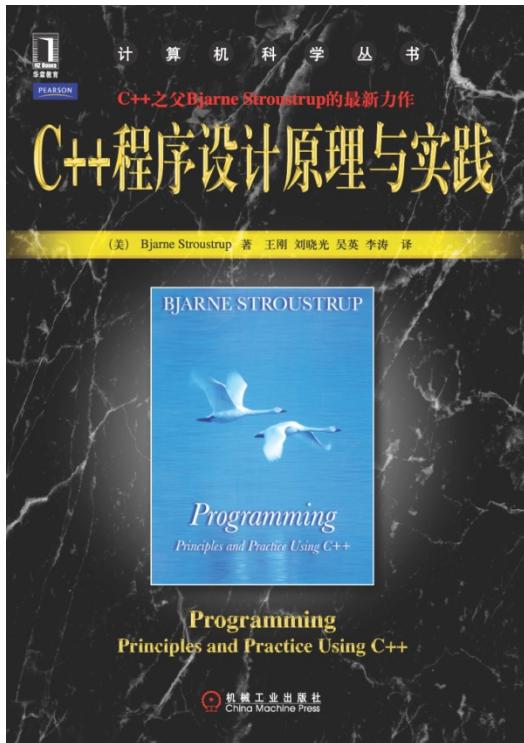


# 《C++程序设计原理与实践》迷你书

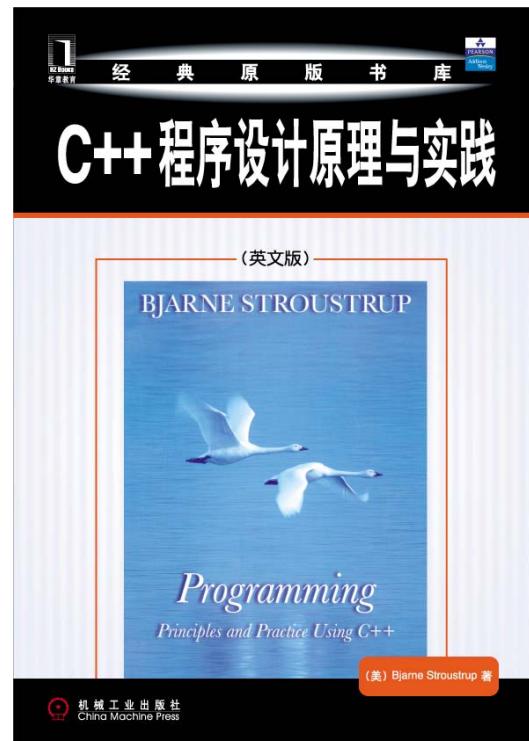


**ISBN: 78-7-111-30322-0**

**定价: 108.00**

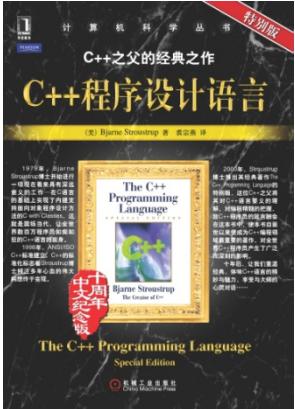
**ISBN: 978-7-111-28248-8**

**定价: 89.00**



机械工业出版社华章公司

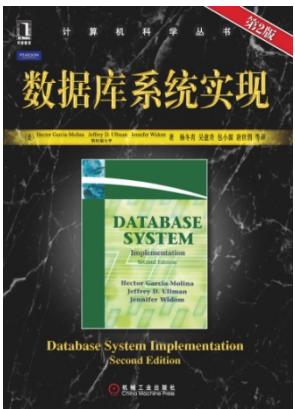
## 插播广告



本书是在 C++ 语言和程序设计领域具有深远影响、畅销不衰的著作，由 C++ 语言之父 Bjarne Stroustrup 撰写，对 C++ 语言进行了最全面、最权威的论述，覆盖标准 C++ 以及由 C++ 所支持的关键性编程技术和设计技术。

本书英文原版一经面世，即引起业内人士的高度评价和热烈欢迎，先后被翻译成德、希、匈、西、荷、法、日、俄、中、韩等近 20 种语言，数以百万计的程序员从中获益，是拥有最多读者、使用最广泛的 C++ 著作。

ISBN: 978-7-111-29885-4

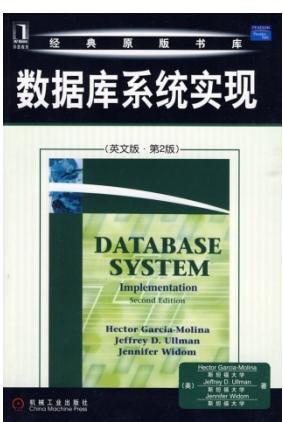


本书是关于数据库系统实现方面内容最为全面的著作之一。

书中从数据库实现者的角度对数据库系统实现原理进行了深入阐述，并具体讨论了数据库管理系统的三个主要成分——存储管理器、查询处理器和事务管理器的实现技术。

本书内容深入且全面，技术实用且先进，叙述深入浅出，是一本难得的高层次的教材，非常适合于作为高等院校计算机专业研究生的教材或本科生的教学参考书，同时也适合作为从事相关研究或开发工作的专业技术人员的高级参考资料。

ISBN: 978-7-111-30287-2



本书是美国斯坦福大学计算机科学专业数据库系列课程第二门课程的指定教材。

本书内容深入且全面，技术实用且先进，叙述深入浅出，是一本难得的高层次的教材，非常适合于作为高等院校计算机专业研究生的教材或本科生的教学参考书，同时也适合作为从事相关研究或开发工作的专业技术人员的高级参考资料。

ISBN: 978-7-111-09161-2

打扰了，请继续……

计算机科学丛书

# C++ 程序设计原理与实践

**Programming: Principles and Practice Using C++**

(美) Bjarne Stroustrup 著

王刚 刘晓光 吴英 李涛 译



机械工业出版社  
China Machine Press

本书是 C++ 之父 Bjarne Stroustrup 的最新力作。书中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。

本书语言通俗易懂、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。

Simplified Chinese edition copyright © 2010 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Programming: Principles and Practice Using C++* (ISBN 978-0-321-54372-1) by Bjarne Stroustrup, Copyright © 2009.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

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

**封底无防伪标均为盗版**

**版权所有，侵权必究**

**本书法律顾问 北京市展达律师事务所**

**本书版权登记号：图字：01-2009-1608**

### **图书在版编目(CIP)数据**

C++ 程序设计原理与实践/(美)斯特劳斯特鲁普(Stroustrup, B.)著；王刚等译. —北京：机械工业出版社，2010.6

(计算机科学丛书)

书名原文：Programming: Principles and Practice Using C++

ISBN 978-7-111-30322-0

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

中国版本图书馆 CIP 数据核字 (2010) 第 061970 号

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：李俊竹

印刷

2010 年 6 月第 1 版第 1 次印刷

184mm × 260mm · 41.75 印张

标准书号：ISBN 978-7-111-30322-0

定价：108.00 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

购书热线：(010) 68326294; 88379649; 68995259

投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com

# 译者序

程序设计是打开计算机世界大门的金钥匙，它使五彩斑斓的软件对你来说不再是“魔术”。C++ 语言则是学习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。本书适合那些从未有过程序设计经验的初学者，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。你所学到的思想，大多数也都可直接用于其他程序设计语言。本书不是初学程序设计语言的简单入门教材，它的目标是能让读者学到基本的实用程序设计技术，因此也可以作为程序设计方面的“第二本书”。基于这样一个目标，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，除了基本的 C++ 程序特性之外，本书还介绍了大量的求解实际问题的程序设计技术：如语法分析器程序的设计、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，为读者指引了明确的方向，教会读者如何才能达到编写有用的、优美的程序这一最终目标。

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。他现在是德州农工大学计算机科学首席教授，美国国家工程院的会员和 AT&T 院士。在进入学术界之前，他在 AT&T 贝尔实验室工作多年。他是 ISO C++ 标准委员会的创始人之一。本书是他在 C++ 程序设计领域奉献给广大读者的又一经典著作。

本书分为五个部分。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、类型和值，运算，错误处理，函数，类等内容，以及一个计算器程序实例。第二部分介绍输入和输出，首先介绍了输入/输出流的基本概念和格式化输出方法，然后第 12~16 章重点介绍了图形/GUI 类和图形化程序设计。第三部分介绍数据结构和算法，重点介绍了向量、自由内存空间、数组、模板和异常、容器和迭代器以及算法和映射。第四部分希望拓宽读者的视野，介绍了程序设计语言理念和历史、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。第五部分为附录，包括 C++ 语言概要、标准库概要、Visual Studio 简要入门、FLTK 安装以及 GUI 实现等内容。

本书的序、第 0 章、8~11 章、23~25 章、27 章、附录、术语表由王刚翻译，第 4、5、22、26 章由刘晓光翻译，第 1~3 章、17~21 章由吴英翻译，第 6~7 章、12~16 章由李涛翻译。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。

译者

2010 年 4 月于南开大学

# 前 言

“该死的鱼雷！全速前进。”

——海军上将 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的初学者，它能帮助你理解使用 C++ 语言进行程序设计的基本原理并获得实践技巧。我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，共 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，并通过从实际应用中获取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——也就是让计算机按你的期望工作的能力，会逐渐稳步地提高。我从不会说：“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？因为计算机文化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，如果认真去做的话，它会是一种非常有用的智力训练，可以锻炼我们的思考能力。然而，由于计算机能做出反馈，程序设计又不像大多数数学形式那么抽象，因而对更多人来说更容易接受。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，程序设计非常有趣。

为什么学习 C++ 这门程序设计语言呢？学习程序设计不可能不借助一门程序设计语言，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和免费的 C++ 实现。而且，你用 C++ 所学到的程序设计思想，大多数都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：能让你学到基本的实用编程技术的最简单的书籍。这是一个雄心勃勃的目标，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。

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、较高质量地完成这个工作；也就是说，我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。请考虑是否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，只是因为你的需求与我所描述的部分软件之间不匹配。信息技术世界中还有很多部分是不要求程序设计知识的，那些领域可能适合你。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，它也可以作为程序设计方面的第二本书，适合那些已经了解一点 C++ 的人，和那些会用其他语言编程，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，多做练习会帮助你解决这个问题。如果你曾经按某种更为传统的方式学习过 C++，那么在进行到第 7 章之前，你会发现一些令你惊奇的和有用的内容。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能是相似的。你不可能仅仅通过读书就学会游泳、演奏乐器或者开车，你必须进行实践。同样，不读程序、不写程序就不可能学会程序设计。本书给出了大量代码实例，都配合有说明文字和图表。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，仅仅读代码是不能学会编程实践技巧的。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，程序设计远非只是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。但本书不能取代对这些领域全面、均衡的学习。

代码可以很有用，同样也可以很优美。本书会帮你了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。祝你学习顺利！

## 致学生

到目前为止，我在德州农工大学已经用本书的初稿教过 1000 名以上的大一新生，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来学习本书，我认为本书会广泛用于自学。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你的思想，这正是测试你对问题理解和确认你的记忆的最有效的方法。你没有必要独自解决所有编程语言和编程环境中的难题。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（不是如果）你产生这样的疑问时，休息一会儿，重新阅读这篇前言，阅读一下第 1 章（“计算机、人和程序设计”）和第 22 章（“思想和历史”）。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，以及为什么我会认为程序设计是能为世界带来积极贡献的重要工具。如果你对我的教学理念和一般方法有疑问，请阅读第 0 章（“致读者”）。

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

本书不是一门传统的计算机科学的 101 课程，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时代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，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。本书试图只做好一件事，计算机科学不是一门课程可以囊括的。如果本书（本课程）被计算机科学、计算机工程、电子工程（很多我们最早的学生都是电子专业的）、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。

请阅读第 0 章，那里有对我的教学理念、一般教学方法等的介绍。请在教学过程中尝试将这

些观点传达给你的学生。

## 资源

本书网站的网址为 [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推移不断改进，但对于初学者，现在可以找到下面一些资料：

- 基于本书的讲义的幻灯片。
- 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。
- 可能有用的一些链接。
- 勘误表。

欢迎随时提出对这些资料的改进意见。

## 致谢

我要特别感谢我已故的同事和联合导师 Lawrence “Pete” Peterson，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，并提供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“Pete 和我”。

我要感谢那些直接或间接帮助过我撰写本书的学生、助教以及德州农工大学讲授 ENGR 112 课程的教师，以及 Walter Daugherty，他曾讲授过这门课程。还要感谢 Damian Dechev、Tracy Hammond、Arne Tolstrup Madsen、Gabriel Dos Reis、Nicholas Stroustrup、J. C. van Winkel、Greg Versoender、Ronnie Ward 和 Leor Zolman，他们对本书初稿提出了一些建设性意见。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，他们帮助我在夏天躲开那些分心的事来完成本书。

感谢 Addison-Wesley 公司为我安排的审阅人：Richard Enbody、David Gustafson、Ron McCarty 和 K. Narayanaswamy，他们基于自身讲授 C++ 课程或者大学计算机科学系 101 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他极大的耐心。我非常感谢 Addison-Wesley 公司为本书组织的制作团队的同仁，他们为本书的高质量出版做出了很多贡献，他们是：Julie Grady（校对）、Chris Keane（排版）、Rob Mauhar（插图）、Julie Nahil（制作编辑）和 Barbara Wood（文字编辑）。

另外，我本人对本书代码的检查很不系统，Bashar Anabtawi、Yinan Fan 和 Yuriy Solodkyy 使用微软 C++ 7.1 版(2003)和 8.0 版(2005)以及 GCC 3.4.4 版检查了所有代码片段。

我还要感谢 Brian Kernighan 和 Doug McIlroy 为程序设计类书籍的撰写定下了非常高的标准，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。

# 目 录

出版者的话

译者序

前言

第 0 章 致读者 ..... 1

  0.1 本书结构 ..... 1

    0.1.1 一般方法 ..... 2

    0.1.2 简单练习、习题等 ..... 2

    0.1.3 进阶学习 ..... 3

  0.2 讲授和学习本书的方法 ..... 4

    0.2.1 本书内容顺序的安排 ..... 6

    0.2.2 程序设计和程序设计语言 ..... 7

    0.2.3 可移植性 ..... 7

  0.3 程序设计和计算机科学 ..... 8

  0.4 创造性和问题求解 ..... 8

  0.5 反馈方法 ..... 8

  0.6 参考文献 ..... 8

  0.7 作者简介 ..... 9

第 1 章 计算机、人与程序设计 ..... 11

  1.1 介绍 ..... 11

  1.2 软件 ..... 11

  1.3 人 ..... 13

  1.4 计算机科学 ..... 15

  1.5 计算机已无处不在 ..... 15

    1.5.1 有屏幕和没有屏幕 ..... 16

    1.5.2 船舶 ..... 16

    1.5.3 电信 ..... 17

    1.5.4 医疗 ..... 18

    1.5.5 信息领域 ..... 19

    1.5.6 一种垂直的视角 ..... 20

    1.5.7 与 C++ 程序设计有何联系 ..... 21

  1.6 程序员的理想 ..... 21

## 第一部分 基本知识

第 2 章 Hello, World!	25
2.1 程序	25
2.2 经典的第一个程序	26
2.3 编译	27
2.4 链接	29
2.5 编程环境	30
第 3 章 对象、类型和值	34
3.1 输入	34
3.2 变量	35
3.3 输入和类型	36
3.4 运算和运算符	37
3.5 赋值和初始化	39
3.5.1 实例：删除重复单词	41
3.6 组合赋值运算符	42
3.6.1 实例：重复单词统计	42
3.7 命名	43
3.8 类型和对象	44
3.9 类型安全	45
3.9.1 安全类型转换	46
3.9.2 不安全类型转换	46
第 4 章 计算	51
4.1 计算	51
4.2 目标和工具	52
4.3 表达式	53
4.3.1 常量表达式	54
4.3.2 运算符	55
4.3.3 类型转换	56
4.4 语句	56
4.4.1 选择语句	57
4.4.2 循环语句	61
4.5 函数	64

4.5.1 使用函数的原因	65	6.3.3 实现单词	106
4.5.2 函数声明	66	6.3.4 使用单词	107
4.6 向量	67	6.3.5 重新开始	108
4.6.1 向量空间增长	67	6.4 文法	109
4.6.2 一个数值计算的例子	68	6.4.1 英文文法	112
4.6.3 一个文本处理的例子	70	6.4.2 设计一个文法	113
4.7 语言特性	71	6.5 将文法转换为程序	114
第5章 错误	75	6.5.1 实现文法规则	114
5.1 介绍	75	6.5.2 表达式	115
5.2 错误的来源	76	6.5.3 项	117
5.3 编译时错误	77	6.5.4 基本表达式	118
5.3.1 语法错误	77	6.6 试验第一个版本	119
5.3.2 类型错误	77	6.7 试验第二个版本	122
5.3.3 警告	78	6.8 单词流	123
5.4 连接时错误	78	6.8.1 实现 Token_stream	124
5.5 运行时错误	79	6.8.2 读单词	125
5.5.1 调用者处理错误	80	6.8.3 读数值	126
5.5.2 被调用者处理错误	81	6.9 程序结构	127
5.5.3 报告错误	82	第7章 完成一个程序	131
5.6 异常	83	7.1 介绍	131
5.6.1 错误参数	83	7.2 输入和输出	131
5.6.2 范围错误	84	7.3 错误处理	133
5.6.3 输入错误	85	7.4 处理负数	135
5.6.4 截断错误	87	7.5 模运算: %	136
5.7 逻辑错误	88	7.6 清理代码	138
5.8 估计	89	7.6.1 符号常量	138
5.9 调试	90	7.6.2 使用函数	139
5.9.1 实用调试技术	91	7.6.3 代码格式	140
5.10 前置条件和后置条件	94	7.6.4 注释	141
5.10.1 后置条件	95	7.7 错误恢复	143
5.11 测试	96	7.8 变量	145
第6章 编写一个程序	100	7.8.1 变量和定义	145
6.1 一个问题	100	7.8.2 引入单词 name	148
6.2 对问题的思考	100	7.8.3 预定义名字	150
6.2.1 程序设计的几个阶段	101	7.8.4 我们到达目的地了吗	150
6.2.2 策略	101	第8章 函数相关的技术细节	153
6.3 回到计算器问题	102	8.1 技术细节	153
6.3.1 第一步尝试	103	8.2 声明和定义	154
6.3.2 单词	104	8.2.1 声明的类别	156

8.2.2 变量和常量声明 .....	157
8.2.3 默认初始化 .....	158
8.3 头文件 .....	158
8.4 作用域 .....	160
8.5 函数调用和返回 .....	163
8.5.1 声明参数和返回类型 .....	163
8.5.2 返回一个值 .....	164
8.5.3 传值参数 .....	165
8.5.4 传常量引用参数 .....	166
8.5.5 传引用参数 .....	168
8.5.6 传值与传引用的对比 .....	169
8.5.7 参数检查和转换 .....	171
8.5.8 实现函数调用 .....	172
8.6 求值顺序 .....	175
8.6.1 表达式求值 .....	176
8.6.2 全局初始化 .....	176
8.7 名字空间 .....	177
8.7.1 using 声明和 using 指令 .....	178
第 9 章 类相关的技术细节 .....	183
9.1 用户自定义类型 .....	183
9.2 类和成员 .....	184
9.3 接口和实现 .....	184
9.4 演化一个类 .....	185
9.4.1 结构和函数 .....	185
9.4.2 成员函数和构造函数 .....	187
9.4.3 保持细节私有性 .....	188
9.4.4 定义成员函数 .....	189
9.4.5 引用当前对象 .....	191
9.4.6 报告错误 .....	191
9.5 枚举类型 .....	192
9.6 运算符重载 .....	193
9.7 类接口 .....	195
9.7.1 参数类型 .....	195
9.7.2 拷贝 .....	197
9.7.3 默认构造函数 .....	197
9.7.4 const 成员函数 .....	199
9.7.5 类成员和“辅助函数” .....	200
9.8 Date 类 .....	201

## 第二部分 输入和输出

第 10 章 输入/输出流 .....	207
10.1 输入和输出 .....	207
10.2 I/O 流模型 .....	208
10.3 文件 .....	209
10.4 打开文件 .....	210
10.5 读写文件 .....	211
10.6 I/O 错误处理 .....	213
10.7 读取单个值 .....	215
10.7.1 将程序分解为易管理的子模块 .....	216
10.7.2 将人机对话从函数中分离 .....	218
10.8 用户自定义输出操作符 .....	219
10.9 用户自定义输入操作符 .....	220
10.10 一个标准的输入循环 .....	220
10.11 读取结构化的文件 .....	222
10.11.1 内存表示 .....	222
10.11.2 读取结构化的值 .....	224
10.11.3 改变表示方法 .....	226
第 11 章 定制输入/输出 .....	230
11.1 有规律的和无规律的输入和输出 .....	230
11.2 格式化输出 .....	230
11.2.1 输出整数 .....	231
11.2.2 输入整数 .....	232
11.2.3 输出浮点数 .....	232
11.2.4 精度 .....	233
11.2.5 域 .....	234
11.3 文件打开和定位 .....	235
11.3.1 文件打开模式 .....	235
11.3.2 二进制文件 .....	236
11.3.3 在文件中定位 .....	238
11.4 字符串流 .....	238
11.5 面向行的输入 .....	239
11.6 字符分类 .....	240
11.7 使用非标准分隔符 .....	241
11.8 还有很多未讨论的内容 .....	246
第 12 章 一个显示模型 .....	249
12.1 为什么要使用图形用户界面 .....	249

12.2 一个显示模型 .....	250	14.1.2 操作 .....	289
12.3 第一个例子 .....	250	14.1.3 命名 .....	290
12.4 使用 GUI 库 .....	252	14.1.4 可变性 .....	291
12.5 坐标系 .....	253	14.2 Shape 类 .....	291
12.6 形状 .....	253	14.2.1 一个抽象类 .....	292
12.7 使用形状类 .....	254	14.2.2 访问控制 .....	293
12.7.1 图形头文件和主函数 .....	254	14.2.3 绘制形状 .....	295
12.7.2 一个几乎空白的窗口 .....	255	14.2.4 拷贝和可变性 .....	297
12.7.3 坐标轴 .....	256	14.3 基类和派生类 .....	298
12.7.4 绘制函数图 .....	257	14.3.1 对象布局 .....	299
12.7.5 Polygon .....	257	14.3.2 类的派生和虚函数定义 .....	300
12.7.6 Rectangle .....	258	14.3.3 覆盖 .....	301
12.7.7 填充 .....	259	14.3.4 访问 .....	302
12.7.8 文本 .....	259	14.3.5 纯虚函数 .....	302
12.7.9 图片 .....	259	14.4 面向对象程序设计的好处 .....	303
12.7.10 还有很多未讨论的内容 .....	260	第 15 章 绘制函数图和数据图 .....	307
12.8 让图形程序运行起来 .....	261	15.1 介绍 .....	307
12.8.1 源文件 .....	261	15.2 绘制简单函数图 .....	307
第 13 章 图形类 .....	264	15.3 Function 类 .....	309
13.1 图形类概览 .....	264	15.3.1 默认参数 .....	310
13.2 Point 和 Line .....	265	15.3.2 更多的例子 .....	311
13.3 Lines .....	267	15.4 Axis 类 .....	311
13.4 Color .....	268	15.5 近似 .....	313
13.5 Line_style .....	270	15.6 绘制数据图 .....	316
13.6 Open_polyline .....	271	15.6.1 读取文件 .....	317
13.7 Closed_polyline .....	271	15.6.2 一般布局 .....	318
13.8 Polygon .....	272	15.6.3 数据比例 .....	319
13.9 Rectangle .....	273	15.6.4 构造数据图 .....	319
13.10 管理未命名对象 .....	276	第 16 章 图形用户界面 .....	324
13.11 Text .....	277	16.1 用户界面的选择 .....	324
13.12 Circle .....	278	16.2 “Next”按钮 .....	325
13.13 Ellipse .....	279	16.3 一个简单的窗口 .....	325
13.14 Marked_polyline .....	280	16.3.1 回调函数 .....	327
13.15 Marks .....	281	16.3.2 等待循环 .....	328
13.16 Mark .....	282	16.4 Button 和其他 Widget .....	329
13.17 Image .....	283	16.4.1 Widget .....	329
第 14 章 设计图形类 .....	288	16.4.2 Button .....	330
14.1 设计原则 .....	288	16.4.3 In_box 和 Out_box .....	330
14.1.1 类型 .....	288	16.4.4 Menu .....	331

16.5	一个实例	332
16.6	控制流的反转	334
16.7	添加菜单	335
16.8	调试 GUI 代码	338

### 第三部分 数据结构和算法

第 17 章	向量和自由空间	343
17.1	介绍	343
17.2	向量的基本知识	344
17.3	内存、地址和指针	345
17.3.1	运算符 sizeof	347
17.4	自由空间和指针	347
17.4.1	自由空间分配	348
17.4.2	通过指针访问数据	349
17.4.3	指针范围	349
17.4.4	初始化	350
17.4.5	空指针	351
17.4.6	自由空间释放	351
17.5	析构函数	353
17.5.1	生成的析构函数	354
17.5.2	析构函数和自由空间	355
17.6	访问向量元素	356
17.7	指向类对象的指针	356
17.8	类型混用：无类型指针和指针 类型转换	357
17.9	指针和引用	359
17.9.1	指针参数和引用参数	359
17.9.2	指针、引用和继承	360
17.9.3	实例：列表	360
17.9.4	列表的操作	362
17.9.5	列表的使用	363
17.10	this 指针	364
17.10.1	关于 Link 使用的更多讨论	365
第 18 章	向量和数组	369
18.1	介绍	369
18.2	拷贝	369
18.2.1	拷贝构造函数	370
18.2.2	拷贝赋值	372
18.2.3	拷贝术语	373

