

TURING

图灵计算机科学丛书

C语言程序设计 现代方法

C Programming: A Modern Approach

[美] K. N. King 著
吕秀锋 译



人民邮电出版社
POSTS & TELECOM PRESS

TURING

图灵计算机科学丛书

C语言程序设计 现代方法

C Programming: A Modern Approach

[美] K. N. King 著
吕秀峰 译

人民邮电出版社
北京

计算机精品学习资料大放送

[软考官方指定教材及同步辅导书下载](#) | [软考历年真题解析与答案](#)

[软考视频](#) | [考试机构](#) | [考试时间安排](#)

[Java 一览无余：Java 视频教程](#) | [Java SE](#) | [Java EE](#)

[.Net 技术精品资料下载汇总：ASP.NET 篇](#)

[.Net 技术精品资料下载汇总：C#语言篇](#)

[.Net 技术精品资料下载汇总：VB.NET 篇](#)

[撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程](#)

[Visual C++\(VC/MFC\)学习电子书及开发工具下载](#)

[Perl/CGI 脚本语言编程学习资源下载地址大全](#)

[Python 语言编程学习资料\(电子书+视频教程\)下载汇总](#)

[最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载](#)

[数据库管理系统\(DBMS\)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇](#)

[平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载](#)

[最强 HTML/xHTML、CSS 精品学习资料下载汇总](#)

[最新 JavaScript、Ajax 典藏级学习资料下载分类汇总](#)

[网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总](#)

[UML 学习电子资下载汇总 软件设计与开发人员必备](#)

[经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通](#)

[天罗地网：精品 Linux 学习资料大收集\(电子书+视频教程\) Linux 参考资源大系](#)

[Linux 系统管理员必备参考资料下载汇总](#)

[Linux shell、内核及系统编程精品资料下载汇总](#)

[UNIX 操作系统精品学习资料<电子书+视频>分类总汇](#)

[FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频](#)

[Solaris/OpenSolaris 电子书、视频等精华资料下载索引](#)

[>> 更多精品资料请访问大家论坛计算机区...](#)



图书在版编目（CIP）数据

C 语言程序设计：现代方法 / (美) 金 (King, K. N.) 著；
吕秀锋译。—北京：人民邮电出版社，2007.11(2009.3 重印)
(图灵计算机科学丛书)
ISBN 978-7-115-16707-1

I . C … II . ①金 … ②吕 … III . C 语 言 — 程 序 设 计
IV . TP312

中国版本图书馆 CIP 数据核字 (2007) 第 130773 号

Copyright © 1996 W. W. Norton & Company, Inc.

All rights reserved. No portion of this book may be reproduced in any form or by any means without the prior written permission of the publisher.

本书中文版由W. W. Norton公司授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

内 容 提 要

时至今日，C语言仍然是计算机领域的通用语言之一，但今天的C语言已经和最初的时候大不相同。本书最主要的一个目的就是通过一种“现代方法”来介绍C语言，实现客观评价C语言、强调标准化C语言、强调软件工程、不再强调“手工优化”、强调与C++语言的兼容性的目标。本书分为C语言的基础特性、C语言的高级特性、C语言标准库和参考资料4个部分。每章都有“问与答”小节，给出一系列与本章内容相关的问题及其答案，此外还包含适量的习题。

本书是为大学本科阶段的C语言课程编写的教材，同时也非常适合作为其他一些课程的辅助用书。

图灵计算机科学丛书 C 语 言 程 序 设 计：现 代 方 法

◆ 著 [美]K. N. King
译 吕秀锋
责任编辑 杨海玲
◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市海波印务有限公司印刷
◆ 开本：787×1092 1/16
印张：26.5
字数：748 千字 2007 年 11 月第 1 版
印数：5 001—6 000 册 2009 年 3 月河北第 2 次印刷
著作权合同登记号 图字：01-2007-3589 号
ISBN 978-7-115-16707-1/TP

定价：55.00 元

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

反盗版热线：(010) 67171154

对本书的赞誉

“问：有什么好的学习C的书？答：……许多活跃在新闻组comp.lang.c的人都推荐K. N. King写的《C语言程序设计：现代方法》。”

——C FAQs

“本书相对于一般的编程书而言简直是鹤立鸡群。它将清晰简明的阐述与对深入问题的深刻讨论完美结合起来。期待作者的更多著作……”

——Mike Biglan

“总之，这是一部杰出的教材，非常适合用于大学本科教学。书中的源代码示例也非常精彩，清晰地说明了C语言的工作原理。”

——*Mathematics and Computer Education*杂志

“我觉得这是我看过的最好的C语言书籍……真希望某天在我的课上可以使用它。”

——Arthur B. Maccabe, 新墨西哥大学

“我在看了USENET上的推荐后购买了本书。真是一本好书！我曾经尝试过Kochan的《C语言编程》，但是到指针和位运算部分就无法继续了。本书挽救了我！”

——Greg McNichol

“本书很好地实现了作者的既定目标：一本示例丰富的理想课堂教材……本书使我受益良多，我不仅了解到许多新知识，还大大改变了对C语言的原有认识。”

——*Computing Reviews*杂志

“作者的教学风格非常独特，各种语言要素都在需要的时候再介绍，这样初学者不会一下子就被大量信息所淹没，而比较复杂的主题放在较后章节介绍，很自然地就达到了循序渐进的效果。书中对指针的阐述是我所见过的最清晰的。……本书有很多优点，我希望其他作者也能借鉴。”

——ACCU.org (C/C++用户协会)

“本书之所以广受欢迎，是因为作者不仅精彩地阐述了C语言本身，而且在其间融合了大量有关C语言实际编程的宝贵经验和真知灼见。”

——*Choice*杂志

“有了本书，学习C语言不再困难了。系统、全面、易懂，习题也非常精彩。”

——Amazon书评

“我认为King的书在这一领域是出类拔萃的……我还没见过哪一本书像他这本这样具有精确且高水平的阐述。”

——Manuel E. Bermudez, 佛罗里达大学

“我所在的加州大学戴维斯分校使用本书作为教材。我非常喜欢，非常好懂，尤其是例子。”

——Tim Tully

“这本书实在是太赏心悦目了……写作手法如此清晰、直接……”

——Carolyn Rosner, 威斯康星大学

“我见过的最好的C语言教材！可以与K&R媲美。”

——Kevin Davis

“我已经在教学中使用本书多年。它深受学生喜爱，的确是一部杰作。我尤其欣赏书中明确地深入讨论了许多其他书里没有或者无力涉及的实践性主题。”

——Diane Horton, 加拿大多伦多大学

译者序

C语言是国际上最流行的、应用最广泛的高级编程语言之一。时至今日，它依然保持着旺盛的生命力，深受广大程序员的欢迎。作为一种“个性鲜明”的编程语言，C语言既具有高级语言的优点，又有着低级语言的特性，因此它在编写操作系统、编译器等系统软件方面有着得天独厚的优势。

目前，市面上介绍C语言的书籍众多，其中很多书的内容可以用“事无巨细，面面俱到”来形容。但是，熟记编程语言的语法和规则并不是掌握程序设计的捷径，只有通过大量的实践练习才能真正做到学以致用，并最终达到灵活高效地开发软件的目标。因此，本书从软件工程的角度，运用简洁、易懂的语言，结合丰富且实用的程序样例来介绍C语言，初学者或有经验的程序员都会从中受到启发。除了上述特色外，本书的另外一大亮点就是每个章节后面丰富的练习和问答题，这些都是原书作者结合多年的软件开发经验和教学经验提炼出来的，具有极强的针对性，非常适合作为高等院校C语言课程的教材使用。

在本书的翻译过程中，得到了许多人的帮助。首先要感谢李凌、陈寿福、马锐对本书内容提出的修改意见。同时，要感谢人民邮电出版社的编辑们为本书的翻译提供的大力支持和帮助，特别要感谢杨海玲编辑在翻译、审校过程中提供的建议。最后，要感谢我的家人，正是他们的支持和鼓励才使本书的翻译工作顺利完成。

原书的内容严谨，结构清晰，具有较强的理论性和实践性。译者力求反映本书的特点和原貌，但由于时间关系及水平所限，疏漏或不当之处在所难免，敬请广大读者批评指正。

2007年7月

前　　言

在计算机领域中，把显而易见的转变为有实用价值的，这一过程是“挫折”一词的生动体现。

我首先要敞开心扉告诉大家：多年来我与C语言之间一直保持着一种“爱恨交加”的关系。一方面，我享受着轻松自如地编写C程序的快乐，并且酷爱当今多种C编译器提供的各式开发环境；另一方面，我讨厌自己在编程中容易犯错，也厌倦了要留意C语言编程中常常要求的细节。更重要的是，我憎恶许多C程序员蔑视其他编程语言的态度。让我们一起客观地评价一下C语言：C语言不是编程语言的终结（当然，C++语言也不是）；然而，C语言却是每位软件开发人员都应该掌握的一种编程语言。无论褒贬如何，C语言都已成为计算机领域的通用语言。

1975年，当C语言还是一种新兴的、尚未完全成熟的编程语言时，我就开始接触它了。随后，我有好几年没有和C打交道。然而，C语言被标准化之后，我决定重新审视一下这种语言。结果我惊喜地发现，一些C语言中致命的缺陷已经在标准化过程中得到了修正。（当然，C语言仍旧有不少值得改进的地方！）于是，我决定写一本书来展示C语言的崭新面貌，并且同时收集整理一下过去多年来C程序员所创造的智慧结晶。

目的

下面这些是本书试图实现的目标：

- **清晰易读，而且尽可能带有趣味性。**对普通读者来说，许多C语言的书籍都过于简洁了。甚至某些C语言书籍不是编写得一塌糊涂，就是平淡无趣。而我试图对C语言进行一种清晰、全面的讲解，并且决定用适当的幽默来激发读者阅读的兴趣。
- **适用于广泛的读者群。**我假设阅读本书的读者都只有一点点编程经验，而且他们都尚未精通某种具体的编程语言。我尽量减少“行话”而改用通俗易懂的词汇来定义用到的术语。同时，为了鼓励初学者，我还尝试了将某些高级内容从基本主题中分离出来。
- **有权威性，但不是学究气十足。**为了减少读者的麻烦，我在书中尽量涵盖了所有标准C的特性和库函数，包括信号、setjmp/longjmp和可变长度实际参数列表。同时，为了避免给读者造成负担，我还忽略了一些不必要的细节。
- **具备简单易学的组织结构。**根据多年教授C语言的经验，我强调循序渐进地展示C语言特性的重要性。针对有一定难度的主题，我采用了螺旋式的介绍方法。也就是，对于较难的主题先进行简要介绍，然后，在后续章节中再多次介绍该主题，逐渐增加细节内容。本书的进度是经过深思熟虑的。每章都按照循序渐进的方式进行组织，并且前后内容由浅入深，相互呼应。对于大多数学生来说，这种循序渐进的方法是最合适的：既避免产生无聊的内容，又防止出现“信息超载”。
- **深入探讨语言特性。**我不是仅描述语言的每个特性，或者只展示几个简单的特性应用的例题，而是尝试深入讲解每一种特性，并且探讨如何将这些特性应用到实际问题中。

- 强调编写风格。对每位C程序员来说，采用一种统一的代码编写风格是非常重要的。但是，与其指定某种风格，我更愿意给出多种编写风格，让读者自己做出合理的选择，因为了解多种编写风格对阅读别人的程序是很有帮助的（有些程序员经常要花费大量时间阅读别人的程序）。
- 避免依赖任何特定的计算机、编译器或者操作系统。因为C语言可以应用在如此多样的平台上，所以我试图避免编写的程序依赖于任何特定的计算机、编译器或操作系统。当然，使用C这样的语言完全忽略机器的细节也是不可能的。当此类问题不可避免时，我都以16位计算机和32位计算机的两种体系结构进行举例说明。当示例要依赖于某种特定操作系统时，我会讨论DOS和UNIX两种系统。
- 用图示的方法阐明关键概念。因为图在理解C语言方方面面都起着至关重要的作用，所以我在书中加入了尽可能多的图。特别是我还试图通过图显示运算中不同阶段的数据状态来动态地描述算法。

现代方法到底是什么

本书最主要的一个目的就是想通过一种“现代方法”来介绍C语言。我试图通过以下这些途径来达成目标：

- 正确看待C语言。我没有把C语言看成是唯一值得学习的编程语言，而是把它作为众多有用语言中的一种进行介绍。我在书中提到了最适合用C语言编写的程序类型。此外，我还展示了如何扬长避短地使用C语言。
- 强调标准化C语言。我较少关注旧版C语言。只在某些章节偶尔提到经典（K&R）C，多在“问与答”部分。附录C列出了标准C和经典C之间的主要差异。
- 揭穿神话。现今的某些书籍的作者常常会在C语言某些常见的假设上争论不休，而我却乐于揭穿C语言的某些神话，或者说是想对长久以来构成C语言传说的某些信条提出挑战。例如，有种说法始终认为指针的算术运算一定比数组下标操作快。我重新审查了C语言的旧惯例，并且保留了那些仍然有帮助的惯例。
- 强调软件工程。我把C语言视为一种成熟的软件工程工具，因此我强调如何运用C语言来处理大规模程序开发过程中产生的问题。我坚持程序要易读、可维护、可靠且容易移植，同时还特别看重信息隐藏。
- 不是在一开始就介绍C语言的低级特性。虽然这些特性曾为早期C语言的各种系统编程提供了便利，但是现在它们已经不再适宜使用，因为C语言已经应用于大量不同的程序中。本书没有像其他大多数C语言书籍那样把这部分内容放在前面章节进行介绍，而是推迟到第20章再进行讲述。
- 不再强调“手工优化”。某些书籍指导读者编写并不简单、清晰的代码，仅仅是为了稍稍提高程序效率。然而，面对现今C语言编译器的大量优化技术，这种代码优化工作常常不再必要；事实上，它们适得其反。
- 强调与C++语言的兼容性。有关这方面的内容，我会稍后进行详细介绍。

“问与答”部分

每章的末尾都有一个“问与答”部分，汇集了与本章内容相关的问题及其答案。“问与答”部分的内容包括：

- **常见问题。**我尽力回答了某些频繁出现在我的课堂里、其他书籍中，以及和C语言相关的新闻组里的问题。
- **对一些难以理解的问题的进一步讨论和澄清。**虽然具有多种编程语言经验的读者会满足于简明扼要的说明和少量的示例，但是缺乏经验的读者却需要更多的内容以帮助理解。
- **非主流的问题。**某些问题所引发的并不是所有读者都感兴趣的技术讨论。
- **某些对普通读者来说过于超前或深奥的内容。**这类问题都用星号 (*) 进行了标记。好奇且有一定编程经验的读者也许希望立刻深入研究这些问题，而另外一些读者则需要在首次阅读时跳过这部分内容。提示：这类问题往往引用后续章节的内容。
- **多种C语言编译器的常见差异。**我讨论了一些经常用到的（而非标准的）特性。DOS系统和UNIX系统都对这类特性提供支持。

“问与答”中的某些问题与对应章中某些特定的内容直接相关，这些特定内容会在对应位置用一种特殊的图标**Q&A**标记，从而提示读者有进一步的信息。

其他特色

除了“问与答”部分，我还加入了另外一些有用的特色，其中一些特色用简单但是独特的图标进行了标识。

- **警告 (Δ)** 警示读者一些常见的陷阱。C语言以其陷阱多而出名。要记录所有的陷阱非常困难。我试图挑选出最常见或最重要的陷阱供大家参考。
- **引用 (>前言)** 提供一种类似超文本的能力来查找信息。多数引用指向稍后的章节中才能提到的主题，但是确有某些引用指向先前的主题供读者回顾。
- **惯用法**是经常可以在C语言程序中看到的代码模式。它常被标记出来以便于速查参考。
- **可移植性技巧**为编写不依赖于特定的计算机、编译器或操作系统的程序所需的技巧和心得。
- **附加说明**包含一些严格来讲并不属于C语言的内容，但却是每位熟练的C程序员都应该知道的知识，比如无符号整数、IEEE浮点数标准以及Unicode编码等。（附加说明的示例可参见本页下面的“源代码”说明。）
- **附录**提供有价值的参考资料信息。

程序

选择作为例证的程序并不是件轻松的工作。如果程序过于简洁和做作，那么读者将无法体会如何将这些特性应用于现实世界里。一方面，如果程序过于真实，那么它的要点将很容易被忽略在过多的细节中。我采取了折中方案。在首次介绍概念时，先通过小而简单的示例使内容清晰，再逐步建立完整的程序。我没有使用过长的程序，因为根据我个人的经验，授课者没有时间介绍这些，而学生们也不会有耐心去阅读。但是，我没有忽视创作大程序所引发的问题，这些在第15章和第19章中进行了详细的介绍。

源 代 码

本书中所有程序的源代码都可以从图灵网站 (<http://www.turingbook.com>) 下载。有关本书原版的校正、修改和最新消息可以从<http://www.gsu.edu/~matknk/cbook>获取。

C++语言的介绍

本书从一开始就考虑到要完全兼容C++语言，因此读者不会被培养出那些稍后必须忘掉的习惯。本书通过下面3种方式为读者继续学习C++语言做了铺垫：

- 强化现代设计规则，比如信息隐藏。
- 一些分散在书中的C++语言的简要论述。其中，每次论述都用特殊的**C++**符号标记。
- 第19章提供的详细的C++语言概述。

读者

本书是为大学本科阶段的C语言课程编写的教材。具有其他高级语言或汇编语言的编程经验会很有帮助，不过这些经验对于会用计算机的读者（我的编辑称他们为“熟练的初学者”）来说并不是必需的。

因为本书内容齐备，自成一体，并且既可用于学习又可作为参考，所以它非常适合作为其他一些课程的辅助读物，如数据结构、编译器设计、操作系统、计算机图形学及其他要用C语言进行项目设计的课程。

“问与答”部分以及对实际问题的强调，使得本书也可以引起另外一些读者的兴趣，如培训班学员，或是自学C语言的人。

组织结构

本书分为4个部分：

- **C语言的基础特性。**第1章~第10章包含的C语言内容足够帮助读者编写出使用数组和函数的单个文件程序。
- **C语言的高级特性。**第11章~第20章建立在前面各章的内容上。这些章的内容有一定的难度，深入介绍了指针、字符串、预处理器、结构、联合、枚举以及C语言的低级特性。此外，第15章和第19章提供了编程设计方面的指导。
- **C语言标准库。**第21章~第26章集中介绍C语言库。这个庞大的函数集合是与每种编译器一起提供的。虽然一部分内容适合讲解，但这些章更适合作为参考资料来使用。
- **参考资料。**附录A介绍了C语言的完整语法，并用注解来解释一些较为模糊的论点。附录B给出了C语言运算符的完整列表。附录C描述了标准C和经典C之间的差异。附录D按字母顺序列出了C标准库中的全部函数，每个都予以充分说明。附录E列出了ASCII字符集。还有一个带注解的参考文献为读者指明其他信息的来源。

C语言的全面课程应该按顺序讲授前20章的内容，并且根据需要选取第21章~第26章中的某些主题。短期课程可以忽略以下一些主题而不会失去内容的连贯性：9.6节（递归函数）、12.4节（指针和多维数组）、14.5节（其他指令）、17.7节（指向函数的指针）、第19章（程序设计）以及20.2节（结构中的位域）和20.3节（其他低级技术）。

练习

作为教材，拥有大量多样化的精选习题显然是非常必要的。我准备了300多道难度各异的习

题。有些习题是简短的训练题，虽然它们不是最激动人心的（事实上，它们可能是完全无趣的），但是我认为这些题对培养C语言开发技巧是必要的，就如同词汇训练是外语教材必备的，或者在代数教材中数学问题也是必要的一样。除了训练题，我还加入了大量的简答题和编程练习。虽然简答题的答案通常都很简短，但是此类习题比训练题需要更多的思考。编程练习要求读者编写小程序或大程序中的某个片段。

少数练习没有明确的答案（有些特别挑剔的人称这些是“刁钻问题”）。因为C语言程序经常包含大量这类代码案例，所以我认为有必要提供一些这样的练习，并用星号(*)进行了标注。对于有星号的练习一定要小心：要么格外小心，认真考虑，要么干脆绕开它。

反馈

为了本书能够精确，我付出了极大的努力。然而，任何书籍都不可避免地会有一些错误。如果读者发现了错误，请通过knking@gsu.edu这个电子邮箱或以下地址与我联系。

K. N. King

Department of Mathematics and Computer Science

Georgia State University

University Plaza

Atlanta, GA 30303-3083

我也同样期望听到读者的其他反馈，比如，哪些内容你最认可，需要删除哪些内容，又或希望添加哪些内容等。

致谢

首先，我要感谢本书的编辑，Norton出版社的Joe Wisonvsky，非常欣赏他的聪明才智和良好的鉴赏力。同时，还要感谢我的前任编辑Jim Jordan，他认为这世界需要一本新的C语言书。还有一位要感谢的是Norton出版社的Deborah Gerish，感谢她对书稿所做的文字编辑工作。

本书的前身是我为TopSpeed C编译器编写的指导手册。我要感谢Karin Ellison和Niels Jensen，正是他们让我有机会编写那本手册，并且允许我在本书中再次使用其中的资料。

我还要感谢下面这些对书稿进行部分或全部审稿的人：

Susan Anderson-Freed和Lisa J. Brown，伊利诺伊韦思利安大学

Manuel E. Bermudez，佛罗里达大学

Steven C. Cater，佐治亚大学

Patrick Harrison，美国海军学院

Brian Harvey，加利福尼亚大学伯克利分校

Henry H. Leitner，哈佛大学

Darrell Long，加利福尼亚大学圣克鲁兹分校

Arthur B. Maccabe，新墨西哥大学

Carolyn Rosner，威斯康星大学

Patrick Terry，罗德斯大学

在这里需要特别提到的是Brian Harvey和Patrick Terry，他们提供了超出职责范围的协助，对他们所做的详尽注释我感激不尽，同时我保证会原谅他们对我偶尔严厉的评论。

我也从亚特兰大的朋友和同事那里得到了大量有价值的反馈信息。他们是：一直乐于讨论

C和C++语言细节的Geoff George，在最终书稿中发现不少错误的Marge Hicks，以及对早期书稿发表很多有用意见的Scott Owen。衷心感谢我的部门主管Fred Turner给予我的支持与鼓励。感谢许多学生提供的反馈，包括用敏锐的眼光阅读了早期书稿的Terry Turner，以及Nina Dalal、Jay Schneider和Michael Sigmond。

感谢Susan Cole不仅仔细地阅读了全部书稿，还为我创造了便于写作的良好家庭氛围。没有她的爱和理解，我是不可能完成本书的。我还必须感谢我的猫咪Bronco和Dennis的帮助，在写作本书的过程中，它们一直陪伴着我。

最后，我还要感谢已故的Alan J. Perlis^①。他的警句出现在本书每一章的开始。在20世纪70年代中期我在耶鲁大学期间，曾经有幸在Alan的指导下进行了短暂学习。我想当他发现他的警句出现在一本C语言书中时一定会非常高兴的。

① Alan J. Perlis (1922—1990) 是计算机科学先驱，1966年首届图灵奖得主。——编者注

目 录

| | |
|----------------------|----|
| 第1章 C语言概述 | 1 |
| 1.1 C语言的历史 | 1 |
| 1.1.1 起源 | 1 |
| 1.1.2 标准化 | 1 |
| 1.1.3 C++语言 | 2 |
| 1.2 C语言的优缺点 | 3 |
| 1.2.1 C语言的优点 | 3 |
| 1.2.2 C语言的缺点 | 4 |
| 1.2.3 高效地使用C语言 | 5 |
| 问与答 | 5 |
| 第2章 C语言基本概念 | 7 |
| 2.1 编写一个简单的C程序 | 7 |
| 2.1.1 程序：显示双关语 | 7 |
| 2.1.2 编译和链接 | 8 |
| 2.2 简单程序的通用形式 | 8 |
| 2.2.1 指令 | 9 |
| 2.2.2 函数 | 9 |
| 2.2.3 语句 | 10 |
| 2.2.4 显示字符串 | 10 |
| 2.3 注释 | 11 |
| 2.4 变量和赋值 | 12 |
| 2.4.1 类型 | 12 |
| 2.4.2 声明 | 12 |
| 2.4.3 赋值 | 13 |
| 2.4.4 显示变量的值 | 13 |
| 2.4.5 程序：计算箱子的空间重量 | 13 |
| 2.4.6 初始化 | 14 |
| 2.4.7 显示表达式的值 | 15 |
| 2.5 读入输入 | 15 |
| 2.6 定义常量 | 16 |
| 2.7 标识符 | 17 |
| 2.8 C语言程序的布局 | 18 |
| 问与答 | 20 |
| 练习 | 21 |
| 第3章 格式化的输入/输出 | 23 |
| 3.1 printf函数 | 23 |

| | |
|--------------------------|----|
| 3.1.1 转换说明 | 24 |
| 3.1.2 程序：用printf函数格式化数 | 25 |
| 3.1.3 转义序列 | 25 |
| 3.2 scanf函数 | 26 |
| 3.2.1 scanf函数的工作方法 | 27 |
| 3.2.2 格式串中的普通字符 | 28 |
| 3.2.3 混淆printf函数和scanf函数 | 29 |
| 3.2.4 程序：计算持有的股票的价值 | 29 |
| 问与答 | 30 |
| 练习 | 31 |
| 第4章 表达式 | 33 |
| 4.1 算术运算符 | 33 |
| 4.1.1 运算符的优先级和结合性 | 34 |
| 4.1.2 程序：计算通用产品代码的校验位 | 35 |
| 4.2 赋值运算符 | 36 |
| 4.2.1 简单赋值 | 36 |
| 4.2.2 左值 | 37 |
| 4.2.3 复合赋值 | 37 |
| 4.3 自增运算符和自减运算符 | 38 |
| 4.4 表达式求值 | 39 |
| 4.5 表达式语句 | 41 |
| 问与答 | 42 |
| 练习 | 43 |
| 第5章 选择语句 | 45 |
| 5.1 逻辑表达式 | 45 |
| 5.1.1 关系运算符 | 45 |
| 5.1.2 判等运算符 | 46 |
| 5.1.3 逻辑运算符 | 46 |
| 5.2 if语句 | 47 |
| 5.2.1 复合语句 | 48 |
| 5.2.2 else子句 | 48 |
| 5.2.3 级联式if语句 | 49 |
| 5.2.4 程序：计算股票经纪人的佣金 | 50 |
| 5.2.5 “悬空else”的问题 | 51 |
| 5.2.6 条件表达式 | 51 |
| 5.2.7 布尔值 | 52 |

2 目 录

| | | | |
|----------------------------------|-----------|--------------------------------|------------|
| 5.3 switch语句..... | 53 | 练习 | 95 |
| 5.3.1 break语句的作用 | 55 | | |
| 5.3.2 程序：显示法定格式的日期 | 55 | | |
| 问与答..... | 56 | 第 8 章 数组 | 98 |
| 练习..... | 58 | 8.1 一维数组 | 98 |
| 第 6 章 循环 | 61 | 8.1.1 数组下标 | 98 |
| 6.1 while语句 | 61 | 8.1.2 程序：数列反向 | 100 |
| 6.1.1 无限循环 | 62 | 8.1.3 数组初始化 | 100 |
| 6.1.2 程序：显示平方值的表格 | 63 | 8.1.4 程序：检查数中重复出现的 数字 | 101 |
| 6.1.3 程序：数列求和 | 63 | 8.1.5 对数组使用 sizeof 运算符 | 101 |
| 6.2 do语句 | 64 | 8.1.6 程序：计算利息 | 102 |
| 6.3 for语句 | 65 | 8.2 多维数组 | 103 |
| 6.3.1 for语句的惯用法 | 66 | 8.2.1 多维数组初始化 | 104 |
| 6.3.2 在 for 语句中省略表达式 | 67 | 8.2.2 常量数组 | 105 |
| 6.3.3 逗号运算符 | 67 | 8.2.3 程序：发牌 | 105 |
| 6.3.4 程序：显示平方值的表格 (改进版) | 68 | 问与答 | 106 |
| 6.4 退出循环 | 69 | 练习 | 107 |
| 6.4.1 break语句 | 69 | 第 9 章 函数 | 110 |
| 6.4.2 continue语句 | 70 | 9.1 函数的定义和调用 | 110 |
| 6.4.3 goto语句 | 71 | 9.1.1 程序：计算平均值 | 110 |
| 6.4.4 程序：账目簿结算 | 71 | 9.1.2 程序：显示倒数计数 | 111 |
| 6.5 空语句 | 73 | 9.1.3 程序：显示双关语 (改进版) | 112 |
| 问与答 | 74 | 9.1.4 函数定义 | 113 |
| 练习 | 76 | 9.1.5 函数调用 | 114 |
| 第 7 章 基本类型 | 78 | 9.1.6 程序：判定素数 | 115 |
| 7.1 整型 | 78 | 9.2 函数声明 | 116 |
| 7.1.1 整型常量 | 79 | 9.3 实际参数 | 117 |
| 7.1.2 读/写整数 | 80 | 9.3.1 实际参数的转换 | 118 |
| 7.1.3 程序：数列求和 (改进版) | 81 | 9.3.2 数组型实际参数 | 119 |
| 7.2 浮点型 | 81 | 9.4 return语句 | 120 |
| 7.2.1 浮点常量 | 82 | 9.5 程序终止 | 121 |
| 7.2.2 读/写浮点数 | 83 | 9.6 递归函数 | 122 |
| 7.3 字符型 | 83 | 9.6.1 快速排序算法 | 123 |
| 7.3.1 转义序列 | 84 | 9.6.2 程序：快速排序 | 124 |
| 7.3.2 字符处理函数 | 85 | 问与答 | 125 |
| 7.3.3 读/写字符 | 86 | 练习 | 128 |
| 7.3.4 程序：确定消息的长度 | 87 | 第 10 章 程序结构 | 131 |
| 7.4 sizeof运算符 | 88 | 10.1 局部变量 | 131 |
| 7.5 类型转换 | 89 | 10.2 外部变量 | 132 |
| 7.5.1 常用算术转换 | 89 | 10.2.1 程序：用外部变量实现栈 | 132 |
| 7.5.2 赋值中的转换 | 90 | 10.2.2 外部变量的利与弊 | 133 |
| 7.5.3 强制类型转换 | 91 | 10.2.3 程序：猜数 | 134 |
| 7.6 类型定义 | 92 | 10.3 程序块 | 137 |
| 问与答 | 93 | 10.4 作用域 | 137 |
| | | 10.5 构建C程序 | 138 |

| | | | |
|-------------------------------------|------------|------------------------------------|------------|
| 问与答 | 144 | 13.3.1 用printf函数和puts函数写字符串 | 173 |
| 练习 | 144 | 13.3.2 用scanf函数 和 gets函数读字符串 | 173 |
| 第 11 章 指针 | 146 | 13.3.3 逐个字符读字符串 | 174 |
| 11.1 指针变量 | 146 | 13.4 访问字符串中的字符 | 175 |
| 11.2 取地址运算符和间接寻址运算符 | 147 | 13.5 使用C语言的字符串库 | 176 |
| 11.2.1 取地址运算符 | 147 | 13.5.1 strcpy函数 | 177 |
| 11.2.2 间接寻址运算符 | 148 | 13.5.2 strcat函数 | 177 |
| 11.3 指针赋值 | 148 | 13.5.3 strcmp函数 | 178 |
| 11.4 指针作为实际参数 | 149 | 13.5.4 strlen函数 | 178 |
| 11.4.1 程序：找出数组中的最大元素 和最小元素 | 151 | 13.5.5 程序：显示一个月的提示 列表 | 179 |
| 11.4.2 用const保护实际参数 | 152 | 13.6 字符串惯用法 | 181 |
| 11.5 指针作为返回值 | 153 | 13.6.1 搜索字符串的结尾 | 181 |
| 问与答 | 153 | 13.6.2 复制字符串 | 182 |
| 练习 | 155 | 13.7 字符串数组 | 184 |
| 第 12 章 指针和数组 | 156 | 13.7.1 命令行参数 | 185 |
| 12.1 指针的算术运算 | 156 | 13.7.2 程序：核对行星的名字 | 186 |
| 12.1.1 指针加上整数 | 157 | 问与答 | 187 |
| 12.1.2 指针减去整数 | 157 | 练习 | 189 |
| 12.1.3 指针相减 | 158 | 第 14 章 预处理器 | 192 |
| 12.1.4 指针比较 | 158 | 14.1 预处理器的工作方式 | 192 |
| 12.2 指针用于数组处理 | 158 | 14.2 预处理指令 | 194 |
| 12.3 用数组名作为指针 | 160 | 14.3 宏定义 | 194 |
| 12.3.1 程序：数列反向（改进版） | 161 | 14.3.1 简单的宏 | 194 |
| 12.3.2 数组型实际参数（改进版） | 161 | 14.3.2 带参数的宏 | 196 |
| 12.3.3 用指针作为数组名 | 162 | 14.3.3 #运算符 | 197 |
| 12.4 指针和多维数组 | 163 | 14.3.4 ##运算符 | 198 |
| 12.4.1 处理多维数组的元素 | 163 | 14.3.5 宏的通用属性 | 199 |
| 12.4.2 处理多维数组的行 | 163 | 14.3.6 宏定义中的圆括号 | 199 |
| 12.4.3 用多维数组名作为指针 | 164 | 14.3.7 创建较长的宏 | 200 |
| 问与答 | 164 | 14.3.8 预定义宏 | 201 |
| 练习 | 166 | 14.4 条件编译 | 202 |
| 第 13 章 字符串 | 168 | 14.4.1 #if 指令和#endif 指令 | 202 |
| 13.1 字符串字面量 | 168 | 14.4.2 defined运算符 | 203 |
| 13.1.1 字符串字面量中的转义序列 | 168 | 14.4.3 #ifdef 指令和#ifndef 指令 | 203 |
| 13.1.2 延续字符串字面量 | 169 | 14.4.4 #elif 指令和#else 指令 | 203 |
| 13.1.3 如何存储字符串字面量 | 169 | 14.4.5 使用条件编译 | 204 |
| 13.1.4 字符串字面量的操作 | 170 | 14.5 其他指令 | 205 |
| 13.1.5 字符串字面量与字符常量 | 170 | 14.5.1 #error 指令 | 205 |
| 13.2 字符串变量 | 170 | 14.5.2 #line 指令 | 205 |
| 13.2.1 初始化字符串变量 | 171 | 14.5.3 #pragma 指令 | 206 |
| 13.2.2 字符数组与字符指针 | 172 | 问与答 | 206 |
| 13.3 字符串的读/写 | 173 | 练习 | 208 |

| | | | |
|------------------------|-----|------------------------------|-----|
| 第 15 章 编写大规模程序 | 211 | 第 17 章 指针的高级应用 | 250 |
| 15.1 源文件 | 211 | 17.1 动态存储分配 | 250 |
| 15.2 头文件 | 212 | 17.1.1 内存分配函数 | 250 |
| 15.2.1 #include 指令 | 212 | 17.1.2 空指针 | 251 |
| 15.2.2 共享宏定义和类型定义 | 213 | 17.2 动态分配字符串 | 251 |
| 15.2.3 共享函数原型 | 214 | 17.2.1 使用 malloc 函数为字符串分配内存 | 252 |
| 15.2.4 共享变量声明 | 215 | 17.2.2 在字符串函数中使用动态存储分配 | 252 |
| 15.2.5 嵌套包含 | 216 | 17.2.3 动态分配字符串的数组 | 253 |
| 15.2.6 保护头文件 | 216 | 17.2.4 程序：显示一个月的提示列表（改进版） | 253 |
| 15.2.7 头文件中的#error 指令 | 217 | 17.3 动态分配数组 | 254 |
| 15.3 把程序划分成多个文件 | 217 | 17.3.1 使用 malloc 函数为数组分配存储空间 | 255 |
| 15.4 构建多文件程序 | 223 | 17.3.2 calloc 函数 | 255 |
| 15.4.1 makefile | 223 | 17.3.3 realloc 函数 | 256 |
| 15.4.2 链接期间的错误 | 224 | 17.4 释放存储 | 256 |
| 15.4.3 重新构建程序 | 225 | 17.4.1 free 函数 | 257 |
| 15.4.4 在程序外定义宏 | 226 | 17.4.2 “悬空指针”问题 | 257 |
| 问与答 | 227 | 17.5 链表 | 257 |
| 练习 | 228 | 17.5.1 声明结点类型 | 258 |
| 第 16 章 结构、联合和枚举 | 229 | 17.5.2 创建结点 | 258 |
| 16.1 结构变量 | 229 | 17.5.3 -> 运算符 | 259 |
| 16.1.1 结构变量的声明 | 229 | 17.5.4 在链表的开始处插入结点 | 259 |
| 16.1.2 结构变量的初始化 | 230 | 17.5.5 搜索链表 | 261 |
| 16.1.3 对结构的操作 | 231 | 17.5.6 从链表中删除结点 | 262 |
| 16.2 结构类型 | 232 | 17.5.7 链表排序 | 264 |
| 16.2.1 结构标记的声明 | 232 | 17.5.8 程序：维护零件数据库（改进版） | 264 |
| 16.2.2 结构类型的定义 | 233 | 17.6 指向指针的指针 | 268 |
| 16.2.3 结构类型的实际参数和返回值 | 233 | 17.7 指向函数的指针 | 269 |
| 16.3 数组和结构的嵌套 | 234 | 17.7.1 函数指针作为实际参数 | 269 |
| 16.3.1 嵌套的结构 | 234 | 17.7.2 qsort 函数 | 270 |
| 16.3.2 结构数组 | 235 | 17.7.3 函数指针的其他用途 | 271 |
| 16.3.3 结构数组的初始化 | 235 | 17.7.4 程序：列三角函数表 | 272 |
| 16.3.4 程序：维护零件数据库 | 236 | 问与答 | 273 |
| 16.4 联合 | 241 | 练习 | 276 |
| 16.4.1 使用联合来节省空间 | 242 | 第 18 章 声明 | 278 |
| 16.4.2 使用联合来构造混合的数据结构 | 243 | 18.1 声明的语法 | 278 |
| 16.4.3 为联合添加“标记字段” | 243 | 18.2 存储类型 | 279 |
| 16.5 枚举 | 244 | 18.2.1 变量的特性 | 279 |
| 16.5.1 枚举标记和枚举类型 | 245 | 18.2.2 auto 存储类型 | 280 |
| 16.5.2 枚举作为整数 | 245 | 18.2.3 static 存储类型 | 280 |
| 16.5.3 用枚举声明“标记字段” | 246 | | |
| 问与答 | 246 | | |
| 练习 | 247 | | |

| | | | |
|----------------------------------------------------|-----|--------------------------------------|-----|
| 18.2.4 extern存储类型 | 281 | 20.3.1 定义依赖机器的类型 | 320 |
| 18.2.5 register存储类型 | 282 | 20.3.2 用联合从多个视角看待数据 | 320 |
| 18.2.6 函数的存储类型 | 282 | 20.3.3 将指针作为地址使用 | 321 |
| 18.2.7 小结 | 283 | 20.3.4 程序：设置Num Lock键 | 322 |
| 18.3 类型限定符 | 284 | 20.3.5 volatile类型限定符 | 323 |
| 18.4 声明符 | 284 | 问与答 | 323 |
| 18.4.1 解释复杂声明 | 285 | 练习 | 323 |
| 18.4.2 使用类型定义来简化声明 | 286 | 第 21 章 标准库 | 325 |
| 18.5 初始化式 | 287 | 21.1 标准库的使用 | 325 |
| 问与答 | 288 | 21.1.1 对标准库中使用的名字 一些限制 | 325 |
| 练习 | 289 | 21.1.2 使用宏隐藏函数 | 325 |
| 第 19 章 程序设计 | 291 | 21.2 标准库概述 | 326 |
| 19.1 模块 | 291 | 21.3 <stddef.h>：常用定义 | 327 |
| 19.1.1 内聚性与耦合性 | 293 | 问与答 | 328 |
| 19.1.2 模块的类型 | 293 | 练习 | 328 |
| 19.2 信息隐藏 | 293 | 第 22 章 输入/输出 | 329 |
| 19.3 抽象数据类型 | 296 | 22.1 流 | 329 |
| 19.4 C++语言 | 297 | 22.1.1 文件指针 | 330 |
| 19.4.1 C语言与C++语言之间的 差异 | 297 | 22.1.2 标准流和重定向 | 330 |
| 19.4.2 类 | 299 | 22.1.3 文本文件与二进制文件 | 330 |
| 19.4.3 类定义 | 300 | 22.2 文件操作 | 331 |
| 19.4.4 成员函数 | 300 | 22.2.1 打开文件 | 332 |
| 19.4.5 构造函数 | 302 | 22.2.2 模式 | 332 |
| 19.4.6 构造函数和动态存储分配 | 303 | 22.2.3 关闭文件 | 333 |
| 19.4.7 析构函数 | 304 | 22.2.4 为流附加文件 | 333 |
| 19.4.8 重载 | 304 | 22.2.5 从命令行获取文件名 | 334 |
| 19.4.9 面向对象编程 | 306 | 22.2.6 程序：检查文件是否可以 打开 | 334 |
| 19.4.10 派生 | 306 | 22.2.7 临时文件 | 335 |
| 19.4.11 虚函数 | 308 | 22.2.8 文件缓冲 | 336 |
| 19.4.12 模板 | 310 | 22.2.9 其他文件操作 | 337 |
| 19.4.13 异常处理 | 310 | 22.3 格式化的输入/输出 | 337 |
| 问与答 | 311 | 22.3.1 ...printf类函数 | 337 |
| 练习 | 312 | 22.3.2 ...printf类函数的转换说明 | 338 |
| 第 20 章 低级程序设计 | 314 | 22.3.3 ...printf类函数的转换说明 示例 | 339 |
| 20.1 按位运算符 | 314 | 22.3.4 ...scanf类函数 | 341 |
| 20.1.1 移位运算符 | 314 | 22.3.5 ...scanf类函数的格式化字 字符串 | 342 |
| 20.1.2 按位求反运算符、按位与 运算符、按位异或运算符 和按位或运算符 | 315 | 22.3.6 ...scanf类函数的转换说明 | 342 |
| 20.1.3 用按位运算符访问位 | 316 | 22.3.7 scanf函数的示例 | 343 |
| 20.1.4 用按位运算符访问位域 | 317 | 22.3.8 检测文件末尾和错误条件 | 344 |
| 20.1.5 程序：XOR加密 | 317 | 22.4 字符的输入/输出 | 346 |
| 20.2 结构中的位域 | 318 | 22.4.1 输出函数 | 346 |
| 20.3 其他低级技术 | 320 | | |

| | | | |
|------------------------------------|------------|-----------------------------------------|------------|
| 22.4.2 输入函数 | 346 | 练习 | 381 |
| 22.4.3 程序：复制文件 | 347 | | |
| 22.5 行的输入/输出 | 348 | 第 25 章 国际化特性 | 383 |
| 22.5.1 输出函数 | 348 | 25.1 <locale.h>：本地化 | 383 |
| 22.5.2 输入函数 | 348 | 25.1.1 类别 | 383 |
| 22.6 块的输入/输出 | 349 | 25.1.2 setlocale 函数 | 384 |
| 22.7 文件的定位 | 350 | 25.1.3 localeconv 函数 | 384 |
| 22.8 字符串的输入/输出 | 352 | 25.2 多字节字符和宽字符 | 386 |
| 问与答 | 353 | 25.2.1 多字节字符 | 387 |
| 练习 | 356 | 25.2.2 宽字符 | 387 |
| 第 23 章 库对数值和字符数据的支持 | 360 | 25.2.3 多字节字符函数 | 388 |
| 23.1 <float.h>：浮点型的特性 | 360 | 25.2.4 多字节字符串函数 | 389 |
| 23.2 <limits.h>：整数类型的大小 | 361 | 25.3 三字符序列 | 389 |
| 23.3 <math.h>：数学计算 | 362 | 问与答 | 390 |
| 23.3.1 错误 | 362 | 练习 | 391 |
| 23.3.2 三角函数 | 363 | | |
| 23.3.3 双曲函数 | 363 | | |
| 23.3.4 指数函数和对数函数 | 363 | | |
| 23.3.5 幂函数 | 364 | | |
| 23.3.6 就近取整函数、绝对值函数 和取余函数 | 364 | | |
| 23.4 <ctype.h>：字符处理 | 365 | | |
| 23.4.1 字符测试函数 | 365 | | |
| 23.4.2 程序：测试字符测试函数 | 366 | | |
| 23.4.3 字符大小写转换函数 | 367 | | |
| 23.4.4 程序：测试大小写转换函数 | 367 | | |
| 23.5 <string.h>：字符串处理 | 367 | | |
| 23.5.1 复制函数 | 368 | | |
| 23.5.2 拼接函数 | 368 | | |
| 23.5.3 比较函数 | 369 | | |
| 23.5.4 搜索函数 | 370 | | |
| 23.5.5 其他函数 | 372 | | |
| 问与答 | 372 | | |
| 练习 | 372 | | |
| 第 24 章 错误处理 | 374 | | |
| 24.1 <assert.h>：诊断 | 374 | | |
| 24.2 <errno.h>：错误 | 375 | | |
| 24.3 <signal.h>：信号处理 | 376 | | |
| 24.3.1 信号宏 | 376 | | |
| 24.3.2 signal 函数 | 377 | | |
| 24.3.3 预定义的信号处理函数 | 377 | | |
| 24.3.4 raise 函数 | 378 | | |
| 24.3.5 程序：测试信号 | 378 | | |
| 24.4 <setjmp.h>：非局部跳转 | 379 | | |
| 问与答 | 380 | | |
| | | 附录 A C 语言语法（图灵网站下载） | |
| | | 附录 B C 语言运算符（图灵网站下载） | |
| | | 附录 C 标准 C 与经典 C 的比较（图灵网站 下载） | |
| | | 附录 D 标准库函数（图灵网站下载） | |
| | | 附录 E ASCII 字符集（图灵网站下载） | |
| | | 参考文献（图灵网站下载） | |
| | | 索引（图灵网站下载） | |

计算机精品学习资料大放送

[软考官方指定教材及同步辅导书下载](#) | [软考历年真题解析与答案](#)

[软考视频](#) | [考试机构](#) | [考试时间安排](#)

[Java 一览无余：Java 视频教程](#) | [Java SE](#) | [Java EE](#)

[.Net 技术精品资料下载汇总：ASP.NET 篇](#)

[.Net 技术精品资料下载汇总：C#语言篇](#)

[.Net 技术精品资料下载汇总：VB.NET 篇](#)

[撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程](#)

[Visual C++\(VC/MFC\)学习电子书及开发工具下载](#)

[Perl/CGI 脚本语言编程学习资源下载地址大全](#)

[Python 语言编程学习资料\(电子书+视频教程\)下载汇总](#)

[最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载](#)

[数据库管理系统\(DBMS\)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇](#)

[平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载](#)

[最强 HTML/xHTML、CSS 精品学习资料下载汇总](#)

[最新 JavaScript、Ajax 典藏级学习资料下载分类汇总](#)

[网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总](#)

[UML 学习电子资下载汇总 软件设计与开发人员必备](#)

[经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通](#)

[天罗地网：精品 Linux 学习资料大收集\(电子书+视频教程\) Linux 参考资源大系](#)

[Linux 系统管理员必备参考资料下载汇总](#)

[Linux shell、内核及系统编程精品资料下载汇总](#)

[UNIX 操作系统精品学习资料<电子书+视频>分类总汇](#)

[FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频](#)

[Solaris/OpenSolaris 电子书、视频等精华资料下载索引](#)

[>> 更多精品资料请访问大家论坛计算机区...](#)

C 语言概述

如果说有人说“我想要一种语言，只需对它说我要干什么就行”，给他一支棒棒糖好了。^①

什么是C语言？它是20世纪70年代初期在贝尔实验室开发出来的一种用途广泛的编程语言。这种简单回答传递出少许C语言特有的风味。在深入学习这门语言之前，让我们先来回顾一下C语言的起源，C语言的设计目标，以及其后续的转变历程（1.1节）。我们还将讨论C语言的优缺点，并且了解一下如何最佳地使用C语言（1.2节）。

1.1 C 语言的历史

C语言的历史可以追溯到计算机领域的“古生代”时期，即20世纪60年代末期。让我们对C语言发展史作一简单回顾，从它在贝尔实验室的起源开始，再到它作为一种标准化语言的时代，最后谈到它对近代编程语言的影响。

1.1.1 起源

C语言是在贝尔实验室由Ken Thompson、Dennis Ritchie及其他同事在开发UNIX操作系统的过程中的副产品。Thompson独自动手编写了UNIX操作系统的最初版本，这套系统运行在DEC PDP-7计算机上。这款早期的小型计算机仅有16K字节内存（毕竟那是在1969年！）。

与同时代的其他操作系统一样，UNIX系统最初也是用汇编语言编写的。用汇编语言编写的程序往往难以调试和改进，UNIX系统也不例外。Thompson意识到需要用一种更加高级的编程语言来完成UNIX系统未来的开发，于是他设计了一种小型的B语言。Thompson的B语言是在BCPL语言的基础上开发的（BCPL语言是20世纪60年代中期产生的一种系统编程语言），**Q&A**而BCPL语言的起源又可以追溯到一种最早的（并且影响最深远的）Algol 60语言。

不久，Ritchie也加入到UNIX项目，并且开始着手用B语言编写程序。1970年，贝尔实验室为UNIX项目争取到一台PDP-11计算机。当B语言经过改进并且运行在了PDP-11计算机上时，Thompson就用B语言重新编写了部分UNIX代码。到了1971年，B语言已经暴露出非常不适合PDP-11计算机的问题，于是Ritchie开始开发B语言的升级版。他最初将新开发的语言命名为NB语言（意为“New B”），但是后来，新语言越来越脱离B语言，于是他决定将它改名为C语言。到1973年，C语言已经足够稳定，可以用来重新编写UNIX系统了。改用C语言编写程序显示出一个非常重要的好处：可移植性。通过在贝尔实验室里为其他类型计算机编写C语言编译器，UNIX系统也同样可以在不同类型的计算机上运行了。

1.1.2 标准化

整个20世纪70年代，特别是1977年到1979年之间，C语言一直在持续发展。这一时期出现

^① 每章章首的警句选自Alan J. Perlis的“Epigrams on Programming”（ACM SIGPLAN Notices (September, 1982): 7-13）。

了第一本有关C语言的书。1978年，Brian Kernighan和Dennis Ritchie合作编写并出版了*The C Programming Language*一书。此书一经出版就迅速成为了C程序员的宝典。由于当时缺少C语言的正式标准，所以这本书就成为了事实上的标准，编程爱好者把它称为“K&R”或者“白皮书”。

在20世纪70年代，C程序员相对较少，而且他们中的大多数人都是UNIX系统的用户。然而到了20世纪80年代，C语言已经超越了UNIX领域的界限。运行在不同操作系统下的多种类型的计算机都开始使用C语言编译器。特别是，迅速壮大的IBM PC机平台也开始使用C语言。

随着C语言的迅速普及，一系列问题也接踵而至。编写新的C语言编译器的程序员们都把“K&R”作为参考。但是遗憾的是，“K&R”对一些语言特性的描述非常模糊，以至于编译器常常会对这些特征进行不同的处理。而且，“K&R”也没有对属于C语言的特性和属于UNIX系统的内容进行明确的区分。何况在“K&R”出版以后，C语言仍在不断变化，增加了新特性并且去除了少量过时的特性。

不久，这种对C语言进行全面、准确并且最新描述的需求开始显现出来。如果没有这样一种标准，就将会出现各种“方言”，这势必会威胁C语言的主要优势之一——程序的可移植性。

