 */
```

这段代码中，`x` “跌落出”了枚举常量的范围，可能我们就是想这么做，但也很有可能是我们无意中犯的错误。

注意，与结构标签类似，枚举类型名也位于自己独立的名字空间中，因此使用时必须使用关键字 `enum` 作为前缀：

```
color c2 = blue;      /* C 中错误：color 不在名字空间；C++ 中合法 */
enum color c3 = red; /* 正确 */
```

### 27.3.7 名字空间

C 不支持名字空间（这里的“名字空间”一词是指在 C++ 中的含义）。那么在大型 C 程序中应该如何避免名字冲突呢？一般的策略是使用前缀和后缀。例如：

```
/* 文件 bs.h: */
typedef struct bs_string { /* ... */ } bs_string; /* Bjarne 的 string 类型 */
typedef int bs_bool;      /* Bjarne 的 Boolean 类型 */
/* 文件 pete.h: */
typedef char* pete_string; /* Pete 的 string 类型 */
typedef char pete_bool;    /* Pete 的 Boolean 类型 */
```

这种方法实在是太流行了，因此使用一两个字母的前缀不是好的选择，很容易与其他人代码中的名字产生冲突。

## 27.4 自由存储空间

C 并未提供 `new` 和 `delete` 操作符来进行对象分配与释放。为了使用自由存储空间，可以用一些处理内存的函数。最重要的几个函数定义在“一般工具”标准头文件 `<stdlib.h>` 中：

```
void* malloc(size_t sz);          /* 分配 sz 个字节 */
void free(void* p);              /* 释放 p 指向的内存空间 */
void* calloc(size_t n, size_t sz); /* 分配 n*sz 个字节并初始化为 0 */
void* realloc(void* p, size_t sz); /* 分配大小为 sz 的空间，指针 p 指向这一空间 */
```

类型 `size_t` 也是在 `<stdlib.h>` 中定义的，它是用 `typedef` 定义的无符号整型。

为什么 `malloc()` 返回一个 `void*`？原因在于它不知道你要在分配到的内存空间中存入什么样的对象。对象的初始化应该是你的责任，例如：



```
struct Pair {
    const char* p;
    int val;
};

struct Pair p2 = {"apple", 78};
struct Pair* pp = (struct Pair*) malloc(sizeof(Pair)); /* 分配 */
pp->p = "pear";          /* 初始化 */
pp->val = 42;
```

注意，无论是在 C 中还是在 C++ 中，都不可以这样写：

```
*pp = {"pear", 42}; /* 在 C 或 C++98 中都错误 */
```

但在 C++ 中，可以为 `Pair` 定义一个构造函数，然后这样写：

```
Pair* pp = new Pair("pear", 42);
```

在 C 中（C++ 中不可以，见 27.3.4 节），可以省略 `malloc()` 之前的类型转换，但我们不推荐这么做：

```
int* p = malloc(sizeof(int)*n); /* 避免这样使用 */
```

省去类型转换的做法很流行，因为可以省去一些打字工作量，而且这样做还能检查到一种罕见的错误：在使用 `malloc()` 之前忘记了包含 `<stdlib.h>`。但是，这种方法也会掩盖内存大小的计算错误：

```
p = malloc(sizeof(char)*m); /* 可能出现的错误——没有 m 个 int 空间用于分配 */
```

不要在 C++ 程序中使用 `malloc()` 和 `free()`，因为 `new/delete` 不需要类型转换，可以进行初始化（构造函数）和清理工作（析构函数），还能报告内存分配错误（抛出异常），而且和 `malloc()/free()` 一样快。也不要使用 `delete` 来释放一个由 `malloc()` 分配的空间，或者用 `free()` 释放用 `new` 创建的空间，例如：

```
int* p = new int[200];
// ...
free(p); // 错误

X* q = (X*)malloc(n*sizeof(X));
// ...
delete q; // 错误
```

这段代码也许会正确运行，但它难以移植。而且，对于具有构造函数和析构函数的对象，如果混合使用 C 风格和 C++ 风格的动态内存分配代码，很容易导致灾难性后果。

函数 `realloc()` 通常用于扩展缓冲区：

```
int max = 1000;
int count = 0;
int c;
char* p = (char*)malloc(max);
while ((c=getchar())!=EOF) { /* 读操作：忽略末尾的 eof 字符 */
    if (count==max-1) { /* 需要扩展缓冲区 */
        max += max; /* 加倍缓冲区 */
        p = (char*)realloc(p,max);
        if (p==0) quit();
    }
    p[count++] = c;
}
```

C 的输入操作的相关内容，可参见 27.6.2 节和附录 C.11.2。

函数 `realloc()` 可以将数据从旧的内存空间移动到新的内存空间，但这种数据移动也有 **⚠** 可能无法完成。绝对不要将 `realloc()` 用于 `new` 分配的内存空间。

在 C++ 中，(大致)等价的代码如下所示 (使用了标准库)：

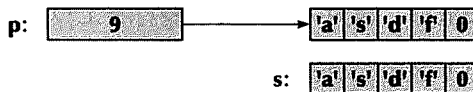
```
vector<char> buf;
char c;
while (cin.get(c)) buf.push_back(c);
```

请参考“Learning Standard C++ as a New Language”一文 (见 27.1 节中给出的参考文献列表)，其中对输入和内容分配策略进行了全面的讨论。

## 27.5 C 风格字符串

在 C 中，字符串 (在 C++ 的文献中，通常称之为 C 字符串或 C 风格字符串) 就是一个以 0 (数值) 结尾的字符数组。例如：

```
char* p = "asdf";
char s[] = "asdf";
```



在 C 中，没有成员函数机制，不能重载函数，也不能为 `struct` 定义运算符 (如 `==`)。因此我们需要一组函数 (非成员函数) 来处理字符串。C 和 C++ 的标准库提供了如下函数 (在 `<string.h>` 中)：


```
size_t strlen(const char* s); /* 字符计数 */
char* strcat(char* s1, const char* s2); /* 字符串 s2 拷贝到 s1 的末尾 */
int strcmp(const char* s1, const char* s2); /* 按照字典序比较 */
char* strcpy(char* s1, const char* s2); /* 把 s2 拷贝到 s1 */

char* strchr(const char *s, int c); /* 在 s 中查找 c */
char* strstr(const char *s1, const char *s2); /* 在 s1 中查找 s2 */

char* strncpy(char*, const char*, size_t n); /* 与 strcpy 功能相同，但最多 n 个字符 */
char* strncat(char*, const char, size_t n); /* 与 strcat 功能相同，但最多 n 个字符 */
int strncmp(const char*, const char*, size_t n); /* 与 strcmp 功能相同，但最多 n 个字符 */
```

这里并没有列出所有字符串函数，但最有用和最常用的函数都已经列出了。我们简单说明一

下如何使用这些函数。

我们可以进行字符串比较。但相等运算符 (==) 比较的是两个字符串的指针值，标准库  函数 `strcmp()` 才是比较字符串内容的：

```
const char* s1 = "asdf";
const char* s2 = "asdf";

if (s1==s2) { /* s1 和 s2 指向同一个地址吗？ */
    /* (一般不是你所希望的) */
}

if (strcmp(s1,s2)==0) { /* s1 和 s2 指向的地址存储相同内容吗？ */
}

```

函数 `strcmp()` 对字符串进行三路比较。如上面程序所示，对于给定的字符串参数 `s1` 和 `s2`，`strcmp(s1,s2)` 返回 0 表示两个字符串完全相等。如果在字典序中，`s1` 位于 `s2` 之前，`strcmp` 会返回一个负数，如果 `s1` 位于 `s2` 之后，返回一个正数。单词 “lexicographical” 表示字典序，例如：

```
strcmp("dog","dog")==0
strcmp("ape","dodo")<0 /* 按照字典序 "ape" 在 "dodo" 之前 */
strcmp("pig","cow")>0 /* 按照字典序 "pig" 在 "cow" 之后 */

```

字符串指针的比较 `s1==s2` 也不一定得到 0 (假, false)。因为某些实现可能将所有相同内容的字符串只保存一份，这样 `s1` 和 `s2` 会指向相同内存空间，于是比较的结果是 1 (真, true)。因此，使用 `strcmp()` 才是正确的 C 风格字符串比较方法。

函数 `strlen()` 用来获得 C 风格字符串的长度：

```
int lgt = strlen(s1);
```

注意，长度不包括结尾的 0。在本例中，`strlen(s1)==4`，但 `s1` 实际占用了 5 个字节来保存 “asdf”。这个微小的差别是很多错一位错误的根源。

我们还可以拷贝 C 风格字符串 (结尾的 0 也会被拷贝)：

```
strcpy(s1,s2); /* 从 s2 到 s1 进行字符拷贝 */
```

你 (调用者) 应该保证目标字符串 (数组) 有足够的空间容纳源字符串中的字符。

函数 `strncpy()`、`strncat()` 和 `strncmp()` 是 `strcpy()`、`strcat()` 和 `strcmp()` 限定处理长度的版本，第三个参数 `n` 指出了最多处理多少个字符。注意，如果源字符串中的字符不足 `n` 个，`strncpy()` 不会拷贝结尾的 0，因此得到的结果是一个非法的 C 风格字符串。

函数 `strchr()` 和 `strstr()` 在第一个参数 (一个字符串) 中搜索第二个参数 (字符或字符串)，并返回匹配的位置。与 `find()` 类似，它们从左至右搜索。

令人惊奇的是这些简单的函数能完成很多功能，同样令人惊奇的是使用这些函数非常容易出现小的错误。考虑这么一个简单问题：将用户名和地址字符串连接起来，之间加入一个 @。在 C++ 中，可以使用 `std::string`：

```
string s = id + '@' + addr;
```

使用 C 风格字符串，我们可以这样写：

```
char* cat(const char* id, const char* addr)
{
    int sz = strlen(id)+strlen(addr)+2;
```

```

char* res = (char*) malloc(sz);
strcpy(res,id);
res[strlen(id)+1] = '@';
strcpy(res+strlen(id)+2,addr);
res[sz-1]=0;
return res;
}

```

这段代码是正确的吗？谁会用 `free()` 操作释放掉 `cat()` 返回的字符串？

### 试一试

测试 `cat()`。为什么计算新字符串的长度时要加 2？我们在 `cat()` 中留下了一个初学者常犯的错误，找到并修正它。我们“忘记”写注释了。请添加注释，假定读者了解标准 C 字符串函数。

## 27.5.1 C 风格字符串和 `const`

考虑下面代码：

```

char* p = "asdf";
p[2] = 'x';

```

- ⚠ 这段代码在 C 中是合法的，但在 C++ 中不合法。在 C++ 中，一个字符串文字常量被视为常量，是不可更改的，因此 `p[2]='x'`（将 `p` 指向的内容改为 "axdf"）是非法的。不幸的是，很少有编译器能捕获这个错误，导致出现问题。如果你足够幸运的话，程序运行时会产生一个错误，但你不能期望总是这么幸运，有可能产生更为严重的后果。你应该这样编写代码：

```

const char* p = "asdf"; // 现在你 cannot 通过 p 修改 "asdf"

```

我们建议在 C 和 C++ 中都这样编写程序。

C 的 `strchr()` 函数有一个相似但更难以发现的问题。考虑如下代码：

```

⚠ char* strchr(const char* s, int c); /* 在常量 s 中查找字符 c (不是 C++) */

const char aa[] = "asdf";          /* aa 是常量数组 */
char* q = strchr(aa, 'd');         /* 查找 'd' */
*q = 'x';                          /* 将 aa 中的 'd' 改为 'x' */

```

这段代码在 C 和 C++ 中都是非法的，但 C 编译器不能捕获这个错误。这种错误有时被称为嬗变 (transmutation)：它把 `const` 类型变为非 `const` 类型，违反了对代码的合理假设。

在 C++ 中，可以用标准库声明的另一个 `strchr()` 函数：

```

char const* strchr(const char* s, int c); // 在常量 s 中查找字符 c
char* strchr(char* s, int c);           // 在 s 中查找字符 c

```

函数 `strstr()` 也存在同样问题。

## 27.5.2 字节操作

在遥远的黑暗时代（20 世纪 80 年代早期），当时 `void*` 尚未发明，C（和 C++）程序员只能使用字符串操作来处理字节。现在的标准库中已经有了基本的内存处理函数，它们接受 `void*` 型参数，返回 `void*` 型值，以此来警告用户——它们直接处理内存，因此本质上讲处理对象应该是无类型的内存数据。

```

/* 从 s2 到 s1 拷贝 n 个字节 (类似 strcpy): */
void* memcpy(void* s1, const void* s2, size_t n);

/* 从 s2 到 s1 拷贝 n 个字节 ( [s1:s1+n] 可能与 [s2:s2+n] 重叠 ): */
void* memmove(void* s1, const void* s2, size_t n);

/* 比较字符串 s2 和 s1 中最多 n 个字节 (类似 strcmp): */
int memcmp(const void* s1, const void* s2, size_t n);

/* 在 s 的前 n 个字节中查找 c (转换为 unsigned char): */
void* memchr(const void* s, int c, size_t n);

/* 拷贝 c (转换为 unsigned char) 到 s 所指向地址的前 n 个字节: */
void* memset(void* s, int c, size_t n);

```

不要在 C++ 中使用这些函数。特别是 `memset()`，它会干扰构造函数的正常工作。

### 27.5.3 实例: `strcpy()`

`strcpy()` 的定义应该是为人们所熟知的了，但它作为 C (和 C++) 简洁性范例的一面，就不那么为人所知道了：

```

char* strcpy(char* p, const char* q)
{
    while (*p++ = *q++);
    return p;
}

```

为什么这段代码的确将 C 风格字符串 `q` 的内容拷贝到 `p` 中，留给大家思考。后缀增量的描述在 A.5 节：`p++` 的值是 `p` 加 1 前的值。

#### 试一试

这个 `strcpy()` 的实现正确吗？解释为什么。

如果你不能解释为什么，我们认为你还不是一个 C 程序员（不过你可能是合格的其他语言的程序员）。每种语言都有自己的风格特色，这就是 C 的特色。

### 27.5.4 一个风格问题

对一个常常引起激烈争论的很大程度上与程序设计本身无关的风格问题，我们已经默默地支持很长时间了。我们可以定义指针类型如下：

```
char* p; // p 是一个指向字符的指针
```

而不是这样定义：

```
char *p; /* p 是某种能够获得字符的引用 */
```

空格放在哪里对于编译器来说毫无意义，但是程序员却很在意。我们的风格（在 C++ 中很常见）强调变量的类型，而第二种风格（在 C 中很常见）强调对指针变量的使用。注意，我们并不推荐在一条语句中声明很多变量：

```
char c, *p, a[177], *f(); /* 合法，但容易混淆 */
```

这种声明语句在老式程序中并不罕见。我们建议用多条语句来声明这些变量，并利用每行剩

余的空间添加注释和初始化代码：

```
char c = 'a';      /* f() 中的输入终结符 */
char* p = 0;      /* f() 最后读入的字符 */
char a[177];     /* 输入缓冲区 */
char* f();       /* 向缓冲区读入数据，返回指向第一个读入字符的指针 */
```

而且，应该为变量取更有意义的名字。

## 27.6 输入 / 输出：stdio

C 中没有 `iostream`，因此我们使用 `<stdio.h>` 中定义的 C 标准 I/O，这组特性通常被称为标准 I/O (`stdio`)。`stdio` 中与 `cin` 和 `cout` 等价的是 `stdin` 和 `stdout`。在一个程序中，可以（对相同的 I/O 流）混合使用 `stdio` 和 `iostream`，但我们不推荐这么用。如果你觉得需要混合使用，请查阅专家级的参考书籍中有关 `stdio` 和 `iostream`（特别是 `ios_base::sync_with_stdio()`）的详细介绍。参见附录 C.11。

### 27.6.1 输出

最常用也最有用的 `stdio` 函数是 `printf()`，它的最基本的用途是打印（C 风格）字符串：

```
#include<stdio.h>

void f(const char* p)
{
    printf("Hello, World!\n");
    printf(p);
}
```

这个例子不是那么有趣。有趣的一点是 `printf()` 能够接受任意数量的参数，第一个参数（一个字符串）控制其后的参数如何打印。C 中 `printf()` 的定义如下所示：

```
int printf(const char* format, ...);
```

“...” 的含义是“可以有更多参数”。我们可以这样调用 `printf()`：

```
void f1(double d, char* s, int i, char ch)
{
    printf("double %g string %s int %d char %c\n", d, s, i, ch);
}
```

这里，`%g` 表示“用一般格式打印一个浮点值”，`%s` 表示“打印一个 C 风格字符串”，`%d` 表示“以十进制格式打印一个整数”，`%c` 表示“打印一个字符”。每个格式限定符都打印下一个未处理的参数，因此 `%g` 打印 `d`，`%s` 打印 `s`，`%d` 打印 `i`，`%c` 打印 `ch`。附录 C.11.2 中给出了 `printf()` 的完整格式限定符列表。

⚠ 不幸的是，`printf()` 不是类型安全的，例如：

```
char a[] = {'a', 'b'};      /* 没有终结符 0 */

void f2(char* s, int i)
{
    printf("goof %s\n", i);    /* 不能捕获的错误 */
    printf("goof %d: %s\n", i); /* 不能捕获的错误 */
    printf("goof %s\n", a);    /* 不能捕获的错误 */
}
```

最后一个 `printf()` 的效果很有趣：它会打印 `a[1]` 之后内存中的每个字节（字符），直至遇到 0 为止。打印出的字符数目可能非常多。

虽然 `stdio` 在 C 和 C++ 中都能正常工作，但由于缺少类型检查，我们更倾向于使用 `iostream`。倾向于 `iostream` 的另一个原因是 `stdio` 函数没有扩展性：你无法扩展 `printf()` 来输出自定义类型，而使用 `iostream` 是可以做到这点的。例如，你无法定义新的格式限定符 `%Y` 来输出自定义类型 `struct Y`。

`printf()` 还有一个很有用的版本，可以向文件中打印数据：

```
int fprintf(FILE* stream, const char* format, ...);
```

例如：

```
fprintf(stdout, "Hello, World!\n"); // 完全类似 printf("Hello, World!\n");
FILE* ff = fopen("My_file", "w"); // 以写方式打开文件 My_file
fprintf(ff, "Hello, World!\n");    // 写 "Hello, World!\n" 到 My_file
```

第一个参数是一个文件句柄（文件描述符），文件句柄将在 27.6.3 节中介绍。

## 27.6.2 输入

最常用的 `stdio` 输入函数包括：

```
int scanf(const char* format, ...); /* 用指定格式从 stdin 读数据 */
int getchar(void);                 /* 从 stdin 读一个字符 */
int getc(FILE* stream);            /* 从 stream 读一个字符 */
char* gets(char* s);               /* 从 stdin 读一串字符 */
```


最简单的读取字符串的方式是使用 `gets()`，例如：

```
char a[12];
gets(a); /* 读一个字符数组到 a 中，以 '\n' 结束 */
```

永远不要这样编写程序！`gets()` 是有害的！曾经有大约 1/4 的成功黑客攻击是由于 `gets()` 和它的近亲 `scanf("%s")` 的漏洞造成的。到现在为止，这仍然是一个主要的安全问题。以上面简单的程序为例，你如何知道在换行之前用户最多输入 11 个字符呢？你是无法知道的。因此，`get()` 几乎肯定会导致内存破坏（缓冲区之后的内存空间），而内存破坏目前仍是黑客的主要工具之一。不要认为你可以猜测一个最大缓冲区规模，能“对所有用户都足够大”。也许在输入流另一端的那个“人”只是一个程序，它会打破你的合理假设。

函数 `scanf()` 使用类似 `printf()` 的格式限定串来指定输入格式。其使用与 `printf()` 一样方便：

```
void f()
{
    int i;
    char c;
    double d;
    char* s = (char*)malloc(100);
    /* 读入变量并以指针形式传递：*/
    scanf("%i %c %g %s", &i, &c, &d, s);
    /* %s 忽略前面的空格并以空格结束 */
}
```

与 `printf()` 类似，`scanf()` 也不是类型安全的。格式字符串和参数（指针）必须严格匹配，



否则在运行时就会产生奇怪的结果。而且，`%s` 将字符串读入 `s` 的过程中还会发生溢出。因此，永远不要使用 `gets()` 或 `scanf("%s")`！

那么如何才能安全地读取字符呢？我们可以在格式限定符 `%s` 中指定要读取的字符的数目，例如：

```
char buff[20];
scanf("%19s",buff);
```

我们需要为结尾的 0 留出存储空间，因此能读入 `buff` 的最大字符数为 19。但是，这又引起一个新的问题：如果用户输入的字符数超过了 19 个，应该怎么办呢？`scanf()` 的处理方式是将“多余”的字符留在输入流中，随后的输入操作可能会“发现”这些字符。

由于 `scanf()` 存在这些问题，一般来说更为谨慎也更为容易的方法是使用 `getchar()`。使用 `getchar()` 读取字符的一般方法如下：

```
while((x=getchar())!=EOF) {
    /* ... */
}
```

`EOF` 是一个 `stdio` 宏，它表示“文件尾”的含义，参见 27.4 节。

C++ 标准库中的流与 `scanf("%s")` 和 `get()` 功能类似，但不存在上述问题：

```
string s;
cin >> s;      // 读一个词
getline(cin,s); // 读一行
```

### 27.6.3 文件

在 C (或 C++) 中，可以使用 `fopen()` 打开文件，使用 `fclose()` 关闭文件。这些函数与文件描述符结构 `FILE` 及 `EOF` 宏 (文件尾) 都定义在 `<stdio.h>` 中：

```
FILE *fopen(const char* filename, const char* mode);
int fclose(FILE *stream);
```

可以这样使用文件：

```
void f(const char* fn, const char* fn2)
{
    FILE* fi = fopen(fn, "r");      /* 以读方式打开 fn */
    FILE* fo = fopen(fn2, "w");    /* 以写方式打开 fn2 */
    if (fi == 0) error("failed to open input file");
    if (fo == 0) error("failed to open output file");

    /* 用 stdio 输入函数从文件读，例如 getc() */
    /* 用 stdio 输出函数向文件写，例如 fprintf() */

    fclose(fo);
    fclose(fi);
}
```

考虑这样一个问题：C 中没有异常机制，那么在发生错误的情况下，我们如何保证文件确实被关闭了？

## 27.7 常量和宏

在 C 中，`const` 绝不是编译时常量：

```

const int max = 30;
const int x;      /* 常量未初始化: C 中合法 (C++ 中错误) */

void f(int v)
{
    int a1[max]; /* 错误: 数组大小不是常量 (C++ 中合法) */
                  /* (max 没有以常量表达式形式定义!) */
    int a2[x];   /* 错误: 数组大小不是常量 */

    switch (v) {
    case 1:
        /* ... */
        break;
    case max: /* 错误: case 标号不是常量 (C++ 中合法) */
        /* ... */
        break;
    }
}

```

在 C 中这样规定 (与 C++ 不同), 是因为考虑到技术上的原因: `const` 隐含地可被所有源文件访问。

```

/* 文件 x.c: */
const int x;      /* 在其他地方初始化 */
/* 文件 xx.c: */
const int x = 7;  /* 这是实际定义的地方 */

```

在 C++ 中, 这是两个不同的对象, 名字都是 `x`, 作用域在自己的文件中。C 程序员不是用 `const` 来表示符号常量, 他们更愿意使用宏。例如:

```

#define MAX 30

void f(int v)
{
    int a1[MAX]; /* OK */

    switch (v) {
    case 1:
        /* ... */
        break;
    case MAX: /* OK */
        /* ... */
        break;
    }
}

```

程序中凡是使用宏名 `MAX` 的地方, 都被替换为文本 `30` (宏的值)。也就是说, `a1` 的元素数目为 `30`, 第二个 `case` 语句的值也是 `30`。我们为宏取名时使用了全部大写的字符串 `MAX`, 这是 C 语言的惯例。这种命名习惯有助于减少宏引起的错误。

## 27.8 宏

使用宏的时候一定要小心: 在 C 中没有真正有效的方法来避免使用宏, 但宏带有严重的副作用, 因为宏不遵守通常的 C (或 C++) 作用域和类型规则——它只是一种文本替换。参见附录 A.17.2。

除了尽量不用宏 (使用 C++ 中的替代方法) 之外, 我们还有什么办法来避免宏引起的问

题吗?

- 所有宏名全部大写。
- 不是宏的结构不要使用全部大写的名字。
- 不要为宏取短的或“有趣”的名字，如 max 或 min。
- 期望其他人也遵守上述简单而常见的规范。

宏的主要用途包括：

- 定义“常量”。
- 定义类似函数的结构。
- “改进”语法。
- 控制条件编译。

另外，还有其他很多不太常见的用途。

我们认为宏被过度使用了，但在 C 程序中，还没有一种合理而完整的替代方法。甚至在 C++ 程序中也很难避免使用宏（特别是当你编写的程序需要移植到很老的编译器上或者有特殊限制的平台时）。

对于那些认为下面介绍的技术是“低级手段”，不应该在此提及的人，我们要说声抱歉了。因为我们认为这种编程技术是现实世界中真实存在的，而且我们所选择的这些（很温和的）例子展示了宏的正确使用和不正确使用，可以帮助初学者避免长时间陷入困境。对宏的愚昧无知并非一件好事。

### 27.8.1 类函数宏

下面是一个非常典型的类函数宏：