18.3	必要的操作	374
18.3.1	显示构造函数	375
18.3.2	调试构造函数与析构函数	376
18.4	访问向量元素	377
18.4.1	对 const 对象重载运算符	378
18.5	数组	379
18.5.1	指向数组元素的指针	380
18.5.2	指针和数组	381
18.5.3	数组初始化	383
18.5.4	指针问题	383
18.6	实例：回文	385
18.6.1	使用 string 实现回文	386
18.6.2	使用数组实现回文	386
18.6.3	使用指针实现回文	387
第 19 章	向量、模板和异常	391
19.1	问题	391
19.2	改变向量大小	393
19.2.1	方法描述	393
19.2.2	reserve 和 capacity	394
19.2.3	resize	394
19.2.4	push_back	395
19.2.5	赋值	395
19.2.6	到现在为止我们设计的 vector 类	397
19.3	模板	397
19.3.1	类型作为模板参数	398
19.3.2	泛型编程	399
19.3.3	容器和继承	401
19.3.4	整数作为模板参数	402
19.3.5	模板参数推导	403
19.3.6	一般化 vector 类	403
19.4	范围检查和异常	405
19.4.1	附加讨论：设计上的考虑	406
19.4.2	使用宏	407
19.5	资源和异常	408
19.5.1	潜在的资源管理问题	409
19.5.2	资源获取即初始化	410
19.5.3	保证	411
19.5.4	auto_ptr	412

19.5.5 vector 类的 RAII .....	412	21.6.3 另一个 map 实例.....	458
第 20 章 容器和迭代器 .....	417	21.6.4 unordered_map .....	459
20.1 存储和处理数据.....	417	21.6.5 集合 .....	461
20.1.1 处理数据 .....	417	21.7 拷贝操作 .....	462
20.1.2 一般化代码 .....	418	21.7.1 拷贝 .....	462
20.2 STL 建议 .....	420	21.7.2 流迭代器 .....	462
20.3 序列和迭代器 .....	423	21.7.3 使用集合保持顺序 .....	464
20.3.1 回到实例 .....	424	21.7.4 copy_if .....	464
20.4 链表 .....	425	21.8 排序和搜索 .....	465
20.4.1 列表操作 .....	426		
20.4.2 迭代 .....	427		
20.5 再次一般化 vector .....	428		
20.6 实例：一个简单的文本编辑器 .....	429	第 22 章 理念和历史 .....	471
20.6.1 处理行 .....	431	22.1 历史、理念和专业水平 .....	471
20.6.2 迭代 .....	431	22.1.1 程序设计语言的目标和哲学 .....	471
20.7 vector、list 和 string .....	434	22.1.2 编程理念 .....	473
20.7.1 insert 和 erase .....	435	22.1.3 风格/范型 .....	477
20.8 调整 vector 类达到 STL 版本 的功能 .....	436	22.2 程序设计语言历史概览 .....	479
20.9 调整内置数组达到 STL 版本 的功能 .....	438	22.2.1 最早的程序语言 .....	480
20.10 容器概览 .....	439	22.2.2 现代程序设计语言的起源 .....	481
20.10.1 迭代器类别 .....	440	22.2.3 Algol 家族 .....	485
第 21 章 算法和映射 .....	444	22.2.4 Simula .....	490
21.1 标准库中的算法.....	444	22.2.5 C .....	491
21.2 最简单的算法：find() .....	444	22.2.6 C++ .....	493
21.2.1 一些一般的应用 .....	446	22.2.7 今天的程序设计语言 .....	495
21.3 通用搜索算法：find_if() .....	447	22.2.8 参考资源 .....	496
21.4 函数对象 .....	448	第 23 章 文本处理 .....	499
21.4.1 函数对象的抽象视图 .....	449	23.1 文本 .....	499
21.4.2 类成员上的谓词 .....	450	23.2 字符串 .....	499
21.5 数值算法 .....	450	23.3 I/O 流 .....	502
21.5.1 累积 .....	451	23.4 映射 .....	503
21.5.2 一般化 accumulate() .....	452	23.4.1 实现细节 .....	507
21.5.3 内积 .....	453	23.5 一个问题 .....	508
21.5.4 一般化 inner_product() .....	453	23.6 正则表达式的思想 .....	510
21.6 关联容器 .....	454	23.7 用正则表达式进行搜索 .....	511
21.6.1 映射 .....	454	23.8 正则表达式语法 .....	513
21.6.2 map 概览 .....	456	23.8.1 字符和特殊字符 .....	514
		23.8.2 字符集 .....	514
		23.8.3 重复 .....	515
		23.8.4 子模式 .....	516

23.8.5 可选项 .....	516
23.8.6 字符集和范围 .....	516
23.8.7 正则表达式错误 .....	518
23.9 与正则表达式进行模式匹配 .....	519
23.10 参考文献 .....	522
第 24 章 数值计算 .....	525
24.1 介绍 .....	525
24.2 大小、精度和溢出 .....	525
24.2.1 数值限制 .....	527
24.3 数组 .....	528
24.4 C 风格的多维数组 .....	528
24.5 Matrix 库 .....	529
24.5.1 矩阵的维和矩阵访问 .....	530
24.5.2 一维矩阵 .....	532
24.5.3 二维矩阵 .....	534
24.5.4 矩阵 I/O .....	536
24.5.5 三维矩阵 .....	536
24.6 实例：求解线性方程组 .....	537
24.6.1 经典的高斯消去法 .....	538
24.6.2 选取主元 .....	539
24.6.3 测试 .....	539
24.7 随机数 .....	540
24.8 标准数学函数 .....	541
24.9 复数 .....	542
24.10 参考文献 .....	543
第 25 章 嵌入式系统程序设计 .....	547
25.1 嵌入式系统 .....	547
25.2 基本概念 .....	549
25.2.1 可预测性 .....	551
25.2.2 理想 .....	551
25.2.3 生活在故障中 .....	552
25.3 内存管理 .....	553
25.3.1 动态内存分配存在的问题 .....	554
25.3.2 动态内存分配的替代方法 .....	556
25.3.3 存储池实例 .....	557
25.3.4 栈实例 .....	557
25.4 地址、指针和数组 .....	558
25.4.1 未经检查的类型转换 .....	559
25.4.2 一个问题：不正常的接口 .....	559
25.4.3 解决方案：接口类 .....	561
25.4.4 继承和容器 .....	564
25.5 位、字节和字 .....	566
25.5.1 位和位运算 .....	566
25.5.2 bitset .....	569
25.5.3 有符号数和无符号数 .....	570
25.5.4 位运算 .....	573
25.5.5 位域 .....	574
25.5.6 实例：简单加密 .....	575
25.6 编码规范 .....	579
25.6.1 编码规范应该是怎样的 .....	579
25.6.2 编码原则实例 .....	580
25.6.3 实际编码规范 .....	584
第 26 章 测试 .....	589
26.1 我们想要什么 .....	589
26.1.1 说明 .....	590
26.2 程序正确性证明 .....	590
26.3 测试 .....	590
26.3.1 回归测试 .....	591
26.3.2 单元测试 .....	591
26.3.3 算法和非算法 .....	596
26.3.4 系统测试 .....	601
26.3.5 测试类 .....	604
26.3.6 寻找不成立的假设 .....	606
26.4 测试方案设计 .....	607
26.5 调试 .....	607
26.6 性能 .....	607
26.6.1 计时 .....	609
26.7 参考文献 .....	610
第 27 章 C 语言 .....	613
27.1 C 和 C++：兄弟 .....	613
27.1.1 C/C++ 兼容性 .....	614
27.1.2 C 不支持的 C++ 特性 .....	615
27.1.3 C 标准库 .....	616
27.2 函数 .....	617
27.2.1 不支持函数名重载 .....	617
27.2.2 函数参数类型检查 .....	618
27.2.3 函数定义 .....	619
27.2.4 C++ 调用 C 和 C 调用 C++ .....	620

27.2.5 函数指针 .....	621	27.6.2 输入 .....	632
27.3 小的语言差异 .....	622	27.6.3 文件 .....	633
27.3.1 结构标签名字空间 .....	622	27.7 常量和宏 .....	633
27.3.2 关键字 .....	623	27.8 宏 .....	634
27.3.3 定义 .....	623	27.8.1 类函数宏 .....	635
27.3.4 C 风格类型转换 .....	624	27.8.2 语法宏 .....	636
27.3.5 void * 的转换 .....	625	27.8.3 条件编译 .....	636
27.3.6 枚举 .....	626	27.9 实例：侵入式容器 .....	637
27.3.7 名字空间 .....	626	术语表 .....	644
27.4 动态内存分配 .....	626	参考书目 .....	648
27.5 C 风格字符串 .....	628		
27.5.1 C 风格字符串和 const .....	629	<b>第五部分 附录<sup>⊖</sup></b>	
27.5.2 字节操作 .....	630	附录 A C++ 语言概要 .....	
27.5.3 实例：strcpy() .....	630	附录 B 标准库概要 .....	
27.5.4 一个风格问题 .....	630	附录 C Visual Studio 简要入门教程 .....	
27.6 输入/输出：stdio .....	631	附录 D 安装 FLTK .....	
27.6.1 输出 .....	631	附录 E GUI 实现 .....	

# 第0章 致读者

“当实际地形与地图不符时，相信实际地形。”

——瑞士军队谚语

本章汇集了多种信息，目的是使你对本书剩余部分的内容有初步了解。你可以略过本章，直接阅读后面你感兴趣的部分。对教师来说，可以立即发现很多有用的内容。如果没有一个好的老师指导你学习本书，请不要试图阅读并理解本章的所有内容，只要阅读“本书结构”一节和“讲授和学习本书的方法”一节的第一部分即可。当你已经能自如编写和执行小程序时，可能需要回过头来重读本章。

## 0.1 本书结构

本书由四个部分和若干个附录组成：

- 第一部分：基本知识，介绍了程序设计的基本概念和技术，以及开始编写代码需要了解的一些 C++ 语言和库的知识。这部分包括类型系统、算术运算、控制结构、错误处理，以及函数和用户自定义类型的设计、实现和使用等内容。
- 第二部分：输入/输出，介绍了如何从键盘和文件获取数值和文本数据，以及如何生成相应的输出到屏幕和文件。然后介绍了如何以图形化方式表示数值数据、文本和几何图形，以及如何从图形用户界面(graphical user interface, GUI) 获取输入数据。
- 第三部分：数据结构和算法，关注 C++ 标准库中的容器和算法框架(标准模板库, standard template library, STL)。展示了容器(如向量、列表和映射)是如何(用指针、数组、动态内存、异常和模板)实现的以及如何使用它们。还展示了标准库算法(如排序、查找和内积)如何设计及使用。
- 第四部分：拓宽视野，通过对 C++ 思想和历史的讨论，通过一些实例(如矩阵运算、文本处理、测试以及嵌入式系统程序设计)，以及通过 C 语言的一个简单描述，为我们呈现了程序设计的一个全景。
- 第五部分：附录，提供了一些不适合作为教学但很有用的内容，如 C++ 语言和标准库的概要介绍，以及集成开发环境(integrated development environment, IDE) 和图形用户界面库(GUI 库)的入门简介等。

不幸的是，现实世界中的程序设计并不能真正分为完全独立的四个部分。因此，这种划分仅仅是对本书内容的一种粗略分类。我们认为这是一种有用的分类方法(这是显然的，否则我们不会采用它)，但现实情况往往与这种简洁的分类法相悖。例如，我们很快就会用到输入操作，但对 C++ 标准 I/O 流(input/output stream, 输入/输出流)的完整介绍却出现在本书比较靠后的部分。书中有些地方在提出某个概念时需要先介绍另外一些内容，而这与全书的布局不符，对此，我们会在此处简明介绍这些内容，以便更好地提出概念，而不是仅仅指出这些内容的完整介绍在书中什么地方。刻板的分类法更适合于手册而不是教材。

本书内容的顺序是由程序设计技术决定的，而不是程序设计语言特性，参见 0.2 节。附录 A 是按语言特性组织的。

### 0.1.1 一般方法

在本书中，人称都是直接的，简单清楚，不像很多科技论文中那种惯用的“专业”的婉转称呼方式。当用到“你”时，我们的意思就是“你、读者”；而“我们”指“我们、作者和教师”，或者指读者和我们一起在讨论某个问题，就好像我们在一个教室中一样。

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，你最终还是会再次回到这个地方。然而，这么做要有度，因为除了索引和交叉引用之外，对本书其他部分，你随便翻开一页，就开始学习并希望成功，这几乎是不可能的。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的单元，这意味着应将其一口气读完（当然这只是从理论上讲，实际上由于学生紧密的学习计划，不总是可行的）。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，一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，做了简单练习和一些习题时，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。我们按主题将章组合成“部分”，例如，第二部分都是关于输入/输出的内容。每一部分都可以作为一个很好的完整的复习单元。

“它回答了我想到的所有的问题”是对一本教材常见的称赞，这对细节技术问题是很理想的，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，我们希望提出更多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。

我们努力尊重你的聪明才智，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，我们宁可有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种语言特性的重要性，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，我们的意思是，你如果不掌握它，或早或晚都会因此而浪费时间。我们喜欢幽默，但在本书中使用很谨慎。经验表明，人们对什么是幽默的看法大相径庭，不恰当地使用幽默会把人弄糊涂。

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、表达你所面临的问题求解方案而已。我们尽量避免“无恶意的谎言”，也就是说，对于那些清晰易理解，但在实际编程和问题求解时容易弄错的内容，对其介绍避免过度简单化。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 (Special Edition)》<sup>①</sup>一书 (Addison-Wesley 出版社，2000 年) 和 ISO 的 C++ 标准。

### 0.1.2 简单练习、习题等

程序设计不仅是一种脑力活动，实际动手编写程序是掌握程序设计技巧必不可少的一个环节。本书提供以下两个层次的程序设计练习：

① 本书中文版已由机械工业出版社引进出版，书名为《C++ 程序设计语言(特别版)》。——编辑注

- **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机械的实际编程技巧。一个简单练习通常由对单个程序的一系列修改练习组成。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：**有些习题比较简单而有些则很难，但大多数习题都是想给学生留下一定的创造和想象的空间。如果时间紧张，你可以做少量习题。但至少应该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，这才是学习成功之道。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）。一个项目就是要编写一个完整的有用的程序。理想情况，一个项目由一个多人小组（比如三个人）共同完成，最好在学习第三部分的同时花大概一个月时间来完成整个项目。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，我们在正文的段落之间放置了一些有“试一试”标识的文字，给出了对于编程实践的一些简单建议。“试一试”本质上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。如果你略去一个“试一试”而没有去尝试它（也许因为你的手边没有计算机，或者你过于沉浸在正文的内容中），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++ 方面的主要词汇表。如果你希望理解别人关于程序设计的陈述，或者想明确表达出你自己的思想，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。

### 0.1.3 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。你不能奢望花四个月时间就成为一名程序设计专家，就像你不能奢望花四个月、半年或一年时间就成为一名生物学专家、数学家、自然语言（如中文、英文或丹麦文）方面的专家或者小提琴演奏家一样。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，最好的进一步学习的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业的书（例如，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++》<sup>①</sup>，Sutter 的《Exceptional C++》<sup>②</sup>或 Gamma 等人的《Design Patterns》<sup>③</sup>）。完整的参考书目，参见 0.6 节或本书最后的“参考书目”一节。

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（即使你并不想做一名程序员）。

## 0.2 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。本书主题是“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服务于两个目的：提出一种技术、概念或原理，介绍一种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

---

<sup>①</sup> 本书英文影印版和中文版均已由机械工业出版社引进出版。——编辑注

<sup>②</sup> 本书英文影印版已由机械工业出版社引进出版。——编辑注

<sup>③</sup> 本书中文版已由机械工业出版社引进出版，书名为《设计模式》。——编辑注

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做“有魔力的东西”。

对于程序员来说，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”，也可以描述为“具体优先”和“基于概念”。首先，我们快速地（好吧，是相对快速的，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。我们着重介绍简单具体的代码实例，因为相对于抽象概念，人们能更快地领会具体实例，这是大多数人的学习方法。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，对刚刚还工作得很好的程序只是稍加改动，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！”但是你见过小孩儿学习走路吗？小孩儿确实在学会平稳地慢慢走路之前就开始自己学着跑了。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儿学习母语的方法，也是教授外语的最有效的方法。有时你不可避免地被难题困住，我们鼓励你向授课老师、朋友、同事、辅导员以及导师等寻求帮助。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，我们向你介绍一些入门技巧，来拓宽你的知识和技巧基础。你通过实例和习题来强化理解，我们为你提供一个程序设计的概念基础。

我们重点强调思想和原理。思想能指导你求解实际问题——可以帮助你知道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“因为事情就是这样的”这种解释。更为重要的是，如果真正理解了思想和原理，你就能将已有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，也是在浪费时间。我们认为你的时间很珍贵，尽力不浪费它。

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你需要的信息，你可以借助索引和目录来查找信息。不要忘了编译器和互联网也有在线帮助功能。但要记住，要对所有互联网资源保持足够高的怀疑，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的代码。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如我们的实例更多。数十万行规模的成功的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。最快的理解现实世界程序的途径是好好研究一些基础的小程序。

另一方面，我们不会用“cute 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。

### 0.2.1 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，我们选择省略内容与选择保留内容至少同样重要。

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)，可能对你有用：

- **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，这样的编程实践很糟糕。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。这是手册适合的用途，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。例如，如果你没有实际体验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组织代码和开发工作的最好方法，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。本书确实在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 章才介绍如何设计一个类，而直到第 12 章才介绍类层次。
- “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，但不介绍其下蕴含的技术和功能。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，以及它们合理的应用范围，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一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，我们不把一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。

## 0.2.2 程序设计和程序设计语言

我们首先介绍程序设计，把程序设计语言作为一种工具放在第二位。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、原理和技术，但是不能孤立地学习这些内容。例如，不同程序设计语言在语法细节、编程想法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（参见第 5 章和第 6 章），构造不变式（参见 9.4.3 节），以及接口和实现细节分离（参见 9.7 节和 14.1 节～14.2 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，设计、组织代码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。本书讲述的与 C++ 语言相关的内容只是它的一个子集，即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。

## 0.2.3 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上！我们认为可移植性和对多种平台架构和操作系统的利用是非常重要的特性。本质上，本书的每个例子不仅是 ISO 标准 C++ 程序，而且是可移植的。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、连接和运行 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 C 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（integrated development environment，集成开发环境）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，下面给出的是用 GNU C++ 编译器，在 UNIX 或 Linux 系统中编译、连接和执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：

```
g++ -o my_program my_file1.cpp my_file2.cpp
my_program
```

是的，这真的就是全部。

## 0.3 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，因为程序设计包含很多专业技巧，通常不能归类于任何一种科学。
- 更窄，因为就我们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），作为软件构造和维护领域第一门基础课程（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，但我们是以理论和经验为基础来讲解程序设计，是把它作为一种实践技能，而不是一门科学。

## 0.4 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你的思想，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的、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来说明这一切，而不是通过“布道”或是对问题求解详细“处方”的描述。

## 0.5 反馈方法

我们认为不存在完美的教材，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。我们也欢迎有关更好的习题、更好的实例、增加内容、删除内容等建议。大家提出的建设性的意见会帮助将来的读者，我们会将勘误表张贴在支持网站上：[www.stroustrup.com/Programming](http://www.stroustrup.com/Programming)。

## 0.6 参考文献

本节列出了本章提及的参考文献，以及可能对你有用的其他一些文献。

- Austern, Matthew H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999. ISBN 0201309564.
- Austern, Matthew H. (editor). “Technical Report on C++ Standard Library Extensions.” ISO/IEC PDTR 19768.
- Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall, 2006. ISBN 0131872493.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0201633612.
- Goldthwaite, Lois (editor). “Technical Report on C++ Performance.” ISO/IEC PDTR 18015.

- Koenig, Andrew (editor). *The C++ Standard*. ISO/IEC 14882:2002. Wiley, 2003. ISBN 0470846747.
- Koenig , Andrew, and Barbara Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 020170353X.
- Langer, Angelika, and Klaus Kreft. *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley, 2000. ISBN 0201183951.
- Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley, 2001. ISBN 0201749625.
- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd 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, 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.
- 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 (Special Edition)*. Addison-Wesley, 2000. ISBN 0201700735.
- Stroustrup, Bjarne. “C and C++: Siblings”; “C and C++: A Case for Compatibility”; and “C and C++: Case Studies in Compatibility.” *C/C++ Users Journal*, July, Aug., Sept. 2002.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 2000. ISBN 0201615622.

更全面的参考文献列表可以在本书最后“参考书目”一节找到。

## 0.7 作者简介

你也许有理由问：“是一些什么人想要教我程序设计？”那么，下面是作者简介。我(即 Bjarne Stroustrup)和 Lawrence “Pete” Petersen 合著了本书。我还设计并讲授了面向大学生初学者(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。在过去的大约 30 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，现在我是该委员会语言演化工作组的主席。

这是我第一本入门级的书。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 研究实验室。

我现在是美国国家工程院的院士，ACM 院士和 IEEE 院士，贝尔实验室院士和 AT&T 院士。我获得了 2005 年度 Sigma Xi 科学研究社区科学成就 William Procter 奖，我是首位获得此奖的计算



机科学家。

至于工作之外的生活，我已婚并有两个孩子，一个是医学博士，另一个目前是博士生。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，我还喜欢访问世界各地有趣的地方和人。为了能够享受美食，我还坚持跑步。

关于我的更多的信息，请浏览我的网站：[www.research.att.com/~bs](http://www.research.att.com/~bs) 和 [www.cs.tamu.edu/people/faculty/bs](http://www.cs.tamu.edu/people/faculty/bs)。特别地，你可以在那里找到我名字的正确发音。

#### Lawrence “Pete” Petersen

在 2006 年末，Pete 如此介绍他自己：“我是一名教师。将近 20 年来，我一直在德州农工大学讲授程序设计。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的。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 ~ 1981 年及 1985 ~ 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、设计和实现供人使用的实用软件。我在佐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。我的信息和运营管理学博士学位是在德州农工大学获得的。

我的妻子芭芭拉和我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们和他们的家人，特别是我们的几个孙子孙女，安吉丽娜、卡洛斯、苔丝、埃弗里、尼古拉斯和乔丹。”

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。

## 附言

很多章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。这样做是因为我们意识到，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，也是可以预料的。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相当胜任的程序员。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。



# 第1章 计算机、人与程序设计

“只有昆虫才专业化。”

——R. A. Heinlein

在本章中，我们讲一些可以使程序设计变得重要和有趣的事情。我们也讲一些基本的思想与理想。我们希望揭穿几个有关程序设计与程序员的流行的神话。本章是现在可以跳过的内容，当你困扰于一些编程问题和怀疑是否值得阅读时，可以返回阅读本章。

## 1.1 介绍

正如大多数的学习一样，学习程序设计就像鸡和蛋的问题。我们希望开始学习一件事，但是我们也希望了解为什么学习它。我们想学习一种实用技能，但是我们也希望确定它不只是短暂的流行。我们希望自己将不会浪费时间，但是我们也希望不被炒作和道德说教所打扰。现在，我们仅抱着有兴趣的态度来阅读本章，并在为什么技术细节会符合课堂外的情况，而感到要更新记忆时返回来阅读。

本章是对如何发现编程的兴趣与重要性的个人陈述。它解释了是什么激励我们数十年后还在这个领域中不断前进。通过阅读本章会得到可能的最终目标和程序员可能是哪种人的想法。针对初学者的技术书籍毫无疑问会包含很多基础的内容。在本章中，我们将眼睛从技术细节上移开，并且考虑一个更大的画面：为什么程序设计是一个有价值的活动？程序设计在我们的文明中扮演怎样的角色？程序员所做的贡献哪里值得骄傲？程序设计如何融入软件开发、部署和维护这一更大的领域？当人们谈论“计算机科学”、“软件工程”、“信息技术”等时，程序设计如何融入其中？一个程序员需要做什么？一个好的程序员需要具备哪些技能？

对于一个学生来说，理解一种思想、一项技术或一章的最紧迫的原因，可能是以好的成绩通过一次考试，但是在里会有更多比学习重要的东西！对于那些在软件公司工作的人来说，理解一种思想、一项技术或一章的最紧迫的原因，可能是找到一些对目前的项目有帮助的东西，并且不会使控制你的薪水、升职和解雇的老板感到厌烦，但是在里会有更多比学习重要的东西！当我们感到自己的工作会在细小的方面改善人们生活的世界，我们会努力将工作做到最好。对于那些在几年内完成的任务（在专业和职业发展中的“事情”），理想和更抽象的思想是决定性的。

我们的文明运行在软件之上。改进软件和发现软件的新用途是个人改进生活的两种方法。程序设计在这里扮演着一个基本的角色。

## 1.2 软件

好的软件是看不见的。你不能看到、感觉、称量或敲击它。软件是运行在计算机上的程序的集合。有时候我们可以看到一台计算机。我们通常仅仅看到由一些东西构成的计算机，例如一部电话机、一台照相机、一台面包机、一辆汽车或一台风力发电机。我们可以看到软件如何工作。如果软件没有按预想的方式工作，我们会感到厌烦或受到伤害。如果软件按预想的方式工作而不符合我们的需要，我们也会感到厌烦或受到伤害。

世界上有多少台计算机？我们并不知道，至少有数十亿台。世界上的计算机数量有可能超过人的数量。2004年，据ITU(国际电信联盟，一个联合国机构)的统计，共有7.72亿台个人计算机(PC)，以及更多的不属于PC的计算机。