1983年，美国国家标准协会(ANSI)开始编制C语言标准。经过多次修订，C语言标准于1988年完成，并且在1989年12月正式通过，成为ANSI标准X3.159-1989。1990年，国际标准化组织(ISO)通过此项标准，将其作为ISO/IEC 9899-1990国际标准^①。我们把这些标准中描述的C语言称为“ANSI C”、“ANSI/ISO C”或者就叫作“标准C”，本书将采用“标准C”的叫法。

虽然经常把“K&R”第1版中描述的C语言称为K&R C，但是自从1988年“K&R”的第2版出版以后，这种叫法就不合适了，因为新版反映出的变化都源自ANSI标准。所以这里将称第1版C语言为“经典C”，这种叫法在C语言里已经变得非常普及了。

本书中关于C语言的描述都是基于ANSI/ISO标准的。但是，由于许多用旧版C编写的“现实世界”的程序一直在使用，所以我们不可能完全忽视经典C。附录C列出了标准C和经典C之间的主要差异。如果你遇到旧版C编写的程序，那么这个附录将会帮你理解这些程序。

1.1.3 C++语言

虽然采纳了ANSI/ISO标准以后C语言自身不再发生变化。但是，从某种意义上来说，随着基于C的新式语言的产生，C语言的演变还在继续。新式语言包括有Concurrent C和Objective C，以及最著名的C++语言。**C++**C++语言是贝尔实验室的Bjarne Stroustrup设计的，它在许多方面对C语言进行了扩展，尤其是增加了支持面向对象编程的特性。

随着C++语言的迅速普及，在不久的将来你很可能会用C++语言编写程序。果真如此，为何还要如此费心学习C语言呢？首先，C++语言比C语言更加难学，在掌握复杂的C++语言（或任何其他源于C的编程语言）之前，最好是先精通C语言；其次，在我们身边存在着大量的C语言代码，这就需要会阅读和维护这些代码；最后，不是每个人都喜欢改用C++语言编程，例如，编写相对小规模程序的人就不会从C++语言中获得多少好处。

倾向于先学C++语言的主要论据是，在使用C++语言时，人们要避免采用C语言的编程习惯，以至于不得不像“从未学过”一样。读者会发现本书通过强调数据抽象、信息隐藏和其他在C++语言中扮演重要角色的原理，很好地避免了这一问题。C++语言包含了C语言的全部特性，因此

^① 该标准对应的中国国家标准是GB/T 15272—1994。C语言目前最新标准是1999年修订的ISO 9899:1999（称为C99），但总体上C99的新特性尚未得到广泛应用。——编者注

读者今后在使用C++语言时可以使用所有从本书中学到的知识。^①

虽然C++不是本书的重点，但是我们不会忽略C++。对C++语言的简要介绍会穿插在书中，并且会用**C++**符号标记出来。此外，本书还将在19.4节对C++语言进行详尽的介绍。

1.2 C 语言的优缺点

就像任何其他编程语言一样，C语言也有自己的优缺点。两者都源于这种语言预期的用途（编写操作系统和其他系统软件）和语言自身的基础理论体系：

- C语言是一种低级语言。作为一种适合系统编程的语言，C语言提供了对机器级概念（例如，字节和地址）的访问，而这些却是其他编程语言试图隐藏的内容。此外，C语言提供了与计算机内在指令紧密协调的操作，使得程序可以快速执行。既然应用程序要依赖操作系统进行输入/输出、存储管理以及其他众多的服务，操作系统一定不能运行得太慢。
- C语言是一种小型语言。与许多其他编程语言相比，C语言提供了一套更有限的特性集合。（在K&R第2版的参考手册中仅用49页就描述了整个C语言。）为了保持少量的特性，C语言在很大程度上依赖一个标准函数“库”（“函数”类似于其他编程语言中描述的“过程”或“子程序”）。
- C语言是一种包容性语言。C语言假设用户知道自己在做什么，因此它提供了比其他许多语言更广阔的自由度。此外，不同于其他语言的是，C语言不提供详细的查错功能。

1.2.1 C 语言的优点

C语言具有的众多优点说明了这种语言如此流行的原因。

- **高效性。**高效性是C语言与生俱来的优点之一。因为C语言原来就用于编写传统的由汇编语言编写的应用程序，所以快速运行并占用有限内存就显得至关重要了。
- **可移植性。**虽然程序的可移植性并不是C语言的主要目标，但是它还是成为了C语言的优点之一。当程序必须在个人计算机到超级计算机这多种机型范围内运行时，常常会用C语言来编写。C程序具有可移植性的一个原因要感谢C语言与UNIX系统的早期结合，以及后来的ANSI/ISO标准化工作。C语言正是由于标准化才没有分裂成不兼容的多种分支。另一个原因是C语言编译器规模小且容易编写，这使得此种编译器得以广泛应用。最后，C语言自身的特性也支持可移植性（尽管它没有阻止程序员编写不具有移植性的程序）。
- **功能强大。**C语言拥有一个数据类型和运算符的庞大集合，这个集合使得C语言具有强大的表达能力，往往寥寥几行代码就可以实现许多功能。
- **灵活性。**虽然C语言最初的设计是为了系统编程，但是没有固有的约束将它限制在此范围内。C语言现在可以用于编写从嵌入式系统到商业数据处理的各种应用程序。此外，C语言在其特性使用上的限制非常少。在其他语言中认定为非法的操作在C语言中往往是允许的。例如，C语言允许一个字符与一个整数值相加（或者是与一个浮点数相加）。虽然灵活性可能会让某些错误溜掉，但是它却使编程变得更加轻松。
- **标准库。**C语言的一个突出优点就是它的标准库，包含了数百个函数，这些函数可以用

3

4

^① 学C++之前是否应该先学C，是一个见仁见智的问题。近来C++界包括Stroustrup、Koenig在内的许多大师都更倾向于直接学C++。本书作者对C的处理方法非常合理，很大程度上解决了从C再过渡到C++的一般问题。

——编者注

于输入/输出、字符串处理、存储分配以及其他一些实用的操作。

- 与UNIX系统的集成。C语言在与UNIX系统结合方面特别强大。事实上，一些UNIX工具甚至假定用户是了解C语言的。

1.2.2 C语言的缺点

C语言的缺点和它的某些优点本是同根生，均来自C语言与机器的紧密性。如果可以把类似Pascal或者Ada这样的语言看成是“高级语言”的话，那么对C语言比较精确的描述应该是“低级语言”，甚至是“结构化汇编语言”。

以下是一些众所周知的问题：

- C程序可能会漏洞百出。C语言的灵活性使得它成为一种会漏洞百出的语言。许多其他语言可以发现的编程错误，C语言编译器却无法检查到。从这方面来说，C语言与汇编语言极为相似，因为直到程序运行时汇编语言才能检查到大多数错误。更糟的是，C语言还包含大量不易觉察的隐患。在后续的几章中，我们将会看到，一个额外的分号可能会导致无限循环，再或者一个遗漏的&可能会引发程序崩溃。
- C程序可能会难以理解。虽然根据大多数衡量标准C语言是一种小型语言，但是它也有许多其他通用语言没有的特性（并且常常被误解）。这些特性可以用多种方式结合使用，其中的一些结合尽管编程者心知肚明，但是其他人恐怕难以理解。另一个问题就是C程序简明扼要的特性。C语言产生的时候正是人机交互最为单调乏味的时期。由此产生的后果是C语言为了保持简明将录入和编辑程序的用时减到最少。C语言的灵活性也可能是一个负面因素，某些程序员实在太高明了，甚至可以编写出除了他们自己几乎没人可以读得懂的程序。
- C程序可能会难以修改。用C语言编写大规模程序时，如果在设计中没有考虑到维护的问题，那么将很难修改。现代的编程语言通常都会提供一种称为“模块”（“单元”或者“包”）的语言特性，这一特性可以把一个大规模的程序分解成许多可管理的块。遗憾的是，C语言恰恰缺少这样的特性。

模糊的C语言

即使是那些最热爱C语言的人也不得不承认C代码难以阅读。每年1次的国际模糊C代码大赛（International Obfuscated C Code Contest）竟然鼓励参赛者编写最难以理解的C程序。获奖作品实在让人感觉莫名其妙。例如，1991年的“最佳小程序”如下：

```
int v,i,j,k,l,s,a[99];
main()
{
    for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,j+=
(v=j<s&&(!k&&!printf(2+"\n\n%c"-(!l<!j), "#Q"[l^v?(
1^j)&1:2])&&++l||a[i]<s&&v&&v-i+j&&v+i-j&&v+i-j))&&(
1=s),v|| (i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i])
    ;
}
```

这个程序是由Doron Osovanski和Baruch Nissenbaum共同编写的，内容是要打印出八皇后问题的全部解决方案。（此问题要求在一个棋盘上放置8个皇后，并且要求皇后之间不会出现相互“攻击”的现象。）事实上，此程序可用于求解皇后数量在4~99范围内的全部问题。其他获奖程序可以在Don Libes编写的*Obfuscated C and Other Mysteries*一书中找到。

1.2.3 高效地使用 C 语言

高效地使用C语言要求在利用C语言优点的同时要避免它的缺点。以下是一些建议：

- **学习如何规避C语言的缺陷。** 规避缺陷的提示遍布全书，只要寻找到△符号。如果想看到更详尽的缺陷列表，可以参考Andrew Koenig的《C陷阱与缺陷》^①一书。现代编译器将可以检查到常见的缺陷并且发出警告，但是没有一个编译器可以侦察出全部缺陷。
- **使用软件工具使程序更加可靠。** C程序员是众多软件工具的制造者（和使用者）。lint是其中一个最著名的C语言工具。**Q&A** lint（传统上由UNIX系统提供）与大多数C语言编译器相比，可以对程序进行更加广泛的错误分析。如果可以得到lint（或某个类似程序），那么使用它应该是个好主意。另一个有益的工具则是调试工具。由于C语言的本性，许多错误无法被C语言编译器查出。这些错误会以运行时错误或不正确输出的形式表现出来。因此，在实践中C程序员都必须能够很好地使用调试工具。
- **利用现有的代码库。** 大家都在使用C是一种好处。把别人编写好的代码用于自己的程序是一个非常好的主意。C代码经常被打包进库（函数集合）。获取适当的库既可以大大减少错误，也可以节省相当多的编程工作。用于常见任务的库很容易获得，常见任务包括用户界面开发、图形学、通信、数据库管理以及网络等。有些库是公用的，而有些则是作为商品销售的。
- **采用一套切合实际的编码规范。** 编码规范是一套设计准则，即使语言本身没有强制要求，程序员也必须遵守。适当的规范可以使程序更加统一，并且易于阅读和修改。使用任何一种编程语言都需要规范，尤其是C语言。正如前面所说的，C语言本身具有较高的灵活性，这使得程序员编写的代码可能会难以理解。本书的编程示例遵循一套编码规范，但是，还有另外一些同样有效的规范可以使用。（本书将穿插讨论一些可供选择的方法。）选用哪种并不是最重要的，重要的是必须采纳某些规范并且坚持使用它们。
- **避免“投机取巧”和极度复杂的代码。** C语言鼓励使用编程技巧。通常用C语言完成某项指定任务时会有多种解决途径，程序员经常会尝试选择最简洁的方式。但是，千万不要没有节制，因为最简略的解决方式往往也是最难理解的。书中将举例说明什么是简洁合理且易于理解的方式。
- **使用标准C，少用经典C。** 标准C绝不仅仅是更好形式的经典C。标准C增加了许多允许编译器检查错误的特性，经典C却忽略了这个问题。
- **避免不可移植性。** 大多数C语言编译器都提供不属于标准C内容的特性和库函数。除非确有必要，否则最好尽量避免使用这些特性和库函数。

6

问与答

问：“问与答”究竟是什么？

答：很高兴有此一问。“问与答”将出现在每章的结尾。设置它主要有以下几个目的。

最主要的是解决学生学习C语言时经常遇到的问题。读者可以（或多或少）参与和作者的对话，这种形式非常像是读者上了一节作者讲的C语言课。

另一个目的是为对应章中涉及的某些主题提供额外的信息。本书的读者可能会有不同的知识背景。有些读者可能具有其他编程语言的经验，而另外一些读者可能是第一次学习编程。有多种语言经验的读者也许会满足于简要的说明和几个示例，而那些缺少经验的读者则需要更多内容。最基本原则是：如果发现内容不够详细，那么请查阅“问与答”部分获取更多的信息。

^① 本书由人民邮电出版社于2002年出版。——编者注

对普通读者来说，“问与答”中某些问题包含的内容可能过于超前或深奥。这类问题都会用星号 (*) 标记。提示：这类问题涉及的内容经常出现在稍后面的几章里。好奇且有一定编程经验的读者也许希望立刻深入研究这些问题，否则需要在首次阅读时跳过这部分内容。

7

必要时，“问与答”中会讨论多种C语言编译器的常见差异。例如，我们将会介绍一些频繁使用（但未标准化）的、DOS编译器和UNIX编译器都支持的特性。

问：除了C语言之外，现在Algol 60语言是否还有一些其他的衍生语言？(p.1)

答：有。由Algol 60语言衍生出来的语言有Pascal、Ada、Modula-2及其他语言。尽管这些衍生语言看上去不同于C语言，但是它们都是C语言的同系，因此它们与C语言有许多共同之处。

问：lint是做什么的？(p.6)

答：lint检查C程序中潜在的错误，它会产生一系列诊断信息。然后，程序员需要从头到尾过滤这些信息。使用lint的好处是，它可以检查出被编译器漏掉的错误。另一方面，我们需要记住使用lint，因为它太容易被忘记了。更糟的是，lint可以产生数百条信息，而这些信息中只有少部分涉及了实际错误。

问：lint这名字是如何得来的？

答：与许多其他UNIX工具不同，lint（棉绒）不是缩写。它的命名是因为它像在程序中“吹毛求疵”。

问：如何获得lint？

答：如果使用UNIX系统，那么将会自动获得lint，因为它是一个标准的UNIX工具。如果采用其他操作系统，则可能没有lint。幸运的是，lint的各种版本可以从第三方那里获得。

*问：听说gcc（GNU C编译器）可以像lint一样彻底检查程序，这是真的吗？

答：当选择-Wall选项运行时，gcc确实可以彻底检查程序。然而，gcc可能会漏掉一些lint能侦察到的问题。

*问：我很关心能让程序尽可能可靠的方法。除了lint和调试工具以外，还有其他有效的工具吗？

答：有的。其他常用的工具包括越界检查工具（bounds-checker）和内存泄漏监测工具（leak-finder）。C语言不要求检查数组下标，而越界检查工具增加了此项功能。内存泄漏监测工具帮助定位“内存泄漏”，即那些动态分配却从未被释放的内存块。

8

C 语言基本概念

某个人的常量可能是其他人的变量。

本章介绍了C语言的一些基本概念，包括预处理指令（preprocessor directive）、函数（function）、变量（variable）和语句（statement）。即使是编写最简单的C程序，也会用到这些基本概念。后续的几章将会对这些概念进行进一步的详细描述。

首先，2.1节给出一个简单的C程序，并且描述了如何对程序进行编译和链接。接着，2.2节讨论如何使程序通用。2.3节说明如何添加说明性解释，即通常所说的注释（comment）。2.4节介绍变量，变量是用来存储程序执行过程中可能会发生改变的数据的。2.5节说明利用scanf函数把数据读入变量的方法。就如2.6节介绍的那样，常量是程序执行过程中不会发生改变的数据，用户可以对其进行命名。最后，2.7节解释C语言的命名（标识符（identifier））规则，2.8节给出了C程序的书写规范。

2.1 编写一个简单的C程序

与用其他语言编写的程序相比，C程序较少要求“死板的格式”。一个完整的C程序可以只有寥寥数行。

2.1.1 程序：显示双关语

在Kernighan和Ritchie编写的经典*The C Programming Language*一书中，第一个程序是极其简短的。它仅仅输出了一条hello, world信息。与大多数C语言书籍的作者不同，这里不打算用这个程序作为第一个C程序示例，反倒更愿意支持另一种C语言的传统：双关语。下面是一条双关语：

To C, or not to C: that is the question.

9

下面这个名为pun.c的程序会在每次运行时显示上述这条消息。

```
pun.c
#include <stdio.h>

main()
{
    printf("To C, or not to C: that is the question.\n");
}
```

2.2节会对这段程序中的一些格式细节进行详尽的说明，这里仅做一个简明扼要的介绍。程序中第一行

```
#include <stdio.h>
```

是必不可少的，它“包含”了C语言标准输入/输出库的相关信息。程序的可执行代码都在main函数中，这个函数代表“主”程序。main函数中唯一的一行代码是用来显示期望的信息的。printf函数来自标准输入/输出库，可以产生完美的格式化输出。代码\n说明printf函数执行完消息显示后要进行换行操作。

2.1.2 编译和链接

尽管pun.c程序十分简短，但是为运行此程序而包含的内容可能比想象的要多。首先，需要生成一个名为pun.c并且含有上述程序代码的文件（任何一种文本编辑器都可以创建该文件）。文件的名字无关紧要，但是编译器往往要求文件的扩展名是.c。

接下来，就需要把程序转化为机器可以执行的形式。对于C程序来说，通常包含下列3个步骤：

- **预处理**。首先会把程序送交给预处理器（preprocessor）。预处理器执行以#开头的命令（通常称为指令，directive）。预处理器有点类似于编辑器，它可以给程序添加内容，也可以对程序进行修改。
- **编译**。修改后的程序现在可以进入编译器（compiler）了。编译器会把程序翻译成机器指令（即目标代码，object code）。然而，这样的程序还是不可以运行的。
- **链接**。在最后一个步骤中，链接器（linker）把由编译器产生的目标代码和任何其他附加代码整合在一起，这样才最终产生了完全可执行的程序。这些附加代码包括很多程序中用到的库函数（例如printf函数）。

幸运的是，上述过程往往是自动实现的，因此人们会发现这项工作不是太艰巨。事实上，由于预处理器通常会和编译器整合在一起，所以人们甚至可能不会注意到它在工作。

根据编译器和操作系统的不同，编译和链接所需的命令也是多种多样的。在UNIX系统环境下，通常把C语言编译器命名为cc。为了编译和链接pun.c程序，需要录入如下命令：

```
% cc pun.c
```

（字符%是UNIX系统的提示符。）在使用编译器cc时，系统自动进行链接操作，而无需单独的链接命令。

在编译和链接好程序后，编译器cc会把可执行程序放到默认名为a.out的文件中。编译器cc有许多选项。其中一个选项（-o选项）允许为含有可执行程序的文件选择名字。例如，假设要把文件pun.c生成的可执行文件命名为pun，那么只需录入下列命令：

```
% cc -o pun pun.c
```

GNU C编译器

gcc（即“GNU C compiler”的首字母缩写）是UNIX系统中最常用的编译器之一。它广泛用于多种不同的平台上。gcc来自自由软件基金会（Free Software Foundation, FSF）。这个基金会由Richard M.Stallman创建，旨在对抗UNIX正版软件的使用限制（和高额费用）。基金会的GNU项目已经重写了大量传统的UNIX软件，并且将它免费发布使用。GNU是“GUN's Not UNIX!”的缩写，可以把它念成guh-new。

如果使用gcc进行编译，那么建议在编译时最好采用-Wall选项：

```
% gcc -Wall -o pun pun.c
```

-Wall选项可以使gcc比平常更彻底地检查程序并且警告可能发生的问题。

在个人计算机上，通常至少有两种方法来编译和链接程序：既可以像在UNIX环境中那样使用命令行的方法，也可以完全在“集成开发环境”中来对程序进行编辑、编译、链接、执行甚至是调试的全部操作。

2.2 简单程序的通用形式

下面一起来仔细研究一下pun.c程序，并且由此归纳出一些通用的程序格式。简单的C程序

一般具有如下形式：

```
指令
main()
{
    语句
}
```

11

在这个模板以及本书其他类似的模板中，所有以Courier字体显示的语句都代表实际的C语言程序代码，而所有以中文楷体显示的部分则表示需要由程序员提供的内容。

注意如何使用大括号来标志出main函数的起始和结束。Q&AC语言使用{和}的方式非常类似于其他语言中begin和end的用法。这也说明对C语言一个普遍认同的特点：C语言极其依赖缩写词和特殊符号，其中一个原因是由于C程序是非常简洁的（或者不客气地说是含义模糊的）。

即使是最简单的C程序也依赖3个关键的语言特性：指令（在编译操作前修改程序的编辑命令），函数（被命名的可执行代码块，例如main函数）和语句（程序运行时执行的命令）。下面将进一步详细讨论上述这些特性。

2.2.1 指令

在编译C程序之前，预处理器会首先对C程序进行编辑。我们把预处理器执行的命令称为指令。后面会详细介绍这部分内容，这里只关注#include指令。

程序pun.c由下列这行指令开始：

```
#include <stdio.h>
```

这条指令说明，在编译前把<stdio.h>中的信息“包含”到程序中。<stdio.h>包含了关于C标准输入/输出库的信息。C语言拥有大量类似于<stdio.h>这样的头文件(header)（▶15.2节）。每个头文件都包含一些标准库的内容。这段程序中包含<stdio.h>的原因是：C语言不同于其他的编程语言，它没有内置的“读”和“写”命令。因此，进行输入/输出操作就需要用标准库中的函数来实现。

所有指令都是以字符#开始的。这个字符可以把C程序中的指令和其他代码区分开来。默认情况下，指令是一行，在每条指令的结尾既没有分号也没有其他特殊标记。

2.2.2 函数

函数类似于其他编程语言中的“过程”或“子程序”，它们是用来构建程序的构建块。事实上，一个C程序就是一个函数的集合。函数分为两大类：一类是程序员编写的函数，另一类则是由C语言的实现所提供的函数。这里更愿意把后者称为库函数(library function)，因为这些函数属于一个函数的“库”，而这个库则是由编译器提供的。

术语“函数”来源于数学。在数学中函数是一条根据一个或多个给定参数进行数值计算的规则：

$$\begin{aligned}f(x) &= x + 1 \\g(y, z) &= y^2 - z^2\end{aligned}$$

12

C语言对“函数”这个术语的使用则更加宽松。在C语言中，函数仅仅是一系列组合在一起并且赋予了名字的语句。某些函数计算一个值，而某些函数不是。计算出一个值的函数可以用return语句来指定所“返回”的值。例如，把参数进行加1操作的函数可以执行语句

```
return x + 1;
```

而当函数要计算参数的平方差时，则可以执行语句

```
return y * y - z * z;
```

虽然C程序可以包含多个函数，但是强制规定每个程序必须有一个main函数。main函数是非常特殊的：在执行程序时系统会自动调用main函数。到本书第9章，我们会学习如何编写其他函数。而在此之前，程序中都只有一个main函数。



主函数的名字main是至关重要的，绝对不能改写成begin或者start，甚至是MAIN。

如果main是一个函数，那么它也会返回一个值吗？是的。主函数在程序终止时会向操作系统返回一个状态码。在下一章中还将详细论述main函数的返回值（>9.5节）。Q&A但是现在这里将始终让main函数的返回值为0。这个值表明程序正常终止。

为了简便，程序pun.c的最初版本中忽略了return语句。下面是添加了return语句的程序：

```
#include <stdio.h>

main()
{
    printf("To C, or not to C: that is the question.\n");
    return 0;
}
```

建议你像这样在main函数的末尾用一条return语句结束。Q&A如果不这样做，某些编译器可能会产生一条警告信息。

2.2.3 语句

语句是程序运行时执行的命令。本书后面的几章（主要集中在第5章和第6章）将进一步探讨语句。程序pun.c只用到了两种语句。一种是返回（return）语句，另一种则是函数调用（function call）语句。要求某个函数执行指派的任务称为调用这个函数。例如，程序pun.c为了在屏幕上显示一条字符串就调用了printf函数：

13

```
printf("To C, or not to C: that is the question.\n");
```

C语言规定每条语句都要以分号结尾。（就像任何好的规则一样，这条规则也有一个例外：稍后会遇到的复合语句（>5.2.1节）就不以分号结尾。）由于语句可以连续占用多行，所以很难确定它的结束位置，因此用分号来向编译器显示语句的结束位置。但是，指令都是一行，因此不需要用分号结尾。

2.2.4 显示字符串

第3章将会进一步介绍printf是一个功能多么强大的函数。到目前为止，我们只是用printf函数显示了一条字符串字面量（string literal）。字符串字面量是用一对双引号包围的一系列字符。当用printf函数显示字符串字面量时，系统不会显示出双引号。

当打印结束时，printf函数不会自动跳转到下一输出行。为了让printf跳转到下一行，必须在要打印的字符串中包含一个\n（换行符）。写换行符就意味着终止当前行，然后把后续的输出转到下一行进行。为了说明这一点，请思考把语句

```
printf("To C, or not to C: that is the question.\n");
```

替换成下面两个printf函数调用后所产生的效果：

```
printf("To C, or not to C: ");
printf("that is the question.\n");
```

第一条printf函数调用语句打印出To C, or not to C:；而第二条调用语句则打印出that is the question.，并且跳转到下一行。最终的效果和前一个版本的printf语句完全一样，用户不会发现什么差异。

换行符可以在一个字符串字面量中出现多次。为了显示下列信息：

```
Brevity is the soul of wit.  
--Shakespeare
```

可以把它写成如下格式：

```
printf("Brevity is the soul of wit.\n --Shakespeare\n");
```

2.3 注释

程序pun.c始终缺乏某些重要内容：文档说明。每一个程序都应该包含识别信息，即程序名、编写日期、作者、程序的用途以及其他内容。C语言把这类信息放在注释中。符号/*标记注释的开始，而符号*/则标记注释的结束。例如：

```
/* This is a comment */
```

注释几乎可以出现在程序的任何位置上。它既可以单独占行也可以和其他程序文本出现在同一行中。下面展示的程序pun.c就把注释加在了程序开始的地方：

```
/* Name: pun. c */  
/* Purpose : Prints a bad pun. */  
/* Author: K. N. King */  
/* Date written: 5/21/95 */  
  
# include < stdio. h>  
  
main()  
{  
    printf("To C, or not to C: that is the question.\n");  
    return 0 ;  
}
```

注释还可以占用多行。一旦遇到符号/*，那么编译器读入（并且忽略）随后的内容直到遇到符号*/才停止。如果愿意，还可以把一串短注释合并成为一条长注释：

```
/* Name: pun. c  
Purpose: Prints a bad pun.  
Author : K. N. King  
Date written: 5/21/95 */
```

但是，上述这样的注释可能难于阅读，因为人们阅读程序时可能不易发现注释的结束位置。所以，单独把*/符号放在一行会很有帮助：

```
/* Name: pun. c  
Purpose: Prints a bad pun.  
Author: K. N. King  
Date written: 5/21/95  
*/
```

更好的方法是用一个“盒形”格式把注释单独标记出来：

```
*****  
*Name: pun.c  
* Purpose: Prints a bad pun.  
* Author: K. N. King  
* Date written: 5/21/95  
*****
```

有些程序员通过忽略3条边框的方法来简化盒形注释：

```
/*  
* Name: pun.c  
* Purpose: Prints a bad pun.
```

15

```
* Author: K. N. King
* Date written: 5/21/95
*/
```

一条简短的注释还可以与程序中的其他代码放在同一行：

```
main() /* Beginning of main program */
```

这类注释有时也称作“翼型注释”。



如果忘记终止注释可能会导致编译器忽略掉一部分程序内容。请思考一下下面的示例：

```
printf("My ");
printf("car ");
printf("has ");
printf("fleas");
```

由于在第一条注释中遗漏了结束标志，使得编译器忽略掉了中间的两条语句，因此程序最终只打印出了My fleas。

2.4 变量和赋值

很少有程序会像2.1节中的示例那样简单。大多数程序在产生输出之前往往需要执行一系列的计算，因此需要在程序执行过程中有一种临时存储数据的方法。和大多数编程语言一样，我们把C语言中的这类存储单元称为变量（variable）。

2.4.1 类型

每一个变量都必须有一个类型（type）。类型用来说明变量所存储的数据的种类。C语言拥有广泛多样的类型。但是现在，我们将只限定在两种类型范围内：int类型和float类型。由于类型会影响变量的存储方式以及允许对变量采取的操作，所以选择合适的类型是非常关键的。数值型变量的类型决定了变量所能存储的最大值和最小值，同时也决定是否允许在小数点后出现数字。

int（即integer的简写）型变量可以存储整数，例如0、1、392或者-2553。但是，整数的取值范围（>7.1节）是受限制的。在某些计算机上，int型数值的最大取值仅仅是32 767。

与int型变量相比，float（即floating-point的简写）型变量可以存储更大的数值。而且，float型变量可以存储带小数位的数据，例如379.125。但是，float型变量有一些缺陷，即这类变量需要的存储空间要大于int型变量。而且，进行算术运算时float型变量通常比int型变量慢。另外，float型变量所存储的数值往往只是实际数值的一个近似值。如果在一个float型变量中存储数据9 999 999 999，那么可能后来会发现变量的数值为10 000 000 000，这是舍入造成的误差。

16

2.4.2 声明

在使用变量之前必须对其进行声明，这也是为了便于编译器工作。为了声明变量，首先要指定变量的类型，然后说明变量的名字。（程序员决定变量的名字，命名规则可见2.7节。）例如，声明变量height和变量profit的方式如下所示：

```
int height;
float profit;
```

第一条声明说明height是一个int型变量，这也就意味着变量height可以存储一个整数值。第二条声明则表示profit是一个float型变量。

如果几个变量具有相同的类型，就可以把它们的声明合并：

```
int height, length, width, volume;
float profit, loss;
```

注意每一条完整的声明语句都要以分号结尾。

在main函数的第一个模版中并没有包含声明。当main函数包含声明时，必须把声明放置在语句之前：

```
main ( )
{
    声明
    语句
}
```

就书写格式而言，建议在声明和语句之间留出空白行。

2.4.3 赋值

变量通过赋值（assignment）的方式获得值。例如，语句

```
height = 8;
length = 12;
width = 10;
```

把数值8、12和10分别赋值给变量height、length和width。

一旦变量被赋值，就可以用它来辅助计算出其他变量的值：

```
volume = height * length * width;
```

在C语言中，符号*表示乘法运算，因此上述语句对存储在height、length和width这3个变量中的数值进行了乘法操作，然后把运算结果赋值给变量volume。通常情况下，赋值运算的右侧可以是一个含有常量、变量和运算符的公式（在C语言的术语中称为表达式，expression）。 17

2.4.4 显示变量的值

用printf可以显示出当前变量的值。例如，

```
Height: n
```

这里的n表示变量height的当前值。这里可以通过调用printf来实现输出上述信息的要求：

```
printf("Height: %d\n", height);
```

占位符%d用来指明在打印过程中变量height的值的显示位置。注意，由于在%d后面放置了\n，所以打印完height的值后程序会跳到下一行。

%d仅用于int型变量。如果要打印float型变量，需要用%f来代替。默认情况下，%f会显示出小数点后6位数字。若需要%f显示小数点后n位数字，则可以把.n放置在%和f之间。例如，为了打印信息

```
Profit: $2150.48
```

可以把printf写为如下形式：

```
printf("Profit: $%.2f\n", profit);
```

C语言没有限制调用一个printf可以显示的变量的数量。为了同时显示变量height和变量length的数值，可以使用下列printf调用语句：

```
printf("Height: %d Length: %d\n", height, length);
```

2.4.5 程序：计算箱子的空间重量

运输公司特别不喜欢箱子又大又轻，因为箱子在卡车或飞机上运输时要占据宝贵的空间。

事实上，对于这类箱子，公司常常要求按照箱子的体积而不是重量来支付额外的费用。通常的做法是把体积除以166（这是每磅允许的立方英寸数）。如果除得的商（也就是箱子的“空间上”或“体积上”的重量）大于箱子的实际重量，那么运费就按照空间重量来计算。

假设运输公司雇用你来编写一个可以计算箱子空间重量的程序。因为刚刚开始学习C语言，所以你决定先编写一个计算特定箱子空间重量的程序来试试身手。其中箱子的长、宽、高分别是12英寸、10英寸和8英寸。C语言中除法运算用符号/表示。所以，很显然计算箱子空间重量的公式如下：

```
weight = volume / 166;
```

18 这里的weight和volume都是整型变量，二者分别用来表示箱子的重量和体积。遗憾的是上面这个公式不是我们所需要的。在C语言中，如果两个整数相除，那么结果会被“取整”：也就是说所有小数点后的数字都会丢失。如果 $12 \times 10 \times 8$ 的箱子体积是960立方英寸，那么体积除以166后的结果将是5而不是5.783。这样会使得重量向下取整，而运输公司则希望结果向上取整。一种解决方案是在除以166之前把体积数加上165：

```
weight = (volume + 165) / 166;
```

这样，体积为166的箱子重量就为331/166，即取整为1，而体积为167的箱子重量则为332/166，即取整为2。下面列出的是利用这种方法编写的计算空间重量的程序：

```
dweight.c
/* Computes the dimensional weight of a 12" x 10" x 8" box */

#include <stdio.h>

main()
{
    int height, length, width, volume, weight;

    height = 8;
    length = 12;
    width = 10;
    volume = height * length * width;
    weight = (volume + 165) / 166;

    printf("Dimensions: %dx%dx%d\n", length, width, height);
    printf("Volume (cubic inches): %d\n", volume);
    printf("Dimensional weight (pounds): %d\n", weight);

    return 0;
}
```

这段程序的输出结果是：

```
Dimensions: 12x10x8
Volume (cubic inches): 960
Dimensional weight (pounds): 6
```

2.4.6 初始化

当程序开始执行时，某些变量会自动设置为零，而大多数变量则不会。这样产生的结果是，人们往往不能预计到变量初始化（>18.5节）后的值是什么。它可能是2568，也可能是-30891，或是其他同样没有意义的数。

当然可以总是采用赋值的方法给变量赋一个初始值。但是，还有更简便的方法：可以在变量声明中加入初始值。例如，可以在一步操作中声明变量height并同时对其进行初始化：

```
int height = 8;
```

19 按照C语言的术语来讲，数值8是一个初始化式（initializer）。

在同一个声明中可以对任意数量的变量进行初始化：

```
int height = 8, length = 12, width = 10;
```

注意上述每个变量都有属于自己的初始化式。在接下来的例子中，只有变量width拥有初始化式10，而变量height或变量length都没有（也就是说这两个变量的值仍然未知）：

```
int height, length, width = 10;
```

2.4.7 显示表达式的值

`printf`不仅可以显示变量存储的数，还可以显示任意数字表达式的值。利用这一特性既可以简化程序，又可以减少变量的数量。例如，语句

```
volume = height * length * width;
printf("%d\n", volume);
```

可以用以下形式代替：

```
printf("%d\n", height * length * width);
```

`printf`能显示表达式的能力说明了C语言的一个通用原则：在任何需要数值的地方，都可以使用具有相同类型的表达式。

2.5 读入输入

程序`dweight.c`并不十分有用，因为它仅仅可以计算出一个箱子的空间重量。为了改进程序，需要允许用户自行录入尺寸。

为了获取输入，就要用到`scanf`函数。它是C函数库中与`printf`相对应的函数。`scanf`中的字母f和`printf`中的字母f含义相同，都是表示“格式化”的意思。`scanf`函数和`printf`函数都需要使用格式串（format string）来说明输入或输出数据的样式。`scanf`函数需要知道将获得的输入数据的格式，而`printf`函数需要知道输出数据的显示格式。

为了读入一个int型数值，可以使用如下的`scanf`函数调用：

```
scanf("%d", &i); /* reads an integer; stores into i */
```

其中字符串"`%d`"说明`scanf`读入的是一个整数，而`i`是一个`int`型变量，用来存储`scanf`读入的输入。`&`运算符（>11.2.1节）在这里很难解释清楚，因此现在只说明它在使用`scanf`函数时通常是（但不总是）必须的。

读入一个`float`型数值时，需要一个形式略有不同的`scanf`调用语句：

```
scanf("%f", &x); /* reads a float value; stores into x */
```

`%f`只用于`float`型变量，因此这里假设`x`是一个`float`型变量。字符串"`%f`"告诉`scanf`函数去寻找一个`float`格式的输入值（此数可以含有小数点，但不是必须含有）。

程序：计算箱子的空间重量（改进版）

下面是计算空间重量程序的一个改进版。在这个改进的程序中，用户可以录入尺寸。注意，每一个`scanf`函数调用都紧跟在一个`printf`函数调用的后面。这样做可以提示用户何时输入，以及输入什么。

```
dweight2.c
/* Computes the dimensional weight of a box */
/* from input provided by the user */

#include <stdio.h>
```

```

main()
{
    int height, length, width, volume, weight;

    printf("Enter height of box: ");
    scanf("%d", &height);
    printf("Enter length of box: ");
    scanf("%d", &length);
    printf("Enter width of box: ");
    scanf("%d", &width);
    volume = height * length * width;
    weight = (volume + 165) / 166;

    printf("Volume (cubic inches): %d\n", volume);
    printf("Dimensional weight (pounds): %d\n", weight);

    return 0;
}

```

此程序的输出显示如下（用户的输入用下划线标注）：

```

Enter height of box: 8
Enter length of box: 12
Enter width of box: 10
Volume (cubic inches): 960
Dimensional weight (pounds): 6

```

2.6 定义常量

21 常量（constant）是在程序执行过程中固定不变的量。当程序含有常量时，建议给这些常量命名。程序dweight.c和程序dweight2.c都用到了常量166。在后期阅读程序时也许有些人会不明白这个常量的含义。所以可以采用称为宏定义（macro definition）的特性给常量命名：

```
#define CUBIC_IN_PER_LB 166
```

这里的#define是预处理指令，就类似于前面所讲的#include，因而在行的结尾也没有分号。

当对程序进行编译时，预处理器会把每一个宏用其表示的值替换回来。例如，语句

```
weight = (volume + CUBIC_IN_PER_LB - 1) / CUBIC_IN_PER_LB;
```

将变为

```
weight = (volume + 166 - 1) / 166;
```

给出的效果就如同在第一个地方编写了后一个语句。

此外，还可以利用宏来定义表达式：

```
#define SCALE_FACTOR (5.0 / 9.0)
```

当宏包含运算符时，必须用括号（►14.3.6节）把表达式括起来。

注意，常量的名字只用了大写字母。这是大多数C程序员遵循的规范，但并不是C语言本身的要求。（至今，C程序员沿用此规范已经几十年了；希望读者不要打破此规范。）

程序：华氏温度转换为摄氏温度

下面的程序提示用户输入一个华氏温度，然后输出一个对应的摄氏温度。此程序的输出格式如下所示（按照惯例，用户的输入信息用下划线标注出来）：

```
Enter Fahrenheit temperature: 212
Celsius equivalent: 100.0
```

此程序可以允许温度值不是整数，这也就是摄氏温度显示是100.0而不是100的原因。首先

来阅读一下整个程序，随后再讨论程序是如何构成的。

```
celsius.c
/* Converts a Fahrenheit temperature to Celsius */

#include <stdio.h>

#define FREEZING_PT 32.0
#define SCALE_FACTOR (5.0 / 9.0)

main()
{
    float fahrenheit, celsius;
    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);

    celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
    printf("Celsius equivalent: %.1f\n", celsius);

    return 0;
}
```

22

语句

```
celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
```

把华氏温度转换成为了相应的摄氏温度。既然FREEZING_PT表示的是常量32.0，而SCALE_FACTOR表示的是表达式(5.0 / 9.0)，所以编译器会把此语句看成是

```
celsius = (fahrenheit - 32.0) * (5.0 / 9.0);
```

在定义SCALE_FACTOR时，表达式采用(5.0 / 9.0)形式而不是(5 / 9)形式，这一点是非常重要的。因为如果两个整数相除，那么C语言会对结果采用取整操作。表达式(5 / 9)的值将为0，这并不是我们想要的。

调用printf函数输出相应的摄氏温度：

```
printf("Celsius equivalent : %.1f\n", celsius);
```

注意，使用%.1f显示celsius的值时，数值只保留小数点后一位数。

2.7 标识符

在编写程序时，需要对变量、函数、宏和其他实体进行命名。这些名字称为标识符(identifier)。在C语言中，标识符可以含有字母、数字和下划线，但是都必须以字母或者下划线开头。下面是一些合法的标识符示例：

```
times10 get_next_char _done
```

接下来这些则是不合法的标识符：

```
10times get-next-char
```

不合法的原因是：符号10times是以数字而不是以字母或下划线开头的。符号get-next-char包含了减号，而不是下划线。

C语言是区分大小写的；也就是说，在标识符中C语言区别大写字母和小写字母。例如，下列所示的标识符全是不同的：

```
job job jOb jOB Job JoB JOB JOB
```

23

上述8个标识符可以全部同时使用，且每一个都有完全不同的意义。（看起来使人困惑！）除非标识符之间存在某种关联，否则明智的程序员会尽量使标识符看起来各不相同。

既然C语言是区分大小写的，许多程序员都会遵循在标识符命名时只使用小写字母的规则（宏命名除外），而且为了使名字清晰，还会在中间插入下划线：

```
symbol_table current_page name_and_address
```

而另外一些程序员则避免使用下划线，他们的方法是把标识符中的每个单词用大写字母开头：

```
SymbolTable CurrentPage NameAndAddress
```

当然还存在其他一些合理的规范。但一定要保证整个程序中对同一个标识符按照同一种方式使用大写字母。

Q&A 标准C对标识符的最大长度没有限制，所以不用担心使用过长的描述性名字。诸如current_page这样的名字比命名为cp更容易让人理解。

关键字

在标准C中，表2-1中的所有关键字（keyword）对编译器而言都有着特殊的意义，因此这些关键字不能作为标识符来使用。

表2-1 关键字

| | | | |
|----------|--------|----------|----------|
| auto | double | int | struct |
| break | else | long | switch |
| case | enum | register | typedef |
| char | extern | return | union |
| const | float | short | unsigned |
| continue | for | signed | void |
| default | goto | sizeof | volatile |
| do | if | static | while |

因为C语言是区分大小写的，所以程序中出现的关键字必须严格按照表2-1所示的全部采用小写字母。标准库中函数的名字（例如printf函数）也只能包含小写字母。某些可怜的程序员用大写字母录入了整个程序，结果却发现编译器不能识别关键字和库函数的调用。应该注意避免这类情况发生。



请注意有关标识符的其他限制。某些编译器把标识符（如asm、far和near）视为附加关键字。属于标准库的标识符也是受限的（>21.1.1节）。意外地使用了其中的某个名字可能会导致在编译或链接时发生错误。以下划线开头的标识符也是受限的。

24

2.8 C语言程序的布局

我们可以把C程序看成是一串记号（token）（>附录A）：在不改变意思的基础上无法再进行分割的字符组。标识符和关键字都是记号。像+和-这样的运算符、逗号和分号这样的标点符号以及字符串字面量，也都是记号。例如，语句

```
printf( "Height: %d\n", height);
```

是由7个记号组成的：

```
printf ( "Height: %d\n" , height ) ;  
①   ②       ③     ④   ⑤   ⑥   ⑦
```

其中记号①和记号⑤都是标识符，记号③是字符串字面量，而记号②、记号④、记号⑥和记号⑦则是标点符号。

大多数情况下，程序中记号之间的空格数量没有严格要求。除非两个记号合并后会产生第

三个记号，否则在一般情况下记号之间根本不需要留有间隔。例如，可以删除掉2.6节的程序celsius.c中的大多数间隔，而只保留诸如float和fahrenheit这样的记号间的空格。

```
/* Converts a Fahrenheit temperature to Celsius */
#include <stdio.h>
#define FREEZING_PT 32.0
#define SCALE_FACTOR (5.0/9.0)
main(){float fahrenheit,celsius;printf(
"Enter Fahrenheit temperature: ");scanf("%f", &fahrenheit);
celsius=(fahrenheit-FREEZING_PT)*SCALE_FACTOR;
printf("Celsius equivalent: %.1f\n", celsius);return 0;}
```

事实上，如果篇幅足够，可以将整个main函数都写在一行中。但是，不能把整个程序写在一行内，因为每条预处理指令都要求独立成行。

当然，用这种方式压缩程序并不是个好主意。事实上，添加足够的空格和空行可以使程序更便于阅读和理解。幸运的是，C语言允许在记号之间插入任意数量的间隔，这些间隔可以是空格符、制表符和换行符。针对程序布局的规则有以下几条重要的原则。

- 语句可以划分在任意多行内。例如，下面的语句就是过长了以至于很难将它压缩在一行内：

```
printf("Dimensional weight (pounds): %d\n",
      (volume + CUBIC_IN_PER_LB - 1) / CUBIC_IN_PER_LB);
```

25

- 记号间的空格应便于肉眼区别记号。基于这个原因，通常会把每个运算符的前后都放上一个空格：

```
volume = height * length * width;
```

此外，还可以在每个逗号后边放一个空格。某些程序员甚至在圆括号和其他标点符号两边都加上空格。

- 缩进有助于轻松识别程序嵌套。**Q&A**例如，为了清晰地表示出声明和语句都嵌套在main函数中，应该对它们都进行缩进。
- 空行可以把程序划分成逻辑单元，从而使读者更容易辨别程序的结构。就像没有章节的书一样，没有空行的程序也很难阅读。

2.6节的程序celsius.c说明了几个上面提到的要求。请大家一起来仔细阅读一下程序中的main函数。

```
main()
{
    float fahrenheit, celsius;

    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);

    celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;

    printf("Celsius equivalent: %.1f\n", celsius);

    return 0;
}
```

首先，观察一下空格是如何环绕运算符=、-和*，从而使这些运算符可以凸现出来的。其次，留心为了明确声明和语句属于main函数，如何对它们采取缩进格式。最后，注意如何利用空行将main划分为5部分：(1) 声明变量fahrenheit和celsius；(2) 获取华氏温度；(3) 计算变量celsius的值；(4) 显示摄氏温度；(5) 返回操作系统。

在讨论程序布局问题的同时，还要注意一下记号{和记号}的放置方法，记号{放在了main()

的下面，而与之匹配的记号}则放在了独立的一行中，并且与记号{排在同一列上。把记号}独立放在一行中可以便于在函数的末尾插入或删除语句，而将记号}与记号{排在一列上是为了便于发现main函数的结尾处。

最后要注意的是：虽然可以在记号之间添加额外的空格，但是绝不能在记号内添加空格，因为这样做可能会改变程序的意思或者引发错误。如果写成

```
f1 oat fahrenheit, Celsius; /* *** WRONG *** /
```

或

```
26 f1
oat fahrenheit, Celsius; /* *** WRONG *** /
```

在程序编译时会报错。虽然把空格加在字符串字面量中是允许的，但这会改变字符串的意思。然而，把换行符加进字符串中（换句话说，就是把字符串分裂成两行）却是非法的：

```
printf("To C, or not to C:
that is the question.\n"); /* *** WRONG *** /
```

把字符串从一行延续到下一行（>13.1.2节）需要一种特殊的方法才可以实现。这种方法将在稍后的章节中学到。

问与答

问：为什么C语言如此简明扼要？如果C语言用begin和end代替{和}，并且用integer代替int，如此等等，这样好像可以使程序更加便于阅读。（p.9）

答：据说，C程序如此简短是由于该语言开发时贝尔实验室的开发环境造成的。第一个C语言编译器是运行在DEC PDP-11计算机（一种早期的微型计算机）上的，而程序员用电传打字机录入程序和打印列表，其中电传打字机实际上是一种与计算机相连的打字机。由于电传打字机的速度非常慢（每秒钟只可以打出10个字符），所以在程序中尽量减少字符数量是十分有利的。

问：在某些C语言书中，main函数的结尾使用的是exit(0)，而不是return 0，二者是否一样呢？（p.10）

答：当出现在main函数中时，这两种语句是完全等价的：二者都终止程序执行，并且向操作系统返回0值。使用哪种语句完全依据个人喜好而定。

问：如果程序终止时没有执行return语句会产生什么后果呢？（p.10）

答：某个值将会返回到操作系统中，但是无法保证这个值是什么。程序终止时只要不用测试程序的状态，那么这样做应该没有问题。

问：编译器是完全移走注释还是用空格替换掉注释呢？

答：一些早期的编译器是简单地删除每条注释中的所有字符。对程序

```
a/**/b = 0;
```

编译器可能会把它编译成

```
27 ab = 0;
```

然而，依据标准C，编译器必须用一个空格符替换每条注释语句，因此上面提到的技巧并不可行。我们实际上会得到下面的语句：

```
a b = 0;
```

问：如何发现程序有没有未终止的注释？

答：如果运气好的话，程序将无法通过编译，因为这样的注释会导致程序非法。如果程序可以通过编译，那也有几种方法可以用来发现问题。通过用调试器逐行地执行程序，就会发现是否有些行被跳过了。一些开发环境使用多种色彩显示程序，注释所用的颜色会不同于周围其他代码的颜色。如果你使用的是这样的开发环境，就会很容易发现未终止的注释，因为正常代码会与意外包含在注释里的代码有不同的颜色。此外，诸如lint（>1.2.3节）这样的程序也可以提供帮助。

问：在注释中嵌套一个新的注释是否合法？

答：在标准C中是不合法的。例如，下面的代码就是不合法的代码：

```
/*
    *** WRONG ***
*/
```

第2行的符号*/会和第一行的/*相匹配，所以编译器将会把第3行的*/标记为一个错误。

C语言禁止注释嵌套有些时候也会是个问题。假设编写了一个很长的程序，而且程序包含了許多短小的注释。为了屏蔽程序的某些部分（比如在测试过程中），那么我们首先会想到用/*和*/“注释掉”相应的程序行。但是，如果这些代码行中包含有注释的话，这种方法就行不通了。后面我们将看到，有一种较好的方法可以屏蔽部分程序（►14.4.5节）。

问：曾见过C程序用//作为注释的开头而不是/*，并且没有*/作为注释的结尾，例如：

```
// This is a comment.
```

这样表示在实践中是否合法？

答：在标准C中是不合法的。用//作为注释的开始是C++的方式，有一些C语言编译器也支持。然而，另外一些编译器可能并不支持这种方式，于是程序就成为了不可移植的程序，因此应尽量避免使用//。

问：float类型这名字是由何而来的？(p.12)

答：float是floating-point的缩写形式，它是一种存储数的方法，而这些数中的小数点是“浮动的”。float类型的值通常分成两部分存储：小数(fraction)部分（或者称为尾数(mantissa)部分）和指数(exponent)部分。例如，12.0这个数以 1.5×2^3 的形式存储，其中1.5是小数部分，而3是指数部分。有些编程语言把这种类型成为real类型而不是float类型。

*问：对标识符的长度是真的没有限制吗？(p.18)

答：是，又不是。标准C声称标识符可以任意长。但是，编译器只能记住前31个字符。因此，如果两个名字的前31个字符都相同，那么编译器可能会无法区别它们。

更复杂的情况是，标准C对外链接（►18.2节）的标识符有特殊的规定，而大多数函数名都属于这类标识符。因为链接器都必须能识别这些名字，而且一些早期的链接器只能处理短名字，所以C语言标准声称只有前6个字符才是有意义的。此外，还不区分字母的大小写。所以，这样产生的后果是ABCDEFG和abcdefg可能会被作为相同的名字处理。

大多数编译器和链接器都比标准所要求的宽松，所以实际使用中这些规则都不是问题。不要担心标识符太长，还是注意不要把它们定义得太短吧。

问：缩进时应该使用多少空格？(p.19)

答：这是个难以回答的问题。如果预留的空间过少，那么会不易察觉到缩进。如果预留的太多，则可能会导致行宽超出屏幕（或篇幅）。一些程序员采用8个空格（即一个制表键）来缩进嵌套语句，当然也有普遍采用4个空格的。特别是针对有80列限制的屏幕和打印机来说，缩进8个空格可能太多了。研究表明：缩进3个空格是最合适的。但是由于纸张篇幅的需要，本书采用了两个空格的缩进方式。