```
#define MAX(x, y) ((x)>=(y)?(x):(y))
```

我们为宏取名为全大写字母的 MAX，以便与（各种程序中）常用的函数名 max 区别开来。显然，它与函数还是有很大区别的：没有参数类型、没有语句块、没有返回语句等。另外，宏定义中的那些括号是起什么作用的？考虑如下代码：

```
int aa = MAX(1,2); -
double dd = MAX(aa++,2);
char cc = MAX(dd,aa)+2;
```

宏替换后，程序扩展为：

```
int aa = ((1)>=( 2)?(1):(2));
double dd = ((aa++)>=(2)?( aa++):(2));
char cc = ((dd)>=(aa)?(dd):(aa))+2;
```

如果在宏定义中没有使用“那些括号”，最后一条语句会扩展为：

```
char cc = dd>=aa?dd:aa+2;
```

也就是说，cc 的值将和你根据其定义推断出的值不同。这个例子说明，在宏的定义中，使用任何参数时都应将其置于括号之中（当作表达式）。

另一方面，对于第二条语句，使用再多括号也解决不了问题。宏参数 x 被替换为 aa++，由于 x 在 MAX 使用了两次，因此 x 进行了两次增 1 运算。记住，不要向宏传递可能引起副作用的参数。

某些天才可能碰巧定义了这样有问题的宏，并将其放入了被广泛使用的头文件中。更不

幸的是，他还将宏命名为 `max` 而不是 `MAX`，这样，当 C++ 标准头文件中定义下面函数时：

```
template<class T> inline T max(T a,T b) { return a<b?b:a; }
```

`max(T a,T b)` 就会被扩展，在编译器看来，语句变为：

```
template<class T> inline T ((T a)>=( T b)?( T a):( T b)) { return a<b?b:a; }
```

编译器给出的错误信息会非常“有趣”，对程序员修正错误毫无帮助。如果遇到这种紧急情况，你可以“取消定义”(undefine)：

```
#undef max
```

幸运的是，这个宏并不是那么重要。但是，在那些广泛应用的头文件中有成千上万个宏，不可能取消每个宏都不引起混乱。

并不是所有宏参数都被用作表达式，例如：

```
#define ALLOC(T,n) ((T*)malloc(sizeof(T)*n))
```

这是来自实际程序中的例子，内存分配时 `sizeof` 中使用的类型与所需类型可能不匹配，这个宏对避免此类错误很有用：

```
double* p = malloc(sizeof(int)*10); /* 可能错误 */
```

不幸的是，如果希望宏还能捕获内存耗尽错误，就不那么好办了。假如已经定义了 `error_var` 和 `error()`，可以这样定义宏：

```
#define ALLOC(T,n) (error_var = (T*)malloc(sizeof(T)*n), \
                    (error_var==0)\
                    ?(error("memory allocation failure"),0)\
                    :error_var)
```

行结尾处的 `\` 并非输入错误，将一个较长的宏分成几行，就必须在非结尾行的最后加上 `\`。如果你使用 C++ 编写程序，建议还是使用 `new`。

## 27.8.2 语法宏

你可以定义这样一类宏，它们能使源程序形式上更符合你的偏好，例如：

```
#define forever for(;;)
#define CASE break; case
#define begin {
#define end }
```

我们强烈建议不要使用这种宏。很多人已经尝试过这种方法了。他们（以及他们编写出的代码的维护人员）发现：

- 对于“好的语法”，很多人的理解是不一样的。
- “改进的语法”是不标准的、奇怪的，会令他人困惑。
- 有过“改进语法”导致难以发现的编译错误的先例。
- 你所看到的并非编译器所看到的，编译器是根据它所知道的（以及在源程序中所看到的）词汇报告错误，而不是根据你所知及你所见。

因此，不要使用语法宏“改进”代码外观。你和你的好朋友可能觉得效果很棒，但经验表明，你只是大社群中的一份子而已，因而其他人将不得不重写你的代码（假如你的代码还“活着”的话）。

### 27.8.3 条件编译

假设某个头文件有两个版本，比如说一个是 Linux 版，另一个是 Windows 版。在程序中你如何选择使用哪个版本呢？常用方法如下：

```
#ifndef WINDOWS
    #include "my_windows_header.h"
#else
    #include "my_linux_header.h"
#endif
```

现在，如果有人人在编译之前定义了宏 `WINDOWS`，则效果为：

```
#include "my_windows_header.h"
```

否则，效果为：

```
#include "my_linux_header.h"
```

`#ifndef WINDOWS` 并不关心 `WINDOWS` 被定义成什么，它只关心 `WINDOWS` 是否被定义。

很多大型系统（包括所有操作系统）都会定义类似 `WINDOWS` 这样的宏，以供你检测。我们可以这样来检测程序是在被 C++ 编译器编译还是在被 C 编译器编译：

```
#ifdef __cplusplus
    // in C++
#else
    /* in C */
#endif
```

还有一种类似的结构，通常被人们称为包含保护（include guard），常常用来防止头文件被包含多次：

```
/* my_windows_header.h: */
#ifndef MY_WINDOWS_HEADER
#define MY_WINDOWS_HEADER
    /* 下面是头文件内容 */
#endif
```

`#ifndef` 检测宏是否未被定义，即它与 `#ifdef` 是相对的。逻辑上，用于源文件控制的宏与其他修改源码（宏替换）的宏有很大不同。它们只是使用了相同的下层语言机制。

## 27.9 实例：侵入式容器

C++ 标准库容器（如 `vector` 和 `map`）是非侵入式容器（non-intrusive container），即它们不要求容器内的数据以单个元素的形式被访问，而是对容器整体进行操作。这也是它们为什么有那么好的通用性——适用于所有内置类型和用户自定义类型，只要类型支持拷贝操作即可。另外一类容器被称为侵入式容器（intrusive container），在 C 和 C++ 中都很常用。下面我们将通过一个非侵入式的容器来说明 C 风格 `struct`、指针和动态内存分配的使用。

我们可以定义一个支持如下九个操作的双向链表：

```
void init(struct List* lst);          /* 初始化 lst 为空 */
struct List* create();               /* 创建一个新的空链表用于空间分配 */
void clear(struct List* lst);        /* 释放 lst 的所有元素的空间 */
void destroy(struct List* lst);      /* 释放 lst 的所有元素的空间，然后释放 lst 本身 */

void push_back(struct List* lst, struct Link* p); /* 将 p 加到 lst 尾部 */
void push_front(struct List*, struct Link* p);   /* 将 p 加到 lst 头部 */
```

```

/* 在 lst 中将 q 插入 p 之前 */
void insert(struct List* lst, struct Link* p, struct Link* q);
struct Link* erase(struct List* lst, struct Link* p); /* 在 lst 中删除 p */

/* 返回 p 之前或之后第 n 个元素: */
struct Link* advance(struct Link* p, int n);

```

基本设计思路是让用户只需提供 List\* 和 Link\* 指针就能完成这些操作。这意味着可以大幅修改这些操作的实现，而无须修改用户程序。显然，我们在命名上受到了 STL 的影响。List 和 Link 显然可以简单定义如下：

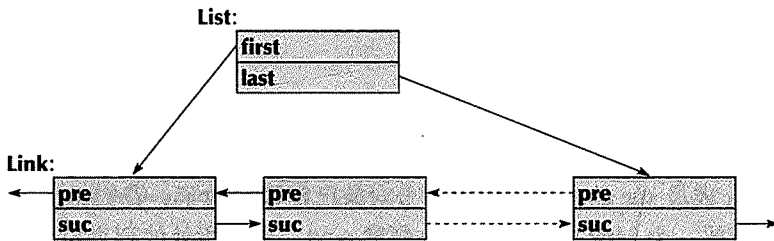
```

struct List {
    struct Link* first;
    struct Link* last;
};

struct Link { /* 双向链表的链接 */
    struct Link* pre;
    struct Link* suc;
};

```

下面是 List 的图示：



我们目的不是介绍高明的描述技术或者高明的算法，因此本节并未展示这些。但是，请注意程序中并未提及 Link 所保存的数据（List 中的元素）。回过头看一下程序，我们发现 Link 和 List 非常像抽象类。Link 保存的数据会随后提供。Link\* 和 List\* 有时被称为不透明类型处理工具，即我们可以在不了解 Link 和 List 的内部结构的情况下，使用 Link\* 和 List\* 来处理 List 中的元素。

为了实现 List 函数，我们首先要用 #include 包含一些标准库头文件：

```

#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

```

C 不支持名字空间，因此我们不必担心 using 声明或者 using 指令的问题。另一方面，我们应该担心的可能是那些非常常见的简单名字（Link、insert、init 等），因此这组函数不应在这个玩具程序之外使用。

初始化代码很简单，但注意 assert() 的使用：

```

void init(struct List* lst) /* *lst 初始化为空链表 */
{
    assert(lst);
    lst->first = lst->last = 0;
}

```

我们决定在运行时不处理非法链表指针错误。通过使用 assert()，我们只是对空链表指针

给出一个（运行时）系统错误。“系统”错误会给出失败的 `assert()` 所在的文件名和行号；`assert()` 是在 `<assert.h>` 中定义的宏，其检测只在调试状态下才执行。由于 C 语言不支持异常，处理非法指针是很困难的。

函数 `create()` 简单地在动态内存空间中创建一个 `List`。它在某种程度上是构造函数（`init()` 进行初始化）和 `new`（`malloc()` 完成内存分配）的结合：

```
struct List* create()          /* 创建一个新的空链表 */
{
    struct List* lst = (struct List*)malloc(sizeof(struct List));
    init(lst);
    return lst;
}
```

函数 `clear()` 假定所有 `Link` 的内存空间都是动态分配的，因此用 `free()` 来释放：

```
void clear(struct List* lst)    /* 释放 lst 的所有元素 */
{
    assert(lst);
    {
        struct Link* curr = lst->first;
        while(curr) {
            struct Link* next = curr->suc;
            free(curr);
            curr = next;
        }
        lst->first = lst->last = 0;
    }
}
```

注意我们使用 `Link` 成员 `suc` 的方法。如果一个对象已经被释放了，我们是无法安全访问其成员的。因此，在释放一个 `Link` 时，我们引入变量 `next`，保存所处的列表位置。

如果我们并不是在动态内存空间中分配全部 `Link`，那么最好不要调用 `clear()` 来释放链表内存空间，否则会造成很大的混乱。

`destroy()` 本质上与 `create()` 是相对的，也就是说，它是析构函数和 `delete` 的结合：

```
void destroy(struct List* lst) /* 释放 lst 的所有元素的空间，然后释放 lst 本身 */
{
    assert(lst);
    clear(lst);
    free(lst);
}
```

注意，我们没有准备为链表元素调用清理函数（析构函数）。也就是说，目前的设计并非是 C++ 技术和普遍方法的准确模拟，实际上，我们不能也不必这样做。

函数 `push_back()` 的设计思路非常直接——向链表中添加一个 `Link`，作为新的表尾：

```
void push_back(struct List* lst, struct Link* p) /* 将 p 加到 lst 的末尾 */
{
    assert(lst);
    {
        struct Link* last = lst->last;
        if (last) {
            last->suc = p;          /* 将 p 加到 last 之后 */
            p->pre = last;
        }
        else {
```

```

        lst->first = p;          /* p 是第一个元素 */
        p->pre = 0;
    }
    lst->last = p;             /* p 是新的最后元素 */
    p->suc = 0;
}
}

```

但是，如果我们不在草稿纸上画一些方块（链表节点）和箭头（链表指针）来分析链表的操作方式，是很难直接写出正确的代码的。注意，在上述代码中，我们“忘记”考虑参数 *p* 为空的情况。将 0 而不是一个合法指针传递给 Link，这段代码就会崩溃。这段代码谈不上糟糕，但它不是一个工业级别的代码。其目的是说明常用的、有用的技术，在本例中，它的另一目的是展示一种常见的弱点 /bug。

函数 `erase()` 可以这样编写：

```

struct Link* erase(struct List* lst, struct Link* p)
/*
    从 lst 中删除 p
    在 p 后返回一个链接的指针
*/
{
    assert(lst);
    if (p==0) return 0;          /* 允许 erase(0) */

    if (p == lst->first) {
        if (p->suc) {
            lst->first = p->suc;    /* 后继变为 first */
            p->suc->pre = 0;
            return p->suc;
        }
        else {
            lst->first = lst->last = 0;    /* 链表变为空 */
            return 0;
        }
    }
    else if (p == lst->last) {
        if (p->pre) {
            lst->last = p->pre;    /* 前驱变为 last */
            p->pre->suc = 0;
        }
        else {
            lst->first = lst->last = 0;    /* 链表变为空 */
            return 0;
        }
    }
    else {
        p->suc->pre = p->pre;
        p->pre->suc = p->suc;
        return p->suc;
    }
}

```

我们将剩余函数的编写作为练习，在我们非常简单的测试中也用不到这些函数。但是，现在我们必须面对目前设计中最费解的一个问题：链表元素中的数据在哪里？例如，我们想实现一个保存名字（C 风格字符串）的链表，应该怎么做？考虑下面代码：



```

struct Name {
    struct Link link;    /* 链表操作需要的 Link */
    char* p;            /* 字符串名 */
};

```

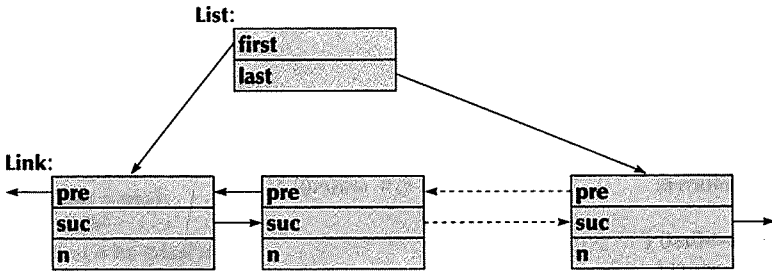
到目前为止，一切尚好，只是我们还没弄清如何使用 Name 的 Link 成员。不过由于我们知道 List 希望其中的 Link 在动态空间中分配内存，因此我们可以编写函数，在动态空间中创建 Name。

```

struct Name* make_name(char* n)
{
    struct Name* p = (struct Name*)malloc(sizeof(struct Name));
    p->p = n;
    return p;
}

```

下图描述了链表的结构：



下面程序展示了其使用方式：

```

int main()
{
    int count = 0;
    struct List names;    /* 创建链表 */
    struct List* curr;
    init(&names);

    /* 给一些名字，并加入链表： */
    push_back(&names, (struct Link*)make_name("Norah"));
    push_back(&names, (struct Link*)make_name("Annemarie"));
    push_back(&names, (struct Link*)make_name("Kris"));

    /* 删除第二个名字： */
    erase(&names, advance(names.first, 1));

    curr = names.first;    /* 写出所有名字 */
    for (; curr != 0; curr = curr->suc) {
        count++;
        printf("element %d: %s\n", count, ((struct Name*)curr)->p);
    }
}

```

可以看出，我们使用了“欺骗手段”。我们将 Name\* 转换为 Link\*。通过这种方式，用户程序能够获得“库类型” Link。然而，“库”却不会（也不必）知道“用户程序类型” Name。这种方法是允许的吗？是的，这是允许的：在 C（和 C++）中，可以将一个 struct 指针当作其第一个成员的指针来处理，反之亦然。

显然，本例也是合法的 C++ 程序。

## 试一试

C++ 程序员常对 C 程序员说的一句话是：“你所能做的每件事，我都能做得更好！”请用 C++ 语言重写侵入式 List 程序来展示如何用更短、更简单的代码实现相同的功能，又不会牺牲速度和内存空间。

## 简单练习

1. 用 C 语言编写 “Hello, World!” 程序，编译、运行它。
2. 定义两个变量，分别保存 “Hello” 和 “World!”，将两个字符串连接在一起，中间加入一个空格，并输出为 “Hello World!”。
3. 定义一个 C 函数，它接受两个参数：一个为 char\* 类型，名为 p；另一个为 int 型，名为 x。函数输出两个参数的值，形式为：p is "foo" and x is 7。用一些实际参数来测试这个函数。

## 思考题

在下面的题目中，假定 C 表示 ISO 标准 C89。

1. C++ 是 C 的子集吗？
2. 谁发明了 C？
3. 说出一本获得极高声誉的 C 教科书。
4. C 和 C++ 是哪个机构发明出来的？
5. 为什么 C++ 与 C（几乎）兼容？
6. 为什么 C++ 只是与 C 几乎兼容？
7. 列出十几个 C 不支持的 C++ 特性。
8. 现在哪个组织 “拥有” C 和 C++？
9. 列出 6 个 C 中无法使用的 C++ 标准库功能。
10. 哪些 C 标准库功能在 C++ 中可以使用？
11. 如何在 C 中实现函数参数类型检查？
12. 哪些 C++ 特性相关的函数在 C 中不存在？列出至少 3 个，并举实例。
13. 如何从 C++ 程序调用 C 函数？
14. 如何从 C 程序调用 C++ 函数？
15. 哪些类型的内存布局在 C 和 C++ 中是兼容的？请举例。
16. 什么是结构标签？
17. 列出 20 个 C 中不存在的 C++ 关键字。
18. “int x;” 在 C++ 中是一个定义吗？在 C 中呢？
19. 什么是 C 风格类型转换？它为什么是危险的？
20. void\* 是什么？它在 C 和 C++ 中有什么不同？
21. 枚举类型在 C 和 C++ 中有什么不同？
22. 在 C 中如何才能避免常用名字所引起的连接问题？
23. C 中最常用的动态内存空间相关函数是哪三个？

24. C 风格字符串的定义是什么?
25. 对于 C 风格字符串, == 和 strcmp() 有何不同?
26. 如何拷贝 C 风格字符串?
27. 如何获得一个 C 风格字符串的长度?
28. 如何拷贝一个大的 int 数组?
29. printf() 的优点是什么? 它的问题 / 局限呢?
30. 为什么永远不要使用 gets()? 替代方法是什么?
31. 在 C 中如何打开一个文件?
32. const 在 C 和 C++ 中有何区别?
33. 我们为什么不喜欢宏?
34. 宏的常见用途是什么?
35. 包含保护是什么?

## 术语

#define	K&R
#ifdef	lexicographical (字典序)
#ifndef	linkage (连接)
Bell Labs (贝尔实验室)	macro (宏)
Brian Kernighan	malloc()
C/C++	non-intrusive (非侵入式)
compatibility (兼容性)	opaque type (不透明类型)
conditional compilation (条件编译)	overloading (重载)
C-style cast (C 风格类型转换)	printf()
C-style string (C 风格字符串)	strcpy()
Dennis Ritchie	structure tag (结构标签)
FILE	three-way comparison (三路比较)
fopen()	void
format string (格式字符串)	void*
intrusive (侵入式)	

## 习题

对于本章的习题, 最好对所有程序都同时在 C 和 C++ 两种编译器下编译。如果只使用 C++ 编译器, 你可能无意中使用了 C 不支持的特性。如果只使用 C 编译器, 类型错误可能无法被检测到。

1. 实现 strlen()、strcmp() 和 strcpy()。
2. 将 27.9 节中的侵入式 List 程序补充完整, 并测试所有函数。
3. 尽可能地“美化”27.9 节中的侵入式 List 程序, 使之更易使用。捕获 / 处理尽量多的错误。改变 struct 定义的细节, 使用宏等方法都是合理的。
4. 为 27.9 节中的侵入式 List 程序编写 C++ 版本, 并测试每个函数。
5. 比较习题 3 和习题 4 的结果。

6. 改变 27.9 节中 Link 和 List 的实现，但不改变用户函数接口。在一个数组中为 Link 分配空间，并将其成员 first、last、pre 和 suc 定义为 int 类型（数组下标）。
7. 与 C++ 标准库容器（非侵入式）相比，侵入式容器的优点和缺点是什么？列出优缺点。
8. 你的机器中的字典序是怎样的？输出你的键盘上的每个字符及其整型值。然后，按整型值的顺序输出所有字符。
9. 只使用 C 语言特性和 C 标准库，从 stdin 读入一个单词序列，然后按字典序将它们输出到 stdout。提示：C 中的排序函数称为 qsort()，查找它是在哪里定义的，使用它来完成题目。另一种方法是，每读入一个单词，就将它插入已排序的列表中。C 标准库中未定义列表结构。
10. 列出从 C++ 语言或支持类的 C 语言中借鉴来的 C 语言特性（见 27.1 节）。
11. 列出没有被 C++ 采纳的 C 特性。
12. 实现一个支持查找功能的表结构，每个表项保存一个（C 风格字符串，int）对，支持 find(struct table\*, const char\*)、insert(struct table\*, const char\*, int) 以及 remove(struct table\*, const char\*) 等操作。表可以用一个 struct 数组或者一对数组（const char\* 和 int\*）来保存，你可以选择其中一种方式。函数返回类型也由你选择。编写文档，将你的设计决策描述清楚。
13. 编写 C 程序，实现 string s; cin>>s; 相同的功能。也就是说，定义一个输入操作，读入任意长度的以空白符结尾的字符序列，存入以 0 结尾的 char 数组中。
14. 编写函数，接受一个 int 数组参数，找出其中的最小值和最大值，并计算均值和中值。使用一个 struct 保存结果，返回值就设定为这个 struct。
15. 在 C 中模拟出单重继承。令每个“基类”包含一个指向指针数组的指针（用一组独立函数模拟虚函数，每个函数的第一个参数为指向一个“基类”对象的指针），参考 27.2.3 节。“派生”机制实现方式为：将派生的第一个成员定义为“基类”类型。对每个类，恰当地对“虚函数”数组进行初始化。为了测试这种模拟方式，用它实现“Shape”，基类的派生类的 draw() 只是简单地打印类名。在完成本题的过程中，只允许使用标准 C 特性和 C 标准库功能。
16. 使用宏简化上一题中的符号。

## 附言

我们曾经提到，兼容性问题不那么令人兴奋。然而，已经有大量的（数十亿行）C 代码“在那里”了，如果你必须阅读或编写 C 代码，本章向你介绍了一些预备知识。从个人角度，我更倾向于使用 C++，本章的一些内容也给出了部分原因。请不要低估“侵入式 List”例程，“侵入式 List”和不透明类型在 C 和 C++ 中都是非常重要和强大的工具。

# 标准库概要

如果可能，所有复杂性都应埋藏于视野之外。

—David J. Wheeler

本附录概述重要的 C++ 标准库特性。本附录内容都是精心选择的，特别适合于那些希望接触一些本书之外内容的初学者。

## C.1 概述

本附录的目的是作为补充参考资料，而不是像其他章节一样需要从头到尾仔细阅读。它（或多或少地）系统描述了 C++ 标准库的一些重要特性。本附录不是完整的参考资料，而只是一些重要特性的概述。通常，你需要查看相关章节来获得更为详细完整的解释。注意，本附录不追求与 C++ 标准相同的精确性和术语，而是追求易于查阅。更详细的信息可参考 Stroustrup 的《The C++ Programming Language》一书。ISO C++ 标准中有标准库的完整定义，但它并不是为了初学者所编写的，因此不适合入门阅读学习。不要忘了使用联机帮助来查找标准库的有关内容。

一个选择性的（因而不完整的）概要介绍有什么用处呢？它的用处是，你可以从中快速查找已经知道的特性，也可以快速浏览一节来了解标准库中有哪些常用的特性。你可能必须到其他地方查找细节内容，但这没有关系：通过本附录，你已经获得了“查找什么”的线索。而且，本附录包含了交叉引用，你可以迅速找到包含详细内容的章节。本附录是 C++ 标准库特性的一个简洁概述。请不要尝试记忆本附录中的内容，这不是本附录的目的，相反，本附录是一个工具，能帮助你避免形成错误的记忆。

你可以在本附录中找到所需要的有用的特性，不要试图自己重新发明。标准库中的所有特性（特别是本附录中所提到的特性），已经被证明对很多程序员来说都是有用的工具。标准库中的工具，几乎总是比你仓促设计实现出的工具有着更为良好的设计、实现、文档以及更好的可移植性。因此，只要可能，你应该优先使用标准库特性，而不是“自行制造”。这样做，你的代码就更容易被他人所理解。

如果你是个理智的人，你会觉得本附录介绍的内容太多了，不要担心，忽略那些你不需要的内容就是了。如果你是个“细节狂人”，你会觉得本附录缺少了很多内容，但是，完整性是那些专家级教材和指南的目标，而且联机帮助中已经有足够完整的信息了。无论你是哪种人，你都会发现很多看来很神秘，而且可能很有趣的内容。试着认真研究其中一些内容！

### C.1.1 头文件

标准库的接口都是在头文件中定义的。请将本节作为 C++ 标准库的一个概览，它可以帮助你找到某个功能的定义和描述在哪里。

## STL (容器、迭代器和算法)

---

<algorithm>	算法; sort()、find() 等等 (见附录 C.5 和 16.1 节)
<array>	固定大小数组 (见 15.9 节)
<bitset>	bool 数组 (见 25.5.2 节)
<deque>	双端队列
<functional>	函数对象 (见附录 C.6.2)
<iterator>	迭代器 (见附录 C.4.4)
<list>	双向链表 (见附录 C.4 和 15.4 节)
<forward_list>	单向链表
<map>	(key, value) 的 map 和 multimap (见附录 C.4 和 16.6.1 ~ 16.6.3 节)
<memory>	供容器用的分配器
<queue>	queue 和 priority_queue
<set>	set 和 multiset (见附录 C.4 和 16.6.5 节)
<stack>	stack
<unordered_map>	哈希映射 (见 16.6.4 节)
<unordered_set>	哈希集合
<utility>	运算符和 pair (见附录 C.6.3)
<vector>	vector (可动态扩展)(见附录 C.4 和 15.8 节)

---

## I/O 流

---

<iostream>	I/O 流对象 (见附录 C.7)
<fstream>	文件流 (见附录 C.7.1)
<sstream>	string 流 (见附录 C.7.1)
<iosfwd>	声明 (但未定义) I/O 流工具
<ios>	I/O 流基类
<streambuf>	流缓冲
<istream>	输入流 (见附录 C.7)
<ostream>	输出流 (见附录 C.7)
<iomanip>	格式化和操纵符 (见附录 C.7.6)

---

## 字符串操作

---

<string>	string (见附录 C.8.2)
<regex>	正则表达式 (见第 23 章)

---

## 数值计算

---

<complex>	复数及其运算 (见附录 C.9.3)
<random>	随机数发生器 (见附录 C.9.6)
<valarray>	数值数组
<numeric>	通用数值算法, 如 accumulate() (见附录 C.9.5)
<limits>	数值限制 (见附录 C.9.1)

---

工具和语言支持	
<exception>	异常类型 (见附录 C.2.1)
<stdexcept>	异常层次 (见附录 C.2.1)
<locale>	本地化格式
<typeinfo>	标准类型信息 (从 typeid 获取)
<new>	内存分配和释放函数
<memory>	资源管理指针, 如 unique_ptr (见附录 C.6.5)
并发支持	
<thread>	线程 (超出了书范围)
<future>	线程间通信 (超出了本书范围)
<mutex>	互斥机制 (超出了本书范围)
C 标准库	
<cstring>	C 风格字符串操作 (见附录 C.11.3)
<cstdio>	C 风格 I/O (见附录 C.11.2)
<ctime>	clock()、time() 等等 (见附录 C.11.5)
<cmath>	标准浮点数学函数 (见附录 C.9.2)
<cstdlib>	其他函数: abort()、abs()、malloc()、qsort() 等 (见第 27 章)
<cerrno>	C 风格错误处理 (见 24.8 节)
<cassert>	断言宏 (见 27.9 节)
<locale>	本地化格式
<climits>	C 风格数值限制 (见附录 C.9.1)
<float>	C 风格浮点限制 (见附录 C.9.1)
<stddef>	C 语言支持: size_t 等
<stdarg>	可变参数宏
<setjmp>	setjmp 和 longjmp() (不要使用)
<signal>	信号处理
<wchar>	C 的宽字符
<cctype>	字符分类 (见附录 C.8.1)
<ctype>	宽字符分类

每个 C 标准库头文件都有一个不带开始字母 c 并有 .h 后缀的版本, 例如 <ctime> 的另一个版本是 <time.h>。 .h 版本定义的名字都位于全局空间, 而不是 std 名字空间。

在本附录下面几节中 (以及前面一些章节中), 我们将介绍这些头文件中的一些 (但不是所有) 特性。如果你希望了解更多内容, 请查阅联机帮助或者专家级 C++ 书籍。

## C.1.2 名字空间 std

标准库特性都定义于名字空间 std 中, 因此使用时需显式使用限定符, 如使用 using 声明或者 using 指令:

```
std::string s;           // 显式限定

using std::vector;      // using 声明
vector<int>v(7);

using namespace std;    // using 指令
map<string,double> m;
```

在本书中，我们采用 `using` 指令。使用 `using` 指令要有节制，见附录 A.15。

### C.1.3 描述风格

如果想完整描述一个标准库操作，即使是一个简单操作，如一个构造函数或者一个算法，都要花费几页的篇幅。因此，我们采用一种非常简化的描述风格。例如：

描述方式示例	
<code>p=op(b,e,x)</code>	对区间 <code>[b:e)</code> 和 <code>x</code> 进行操作 <code>op</code> ，将返回值赋予 <code>p</code>
<code>foo(x)</code>	对 <code>x</code> 进行操作 <code>foo</code> ，不返回结果
<code>bar(b,e,x)</code>	对 <code>x</code> 和区间 <code>[b:e)</code> 进行一些操作，结果为真？

我们尽量使用一些含义明确的助记符，如 `b`、`e` 表示迭代器，指出了范围；`p` 表示指针或者迭代器；`x` 表示值；当然，所有这些含义都与上下文相关。在这种表示方法中，只有借助注释才能分辨不返回值和返回布尔值这两种情况，因此有可能产生混淆。对于返回 `bool` 值的操作，注释通常以问号结束。

如果算法遵循习惯方式，通过返回输入序列结束标记来表示“故障”“未找到”等含义（见 C.3.1），我们不再重复说明。

## C.2 错误处理

标准库中不同部分的开发时间可能相差 40 年，因此错误处理的风格和方法是不一致的。

- C 标准库由函数构成，这些函数中很多都是通过设置 `errno` 来表示发生了错误，见 24.8 节。
- 很多对元素序列进行操作的算法会返回一个指向末尾元素之后位置的迭代器，来表示“未找到”或“错误”。
- I/O 流库依赖每个流中的状态来指示错误，而且可能会抛出异常来表示错误发生（如果用户要求这样的话），见 10.6 节和附录 C.7.2。
- 一些标准库特性，如 `vector`、`string` 和 `bitset`，通过抛出异常来表示错误。

标准库的一个设计原则就是，所有的特性都要遵循“基本保证”（见 14.5.3 节）——即使发生错误，抛出了异常，也不会发生资源泄漏（如内存泄漏）以及破坏标准库类的不变式。

### C.2.1 异常

一些标准库特性通过抛出异常来表示错误：

标准库异常	
<code>bitset</code>	抛出 <code>invalid_argument</code> 、 <code>out_of_range</code> 、 <code>overflow_error</code>
<code>dynamic_cast</code>	如果无法进行转换，抛出 <code>bad_cast</code>
<code>istream</code>	如果异常开启的话，在错误时抛出 <code>ios_base::failure</code>
<code>new</code>	如果无法分配内存，抛出 <code>bad_alloc</code>
<code>regex</code>	抛出 <code>regex_error</code>
<code>string</code>	抛出 <code>length_error</code> 、 <code>out_of_range</code>
<code>typeid</code>	如果无法获得 <code>type_info</code> ，抛出 <code>bad_typeid</code>
<code>vector</code>	抛出 <code>out_of_range</code>



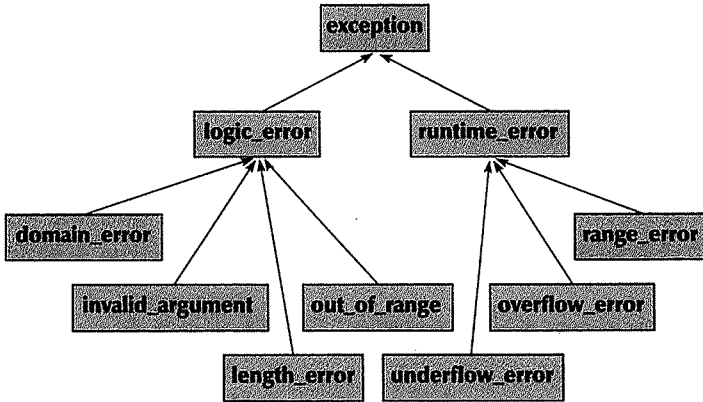
在任何直接或间接使用这些特性的代码中，都可能遇到这些异常。除非你确定你使用这些特性的方式不会产生异常，否则最好保证在某处（比如在 main 中）捕获标准库异常类层次中的根类（如 exception），这样就不会遗漏任何异常。

我们强烈建议你不要抛出内置类型，如 int 或 C 风格字符串，应该抛出专门定义用作异常的类型对象。比如，可以使用从标准库类 exception 派生出的类：

```
class exception {
public:
    exception();
    exception(const exception&);
    exception& operator=(const exception&);
    virtual ~exception();
    virtual const char* what() const;
};
```

函数 what() 可以用来获取一个字符串，这个字符串说明了导致异常的错误是什么。

通过下面的异常分类，我们可以很好地了解标准库异常类的层次关系：

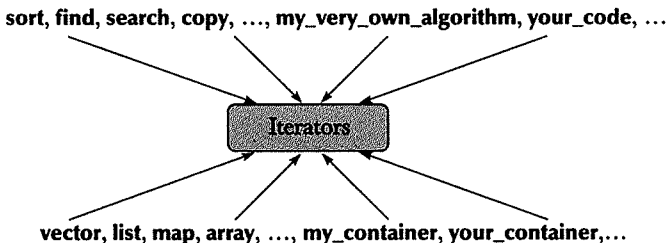


我们可以像下面这样通过派生标准库异常来定义自己的异常类：

```
struct My_error : runtime_error {
    My_error(int x) : interesting_value(x) {}
    int interesting_value;
    const char* what() const override { return "My_error"; }
};
```

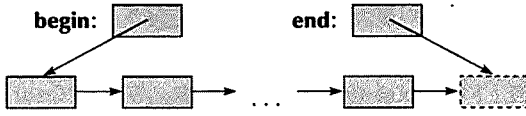
### C.3 迭代器

迭代器是联系标准库算法及其数据的纽带。从相反的角度，你也可以说迭代器是最小化算法和它所处理的数据之间依赖关系的机制（见 15.3 节）：



### C.3.1 迭代器模型

迭代器提供了间接访问数据元素（如用 \* 解引用）以及移动到新元素（如用 ++ 移动到下一元素）的能力，在这些方面它与指针很接近。我们可以用两个迭代器所构成的半开区间 [begin:end) 来定义一个元素序列：



即，begin 指向序列的第一个元素，而 end 指向序列最后一个元素之后的位置。因此，不要读写 \*end。注意，对于空序列 begin==end；即，对任意 p，[p:p) 都表示空序列。

读取序列内容的一般方法是：使用一对迭代器 (h, e)，通过 ++ 来遍历序列，直至到达末尾 e：

```
while (h!=e) { // 使用 != 而不是 <
    // 进行一些操作
    ++h; // 移动到下一个元素
}
```

在序列中进行搜索的算法通常返回指向序列末尾的迭代器，来表示“未找到”，例如：

```
p = find(v.begin(),v.end(),x); // 在 v 中查找 x
if (p!=v.end()) {
    // 在位置 p 找到了 x
}
else {
    // 在范围 [v.begin():v.end()) 中未找到 x
}
```

见 15.3 节。

向序列中写入数据一般只需一个指向首元素的迭代器，保证不超出序列末尾是程序员的责任。例如：

```
template<class Iter> void f(Iter p, int n)
{
    while (n>0) *p++ = --n;
}
vector<int> v(10);
f(v.begin(),v.size()); // 正确
f(v.begin(),1000); // 大麻烦
```

一些标准库实现了范围检查，即对于上面程序中最后一次 f() 调用，会抛出一个异常。但是，当你编写可移植的程序时，不能假定标准库提供了这一功能，有很多实现并不支持范围检查。

我们可以对迭代器进行如下运算：

---

#### 迭代器运算

---

++p	前增：使 p 指向序列中的下一个元素或者尾元素之后的位置（“前进一个元素”），表达式的值为 p+1
-----	---

---

(续)

迭代器运算	
<code>p++</code>	后增: 使 <code>p</code> 指向序列中的下一个元素或者尾元素之后的位置 (“前进一个元素”), 表达式的值为 <code>p</code> (操作之前的值)
<code>--p</code>	前减: 使 <code>p</code> 指向前一个元素 (“后退一个元素”), 表达式值为 <code>p-1</code>
<code>p--</code>	后减: 使 <code>p</code> 指向前一个元素 (“后退一个元素”), 表达式值为 <code>p</code> (操作之前的值)
<code>*p</code>	访问 (解引用): <code>*p</code> 引用 <code>p</code> 指向的元素
<code>p[n]</code>	访问 (下标): <code>p[n]</code> 引用 <code>p+n</code> 指向的元素, 等价于 <code>*(p+n)</code>
<code>p&gt;m</code>	访问 (成员访问): 等价于 <code>(*p).m</code>
<code>p==q</code>	相等判定: 如果 <code>p</code> 和 <code>q</code> 指向相同元素或者都指向最后元素之后位置, 结果为 <code>true</code>
<code>p!=q</code>	不等判定: <code>!(p==q)</code>
<code>p&lt;q</code>	<code>p</code> 指向的元素位于 <code>q</code> 指向的元素之前?
<code>p&lt;=q</code>	<code>p&lt;q    p==q</code>
<code>p&gt;q</code>	<code>p</code> 指向的元素位于 <code>q</code> 指向的元素之后?
<code>p&gt;=q</code>	<code>p&gt;q    p==q</code>
<code>p+=n</code>	前进 <code>n</code> 个元素: 使 <code>p</code> 指向当前元素之后第 <code>n</code> 个元素
<code>p-=n</code>	后退 <code>n</code> 个元素: 使 <code>p</code> 指向当前元素之前第 <code>n</code> 个元素
<code>q=p+n</code>	<code>q</code> 指向 <code>p</code> 所指向的元素之后第 <code>n</code> 个元素
<code>q=p-n</code>	<code>q</code> 指向 <code>p</code> 所指向的元素之前第 <code>n</code> 个元素
<code>advance(p,n)</code>	类似 <code>p+=n</code> , 但 <code>advance()</code> 可用于 <code>p</code> 不是随机访问迭代器的情况, 它花费 <code>n</code> 个步骤 (在列表中移动) 来获得最终位置
<code>x=distance(p,q)</code>	类似 <code>q-p</code> , 但 <code>distance()</code> 可用于 <code>p</code> 不是随机访问迭代器的情况, 它花费 <code>n</code> 个步骤 (在列表中移动) 来获得两个迭代器的距离

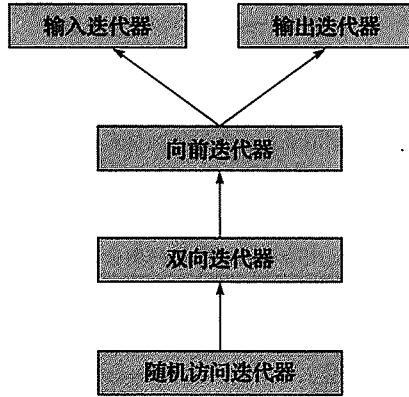
注意, 不是每种迭代器 (见附录 C.3.2) 都支持所有运算。

### C.3.2 迭代器类别

标准库提供五种迭代器 (五个 “迭代器类别”):

迭代器类别	
输入迭代器	可以使用 <code>++</code> 向前移动, 只能读取元素——使用 <code>*</code> 。可以使用 <code>==</code> 和 <code>!=</code> 判断相等性。 <code>istream</code> 提供的就是这种迭代器, 见 16.7.2 节
输出迭代器	可以使用 <code>++</code> 向前移动, 只能写元素——使用 <code>*</code> 。 <code>ostream</code> 提供的就是这种迭代器, 见 16.7.2 节
向前迭代器	可以反复使用 <code>++</code> 向前移动, 可以使用 <code>*</code> 读写元素 (除非元素是 <code>const</code> , <code>const</code> 元素只能读)。如果指向的是类对象, 可以使用 <code>-&gt;</code> 访问成员
双向迭代器	既可向前 (使用 <code>++</code> ) 也可向后 (使用 <code>--</code> ), 可以使用 <code>*</code> 读写元素 ( <code>const</code> 元素只能读)。 <code>list</code> 、 <code>map</code> 和 <code>set</code> 提供的就是这种迭代器
随机访问迭代器	既可向前 (使用 <code>++</code> 或 <code>+=</code> ) 也可向后 (使用 <code>--</code> 或 <code>-=</code> ), 可以使用 <code>*</code> 或 <code>[]</code> 读写元素 ( <code>const</code> 元素只能读)。可以使用下标, 可以使用 <code>+</code> 加上一个整数, 可以使用 <code>-</code> 减去一个整数。可以使用减法获得指向同一序列的两个随机访问迭代器的距离。可以使用 <code>&lt;</code> 、 <code>&lt;=</code> 、 <code>&gt;</code> 和 <code>&gt;=</code> 比较两个随机访问迭代器。 <code>vector</code> 提供的是这种迭代器

逻辑上，这些迭代器构成如下层次（见 15.8 节）：



注意，由于迭代器类别不是类，因此迭代器层次不是利用类派生实现的类层次。如果你需要深入了解迭代器类别，请查阅一些进阶材料，阅读 `iterator_traits` 有关内容。

每个容器都提供特定类别的迭代器：

- `vector`——随机访问迭代器
- `list`——双向迭代器
- `forward_list`——前向迭代器
- `deque`——随机访问迭代器
- `biset`——无
- `set`——双向迭代器
- `multiset`——双向迭代器
- `map`——双向迭代器
- `multimap`——双向迭代器
- `unordered_set`——向前迭代器
- `unordered_multiset`——向前迭代器
- `unordered_map`——向前迭代器
- `unordered_multimap`——向前迭代器

## C.4 容器

容器保存一个对象序列，序列中元素的类型是称为 `value_type` 的成员类型。最常用的容器包括：

### 顺序容器

<code>array&lt;T, N&gt;</code>	固定大小数组，由 $N$ 个类型为 $T$ 的元素构成
<code>deque&lt;T&gt;</code>	双端队列
<code>list&lt;T&gt;</code>	双向链表
<code>forward_list&lt;T&gt;</code>	单向链表
<code>vector&lt;T&gt;</code>	元素类型为 $T$ 的动态数组

关联容器	
<code>map&lt;K, V&gt;</code>	K 到 V 的映射, (K, V) 对的序列
<code>multimap&lt;K, V&gt;</code>	K 到 V 的映射, 允许重复关键字
<code>set&lt;K&gt;</code>	K 的集合
<code>multiset&lt;K&gt;</code>	K 的集合 (允许重复关键字)
<code>unordered_map&lt;K, V&gt;</code>	使用哈希函数从 K 映射到 V
<code>unordered_multimap&lt;K, V&gt;</code>	使用哈希函数从 K 映射到 V, 允许重复关键字
<code>unordered_set&lt;K&gt;</code>	使用哈希函数的 K 的集合
<code>unordered_multiset&lt;K&gt;</code>	使用哈希函数的 K 的集合, 允许重复关键字

有序关联容器 (`map`、`set` 等) 具有一个额外的可选模板参数, 它指出比较器的类型, 例如, `set<K, C>` 使用一个 `C` 来比较 `K` 值。

容器适配器	
<code>priority_queue&lt;T&gt;</code>	优先队列
<code>queue&lt;T&gt;</code>	队列, 支持 <code>push()</code> 和 <code>pop()</code> 操作
<code>stack&lt;T&gt;</code>	栈, 支持 <code>push()</code> 和 <code>pop()</code> 操作

这些容器定义于 `<vector>`、`<list>` 等头文件中 (见附录 C.1.1)。顺序容器的空间是连续分配或是链表, 元素类型为 `value_type` (上表中的符号 `T`)。关联容器是链接结构 (树), 节点类型为 `value_type` (上表中的 (K, V) 对)。`set`、`map` 和 `multimap` 的序列是按关键字值 (`K`) 排序的。`unordered_*` 容器的序列不保证顺序。`multimap` 与 `map` 的不同之处在于, 允许一个关键字值出现多次。容器适配器是在其他容器之上构造而来的, 并定义了专门的操作。

如果对选择哪种容器有疑惑, 请使用 `vector`, 除非你有充分的理由选用其他容器。

容器使用“分配器”分配和释放内存 (见 14.3.6 节)。本附录不讨论分配器, 如果需要, 请查阅专家级文献。默认情况下, 分配器使用 `new` 和 `delete` 为其元素申请和释放内存。

一个访问操作一般有两个版本: 一个用于 `const` 对象, 另一个版本用于非 `const` 对象 (如果都有意义的话, 见 13.5 节)。

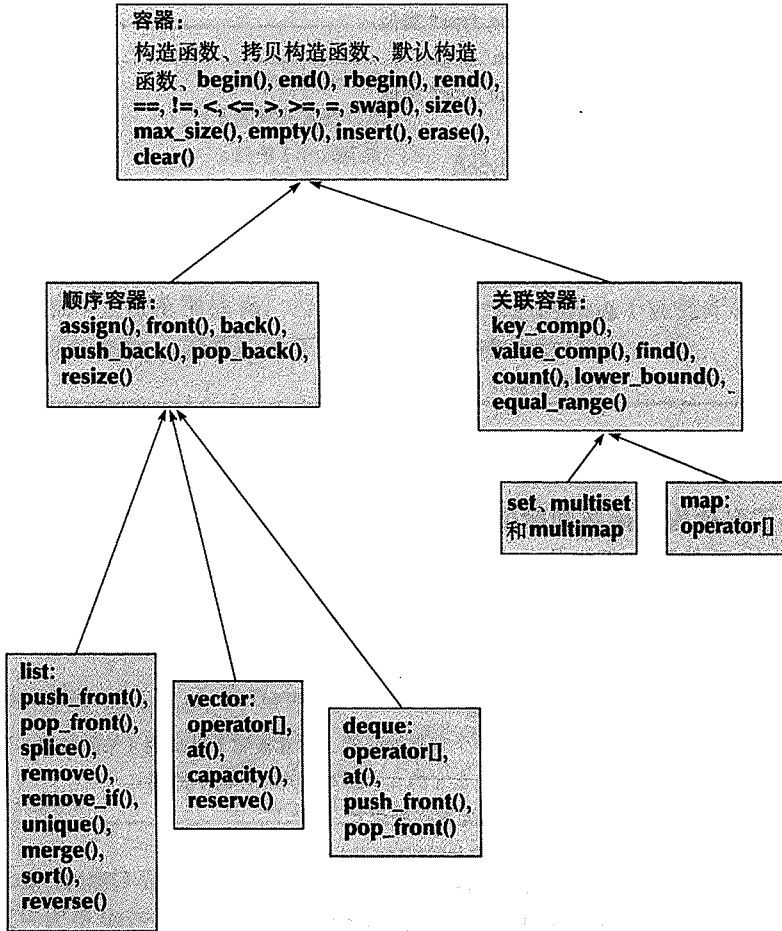
本节列出的成员都是标准容器共有的或者几乎是共有的, 更多细节请参考第 15 章。某个特定容器特有的成员, 如 `list` 的 `splice()`, 并不在本节讨论范围内, 这方面内容请查阅专家级书籍。

某些数据类型提供了标准容器所能提供的大部分功能, 但并不是全部。我们有时称这些数据类型为“拟容器”, 一些最常用的拟容器如下表所示:

“拟容器”	
<code>T[n]</code> (内置数组)	不具备 <code>size()</code> 和其他成员函数, 如果可能的话, 尽量使用 <code>vector</code> 、 <code>string</code> 或 <code>array</code> 等容器, 而不要使用内置数组
<code>string</code>	只包含字符, 但提供了有用的文本处理操作, 如连接 ( <code>+</code> 和 <code>+=</code> ), 应尽量使用标准库 <code>string</code> , 而不要使用其他类型的字符串
<code>valarray</code>	数值向量, 支持向量操作, 但为了提高性能, 有很多限制; 除非你需要做大量向量算术运算, 否则不要使用它

## C.4.1 容器操作概述

标准容器提供的操作可总结如下：



我们忽略了 `array` 和 `forward_list`，因为它们无法完美地适合标准库可互换性方面的理想：

- `array` 不是一个句柄，在初始化后就不能再改变其元素数目了，其初始化必须使用初始化器列表方式，而不能使用构造函数。
- `forward_list` 不支持后退操作。特别是，它不支持 `size()`。我们最好将它看作为空和接近空的序列而特别优化的一种容器。

## C.4.2 成员类型

容器可以定义一组成员类型：

成员类型	
<code>value_type</code>	元素类型
<code>size_type</code>	下标、元素数量等的类型
<code>difference_type</code>	迭代器距离的类型

(续)

成员类型	
iterator	与 value_type* 类似
const_iterator	与 const value_type* 类似
reverse_iterator	与 value_type* 类似
const_reverse_iterator	与 const value_type* 类似
reference	value_type&
const_reference	const value_type&
pointer	与 value_type* 类似
const_pointer	与 const value_type* 类似
key_type	关键字类型 (关联容器才具有)
mapped_type	映射值类型 (关联容器才有)
key_compare	比较标准的类型 (关联容器才有)
allocator_type	内存管理器类型

### C.4.3 构造函数、析构函数和赋值

容器提供了多种构造函数和赋值操作。对于一个称为 **C** 的容器 (如 `vector<double>` 或 `map<string,int>`), 可使用如下操作:

构造函数、析构函数和赋值	
<code>C c;</code>	<code>c</code> 为空容器
<code>C {}</code>	创建一个空容器
<code>C c(n);</code>	<code>c</code> 被初始化为 <code>n</code> 个元素的容器, 元素被赋予默认值 (关联容器不适用)
<code>C c(n,x);</code>	<code>c</code> 被初始化为包含 <code>x</code> 的 <code>n</code> 个副本 (关联容器不适用)
<code>C c(b,e);</code>	用 <code>[b:e)</code> 之间的元素初始化 <code>c</code>
<code>C c(elems);</code>	用包含 <code>elems</code> 的 <code>initializer_list</code> 初始化 <code>c</code>
<code>C c(c2);</code>	<code>c</code> 被初始化为容器 <code>c2</code> 的副本
<code>~C()</code>	销毁容器及其所有元素 (通常被隐式调用)
<code>c1=c2</code>	拷贝赋值, 将 <code>c2</code> 的所有元素复制到 <code>c1</code> 中, 因此, 赋值完成后 <code>c1==c2</code>
<code>c.assign(n,x)</code>	将 <code>x</code> 的 <code>n</code> 个副本赋予 <code>c</code>
<code>c.assign(b,e)</code>	将 <code>[b:e)</code> 之间的元素赋予 <code>c</code>

注意, 对一些容器和一些元素类型, 构造函数或元素复制可能会抛出异常。

### C.4.4 迭代器

一个容器可以看作按其迭代器定义的顺序或相反顺序排列的元素序列。关联容器的顺序则是由比较操作 (默认比较操作是运算符 `<`) 决定的。

迭代器	
<code>p=c.begin()</code>	<code>p</code> 指向 <code>c</code> 的首元素
<code>p=c.end()</code>	<code>p</code> 指向 <code>c</code> 的尾元素之后的位置
<code>p=c.rbegin()</code>	<code>p</code> 指向 <code>c</code> 的逆序的首元素
<code>p=c.rend()</code>	<code>p</code> 指向 <code>c</code> 的逆序的尾元素之后的位置

## C.4.5 元素访问

一些元素可以直接访问：

元素访问	
<code>c.front()</code>	<code>c</code> 的首元素的引用
<code>c.back()</code>	<code>c</code> 的尾元素的引用
<code>c[i]</code>	<code>c</code> 的第 <code>i</code> 个元素的引用，不做范围检查（ <code>list</code> 不适用）
<code>c.at(i)</code>	<code>c</code> 的第 <code>i</code> 个元素的引用，有范围检查（只适用于 <code>vector</code> 和 <code>deque</code> ）

一些标准库实现总是进行范围检查，特别是在调试状态下。但你如果想编写可移植的代码，不能假定标准库会做范围检查，以此来保证正确性，同样也不能假定标准库不会做类型检查，以此获得好的性能。如果这很重要，请认真检查你的代码。

## C.4.6 栈和队列操作

标准库 `vector` 和 `deque` 提供了在序列末端的高效操作。另外，`list` 和 `deque` 提供了在序列首的高效操作。

栈和队列操作	
<code>c.push_back(x)</code>	将 <code>x</code> 添加到 <code>c</code> 的末尾
<code>c.pop_back()</code>	删除 <code>c</code> 的尾元素
<code>c.emplace_back(args)</code>	将 <code>T(args)</code> 添加到 <code>c</code> 的末尾； <code>T</code> 为 <code>c</code> 的值类型
<code>c.push_front(x)</code>	将 <code>x</code> 添加到 <code>c</code> 的首元素之前（只适用于 <code>list</code> 和 <code>deque</code> ）
<code>c.pop_front()</code>	删除 <code>c</code> 的首元素（只适用于 <code>list</code> 和 <code>deque</code> ）
<code>c.emplace_front(args)</code>	将 <code>T(args)</code> 添加到 <code>c</code> 的首元素之前； <code>T</code> 为 <code>c</code> 的值类型

注意，`push_front()` 和 `push_back()` 向容器中拷贝一个元素。这意味着容器规模增加 1。如果元素类型的拷贝构造函数能抛出异常，压栈操作可能会失败。

`push_front()` 和 `push_back()` 操作将实参对象拷贝到容器中。例如：