你每天会使用多少台计算机(直接或间接)？在一辆汽车中有超过30台计算机，在移动电话中有2台计算机，在MP3播放器中有1台计算机，在照相机中有1台计算机。我有自己的笔记本电脑与台式电脑。在夏天保持温度与湿度的空调也是1台简单的计算机。控制计算机科学系的电梯也是1台计算机。如果你使用的是现代的电视机，那么其中至少会有1台计算机。如果你进行一次网上冲浪，将会通过通信系统接触几十、也可能几百台服务器，通信系统中又包含数千台计算机(电话交换机、路由器等)。

我并不是在驾驶一辆后座上带着30台笔记本电脑的汽车！重点是这些计算机看起来不像通常的计算机(带有一个屏幕、一个键盘和一个鼠标等)；它们作为很小的一个“零件”嵌入我们使用的设备中。正因为如此，汽车中没有哪个东西看起来像计算机，即使是用于显示地图和行驶方向的屏幕(这种小工具在汽车中很流行)。但是，在汽车引擎中会包含几台计算机，用于完成燃油喷射控制与温度监控工作。汽车的助力转向系统包含至少1台计算机，广播与安全系统包含多台计算机，我们甚至怀疑车窗的开启/关闭都由计算机来控制。新型号的汽车甚至有用于持续检测轮胎气压的计算机。

你在日常生活中所做的事情需要依赖于多少台计算机？你需要吃饭；如果你生活在一个现代化的城市中，为了将食物提供给你，需要计划、运输和存储。对这一分配网络的管理当然是计算机化的，它们之间通过通信系统连接起来。现代化农业也是高度计算机化的，你可以在牛舍附近发现用于监控牛群(年龄、健康、产奶量等)的计算机，农业设备也越来越计算机化。如果某些事情出错，你可以在报纸上阅读到它。当然，报纸上的文章通过计算机来书写，通过计算机来进行页面设置，以及通过计算机设备来印刷(如果你仍阅读纸质的报纸)，通常需要以电子形式传输到印刷厂。书籍的生产采用的是同样的方式。如果你需要上下班，计算机通过监控交通流量以避免交通堵塞的。你喜欢乘坐火车？火车也是计算机化的。有些操作甚至不需要司机来完成，火车的子系统(广播、刹车和监票)包括很多计算机。今天的娱乐业(音乐、电影、电视、舞台表演)是大量使用计算机的用户。即使非卡通的电影也在大量使用(计算机)动画，音乐和摄影也趋向于数字化存储和传输(使用计算机)。如果你生病，医生为你做的检查要使用计算机，病历通常是计算机化的，大多数你遇到的用于治疗的医学仪器也包含计算机。除非你碰巧住在树林中的草屋中，并且不使用任何电动工具(包括电灯)，否则你都会使用能源。石油被发现、提炼、加工和传输的过程，从钻头深入地下到本地的汽油(天然气)加油站，整个过程中的每个步骤都要使用计算机。如果你使用信用卡来购买汽油，你也会访问一组计算机。对于煤炭、天然气、太阳能和风力发电，它们都会经过同样的过程。

迄今为止的例子都是“可操作的”，它们都直接包含在你所做的事情中。你未参与其中的事情是设计中的重要和有趣的部分。你穿着的衣服、你交谈用的电话和你调制自己喜欢的饮料用的咖啡机，这些都是通过计算机来设计与生产的。优质的现代摄影镜头、造型精美的日常工具和器具，这些几乎都要归功于基于计算机的设计与生产方式。那些设计我们周围环境的工匠、设计师、艺术家和工程师，他们从很多以前被认为是基本工作的物理限制中解脱出来。如果你生病，那些用来治愈你的药品也是使用计算机设计的。

最后，科学研究本身严重依赖于计算机。对于用于探索遥远的恒星秘密的望远镜，它们离开

计算机是无法设计、制造和操作的，它们产生的大量数据离开计算机是无法处理的。个别生物学领域的研究人员没有被严重计算机化(不包括照相机、数字录音机、电话等的使用)，但是回到实验室中，数据要使用计算机模型来储存、分析和检查，并且要和其他科研人员通信。现代化学和生物学(包括医学)大量使用计算机，在几年前就达到人们做梦也想不到的程度，并且至今对大多数人仍是难以想象的。人类基因测序是通过计算机完成的。让我们描述得更准确一些，人类基因测序是人使用计算机完成的。在所有这些例子中，我们可以看到计算机可以帮助我们完成某些事，在没有计算机的情况下需要花费更多时间。

每台计算机都需要运行软件。如果没有软件，计算机就是由硅、金属和塑料组成的昂贵的大块头，其与门吸、船锚和空间加热器没有多大区别。软件中的每行代码都是由不同的人编写的。如果软件在运行中有错误，实际执行的每行在合理的最低限度内。它们都正常运行是很惊人的事。我们谈论的是用几百种编程语言编写的几十亿行程序代码(或程序文本)。使它们都正常运行需要付出惊人的代价和令人难以想象的技巧。我们希望对所依赖的每种服务和工具进行更多的改变。思考一下你所依赖的某种服务和工具，你希望看到它们有怎样的改进？如果没有的话，我们希望服务和工具更小(或更大)、更快、更可靠、更有特点、更容易使用、更高性能、更好看和更便宜。这些我们想做的改进的相似点是都需要编程。

### 1.3 人

计算机是人制造的，供人使用的。计算机是一种非常通用性的工具，它可以被用于很多你无法想象的任务。计算机运行一个程序，以使它对一些人有用。换句话说，计算机只是一个硬件，直到一些人(程序员)编写代码使它能做某些事情。我们经常会忘记软件，甚至经常会忘记程序员。

好莱坞和类似的“流行文化”等谣言的来源已经给程序员带来很大的负面影响。例如，我们总是看到孤独的、肥胖的、丑陋的、不懂社交技巧的讨厌鬼，并且总是痴迷于视频游戏和闯入其他人的计算机。他(几乎总是男人)可能是想毁灭世界，也可能是想拯救世界。很明显，这种漫画人物的“温和版本”在现实生活中确实存在，但是依我们的经验他们中的软件开发者，并不比律师、警官、汽车销售员、记者、艺术家或政治家更多。

我们思考一下计算机在现实生活中的应用。它们是在一个黑屋子中独立工作吗？当然不是，一个成功的软件、计算机设备或系统的创建，需要包括几十、几百或几千人扮演一系列扑朔迷离的角色，例如程序员、(程序)设计者、测试人员、美工人员、开发小组管理者、实验心理学家、用户界面设计者、分析人员、系统管理员、客户关系人员、音效工程师、项目经理、质量工程师、统计人员、硬件接口工程师、需求分析工程师、安全主管、数学家、销售支持人员、答疑人员、网络设计人员、方法论学家、软件工具管理员、软件库管理员等。这些角色的范围很广，随着组织之间的变换使人更加迷惑。一个组织中的“工程师”可能是另一个组织中的“程序员”，也可能是另一个组织中的“开发人员”、“技术组成员”或“结构设计师”。这里有多个组织，不同成员可以在其中找到自己的头衔。并不是所有角色都与编程直接相关。但是，我们每个人都曾看到人们扮演每个角色的例子，他们将读写代码作为自己工作的重要部分。另外，一个程序员(扮演这些角色中的一个或多个)在短时期内会和不同应用领域的人打交道，例如生物学家、发动机设计师、律师、汽车销售员、医学研究员、历史学家、地理学家、宇航员、飞机工程师、木材库经理、火箭科学家、保龄球馆建设者、记者和漫画家(这是从个人经验中得到的)。有些人可能有时是一个程序

员，而在职业生涯的另一个阶段扮演非程序员的角色。

程序员是孤立的这一神话本身只是一个神话而已。那些喜欢工作在自己选择的工作领域的人，经常痛苦地抱怨被“打扰”或开会的次数。由于现代软件开发是一种团队行为，因此那些喜欢和别人交互的人会感到更轻松。这意味着社会和沟通能力是必不可少的，并且其价值远远大于陈规旧习。在一份对程序员很有用的技能的简短列表中(你是在从现实的角度定义程序员)，你会发现与不同背景的人进行沟通的能力最重要，这种沟通可能是非正式的会议、书面形式和正式介绍。我们相信，除非你完成过一个或两个团队项目，否则你不会知道什么是编程以及是否喜欢它。我们喜欢编程的理由是我们遇到的都是很好的、有趣的人，并且将访问风格各异的地方作为我们职业生涯的一部分。

所有这些是指那些有各种各样的技能、兴趣和工作习惯的人，对开发一个好的软件来说是必不可少的。我们的生活质量(有时甚至是生活本身)依赖于那些人。没有人可以扮演我们这里提到的所有角色，也没有明智的人希望扮演每个角色。重点是你有比自己所能想到的更大的选择空间，而不是不得不做出某个特定的选择。作为个人，你将“流向”那些符合你的技能、才智和兴趣的工作领域。

我们谈论的是“程序员”与“编程”，但是编程很明显只是整个画面的一个部分。那些设计船只或移动电话的人不会将自己做程序员。编程是软件开发中的一个重要部分，但并不是全部。相似地，对于大多数产品来说，软件开发是产品开发中的一个重要部分，但并不是全部。

我们并不假设你(我们的读者)希望成为一个专业程序员，将剩余的工作生涯用于编写代码。即使是那些优秀的程序员，他们也不会将大部分时间用在编写代码上。理解问题需要占用更多的时间，并且通常更消耗智力。智力挑战是很多程序员认为编程有趣的出发点。很多优秀程序员通常不是计算机科学专业的。例如，如果你进行基因研究方面的软件开发，理解分子生物学将会对你有更大的帮助。如果你进行中世纪文学分析方面的程序设计，阅读一些这类文学作品和掌握一门或多门相关语言将会对你有更大的帮助。特别是对于抱有“只关心计算机和编程”态度的人，他们将会难以与那些非程序员的同事交流。这样的人不仅会错过人与人交流(即生活)中最棒的那部分，而且也不会成为成功的软件开发人员。

于是，我们如何假设呢？编程是一种挑战智力的技能，需要经过很多重要与有趣的训练。另外，编程是我们这个世界的重要组成部分，不了解基本的编程知识就像不了解基本的物理、历史、生物或文学知识一样。那些对编程完全无知的人相信它是魔术，让这样的人承担很多技术工作是危险的。另外，编程可以带来乐趣。

但是，我们如何假设编程的用途？你也许将编程作为未来学习和工作的重要工具，而不是成为一个专业的程序员。你也许将与其他人进行专业和个人的交流，这些人可能是设计师、作家、经理或科学家，具有编程方面的基础知识将会有一定优势。你也许将专业水平的编程作为你学习和工作的一部分。即使你成为一个专业的程序员，也不意味着你除了编程之外不做任何事。

你可能成为一名计算机或计算机科学方面的工程师，但是这样也并不是“编程占据所有时间”。编程是用代码表达你的思想的方式，也是一种协助求解问题的方式。除非你有值得表达的思想和值得解决的问题，否则编程没有用处(纯粹是浪费时间)。

这是一本关于编程的书，我们曾经承诺本书可以帮助你学习如何编程，那么为什么我们要强调非编程的内容与编程的有限作用呢？一个优秀的程序员会理解代码和编程技术在一个项目中的作用。一个优秀的程序员(在多数情况下)是一个优秀的团队成员，并且会努力理解代码和其产

品如何很好地支持整个项目。例如，想象我在为一个新的 MP3 播放器进行编程，则我关心的只是代码优美程度和提供的简洁功能的数量。我可能一直在大型的、功能强大的计算机上运行这些代码。我可能对声音编码理论不屑一顾，因为它是“与编程无关的”。我将会待在自己的实验室里，而不是走出去与潜在的用户交流。这样，用户毫无疑问将对音乐有不好的体验，并且不欣赏图形用户界面(GUI)编程的最新发展。这样做的后果是可能对项目带来一场灾难。更大的计算机意味着更昂贵的 MP3 播放器和更短的电池寿命。编码是数字化音乐控制的重要部分，忽视编码技术的发展会导致每首歌曲增加所需的存储空间(不同编码获得相同品质的输出时的存储空间大小差异超过 100%)。无视用户的喜好(在你看来是奇怪的和过时的)，通常会导致用户选择其他产品。编写一个好的程序的重要部分是理解用户的需求，并且在执行(即编码)时实现这些需求。为了完成上述对一个坏程序员的描绘，我倾向于将其描述成对细节的狂热和对简单测试的代码正确性的过分自信。我们鼓励你成为一个好的程序员，只有具有广阔的视野才能生产出好的软件。这既是社会价值也是个人自我实现之所在。

## 1.4 计算机科学

即使是在最广泛的定义中，也最好将编程看做某些更大事物的一部分。我们可以将编程看做计算机科学、计算机工程、软件工程、信息技术或其他软件相关的学科。我们将编程看做科学和工程中的计算机和信息领域的实现技术，同样也是物理学、生物学、医学、历史学、文学和其他学术或研究领域的实现技术。

请思考计算机科学。在 1995 年，美国政府的“蓝皮书”对它的定义如下：“对计算系统和计算的系统研究。这个学科造就的知识体系包含理解计算系统和方法的理论，设计方法学、算法和工具，测试概念的方法，分析和验证的方法，以及知识的表示和实现。”正如我们所预料的那样，维基百科条目所给出的概念不太正式：“计算机科学或计算科学是对信息和计算的理论基础的研究，以及它们在计算机系统中的实现和应用。计算机科学包含很多子领域，有些强调特定结果的计算(例如计算机图形学)，另一些关于计算问题的性能(例如计算复杂度理论)。有些集中在实现计算的挑战上。例如，编程语言理论研究描述计算的方法，而计算机编程使用特定的编程语言来解决特定的计算问题。”

编程是一种工具。它是一种针对基础和实践问题的基本工具，使这些问题可以通过实验来测试、改进和应用。编程是思想和理论的实际交汇。这是计算机科学可以成为一种实践训练而不是纯理论，并且影响世界的原因所在。在这方面，和很多其他事情一样，编程必不可少的是训练和实践的良好结合。它不应退化为这样一种活动：只是编写一些代码，用旧的方式满足眼前的简单需求。

## 1.5 计算机已无处不在

没有人知道计算机或软件相关的所有事。本节内容只是给出一些例子。你也许会看到自己想看的东西。你至少可以理解计算机的应用范围和编程所涵盖的范围远远超出任何个人可以完全掌握的程度。

大多数人认为计算机只是一个带有显示器和键盘的灰色盒子。这种计算机往往被放置在桌子下面，用于玩游戏、收发消息和邮件、播放音乐。另一些计算机称为笔记本电脑，它们被无聊的商人们在飞机上使用，查看报表、玩游戏和观看视频。这种讽刺性的描述只是冰山一角。大多

数的计算机工作在我们看不到的地方，并且作为保持社会运转的系统的一部分。它们中的一些可能占据整个房间，另一些可能比一枚小的硬币还小。这些有趣的计算机不是通过键盘、鼠标或类似的设备直接与人进行交互。

### 1.5.1 有屏幕和没有屏幕

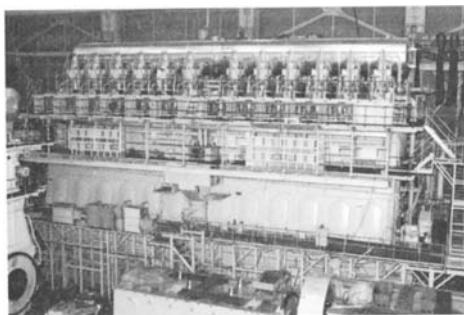
计算机是一个相当大的、带有屏幕和键盘的方盒子的想法很普遍，并且通常是难以动摇的。但是，我们考虑一下这两种计算机：



这两种用于计时的工具本质上也是计算机。实际上，我们猜测它们基本上是带有不同 I/O(输入/输出)系统的相同模式计算机。左边那个驱动一个小的屏幕(与普通计算机的屏幕类似，但是更小)，第二个驱动小的电子发动机来控制传统的表针和用于表示日、月的数字表盘。这些输入系统都有4个按钮(右边那个更容易看清楚)和1个无线电接收器，它被用于与非常精确的“原子”时钟保持同步。控制这两个计算机的大多数程序都是相同的。

### 1.5.2 船舶

这两张图片显示的是一台大型的船用柴油机和它可能驱动的巨大的船舶：



我们考虑一下计算机和软件在这里扮演的重要角色：

- **设计：**当然，船舶和引擎都是使用计算机来设计的。这个过程中用到计算机的地方非常多，主要包括结构和工程制图、一般的计算、空间和部件的可视化以及性能的模拟。
- **建造：**现代化的造船厂是高度计算机化的。船舶组装是通过计算机来严格规划的，工作是通过计算机来指导的。焊接是由机器人来完成的。特别是双壳油船，没有小的焊接机器人在壳体之间焊接是无法完成的。那里没有可以容纳一个人的空间。为船舶切割钢板是世界上最早的 CAD/CAM(计算机辅助设计和计算机辅助制造)应用之一。
- **引擎：**引擎采用电子燃料喷射技术，和由数十台计算机控制。对于一台 10 万马力的引擎(就像照片中的那台)，这是一个非同一般的任务。例如，引擎管理计算机要持续调节燃

料注入，以便将引擎不协调可能导致的污染降到最小。很多与引擎相连接的泵（船舶的其他部分）本身也是计算机化的。

- 管理：船舶需要航行到特定的港口去装卸货物。船队的日程安排是一个持续的过程（当然是计算机化的），其目标是可以根据气象、供应和需求、港口的空间和吞吐量来调整航线。甚至有网站可以用来查询大型商船在某个时刻的位置。照片中的船舶碰巧是一艘集装箱船（这个世界上最大的船舶，397米长和56米宽）。其他类型的大型现代化船舶都以相似的方式进行管理。
- 监控：一艘远洋船舶在很大程度上是自治的，它的全体船员可以在到达下一个港口前处理大多数可能产生的紧急事件。但是，它们仍是一个全球网络中的一部分。船员可以访问相当精确的气象信息（通过计算机化的人造卫星）。它们拥有 GPS（全球定位系统）和计算机控制、计算机增强的雷达。如果船员需要休息，大多数系统（包括引擎、雷达等）可以在航线控制室中进行监控（通过卫星）。如果发现任何异常或通信连接中断，船员会收到通知。

让我们考虑一下，如果在这段简短的介绍中明确提到或暗示的数百台计算机之一出现故障将意味着什么。在第25章（“嵌入式系统编程”）中，将会对这种情况做出稍微详细的解释。为一艘现代化船舶编写代码是一件讲究技巧且有趣的事。它也是很有用的。运输成本实际上是很低廉的。你在购买那些不在本地生产的东西时会赞成这一点。海洋运输总是比陆地运输更便宜，其主要原因在于计算机和信息的大量使用。

### 1.5.3 电信

这两张图片显示的是一台电话交换机和一部电话（它碰巧还是一台照相机、一台MP3播放器、一台FM收音机和一个Web浏览器）：



我们考虑一下计算机和软件在这里扮演的角色。你拿起一部电话拨号，你所拨打的人响应，然后你们可以通话。你可能是在与一台应答器通话，可能通过电话中的照相机发送一张照片，或者发送一条文本消息（按“发送”使电话完成拨号）。很明显，电话是一台计算机。如果电话（像多数的移动电话）拥有一个屏幕，并且提供更多超过传统“老式电话服务”的功能（如Web浏览器），则这种情况将会特别明显。实际上，这类电话通常包含多台计算机：一台用于管理屏幕，一台用于与电话系统通话，可能还会包含更多计算机。

计算机用户最熟悉的可能是电话中的管理屏幕、进行Web浏览等：它为“所有常见的功能”提供一个图形化用户界面。大多数用户不知道、甚至感到意外的是与小的电话协作，完成通话工作

背后的庞大系统。我拨叫一个德克萨斯州的号码，而这时你正在纽约城度假，但是你的电话铃声在几秒钟内响起，并且我听到你伴着城市交通的嘈杂声说“你好！”。很多电话可以在地球上的两个位置之间通话，我们认为这是理所当然的。我的电话如何找到你的电话？声音如何被传输？声音如何被编码加入数据包？这些问题的答案可以填满比本书更厚的几本书，但是它会涉及分布在相关地理区域中的数百台计算机中的软件和硬件。如果你是不幸的，还会涉及几个通信卫星（它们也是计算机化的）。“不幸”的原因是不能完全补偿进入2万英里的太空的代价，光速（因此是你的声音传播的速度）是有限的（光纤电缆更好：更短、更快和传输更多数据）。这些工作多数是很好的，骨干通信系统的可靠性可以达到99.9999%（例如，在20年中有20分钟断线，等于 $20/20 \times 365 \times 24 \times 60$ ）。我们遇到的麻烦通常出现在移动电话和最近的电话交换机之间的通信。

在这里，软件用于在电话之间建立连接，用于将语音编码为数据包通过有线或无线链路传输，用于路由这些消息，用于恢复各种故障，用于持续监控服务的质量和可靠性，当然也用于记账。跟踪系统中所有物理部分，也需要大量智能软件：谁和谁通话？哪部分进入一个新的系统？何时需要进行一些预防性维护？

这个世界的主干通信系统由很多半独立但互连的系统组成，它可能是最大和最复杂的人工产品。为了使事情更真实一些，记住，这不只是令人厌烦的老式电话带有一些新的铃声或哨音。各种基础设施已经合并。它们也是Internet（Web）、金融和贸易系统、广播电台播放的电视节目运行的基础。因此，我们提供另一对图片来说明通信：

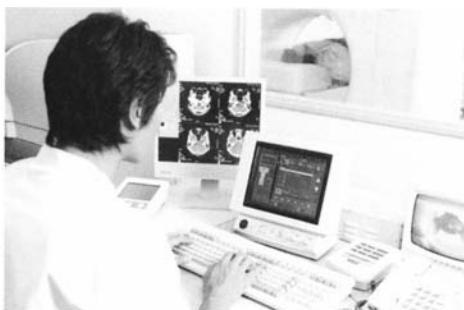


左图是位于纽约华尔街的美国证券交易所的“交易大厅”，右边的地图表示部分的Internet骨干网（一张完整的地图将会更加零乱）。

碰巧，我们也喜欢数字照片和用计算机绘制的地图来使知识可视化。

#### 1.5.4 医疗

这两张图片显示的是一台CAT（计算机轴向断层）扫描仪和一间计算机辅助手术室（也称为“机器人辅助手术”或“机器人手术”）：

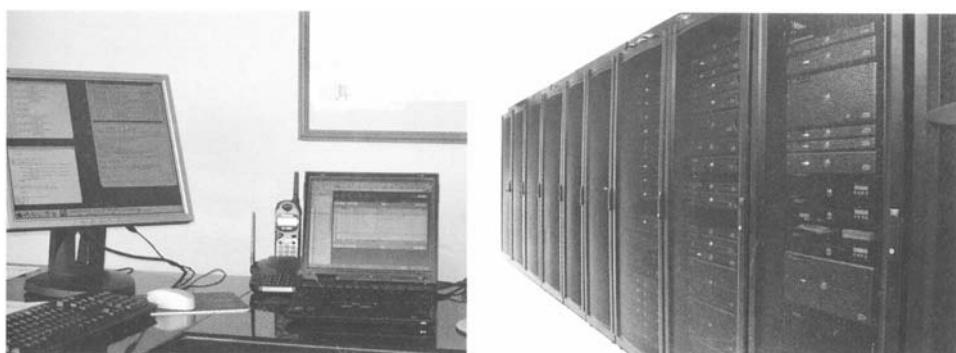


考虑一下计算机和软件在这里扮演的重要角色。扫描仪基本上就是计算机，它发出的脉冲由一台计算机来控制，它读取的内容对我们来说是杂乱无章的，除非将其通过复杂的算法转换成我们可以识别的人体相应部位的(三维)图像。为了进行计算机化的手术，必须分为几个步骤。各种成像技术用于使外科医生看清患者的身体内部，以便尽可能大和更亮地看清手术的部位。外科医生借助计算机可以使用人手无法握住的工具，或者不必割开身体就可到达某些部位。微创手术(腹腔镜手术)是一个最简单的例子，它减少了数百万人的痛苦和康复时间。计算机可以帮助稳定外科医生的“手”，以便完成正常情况下不可能完成的更细致的工作。最后，“机器人”系统可以远程操作，因此医生有可能远程(通过 Internet)医治病人的。计算机和编程是难以置信的、复杂的和有趣的。用户界面、设备控制、成像技术中的每一项，都足以让数千名研究人员、工程师和程序员忙碌几十年。

我们听到很多医生关于哪种新的工具对他们的工作最有帮助的讨论：CAT 扫描仪？MRI 扫描仪？自动血液分析仪？高分辨率超声波仪？PDA？在经过讨论以后，令人惊讶的“胜利者”从这场“竞争”中出现：即时访问病历。了解患者的医疗史(早期疾病、早期用药、过敏史、遗传问题、一般健康状况、当前用药等)会简化问题的诊断和减小发生错误的机会。

### 1.5.5 信息领域

这两张图片显示的是一台普通 PC(也可能是两台)和服务器机群的一部分：



我们曾经将注意力集中在普通用途的“工具”上：你不能看到、感觉到或听到软件。我们无法给你提供一张程序的图片，因此我们给你看一下运行它的“工具”。但是，很多软件直接处理“信息”。因此，让我们来考虑一下运行“普通软件”的“普通计算机”的“普通用途”。

一个“服务器机群”是提供 Web 服务的多台计算机的集合。通过使用 Web 搜索引擎，我们找到由维基百科(一个 Web 目录)提供的下列知识。在 2004 年，据估计搜索引擎的服务器机群是以下规模：