练习

2.1节

1. 建立并运行由Kernighan和Ritchie编写的著名的“hello, world”程序：

```
#include <stdio.h>

main ()
{
    printf("hello, world\n");
}
```

在编译时是否有警告信息？如果有，需要如何进行修改呢？

2.2节

2. 思考下面的程序：

```
#include <stdio.h>

main()
{
    printf("Parkinson's Law:\nWork expands so as to ");
    printf("fill the time\n");
    printf("available for its completion.\n");
    return 0;
}
```

29

- (a) 请指出程序中的指令和语句。
 (b) 程序的输出是什么？

3. 编写一个程序，程序要使用printf在屏幕上显示出下面的图形：

```
*
 *
*
 * *
 *
 *
```

2.4节

4. 通过下列方法缩写程序dweight.c：(1)用初始化语句替换对变量height、length和width的赋值语句；(2)去掉变量weight，在最后的printf语句中计算 $(volume + 165) / 166$ 。
5. 编写一个计算球体体积的程序，其中球体半径为10 m，参考公式 $V = \frac{4}{3}\pi r^3$ 。注意，分数 $\frac{4}{3}$ 应写为 $4.0/3.0$ 。（如果分数写成 $4/3$ ，会产生什么结果？）
6. 编写一个程序用来声明几个int型和float型变量，不对这些变量进行初始化，然后打印出它们的值。这些数值是否有规律？（通常情况下没有。）

2.5节

7. 修改练习5中的程序，使用户可以自行录入球体的半径。
 8. 编写一个程序，要求用户输入一个美金数量，然后显示出加了5%税率的相应金额。格式如下所示：

```
Enter a dollar amount: 100.00
With tax added: 105.00
```

2.6节

9. 修改练习7，要求用名为PI的宏表示 π 的值。

2.7节

10. 判断下列哪些是不合法的C语言标识符？

- (a) 100_bottles
- (b) _100_bottles
- (c) one_hundred_bottles
- (d) bottles_by_the_hundred_

11. 判断下列哪些是C语言的关键字？

- (a) for
- (b) If
- (c) main
- (d) printf
- (e) while

2.8节

12. 下面的语句中有多少记号？

```
a=(3*q-p*p)/3;
```

30

13. 在练习12的记号之间插入足够多的空格，使得上面的语句易于阅读。

格式化的输入/输出

在探索难以实现的问题时，简化是唯一的方法。

`scanf`函数和`printf`函数是C语言使用最频繁的两个函数，它们用来支持格式化的读和写。正如本章要展示的那样，虽然这两个函数功能强大，但是却很难正确地使用。3.1节描述`printf`函数，3.2节则介绍`scanf`函数。但是这两节都没有提供完整的细节，这将留到第22章中介绍。

3.1 `printf` 函数

`printf`函数被设计用来显示格式串（format string）的内容，并且在字符串指定位置插入可能的值。调用，`printf`函数时必须提供格式串，接着是用来在打印时插入到字符串中的任意值：

```
printf(格式串, 表达式1, 表达式2, ...);
```

显示的值可以是常量、变量或者是更加复杂的表达式。单独调用一次`printf`函数时可以打印的值的个数没有限制。

格式串包含普通字符和转换说明（conversion specification），其中转换说明以字符%开头。转换说明是用来表示打印过程中填充了值的占位符。跟随在字符%后边的信息指定了把数值从内部（二进制）形式转换成打印（字符）形式的方法，这也就是“转换说明”这一术语的由来。例如，转换说明%d指定`printf`函数把int型数值从二进制形式转换成十进制数字组成的字符串，同时转换说明%f对float型数值也进行了类似的转换。

格式串中的普通字符完全如在字符串中显示的那样打印出来，而转换说明则要用打印出的数值来替换。请思考下面的例子：

```
int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892;
y = 5527.0;

printf("i = %d, j = %d, x = %f, y = %f\n", i, j, x, y);
```

这个`printf`函数调用会产生如下的输出：

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

格式串中的普通字符被简单复制给输出行，而变量i、j、x和y的值则依次了替换4个转换说明。



C语言编译器不会检测格式串中转换说明的数量是否和输出项的数量相匹配。下面这个`printf`函数调用所拥有的转换说明的数量就多于要显示的值的数量：

```
printf("%d %d\n", i); /* *** WRONG ***/
```

`printf`函数将正确显示变量*i*的值，接着显示下一个（无意义的）整数值。函数调用带有太少的转换说明也会出现类似的问题：

```
printf("%d\n", i, j); /* WRONG */
```

在这种情况下，`printf`函数会显示变量*i*的值，但是不显示变量*j*的值。

此外，C语言编译器也不检测转换说明是否适合要显示项的数据类型。如果程序员使用不正确的转换说明，程序将会只简单地产生无意义的输出。思考下面`printf`函数的调用，调用中int型变量*i*和float型变量*x*顺序放置错误：

```
printf("%f %d\n", i, x); /* WRONG */
```

因为`printf`函数必须服从于格式串，所以它将如实地显示出一个float型数值，接着是一个int型数值。可惜这两个数值都将是无意义的。

3.1.1 转换说明

转换说明给程序员提供了大量对输出格式的控制方法。另一方面，转换说明很可能是复杂而难以阅读的。事实上，在本节中想要完整详尽地介绍转换说明是不可能的，这里将只是简要介绍一些较为重要的转换说明的性能。

在第2章中已经看到，转换说明可以包含格式化信息。具体而言，用`%.mf`来显示带小数点后一位数字的float型数值。更加通用的情况下，转换说明可以有`%m.pX`格式或`%-m.pX`格式，这里的*m*和*p*都是整型常量，而*X*是字母。*m*和*p*都是可选项；如果省略*p*，那么分割*m*和*p*的小数点也要忽略掉。在转换说明`%10.2f`中，*m*是10，*p*是2，而*X*是*f*。在转换说明`%10f`中，*m*是10，而丢失了*p*（连同小数点一起）；但是在转换说明`%.2f`中，*p*是2，而丢失了*m*。

最小字段宽度（minimum field width）*m*指定了要显示的最小字符数量。如果要打印的数值比*m*个字符少，那么值在字段内是右对齐的。（换句话说，在数值前面放置额外的空格。）例如，转换说明`%4d`将以`•123`的形式显示数123。（这里用符号`•`表示空格字符）如果要显示的数值比*m*个字符多，那么字段宽度会自动扩展为需要的尺寸。因此，转换说明`%4d`将以`12345`的形式显示数12345，而不会丢失数字。在*m*前放上一个负号会发生左对齐；转换说明`%-4d`将以`123•`的形式显示123。

精度（precision）*p*的含义很难描述，因为它依赖于转换说明符（conversion specifier）*X*的选择。*X*表明在显示数值前需要对其进行哪种转换。对数来说最常用的转换说明符有：

- `d` —— 表示十进制（基数为10）形式的整数。**Q&A** *p*说明可以显示的数字的最少个数（如果需要，就在数前加上额外的零）；如果忽略掉*p*，则默认它的值为1。
- `e` —— 表示指数（科学记数法）形式的浮点数。*p*说明小数点后应该出现的数字的个数（默认值为6）。如果*p*为0，则不显示小数点。
- `f` —— 表示“定点十进制”形式的浮点数，没有指数。*p*的含义与在说明符`e`中的一样。
- `g` —— 表示指数形式或者定点十进制形式的浮点数，形式的选择根据数的大小决定。*p*说明可以显示的有效数字（没有小数点后的数字）的最大数量。与转换说明符`f`不同，`g`的转换将不显示尾随零。此外，如果要显示的数值没有小数点后的数字，那么`g`不会显示小数点。

编写程序时无法预知显示数的大小或者数值变化范围很大的情况下，说明符`g`对于数的显示是特别有用的。在用于显示大小适中的数时，说明符`g`采用定点十进制形式。但是，在显示非常大或非常小的数时，说明符`g`会转换成指数形式以便可以需要较少的字符。

除了说明符`%d`、`%e`、`%f`和`%g`以外，还有其他一些说明符（➤7.1节、➤7.2节、➤7.3节、➤13.3

节) 将在后续的章节陆续进行介绍。转换说明符的全部列表以及转换说明符其他性能的完整解释见22.3节。

3.1.2 程序：用 printf 函数格式化数

下面的程序举例说明了用printf函数以各种格式显示整数和浮点数的方法。

tprintf.c

```
/* Prints int and float values in various formats */

#include <stdio.h>

main()
{
    int i;
    float x;

    i = 40;
    x = 839.21;

    printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
    printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

    return 0;
}
```

当显示时，printf函数格式串中的字符|只是用来帮助显示每个数所占用的空格数量；不同于%或\，字符|对printf函数而言没有任何特殊意义。此程序的输出如下：

| | | | | |
|----|---------|----|-----------|--------|
| 40 | 40 | 40 | 040 | |
| | 839.210 | | 8.392e+02 | 839.21 |

下面仔细观察上述程序中使用的转换说明：

- %d —— 显示十进制形式的变量i，且使用了空间的最小字段宽度。
- %5d —— 显示十进制形式的变量i，且使用了5个字符的最小字段宽度。因为变量i只占两个字符，所以其他3位添加空格。
- %-5d —— 显示十进制形式的变量i，且使用了5个字符的最小字段宽度；由于变量i的值不到5个字符，所以在后续位置上添加空格（更确切地说，变量i在长度为5的字段内是左对齐的）。
- %5.3d —— 显示十进制形式的变量i，且使用了总数为5个字符的最小字段宽度，而且最少要有3位数字。因为变量i只有两个字符长度，所以添加额外的零来保证另外3位数字。为了保证占有5个字符，因为结果数只有3个字符长度，所以添加2个空格（变量i是右对齐的）。
- %10.3f —— 显示定点十进制形式的变量x，且总共用10个字符，其中小数点后保留3位数字。因为变量x只有7个字符（即小数点前3位，小数点后3位，再加上小数点本身1位），所以在变量x前面有3个空格。
- %10.3e —— 显示指数形式的变量x，且总共用10个字符，其中小数点后保留3位数字。变量x总占有9位字符（包括指数），所以在变量x前面有1个空格。
- %-10.3g —— 既可以显示定点十进制形式的变量x，也可以显示指数形式的变量x，且总共用10个字符。在这种情况下，printf函数选择用定点十进制形式显示变量x。负号的出现强制进行左对齐，所以有4个空格跟在变量x后面。

3.1.3 转义序列

我们经常把在格式串中用的代码\n称为转义序列（escape sequence）。转义序列（>7.3.1节）

使字符串包含一些特殊字符而又不会使编译器引发问题，这些字符包括非打印的（控制）字符和对编译器有特殊含义的字符（诸如"）。稍后会提供完整的转义序列表。但是现在，请看下面这个示例：

- 警报（响铃）符：\a。
- 回退符：\b。
- 换行符：\n。
- 横向制表符：\t。

当这些转义序列出现在printf函数的格式串中时，它们表示在显示中执行的操作。在大多数机器上，输出\a会产生一声鸣响，输出\b会使光标从当前位置回退一个位置，输出\n会使光标跳到下一行的起始位置，Q&A输出\t会把光标移动到下一个制表符停止的位置。

字符串可以包含任意数量的转义序列。思考下面的printf函数示例，这个例子中的格式串包含了6个转义序列：

```
printf("Item\tUnit\tPurchase\n\tPrice\tDate\n");
```

执行上述语句打印出一条两行的标题：

| | | |
|-------|------|----------|
| Item | Unit | Purchase |
| Price | Date | |

另一个常用的转义序列是\"，它表示字符"。因为字符"标记字符串的开始和结束，所以它不能出现在没有使用上述转义序列的字符串内。下面是一个示例：

```
printf("\ "Hello!\"");
```

这条语句产生如下输出：

35 "Hello!"

附带提一下，不能在字符串中只放置单独一个字符\，编译器将认为它是一个转义序列的开始。为了显示单独一个字符\，需要在字符串中放置两个\字符：

```
printf("\\\\"); /* prints one \ character */
```

3.2 scanf 函数

就如同printf函数用特定的格式显示输出一样，scanf函数也根据特定的格式读取输入。像printf函数的格式串一样，scanf函数的格式串也可以包含普通字符和转换说明两部分。scanf函数转换说明的用法和printf函数转换说明的用法本质上是一样的。

在一些情况下，scanf函数的格式串将只会包含转换说明，如下例所示：

```
int i, j;
float x, y;
scanf ("%d%d%f%f", &i, &j, &x, &y);
```

假设用户录入了下列输入行：

```
1 -20 .3 -4.0e3
```

scanf函数将读入上述行的信息，并且把这些符号转换成它们表示的数，然后分别把1、-20、0.3和-4000.0赋值给变量i、j、x和y。scanf函数调用中像"%d%d%f%f"这样“紧密压缩”的格式串是很普遍的，而printf函数的格式串很少有这样紧贴的转换说明。

像printf函数一样，scanf函数也有一些不易觉察的陷阱。使用scanf函数时，程序员必须检查转换说明的数量是否与输入变量的数量相匹配，并且检查每个转换是否适合相对应的变量。和用printf函数一样，编译器无法检查出可能的匹配不当。另一个陷阱涉及符号&，通常把符

号&放在scanf函数调用中的每个变量的前面。符号&常常（但不总是）是需要的，而且记住使用它是程序员的责任。



如果scanf函数调用中忘记在变量前面放置符号&，将会产生不可预知且可能是毁灭性的结果。程序崩溃是常见的结果。最起码将不会把从输入读进来的值存储到变量中；取而代之的，变量将保留原有的值（如果没有给变量赋初值，那么这个原有值可能是没有意义的）。忽略符号&是极为常见的错误，一定要小心！一些编译器可以检查出这种错误，但通常不是所有的时候都可以。如果丢失符号&的变量（比如说变量i）没有被赋值，可能得到诸如“Possible use of ‘i’ before definition.”这样的警告。如果遇到这样的警告，就要检查是否丢失符号&。

36

调用scanf函数是读数据的一种有效但不理想的方法。许多专业C程序员避免用scanf函数，而是采用字符格式读取所有数据，然后再把它们转换成数值形式。在本书中，特别是前面的几章将相当多地用到scanf函数，因为它提供了一种读入数的简单方法。但是要注意，如果用户录入了非预期的输入，那么许多程序都将无法正常执行。正如稍后将会看到的那样，可以用程序测试scanf函数（>22.3节）是否成功读入了要求的数据（若不成功，还可以试图恢复）。但是，这样的测试对于本书的示例是不切实际的，因为这类测试将添加太多语句而掩盖掉示例的要点。

3.2.1 scanf 函数的工作方法

实际上scanf函数可以做的事情远远多于目前为止已经提到的这些。scanf函数本质上是一种“模式匹配”函数，也就是试图把输入的字符组与转换说明匹配成组。

像printf函数一样，scanf函数是由格式串控制的。调用时，scanf函数从左边开始处理字符串中的信息。对于格式串中的每一个转换说明，scanf函数努力从输入的数据中定位适当类型的项，并且跳过必要的空格。然后，scanf函数读入数据项，并且在遇到不可能属于此项的字符时停止。如果读入数据项成功，那么scanf函数会继续处理格式串的剩余部分。如果任何项都不能成功读入，那么scanf函数将不再查看格式串的剩余部分（或者余下的输入数据）而立即返回。

在寻找数的起始位置时，scanf函数会忽略空白（white-space）字符（空格符、横向和纵向制表符、换页符和换行符）。这样的结果是可以把数字放在单独一行或者分散在几行内输入。考虑下面的scanf函数调用：

```
scanf ("%d%d%f%f", &i, &j, &x, &y);
```

假设用户录入3行输入：

```
1
-20 .3
-4.0e3
```

scanf函数会把它们看成是一条连续的字符流：

```
..1..-20...3...-4.0e3..
```

（这里使用符号·表示空格符，用符号¤表示换行符。）因为scanf函数在寻找每个数的起始位置时会跳过空白字符，所以它可以成功读取数。在接下来的图中，字符下方的s说明跳过此项，而字符下面的r表示作为输入项的部分读入此项：

```
..1..-20...3...-4.0e3..
ssrsrrrssrrssssrrrrr
```

37

scanf函数“忽略”了最后的换行符，没有真正地读取它。这个换行符将是下一次scanf函数调

用的第一个字符。

`scanf` 函数遵循什么原则来识别整数或浮点数呢？在要求读入整数时，`scanf` 函数首先找到一个数字、正号或负号；然后，它将继续读取数字直到读到一个非数字时才停止。当要求读入浮点数时，`scanf` 函数会寻找一个正号或负号（可选的），随后是一串数字（可能含有小数点），再后是一个指数（可选的）。指数由字母e（或者字母E）、可选的符号和一个或多个数字构成。在使用`scanf` 函数时，转换说明`%e`、`%f` 和`%g` 是可以互换的；这3种转换说明在识别浮点数方面都遵循相同的原则。

Q&A 当`scanf` 函数遇到的字符不是当前项的内容时，会把此字符“放回原处”，在扫描下一个输入项时或者在下一次调用`scanf` 函数时，才会再次读入此字符。思考下面（公认有问题的）4个数的排列：

1-20.3-4.0e3¤

让我们像前面一样使用`scanf` 函数调用：

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

下面列出了`scanf` 函数处理这组新输入的方法：

- 转换说明`%d`。第一个非空的输入字符是1；因为整数可以用1开始，所以`scanf` 函数接着读取下一个字符，即-。`scanf` 函数识别出字符-不能出现在整数内，所以把1存入变量*i*中，而把字符-放回原处。
- 转换说明`%d`。随后，`scanf` 函数读取字符-、2、0和.（句点）。因为整数不能包含小数点，所以`scanf` 函数把-20存入变量*j*中，而把字符. 放回原处。
- 转换说明`%f`。接下来`scanf` 函数读取字符.、3和-。因为浮点数不能在数字后边有负号，所以`scanf` 函数把0.3存入变量*x*中，而将字符-放回原处。
- 转换说明`%f`。最后，`scanf` 函数读取字符-、4、.、0、e、3和¤（换行符）。因为浮点数不能包含换行符，所以`scanf` 函数把-4.0×10³存入变量*y*中，而把换行符放回原处。

在这个例子中，`scanf` 函数可以把格式串中每个转换说明与一个输入项进行匹配。因为没有读取换行符，所以换行符将留给下一次`scanf` 函数调用。

38

3.2.2 格式串中的普通字符

通过编写含有普通字符和转换说明的格式串能更进一步地理解模式匹配的概念。处理格式串中普通字符时，`scanf` 函数采取的动作依赖于这个字符是否为空白字符。

- 空白字符。当在格式串中遇到一个或多个连续的空白字符时，`scanf` 函数从输入中重复读空白字符直到遇到一个非空白字符（把该字符“放回原处”）为止。格式串中空白字符的数量无关紧要，格式串中的一个空白字符可以与输入中任意数量的空白字符相匹配。在格式串中，所有空白字符都是等价的。格式串中一个空格字符或者空白转义序列都可以匹配任意数量的空格、换行符或其他空白字符。
- 其他字符。当在格式串中遇到一个非空白字符时，`scanf` 函数将把它与下一个输入字符进行比较。如果两个字符相匹配，那么`scanf` 函数会放弃输入字符而继续处理格式串。如果两个字符不匹配，那么`scanf` 函数会把不匹配的字符放回输入中，然后异常退出，而不进一步处理格式串或者从输入中读取字符。

例如，假设格式串是`"%d/%d"`。如果输入是

•5/•96

在寻找整数时，`scanf` 函数会跳过第一个空格，把`%d`与5相匹配，把/与/相匹配，在寻找下一个整数时跳过一个空格，并且把`%d`与96相匹配。另一方面，如果输入是

• 5 • / • 96

scanf函数会跳过一个空格，把%d与5相匹配，然后试图把格式串中的/与输入中的空格相匹配。但是二者不匹配，所以scanf函数把空格放回原处，也就是把字符• 96留给下一次scanf函数调用来读取。为了允许第一个数后边有空格，必须用格式串"%d / %d"来代替。

3.2.3 混淆 printf 函数和 scanf 函数

虽然scanf函数调用和printf函数调用看起来很相似，但两个函数之间有着显著的差异。忽略这些差异就是拿程序的正确性来冒险。

一个常见的错误就是：在printf函数调用时在变量前面放置&：

```
printf("%d %d\n", &i, &j); // *** WRONG***
```

幸运的是，这种错误是很容易发现的：printf函数将显示一对样子奇怪的数来代替变量i和j的值。

39

因为在寻找数据项时scanf函数通常会跳过空白字符，所以除了转换说明，格式串常常不需要包含字符。另一个常见错误，假设scanf格式串应该类似于printf格式串，这种不正确的假定可能引发scanf函数行为异常。让我们看一下执行下面这个scanf函数调用时，到底发生了什么：

```
scanf("%d, %d", &i, &j);
```

scanf函数将首先寻找输入中的整数，把这个整数存入变量i中。然后，scanf函数将试图把逗号与下一个输入字符相匹配。如果下一个输入的字符是空格而不是逗号，那么scanf函数将不再读取变量j的值而终止操作。



虽然printf格式串经常以\n结尾，但是在scanf格式串末尾放置换行符通常是一个坏主意。对scanf函数来说，格式串中的换行符等价于一个空格，两者都会引发scanf函数提前进入到下一个非空白的字符。例如，如果有格式串"%d\n"，那么scanf函数将跳过空白字符，读取一个整数，然后跳到下一个非空白字符处。像这样的格式串可能会导致交互式程序一直“挂起”直到用户输入一个非空白字符为止。

3.2.4 程序：计算持有的股票的价值

股票价格通常是由美元数量表示的，而且可能会包含分数。例如， $4\frac{1}{2}$ ， $63\frac{17}{32}$ 等。如果持有100股，且每股价值是 $4\frac{1}{2}$ ，那么持有的股票价值将是450美元。如果持有股票是1000，且每股价值是 $63\frac{17}{32}$ ，那么持有的股票价值就是63 531.25美元。下面的程序用scanf函数读入股票价格和股份数量，然后显示持有的股票的价值。

```
stocks.c
/* Computes the value of stock holdings */

#include <stdio.h>

main()
{
    int price, shares;
    float num, denom, value;

    printf("Enter share price (must include a fraction): ");
    scanf("%d%f/%f", &price, &num, &denom);
```

```

printf("Enter number of shares: ");
scanf("%d", &shares);

value = (price + num / denom) * shares;

printf("Value of holdings: $%.2f\n", value);
return 0;
}

```

40

运行此程序，可能的显示如下：

```

Enter share price (must include a fraction): 63 17/32
Enter number of shares: 1000
Value of holdings: $63531.25

```

注意，用户必须输入股票价格中的分数部分。还要注意把变量num和变量denom都声明成float型而不是int型。用整数方式处理变量num和变量denom会引发问题，因为整除时运算符/会对int型数据进行截取。例如， $17/32$ 的结果将为0。

问与答

问：转换说明%*i*用于读和写整数。*%i*和*%d*之间有什么区别？(p.24)

答：在printf格式串中使用时，二者没有区别。但是，在scanf格式串中%d只能与十进制（基数为10）形式的整数相匹配，而%i则可以匹配用八进制（基数为8）、十进制或十六进制（基数为16）表示的整数。如果输入的数有前缀0（例如056），那么%i会把它作为八进制数来处理；如果输入的数有前缀0x或0X（例如0x56），那么%i把它作为十六进制数来处理。如果用户意外地将0放在数的开始处，那么用%i代替%d读取数可能有意料之外的结果。由于这是一个陷阱，所以建议坚持采用%d。

问：如果printf函数将%作为转换说明的开始，那么如何显示字符%呢？

答：如果printf函数在格式串中遇到两个连续的字符%，那么它将显示出一个字符%。例如，语句

```
printf("Net profit: %d%%\n", profit);
```

可以显示出

```
Net profit: 10%
```

问：转义序列\t将会使得printf函数跳到下一个横向制表符处停止。如何知道横向制表符到底跳多远呢？(p.26)

答：不可能知道。当输出一个横向制表符时，输出\t的效果不是由标准C定义的，而是依赖于所采用的操作系统。典型的横向制表符一次跳8个字符宽度，但C语言本身无法保证这一点。

41 问：如果要求读入一个数，而用户却录入了非数值的输入，那么scanf函数会如何处理？

答：请看下面的例子：

```
printf("Enter a number:");
scanf("%d", &i);
```

假设用户录入了一个有效数，后边跟着一些非数值的字符：

```
Enter a number : 23foo
```

这种情况下，scanf函数读取2和3，并且将23存储在变量i中，而剩下的字符(foo)则留给下一次scanf函数调用（或者某些其他的输入函数）来读取。另一方面，假设输入从开始就是无效的：

```
Enter a number : foo
```

这种情况下，变量i的值未定义，并且字符foo会留给下一次scanf函数调用。

如何处理这种糟糕的情况呢？稍后将看到检测scanf函数调用是否成功（>22.3.4节）的方法。如果调用失败，可以终止或者尝试恢复程序，可能的恢复方法包括丢掉有问题的输入和要求用户重新输入。（在第22章结尾的“问与答”会讨论有关丢弃错误输入的方法。）

问：如何理解scanf函数对字符进行“放回原处”并且稍后再次读取的操作？(p.28)

答：据我们所知，用户在键盘输入时，程序并没有读取输入，而是把用户的输入放入隐藏的缓冲区中，由scanf函数来读取。为后续读取，scanf函数把字符放回到缓冲区中是非常容易的。第22章将会讨论输入缓冲区的详细内容。

问：如果用户在两个数之间加入了标点符号（例如逗号），那么scanf函数将如何处理？

答：首先看一个简单的例子。假设试图用scanf函数读取一对整数：

```
printf("Enter two numbers: ");
scanf("%d%d", &i, &j);
```

如果用户录入

4,28

scanf函数将读取4并且把它存储在变量i中。在寻找第二个数字的起始位置时，scanf函数遇到了逗号。因为数字不能以逗号开头，所以scanf函数立刻返回，而把逗号和第二个数留给下一次scanf函数调用。

当然，如果确信这些数将始终用逗号进行分割，那么通过为格式串添加逗号的方法可以很容易地解决这个问题：

```
printf("Enter two numbers, separated by a comma: ");
scanf("%d,%d", &i, &j);
```

42

练习^①

3.1节

1. 下面的printf函数调用产生的输出分别是什么？

- (a) printf("%6d,%4d", 86, 1040);
- (b) printf("%12.5e", 30.253);
- (c) printf("%.4f", 83.162);
- (d) printf("%-6.2g", .0000009979);

2. 编写printf函数调用以下列格式来显示float型变量x：

- (a) 指数表示形式；最小为8的字段宽度内左对齐；小数点后保留1位数字。
- (b) 指数表示形式；最小为10的字段宽度内右对齐；小数点后保留6位数字。
- (c) 定点十进制表示形式；最小为8的字段宽度内左对齐；小数点后保留3位数字。
- (d) 定点十进制表示形式；最小为6的字段宽度内右对齐；小数点后无数字。

3.2节

3. 说明下列每对scanf格式串是否等价？如果不等价，请指出它们的差异。

- (a) "%d"与"%d"
- (b) "%d-%d-%d"与"%d - %d - %d"
- (c) "%f"与"%f"
- (d) "%f,%f"与"%f, %f"

4. 编写一个程序，接收用户录入的日期信息并且将其显示出来。其中，输入日期的形式为月/日/年（即mm/dd/yy），输出日期的形式为年月日（即yyymmdd）。格式如下所示：

```
Enter a date (mm/dd/yy): 2/17/96
You entered the date 960217
```

5. 编写一个程序，对用户录入的产品信息进行格式化。程序运行后需有如下会话：

①用*标注的练习题较复杂，正确答案往往不是显而易见的。仔细通读问题，如果需要，还要复习相关小节的内容，一定要注意！

```

Enter item number: 583
Enter unit price: 13.5
Enter purchase date (mm/dd/yy): 10/24/95
Item      Unit      Purchase
          Price     Date
583       $ 13.50   10/24/95

```

其中，数字项和日期项采用左对齐方式；单位价格采用右对齐方式。美元数量的最大取值为9999.99。
提示：使用制表符控制列坐标。

6. 图书用国际标准图书编号（ISBN）进行标识，如0-393-30375-6。编号中的第一个数字说明编写书籍所用的语言（例如，0表示英语，3表示德语）。接下来的一组数字表示出版社（例如，393是W.W.Norton出版社的编号），而随后的数字则是出版社设定的，用来识别图书（例如，30375是Stephen Jay Gould的*The Flamingo's Smile*一书的编号）。最后，结尾数字是“校验数字”，它用来验证前面数字的准确性。
编写一个程序来分解用户录入的ISBN信息，格式如下：

43
 Enter ISBN: 0-393-30375-6
 Language: 0
 Publisher: 393
 Book Number: 30375
 Check digit: 6

用实际的ISBN值检测编写好的程序（通常ISBN编号会放在书的背面和版权页上）。

- *7. 假设scanf函数调用的格式如下：

```
scanf("%d%f%d", &i, &x, &j);
```

如果用户录入如下信息：

10.3 5 6

调用执行后，变量i、x和j的值分别是多少？（假设变量i和变量j都是int型，而变量x是float型。）

- *8. 假设scanf函数调用的格式如下：

```
scanf("%f%d%f", &x, &i, &y);
```

如果用户录入如下信息：

12.3 45.6 789

调用执行后，变量x、i和y的值分别是多少？（假设变量x和变量y都是float型，而变量i是int型。）

表达式

人不是通过用计算器学会的计算，但却是靠此手段忘记了算术。

C语言的显著特征之一就是它更多地强调表达式（expression）而不是语句，表达式是显示如何计算值的公式。最简单的表达式是变量和常量。变量表示程序运行时计算出的值；常量表示不变的值。更加复杂的表达式把运算符用于操作数（操作数自身就是表达式）。在表达式 $a + (b * c)$ 中，运算符+用于操作数a和 $(b * c)$ ，而这两者自身都是表达式。

运算符是构建表达式的基本工具，而且C语言拥有一个异常丰富的运算符集合。首先，C提供了基本运算符，这类运算符在大多数编程语言中都有：

- 算术运算符包括加、减、乘和除。
- 关系运算符进行诸如“i比0大”这样的比较运算。
- 逻辑运算符实现诸如“i比0大并且i比10小”这样的关系运算。

但是C语言不只包括这些运算符，它还提供了许多其他种类的运算符。事实上，如此多的运算符将会在本书前20章中逐步进行介绍。虽然掌握如此众多的运算符可能是一件非常繁琐的事，但这对于成为C语言专家是特别重要的。

本章将涵盖一些C语言中最基础的运算符：算术运算符（4.1节）、赋值运算符（4.2节）和自增及自减运算符（4.3节）。4.1节除了讨论算术运算符，还解释了运算符的优先级和结合性，这两个特性对含有多个运算符的表达式而言非常重要。4.4节描述C语言表达式的求值方法；某些情况下表达式的值可能与使用哪种编译器有关，所以4.4节还将讨论如何避免这类情况发生。最后，4.5节介绍表达式语句（expression statement），即一种允许把任何表达式都当作语句来使用的特性。

45

4.1 算术运算符

算术运算符是包括C语言在内的许多编程语言中都广泛应用的一种运算符，这类运算符可以执行加法、减法、乘法和除法。表4-1显示了C语言的算术运算符。

表4-1 算术运算符

| 一元运算符 | 二元运算符 | |
|-----------|---------|--------------------|
| | 加法类 | 乘法类 |
| + 一元正号运算符 | + 加法运算符 | * 乘法运算符 |
| - 一元负号运算符 | - 减法运算符 | / 除法运算符 % 取余运算符 |

加法类运算符和乘法类运算符都属于二元运算符。二元运算符要求有两个操作数，而一元运算符只要有一个操作数：

```
i = +1; /* + used as a unary operator */
j = -i; /* - used as a unary operator */
```

一元运算符+无任何操作；实际上，经典C中不存在这种运算符。它主要是为了强调某数值常量是正的。

二元运算符或许看上去很熟悉，只有运算符%可能不熟悉。在其他编程语言中，经常把%称为mod（求模）或rem（求余）。 $i \% j$ 的数值是*i*除以*j*后的余数。例如，10%3的值是1，而12%4的值是0。

Q&A 除%运算符以外，表4-1中的二元运算符既允许操作数是整数也允许操作数是浮点数，或者允许两者的混合。当把int型操作数和float型操作数混合在一起时，运算结果是float型的。因此，9+2.5的值为11.5，而6.7/2的值为3.35。

运算符/和运算符%需要特别注意：

- 运算符/可能产生意外的结果。当两个操作数都是整数时，运算符/通过丢掉分数部分的方法截取结果。因此，1/2的结果是0而不是0.5。
- 运算符%要求整数操作数；如果两个操作数中有一个不是整数，那么程序将无法编译通过。
- 当运算符/和运算符%用于负的操作数时，其结果与具体实现有关。如果两个操作数中有一个为负数，那么除法的结果既可以向上取整也可以向下取整。（例如，-9/7的结果既可以是-1也可以是-2。）如果*i*或者*j*是负数，那么*i % j*的符号与具体实现有关。（例如，-9%7的值既可能是2也可能是-2。）

46

“由实现定义”

术语由实现定义（implementation-defined）出现频率很高，因此值得花些时间讨论一下。C标准故意漏掉了语言的未定义部分，并认为这部分内容会由“实现”来具体定义。所谓实现是指软件在特定的平台上编译、链接和执行。因此，根据实现的不同，程序的行为可能会稍有差异。运算符/和运算符%对负操作数的操作就是一个由实现定义行为的例子。

留下语言的未定义部分看起来可能有点奇怪，甚至很危险。但这正反映出C语言的基本理念。C语言的目的之一是达到高效率，这经常意味着要与硬件行为相匹配。当-9除以7时，一些机器可能产生的结果是-1，而另一些机器的结果为-2。C标准简单地反映了这一现实。

最好避免编写与实现定义的行为相关的程序。如果不可能做到，起码要仔细查阅手册。C标准要求由实现定义的行为必须在文档中说明。

4.1.1 运算符的优先级和结合性

当表达式包含多个运算符时，它的解释可能不会立刻清楚。例如，执行表达式*i+j*k*是意味着“*i*加上*j*，然后结果再乘以*k*”还是意味着“*j*乘以*k*，然后加上*i*”到底哪种正确呢？解决这个问题的一种方法就是添加圆括号，表达式既可以写为(*i+j*)**k*，也可以写为*i+(j*k)*。作为通用规则，C语言允许在所有表达式中用圆括号进行分组。

可是，如果不使用圆括号结果会如何呢？编译器将把表达式*i+j*k*解释为(*i+j*)**k*还是*i+(j*k)*呢？和许多其他语言一样，C语言采用运算符优先级（operator precedence）的规则来解决这个隐含的二义性。算术运算符有下列相对优先级：

最高优先级： + - （一元运算符）

* / %

最低优先级： + - （二元运算符）

当两个或更多个运算符出现在同一个表达式中时，可以通过以运算符优先级从高到低的次序重复给子表达式添加圆括号来确定编译器解释表达的方法。下面的例子说明了这种结果：

$$\begin{array}{lll} i + j * k & \text{等价于} & i + (j * k) \\ -i * -j & \text{等价于} & (-i) * (-j) \\ +i + j / k & \text{等价于} & (+i) + (j / k) \end{array}$$

当一个表达式包含两个或更多个相同优先级的运算符时，单独的运算符优先级规则是不够用的。这种情况下，运算符的结合性（associativity）开始发挥作用。如果运算符是从左向右结合的，那么称这种运算符是左结合的（left associative）。二元算术运算符（即*、/、%、+和-）都是左结合的，所以

$$\begin{array}{ll} i - j - k & \text{等价于} (i - j) - k \\ i * j / k & \text{等价于} (i * j) / k \end{array}$$

如果运算符是从右向左结合的，那么称这种运算符是右结合的（right associative）。一元算术运算符（+和-）都是右结合的，所以

$$- + i \quad \text{等价于} \quad -(+i)$$

在许多语言（特别是C语言）中，优先级和结合性规则都是非常重要的。然而，C语言有如此多的运算符（几乎50种！）以致很少有程序员会费心记住这些优先级和结合性的规则。相反，程序员在有疑问时会参考运算符表（见附录B），或者只是使用足够的圆括号。

4.1.2 程序：计算通用产品代码的校验位

许多年来，货物生产商都会把在超市销售的每件商品上放置一个条码。这种被称为通用产品代码（Universal Product Code, UPC）的条码可以识别生产商和产品。超市可以通过扫描商品上的条码来确定该商品的价格。每个条码表示成一个12位的数，通常会把这个数打印在条码下面。例如，包装为26盎司的Morton牌碘化盐所用条码下的数字是：

0 24600 01003 0

第1个数字表示商品的种类（0表示大部分的食品杂货，2表示需要称量的商品，3表示药品或与健康相关的商品，而5表示优惠券）。第一组5位数字用来识别生产商。第二组5位数字用来区分产品（包括包装尺寸）。最后一位数字是“校验位”，它唯一的目的是用来帮助识别先前数字中的错误。如果条码扫描出现错误，那么前11位数字可能会和最后一位数字不一致，而超市扫描机将拒绝整个条码。

下面是一种计算校验位的方法：首先把第1位、第3位、第5位、第7位、第9位和第11位数字相加；然后把第2位、第4位、第6位、第8位和第10位数字相加；接着把第一次加法结果乘以3后再加上第二次加法的结果；随后，再把上述结果减去1；减法后的结果除以10取余数。最后，用9减去上一步骤中得到的余数。

以Morton碘盐为例，我们从表达式 $0+4+0+0+0+3=7$ 得到第一个和，而从表达式 $2+6+0+1+0=9$ 得到第二个和。把第一个和乘以3后再加上第二个和得到的结果是30。再减去1得到29。把这个值除以10取余数为9。9再减去余数9，结果为0。下面还有两个通用产品代码，试着手工算出各自的校验位数字^①：

Jif牌奶油花生黄油（18盎司）： 0 37000 00407 ?
Ocean Spray牌蔓越桔果酱（8盎司）： 0 31200 01005 ?

下面编写一个程序来计算任意通用产品代码的校验位。要求用户录入通用产品代码的前11位数字，然后程序显示出相应的校验位。为了避免混淆，要求用户分3部分录入数字：左边的第一个数字，第一组5位数字，还有第二组5位数字。程序运行的形式如下所示：

```
Enter the first (single) digit: 0
Enter first group of five digits: 24600
```

^① 要计算的校验位分别为3（Jif）和6（Ocean Spray）。

```
Enter second group of five digits: 01003
Check digit: 0
```

程序不是按一个五位数来读取每位数字，而是将它们读作5个一位数字。把数看成独立的一位的数字进行读取是比较方便的，而且也无需担心由于五位数过大而无法存储到int型变量中。（某些编译器限定int型变量最大值为32 767。）为了读取单个数字，需要使用scanf函数，其中转换说明为了配合一位数字的录入采用%1d的格式。

```
upc.c
/* Computes a Universal Product Code check digit */

#include <stdio.h>

main()
{
    int d, i1, i2, i3, i4, i5, j1, j2, j3, j4, j5,
        first_sum, second_sum, total;

    printf("Enter the first (single) digit: ");
    scanf("%1d", &d);
    printf("Enter first group of five digits: ");
    scanf("%1d%1d%1d%1d%1d", &i1, &i2, &i3, &i4, &i5);
    printf("Enter second group of five digits: ");
    scanf("%1d%1d%1d%1d%1d", &j1, &j2, &j3, &j4, &j5);

    first_sum = d + i2 + i4 + j1 + j3 + j5;
    second_sum = i1 + i3 + i5 + j2 + j4;
    total = 3 * first_sum + second_sum;

    printf("Check digit: %d\n", 9 - ((total - 1) % 10));
}

49
```

4.2 赋值运算符

一旦计算出表达式的值就常常需要把这个值存储在变量中，以便后面使用。C语言的=（简单赋值（simple assignment）运算符可以用于此目的。为了更新已经存储在变量中的值，C语言还提供了一种复合赋值（compound assignment）运算符。

4.2.1 简单赋值

表达式 $v = e$ 的赋值效果是求出表达式 e 的值，并把此值复制给 v 。正如下面的示例显示的那样， e 可以是常量、变量或较为复杂的表达式：

```
i = 5;           /* i is now 5 */
j = i;           /* j is now 5 */
k = 10 * i + j; /* k is now 55 */
```

如果 v 和 e 的类型不同，那么赋值运算发生时会把 e 的值转化为 v 的类型：

```
int i;
float f;

i = 72.99 /* i is now 72 */
f = 136;   /* f is now 136.0 */
```

稍后将会回到类型转换的主题（>7.5节）。

在许多编程语言中，赋值是语句；然而，在C语言中，赋值就像+那样是运算符。换句话说，赋值操作产生结果，这就如同两个数相加产生结果一样。赋值表达式 $v = e$ 的值就是赋值运算后 v 的值。因此，表达式 $i = 72.99$ 的值是72（不是72.99）。

副 作 用

通常不希望运算符修改它们的操作数，因为数学中的运算符就是如此。编写表达式 $i+j$ 不会改变 i 或 j 的值，它只是计算出 i 加 j 的结果。

大多数C语言运算符不会改变操作数的值，但是也有一些会改变。由于这类运算符所做的不再仅仅是计算出值，所以称它们有副作用（side effect）。简单赋值运算符是已知的第一个有副作用的运算符，它改变了运算符左边的操作数，表达式 $i=0$ 产生的结果为 0，作为副作用，把 0 赋值给 i 。

50

既然赋值是运算符，那么一些赋值可以串联在一起：

```
i = j = k = 0;
```

运算符 = 是右结合的，所以上述赋值表达式等价于

```
i = (j = (k = 0));
```

效果是先把 0 赋值给 k ，再赋值给 j ，最后再赋值给 i 。



注意由于结果发生了类型转换，所以串联的赋值运算最终的结果不是预计的结果：

```
int i;
float f;
```

```
f = i = 33.3;
```

首先把数值 33 赋值给变量 i ，然后把 33.0（而不是预计的 33.3）赋值给变量 f 。

通常情况下，赋值公式 $v = e$ 中 v 可以是任何类型的值。在下面的例子中，表达式 $j=i$ 把 i 的值复制给 j ，然后 j 的新值加上 1，最后产生出 k 的新值：

```
i = 1;
k = 1 + (j = i);
printf("%d %d %d\n", i, j, k); /* prints "1 1 2" */
```

但是，通常按照上述这种形式使用赋值运算符不是一个好主意。其一，“嵌入式赋值”不便于程序的阅读；其二，4.4 节将会看到，这样做也会是隐含错误的根源。

4.2.2 左值

大多数C语言运算符允许它们的操作数是变量、常量或者是包含其他运算符的表达式。然而，赋值运算符要求它左边的操作数必须是左值（lvalue）。Q&A 左值表示存储在计算机内存中的对象，而不是常量或计算结果。变量是左值，而诸如 10 或 $2*i$ 这样的表达式则不是左值。目前为止，变量是已知的唯一左值，在后面的几章中，我们将介绍其他类型的左值。

既然赋值运算符要求左边的操作数是左值，那么在赋值表达式的左侧放置任何其他类型的表达式都是不合法的：

```
12 = i;      /*** WRONG ****/
i + j = 0;   /*** WRONG ****/
-i = j;      /*** WRONG ****/
```

编译器将会检查出这个自然错误，并且显示出诸如“Lvalue required.” 这样的错误信息。

51

4.2.3 复合赋值

利用变量原有值计算出新值并重新赋值给这个变量在C语言程序中是非常普遍的。例如，下列这条语句就是把变量 i 加上 2 后再赋值给它自己：

```
i = i + 2;
```

C语言的复合赋值运算符 (compound assignment operator) 允许缩短这种语句和其他类似的语句。使用 $+=$ 运算符，可以将上面的表达式简写为

```
i += 2; /* same as i = i + 2; */
```

$+=$ 运算符把右侧操作数的值加上左侧的变量，并把结果赋值给左侧的变量。

还有另外9种复合赋值运算符，包括

```
-= *= /= %=
```

(其他复合赋值运算符 (>20.1 节) 将在后面的章节中介绍。) 所有复合赋值运算符的工作方式大体相同：

- $v += e$ 表示 v 加上 e ，然后结果存储在 v 中。
- $v -= e$ 表示 v 减去 e ，然后结果存储在 v 中。
- $v *= e$ 表示 v 乘以 e ，然后结果存储在 v 中。
- $v /= e$ 表示 v 除以 e ，然后结果存储在 v 中。
- $v \% e$ 表示 v 除以 e 取余数，然后余数的结果存储在 v 中。

注意，这里已经不说 $v += e$ “等价于” $v = v + e$ 。一个问题是运算符的优先级：表达式 $i * j + k$ 和表达式 $i = i * j + k$ 是不一样的。**Q&A**在极少的情况下，由于 v 自身的副作用， $v += e$ 不等同于 $v = v + e$ 。类似这样的说明也适用于其他复合赋值运算符。



在使用复合赋值运算符时，注意不要交换组成运算符的两个字符的位置。交换字符位置产生的表达式也许可以被编译器接受，但不会有预期的意义。例如，原打算写表达式 $i += j$ 但却写成了 $i =+ j$ ，程序将继续编译。但是，后一个表达式 $i =+ j$ 等价于表达式 $i = (+ j)$ ，这只是把 j 的值赋值给 i 。

复合赋值运算符有着和 $=$ 运算符一样的特性。特别是，它们都是右结合的，所以语句

```
i += j += k;
```

意味着

52

```
i += (j += k);
```

4.3 自增运算符和自减运算符

变量方面最常用到的两个操作是“自增”(加1)和“自减”(减1)。当然，也可以通过下列方式完成这类操作：

```
i = i + 1;
j = j - 1;
```

复合赋值运算符可以将上述这些语句缩短一些：

```
i += 1;
j -= 1;
```

Q&A但是C语言允许用 $++$ (自增)和 $--$ (自减)运算符将这些语句缩得更短些。

乍一看，自增和自减运算符是在简化自身： $++$ 表示操作数加1，而 $--$ 表示操作数减1。但是，这种简化是一种误导，实际上自增和自减运算符的使用是很复杂的。复杂的原因之一就是 $++$ 和 $--$ 运算符既可以作为前缀(prefix)运算符(例如 $++i$ 和 $--i$)使用也可以作为后缀(postfix)运算符(例如 $i++$ 和 $i--$)使用。程序的正确性可能和选取适合的运算符形式紧密相关。

另一个复杂的原因是，和赋值运算符一样， $++$ 和 $--$ 也有副作用：它们会改变操作数的值。

计算表达式`++i`（“前缀自增”）的结果是`i+1`，并且作为副作用的效果是自增`i`:

```
i = 1;
printf("i is %d\n", ++i); /* prints "i is 2" */
printf("i is %d\n", i);   /* prints "i is 2" */
```

计算表达式`i++`（“后缀自增”）的结果是`i`，但是引发`i`随后进行自增:

```
i = 1;
printf("i is %d\n", i++); /* prints "i is 1" */
printf("i is %d\n", i);   /* prints "i is 2" */
```

第一个`printf`函数显示了`i`自增前的原始值，第二个`printf`函数显示了`i`变化后的newValue。正如这些例子说明的那样，`++i`意味着“立即自增`i`”，而`i++`则意味着“现在先用`i`的原始值，稍后再自增`i`”。这个稍后有多久呢？Q&A C语言标准没有给出精确的时间，但是可以放心地假设是在下一条语句执行前`i`将进行自增。

--运算符具有相似的特性:

```
i = 1;
printf("i is %d\n", --i); /* prints "i is 0" */
printf("i is %d\n", i);   /* prints "i is 0" */
i = 1;
printf("i is %d\n", i--); /* prints "i is 1" */
printf("i is %d\n", i);   /* prints "i is 0" */
```

53

在同一个表达式中使用了过多的`++`或`--`运算符，结果可能会很难理解。思考下列语句:

```
i = 1;
j = 2;
k = ++i + j++;
```

在上述语句执行后，`i`、`j`和`k`的值分别是多少呢？既然`i`是在值被使用前进行自增，而`j`是在值被使用后进行自增，所以最后一语句等价为:

```
i = i + 1;
k = i + j;
j = j + 1;
```

所以最终`i`、`j`和`k`的值分别是2、3和4。与此对应，执行语句

```
i = 1;
j = 2;
k = i++ + j++;
```

`i`、`j`和`k`的值将分别是2、3和3。

需要记住的是，后缀`++`和后缀`--`比一元的正号、负号优先级高，而且都是左结合的。前缀`++`和前缀`--`与一元的正号、负号优先级相同，而且都是右结合的。

4.4 表达式求值

表4-2总结了到目前为止讲到的运算符。（附录B有一个类似的显示了全部运算符的表格。）表4-2的第一列显示了每种运算符相对于表中其他运算符的优先级（最高优先级为1，最低优先级为5），最后一列显示了每种运算符的结合性。

表4-2（或者附录B中的运算符汇总表）有广泛的用途。先看其中的一种用途。假设我们读某人的程序时遇到类似这样的复杂表达式:

```
a = b += c++ - d + --e / -f
```

如果有圆括号显示表达式是如何由子表达式构成的，那么这个复杂的表达式将会很容易理解。借助表4-2，为表达式添加圆括号是非常容易的：检查表达式，找到最高优先级的运算符后，用



圆括号把运算符和相应的操作数括起来，这表明在此之后圆括号内的内容将被看成是一个单独的操作数。然后重复此类操作直到将表达式完全加上圆括号。

表4-2 部分C语言运算符表

| 优先级 | 类型名称 | 符 号 | 结合性 |
|-----|---------|------------------|-----|
| 1 | (后缀) 自增 | + + | 左结合 |
| | (后缀) 自减 | - - | |
| 2 | (前缀) 自增 | + + | 右结合 |
| | (前缀) 自减 | - - | |
| | 一元正号 | + | |
| | 一元负号 | - | |
| 3 | 乘法类 | * / % | 左结合 |
| 4 | 加法类 | + - | 左结合 |
| 5 | 赋值 | = *= /= %= += -= | 右结合 |

在此示例中，用作后缀运算符的++具有最高优先级，所以在后缀++和相关操作数的周围加上圆括号：

```
a = b += (c++) - d + --e / -f
```

现在在表达式中发现了前缀--运算符和一元负号运算符（二者的优先级都为2）：

```
a = b += (c++) - d + (--e) / (-f)
```

注意，另外一个负号的左侧紧跟着一个操作数，所以这个运算符一定是减法运算符，而不是一元负号运算符。

接下来，注意到运算符/（优先级为3）：

```
a = b += (c++) - d + ((--e) / (-f))
```

这个表达式包含两个优先级为4的运算符，减号和加号。任何时候两个具有相同优先级的运算符和一个操作数相邻时，都需要注意结合性。在此例中，-运算符和+运算符都和d毗邻，所以应用结合性规则。-运算符和+运算符都是自左向右结合，所以圆括号先环绕减号，然后再环绕加号：

```
a = b += (((c++) - d) + ((--e) / (-f)))
```

最后剩下运算符=和运算符+=。这两个运算符都和b相连，所以必须考虑结合性。赋值运算符从右向左结合。所以括号先加在表达式+=周围，然后加在表达式=周围：

```
(a = (b += (((c++) - d) + ((--e) / (-f))))))
```

现在这个表达式完全加上了括号。

子表达式的求值顺序

54
55

运算符的优先级和结合性的规则允许将任何C语言表达式划分成若干子表达式；而且如果表达式是完全括号化的，那么这些规则还可以确定添加圆括号的唯一方式。与之相矛盾的是，这些规则并不总是允许我们来确定表达式的值，这些表达式的值可能依靠子表达式的求值顺序来确定。