```
vector<pair<string,int>> v;
v.push_back(make_pair("Cambridge",1209));
```

如果首先创建一个对象然后再拷贝它显得有些笨拙或低效，我们可以在序列中一个新分配的元素位置上直接构造对象：

```
v.emplace_back("Cambridge",1209);
```

`Emplace`（安放）的含义是“放在此空间”或“放在此位置”。

注意，弹出栈操作不返回值。如果这些操作返回值，而且元素类型的拷贝构造函数能抛出异常的话，可能会使实现变得非常复杂。访问栈和队列元素请使用 `front()` 和 `back()`（见附录 C.4.5）。本小节中没有给出每个操作的完整要求，你可以猜测（如果猜错，编译器会告知你）或者查阅更详细的文档。

## C.4.7 链表操作

容器还提供了链表操作：



**链表操作**

<code>q=c.insert(p,x)</code>	将 <code>x</code> 添加到 <code>p</code> 之前
<code>q=c.insert(p,n,x)</code>	将 <code>x</code> 的 <code>n</code> 份副本添加到 <code>p</code> 之前
<code>q=c.insert(p,first,last)</code>	将 <code>[first:last)</code> 之间的元素添加到 <code>p</code> 之前
<code>q=c.emplace(p,args)</code>	将 <code>T(args)</code> 添加到 <code>p</code> 之前; <code>T</code> 为 <code>c</code> 的值类型
<code>q=c.erase(p)</code>	将 <code>c</code> 中位置 <code>p</code> 处的元素删除
<code>q=c.erase(first,last)</code>	删除 <code>c</code> 中 <code>[first:last)</code> 之间的元素
<code>c.clear()</code>	删除 <code>c</code> 中所有元素

对于 `insert()` 函数, 结果 `q` 指向插入的最后一个元素。对于 `erase()` 函数, `q` 指向最后一个被删元素之后的元素。

**C.4.8 大小和容量**

大小是指容器中的元素个数, 容量是指容器在不扩充内存的情况下所能容纳的最大元素数:

**大小和容量**

<code>x=c.size()</code>	<code>x</code> 为 <code>c</code> 的元素数目
<code>c.empty()</code>	<code>c</code> 空?
<code>x=c.max_size()</code>	<code>x</code> 为 <code>c</code> 能容纳的最大元素数
<code>x=c.capacity()</code>	<code>x</code> 是为 <code>c</code> 分配的内存空间大小 (仅适用于 <code>vector</code> 和 <code>string</code> )
<code>c.reserve(n)</code>	为 <code>c</code> 留出 <code>n</code> 个元素的空间 (仅适用于 <code>vector</code> 和 <code>string</code> )
<code>c.resize(n)</code>	将 <code>c</code> 的大小改变为 <code>n</code> (仅适用于 <code>vector</code> 、 <code>string</code> 、 <code>list</code> 和 <code>deque</code> )

当改变大小或容量时, 元素可能被移动到新的内存位置。这意味着指向元素的迭代器 (以及指针和引用) 将变为无效 (它们仍指向旧的元素存储位置)。

**C.4.9 其他操作**

容器可以拷贝 (见附录 C.4.3)、比较以及交换:

**比较和交换**

<code>c1==c2</code>	<code>c1</code> 和 <code>c2</code> 的所有对应元素都相等?
<code>c1!=c2</code>	<code>c1</code> 和 <code>c2</code> 的某个对应元素不等?
<code>c1&lt;c2</code>	<code>c1</code> 字典序在 <code>c2</code> 前?
<code>c1&lt;=c2</code>	<code>c1</code> 字典序在 <code>c2</code> 之前, 或与 <code>c2</code> 相同?
<code>c1&gt;c2</code>	<code>c1</code> 字典序在 <code>c2</code> 后?
<code>c1&gt;=c2</code>	<code>c1</code> 字典序在 <code>c2</code> 之后, 或与 <code>c2</code> 相同?
<code>swap(c1,c2)</code>	交换 <code>c1</code> 和 <code>c2</code> 的元素
<code>c1.swap(c2)</code>	交换 <code>c1</code> 和 <code>c2</code> 的元素

当用一个运算符 (如 `<`) 比较两个容器时, 实际是用等价的元素运算符 (即 `<`) 来比较对应元素。

## C.4.10 关联容器操作

关联容器提供了基于关键字的查找：

### 关联容器操作

<code>c[k]</code>	引用关键字为 <code>k</code> 的元素（适用于关键字唯一的容器）
<code>p=c.find(k)</code>	<code>p</code> 指向第一个关键字为 <code>k</code> 的元素
<code>p=c.lower_bound(k)</code>	<code>p</code> 指向第一个关键字为 <code>k</code> 的元素
<code>p=c.upper_bound(k)</code>	<code>p</code> 指向第一个关键字大于 <code>k</code> 的元素
<code>pair(p1,p2)=c.equal_range(k)</code>	<code>[p1, p2)</code> 为关键字为 <code>k</code> 的所有元素
<code>r=c.key_comp()</code>	<code>r</code> 为关键字比较对象（比较函数）的副本
<code>r=c.value_comp()</code>	<code>r</code> 为映射值比较对象（比较函数）的副本。如果关键字未找到，返回 <code>c.end()</code>

`equal_range` 返回的 `pair` 中的第一个迭代器为 `lower_bound`，第二个迭代器为 `upper_bound`。如果你希望打印 `multimap<string,int>` 中所有关键字值为 "Marian" 的元素，可以这样做：

```
string k = "Marian";
auto pp = m.equal_range(k);
if (pp.first!=pp.second)
    cout << "elements with value " << k << ":\n";
else
    cout << "no element with value " << k << "\n";
for (auto p = pp.first; p!=pp.second; ++p)
    cout << p->second << '\n';
```

其中对 `equal_range` 的调用等价于：

```
auto pp = make_pair(m.lower_bound(k),m.upper_bound(k));
```

但是，第二种方式的执行时间可能是 `equal_range` 的两倍。有序序列也提供了 `equal_range`、`lower_bound` 和 `upper_bound` 算法（见附录 C.5.4）。`pair` 的定义见附录 C.6.3。

## C.5 算法

`<algorithm>` 定义了大约 60 个标准算法。所有这些算法都是对一对迭代器所标定的序列（输入）或者一个单一迭代器（输出）进行操作。

当复制、比较、…两个序列时，第一个序列用一对迭代器 `[b:e)` 表示，但第二个序列只用单一迭代器 `b2` 表示，它应该包含足够的元素以便算法能正确运行，比如说包含与第一个序列一样多的元素——`[b2:b2+(e-b))`。

某些算法（例如 `sort`）要求随机访问迭代器，而其他很多算法（如 `find`）只顺序读取元素，因此只需一个向前迭代器就能工作。

很多算法遵循这样一个常用约定：通过返回序列尾来表示“未找到”。我们不再对每个算法重复这一点。

### C.5.1 非变动性序列算法

非变动性算法只读取序列元素，但不重排元素顺序，也不修改元素的值。

## 非变动性序列算法

<code>f=for_each(b,e,f)</code>	对 [b:e) 中的每个元素执行 f, 返回 f
<code>p=find(b,e,v)</code>	p 指向 [b:e) 中第一次出现 v 的位置
<code>p=find_if(b,e,f)</code>	p 指向 [b:e) 中第一个使 f(*p) 为真的元素
<code>p=find_first_of(b,e,b2,e2)</code>	p 指向 [b:e) 中第一个满足 *p==*q 的元素, q 为 [b2:e2) 中位置
<code>p=find_first_of(b,e,b2,e2,f)</code>	p 指向 [b:e) 中第一个使 f(*p,*q) 为真的元素, q 为 [b2:e2) 中对应位置
<code>p=adjacent_find(b,e)</code>	p 指向 [b:e) 中第一个满足 *p==*(p+1) 的元素
<code>p=adjacent_find(b,e,f)</code>	p 指向 [b:e) 中第一个使 f(*p,*p+1) 为真的元素
<code>equal(b,e,b2)</code>	[b:e) 中所有元素都和 [b2:b2+(e-b)) 中对位元素相等吗?
<code>equal(b,e,b2,f)</code>	对 [b:e) 和 [b2:b2+(e-b)) 中所有对位元素 *p, *q, f(*p,*q) 都为真吗?
<code>pair(p1,p2)=mismatch(b,e,b2)</code>	(p1,p2) 指向 [b:e) 和 [b2:b2+(e-b)) 中第一对不等的对位元素, 即 !(*p1==*p2)
<code>pair(p1,p2)=mismatch(b,e,b2,f)</code>	(p1,p2) 指向 [b:e) 和 [b2:b2+(e-b)) 中第一对满足 !f(*p1,*p2) 的对位元素
<code>p=search(b,e,b2,e2)</code>	p 指向 [b:e) 中第一个在 [b2:e2) 中有相等元素的元素
<code>p=search(b,e,b2,e2,f)</code>	p 指向 [b:e) 中第一个这样的元素: 在 [b2:e2) 中存在一个元素 *q, 满足 f(*p,*q)
<code>p=find_end(b,e,b2,e2)</code>	p 指向 [b:e) 中最后一个在 [b2:e2) 中有相等元素的元素
<code>p=find_end(b,e,b2,e2,f)</code>	p 指向 [b:e) 中最后一个这样的元素: 在 [b2:e2) 中存在一个元素 *q, 满足 f(*p,*q)
<code>p=search_n(b,e,n,v)</code>	p 指向 [b:e) 中第一个满足下面条件的区间 [p:p+n) 的第一个元素: 其全部元素都等于 v
<code>p=search_n(b,e,n,v,f)</code>	p 指向 [b:e) 中第一个全部元素都满足 f(*p,v) 的区间 [p:p+n) 的第一个元素
<code>x=count(b,e,v)</code>	x 为 [b:e) 中 v 出现的次数
<code>x=count_if(b,e,v,f)</code>	x 为 [b:e) 中满足 f(*p,v) 的元素的数目

注意, 标准库并不会阻止将一个修改元素的操作传递给 `for_each`, 这是可接受的。但对于其他一些算法 (如 `count` 或 `==`), 则不接受修改元素的操作。

下面是一个 (正确使用的) 例子:

```
bool odd(int x) { return x&1; }
```

```
int n_even(const vector<int>& v) // 统计 v 中偶数值的个数
{
    return v.size()-count_if(v.begin(),v.end(),odd);
}
```

## C.5.2 变动性序列算法

变动性算法 (也称为改变型序列算法 (mutating sequence algorithm)) 可以 (通常也确实) 修改参数序列中元素。

## 变动性序列算法

<code>p=transform(b,e,out,f)</code>	对 [b:e) 中每个 *p1 执行 *p2=f(*p1), 将结果 *p2 写到 [out:out+(e-b)) 中对应位置; p=out+(e-b)
<code>p=transform(b,e,b2,out,f)</code>	对 [b:e) 和 [b2:b2+(e-b)) 中每对对应元素 *p1 和 *p2 执行 *p3=f(*p1,*p2), 将结果 *p3 写到 [out:out+(e-b)) 中对应位置; p=out+(e-b)
<code>p=copy(b,e,out)</code>	将 [b:e) 拷贝至 [out:p)

(续)

## 变动性序列算法

<code>p=copy_backward(b,e,out)</code>	从最后一个元素开始将 [b:e] 拷贝至 [out:p]
<code>p=unique(b,e)</code>	移动 [b:e] 中元素, 使得 [b:p] 中相邻元素不重复 (“重复”由 == 判定)
<code>p=unique(b,e,f)</code>	移动 [b:e] 中元素, 使得 [b:p] 中相邻元素不重复 (“重复”由 f 判定)
<code>p=unique_copy(b,e,out)</code>	将 [b:e] 拷贝至 [out:p], 不拷贝相邻的重复元素
<code>p=unique_copy(b,e,out,f)</code>	将 [b:e] 拷贝至 [out:p], 不拷贝相邻的重复元素 (“重复”由 f 判定)
<code>replace(b,e,v,v2)</code>	将 [b:e] 中所有值为 v 的元素用 v2 替换
<code>replace(b,e,f,v2)</code>	将 [b:e] 中所有满足 f(*q) 的元素用 v2 替换
<code>p=replace_copy(b,e,out,v,v2)</code>	将 [b:e] 拷贝至 [out:p], 其中所有值为 v 的元素用 v2 替换
<code>p=replace_copy(b,e,out,f,v2)</code>	将 [b:e] 拷贝至 [out:p], 其中所有满足 f(*q) 的元素用 v2 替换
<code>p=remove(b,e,v)</code>	移动 [b:e] 中元素, 使得 [b:p] 中不出现 v
<code>p=remove(b,e,v,f)</code>	移动 [b:e] 中元素, 使得 [b:p] 中无满足 f(*q) 的元素
<code>p=remove_copy(b,e,out,v)</code>	将 [b:e] 中值不等于 v 的元素拷贝至 [out:p]
<code>p=remove_copy_if(b,e,out,f)</code>	将 [b:e] 中不满足 f(*q) 的元素拷贝至 [out:p]
<code>reverse(b,e)</code>	倒转 [b:e] 中元素的顺序
<code>p=reverse_copy(b,e,out)</code>	将 [b:e] 中元素按逆序拷贝至 [out:p]
<code>rotate(b,m,e)</code>	元素循环移位: 将 [b:e] 看作一个圈——首元素在尾元素之后。将 *b 移动到 *m, 其他元素依此类推——将 *(b+i) 移动到 *((b+i+(e-m))%(e-b))
<code>p=rotate_copy(b,m,e,out)</code>	将 [b:e] 拷贝至 [out:p], 元素循环右移 m 位
<code>random_shuffle(b,e)</code>	随机混洗: 利用默认均匀分布随机数发生器将 [b:e] 中元素重新排列
<code>random_shuffle(b,e,f)</code>	随机混洗: 用 f 作为随机数发生器

shuffle 算法充分混洗序列, 就像我们洗牌一样。也就是说, 混洗之后, 元素是随机排列的, “随机”是由随机数发生器所产生的分布决定的。

请注意, 这些算法并不知道其参数是否是一个容器, 因而它们并没有添加或删除元素的能力。因此, 像 remove 这样的算法不会删除元素、缩短序列, 而是将保留下来的元素移动到序列的前端:

```
template<typename Iter>
void print_digits(const string& s, Iter b, Iter e)
{
    cout << s;
    while (b!=e) { cout << *b; ++b; }
    cout << '\n';
}

void ff()
{
    vector<int> v {1,1,1, 2,2, 3, 4,4,4, 3,3,3, 5,5,5,5, 1,1,1};
    print_digits("all: ", v.begin(), v.end());

    auto pp = unique(v.begin(), v.end());
    print_digits("head: ", v.begin(), pp);
    print_digits("tail: ", pp, v.end());

    pp=remove(v.begin(), pp, 4);
    print_digits("head: ", v.begin(), pp);
    print_digits("tail: ", pp, v.end());
}
```

输出结果如下：

```
all: 111223444333555111
head: 1234351
tail: 44333555111
head: 123351
tail: 144333555111
```

### C.5.3 工具算法

从技术角度，这些工具算法也都是变动性序列算法，但我们觉得将它们单独列出更好，以免被忽略。

工具算法	
swap(x,y)	交换 x 和 y
iter_swap(p,q)	交换 *p 和 *q
swap_ranges(b,e,b2)	交换 [b:e) 和 [b2:b2+(e-b)) 的元素
fill(b,e,v)	将 v 赋予 [b:e) 中每个元素
fill_n(b,n,v)	将 v 赋予 [b:b+n) 中每个元素
generate(b,e,f)	将 f() 赋予 [b:e) 中每个元素
generate_n(b,n,f)	将 f() 赋予 [b:b+n) 中每个元素
uninitialized_fill(b,e,v)	将 [b:e) 中所有元素初始化为 v
uninitialized_copy(b,e,out)	用 [b:e) 中元素初始化 [out:out+(e-b)) 中对应元素

注意，未初始化的序列只能出现在最底层的程序中，通常是在容器的实现代码中。`uninitialized_fill` 和 `uninitialized_copy` 处理的元素必须是内置类型或者是未初始化的。

### C.5.4 排序和搜索

排序和搜索是非常基础的操作，而程序员对其的要求可能差别很大。默认情况下，比较操作通过 < 运算符来完成，而元素 a 和 b 的值相等通过 `!(a<b)&&!(b<a)` 来判定，而不是使用 `==` 运算符。

排序和搜索	
sort(b,e)	排序 [b:e)
sort(b,e,f)	排序 [b:e)，使用 f(*p,*q) 进行比较操作
stable_sort(b,e)	排序 [b:e)，保持相等元素的原有顺序
stable_sort(b,e,f)	排序 [b:e)，使用 f(*p,*q) 进行比较操作，保持相等元素的原有顺序
partial_sort(b,m,e)	排序 [b:e)，保证 [b:m) 有序，[m:e) 可以不必有序
partial_sort(b,m,e,f)	排序 [b:e)，使用 f(*p,*q) 进行比较操作，保证 [b:m) 有序，[m:e) 可以不必有序
partial_sort_copy(b,e,b2,e2)	排序 [b:e)，保证有足够多 (e2-b2 个) 的有序元素复制到 [b2:e2) 即可
partial_sort_copy(b,e,b2,e2,f)	排序 [b:e)，使用 f 作为比较操作，保证有足够多的有序元素复制到 [b2:e2) 即可
nth_element(b,e)	将 [b:e) 中排在第 n 位的元素移动到正确位置
nth_element(b,e,f)	将 [b:e) 中排在第 n 位的元素移动到正确位置，使用 f 作为比较操作
p=lower_bound(b,e,v)	p 指向 [b:e) 中 v 第一次出现的位置

(续)

## 排序和搜索

<code>p=lower_bound(b,e,v,f)</code>	<code>p</code> 指向 <code>[b:e)</code> 中 <code>v</code> 第一次出现的位置, 使用 <code>f</code> 作为比较操作
<code>p=upper_bound(b,e,v)</code>	<code>p</code> 指向 <code>[b:e)</code> 中第一个大于 <code>v</code> 的元素
<code>p=upper_bound(b,e,v,f)</code>	<code>p</code> 指向 <code>[b:e)</code> 中第一个大于 <code>v</code> 的元素, 使用 <code>f</code> 作为比较操作
<code>binary_search(b,e,v)</code>	有序序列 <code>[b:e)</code> 中包含 <code>v</code> 吗?
<code>binary_search(b,e,v,f)</code>	有序序列 <code>[b:e)</code> 中包含 <code>v</code> 吗? <code>f</code> 作为比较操作
<code>pair(p1,p2)=equal_range(b,e,v)</code>	<code>[p1:p2)</code> 为 <code>[b:e)</code> 的子序列, 其中元素均为 <code>v</code> , 大致是在序列中二分搜索 <code>v</code>
<code>pair(p1,p2)=equal_range(b,e,v,f)</code>	<code>[p1:p2)</code> 为 <code>[b:e)</code> 的子序列, 其中元素均为 <code>v</code> , <code>f</code> 作为比较操作, 大致是在序列中二分搜索 <code>v</code>
<code>p=merge(b,e,b2,e2,out)</code>	将有序序列 <code>[b2:e2)</code> 和 <code>[b:e)</code> 合并为一个有序序列, 结果存入 <code>[out:p)</code>
<code>p=merge(b,e,b2,e2,out,f)</code>	将有序序列 <code>[b2:e2)</code> 和 <code>[b:e)</code> 合并为一个有序序列, 结果存入 <code>[out:p)</code> , <code>f</code> 作为比较操作
<code>inplace_merge(b,m,e)</code>	原址合并, 将有序序列 <code>[b:m)</code> 和 <code>[m:e)</code> 合并为一个有序序列 <code>[b:e)</code>
<code>inplace_merge(b,m,e,f)</code>	原址合并, <code>f</code> 作为比较操作
<code>p=partition(b,e,f)</code>	将满足 <code>f(*p1)</code> 的元素置于 <code>[b:p)</code> , 其他元素置于 <code>[p:e)</code>
<code>p=stable_partition(b,e,f)</code>	将满足 <code>f(*p1)</code> 的元素置于 <code>[b:p)</code> , 其他元素置于 <code>[p:e)</code> , 保持相对顺序

例如:

```
vector<int> v {3,1,4,2};
list<double> lst {0.5,1.5,3,2.5}; // lst 已有序
sort(v.begin(),v.end()); // 排序 v
vector<double> v2;
merge(v.begin(),v.end(),lst.begin(),lst.end(),back_inserter(v2));
for (auto x : v2) cout << x << ", ";
```

关于 `inserter`, 参见附录 C.6.1。输出结果为:

0.5, 1, 1.5, 2, 2, 2.5, 3, 4,

`equal_range`、`lower_bound` 和 `upper_bound` 的使用方法与关联容器的对应版本相同, 参见附录 C.4.10。

## C.5.5 集合算法

这些算法将序列视为元素集合, 提供了基本的集合操作。这些算法假定输入序列是有序的, 而它们生成的输出也是有序的:

## 集合算法

<code>includes(b,e,b2,e2)</code>	<code>[b2:e2)</code> 中所有元素都在 <code>[b:e)</code> 中?
<code>includes(b,e,b2,e2,f)</code>	<code>[b2:e2)</code> 中所有元素都在 <code>[b:e)</code> 中? <code>f</code> 作为比较操作
<code>p=set_union(b,e,b2,e2,out)</code>	构造有序序列 <code>[out:p)</code> , 包含 <code>[b:e)</code> 和 <code>[b2:e2)</code> 中所有元素
<code>p=set_union(b,e,b2,e2,out,f)</code>	构造有序序列 <code>[out:p)</code> , 包含 <code>[b:e)</code> 和 <code>[b2:e2)</code> 中所有元素, <code>f</code> 作为比较操作
<code>p=set_intersection(b,e,b2,e2,out)</code>	构造有序序列 <code>[out:p)</code> , 包含 <code>[b:e)</code> 和 <code>[b2:e2)</code> 中同时出现的元素
<code>p=set_intersection(b,e,b2,e2,out,f)</code>	构造有序序列 <code>[out:p)</code> , 包含 <code>[b:e)</code> 和 <code>[b2:e2)</code> 中同时出现的元素, <code>f</code> 作为比较操作

(续)

集合算法	
<code>p=set_difference(b,e,b2,e2,out)</code>	构造有序序列 [out:p]，包含在 [b:e] 中出现但在 [b2:e2] 没有出现的元素
<code>p=set_difference(b,e,b2,e2,out,f)</code>	构造有序序列 [out:p]，包含在 [b:e] 中出现但在 [b2:e2] 没有出现的元素，f 作为比较操作
<code>p=set_symmetric_difference(b,e,b2,e2,out)</code>	构造有序序列 [out:p]，包含 [b:e] 中出现或 [b2:e2] 中出现但不同时在两者中出现的元素
<code>p=set_symmetric_difference(b,e,b2,e2,out,f)</code>	构造有序序列 [out:p]，包含在 [b:e] 中出现或 [b2:e2] 中出现但不同时在两者中出现的元素，f 作为比较操作

## C.5.6 堆

堆是这样一种数据结构：它将值最大的元素保存在最前面。堆算法允许程序员将一个随机访问序列作为一个堆来处理：

堆操作	
<code>make_heap(b,e)</code>	将序列 [b:e] 整理为一个堆
<code>make_heap(b,e,f)</code>	将序列 [b:e] 整理为一个堆，f 作为比较操作
<code>push_heap(b,e)</code>	将元素加入堆（中正确位置）
<code>push_heap(b,e,f)</code>	将元素加入堆，f 作为比较操作
<code>pop_heap(b,e)</code>	将最大元素从堆中删除
<code>pop_heap(b,e,f)</code>	将最大元素从堆中删除，f 作为比较操作
<code>sort_heap(b,e)</code>	堆排序
<code>sort_heap(b,e,f)</code>	堆排序，f 作为比较操作

堆的目标是提供快速的元素添加和最大元素访问操作，其主要用途是实现优先队列。

## C.5.7 排列

排列用于生成序列元素的组合。例如，abc 的排列有 abc、acb、bac、bca、cab 和 cba。

排列	
<code>x=next_permutation(b,e)</code>	生成字典序中 [b:e] 的下一个排列
<code>x=next_permutation(b,e,f)</code>	生成字典序中 [b:e] 的下一个排列，f 作为比较操作
<code>x=prev_permutation(b,e)</code>	生成字典序中 [b:e] 的前一个排列
<code>x=prev_permutation(b,e,f)</code>	生成字典序中 [b:e] 的前一个排列，f 作为比较操作

如果 [b:e] 当前的排列已经是最后一个排列（上例中的 cba），`next_permutation` 的返回值 (x) 为 false，在这种情况下，返回第一个排列（上例中的 abc）。如果 [b:e] 当前的排列已经是第一个排列（上例中的 abc），`prev_permutation` 的返回值 (x) 为 false，在这种情况下，返回最后一个排列（上例中的 cba）。

## C.5.8 min 和 max

比较操作在很多场合下都是很有用的：

**min 和 max**

<code>x=max(a,b)</code>	x 为 a 和 b 中较大者
<code>x=max(a,b,f)</code>	x 为 a 和 b 中较大者, f 作为比较操作
<code>x=max({elems})</code>	x 为 {elems} 中最大元素
<code>x=max({elems},f)</code>	x 为 {elems} 中最大元素, f 作为比较操作
<code>x=min(a,b)</code>	x 为 a 和 b 中较小者
<code>x=min(a,b,f)</code>	x 为 a 和 b 中较小者, f 作为比较操作
<code>x=min({elems})</code>	x 为 {elems} 中最小元素
<code>x=min({elems},f)</code>	x 为 {elems} 中最小元素, f 作为比较操作
<code>pair(x,y)=minmax(a,b)</code>	x 为 max(a,b), y 为 min(a,b)
<code>pair(x,y)=minmax(a,b,f)</code>	x 为 max(a,b,f), y 为 min(a,b,f)
<code>pair(x,y)=minmax({elems})</code>	x 为 max({elems}), y 为 min({elems})
<code>pair(x,y)=minmax({elems},f)</code>	x 为 max({elems},f), y 为 min({elems},f)
<code>p=max_element(b,e)</code>	p 指向 [b:e) 中最大元素
<code>p=max_element(b,e,f)</code>	p 指向 [b:e) 中最大元素, f 作为元素比较操作
<code>p=min_element(b,e)</code>	p 指向 [b:e) 中最小元素
<code>p=min_element(b,e,f)</code>	p 指向 [b:e) 中最小元素, f 作为元素比较操作
<code>lexicographical_compare(b,e,b2,e2)</code>	[b:e)<[b2:e2) ?
<code>lexicographical_compare(b,e,b2,e2,f)</code>	[b:e)<[b2:e2) ? f 作为元素比较操作

## C.6 STL 工具

STL 提供了一些工具, 能方便使用 STL 算法。

### C.6.1 插入器

一些算法通过迭代器将结果输出到容器中, 这意味着迭代器指向的元素和之后一个元素可以被改写。这也意味着可能出现溢出, 从而导致内存错误。例如:

```
void f(vector<int>& vi)
{
    fill_n(vi.begin(), 200,7);    // 将 7 赋予 vi[0]..vi[199]
}
```

如果 vi 中元素数少于 200 的话, 我们就有麻烦了。

在 <iterator> 中, 标准库提供了三种迭代器来解决问题, 它们并不改写旧元素, 而是向容器中加入 (插入) 新元素。标准库提供了三个函数来生成这些插入迭代器:

**插入器**

<code>r=back_inserter(c)</code>	*r=x, 引起一次 c.push_back(x) 操作
<code>r=front_inserter(c)</code>	*r=x, 引起一次 c.push_front(x) 操作
<code>r=inserter(c,p)</code>	*r=x, 引起一次 c.insert(p,x) 操作

对于 `inserter(c,p)`, p 必须是容器 c 的一个合法迭代器。当用插入迭代器将一个值写入容器时, 容器的规模增长一个元素。插入器通过 `push_back()`、`push_front()` 或 `insert()` 向容器插入一个新元素, 而不是改写已有元素, 例如:



```
void g(vector<int>& vi)
{
    fill_n(back_inserter(vi), 200, 7);    // 在 vi 的末尾添加 200 个 7
}
```

## C.6.2 函数对象

很多标准库算法都接受函数对象（或函数）参数，这允许程序员控制其工作方式。函数对象的常见用途是比较操作、断言（返回 bool 值的函数）和算术运算。在 `<functional>` 中，标准库提供了一些常用的函数对象。

---

### 断言

<code>p=equal_to&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $p(x,y)$ 表示 $x==y$
<code>p=not_equal_to&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $p(x,y)$ 表示 $x!=y$
<code>p=greater&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $p(x,y)$ 表示 $x>y$
<code>p=less&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $p(x,y)$ 表示 $x<y$
<code>p=greater_equal&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $p(x,y)$ 表示 $x>=y$
<code>p=less_equal&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $p(x,y)$ 表示 $x<=y$
<code>p=logical_and&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $p(x,y)$ 表示 $x\&\&y$
<code>p=logical_or&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $p(x,y)$ 表示 $x\ \ y$
<code>p=logical_not&lt;T&gt;()</code>	当 $x$ 是类型 $T$ 时， $p(x)$ 表示 $!x$

---

例如：

```
vector<int> v;
// ...
sort(v.begin(), v.end(), greater<int>{});    // 将 v 排序为递减序
```

注意，`logical_and` 和 `logical_or` 总是对两个参数都进行求值（`&&` 和 `\|\|` 则不是这样）。此外，`lambda` 被动式（见 20.3.3 节）通常可以用来代替简单的函数对象：

```
sort(v.begin(), v.end(), [](int x, int y) { return x>y; });    // 将 v 排序为递减序
```

---

### 算术运算

<code>f=plus&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $f(x,y)$ 表示 $x+y$
<code>f=minus&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $f(x,y)$ 表示 $x-y$
<code>f=multiplies&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $f(x,y)$ 表示 $x*y$
<code>f=divides&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $f(x,y)$ 表示 $x/y$
<code>f=modulus&lt;T&gt;()</code>	当 $x$ 和 $y$ 是类型 $T$ 时， $f(x,y)$ 表示 $x\%y$
<code>f=negate&lt;T&gt;()</code>	当 $x$ 是类型 $T$ 时， $f(x,y)$ 表示 $-x$

---

### 适配器

<code>f=bind(g,args)</code>	$f(x)$ 表示 $g(x,args)$ ，其中 $args$ 可以是一个或多个实参
<code>f=mem_fn(mf)</code>	$f(p,args)$ 表示 $p->mf(args)$ ，其中 $args$ 可以是一个或多个实参
<code>Function&lt;f&gt; f(g)</code>	$f(args)$ 表示 $g(args)$ ，其中 $args$ 可以是一个或多个实参， $F$ 为 $g$ 的类型
<code>f=not1(g)</code>	$f(x)$ 表示 $!g(x)$
<code>f=not2(g)</code>	$f(x,y)$ 表示 $!g(x,y)$

---

注意，`function` 是一个模板，从而你可以定义类型为 `function<T>` 的变量，并将可调用对象赋值给这种变量。例如：

```
int f1(double);
function<int(double)> fct {f1};    // 初始化为 f1
int x = fct(2.3);                // 调用 f1(2,3)
function<int(double)> fun;       // fun 可以保存任意的 int(double)
fun = f1;
```

### C.6.3 pair 和 tuple

在 `<utility>` 中，标准库提供了一些“工具组件”，`pair` 就是其中之一：

```
template <class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;

    // ...拷贝和移动操作...
};
```

```
template <class T1, class T2>
constexpr pair<T1,T2> make_pair(T1 x, T2 y) { return pair<T1,T2>(x,y); }
```

函数 `make_pair` 使 `pair` 的使用变得简单。例如，下面函数返回一个值和一个错误指示器：

```
pair<double,error_indicator> my_fct(double d)
{
    errno = 0;    // 清除 C 风格的全局错误指示器
    // ... 进行大量涉及 d 的计算来算出 x ...
    error_indicator ee = errno;
    errno = 0;    // 清除 C 风格的全局错误指示器
    return make_pair(x,ee);
}
```

下面是一种常见的用法：

```
pair<int,error_indicator> res = my_fct(123.456);
if (res.second==0) {
    // ... 使用 res.first ...
}
else {
    // 糟糕：错误
}
```

当我们就是需要两个元素，而不在意是否为它们定义一个类型时，就应该使用 `pair`。如果我们需要零个或多个元素，我们可以使用 `<tuple>` 中定义的 `tuple`：

```
template <typename... Types>
struct tuple {
    explicit constexpr tuple(const Types& ...);    // 用 N 个值构造
    template<typename... Atypes>
    explicit constexpr tuple(const Atypes&& ...); // 用 N 个值构造

    // ...复制和移动操作
};
```

```
template <class... Types>
constexpr tuple<Types...> make_tuple(Types&&...); // 用 N 个值构造 tuple
```

tuple 的实现采用了一种称为可变参数模板的特性，它已经超出了本书的范围。上面代码中的省略号 (...) 就是这种特性。但这并没有关系，我们可以像使用 pair 那样使用 tuple。例如：

```
auto t0 = make_tuple();           // 无元素
auto t1 = make_tuple(123.456);   // 一个 double 类型元素
auto t2 = make_tuple(123.456, 'a'); // 两个元素，double 和 char 类型
auto t3 = make_tuple(12, 'a', string("How?")); // 三个元素，类型为 int、char 和 string
```

一个 tuple 可以包含很多有元素，因此我们不可能还是用 first 和 second 来访问它们。我们需要使用 get 函数：

```
auto d = get<0>(t1);           // double 元素
auto n = get<0>(t3);           // int 元素
auto c = get<1>(t3);           // char 元素
auto s = get<2>(t3);           // string 元素
```

get 的下标是作为模板实参提供的。如例子中所示，tuple 中元素的下标是从零开始的。元组大多用于泛型编程中。

## C.6.4 initializer\_list

<initializer\_list> 中定义了 initializer\_list：

```
template<typename T>
class initializer_list {
public:
    initializer_list() noexcept;

    size_t size() const noexcept;           // 元素个数
    const T* begin() const noexcept;       // 第一个元素
    const T* end() const noexcept;         // 尾后位置

    // ...
};
```

当编译器看到一个元素类型为 X 的 {} 初始化器列表时，就会用它构造一个 initializer\_list<X>（见 19.2.1 节和 13.2 节）。不幸的是，initializer\_list 不支持下标运算符（[]）。

## C.6.5 资源管理指针

C++ 内置指针并不能表明它是否拥有所指向对象的所有权。这会令编程严重复杂化（见 14.5 节）。定义在 <memory> 中的资源管理指针 unique\_ptr 和 shared\_ptr 可以解决这个问题：

- unique\_ptr（见 14.5.4 节）表示排他所有权；对于一个对象，只能有一个 unique\_ptr 指向它，当此 unique\_ptr 被销毁时，对象被释放。
- shared\_ptr 表示共享的所有权；可以有很多 shared\_ptr 指向同一个对象，当其中最后一个 shared\_ptr 销毁时，对象被释放。

---

### unique\_ptr<p>（简化的）

---

unique_ptr up0;	默认构造函数：up 拥有 nullptr
unique_ptr up(p);	up 拥有 p
unique_ptr up(up2);	移动构造函数：up 拥有 up2 的 p；up2 拥有空指针
up.~unique_ptr()	释放 up 拥有的指针

---

(续)

**unique\_ptr<p> (简化的)**

p=up.get()	p 为 up 拥有的指针
p=up.release()	p 为 up 拥有的指针; up 拥有 nullptr
up.reset(p)	释放 up 拥有的指针; up 拥有 p
up=make_unique<X>(args)	up 拥有 new<X>(args) (C++14)

我们可以对 `unique_ptr` 使用 `*`、`->`、`==` 和 `<` 这些常用的指针运算。此外, 定义一个 `unique_ptr` 时还可以指定使用不同于普通 `delete` 的释放操作。

**shared\_ptr<p> (简化的)**

shared_ptr sp{};	默认构造函数: sp 拥有 nullptr
shared_ptr sp(p);	sp 拥有 p
shared_ptr sp(sp2);	拷贝构造函数: sp 和 sp2 都拥有 sp2 的 p
shared_ptr sp(move(sp2));	移动构造函数: sp 拥有 sp2 的 p; sp2 拥有空指针
sp.~shared_ptr()	若 sp 是其拥有的指针的最后一个 shared_ptr, 则释放该指针
sp = sp2	拷贝赋值: 若 sp 是其拥有的指针的最后一个共享指针, 则释放该指针; sp 和 sp2 都拥有 sp2 的 p
sp = move(sp2)	移动赋值: 若 sp 是其拥有的指针的最后一个共享指针, 则释放该指针; sp 拥有 sp2 的 p; sp2 拥有空指针
p=sp.get()	p 为 sp 拥有的指针
p=sp.use_count()	有多少 shared_ptr 共享 sp 所拥有的指针?
sp.reset(p)	若 sp 是其拥有的指针的最后一个共享指针, 则释放该指针; sp 拥有 p
sp=make_shared<X>(args)	sp 拥有 new<X>(args)

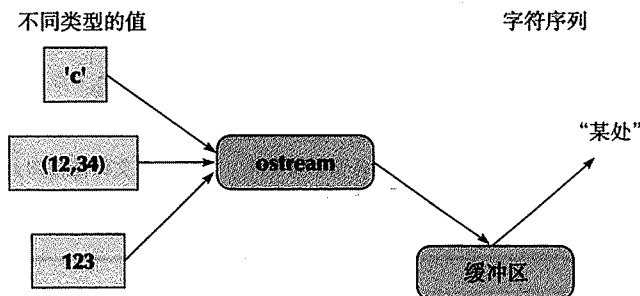
我们可以对 `shared_ptr` 使用 `*`、`->`、`==` 和 `<` 这些常用的指针运算。此外, 定义一个 `shared_ptr` 时还可以指定使用不同于普通 `delete` 的释放操作。

标准库还提供了一个 `weak_ptr` 用于打破 `shared_ptr` 的循环。

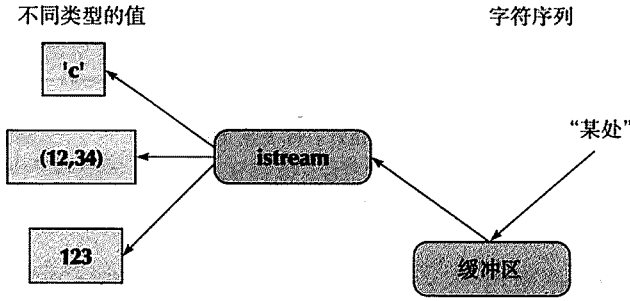
## C.7 I/O 流

I/O 流库提供了文本和数值的格式化和未格式化的缓冲 I/O 功能。I/O 流特性在 `<istream>`、`<ostream>` 等头文件中定义, 参见附录 C.1.1。

一个 `ostream` 可以将有类型的对象转换为字符(字节)流:



`istream` 可以将字符(字节)序列转换为有类型的对象:



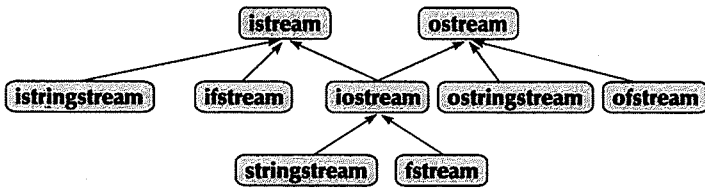
一个 `iostream` 既可以像 `istream` 一样工作，也可以像 `ostream` 一样工作。上图中的缓冲区是“流缓冲区”（`streambuf` 对象）。如果你需要将 `iostream` 映射到一种新设备、文件或者内存，请查阅专家级书籍来获得 `streambuf` 的更详细的内容。

STL 中提供了三种标准流：

标准 I/O 流	
<code>cout</code>	标准字符输出（通常默认是显示器）
<code>cin</code>	标准字符输入（通常默认是键盘）
<code>cerr</code>	标准字符错误输出（未缓冲的）

### C.7.1 I/O 流层次

一个 `istream` 可以连接至一个输入设备（如键盘）、一个文件或者一个 `string`。类似地，一个 `ostream` 可以连接至一个输出设备（如文本窗口）、一个文件或者一个 `string`。I/O 流特性组织为一个类层次：



我们可以通过构造函数或者 `open()` 调用来打开一个流：

流类型	
<code>stringstream(m)</code>	以模式 <code>m</code> 创建一个空的字符流
<code>stringstream(s,m)</code>	以模式 <code>m</code> 创建一个包含字符串 <code>s</code> 的字符串流
<code>fstream()</code>	创建一个文件流，稍后可以打开
<code>fstream(s,m)</code>	以模式 <code>m</code> 打开名为 <code>s</code> 的文件，创建一个文件流指向此文件
<code>fs.open(s,m)</code>	以模式 <code>m</code> 打开名为 <code>s</code> 的文件，令 <code>fs</code> 指向它
<code>fs.is_open()</code>	<code>fs</code> 已经打开？

对于文件流，文件名是 C 风格字符串。

你可以以一种或者多种模式打开一个文件：

**流模式**

<code>ios_base::app</code>	追加模式（即添加到文件末尾）
<code>ios_base::ate</code>	“文件尾”模式（打开文件，并定位到文件尾）
<code>ios_base::binary</code>	二进制模式——要小心系统相关的行为
<code>ios_base::in</code>	读模式
<code>ios_base::out</code>	写模式
<code>ios_base::trunc</code>	将文件长度截断为 0

对于每种模式，确切的效果依赖于操作系统。如果操作系统不允许以某种特定方式打开文件，则流会进入非 `good()` 状态。

下面是一个例子：

```
void my_code(ostream& os);           // 我的代码可以使用任意 ostream

ostream os;                          // o 表示 “output”
ofstream of("my_file");
if (!of) error("couldn't open 'my_file' for writing");
my_code(os);                          // 使用一个字符串
my_code(of);                          // 使用一个文件
```

参见 11.3 节。

**C.7.2 I/O 错误处理**

一个 `iostream` 可以处于下列四种状态之一：

**流状态**

<code>good()</code>	操作成功
<code>eof()</code>	到达输入结尾（“文件尾”）
<code>fail()</code>	发生了不期望的事情（例如寻找数字却遇到 'x'）
<code>bad()</code>	发生了不期望的严重问题（例如磁盘读错误）

程序员可以通过使用 `s.exceptions()`，来要求 `iostream` 在从 `good()` 状态转到其他状态时抛出一个异常（见 10.6 节）。

如果流处于非 `good()` 状态，试图对其进行操作将不会有任何效果，即“无操作”。

`iostream` 可以作为条件来使用——如果流状态为 `good()` 则条件为真（成功）。这样，读取流的程序通常就可以如下编写：

```
for (X buf; cin>>buf; ) { // buf 是一个 “输入缓冲区”，用来保存一个 X 类型的值
    // ... 对 buf 进行一些操作 ...
}
// 当 >> 不能从 cin 读取更多的 X 时，我们会到达这里
```

**C.7.3 输入操作**

除了字符串流的输入操作定义于 `<string>` 中之外，其他流输入操作都定义于 `<istream>` 中：

**格式化输入**

<code>in &gt;&gt; x</code>	根据 <code>x</code> 的类型，从 <code>in</code> 中读取数据存入 <code>x</code>
<code>getline(in,s)</code>	从 <code>in</code> 中读取一行，存入字符串 <code>s</code>

除非特别指出，否则 `istream` 操作都会返回指向其 `istream` 的引用，因此我们可以将操作“链接”起来，如 `cin>>x>>y;`

未格式化输入	
<code>x=in.get()</code>	从 <code>in</code> 读取一个字符，返回其整型值
<code>in.get(c)</code>	从 <code>in</code> 读取一个字符，存入 <code>c</code>
<code>in.get(p,n)</code>	从 <code>in</code> 读取最多 <code>n</code> 个字符，存入 <code>p</code> 开始的数组
<code>in.get(p,n,t)</code>	从 <code>in</code> 读取最多 <code>n</code> 个字符，存入 <code>p</code> 开始的数组， <code>t</code> 作为结束符
<code>in.getline(p,n)</code>	从 <code>in</code> 读取最多 <code>n</code> 个字符，存入 <code>p</code> 开始的数组，从 <code>in</code> 中删除结束符
<code>in.getline(p,n,t)</code>	从 <code>in</code> 读取最多 <code>n</code> 个字符，存入 <code>p</code> 开始的数组， <code>t</code> 作为结束符，从 <code>in</code> 中删除结束符
<code>in.read(p,n)</code>	从 <code>in</code> 读取最多 <code>n</code> 个字符，存入 <code>p</code> 开始的数组
<code>x=in.gcount()</code>	<code>x</code> 为 <code>in</code> 的最后一次未格式化输入操作所读取的字符数
<code>in.unget()</code>	回退流，因此读入的下一个字符与上一个字符一样
<code>in.putback(x)</code>	将 <code>x</code> “退还”到流中，因此读入的下一个字符就是 <code>x</code>

函数 `get()` 和 `getline()` 在写入到 `p[0]...` 的字符序列（如果有的话）末尾放置一个 `0`；如果读取到结束符 (`t`)，`getline()` 将其从输入流中删除，而 `get()` 则不会删除。`read(p,n)` 不会在字符序列末尾写入一个 `0`。显然，与非格式化输入操作相比，格式化输入操作更容易使用，也更不容易出错。

## C.7.4 输出操作

除了字符串流的输出操作定义于 `<string>` 之外，其他流输出操作都定义于 `<ostream>`：

输出操作	
<code>out&lt;&lt;x</code>	根据 <code>x</code> 的类型，将 <code>x</code> 写入 <code>out</code>
<code>out.put(c)</code>	将字符 <code>c</code> 写入 <code>out</code>
<code>out.write(p,n)</code>	将字符 <code>p[0]..p[n-1]</code> 写入 <code>out</code>

除非特别指出，否则 `ostream` 操作都会返回指向其 `ostream` 的引用，因此我们可以将操作“链接”起来，如 `cout<<x<<y;`

## C.7.5 格式化

流 I/O 的格式由对象类型、流状态、本地化信息（见 `<locale>`）及显式操作共同控制。第 10 章和第 11 章对此进行了详细介绍，因此本节只是列出标准的格式操纵符（改变流状态的操作），因为这些操纵符提供了最直接的改变格式的方法。

本地化的相关内容已经超出了本书的范围。

## C.7.6 标准格式操纵符

标准库提供了一些操纵符，与不同的格式状态和状态改变相对应。标准操纵符定义于 `<ios>`、`<istream>`、`<ostream>`、`<iostream>` 和 `<iomanip>`（接受参数的操纵符）中：

## I/O 操纵符

<code>s&lt;&lt;boolalpha</code>	使用 true 和 false 的符号表示 (输入和输出)
<code>s&lt;&lt;noboolalpha</code>	<code>s.unsetf(ios_base::boolalpha)</code>
<code>s&lt;&lt;showbase</code>	八进制输出加前缀 0, 十六进制加前缀 0x
<code>s&lt;&lt;noshowbase</code>	<code>s.unsetf(ios_base::show_base)</code>
<code>s&lt;&lt;showpoint</code>	总是显示小数点
<code>s&lt;&lt;noshowpoint</code>	<code>s.unsetf(ios_base::showpoint)</code>
<code>s&lt;&lt;showpos</code>	对于正数显示 +
<code>s&lt;&lt;noshowpos</code>	<code>s.unsetf(ios_base::showpos)</code>
<code>s&gt;&gt;skipws</code>	跳过空白符
<code>s&gt;&gt;noskipws</code>	<code>s.unsetf(ios_base::skipws)</code>
<code>s&gt;&gt;uppercase</code>	在数值输出中使用大些字母, 如 1.2E10 和 0X1A2, 而不是 1.2e10 和 0x1a2
<code>s&gt;&gt;nouppercase</code>	使用 x 和 e 而不是 X 和 E
<code>x&lt;&lt;internal</code>	在格式模式中指定的位置进行填充
<code>x&lt;&lt;left</code>	在值之后填充
<code>x&lt;&lt;right</code>	在值之前填充
<code>s&lt;&lt;dec</code>	整数基底设置为 10
<code>s&lt;&lt;hex</code>	整数基底设置为 16
<code>s&lt;&lt;oct</code>	整数基底设置为 8
<code>s&lt;&lt;fixed</code>	采用浮点格式 dddd.dd
<code>s&lt;&lt;scientific</code>	采用科学记数法格式 d.ddddEdd
<code>s&lt;&lt;defaultfloat</code>	所有格式给出最精确的浮点输出
<code>s&lt;&lt;endl</code>	输出 '\n' 并清除缓冲
<code>s&lt;&lt;ends</code>	输出 '\0'
<code>s&lt;&lt;flush</code>	清除流
<code>s&gt;&gt;ws</code>	吃掉空白符
<code>s&lt;&lt;resetiosflags(f)</code>	清空标志 f
<code>s&lt;&lt;setiosflags(f)</code>	将标志设置为 f
<code>s&lt;&lt;setbase(b)</code>	以基底 b 输出整数
<code>s&lt;&lt;setfill(c)</code>	将填充字符设置为 c
<code>s&lt;&lt;setprecision(n)</code>	精度设置为 n 个数字
<code>s&lt;&lt;setw(n)</code>	下个域的宽度为 n 个字符

每个操作都返回指向第一个运算对象 s (流) 的引用, 例如:

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << endl;
```