- 719 个机架
- 63272 台机器
- 126544 个 CPU
- 253THz 的处理能力
- 126544GB 的内存
- 5062TB 的硬盘空间

一个 GB 是 1G 字节，大约是 1000000000 个字符。一个 TB 是 1T 字节，等于 1000 GB，大约是 1000000000000 个字符。最近，这个“机群”变得更加庞大。这是一个相当极端的例子，但是每个

大公司都在 Web 上运行程序，并通过它与用户或消费者进行交互。更多的例子包括 Amazon(销售图书和其他商品)、Amadeus(航空票务和汽车租赁)和 eBay(在线拍卖)。数以百万计的小公司、组织和个人也存在于 Web 上。它们中的大多数并不运行自己的软件，但是也有很多在运行并且相当复杂。

其他更传统、更大规模的计算机应用主要涉及：会计、订单处理、发薪水、记录保存、账单、库存管理、个人记录、学生记录、病历等，基本上每个组织都需要保存记录(商业和非商业、政府和个人)。这些记录是每个组织的支柱。通过计算机处理这些记录看起来很简单：这些信息(记录)中的大多数只需要存储和检索，只有非常少的部分需要处理。这方面的例子主要包括：

- 12:30 飞往芝加哥的航班是否仍然准时？
- Gilbert Sullivan 是否曾经患过麻疹？
- Juan Valdez 订购的咖啡是否已经启运？
- Jack Sprat 在 1996 年购买的是哪种餐椅？
- 2006 年 8 月从 212 区号拨出电话的数量是多少？
- 1 月售出的咖啡壶数量和总价是多少？

规模庞大的数据库使得这些系统非常复杂。这样，就对响应时间(对每个查询的响应通常不超过 2 秒钟)和准确性(至少在大多数情况下)的需求。如今，人们谈论 T 字节的数据(一个字节等于用于存储一个普通字符的内存大小)已经很常见了。这就是传统的“数据处理”，它正在和“Web”相融合，这是由于当前多数的数据库访问都通过 Web 接口。

这种计算机应用通常称为信息处理。它将重点集中在数据上，通常是大量的数据。这就导致了在数据的组织和传输上的挑战，以及在怎样以可以理解的形式来表示大量数据的大量有趣的工作：“用户接口”是处理数据中的重要方面。例如，对古典文学(Chaucer 的《Canterbury Tales》或 Cervantes 的《Don Quixote》)的分析工作，通过比较几十个版本以找出哪个才是作者的实际创作。我们需要以分析人员提供的多种标准来搜索文本，并且以有助于发现要点的方式来显示结果。思考一下文本分析和出版：当前，几乎所有的文章、书籍、小册子、报纸等都通过计算机生产。设计出能够很好地支持这一切的软件，对大多数人仍是一个缺乏真正好的解决方案的问题。

### 1.5.6 一种垂直的视角

有人曾经宣布古生物学家可以重构一个完整的恐龙，并且通过研究一块小的骨骼来描述它的生活方式和自然环境。这有可能是一种夸张的说法，但是从中可以体会到通过观察一个简单的产品来思考它的含义的思想。我们考虑一下这张显示火星风景的照片，它由 NASA 的火星探测器携带的照相机所拍摄：



如果你希望研究“火箭科学”，那么成为好的程序员是一种方式。各种空间计划需要大量软件设计人员，特别是根据载人或非载人空间计划需要的懂得物理、数学、电子工程、机械工程、医疗工程

等的人员。人类已成功发射两颗探测器围绕火星运行四年多(估计的设计寿命是3个月)，这是人类文明最伟大的成就技术之一。

这张照片通过一条通信信道经过单向25分钟的传输延时传输到地球，这里需要很多巧妙的编程和高等数学应用，以便保证以最少的比特数、同时无差错地传输图片。在地球上，通过某些算法对这张照片进行渲染以恢复颜色和减小失真，这些问题都是由光学和电子传感器引起的。

火星探测器的控制程序当然也是程序，探测器每24小时会自动驱动一次，执行前一天从地球发送的指令。数据传输也是由程序来管理的。

探测器中的各种计算机使用的操作系统、传输和照片重构都是程序，在这点上和用来编写本章的各种应用程序相似。运行这些程序的计算机是通过CAD/CAM(计算机辅助设计和计算机辅助制造)程序来设计和生产的。这些计算机中的芯片是通过计算机化生产线用精密工具组装的，这些工具在它们的设计和制造中也使用计算机(或软件)。对这个长期组装过程的质量监控涉及大量计算。所有这些代码都由程序员用高级编程语言编写，并且通过编译器(本身就是一个程序)转换成机器代码。很多程序使用GUI与用户进行交互，以及使用输入/输出流进行数据交换。

最后，在图像处理(包括来自火星探测器的照片处理)、动画和照片编辑(在网络上有很多描述围绕“火星”的探测器照片)方面也需要大量编程工作。

### 1.5.7 与C++程序设计有何联系

这些“多样和复杂的”应用和软件系统与学习编程和使用C++有什么关系？这个关系很简单，很多程序员会参加这样的项目。这些事是好的程序设计可以帮助实现的。本章中用到的每个例子都涉及C++和本书中描述的几种技术。是的，在MP3播放器、船舶、风力发电机组、火星探测和人类基因工程中都会用到C++编程。如果想获得更多的使用C++的例子，你可以查看[www.research.att.com/~bs/applications.html](http://www.research.att.com/~bs/applications.html)。

## 1.6 程序员的理想

我们希望从自己的程序中获得什么？我们通常(而不是从特定程序的特定功能)希望获得什么？我们希望保证正确性和作为其中一部分的可靠性。如果程序没有按照设想工作，没有按我们可以信赖的方式工作，小则是一个严重干扰、大则是一个危险。我们希望它得到良好的设计，这样它可以很好地满足实际需要；如果它所做的事情与我们无关，或者以某种我们厌烦的方式完成，则不能说程序是正确的。我们同样希望可以负担得起；我可能喜欢用Rolls-Royce汽车或行政专机作为日常交通工具，但是除非我是一名亿万富翁，否则开销会影响我的选择。

这些方面是软件(工具、系统)能得到外部的、非程序员的赞赏的所在。如果我们希望开发出成功的软件，那么这些内容必须成为程序员的理想，我们必须将它们永远记在心中，特别是在程序开发的早期阶段。此外，我们必须关注与代码本身相关的理想：我们的代码必须是可维护的，那些没有编写它的人可以理解它的结构和进行修改。一个成功的程序可以“生存”很长时间(通常有几十年)并经过反复修改。例如，它将会移植到新的硬件上，它将会增加新的功能，它将会修改以使用新的I/O设备(屏幕、视频、音频)，它将会用新的自然语言进行交互等。只有失败的程序才永远不会被修改。为了保证可维护性，一个程序必须只与它的需求相关，它的代码必须直接体现要表达的思想。复杂性是简单性和可维护性的敌人，它对程序员来说可能是必需的(在那种情况下我们不得不处理它)，但是也可能是由于没有用代码清晰地表达出思想而产生。我们必须通过良好的编码风格来尽量避免它——风格是很重要的！

这听起来不太难，但是做起来确实很难。为什么？编程基本上是简单的：就是告诉机器你打算做什么。但是，为什么编程中要面对很多挑战？计算机基本上是简单的，它只能做很少几种操作，例如两个数相加和基于两个数的比较来选择要执行的下一条指令。问题是我们并不希望计算机做简单的事情。我们希望“机器”帮助我们做那些难以完成的事，但是计算机是挑剔的、无情的和不会说话的东西。另外，这个世界要比我们所相信的更复杂，因此我们并不能理解自己需求的实际含义。我们只是希望一个程序能够“像这样做一些事情”，但是我们并不希望被技术细节所困扰。我们通常假设“基本常识”。不幸的是，人们认为很普通的基本常识，在计算机中通常完全不存在（通过某些精心设计的程序，可以在具体的、很好理解的情况下模拟它）。

这种思路导致的想法是“编程就是理解”：当你需要编写一个任务时，你需要理解它。相反，当你彻底理解一个任务，你可以编写程序去执行它。换句话说，我们可以将编程看做努力去彻底理解一个课题的一部分。程序是我们对一个课题的理解的精确表示。

当你在进行编程时，你会花费很多时间尝试理解你试图自动化的任务。

我们可以将描述开发程序的过程分为四个阶段：

- 分析：问题是什么？用户想要做什么？用户需要什么？用户可以负担什么？我们需要哪种可靠性？
- 设计：我们如何解决问题？系统的整体结构将是怎样的？系统包括哪些部分？这些部分之间如何通信？系统与用户之间如何通信？
- 编程：用代码表达问题（或设计）求解的方法，以满足所有约束（时间、空间、金钱、可靠性等）的方式编写代码。保证这些代码是正确的和可维护的。
- 测试：系统化地尝试各种情况，保证系统在所要求的所有情况下都能正确工作。

编程和测试相加通常称为实现。很明显，将软件开发简单分为四个部分是一种简化。分别针对这四个主题编写的书都很厚，并且很多书仍在讨论它们之间的关系。需要说明的一件重要的事情是开发的这四个阶段并不是独立的，并且不一定严格按照顺序依次出现。我们通常从分析开始，但是通过测试的反馈有助于对编程的改进；编程工作带来的问题可能表明设计带来的问题；按设计进行工作可能发现在设计中至今仍被忽视的某些方面的问题。系统的实际使用通常会暴露分析中的一些弱点。

这里的关键概念是反馈。我们从经验中学习，根据学到的东西改变我们的行为。这是高效软件开发的根本。对于很多大的项目，我们在开始之前不可能理解有关问题的所有事情和解决方案。我们可以尝试自己的想法和从编程中得到反馈，但是在开发的早期阶段更容易（更快）从设计方案的书写、按设计思路编程和朋友的使用中得到反馈。我们知道的最好的设计工具是黑板。你要尽可能避免独立设计。在已将设计思路解释给其他人之前，不要开始进行编码工作。在接触键盘之前，与朋友、同事、潜在用户等讨论设计和编程技术。从简单尝试到阐明思路的过程中，你所学到的东西是令人惊讶的。最终，程序只不过是对某些思路的表达（用代码）。

同样，当你实现一个程序时遇到问题，将目光从键盘上移开。考虑一下问题本身，而不是你的不完整的方案。与别人交流：解释你希望做什么和为什么它不工作。令人惊讶的是，你向有些人详细解释问题的过程中经常会找到解决方案。除非不得已，否则不要单独进行调试（找到程序错误）！

本书的重点是实现，特别是编程。我们不讲授“解决问题”，以及提供有关问题的足够例子和它们的解决方案。很多问题的解决是认识到一个已知的问题，以及使用一个已知的解决方案。只有当大多数子问题以这种方式解决后，你才可能专注于令人兴奋的和有创造性的“跳出固有模式”。

的思维”。因此，我们重点介绍如何用代码明确表达思路。

用代码直接表达思路是编程的基本的理想。这确实是很明显，但是至今我们还缺少好的例子。我们将会反复回到这里。当我们需要在自己的代码中使用一个整数时，我们将它保存在一个 int 类型中，它会提供基本的整数操作。当我们需要使用一个字符串时，我们将它保存在一个 string 类型中，它会提供基本的文本控制操作。在最基本的层次上，理想是当我们有一个思路、概念和实体时，即那些我们可以作为“事情”考虑的、可以写在黑板上的、可以加入讨论的、(非计算机科学)教科书中讨论的东西，我们需要这些东西在程序中作为一个命名实体(类型)存在，并且提供我们需要它们执行的操作。如果我们需要进行数学计算，则需要一个复数的 complex 类型和一个线性代数的 Matrix 类型。如果我们需要进行图形处理，则需要一个 Shape 类型、一个 Circle 类型、一个 Color 类型和一个 Dialog\_box 类型。当我们处理来自比如说一个温度传感器的数据流时，我们需要一个 istream 类型(“i”表示输入)。很明显，每种类型将提供适当的操作，并且只提供适当的操作。这些只是本书中提到的几个例子。基于上述内容，我们提供用于构建自己的类型的工具和技术，以便用程序直接表达你希望体现的概念。

编程是实践和理论相结合的。如果你只重视实践，你将制造出不可扩展的、不可维护的程序。如果你只重视理论，你将制造出无法使用的(或无法负担的)玩具。

如果你想获得有关编程理想的不同类型的观点，以及少数在编程语言方面对软件做出重要贡献的人，请看第 22 章“理想和历史”。

## 思考题

思考题的目的是说明本章所解释的关键思想。可以把它们看做是练习的补充：练习关注的是编程的实践方面，而思考题尝试帮助你阐明思想和概念。在这方面，它们类似于好的面试题。

1. 什么是软件？
2. 软件为什么重要？
3. 软件重要在哪里？
4. 如果有些软件失败，那么导致错误的原因是什么？列举一些例子。
5. 软件在哪里扮演重要角色？列举一些例子。
6. 哪些工作与软件开发相关？列举一些例子。
7. 计算机科学和编程之间的区别是什么？
8. 在船舶的设计、建造和使用中，软件使用在哪些地方？
9. 什么是服务器机群？
10. 你在线提出哪种类型的查询？列举一些例子。
11. 软件在科学方面有哪些应用？列举一些例子。
12. 软件在医疗方面有哪些应用？列举一些例子。
13. 软件在娱乐方面有哪些应用？列举一些例子。
14. 我们期待中的好软件的一般特点有哪些？
15. 一个软件开发者看起来像什么？
16. 软件开发有哪些阶段？
17. 软件开发为什么困难？列举一些原因。
18. 软件的哪些用途为人类生活带来便利？
19. 软件的哪些用途为人类生活带来更多困难？

## 术语

这些术语是编程和 C++ 方面的基本词汇。如果你希望理解人们谈到的关于编程的主题和阐明自己的思

路，那么你应该知道每个术语的含义。

负担得起	客户	程序员	分析	设计	程序设计
黑板	反馈	软件	CAD/CAM	图形用户界面(GUI)	陈规旧习
沟通	理想	测试	正确性	实现	用户

## 习题

- 选择一种你最常做的活动(例如上学、吃饭或看电视)。列举计算机直接或间接参与其中的方式。
- 选择一种你最感兴趣或了解的职业。列举这些职业的人涉及计算机的活动。
- 将你从习题2中得到的列表与选择不同职业的朋友交换，并且改进他或她的列表。当你完成这个练习，比较你们的结果。记住，这种开放式的练习没有完美的解决方案，永远有可能需要改进。
- 根据你自己的经验，描述一种离开计算机不可能进行的活动。
- 列举你已经使用过的程序(软件应用)。列举那些你与程序有明显交互的例子(例如在一台MP3播放器中选择一首新歌)，而不是那些可能涉及计算机的例子(例如转动你的汽车方向盘)。
- 列举十种完全不会涉及(即使是间接的)计算机的人类活动。这可能会比你想象得更难。
- 列举五个当前没有使用计算机，但是你认为在将来会使用的任务。为你选择的每个任务写几句话加以说明。
- 解释(至少100字，但不超过500字)为什么你想成为一名计算机程序员。另一方面，如果你认为自己不想成为一名程序员，请解释原因。在这两种情况下，提供深思熟虑、合乎逻辑的论据。
- 解释(至少100字，但不超过500字)除了程序员之外，你希望在计算机工业中扮演的角色(“程序员”是否是你的首要选择)。
- 你认为计算机将会发展到有意识、有思想、有能力与人类竞争的程度吗？写一段支持你的观点的话(至少100字)。
- 列举最成功的程序员共有的特点。列举通常认为程序员应该具有的特点。
- 列举五种在本章中提到的对计算机程序的应用，并选择一种你最感兴趣和将来最想参加的应用。并解释为什么选择这种应用(至少100字)。
- 保存本页文字、本章、莎士比亚的所有著作各需要多大内存？假设1字节的内存可以保存一个字符，在这里只是尽量精确到20%。
- 你的计算机拥有多大的内存？主存储器多大？磁盘多大？

## 附言

我们的文明运行在软件之上。软件是一种有趣的、对社会有益的、有利可图的工作，它是具有无与伦比的多样性和机会的领域。当你接触软件时，以有原则和严肃的方式工作：你要成为解决方案的一部分，而不是增加问题。

我们对贯穿技术文明的各种软件很敬畏。当然，软件并不是在所有应用中都表现良好，但那是另外一回事。在这里，我们想强调的是软件是多么普遍，以及我们在日常生活中多么依赖软件。它们都是由像我们这样的人来编写的。所有的科学家、数学家、工程师、程序员等，他们构建这里简略提到的软件的起点和你是一样的。

现在，让我们回到脚踏实地的业务上，学习那些编程需要的技术性的技能。如果你开始怀疑自己艰苦工作是否值得(大多数深思熟虑的人有时会怀疑它)，你可以返回并重新阅读本章、前言和第0章。如果你开始怀疑自己是否能掌握这一切，请记住已经有数百万人成为称职的程序员、设计师、软件工程师等。你也可以做到。

# 第一部分 基本知识

## 第2章 Hello, World!

“程序设计要通过编写程序的实践来学习”

——Brian Kernighan

在本章中，我们提出最简单的 C++ 程序，它们实际上可以做任何事。编写这些程序的目的如下：

- 让你试用自己的编程环境。
- 给你一个最初的体验：如何让计算机为你做某些事。

因此，我们提出程序的概念，使用编译器将程序从人类可读的形式转换到机器指令，以及最终执行机器指令的思想。

### 2.1 程序

为了使计算机能够做某件事，你(或其他人)需要在繁琐的细节上明确告诉它怎么做。对“怎么做”的描述称为程序，编程是书写和测试这个程序的行为。

在某种意义上，我们以前都编写过程序。毕竟，我们曾描述过所要完成的任务，例如“如何开车去最近的电影院”、“如何找到楼上的浴室”和“如何用微波炉热饭”。这种描述和程序之间的不同表现在精确度上：人类往往通过常识对不明确的指示加以补偿，但是计算机不会这样。例如，“沿走廊右转，上楼，它就在你的左边”可能是对如何找到楼上的浴室的很好描述。但是，当你看到这些简单的指令时，你会在其中找到草率的语法和不完整的指令。人类很容易做出补偿。例如，假设你坐在桌子旁询问浴室的方向。你不需要被告知离开桌子来到走廊、绕过(不是跨过或钻过)桌子、不要踩到猫等。你不需要被告知不要带走刀子和叉子，以及记住打开灯才能看到楼梯。你也不需要被告知进入浴室之前首先要开门。

与此相反，计算机确实是不很笨的。它们做的所有事都要准确、详细地描述。我们考虑“沿走廊右转，上楼，它就在你的左边”。走廊在哪里？什么是走廊？什么是“右转”？什么是楼梯？我如何上楼梯？(每次迈出一步？两步？沿扶手滑上楼梯？)什么是我的左边？它什么时候会在我的左边？为了向计算机精确描述这些“事情”，我们需要一种由特定语法精确定义的语言(英语对它来说有太多的松散结构)和针对我们要执行的多种行动明确定义的词汇。这种语言称为编程语言，C++ 是为各种编程任务而设计的编程语言。

如果你想知道有关计算机、程序和编程的更多哲学上的细节，请阅读第 1 章。在这里，让我

们来看一些代码，从一个很简单的程序和运行它的工具和技术开始学习。

## 2.2 经典的第一个程序

这是经典的第一个程序的一个版本。它在你的屏幕上输出“Hello, World!”：

```
// This program outputs the message "Hello, World!" to the monitor
```

```
#include "std_lib_facilities.h"

int main() // C++ programs start by executing the function main
{
    cout << "Hello, World!\n"; // output "Hello, World!"
    return 0;
}
```

我们可以将这段文字看做是交给计算机执行的一组指令，就像我们交给一个厨师的一张菜谱，或我们用于使一个新玩具工作的一组指令集合。我们从这下面行开始讨论这个程序的每行如何工作。

```
cout << "Hello, World!\n"; // output "Hello, World!"
```

这是实际生成输出内容的一行。它打印字符串 Hello, World!，并且紧跟换一个新行。也就是说，在打印出 Hello, World! 之后，光标将位于下一行的开始位置。光标是一个小的、闪烁的字符或行，它用来显示你可以输入下一个字符的位置。

在 C++ 中，字符串常量是由双引号 (" ) 来分隔。也就是说，“Hello, World! \n”是一个字符串。 \n 是一个用于指定新行的“特殊字符”。名称 cout 是一个标准的输出流。使用输出操作符 << “放入 cout”的字符将显示在屏幕上。名称 cout 的发音是“see-out”，它是“character output stream”的缩写。你会发现在编程时缩写很常见。很自然，在你第一次看到和要记住一个缩写时会觉得有点儿烦，但是当你开始重复使用一个缩写时，它们将会变得很自然，并且对保持程序文本的简短和可控制是必不可少的。

这行的结尾

```
// output "Hello, World!"
```

是一个注释。在一行中的//符号(/ 符号称为“斜杠”，这里是两个斜杠)之后的内容都是注释。注释会被编译器忽略，但对人们读懂代码很有帮助。在这里，我们使用注释告诉你这一行的开始部分实际在做什么。

注释用于描述这个程序打算做的事情，它通常提供的是对人们有用但是不能用代码直接来表达的信息。当你在下一星期或下一年回过头来阅读代码，并且已经忘记为什么以这种方式编写代码时，你最有可能通过代码中的注释得到帮助。因此，做好你的程序的文档工作。在 7.6.4 节，我们将讨论如何做好注释。

程序是为两个读者编写的。理所当然，我们编写代码是为了在计算机中执行。然而，我们阅读和修改代码也花费很长时间。于是，程序员是程序的另一个读者。因此，编写代码也是人与人之间沟通的一种方式。实际上，考虑将人类作为我们的代码的主要读者是有意义的：如果他们(我们)发现代码不是那么容易理解，那么代码永远不可能变得正确。总而言之，注释只是对人类读者有帮助的，计算机并不会看注释中的文本。

这个程序中的第一行是一个典型的注释，它简单地告诉人类读者程序打算做什么：

```
// This program outputs the message "Hello, World!" to the monitor
```

由于这段代码本身说明了程序做什么，而不是我们想让它做什么，因此这个注释是有用的。我们通常向人类(粗略地)解释一个程序将做什么，比我们用代码向计算机(详细)表达它要简单得多。这种

注释通常是我们编写的程序的第一部分。如果没有其他内容，它会提醒我们正在尝试做什么。

下一行

```
#include "std_lib_facilities.h"
```

是一个“#include 指令”。它指示计算机从名为 std\_lib\_facilities.h 的文件中提供(“包含”)功能。我们编写这个文件以简化使用所有 C++ 实现(“C++ 标准库”)中的功能。我们将随着学习的深入解释它的内容。它完全是普通的标准 C++ 程序，但是它包含我们不得不介绍的细节，在后续章中将会介绍它们。对于这个程序来说，std\_lib\_facilities.h 的重要性表现在我们可以使用标准 C++ 流 I/O 功能。在这里，我们只使用标准输出流 cout 和它的输出操作符 <<。使用#include 包含的文件通常有后缀 .h，称为头或头文件。在头文件中包含术语的定义，例如在我们的程序中使用的 cout。

一台计算机如何知道从哪里开始执行一个程序？它会查找一个称为 main 的函数，并且在那里开始执行找到的指令。下面是“Hello, World!”程序的 main 函数：

```
int main() // C++ programs start by executing the function main
{
    cout << "Hello, World!\n"; // output "Hello, World!"
    return 0;
}
```

每个 C++ 程序必须有一个称为 main 的函数，以便告诉计算机从哪里开始执行。一个函数基本上是一个命名过的指令序列，计算机会按照它们的编写顺序来执行。一个函数包括 4 个组成部分：

- 返回值类型，在这里是 int(表示“整数”)，它用来指定返回结果的类型。如果有的话，这个函数会将值返回给要求它执行的程序。单词 int 在 C++ 中是保留字(一个关键字)，因此 int 不能用于作为其他任何东西的名字(见 A. 3.1 节)。
- 名字，在这里是 main。
- 参数列表，封闭在一对括号中(见 8.2 节和 8.6 节)，在这里是()，在这种情况下，参数列表是空的。
- 函数体，封闭在一对大括号中，在这里是{}中，列出了这个函数将要执行的动作(称为语句)。

下面是最简单的 C++ 程序

```
int main() {}
```

由于这个程序没有做任何事情，因此它并没有什么用处。我们的“Hello, World!”程序的 main() (“主函数”) 体中有两条语句：

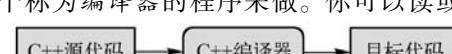
```
cout << "Hello, World!\n"; // output "Hello, World!"
return 0;
```

首先，它在屏幕上书写 Hello, World!，然后返回一个值 0(零)给它的调用者。由于 main() 是由“系统”来调用的，因此我们不会使用返回值。但是，在有些系统(特别是 UNIX/Linux)中，返回值可以用于检查程序是否成功。由 main() 返回的一个零(0)表示程序成功终止。

在 C++ 程序中用于指定一个行为并且不是一个#include 指令(或其他预处理器指令，见 4.4 节和 A. 17 节)的部分称为语句。

## 2.3 编译

C++ 是一种编译语言。这意味着要想使一个程序可以运行，你首先必须将它从人类可读的格式转换为机器可以“理解”的东西。这个转换过程由一个称为编译器的程序来做。你可以读或写的称为源代码或程序文本，计算机可以执行的称为可执行代码、目标代码或机器代码。典型的 C++ 源代