C语言没有定义子表达式的求值顺序（除了含有逻辑与运算符及逻辑或运算符（>5.1节）、条件运算符（>5.2节）以及逗号运算符（>6.3.3节）的子表达式）。因此，在表达式(a + b) * (c - d)中，无法确定子表达式(a + b)是否在子表达式(c - d)之前求值。

不管子表达式的计算顺序如何，大多数表达式都有相同的值。但是，当子表达式改变某个

操作数的值时，产生的值就可能不一致了。思考下面这个例子：

```
a = 5;
c = (b = a + 2) - (a = 1);
```

在执行这些语句以后，`c`既可能是6也可能是2。如果先计算子表达式(`b = a + 2`)，那么把7赋值给`b`，而把6赋值给`c`。但是，如果先计算子表达式(`a = 1`)，那么将把3赋值给`b`，而把2赋值给`c`。



如果表达式的值依赖于子表达式的计算顺序，这相当于设置了一种可能会在未来引爆的“定时炸弹”。最初编写时程序可能按照预期进行工作，但是后来用不同编译器进行编译时，程序就会出现异常。

为了避免出现此类问题，一个好主意就是：不在子表达式中使用赋值运算符，而是采用一串分离的赋值表达式。例如，上述语句可以改写成如下形式：

```
a = 5;
b = a + 2;
a = 1;
c = b - a;
```

在执行完这些语句后，`c`的值将始终是6。

除了赋值运算符，自增和自减运算符是唯一可以改变操作数的运算符。使用这些运算符时，要注意表达式不能依赖特定的计算顺序。在下面的例子中，`j`可能会有两个值：

```
i = 2;
j = i * i++;
```

很自然地就会认定`j`赋值为4。但是，`j`也可能赋值为6。这里特定的情况是：(1) 取出第二个操作数(`i`的原始值)，然后`i`自增；(2) 取出第一个操作数(`i`的新值)；(3) `i`的原始值和新值相乘结果为6。

56

4.5 表达式语句

C语言有一条不同寻常的规则，那就是任何表达式都可以用作为语句。换句话说，不论任何类型或计算结果，任何表达式都可以通过添加分号的方式转换成为语句。例如，可以把表达式`++i`转换成语句

```
++i;
```

执行这条语句时，`i`先进行自增，然后把新产生的`i`值取出(这就像在闭合表达式中的用法)。**Q&A**但是，由于`++i`不是长表达式中的一部分，所以会丢弃它的值，同时执行下一条语句。(当然对`i`的改变是一定的。)

既然会丢掉`++i`的值，那么除非表达式有副作用，否则将表达式用作语句并没有什么意义。一起来看看下面这3个例子。在第一个例子中，`i`存储了1，然后取出`i`的新值但是未使用：

```
i = 1;
```

在第二个例子中，取出`i`的值但没有使用；随后，`i`进行自减：

```
i --;
```

在第三个例子中，计算出表达式`i * j - 1`的值后丢弃掉：

```
i * j - 1;
```

因为`i`和`j`没有变化，所以这条语句没有任何作用。



键盘上的误操作很容易造成“什么也不做”的表达式语句。例如，本想输入

`i = j;`

但是却错误地输入成

`i + j;`

(因为通常把=和+两个字符设置在键盘的同一个键上，所以这种错误发生的频率可能会超出想象。) 某些编译器可能会检查出无意义的表达式语句；这样的话，会显示类似“Code has no effect.”的警告。

57

问与答

问：我注意到C语言没有指数运算符。如何对数进行幂操作呢？

答：通过重复乘法运算的方法可以进行整数的小范围正整数次幂运算 ($i * i * i$ 是 i 的立方运算)。如果想计算数的非正整数次幂运算，可以调用pow函数 (►23.3.5节)。

问：如果想把%运算符用于浮点数，而程序又无法编译通过。这该怎么办？(p.34)

答：%运算符要求整数操作数，所以可以试试fmod函数 (►23.3.6节)。

问：如果C语言有左值，那它也有右值吗？(p.37)

答：是的，当然。左值是可以出现在赋值左侧的表达式，而右值则是可以出现在赋值右侧的表达式。因此，右值可以是变量、常量或更加复杂的表达式。本书和C语言标准一样，采用“表达式”这一术语来代替“右值”。

*问：前面提到如果v有副作用，那么 $v += e$ 不等价于 $v = v + e$ 。可以解释一下吗？(p.38)

答：计算 $v += e$ 只是导致求一次 v 的值，而计算 $v = v + e$ 则会求两次 v 的值。任何副作用都能导致两次求 v 的值。在下面的例子中， i 自增一次：

`a [i++] += 2;`

如果用=代替+=，那么 i 会自增两次：

`a [i++] = a[i++] + 2;`

问：C语言为什么提供++、--运算符？它们是否比其他的自增、自减方法执行得快，或者说它们更便捷？(p.38)

答：C语言从Ken Thompson早期的B语言中继承了++和--。显然，Thompson创造这类运算符是因为他的B语言编译器可能对 $++i$ 产生比 $i=i+1$ 更简洁的翻译。这些运算符成为C语言的根深蒂固的组成部分（事实上，许多C语言最著名的习惯用语都依赖于这些运算符）。对于现代编译器而言，使用++和--不会使编译后的程序变得更短小或更快，继续普及这些运算符主要是由于它们的简洁和便利。

问：++和--是否可以处理float型变量？

答：可以。规定自增和自减运算可以用于所有数值类型。但是极少采用自增和自减运算符处理float型变量。

*问：在使用后缀形式的++或--时，何时执行自增或自减操作？(p.39)

答：这是一个非常好的问题。但是，它也是一个非常难回答的问题。C语言标准介绍了“顺序点”的概念，并且提到“在前一个顺序点和下一个顺序点之间应该对存储的操作数的值进行更新”。在C语言中有各种不同类型的顺序点，语句是其中一种。到语句末尾，必须已经执行完所有语句中的自增和自减操作，只有在满足这些条件的情况下才可以执行下一条语句。

思考下面这个例子：

```
i = 1;
j = i++ + i++;
```

在执行完第二条语句时， i 应该已经自增了两次。但是不知道这两次自增是在 i 自增之后（给 j 的值

为2), 还是在其中一个i先行自增之后(给j的值为3)。

在后几章中遇到的特定运算符(逻辑与、逻辑或、条件和逗号)也强调顺序点。函数调用也是如此: 函数调用执行之前实际参数必须全部计算出来。如果实际参数恰巧是含有++或--运算符的表达式, 那么必须在调用发生前进行自增或自减操作。

问: 当说到丢掉表达式语句的值时意味着什么? (p.41)

答: 根据定义一个表达式表示一个值。例如, 如果i的值为5, 那么计算i+1产生的值为6。通过在末尾添加分号的方法把i+1变成语句:

```
i + 1;
```

在执行这条语句时, 计算出i+1的值。既然放弃保存此值, 或者至少放弃以某种方式使用这个值, 那么此值就丢失了。

问: 但是类似i = 1;这样的语句会如何呢? 没有发现它被丢掉。

答: 不要忘记在C语言中=是一种运算符, 它可以和其他运算符一样产生值。如下赋值语句:

```
i = 1;
```

把1赋值给i。整个表达式的值就是1, 这个值将被丢掉。既然编写语句的首要目的是为了改变i的值, 那么丢掉表达式的值就没有什么大的损失。

59

练习

4.1节

1. 列出下列每段代码的输出结果。假设i、j和k都是int型变量。

- (a) i = 5; j = 3;
printf("%d %d", i / j, i % j);
- (b) i = 2; j = 3;
printf("%d", (i + 10) % j);
- (c) i = 7; j = 8; k = 9;
printf("%d", (i + 10) % k / j);
- (d) i = 1; j = 2; k = 3;
printf("%d", (i + 5) % (j+2) / k);

*2. 如果i和j都是正整数, 是否(-i)/j的值和-(i/j)的值始终一样? 验证你的答案。

3. 编写程序实现数字反向, 即根据用户输入的两位数, 反向显示出该数相应位上数字。要求程序执行过程中需要具有下列显示信息:

```
Enter a two-digit number: 28  
The reversal is: 82
```

用%d读入两位数, 然后分解成两个数字。提示: 如果n是整数, 那么n%10的结果是数n中的最后一位数字, 而n/10的结果则是移除n中的最后一位数字。

4. 扩展练习3中的程序使其可以处理三位数的反向。

5. 重新编写练习4中的程序, 使新程序不用数学式的分割数字方法就可以显示出三位的反向数。提示: 复习程序upc.c。

4.2节

6. 列出下列每段代码的输出结果。假设i、j和k都是int型变量。

- (a) i = 7; j = 8;
i *= j + 1;
printf("%d %d", i, j);
- (b) i = j = k = 1;
i += j += k;
printf("%d %d %d", i, j, k);

- (c) `i = 1; j = 2; k = 3;`
`i -= j -= k;`
`printf("%d %d %d", i, j, k);`
- (d) `i = 2; j = 1; k = 0;`
`i *= j *= k;`
`printf("%d %d %d", i, j, k);`

4.3节

*7. 列出下列每段代码的输出结果。假设i、j和k都是int型变量。

- 60 (a) `i = 1;`
`printf("%d ", i++ - 1);`
`printf("%d", i);`
- (b) `i = 10; j = 5;`
`printf("%d ", i++ - ++j);`
`printf("%d %d", i, j);`
- (c) `i = 7; j = 8;`
`printf("%d ", i++ - --j);`
`printf("%d %d", i, j);`
- (d) `i = 3; j = 4; k = 5;`
`printf("%d ", i++ - j++ + --k);`
`printf("%d %d %d", i, j, k);`

8. 表达式`++i`和`i++`中只有一个与表达式`(i += 1)`完全相同的，哪一个是呢？验证你的答案。

4.4节

9. 用圆括号来显示C语言编译器解释下列表达式的方法。

- (a) `a * b - c * d + e`
(b) `a / b % c / d`
(c) `- a - b + c - + d`
(d) `a * - b / c - d`

*10. 表达式`(i++) + (i--)`共有多少种可能的值？假设i初始值为1，这些值都是什么？

4.5节

11. 请描述执行下列每条表达式语句后的效果。（假设i的初始值为1，j的初始值为2。）

- 61 (a) `i += j;`
(b) `i--;`
(c) `i * j / i;`
(d) `i % ++j;`

选择语句

不应该以聪明才智和逻辑分析能力来评判程序员，而要看其分析问题的全面性。

尽管C语言有许多运算符，但是它所拥有的语句却相对很少。到目前为止只遇到了两种语句：return语句（>2.2.3节）和表达式语句（>4.5节）。根据语句执行过程中顺序所产生的影响方式，C语言的其他语句大多属于以下3大类：

- **选择语句 (selection statement)**。`if`语句和`switch`语句允许程序在一组可选项中选择一条特定的执行路径。
- **循环语句 (iteration statement)**。`while`语句、`do`语句和`for`语句支持重复（循环）操作。
- **跳转语句 (jump statement)**。`break`语句、`continue`语句和`goto`语句引起无条件地跳转到程序中的某个位置。（`return`语句也属于此类。）

C语言还有其他两类语句，一类是由几条语句组合成一条语句的复合语句，一类是不执行任何操作的空语句。

本章讨论选择语句和复合语句。（第6章介绍循环语句、跳转语句和空语句。）5.2节说明`if`语句和复合语句，也介绍了可以用来测试表达式中条件的**条件运算符**（`?:`）。5.3节描述`switch`语句。然而，在能够使用`if`语句之前，我们需要了解**逻辑表达式**：`if`语句可以测试的条件。5.1节说明如何用**关系运算符**（`<`、`<=`、`>`和`>=`）、**判等运算符**（`==`和`!=`）和**逻辑运算符**（`&&`、`||`和`!`）构造逻辑表达式。

63

5.1 逻辑表达式

包括`if`语句在内的某些C语句都必须测试表达式的值是“真”还是“假”。例如，`if`语句可能需要检测表达式`i < j`，真值将说明`i`是小于`j`的。在某些编程语言中，类似`i < j`这样的表达式都具有特殊的“布尔”类型或“逻辑”类型。这样的类型只有两个值，即假值和真值；而C语言没有这种类型。相反，诸如`i < j`这样的比较运算会产生整数：0（假值）或1（真值）。依据这一点，下面来看看用于构建逻辑表达式的运算符。

5.1.1 关系运算符

C语言的关系运算符（relational operator）（表5-1）和数学的`<`、`>`、`≤`和`≥`运算符相对应，只是前者用在C语言的表达式中时产生的结果是0（假）或1（真）。例如，表达式`10 < 11`的值为1，而表达式`11 < 10`的值为0。

可以用关系运算符比较整数和浮点数，以及允许的混合类型的操作数。因此，表达式`1 < 2.5`的值为1，而表达式`5.6 < 4`的值为0。

表5-1 关系运算符

| 符 号 | 含 义 |
|-----|-------|
| < | 小于 |
| > | 大于 |
| <= | 小于或等于 |
| >= | 大于或等于 |

关系运算符的优先级低于算术运算符。例如，表达式 $i+j < k-1$ 意味着 $(i+j) < (k-1)$ 。关系运算符都是左结合的。



表达式 $i < j < k$ 在C语言中是合法的，但是它不是你所期望的意思。因为<运算符是左结合的，所以这个表达式等价于

$(i < j) < k$

换句话说，表达式首先检测 i 是否小于 j ，然后用比较后产生的结果 1 或 0 来和 k 进行比较。表达式不测试 j 是否位于 i 和 k 之间。（在本节后面的内容中将会看到正确的表达式应该是 $i < j \&\& j < k$ 。）

64

5.1.2 判等运算符

虽然C语言的关系运算符和许多其他编程语言中表示的符号相同，但是判等运算符（equality operator）却有着独一无二的形式（表5-2）。因为单独一个=字符表示赋值运算符，所以“等于”运算符是两个紧邻的=字符，而不是一个=字符。“不等于”运算符也是两个字符，即!=和==。

表5-2 判等运算符

| 符 号 | 含 义 |
|-----|-----|
| == | 等于 |
| != | 不等于 |

和关系运算符一样，判等运算符也是左结合的，而且也是产生 0（假）或 1（真）作为结果。然而，判等运算符的优先级低于关系运算符。例如，表达式 $i < j == j < k$ 等价于表达式 $(i < j) == (j < k)$ 。如果 $i < j$ 和 $j < k$ 的结果同为真或同为假，那么表达式的结果为真。

聪明的程序员有时会利用关系运算符和判等运算符返回整数值这一事实。例如，依据 i 是否小于、大于或等于 j ，得出表达式 $(i >= j) + (i == j)$ 的值分别是 0、1 或 2。然而，这种技巧性编码通常不是一个好主意，因为这样会使程序难以阅读。

5.1.3 逻辑运算符

用逻辑运算符（logical operator）构成的较简单的表达式可以构建出更加复杂的逻辑表达式。这些逻辑运算符包括与、或和非（表5-3）。!是一元运算符，而&&和||是二元运算符。

表5-3 逻辑运算符

| 符 号 | 含 义 |
|-----|-----|
| ! | 逻辑非 |
| && | 逻辑与 |
| | 逻辑或 |

逻辑运算符所产生的结果是0或1。操作数经常会有0或1的值，但这不是必需的。逻辑运算符将任何非零值操作数作为真值来处理，同时将任何零值操作数作为假值来处理。

逻辑运算符的操作如下：

- 如果表达式的值为0，那么!表达式的结果为1。
- 如果表达式1和表达式2的值都是非零值，那么表达式`1 && 表达式2`的结果为1。
- 如果表达式1或表达式2的值中任意一个是（或者两者都是）非零值，那么表达式`1 || 表达式2`的结果为1。

在所有其他情况下，这些运算符产生的结果都为0。

运算符`&&`和运算符`||`都对操作数进行“短路”计算。也就是说，这些运算符首先计算出左侧操作数的值，然后是右侧操作数；如果表达式的值可以由左侧操作数的值单独推导出来，那么将不计算右侧操作数的值。思考下面的表达式：

```
(i != 0) && (j / i > 0)
```

为了得到此表达式的值，首先必须计算表达式`(i != 0)`的值。如果*i*不等于0，那么需要计算表达式`(j / i > 0)`的值，从而确定整个表达式的值为真还是为假。但是，如果*i*等于0，那么整个表达式的值一定为假，所以就不需要计算表达式`(j / i > 0)`的值了。短路计算的优势是显而易见的，如果没有短路计算，那么表达式的求值将会导致除以零的运算。



注意逻辑表达式的副作用。有了运算符`&&`和运算符`||`的短路特性，操作数的副作用并不会总发生。思考下面的表达式：

```
i > 0 && ++j > 0
```

虽然*j*因为表达式计算的副作用进行了自增操作，但是并不总是这样。如果*i > 0*的结果为假，将不会计算表达式`++j > 0`，那么*j*也就不会进行自增。把表达式的条件变成`++j > 0 && i > 0`，就可以解决这种短路问题。或者更好的办法是单独对*j*进行自增操作。

运算符`!`的优先级和一元正号、负号的优先级相同。运算符`&&`和运算符`||`的优先级低于关系运算符和判等运算符。例如，表达式`i < j && k == m`等价于表达式`(i < j) && (k == m)`。运算符`!`是右结合的，而运算符`&&`和运算符`||`都是左结合的。

5.2 if语句

`if`语句允许程序通过测试表达式的值从两种选项中选择一种。`if`语句的最简单格式如下：

[if语句] `if (表达式) 语句`

注意，表达式两边的圆括号是必需的，它们是`if`语句的组成部分，而不是表达式的内容。还要注意的是，和在其他一些语言中的用法不同，单词`then`没有出现在圆括号的后边。

执行`if`语句时，先计算圆括号内表达式的值。如果表达式的值非零，那么接着执行圆括号后边的语句，C语言把此非零值解释为真值。下面是一个示例：

```
if (line_num == MAX_LINES)
    line_num = 0;
```

如果条件`line_num == MAX_LINES`的结果为真（有非零值），那么执行语句`line_num = 0;`。



不要混淆（判等）运算符`==`和（赋值）运算符`=`。语句`if (i == 0) ...`测试*i*是否等于0。然而，语句`if (i = 0) ...`则是先把0赋值给*i*，然后测试赋值表达式

的结果是否是非零值。在这种情况下，测试总是会失败的。

也许因为在数学（和许多其他编程语言）中意味着“等于”，所以把（判断）运算符`==`与（赋值）运算符`=`相混淆是最常见的C语言编程错误。**Q&A**如果注意到应该正常出现运算符`==`的地方出现的是运算符`=`，那么一些编译器会产生诸如“Possibly incorrect assignment”这类警告。

通常，`if`语句中的表达式能判定变量是否落在某个数值范围内。例如，为了判定 $0 \leq i < n$ 是否成立，最好写成

[惯用法] `if (0 <= i && i < n) ...`

为了判定相反的情况（`i`在范围之外），最好写成

[惯用法] `if (i < 0 || i >= n) ...`

注意用运算符`||`代替运算符`&&`。

5.2.1 复合语句

在`if`语句模板中，注意语句是单独一条语句而不是多条语句：

`if (表达式) 语句`

如果想用`if`语句处理两条或更多条语句，那么该怎么办呢？可以引入复合语句（compound statement）。复合语句有如下格式：

[复合语句] { 多条语句 }

67 通过在一组语句周围放置大括号的方法，可以强制编译器将其作为单独一条语句来处理。

下面是一个复合语句的示例：

```
{ line_num = 0; page_num++; }
```

为了表示清楚，通常将一条复合语句放在多行内，每行有一条语句，如下所示：

```
{
    line_num = 0;
    page_num++;
}
```

注意，每条内部语句始终是以分号结尾，而复合语句本身却不是。

下面是在`if`语句内部使用复合语句的形式：

```
if (line_num == MAX_LINES) {
    line_num = 0;
    page_num++;
}
```

复合语句也常出现在循环和其他需要多条语句，但C语言的语法却要求单独的一条语句的地方。

5.2.2 `else`子句

`if`语句可以有`else`子句：

[含有else子句的if语句] `if (表达式) 语句 else 语句`

如果在圆括号内的表达式的值为0，那么就执行`else`后边的语句。

下面是一个含有else子句的if语句的示例：

```
if (i > j)
    max = i;
else
    max = j;
```

注意，两条“内部”语句都是以分号结尾的。

if语句包含else子句时，出现了格式设计问题：应该把else放置在哪里呢？和前面的例子一样，许多C语言程序员把它和if对齐排列在语句的起始位置。内部语句通常采用缩进格式。但是，如果内部语句很短，则可以把它们与if和else放置在同一行内：

```
if (i > j) max = i;
else max = j;
```

C语言对可以出现在if语句内部的语句的类型没有限制。事实上，在if语句内部嵌套其他if语句是非常普遍的：

```
if (i > j)
    if (i > k)
        max = i;
    else
        max = k;
else
    if (j > k)
        max = j;
    else
        max = k;
```

if语句可以嵌套任意层数。注意，把每个else同与它匹配的if对齐排列，这样做很容易辨别嵌套层次。如果发现嵌套仍然很混乱，那么不要犹豫，直接增加大括号就可以了：

```
if (i > j) {
    if (i > k)
        max = i;
    else
        max = k;
} else {
    if (j > k)
        max = j;
    else
        max = k;
}
```

为语句增加大括号就像在表达式中使用圆括号一样，即使有时并不是必需的。这两种方法都可以使程序更加容易阅读，而且同时可以避免编译器不能像程序员一样去理解程序。

5.2.3 级联式if语句

编程时常常需要判定一系列的条件，一旦其中某一种条件为真就立刻停止。“级联式”if语句常常是编写这类系列判定的最好方法。例如，下面这个级联式if语句用来判定n是小于0，等于0，还是大于0：

```
if (n < 0)
    printf("n is less than 0\n");
else
    if (n == 0)
        printf("n is equal to 0\n");
    else
        printf("n is greater than 0\n");
```

虽然第二个if语句是嵌套在第一个if语句内部的，但是C语言程序员通常不会对它进行缩进，而是把每个else都与最初的if对齐：

```

if (n < 0)
    printf("n is less than 0\n");
else if (n == 0)
    printf("n is equal to 0\n");
else
    printf("n is greater than 0\n");

```

这样的安排带给级联式if语句独特的书写形式：

```

if (表达式)
    语句
else if (表达式)
    语句
...
else if (表达式)
    语句
else
    语句

```

当然，这种格式中的最后两行（else语句）不是总出现。这种缩进级联式if语句的方法避免了判定数量过多时过度缩进的问题。此外，这样也向读者证明了这组语句只是一连串的判定。

请记住，级联式if语句不是新的语句类型，它仅仅是普通的if语句，只是碰巧有另外一条if语句作为else子句的替换（而且这条替换的if语句又有了另外一条if语句作为它的else子句的替换，依次类推）。

5.2.4 程序：计算股票经纪人的佣金

当股票通过经纪人进行买卖时，经纪人的佣金往往根据股票交易额采用某种变化的比例进行计算。下面的表格显示了实际支付给经纪人的费用的数量：

| 交易额范围 | 佣金费用 |
|------------------|---------------|
| 低于2 500美元 | 30美元 + 1.7% |
| 2 500~6 250美元 | 56美元 + 0.66% |
| 6 250~20 000美元 | 76美元 + 0.34% |
| 20 000~50 000美元 | 100美元 + 0.22% |
| 50 000~500 000美元 | 155美元 + 0.11% |
| 超过500 000美元 | 255美元 + 0.09% |

最低收费是39美元。下面的程序要求用户录入交易额，然后显示出佣金的数额：

70 Enter value of trade: 30000
Commission: \$166.00

此程序的重点是用级联式if语句来确定交易额所在的范围。

```

broker.c
/* Calculates a broker's commission */

#include <stdio.h>

main()
{
    float commission, value;

    printf("Enter value of trade: ");
    scanf("%f", &value);

    if (value < 2500.00)
        commission = 30.00 + .017 * value;
    else if (value < 6250.00)

```

```

commission = 56.00 + .0066 * value;
else if (value < 20000.00)
    commission = 76.00 + .0034 * value;
else if (value < 50000.00)
    commission = 100.00 + .0022 * value;
else if (value < 500000.00)
    commission = 155.00 + .0011 * value;
else
    commission = 255.00 + .0009 * value;

if (commission < 39.00)
    commission = 39.00;

printf("Commission: $%.2f\n", commission);

return 0;
}

```

5.2.5 “悬空 else”的问题

当嵌套if语句时，千万当心著名的“悬空else”的问题。思考下面这个例子：

```

if (y != 0)
    if (x != 0)
        result = x / y;
    else
        printf("Error: y is equal to 0\n");

```

上面的else子句究竟属于哪一个if语句呢？缩进格式暗示它属于最外层的if语句。然而，C语言遵循的规则是else子句应该属于离它最近的且还未和其他else匹配的if语句。在此例中，else子句实际上属于最内层的if语句，所以正确的缩进格式应该如下所示：

```

if (y != 0)
    if (x != 0)
        result = x / y;
    else
        printf("Error: y is equal to 0\n");

```

为了使else子句属于外层的if语句，可以把内层的if语句用大括号括起来：

```

if (y != 0) {
    if (x != 0)
        result = x / y;
} else
    printf("Error: y is equal to 0\n");

```

这个示例表明了大括号的作用。如果把大括号用在本节第一个示例的if语句上，那么就不会有这样的问题了。

5.2.6 条件表达式

C语言的if语句允许程序根据条件表达式的值来执行两个操作中的一个。C语言还提供一种特殊的运算符，这种运算符允许表达式依据条件的值产生两个值中的一个。

条件运算符 (conditional operator) 由符号?和符号:组成，两个符号必须按如下格式一起使用：

[条件表达式] 表达式1 ? 表达式2 : 表达式3

表达式1、表达式2和表达式3可以是任何类型的表达式。上述结果表达式被称为**条件表达式** (conditional expression)。条件运算符是C运算符中唯一一个要求3个操作数的运算符。因此，

经常把它称为三元(ternary)运算符。

应该把条件表达式表达式1?表达式2:表达式3读作“如果表达式1成立，那么表达式2，否则表达式3。”条件表达式求值的步骤是：首先计算出表达式1的值。如果此值不为零，那么计算表达式2的值，并且计算出来的值就是整个条件表达式的值；如果表达式1的值为零，那么计算表达式3的值，并且此值是整个条件表达式的值。

下面的示例对条件运算符进行了说明：

```
int i, j, k;

i = 1;
j = 2;
k = i > j ? i : j;           /* k is now 2 */
k = (i >= 0 ? i : 0) + j;   /* k is now 3 */
```

72 在k的第一个赋值语句中，表达式*i > j*比较的结果为假，所以条件表达式*i > j ? i : j*的值为2，把这个值赋给k。在k的第二个赋值语句中，表达式*i >= 0*比较的结果为真，所以条件表达式(*i >= 0 ? i : 0*)的值为1，然后把这个值和j相加的结果为3。顺便说一下，这里的圆括号是非常必要的，因为除了赋值运算符，条件运算符的优先级低于先前介绍过的所有运算符。

条件表达式使程序更短小但也更难以阅读，所以最好是避免使用。然而，在少数地方仍会使用条件表达式，其中一个就是return语句。不再编写成下列形式：

```
if (i > j)
    return i;
else
    return j;
```

一些程序员将写为

```
return (i > j ? i : j);
```

printf函数的调用有时可能会得益于条件表达式。不用下列代码：

```
if (i > j)
    printf("%d\n", i);
else
    printf("%d\n", j);
```

而是简单写为

```
printf("%d\n", i > j ? i : j);
```

条件表达式也普遍用于某些类型的宏定义中(▶14.3.2节)。

5.2.7 布尔值

因为许多程序需要变量能存储假值或真值，所以C语言缺少适当的布尔类型可能会很麻烦。(C++因为认识到这个问题，所以新版的C++提供了内建的布尔类型。)一直采用模拟布尔型变量的方法来解决麻烦，这种模拟的方法是先声明int型变量，然后将其赋值为0或1：

```
int flag;

flag = 0;
...
flag = 1;
```

虽然这种方法可行，但是它对于程序的可读性没有多大贡献；因为没有明确地表示flag的赋值只能是布尔值，并且也没有明确指出0和1就是表示假和真。

为了使程序更加便于理解，一个好的方法是用类似TRUE和FALSE这样的名字定义宏(▶2.6节)。

```
#define TRUE 1
#define FALSE 0
```

现在对flag的赋值有了更加自然的形式：

```
flag = FALSE;
...
flag = TRUE;
```

为了判定flag是否为真，可以写成

```
if (flag == TRUE) ...
```

或者只是写成

```
if (flag) ...
```

为了判定flag是否为假，可以写成

```
if (flag == FALSE) ...
```

或者是

```
if (!flag) ...
```

为了更进一步实现这个想法，甚至可以定义用作类型的宏：

```
#define BOOL int
```

声明布尔型变量时可以用BOOL代替int：

```
BOOL flag;
```

现在就非常清楚flag不是普通的整型变量，而是表示布尔条件。（当然，编译器始终把flag看成是int型变量。）后面的章节将介绍设置布尔类型的更好的方法（类型定义（>7.6节）和枚举（>16.5节））。

5.3 switch 语句

在日常编程中，常常需要把表达式和一系列值进行比较，从中找出当前匹配的值。在5.2节已经看到，级联式if语句可以达到这个目的。例如，下面的级联式if语句根据数字的级别显示出相应的英语单词：

```
if (grade == 4)
    printf("Excellent");
else if (grade == 3)
    printf("Good");
else if (grade == 2)
    printf("Average");
else if (grade == 1)
    printf("Poor");
else if (grade == 0)
    printf("Failing");
else
    printf("Illegal grade");
```

C语言提供了switch语句作为这类级联式if语句的替换。下面的switch语句等价于前面的级联式if语句：

```
switch (grade) {
    case 4:   printf("Excellent");
               break;
    case 3:   printf("Good");
               break;
    case 2:   printf("Average");
               break;
```

```

        break;
case 1: printf("Poor");
        break;
case 0: printf("Failing");
        break;
default: printf("Illegal grade");
        break;
}

```

执行这条语句时，判定变量grade的值是否等于4、3、2、1和0。例如，如果值和4相匹配，那么显示信息Excellent，然后break语句（>6.4.1节）把控制传递给switch后边的语句。如果grade的值和列出的任何选择都不匹配，那么使用default情况，并且显示信息Illegal grade。

switch语句往往比级联式if语句更容易阅读。此外，switch语句往往比if语句执行速度快，特别是在有一连串情况要判定的时候。

Q&A switch语句的最常用格式如下：

| | |
|-------------------|--------------------------|
| [switch语句] | switch (表达式) { |
| | case 常量表达式 : 多条语句 |
| | ... |
| | case 常量表达式 : 多条语句 |
| | default : 多条语句 |
| | } |

switch语句是十分复杂的，下面逐一看一下它的组成部分：

- **控制表达式。** switch后边必须跟着由圆括号括起来的整型表达式。C语言把字符（>7.3节）当成整数来处理，因此可以在switch语句中对字符进行判定。但是，不能用浮点数和字符串。
- **情况标号。** 格式中的每一种情况都以标号开始

75

case 常量表达式 :

除了不能包含变量或函数调用，常量表达式（constant expression）更像是普通的表达式。因此，5是常量表达式， $5 + 10$ 也是常量表达式，而 $n + 10$ 不是常量表达式（除非n是表示常量的宏）。情况标号中的常量表达式必须计算出整数值（字符也可以接受）。

- **语句。** 每个情况标号的后边可以跟任意数量的语句。不需要用大括号把这些语句括起来。（好好享受这一点，这可是C语言中少数几个不需要大括号的地方。）每组语句的最后一条通常是break语句。

C语言不允许有重复的情况标号，但对情况的顺序没有要求，特别是default这种情况不一定要放置在最后。

case后边只可以跟随一个常量表达式。但是，几个情况标号可以放置在同一组语句的前面：

```

switch (grade) {
    case 4:
    case 3:
    case 2:
    case 1: printf("Passing");
              break;
    case 0: printf("Failing");
              break;
    default: printf("Illegal grade");
              break;
}

```

为了节省空间，程序员有时还会把几个情况标号放置在同一行中：

```
switch (grade) {
    case 4: case 3: case 2: case 1:
        printf("Passing");
        break;
    case 0: printf("Failing");
        break;
    default: printf("Illegal grade");
        break;
}
```

可惜的是，就像其他一些编程语言一样，C语言没有表示数值范围的情况标号。

switch语句不要求一定有default情况。如果default不存在，而且控制表达式的值和任何一种情况标号都不匹配的话，直接把控制传给switch语句后面的语句。

5.3.1 break语句的作用

现在仔细讨论一下break语句。正如已经看到的那样，执行break语句会导致程序“跳出”switch语句，并且继续执行switch后面的语句。

需要break语句的原因是由于switch语句实际上是一种“基于计算的跳转”。当计算控制表达式的数值时，控制跳到与switch表达式的值相匹配的情况标号处。情况标号只是说明switch内部位置的标记。在执行完情况中的最后一条语句后，程序控制“向下跳转”到下一种情况的第一条语句上，而忽略下一种情况的情况标号。如果没有break（或者一些其他的跳转语句）语句，程序控制将会从一种情况继续到下一种情况。思考下面的switch语句：

```
switch (grade) {
    case 4: printf("Excellent");
    case 3: printf("Good");
    case 2: printf("Average");
    case 1: printf("Poor");
    case 0: printf("Failing");
    default: printf("Illegal grade");
}
```

如果grade的值为3，那么显示的信息是

GoodAveragePoorFailingIllegal grade



忘记使用break语句是编程时常犯的错误。虽然有时会故意忽略break以便在几个情况下共享代码，但是通常情况下是因为疏忽。

既然故意从一种情况跳转到接下来的一种情况是非常少见的，那么明确指出任何break语句的故意省略会是一个好主意：

```
switch (grade) {
    case 4: case 3: case 2: case 1:
        num_passing++;
        /* FALL THROUGH */
    case 0: total_grades++;
        break;
}
```

如果没有注释，那么稍后某些人可能会通过增加不必要的break语句来修正“错误”。

虽然switch语句中最后一个case不需要break语句，但常见的做法是放一个break语句来提醒若增加case不要出现“丢失break”的问题。

5.3.2 程序：显示法定格式的日期

合同和其他法律文档中经常使用下列日期格式：

Dated this _____ day of _____, 19 ____.

编写程序用来显示这种格式的日期。用户以月/日/年的格式的录入日期，然后计算机显示出“法定”格式的日期：

```
Enter date (mm/dd/yy): 7/19/96
Dated this 19th day of July, 1996.
```

可以使用printf函数实现主要的格式化。然而，还有两个问题：如何为日添加“th”（或者“st”、“nd”、“rd”），以及如何用单词代替数字显示月份。幸运的是，switch语句可以很好地解决这两种问题；用一个switch语句负责显示日的后缀，再用另一个switch语句显示出月份名。

```
date.c
/* Prints a date in legal form */

#include <stdio.h>

main()
{
    int month, day, year;

    printf("Enter date (mm/dd/yy): ");
    scanf("%d %d %d", &month, &day, &year);

    printf("Dated this %d", day);
    switch (day) {
        case 1: case 21: case 31:
            printf("st"); break;
        case 2: case 22:
            printf("nd"); break;
        case 3: case 23:
            printf("rd"); break;
        default: printf("th"); break;
    }
    printf(" day of ");

    switch (month) {
        case 1: printf("January"); break;
        case 2: printf("February"); break;
        case 3: printf("March"); break;
        case 4: printf("April"); break;
        case 5: printf("May"); break;
        case 6: printf("June"); break;
        case 7: printf("July"); break;
        case 8: printf("August"); break;
        case 9: printf("September"); break;
        case 10: printf("October"); break;
        case 11: printf("November"); break;
        case 12: printf("December"); break;
    }

    printf(", 19%.2d.\n", year);
    return 0;
}
```

78

注意，%.2d用于显示年的最后两位数字。如果用%d代替的话，那么将错误地显示单一数字的年份（将会把1900显示成190）。

问与答

问：当我用=代替==时我所用的编译器没有发出警告。是否有一些方法可以强制编译器注意这类问题？

(p.48)

答：下面是一些程序员使用的技巧，将

```
' if (i == 0) ...
```

习惯上改写成

```
if (0 == i) ...
```

现在假设运算符==意外地写成了=:

```
if (0 = i) ...
```

因为不可能给0赋值，所以编译器将会产生错误信息。但是我没有用这种技巧，因为这样会使程序看上去很不自然。而且，这种技巧也只能用于判定条件中的一个操作数不是左值的时候。

问：针对复合语句，C语言的书籍好像使用了很多种缩进和放置大括号的风格。哪种风格最好呢？

答：根据*The New Hacker's Dictionary* (Cambridge, Mass.: MIT Press 1993) 的内容，共有4种缩进和放置大括号的风格：

- K&R风格，它是Kernighan和Ritchie合著的*The C Programming Language*一书中使用的风格，也是本书中的程序所采用的风格。在此风格中，左大括号出现在行的末尾：

```
if (line_num == MAX_LINES) {
    line_num = 0;
    page_num++;
}
```

K&R风格通过不单独在一行放置左大括号的方法来保持程序的紧凑，这也类似于许多现代语言中采用的缩进风格。缺点是：很难找到左大括号。（因为内部语句的缩进可以清楚地显示出左大括号的位置，所以这不是什么问题。）

- Allman风格，它是因Eric Allman (sendmail和其他UNIX工具的作者) 而得名的，这种风格类似于用Pascal语言编写的程序的布局方式：

```
if (line_num == MAX_LINES)
{
    line_num = 0;
    page_num++;
}
```

为了易于检查匹配，每一个大括号都单独放在一行上。

- Whitesmiths风格，它是因Whitesmiths C编译器而普及起来的风格，它指定大括号采用缩进格式：

```
if (line_num == MAX_LINES)
{
    line_num = 0;
    page_num++;
}
```

- GNU风格，它用于自由软件基金会发布的GNU软件中，它对大括号采用缩进形式，然后再进一步缩进内层的语句：

```
if (line_num == MAX_LINES)
{
    line_num = 0;
    page_num++;
}
```

使用哪种风格因人而异；没有证据表明某种风格明显比其他风格更好。无论如何，与始终如一的坚持使用相比，选择适当的风格并不是十分重要。

*The New Hacker's Dictionary*声称，Allman和Whitesmiths风格应用最为广泛，紧随其后的是K&R风格，而GNU风格则是最少使用的风格。尽管有关风格的讨论经常退化为“网络上的激烈争论”，但是一直可以在C语言的讨论区中找到这种争论。比如，K&R风格的支持者经常称其为“唯一正确的风格”。

问：如果i是int型变量，而f是float型变量，那么条件表达式(i > 0 ? i : f)是哪一种类型的值？

答：如问题中出现的那样，当int型和float型的值混合在一个条件表达式中时，表达式的类型为float型。如果i > 0为真，那么变量i转化为float型后的值就是表达式的值。

问：本章中的switch语句模板被称为是“最常用格式”，是否有其他格式呢？(p.54)

答：本章中描述的switch语句格式是较为通用的一种。例如，switch语句包含的标号（>6.4.3节）可以不用在前面放置单词case，这可能会产生有趣的（?）陷阱。假设意外地拼错了单词default：

```
switch (...) {
    ...
    defualt : ...
}
```

80

因为编译器认为defualt是一个普通标号，所以不会检查出错误。

虽然忽略掉了switch语句的某些细节，但是本章描述的格式通常足以适用于所有实际的程序。附录A对此语句给出了更加精确的描述。

问：我已见过一些缩进switch语句的方法，哪种方法最好呢？

答：至少有两种常用方法。一种方法是在每种情况标号后边放置语句：

```
switch (coin) {
    case 1: printf("Cent");
              break;
    case 5: printf("Nickel");
              break;
    case 10: printf("Dime");
              break;
    case 25: printf("Quarter");
              break;
}
```

如果每种情况只有一个简单操作（例如本例中只有printf函数的调用），那么break语句甚至可以和操作放在同一行中：

```
switch (coin) {
    case 1: printf("Cent"); break;
    case 5: printf("Nickel"); break;
    case 10: printf("Dime"); break;
    case 25: printf("Quarter"); break;
}
```

另一种方法是把语句放在情况标号的下面，并且要对语句进行缩进，从而凸现出情况标号：

```
switch (coin) {
    case 1:
        printf("Cent");
        break;
    case 5:
        printf("Nickel");
        break;
    case 10:
        printf("Dime");
        break;
    case 25:
        printf("Quarter");
        break;
}
```

81

在此格式的一种变异形式中，每一个情况标号都会和单词switch对齐排列。

在每种情况中的语句都较短小而且仅有相对较少的情况下，使用第一种方法是非常好的。第二种方法更加适合于大规模的switch语句，这种switch语句的每种情况中的语句都很复杂或数量较多。

练习

5.1节

1. 下列代码段对关系运算符和判等运算符进行了说明。假设i、j和k都是int型变量，请列出每道题中的输出结果。

- (a) i = 2; j = 3;
k = i * j = 6;
printf("%d", k);
- (b) i = 5; j = 10; k = 1;
printf("%d", k > i < j);
- (c) i = 3; j = 2; k = 1;
printf("%d", i < j = j < k);
- (d) i = 3; j = 4; k = 5;
printf("%d", i & j + i < k);
2. 下列代码段对逻辑运算符进行了说明。假设i、j和k都是int型变量，请列出每道题中的输出结果。
- (a) i = 10; j = 5;
printf("%d", !i < j);
- (b) i = 2; j = 1;
printf("%d", !!i + !j);
- (c) i = 5; j = 0; k = -5;
printf("%d", i && j || k);
- (d) i = 1; j = 2; k = 3;
printf("%d", i < j || k);
- *3. 下列代码段对逻辑表达式的短路行为进行了说明。假设i、j和k都是int型变量，请列出每道题中的输出结果。
- (a) i = 3; j = 4; k = 5;
printf("%d", i < j || ++j < k);
printf("%d %d %d", i, j, k);
- (b) i = 7; j = 8; k = 9;
printf("%d", i - 7 && j++ < k);
printf("%d %d %d", i, j, k);
- (c) i = 7; j = 8; k = 9;
printf("%d", (i = j) || (j = k));
printf("%d %d %d", i, j, k);
- (d) i = 1; j = 1; k = 1;
printf("%d", ++i || ++j && ++k);
printf("%d %d %d", i, j, k);
- *4. 编写一个单独的表达式，要求这个表达式的值根据i是否小于、等于或大于j而分别为-1、0或+1。 82

5.2节

5. 编写一个程序用来确定一个数的位数：

```
Enter a number: 374
The number 374 has 3 digits
```

假设输入的数最多不超过四位。提示：利用if语句进行数的判定。例如，如果数是在0到9之间的，那么位数为一。如果数是在10到99之间的，那么位数为二。

6. 编写一个程序，要求用户输入24小时制的时间，然后显示12小时制的格式：

```
Enter a 24-hour time: 21:11
Equivalent 12-hour time: 9:11 PM
```

注意不要把12:00显示成0:00。

7. 同时采用下列两种变化对程序broker.c进行修改。

- (a) 不再直接用交易额，而是要求用户输入股票的数量和每股的价格。
(b) 增加语句用来计算经纪人竞争对手的佣金（少于2000股时佣金为每股33美元+3美分，2000股或更大股时佣金为每股33美元+2美分）。显示原有的经纪人的佣金的同时，显示竞争对手的佣金。

8. 下面是蒲福风力等级的简单版本。蒲福风力等级是用于测量风力的。

| 速率 (=海里 / 小时) | 描述 |
|---------------|----------------|
| 小于1 | Calm (无风) |
| 1~3 | Light air (轻风) |
| 4~27 | Breeze (微风) |
| 28~47 | Gale (大风) |
| 48~63 | Storm (暴风) |
| 大于63 | Hurricane (飓风) |

编写一个程序，要求用户输入风速（按照海里/小时），然后显示相应的描述。

9. 在美国的某个州，未婚居民需要担负下列所得税：

| 收入 | 税金 |
|---------------|--------------------------|
| 未超过750美元 | 收入的1% |
| 750~2 250美元 | 7.50美元加上超出750美元部分的2% |
| 250~3 750美元 | 37.50美元加上超出2 250美元部分的3% |
| 3 750~5 250美元 | 82.50美元加上超出3 750美元部分的4% |
| 5 250~7 000美元 | 142.50美元加上超出5 250美元部分的5% |
| 超过7 000美元 | 230.00美元加上超出7 000美元部分的6% |

编写一个程序，要求用户输入需纳税的收入，然后显示税金。

10. 修改4.1节的程序upc.c，使其可以检测UPC的有效性。在用户输入UPC后，程序将显示VALID或NOT VALID。

- *11. 下列if语句在C语言中是否合法？

```
if (n >= 1 <= 10)
    printf ("n is between 1 and 10\n");
```

如果合法，那么当n等于0时语句如何执行？

- *12. 下列if语句在C语言中是否合法？

```
if (n = = 1 - 10)
    printf ("n is between 1 and 10\n");
```

如果合法，那么当n等于5时语句如何执行？

13. 如果i的值为17，那么下列语句显示的结果是什么？如果i的值为-17，那么下列语句显示的结果又是什么？

```
printf ("%d\n", i >= 0 ? i : -i);
```

5.3节

14. 利用switch语句编写一个程序，把数字显示的成绩转化为字母显示的等级：

```
Enter numerical grade: 84
Letter grade: B
```

使用下面这套等级评定规则：A=90~100，B=80~89，C=70~79，D=60~69，F=0~59。如果成绩高于100或低于0显示出错信息。提示：把成绩拆分成2个数字，然后使用switch判定十位上的数字。

15. 编写一个程序，要求用户输入一个两位的数，然后显示这个数的英文单词：

```
Enter a two-digit number: 45
You entered the number forty-five
```

提示：把数分解为两个数字。用一个switch语句显示第一位数字对应的单词（“twenty”、“thirty”等），用第二个switch语句显示第二位数字对应的单词。不要忘记11~19的数有特殊的处理要求。

- *16. 请写出下面程序段的输出结果？（假设i是整型变量。）

```
i = 1;
switch (i % 3) {
    case 0: printf("zero");
    case 1: printf("one");
    case 2: printf("two");
}
```

循 环

不值得编写没有循环和结构化变量的程序。

第5章介绍了C语言的选择语句，例如if语句和switch语句。本章将介绍C语言的重复语句，这种语句允许用户设置循环。

循环（loop）是重复执行某些其他语句（循环体）的一种语句。在C语言中，每个循环都有一个控制表达式（controlling expression）。每次执行循环体（循环重复一次）时都要对控制表达式进行计算。如果表达式为真，也就是值不为零，那么继续执行循环。

C语言提供了3种循环语句：while语句、do语句和for语句。这些语句将在6.1节、6.2节和6.3节分别进行介绍。while语句用于判定控制表达式在循环体执行之前的循环。do语句用于判定控制表达式在循环体执行之后的循环。for语句对于自增或自减计数变量的循环是十分方便的。6.3节还介绍了主要用于for语句的逗号运算符。

本章的最后两个小节致力于讨论与循环相关的C语言的特性。6.4节描述了break语句、continue语句和goto语句。break语句用来跳出循环并且把程序控制传递到循环后的下一条语句。continue语句用来跳过循环重复的剩余部分。而goto语句则可以跳到函数内任何语句上。6.5节包含了空语句的介绍，空语句可以用来构造空循环体的循环。

6.1 while语句

在C语言所有设置循环的方法中，while语句是最简单也是最基本的一种方法。while语句的格式如下所示：

85

[while语句] while (表达式) 语句

圆括号内的表达式是控制表达式；圆括号后边的语句是循环体。下面是一个示例：

```
while (i < n) /* controlling expression*/  
    i = i * 2; /* loop body */
```

注意这里的圆括号是强制要求的，而且在右括号和循环体之间没有任何内容。（一些语言要求单词do。）

执行while语句时，首先计算控制表达式的值。如果值不为零（即真值），那么执行循环体，接着再次判定表达式。先判定控制表达式，再执行循环体，这个过程持续进行直到最终控制表达式的值变为零才停止。

下面的例子使用while语句计算大于或等于数n的最小的2次幂：

```
i = 1;  
while (i < n)  
    i = i * 2;
```

假设n的值为10。下面的跟踪显示了while语句执行时的情况：

- i = 1; i现在值为1。

- $i < n$ 成立吗? 是的, 继续。
- $i = i * 2;$ i 现在值为2。
- $i < n$ 成立吗? 是的, 继续。
- $i = i * 2;$ i 现在值为4。
- $i < n$ 成立吗? 是的, 继续。
- $i = i * 2;$ i 现在值为8。
- $i < n$ 成立吗? 是的, 继续。
- $i = i * 2;$ i 现在值为16。
- $i < n$ 成立吗? 不成立, 退出循环。

注意, 只有在控制表达式 ($i < n$) 为真的情况下循环才会继续。当表达式值为假时, 循环终止, 而且就像描述的那样, 此时 i 的值是大于或等于 n 的。

虽然循环体必须是单独的一条语句, 但这只是个技术问题; 如果需要多条语句, 那么只要用一对大括号构造出一条复合语句就可以了:

```
while (i > 0) {
    printf("T minus %d and counting\n", i);
    i--;
}
```

86

即使没有严格要求的时候, 一些程序员始终都在使用大括号:

```
while (i < n) { /* braces allowed, but not required */
    i = i * 2;
}
```

作为第二个示例, 让我们一起跟踪下列语句的执行, 这些语句用来显示一串“倒数计数”信息。

```
i = 10;
while (i > 0) {
    printf("T minus %d and counting\n", i);
    i--;
}
```

在 `while` 语句执行前, 把变量 i 赋值为 10。因为 10 大于 0, 所以执行循环体, 这导致显示出信息 `T minus 10 and counting`, 同时变量 i 进行自减。然后再次判定条件 $i > 0$ 。因为 9 大于 0, 所以再次执行循环体。整个过程持续到显示信息为 `T minus 1 and counting`, 并且变量 i 的值变为 0 时停止。然后判定条件 $i > 0$ 的结果为假, 导致循环终止。

“倒数计数”这个例子可以引发对 `while` 语句的一些讨论:

- 在 `while` 循环终止时, 控制表达式的值为假。因此, 当通过表达式 $i > 0$ 控制循环终止时, i 必须是小于或等于 0 的。(否则将始终执行循环!)
- 可能根本不执行 `while` 循环体。因为控制表达式是在循环体执行之前进行判定, 所以循环体有可能一次也不执行。第一次进入“递减计数”循环时, 如果变量 i 的值是负数或零, 那么将不会执行循环。
- `while` 语句常常可以有多种写法。例如, 在 `printf` 函数调用的内部进行变量 i 的自减操作, 这种方法可以使“递减计数”循环更加简明:

Q&A `while (i > 0) printf("T minus %d and counting\n", i--);`

6.1.1 无限循环

如果控制表达式的值始终是非零值的话, `while` 语句将无法终止。事实上, C 语言程序员有时故意用非零常量作为控制表达式来构造无限循环:

[惯用法] `while (1) ...`

除非循环体含有跳出循环控制的语句(`break`、`goto`、`return`)或者调用了导致程序终止的函数，否则上述这种形式的`while`语句将永远执行下去。

6.1.2 程序：显示平方值的表格

现在要编写一个程序来显示平方值的表格。首先程序提示用户输入数`n`。然后显示出`n`行的输出，其中每一行包含两个数：一个是 $1 \sim n$ 的数，另一个则是此数的平方值。

```
This program prints a table of squares.  
Enter number of entries in table: 5
```

| | |
|---|----|
| 1 | 1 |
| 2 | 4 |
| 3 | 9 |
| 4 | 16 |
| 5 | 25 |

把期望的数的平方值存储在变量`n`中。程序需要用循环来重复显示数`i`和它的平方值，并且循环要从`i`等于1开始。如果`i`小于或等于`n`，那么循环将反复进行。需要保证的是每次执行循环时对`i`值加1。