输出结果为:

```
1234,4d2,2322
```

而

```
cout << '(' << setw(4) << setfill('#') << 12 << ")" (" << 12 << ")n";
```

输出

```
(##12) (12)
```

为了显式设置浮点数的输出格式, 我们可以使用

```
b.setf(ios_base::fmtflags(0), ios_base::floatfield)
```

参见第 11 章。



## C.8 字符串处理

标准库在 `<cctype>` 中提供了字符分类操作，在 `<string>` 中提供了字符串及相关操作，在 `<regex>` 中提供了正则表达式操作，在 `<cstring>` 中提供了 C 风格字符串的支持。

### C.8.1 字符分类

基本字符集中的字符可分类如下：

字符分类	
<code>isspace(c)</code>	c 是空白符 (' ', '\t', '\n' 等) ?
<code>isalpha(c)</code>	c 是字母 ('a'..'z', 'A'..'Z') ? (注意: 不包括 '_')
<code>isdigit(c)</code>	c 是十进制数字 ('0'..'9') ?
<code>isxdigit(c)</code>	c 是十六进制数字 (十进制数字或 'a'..'f' 或 'A'..'F') ?
<code>isupper(c)</code>	c 是大写字母?
<code>islower(c)</code>	c 是小写字母?
<code>isalnum(c)</code>	c 是字母或十进制数字?
<code>iscntrl(c)</code>	c 是控制字符 (ASCII 0..31 和 127) ?
<code>ispunct(c)</code>	c 不是字母、数字、空白符或可见控制字符?
<code>isprint(c)</code>	c 是可打印的 (ASCII '!..'~') ?
<code>isgraph(c)</code>	c 满足 <code>isalpha() isdigit() ispunct()</code> ? (注意: 不包括空格)

另外，标准库提供了两个有用的函数，可以消除大小写差别：

大小写	
<code>toupper(c)</code>	返回 c 或者 c 的对应大写形式
<code>tolower(c)</code>	返回 c 或者 c 的对应小写形式

标准库还支持扩展字符集，如 Unicode，但这些内容已经超出了本书的范围。

### C.8.2 字符串

标准库字符串类 `string` 是字符串模板 `basic_string` 对字符类型 `char` 的一个特例化，即 `string` 是 `char` 序列：

字符串运算	
<code>s=s2</code>	将 s2 赋予 s, s2 可以是一个字符串或者一个 C 风格字符串
<code>s+=x</code>	将 x 附加于 s 的末尾, x 可以是一个字符、一个字符串或者一个 C 风格字符串
<code>s[i]</code>	下标
<code>s+s2</code>	连接, 结果字符串中字符来自 s 后接来自 s2 的字符
<code>s==s2</code>	字符串比较, s 或 s2 可以为 C 风格字符串, 但不能两者均是
<code>s!=s2</code>	字符串比较, s 或 s2 可以为 C 风格字符串, 但不能两者均是
<code>s&lt;s2</code>	字符串字典序比较, s 或 s2 可以为 C 风格字符串, 但不能两者均是
<code>s&lt;=s2</code>	字符串字典序比较, s 或 s2 可以为 C 风格字符串, 但不能两者均是
<code>s&gt;s2</code>	字符串字典序比较, s 或 s2 可以为 C 风格字符串, 但不能两者均是
<code>s&gt;=s2</code>	字符串字典序比较, s 或 s2 可以为 C 风格字符串, 但不能两者均是
<code>s.size()</code>	s 中字符数
<code>s.length()</code>	s 中字符数

(续)

**字符串运算**

<code>s.c_str()</code>	s 的 C 风格字符串版本 (以 0 结束)
<code>s.begin()</code>	指向首字符的迭代器
<code>s.end()</code>	指向尾字符之后位置的迭代器
<code>s.insert(pos,x)</code>	将 x 插入到 <code>s[pos]</code> 之前, x 可以是字符串或 C 风格字符串
<code>s.append(x)</code>	将 x 插入到 s 的最后一个字符之后, x 可以是字符串或 C 风格字符串
<code>s.erase(pos)</code>	删除 s 中从 <code>s[pos]</code> 开始到末尾的所有字符。s 的大小变为 pos
<code>s.erase(pos,n)</code>	删除 s 中从 <code>s[pos]</code> 开始的 n 个字符。s 的大小变为 <code>max(pos,size-n)</code>
<code>s.push_back(c)</code>	追加字符 c
<code>pos=s.find(x)</code>	在 s 中查找 x, x 可以是字符、字符串或 C 风格字符串。pos 为找到的第一个字符的下标, 或者是 <code>string::npos</code> (s 末尾之后的位置)
<code>in&gt;&gt;s</code>	从 in 中读取一个词, 存入 s

**C.8.3 正则表达式匹配**

正则表达式工具定义在 `<regex>` 中, 其主要功能包括:

- 在 (任意长度) 数据流中搜索与正则表达式相匹配的字符串——`regex_search()`。
- 判定字符串 (已知长度) 是否与正则表达式匹配——`regex_match()`。
- 替换匹配成功的串——`regex_replace()`: 本书中并未讨论, 请参考专家级书籍或手册。

`regex_search()` 或 `regex_match()` 的结果是一个匹配结果的集合, 通常表示为 `smatch`:

```
regex row("^[\\w ]+( \\d+)( \\d+)( \\d+)$"); // 数据行

while (getline(in,line)) { // 检查数据行
    smatch matches;
    if (!regex_match(line, matches, row))
        error("bad line", lineno);
    // 检查行
    int field1 = from_string<int>(matches[1]);
    int field2 = from_string<int>(matches[2]);
    int field3 = from_string<int>(matches[3]);
    // ...
}
```

正则表达式的语法基于具有特殊含义的字符 (见第 23 章):

**正则表达式特殊字符**

<code>.</code>	任意单个字符 (“通配符”)
<code>[</code>	字符分类
<code>{</code>	计数
<code>(</code>	分组开始
<code>)</code>	分组结束
<code>\</code>	转义字符——下一字符具有特殊含义
<code>*</code>	零次或多次重复
<code>+</code>	一次或多次重复
<code>?</code>	可选 (零次或一次)
<code> </code>	二选一 (或)
<code>^</code>	行开始; 非
<code>\$</code>	行结束

重复	
{n}	重复 n 次
{n,}	重复 n 次或更多次
{n,m}	重复至少 n 次, 至多 m 次
*	重复零次或多次, 即 {0,}
+	重复一次或多次, 即 {1,}
?	可选 (零次或一次), 即 {0,1}

字符分类	
alnum	任意字母数字字符或者下划线
alpha	任意字母
blank	任意空白符, 行分隔符除外
cntrl	任意控制字符
d	任意数字
digit	任意数字
graph	任意图形字符
lower	任意小写字母
print	任意可打印字符
punct	任意标点符号
s	任意空白符
space	任意空白符
upper	任意大写字母
w	任意单词字符 (字母数字字符)
xdigit	任意十六进制数字字符

一些字符分类支持简写形式:

字符分类简写		
\d	十进制数字	[[digit:]]
\l	小写字母	[[lower:]]
\s	空白符 (空格、制表符等)	[[space:]]
\u	大写字母	[[upper:]]
\w	字母、十进制数字或者下划线 (_)	[[alnum:]]
\D	除 \d 之外任意字符	[^digit:]]
\L	除 \l 之外任意字符	[^lower:]]
\S	除 \s 之外任意字符	[^space:]]
\U	除 \u 之外任意字符	[^upper:]]
\W	除 \w 之外任意字符	[^alnum:]]

## C.9 数值

C++ 对数学计算 (如科学计算、工程计算等) 提供最基础的支持。

## C.9.1 数值限制

每个 C++ 编译器实现都指定了内置类型的属性，程序员可以使用这些属性来检查数值限制，设置“哨兵”（边界检测）等等。

我们可以从 `<limits>` 中获得每个内置类型或者库类型 `T` 的限制 `numeric_limits<T>`。此外，程序员还可以为用户自定义类型 `X` 定义限制 `numeric_limits<X>`。例如：

```
class numeric_limits<float> {
public:
    static const bool is_specialized = true;

    static constexpr int radix = 2;    // 指数的基（本例为二进制）
    static constexpr int digits = 24;  // 尾数部分数字位数
    static constexpr int digits10 = 6; // 尾数部分十进制数字位数

    static constexpr bool is_signed = true;
    static constexpr bool is_integer = false;
    static constexpr bool is_exact = false;

    static constexpr float min() { return 1.17549435E-38F; } // 数值实例
    static constexpr float max() { return 3.40282347E+38F; } // 数值实例
    static constexpr float lowest() { return -3.40282347E+38F; } // 数值实例

    static constexpr float epsilon() { return 1.19209290E-07F; } // 数值实例
    static constexpr float round_error() { return 0.5F; } // 数值实例

    static constexpr float infinity() { return /* 某个数值 */; }
    static constexpr float quiet_NaN() { return /* 某个数值 */; }
    static constexpr float signaling_NaN() { return /* 某个数值 */; }
    static constexpr float denorm_min() { return min(); }

    static constexpr int min_exponent = -125; // 数值实例
    static constexpr int min_exponent10 = -37; // 数值实例
    static constexpr int max_exponent = +128; // 数值实例
    static constexpr int max_exponent10 = +38; // 数值实例
    static constexpr bool has_infinity = true;
    static constexpr bool has_quiet_NaN = true;
    static constexpr bool has_signaling_NaN = true;
    static constexpr float_denorm_style has_denorm = denorm_absent;
    static constexpr bool has_denorm_loss = false;

    static constexpr bool is_iec559 = true; // 服从 IEC-559
    static constexpr bool is_bounded = true;
    static constexpr bool is_modulo = false;
    static constexpr bool traps = true;
    static constexpr bool tinyness_before = true;

    static constexpr float_round_style round_style = round_to_nearest;
};
```

从 `<limits.h>` 和 `<float.h>` 中，我们可以获得指明整数和浮点数的一些关键属性的宏，包括：

---

### 数值限制宏

CHAR_BIT	一个 char 中的二进制位数（通常为 8）
CHAR_MIN	char 的最小值
CHAR_MAX	char 的最大值（如果 char 是有符号的，通常为 127；如果 char 是无符号的，通常为 255）

---

## C.9.2 标准数学函数

标准库提供了最常用的数学函数（定义于 `<cmath>` 和 `<complex>`）：

标准数学函数	
<code>abs(x)</code>	绝对值
<code>ceil(x)</code>	$\geq x$ 的最小整数
<code>floor(x)</code>	$\leq x$ 的最大整数
<code>round(x)</code>	取整到最接近的整数（0.5 认为离 0 更远）
<code>sqrt(x)</code>	平方根，x 必须是非负数
<code>cos(x)</code>	余弦
<code>sin(x)</code>	正弦
<code>tan(x)</code>	正切
<code>acos(x)</code>	反余弦，结果是非负数
<code>asin(x)</code>	反正弦，返回最接近 0 的值
<code>atan(x)</code>	反正切
<code>sinh(x)</code>	双曲正弦
<code>cosh(x)</code>	双曲余弦
<code>tanh(x)</code>	双曲正切
<code>exp(x)</code>	基底为 e 的指数
<code>log(x)</code>	自然对数，基底为 e，x 必须是正数
<code>log10(x)</code>	基底为 10 的对数

这些函数都有分别针对 `float`、`double`、`long double` 和 `complex` 的不同版本。对每个函数，返回值类型与参数类型一致。

如果一个标准数学函数不能生成数学上有效的结果，它会恰当设置 `errno`。

## C.9.3 复数

标准库提供了复数类型 `complex<float>`、`complex<double>` 和 `complex<long double>`。如果 `Scalar` 是其他某种支持常用算术运算的类型，`complex<Scalar>` 通常也能正常工作，但不保证是可移植的。

```
template<class Scalar> class complex {
    // 一个复数是一对标量值，基本上可以认为是一个坐标对
    Scalar re, im;
public:
    constexpr complex(const Scalar & r, const Scalar & i) :re(r), im(i) {}
    constexpr complex(const Scalar & r) :re(r), im(Scalar{}) {}
    constexpr complex() :re{Scalar{}}, im{Scalar[]} {}

    Scalar real() { return re; } // 实部
    Scalar imag() { return im; } // 虚部

    // 运算符: = += -= *= /=
};
```

除了复数的成员之外，`<complex>` 还提供了大量有用的操作：

## 复数运算符

<code>z1+z2</code>	加
<code>z1-z2</code>	减
<code>z1*z2</code>	乘
<code>z1/z2</code>	除
<code>z1==z2</code>	相等性
<code>z1!=z2</code>	不相等
<code>norm(z)</code>	<code>abs(z)</code> 的平方
<code>conj(z)</code>	共轭: 若 $z$ 为 $\{re,im\}$ , 则 <code>conj(z)</code> 为 $\{re,-im\}$
<code>polar(x,y)</code>	由极坐标 ( $\rho, \theta$ ) 构造一个复数
<code>real(z)</code>	复数的实部
<code>imag(z)</code>	虚部
<code>abs(z)</code>	也称为 $\rho$
<code>arg(z)</code>	也称为 $\theta$
<code>out&lt;&lt;z</code>	输出复数
<code>in&gt;&gt;z</code>	输入复数

标准数学函数 (见附录 C.9.2) 也都有复数版本。注意: `complex` 不提供 `<` 或 `%`, 参见 24.9 节。

## C.9.4 valarray

标准库 `valarray` 是一维数值数组, 即, 它为数组类型 (更像第 24 章中的 `Matrix`) 提供了算术运算, 并提供了对切片和跨越访问的支持。

## C.9.5 泛型数值算法

下面函数均来自 `<numeric>`, 它们提供了数值序列的通用运算的泛型版本:

## 数值算法

<code>x=accumulate(b,e,i)</code>	$x$ 是 $i$ 和 $[b:e)$ 中元素之和
<code>x=accumulate(b,e,i,f)</code>	累计, 但使用 $f$ 代替 $+$
<code>x=inner_product(b,e,b2,i)</code>	$x$ 是 $[b:e)$ 和 $[b2:b2+(e-b))$ 的内积, 即, $x$ 是 $i$ 和所有 $(*p1)*(*p2)$ 之和, $p1$ 是 $[b:e)$ 中元素, $p2$ 是 $[b2:b2+(e-b))$ 中对应元素
<code>x=inner_product(b,e,b2,l,f,f2)</code>	内积, 但分别用 $f$ 和 $f2$ 代替 $+$ 和 $*$
<code>p=partial_sum(b,e,out)</code>	$[out:p)$ 中第 $i$ 个元素是 $[b:e)$ 中第 $0$ 个到第 $i$ 个元素之和
<code>p=partial_sum(b,e,out,f)</code>	部分和, 使用 $f$ 代替 $+$
<code>p=adjacent_difference(b,e,out)</code>	对 $i>1$ , $[out:p)$ 中第 $i$ 个元素是 $*(b+i)-*(b+i-1)$ ; 若 $e-b>0$ , 则 $*out$ 是 $*b$
<code>p=adjacent_difference(b,e,out,f)</code>	相邻差, 用 $f$ 代替 $-$
<code>iota(b,e,v)</code>	将 $[b:e)$ 中所有元素都赋值为 $++v$

例如:

```
vector<int> v(100);
iota(v.begin(),v.end(),0); // v={1,2,3,4,5,...,100}
```

### C.9.6 随机数

标准库在 `<random>` 中提供了随机数发生器和分布（见 24.7 节）。默认使用 `default_random_engine`，它应用范围最广、代价最低。

随机数分布包括：

分布	
<code>uniform_int_distribution&lt;int&gt; {low, high}</code>	[low:high] 中的值
<code>uniform_real_distribution&lt;int&gt; {low, high}</code>	[low:high) 中的值
<code>exponential_distribution&lt;double&gt;{lambda}</code>	[0: $\infty$ ) 中的值
<code>bernoulli_distribution(p)</code>	[true:false] 中的值
<code>normal_distribution&lt;double&gt;{median,spread}</code>	$[-\infty : \infty)$ 中的值

我们可以将发生器作为分布的参数。例如：

```
uniform_real_distribution<> dist;
default_random_engine engn;
for (int i = 0; i<10; ++i)
    cout << dist(engn) << ' ';
```

### C.10 时间组件

标准库在 `<chrono>` 中提供了计时工具。一个时钟会以时钟周期为单位计数时间，可通过调用 `now()` 报告当前时间点。标准库定义了三种时钟：

- `system_clock`：默认的系统时钟。
- `steady_clock`：对于这种时钟 `c`，连续调用 `now()` 满足 `c.now()<=c.now()`，且时钟周期的间隔时间固定。
- `high_resolution_clock`：一个系统中精度最高的时钟。

通过函数 `duration_cast<>()`，我们可以将一个给定时钟的时钟周期数转换为传统的时间单位，如 `seconds`、`milliseconds` 和 `nanoseconds`。例如：

```
auto t = steady_clock::now();
// ... 执行一些操作 ...
auto d = steady_clock::now()-t;    // 操作花费了 d 个时间单位

cout << "something took "
    << duration_cast<milliseconds>(d).count() << "ms";
```

这段代码会打印出“一些操作”花费了多少微秒。另可参考 26.6.1 节。

### C.11 C 标准库函数

C 语言标准库在集成入 C++ 标准库时只进行了很小的修改。C 标准库中提供的函数，都是在大量不同实际环境中，特别是相对低层的程序设计领域，经过长期验证，被证明是非常有用的函数。在本节中，我们将它们组织为几个传统的类别来进行介绍：

- C 风格 I/O。
- C 风格字符串。
- 内存。
- 日期和时间。

- 其他。

还有很多 C 标准库函数本节没有介绍，如果你需要了解这些函数，请参考一本好的教材，如 Kernighan 和 Ritchie 的《The C++ Programming Language》(K&R)。

### C.11.1 文件

<stdio> 定义的 I/O 系统是基于“文件”的。文件 (FILE \*) 可以指向文件或者一个标准输入输出流 `stdin`、`stdout` 和 `stderr`。标准流可直接使用，其他文件需要显式打开：

文件打开和关闭	
<code>f=fopen(s,m)</code>	以模式 <code>m</code> 打开一个文件 <code>s</code>
<code>x=fclose(f)</code>	关闭文件 <code>f</code> ，若成功返回 0

“模式”是一个字符串，包含一个或多个指明文件如何打开的指令：

文件模式	
"r"	读
"w"	写 (丢弃已有内容)
"a"	追加 (添加到末尾)
"r+"	读和写
"w+"	读和写 (丢弃已有内容)
"b"	二进制模式，与其他一个或多个模式一起使用

在一个特定系统中，可能有更多模式 (通常也确实是的)。一些模式可以组合，例如：`fopen("foo","rb")` 试图打开文件 `foo`，用于二进制读。`stdio` 和 `iostream` (见附录 C.7.1) 的 I/O 模式应该是相同的。

### C.11.2 printf() 函数家族

最常用的 C 标准库函数是 I/O 函数。但是，我们推荐使用 `iostream`，因为它是类型安全且可扩展的。格式化输出函数 `printf()` 应用非常广泛 (在 C++ 程序中也广泛使用)，也被其他程序设计语言所广泛模仿。

printf	
<code>n=printf(fmt,args)</code>	将参数 <code>args</code> 恰当地插入“格式串” <code>fmt</code> 中，将其输出到 <code>stdout</code>
<code>n=fopen(f,fmt,args)</code>	将参数 <code>args</code> 恰当地插入“格式串” <code>fmt</code> 中，将其输出到文件 <code>f</code>
<code>n=sprintf(s,fmt,args)</code>	将参数 <code>args</code> 恰当地插入“格式串” <code>fmt</code> 中，将其输出到 C 风格字符串 <code>s</code>

对于每个版本，`n` 返回打印的字符数，或者是一个负数表示输出错误。`printf()` 的返回值一般应该忽略。

`printf()` 的声明如下：

```
int printf(const char* format...);
```

换句话说，它接受一个 C 风格字符串 (通常是一个字符串文字常量)，后跟任意数量、任意类型的参数。这些“额外参数”的含义由格式串中的转换说明符，如 `%c` (打印字符) 和 `%d`



(打印十进制整数)来控制。例如:

```
int x = 5;
const char* p = "asdf";
printf("the value of x is '%d' and the value of p is '%s'\n",x,p);
```

% 后跟一个字符控制了如何处理参数。第一个 % 应用于第一个“额外参数”(本例中, %d 应用于 x), 第二个 % 应用于第二个“额外参数”(本例中, %s 应用于 p), 依此类推。本例中 printf() 的输出结果为:

```
the value of x is '5' and the value of p is 'asdf'
```

后接一个换行。

一般来说, 无法检查 % 转换指令与其所应用的参数类型之间的对应关系, 即便可以, printf() 通常也不会检查。例如:

```
printf("the value of x is '%s' and the value of p is '%d'\n",x,p); // 糟糕
```

C 标准库提供了大量转换说明符, 从而提供了极大的灵活性(另一方面也容易导致混淆)。在 % 之后, 可使用下面这些转换说明符:

- 可选的减号, 指明输出值在域中左对齐。
- + 可选的加号, 指明有符号类型的值总是以 + 和 - 开始, 来表明正负值。
- 0 可选的零, 指明使用前导 0 来对数值进行填充。如果指定了 - 或者精度, 则忽略 0。
- # 可选的 # 号, 指明浮点值必须打印小数点, 即便之后没有非 0 数字——这种情况打印结尾 0, 它的另外两种含义是八进制数打印一个前缀 0, 及十六进制数打印前缀 0x 或 0X。
- d 可选的数字串, 指明域宽, 如果输出值的字符数小于域宽, 则在左侧填充空格(或者右侧, 如果指定了左对齐指示符的话); 如果域是以 0 开始的话, 则填充 0 而不是空格。
- . 可选的句点, 用来将域宽与下个数字串分开。
- dd 可选的数字串, 指明精度: 对于转换符 e 和 f 来说, 指出小数点后有多少位数字; 或者是一个字符串中最多打印多少个字符。
- \* 域宽和精度可以用 \* 而不是数字串来指定, 这种情况下, 额外参数中的一个整型值用来指定域宽和精度。
- h 可选的字母 h, 指明后面的 d、o、x 或 u 对应短整型参数。
- l 可选的字母 l, 指明后面的 d、o、x 或 u 对应长整型参数。
- L 可选的字母 L, 指明后面的 e、E、g、G 或 f 对应 long double 型参数。
- % 指出打印一个字符 %, 不使用任何参数。
- c 一个指明转换类型的字符, C 标准库提供的转换符及其含义如下:
  - d 将整数参数转换为十进制表示。
  - i 将整数参数转换为十进制表示。
  - o 将整数参数转换为八进制表示。
  - x 将整数参数转换为十六进制表示。
  - X 将整数参数转换为十六进制表示。
  - f 将 float 或 double 参数转换为十进制表示, 格式为 [-]ddd.ddd。小数点之后的数字个数与参数对应的精度相等。如果需要的话, 对数值进行舍入。如果未指定精

度，则打印小数点后六位。如果显式地指定了精度为 0，且未指定 #，则不打印小数点。

e 将 float 或 double 参数转换为十进制表示，格式为科学计数法 `[-]d.ddde+dd` 或者 `[-]d.ddde-dd`，小数点之前只有一位数字，小数点之后的数字个数与参数对应的精度相等。如果需要的话，对数值进行舍入。如果未指定精度，则打印小数点后六位。如果显式地指定了精度为 0，且未指定 #，则不打印小数点。