码文件的后缀为 .cpp(例如 hello\_world.cpp)或 .h(例如 std\_lib\_facilities.h)，目标代码文件的后缀为 .obj(在 Windows 中)或 .o(在 UNIX 中)。代码一词是模棱两可的并且会引起混淆，注意只有在可以明确表达含义时才使用它。除非特别说明，否则我们用代码来表示“源代码”或“不包含注释的源代码”，这是由于注释只是供人类阅读的，在编译器生成目标代码时不会看到它。

编译器会阅读你的源代码，并且尽力理解你所写的内容。编译器会检查你的程序在语法上是否正确，每个单词是否已明确定义，在程序中是否有不必实际执行就可以检测到的明显错误。你会发现编译器在语法上相当挑剔。忽略程序中的有些细节(例如#include 文件、分号或大括号)将会引起错误。与此类似，编译器绝对不会容忍拼写错误。我们将通过一系列例子来解释这些，在每个例子中有一个小错误。每个错误是我们经常犯的一类错误的例子：

```
// no #include here
int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

我们没有包括任何文件以告诉编译器 cout 是什么，因此编译器会抱怨。为了纠正这个错误，增加一个头文件：

```
#include "std_facilities.h"
int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

不幸的是，编译器再次抱怨：我们拼写错了 std\_lib\_facilities.h。编译器也不支持这样：

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

我们没有用一个" 来终止字符串。编译器也不支持这样：

```
#include "std_lib_facilities.h"
integer main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

在 C++ 中使用缩写 int 而不是单词 integer。编译器也不支持这样：

```
#include "std_lib_facilities.h"
int main()
{
    cout < "Hello, World!\n";
    return 0;
}
```

我们使用 <(小于操作符)而不是 <<(输出操作符)。编译器也不支持这样：

```
#include "std_lib_facilities.h"
int main()
{
    cout << 'Hello, World!\n';
    return 0;
}
```

我们使用单引号而不是双引号来限制字符串。最后，编译器发现这样的错误：

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Hello, World!\n"
    return 0;
}
```

我们忘记使用分号来终止输出语句。需要注意的是，很多 C++ 语句是用一个分号(;)来终止的。编译器通过这些分号来识别一个语句在哪里终止，以及另一个语句从哪里开始。这里没有简短的、完全正确的、非技术方式的有关哪里需要分号的总结。现在，我们只是复制自己的应用模式，它可以归纳为：在每个没有由右侧大括号({})表示结束的表达式后面放置一个分号。

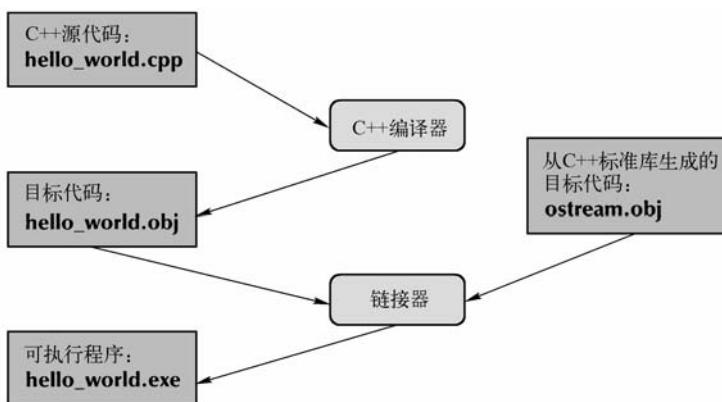
为什么我们要花费篇幅和你宝贵的时间给你看这些带有琐碎错误的小程序的例子呢？像所有的程序员一样，你会花费大量时间在程序源文本中查找错误。在大多数时候，我们看到的是含有错误的文本。毕竟，如果我们确信一些代码是正确的，那么我们通常会看其他代码或将时间用在别的地方。这令早期的计算机先驱们非常惊讶，他们曾经不得不花费自己的绝大部分时间来发现编程时产生的错误。这也令大多数的编程新手感到惊讶。

当你编程时，有时会对编译器感到相当懊恼。有时候，编译器会抱怨无关紧要的细节（例如缺少一个分号），或者一些你认为“明显正确”的东西。但是，编译器通常是正确的：当它给出一个错误消息并拒绝为你的源代码生成目标代码时，在你的程序中确实有不正确的地方，这意味着你写的程序不符合 C++ 标准的精确定义。

编译器没有常识（它不是人类），它对细节是非常挑剔的。由于编译器没有常识，因此你不能让它尝试着去猜测那些“看起来正确”但是不符合 C++ 定义的代码所表达的意思。如果你这样做并且编译器的猜测与你不同，这时你需要花费很多时间找出为什么程序没有按你的要求工作。当所有的事都说清和做到后，编译器会帮你从大量自己造成的问题中解脱出来。编译器将我们从问题中拯救出来，而不是引起问题。因此，请大家记住，编译器是你的朋友，编译器可能是你在编程时最好的朋友。

## 2.4 链接

程序通常由几个单独的部分组成，它们经常由不同的人来开发。例如，“Hello, World!”程序包含我们编写的部分和 C++ 标准类库。这些单独的部分（有时称为翻译单元）必须被编译，其目标代码必须被链接起来以形成一个可执行程序。用于将这些部分链接起来的程序通常称为链接器。



请注意目标代码和可执行程序是不能在系统之间移植的。例如，当你为一台 Windows 机器进行编译时，你得到的支持 Windows 的目标代码无法在 Linux 机器上运行。

库是一些代码的集合，它们通常是由其他人编写的，我们用#include 文件中的声明来访问这些代码。声明用于指出一段程序如何使用一条语句，我们将在后面的章节(如 4.5.2 节)中详细介绍声明。

由编译器发现的错误称为编译时错误，由链接器发现的错误称为链接时错误，直到程序运行时仍未发现的错误称为运行时错误或逻辑错误。通常来说，编译时错误比链接时错误更容易理解和修正，链接时错误比运行时错误和逻辑错误更容易发现和修正。在第 5 章中，我们将详细讨论这些错误和它们的解决方式。

## 2.5 编程环境

我们使用编程语言来编写程序。我们使用编译器将自己的源代码转换成目标代码，使用链接器将我们的目标代码链接成一个可执行程序。另外，我们使用一些程序在计算机中输入源代码文本并且编辑它。这些是最初的和最重要的工具，它们构成程序员的工具集合或“程序开发环境”。

如果你使用的是命令行窗口，就像很多专业程序员所做的那样，你将不得不自己来编写编译和链接命令。如果你使用 IDE(“交互式开发环境”或“集成式开发环境”)，就像很多程序员所做的那样，简单地点击正确按钮就可以完成这个工作。附录 C 介绍了如何在你的 C++ 实现中编译和链接。

IDE 通常包括一个具有有用特性的编辑器，例如用不同颜色的代码来区分你的源代码中的注释、关键字和其他部分，以及其他帮助你来调试代码、编译和运行代码的功能。调试是发现程序中的错误和排除错误的活动，你在前进的道路上会听到很多有关它的内容。

在本书中，我们使用微软的 Visual C++ 作为编程开发环境实例。如果我们简单地说“编译器”或是“IDE”的某些部分，那就是所指 Visual C++ 系统。但是，你可以使用一些提供最新的、符合标准的 C++ 实现的系统。我们所说的大多数内容(经过微小的修改)对所有的 C++ 实现都将是正确的，并且其代码可以在任何地方运行。在工作中，我们使用几种不同的实现。

### 简单练习

迄今为止，我们讨论了很多有关编程、代码和工具(例如编译器)的内容。现在，你可以运行一个程序。这是本书中和学习编程过程中的一个重点。这是你培养实践技能和好的编程习惯的开始。本章练习的重点在于使你熟悉软件开发环境。当你运行“Hello, World!”程序时，你将通过成为程序员的第一个重要的里程碑。

这个简单练习的目的是建立或加强你的实际编程技能，以及为你提供编程环境工具方面的经验。典型的简单练习是对一个独立程序的一系列修改，将琐碎的程序“发展”成一个实际程序的有用的部分。一套传统的练习是用于测试你的主动性、灵活性或创造性的。与此相反，简单练习很少需要你发挥创造力。典型的情况是顺序很关键，每个单独的步骤可能是容易(或琐碎)的。请不要自认为聪明地试图跳过某些步骤，这样通常会减慢你的进展或使你感到迷惑。

你可能会认为自己已经理解了阅读过的和导师或辅导员告诉你的任何事，但是重复和实践对提高编程能力是很必要的。在这方面，编程和体育、音乐、舞蹈或任何基于技能的行业是相似的。请想象人们试图经过常规练习就能在这类领域中参与竞争，你知道结果会如何。坚持练习对专业人员意味着终身的长期练习，是发展和维持一种高水平的实用技能的唯一方式。

因此，不要跳过这个简单练习，即使你多么想跳过去，它们从本质上来说是学习的过程。现在，从第一步开始并继续下去，测试每个步骤以确保你做得正确。

如果你不理解所使用的语法的细节也不必担忧，不要害怕向辅导员或朋友寻求帮助。坚持完成所有的简单练习和部分的习题，所有一切都会在适当的时候变得清晰。

那么，这是你的第一个简单练习：

1. 查看附录 C 并按照这些步骤的要求建立一个工程。建立一个名为 hello\_world 的空白的 C++ 工程控制台。
2. 输入 hello\_world.cpp，完全按照下面的要求，将它保存在你的练习目录中，并将它包含在你的 hello\_world 工程中。

```
#include "std_lib_facilities.h"
int main() // C++ programs start by executing the function main
{
    cout << "Hello, World!\n"; // output "Hello, World!"
    keep_window_open(); // wait for a character to be entered
    return 0;
}
```

在一些 Windows 机器中需要调用 `keep_window_open()`，以防止在你在有机会阅读输出之前窗口被关闭。这是 Windows 的一个特点，而不是 C++ 的。我们在 `std_lib_facilities.h` 中定义 `keep_window_open()`，以便简化简单文本程序的编写。

你如何找到 `std_lib_facilities.h`? 如果你在上课，你可以问辅导员。否则，你可以从我们的支持站点 [www.stroustrup.com/Programming](http://www.stroustrup.com/Programming) 下载它。但是，如果你既没有辅导员又无法访问网站? (只有)在这种情况下，用下列语句代替`#include`语句：

```
#include<iostream>
#include<string>
#include<vector>
#include<algorithm>
#include<cmath>
using namespace std;
inline void keep_window_open() { char ch; cin>>ch; }
```

这里直接使用了标准库，它将会跟随你直到第 5 章，并将在 8.7 节中详细解释。

3. 编译和运行“Hello, World!”程序。有些地方很可能不会正常工作。很少有人在初次尝试时使用一种新的编程语言或一个新的编程环境时就能成功。找到问题并修改它! 这里的关键是向一个有经验的人寻求帮助，但是你要确定能够理解你看到的东西，这样你在进一步处理之前可以完全自己来做。
4. 现在，你可能遇到一些错误并不得不纠正它。现在是更好地熟悉编译器的错误发现和错误报告功能的时候了! 尝试来自 2.3 节的 6 个错误，以便查看编程环境对它们的反应。考虑至少 5 个可能发生的错误，它们是你在输入程序时造成的(例如忘记 `keep_window_open()`，在输入单词时按下 Caps Lock 键，或者将分号输入成逗号)，并查看在编译和运行每种错误时会发生什么?

## 思考题

这些思考题的基本思想是给你一个机会，以查看你是否注意到和理解了本章的关键点。在回答问题时你可能需要返回正文中，这是正常的和可以预料的。你可能需要重新阅读整个章节，这也是正常的和可以预料的。但是，如果你需要重新阅读整个章节或对每道思考题都有问题，那么你可能需要考虑自己的学习方式是否有效。你是否阅读得过快? 你可能要停下来并做一下“试一试”吗? 你会和朋友一起学习以讨论正文解释方面的问题吗?

1. “Hello, World!”程序的目的是什么?
2. 函数的 4 个部分的名字。
3. 函数命名必须出现在每个 C++ 程序中。
4. 在“Hello, World!”程序中，`return 0;` 这行的目的是什么?
5. 编译器的目的是什么?
6. `#include` 语句的目的是什么?
7. 文件名后缀为 .h 在 C++ 中表示什么?
8. 链接器为你的程序做什么?

9. 源文件和对象文件之间的区别是什么？
10. IDE 是什么以及它能为你做什么？
11. 如果你理解教材中的所有内容，为什么练习还是必要的？

大多数的思考题在它们出现的章节中有明确的回答。但是，我们偶尔会包含问题以提醒你在其他章节中有相关的信息，甚至有些内容可能不在本书范围内。我们认为这是正常的，更为重要的是编写好的软件和思考这样做的含义，而不是更适合于作为独立的章节或书籍出现。

## 术语

这些术语是编程和 C++ 方面的基本词汇。如果你希望理解人们谈到的关于编程的主题和阐明自己的思路，你应该知道每个术语的含义。

//	可执行程序	main()	<<	函数	目标代码
C++	头文件	输出	注释	集成开发环境(IDE)	程序
编译器	#include	源代码	编译时错误	库	语句
cout	链接器				

你也可以以自己的语言逐步发展出一个词汇表。你可以通过重复每章后面的练习 5 来完成它。

## 习题

我们将简单练习与习题分别列出，在尝试做习题之前，请先完成本章中的简单练习。这样做将会节约你的时间。

1. 修改程序以输出下面 2 行

**Hello, programming!  
Here we go!**

2. 扩展你曾经学习的知识，编写程序列出使计算机找到楼上浴室的指令（在 2.1 节中讨论）。你能讨论更多的人类可以假设而计算机不会的步骤吗？将它们加入你的列表。这是一个“像计算机一样思考”的好的开始。注意，对于大多数人来说，“去浴室”是一个完全充分的指令。对于那些对房子或浴室没有任何经验的人（想象一个石器时代的人被传送到你的餐厅），这个包含必要指令的列表可能会很长。请不要使用超过一页的指令。为了便于读者们阅读，你可以增加一个关于你所想象的房子布局的简短描述。
3. 编写一个有关如何从你的宿舍、公寓、房屋等的前门到你的教室（假设你在参观某个学校，如果你无法想象，选择其他目的地）前门的描述。请一个朋友尝试按照这些指令走，并且对他或她改进的路线加以解释。为了保持朋友关系，在将这些指令交给一个朋友之前进行“实地测试”是一个好的主意。
4. 找到一本好的菜谱。阅读有关烤制蓝莓松饼的指令（如果你在一个“蓝莓松饼”是一种陌生的、异国食品的国家，你可以用自己更熟悉的食品来代替）。请注意，在得到一点帮助和指令的情况下，这个世界上的大多数人都可以烤出美味的蓝莓松饼。这里不需要考虑高级的或困难的精细食品。但是，对于作者来说，本书中很少有习题像这个这样困难。只是通过一点儿练习，你所能做的事令人吃惊。

- 重新写这些指令使每个单独的动作位于它们自己编号的段落中。认真列出每个步骤使用的所有原料和厨房用具。注意关键的细节，例如理想的烤箱温度、预热烤箱、准备烘烤的薄饼、烹调技术的方式，以及将松饼从烤箱中取出时的注意事项。
  - 从一个烹调初学者（如果你不是，请你认识的不会烹调的朋友帮忙）的角度来考虑这些指令。填写菜谱作者（几乎可以肯定是一个有烹调经验的人）漏掉的很明显的步骤。
  - 建立一个包含使用过的术语的词汇表。（松饼平底锅是什么？如何预热？你认为“烤箱”意味着什么？）
  - 现在，烤制一些松饼和享用你的成果。
5. 为“术语”中的每个术语书写一个定义。首先尽力看你是否不看本章就可以做到，然后浏览本章以找到这些定义。你会发现在你初次尝试定义和看完本书之后的定义之间有趣的区别。你可以参考一些合适的在线词汇表，例如 [www.research.att.com/~bs/glossary.html](http://www.research.att.com/~bs/glossary.html)。通过在查找之前书写自己的定义，你会加强自己从学习中获得的知识。如果你需要重新读某个部分以形成一个定义，这也可以帮助你来理解它。你可以自由使用自己的词汇来进行定义，并且按照你认为合理的细节来完成定义。在通常情况下，主要定义

后面跟着一个例子将是有帮助的。你可以将这些定义存储在一个文件中，这样你可以通过增加它们形成后面章节的“术语”部分。

## 附言

“Hello, World!”程序有多么重要？它的目的是使我们熟悉基本的编程工具。当接触一个新的工具时，我们通常做一个非常简单的例子，例如“Hello, World!”。在这种方式下，我们将自己所学的东西分为两个部分：首先，我们通过简单程序学习有关工具的基础知识，然后，我们在没有被工具影响的情况下学习更复杂的程序。同时学习工具和语言的难度比先学一种后学另一种要大得多。这种将复杂任务分解成一系列小的（更容易管理的）步骤的学习方式，并不仅仅局限于编程和计算机方面。它在生活中的大多数领域是普遍的和有用的，特别是在那些需要实用技能的领域。

# 第3章 对象、类型和值

“幸运只青睐有准备的人。”

——Louis Pasteur

本章介绍程序中的数据存储和使用的基础知识。这样，我们首先关注从键盘读取数据。在建立起对象、类型、数值和变量的基本概念之后，我们介绍几种操作符，并且给出有关 char、int、double 和 string 类型的变量使用的例子。

## 3.1 输入

“Hello, World!”只是打印到屏幕。它产生输出，不读取任何内容，也就是不从用户那里得到输入。这令人相当厌烦。实际的程序通常基于我们给它的输入产生结果，而不是当我们每次执行它们时都做相同的事。

为了读取数据，我们需要将其读入某个地方，即我们需要在计算机内存中的某个地方放置读取的内容。我们将这样一个“地方”称为一个对象。一个对象是一个某种类型的内存区域，类型指定了可以放置什么样的信息。一个有名字的对象称为一个变量。例如，字符存放在 string 变量中，整数存放在 int 变量中。你可以将对象看成一个“盒子”，你可以在其中放置该对象类型的数值，如右图所示。

上图表示一个名为 age 的 int 类型的对象，其中保存的是整数值 42。通过使用一个字符变量，我们可以从输入中读取一个字符，然后将它打印出来，具体如下：

```
// read and write a first name
#include "std_lib_facilities.h"

int main()
{
    cout << "Please enter your first name (followed by 'enter'):\n";
    string first_name; // first_name is a variable of type string
    cin >> first_name; // read characters into first_name
    cout << "Hello, " << first_name << "!\n";
}
```

#include 与 main() 与第 2 章中的内容相似。由于我们的所有程序（直到第 12 章）都需要#include，我们将不再介绍它以避免引起你分心。同样，我们有时需要将代码放入 main() 或其他函数中才能工作，就像下面这样：

```
cout << "Please enter your first name (followed by 'enter'):\n";
```

我们假设你可以理解如何将这个代码加入一个完整的程序以便测试。

main() 中的第一行简单地输出一个信息，以便鼓励用户输入一个名字。这个信息通常称为提示符，这是由于它提示用户完成某个操作。下一行定义了一个名为 first\_name 的 string 变量，接下来的程序读取键盘输入存入变量，然后输出一个欢迎词。接下来，我们逐行分析三行：

```
string first_name; // first_name is a variable of type string
```

本行会划分一个可以保存一个字符串的内存区域，并将它命名为 first\_name，如右图所示。

string:

first\_name:

定义是用于将一个新的名字加入一个程序，并为一个变量分配内存空间的语句。

下一行将(键盘)输入的字符串读取到变量：

```
cin >> first_name; // read characters into first_name
```

名字 cin 是由标准库定义的标准输入流(读为“see-in”，“character input”的缩写)。操作符 >> 的第二个操作指定对象输入到哪里。因此，如果我们输入名字(例如 Nicholas)，后接一个换行，字符串"Nicholas"将会变成 first\_name 的值，如右图所示。新行是必要的以引起计算机的注意。

string:

Nicholas

计算机简单地收集输入的字符，直到输入一个新行(按下回车键)。这段“延迟”给你改变主意的机会，在按回车键之前可以删除或修改某些字符。这个新行不会成为保存在内存中的字符串的一部分。

在将输入的字符串放入 first\_name 之后，我们就可以使用它了：

```
cout << "Hello, " << first_name << "\n";
```

本行会在屏幕上打印 Hello，接着是 Nicholas(first\_name 的值)，接着是! 和一个新行('\'\n')：

**Hello, Nicholas!**

如果我们喜欢重复和额外的输入，我们可以用三个单独的输出语句来代替：

```
cout << "Hello, ";  
cout << first_name;  
cout << "\n";
```

但是，我们不是打字员，更重要的是非常不喜欢不必要的重复(由于重复为出错提供了机会)，因此我们将三个输出语句合并为一个语句。

注意，我们在"Hello," 处使用的是引号，而在 first\_name 处没有这样做。我们希望输出字符串常量时使用引号。当我们不使用引号时，我们希望输出的是名字中的值。考虑一下：

```
cout << "first_name" << " is " << first_name;
```

在这里，"first\_name" 输出的是十个字符 first\_name，而 first\_name 输出的是变量 first\_name 中的值，在这种情况下是 Nicholas。因此，我们得到：

**first\_name is Nicholas**

## 3.2 变量

如果没有存储在内存中的数据，我们基本上不能用计算机做任何有趣的事，正如上面的例子中所说的字符串输入。我们用来存储数据的“位置”称为对象。我们需要使用一个名字来访问一个对象。一个命名后的对象称为变量，它有特定的类型(例如 int 或 string)，类型决定我们将什么赋给对象(例如，123 可以赋给 int 型和"Hello, World! \n"可以赋给 string 型)，以及可以使用的操作(例如，我们可以对多个 int 型使用 \* 操作符进行乘法运算，对多个 string 型使用 <= 操作符进行比较)。我们赋给变量的数据项称为值。一个用来定义变量的语句(通常)称为定义，一个定义可以(通常会)提供一个初始值。考虑一下：

```
string name = "Annemarie";  
int number_of_steps = 39;
```

我们可以像这样来可视化这些变量：

int: number_of_steps: <span style="border: 1px solid black; padding: 2px;">39</span>	string: name: <span style="border: 1px solid black; padding: 2px;">Annemarie</span>
---	--

我们不能将类型错误的值赋给一个变量：

```
string name2 = 39;           // error: 39 isn't a string
int number_of_steps = "Annemarie"; // error: "Annemarie" is not an int
```

编译器将会记录每个变量的类型，并确认你对它的使用是否与它的类型一致，就像你在它的定义中所指定的那样。

C++ 提供了相当多的类型(参见 A.8 节)。但是，你只使用其中五种类型，就完全可以写出下面的好程序：

```
int number_of_steps = 39;      // int for integers
double flying_time = 3.5;     // double for floating-point numbers
char decimal_point = '.';     // char for individual characters
string name = "Annemarie";    // string for character strings
bool tap_on = true;           // bool for logical variables
```

名字 double 的原因是历史的：double 是“双精度浮点”的简称。浮点是数学上实数概念在计算机中的近似。

注意，每种类型的文字常量都有自己的特殊风格：

```
39          // int: an integer
3.5         // double: a floating-point number
'.'         // char: an individual character enclosed in single quotes
"Annemarie" // string: a sequence of characters delimited by double quotes
true        // bool: either true or false
```

一串数字(例如 1234、2 或 976)表示一个整数，在单引号中的一个字符(例如 '1'、'@'或 'x')表示一个字符，带小数点的一串数字(例如 1.234、0.12 或 .98)表示一个浮点值，在双引号中的一串字符(例如"1234"、"Howdy!"或"Annemarie")表示一个字符串。如果要获得文字常量的细节描述，参见 A.2 节。

### 3.3 输入和类型

输入操作 `>>` (“get from”)是对类型敏感的，它读取的值与变量类型需要一致。例如：

```
// read name and age
int main()
{
    cout << "Please enter your first name and age\n";
    string first_name; // string variable
    int age;           // integer variable
    cin >> first_name; // read a string
    cin >> age;        // read an integer
    cout << "Hello, " << first_name << " (age " << age << ")\n";
}
```

因此，如果你输入 Carlos 22，`>>` 操作符将 Carlos 读入 `first_name`，将 22 读入 `age`，并且生成这个输出：

**Hello, Carlos (age 22)**

为什么它不将 Carlos 22(全部)读入 `first_name`? 这是由于按照规定，字符串的读取会被空白符所终止，包括空格、换行和 tab 字符。否则，空白符在默认情况下会被 `>>` 忽略。例如，你可以在读取的数字之前添加任意多的空格，`>>` 将会跳过它们并读取这个数字。

如果你输入 22 Carlos，你将看到奇怪的东西，直到你能够理解这一切。22 将会读入 `first_name`，这是由于 22 毕竟是一个字符串。另一方面，Carlos 并不是一个整数，因此它不会被读取。这时的输出将是 22 和一些随机数，例如 -96739 或 0。为什么? 因为你没有给 `age` 赋一个初始值，并且没能成功地读取一个值存入它。因此，当你开始执行时，就会得到内存中的某个部分的“垃圾值”。在 10.6 节中，我们讨论“输入格式错误”的处理方式。现在，我们只是初始化 `age`，这样在输入错误时，我们会获得一个可预测的值：

```
// read name and age (2nd version)
int main()
{
    cout << "Please enter your first_name and age\n";
    string first_name = "???"; // string variable
                                // ("???" means "don't know the name")
    int age = -1; // integer variable (-1 means "don't know the age")
    cin >> first_name >> age; // read a string followed by an integer
    cout << "Hello, " << first_name << " (age " << age << ")\n";
}
```

现在，输入 22 Carlos 将会输出：

```
Hello, 22 (age -1)
```

注意，我们可以在一个输入语句中读取几个值，就像我们可以在一个输出语句中写入几个值一样。注意，`<<` 和 `>>` 都是对类型敏感的，因此我们可以输出 `int` 型变量 `age` 和字符文字 `'\n'`，以及 `string` 型变量 `first_name` 和字符串文字 `"Hello, " "(age" 和 ")" \n"`。