可以使用`while`语句编写循环。(坦白地说，现在没有其他更多的选择，因为`while`语句是目前为止我们唯一掌握的循环类型。)下面是完成后的程序。

```
square.c  
/* Prints a table of squares using a while statement */  
  
#include <stdio.h>  
  
main()  
{  
    int i, n;  
  
    printf("This program prints a table of squares.\n");  
    printf("Enter number of entries in table: ");  
    scanf("%d", &n);  
  
    i = 1;  
    while (i <= n) {  
        printf("%10d%10d\n", i, i * i);  
        i++;  
    }  
  
    return 0;  
}
```

注意，程序`square.c`是如何把输出整齐地排列成两列的。窍门是使用类似`%10d`这样的转换说明代替`%d`，这样做好处是使`printf`函数在指定宽度内将输出右对齐。

6.1.3 程序：数列求和

作为`while`语句的第二个示例，现在来编写一个程序对用户输入的整数数列进行求和计算。下面显示的是用户可见的内容：

```
This program sums a series of integers.  
Enter integers (0 to terminate): 8 23 71 5 0  
The sum is: 107
```

很明显，程序需要循环，循环采用`scanf`函数读入数，然后再把这个数加到运算的总和中。

用`n`表示当前读入的数，而`sum`表示所有先前读入的数的总和，从而得到如下程序：

```
sum.c  
/* Sums a series of numbers */
```

```
#include <stdio.h>

main()
{
    int n, sum = 0;

    printf("This program sums a series of integers.\n");
    printf("Enter integers (0 to terminate): ");

    scanf("%d", &n);
    while (n != 0) {
        sum += n;
        scanf("%d", &n);
    }
    printf("The sum is: %d\n", sum);

    return 0;
}
```

注意，条件`n!=0`只在数被读入后才进行判断，而且一旦条件成立就会立刻终止循环。

6.2 do语句

do语句和while语句关系紧密。事实上，do语句本质上就是while语句，只不过do语句是在每次执行循环体之后对控制表达式进行判定的。do语句的格式如下所示：

[do语句] do 语句 while (表达式);

和处理while语句一样，do语句的循环体也必须是一条语句（当然可以用复合语句），并且控制表达式也要求用圆括号。

执行do语句时，先执行循环体，再计算控制表达式的值。如果表达式的值是非零的，那么再次执行循环体，然后再次计算表达式的值。在循环体执行后控制表达式的值变为0时，终止do语句的执行。

下面使用do语句重写前面的“倒数计数”程序：

```
i = 10;
do {
    printf("T minus %d and counting\n", i);
    --i;
} while (i > 0);
```

执行do语句时，先执行循环体，这导致显示出信息T minus 10 and counting，并且i自减。现在对条件`i > 0`进行判定。因为9大于0，所以再次执行循环体。这个过程持续到显示出信息T minus 1 and counting并且i的值变为0时终止。现在判定表达式`i > 0`的值为假，所以循环终止。正如此例中显示的一样，do语句和while语句往往没有什么区别。两种语句的区别是，至少要执行一次do语句的循环体，而while语句在控制表达式初始为0时会完全跳过不执行循环体。

顺便提一下，无论需要与否，一些C语言的程序员对所有do语句都使用大括号，这是因为没有大括号的do语句很容易被误认为是while语句：

```
do printf("T minus %d and counting\n", i--);
while (i > 0);
```

粗心的读者可能会把单词while误认为是while语句的开始。

程序：计算整数中数字的位数

虽然C程序中while语句的出现次数远远多于do语句，但是后者对于需要至少执行一次的循环来说是非常方便的。为了说明这一点，现在编写一个程序计算用户输入的整数中数字的位数：

```
Enter a nonnegative integer: 60
The number has 2 digit(s).
```

方法是把输入的整数反复除以10，直到结果变为0停止；除法的次数就是所含数字的位数。因为不知道到底需要多少次除法运算才能达到0，所以很明显程序需要某种循环。但是应该用while语句还是do语句呢？do语句显然更合适，因为每个整数，甚至是0，都至少有一位数字。下面是程序。

```
numdigit.c
/* Calculates the number of digits in an integer */

#include <stdio.h>
main()
{
    int digits = 0, n;

    printf("Enter a nonnegative integer: ");
    scanf("%d", &n);

    do {
        n /= 10;
        digits++;
    } while (n > 0);

    printf("The number has %d digit(s).\n", digits);

    return 0;
}
```

90

为了说明do语句是正确的选择，下面观察一下如果用相似的while循环替换do循环会发生什么：

```
while (n > 0) {
    n /= 10;
    digits++;
}
```

如果n初始值为0，那么将根本不执行上述这个循环，而且程序将显示出

```
The number has 0 digit(s).
```

6.3 for语句

现在开始介绍C语言循环中最后一种，也是功能最强大的一种循环：for语句。不要因为for语句表面上的复杂性而灰心；实际上，它是编写许多循环的最佳方法。for语句适和应用在使用“计数”变量的循环中，然而它也灵活用于许多其他类型的循环中。

for语句的格式如下所示：

[**for语句格式**] **for (表达式1; 表达式2; 表达式3) 语句**

这里的表达式1、表达式2和表达式3全都是表达式。下面是一个例子：

```
for (i = 10; i > 0; i--)
    printf("T minus %d and counting\n", i);
```

在执行for语句时，变量*i*先初始化为10，接着判定*i*是否大于0。因为判定的结果为真，所以打印信息T minus 10 and counting，然后变量*i*进行自减操作。随后再次对条件*i > 0*进行判定。随着变量*i*从10变化到1的过程将总共执行10次循环体。

91 for语句和while语句关系紧密。**Q&A**事实上，除了一些极少数的情况以外，for循环总可以用等价的while循环替换：

```
表达式1;
while ( 表达式2 ) {
    语句
    表达式3;
}
```

就像这个扩展显示的那样，循环开始执行前，表达式1是初始化步骤，并且只执行一次，表达式2用来控制循环的终止（只要表达式2不为零，那么将继续执行循环），而表达式3是在每次循环的最后被执行的一个操作。把这种扩展用于先前的for循环的示例中，就可以获得下面的内容：

```
i = 10;
while ( i > 0 ) {
    printf("T minus %d and counting\n" i);
    i--;
}
```

研究等价的while语句有助于更好地理解for语句。例如，假设把先前for循环示例中的*i--*用*--i*来替换：

```
for ( i = 10; i > 0; --i )
    printf("T minus %d and counting\n", i);
```

这样做会对循环产生什么样的影响呢？看看等价的while循环就会发现，这种做法对循环没有任何影响：

```
i = 10;
while ( i > 0 ) {
    printf("T minus %d and counting\n" i);
    i--;
}
```

因为for语句中第一个表达式和第三个表达式都是以语句的方式执行的，所以它们的值互不相关，也就是说刚好可以利用它们的副作用。结果是，这两个表达式常常作为赋值表达式或自增/自减表达式。

6.3.1 for语句的惯用法

92 对于“向上加”（变量自增）或“向下减”（变量自减）的循环来说，for语句通常是最好的选择。对于向上加或向下减共有n次的情况，for语句经常会采用下列形式中的一种。

- 从0向上加到n-1：

[惯用法] `for (i = 0; i < n; i++) ...`

- 从1向上加到n：

[惯用法] `for (i = 1; i <= n; i++) ...`

- 从n-1向下减到0：

[惯用法] `for (i = n-1; i >= 0; i--) ...`

- 从n向下减到1：

[惯用法] `for (i = n; i > 0; i--) ...`

模仿这些书写格式将有助于避免一些C语言初学者经常会犯的错误：

- 在控制表达式中用<代替>（或者反之亦然）。注意“向上加”的循环使用运算符<或运算符<=，而“向下减”的循环则依赖于运算符>或运算符>=。
- 在控制表达式中用 == 代替 <、<=、> 或 >=。循环开始时，控制表达式的值应该为真，为了能终止循环，稍后控制表达式的值会变为假。类似 i == n 这样的判定不能产生这种效果，因为这样的表达式的初始值不会为真。
- 编写的控制表达式用 i<=n 替代了 i<n，这类写法会“循环次数差一次”错误。

6.3.2 在 for 语句中省略表达式

for语句甚至远比目前已知的更加灵活。通常for语句用三个表达式控制循环，但是一些for循环可能不需要所有的表达式，因此C语言允许忽略任意或全部的表达式。

如果忽略第一个表达式，那么在执行循环前没有初始化的操作：

```
i = 10;  
for (; i > 0; --i)  
    printf("T minus %d and counting\n" i);
```

在这个例子中，变量i由一条单独的赋值语句实现了初始化，所以在for语句中忽略了第一个表达式。（注意，保留第一个表达式和第二个表达式之间的分号。即使已经忽略掉某些表达式，但是控制表达式必须始终有两个分号。）

如果忽略了for语句中的第三个表达式，循环体有责任要确认第二个表达式的值最终会变为假。for语句的示例可以写成如下形式：

```
for (i = 10; i > 0;)  
    printf("T minus %d and counting\n" i--);
```

93

为了补偿忽略第三个表达式产生的后果，已经安排变量i在循环体中进行自减。

当for语句同时忽略掉第一个和第三个表达式时，循环的结果和while语句没有任何分别。例如，循环

```
for (; i > 0;)  
    printf("T minus %d and counting\n", i--);
```

等价于

```
while (i > 0)  
    printf("T minus %d and counting\n", i--);
```

这里while语句的形式更清楚，也因此更可取。

如果省略第二个表达式，那么它默认为真值，因此for语句不会终止（除非以某种其他形式停止）。Q&A 例如，某些程序员用下列的for语句建立无限循环：

[惯用法] for (; ;)...

6.3.3 逗号运算符

有些时候，我们可能喜欢编写有两个（或更多个）初始表达式的for语句，或者希望在每次循环时一次对几个变量进行自增操作。使用逗号表达式（comma expression）作为for语句中第一个或第三个表达式可以实现这些想法。

逗号表达式的格式如下所示：

[逗号表达式] 表达式1, 表达式2

这里的表达式1和表达式2是任何两个表达式。逗号表达式的计算要通过两步来实现：第一步，计算表达式1并且扔掉计算出的值。第二步，计算表达式2，把这个值作为整个表达式的值。

计算表达式1始终会有副作用；如果没有，那么表达式1就没有了存在的意义。

例如，假设变量*i*和变量*j*的值分别为1和5，当计算逗号表达式 $++i, i+j$ 时，变量*i*先进行自增，然后计算*i+j*，所以表达式的值为7。（而且，显然现在变量*i*的值为2。）顺便说一句，逗号运算符的优先级低于所有其他运算符，所以不需要在 $++i$ 和*i+j*周围加圆括号。

有时需要把一串逗号表达式串联在一起，就如同某些时候把赋值表达式串联在一起一样。逗号运算符是左结合性，所以编译器把下列表达式

```
i = 1, j = 2, k = i + j
```

解释为

94

```
((i = 1), (j = 2)), (k = (i + j))
```

这样的结果是可以保证表达式*i = 1*, *j = 2*和*k = i + j*是从左向右进行计算的。

提供逗号运算符是为了在C语言要求单独一个表达式的情况下可以使用两个或多个表达式。换句话说，逗号运算符允许将两个表达式“黏贴”在一起构成单独的一个表达式。（注意，与复合语句的相似之处是，复合语句允许用户把一组语句当作是唯一的一条语句来使用。）

不是经常需要把多个表达式粘连在一起的。正如后面的章节将介绍的那样，某些宏（>14.3节）的类型可以从逗号运算符中受益。`for`语句是唯一除上述之外还可以发现逗号运算符的地方。例如，假设在进入`for`语句时希望初始化两个变量。可以把原来的程序

```
sum = 0;
for (i = 1; i <= N; i++)
    sum += i;
```

改写为

```
for (sum = 0, i = 1; i <= N; i++)
    sum += i;
```

表达式`sum = 0, i = 1`首先把0赋值给`sum`，然后把1赋值给`i`。利用附加的逗号运算符，`for`语句符可以初始化两个以上的变量。

6.3.4 程序：显示平方值的表格（改进版）

程序square.c（6.1节）可以通过将`while`循环转化为`for`循环的方式进行改进：

```
square2.c
/* Prints a table of squares using a for statement */

#include <stdio.h>

main()
{
    int i, n;

    printf("This program prints a table of squares.\n");
    printf("Enter number of entries in table: ");
    scanf("%d", &n);

    for (i = 1; i <= n; i++)
        printf("%10d%10d\n", i, i * i);

    return 0;
}
```

利用这个程序可以说明关于`for`语句的一个重要的观点。C语言中的`for`语句比其他相似编程语言中的`for`语句功能更加强大，而且也会带来更多潜在的问题，这是因为C语言对控制循环行为的三个表达式没有加任何限制。虽然这些表达式通常对同一个变量进行初始化、判定和更

新，但是没有要求它们之间以任何方式进行相互关联。思考同一个程序的另一个书写版本：

```
square3.c
/* Prints a table of squares using an odd method */

#include <stdio.h>

main()
{
    int i, n, odd, square;

    printf("This program prints a table of squares.\n");
    printf("Enter number of entries in table: ");
    scanf("%d", &n);

    i = 1;
    odd = 3;
    for (square = 1; i <= n; odd += 2) {
        printf("%10d%10d\n", i, square);
        ++i;
        square += odd;
    }

    return 0;
}
```

上述程序中的for语句初始化了一个变量（square），对另一个变量（i）进行判定，并且对第三个变量（odd）进行了自增操作。变量i是要计算平方值的数，变量square是变量i的平方值，而变量odd是一个奇数，需要用它来和当前平方值相加而获得下一个平方值（允许程序不执行任何乘法操作而计算连续的平方值）。

for语句这种极大的灵活性有时是十分有用的。将会发现在链表（>17.5节）中使用for语句的这种灵活性会获得极大的收益。但是，很容易会错误地使用for语句，所以请不要走极端。为了清楚地表示循环是由变量i来控制，可以对循环的各部分内容进行重新安排，这样会使得程序square3.c中的for语句更加清楚明了。事实上，在习题中会要求读者完成上述这个练习。

6.4 退出循环

我们已经知道编写循环时在循环体之前（使用while语句和for语句）或之后（使用do语句）设置退出点的方法。然而，有些时候也会需要在循环中间设置退出点。甚至可能需要对循环设置多个退出点。break语句可以用于有上述这些需求的循环中。

96

在学习完break语句之后，还将看到一对相关的语句：continue语句和goto语句。continue语句会跳过部分循环重复执行的内容，但是不跳出整个循环。goto语句允许程序从一条语句跳转到另一条语句。由于已经有了break和continue这样有效的语句，所以很少使用goto语句。

6.4.1 break语句

前面已经讨论过break语句把程序控制从switch语句中转移出来的方法。break语句还可以用于跳出while、do或for循环。

假设编写程序用来测试数n是否为素数。计划编写for语句，这条语句可以用n除以2到n-1之间的所有数。一旦发现任何约数就跳出循环，而不需要继续尝试下去。在循环终止后，可以用if语句来确定是提前终止（因此n不是素数）还是正常终止（n是素数）：

```
for (d = 2; d < n; d++)
```

```

if (n % d == 0) break;
if (d < n)
    printf("%d is divisible by %d\n", n, d);
else
    printf("%d is prime\n", n);

```

对于在循环体的中间设置退出点，而不是在循环的开始或结束处设置退出点的情况，break语句是特别有用的。有些循环读入用户输入，并且在遇到特殊输入值时终止，这些循环常常属于下面这种类型：

```

for (;;) {
    printf("Enter a number (enter 0 to stop): ");
    scanf("%d", &n);
    if(n == 0) break;
    printf("%d cubed is %d\n", n, n*n*n);
}

```

break语句把程序控制从最内层封闭的while、do、for或switch语句中转移出来。因此，当这些语句出现嵌套时，break语句只能跳出一层嵌套。思考一下switch语句嵌套在while语句中的情况：

```

while (...) {
    switch (...) {
        ...
        break;
        ...
    }
}

```

97

break语句可以把程序控制从switch语句中转移出来，但是却不能跳出循环。后面会继续讨论这一点。

6.4.2 continue语句

continue语句并不是真的属于这一部分内容，因为它无法跳出循环。但是，它和break类似，所以归结在本节也不是完全随意的。break语句把程序控制正好转移到循环体末尾之后，而continue语句把程序控制正好转移到循环体结束之前的一点。用break语句会使程序控制跳出循环；用continue语句，会把程序控制留在循环内。break语句和continue语句的另外一个区别：break语句可以用于switch语句和循环（while、do和for），而continue语句只能用于循环。

下面的例子通过读入一串数字并求和的操作说明了continue语句的简单应用。例子要求在读入10个非零数后循环终止。无论何时读入数0就执行continue语句，控制将跳过循环体的剩余部分（即语句sum += i;和语句n++;），但是还将继续留在循环中。

```

n = 0;
sum = 0;
while (n < 10) {
    scanf("%d", &i);
    if (i == 0) continue;
    sum += i;
    n++;
    /* continue jumps to here */
}

```

如果不用continue语句，上述示例可以写成如下形式：

```

n = 0;
sum = 0;
while (n < 10) {
    scanf("%d", &i);
}

```

```

if (i != 0) {
    sum += i;
    n++;
}
}

```

6.4.3 goto语句

break语句和continue语句都是跳转语句：它们把控制从程序中的一个位置转移到另一个位置。然而，这两者都是受限制的；break语句的目标是在闭合的循环结束之后的那一点，而continue语句的目标是循环结束之前的那一点。另一方面，goto语句则可以跳转到函数中任何有标号的语句处。

98

标号只是放置在语句开始处的标识符：

[标号语句] 标识符 : 语句

语句可以有多个标号。goto语句自身的格式如下：

[goto语句] goto 标识符;

执行goto语句可以把控制转移到标号后的语句上，而且这些语句必须和goto语句本身在同一个函数中。

如果C语言没有break语句，下面是能用goto语句提前退出循环的方法：

```

for (d = 2; d < n; d++)
    if (n % d == 0) goto done;
done:
if (d < n)
    printf("%d is divisible by %d\n", n, d);
else
    printf("%d is prime\n", n);

```

Q&A goto语句是早期编程语言的主要内容，但在日常C语言编程中却很少用到它了。break、continue和return这些语句本质上都是受限制的goto语句，它们和exit函数（>9.5节）一起足够应付其他编程语言中大多数需要goto语句的情况。

虽然如此，goto语句偶尔还是很有用的。考虑从包含switch语句的循环中退出的问题。正如前面看到的那样，break语句不会产生期望的效果：它可以跳出switch语句，但是无法跳出循环。而goto语句解决了这个问题：

```

while (...) {
    switch (...) {
    ...
    goto loop_done; /* break won't work here */
    ...
}
loop_done: ...

```

goto语句对于嵌套循环的退出也是很有用的。

6.4.4 程序：账目簿结算

许多简单的交互式程序都是基于菜单的：它们向用户显示可供选择的命令列表；一旦用户选择了某条命令，程序就执行相应的操作，然后提示用户输入下一条命令；这个过程一直会持续到用户选择“退出”或“停止”命令。

99

这类程序的核心显然是循环。循环内将有语句提示用户输入命令，读命令，然后确定执行操作的内容：

```
for (;;) {
    提示用户录入命令;
    读入命令;
    执行命令;
}
```

执行这个命令将需要switch语句（或者级联式if语句）：

```
for (;;) {
    提示用户录入命令;
    读入命令;
    switch(命令) {
        case 命令1: 执行操作1; break;
        case 命令2: 执行操作2; break;
        ...
        case 命令n: 执行操作n; break;
        default: 显示错误信息; break;
    }
}
```

为了说明这种格式，开发一个程序用来维护账目簿的余额。程序将为用户提供选择菜单：刷新账户余额，往账户上存钱，从账户上取钱，显示当前余额，退出程序。选择项分别用整数0、1、2、3和4表示。程序运行后的形式显示如下：

```
*** ACME checkbook-balancing program ***
Commands: 0=clear, 1=credit, 2=debit, 3=balance, 4=exit

Enter command: 1
Enter amount of credit: 1042.56
Enter command: 2
Enter amount of debit: 133.79
Enter command: 1
Enter amount of credit: 1754.32
Enter command: 2
Enter amount of debit: 1400
Enter command: 2
Enter amount of debit: 68
Enter command: 2
Enter amount of debit: 50
Enter command: 3
Current balance: $1145.09
Enter command: 4
```

100

当用户录入命令4（即退出）时，程序需要从switch语句以及外围的循环中退出。break语句不可能做到，同时我们又不想使用goto语句。因此，决定采用return语句，这条语句将可以使程序终止并且返回操作系统。

```
checking.c
/* Balances a checkbook */

#include <stdio.h>

main()
{
    int cmd;
    float balance = 0.0, credit, debit;

    printf("**** ACME checkbook-balancing program ***\n");
    printf("Commands: 0=clear, 1=credit, 2=debit, ");
    printf("3=balance, 4=exit\n\n");
    for (;;) {
        printf("Enter command: ");
```

```

scanf("%d", &cmd);
switch (cmd) {
    case 0:
        balance = 0.0;
        break;
    case 1:
        printf("Enter amount of credit: ");
        scanf("%f", &credit);
        balance += credit;
        break;
    case 2:
        printf("Enter amount of debit: ");
        scanf("%f", &debit);
        balance -= debit;
        break;
    case 3:
        printf("Current balance: $%.2f\n", balance);
        break;
    case 4:
        return 0;
    default:
        printf("Commands: 0=clear, 1=credit, 2=debit, ");
        printf("3=balance, 4=exit\n\n");
        break;
}
}

```

注意，`return`语句后不能紧跟`break`语句。紧跟在`return`语句后边的`break`语句永远不会执行，而且许多编译器还将显示警告信息。

101

6.5 空语句

语句可以为空，也就是除了末尾处的分号以外什么符号也没有。下面是一个示例：

i = 0; i < j = 1;

这行含有三条语句：一条语句是给 i 赋值，一条是空语句，还有一条是给 j 赋值。

Q&A 空语句主要有一个好处：编写空循环体的循环。正如前面6.4节中寻找素数的循环：

```
for (d = 2; d < n; d++)
    if (n % d == 0) break;
```

如果把条件 $n \% d == 0$ 移到循环控制表达式中，那么循环体就会变为空：

```
for (d = 2; d < n && n % d != 0; d++)
    ; /* empty body */
```

每次执行循环时，先判定条件 $d < n$ 。如果结果为假，循环终止。否则，判定条件 $n \% d == 0$ ，如果结果为假则终止循环。（在后面的例子中， $n \% d == 0$ 必须为真；换句话说，找到了 n 的一个约数。）

注意，如何把空语句单独放置在一行，而不是写成

```
for (d = 2; d < n && n % d != 0; d++) ;
```

Q&A C程序员习惯性把空语句单独放置在一行。否则，一些人阅读程序时可能会混淆for语句后边的语句是否是其循环体：

```
for (d = 2; d < n && n % d != 0; d++);
if (d < n)
    printf("%d is divisible by %d\n", n, d);
```

把普通循环转化成带空循环体的循环不会带来很大的好处：新循环往往更简洁，但通常不

102



空语句会引发一类缺陷。不小心在if、while或for语句的圆括号后放置了分号，会造成语句的提前结束，而编译器是无法检测出这类错误的。

- if语句中，在圆括号后边放置分号，这样无论控制表达式的值是什么，显然会使得if语句执行同样的动作：

```
if (d == 0);                                /*** WRONG ***/
    printf("Error: Division by zero\n");
```

因为printf函数调用不在if语句内，所以无论d的值是否等于0，都会执行此函数调用。

- while语句中，在圆括号后边放置分号，这样做会产生无限循环：

```
i = 10;
while (i > 0);                                /*** WRONG ***/
{
    printf("T minus %d and counting\n", i);
    --i;
}
```

另一种可能是循环终止，但是在循环终止后只执行一次循环体语句。

```
i = 11;
while (--i > 0);                                /*** WRONG ***/
    printf("T minus %d and counting\n", i);
```

这个例子显示如下信息：

T minus 0 and counting

- for语句中，在圆括号后边放置分号会导致只执行一次循环体语句：

```
for (i = 10; i > 0; i--);                      /*** WRONG ***/
    printf("T minus %d and counting\n", i);
```

这个例子也显示出如下信息：

T minus 0 and counting

问与答

问：出现在6.1节中的循环

```
while (i > 0) printf ("T minus %d and counting\n", i --);
```

为什么不能通过删除“> 0”来缩短循环呢？

103

```
while (i) printf ("T minus %d and counting\n", i --);
```

这种写法的循环会在i达到0值时停止，所以它应该和原始版本一样好。（p.62）

答：新写法可能是更加简洁，而且许多C程序员愿意书写这种形式的循环。但是，它也有缺点。

首先，新循环不像原始版本那样容易阅读。新循环可以清楚地显示出在i达到0值时循环终止，但是不能清楚地表示是向上计数还是向下计数。而在原始的循环中，根据控制表达式*i > 0*可以推导出来这种向上还是向下的信息。

其次，如果循环开始执行时i碰巧为负值，那么新循环的行为会不同于原始版本。原始循环会立刻终止，而新循环则不会。

问：6.3节提到，大多数for循环可以利用标准模式转换成while循环。为什么不能是所有for循环呢？（p.66）

答：当for循环体中含有continue语句时，6.3节中显示的while语句格式将不再有效。思考下面这个来自6.4节的示例：

```
n = 0;
sum = 0;
while (n < 10) {
    scanf("%d", &i);
    if (i == 0) continue;
    sum += i;
    n++;
}
```

乍看之下，好像可以把while循环转化成for循环：

```
sum = 0;
for (n = 0; n < 10; n++) {
    scanf("%d", &i);
    if (i == 0) continue;
    sum += i;
}
```

但是，这个循环并不等价于原始循环。当i等于0时，原始循环并没有对n进行自增操作，但是新循环却做了。

问：哪个无限循环格式是更可取的，while(1)还是for(;;)? (p.67)

答：C程序员传统上喜欢for(;;)的高效性；因为早期的编译器经常强制程序在每次执行while循环时测试条件1。但是，对于现代编译器来说，在性能上两种无限循环没有差别。

问：听说程序员应该永不使用continue语句。这是真的吗？

答：continue语句的确很少使用。尽管如此，continue语句有时还是非常方便的。假设编写的循环要读入一些输入的数据，并且测试它是否有效，接下来，如果有效则以某种方法进行处理。如果有许多有效性测试，或者如果它们都很复杂，那么continue语句就非常有用了。循环将类似于下面这样：

```
for(;;) {
    读入数据;
    if (数据的第一条测试失败)
        continue;
    if (数据的第二条测试失败)
        continue;
    ...
    if (数据的最后一条测试失败)
        continue;
    处理数据;
}
```

问：关于goto语句什么是不好的？(p.71)

答：goto语句不是天生的魔鬼；只是通常它有更好的替代方式。使用过多goto语句的程序会迅速退化成“垃圾代码”，因为控制可以随意地跳来跳去。垃圾代码是非常难于理解和修改的。

由于goto语句可以跳前跳后，所以使得程序难于阅读。（对应的，break语句和continue语句只是往前跳。）含有goto语句的程序经常为了跟着流程控制要求阅读者跳后和跳前。

goto语句使程序难于修改，因为它可能会使某段代码用于多种不同的目的。例如，既可以通过前面语句的“失败”，也可以通过多条goto语句中的一条到达前面放置了标号的语句上。

问：除了说明循环体为空外，空语句还有其他用途吗？(p.73)

答：非常少。因为空语句可以放在任何允许放语句的地方，所以仍然有一些潜在的空语句的用途。但事实上，空语句还有另外一种用途，只是极少用到。

假设需要在复合语句的末尾放置标号。标号不能独立存在，它必须有语句跟在后边。在标号后放置空语句就可以解决这个问题：

```
{
    ...
}
```

105

```

    goto end_of_stmt;
    ...
end_of_stmt: ;
}

```

问：除了把空语句单独放置在一行以外，是否还有其他一些方法可以凸现出空循环体？(p.73)

答：一些程序员使用虚空的continue语句：

```

for (d = 2; d < n && n % d != 0; d++)
    continue;

```

另一些人使用空的复合语句：

```

for (d = 2; d < n && n % d != 0; d++)
{
}

```

练习

6.1节

1. 编写程序，要求找到用户输入的一串数中的最大数。程序需要提示用户一个一个输入数。当用户输入0或负数时，程序必须显示输入的最大非负数：

```

Enter a number: 60
Enter a number: 38.3
Enter a number: 4.89
Enter a number: 100.62
Enter a number: 75.2295
Enter a number: 0

```

The largest number entered was 100.62

注意，输入的数不要求一定是整数。

2. 编写程序，要求用户输入两个整数，然后计算并显示这两个整数的最大公约数 (GCD)：

```

Enter two integers: 12 28
Greatest common divisor: 4

```

提示：求最大公约数的经典算法是Euclid算法，方法如下：分别让变量m和n存储两个数的值；用m除以n；把除数保存在m中，而把余数保存在n中；如果n为0，那么停止操作，m中的值是GCD；否则，从m除以n开始，重复上述除法过程。

3. 编写程序，要求用户输入一个分数，然后将其约分为最简分式：

```

Enter a fraction: 6/12
In lowest terms: 1/2

```

提示：为了把分数约分为最简分式，首先计算分子和分母的最大公约数；然后分子和分母分别都除以最大公约数。

4. 在5.2节的broker.c程序中添加循环以便用户可以输入多笔交易，并且程序可以计算每次的佣金。程序在用户输入的交易额为0时终止。

```

Enter value of trade: 30000
Commission: $166.00
Enter value of trade: 20000
Commission: $144.00
Enter value of trade: 0

```

106

6.2节

5. 第4章中的练习3要求编写程序可以显示两位数字的反向。设计一个程序可以实现一位、两位、三位或者多位数的反向。提示：使用do循环重复除以10，直到值达到0为止。

6.3节

6. 编写程序，要求提示用户输入一个数n，然后显示出1~n的所有偶数平方。例如，如果用户输入100，那么程序员应该显示出下列内容：

```

4
16
36
64
100

```

7. 重新安排程序square3.c以便for循环对变量i进行初始化，对变量i进行判定，并且对变量i进行自增操作。不需要重写程序，特别是不要使用任何乘法。
8. 编写程序，要求显示出单月的日历。用户说明这个月的天数和本月起始日是星期几：

```

Enter number of days in month: 31
Enter starting day of the week (1=Sun, 7=Sat): 3

```

```

1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

```

提示：此程序不像看上去那么难。最重要的内容是for语句使用变量i从1计数到n，n是此月的天数，显示出i的每个值。在循环中，用if语句判定i是否是一个星期的最后一天，如果是，就显示一个换行符。

- *9. 下面这条for语句的输出是什么？

```

for (i = 5, j = i - 1; i > 0, j > 0; --i, j = i - 1)
    printf("%d ", i);

```

10. 下列哪条语句和其他两条语句不等价（假设循环体都是一样的）？

(a) for (i = 0; i < 10; i++)...
(b) for (i = 0; i < 10; ++i)...
(c) for (i = 0; i++ < 10;)...

11. 下列哪条语句和其他两条语句不等价（假设循环体都是一样的）？

(a) while (i < 10) {...}
(b) for (; i < 10;) {...}
(c) do {...} while (i < 10);

6.4节

12. 显示如何用等价的goto语句替换continue语句。

13. 下面的程序段产生的输出是什么？

```

sum = 0;
for (i = 0; i < 10; i++) {
    if (i % 2) continue;
    sum += i;
}
printf("%d\n", sum);

```

14. 下面的“素数判定”循环作为示例出现在6.4节中：

```

for (d = 2; d < n; d++)
    if (n % d == 0) break;

```

这个循环不是很有效。用n除以2~n-1所有数的方法来判断它是否为素数是没有必要的。事实上，只需要检查不大于n的平方根的除数。利用这一点来修改循环。提示：不要试图计算出n的平方根，而是用d*d和n进行比较。

6.5节

- *15. 重写下面的循环，从而使其循环体为空。

```

for (n = 0; m > 0; n++)
    m /= 2;

```

- *16. 找出下面程序段的错误并且修改它。

```

if (n % 2 == 0);
    printf("n is even\n");

```

107

108

基本类型

请别搞错：计算机处理的是数而不是符号。我们用对活动算术化的程度来衡量我们的理解力（和控制力）。

到目前为止，本书只使用了C语言的两种基本（内置的）类型：`int`和`float`。本章讲述其余的基本类型，并且在这个过程中提供关于`int`类型和`float`类型的附加信息。7.1节展示了整型的取值范围，包括长整型、短整型和无符号整型。7.2节介绍了`double`类型和`long double`类型，这些类型提供了更大的取值范围，以及比`float`类型更高的精度。7.3节涵盖了`char`（字符）类型，这种类型将用于字符数据的处理。7.4节描述了`sizeof`运算符，这种运算符用来计算一种类型需要的存储空间大小。7.5节解决了重要的类型转换问题，即把一种类型的值转换成另外一种类型的等价值。最后，7.6节展示了利用`typedef`定义新类型名的方法。

7.1 整型

C语言支持两种根本不同的数值类型：整型和浮点型。整型的值全都是数，而浮点型的值则可能还有小数部分。整型又分为两大类：有符号的和无符号的。

有符号整数和无符号整数

整数通常以16位或32位方式存储。在有符号数中，如果数为正数或零，那么最左边的位（符号位）为0，如果是负数，符号位则为1。因此，最大的16位整数的二进制表示形式是0111111111111111，对应的值是32 767（即 $2^{15} - 1$ ）。而最大的32位整数是01111111111111111111111111，对应的数值是2 147 483 647（即 $2^{31} - 1$ ）。把不带符号位（把最左边的位看成是数值部分）的整数称为无符号数。最大的16位无符号整数是65 535（即 $2^{16} - 1$ ），而最大的32位无符号整数是4 294 967 295（即 $2^{32} - 1$ ）。

默认情况下，C语言中的整型变量都是有符号的，也就是说最左位保留为符号位。为了告诉编译器变量没有符号位，需要把它声明成`unsigned`类型。无符号数主要用于系统编程和低级别的、与机器相关的应用。第20章将讨论无符号数的典型应用，在此之前，我们通常回避无符号数。

C语言的整型有不同的尺寸。`int`类型是计算机给出的整数的“正常尺寸”（通常为16位或32位）。由于16位整数的上限值为32 767，这会对许多应用产生限制，所以C语言还提供了长整型。某些时候，为了节省空间，我们会指示编译器存储比正常尺寸小的数，称这样的数为短整型。

为了构造的整型正好满足需要，可以指明变量是`long`型或`short`型，`singed`型或`unsigned`型。甚至可以把说明符组合起来（如`long unsigned int`）。然而，实际上只有下列6种组合可以产生不同的类型：

`short int`

```
unsigned short int
int
unsigned int
long int
unsigned long int
```

其他组合都是上述6种类型其中一种的同义词。(例如,除非额外说明,否则所有整数都是有符号的。因此, long signed int和long int是一样的类型。)另外,说明符的顺序没有要求,所以unsigned short int和short unsigned int是一样的。

C语言允许通过省略单词int来缩写整型的名字。例如, unsigned short int可以缩写为unsigned short,而long int则可以缩写为long。

6种整型的每一种所表示的取值范围都会根据机器的不同而不同。但是有两条所有编译器都必须遵守的原则。首先,C标准要求short int、int和long int中的每一种类型都要覆盖一个确定的最小取值范围。其次,标准要求int类型不能比short int类型短,而且long int类型不能比int类型短。但是,short int类型的取值范围有可能和int类型的范围是一样的;int类型的取值范围也可以和long int的一样。

110

表7-1说明了在16位机上整型通常的取值范围,注意short int和int有相同的取值范围。

表7-1 16位机的整型

| 类 型 | 最 小 值 | 最 大 值 |
|--------------------|----------------|---------------|
| short int | -32 768 | 32 767 |
| unsigned short int | 0 | 65 535 |
| int | -32 768 | 32 767 |
| unsigned int | 0 | 65 535 |
| long int | -2 147 483 648 | 2 147 483 647 |
| unsigned long int | 0 | 4 294 967 295 |

表7-2说明了32位机上整型的通常取值范围。这里的int和long int有着相同的取值范围。(见23.2节)在<limits.h>中可以找到定义了每种整型最大值和最小值的宏,其中<limits.h>属于标准库。

表7-2 32位机的整型

| 类 型 | 最 小 值 | 最 大 值 |
|--------------------|----------------|---------------|
| short int | -32 768 | 32 767 |
| unsigned short int | 0 | 65 535 |
| int | -2 147 483 648 | 2 147 483 647 |
| unsigned int | 0 | 4 294 967 295 |
| long int | -2 147 483 648 | 2 147 483 647 |
| unsigned long int | 0 | 4 294 967 295 |

注意,short和long整型在16位机和32位机上都有着各自相同的取值范围。这个发现也引出第一个可移植性技巧。

可移植性技巧 为了最大限度保证可移植性,对不超过32 767的整数采用int(或short int)类型,而对其他的整数采用long int类型。

然而,不要不分差别地使用长整型,因为在长整型上的操作需要的时间可能会多过在较小整数上的。

7.1.1 整型常量

现在把注意力转向常量,它是在程序中以文本形式显示的数,而不是读、写或计算出来的

数。C语言允许用十进制（基数为10）、八进制（基数为8）和十六进制（基数为16）形式书写整型常量。

八进制数和十六进制数

八进制数是用0~7的数字编写的。八进制数的每一位表示一个8次幂（这就如同十进制数的每一位表示的是10次幂一样）。因此，八进制的数237表示成十进制数就是 $2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0 = 128 + 24 + 7 = 159$ 。

十六进制数是用0~9的数字加上A~F的字母编写的，其中字母A~F分别表示了10~15的数。十六进制数的每一位表示一个16的幂；十六进制数1AF的十进制数值是 $1 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 256 + 160 + 15 = 431$ 。

- **十进制常量**包含数字0~9，但是一定不能以零开头：

15 255 32767

- **八进制常量**只包含数字0~7，而且必须要以零开头：

017 0377 077777

- **十六进制常量**包含数字0~9和字母a~f，而且总是以0x开头：

0xf 0xff 0x7fff

十六进制常量中的字母既可以是大写字母也可以是小写字母：

0xff 0xFF 0xFF 0xFF 0Xff 0XFf 0Xff 0XFf

请记住八进制和十六进制只是数书写的另一种方式；它们不会对数实际存储的方式产生影响。（整数都是以二进制形式存储的，而不考虑实际书写的方式。）任何时候都可以从一种书写方式切换到另一种书写方式，甚至是混合使用：10 + 015 + 0x20的值为55（十进制）。八进制和十六进制更适合应用在低级程序的编写上，到第20章会较多地用到它们。

当程序中出现整型常量时，如果它属于int类型的取值范围，那么编译器会把此常量作为普通整数来处理，否则作为长整型数来处理。为了迫使编译器把常量作为长整型数来处理，只需在后边加上一个字母L（或l）：

15L 0377L 0x7ffffL

为了指明是无符号常量，可以在常量后边加上字母U（或u）：

15U 0377U 0x7ffffU

为了表示常量是长且无符号的可以组合使用字母L和U：0xffffffffUL（字母L和U的顺序和大小写没有关系）。

7.1.2 读/写整数

112 假设有一个程序无法工作，**Q&A**原因是它的其中一个int变量发生“溢出”，也就是程序赋给变量的值太大以致于无法存储在int类型中。第一个想法是改变变量的类型，从int型变为long int型。但是，仅仅这样做是不够的，我们必须检查数据类型的改变对程序其他部分的影响。尤其是需要检查变量是否用在printf函数或scanf函数的调用中。如果用到了，那么将需要改变调用中的格式串，因为%d的转换只针对int型数值。

读和写无符号、短的和长的整数需要一些新的转换说明符。

- 当读或写无符号整数时，**Q&A**使用字母u、o或x代替转换说明中的d。如果使用了u说明符，那么读（或写）的数是十进制形式；o指明是八进制形式，而x指明十六进制形式。

```

unsigned int u;
scanf("%u", &u); /* reads u in base 10 */
printf("%u", u); /* writes u in base 10 */
scanf("%o", &u); /* reads u in base 8 */
printf("%o", u); /* writes u in base 8 */
scanf("%x", &u); /* reads u in base 16 */
printf("%x", u); /* writes u in base 16 */

```

- 当读或写短整型数时，在d、o、u或x前面加上字母h:

```

short int s;
scanf("%hd", &s);
printf("%hd", s);

```

- 当读或写长整型数时，在d、o、u或x前面加上字母l:

```

long int l;
scanf("%ld", &l);
printf("%ld", l);

```

7.1.3 程序：数列求和（改进版）

在6.1节编写了一个程序对一个用户输入的整数数列求和。这个程序的一个问题就是所求出的和（或其中某个输入数）可能会超出int型变量允许的最大值。如果程序运行在用16位长度表示整数的机器上，可能会发生这类情况：

```

This program sums a series of integers.
Enter integers (0 to terminate): 10000 20000 30000 0
The sum is: -5536

```

这个和的结果应该为60 000，但这个值不在int型变量表示的范围内，所以我们得到了一个毫无意义的结果。为了改进这个程序，可以把变量改换成long int型。

113

```

sum2.c
/* Sums a series of numbers (using long int variables) */

#include <stdio.h>

main()
{
    long int n, sum = 0;

    printf("This program sums a series of integers.\n");
    printf("Enter integers (0 to terminate): ");

    scanf("%ld", &n);
    while (n != 0) {
        sum += n;
        scanf("%ld", &n);
    }
    printf("The sum is: %ld\n", sum);

    return 0;
}

```

这种改变比较简单：将声明n和sum为int型变量替换成long int型变量。然后改变scanf函数和printf函数中的转换说明，用%ld代替%d。

7.2 浮点型

整型并不适用于所有应用。有些时候需要变量能存储带小数点的数，或者能存储极大数或



极小数。这类数可以用浮点（因小数点是“浮动的”而得名）格式进行存储。C语言提供3种浮点型，它们对应不同的浮点格式。

- float：单精度浮点数。
- double：双精度浮点数。
- long double：扩展双精度浮点数。

具体采用哪种类型依赖于程序对精度（和数量级）的要求。当精度要求不严格时，float型是很适合的类型（例如，计算带一个小数点的温度）。double提供更高的精度，足够适用于绝大多数程序。long double支持极高精度的要求，很少会用到。

C标准没有说明float、double和long double类型提供的精度到底是多少，因为不同的计算机可以用不同方法存储浮点数。大多数现代计算机和工作站都遵循IEEE 754标准的规范，所以这里也用它做一个示例。

114

IEEE浮点标准

由IEEE开发的IEEE标准提供了两种主要的浮点数格式：单精度（32位）和双精度（64位）。数值以科学计数法的形式存储，每一个数都是由3部分组成：符号、指数和小数。为指数部分保留的位数说明了数可能大（或小）的程度，而小数部分的位数说明了精度。单精度格式中，指数长度为8位，而小数部分占了23位。结果是，单精度数可以表示的最大值大约是 3.40×10^{38} ，其中精度是6个十进制数字。

IEEE标准也描述另外两种格式，单扩展精度和双扩展精度。标准没有说明这些格式中的位数，但是它要求单扩展精度类型至少为43位，而双扩展精度类型至少要为79位。为了获得更多有关IEEE标准和浮点算术的信息，可以参阅David Goldberg的“What every computer scientist should know about floating-point arithmetic”(ACM Computing Surveys, vol. 23, no. 1:5-48)。

表7-3显示了根据IEEE标准实现时浮点型的特征。long double类型没有显示在此表中，因为它的长度随着机器的不同而变化，而最常通用的尺寸是80位和128位。在不遵循IEEE标准的计算机上，表7-3是无效的。事实上，在一些机器上，float可以有和double相同的数值集合，或者double可以有和long double相同的数值。可以在<float.h>(>23.1节)中找到定义浮点型特征的宏。

表7-3 浮点型的特征 (IEEE标准)

| 类 型 | 最小正值 | 最大值 | 精 度 |
|--------|-------------------------|------------------------|-------|
| float | 1.17×10^{-38} | 3.40×10^{38} | 6个数字 |
| double | 2.22×10^{-308} | 1.79×10^{308} | 15个数字 |

7.2.1 浮点常量

浮点常量可以有许多种书写方式。例如，下面这些常量全都是表示数57.0的有效方式：

57.0 57. 57.0e0 57E0 5.7e1 5.7e+1 .57e2 570.e-1

浮点常量必须包含小数点或指数；其中，指数指明衡10的幂。如果表示指数，需要在指数数值前放置字母E（或e）。可选项+或-可以出现在字母E（或e）的后边。

默认情况下，浮点常量都以双精度数的形式存储。**Q&A**换句话说，当C语言编译器在程序中发现常量57.0时，它会安排数据以double型变量的格式存储在内存中。通常这条规则不会引发任何问题，因为在需要时double类型的值可以自动转化为float类型值。

115

在某些极个别的的情况下，可能会需要强制编译器以float或long double格式存储浮点常量。为了表明只需要单精度，可以在常量的末尾处加上字母F（或f）（如57.0F）。而为了说明常量必须以long double格式存储，要在常量的末尾处加上字母L（或l）（如57.0L）。

7.2.2 读/写浮点数

正如已经讨论过的一样，转换说明%e、%f和%g用于读和写单精度浮点数，而double和long double类型值则要求略微不同的转换。

- 当读取double类型的数值时，在e、f或g前放置字母l：

```
double d;
scanf("%lf", &d);
```

注意：只能在scanf函数格式串中使用l，不能在printf函数格式串中使用。**Q&A**在printf函数格式串中，转换e、f和g可以用来写float型或double型值。

- 当读或写long double类型的值时，在e、f或g前放置字母L：

```
long double ld;
scanf("%Lf", &ld);
printf("%Lf", ld);
```

7.3 字符型

除了整型和浮点型外，char是唯一还没有讨论的基本类型，即字符型。**Q&A**char类型的值可以根据计算机的不同而不同，因为不同的机器可能会有不同的字符集。

字符集

当今最常用的字符集是ASCII（美国信息交换标准码）（>附录E），它是用7位代码表示128个字符。在ASCII码中，数字0~9用0110000~01111001码来表示，同时大写字母A~Z则用10000001~1011010码表示。一些计算机把ASCII码扩展为8位代码以便可以表示256个字符。

其他一些计算机使用完全不同的字符集。例如，IBM主机依赖一种早期的EBCDIC代码。**116**未来的机器可能会使用Unicode代码（>25.2.2节），这是一种用16位代码表示65 536个字符的编码集合。

用计算机能表示的任意字符给char类型的变量赋值：

```
char ch;
ch = 'a'; /* lower-case a */
ch = 'A'; /* upper-case A */
ch = '0'; /* zero */
ch = ' '; /* space */
```

注意，字符常量需要用单引号括起来，而不是双引号。

在C语言中字符的操作非常简单，因为存在这样一个事实：C语言会按小整数的方式处理字符。毕竟所有字符都是以二进制的形式进行编码的，而且无需花费太多的想象就可以将这些二进制代码看成是整数。例如，在ASCII码中，字符的取值范围是0000000~1111111，这个范围可以看成是0~127的整数。字符'a'的值为97，'A'的值为65，'0'的值为48，而' '的值为32。

当计算中出现字符时，C语言只是使用它对应的整数值。思考下面这个例子，假设采用ASCII码字符集：

```

char ch;
int i;

i = 'a';           /* i is now 97 */
ch = 65;          /* ch is now 'A' */
ch = ch + 1;      /* ch is now 'B' */
ch++;             /* ch is now 'C' */

```

可以像对数那样对字符进行比较。下面的if语句测试ch是否含有小写字母；如果有，那么它会把ch转化为相应的大写字母。

```

if ('a' <= ch && ch <= 'z')
    ch = ch - 'a' + 'A';

```

诸如'a'<= ch这样的比较使用的是字符所对应的整数值，这些数值依据使用的字符集有所不同，所以程序使用<、<=、>和>=来进行字符比较可能不易移植。

字符拥有和数相同的属性，这一事实会带来一些好处。例如，for语句中的控制变量可以简单采用大写字母：

```

for (ch = 'A'; ch <= 'Z'; ch++)...