E 与 e 相同，但指数之前用大写的 E。

g 将 float 或 double 参数打印为格式 d、f 或 e，哪种格式能以最少的空间表示最大的精度，就采用哪种格式。

G 与 g 相同，但指数之前用大写的 E。

c 打印一个字符参数，忽略空字符。

s 接受一个字符串参数（字符指针），打印其中的字符，直至遇到空字符或者达到精度上限。但是，如果精度为 0 或未指定精度，则空字符之前的所有字符均被打印。

p 打印一个指针，打印格式依赖于具体实现。

u 将无符号整数转换为十进制表示。

n 将 printf()、fprintf() 或 sprintf() 所打印的字符数写入整型指针参数指向的地址。

如果未指定域宽，或者指定的域宽很小，不会对域进行截断；只有当指定域宽超出实际域宽时，才会进行填充。

由于 C 的用户自定义类型与 C++ 含义不同，因此不能为 complex、vector 或 string 等用户自定义类型定义输出格式。

C 的标准输出 stdout 与 cout 相对应。C 的标准输入 stdin 与 cin 相对应。C 的标准错误输出 stderr 与 cerr 相对应。C 标准 I/O 和 C++ I/O 流之间的关系是非常紧密的，甚至两者的缓冲区都是共享的。例如，混合使用 cout 和 stdout 可以生成单一的输出流（这在 C/C++ 混合程序中并不罕见）。但这种灵活性也会带来额外的代价。出于性能考虑，不要对同一个流混合使用 stdio 和 iostream 操作，并且在第一个 I/O 操作前调用 `ios_base::sync_with_stdio(false)`。

stdio 库提供了一个名为 scanf() 的函数，它是输入操作，风格与 printf() 类似。例如：

```
int x;
char s[buf_size];
int i = scanf("the value of x is '%d' and the value of s is '%s'\n",&x,s);
```

在这段代码中，scanf() 试图读取一个整数存入 x，并读取一个非空白字符序列存入 s。格式串中包含一些非格式控制字符，用户的输入中必须包含这些字符，如输入以下内容：

```
the value of x is '123' and the value of s is 'string '\n'
```

则上面例程会读取 123 存入 x，并将 string 后跟一个 0 存入 s。如果 scanf() 操作成功，返回结果（上例的 i）将是成功赋值的参数指针数（上例中期望是 2）；否则，返回 EOF 表示失败。这种指定输入格式的方式很容易出错（例如，在上例中，如果忘记输入 string 后面的空格，会发生什么情况？）。传递给 scanf() 的参数必须都是指针，我们强烈建议不使用 scanf()。

那么，如果不得不使用 stdio，如何来进行输入呢？一个常见的回答是“使用标准库函数 gets()”：

```
// 非常危险的代码
char s[buf_size];
char* p = gets(s);    // 读取一行存入 s
```

`p=gets(s)` 会一直读取字符存入 `s`，直至遇到换行或者文件尾，在最后一个字符之后放置一个 0。如果开始就遇到文件尾，或者发生了错误，则将 `p` 置为 `NULL`（也就是 0）；否则将 `s` 赋予 `p`。不要使用 `gets(s)` 或大致等价的 `scanf("%s",s)`！长久以来，它们一直是病毒设计者的最爱：通过输入大量数据，使缓冲区（上例中的 `s`）溢出，黑客就可以使程序崩溃甚至接管计算机。函数 `sprintf()` 也存在这种缓冲区溢出问题。

`stdio` 库也提供了简单有效的字符输入输出函数：

stdio 字符函数	
<code>x=getc(st)</code>	从输入流 <code>st</code> 读入一个字符，返回字符的整型值；如果到达文件尾或发生错误返回 <code>EOF</code>
<code>x=putc(c,st)</code>	将字符 <code>c</code> 写入输出流 <code>st</code> ，返回 <code>c</code> 的整型值；若发生错误返回 <code>EOF</code>
<code>x=getchar()</code>	从 <code>stdin</code> 读取一个字符，返回字符的整型值，遇到文件尾或发生错误返回 <code>EOF</code>
<code>x=putchar(c)</code>	将字符 <code>c</code> 写入 <code>stdout</code> ，返回字符的整型值，若发生错误返回 <code>EOF</code>
<code>x=ungetc(c,st)</code>	将 <code>c</code> 退回输入流 <code>st</code> ，返回字符的整型值，若发生错误返回 <code>EOF</code>

注意，这些函数的返回值是一个 `int`（而不是一个 `char`，否则就不能返回 `EOF` 了）。例如，下面是一个典型的 C 风格输入循环：

```
int ch; /* 不是 char ch; */
while ((ch=getchar())!=EOF) { /* 执行一些操作 */}
```

不要对一个流连续做两次 `ungetc()`，这种操作方式的结果是未定义的，因此这种代码是不具有可移植性的。

除上述函数之外，`stdio` 库还包含其他很多函数，如果你希望了解更多，请参考 K&R 这类好的 C 教材。

### C.11.3 C 风格字符串

C 风格字符串是以 0 结尾的字符数组。支持这种字符串表示法的函数都定义于 `<cstring>`（或 `<string.h>`，注意，不是 `<string>`）和 `<cstdlib>` 中。这些函数对 `char*` 指针指向的字符串进行操作（如果是 `const char*`，则是只读的）：

C 风格字符串操作	
<code>x=strlen(s)</code>	统计字符数（不包括结尾 0）
<code>p=strncpy(s,s2)</code>	将 <code>s2</code> 拷贝到 <code>s</code> ； <code>[s:s+n)</code> 和 <code>[s2:s2+n)</code> 两个区间不能重叠；将 <code>s</code> 返回给 <code>p</code> ；结尾 0 也拷贝
<code>p=strcat(s,s2)</code>	将 <code>s2</code> 拷贝到 <code>s</code> 末尾；将 <code>s</code> 返回给 <code>p</code> ；结尾 0 也拷贝
<code>x=strcmp(s,s2)</code>	字典序比较：如果 <code>s&lt;s2</code> 返回负值；若 <code>s==s2</code> 返回 0；若 <code>s&gt;s2</code> 返回正值
<code>p=strncpy(s,s2,n)</code>	类似 <code>strncpy</code> ；最多拷贝 <code>n</code> 个字符；可能无法拷贝结尾 0；返回 <code>s</code>
<code>p=strncat(s,s2,n)</code>	类似 <code>strcat</code> ；最多连接 <code>n</code> 个字符；可能无法拷贝结尾 0；返回 <code>s</code>
<code>p=strncmp(s,s2,n)</code>	类似 <code>strcmp</code> ；最多比较 <code>n</code> 个字符
<code>p=strchr(s,c)</code>	返回指针，指向 <code>s</code> 中 <code>c</code> 第一次出现的位置
<code>p=strrchr(s,c)</code>	返回指针，指向 <code>s</code> 中 <code>c</code> 最后一次出现的位置
<code>p=strstr(s,s2)</code>	返回指针，指向 <code>s</code> 中与 <code>s2</code> 相等的字符串的首字符

(续)

**C 风格字符串操作**

<code>p=strpbrk(s,s2)</code>	返回指针, 指向 <code>s</code> 中第一个也在 <code>s2</code> 中出现的字符
<code>x=atof(s)</code>	从 <code>s</code> 中提取出一个 <code>double</code> 值
<code>x=atoi(s)</code>	从 <code>s</code> 中提取出一个 <code>int</code> 值
<code>x=atol(s)</code>	从 <code>s</code> 中提取出一个 <code>long int</code> 值
<code>x=strtod(s,p)</code>	从 <code>s</code> 中提取出一个 <code>double</code> 值, <code>p</code> 指向 <code>double</code> 值之后的第一个字符
<code>x=strtol(s,p)</code>	从 <code>s</code> 中提取出一个 <code>long int</code> 值, <code>p</code> 指向 <code>long int</code> 值之后的第一个字符
<code>x=strtoul(s,p)</code>	从 <code>s</code> 中提取出一个 <code>unsigned long int</code> 值, <code>p</code> 指向 <code>unsigned long int</code> 值之后的第一个字符

注意, 在 C++ 中, `strchr()` 和 `strstr()` 都有两个版本, 来实现类型安全 (它们不能将 `const char*` 转换为 `char*`, 而 C 版本是可以的), 参见 27.5 节。

最后几个函数在 C 风格字符串内查找常规的数值表示形式, 如 “124” 和 “1.4”。如果未找到, 则返回 0。例如:

```
int x = atoi("fortytwo");    /* x 变为 0 */
```

**C.11.4 内存**

内存处理函数通过 `void*` 指针在 “原始内存” (类型未知的内存区域) 上进行操作 (`const void*` 指针对应只读内存):

**C 风格内存操作**

<code>q=memcpy(p,p2,n)</code>	从 <code>p2</code> 开始拷贝 <code>n</code> 个字节到 <code>p</code> (类似 <code>strcpy</code> ); [ <code>p:p+n</code> ] 和 [ <code>p2:p2+n</code> ] 两个区间不能重叠; 将 <code>p</code> 返回给 <code>q</code>
<code>q=memmove(p,p2,n)</code>	从 <code>p2</code> 开始拷贝 <code>n</code> 个字节到 <code>p</code> ; 将 <code>p</code> 返回给 <code>q</code>
<code>x=memcmp(p,p2,n)</code>	比较 <code>p2</code> 开始的 <code>n</code> 个字节和 <code>p</code> 中对应的 <code>n</code> 个字节 (类似 <code>strcmp</code> )
<code>q=memchr(p,c,n)</code>	在 <code>p[0]..p[n-1]</code> 中查找 <code>c</code> (转换为 <code>unsigned char</code> ), 若找到, 返回指向该字节的指针, 否则返回 0
<code>q=memset(p,c,n)</code>	将 <code>c</code> (转换为 <code>unsigned char</code> ) 赋予 <code>p[0]..p[n-1]</code> 中每个字节, 返回 <code>p</code>
<code>p=calloc(n,s)</code>	在自由内存空间中分配 <code>n*s</code> 个字节, 全部初始化为 0; 若分配失败返回 0
<code>p=malloc(s)</code>	在自由内存空间中分配 <code>s</code> 个字节, 不进行初始化, 若分配失败返回 0
<code>q=realloc(p,s)</code>	在自由内存空间中分配 <code>s</code> 个字节; <code>p</code> 必须是 <code>malloc()</code> 或 <code>calloc()</code> 返回的指针; 如果可能, 继续使用 <code>p</code> 指向的空间。如果不行, 将 <code>p</code> 指向的内存空间中的所有字节都拷贝到新空间中; 若分配失败返回 0
<code>free(p)</code>	释放 <code>p</code> 指向的内存空间; <code>p</code> 必须是 <code>malloc()</code> 、 <code>calloc()</code> 或 <code>realloc()</code> 返回的指针

注意, `malloc()` 等函数不调用构造函数, `free()` 也不调用析构函数。对具有构造函数和析构函数的类型, 不要使用这些函数。同样, 对具有构造函数的类型也不要使用 `memset()`。

以 `mem`<sup>®</sup> 开头的函数都定义于 `<cstring>`, 而分配函数都在 `<cstdlib>` 中定义。

另请参考 27.5.2 节。

**C.11.5 日期和时间**

`<ctime>` 中定义了一些日期和时间相关的类型和函数。

---

**日期和时间类型**


---

clock_t	算术类型，用于保存较短的时间间隔（可能只是几分钟的间隔）
time_t	算术类型，用于保存较长的时间间隔（可能几个世纪）
tm	结构类型，保存（自 1900 年以来的）日期和时间

---

struct tm 定义如下：

```
struct tm {
    int tm_sec; // 分钟内的秒值 [0,61]; 60 和 61 表示闰秒
    int tm_min; // 小时内的分钟值 [0,59]
    int tm_hour; // 天内的小时值 [0,23]
    int tm_mday; // 月内的天值 [1,31]
    int tm_mon; // 年内的月值 [0,11]; 0 表示一月（注意：不是 [1,12]）
    int tm_year; // 自 1900 年起的年值；0 表示 1900 年，102 表示 2002 年
    int tm_wday; // 自星期一起的星期内天值 [0,6]; 0 表示星期天
    int tm_yday; // 自一月一日起年内天值 [0,365]; 0 表示一月一日
    int tm_isdst; // 夏时制小时值
};
```

日期和时间函数如下：

```
clock_t clock(); // 自程序开始到当前的时钟周期数

time_t time(time_t* pt); // 当前日历时间
double difftime(time_t t2, time_t t1); // t2-t1 的秒数

tm* localtime(const time_t* pt); // *pt 对应的本地时间
tm* gmtime(const time_t* pt); // *pt 对应的格林威治标准时间 tm 结构，或是 0

time_t mktime(tm* ptm); // *ptm 对应的 time_t 或 time_t(-1)

char* asctime(const tm* ptm); // *ptm 对应的 C 风格字符串表示
char* ctime(const time_t t) { return asctime(localtime(t)); }
```

调用 asctime() 返回的结果可能是 “Sun Sep 16 01:03:52 1973\n” 这样的字符串。

一个名为 strftime() 的函数为 tm 提供了令人惊讶的格式化选项。你如果需要使用它，请查阅相关资料。

### C.11.6 其他函数

<cstdlib> 中定义了如下函数：

---

**其他 stdlib 函数**


---

abort()	“非正常”结束程序
exit(n)	结束程序，返回 n；n==0 表示程序运行成功
system(s)	将 C 风格字符串参数作为命令执行（具体行为依赖于系统）
qsort(b,n,s,cmp)	使用比较函数 cmp 排序 b 开始的 n 个元素，元素大小为 s
bsearch(k,b,n,s,cmp)	使用比较函数 cmp 在 b 开始的 n 个元素中搜索 k，元素大小为 s

---

qsort() 和 bsearch() 所使用的比较函数 (cmp) 必须是下面类型：

```
int (*cmp)(const void* p, const void* q);
```

即，排序函数并不了解类型信息，它将数组简单看作字节序列。cmp 返回值的含义为：

- 负数表示 \*p 小于 \*q。
- 0 表示 \*p 等于 \*q。
- 正数表示 \*p 大于 \*q。

注意，`exit()` 和 `abort()` 不调用析构函数。如果你希望对构造的自动对象和静态对象调用析构函数（见附录 A.4.2），应抛出一个异常。

C 标准库的更多内容，请参考 K&R 或其他有名的 C 语言参考资料。

## C.12 其他库

当你浏览标准库功能，毫无疑问可能找不到一些你需要的功能。与程序员面临的挑战以及世界上已有的众多库相比，C++ 是很渺小的。还存在很多其他用途的库：

- 图形用户界面库；
- 高级的数学库；
- 数据库访问库；
- 网络库；
- XML 库；
- 日期和时间库；
- 文件系统处理库；
- 三维图形库；
- 动画库；
- 其他用途的库。

但是，这些库不是标准库的一部分。你可以通过搜索互联网或者请教朋友和同事来寻找这些库。请不要形成这样一种观念：所有有用的库都是标准库的一部分。

# 安装 FLTK

如果代码和注释不一致，那么很可能是两者都错了。

——Norm Schryer

本附录说明了如何下载、安装 FLTK 图形和 GUI 工具包，以及如何将程序与之连接。

## D.1 介绍

我们选择 FLTK——快速轻量级工具包 (Fast Light Tool Kit, 发音为 “full tick”), 用来介绍图形和 GUI 的内容, 是因为它移植性好、相对简单、符合一般习惯、易于安装。我们将展示如何在微软 Visual Studio 下安装 FLTK, 因为这是我们的学生最常用的方式, 也是最难的。如果你使用其他一些系统 (像我们的一些学生那样), 在下载文件 (参见附录 D.3) 主文件夹 (目录) 中查找相应说明即可。

当使用的库不是 ISO C++ 标准库的一部分时, 你 (或者其他人) 必须下载、安装库, 并在自己的代码中正确使用它。这通常不是一个简单的工作, 安装 FLTK 可能是一个很好的练习——因为如果以前没有尝试过的话, 即使是下载、安装最简单的库也是比较困难的。不要不情愿再次向同一个人求教, 但要注意, 不要仅仅让他帮你完成, 而是要向他学习。

注意, 实际情况和本附录所描述的可能有细微差别。例如, 你使用的可能是 FLTK 的新版本, 或者你使用的 Visual Studio 与附录 D.4 介绍的不是同一个版本, 又或者你使用的是完全不同的 C++ 实现。

## D.2 下载 FLTK

在开始之前, 首先查看一下在你的计算机上是否已经安装了 FLTK, 参见附录 D.5。如果未安装, 首先要将所需安装文件下载到你的计算机上:

- 1) 进入网站 <http://fltk.org>。(紧急情况下, 可以从本书的支持网站下载: [www.stroustrup.com/Programming/FLTK](http://www.stroustrup.com/Programming/FLTK)。)
- 2) 在导航菜单点击 Download。
- 3) 在下拉菜单中选择 FLTK 1.1.x, 并点击 Show Download Locations。
- 4) 选择一个下载位置, 并下载 .zip 文件。

下载的文件是 .zip 格式。这种压缩格式适合于在网络上传输大量文件。你需要相应工具将其“解压”为普通文件。在 Windows 平台, 常用 WinZip 和 7-Zip 这两个工具。

## D.3 安装 FLTK

对于下面列出的安装步骤, 你可能遇到的主要问题是: 在我们写下并测试了这样的安装步骤后, 软件发生了变化 (这确实会发生); 其中的术语不合你的习惯 (抱歉, 这方面我们没

法帮你)。对于后一种情况，请朋友帮你翻译一下。

1) 解压下载的文件，并打开主文件夹 fltk-1.1.?. 在 Visual C++ 文件夹中 (如 vc2005 或 vnet)，打开 fltk.dsw。如果 Visual C++ 提问是否更新旧的项目文件，选择“全是”。

2) 在“生成”菜单中选择“生成解决方案”。这会花费几分钟。编译器将 FLTK 源码编译为静态链接库，这样，你在新项目中每次使用 FLTK 时就不必重新编译。当编译完成，将 Visual Studio 关闭。

3) 在 FLTK 主目录中打开 lib 文件夹。将除 README.lib 之外的所有 .lib 文件 (应该共七个) 拷贝 (不要拖拽) 到 C:\Program Files\Microsoft Visual Studio\Vc\lib。

4) 回到 FLTK 主目录，将 FL 文件夹拷贝到 C:\Program File\Microsoft Visual Studio\Vc\include。

专家会告诉你，有比拷贝文件到 C:\Program Files\Microsoft Visual Studio\Vc\lib 和 C:\Program Files\Microsoft Visual Studio\Vc\include 更好的安装方法。这是对的，但我们的目的不是使你成为 VS 专家。如果专家坚持应该这样做，请他们向你演示更好的方法。

## D.4 在 Visual Studio 中使用 FLTK

1) 在 Visual Studio 中创建一个新项目，与以往有一点不同：要创建一个“Win32 项目”而非“Win32 控制台应用程序”。确认创建一个“空项目”，否则“软件向导”就会向项目中添加你可能不需要或者不理解的内容。

2) 在 Visual Studio 中，选择“项目”菜单，在下拉菜单选择“属性”。

3) 在“属性”对话框中，在左侧的菜单中点击“链接器”文件夹。这会扩展出一个子菜单，在这个子菜单中，点击“输入”。在右侧的“附加依赖项”文本域中，输入：

```
fltkd.lib wsock32.lib comctl32.lib fltkjpegd.lib fltkimagesd.lib
```

(下面步骤不是必需的，因为不是缺省步骤。) 在忽略特定库文本域内，输入：

```
libcd.lib
```

4) (此步骤不是必需的，因为 /MDd 现在不是缺省选项。) 在“属性”窗口中左侧的菜单中，点击 C/C++ 扩展出一个不同的子菜单。点击“代码生成”子菜单项。在右侧菜单中，在“运行时库”下拉框中选择“多线程调试 DLL (/MDd)”。点击确定关闭“属性”窗口。

## D.5 测试是否工作正常

在一个新建的项目中创建一个新的 .cpp 文件，输入下面代码。正常情况下，这段代码应该会顺利编译通过。

```
#include <FL/Fl.h>
#include <FL/Fl_Box.h>
#include <FL/Fl_Window.h>

int main()
{
    Fl_Window window(200, 200, "Window title");
    Fl_Box box(0,0,200,200, "Hey, I mean, Hello, World!");
    window.show();
    return Fl::run();
}
```



如果这段代码不能正常工作：

- “未找到 .lib 文件的编译错误”：问题很可能出在安装环节。注意第 3 步，这一步将链接库 (.lib) 文件放置于编译器可以容易找到的地方。
- “不能打开 .h 文件的编译错误”：问题很可能出在安装环节。注意第 4 步，这一步将头 (.h) 文件放置于编译器容易找到的地方。
- “不能解析的外部符号链接错误”：问题很可能出在创建项目环节。

如果这些解决方法没有效果，请寻求朋友的帮助。

## GUI 实现

当你最终理解了你在做什么时，事情就会向正确方向发展了。

——Bill Fairbank

本附录介绍回调函数、Window、Widget 和 Vector\_ref 的实现细节。在第 21 章中，由于还没有介绍指针和类型转换的相关知识，我们无法给出 GUI 实现的完整描述，因此将这部分内容放到本附录中。

## E.1 回调函数实现

回调函数可实现如下：

```
void Simple_window::cb_next(Address, Address addr)
    // 对位于 addr 的窗口调用 Simple_window::next()
{
    reference_to<Simple_window>(addr).next();
}
```

如果你已经理解了第 12 章，很显然 Address 必须是 void\*。而且，当然，reference\_to<Simple\_window>(addr) 必须是由名为 addr 的 void\* 创建的 Simple\_window 的引用。但是，除非你以前有相关的程序设计经验，否则在阅读第 12 章之前，你不会有“很显然”或者“当然”的感觉。因此，让我们好好看一下地址使用的细节。

如附录 A.17 所述，C++ 提供了类型命名的功能。例如：

```
typedef void* Address;           // Address 是 void* 的别名
```

这意味着现在就可以用名字 Address 来代替 void\* 了。在此，我们用 Address 这个名字来强调传递了一个地址，并掩盖这样一个事实：void\* 是指向未知类型的对象的指针。

因此，cb\_next() 接受一个名为 addr 的 void\* 参数，并立即用某种方法将其转换为 Simple\_window&：

```
reference_to<Simple_window>(addr)
```

reference\_to 是一个模板函数（见附录 A.13）：

```
template<class W> W& reference_to(Address pw)
    // 将一个地址作为一个 W 的引用来处理
{
    return *static_cast<W*>(pw);
}
```

此处，我们将模板函数设计为类似一个类型转换——它将 void\* 转换为 Simple\_window&。类型转换操作 static\_cast 在 12.8 节中有详细描述。

编译器无法验证 addr 是否指向一个 Simple\_window 对象，但语言规则要求编译器此时信任程序员。幸运的是，我们是正确的。之所以确定我们是正确的，是因为 FLTK 会将我们传递给它的指针传递回来。由于我们知道传递给 FLTK 的指针是什么类型，因此可以使用

reference\_to 将其“取回来”。这种方法有些混乱，也未经检查，但在系统底层并不罕见。

一旦你拥有了一个 Simple\_window 的引用，就可以使用它来调用 Simple\_window 的成员函数。例如（见 21.3 节）：

```
void Simple_window::cb_next(Address, Address pw)
    // 对位于 addr 的窗口调用 Simple_window::next()
{
    reference_to<Simple_window>(pw).next();
}
```

我们简单地使用了一个混乱的回调函数 cb\_next() 来调整类型，以便调用一个普普通通的成员函数 next()。

## E.2 Widget 实现

Widget 接口类如下所示：

```
class Widget {
    // Widget 是一个 Fl_widget 的句柄——它 * 不是 * 一个 Fl_widget
    // 我们试图令我们的接口类与 FLTK 保持距离
public:
    Widget(Point xy, int w, int h, const string& s, Callback cb)
        :loc(xy), width(w), height(h), label(s), do_it(cb)
    {}

    virtual ~Widget() {}           // 析构函数

    virtual void move(int dx,int dy)
        { hide(); pw->position(loc.x+=dx, loc.y+=dy); show(); }
    virtual void hide() { pw->hide(); }
    virtual void show() { pw->show(); }

    virtual void attach(Window&) = 0; // 每个 Widget 至少为窗口
                                        // 定义了一种动作

    Point loc;
    int width;
    int height;
    string label;
    Callback do_it;

protected:
    Window* own;           // 每个 Widget 都属于一个 Window
    Fl_Widget* pw;        // 一个 Widget “了解” 其 Fl_Widget
};
```

注意，Widget 会跟踪 FLTK widget 及关联的 Window 对象。这需要使用指针，因为一个 Widget 在其生命期内可能与不同的 Window 对象相关联。引用或者命名对象是不能达到要求的（为什么？）。

每个 Widget 有一个位置（loc）、一个矩形形状（width 和 height）和一个标签（label）。有趣的地方是，它还有一个回调函数（do\_it），这个回调函数将屏幕上 Widget 的图像与代码连接起来。其他操作（move()、show()、hide() 和 attach()）的含义是显然的。

Widget 看起来还只是“半成品”。其设计目标是：如果就是作为一个实现类，用户不必经常查看其细节。它很适合于重新设计。我们对所有公有数据成员存有疑问，而那些“显

然”的操作通常需要针对意料之外的细微问题进行重新检查。

Widget 包含虚函数，可以作为基类，因此它有一个虚析构函数（见 12.5.2 节）。

### E.3 Window 实现

我们应该在什么时候使用指针，又应该在什么时候使用引用呢？在 8.5.6 节中，我们对这方面的一些一般性问题进行了讨论。在本附录中，我们只是注意到，一些程序员喜欢使用指针，而当需要在程序中在不同时刻引用不同对象时，应该使用指针。