使用 `>>` 读取的 `String`(默认情况下)会被空格所终止，也就是说，它只能读取一个字。但是，我们有时需要读取多个字。当然会有多种方法来解决这个问题。例如，我们可以像这样来读取一个包括两个字的名字：

```
int main()
{
    cout << "Please enter your first and second names\n";
    string first;
    string second;
    cin >> first >> second; // read two strings
    cout << "Hello, " << first << ' ' << second << '\n';
}
```

我们简单地使用 `>>` 两次，每次针对一个名字。当我们想输出多个名字时，我们必须在它们之间插入一个空白符。

**试一试** 运行这个“名字和年龄”的例子。然后，修改它以月份的形式输出年龄：读取输入的年龄并(使用 `*` 操作符)乘以 12。将年龄读入一个 `double` 型的变量，使 5.5 岁的孩子骄傲于他比 5 岁的孩子大。

### 3.4 运算和运算符

除了指定什么值可以存储在一个变量中之外，变量类型还决定了我可以对它进行什么操作和它们表示什么。例如：

```
int count;
cin >> count; // >> reads an integer into count
string name;
cin >> name; // >> reads a string into name

int c2 = count+2; // + adds integers
string s2 = name + " Jr. "; // + appends characters

int c3 = count-2; // - subtracts integers
string s3 = name - "Jr. "; // error: - isn't defined for strings
```

通过“错误”，我们认识到编译器拒绝程序对字符串进行减。编译器确切地知道哪种操作可以应用于哪种变量，这样可以防止很多错误的发生。但是，编译器不知道哪种操作对你有用，因此它很高兴接受合法的操作，即使它们在你看来可能是荒谬的。例如：

```
int age = -100;
```

很明显，你的年龄不能是一个负数，但是没有人会告诉编译器，因此它会为这个定义生成代码。

下表给出了一些常见的和有用的操作符：

	bool	char	int	double	string
赋值	=	=	=	=	=
加			+	+	
连接					+
减			-	-	
乘			*	*	
除			/	/	
余数(模)			%		
递加 1			++	++	
递减 1			--	--	
加 n			+ = n	+ = n	
添加到结尾					+ =
减 n			- = n	- = n	
乘并赋值			* =	* =	
除并赋值			/ =	/ =	
余数并赋值			% =		
从 s 读到 x	s >> x				
从 x 写到 s	s << x				
等于	==	==	==	==	==
不等于	!=	!=	!=	!=	!=
大于	>	>	>	>	>
大于或等于	>=	>=	>=	>=	>=
小于	<	<	<	<	<
小于或等于	<=	<=	<=	<=	<=

空白表示一个操作符不能直接用于一种类型(尽管可能有间接使用这种操作符的方式，见 3.7 节)。我们将在后面的内容中解释这些操作符。这里的关键是有很多有用的操作符，它们对相似的类型通常是相同的。

我们来介绍一个涉及浮点数的例子：

```
// simple program to exercise operators
int main()
{
    cout << "Please enter a floating-point value: ";
    double n;
    cin >> n;
    cout << "n == " << n
        << "\nn+1 == " << n+1
        << "\nthree times n == " << 3*n
        << "\ntwice n == " << n+n
        << "\nn squared == " << n*n
        << "\nhalf of n == " << n/2
        << "\nsquare root of n == " << sqrt(n)
    cout << endl; // another name for newline ("end of line")
}
```

很明显，常见的数学操作有常见的表示法和含义，这点和我们在小学学到的知识一样。很自然，并不是我们想对一个浮点数做的任何事(例如得到它的平方根)都有相应的操作符。很多操作都表示为命名函数的形式。在这种情况下，我们使用标准库中的 `sqrt()` 来得到 n 的平方根：`sqrt(n)`。这种表示法与数学中相似。我们将会逐渐学习使用函数，并在 4.5 节和 8.5 节中讨论它们

的细节。

**试一试** 运行这个小程序。然后，修改它以读取一个 int 型，而不是一个 double 型。

注意，sqrt() 不是针对 int 型定义的，因此将 n 赋值给一个 double 型并执行 sqrt()。另外，“练习”一些其他操作。注意，对于 int 型来说，/ 是整除，% 是余数（模），因此  $5/2$  等于 2（而不是 2.5 或 3）， $5\%2$  等于 1。对整数 \*、/ 和 % 的定义，保证两个正整数 a 和 b 可以得到  $a/b * a + a \% b == a$ 。

字符串拥有更少的操作符，但在第 23 章中将看到足够多的命名操作。但是，所支持的操作符都可以按常规方式使用。例如：

```
// read first and second name
int main()
{
    cout << "Please enter your first and second names\n";
    string first;
    string second;
    cin >> first >> second;           // read two strings
    string name = first + ' ' + second; // concatenate strings
    cout << "Hello, " << name << '\n';
}
```

字符串 + 意味着连接，也就是说，当 s1 和 s2 是字符串时， $s1 + s2$  也是字符串，包含来自 s1 的字符后接来自 s2 的多个字符。例如，如果 s1 的值为 "Hello"，s2 的值为 "World"，那么  $s1 + s2$  的值为 "HelloWorld"。字符串比较操作特别有用：

```
// read and compare names
int main()
{
    cout << "Please enter two names\n";
    string first;
    string second;
    cin >> first >> second;           // read two strings
    if (first == second) cout << "that's the same name twice\n";
    if (first < second)
        cout << first << " is alphabetically before " << second << '\n';
    if (first > second)
        cout << first << " is alphabetically after " << second << '\n';
}
```

在这里，我们使用 if 语句来根据条件选择动作，该语句将在 4.4.4.1 节中详细介绍。

### 3.5 赋值和初始化

在很多方面，最有趣的操作符是赋值，表示为 =。它为一个变量赋予一个新的值。例如：

```
int a = 3;      // a starts out with the value 3
```

a: 3

```
a = 4;         // a gets the value 4 ("becomes 4")
```

a: 4

```
int b = a;      // b starts out with a copy of a's value (that is, 4)
```

a: 4

b: 4

```
b = a+5; // b gets the value a+5 (that is, 9)
```

a:	<b>4</b>
b:	<b>9</b>

```
a = a+7; // a gets the value a+7 (that is, 11)
```

a:	<b>11</b>
b:	<b>9</b>

最后一次赋值需要注意。首先，很明显 = 并不意味着等于，a 不等于  $a + 7$ 。它意味着赋值，也就是将一个新的值赋予一个变量。 $a = a + 7$  所做的事如下：

- 1) 首先，得到 a 的值，这里是整数 4。
- 2) 其次，将 7 和 4 相加，得到整数 11。
- 3) 最后，将整数 11 赋予 a。

我们也可以通过字符串来说明赋值：

```
string a = "alpha"; // a starts out with the value "alpha"
```

a:	<b>alpha</b>
----	--------------

```
a = "beta"; // a gets the value "beta" (becomes "beta")
```

a:	<b>beta</b>
----	-------------

```
string b = a; // b starts out with a copy of a's value (that is, "beta")
```

a:	<b>beta</b>
b:	<b>beta</b>

```
b = a+"gamma"; // b gets the value a+"gamma" (that is, "betagamma")
```

a:	<b>beta</b>
b:	<b>betagamma</b>

```
a = a+"delta"; // a gets the value a+"delta" (that is, "betadelta")
```

a:	<b>betadelta</b>
b:	<b>betagamma</b>

以上，我们使用“以…开始”和“获得”来区别两种相似的操作，但两者在逻辑上是有区别的：

- 初始化(给一个变量它的初值)。
- 赋值(给一个变量一个新的值)。

这些操作是如此相似，因此 C++ 允许我们对它们使用相同的符号(=)：

```
int y = 8; // initialize y with 8
x = 9; // assign 9 to x
```

```
string t = "howdy!"; // initialize t with "howdy!"
s = "G'day"; // assign "G'day" to s
```

但是，赋值和初始化在逻辑上是不同的。你可以通过类型描述(如 int 或 string)来区分它们，初始化总是从类型描述开始，而赋值并不需要这样做。从原则上来说，初始化时变量总是空的。另一方面，赋值在放入一个新的值之前，首先必须将旧的值清空。你可以将变量看做是一种小的盒子，值是一个可以放入其中的具体东西(例如一枚硬币)。在初始化之前盒子是空的，但是在初始化之后它总是包含一枚硬币，因此为了在里面放入一枚新的硬币，你(即赋值操作符)首先需要移

走旧的东西(“销毁旧的值”),而且你不能留下一个空盒子(必须赋予一个值)。在计算机内存中并不完全如此,但是它对于我们理解后面的内容没有坏处。

### 3.5.1 实例: 删除重复单词

当我们想将一个新的值放入一个对象,就需要赋值操作。当你考虑赋值操作时,很明显它在多次重复做一些事情时赋值是最有用的。当我们想以一个不同的值重复做某事时,我们需要进行一次赋值。让我们来看一个小的程序,它在一连串单词中找到相邻的重复字符。这段代码是大多数的语法检查程序的一部分:

```
int main()
{
    string previous = " "; // previous word; initialized to "not a word"
    string current; // current word
    while (cin>>current) { // read a stream of words
        if (previous == current) // check if the word is the same as last
            cout << "repeated word: " << current << '\n';
        previous = current;
    }
}
```

由于它没有告诉我们重复单词在文本中的哪个位置出现,因此这个程序对我们并不是很有帮助,但是现在它够用了。我们从如下一行开始逐行分析这个程序:

```
string current; // current word
```

这是一个字符串变量,我们使用它来读取当前(即最近阅读)的单词:

```
while (cin>>current)
```

这个结构称为一个 while 语句,它对右侧的程序结构感兴趣,我们将在 4.4.2.1 节中详细介绍。while 的意思是当输入操作 `cin >> current` 成功的情况下,(`cin >> current`)后面的语句将反复执行,而 `cin >> current` 成功的条件是从标准输入中读取字符。记住,对于一个 `string`,`>>` 读取的是用空格分开的单词。你可以通过给程序一个终止输入符号(通常是指文件结尾)来终止这个循环。在 Windows 系统的计算机中,使用 `Ctrl + Z`(同时按 `Control` 和 `Z`)紧接着一个回车。在 UNIX 或 Linux 系统的计算机中,使用 `Ctrl + D`(同时按 `Control` 和 `D`)。

因此,我们所做的是读取一个单词到 `current`,然后将它与前一个单词(存储在 `previous` 中)比较。如果它们是相同的,我们将会:

```
if (previous == current) // check if the word is the same as last
    cout << "repeated word: " << current << '\n';
```

然后,我们准备好对下一个单词重复进行上述操作。我们通过将 `current` 单词拷贝到 `previous` 中来进行这个操作:

```
previous = current;
```

这可以处理我们开始后的所有情况。当第一个单词没有前一个单词可以比较时,这段代码将会如何处理呢?这个问题可以在定义 `previous` 时得到解决:

```
string previous = " "; // previous word; initialized to "not a word"
```

`" "` 只包含一个字符(空格字符,通过按键盘中的空格键来得到)。输入操作符 `>>` 会跳过空格,我们不可能通过输入得到它。因此,第一次执行 while 语句时,检测

```
if (previous == current)
```

失败(正如我们所希望的)。

理解程序流程的一种方式是“推演计算机的运行”,也就是按程序的顺序逐行执行指定的工作。在一张纸上画出很多方块,然后在里面写入程序运行的结果。按程序指定的方式修改储存在

其中的值。

**试一试** 你亲自用一张纸来执行这个程序。输入是“The cat cat jumped”。即使是有经验的程序员，当某段代码不那么清晰时，也会用这种技术来推演其结果。

**试一试** 运行“重复单词检测程序”。用句子“She she laughed He He He because what he did did not look very very good good”来测试它。这里有多少个重复的单词？为什么？在这里，单词的定义是什么？重复单词的定义又是什么？（例如，“She she”是否重复？）

## 3.6 组合赋值运算符

一个变量的递增（增加 1）在程序中很常用，C++ 为它提供了一个特定的语法。例如：

**++counter**

意味着

**counter = counter + 1**

这里有很多其他常用方式，可以基于变量的当前值来修改它。例如，我们可能想将它加 7、减 9 或乘 2。C++ 直接支持这些操作。例如：

```
a += 7; // means a = a+7
b -= 9; // means b = b-9
c *= 2; // means c = c*2
```

通常，对一些二进制操作符 oper， $a \text{ oper} = b$  意味着  $a = a \text{ oper} b$ （参见附录 A.5）。首先，这一规则提供了操作符  $+=$ 、 $-=$ 、 $*=$ 、 $/=$  和  $\% =$ 。这提供了一种令人愉快的、紧凑的、直接反映我们观点的表示方法。例如，在很多应用领域中， $/=$  和  $\% =$  被称为“缩放”。

### 3.6.1 实例：重复单词统计

考虑上面的检测重复的相邻单词的例子。我们可以通过得到重复的单词在序列中的位置来改进程序。我们可以设置一个简单的变量，简单地统计单词数并输出重复的单词数：

```
int main()
{
    int number_of_words = 0;
    string previous = " "; // not a word
    string current;
    while (cin >> current) {
        ++number_of_words; // increase word count
        if (previous == current)
            cout << "word number " << number_of_words
            << " repeated: " << current << '\n';
        previous = current;
    }
}
```

我们将单词计数器设置为 0。我们每次看到一个单词，就会将这个计数器递增：

**++number\_of\_words;**

这样，第一个单词变为数值 1，下一个单词变为数值 2，依次类推。我们也可以按以下方式完成相同功能

**number\_of\_words += 1;**

或者是

**number\_of\_words = number\_of\_words + 1;**

但是，**++ number\_of\_words** 更加简短，并且直接表达递增的思想。

注意，这个程序与 3.5.1 节中的程序是如此相似。很明显，我们只是将这个程序从 3.5.1 节拿来，并对它进行一点儿修改以实现我们的目标。这是我们解决一个问题时用到的一个非常通用

的技术：当遇到一个问题需要解决时，我们找到一个相似的问题并用我们的方案加以适当修改。不要从头开始，除非你不得不这样做。在一个程序早期版本的基础上修改通常会节省大量时间，我们将会从成功深入原始程序中受益良多。

## 3.7 命名

我们命名自己的变量，这样我们可以记住它们，并在程序的其他部分使用。在 C++ 中什么可以作为一个名字呢？在一个 C++ 程序中，一个名字必须以一个字母开始，并且只能包含字母、数字和下划线。例如：

```
x
number_of_elements
Fourier_transform
z2
Polygon
```

以下这些不是名字：

```
2x          // a name must start with a letter
time$to$market // $ is not a letter, digit, or underscore
Start menu   // space is not a letter, digit, or underscore
```

当我们说“不是名字”时，我们的意思是 C++ 编译器不认为它们是名字。

如果你阅读系统代码或机器生成的代码，你可能看到以下划线开始的名字，例如 \_foo。你自己不要这样写，这样的名字是为编译器和系统实体保留的。尽量避免使用下划线，这样你的名字将不会与编译器生成的名字冲突。

名字是区分大小写的，也就是说，大写字母和小写字母是不同的，因此 x 和 X 是不同的名字。下面这个小程序至少有 4 个错误：

```
#include "std_lib_facilities.h"

int Main()
{
    String s = "Goodbye, cruel world!";
    cOut << s << '\n';
}
```

在定义名字时只用一个字符来区分通常不是一个好主意，例如 one 和 One，它不会使编译器混淆，但是它容易使程序员混淆。

**试一试** 编译“Goodbye, cruel world!”程序，并检查错误信息。编译器是否能发现所有错误？它遇到问题时的建议是什么？编译器是否混淆并发现超过 4 个错误？依次改正这些错误，首先从词法开始，看错误信息如何改变（与改进）。

C++ 语言保留了很多（大约 70 个）名字作为“关键字”。我们将在附录 A.3.1 中列出它们。你不能使用它们作为你的变量、类型、函数等的名字。例如：

```
int if = 7; // error: "if" is a keyword
```

你可以使用标准库中的名字（例如 string），但是你不应该这样做。如果你想要使用标准库的话，这样一个通用名字的重用将会带来麻烦：

```
int string = 7; // this will lead to trouble
```

当你为自己的变量、函数、类型等选择名字时，最好选择有特定含义的名字。也就是说，选择有助于人们理解你的程序的名字。如果你将变量胡乱命名为“简单型”的名字，例如 x1、x2、s3 与 p7，则你在理解程序要做什么时也会遇到问题。缩写和仅有首字母的缩写会使人糊涂，因此要谨

慎使用。我们在编程时明白这些缩略语的含义，但是对于下面的名字，我们预料至少有一个使你感到困惑：

**mthf  
TLA  
myw  
NBV**

我们预料在几个月之内，我们自己也至少会忘掉其中一个名字的含义。

短名字(例如 `x` 与 `i`)在常规使用时是有意义的。也就是说，`x` 可能是一个本地变量或一个参数(参见 4.5 节与 8.4 节)，`i` 可能是一个循环的索引(见 4.4.2.3 节)。

不要使用很长的名字，它们难以输入，由于太长难以在一个屏幕中显示，也难以快速读取。下面这些名字可能是合适的：

**partial\_sum  
element\_count  
stable\_partition**

下面这些名字可能太长：

**the\_number\_of\_elements  
remaining\_free\_slots\_in\_symbol\_table**

我们的“风格”是在一个标识符中使用下划线来区分单词，例如 `element_count`，而不是其他方案(例如 `elementCount` 与 `ElementCount`)。我们不使用全部大写字母的名字，例如 `ALL_CAPITAL LETTERS`，这是由于它们通常保留作为宏(参见 27.8 节与附录 A.17.2)，因此我们避免这样使用。我们使用首字母大写来定义自己的类型，例如 `Square` 与 `Graph`。C++ 语言和标准库不使用大写字母，因此它们使用 `int` 而不是 `Int`，使用 `string` 而不是 `String`。因此，我们的约定会帮助你减少对自己类型和标准类型的混淆。

避免使用容易错误、误读或混淆的名字。例如：

Name	names	nameS
foo	foo	fl
f1	fl	fi

字符 0、o、O、1、l 和 I 容易引起麻烦。

### 3.8 类型和对象

类型的概念是 C++ 和大多数编程语言的核心。让我们以更紧密和稍微带有技术性的观点来看待类型，特别是我们在计算过程中用来存储数据的对象类型。它将会在长时间运行时节省时间，它也可能避免引起你的混淆。

- 类型定义一组可能的值与一组操作(对于一个对象)。
- 对象是用来保存一个指定类型值的一些内存单元。
- 值是被解释为一个类型的内存中的一组比特。
- 变量是一个命名过的对象。
- 声明是命名一个对象的一条语句。
- 定义是为一个对象分配内存空间的声明。

我们可以非正式地将一个对象看做一个盒子，我们可以将指定类型的值放入这个盒子。一个 `int` 盒子可以保存整数，例如 7、42 与 -399。一个 `string` 盒子可以保存字符串值，例如 "Interoperability"、"tokens: ! @# \$ % ^& \* " 与 "Old McDonald had a farm"。我们可以生动地将它想象为：

<b>int a = 7;</b>	a: <table border="1"><tr><td>7</td></tr></table>	7	
7			
<b>int b = 9;</b>	b: <table border="1"><tr><td>9</td></tr></table>	9	
9			
<b>char c = 'a';</b>	c: <table border="1"><tr><td>a</td></tr></table>	a	
a			
<b>double x = 1.2;</b>	x: <table border="1"><tr><td>1.2</td></tr></table>	1.2	
1.2			
<b>string s1 = "Hello, World!";</b>	s1: <table border="1"><tr><td>13</td><td>Hello, World!</td></tr></table>	13	Hello, World!
13	Hello, World!		
<b>string s2 = "1.2";</b>	s2: <table border="1"><tr><td>3</td><td>1.2</td></tr></table>	3	1.2
3	1.2		

由于 string 要跟踪它保存的字符数，因此 string 比 int 的表示方法更复杂。注意，一个 double 保存一个数字，而一个 string 保存多个字符。例如，x 保存数字 1.2，而 s2 保存三个字符 '1'、'.' 与 '2'。字符的单引号和字符串常量并不保存。

每个 int 的大小是相同的。也就是说，编译器为每个 int 分配相同的固定大小的内存。在一个典型的台式电脑中，这个大小是 4 个字节(32 个比特)。与此类似，bool、char 与 double 是固定大小的。在通常情况下，你会发现台式电脑为一个 bool 或一个 char 分配 1 个字节(8 个比特)，为一个 double 分配 8 个字节。注意，不同类型的对象使用不同大小的空间。特别地，一个 char 比一个 int 占用更少的空间，string 不同于 double、int 与 char，不同大小的字符串占用不同大小的空间。

在内存中比特的含义完全依赖于访问它时所用的类型。我们这样考虑：计算机内存不知道我们的类型，只是将它保存起来。只有当我们决定内存如何解释时，在内存中的比特才有意义。这个过程与我们每天使用数字相似。12.5 的含义是什么？我们并不知道。它可以是 \$ 12.5、12.5cm 或 12.5gallons。只有当我们使用单位时，才会定义 12.5 的含义。

例如，当值 120 表示的是一个 int，而值 'x' 表示的是一个 char 时，它们在内存中的比特值相同。如果我们将它看成一个 string，它将不会有意义，并在我们试图使用它时出现运行错误。我们可以像下面这样生动地解释它，使用 1 和 0 表示内存中的比特值：

00000000 00000000 00000000 01111000
-------------------------------------

这是一个内存区域(一个字)中的一组比特，它们可以被读取为一个 int(120)或一个 char('x'，只看最右侧的 8 个比特)。一个比特是计算机中的一个内存单元，它可以保存一个值 0 或 1。想理解二进制数的含义，见 A.2.1.1 节。

### 3.9 类型安全

每个对象在定义时被分配一个类型。对于一个程序或程序的一个部分，如果使用的对象符合它们规定的类型，那么它们是类型安全的。不幸的是，有很多执行操作的方式不是类型安全的。例如，在一个变量没有初始化之前使用它，则认为不是类型安全的：

```
int main()
{
    double x;           // we "forgot" to initialize:
                        // the value of x is undefined
    double y = x;       // the value of y is undefined
    double z = 2.0+x;   // the meaning of + and the value of z are undefined
}
```

当使用没有初始化的 x 时，一个实现甚至被允许出现一个硬件错误。记得初始化你的变量！这个规则的例外只有很少(非常少)，例如我们立即将一个变量作为输入操作的目标，但是记住初始化变量是一个好习惯，它会为我们减少很多的麻烦。

完全的类型安全是最为理想的，因此它对语言来说应是一般规则。不幸的是，C++ 编译器不能保证完全的类型安全，但是通过良好的编码训练和运行时检查，我们可以避免违反类型安全。理想状态是永远不要使用编译器也不能保证是安全的语言功能：静态类型安全。不幸的是，这对于大多数有趣的编程应用过于严格。一种显然的退而求其次的方法是，由编译器隐式生成代码，检查是否有违反类型安全的情况并捕获它们，但这已超出了 C++ 的能力。当我们决定做（类型）不安全的事时，我们必须自己做相应的检查工作。当我们在本节中遇到这种情况时，我们将会指出来。

类型安全的思想在编写代码时非常重要。这是我们在前面章节花费时间介绍它的原因。请注意陷阱并避开它们。

### 3.9.1 安全类型转换

在 3.4 节中，我们发现不能直接将 char 相加，或者将一个 double 与一个 int 比较。但是，C++ 提供间接方式来完成这些操作。在有必要时，一个 char 可以转换成一个 int，而一个 int 也可以转换成一个 double。例如：

```
char c = 'x';
int i1 = c;
int i2 = 'x';
```

这里的 i1 和 i2 都被赋值为 120，它是字符 'x' 在大多数常见的 8 比特字符集（例如 ASCII）中的整数值。这是一个简单和安全的方法，通过它可以得到一个字符的数字表示。由于没有信息丢失，我们称这种 char-int 的转换为安全的。也就是说，我们可以将 int 结果拷贝到一个 char 中，并且得到原始的值：

```
char c2 = i1;
cout << c << ' ' << i1 << ' ' << c2 << '\n';
```

这里将会打印

x 120 x

在这种情况下，一个值总是被转换成一个等价的值，或者一个最接近等价的值（对于 double），那么这些转换就是安全的：

```
bool 到 char
bool 到 int
bool 到 double
char 到 int
char 到 double
int 到 double
```

最常用的转换是从 int 到 double，这是由于它允许在表达式中混合使用 int 和 double：

```
double d1 = 2.3;
double d2 = d1+2; // 2 is converted to 2.0 before adding
if (d1 < 0) // 0 is converted to 0.0 before comparison
    error("d1 is negative");
```

对于一个确实很大的整数，当它被转换成 double 时，我们（在有些计算机中）可能承受一些精度上的损失。这种情况不是很常见。

### 3.9.2 不安全类型转换

安全的转换对程序员通常是一个福音，它可以简化编写代码的过程。不幸的是，C++ 也允许（隐式的）不安全转换。所谓的不安全，我们指的是一个值可以转换成一个其他类型的值，这个值不等于原始的值。例如：

```

int main()
{
    int a = 20000;
    char c = a; // try to squeeze a large int into a small char
    int b = c;
    if (a != b) // != means "not equal"
        cout << "oops!: " << a << "!=" << b << "\n";
    else
        cout << "Wow! We have large characters\n";
}