```

另一方面，以数的方式处理字符可能会导致编译器无法检查出来的多种编程错误，还可能会导致我们编写出诸如'a' * 'b' / 'c'这类无意义的表达式。此外，这样做也可能会妨碍程序的可移植性，因为程序可能会基于一些对字符集的假设。（例如，上述的for循环是假设从字母A到字母Z是连续的代码。）

既然C语言允许把字符作为整数来使用，那么像整型一样，char类型也存在有符号型和无符号型两种，通常有符号型字符的取值范围是-128~127，而无符号型字符的取值范围则是0~255。

C语言标准没有说明普通char型数据是有符号型还是无符号型；一些编译器把它们按照有符号型数据来处理，而另外一些编译器则将它们处理成无符号型数据。（甚至还有一些编译器允许程序员通过编译器选项来选择char类型是有符号型还是无符号型。）

大多数时候，人们并不真的关心char型是有符号型还是无符号型。但是，我们偶尔确实需要注意，特别是当使用字符型变量存储一个数值的时候。**Q&A** 基于上述原因，标准C允许使用单词signed和unsigned来修饰char类型：

```

signed char sch;
unsigned char uch;

```

可移植性技巧 不要假设char类型默认为signed或unsigned。如果需要注意，要用signed char或unsigned char代替char。

由于字符和整数之间有密切关系，本书将采用术语整值类型来（统称）包含整型和字符型。

7.3.1 转义序列

正如在前面示例中见到的那样，字符常量通常是用单引号括起来的字符。然而，一些特殊符号是无法采用上述这种书写方式的，比如换行符，因为它们是不可见的（无法打印的），或者是无法从键盘输入的。因此，为了使程序可以处理字符集中的每一个字符，C语言提供了一种特殊的符号——转义序列（escape sequence）。

转义序列共有两种：字符转义序列（character escape）和数字转义序列（numeric escape）。在3.1节已经见过字符转义序列的部分列表，下面的表7-4给出了一个完整的字符转义序列集。转义序列\ a、\ b、\ f、\ r、\ t 和\ v 表示了通用的ASCII码控制字符，**Q&A** 转义序列\ n 表示了ASCII码的回行符，转义序列\\允许字符常量或字符串包含字符\，转义序列\'允许字符常量包含字符'，而转义序列\"则允许字符串包含字符"，**Q&A** 转义序列\? 使用极少。

表7-4 字符转义序列

| 名 称 | 转义序列 | 名 称 | 转义序列 |
|---------|------|-------|------|
| 警报(响铃)符 | \a | 纵向制表符 | \v |
| 回退符 | \b | 反斜杠 | \＼ |
| 换页符 | \f | 问号 | \? |
| 换行符 | \n | 单引号 | \' |
| 回车符 | \r | 双引号 | \" |
| 横向制表符 | \t | | |

字符转义序列使用起来很容易，但是它们有一个问题：转义序列列表没有包含所有无法打印的ASCII字符，只包含了最常用的字符。字符转义序列也无法用于表示基本的128个ASCII码字符以外的字符。(一些计算机提供的是扩展的ASCII码字符集，比如IBM个人计算机系列就是一个明显的例子。)数字转义序列可以表示任何字符，所以它可以解决上述这个问题。

为了把特殊字符书写成数字转义序列，首先需要在类似附录E那样的表中查找字符的八进制或十六进制值。例如，某个ASCII码转义字符(十进制值为27)对应的八进制值为33，对应的十六进制值则为1B。上述这些八进制和十六进制的代码可以用来书写转义序列：

- 八进制转义序列由字符\和跟随着其后的一个最多含有三位数字的八进制数组成。(此数必须表示为无符号字符型，所以最大值通常是八进制的377。)例如，可以将转义字符写成\33或\033。转义序列中的八进制数不一定要用0开头，这一点不像通常表示的八进制数。
- 十六进制转义序列由\x和跟随着其后的一个十六进制数组成。虽然标准C对于十六进制数中的数字个数没有限制，但是必须可以将数表示成无符号型字符(因此，如果字符是八位长度，那么十六进制数中的数不能超过FF)。若采用这种符号，那么可以把转义字符写成\x1b或\x1B的形式。字符x必须小写，但是十六进制的数字(例如b)不限大小写。

作为字符常量使用时，转义序列必须用一对单引号括起来。例如，一个表示成转义字符的常量可以写成'\33'(或'\x1b')的形式。转义序列可能有点隐晦，所以采用#define的方式给它们命名通常会是个不错的主意：

```
#define ESC '\33' /* ASCII escape character */
```

正如3.1节看到的那样，转移序列也可以嵌入在字符串中使用。

转义序列不是用于表示字符的唯一一种特殊符号。20世纪80年代，为了使C语言成为更加国际化的编程语言，C语言加入了一些其他表示字符的方式。三字符序列(trigraph sequence)(►25.3节)是一些特殊的ASCII字符的代码，这些字符在美国以外的某些计算机上的是不可用的。多字节字符(multibyte character)(►25.2节)和宽字符(wide character)(►25.2节)用于某些字符集，这些字符集的编码都太大以致于无法存储在一个字节内。

118

119

7.3.2 字符处理函数

本节的前面已经讲过如何使用if语句把小写字母转换成大写字母：

```
if ('a' <= ch && ch <= 'z')
    ch = ch - 'a' + 'A';
```

但是这种方法还不是最好的。有一种更快捷、更易于移植的方法是调用C语言的toupper库函数：

```
ch = toupper(ch); /* converts ch to upper case */
```

被调用时，toupper函数检测自身的参数(在本例中参数为ch)是否是小写字母。如果是，那么它会把参数转换成相应的大写字母；否则，toupper函数会返回参数的值。上面的例子采用

赋值运算符把`toupper`函数返回的值存储在变量`ch`中。当然也可以同样简单地进行其他的处理，如存储到其他变量中，或用`if`语句进行测试：

```
if (toupper(ch) == 'A') ...
```

程序调用`toupper`函数的程序需要在顶部放置下面这条`#include`指令：

```
#include <ctype.h>
```

在C函数库中，`toupper`函数不是唯一实用的字符处理函数。23.4节描述了全部字符处理函数，并且给出了使用它们的示例。

7.3.3 读/写字符

转换说明`%c`允许`scanf`函数和`printf`函数对单独一个字符进行读/写操作：

```
char ch;
scanf("%c", &ch);      /* reads a single character */
printf("%c", ch);      /* writes a single character */
```

在读入字符前，`scanf`函数不会跳过空白字符。如果下一个未读字符是空格，那么前面例子中，`scanf`函数返回后变量`ch`将包含一个空格。为了强制`scanf`函数在读入字符前跳过空白字符，需要在格式串转换说明`%c`前面加上一个空格：

```
scanf(" %c", &ch);      /* skips white space, then reads ch */
```

回顾3.2节介绍的内容，`scanf`函数格式串中的空白意味着“跳过零个或多个空白字符”。

120 既然通常情况下`scanf`函数不会跳过空白，所以它很容易检查到输入行的结尾：检查刚读入的字符是否为换行符。例如，下面的循环将读入并且忽略掉所有当前输入行中其余的字符：

```
do {
    scanf("%c", &ch);
} while (ch != '\n');
```

当下次调用`scanf`函数时，将读入下一输入行中的第一个字符。

C语言还提供了读/写单独一个字符的其他方法。特别是，可以使用`getchar`函数和`putchar`函数来代替调用`scanf`函数和`printf`函数。**Q&A**每次调用`getchar`函数时，它会读入一个字符，并返回这个字符。为了保存`getchar`函数返回的字符，需要使用赋值操作将返回值存储在变量中：

```
ch = getchar();      /* reads a character and stores it in ch */
```

和`scanf`函数一样，`getchar`函数也不会在读取时跳过空白字符。`putchar`函数用来写单独的一个字符：

```
putchar(ch);
```

当执行程序时，使用`getchar`函数和`putchar`函数（胜于`scanf`函数和`printf`函数）可以节约时间。`getchar`函数和`putchar`函数执行速度快有两个原因。第一个原因是，这两个函数比`scanf`函数和`printf`函数简单，因为`scanf`函数和`printf`函数是设计用来读/写多种不同格式类型数据的。第二个原因是，为了额外的速度提升，通常`getchar`函数和`putchar`函数是作为宏（>14.3节）来实现的。

`getchar`函数还有一个优于`scanf`函数的地方：因为返回的是读入的字符，所以`getchar`函数可以应用在多种不同的C语言惯用法中，包括用循环搜索字符或跳过所有出现的同一字符。思考下面这个`scanf`函数循环，它用来跳过输入行的剩余部分：

```
do {
    scanf("%c", &ch);
} while (ch != '\n');
```

用`getchar`函数重写上述循环，内容如下所示：

```
do {
    ch = getchar();
} while (ch != '\n');
```

为了精简循环，允许把getchar函数的调用放入到循环的控制表达式中，形式如下：

```
while ((ch = getchar()) != '\n')
```

这个循环读入一个字符，把它存储在变量ch中，然后测试变量ch是否不是换行符。如果测试结果为真，那么执行循环体(循环体实际为空)，接着再次测试循环条件，从而引发读入新的字符。[121]
实际上不是真的需要变量ch；可以只把getchar函数的返回值与换行符进行比较：

[惯用法] `while (getchar() != '\n') /* skips rest of line */ ;`

这个循环是非常著名的C语言惯用法，虽然这种用法的含义是十分隐晦的，但是值得学习。

getchar函数在用于循环中搜寻字符时和跳过字符一样有效。思考下面的语句，利用getchar函数跳过无限数量的空格字符：

[惯用法] `while ((ch = getchar()) == ' ') /* skips blanks */ ;`

当循环终止时，变量ch将包含getchar函数遇到的第一个非空字符。



如果在同一个程序中混合使用getchar函数和scanf函数，请一定注意scanf函数有一种留下后边字符的趋势，也就是说对于输入后面的字符（包括换行符）只是“看了一下”，并没有读入。思考一下，如果试图先读入数再读入字符的话，下面的程序段会发生什么：

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

在读入i的同时，scanf函数调用将会留下后面没有消耗掉的任意字符，包括换行符（但不仅限于换行符）。getchar函数随后将取回第一个剩余字符，但这不是我们所希望的结果。

7.3.4 程序：确定消息的长度

为了说明字符的读取方式，下面编写一个程序来计算消息的长度。在用户输入消息后，程序显示的长度如下：

```
Enter a message: Brevity is the soul of wit.
Your message was 27 character(s) long.
```

消息的长度包括空格和标点符号，但是不包含消息结尾处的换行符。

程序需要采用循环结构来实现读入字符和计数器自增操作，循环在遇到换行符时立刻终止。我们可以采用scanf函数也可以采用getchar函数读取字符，但大多数C程序员愿意采用getchar函数。采用一个简明的while循环书写的程序如下：[122]

```
length.c
/* Determines the length of a message */

#include <stdio.h>

main()
{
    char ch;
```

```

int len = 0;
printf("Enter a message: ");
ch = getchar();
while (ch != '\n') {
    len++;
    ch = getchar();
}
printf("Your message was %d character(s) long.\n", len);

return 0;
}

```

重新回顾有关while循环和getchar函数惯用法的讨论，我们发现程序可以缩短成如下形式：

```

length2.c
/* Determines the length of a message */

#include <stdio.h>

main()
{
    int len = 0;

    printf("Enter a message: ");
    while (getchar() != '\n')
        len++;
    printf("Your message was %d character(s) long.\n", len);

    return 0;
}

```

7.4 sizeof 运算符

运算符sizeof允许程序确定用来存储指定类型值所需空间的大小。

[sizeof表达式] sizeof(类型名)

上述表达式的值是无符号整数，这个整数表示用来存储属于类型名的值所需要的字节数。表达式sizeof(char)的值始终为1，但是对其他类型计算出的值可能会有所不同。在16位机上，表达式sizeof(int)的值通常为2；在大多数32位机上，表达式sizeof(int)的值为4。注意，因为编译器本身就可以计算sizeof表达式的值，所以运算符sizeof是一种特殊的运算符。

通常情况下，运算符sizeof也可以应用于常量、变量和表达式。如果i和j是整型变量，那么sizeof(i)在16位机上的值为2，这和表达式sizeof(i+j)的值一样。与应用于类型时相反，当应用于表达式时sizeof不要求圆括号；可以用sizeof i代替sizeof(i)。但是，由于运算符优先级的问题，圆括号可能还是会需要的。编译器会把表达式sizeof i + j解释为(sizeof i) + j，这是因为sizeof作为一元运算符的优先级高于二元运算符+。为了避免出现此类问题，所以建议始终在sizeof表达式中采用圆括号。

显示sizeof的值时要注意，因为sizeof表达式的类型是由实现定义的。窍门是在显示前把表达式的值转换成一种已知的类型。既然sizeof返回无符号的整型，所以最安全的方法是把sizeof表达式转换成unsigned long型（最大的无符号类型），然后用转换说明%lu进行。把表达式转换成不同类型要求“强制类型转换”，这种技术将在7.5节描述。下面是采用强制方法显示int类型大小的方式：

```
printf("Size of int: %lu\n", (unsigned long)sizeof(int));
```

符号(unsigned long)告诉编译器将随后的表达式（本例中是sizeof (int)）的值转换成无符号的长整型数据。

7.5 类型转换

在执行算术运算时，计算机比C语言的限制更多。为了让计算机执行算术运算，通常要求操作数有相同的大小（即位的数量相同），并且要求存储的方式也相同。计算机可能可以将两个16位整数相加，但是不能直接将16位整数和32位整数相加，也不能直接将32位整数和32位浮点数相加。

另一方面，C语言允许在表达式中混合使用基本数据类型。在单独一个表达式中可以组合整数、浮点数，甚至是字符。当然，在这种情况下C语言编译器可能需要生成一些指令将某些操作数转换成不同类型，使得硬件可以对表达式进行计算。例如，如果对16位int型数和32位long int型数进行加法操作，那么编译器将安排把16位int型值转换成32位值。如果是int型数据和float型数据进行加法操作，那么编译器将安排把int型值转换成为float格式。这个转换过程稍微复杂一些，因为int型值和float型值的存储方式不同。

因为编译器可以自动处理这些转换而无需程序员介入，所以这类转换称为隐式转换（*implicit conversion*）。C语言还允许程序员通过使用强制运算符执行显式转换（*explicit conversion*）。首先讨论隐式转换，而显式转换将会推迟到本节的后半部分进行介绍。遗憾的是，执行隐式转换的规则有些复杂，主要是因为C语言有大量不同的基本数据类型（6种整型和3种浮点型，这还不包括字符型）。

当发生下列情况时会进行隐式转换：

- 当算术表达式或逻辑表达式中操作数的类型不相同时。（C语言执行所谓的常用算术转换。）
- 当赋值运算符右侧表达式的类型和左侧变量的类型不匹配时。
- 当函数调用中使用的参数类型与其对应的参数的类型不匹配时。
- 当return语句中表达式的类型和函数返回值的类型不匹配时。

这里将讨论前两种情况，而其他情况将留到后边的第9章进行介绍。

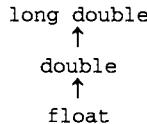
7.5.1 常用算术转换

常用算术转换多用于二元运算符的操作数上，包括算术运算符、关系运算符和判等运算符。例如，假设变量x的类型为float型，而变量i为int类型。常用算术转换将会应用在表达式x+i的操作数上，因为两者的类型不同。显然把变量i转换成float型（匹配变量x的类型）比把变量x转换成int型（匹配变量i的类型）更安全。整数始终可以转换成为float类型；可能会发生的最糟糕的事是精度会有少量损失。相反，把浮点数转换成为int类型，将有小数部分的损失；更糟糕的是，如果原始数大于最大可能的整数或者小于最小的整数，那么将会得到一个完全没有意义的结果。

常用算术转换的策略是把操作数转换成可以安全的适用于两个数值的“最狭小的”数据类型。（粗略的说，如果某种类型要求的存储字节比另一种类型少，那么这种类型就比另一种类型更狭小。）为了统一操作数的类型，通常可以将相对较狭小类型的操作数转换成另一个操作数的类型来实现（这就是所谓的提升）。Q&A最常用的提升是整型提升（integral promotion），它把字符或短整数转换成int类型（或者某些情况下是unsigned int类型）。

执行常用算术转换的规则时可以划分成两种情况。

- 任一操作数的类型是浮点型的情况。按照下图将类型较狭小的操作数进行提升：

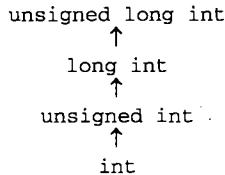


124

125

也就是说，如果一个操作数的类型为long double，那么把另一个操作数的类型转换成long double类型。否则，如果一个操作数的类型为double类型，那么把另一个操作数转化成double类型；如果一个操作数的类型是float类型，那么把另一个操作数转换成float类型。注意，这些规则涵盖了混合整数和浮点数类型的情况。例如，如果一个操作数的类型是long int类型，并且另一个操作数的类型是double类型，那么把long int类型的操作数转换成double类型。

- **两个操作数的类型都不是浮点型的情况。**首先对两个操作数进行整型提升（保证没有一个操作数是字符型或短整型）。然后按照下图对操作数的类型进行提升：



有一种特殊情况，只有在long int类型和unsigned int类型长度相同（比如32位）时才会发生。在这类情况下，如果一个操作数的类型是long int，而另一个的类型是unsigned int，那么两个操作数都会转换成unsigned long int类型。



当把有符号操作数和无符号操作数整合时，会通过把符号位看成数的位的方法把有符号操作数“转换”成无符号的值。这条规则可能会导致某些隐蔽的编程错误。

假设int型的变量i的值为-10，而且unsigned int型的变量u的值为10。如果用<运算符比较变量i和变量u，那么期望的结果应该是1（真）。但是，在比较前，变量i转换成为unsigned int类型。因为负数不能被表示成无符号整数，所以转换后的数值将不再为-10，而是一个大的正数（将变量i中的位看作是无符号数）。因此i < u比较的结果将为0。

由于此类陷阱的存在，所以最好尽量避免使用无符号整数，特别是不要把它和有符号整数混合使用。

下面的例子显示了常用算术转换的实际执行情况：

```

126
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

i = i + c;          /* c is converted to int           */
i = i + s;          /* s is converted to int           */
u = u + i;          /* i is converted to unsigned int */
l = l + u;          /* u is converted to long int     */
ul = ul + l;         /* l is converted to unsigned long int */
f = f + ul;          /* ul is converted to float       */
d = d + f;          /* f is converted to double        */
ld = ld + d;         /* d is converted to long double   */
  
```

7.5.2 赋值中的转换

常用算术转换不适用于赋值运算。C语言会遵循另一条简单的转换规则，那就是把赋值运算右边的表达式转换成左边变量的类型。如果变量的类型至少和表达式类型一样“宽”，那么这

种转换将没有任何障碍。例如：

```
char c;
int i;
float f;
double d;

i = c;      /* c is converted to int */
f = i;      /* i is converted to float */
d = f;      /* f is converted to double */
```

其他情况下是有问题的。把浮点数赋值给整型变量会去掉该数的小数部分：

```
int i;
i = 842.97;      /* i is now 842 */
i = -842.97;     /* i is now -842 */
```

而且，**Q&A**如果取值在变量类型范围之外，那么把值赋给一个较狭小类型的变量将会得到无意义的结果（甚至更糟）。

```
c = 10000;      /*** WRONG ***/
i = 1.0e20;      /*** WRONG ***/
f = 1.0e100;     /*** WRONG **/
```

这类赋值可能会导致编译器或lint发出的警告。

7.5.3 强制类型转换

虽然C语言的隐式转换使用起来非常方便，但是有些时候会需要更大程度的控制类型转换。基于这种原因，C语言提供了强制类型转换。强制类型转换表达式的格式如下：

127

[强制转化表达式] (类型名) 表达式

这里的类型名表示的是表达式应该转换成的类型。

下面的例子显示了使用强制类型转换表达式计算float型值小数部分的方法：

```
float f, frac_part;
frac_part = f - (int) f;
```

强制类型转换表达式 (int) f 表示把 f 的值转换成 int 类型后的结果。C语言的常用算术转换则要求在进行减法运算前把 (int) f 转换回 float 类型。f 和 (int) f 的不同就是在 f 的小数部分，这部分在强制类型转换时被忽略掉了。

强制类型转换表达式可以被用来显示那些肯定会发生类型的转换：

```
i = (int) f; /* f is converted to int */
```

它也可以用来控制编译器并且强制它进行我们需要的转换。思考下面的例子：

```
float quotient;
int dividend, divisor;
quotient = dividend / divisor;
```

正如现在写的那样，除法的结果是一个整数，在把结果存储在 quotient 变量中之前，将要把结果转换成 float 格式。但是，为了得到更精确的结果，可能需要在除法执行之前把 dividend 和 divisor 的类型转换成 float 格式的。强制类型转换表达式可以完成这样的小技巧：

```
quotient = (float) dividend / divisor;
```

变量 divisor 不需要进行强制类型转换，因为把变量 dividend 强制类型转换成 float 类型会迫使编译器把 divisor 也转换成 float 类型。

顺便提一下，C语言把(类型名)视为一元运算符。一元运算符的优先级高于二元运算符，

所以编译器会把表达式

```
(float) dividend / divisor
```

解释为

```
((float) dividend) / divisor
```

如果感觉有点混淆，那么注意还有其他方法可以实现同样的效果：

128

```
quotient = dividend / (float) divisor;
```

或者

```
quotient = (float) dividend / (float) divisor;
```

有些时候，需要使用强制类型转换来避免溢出。思考下面这个例子：

```
long int i;
int j = 1000;
i = j * j; /* *** WRONG ***/
```

乍看之下，这条语句没有问题。表达式 $j*j$ 的值是1 000 000，并且变量*i*是long int型的，所以*i*应该能很容易地存储这种大小的值，不是吗？问题是，当两个int型值相乘时，结果也应该是int类型的，但是 $j*j$ 的结果太大，以致于在某些机器上无法表示成int类型。在这样的机器上，会给变量*i*赋一个无意义的值。幸运的是，可以使用强制类型转换避免这种问题的发生：

```
i = (long int) j * j;
```

因为强制运算符的优先级高于*，所以第一个变量*j*会被转换成long int类型，同时也迫使第二个*j*进行转换。注意语句

```
i = (long int) (j * j); /* *** WRONG ***/
```

是不对的，因为溢出在强制类型转换之前就已经发生了。

7.6 类型定义

5.2节中，我们使用#define指令创建了一个宏，可以用来定义布尔型数据：

```
#define BOOL int
```

Q&A但是，一个更好的设置布尔型的方法是利用所谓的**类型定义**（type definition）的特性：

```
typedef int Bool;
```

注意，所定义的类型的名字放在最后。还要注意，我们使用大写的单词Bool。将类型名的首字母大写不是必需的，它只是一些C语言程序员使用的一种习惯。

采用typedef定义Bool会导致编译器在它所识别的类型名列表中加入Bool。现在，Bool类型可以和内置的类型名一样用于变量声明，强制类型转换表达式和其他地方了。例如，可以使用Bool声明下列变量：

129

```
Bool flag; /* same as int flag; */
```

编译器将会把Bool类型看成是int类型的同义词；因此，变量flag实际就是一个普通的int型变量。

类型定义使得程序更加易于理解（假定程序员是仔细选择了有意义的类型名）。例如，假设变量cash_in和变量cash_out将用于存储美元数量。把Dollars声明成

```
typedef float Dollars;
```

并且随后写出

```
Dollars cash_in, cash_out;
```

这样的写法比下面的写法更有实际意义：

```
float cash_in, cash_out;
```

类型定义还可以使程序更容易修改。如果稍后决定Dollars实际应该定义为double类型的，那么只需要改变类型定义就足够了：

```
typedef double Dollars;
```

Dollars变量的声明不需要进行改变。如果不使用类型定义，则需要找到所有用于存储美元数量的float型变量（这显然不是件容易的工作）并且改变它们的声明。

类型定义是编写可移植程序的一种重要工具。程序从一台计算机移动到另一台计算机可能引发的问题之一就是不同计算机上的类型取值范围可能不同。如果i是int型的变量，那么赋值语句

```
i = 100000;
```

在一台使用32位整数的机器上是没问题的，但是在一台使用16位整数的机器上就会出错。

可移植性技巧 为了更大的可移植性，可以考虑使用typedef定义新的整型名。

假设编写的程序需要用变量来存储产品数量，取值范围在0~50 000。为此可以使用long int型的变量（因为这样保证可以存储至少在2 147 483 647以内的数），但是用户更愿意使用int型的变量，因为算术运算时int型值比long int型值运算速度快；同时，int型变量可以占用较少的空间。

我们可以定义自己的“数量”类型，而避免使用int类型声明数量变量：

```
typedef int Quantity;
```

并且使用这种类型来声明变量：

```
Quantity q;
```

当把程序转到使用小值整数的机器上时，需要改变Quantity的定义：

```
typedef long int Quantity;
```

可惜的是，这种技术无法解决所有的问题，因为Quantity定义的变化可能会影响Quantity类型变量的使用方式。至少使用了Quantity类型的变量在进行scanf函数和printf函数调用时也需要改动，用转换说明%ld替换%d。

C语言库自身使用typedef为那些可能依据C语言实现的不同而不同的类型创建类型名；这些类型的名字经常以_t结尾，比如ptrdiff_t、size_t和wchar_t。编译器可能在它的库中有下列类型定义：

```
typedef int ptrdiff_t;
typedef unsigned size_t;
typedef char wchar_t;
```

其他编译器可能采用不同的方式定义这些类型。例如，ptrdiff_t可能在某些机器上是long int类型。

问与答

问：如果“溢出”会发什么？比如，两个数相加的结果过大而无法存储。（p.80）

答：这取决于数是有符号型的还是无符号型的。当溢出发生在有符号数的操作上时，依据C语言的标准，结果是“未定义的”。我们无法准确说出结果是什么，因为这依赖于机器的行为。程序甚至可能会异常中断（对除以零的典型反应）。

但是，当溢出发生在无符号数的操作上时，结果是定义了的：可以获得正确答案对 2^n 进行取模运算

的结果，这里的n是用于存储结果使用的位数。例如，如果用1加上无符号的16位数65 535，那么结果肯定是0。

问：7.1节说到%o和%x分别用于以八进制和十六进制书写无符号整数。那么如何以八进制和十六进制书写普通的（有符号的）整数呢？(p.80)

答：只要它的值不是负值，我们就可以用%o和%x显示有符号的整数。这些转换导致printf函数把有符号整数看成是无符号的；换句话说，printf函数将假设符号位是数的绝对值部分。只要符号位为0，就不是问题。如果符号位为1，那么printf函数将显示出一个超出预期的大数。

问：但是，如果数是负数该怎么办呢？如何以八进制或十六进制形式书写它？

答：没有直接的方法可以书写负数的八进制或十六进制形式。幸运的是，需要这样做的情况非常少。当然，可以判定这个数是否是负数，并且自行显示一个负号：

```
if (i < 0)
    printf("-%x", -i);
else
    printf("%x", i);
```

问：浮点常量为什么存储成double格式而不是float格式？(p.82)

答：由于历史的原因，C语言更倾向于使用double类型；float类型则被看成是“二等公民”。思考Kernighan和Ritchie的*The C Programming Language*一书中关于float的论述：“在大型数组中使用float类型的主要原因是节省存储空间，或者有时是为了节省时间，因为在一些机器上双精度计算花销格外大。”时至今日，经典C一直要求所有浮点计算都采用双精度的形式。（标准C没有这样的要求。）

*问：为什么使用%lf读取double型的值，而用%f进行显示呢？(p.82)

答：这是一个十分难回答的问题。首先，注意scanf函数和printf函数都是不同寻常的函数，因为它们都没有将函数的参数限制为固定数量。scanf函数和printf函数有可变长度的参数列表(>26.1节)。当调用带有可变长度参数列表的函数时，编译器会安排float参数自动转换成为double类型，其结果是printf函数无法区分float型和double型的参数。这解释了在printf函数调用中为何可以用%f既表示float型又表示double型的参数。

另一方面，scanf函数是通过指针指向变量的。%f告诉scanf函数在所传地址位置上存储一个float型值，而%lf告诉scanf函数在该地址上存储一个double型值。这里float和double的区别是非常重要的。如果给出了错误的转换说明，那么scanf函数将可能存储错误的字节数量（没有提到的是，float型的位模式可能不同于double型的位模式）。

问：char的正确发音是什么？(p.83)

答：没有普遍接受的发音。一些人把char发音成“character”的第一个音节Kæ，还有一些人则把它念成tja:(r)，就像在char broiled;中那样。

问：什么时候需要考虑字符变量是有符号的还是无符号的？(p.84)

答：如果在变量中只存储7位的字符，那么不需要考虑，因为符号位将为零。但是，如果计划存储8位字符，那么将希望变量是unsigned char类型。思考下面的例子：

```
ch = '\xdb';
```

如果已经把变量ch声明成char类型，那么编译器可能选择把它看作是有符号的字符来处理（许多编译器这么做）。只要变量ch只是作为字符来使用，就不会有什么问题。但是如果ch用在一些需要编译器将其值转换为整数的上下文中，那么可能就有问题了：转换为整数的结果将是负数，因为变量ch的符号位为1。

还有另外一种情况：在一些程序中，习惯使用char型变量存储单字节的整数。如果编写了这类程序，就需要决定每个变量应该是signed char类型的还是unsigned char类型的，这就像需要决定普通整型变量应该是int类型还是unsigned int类型一样。

问：我无法理解换行符(new-line)怎么会是ASCII码的回行符(line feed)。当用户录入输入内容并且按回车键时，程序不会把它作为回车符或者回车加回行符读取吗？(p.84)

答：不会的。作为C语言的UNIX继承部分，一直把行的结束位置标记作为单独的回行字符来看待。（在UNIX文本文件中，单独一个回行符（但不是回车符）会出现在每行的结束处。）C语言函数库会把用户的按键翻译成回行符。当程序读文件时，输入/输出函数库将文件的行结束标记（不管它是什么）翻译成单独的回行符。与之相对应的反向转换发生在将输出往屏幕或文件中写的时候。（细节请见22.1节。）

虽然这些翻译可能看上去很混乱，但是它们都为了一个重要的目的：使程序不受不同操作系统的影响。

*问：使用转义序列\?的目的是什么？(p.84)

答：转义序列\?与三字符序列有关，因为三字符序列以??开头。如果需要在字符串中加入??，那么编译器很可能把它误当作成三字符序列（>25.3节）的开始。用\?代替第二个?可以解决这个问题。

问：既然getchar函数读取速度快，为什么仍然需要使用scanf函数读取单个的字符呢？(p.86)

答：虽然scanf函数没有getchar函数读取的速度快，但是它更灵活。正如前面已经看到的，格式串"%c"可以使scanf函数读入下一个输入字符；"%c"则可以使scanf函数读入下一个非空白字符。而且，scanf函数也很擅长读取混合了其他数据类型的字符。假设输入数据中包含有一个整数、一个单独的非数值型字符和另一个整数。通过使用格式串"%d%c%d"，就可以利用scanf函数读取全部三项内容。[133]

*问：在什么情况下，整型提升会把字符或短整数转换成unsigned int型整数？(p.89)

答：如果int型整数不够大到可以包含所有可能的原始类型值的情况下，整型提升会产生unsigned int型。因为字符通常是8位的长度，而且至少可以保证int型为16位的长度，所以整型提升几乎总会把字符转化为int型。但是，无符号短整数却是有疑问的。如果短整数和普通整数的长度相同（假设它们都用于16位机上），那么整型提升必将会把无符号短整数转化为unsigned int型，因为最大的无符号短整数（在16位机上为65 535）要大于最大的int型数（即32 767）。

问：如果把超出变量承受范围的值赋值给变量，究竟会发生什么？(p.91)

答：很难讲，如果值是整型并且变量是无符号类型，那么会舍丢掉超出的位数；如果变量是有符号类型，那么结果是由实现定义的。把浮点数赋值给整数型或浮点数型变量的话，这两种类型变量都会因为太小而无法承受，因此产生未定义的行为：任何事情都可能发生，包括程序终止。

*问：为什么C语言担心提供类型定义呢？定义一个BOOL宏不是和用typedef定义一个Bool类型一样有用吗？(p.92)

答：在类型定义和宏定义之间存在两个重要的不同点。首先，类型定义比宏定义功能更强大。特别是，数组和指针类型是不能定义为宏的。假设我们试图使用宏来定义“指向整数的指针”类型：

```
#define PTR_TO_INT int *
```

声明

```
PTR_TO_INT p, q, r;
```

在处理以后，将会变成

```
int * p, q, r;
```

可惜的是，只有p是指针；q和r都成了普通的整型变量。类型定义不会有这样的问题。

其次，typedef命名的对象具有和变量相同范围的规则；定义在函数体内的typedef名字在函数外是无法识别的。另一方面，宏的名字在预处理时会在任何出现的地方被替换掉。[134]

练习

7.1节

1. 请给出下列整型常量的十进制数值。

- (a) 077
- (b) 0x77

- (c) 0XABC
2. 如果 $i * i$ 超出了int型的最大取值，那么6.3节的程序square2.c将失败（通常会显示奇怪的答案）。运行程序，并且确定导致失败的n的最小值。尝试把变量i的类型改换成short int类型，并且再次运行程序。（不要忘记更新printf函数调用中的转换说明！）然后尝试改换成long int类型。从这些实验中，你能总结出在你的机器上用于存储整型的位数是多少吗？

7.2节

3. 下列哪些在C语言中不是合法的数？区分每一个合法的数是整数还是浮点数。
- (a) 010E2
 - (b) 32.1E+5
 - (c) 0790
 - (d) 100_000
 - (e) 3.978e-2
4. 下列哪些在C语言中不是合法的类型？
- (a) short unsigned int
 - (b) short float
 - (c) long double
 - (d) unsigned long
5. 修改程序sum2.c（7.1节）以便可以进行一串double型值的求和计算。

7.3节

6. 如果变量c是char类型，那么下列哪条语句是非法的？
- (a) `i += c; /* i has type int */`
 - (b) `c = 2 * c - 1;`
 - (c) `putchar(c);`
 - (d) `printf(c);`
7. 下列哪条在书写数65上不是合法的方式？（假设字符集是ASCII。）
- (a) 'A'
 - (b) 0b1000001
 - (c) 0101
 - (d) 0x41
8. 修改6.3节的程序square2.c以便它在每24次平方后暂停并且显示下列信息：

`Press Enter to continue...`

在显示完消息后，程序应该使用getchar函数读入一个字符。getchar函数将不允许程序继续直到用户录入回车（或返回）键。

9. 编写程序可以把字母格式的电话号码翻译成数值格式：

`Enter phone number: CALLATT
2255288`

（万一没有电话在身边，后面有字母在键盘上的对应关系：2=ABC，3=DEF，4=GHI，5=JKL，6=MNO，7=PRS，8=TUV，9=WXY。）如果原始的电话号码包含非字母的字符（例如，数字或标点符号），那么保留下来不做变化：

`Enter phone number: 1-800-CO-L-ECT
1-800-265-5328`

可以假设任何用户输入的字母都是大写字母。

10. 在十字拼字游戏中，玩家利用小卡片组成单词，每个卡片包含字母和面值。面值根据字母的不同而不同，也就是说面值是基于字母变化的。（面值有：1——AEILNORSTU，2——DG，3——BCMP，4——FHVWY，5——K，8——JX，10——QZ。）编写程序通过对字母对应的面值求和来计算单词的值：

`Enter a word: pitfall
Scrabble value: 12`

编写的程序应该允许单词中混合出现大小写字母。提示：使用`toupper`库函数。

11. 飞机票有冗长的标识数字，例如47715497443。为了有效，最后一位数字必须与以其他位的数字为整体除以7后的余数相匹配。（例如，4771549744除以7的余数为3。）编写程序检查机票号是否有效：

```
Enter ticket number: 47715497443  
VALID
```

提示：不要试图在单步操作中读取数，而是使用`getchar`函数逐个获取数字。一次执行一个数字的除法，小心除法中不要包含最后一位数字。

7.4节

12. 编写程序显示`sizeof(int)`、`sizeof(short int)`、`sizeof(long int)`、`sizeof(float)`、`sizeof(double)`和`sizeof(long double)`的值。

7.5节

13. 假设变量*i*和变量*j*都是int类型，那么表达式*i / j + 'a'*是什么类型？
14. 假设变量*i*是int类型，变量*j*是long int类型，并且变量*k*是unsigned int类型，那么表达式*i + (int)j * k*是什么类型？
15. 假设变量*i*是int类型，变量*f*是float类型，并且变量*d*是double类型，那么表达式*i * f / d*是什么类型？
16. 假设变量*i*是int类型，变量*f*是float类型，并且变量*d*是double类型，请解释在执行下列语句时发生了什么转换？

```
d = i + f;
```

136

17. 假设程序包含下列声明：

```
char c = '\1';  
short int s = 2;  
int i = -3;  
long int m = 5;  
float f = 6.5;  
double d = 7.5;
```

请给出下列每个表达式的值和类型。

- (a) $c * i$ (c) f / c (e) $f - d$
(b) $s + m$ (d) d / s (f) $(int) f$

18. 下列语句是否始终可以正确地计算出*f*的小数部分（假设*f*和`frac_part`都是float型的变量）？

```
frac_part = f - (int) f;
```

如果不是，那么出了什么问题？

7.6节

19. 使用`typedef`产生名为Int8、Int16和Int32的类型。定义这些类型以便它们可以在你的机器上分别表示8位、16位和32位的整数。

137

如果程序操纵着大量的数据，那它一定是用较少的方法办到的。

到目前为止所见的变量都只是标量 (scalar)：标量具有保存单一数据项的能力。C语言也支持聚合 (aggregate) 变量，这类变量可以存储数值的集合。在C语言中一共有两种聚合类型：数组 (array) 和结构 (structure) (记录) (►16.1节)。本章会介绍一维数组 (8.1节) 和多维数组 (8.2节) 的声明和使用。内容将主要集中讨论一维数组，因为与多维数组相比，一维数组在C语言中占有更加重要的角色。后面的几章 (特别是第12章) 对数组还提供了附加信息；第16章介绍结构。

8.1 一维数组

数组是含有多个数据值的数据结构，并且每个数据值具有相同的数据类型。这些数据值被称为元素 (element)，数组内可以根据元素所处的位置对其进行单独选择。

最简单的数组类型就是一维数组，一维数组中的元素一个接一个地编排在单独一行 (如果你喜欢，也可以说成是一列) 内。这里可以假设有一个名为a的一维数组：



为了声明数组，需要说明数组元素的类型和数量。例如，为了声明数组a有10个int型的元素，可以写成

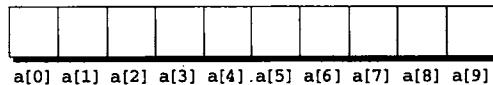
`int a[10];`

数组的元素可以是任何类型；数组的长度可以用任何 (整数) 常量表达式 (►5.3节) 说明。因为在程序后面变化时可能需要调整数组的长度，所以较好的方法是用宏来定义数组的长度：

```
#define N 10
int a[N];
```

8.1.1 数组下标

为了存取特定的数组元素，可以在写数组名的同时在后边加上一个用方括号围绕的整数值 (称这是对数组进行下标 (subscripting) 或索引 (indexing))。Q&A 数组元素始终从0开始，所以长度为n的数组元素的索引是从0到n-1。例如，如果a是含有10个元素的数组，那么这些元素可以如下图所示的依次标记为a[0], a[1], ..., a[9]：



`a[i]`的表达式格式是左值 (►4.2.2节)，所以数组元素可以和普通变量一样使用：

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```

数组和for循环结合在一起使用。许多程序所包含的for循环都是为了对数组中的每个元素执行一些操作。这里有一些示例是关于长度为N的数组的典型操作：

```
[惯用法] for (i = 0; i < N; i++)
    a[i] = 0;                                /* clear a */

[惯用法] for (i = 0; i < N; i++)
    scanf("%d", &a[i]);                     /* reads data into a */

[惯用法] for (i = 0; i < N; i++)
    sum += a[i];                            /* sums the elements of a */
```

注意，在调用scanf函数读取数组元素时，就像对待普通变量一样，必须使用取地址符号&。



140C语言不要求检查下标的范围；当下标超出范围时，程序可能执行不可预知的行为。下标超出范围的原因之一：是忘记了对n个元素数组的索引是从0到n-1，而不是从1到n。（正如我的一位教授喜欢说的，“在这件事情上，你要永远远离1。”他显然是对的。）下面的例子说明由于这种常见错误从而导致的奇异效果：

```
int a[10], i;
for (i = 1; i <= 10; i++)
    a[i] = 0;
```

对于某些编译器来说，这个表面上正确的for语句却产生了一个无限循环！当变量i的值变为10时，程序将数值0存储在a[10]中。但是a[10]这个元素并不存在，所以在元素a[9]后数值0立刻进入内存。如果内存中变量i放置在a[9]的后边，因为这是可能发生的一种情况，那么变量i将会重新设置为0，这也就会导致再次开始循环。

数组下标可以是任何整数表达式：

```
a[i+j*10] = 0;
```

表达式甚至可能会有副作用：

```
i = 0;
while (i < N)
    a[i++] = 0;
```

让我们一起来跟踪一下这段代码。在把变量i设置为0后，while语句判断变量i是否小于N。如果是，那么将数值0赋值给a[0]，随后i自增，然后重复循环。注意，a[++i]是不正确的，因为第一次循环操作期间将会把0赋值给a[1]。



当数组下标有副作用时一定要注意。例如，下面这个循环用来把数组b复制给数组a，可是它可能无法正常工作：

```
i = 0;
while (i < N)
    a[i] = b[i++];
```

在每次对a[i]赋值之前，必须确定好对应于a[i]和b[i++]内存位置。如果运气好的话，将会首先确定好对应a[i]的内存位置以便于把b[i]复制给a[i]。如果不走运的话，将可能首先确定好对应b[i++]内存位置，变量i进行自增，然后才把b[i]的值复制给a[i+1]。当然，通过从下标中移走自增操作的方法可以很容易避免此类问题的发生：

```
for (i = 0; i < N; i++)
    a[i] = b[i];
```

8.1.2 程序：数列反向

第一个关于数组的程序要求用户录入一串数，然后按反向顺序输出这些数：

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
In reverse order: 31 50 11 23 94 7 102 49 82 34
```

方法是在读入数时将其存储在一个数组中，然后通过数组反向开始一个接一个地显示出数组元素。换句话说，不会真的对数组中的元素进行反向，只是使用户这样认为。

```
reverse.c
/* Reverses a series of numbers */

#include <stdio.h>

#define N 10

main()
{
    int a[N], i;

    printf("Enter %d numbers: ", N);
    for (i = 0; i < N; i++)
        scanf("%d", &a[i]);

    printf("In reverse order:");
    for (i = N - 1; i >= 0; i--)
        printf(" %d", a[i]);
    printf("\n");

    return 0;
}
```

这个程序正好显示了宏和数组联合使用可以多么的有效。程序一共4次使用到了宏N：在数组a的声明中，在显示提示的printf函数中，还有两个for循环中。如果需要稍后决定变化数组的大小，只需要编辑N的定义并且重新编译程序就可以了，其他什么也不需要改变，甚至是提示也始终是正确的而无需更换。

8.1.3 数组初始化

像其他变量一样，数组也可以在声明时获得一个初始值。但是，数组初始化的规则需要有些技巧，所以现在将会介绍一些，其他的留在后面介绍（见18.5节）。

数组初始化式（array initializer）最通用的格式是一个常量表达式列表，列表用大括号括起来，并且内部数值用逗号进行分隔：

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

如果初始化式比数组短，那么数组中剩余的元素赋值为0：

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

利用这一特性，可以很容易地给全部数组元素初始化为0：

```
int a[10] = {0};
/* initial value of a is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

初始化式完全为空是非法的，所以要在大括号内放上一个单独的0。初始化式长过要初始化的数组也是非法的。

如果显示一个初始化式，那么可以忽略掉数组的长度：

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

编译器利用初始化式的长度来确定数组的大小。数组始终有元素的固定数量（在此例中，数量

为10), 这就好像已经明确地指明了长度。

8.1.4 程序：检查数中重复出现的数字

接下来这个程序用来检查数中是否有出现多于一次的数字。用户输入数后, 程序显示信息

Repeated digit或No Repeated digit:

```
Enter a number: 28212
Repeated digit
```

数28212有一个重复的数字(2); 而数9357则没有。

程序采用布尔型值的数组跟踪数中出现的数字。名为digit_seen的数组有十个可能的数字, 数组元素的下标索引从0到9。最初的时候, 每个数组元素的值都为0(假的)。当给出数n时, 程序一次一个地检查n的数字, 并且把每次的数字存储在变量digit中, 然后用这个数字作为数组digit_seen的下标索引。如果digit_seen[digit]为真, 那么表示digit至少在n中出现了两次。另一方面, 如果digit_seen[digit]为假, 那么表示digit之前未出现过, 因此程序会把digit_seen[digit]设置为真并且继续执行。

```
repdigit.c
/* Checks numbers for repeated digits */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

typedef int Bool;

main()
{
    Bool digit_seen[10] = {0};
    int digit;
    long int n;

    printf("Enter a number: ");
    scanf("%ld", &n);

    while (n > 0) {
        digit = n % 10;
        if (digit_seen[digit])
            break;
        digit_seen[digit] = TRUE;
        n /= 10;
    }

    if (n > 0)
        printf("Repeated digit\n\n");
    else
        printf("No repeated digit\n\n");

    return 0;
}
```

143

注意, 数n的类型为long int, 因此允许用户录入的数的上限为2 147 483 647(或者在某些机器上可能允许的数值更大)。

8.1.5 对数组使用 sizeof 运算符

运算符sizeof可以确定数组的大小(字节数)。如果数组a有十个整数, 那么sizeof(a)可以代表为20(如果整数是16位长)或者40(如果整数是32位长)。

还可以用`sizeof`来计算数组元素的大小。用数组的大小除以数组元素的大小可以得到数组的长度：

```
sizeof(a) / sizeof(a[0])
```

当需要数组长度时，一些程序员采用上述这个表达式。例如，数组a的清零操作可以写成如下形式

```
for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)
    a[i] = 0;
```

利用这种技术，即使数组长度在日后需要改变，也不需要改变循环。当然，利用宏来表示数组的长度也可以获得同样的好处，但是`sizeof`技术稍微更好一些，因为没有宏的名字需要记住（并且可能由此产生错误）。

[144] 表达式`sizeof(a) / sizeof(a[0])`有一点难于使用；定义宏来表示它常常是很有帮助的：

```
#define SIZE (sizeof(a) / sizeof(a[0]))
for (i = 0; i < SIZE; i++)
    a[i] = 0;
```

但是，返回来使用宏的话，`sizeof`的优势又是什么呢？后面的章中将对这个问题进行回答（窍门是给宏加上“参数”，即带参数的宏（>14.3.2节））。

8.1.6 程序：计算利息

下面这个程序打印出一个表格，这个表格显示了在几年时间内100美金投资在不同利率上的价值。用户将输入利率和要投资的年数。假设整合利息一年一次，表格将显示出一年间在此输入利率下和后边4个更高利率下投资的价值。下面是程序运行时的情况：

```
Enter interest rate: 6
Enter number of years: 5

Years      6%      7%      8%      9%      10%
1          106.00  107.00  108.00  109.00  110.00
2          112.36  114.49  116.64  118.81  121.00
3          119.10  122.50  125.97  129.50  133.10
4          126.25  131.08  136.05  141.16  146.41
5          133.82  140.26  146.93  153.86  161.05
```

很明显地，可以使用`for`语句显示出第一行信息。第二行的显示有点小窍门，因为它的值要依赖于第一行的数。解决方案是把第一行的数存储在数组中，就像计算它一样，然后使用数组中的这些值计算第二行的内容。当然，从第三行到最后一行可以重复这个过程。程序将以两个`for`语句结束，其中一个嵌套在另一个里面。外层循环将从1计数到用户要求的年数，内层循环将从利率的最低值自增到最高值。

```
interest.c
/* Prints a table of compound interest */

#include <stdio.h>

#define NUM_RATES (sizeof(value)/sizeof(value[0]))
#define INITIAL_BALANCE 100.00

main()
{
    int i, low_rate, num_years, year;
    float value[5];

    printf("Enter interest rate: ");
```

```

scanf("%d", &low_rate);
printf("Enter number of years: ");
scanf("%d", &num_years);

printf("\nYears");
for (i = 0; i < NUM_RATES; i++) {
    printf("%6d", low_rate+i);
    value[i] = INITIAL_BALANCE;
}
printf("\n");

for (year = 1; year <= num_years; year++) {
    printf("%3d ", year);
    for (i = 0; i < NUM_RATES; i++) {
        value[i] += (low_rate+i) / 100.0 * value[i];
        printf("%7.2f", value[i]);
    }
    printf("\n");
}

return 0;
}

```

注意，这里使用NUM_RATES控制两个for循环。如果后面改变数组value的大小，循环将会自动调整。

8.2 多维数组

数组可以有任意维数。例如，下面的声明产生了一个二维数组（或者按数学概念称为矩阵（matrix））：

```
int m[5][9];
```

数组m有5行9列。如下图所示，数组的行和列下标都是从0开始索引：

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | | |
| 1 | | | | | | | | | |
| 2 | | | | | | | | | |
| 3 | | | | | | | | | |
| 4 | | | | | | | | | |

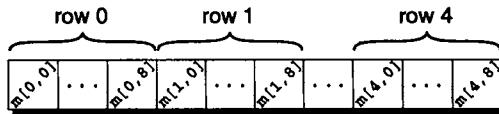
146

为了在i行j列中存取数组m的元素，需要写成m[i][j]的形式。表达式m[i]指明了数组m的第i行，而m[i][j]才是选择了此行中的第j个元素。



抵制诱惑把m[i][j]替换写成m[i,j]。在此处，C语言把逗号看成是逗号运算符（>6.3.3节），所以m[i,j]就等同于m[j]。

虽然以表格形式显示二维数组，但是实际上它们在计算机的内存中不是这样存储的。C语言是按照行主序存储数组的，也就是从第0行开始，接着第1行，如此下去。例如，下面显示了数组m的存储：



就像for循环和一维数组相结合一样，嵌套的for循环是处理多维数组的理想选择。例如，思考用作单位矩阵（identity matrix）的数组的初始化问题。（数学中，单位矩阵在主对角线上的值为1，而其他地方的值为0，其中主对角线上行、列的索引值是完全相同的。）在某些系统方式中将会需要访问数组中的每一个元素。一对嵌套的for循环可以很好地完成这项工作，因为嵌套循环中执行一步可以穿过每行的索引，也可以执行一步穿过每列的索引：

```
#define N 10
float ident[N][N];
int row, col;

for (row = 0; row < N; row++)
    for (col = 0; col < N; col++)
        if (row == col)
            ident[row][col] = 1.0;
        else
            ident[row][col] = 0.0;
```

和其他编程语言中的多维数组相比，C语言中的多维数组扮演的角色相对较弱，这主要是因为C语言为存储多维数据提供了更加灵活的方法：指针数组（>13.7节）。

8.2.1 多维数组初始化

147

通过嵌套一维初始化式的方法可以产生二维数组的初始化式：

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
                {0, 1, 0, 1, 0, 1, 0, 1, 0},
                {0, 1, 0, 1, 1, 0, 0, 1, 0},
                {1, 1, 0, 1, 0, 0, 0, 1, 0},
                {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

每一个内部初始化式提供了一行矩阵的数值。构造高维数组的初始化式采用类似的方法。

C语言为多维数组提供了多种方法来缩写初始化式：

- 如果初始化式不大到足以填满整个多维数组，那么把数组中剩余的元素赋值为0。例如，下面的初始化式只填满了数组m的前三行；后边的两行将赋值为0：

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
                {0, 1, 0, 1, 0, 1, 0, 1, 0},
                {0, 1, 0, 1, 1, 0, 0, 1, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0}};
```

- 如果内层的列表不大到足以填满数组的一行，那么把此行剩余的元素初始化为0：

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
                {0, 1, 0, 1, 0, 1, 0, 1, 1},
                {0, 1, 0, 1, 1, 0, 0, 1, 1},
                {1, 1, 0, 1, 0, 0, 0, 1, 1},
                {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

- 甚至可以忽略掉内层的大括号：

```
int m[5][9] : {1, 1, 1, 1, 1, 0, 1, 1, 1,
               0, 1, 0, 1, 0, 1, 0, 1, 0,
               0, 1, 0, 1, 1, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 1, 1, 1};
```

因为一旦编译器发现足够的数值填满一行，它就开始填充下一行。



在多维数组中忽略掉内层的大括号可能是很危险的，因为额外的元素（或者设置更糟的是丢失的元素）将会影响剩下的初始化式。省略括号会引起某些编译器产生类似“Initialization is only partially bracketed.”这样的警告消息。

8.2.2 常量数组

无论一维数组还是多维数组，通过把单词const作为数组声明开始这种方法可以把任何数组变为“常量”：

148

```
const int months[] =
{31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

程序不应该对声明为const的数组进行修改；编译器为了修改元素将检查到这种直接的意图。

声明数组是const的有两个主要的好处。它表明程序不会改变数组，因为对于某些后面阅读程序的人来说，数组可能是有价值的信息。告知不打算修改数组对编译器发现错误也是很有帮助的。

const（>18.3节）不只限应用于数组；就像后面将看到的，它可以和任何变量一起使用。但是，const时常和数组联合使用，因为他们经常含有参考信息，这些信息在程序执行过程中是不希望发生改变的。

8.2.3 程序：发牌

下面这个程序说明了二维数组和常量数组的用法。程序负责发一副标准纸牌。（标准纸牌的花色有梅花、方块、红桃或黑桃，而且纸牌的等级有2、3、4、5、6、7、8、9、10、J、Q、K或A。）程序需要用户指明手里应该握有几张牌：

```
Enter number of cards in hand: 5
Your hand: 7c 2s 5d as 2h
```

不会立刻很明显地看出如何编写这样一个程序。如何从一副牌中随机抽取纸牌呢？而且如何避免两次抽到同一张牌呢？现在将分别处理这些疑问。

为了随机抽取纸牌，可以采用一些C语言的库函数。time函数（>26.3.1节）（来自于<time.h>）返回当前的时间，且这个时间是被编码成单独的数。srand函数（来自于<stdlib.h>）（>26.2.3节）初始化C语言的随机数生成器。通过把time函数的返回值传递给函数srand这种方法可以避免程序在每次运行时发同样的牌。rand函数（来自于<stdlib.h>）（>26.2.3节）在每次调用时会产生一个显然随机的数。通过采用运算符%，可以标量来自rand函数的返回值，这样可以使得这个值落在0~3（为了表示牌的花色）的范围内，或者是落在0~12（为了表示纸牌的等级）的范围内。

149

为了避免每次都拿到同一张牌，需要跟踪已经选择好的牌。为了这个目的，程序将采用一个名为in_hand的二维数组，其中数组有4行（每行表示一种纸牌的花色）和13列（每一列表示纸牌的一种等级）。换句话说，数组中的所有元素分别对应着52张纸牌中的一张。在程序开始时，所有数组元素都将为0（假的）。每次随机抽取一张纸牌时，将检查数组in_hand的元素与此牌是否相对应，对应就为真，不对应则为假。如果判定结果为真，那么就需要抽取其他纸牌；如果判定结果为假，则将把数值1存储到与此张纸牌相对应的数组元素中，这样做是为了以后提醒此张纸牌已经抽取过了。

一旦证实纸牌是“新”的，也就是说还没有选取过此张纸牌，就需要把牌的等级和花色数值翻译成字符，然后显示出来。为了把纸牌的等级和花色翻译成字符格式，程序将设置两个字符数组，一个用于纸牌的等级，而另一个用于纸牌的花色，然后利用数对数组进行下标。这两

一个字符数组在程序执行期间不会发生改变，所以可以把它们声明为const。

```
deal.c
/* Deals a random hand of cards */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_SUITS 4
#define NUM_RANKS 13
#define TRUE 1
#define FALSE 0

typedef int Bool;

main()
{
    Bool in_hand[NUM_SUITS][NUM_RANKS] = {0};
    int num_cards, rank, suit;
    const char rank_code[] = {'2','3','4','5','6','7','8',
                             '9','t','j','q','k','a'};
    const char suit_code[] = {'c','d','h','s'};

    srand((unsigned) time(NULL));

    printf("Enter number of cards in hand: ");
    scanf("%d", &num_cards);

    printf("Your hand:");
    while (num_cards > 0) {
        suit = rand() % NUM_SUITS;      /* picks a random suit */
        rank = rand() % NUM_RANKS;      /* picks a random rank */
        if (!in_hand[suit][rank]) {
            in_hand[suit][rank] = TRUE;
            num_cards--;
            printf(" %c%c", rank_code[rank], suit_code[suit]);
        }
    }
    printf("\n");
}

return 0;
}
```

注意，数组in_hand的初始化式：

```
Bool in_hand[num_SUITS][NUM_RANKS] = {0};
```

即使in_hand是二维数组，C语言仍允许使用单独一对大括号。而且，由于只给出了初始化式中的一个值，所以根据前面的内容可以知道C语言将会把其他数组元素填充为0值。

问与答

问：为什么数组下标从0开始而不是从1开始？(p.98)

答：拥有从0开始的下标可以使编译器简化一点。而且，这样也可以使得数组下标运算的速度有少量的提高。

问：如果希望数组的下标从1到10而不是从0到9，该怎么做呢？

答：这有一个常用的窍门：声明数组有11个元素而不是10个元素。这样数组的下标将会从0到10，但是可以忽略掉下标为0的元素。

问：使用字符作为数组的下标是否可行呢？

答：是可以的，因为C语言把字符作为整数来处理。但是，在使用字符作为下标前，可能需要对字符进行“标量化”。假设希望数组letter_count可以对字母表中每个字母进行跟踪计数。这个数组将需要26个元素，所以采用下列方式对其进行声明：

```
int letter_count[26];
```

然而，不能直接使用字母作为数组letter_count的下标，因为字母的整数值不是落在0到25的区间内的。为了把小写字母标量到合适的范围内，可以简单采用减去'a'的方法；为了标量到大写字母，则可以减去'A'。例如，如果ch含有小写字母，那么为了对相应的ch字母进行计数，所以ch的清零操作就可以写成

```
letter_count[ch-'a'] = 0;
```

问：如果企图通过使用赋值运算符在数组间进行复制操作，编译器将给出出错信息。错误是什么呢？

答：虽然表达式

```
a = b; /* a and b are arrays */
```

看上去似乎合理，但它确实是非法的。非法的理由不是显而易见的；这需要用到C语言中数组和指针之间的特殊关系，这一点将会在第12章进行探讨。

把一个数组复制给另一个数组，最简单的实现方法是利用循环对数组元素逐个进行复制：

```
for (i = 0; i < N; i++)
    a[i] = b[i];
```

另一种可行的方法是使用来自<string.h>的函数memcpy（意思是“内存复制”）memcpy函数（>23.5.1节）是低级函数，它把字节从一个地方简单复制到另一个地方。为了把数组b复制给数组a，使用函数memcpy的格式如下：

```
memcpy (a, b, sizeof(a));
```

一些程序员喜欢memcpy函数，特别是对大型数组，因为它潜在的速度比普通循环更快。

151

练习

8.1节

- 修改程序repdigit.c，要求修改后的程序可以显示出重复的数字（如果有的话）：

```
Enter a number: 939577
Repeated digit(s): 7 9
```

- 修改程序repdigit.c，要求修改后的程序可以显示出一张列表，表内显示出每种数字在数中出现的次数：

```
Enter a number: 41271092
Digit:      0 1 2 3 4 5 6 7 8 9
Occurrences: 1 2 2 0 1 0 0 1 0 1
```

- 修改程序repdigit.c，要求修改后的程序可以让用户录入多于一个的数进行重复数字的判断。当用户录入的数小于或等于0时，程序终止。
- 已经讨论过利用表达式sizeof(a) / sizeof(a[0])进行数组元素个数的计算。表达式sizeof(a) / sizeof(t)也可以完成同样的工作，其中t表示数组a中元素的类型，但是这种方法被认为是一种较差的技术。这是为什么呢？
- 修改程序reverse.c，利用表达式sizeof(a) / sizeof(a[0])（或者这个值的宏）来计算数组的长度。
- 修改程序interest.c，使得修改后的程序可以每月整合一次利息，而不再是每年整合一次利息。程序的输出格式不用改变；余额应该始终按每年一次的间隔显示。
- 在线运动的名人之一是一个称为B1FF的家伙，他在编写消息上有一种独一无二的方法。下面是一条典型的B1FF公告：

H3Y DUD3, C 15 R1LLY COOL!!!!!!