到目前为止，我们还未展示图形和 GUI 库中的一个中心类——Window。最主要的原因是它使用了指针，而且其实现使用了基于自由内存空间分配的 FLTK。下面是 Window.h 中的 Window 类定义：

```
class Window : public Fl_Window {
public:
    // 令系统选择位置
    Window(int w, int h, const string& title);
    // 左上角在 xy 处
    Window(Point xy, int w, int h, const string& title);

    virtual ~Window() {}

    int x_max() const { return w; }
    int y_max() const { return h; }

    void resize(int ww, int hh) { w=ww, h=hh; size(ww,hh); }

    void set_label(const string& s) { label(s.c_str()); }

    void attach(Shape& s) { shapes.push_back(&s); }
    void attach(Widget&);

    void detach(Shape& s);           // 从形状中删除 w
    void detach(Widget& w);        // 从窗口中删除 w
                                    // (吊销回调函数)

    void put_on_top(Shape& p);     // 将 p 置于其他形状之上
protected:
    void draw();
private:
    vector<Shape*> shapes;         // 附属窗口的形状
    int w,h;                       // 窗口大小

    void init();
};
```

这样，当 attach() 一个 Shape 时，我们在 shapes 中保存了一个指针，Window 对象即可利用它来绘出图形。由于允许随后 detach() 这个形状，因此需要一个指针。本质上，被 attach() 的形状还是由用户代码拥有，我们只是传递给 Window 对象一个引用而已。Window::attach() 将其参数转换为一个指针，方便存储。如上所示，attach() 很简单，detach() 稍微复杂些。查看 Windows.cpp，我们发现：

```
void Window::detach(Shape& s)
    // 猜测最后附着的最先释放
{
```

```

    for (vector<Shape*>::size_type i = shapes.size(); 0<i; --i)
        if (shapes[i-1]==&s)
            shapes.erase(shapes.begin()+i-1);
}

```

成员函数 `erase()` 从 `vector` 中删除 (“擦除”) 一个值, 将 `vector` 的规模减小 1 (见 15.7.1 节)。

`Window` 的设计意图就是要作为一个基类来使用, 因此它有一个虚析构函数 (见 12.5.2 节)。

## E.4 Vector\_ref

本质上, `Vector_ref` 是模拟引用的 `vector`。你可以用引用或指针来初始化一个 `Vector_ref`:

- 如果将一个对象以引用方式传递给 `Vector_ref`, 则假定调用者会负责对象的生命期 (如对象是作用域之内的变量)。
- 如果对象以指针方式传递给 `Vector_ref`, 则假定它是用 `new` 分配的, 由 `Vector_ref` 负责释放空间。

元素以指针而不是对象副本的形式存入 `Vector_ref`, 其访问遵循引用语义。例如, 你可以将一个 `Circle` 放入 `Vector_ref<Shape>` 中, 而不会面临子数组问题:

```

template<class T> class Vector_ref {
    vector<T*> v;
    vector<T*> owned;
public:
    Vector_ref() {}
    Vector_ref(T* a, T* b = 0, T* c = 0, T* d = 0);

    ~Vector_ref() { for (int i=0; i<owned.size(); ++i) delete owned[i]; }

    void push_back(T& s) { v.push_back(&s); }
    void push_back(T* p) { v.push_back(p); owned.push_back(p); }
    T& operator[](int i) { return *v[i]; }
    const T& operator[](int i) const { return *v[i]; }

    int size() const { return v.size(); }
};

```

`Vector_ref` 的析构函数释放每个以指针传递来的对象。

## E.5 实例：操作 Widget

下面是一个完整示例程序, 它试验了很多 `Widget/Window` 特性。代码中只给出了很少的注释。不幸的是, 在现实中像这样缺乏必要注释的情况并不罕见。作为一个练习, 请将这个程序运行起来, 并给出必要的注释。

当你运行这个程序时, 它会定义四个按钮:

```

#include "../GUI.h"
using namespace Graph_lib;

class W7 : public Window {
    // 有四种方法可以令按钮向四处移动
    // 显示 / 隐藏、改变位置、创建一个新按钮以及附着 / 分离
public:

```

```

W7(int w, int h, const string& t);

Button* p1;      // 显示 / 隐藏
Button* p2;
bool sh_left;

Button* mvp;    // 移动
bool mv_left;

Button* cdp;    // 创建 / 销毁
bool cd_left;

Button* adp1;   // 激活 / 吊销
Button* adp2;
bool ad_left;

void sh();      // 动作
void mv();
void cd();
void ad();
static void cb_sh(Address, Address addr) // 回调函数
    { reference_to<W7>(addr).sh(); }
static void cb_mv(Address, Address addr)
    { reference_to<W7>(addr).mv(); }
static void cb_cd(Address, Address addr)
    { reference_to<W7>(addr).cd(); }
static void cb_ad(Address, Address addr)
    { reference_to<W7>(addr).ad(); }
};

```

但是，W7（7号 Window 实验）实际包含六个按钮，它将其中两个隐藏了起来：

```

W7::W7(int w, int h, const string& t)
:Window(w,h,t),
sh_left{true}, mv_left{true}, cd_left{true}, ad_left{true}
{
    p1 = new Button(Point{100,100},50,20,"show",cb_sh);
    p2 = new Button(Point{200,100},50,20,"hide",cb_sh);

    mvp = new Button(Point{100,200},50,20,"move",cb_mv);

    cdp = new Button(Point{100,300},50,20,"create",cb_cd);

    adp1 = new Button(Point{100,400},50,20,"activate",cb_ad);
    adp2 = new Button(Point{200,400},80,20,"deactivate",cb_ad);

    attach(*p1);
    attach(*p2);
    attach(*mvp);
    attach(*cdp);
    p2->hide();
    attach(*adp1);
}

```

程序中使用了四个回调函数。每个回调函数使你按下的按钮消失，并显示一个新的按钮。但是，这一效果是经过四个步骤来实现的：

```

void W7::sh() // 隐藏一个按钮，显示另外一个
{
    if (sh_left) {

```

```

        p1->hide();
        p2->show();
    }
    else {
        p1->show();
        p2->hide();
    }
    sh_left = !sh_left;
}

void W7::mv() // 移动按钮
{
    if (mv_left) {
       .mvp->move(100,0);
    }
    else {
       .mvp->move(-100,0);
    }
    mv_left = !mv_left;
}

void W7::cd() // 释放按钮并创建一个新的
{
   .cdp->hide();
    delete.cdp;
    string lab = "create";
    int x = 100;
    if (cd_left) {
        lab = "delete";
        x = 200;
    }
   .cdp = new Button(Point(x,300), 50, 20, lab, cb_cd);
    attach(*.cdp);
    cd_left = !cd_left;
}

void W7::ad() // 将按钮与窗口分离并将其替代者附着于窗口
{
    if (ad_left) {
        detach(*adp1);
        attach(*adp2);
    }
    else {
        detach(*adp2);
        attach(*adp1);
    }
    ad_left = !ad_left;
}

int main()
{
    W7 w(400,500,"move");
    return gui_main();
}

```

这个程序演示了在窗口中添加和去掉 Widget 或者是仅仅显示 Widget 的基本方法。

# 术 语 表

通常，精心挑选的几个词就胜过几千幅图。

——匿名

术语表 (glossary) 是正文中词汇的简单解释。本章是一个非常简短的术语表，列出了我们认为最重要的，特别是在学习编程初期尤为重要的术语。每章的“术语”一节也能帮助你查找术语。更完整的 C++ 相关术语表可在 [www.stroustrup.com/glossary.html](http://www.stroustrup.com/glossary.html) 找到，在互联网上还能找到非常多的专门的术语表 (质量也参差不齐)。请注意，一个术语可能有多个相关的含义 (因此我们偶尔可能会列出一些其他含义)，而我们列出的大多数术语都在其他场景下有相关的含义 (通常是弱相关的)；例如，我们定义抽象 (abstract) 一词时不会考虑它在现代绘画、法律事务或是哲学中的含义。

**abstract class (抽象类)** 不能直接用来创建对象的类；通常用来定义派生类的接口。如果一个类具有纯虚函数或保护的构造函数，则它就成为抽象类。

**abstraction (抽象)** 描述某实体选择性地、故意地忽略 (隐藏) 细节 (如实现细节)；选择性忽略。

**address (地址)** 一个值，用来在计算机内存中查找一个对象。

**algorithm (算法)** 求解问题的一个过程或一个公式；一个计算步骤的有限序列，生成一个结果。

**alias (别名)** 引用一个对象的一种替代方法；通常是一个名字、一个指针或一个引用。

**application (应用)** 一个程序或一组程序，被其用户看作单一实体。

**approximation (近似)** 接近最优或理想 (值或设计) 的东西 (如一个值或一个设计)。通常一个近似是理想结果的一个折中。

**argument (实参)** 传递给函数或模板的值，函数或模板通过参数访问它。

**array (数组)** 元素的同构序列，元素通常是编号的，如 [0:max)。

**assertion (断言)** 插入程序中的陈述，声明 (断言) 在此程序点上某事必须始终为真。

**base class (基类)** 作为类层次根基的类。通常

一个基类都会有一个或多个虚函数。

**bit (位)** 计算机中基本信息单元。一个位的值可以是 0 或 1。

**bug (程序漏洞)** 程序中的错误。

**byte (字节)** 大多数计算机中的基本寻址单元。一个字节通常包含 8 位。

**class (类)** 用户自定义类型，可以包含数据成员、函数成员和成员类型。

**code (代码)** 程序或程序的一部分；既可表示源代码也可表示目标代码，从这个角度来说有二义性。

**compiler (编译器)** 将源代码转换为二进制代码的程序。

**complexity (复杂性)** 描述问题求解方案构造难度或方案本身难度的概念和衡量标准，很难准确定义。有时复杂性 (简单) 表示为执行算法所需的操作次数估计。

**computation (计算)** 执行某段代码，通常接受一些输入并生成一些输出。

**concept (概念)** (1) 一种概念 (notion)、一种思想；(2) 一组要求，通常用于模板实参。

**concrete class (具体类)** 可创建对象的类。

**constant (常量)** (在给定作用域中) 不能改变的值；不可变。

**constructor (构造函数)** 初始化 (“构造”) 对象的操作。通常一个构造函数会建立一个不变



- 式，并且通常会获取对象要使用的资源（通常由析构函数释放这些资源）。
- container(容器)** 保存元素（其他对象）的对象。
- copy(拷贝)** 令两个对象具有相同值的操作。参见 **move(移动)**。
- correctness(正确性)** 若一个程序或一段程序满足其说明，则它是正确的。不幸的是，说明可能不完整或不一致，又或是无法满足用户的合理期望。因此，为了生成可接受的代码，我们有时不得不比形式化说明做得更多。
- cost(代价)** 生成一个程序或执行它的花费（如编程时间、运行时间或空间）。理想情况下，代价应该是复杂性的一个函数。
- data(数据)** 计算中用到的值。
- debugging(调试)** 在程序中查找并消除错误的工作；通常远没有测试那么系统化。
- declaration(声明)** 在程序中关于一个名字具有某个类型的说明。
- definition(定义)** 一个实体的声明，提供了所有必要信息，令程序可使用此实体。一种更简单的说法：分配了内存的声明。
- derived class(派生类)** 派生自一个或多个基类的类。
- design(设计)** 关于一个软件应该如何操作以满足其说明的总体描述。
- destructor(析构函数)** 当对象被销毁时（例如在作用域尾）被隐式调用的操作。它通常释放资源。
- encapsulation(封装)** 保护想要为私有性质的东西（如实现细节）不被未经授权用户访问。
- error(错误)** 对程序行为的合理期望（通常表达为要求或用户指南）与程序实际表现不吻合。
- executable(可执行程序)** 已准备好运行于计算机上的程序。
- feature creep(特性膨胀)** 向程序添加过多功能只是“以备万一”的倾向。
- file(文件)** 计算机中永久保存的信息的容器。
- floating-point number(浮点数)** 计算机对实数的一种近似，例如 7.93 和 10.73e-3。
- function(函数)** 一个命名的代码单元，可以从程序中其他部分调用它；计算的逻辑单元。
- generic programming(泛型编程)** 一种程序设计风格，关注算法的设计和高效实现。一个泛型算法能正确用于所有满足其要求的实参类型，在 C++ 中，泛型编程通常使用模板。
- handle(句柄)** 一个类，允许用户通过其成员指针或引用访问另一个类。参见 **copy**、**move**、**resource**。
- header(头文件)** 一个包含声明的文件，其中的声明用于程序中不同部分共享接口。
- hiding(隐藏)** 防止一部分信息被直接看到或直接访问的动作。例如，嵌套（内层）作用域中的名字可以防止来自外层（包围）作用域中的相同名字被直接使用。
- ideal(理想)** 我们所追求的完美版本。通常我们不得不做出折中、寻求近似版本。
- implementation(实现)** (1) 编写、测试代码的工作；(2) 实现程序的代码。
- infinite loop(无限循环)** 终止条件永不真的循环。参见 **iteration**。
- infinite recursion(无限递归)** 直到用于保存调用数据的机器内存都耗光也未结束的递归。这种递归实际上不会无限执行下去，而是会由于某种硬件错误而终止。
- information hiding(信息隐藏)** 分离接口和实现的活动，从而隐藏了不希望吸引用户注意的实现细节并提供了抽象。
- initialize(初始化)** 为对象赋予最初的（初始）值。
- input(输入)** 计算用到的值（如函数实参和从键盘敲入的字符）。
- integer(整数)** 一个整数，如 42 和 -99。
- interface(接口)** 一个声明或一组声明，指明了一段代码（如一个函数或一个类）如何被调用。
- invariant(不变式)** 在程序给定位置（可能有多个位置）必须始终为真的东西；通常用于描述一个对象的状态（一组值）或进入重复语句之前的循环的状态。
- iteration(迭代)** 重复执行一段代码的动作；参见 **recursion**。
- iterator(迭代器)** 标识序列中元素的对象。
- library(库)** 一组类型、函数、类等等，实现了一组特性（抽象），这些特性会被很多程序用作其一部分。

- lifetime (生命期)** 从对象的初始化时间一直到其不可用的时间 (离开作用域、被释放或程序终止)。
- linker (连接器)** 将目标代码文件和库组合为一个可执行文件的程序。
- literal (字面值常量)** 直接指出值的语法表示方式, 例如 12 指明整型值“十二”。
- loop (循环)** 反复执行的一段代码; 在 C++ 中通常是 for 语句或 while 语句。
- move (移动)** 将一个值从一个对象转移到另一个对象的操作, 原对象将获得表示“空”的值。参见 copy。
- mutable (可变的)** 可改变的; 与不可变、常量相对, 变量。
- object (对象)** (1) 已知类型、已初始化的一块内存, 保存该类型的一个值; (2) 一块内存。
- object code (目标代码)** 编译器的输出, 连接器的输入 (连接器用它生成可执行代码)。
- object file (目标文件)** 包含目标代码的文件。
- object-oriented programming (面向对象编程)** 一种程序设计风格, 关注类和类层次的设计和使用。
- operation (操作)** 用于执行某个动作, 例如函数和运算符。
- output (输出)** 计算生成的值 (如函数返回结果或写到显示屏的一行字符)。
- overflow (溢出)** 生成的值不能保存到目标中。
- overload (重载)** 定义两个同名的函数或操作, 但具有不同的参数 (运算对象) 类型。
- override (覆盖)** 在派生类中定义一个函数, 与基类中的虚函数具有相同的名字和参数类型, 因此可透过基类定义的接口被调用。
- owner (所有者)** 负责释放资源的对象。
- paradigm (范型)** 多少有些自命不凡的设计或编程风格术语; 常常与 (错误的) 暗示一起使用——存在一种优于所有其他风格的范型。
- parameter (参数)** 函数或模板显式输入的声明。当被调用时, 函数可以通过参数的名字访问传递来的实参。
- pointer (指针)** (1) 用于标识内存中带类型对象的值; (2) 保存这种值的变量。
- post-condition (后置条件)** 当退出一段代码 (如一个函数或一个循环) 时必须满足的条件。
- pre-condition (前置条件)** 当进入一段代码 (如一个函数或一个循环) 时必须满足的条件。
- program (程序)** 足够完整可被计算机执行的代码 (可能还有关联的数据)。
- programming (程序设计或编程)** 将问题求解方案表达为代码的艺术。
- programming language (程序设计语言)** 用于表达程序的语言。
- pseudo code (伪代码)** 计算的描述, 用非正式表示方式而非程序设计语言书写。
- pure virtual function (纯虚函数)** 必须在派生类中覆盖的虚函数。
- RAII (Resource Acquisition Is Initialization, 资源获取即初始化)** 一种基于作用域的资源管理基本技术。
- range (范围)** 可以用一个起始点和一个结束点描述的值的序列。例如, [0:5) 表示值 0、1、2、3 和 4。
- recursion (递归)** 一个函数调用自身的动作; 参见 iteration。
- reference (引用)** (1) 描述内存中一个带类型值的位置的值; (2) 保存这种值的变量。
- regular expression (正则表达式)** 字符串中模式的表示方式。
- requirement (要求)** (1) 程序或程序的一部分应具有行为的描述; (2) 函数或模板对其实参的假设的描述。
- resource (资源)** 需要获取并在随后释放的东西, 例如文件句柄、锁或者内存。参见 handle、owner。
- rounding (舍入)** 一个值转换为数学上最接近的一个值, 目标值的类型精确度更低。
- scope (作用域)** 程序文本 (代码) 的范围, 给定名字在其中可被引用。
- sequence (序列)** 可线性顺序访问的一些元素。
- software (软件)** 一组代码段及相关数据; 常常与 program 互换使用。
- source code (源代码)** 程序员生成的代码, (原则上) 对其他程序员是可读的。
- specification (说明)** 一段代码应该做什么的说明。

**standard (标准)** 官方认可的某种东西的定义, 例如一种程序设计语言。

**state (状态)** 一组值。

**string (字符串)** 字符序列。

**style (风格)** 一组程序设计技术, 对语言特性的使用是一致的; 有时会用于非常局限的意义——仅仅指代一些代码命名和外观的底层规则。

**subtype (子类型)** 派生类型; 一个类型, 具有另一个类型的所有属性, 可能还更多。

**supertype (超类型)** 基类型; 一个类型, 具有另一个类型的属性的子集。

**system (系统)** (1) 一个程序或一组程序, 在计算机上执行某个任务; (2) “操作系统”的简称, 即, 计算机的基本执行环境和工具。

**template (模板)** 一个类或一个函数, 用一个或多个类型或(编译时)值进行参数化; 支持泛型编程的基本 C++ 语言结构。

**testing (测试)** 对程序错误的系统化查找。

**trade-off (折中)** 平衡多个设计和实现标准的结果。

**truncation (截断)** 从一个类型转换到另一个类型所产生的信息损失, 原因是目标类型不能准确表达要转换的值。

**type (类型)** 定义了对象的一组可能取值及其一组操作的东西。

**uninitialized (未初始化)** 对象在初始化之前的(未定义的)状态。

**unit (1) (单位)** 标准度量, 给值以含义(如表示距离的千米); (2) (单元) 一个完整东西的与众不同的(如命名的)部分。

**use case (用例)** 程序的特殊的(通常是简单的)使用, 用于测试其功能、展示其目的。

**value (值)** 内存中的一组位, 根据类型解释其含义。

**variable (变量)** 给定类型的命名对象; 除非未初始化, 否则包含一个值。

**virtual function (虚函数)** 可在派生类中覆盖的成员函数。

**word (字)** 计算机内存的基本单元, 通常是用于保存一个整数的单元。

## 参考文献

- Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition* (usually called “The Dragon Book”). Addison-Wesley, 2006. ISBN 0321486811.
- Andrews, Mike, and James A. Whittaker. *How to Break Software: Functional and Security Testing of Web Applications and Web Services*. Addison-Wesley, 2006. ISBN 0321369440.
- Bergin, Thomas J., and Richard G. Gibson, eds. *History of Programming Languages, Volume 2*. Addison-Wesley, 1996. ISBN 0201895021.
- Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall, 2006. ISBN 0131872494.
- Boost.org. “A Repository for Libraries Meant to Work Well with the C++ Standard Library.” [www.boost.org](http://www.boost.org).
- Cox, Russ. “Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, . . .).” <http://swtch.com/~rsc/regexp/regexpl.html>.
- dmoz.org. <http://dmoz.org/Computers/Programming/Languages>.
- Freeman, T. L., and C. Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992. ISBN 0136515975.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0201633612.
- Goldthwaite, Lois, ed. *Technical Report on C++ Performance*. ISO/IEC PDTR 18015. [www.stroustrup.com/performanceTR.pdf](http://www.stroustrup.com/performanceTR.pdf).
- Gullberg, Jan. *Mathematics – From the Birth of Numbers*. W. W. Norton, 1996. ISBN 039304002X.
- Hailpern, Brent, and Barbara G. Ryder, eds. *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*. San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.
- ISO/IEC 9899:2011. *Programming Languages – C*. The C standard.
- ISO/IEC 14882:2011. *Programming Languages – C++*. The C++ standard.
- Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988. ISBN 0131103628.
- Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1997. ISBN 0201896842.
- Koenig, Andrew, and Barbara E. Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 020170353X.
- Langer, Angelika, and Klaus Krefl. *Standard C++ IOStreams and Locales: Advanced Programmer’s Guide and Reference*. Addison-Wesley, 2000. ISBN 0321585585.
- Lippman, Stanley B., José Lajoie, and Barbara E. Moo. *The C++ Primer, Fifth Edition*. Addison-Wesley, 2005. ISBN 0321714113. (Use only the 5th edition.)
- Lockheed Martin Corporation. “Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program.” Document Number 2RDU0001 Rev C. December 2005. Colloquially known as “JSF++.” [www.stroustrup.com/JSF-AV-rules.pdf](http://www.stroustrup.com/JSF-AV-rules.pdf).
- Lohr, Steve. *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts – The Programmers Who Created the Software Revolution*. Basic Books, 2002. ISBN 978-0465042265.
- Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley, 2001. ISBN 0201749629.
- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*. Addison-Wesley, 2005. ISBN 0321334876.
- Programming Research. *High-Integrity C++ Coding Standard Manual Version 2.4*. [www](http://www).

- programmingresearch.com.
- Richards, Martin. *BCPL – The Language and Its Compiler*. Cambridge University Press, 1980. ISBN 0521219655.
- Ritchie, Dennis. “The Development of the C Programming Language.” *Proceedings of the ACM History of Programming Languages Conference (HOPL-2)*. *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
- Salus, Peter H. *A Quarter Century of UNIX*. Addison-Wesley, 1994. ISBN 0201547775.
- Sammet, Jean E. *Programming Languages: History and Fundamentals*. Prentice Hall, 1969. ISBN 0137299885.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2002. ISBN 0201604647.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2003. ISBN 0201795256.
- Schwartz, Randal L., Tom Phoenix, and Brian D. Foy. *Learning Perl, Fourth Edition*. O’Reilly, 2005. ISBN 0596101058.
- Scott, Michael L. *Programming Language Pragmatics*. Morgan Kaufmann, 2000. ISBN 1558604421.
- Sebesta, Robert W. *Concepts of Programming Languages, Sixth Edition*. Addison-Wesley, 2003. ISBN 0321193628.
- Shepherd, Simon. “The Tiny Encryption Algorithm (TEA).” [www.taloredge.com/reference/Mathematics/TEA-XTEA.pdf](http://www.taloredge.com/reference/Mathematics/TEA-XTEA.pdf) and <http://143.53.36.235:8080/tea.htm>.
- Stepanov, Alexander. [www.stepanovpapers.com](http://www.stepanovpapers.com).
- Stewart, G. W. *Matrix Algorithms, Volume I: Basic Decompositions*. SIAM, 1998. ISBN 0898714141.
- Stone, Debbie, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. *User Interface Design and Evaluation*. Morgan Kaufmann, 2005. ISBN 0120884364.
- Stroustrup, Bjarne. “A History of C++: 1979–1991.” *Proceedings of the ACM History of Programming Languages Conference (HOPL-2)*. *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
- Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
- Stroustrup, Bjarne. “Learning Standard C++ as a New Language.” *C/C++ Users Journal*, May 1999.
- Stroustrup, Bjarne. “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, Aug., and Sept. 2002.
- Stroustrup, Bjarne. “Evolving a Language in and for the Real World: C++ 1991–2006.” *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*. San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.
- Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 0321563840.
- Stroustrup, Bjarne. *A Tour of C++*. Addison-Wesley, 2013. ISBN 978-0321958310.
- Stroustrup, Bjarne. Author’s home page, [www.stroustrup.com](http://www.stroustrup.com).
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 2000. ISBN 0201615622.
- Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, 2004. ISBN 0321113586.
- University of St. Andrews. The MacTutor History of Mathematics archive. <http://www-gap.dcs.st-and.ac.uk/~history>.
- Wexelblat, Richard L., ed. *History of Programming Languages*. Academic Press, 1981. ISBN 0127450408.
- Whittaker, James A. *How to Break Software: A Practical Guide to Testing*. Addison-Wesley, 2002. ISBN 0201796198.
- Wood, Alastair. *Introduction to Numerical Analysis*. Addison-Wesley, 2000. ISBN 020134291X.