```

这种转换又称为“缩小”转换，这是由于它们将一个值放入一个对象，而这个对象大小难以存放这个值。不幸的是，只有少数编译器会警告将 char 初始化为 int 的不安全。问题是一个 int 通常比一个 char 大，因此(在这种情况下)它可以保存一个 int 值，但是这个值并不能表示为一个 char。尝试执行这个程序，查看你计算机中的值 b(常见的结果是 32)。更进一步，完成实验：

```

int main()
{
    double d = 0;
    while (cin >> d) { // repeat the statements below
        // as long as we type in numbers
        int i = d; // try to squeeze a double into an int
        char c = i; // try to squeeze an int into a char
        int i2 = c; // get the integer value of the character
        cout << "d==" << d // the original double
        << " i==" << i // converted to int
        << " i2==" << i2 // int value of char
        << " char(" << c << ")\\n"; // the char
    }
}

```

我们使用 while 语句允许尝试很多值，这个语句将在 4.4.2.1 节中解释。

**试一试** 输入各种各样的值来运行这个程序。尝试小的值(例如 2 和 3)；尝试大的值(大于 127 和大于 1000)；尝试负值；尝试 56；尝试 89；尝试 128；尝试非整数值(例如 56.9 和 56.2)。另外，如何在你的机器上从 double 转换成 int，以及如何从 int 转换成 char。本程序将显示，对一个给定的整数值，你的机器将打印什么字符(如果存在的话)。

你将发现很多输入值产生“不合理”的结果。基本上，我们是在尝试将 1 加仑水倒入容量为 1 品脱的桶中(大约是将 4 升水倒入一个 500 毫升的杯子)。

```

double 到 int
double 到 char
double 到 bool
int 到 char
int 到 bool
char 到 bool

```

所有这些转换被编译器接受，即使它们是不安全的。所谓不安全是指它们保存的值可能与被赋予的值不同。为什么这会是个问题？这是由于我们经常不会怀疑一个不安全的转换会发生。考虑：

```

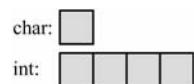
double x = 2.7;
// lots of code
int y = x; // y becomes 2

```

在我们定义 y 时可能忘记 x 是一个 double，或者我们临时忘记 double 到 int 转换会截短(总是去掉小数点后的尾数)，而不是使用常用的四舍五入。发生的事情是完全可以预测的，但是在 int y = x；处没有任何东西能提醒我们信息(.7)被丢掉了。

从 int 到 char 的转换不会出现截短的问题，int 和 char 都不能表示一个整数的一部分。但是，一个 char 只能保存非常小的整数值。在一台 PC 机中，一个 char 占用 1 个字节，而一个 int 占用 4 个字节，如右图所示。因此，我们不能将一个大的数（例如 1000）放入一个 int 而不丢失任何信息：这个值是“缩小的”。例如：

```
int a = 1000;
char b = a; // b becomes -24 (on some machines)
```



不是所有的 int 值都有等价的 char，而 char 值的确切范围依赖于特定的实现。在一台 PC 机中，char 值的范围是 [ -128: 127 ]，但是只有 [ 0, 127 ] 可以方便地移植。这是由于并不是每台计算机都是 PC 机，不同计算机的 char 值的范围不同，例如 [ 0: 255 ]。

为什么人们接受缩小转换的问题？主要原因是历史性的：C++ 从它的前辈语言 C 继承了缩小转换，因此从 C++ 出现时很多代码就依赖于缩小转换。很多这种转换实际上不会引起问题，这是由于它们所涉及的值碰巧在范围内，并且很多程序员反对编译器“告诉它们做什么”。特别是对有经验的程序员来说，这些不安全转换问题在小的程序中是可管理的。它们在大的程序中可能是错误的来源，并且是一个新程序员出现问题的重要原因。但是，编译器可以对多数的缩小转换发出警告。

如果你认为转换可能导致一个错误值，那么你需要做什么？在我们做本节中的第一个例子时，你在赋值之前要简单检查这个值。5.6.4 节与 7.5 节介绍做这种检查的简单方式。

## 简单练习

在完成这个练习的所有步骤之后，运行你的程序以确认它确实在做你希望它做的事。列出那些曾经出现的错误，这样以后可以尽量避免它们。

1. 这个练习是编写一个程序，基于用户输入生成一封简单格式的信。首先，输入来自 3.1 节的代码，提示用户输入他或她的名字，并且输出“Hello, first\_name”，这里的 first\_name 是用户输入的名字。然后，按以下要求修改你的代码：将提示修改为“Enter the name of the person you want to write to”，并将输出修改为“Dear first\_name,。”。不要忘记逗号。
2. 增加一行或两行前言，例如“How are you? I am fine. I miss you.”确定首行需要缩进。增加由你选择的几行，这是你的信。
3. 现在，提示用户输入另一个朋友的名字，并将它保存在 friend\_name 中。在你的信中增加一行：“Have you seen first\_name lately?”
4. 声明一个名为 friend\_sex 的 char 变量，并将它的值初始化为 0。如果这个朋友是男性，提示用户输入一个 m；如果这个朋友是女性，提示用户输入一个 f。为变量 friend\_sex 分配输入值。然后，使用两个 if 语句完成以下输出：

如果这个朋友是男性，输出“If you see friend\_sex please ask him to call me.”。

如果这个朋友是女性，输出“If you see friend\_sex please ask her to call me.”。

5. 提示用户输入收信人的年龄，并为它分配一个 int 变量 age。让你的程序输出“I hear you just had a birthday and you are age years old.”如果 age 小于等于 0 或大于等于 110，调用 error( "you'r kidding!" )。

6. 在你的信中增加下面的内容：

如果你朋友的年龄小于 12，输出“Next year you will be age + 1.”。

如果你朋友的年龄等于 17，输出“Next year you will be able to vote.”。

如果你朋友的年龄大于 70，输出“I hope you are enjoying retirement.”。

7. 添加“Yours sincerely,”接着是两个空行，后面是你的名字。

## 思考题

1. 术语 prompt 的含义是什么？
2. 哪种操作符用于读取值并存入变量中？

3. 如果你希望用户在你的程序中为一个命名为 number 的变量输入一个整数值，如何用两行代码来完成它并将值输入你的程序中？
4. \n 的名称是什么和它的目的是什么？
5. 怎样终止输入一个字符串？
6. 怎样终止输入一个整数？
7. 如何将你书写的
 

```
cout << "Hello, ";
cout << first_name;
cout << "!\n";
```

 作为一行代码来输入？
8. 对象是什么？
9. 文字常量是什么？
10. C++ 中有哪几种文字常量？
11. 变量是什么？
12. 一个 char、int 和 double 的典型大小是多少？
13. 我们用哪种方式测试内存中的小实体（例如 int 和 string）的大小？
14. 操作符 = 与 == 之间的区别是什么？
15. 定义是什么？
16. 什么是一次初始化，它和一次赋值的区别是什么？
17. 什么是字符串连接，如何使它在 C++ 中正确工作？
18. 在以下名字中，哪些在 C++ 中是合法的？如果一个名字是不合法的，为什么？
 

<b>This_little_pig</b>	<b>This_1_is_fine</b>	<b>2_For_1_special</b>
<b>latest thing</b>	<b>the_\$12_method</b>	<b>_this_is_ok</b>
<b>MiniMineMine</b>	<b>number</b>	<b>correct?</b>
19. 请举出 5 个容易引起混淆、你不会使用的合法名字。
20. 选择名字的好规则有哪些？
21. 什么是类型安全，为什么它是重要的？
22. 为什么从 double 转换成 int 是一件坏事？
23. 请定义一个协助判断从一种类型到另一种类型的转换是否安全的规则。

## 术语

赋值	定义	运算	cin	递增	运算符
连接	初始化	类型	转换	名字	类型安全
声明	缩小	值	递减	对象	变量

## 习题

1. 如果你还没有开始这样做，请先做本章的“试一试”练习。
2. 编写一个 C++ 程序，将英里转换成公里。你的程序应该有一个合理的提示，要求用户输入一个表示英里的数字。提示：这里是 1.609 公里要转换成英里。
3. 编写一个程序，不做其他的任何事情，只声明一系列合法与不合法的变量名（例如 int double = 0;），这样你可以看到编译器的反应。
4. 编写一个程序，提示用户输入两个整数值。将这些值保存在 int 变量 val1 和 val2 中。编写程序决定这些值中的最小值、最大值、和、差、乘积和比率，并且将它们报告给用户。
5. 修改上面程序，让用户输入浮点数值并将它们保存在 double 变量中。比较你选择的多种输入在两个程序的输出。这些结果是否相同？它们是否对？区别是什么？
6. 编写一个程序，提示用户输入三个整数值，然后按数值次序输出这些值并以逗号隔开。因此，如果用户输入值为 10 4 6，输出值为 4, 6, 10。如果有两个值相同，那么将它们连续输出。因此，输入 4 5 4 将会输出 4, 4, 5。

7. 重做练习 6，但是输入 3 个字符串。因此，如果用户输入“Steinbeck”、“Hemingway”和“Fitzgerald”，输出将是“Fitzgerald, Hemingway, Steinbeck”。
8. 编写一个程序，测试一个整数值是奇数还是偶数。记住确认你的输出是清楚和完整的。换句话说，不要只是输出“yes”或“no”。你的输出应该是独立的，例如“The value 4 is an even number.”提示：阅读 3.4 节中的余数(模)操作。
9. 编写一个程序，将数字的英文单词(例如“zero”和“two”)转换成数字(例如 0 和 2)。当用户输入一个数字的英文拼写，程序将打印出对应的数字。针对数值 0、1、2、3 和 4 完成这个操作，如果用户输入无法对应的值(例如“stupid computer!”)，程序输出“not a number I know”。
10. 编写一个程序，执行一个包括两个运算的操作，然后输出结果。例如：

+ 100 3.14

\* 4 5

将操作读入一个字符串称为 operation，用一个 if 语句判断哪个操作是用户希望的，例如 if (operation == "+")。将运算读入 double 类型的变量。实现这些称为 +、-、\*、/ 的操作，加、减、乘、除都有各自明显的意义。

11. 编写一个程序，提示用户输入一美分(1 美分硬币)、五美分(5 美分硬币)、十美分(10 美分硬币)、二十五美分(25 美分硬币)、半美元(50 美分硬币)和一美元(100 美分硬币)的数量。对每种面值的硬币，分别提示用户输入其数量，例如“How many pennies do you have?”然后，程序将输出类似下面的内容：

You have 23 pennies.

You have 17 nickels.

You have 14 dimes.

You have 7 quarters.

You have 3 half dollars.

你可能需要发挥想象力才能将合计值右对齐输出，但请尽力尝试，你可以完成它。然后做出一些改进：如果某个面值只有一枚硬币，确保输出语法是正确的，例如，应输出“14 dimes”和“1 dime”(而不是“1 dimes”)。另外，将合计值用美元和美分来表示，例如用 \$ 5.73 来代替 573 美分。

## 附言

请不要低估类型安全概念的重要性。类型是大多数正确程序的核心概念，大多数用于构建程序的有效技术依赖于类型的设计与使用，正如我们将在第 6 章、第 9 章、第二部分、第三部分、第四部分中看到的一样。

# 第 14 章 设计图形类

“实用的，持久的，优美的。”

——Vitruvius

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，我们不得不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，或许你应该把本章看做“图形类的设计与实现”。

## 14.1 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？

### 14.1.1 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。希望我们的图形类能够降低程序员学习和使用的难度。

我们的程序设计理念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——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——所有形状的统称——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，它是一个一般性的、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的类时，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所有方面进行建模。这不仅是不可能的，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？快餐店的地点呢？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。但是这样的库就“全面”了吗？它不太可能还涵盖其他图形应用，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、清晰地建模其应用领域，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图形化输出数值计算/科学计算/工程计算等应用的数据的用户。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，如果你决定了这样一条路线，请首先掌握第 17 章和第 18 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的类（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 Shape 类，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、更加有效地建模我们的图形领域。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感到混乱。

## 14.1.2 操作

我们为每个类提供了最少的操作。我们的设计理念是用最小的接口来实现我们想做的事情。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：形状需要一个位置，因此构造函数接受一个 Point 作为第一个参数：

```
Line In(Point(100,200),Point(300,400));
Mark m(Point(100,200),'x');           // display a single point as an "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，就可以有两种不同的风格：

```
void draw_line(Point p1, Point p2);          // from p1 to p2 (our style)
void draw_line(int x1, int y1, int x2, int y2); // from (x1,y1) to (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); // our style
draw_rectangle (100,200,300,400);      // alternative
```

第一个调用利用 Point、宽度和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形式（对某些人可能是合理的形式）？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、线等的函数都叫做add()，所有画线函数叫做draw\_lines()。这种一致性能帮助我们记忆（只需要记住更少的细节），同时在我们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。这种代码称为泛型程序，参见第19~21章。

### 14.1.3 命名

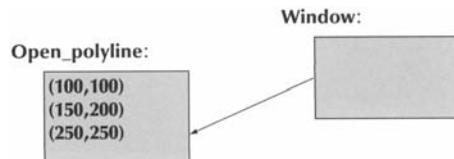
逻辑上，不同的操作应该有不同的名字。这看起来似乎是很显然的，但是请考虑：为什么我们将一个Shape“添加”（attach）到一个Window中，但却将一个Line“加入”（add）一个Shape中呢？两个操作都是“将某物放到某物之中”，那么这种相似性是不是应该反映为相同的名字呢？不是！这种相似性背后隐藏了一个根本的不同点。考虑如下代码：

```
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，我们建立了一个从窗口win到我们的形状opl的连接；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()的区别如右图所示。基本上，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);
} // oops, the lifetime of r ends here

int main()
```

```

{
    Simple_window win(Point(100,100),600,400,"My window");
    //...
    f(win);      // asking for trouble
    //...
    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 17 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，我们还是要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 13.10 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。

#### 14.1.4 可变性

当设计一个类时，“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;      // radius
};

Circle c(Point(100,200),50);
c.r = -9;      // OK? No — compile-time error: Circle::r is private

```

正像你可能在第 13 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，使我们有机会检查那些“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，我们只是进行了有限的检查，所以要小心处理你的数据。我们决定不进行一致的、全面的检查，一方面是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(可看做一组 Window)当做一种纯粹的输出设备。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。

## 14.2 Shape 类

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会出现问题。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 { // deals with color and style and holds sequence of lines
public:
    void draw() const; // deal with color and draw lines
    virtual void move(int dx, int dy); // move the shape +=dx and +=dy

    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; // read-only access to points
    int number_of_points() const;

    virtual ~Shape() {}

protected:
    Shape();
    virtual void draw_lines() const; // draw the appropriate lines
    void add(Point p); // add p to points
    void set_point(int i, Point p); // points[i]=p;
private:
    vector<Point> points; // not used by all shapes
    Color lcolor; // color for lines and characters
    Line_style ls;
    Color fcolor; // fill color

    Shape(const Shape&); // prevent copying
    Shape& operator=(const Shape&);
};
```

这是一个相对复杂的类，用以支持各种各样的图形类以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。

#### 14.2.1 一个抽象类

考虑 Shape 类的第一个构造函数：

```
protected:
    Shape();
```

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 只能用做其他类(如 Line 和 Open\_polyline)的基类。“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; // error: cannot construct Shape
```

Shape 被设计为只能当做一个基类。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：“是什么形状？”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。构造函数可以定义如下：

```
Shape::Shape()
    : lcolor(fl_color()),           // default color for lines and characters
    ls(0),                         // default style
    fcolor(Color::invisible)       // no fill
{                                }
```

这是一个默认构造函数，所以它将成员设置为默认值。再次强调一下，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。

如果一个类只能被用做基类，它就是一个抽象类。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 14.3.5 节。与抽象类相对的是具体类，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”。照相机是什么牌子的？具体是什么型号？单词“照相机”是一个通称；它代表一个抽象的概念。而“Olympus E-3”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，“照相机”更像一个抽象类（基类），“Olympus E-3”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）则更像是一个对象。

声明“virtual ~Shape(){}”定义了一个虚析构函数。我们现在还不会用到它，所以将在 17.5.2 节进行介绍，在那里我们介绍如何使用它。

## 14.2.2 访问控制

Shape 类将所有数据成员均声明为 private：

```
private:
    vector<Point> points;
    Color lcolor;
    Line_style ls;
    Color fcolor;
```

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，而私有变量的命名就不那么重要

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)      // protected
{
    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(`points`)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 12.7.6 节)？
- `Lines` 只允许添加成对的点(而不是一个单独的点；参见 13.3 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 13.8 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设想。

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 6 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` 的每个属性的设计原则仍旧成立。例如，如果要实现一个 `Mutable_rectangle` 类，我们可以从 `Rectangle` 类派生，并且提供更改点的操作。

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)      // not used; not necessary so far
{
    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
    // draw lines connecting pairs of points
{
    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 {      // close-to-minimal definition — too simple — not used
    Shape();
    void draw() const;      // deal with color and call draw_lines
    virtual void draw_lines() const; // draw the appropriate lines
    virtual void move(int dx, int dy); // move the shape +=dx and +=dy
    vector<Point> points;      // not used by all shapes
    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”(参见 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(`public`)的，并被用户代码所使用，那么改进接口库就只能伴随着重写这些用户代码(因为在用户代码中使用了名字 `line_color` 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。

### 14.2.3 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;      // deal with color and call draw_lines
virtual void draw_lines() const; // draw the lines appropriately
```

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 14.4 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，它只是提供了以下两个函数：

- `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();
    // there is no good portable way of retrieving the current style
```

```

fl_color(lcolor.as_int());           // set color
fl_line_style(ls.style(),ls.width()); // set style
draw_lines();
fl_color(oldc);      // reset color (to previous)
fl_line_style(0);      // reset line style to default
}

```

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。如果那样做，可能需要在 Shape 对象中保存每个形状的最后一个像素。如果我们继续使用 vector < Point > 模型，将会存储非常多的点。更糟糕的是，“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 类都可能有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; // let each derived class define its
                                    // own draw_lines() if it so chooses
    // ...
};

struct Circle : Shape {
    // ...
    void draw_lines() const; // "override" Shape::draw_lines()
    // ...
};

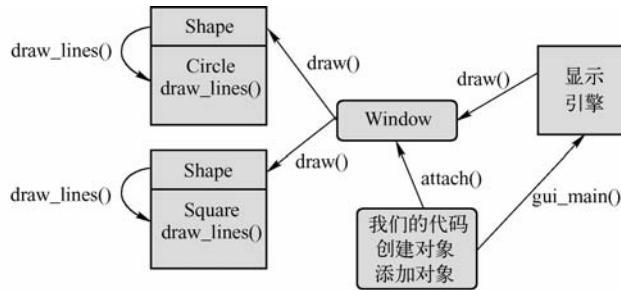
```

所以，如果 Shape 是一个 Circle 对象，Shape 的 draw\_lines() 就会以某种方式调用 Circle 的一个版本；同样，如果是一个 Rectangle 对象，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 13 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() 函数以及 Shape 的派生类使用。

这样就完成了 12.2 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。我们的用户代码对 gui\_main() 函数的调用会启动这个显示引擎，如下图所示：



gui\_main() 函数是什么？到目前为止，我们还没有在代码中实际看到过它。作为替代，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)      // move the shape +=dx and +=dy
{
    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 并不知道。例如，参考 Axis( 参见 12.7.3 节和 15.4 节 )。

逻辑上，move() 函数对于 Shape 类并不是必须的，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。

#### 14.2.4 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 private：

```

private:
    Shape(const Shape&);          // prevent copying
    Shape& operator=(const Shape&);
  
```

这样做的效果是只有 Shape 的成员可以使用默认的拷贝操作来拷贝 Shape 对象。这是为了防止意外拷贝而经常使用的一种方法。例如：

```

void my_fct(const Open_polyline& op, const Circle& c)
{
    Open_polyline op2 = op; // error: Shape's copy constructor is private
    vector<Shape> v;
    v.push_back(c);           // error: Shape's copy constructor is private
    // ...
    op = op2;                // error: Shape's assignment is private
}
  
```

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”的大小为一个 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，如右图所示。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); // the Open_polyline argument refers to a Marked_polyline
```

Shape:  
points  
line\_color  
ls

Circle:  
points  
line\_color  
ls  
r

现在 Open\_polyline 的拷贝操作会将对象 mp 的 string 成员 mark“切断”。

基本上，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，则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，如果窗口只保存了 Shape 的一个副本，而不是引用，那么，对原 Shape 对象的任何修改都不会影响到副本。那么，如果我们改变形状的颜色，窗口将不会知道这个变化，仍会用原来的颜色来显示副本。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，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，而所有的形状类恰好都是这种情况。

## 14.3 基类和派生类

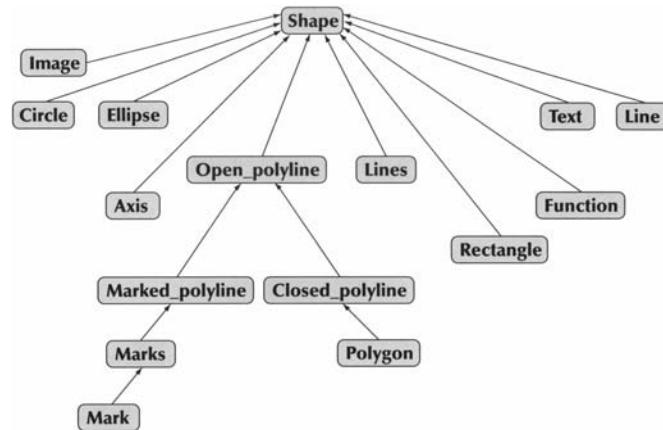
让我们从一个更为技术性的角度来观察基类和派生类，也就是说，在本节中(只在本节)我们将讨论的焦点从程序设计、应用设计和图形转移到程序设计语言的特性上来。当设计一个图形接口库时，我们依赖以下 3 个关键的语言机制：

- 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，派生类称为子类(subclass)，而基类称为父类(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。这通常称为运行时多态(run-time polymorphism)、动态分派(dynamic dispatch)或运行时分派(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，这通常称为封装(encapsulation)。

继承、运行时多态和封装的使用，实际上就是面向对象程序设计(object-oriented programming)最常见的标志。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 20 和 21 章中，我们将看到 C++ 如何支持泛型编程。C++ 借用了 Simu-

la67 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,虽然代表的是一个抽象概念,我们永远不能直接创建其对象,公共基类(Shape)仍是最的一个类。

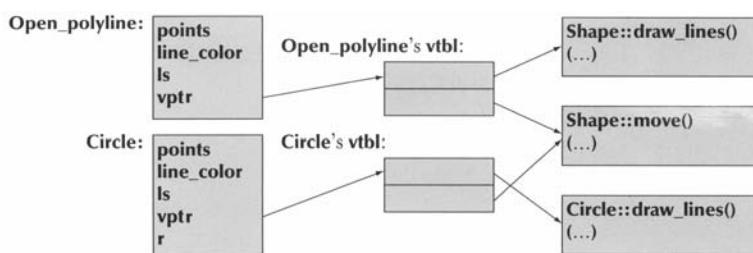
#### 14.3.1 对象布局

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,如右图所示。一个 Circle 对象包含 Shape 类的数据成员(毕竟,它也是一种 Shape),并且可以当做 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。

Shape: points  
line\_color  
ls

Circle: points  
line\_color  
ls  
r

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,这个表通常称为 vtbl(即“virtual table”或“virtual function table”,虚函数表),它的地址通常称为 vptr(即“virtual pointer”,虚指针)。我们将在第 17~18 章中讨论指针,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 加入布局图中可得到下图:



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() 的对应位置，调用找到的函数。基本上，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()。基本上，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。例如，Circle :: draw\_lines() 覆盖了 Shape :: draw\_lines()。

我们为什么要告诉你这些关于 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？和谁相比？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后，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。

### 14.3.2 类的派生和虚函数定义

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，结构体(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体的公有成员。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，可能你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: /* ... */}; // probably a mistake
```

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 { /* ... */ } // error
void Shape::move() { /* ... */ } // OK

```

### 14.3.3 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; // probably a mistake (int argument?)
    void drawlines() const; // probably a mistake (misspelled name?)
    void draw_lines(); // probably a mistake (const missing?)
    //...
};

```

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，难以模仿其所有细节来学习覆盖技术。因此，我们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 "; } // not virtual
};

struct D : B {
    void f() const { cout << "D::f "; } // overrides B::f
    void g() { cout << "D::g "; }
};

struct DD : D {
    void f() { cout << "DD::f "; } // doesn't override D::f (not 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)
    // a D is a kind of B, so call() can accept a D
    // a DD is a kind of D and a D is a kind of B, so call() can accept a 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，你就会明白继承和虚函数机制了。

#### 14.3.4 访问

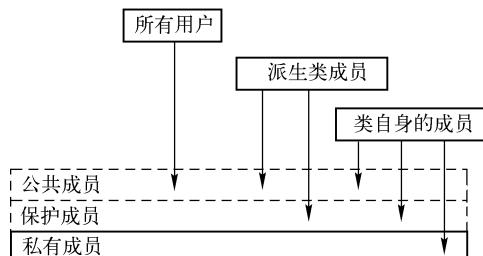
C++ 为类成员访问提供了一个简单的模型。类的成员可以是：

- 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。
- 受保护的(protected)：如果一个成员是受保护的，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见右图。基类可以是私

有的、受保护的或公有的：

- 如果类 D 的一个基类是 private 的，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 的，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 的，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《2003 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)，作为一名程序员(一位软件开发者、工程师、用户以及所有实际使用语言的人)乐趣更多，对社会也更有价值。

#### 14.3.5 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，即相关实体共性的一般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(或抽象或一般性等)。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 14.2.1 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：声明一个或者多个必须在派生类中被覆盖的虚函数。例如：

```

class B { // abstract base class
public:
    virtual void f() =0; // pure virtual function
    virtual void g() =0;
};

B b; // error: B is abstract