编写一个“B1FF过滤器”，它可以读取用户录入的消息并且把此消息翻译成B1FF的表达风格：

Enter message: Hey_dude, C is rilly cool

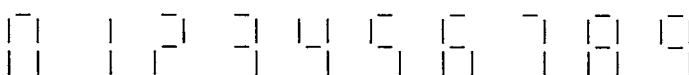
In B1FF-speak: H3Y DUD3, C is R1LLY COOL!!!!!!

程序需要把消息转换成大写字母，用数字代替特定的字母（A→4, B→8, E→3, I→1, O→0, S→5），然后添加10个感叹号。提示：在字符数组中存储原始消息，然后从数组头开始逐个翻译并且显示字符。

8. “问与答”小节介绍了使用字母作为数组下标的方法。请描述一下如何使用数字（字符格式的）作为数组的下标。

8.2节

9. 计算器、手表和其他电子设备经常依靠七段显示器进行数值的输出。为了组成数字，这类设备需要“打开”7个显示段中的某些部分，同时还需要“关闭”7个显示段中的其他部分：



假设需要设置一个数组，用它来记住每个数字中需要“打开”的显示段。各显示段的编号如下所示：



下面是数组可能的样子，其中数组的每一行表示一个数字：

```
const int segments[10][7] = {{1,1,1,1,1,1,0}, ...};
```

上面已经给出了第一行的初始化式，请填充余下的部分。

10. 利用8.2节的简洁描述对数组segments（练习9中的）的初始化式尽可能地进行化简。
11. 编写程序，要求此程序可以用来读取一个 5×5 的整数数组，然后显示出每行的求和结果和每列的求和结果。

```
Enter row 1: 8 3 9 0 10
Enter row 2: 3 5 17 1 1
Enter row 3: 2 8 6 23 1
Enter row 4: 15 7 3 2 9
Enter row 5: 6 14 2 6 0
```

Row totals: 30 27 40 36 28

Column totals: 34 37 37 32 21

12. 修改练习11，要求修改后的程序可以提示每个学生5门测验的成绩，一共有5个学生，然后计算每个学生的5门测验的总分和平均分，还要列出每门测验的平均分、高分和低分。
13. 编写程序，要求此程序可以产生一种贯穿 10×10 数组的“随机步”。数组将包含字符（初始时所有数组元素为字符‘.’）。程序必须是从一个元素随机“走到”另一个元素，对一个元素来说这种走始终向上、向下、向左或向右。程序访问到的元素将用从A到Z的字母进行标记，而且是按顺序进行的访问。
下面是期望输出的一个示例：

```
A . . . . .
B C D . . . .
. F E . . . .
H G . . . .
I . . . .
J . . . . Z .
K . . R S T U V Y .
L M P Q . . . W X .
. N O . . . .
. . . . .
```

提示：利用rand函数和 srand函数（参考程序deal.c）产生随机数。在产生数后，查看此数除以4的余数。余数一共有4种可能的值：0、1、2和3，这些数字分别说明了下一次移动的方向。在执行移

动之前，需要检查两项内容：一是不能超出数组的范围，二是不要选取已经标记了字母的元素。如果两个条件都不满足，尝试换个方向移动。如果全部锁定了下一步的4个方向，那么程序就必须终止了。
下面是提前结束的一个示例：

153

```
A B G H I . . . .  
. C F . J K . . . .  
. D E . M L . . . .  
. . . N O . . . .  
. . W X Y P Q . . . .  
. . V U T S R . . . .  
. . . . . . . . .  
. . . . . . . . .  
. . . . . . . . .  
. . . . . . . . .
```

因为Y的4边都已经锁定了，所以没有地方可以放置下一步的Z了。

154

如果你有一个带了10个参数的过程，那么你很可能还遗漏了一些参数。

在第2章中已经看到，函数简单来说就是一连串组合在一起并且命名的语句。虽然“函数”这个术语来自数学，但是C语言的函数不同于数学函数。在C语言中，函数不一定要有参数，也不一定要计算数值。（在某些编程语言中，“函数”返回一个值，而“过程”不返回值，C语言没有这样的区别。）

函数是C程序的构建块。每个函数本质上是一个自带声明和语句的小程序。可以利用函数把程序划分成小块，这样便于人们理解和修改程序。由于允许避免重复多次使用的代码，函数可以使编程不那么单调乏味。此外，函数可以复用：一个函数最初可能是某个程序的一部分，但可以将其用于其他程序中。

虽然我们的程序调用了库函数，但是到目前为止，都只是由一个函数构成的，即main函数。本章将集中讨论编写自己的函数。9.1节介绍定义和调用函数的方法，9.2节讨论函数的声明，以及它和函数定义的差异，9.3节介绍在函数间传递参数的方式。本章余下的部分则描述return语句（>9.4节）、与程序终止相关的问题（>9.5节）和递归（recursive）函数（>9.6节）。

9.1 函数的定义和调用

155

在介绍定义函数的规则之前，先来看3个简单的定义函数的程序。

9.1.1 程序：计算平均值

假设我们经常需要计算两个float型数值的平均值。C语言库没有“求平均值”函数，但是可以自己定义一个。下面就是这个函数的形式：

```
float average(float a, float b)
{
    return (a + b) / 2;
}
```

在函数开始处放置的单词float表示了average函数的返回类型（return type），也就是，每次调用函数返回数据的类型。**Q&A**标识符a和标识符b（即函数的形式参数（parameter））表示在调用average函数时提供的求平均值的两个数。每一个形式参数都必须有类型（正像每个变量有类型一样）；这里选择了float类型作为a和b的类型。（这看上去有点奇怪，但是单词float必须出现两次，一次为a而另一次为b。）函数的形式参数本质上是变量，其初始值在调用函数的时候才提供。

每个函数都有一个用大括号括起来的执行部分，称为函数体（body）。average函数的函数体由一条return语句构成。执行这条语句将会使函数“返回”到调用它的地方，表达式(a+b)/2的值将作为函数的返回值。

为了激活（即调用（call））函数，需要写出函数名及跟随其后的实际参数（argument）列表，例如，average(x, y)。实际参数用来给函数提供信息；在此例中，函数average需要知

道是要求哪两个数的平均值。调用average(x, y)的效果就是把变量x和y的值复制给形式参数a和b，然后执行average函数的函数体。顺便说一句，实际参数不一定要是变量；任何正确类型的表达式都可以，也就是说，既允许写成average(5.1, 8.9)，也允许写成average(x/2, y/3)。

我们把average函数的调用放在需要使用其返回值的地方。例如，为了计算并显示出x和y的平均值，可以写成

```
printf("Average: %g\n", average(x, y));
```

这条语句产生如下效果：

(1) 程序调用average函数，并且把变量x和y作为实际参数进行传递。

(2) average函数执行自己的return语句，返回x和y的平均值。

(3) printf函数显示出函数average的返回值。(average函数的返回值成为了函数printf的实际参数之一。)

注意，没有把average函数的返回值保存在任何地方；程序显示出这个值后就把它丢弃了。如果需要在稍后的程序中用到返回值，可以把这个返回值赋值给变量：

```
avg = average(x, y);
```

这条语句调用了average函数，然后把它的返回值存储在变量avg中。

现在把average函数放在一个完整的程序中来使用。下面的程序读取了3个数并且计算它们的平均值，其中，一次计算一对数的平均值：

```
Enter three numbers:3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

这个程序表明只要需要可以频繁调用函数。

```
average.c
/* Computes pairwise averages of three numbers */

#include <stdio.h>

float average(float a, float b)
{
    return (a + b) / 2;
}

main()
{
    float x, y, z;

    printf("Enter three numbers: ");
    scanf("%f%f%f", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
    printf("Average of %g and %g: %g\n", y, z, average(y, z));
    printf("Average of %g and %g: %g\n", x, z, average(x, z));

    return 0;
}
```

注意，这里把average函数的定义放在了main函数的前面。在9.2节还将看到把average函数的定义放在main函数的后面的情况。目前，将简单显示它的安全性，并且把函数定义在main函数之前。

9.1.2 程序：显示倒数计数

不是每个函数都返回一个值。例如，进行输出操作的函数可能不需要返回任何值。为了指

示出不带返回值的函数，需要指明这类函数的返回类型是void。（在C语言中，单词void用作占位符，这更像是计算机手册上发现的信息“此页故意留白”。）思考下面的函数，这个函数用来显示出信息T minus n and counting，这里的n在调用函数时才可以获得值：

```
157 void print_count (int n)
{
    printf("T minus %d and counting\n", n);
```

函数print_count有一个形式参数n，参数的类型为int。此函数没有返回任何值，所以用void指明它的返回值类型，并且略掉了return语句。既然print_count函数没有返回值，那么不能使用调用average函数的方法来调用它。print_count函数的调用必须是语句，而不能是表达式：

```
print_count(i);
```

下面这个程序在循环内调用了10次print_count函数：

```
countdown.c
/* Prints a countdown */

#include <stdio.h>

void print_count(int n)
{
    printf("T minus %d and counting\n", n);
}

main()
{
    int i;

    for (i = 10; i > 0; --i)
        print_count(i);
    return 0;
}
```

最开始，变量i的值为10。当调用print_count函数开始后，就把i复制给n，所以变量n也得到了值10。作为结果，第一次调用print_count函数会显示出

```
T minus 10 and counting
```

随后，函数print_count返回到调用的地方，而这个地方恰好是for语句的循环体。for语句再从调用离开的地方重新开始，变量i自减变成9，并且判断i是否大于0。如果判断结果为真，那么就再次调用函数print_count，这次显示出

```
T minus 9 and counting
```

每次调用print_count函数，变量i的值都不同，所以print_count函数显示出来的信息也会不同。

9.1.3 程序：显示双关语（改进版）

一些函数根本没有形式参数。思考下面这个print_pun函数，它在每次调用时显示一条坏的双关语：

```
void print_pun(void)
{
    printf("To C, or not to C: that is the question.\n");
```

在圆括号中的单词void表明print_pun函数没有实际参数。（又一次使用void作为占位符，这意味着“这里没有任何东西”。）

为了调用不带实际参数的函数，需要写出函数名并且后面跟上一对圆括号：

```
print_pun();
```

即使没有实际参数也必须显示圆括号。

下面这个极小的程序测试了print_pun函数：

```
pun2.c
/* Prints a bad pun */

#include <stdio.h>

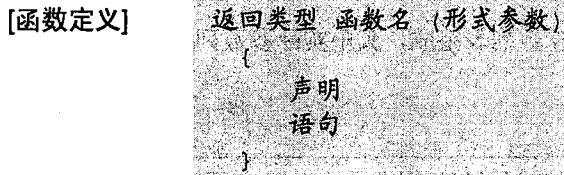
void print_pun(void)
{
    printf("To C, or not to C: that is the question.\n");
}

main()
{
    print_pun();
    return 0;
}
```

首先从main函数中的第一条语句开始执行此程序，这里碰巧第一句就调用了print_pun函数。接着，开始执行print_pun函数，也就是调用printf函数显示字符串。当printf函数返回时，print_pun函数也就返回了main函数。

9.1.4 函数定义

现在已经看过了一些例子，该来看看函数定义的通用格式了：



函数的“返回类型”是函数返回值的类型。下列规则用来管理返回类型：

- 函数无法返回数组，但是没有其他关于返回类型的限制。
- 如果忽略返回类型，那么会假定函数返回值的类型是int型。
- 指定返回类型是void型说明函数没有返回值。

为每个函数指定一个明显的返回类型是一个很好的方法。经典C缺少void的概念，所以，如果函数没有返回值，程序员经常会忽略掉返回类型：

```
print_count(int n)
{
    printf("T minus %d and counting\n", n);
```

这里建议大家避免采用上述方法，因为这样做无法立刻弄清楚函数是没有返回值还是实际上返回了int型值。

顺便提一句，一些程序员把返回类型放在函数名的上边：

```
float
average(float a, float b)
{
    return (a + b) / 2;
```

如果返回类型很长，比如unsigned long int类型，那么把返回类型单独放在一行是非常

有用的。

Q&A 函数名后边有一串形式参数列表。需要在每个形式参数的前面说明其类型；形式参数间用逗号进行分隔。如果函数没有形式参数，那么在圆括号内应该出现void。注意：即使是有形参具有相同数据类型的时候，也必须对每个形式参数分别进行类型说明。

```
Float average(float a, b)      /*** WRONG ***/
{
    return (a + b) / 2;
}
```

函数体可以包含声明和语句。例如，average函数可以写成

```
float average(float a, float b)
{
    return sum;      /* declaration */

    sum = a + b      /* statement */
    return sum / 2   /* statement */
}
```

函数体内声明的变量专属于此函数，其他函数不能对这些变量进行检查或修改。

函数体可以为空：

```
void print_pun(void)
{ }
```

160 程序开发过程中留下空函数体是有意义的；由于没有时间完成函数，所以为它预留下空间，以后可以回来编写它的函数体。

9.1.5 函数调用

函数调用由函数名和跟随其后的实际参数列表组成，其中实际参数列表用圆括号括起来：

```
average(x, y)
print_count(i)
print_pun()
```



如果丢失圆括号，那么将无法进行函数调用：

```
print_pun; /*** WRONG ***/
```

Q&A 这样的结果是合法的（虽然没有意义）表达式语句，而且看上去这语句是正确的，但是这条语句不起作用。一些编译器会发出一条类似“Code has no effect.”这样的警告。

void型的函数调用是语句，所以调用后边始终跟着分号：

```
print_count(i);
print_pun();
```

另一方面，非void型的函数调用是表达式，可以把调用产生的值存储在变量中，还可以进行测试、显示或者其他用途：

```
avg = average(x, y);
if (average(x, y) > 0) printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```

如果需要，可以一直丢掉非void型的函数返回值：

```
average(x, y); /* discards return value */
```

average函数的这个调用就是一个表达式语句（>4.5节）的例子：语句计算出值，但是不保存它。

当然，丢掉average函数的返回值是很奇怪的一件事，因为这正是在调用函数后的内容。

然而，有些情况下，丢掉函数的返回值是有意义的。例如，printf函数返回显示的字符的个数。在下面的调用后，变量num_chars将获得值9：

```
num_chars = printf ("Hi, Mom!\n");
```

因为可能对显示出的字符数量不感兴趣，所以通常会丢掉printf函数的返回值：

```
printf ("Hi, Mom!\n"); /* discards return value */
```

161

为了清楚地表示故意丢掉函数返回值，C语言允许在函数调用前加上(void)：

```
(void) printf ("Hi, Mom!\n");
```

需要做的工作就是把printf函数的返回值强制类型转换成(►7.5.3节)void类型。(在C语言中，“强制转换成void”是对“扔掉”的一种客气说法。)使用(void)可以使别人清楚编写者是故意扔掉返回值的，而不是忘记了。可惜的是，在C语言库中存在大量函数，它们的值被例行公事地丢掉了；在调用它们时使用(void)，所有人可能都会觉得很麻烦，所以本书中没有这样做。

9.1.6 程序：判定素数

为了弄清楚函数如何使程序变得更加容易理解，现在来编写一个程序来检查一个数是否是素数。这个程序将提示用户录入数，然后反映出信息说明此数是否是素数：

```
Enter a number: 34
Not prime
```

我们不是在main函数中加入素数判定的细节，而是定义一个单独的函数，此函数返回值为TRUE就表示它的形式参数是素数，返回FALSE就表示它的形式参数不是素数。给定数n后，is_prime函数把n除以从2到n的平方根之间的每一个数；如果余数永远为0，就知道n不是素数。

```
prime.c
/* Tests whether a number is prime */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

typedef int Bool;

Bool is_prime(int n)
{
    int divisor;

    if (n <= 1) return FALSE;
    for (divisor = 2; divisor * divisor <= n; divisor++)
        if (n % divisor == 0)
            return FALSE;
    return TRUE;
}

main()
{
    int n;

    printf("Enter a number: ");
    scanf("%d", &n);
    if (is_prime(n))
        printf("Prime\n");
    else
        printf("Not prime\n");
}
```

162

```
    return 0;
}
```

注意，main函数包含一个名为n的变量，而is_prime函数的形式参数也叫n。在一个函数中用作参数名或变量名的标识符可以在其他函数中复用。（10.1节更详细地讨论这个问题。）

正如此程序说明的那样，函数可以有多条return语句。当然，在给定的函数调用内只能执行一条return语句。

9.2 函数声明

9.1节中的程序始终小心地把函数的定义放置在调用此函数的位置的上面。事实上，C语言没有要求函数的定义必须放置在调用它之前。假设重新编排程序average.c（9.1节），使average函数的定义放置在main函数的定义之后：

```
# include < stdio. h>

main()
{
    float x, y, z;

    print f("Enter three numbers: ");
    scanf ( "%f%f%f", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
    printf("Average of %g and %g: %g\n", y, z, average (y, z)),
    printf("Average of %g and %g: %g\n", x, z, average(x, z)),

    return 0 ;
}

float average (float a, float b)
{
    return (a + b) / 2;
}
```

163

当遇到main函数中第一个average函数调用时，编译器没有任何关于average函数的信息：编译器不知道average函数有多少形式参数，形式参数的类型是什么，也不知道average函数的返回值是什么类型。但是，编译器没有产生错误信息，而是对average函数给出几个假设。它假设average函数返回int型的值（回顾9.1节的内容可以知道函数返回值的类型默认为int型。）；它假设给average函数传递了正确数量的实际参数；最后，它还假设在提升后实际参数（>9.3.1节）拥有正确的类型。因为关于average函数的这些假设有些是错误的，所以程序无法工作。

为了避免定义前调用这类问题的发生，一种方法是安排程序，使每个函数的定义都在此函数调用之前进行。可惜的是，这类安排不总是存在的，而且即使真的做了这类安排，也会因为按照不自然的顺序放置函数定义，使程序难以阅读。

幸运的是，C语言提供了一种更好的解决办法：在调用前声明（declare）每个函数。函数声明使得编译器对函数进行概要浏览，而函数的完整定义稍后再出现。函数声明类似于函数定义的第一行，不同之处是在其结尾处有分号：

| | |
|--------|-----------------|
| [函数声明] | 返回类型 函数名 (形式参数) |
|--------|-----------------|

无需多言，**Q&A** 函数的声明必须与函数的定义一致。

下面是为average函数添加了声明后程序的样子：

```
# include < stdio. h>
```

```

float average ( float a, float b) ; /* DECLARATION */

main()
{
    float x, y, z ;
    printf ("Enter three numbers: " ) ;
    scanf ( "%f%f%f", &x, &y, &z) ;
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
    printf("Average of %g and %g: %g\n", y, z, average(y, z)),
    printf("Average of %g and %g: %g\n", x, z, average(x, z));

    return 0;
}

float average ( float a, float b) /* DEFINITION */
{
    return (a + b) / 2;
}

```

164

为了与经典C的函数声明相区别，**Q&A**我们把正在讨论的这类函数声明称为**函数原型**（function prototype）。原型为如何调用函数提供了一个完整的描述：提供了多少实际参数，这些参数应该是什么类型，以及返回的结果是什么类型。

顺便提一句，函数原型不需要说明函数形式参数的名字，只要显示它们的类型就可以了：

```
float average(float, float);
```

然而，通常最好是不要忽略形式参数的名字，因为这些名字可以注释每个形式参数的目的，并且提醒程序员在函数调用时有关实际参数出现时必须依据的次序。

9.3 实际参数

复习一下形式参数和实际参数之间的差异。**形式参数**（parameter）出现在函数定义中，它们以假名字来表示函数调用时提供的值；**实际参数**（argument）是出现在函数调用中的表达式。在形式参数和实际参数的差异不是很重要的时候，有时将会用参数表示两者中的任意一个。

在C语言中，实际参数是通过值传递的：调用函数时，计算出每个实际参数的值并且把它赋值给相应的形式参数。在函数执行过程中，对形式参数的改变不会影响实际参数的值。从效果上来说，每个形式参数的行为好像是把变量初始化成与之匹配的实际参数的值。

实际参数按值传递既有利也有弊。既然形式参数的修改不会影响到相应的实际参数，那么可以把形式参数作为函数内的变量来使用，因此可以减少真正需要的变量的数量。思考下面这个函数，此函数用来计算数x的n次幂：

```

int power(int x, int n)
{
    int i, result = 1;

    for (i = 1; i <= n; i++)
        result = result * x;
    return result
}

```

因为n只是原始指数的副本，所以可以在函数体内修改它，因此就不需要使用变量i了：

```

int power(int x, int n)
{
    int result = 1;

    while (n-- > 0)
        result = result * x;
}

```

165

```
    return result;
}
```

可惜的是，C语言关于实际参数按值传递的要求使它很难编写确定的函数类型。例如，假设我们需要一个函数，它将把float型的值分解成整数部分和小数部分。因为函数无法返回两个数，所以可以尝试把两个变量传递给函数并且修改它们：

```
void decompose(float x, int int_part, float frac_part)
{
    int_part = (int) x, /* drops the fractional part of x */
    frac_part = x - int_part;
}
```

假设采用下面的方法调用这个函数：

```
decompose(3.14159, i, f);
```

在调用开始，程序把3.14159复制给x，把i的值复制给int_part，而且把f的值复制给frac_part。然后，decompose函数内的语句把3赋值给int_part而把.14159赋值给frac_part，接着函数返回。可惜的是，变量i和f不会因为赋值给int_part和frac_part而受到影响，所以它们在函数调用前后的值是完全一样的。正如在11.4节将会看到的那样，稍做一点额外的工作就可以使decompose函数工作。但是，我们将首先需要介绍更多C语言的特性。

9.3.1 实际参数的转换

C语言允许在实际参数的类型与形式参数的类型不匹配的情况下进行函数调用。管理如何转换实际参数的规则与编译器是否在调用前遇到函数（或者函数的完整定义）的原型有关。

- 编译器在调用前遇到原型。就像使用赋值一样，每个实际参数的值被隐式地转换成相应形式参数的类型。例如，如果把int类型的实际参数传递给期望得到float型数据的函数，那么会自动把实际参数转换成float类型。
- 编译器在调用前没有遇到原型。编译器执行默认的实际参数提升：(1) 把float型的实际参数转换成double类型，(2) 执行整数的提升（即把char型和short型的实际参数转换成int型）。

166



默认的实际参数提升可能无法产生期望的结果。思考下面的例子：

```
main()
{
    int i;

    printf("Enter number to be squared: ");
    scanf("%d", &i);
    printf("The answer is %g\n", square(i)); /* WRONG */

    return 0;
}

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

在调用square函数时，编译器没有遇到原型，所以它不知道square函数期望有double类型的实际参数。取而代之的，编译器在变量i上执行了没有效果的默认的实际参数提升。因为square函数期望有double类型的实际参数，但是却获得了int型值，所以square函数将产生无效的结果。通过在调用前声明square函数或者把变量i强制转换为正确的类型的方法，都可以解决这个问题：

```
printf("The answer is %g\n", square((double) i));
```

默认的实际参数提升不会总获得期望的效果这一事实使我们始终在调用函数前声明函数变得更加必要。

9.3.2 数组型实际参数

数组经常被用作实际参数。**Q&A**当形式参数是一维数组时，可以（而且是通常情况下）不说明数组的长度：

```
int f(int a[])
{
    ...
}
```

实际参数可以是任何一维数组，且数组元素拥有正确的类型。存在一个问题：f函数如何知道数组是多长呢？可惜的是，C语言没有为函数提供任何简便的方法来确定传递给它的数组的长度。但是，如果函数需要，必须把长度作为额外的实际参数提供出来。

167



虽然可以用运算符`sizeof`计算出数组变量的长度，但是它无法给出关于数组型形式参数的正确答案：

```
int f(int a[])
{
    int len = sizeof(a) / sizeof(a[0]); /* *** WRONG *** */
    ...
}
```

12.3节解释了原因。

下面的函数说明了一维数组型实际参数的用法。当给出具有`int`型值的数组`a`时，`sum_array`函数返回数组`a`中元素的和。因为`sum_array`函数需要知道数组`a`的长度，所以必须把长度作为第二个实际参数提供出来。

```
int sum_array(int a[], int n)
{
    int i, sum = 0;

    for (i = 0; i < n; i++)
        sum += a[i];

    return sum;
}
```

`sum_array`函数的原型有下列形式：

```
int sum_array(int a[], int n);
```

通常情况下，如果愿意可以忽略形式参数的名字：

```
int sum_array(int [], int);
```

在调用`sum_array`函数时，第一个参数是数组的名字，而第二个参数是这个数组的长度。例如：

```
#define LEN 100

main()
{
    int b[LEN], total;
    ...
    total = sum_array(b, LEN);
    ...
}
```

注意，在把数组名传递给函数时，不要在数组名的后边放置方括号：

168 total = sum_array(b[], LEN); /* *** WRONG *** /

一个关于数组型实际参数的重要论点：函数无法检测通过传递获得了正确的数组长度。人们可以利用这个事实，方法是告诉函数数组比实际小得多。假设，虽然数组b可以拥有100个元素，但是实际仅存储了50个元素。通过书写下列语句可以对数组的前50个元素进行求和：

total = sum_array(b, 50); /* sums first 50 elements */

sum_array函数将忽略另外50个元素。（当然，sum_array函数甚至不知道另外50个元素的存在！）



注意不要通知函数数组型实际参数要比实际的大：

total = sum_array(b, 150); /* *** WRONG *** /

在这个例子中，sum_array函数将超出数组的末尾；结果是，total将包含50个不存在的数组元素的值。

当形式参数是多维数组时，只能忽略第一维的长度。**Q&A**例如，修改sum_array函数使得a是一个二维数组，虽然不需要指出数组a中行的数量，但是必须说明数组a中列的数量：

```
#define LEN 10

int sum_array(int a[][LEN], int n)
{
    int i, j, sum = 0;

    for (i = 0; i < n; i++)
        for (j = 0; j < LEN; j++)
            sum += a[i][j];

    return sum;
}
```

不能传递令人困扰的具有任意列数的多维数组。幸运的是，我们经常可以通过使用指针数组（>13.7节）的方式处理这种困难。

9.4 return语句

非void的函数必须使用return语句来说明将要返回的值。return语句有如下格式：

169 [return语句] return 表达式;

表达式经常只是常量或变量：

```
return 0;
return status;
```

也可能是更加复杂的表达式。在return语句的表达式中看到条件运算符是很平常的：

```
return i > j ? i : j;
```

如果return语句中表达式的类型和函数的返回类型不匹配，那么系统将会把表达式的类型隐式转换成返回类型。例如，如果声明函数返回int型值，但是return语句包含float型表达式，那么系统将会把表达式的值转换成int型。

如果没有给出表达式，return语句可以出现在返回类型为void的函数中：

```
return; /* return in a void function */
```

（如果把表达式放置在上述这种return语句中将会获得一个编译时错误。）下面的例子中，在给出负的实际参数时，return语句会导致函数立刻返回：

```
void print_int(int i)
{
    if (I < 0) return;
    printf("%d", i);
}
```

在void函数末尾的return语句不会造成任何损害：

```
void print_pun(void)
{
    printf("To C, or not to C: that is the question.\n");
    return; /* OK, but not needed */
}
```

但是，不是必须使用return语句，因为在执行最后一条语句后函数将自动返回。

如果非void函数要永远达到函数体的末尾，那么返回的值是未定义的。不需要多说，这种实践是不推荐用的。如果一些编译器检查出非void函数有“离开”函数体末尾的可能性，那么它们将产生诸如“Function should return a value”这样的消息。

9.5 程序终止

既然main是函数，那么它必须有返回类型。前面从未说明过main函数的返回类型，这意味着默认情况下它的返回类型是int型。如果选择下列方式可以使返回类型成为显式的：

170

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

main函数返回的值是状态码，在某些操作系统中程序终止时可以检测到状态码。**Q&A**如果程序正常终止，main函数应该返回0；为了说明异常终止，main函数应该返回非0的值。（实际上，这一返回值也可以用于其他目的。）即使不打算使用状态码，确信每个C程序都返回状态码也是一个很好的实践，因为某些运行程序的人可能稍后再决定测试状态码。

exit 函数

在main函数中执行return语句是终止程序的一种方法，另一种方法是调用exit函数，此函数属于<stdlib.h>。传递给exit函数的实际参数和main函数的返回值具有相同的含义：两者都说明程序终止时的状态。为了说明正常终止，传递0：

```
exit(0); /* normal termination */
```

因为0是模糊的位，所以C语言允许用传递EXIT_SUCCESS来代替（效果是相同的）：

```
exit(EXIT_SUCCESS); /* normal termination */
```

传递EXIT_FAILURE说明异常终止：

```
exit(EXIT_FAILURE); /* abnormal termination */
```

EXIT_SUCCESS 和 EXIT_FAILURE 都是定义在<stdlib.h> 中的宏。EXIT_SUCCESS 和 EXIT_FAILURE 的值都是由实现定义的；典型值分别是0和1。

作为终止程序的方法，return语句和exit函数关系紧密。事实上，语句

```
return 表达式;
```

在main函数中等价于

```
exit(表达式);
```

return语句和exit函数之间的差异是，不仅是main函数，任何函数都可以调用exit函数。一些程序员专门使用exit函数以便于模式匹配程序可以很容易地定位程序中全部的退出点。

171

9.6 递归函数

如果函数调用它本身，那么此函数就是递归的（recursive）。例如，利用公式 $n!=n\times(n-1)!$ ，下面的函数可以递归地计算出 $n!$ 的结果：

```
int fact(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * fact(n-1);
}
```

一些编程语言极度地依赖递归，而另一些编程语言甚至不允许使用递归。C语言介于中间：它允许递归，但是大多数C程序员并不经常使用递归。

为了观察递归的工作情况，一起来跟踪下面语句的执行：

```
i = fact(3);
```

下面是实现过程：

```
fact(3)发现3不是小于或等于1的，所以fact(3)调用
fact(2)，此函数发现2不是小于或等于1的，所以fact(2)调用
fact(1)，此函数发现1是小于或等于1的，所以fact(1)返回1，从而导致
fact(2)返回 $2\times1=2$ ，从而导致
fact(3)返回 $3\times2=6$ 。
```

注意，在fact函数最终传递1之前，未完成的fact函数的调用是如何“堆积”的。在最终传递1的那一点上，fact函数的先前的调用开始逐个地“解开”，直到fact(3)的原始调用最终返回结果6为止。

下面是递归的另一个示例：利用公式 $x^n=x\times x^{n-1}$ ，函数计算出 x^n 的值。

```
int power(int x, int n)
{
    if (n == 0)
        return 1;
    else
        return x * power(x, n-1);
}
```

调用power(5, 3)将会按照如下方式执行：

```
power(5, 3)发现3不等于0，所以power(5, 3)调用
power(5, 2)，此函数发现2不等于0，所以power(5, 2)调用
power(5, 1)，此函数发现1不等于0，所以power(5, 1)调用
power(5, 0)，此函数发现0是等于0，所以返回1，从而导致
power(5, 1)返回 $5\times1=5$ ，从而导致
power(5, 2)返回 $5\times5=25$ ，从而导致
power(5, 3)返回 $5\times25=125$ 。
```

顺便说一句，通过把条件表达式放入return语句中的方法可以精简power函数：

```
int power(int x, int n)
{
    return n == 0 ? 1 : x * power(x, n-1);
}
```

一旦被调用，fact函数和power函数都仔细地测试“终止条件”。调用fact函数时，它会立刻检查参数是否小于或等于1。调用power函数时，它先检查第二个参数是否等于0。为了防止无限递归，所有递归函数都需要某些类型的终止条件。

9.6.1 快速排序算法

此处读者可能会好奇为什么要为递归费心；毕竟无论是fact函数还是power函数都不是真的需要递归。好的，这里得出论点。没有函数会对递归情况进行多次，因为每个函数调用它自身只有一次。递归对要求函数调用自身两次或多次的复杂算法非常有帮助。

实际上，递归经常作为分治法（divide-and-conquer）技术的结果自然地出现。这种称为分治法的算法设计技术把一个大问题划分成多个较小的问题，然后采用相同的算法分别解决这些小问题。分治法的经典示例就是流行的排序算法——快速排序（quicksort）。快速排序算法的操作如下（为了简化，假设要排序的数组的下标从1到n）：

(1) 选择数组元素 e （作为“分割元素”），然后重新排列数组使得元素从1一直到 $i-1$ 都是小于或等于元素 e 的，元素 i 包含 e ，而元素从 $i+1$ 一直到 n 都是大于或等于 e 的。

(2) 通过递归地采用快速排序方法，对从1到 $i-1$ 的元素进行排序。

(3) 通过递归地采用快速排序方法，对从 $i+1$ 到 n 的元素进行排序。

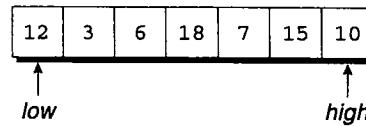
执行完第1步后，元素 e 处在正确的位置上。因为 e 左侧的元素全部都是小于或等于 e 的，所以一旦第2步对这些元素进行排序，那么这些小于或等于 e 的元素也将会处在正确的位置上；类似的理由也可以应用于 e 右侧的元素。

显然快速排序中的第1步是很关键的。有许多种方法可以用来分割数组，有些方法比其他的方法好。下面将采用的方法是很容易理解的，但是它不是特别有效。首先将概括地描述分割算法，稍后将会把这种算法翻译成C代码。

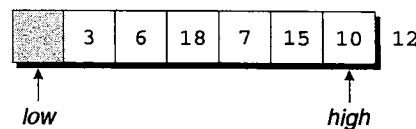
该算法依赖于两个名为 low 和 $high$ 的标记，这两个标记用来跟踪数组内的位置。开始， low 指向数组中的第一个元素，而 $high$ 指向末尾元素。首先把第一个元素（分割元素）复制给其他地方的一个临时存储单元，从而在数组中留出一个“空位”。接下来，从右向左移动 $high$ ，直到 $high$ 指向小于分割元素的数时停止。然后把这个数复制给 low 指向的空位，这将产生一个新的空位（ $high$ 指向的）。现在从左向右移动 low ，寻找大于分割元素的数。在找到时，把这个找到的数复制给 $high$ 指向的空位。重复执行此过程，交替操作 low 和 $high$ 直到两者在数组中间的某处相遇时停止。此时，两个标记都将指向空位；只要把分割元素复制给空位就够了。下面的图演示了这个过程：

173

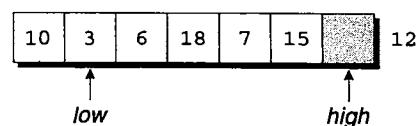
首先，假设数组包含7个元素。 low 指向第一个元素； $high$ 指向最后一个元素。



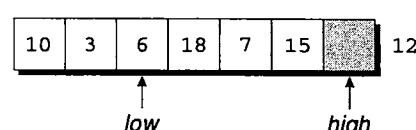
第一个元素12是分割元素。把它复制到某个位置，留出数组开始处的空位。



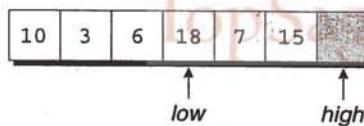
现在把 low 和 $high$ 指向的元素进行比较。因为10小于12，它是处在数组的错误一侧的，所以把10移动到空位，并且把 low 向右移动一位。



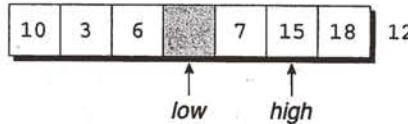
low 指向的数3是小于12的，因此不需要进行移动。只是把 low 向右移动一位。



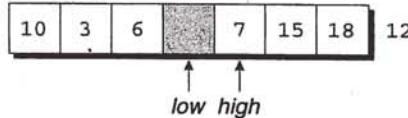
因为6也是小于12的，所以再把low向右移动一位。



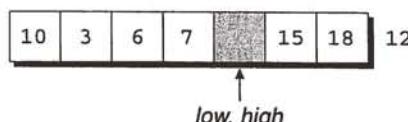
现在low指向的数18是大于12的，因此18超出范围。在把数18移动到空位后，high向左移动一位。



high指向的数15是大于12的，因此不需要进行移动。只是把high向左移动一位然后继续。



174 high指向的数7超出范围。在把7移动到空位后，把low向右移动一位。



low和high现在是相等的，所以把分割元素移到空位上。



此时我们已经实现了目标：分割元素左侧的所有元素都小于或等于12，而其右侧的所有元素都大于或等于12。既然已经分割了数组，那么可以使用快速排序法对数组的前4个元素（10、3、6和7）和后2个元素（15和18）进行递归快速排序了。

9.6.2 程序：快速排序

先来开发一个名为quicksort的递归函数，此函数采用快速排序算法对数组元素进行排序。为了测试函数，将由main函数往数组中读入10个元素，调用quicksort函数对数组进行排序，然后显示数组元素：

```
Enter 10 numbers to be sorted: 9 16 47 82 4 66 12 3 25 51
In sorted order: 3 4 9 12 16 25 47 51 66 82
```

因为分割数组的代码有一点长，所以把这部分代码放置在名为split的独立的函数中。

```
qsrt.c
/* Sorts an array of integers using Quicksort algorithm */

#include <stdio.h>

#define N 10

void quicksort(int a[], int low, int high);
int split(int a[], int low, int high);

main()
{
    int a[N], i;

    printf("Enter %d numbers to be sorted: ", N);
    for (i = 0; i < N; i++)
        scanf("%d", &a[i]);

    quicksort(a, 0, N - 1);

    printf("In sorted order: ");
}
```

```

for (i = 0; i < N; i++)
    printf("%d ", a[i]);
printf("\n");

return 0;
}

void quicksort(int a[], int low, int high)
{
    int middle;

    if (low >= high) return;
    middle = split(a, low, high);
    quicksort(a, low, middle - 1);
    quicksort(a, middle + 1, high);
}

int split(int a[], int low, int high)
{
    int part_element = a[low];

    for (;;) {
        while (low < high && part_element <= a[high])
            high--;
        if (low >= high) break;
        a[low++] = a[high];

        while (low < high && a[low] <= part_element)
            low++;
        if (low >= high) break;
        a[high--) = a[low];
    }

    a[high] = part_element;
    return high;
}

```

虽然此版本的快速排序可行，但是它不是最好的。有许多方法可以用来改进这个程序的性能，包括：

- **改进分割算法。**上面介绍的方法不是最有效的。我们不再选择数组中的第一个元素作为分割元素，较好的方法是取第一个元素、中间元素和最后一个元素的中间值。分割过程本身也可以加速。特别是，在两个while循环中避免测试`low < high`是可能的。
- **采用不同的方法进行小数组排序。**不再递归地使用快速排序法用一个元素全部下至数组尾，针对小数组更好的方法是（比方说，拥有的元素数量少于25个的数组）采用较为简单的方法。
- **使得快速排序非递归。**虽然快速排序本质上是递归算法，并且递归格式的快速排序是最容易理解的，但是实际上若去掉递归会更有效率。

关于改进快速排序法的细节，可以参考算法设计方面的书，如Robert Sedgewick写的*Algorithms in C* (Reading, Mass.: Addison-Wesley, 1990)。

问与答

问：一些C语言书出现了采用了不同于形式参数和实际参数的术语，是否有标准术语？(p.110)

答：正如对待C语言的许多其他概念一样，没有通用的术语标准，但是C标准采用形式参数和实际参数。

下面的表格应该对翻译有帮助：

| 本 书 | 其 他 书 |
|----------|------------------------------------------------------------------------|
| 形式参数（形参） | parameter |
| 实际参数（实参） | formal argument, formal parameter actual argument, actual parameter |

请记住，在不会产生混乱的情况下，有时会故意模糊两个术语的差异，采用参数表示两者中的任意一个。

问：在程序的形式参数列表的后边，我们遇见过把形式参数的类型用单独的声明进行说明：

```
float average(a, b)
float a, b;
{
    return (a + b)/2;
}
```

这种实践是合法的吗？(p.114)

答：这种定义函数的方法来自于经典C，所以可能会在较早的书籍和程序中遇到这种方法。标准C支持这种格式以便于可以继续编译旧的程序。然而，由于下面两个原因本书避免在新程序中采用此种方法。首先，用经典C的方法定义的函数不会遭受和新格式函数一样程度的错误检查。当函数是采用经典方法定义时，并且没有显示原型，编译器将不检测是否是用正确数量的实际参数调用函数的，也不检测实际参数是否具有正确的类型。相反，编译器会执行默认的实际参数提升（>9.3.1节）其次，C标准提到经典格式是“逐渐消亡的”，这意味着不鼓励此种用法，并且这种格式最终可能会从C语言中消失。

问：一些编程语言允许过程和函数互相嵌套。C语言是否允许函数定义嵌套呢？

177

答：不允许。C语言不允许一个函数的定义出现在另一个函数体中。这个限制可以使编译器简单化。

*问：为什么编译器允许函数名不跟着圆括号？(p.114)

答：在下一章中将会看到，编译器把不跟圆括号的函数名看成是指向函数的指针。指向函数的指针有合法的应用，所以编译器不能自动假定函数名不带圆括号是错误的。

*问：有些问题困扰着我。在函数调用f(a, b)中，编译器如何知道逗号是标点符号还是运算符呢？

答：由此引出函数调用中的实际参数不能是任意的表达式，而必须是“赋值表达式”，且这类表达式不能用逗号作为运算符，除非逗号是在圆括号中。换句话说，在函数调用f(a, b)中，逗号是标点符号；而在f((a, b))中，逗号是运算符。

问：函数原型中的形式参数的名字是否需要和后面函数定义中给出的名字相匹配？(p.116)

答：不需要。一些程序员利用给定原型中参数一个长名字的特性，然后在实际定义中使用较短的名字。或者，说法语的程序员可以在函数原型中使用英文名字，然后在函数定义中切换成更为熟悉的法语名字。

问：我始终不明白为什么还要麻烦的函数原型。如果只是把所有函数的定义放置在main函数的前面，就没有问题了吗？

答：错。首先，你是假设只有main函数调用其他函数，当然这是不切实际的。实际上，某些函数将会相互调用。如果把所有的函数定义放在main的上面，就必须仔细斟酌它们之间的顺序，因为调用未定义的函数可能会导致大问题。

但是，不仅如此。假设有两个函数相互调用（这可不是刻意找麻烦）。无论先定义哪个函数，都将结束于对未定义的函数的调用。

但是，还有更麻烦的！一旦程序达到一定的规模，在一个文件中放置所有的函数是不可行的。当遇到这种情况时，就需要函数原型告诉编译器函数在其他文件中定义的函数。

问：已经看到函数声明忽略掉形式参数的全部信息：

```
float average();
```

这种习惯是合法的吗？(p.117)

答：是的。这种声明提示编译器average函数返回float型的值，但不提供关于参数数量和类型的任何

信息。（留下空的圆括号不意味着average函数没有参数。）

在经典C中，这是唯一允许的一种声明格式；采用的函数原型格式是包含参数信息的，这是标准C的新特性。旧式的函数声明虽然还允许使用，但现在已逐渐废弃了。本书将专门采用函数原型。

178

问：把函数的声明放在另一个函数体内是否合法？

答：是合法的。下面是一个示例：

```
main()
{
    float average(float a, float b);
    ...
}
```

average函数的声明只有在main函数体内是有效的；如果其他函数需要调用average函数，那么它们每一个都需要声明它。

这种做法的好处是便于读者清楚函数间的调用关系。（在这个例子中，看到main函数将会调用average函数。）另一方面，如果几个函数需要调用同一个函数，这可能是件麻烦事。最糟糕的情况是，在程序修改过程中试图添加或移动声明可能会很麻烦。基于这些原因，本书将始终把函数声明放在函数体外。

问：如果几个函数具有相同的返回类型，能否把它们的声明合并？例如，既然print_pun函数和print_count函数都具有void型的返回类型，那么下面的声明合法吗？

```
void print_pun(void), print_count(int n);
```

答：合法。事实上，C语言甚至允许把函数声明和变量声明一起合并：

```
float x, y, average(float a, float b);
```

但是，此种方式的合并声明通常不是个好方法；它可能会使得程序有点混乱。

问：如果指定一维数组型形式参数的长度，会发生什么？(p.119)

答：编译器会忽略长度值。思考下面的例子：

```
float inner_product(float v[3], float w[3]);
```

除了注明inner_product函数的参数应该是长度为3的数组以外，指定长度并不会带来什么其他好处。编译器不会检查参数实际上的长度是否为3，所以没有额外的安全性考虑。事实上，在实际可以传递任意长度的数组时，这种做法会产生误导，因为这种写法暗示只能把长度为3的数组传递给inner_product函数。

*问：为什么可以留着数组中第一维的参数不进行说明，但是其他维数必须说明呢？(p.120)

答：首先，需要知道C语言是如何传递数组的。就像12.2节解释的那样，在把数组传递给函数时，是指向数组第一个元素的指针给了函数。

其次，需要知道下标运算符是如何工作的。假设a是要传给函数的一维数组。在书写语句

```
a[i] = 0;
```

时，编译器计算出a[i]的地址，方法是把i乘以每个元素的大小，并且把乘积的结果加上数组a表示的地址（传递给函数的指针）。这个计算过程没有依靠数组a的长度，这说明了为什么可以在定义函数时忽略数组长度。

那么多维数组怎么样呢？回顾一下就知道，C语言是按照行主序存储数组的，即首先存储第0行的元素，然后是第1行的元素，依此类推。假设a是二维数组型的形式参数，并且写了语句

```
a[i][j] = 0;
```

编译器产生指令执行如下：(1)把i乘以数组a中每行的大小；(2)把乘积的结果加上数组a表示的地址；(3)把j乘以数组a中每个元素的大小；(4)把乘积的结果加上第二步计算出的地址。为了产生这些指令，编译器必须知道a数组中每一行的大小，行的大小由列数决定。底线：程序员必须声明数组a拥有的列的数量。

179

问：为什么一些程序员把return语句中的表达式用圆括号括起来？

答：虽然不要求，但是Kernighan和Ritchie写的*The C Programming Language*（第1版）一直在return语句中有圆括号。程序员（和后续书的作者）也采用K&R的这种习惯。因为这种写法不是必须的，而且对可读性没有任何帮助，所以本书不使用这些圆括号。（Kernighan和Ritchie显然也同意：在*The C Programming Language*（第2版）中，return语句就没有圆括号了。）

问：如何测试main的返回值来判断程序是否正常终止？(p.121)

答：这依赖于使用的操作系统。许多操作系统允许在“批处理文件”或“外壳文件”内测试main的返回值，这类文件包含可以运行几个程序的命令。例如，