```

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();
    void g();
};

D1 d1; // ok

```

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();
    // no g()
};

D2 d2; // error: D2 is (still) abstract
class D3 : public D2 {
public:
    void g();
};

D3 d3; // ok

```

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），因此没用任何构造函数（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）。

## 14.4 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做其公有基类的对象使用）是一个拙劣且容易出错的设计。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、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，而无需知道具体的派生类（“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：通过使用基类提供的功能，简化了派生类的实现。

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。特别是，“图形引擎”（FLTK 加上操作系统的图形功能）是在设计图形类之前若干年就编写、编译好的！它根本无法知道图形类的任何信息。我们只是定义了特定的形状并且将它们当做 `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（例如，`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 17~19 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的技术之一。

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地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要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 14.3.5 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，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，重做练习 1~3，并解释其结果。
5. 在 `B1` 中定义一个纯虚函数 `pvf()`，重做练习 1~4，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，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

抽象类	可变性	纯虚函数	访问控制	对象布局	子类
基类	面向对象	超类	派生类	多态	虚函数
分派	私有	虚函数调用	封装	保护	虚函数表
继承	公有				

## 习题

1. 定义两个类 Smiley 和 Frowny, 它们都派生自 Circle 类, 并且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。注意: 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，本题和习题 12 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“lrlr”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. 大多数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()。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(如果向量不为空的话)。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 对象的线的颜色，“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，这些类可以准确反应我们的思想，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)都是我们构建系统的最有力的工具。

# 第 25 章 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，即介绍为“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，重点是低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/解密算法的实现的例子。

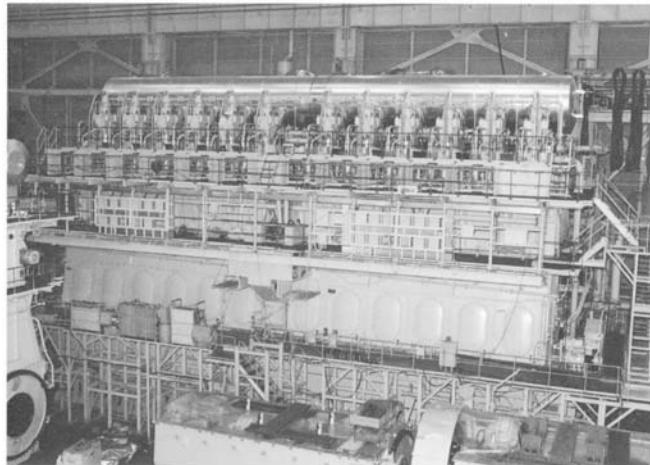
## 25.1 嵌入式系统

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，甚至每个镜头都由独立的处理器来控制。
- 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 70 米（210 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做计算机。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，如下图所示：



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系统和引擎控制系统(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，在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“达到数十亿美元”)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，在嵌入式系统中资源(内存、处理器、能源等)是有限的：对于引擎中的计算机，这可能不是一个大问题。但请考虑手机、传感器、PDA、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 2 GHz 的双核 CPU 和 2GB 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 60 MHz 的处理器和 256 KB 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 1 MHz 的处理器和几百个字节的内存。

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)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。显然你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(这甚至没有考虑更新交换机程序所需的停机时间)。
- 通常，对于嵌入式系统，手工维护(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，这时你就可以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。不定期的人工维护是不可行的，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。我们的程序设计工作中，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，完整的定义只有在试验之后才能得。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“故障会导致极大的损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。“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一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，也无需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(“申请资源”、“分配”)，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)和锁等。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 constrains)：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之前完成，那么我们称它是硬实时(hard real time)的。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，那么就称为软实时(soft real time)系统。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。例如，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（如 `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至少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`（参见 25.3 节）
- 异常（参见 19.5 节）
- 动态类型转换 `dynamic_cast`（参见附录 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）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`。因为我们不能只依赖调试工具的 C++ 程序员发现这类问题。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层语言特性的地步。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（已给定的所有条件）。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(时间或空间)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”：

- 第一法则：不要做优化。
- 第二法则(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，再根据需要仔细测试、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，也不要报太大期望，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需考虑。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 华氏度(54 摄氏度)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，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回者号上的软件和硬件故障，并能在弄清问题后通过软件更新的方式来修复故障，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，都有几十年的研究和开发历史，相关的文献都数以千计。

- 避免资源泄漏：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。

- **复制：**如果某个硬件资源(如计算机、输出设备、轮子等)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备份？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备份，备份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。
- **自检测：**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(参见 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，自检测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。
- **能够迅速离开有错误的代码：**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，但你可以把模块看做一个类、一个库、一个程序或者一台计算机上的所有程序。
- **监控对子系统——**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)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(参见 17.4 节和附录 A.4.2)：

- **静态内存：**是由连接器分配的，其生命期为整个程序的运行期间。
- **栈内存(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(也称为堆)：**用 new 操作分配，用 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（参见 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不会太长（例如，f1 调用 f2，f2 调用 f3，…，调用 fn）。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（参见 15.5 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现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。观察下面程序中内存分配和释放的过程：

```
Message* get_input(Device&);           // make a Message on the free store

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。看看这段代码，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。

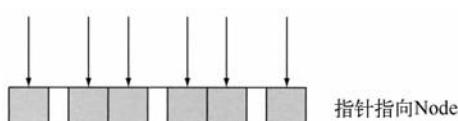
我们假定系统使用一个简单(但并非与实际不符)的内存管理程序。另外假定一个 Message 比一个 Node 稍大。下图展示了动态内存的使用情况，其中 Message 用黑色表示，Node 用灰色表示，而白色表示“空洞”(即“未使用空间”)：



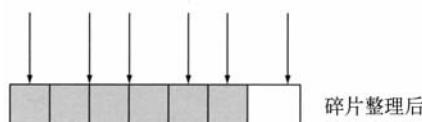
由此可见，每执行一个循环步，我们就会在动态内存中留下一些未用空间(“空洞”)。这些空洞可能只有几个字节大小，但如果不能加以有效利用，其危害与内存泄漏是一样的——即使是微小的泄漏，在长时间运行后也会导致系统崩溃。在内存中，空闲空间分散，形成很多小“空洞”，无法满足新的内存需求的情况，就称为内存碎片。内存管理程序最终会把足够大的“空洞”用尽，只留下无法使用的小空洞。这是任何频繁使用 new 和 delete 的系统长期运行后都会遇到的一个严重问题，最终，内存中布满了无法使用的碎片。此时再执行 new 操作，就需要在大量对象和碎片中搜索足够大的区域，所花费的时间就会急剧增加。显然，这对于嵌入式系统来说是不可接受的。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++ 是直接用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，是不会有多余额外内存空间的。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);
    //... store information in nodes ...
    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。最乐观地估计，既避免碎片，同时又要保证代码仍旧可靠，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，输出创建的对象的地址和大小，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（就像前面那几个图一样），这能帮你更好地理解这个问题。

### 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。在这方面更准确的规则应作为系统编程规范的一部分（参见 25.6 节）

有以下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：**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，其正确性是极难保证的。

### 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，其中灰色表示“已分配的对象”，而白色表示“空闲空间”：



我们可以定义 Pool 如下：

```
template<class T, int N>class Pool { // Pool of N objects of type T
public:
    Pool(); // make pool of N Ts
    T* get(); // get a T from the pool; return 0 if no free Ts
    void free(T*); // return a T given out by get() to the pool
    int available() const; // number of free Ts
private:
    // space for T[N] and data to keep track of which Ts are allocated
    // and which are not (e.g., a list of free objects)
};
```

每个 Pool 对象都包含类型相同的一组对象，对象的数目有上限值。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;

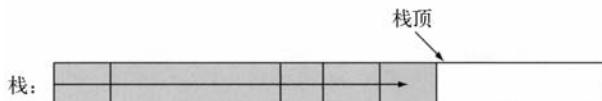
Small_buffer* p = sb_pool.get();
// ...
sb_pool.free(p);
```

程序员应该确保存储池不会被耗尽，“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存储池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(如 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 模板仅仅是存储池一般思想的一种实现，还可以根据需要选择其他实现方式。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，其中灰色表示“已分配内存”，白色表示“空闲空间”：



如上图所示，栈向右“生长”。

定义一个对象栈，与定义一个对象存储池类似：

```
template<class T, int N> class Stack {           // stack of Ts
    // ...
};
```

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 {      // stack of N bytes
public:
    Stack();                  // make an N-byte stack
    void* get(int n);         // allocate n bytes from the stack;
                               // return 0 if no free space
    void free();               // return the last value returned by get() to the stack
    int available() const;    // number of available bytes
private:
    // space for char[N] and data to keep track of what is allocated
    // and what is not (e.g., a top-of-stack pointer)
};
```

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; // 50K worth of storage to be used as a stack

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 17.8 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。

自然地，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 20.9 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 17.8 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，参见附录 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 18.5.1 节所述，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(但很不幸，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) // poor interface design
{
    for (int i = 0; i < sz; ++i) p[i].draw();
}

void f(Shape* q, vector<Circle>& s0) // very bad code
{
    Polygon s1[10];
    Shape s2[10];
    // initialize
    Shape* p1 = new Rectangle(Point(0,0),Point(10,20));
    poor(&s0[0],s0.size()); // #1 (pass the array from the vector)
    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)(此处是正确的)。

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。

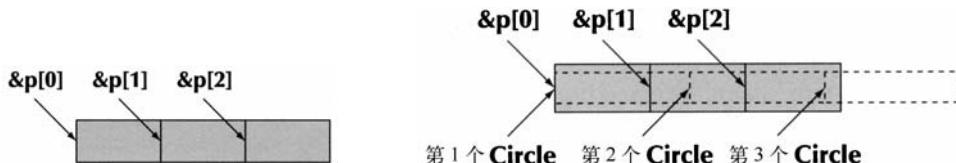
理论上，编译器可以找到其中一些错误(如第二个 poor(p1, 1) 中 p1 == 0 的情形)。但在现实中，我们之所以能免受这些错误的困扰，主要还是因为编译器发现了程序试图定义抽象类 Shape 的对象。但是，这并未解决 poor() 接口方面的问题，因此我们无法松口气。接下来，我们将使用一个非抽象的 Shape，这样就能专注于接口问题了。

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(参见 14.2 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] 等上的对象如下图所示：

就内存地址而言，这些指针的间距为 sizeof(Shape)(参见 17.3.1 节)。但不幸的是，对于此次 poor() 的调用，sizeof(Circle) 大于 sizeof(Shape)，因此内存布局如下图所示：



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，就不会遇到 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 13.8 节和 13.12 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(参见 19.3.3 节)。例如：

```
class Circle : public Shape {/* ... */};

void fv(vector<Shape>&); // OK: implicit conversion from Circle to Shape
void f(Shape &); // error: no conversion from vector<Circle> to vector<Shape>

void g(vector<Circle>& vd, Circle & d)
{
    f(d);           // OK: implicit conversion from Circle to Shape
    fv(vd);         // error: no conversion from vector<Circle> to vector<Shape>
}
```

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，在安全性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式界面等。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，以避免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所存在的那些问题：

```
void general(vector<Shape>&);
```

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` 功能相似但又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、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。

- 它不能显式地转换为一个指针。
- 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<class 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; }

    // default copy operations:
    //     Array_ref doesn't own any resources
    //     Array_ref has reference semantics
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，因此复制操作只复制 (p, size)，而不会复制引用的对象。
- 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, size) 的值，这样就可以改变 Array\_ref 的大小 (很多算法要求指定子区域)。
- 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描述的一个范围。

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 (参见 20.9 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<class T> Array_ref<T> make_ref(T* pp, int s)
{
    return (pp) ? Array_ref<T>(pp,s) : Array_ref<T>(0,0);
}
```

如果我们用一个指针来初始化 `Array_ref`, 那么就必须显式地提供数组的大小。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 节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), 我们同样也应提防空 `vector`:

```
template<class T> Array_ref<T> make_ref(vector<T>& v)
{
    return (v.size()) ? Array_ref<T>(&v[0],v.size()) : Array_ref<T>(0,0);
}
```

这段代码实现了用 `vector` 初始化 `Array_ref`, 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 <class 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]; // error: no elements
```

有了 `Array_ref` 后, 我们就可以重写 25.4.2 中的例程了:

```
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];
    // initialize
    Shape* p1 = new Rectangle(Point(0,0),Point(10,20));
    better(make_ref(s0)); // error: Array_ref<Shape> required
    better(make_ref(s1)); // error: Array_ref<Shape> required
    better(make_ref(s2)); // OK (no conversion required)
    better(make_ref(p1,1)); // OK: one element
    delete p1;
    p1 = 0;
    better(make_ref(p1,1)); // OK: no elements
    better(make_ref(q,max)); // OK (if max is OK)
}
```

新的程序有如下改进:

- 代码更简洁。程序员大多数情况下无需考虑大小, 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(以及其他指针指向的元素数目问题)变得更为明显了, 这里是我们唯

—需要显式地处理大小的地方。

- 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 19.3.2 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 19.3.3 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 19.3.3 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。

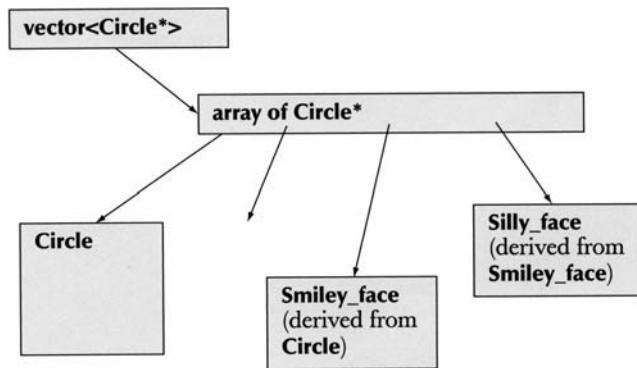
而且，为了防止运行时的多态行为，我们必须通过指针(或者引用)来访问多态对象：better() 中是不该使用 `p[i].draw()` 的。当我们一看到对多态对象使用点操作符而不是箭头( ->) 时，就应该想到可能要出问题了。

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，而不是 `Array_ref < Circle >`、`Array_ref < Shape >` 等。

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已经过测试的工具，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，所以我们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，我们使用了两个 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<sup>\*</sup>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<class T>
class Array_ref {
public:
    // as before

    template<class Q>
    operator const Array_ref<const Q>()
    {
        // check implicit conversion of elements:
        static_cast<Q>(*static_cast<T*>(0));

        // cast Array_ref:
        return Array_ref<const Q>(reinterpret_cast<Q*>(p),sz);
    }

    // as before
};
```

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（参见附录 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];
    // initialize
    Shape* p1 = new Rectangle(Point(0,0),10);
    better2(make_ref(s0));      // OK: converts to Array_ref<Shape*const>
    better2(make_ref(s1));      // OK: converts to Array_ref<Shape*const>
    better2(make_ref(s2));      // OK (no conversion needed)
    better2(make_ref(p1,1));    // error
    better2(make_ref(q,max));  // error
}

```

最后两条语句对指针的使用是错误的，因为两个指针是 Shape<sup>\*</sup> 类型，而 better2() 需要的是一个 Array\_ref < Shape<sup>\*</sup> > 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 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。

总之，我们可以创建简单、安全、易于使用并且高效的接口，来弥补数组的不足。这就是本节的主要目的。“通过间接方式解决每个问题”（引自 David Wheeler）已经被作为“计算机科学第一定律”。这就是我们解决这个接口问题所采用的方法。

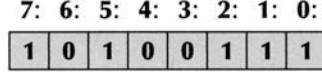
## 25.5 位、字节和字

在本书前面的章节中，我们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做特定类型的对象，如 double、string、Matrix 以及 Simple\_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。

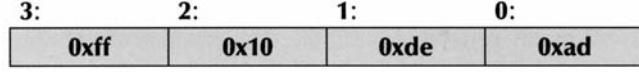
如果你对整数的二进制和十六进制表示的相关知识不太了解，请参考附录 A.2.1.1。

### 25.5.1 位和位运算

一个字节可以看做 8 个位的序列：



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，一个字也可看做 4 个字节的序列：



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 10 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>（参见 24.2.1 节）以确保类型大小

不会弄错。

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做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

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（参见 21.6.5 节）
- 文件：海量的二进制位（参见 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
$\wedge$	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 $x\wedge y$ 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
$\sim$	补	$\sim x$ 的第 $n$ 位是 $x$ 的第 $n$ 位的取反

你可能觉得将“异或”( $\wedge$ ，有时称为“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，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（参见附录 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 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(参见 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; // complement of a`

a: 0xaa

~a: 0x55

`unsigned char b = 0x0f;`

`unsigned char x1 = a&b; // a and b`

a: 0xaa

b: 0xf

a&b: 0xa

`unsigned char x2 = a^b; // exclusive or: a xor b`

a: 0xaa

b: 0xf

a^b: 0xa5

```
unsigned char x3 = a<<1; // left shift 1

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，可以看做是从第 0 位右边“移来”的。而原来的最高位(第 7 位)被简单丢弃。

```
unsigned char x4 == a>>2; // right shift 2

a: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

 0xaa

a>>2: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

 0x2a
```

最高两位(第 6 位和第 7 位)都填入了 0，可以看做是从第 7 位左边“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位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繁琐了。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()
{
    int i;
    while (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那么多二进制位，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由 0 和 1 组成的字符串。例如：

```
bitset<4> flags = 0xb;
bitset<128> dword_bits(string("1010101010101010"));
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); // may throw 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; // and
b1 = b2|b3; // or
b1 = b2^b3; // xor
b1 = ~b2; // complement
b1 = b2<<2; // shift left
b1 = b2>>3; // shift right
```

基本上，对于位运算而言，bitset 就像 unsigned int（参见 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; // read a bitset from input
cout<<bitset<8>('c'); // output the bit pattern for the character 'c'
```

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  $2^7$ ：

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：

7:	6:	5:	4:	3:	2:	1:	0:
1	0	1	0	0	1	1	1

```
int main()
{
    const int max = 10;
    bitset<max> b;
    while (cin>>b) {
        cout << b << '\n';
        for (int i=0; i<max; ++i) cout << b[i]; // reverse order
        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）用来作为“符号位”：



如果符号位为 1，就表示负数。二进制补码表示法事实上已经成为标准方法。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数值, 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  $32 * 1024$  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  $2 * 1024 * 1024 * 1024$  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';
```

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;      // very large
    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。

基本上, 我们将无符号数当做整数来使用有两个原因, 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（在 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  $(-1 + 2 == 1)$ ，ui2 也被赋予同样的值。ui2 的计算结果应该会让你感到惊讶：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使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); // print i as an integer followed by a space
```

这段程序会输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; // doesn't fit into a char
char c = si; // implicit conversion to 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 是无符号的，因为 c 的行为与 uc 一致，而与 sc 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。

插一句：我们为什么要引入 print() 函数？我们可以试试：

```
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，而不是其整数值。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<class 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(bytten)”。

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) // assume 16-bit, 2-byte short integer
{
    unsigned char left = val&0xff;           // leftmost (least significant) byte
    unsigned char right = (val>>8)&0xff; // rightmost (most significant) byte
    // ...
    bool negative = val&0x8000;           // sign bit
    // ...
}

```

这种运算是很常见的，通常称为“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位)，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：

	<b>out_of_color</b>	<b>16</b>	<b>0x10</b>	<b>0001 0000</b>
	<b>out_of_black</b>	<b>8</b>	<b>0x8</b>	<b>0000 1000</b>
	<b>busy</b>	<b>4</b>	<b>0x4</b>	<b>0000 0100</b>
	<b>paper_empty</b>	<b>2</b>	<b>0x2</b>	<b>0000 0010</b>
	<b>acknowledge</b>	<b>1</b>	<b>0x1</b>	<b>0000 0001</b>

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 becomes 24 (16+8)
x |= paper_empty; // x becomes 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，例如：

```

if (x & out_of_color) { // is out_of_color set? (yes, it is)
// ...
}

```

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); // x becomes 24

```

此时，y 的内容就是 x 的第 3 位和第 4 位(`out_of_black` 和 `out_of_color` 对应第 3、4 位)。

将 enum 作为二进制位集合来使用，是一种十分常用的方法。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); // the cast is necessary

```

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”。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(bitfields)。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：



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 4 个标识位，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 { // R6000 Physical Page Number
    unsigned int PFN : 22;      // Page Frame Number
    int : 3;                  // unused
    unsigned int CCA : 3;      // Cache Coherency Algorithm
    bool nonreachable : 1;
    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，但如果我没有手册的帮助，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) { // contents changed
        // copy to disk
        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;           // extract 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;    // extract 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，这并不是一种好的风格。

## 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(参见 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”对其进行解密，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)
{
    unsigned long y = v[0];
    unsigned long z = v[1];
    unsigned long sum = 0;
    unsigned long delta = 0x9E3779B9;
    unsigned long n = 32;
    while(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。据我们所知，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)
{
    unsigned long y = v[0];
    unsigned long z = v[1];
    unsigned long sum = 0xC6EF3720;
    unsigned long delta = 0x9E3779B9;
    unsigned long n = 32;
    // sum = delta<<5, in general sum = delta * n
    while(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加密：

```

int main() // sender
{
    const int nchar = 2*sizeof(long); // 64 bits
    const int kchar = 2*nchar; // 128 bits

    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'; // pad key
    ifstream inf(infile.c_str());
    ofstream outf(outfile.c_str());
    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; // use hexadecimal output
        if (++count == nchar) {
            encipher(inptr,outptr,k);
            // pad with leading zeros:
            outf << setw(8) << setfill('0') << outptr[0] << ' '
                << setw(8) << setfill('0') << outptr[1] << ' ';
            count = 0;
        }
    }

    if (count) { // pad
        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，每次都将 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 unsinged 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（参见 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 d3bcede62 4fdb7dc8 43d565e5
f1d3f026 b2887412 97580690 d2ea4f8b 2d8fb3b7 936cfa6d
6a13ef90 fd036721 b80035e1 7467d8d8 d32bb67e 29923fde
197d4cd6 76874951 418e8a43 e9644c2a eb10e848 ba67dc8
7115211f dbe32069 e4e92f87 8bf3e33e b18f942c c965b87a
44489114 18d4f2bc 256dalbf c57b1788 9113c372 12662c23
eeb63c45 82499657 a8265f44 7c86aae 7c80a631 e91475e1
5991ab8b 6aedbb73 71b642c4 8d78f68b d602bfe4 dleadde7
55f20835 1a6d3a4b 202c36b8 66ale0f2 771993f3 11d1d0ab
74a8cf4 4ce54f5a e5fda09d acbdf110 259ala19 b964a3a9
456fd8a3 1e78591b 07c8f5a2 101641ec d0c9d7e1 60dbeb11
b9ad8e72 ad30b839 201fc553 a34a79c4 217ca84d 30f666c6
d018e61c d1c94ea6 6ca73314 cd60def1 6e16870e 45b94dc0
d7b44fc9 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 dcdbd64c5 dddale73
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 f8fbff67 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; // terminator
unsigned long* outptr = reinterpret_cast<unsigned long*>(outbuf);
inf.setf(ios_base::hex ,ios_base::basefield); // use hexadecimal input

while (inf>>inptr[0]>>inptr[1]) {
    decipher(inptr,outptr,k);
    outf<<outbuf;
}

```

注意这条语句：

```
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()` 的接口风格并不很符合我们的习惯。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。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，而无需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衷，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 Camel-CodingStyle）“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“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（需要很多人，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（参见 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。

## 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: 2003 (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。

## 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，常量通常被看做接口的一部分；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。

### 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);      // violation
int a = 7;    // OK
x = a+7;    // OK
f(x,9);    // OK
```

例子：

```
if (p < q) cout << *p;    // violation
```

例子：

```
if (p < q)
    cout << *p;    // OK
```

**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 等是按习惯方式使用，可以认为是有描述性的。

使用下划线风格(number\_of\_elements)而不是字头缩略风格(numberOfElements)。

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，来自库中的名字是可以使用的。

例外：宏名用于保护#include 不被重复包含。

**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 // 违反了原则。

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 // 违反了原则。

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。

## 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; } // violation: var hides var
```

**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; // violation: var is not initialized
```

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。

**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。

## 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，必须为其定义或禁止（即不能使用默认的）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。

### 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)。

### 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]; // violation
```

例子：

```
++i;  
int x = v[i]; // OK
```

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; // OK
```

例子：

```
if (a<b || c<=d) // violation: parenthesize (a<b) and (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按顺序编号，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

已经有很多 C++ 编码规范，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：

Henricson, Mats, and Erik Nyquist. *Industrial Strength C++ : Rules and Recommendations*. Prentice Hall, 1996. ISBN 0131209655. 这是一个电信公司编制的规范。不幸的是，其中的编码原则有些过时了，因为这本书的出版日期早于 ISO C++ 标准，特别是它没有将模板纳入讨论范围。以现在的眼光来看，这本书中的编码原则都应该重写了。

Lockheed Martin Corporation. “Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program.” 文档编号 2RDU00001 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.research.att.com/](http://www.research.att.com/~)

bs/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做“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.research.att.com/~bs/PerformanceTR.pdf](http://www.research.att.com/~bs/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(最低有效位置为 1)。

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开它? 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? 它为什么会导致错误?
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. 0x7 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

地址	加密	存储池	位	异或
可预测性	位域	小设备	实时	bitset
垃圾收集器	资源	编码规范	硬实时	软实时
嵌入式系统	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(参见 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(参见 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 E-mail。
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(参见 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(不要简单地复制 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 节和 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，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很多都是嵌入式系统，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。