```
if errorlevel 1...
```

在DOS批处理文件中，上面这行命令测试最后程序是否以大于或等于1的状态码终止。

在UNIX系统中，每种外壳都有自己测试状态码的方法。在Bourne外壳中，变量\$?包含最后程序运行的状态。C外壳也有类似的变量，但是名字是\$status。

180 问：在编译main函数时，为什么编译器会产生“Function should return a value”这样的警告？

答：尽管main函数有int作为返回类型，但编译器已经注意到main函数没有return语句。在main的末尾放置语句

```
return 0;
```

将保证编译顺利通过。顺便说一下，即使编译器不反对没有return语句，这也是一种好习惯。

问：对于前一个问题：为什么不把main函数的返回类型定义为void型呢？

答：虽然这种做法非常普遍，但是根据C标准却是非法的。即使它不是非法的，这种做法也不是个好主意，因为它假设没有人会始终测试程序终止时的状态。

问：请参考前一个问题：为什么不把main的返回类型定义成void型呢？

答：虽然这种实践非常普遍，但是根据C标准这种做法是不合法的。即使这样做不是非法的，也不是一种好方法，因为这样做就意味着永远不会在终止前检测程序状态。

问：如果函数f1调用函数f2，而函数f2稍后又调用函数f1，这样合法吗？

答：是合法的。这是一种间接递归的形式，即函数f1的调用导致其他调用。（但是必须确保函数f1和函数f2最终都可以终止！）

练习

9.1节

- 下列计算三角形面积的函数有两处错误。找出这些错误，并且说明修改它们的方法。（提示：公式没有错误。）

```
float triangle_area(float base, height)
float product;
{
    product = base * height;
    return (product / 2);
}
```

- 编写函数check(x, y, n)：如果x和y都落在0到n-1的闭区间内，那么使得函数check返回1。否则，函数应该返回0。假设x、y和n都是int类型。
- 编写函数gcd(m, n)用来计算整数m和n的最大公约数。（第6章中的练习2描述了计算最大公约数的Euclid算法。）
- 编写函数day_of_year(month, day, year)，使得函数返回某month某day是year这一年中的第几天（1和366之间的整数）。
- 编写函数num_digits(n)，使得函数返回正整数n中数字的个数。提示：为了确定n中的数字的个数，

把这个数反复除以10。当n达到0时，除法的次数表明了n最初拥有的数字的个数。

6. 编写函数digit(n, k)，使得函数返回正整数n中第k个数字（从右边算起）。例如，digit(829, 1)返回9，digit(829, 2)返回2，而digit(829, 3)则返回8。如果k大于n所含有的数字的个数，那么函数返回-1。181

7. 假设函数f有下列定义：

```
int f(int a, int b) {...}
```

那么下列哪条语句是合法的？（假设i的类型为int而x的类型为float。）

- (a) i=f(83, 12);
- (b) x=f(83, 12);
- (c) i=f(3.15, 9.28);
- (d) x=f(3.15, 9.28);
- (e) f(83, 12);

9.2节

8. 对于返回为空且有一个float型形式参数的函数，下列哪个函数原型是有效的？

- (a) void f(float x);
- (b) void f(float);
- (c) void f(x);
- (d) f(float x);

9.3节

- *9. 下列程序的输出是什么？

```
#include < stdio.h>

void swap(int a, int b);

main()
{
    int x = 1, y = 2;

    swap (x, y);
    printf("x = %d, y = %d\n", x, y);
    return 0 ;
}

void swap(int a, int b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
}
```

10. 编写函数，使得函数返回下列值。（假设a和n是形式参数，其中a是有int型值的数组，而n则是数组的长度。）

- (a) 数组a中的最大元素。
- (b) 数组a中所有元素的平均值。
- (c) 数组a中正数元素的数量。

9.4节

11. 如果数组a的所有元素值都为0，那么假设下列函数返回TRUE；如果数组的所有元素都是非零的，则函数返回FALSE。可惜的是，此函数有错误。请找出错误并且说明修改它的方法。182

```
Bool has_zero(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        if (a[i] == 0)
```

```

        return TRUE;
    else
        return FALSE;
}

```

12. 下面的（不要弄混）函数用来找到三个数的中间数。重新编写函数，使得它只有一条return语句。

```

float median(float x, float y, float z)
{
    if (x <= y)
        if (y <= z) return y;
        else if (x <= z) return z;
        else return x;
    if (z <= y) return y;
    if (x <= z) return x;
    return z;
}

```

9.6节

13. 请采用精简power函数的方法，来简化fact函数。
 14. 请重新编写fact函数，使得编写后的函数不再递归。
 15. 编写递归版本的gcd函数（参见练习3）。有一种用于计算gcd(m, n)的策略：如果n为0，那么返回m；否则，递归地调用gcd函数，把n作为第一个实际参数进行传递，而把m%n作为第二个实际参数进行传递。

- *16. 思考下面这个“神秘的”函数：

```

void pb(int n)
{
    if (n != 0) {
        pb(n / 2);
        putchar('O' + n % 2);
    }
}

```

手动跟踪函数的执行。然后编写程序调用此函数，把用户录入的数传递给此函数。函数做了什么？

17. 编写程序，要求用户录入一串整数（把这串整数存储在数组中），然后通过调用selection_sort函数来排序这些整数。在给定n个元素的数组后，selection_sort函数必须做下列工作：
 (a) 搜索数组找出数组中最大的元素，然后把它移到数组的最后。
 (b) 递归地调用函数本身来对前n-1个数组元素进行排序。

程序结构

正如罗杰斯可能会说的那样：“没有像自由变量这样的东西。”

第9章已经介绍过函数，现在准备来面对程序包含多个函数时所产生的几个问题。本章的前两节讨论局部变量（local variable）和外部变量（external variable）之间的差异，10.3节考虑程序块（block）（含有声明的复合语句）问题，10.4节解决用于局部名、外部名和在程序块中声明的名字的作用域规则问题，10.5节介绍用来组织原型、函数定义、变量声明和程序其他部分的方法。

10.1 局部变量

我们把在函数体内声明的变量称为相对于函数的局部。在下面的函数中，`log`是局部变量：

```
int log2(int n)
{
    int log = 0; /* local variable */

    while (n > 1) {
        n /= 2;
        log++;
    }
    return log;
}
```

默认情况下，局部变量具有下列性质。

- **自动存储期限。** 变量的存储期限（storage duration）（或存储长度）是在变量存储有效期 内程序执行的部分。调用闭合函数时“自动”分配局部变量的存储单元，函数返回时收回分配，所以称这种变量具有自动的存储期限。在闭合函数返回时，局部变量并不保留值。当再次调用函数时，无法保证变量始终保留原有的值。
- **程序块作用域。** 变量的作用域是可以参考变量的程序文本的部分。局部变量拥有程序块 作用域：从变量声明的点开始一直到闭合函数体的末尾。因为局部变量的作用域不能延 伸到其所属函数之外，所以其他函数可以使用同名变量。

185

18.2节会详细地介绍上述这些内容和其他相关的概念。

在局部变量声明中放置单词`static`可以使变量从自动存储期限变为静态存储期限。因为具有静态存储期限的变量拥有永久的存储单元，所以在整个程序执行期间会保留变量的值。思考下面的函数：

```
void f(void)
{
    static int i;
    ...
}
```

Q&A因为局部变量*i*已经声明为`static`，所以在程序执行期间它占有同样的存储单元。在*f*返 回时，变量*i*不会丢失自身的值。

静态局部变量始终有程序块作用域，它对其他函数而言是不可见的。概括来说，静态变量是隐藏来自其他函数的数据的地方，但是它会为将来同一个函数的调用保留这些数据。

形式参数

形式参数拥有和局部变量一样的性质，即自动存储期限和程序块作用域。事实上，形式参数和局部变量唯一真正的区别是，在每次函数调用时对形式参数自动进行初始化（调用中通过赋值获得实际参数的值）。

10.2 外部变量

传递参数是给函数传送信息的一种方法。函数还可以通过外部变量（external variable）进行交流。这些外部变量是声明在任何函数体外的。

186

外部变量（有时称为全局变量）的性质不同于局部变量的性质：

- **静态存储期限。**就如同声明为static的局部变量一样，外部变量拥有静态存储期限。存储在外部变量中的值将永久保留下。
- **文件作用域。**外部变量拥有文件作用域：从变量声明的点开始一直到闭合文件的末尾。结果是，跟随在外部变量声明后的所有函数都可以访问它。

10.2.1 程序：用外部变量实现栈

为了说明外部变量的使用方法，一起来看看称为栈（stack）的数据结构。（栈是抽象的概念，它不是C语言的特性。大多数编程语言都可以实现栈。）像数组一样，栈可以存储具有相同数据类型的多个数据项。然而，栈中数据项的操作是十分受限制的：可以往栈中压入数据项（把数据项加上1结束作为“栈顶”），或者从栈中弹出数据项（从同样的末尾移走数据项）。由于显而易见的原因，经常把栈称为LIFO（后进先出）数据结构。禁止测试或修改不在栈顶的数据项。

C语言中实现栈的一种方法是把元素存储在数组中，我们称这个数组为contents。命名为top的一个整型变量用来标记栈顶的位置。栈为空时，top值为0。为了往栈中压入数据项，可以把数据项简单存储在contents中标记为top的位置上，然后自增top。弹出数据项则要求自减top，然后用它作为contents的索引取回弹出的数据项。

基于上述这些概要，这里有一段代码（不是完整的程序）为栈留出了变量并且提供一组函数来表示栈的操作。全部5个函数都需要访问变量top，而且其中2个函数还都需要访问contents，所以将把contents和top设为外部变量。

```
#define STACK_SIZE 100
#define TRUE 1
#define FALSE 0

typedef int Bool;

int contents[STACK_SIZE];           /* external */
int top = 0;                      /* external */

void make_empty (void)
{
    top = 0;
}

Bool is_empty (void)
{
    return top == 0;
}
```

187

```

Bool is_full (void)
{
    return top == STACK_SIZE;
}

void push(int i)
{
    if (is_full ())
        stack_overflow();
    else
        contents [top++] = i;
}

int pop (void)
{
    if (is_empty())
        stack_underflow ();
    else
        return contents [--top];
}

```

10.2.2 外部变量的利与弊

在多个函数必须共享一个变量时或者少数几个函数共享大量变量时，外部变量是很有用的。然而，在大多数情况下，通过形式参数进行函数交流比通过共享变量的方法更好。原因是：

- 在程序修改期间，如果改变外部变量（比方说改变它的类型），那么将需要检查同一文件中的每个函数，以确认该变化对函数的影响程度。
- 如果外部变量被赋了错误的值，那么它可能很难确定这个有错误值的函数。就好像是处理聚集很多人的晚会上的谋杀案很难有方法缩小嫌疑犯范围一样。
- 很难在其他程序中复用依赖于外部变量的函数。依赖外部变量的函数不是“独立的”。为了在另一个程序中使用该函数，将不得不带上任何此函数需要的外部变量。

许多程序员过于依赖外部变量。一个普遍的弊端是：在不同的函数中为不同的目的使用同样的外部变量。假设几个函数都需要变量（比如说变量*i*）来控制for语句。一些程序员不是在使用变量*i*的每个函数中都声明它，而是在程序的顶部声明它从而使得变量对所有函数都是可见的。这种方式不但不利于前面列出的几个原因而且还会产生误导；一些人在稍后阅读程序时可能认为变量的使用彼此关联，而实际并非如此。

使用外部变量时，要确保它们都拥有有意义的名字。（局部变量不是总需要有意义的名字的，因为经常很难为for循环中的控制变量考虑一个好名字。）如果你发现为外部变量使用的名字就像*i*和*temp*一样，那么就暗示着或许这些变量实际应该是局部变量。

188



把应该是局部变量的变量变为外部变量可能导致一些令人厌烦的错误。思考下面的例子，它假设显示一个由星号组成的10×10的图形：

```

int i;
void print_row(void)
{
    for (i = 1; i <= 10; i++)
        printf("*");
}

void print_matrix(void)
{
    for (i = i; i <= i0; i++) {
        print_row ();
        printf("\n");
    }
}

```

```

    }
}

```

print_matrix函数不是显示10行，而是只显示出1行。在第一次调用print_row函数后返回时，*i*的值将为11。然后，print_matrix函数中的for语句对变量*i*进行自增并且把它与10进行比较，这导致循环终止并且print_matrix函数返回。

10.2.3 程序：猜数

为了获得更多关于外部变量的经验，现在编写一个简单的游戏程序。这个程序产生一个1~100的随机数，用户尝试用尽可能少的次数猜出这个数。下面是程序运行时用户将会看到的内容：

```

Guess the secret number between 1 and 100.

A new number has been chosen.
Enter guess: 55
Too low; try again.
Enter guess: 65
Too high; try again.
Enter guess: 60
Too high; try again.
Enter guess: 58
You won in 4 guesses!

```

189

```

Play again? (Y/N) y
A new number has been chosen.
Enter guess: 78
Too high; try again.
Enter guess: 34
You won in 2 guesses!

```

```
Play again? (Y/N) n
```

这个程序需要实现几个任务：初始化随机数生成器，选择神秘数，以及与用户交互直到选择出正确数为止。如果编写独立的函数来处理每个任务，那么可能会得到下面的程序。

```

guess.c
/* Asks user to guess a hidden number */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 100

int secret_number;

void initialize_number_generator(void);
void choose_new_secret_number(void);
void read_guesses(void);

main()
{
    char command;

    printf("Guess the secret number between 1 and %d.\n\n",
           MAX_NUMBER);
    initialize_number_generator();
    do {
        choose_new_secret_number();
        printf("A new number has been chosen.\n");

```

```

read_guesses();
printf("Play again? (Y/N) ");
scanf(" %c", &command);
printf("\n");
} while (command == 'y' || command == 'Y');
return 0;
}

/*****************
 * initialize_number_generator: Initializes the random *
 *                               number generator using *
 *                               the time of day. *
 *****************/
void initialize_number_generator(void)
{
    srand((unsigned) time(NULL));
}

/*****************
 * choose_new_secret_number: Randomly selects a number *
 *                           between 1 and MAX_NUMBER and *
 *                           stores it in secret_number. *
 *****************/
void choose_new_secret_number(void)
{
    secret_number = rand() % MAX_NUMBER + 1;
}

/*****************
 * read_guesses: Repeatedly reads user guesses and tells *
 *                 the user whether each guess is too low, *
 *                 too high, or correct. When the guess is *
 *                 correct, prints the total number of *
 *                 guesses and returns. *
 *****************/
void read_guesses(void)
{
    int guess, num_guesses = 0;

    for (;;) {
        num_guesses++;
        printf("Enter guess: ");
        scanf("%d", &guess);
        if (guess == secret_number) {
            printf("You won in %d guesses!\n\n", num_guesses);
            return;
        } else if (guess < secret_number)
            printf("Too low; try again.\n");
        else
            printf("Too high; try again.\n");
    }
}

```

190

对于随机数的生成, `guess.c`程序与`time`(>26.3.1节)、`srand`(>26.2.3节)和`rand`(>26.2.3节)这三个函数有关, 这些函数第一次是用在`deal.c`程序(8.2节)中。这次将规划`rand`函数的返回值使其落在1~MAX_NUMBER范围内。

虽然`guess.c`程序工作正常, 但是它依赖于外部变量。把变量`secret_number`外部化以便`choose_new_secret_number`函数和`read_guesses`函数都可以访问它。如果对`choose_new_secret_number`函数和`read_guesses`函数稍做改动, 应该能把变量`secret_number`移到`main`函数中。现在我们要修改`choose_new_secret_number`函数以便函数返回新值, 而且重写`read_guesses`函数以便变量`secret_number`可以作为参数传递给它。

下面是新程序，修改的部分用粗体标注出来：

guess2.c

```

/* Asks user to guess a hidden number */

191
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 100

void initialize_number_generator(void);
int new_secret_number(void);
void read_guesses(int secret_number);

main()
{
    char command;
    int secret_number;

    printf("Guess the secret number between 1 and %d.\n\n",
           MAX_NUMBER);
    initialize_number_generator();
    do {
        secret_number = new_secret_number();
        printf("A new number has been chosen.\n");
        read_guesses(secret_number);
        printf("Play again? (Y/N) ");
        scanf(" %c", &command);
        printf("\n");
    } while (command == 'y' || command == 'Y');
    return 0;
}

/*****************
 * initialize_number_generator: Initializes the random      *
 *                               number generator using      *
 *                               the time of day.          *
 *****************/
void initialize_number_generator(void)
{
    srand((unsigned) time(NULL));
}

/*****************
 * new_secret_number: Returns a randomly chosen number      *
 *                     between 1 and MAX_NUMBER.            *
 *****************/
int new_secret_number(void)
{
    return rand() % MAX_NUMBER + 1;
}

/*****************
 * read_guesses: Repeatedly reads user guesses and tells   *
 *                 the user whether each guess is too low,    *
 *                 too high, or correct. When the guess is    *
 *                 correct, prints the total number of       *
 *                 guesses and returns.                      *
 *****************/
void read_guesses(int secret_number)
{
    int guess, num_guesses = 0;
}

```

```

for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
        printf("You won in %d guesses!\n\n", num_guesses);
        return;
    } else if (guess < secret_number)
        printf("Too low; try again.\n");
    else
        printf("Too high; try again.\n");
}
}

```

10.3 程序块

5.2节遇到下列形式的复合语句：

{ 多条语句 }

C语言也允许包含声明的复合语句：

[程序块] { 多条声明 多条语句 }

这里将采用术语程序块来描述这类复合语句。下面是程序块的示例：

```

if (i > j) {
    int temp;

    temp = i; /*swaps values of i and j */
    i = j;
    j = temp;
}

```

默认情况下，声明在程序块中的变量的存储期限是自动的：进入程序块时为存储变量分配单元，而在退出程序块时解除分配。变量具有程序块作用域；也就是说，不能在程序块外引用。

函数体是程序块。在需要临时使用的变量时，函数体内程序块也是非常有用的。在上面这个例子中，需要临时变量以便可以交换i和j的值。在程序块中放置临时变量有两个好处：(1) 避免函数体起始位置的声明与只是临时使用的变量相混淆，(2) 减少了名字冲突。在此例中，名字temp可以根据不同的目的用于同一函数中的其他地方，在程序块中声明的变量temp严格局限于程序块。

193

10.4 作用域

在C程序中，相同的标识符可以有不同的含义。C语言的作用域规则使得程序员（和编译器）可以确定与程序中给定点的相关意义。

下面是最重要的作用域规则：当程序块内的声明命名一个标识符时，此标识符已经是可见的（因为此标识符拥有文件作用域，或者因为它是声明在闭合的程序块内），新的声明临时“隐藏”了旧的声明，同时标识符获得了新的含义。在程序块的末尾，标识符重新获得旧的含义。

思考下面的例子，例子中的标识符i有4种不同的含义：

```

int i;          /* Declaration 1 */
void f(int i)  /* Declaration 2 */
{
    i = 1;
}

void g(void)
{
    int i = 2;      /* Declaration 3 */
    if (i > 0) {
        int i;     /* Declaration 4 */
        i = 3;
    }
    i = 4;
}

void h(void)
{
    i = 5;
}

```

- 在声明1中，`i`是具有静态存储期限和文件作用域的变量。
- 在声明2中，`i`是具有程序块作用域的形式参数。
- 在声明3中，`i`是具有程序块作用域的自动变量。
- 在声明4中，`i`也是具有程序块作用域的自动变量。

194

一共使用了5次`i`。C语言的作用域规则允许确定每种情况中的`i`的含义：

- 因为声明2隐藏了声明1，所以赋值`i = 1`引用了声明2中的形式参数，而不是声明1中的变量。
- 因为声明3隐藏了声明1，而且声明2超出了作用域，所以判定`i > 0`引用了声明3中的变量。
- 因为声明4隐藏了声明3，所以赋值`i = 3`引用了声明4中的变量。
- 赋值`i = 4`引用了声明3中的变量。声明4超出了作用域，所以不能引用。
- 赋值`i = 5`引用了声明1中的变量。

10.5 构建C程序

既然已经看过构建C程序的主要元素，那么现在应该为编排这些元素开发一套方法。目前假设程序始终适合单个文件。第15章说明了用分离的几个文件构造一个程序的方法。

迄今为止，已经知道程序可以包含：

- 诸如`#include`和`#define`这样的预处理指令。
- 类型定义。
- 函数声明和外部变量声明。
- 函数定义。

C语言对上述这些项的顺序要求极少：预处理指令直到行出现时才会起作用；类型名直到定义好才可以使用；变量直到声明后才可以使用；虽然C语言对函数没有过分要求，但是这里强烈建议在第一次调用函数前要对每个函数进行定义或声明。

为了遵守这些规则，这里有几个构建程序的方法。下面是一种可能的编排顺序：

- `#include`指令
- `#define`指令

- 类型定义
- 外部变量声明
- 除main函数之外的函数原型
- main函数的定义
- 其他函数的定义

因为#include指令带来的信息将可能在程序中的几个地方都需要，所以先放置这条指令。

#define指令产生宏，对这些宏的使用通常遍布整个程序。类型定义放置在外部变量声明的上面是合理的，因为这些外部变量的声明可能引用了刚刚定义的类型名。接下来，外部变量声明对于跟随在其后的所有函数都是有效的。在编译器看见原型之前调用函数时，可能会产生问题，而除了main函数以外，声明所有函数可以避免这些问题。这种方法也使得无论用什么顺序编排函数定义都是可能的。例如，根据函数名的字母顺序编排，或者依据相关函数组合在一起进行编排。在其他函数前定义main函数使得读者容易定位程序的起始点。

最后的建议：在每个函数定义前放盒型注释可以给出函数名、描述函数的目的、讨论每个形式参数的含义、描述返回值和罗列任何的副作用。

程序：给一手牌分类

为了说明构造C程序的方法，下面编写一个比前面的例子更复杂的程序。这个程序会对一手牌进行读取和分类。手中的每张牌都有花色（方块、梅花、红桃和黑桃）和等级（2、3、4、5、6、7、8、9、10、J、Q、K和A）。不允许使用王牌（纸牌中可当任何点数用的一张），而且假设A是最高等级的。程序将读取一手5张的牌，然后根据下列类别把手中的牌分类（列出的顺序从最好到最坏）：

- 同花顺的牌（即顺序相连又都是同花色）。
- 4张相同的牌（4张牌等级相同）。
- 3张相同和2张相同的牌（3张牌是同样的花色，而另外2张牌是同样的花色）。
- 同花色的牌（5张牌是同花色的）。
- 同顺序的牌（5张牌的等级顺序相连）。
- 3张相同的牌（3张牌的等级相同）。
- 2对子。
- 1对（2张牌的等级相同）。
- 其他牌（任何其他情况的牌）。

如果一手牌有两种或多种类别，程序将选择最好的一种。

为了便于输入，将把牌的等级和花色简化（字母可以是大写的也可以是小写的）。

- 等级：2 3 4 5 6 7 8 9 t j q k a。
- 花色：c d h s。

如果用户输入非法牌或者输入同张牌2次，程序将把此牌忽略掉，产生报错信息，然后要求输入另外一张牌。如果输入为0而不是一张牌，就会导致程序终止。

与程序的会话如下显示：

```
Enter a card: 2s
Enter a card: 5s
Enter a card: 4s
Enter a card: 3s
Enter a card: 6s
Straight flush
```

```
Enter a card: 8c
```

195

196

```

Enter a card: as
Enter a card: 8c
Duplicate card; ignored.
Enter a card: 7c
Enter a card: ad
Enter a card: 3h
Pair

Enter a card: 6s
Enter a card: d2
Bad card; ignored.
Enter a card: 2d
Enter a card: 9c
Enter a card: 4h
Enter a card: ts
High card

Enter a card: 0

```

从上述程序的描述可以看出它有3个任务：

- 读入一手5张牌。
- 分析一手牌的对、顺序和其他。
- 显示一手牌的分类。

把程序分为3个函数来分别完成上述3个任务，即read_cards函数、analyze_hand函数和print_result函数。main函数只负责在无限循环中调用这些函数。这些函数需要共享大量的信息，所以让它们通过外部变量来进行交流。read_cards函数将存储放进几个外部变量中的信息，然后analyze_hand函数将检查这些外部变量，把结果分类放在便于print_result函数显示的其他外部变量中。

基于这些初步设计可以开始勾画程序的轮廓：

```

/* #include directives */

/* #define directives */

/* declarations of external variables */

void read_cards(void);
void analyze_hand(void);
void print_result(void);
***** 
* main: Calls read_cards, analyze_hand, and print_result      *
*          repeatedly.                                              *
***** 
main ()
{
    for (;;) { /* infinite loop */
        read_cards();
        analyze_hand();
        print_result();
    }
}

***** 
* read_cards: Reads the cards into external variables;      *
*           checks for bad cards and duplicate cards.       *
***** 
void read_cards (void)
{
    ...
}

```

```

*****
 * analyze_hand: Determines whether the hand contains a      *
 * straight, a flush, four-of-a-kind,                      *
 * and/or a three-of-a-kind; determines the    *
 * number of pairs; stores the results into   *
 * external variables.                                     *
 *****
void analyze_hand(void)
{
    ...
}

*****
 * print_result: Notifies the user of the result, using      *
 *                 the external variables set by               *
 *                 analyze_hand.                                *
 *****
void print_result(void)
{
    ...
}

```

余下的最紧迫的问题是如何表示一手牌。看看read_cards函数和analyze_hand函数将对这手牌执行什么操作。分析这手牌期间，analyze_hand函数将需要知道每个等级和每个花色的牌的数量。建议使用两个数组，即num_in_rank和num_in_suit。num_in_rank[r]的值是等级为r的牌的数量，而num_in_suit[s]的值是花色为s的牌的数量。（把0~12的数编码为等级，把0~3的数编码为花色。）为了便于read_cards函数检查重复的牌，还需要第3个数组card_exists。每次读取等级为r且花色为s的牌时，read_cards函数都会检查card_exists[r][s]的值是否为TRUE。如果是，就表示此张牌已经录入过；如果不是，那么read_cards函数把TRUE赋值给card_exists[r][s]。

198

read_cards函数和analyze_hand函数都需要访问数组num_in_rank和num_in_suit，所以这两个数组必须是外部变量。然而，数组card_exists只用于read_cards函数，所以将设为此函数的局部变量。作为原则，只有在必要时才把变量设为外部变量。

已经确定了主要的数据结构，现在可以完成程序了：

```

poker.c
/* Classifies a poker hand */

#include <stdio.h>
#include <stdlib.h>

#define NUM_RANKS 13
#define NUM_SUITS 4
#define NUM_CARDS 5
#define TRUE 1
#define FALSE 0

typedef int Bool;

int num_in_rank[NUM_RANKS];
int num_in_suit[NUM_SUITS];
Bool straight, flush, four, three;
int pairs; /* can be 0, 1, or 2 */

void read_cards(void);
void analyze_hand(void);
void print_result(void);

```

```

*****
* main: Calls read_cards, analyze_hand, and print_result *
*      repeatedly.
*****
main()
{
    for (;;) { /* infinite loop */
        read_cards();
        analyze_hand();
        print_result();
    }
}

*****
* read_cards: Reads the cards into the external
*             variables num_in_rank and num_in_suit;
*             checks for bad cards and duplicate cards.
*****
void read_cards(void)
{
    Bool card_exists[NUM_RANKS][NUM_SUITS];
    char ch, rank_ch, suit_ch;
    int rank, suit;
    Bool bad_card;
    int cards_read = 0;

    for (rank = 0; rank < NUM_RANKS; rank++) {
        num_in_rank[rank] = 0;
        for (suit = 0; suit < NUM_SUITS; suit++)
            card_exists[rank][suit] = FALSE;
    }

    for (suit = 0; suit < NUM_SUITS; suit++)
        num_in_suit[suit] = 0;

    while (cards_read < NUM_CARDS) {

        bad_card = FALSE;

        printf("Enter a card: ");

        rank_ch = getchar();
        switch (rank_ch) {
            case '0': exit(EXIT_SUCCESS);
            case '2': rank = 0; break;
            case '3': rank = 1; break;
            case '4': rank = 2; break;
            case '5': rank = 3; break;
            case '6': rank = 4; break;
            case '7': rank = 5; break;
            case '8': rank = 6; break;
            case '9': rank = 7; break;
            case 't': case 'T': rank = 8; break;
            case 'j': case 'J': rank = 9; break;
            case 'q': case 'Q': rank = 10; break;
            case 'k': case 'K': rank = 11; break;
            case 'a': case 'A': rank = 12; break;
            default: bad_card = TRUE;
        }

        suit_ch = getchar();
        switch (suit_ch) {
            case 'c': case 'C': suit = 0; break;

```

```

        case 'd': case 'D': suit = 1; break;
        case 'h': case 'H': suit = 2; break;
        case 's': case 'S': suit = 3; break;
        default:           bad_card = TRUE;
    }

    while ((ch = getchar()) != '\n')
        if (ch != ' ') bad_card = TRUE;

    if (bad_card)
        printf("Bad card; ignored.\n");
    else if (card_exists[rank][suit])
        printf("Duplicate card; ignored.\n");
    else {
        num_in_rank[rank]++;
        num_in_suit[suit]++;
        card_exists[rank][suit] = TRUE;
        cards_read++;
    }
}

/*****
 * analyze_hand: Determines whether the hand contains a      *
 * straight, a flush, four-of-a-kind,      *
 * and/or a three-of-a-kind; determines the      *
 * number of pairs; stores the results into      *
 * the external variables straight, flush,      *
 * four, three, and pairs.      *
 *****/
void analyze_hand(void)
{
    int num_consec = 0;
    int rank, suit;

    straight = FALSE;
    flush = FALSE;
    four = FALSE;
    three = FALSE;
    pairs = 0;

    /* check for flush */
    for (suit = 0; suit < NUM_SUITS; suit++)
        if (num_in_suit[suit] == NUM_CARDS)
            flush = TRUE;

    /* check for straight */
    rank = 0;
    while (num_in_rank[rank] == 0) rank++;
    for (; rank < NUM_RANKS && num_in_rank[rank]; rank++)
        num_consec++;
    if (num_consec == NUM_CARDS) {
        straight = TRUE;
        return;
    }

    /* check for 4-of-a-kind, 3-of-a-kind, and pairs */
    for (rank = 0; rank < NUM_RANKS; rank++) {
        if (num_in_rank[rank] == 4) four = TRUE;
        if (num_in_rank[rank] == 3) three = TRUE;
        if (num_in_rank[rank] == 2) pairs++;
    }
}

```

201

```

}
***** print_result: Notifies the user of the result, using ****
*          the external variables straight, flush,   *
*          four, three, and pairs.                 *
***** void print_result(void)
{
    if (straight && flush) printf("Straight flush\n\n");
    else if (four)         printf("Four of a kind\n\n");
    else if (three &&
             pairs == 1)   printf("Full house\n\n");
    else if (flush)        printf("Flush\n\n");
    else if (straight)     printf("Straight\n\n");
    else if (three)        printf("Three of a kind\n\n");
    else if (pairs == 2)   printf("Two pairs\n\n");
    else if (pairs == 1)   printf("Pair\n\n");
    else                   printf("High card\n\n");
}

```

注意在read_cards函数中exit函数的使用。由于exit函数具有在任何函数中终止程序执行的能力，所以它对于此程序是十分方便的。

问与答

*问：如果局部变量具有静态存储期限，那么会对递归函数产生什么影响？(p.131)

答：当函数是递归函数时，每次调用它时都会产生其自动变量的新副本。静态变量就不会发生这样的情况，相反，所有的函数调用都共享同一个静态变量。

问：在下面的例子中，j初始化的值和i一样，但是有两个命名为i的变量：

```

int i = 1;

void f(void)
{
    int j = i;
    int i = 2;
}

```

这段代码是否合法？如果合法，j的初始值是1还是2？

202 答：局部变量的作用域是从声明处开始的。因此，j的声明引用了名为i的外部变量。j的初始值将是1。

练习

10.2节

- 修改栈示例使它存储字符而不是整数。接下来，增加main函数，用来要求用户输入一串圆括号或大括号，然后指出它们是否是正确的嵌套：

```

Enter parentheses and/or braces: {{(){}{()}}}
Parentheses/braces are nested properly

```

提示：与程序读入字符一样，假设已经把每个左边圆括号或左边大括号压入栈中。当读入右边圆括号或右边大括号时，把栈顶的项弹出，并且检查弹出项是否是匹配的圆括号或大括号。（如果不是，那么圆括号或大括号嵌套不正确。）当程序读入换行符时，检查栈是否为空。如果为空，那么圆括号或大括号匹配；如果栈不为空（或者如果曾经调用过stack_underflow函数），那么圆括号或大括号不匹配。如果调用stack_underflow函数，程序显示信息Stack overflow，并且立刻终止。

10.4节

- 下面的程序框架只显示了函数定义和变量声明。

```
int a;  
void f(int b)  
{  
    int c;  
}  
void g(void)  
{  
    int d;  
    {  
        int e;  
    }  
}  
main()  
{  
    int f;  
}
```

对于下面每种作用域，列出在此作用域内的所有变量的名字和形式参数的名字：

- (a) f函数。
- (b) g函数。
- (c) 声明e的程序块。
- (d) main函数。

10.5节

3. 修改poker.c程序，把数组num_in_rank和数组num_in_suit移入main函数。main函数将把这两个数组作为实际参数传递给read_cards函数和analyze_hand函数。
4. 把数组num_in_rank、数组num_in_suit和数组card_exists从poker.c程序中去掉。程序改用5x2的数组来代替存储牌。
5. 修改poker.c程序，使其识别牌的额外类别——“同花大顺”(A、K、Q、J，以及同样花色的10)。同花大顺的级别高于其他所有的类别。
6. 修改poker.c程序，使其允许“小A顺”(即A、2、3、4和5)。

203

第11章

指 针

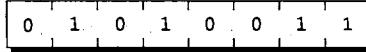
我忘记了第十一条戒律是“你应该计算”，还是“你不应该计算”。

指针是C语言最重要也是最常被误解的特性之一。由于指针的重要性，本书将用3个章对其进行讨论。本章将集中在指针的基础上，而第12章和第17章则介绍指针的更高级应用。

本章将从机器地址以及与其相关的指针变量的讨论开始（11.1节）；然后，11.2节介绍取地址运算符和间接寻址运算符；11.3节涵盖了指针赋值的内容；11.4节说明了给函数传递指针的方法，而11.5节则讨论从函数返回指针。

11.1 指针变量

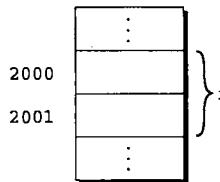
理解指针的第一步是在机器层上观察指针表示的内容。大多数现代计算机用字节（byte）来分割内存，每个字节可以存储8位的信息。



内存为16MB的机器拥有16 777 216个字节。每个字节都有唯一的地址（address），用来和内存中的其他字节进行区别。如果内存中有 n 个字节，那么可以认为作为地址的数的范围是0~ $n-1$ 。

| 地址 | 内容 |
|-------|----------|
| 0 | 01010011 |
| 1 | 01110101 |
| 2 | 01110011 |
| 3 | 01100001 |
| 4 | 01101110 |
| ⋮ | ⋮ |
| $n-1$ | 01000011 |

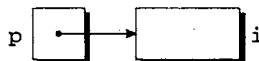
可执行程序由代码（原始C程序中与语句对应的机器指令）和数据（原始程序中的变量）两部分构成。程序中的每个变量占有一个或多个内存字节，把第一个字节的地址称为是变量的地址。下图中，变量*i*占有的字节从地址2000到地址2001，所以变量*i*的地址是2000：



这就是指针的出处。虽然用数表示地址，但是其取值范围可能不同于整数的范围，所以一定不能用普通整型变量存储地址。但是，可以用特殊的指针变量（pointer variable）存储地址。

在用指针变量p存储变量i的地址时，我们说成是p“指向”i。**Q&A**换句话说，指针就是地址，而且指针变量是只存储地址的变量。

本书的例子不再用数作为地址显示，而将采用更加简单的符号。为了说明指针变量p存储变量i的地址，将把p的内容显示为指向i的箭头：



声明指针变量

对指针变量的声明与对普通变量的声明基本一样，唯一的不同就是必须在指针变量名字前放置星号：206

```
int *p;
```

上述声明说明p是指向int型对象的指针变量。正如第17章将看到的那样，用术语对象来代替变量，这是因为p可以指向不用作变量的内存区域。（在19.1.2节讨论程序设计时会知道对象将有不同的含义。）

指针变量可以和其他变量一起出现在声明中：

```
int i, j, a[10], b[20], *p, *q;
```

在这个例子中，i和j都是普通整型变量，a和b是整型数组，而p和q是指向整型对象的指针。

C语言要求每个指针变量唯一指向特定类型（引用类型（referenced type））的对象：

```
int *p /*points only to integers */
float *q /*points only to floats */
char *r /*points only to characters */
```

对于什么可以作为引用类型没有限制。（指针变量甚至可以指向另一个指针，即指向指针的指针（>17.6节）。）

11.2 取地址运算符和间接寻址运算符

为使用指针，C语言提供了一对特殊设计的运算符。为了找到变量的地址，可以使用&（取地址）运算符。如果x是变量，那么&x就是x在内存中的地址。为了获得对指针所指向对象的访问，可以使用*（间接寻址）运算符。如果p是指针，那么*p表示p当前指向的对象。

11.2.1 取地址运算符

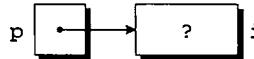
声明指针变量是为指针留出空间，但是并没有把它指向对象：

```
int *p; /* points nowhere in particular */
```

（关于这点，指针和其他变量没有区别。）在使用前初始化p是至关重要的。一种初始化指针变量的方法是把某个变量的地址赋给它，或者更常采用左值（>4.2.2节），即使用&运算符：

```
int i, *p;
p = &i;
```

通过把i的地址赋值给变量p的方法，上述语句把p指向了i：207



顺便说一句，把&i赋值给p不会影响i的值。

取地址运算符可以出现在声明中，所以在声明指针的同时对它进行初始化是可行的：

```
int i;
```

```
int *p = &i;
甚至可以把i的声明和p的声明合并，但是需要首先声明i：
int i, *p = &i;
```

11.2.2 间接寻址运算符

一旦指针变量指向了对象，就可以使用*（间接寻址）运算符访问存储在对象中的内容。例如，如果p指向i，那么可以如下所示显示出i的值：

```
printf("%d\n", *p);
```

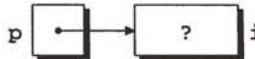
printf函数将会显示i的值，而不是i的地址。

有算术倾向的读者可能希望把*想象成&的反向操作。对变量使用&运算符产生指向变量的指针；而对指针使用*运算符则可以返回到原始变量：

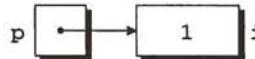
```
j = *&i; /* same as j = i; */
```

只要p指向i，那么*p就是i的别名。*p不仅拥有和i同样的值，而且对*p的改变也会改变i的值。（*p是左值，所以对它赋值是合法的。）下面的例子说明了*p和i的等价关系。下图显示了在计算中不同的点上p和i的值。

```
p = &i;
```



```
i = 1;
```



208

```
printf("%d\n", i);      /* prints 1 */
printf("%d\n", *p);      /* prints 1 */
*p = 2;
```



```
printf("%d\n", i);      /* prints 2 */
printf("%d\n", p);      /* prints 2 */
```



不要把间接寻址运算符用于未初始化的指针变量。如果指针变量p没有初始化，那么*p的值是未定义的：

```
int *p;
printf("%d", *p); /* prints garbage */
```

给*p赋值甚至会更糟；p可以指向内存中的任何地方，所以赋值改变了某些未知的内存单元：

```
int *p;
*p = 1;           /*** WRONG ***/

```

上述赋值改变的内存单元可能属于程序（可能导致不规律的行为）或者属于操作系统（可能导致系统崩溃）。

11.3 指针赋值

C语言允许使用赋值运算符进行指针的复制，前提是两个指针具有相同的类型。假设i、j、

p和q有如下声明：

```
int i, j, *p, *q;
```

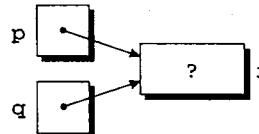
语句

```
p = &i;
```

是指针赋值的示例：把i的地址复制给p。下面是另一个指针赋值的示例：

```
q = p;
```

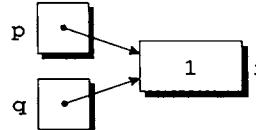
这条语句是把p的内容（即i的地址）复制给q，效果是把q指向了和p指向相同的地方：



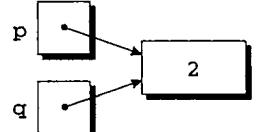
209

现在p和q都指向了i，所以可以用对*p或*q赋新值的方法来改变i：

```
*p = 1;
```



```
*q = 2;
```



任意数量的指针变量都可以指向同一个对象。

注意不要把

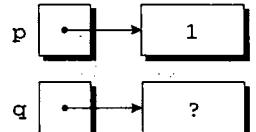
```
q = p;
```

和

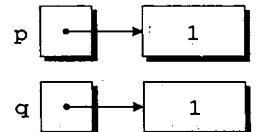
```
*q = *p;
```

搞混。第一条语句是指针赋值；而第二条语句不是。就如下面的例子显示的：

```
p = &i;
q = &j;
i = 1;
```



```
*q = *p;
```



赋值语句*q = *p是把p指向的值（i的值）复制到q指向的内存单元（变量j）中。

210

11.4 指针作为实际参数

到目前为止，我们回避了一个十分重要的问题：指针对什么有益呢？因为C语言中指针有

几个截然不同的应用，所以针对此问题没有唯一的答案。在本节中，只会看到指针的一种应用：调用函数时传递指向变量的指针，通过这种方法使得函数可以改变变量的值。

在9.3节中看到，因为C语言用值进行参数传递，所以在函数调用中用变量作为实际参数会阻止对变量的改变。如果需要函数能够改变变量，那么C语言的这种特性可能是很麻烦的。9.3节中，无法编写用来改变两个参数的decompose函数。

指针提供了此问题的解决方法：不再传递变量x作为函数的实际参数，而是采用&x，即指向x的指针。声明相应的形式参数p是指针。调用函数时，p将有&x值，因此*p将是x的别名。函数体内*p的每次出现都将是对x的间接引用，而且允许函数既可以读取x也可以修改x。

为了用实例证明这种方法，下面通过把形式参数int_part和frac_part声明成指针的方法来修改decompose函数。现在decompose函数的定义形式如下：

```
void decompose(float x, int *int_part, float *frac_part);
{
    *int_part = (int) x;
    *frac_part = x - *int_part;
}
```

decompose函数的原型既可以是

```
void decompose(float x, int *int_part, float *frac_part);
```

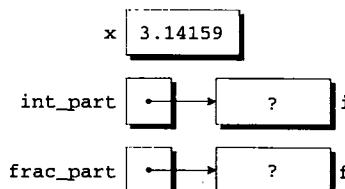
也可以是

```
void decompose(float, int *, float *);
```

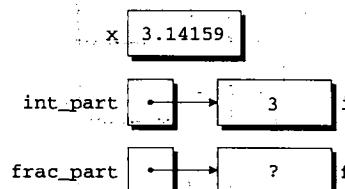
以下列方式调用decompose函数：

```
decompose(3.14159, &i, &f);
```

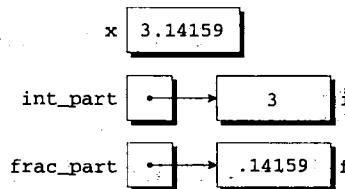
因为i和f前有地址运算符&，所以decompose函数的实际参数是指向i和f的指针，而不是i和f的值。调用decompose函数时，把值3.14159复制到x中，把指向i的指针存储在int_part中，而把指向f的指针存储在frac_part中：



decompose函数体内的第一个赋值把x的值转换为int型，并且把此值存储在int_part指向的内存单元中。因为int_part指向i，所以赋值把值3放到i中：



第二个赋值把int_part指向的值（即i的值）取出，现在这个值是3。把此值转换为float类型，并且用x减去它，得到.14159。然后把这个值存储在frac_part指向的内存单元中：



当decompose函数返回时，就像原来希望的那样，i和f将分别有值3和14159。

用指针作为函数的实际参数实际上不新鲜。因为第2章已经在scanf函数调用中使用过了。思考下面的例子：

```
int i;
scanf("%d", &i);
```

必须把&放在i的前面以便给scanf函数一个指向i的指针；指针会告诉scanf函数读取的值所放置的位置。没有&运算符，scanf函数将无法应用i的值。

虽然需要scanf函数的实际参数是指针，但是不是每个实际参数总是需要&运算符的。在下面的例子中，scanf函数传递了指针变量：

```
int i, *p;
p = &i;
scanf("%d", p);
```

既然p包含了i的地址，那么scanf函数将读入整数并且把它存储在i中。在调用中使用&运算符将是错误的：

```
scanf("&d", &p); // *** WRONG ***
```

scanf函数读入整数并且把它存储在p中而不是i中。



向函数传递需要的指针却失败了可能会产生严重的后果。假设调用在i和f前不带&运算符的decompose函数：

```
decompose(3.14159, i, f);
```

decompose函数期望指针作为第二个和第三个实际参数，但是却用i和f的值代替了。decompose函数没有办法表明差异，所以它将会把i和f的值假设为指针来使用。当decompose函数把值存储到*int_part和*frac_part中时，它会把值写入到未知的内存单元中，而不是修改i和f。

如果已经提供了decompose函数的原型（当然，应该始终这样做），那么编译器将让我们知道正在试图传递错误的实际参数类型。然而，在scanf例子中，编译器通常不会检查出传递指针失败，而是认为scanf是有特别错误倾向的函数。

C++ C++语言提供了可以不需要传递指针就能修改函数的实际参数的方法。19.4节会给出详细介绍。

11.4.1 程序：找出数组中的最大元素和最小元素

为了说明如何在函数中传递指针，下面来看一个名为max_min的函数，该函数找到数组中的最大元素和最小元素。调用max_min函数时，将传递两个指向变量的指针；max_min函数将把答案存储在这些变量中。max_min函数具有下列原型：

```
void max_min(int a[], int n, int *max, int *min);
```

max_min函数的调用可以具有下列的形式：

```
max_min(b, N, &big, &small);
```

b是整型数组，而N是数组b中的元素数量。big和small是普通的整型变量。当max_min函数找到数组b中的最大元素时，通过给*max赋值的方法把值存储在big中。（因为max指向big，所以给*max赋值将会修改big的值。）通过给*min赋值把最小元素的值存储在small中。

为了测试max_min函数，将编写程序用来往数组中读入10个数，然后把数组传递给max_min函数，并且显示出结果：

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
Largest: 102
Smallest: 7
```

下面是完整的程序：

```
maxmin.c
/* Finds the largest and smallest elements in an array */

#include <stdio.h>

#define N 10

void max_min(int a[], int n, int *max, int *min);

main()
{
    int b[N], i, big, small;

    printf("Enter %d numbers: ", N);
    for (i = 0; i < N; i++)
        scanf("%d", &b[i]);

    max_min(b, N, &big, &small);

    printf("Largest: %d\n", big);
    printf("Smallest: %d\n", small);

    return 0;
}

void max_min(int a[], int n, int *max, int *min)
{
    int i;

    *max = *min = a[0];
    for (i = 1; i < n; i++) {
        if (a[i] > *max)
            *max = a[i];
        else if (a[i] < *min)
            *min = a[i];
    }
}
```

11.4.2 用 const 保护实际参数

当调用函数并且传递给它指向变量的指针时，通常会假设函数将修改变量（否则，为什么函数需要指针呢？）。例如，如果在程序中看到语句

f(&x);

大概是希望f改变x的值。但是，f仅需要检查x的值而不是改变它的值也是可能的。指针可能高效的原因是：如果变量需要大量的存储空间，那么传递变量的值可能浪费时间和空间。（12.3节更详细地介绍这方面内容。）

可以使用单词const来证明函数不会改变传递给函数的指针所指向的对象。**Q&A**为了允许f检查传递的指针所指向的实际参数，而不是修改它，可以在参数声明中把const放置在形式参数的类型说明之前：

```
void f(const int *p)
{
    *p = 0;    /* *** WRONG ***/
}
```

const的使用说明p是指向“整型常量”的指针。试图改变*p将会引发编译器发出特定消息。

11.5 指针作为返回值

不仅可以为函数传递指针，还可以编写返回指针的函数。例如，我们可能希望函数返回结果的内存位置而不是返回值。返回指针的函数是相对普遍的；第13章将会遇到几个。

当给定指向两个整数的指针时，下列函数返回指向两整数中较大数的指针：

```
int *max(int *a, int *b)
{
    if (*a > *b)
        return a;
    else
        return b;
}
```

调用max函数时，将传递指向两个int型变量的指针，并且把结果存储在指针变量中：

```
int *p, x, y;
p = max(&x, &y);
```

调用max期间，*a是x的别名，而*b是y的别名。如果x的值大于y，那么max返回x的地址；否则，max返回y的地址。调用函数后，p可以指向x也可以指向y。

虽然max函数返回的指针是作为实际参数传递的两个指针中的一个，但是这不是函数可以返回的唯一事情。一些函数返回的指针指向作为实际参数传递的数组中的一个元素。另外一种可能是返回指向外部变量或指向声明为static的局部变量的指针。



永远不会返回指向自动局部变量的指针：

```
int *f(void)
{
    int i;
    ...
    return &i;
}
```

一旦f返回，变量i就不存在了，所以指向变量i的指针将是无效的。

问与答

*问：指针总是和地址一样吗？(p.147)

答：通常是，但不总是。考虑用字而不是字节划分内存的计算机。字可以包含36位、60位，或者更多位。

如果假设36位的字，那么内存将有如下的显示：

| Address | Contents |
|---------|--------------------------------------|
| 0 | 001010011001010011001010011001010011 |
| 1 | 001110101001110101001110101001110101 |
| 2 | 001110011001110011001110011001110011 |
| 3 | 001100001001100001001100001001100001 |
| 4 | 001101110001101110001101110001101110 |
| ⋮ | |
| n-1 | 001000011001000011001000011001000011 |

当用字划分内存时，每个字都有一个地址。通常整数占一个字长度，所以指向整数的指针可以就是一个地址。但是，字可以存储多于一个的字符。例如，36位的字可以存储6个6位的字符：

| | | | | | |
|--------|--------|--------|--------|--------|--------|
| 010011 | 110101 | 110011 | 100001 | 101110 | 000011 |
|--------|--------|--------|--------|--------|--------|

或者4个9位的字符：

| | | | |
|-----------|-----------|-----------|-----------|
| 001010011 | 001110101 | 001110011 | 001100001 |
|-----------|-----------|-----------|-----------|

由于这个原因，可能需要用不同于其他指针的格式存储指向字符的指针。指向字符的指针可以由地址（存储字符的字）加上一个小整数（字符在字内的位置）组成。

在一些计算机上，指针可能是“偏移量”而不完全是地址。例如，Intel微处理器（用于IBM PC和其他产品）具有复杂的模式，模式中的地址有时用单独的16位数（偏移量）表示，有时用两个16位数（段：偏移量对）表示。偏移量不是真正的内存地址；CPU必须把它和存储在特殊寄存器中段的值联合起来。

用于IBM PC家族的C语言编译器通过提供两种指针的方式处理Intel的分段结构：近指针（16位偏移量）和远指针（32位段：偏移量对）。由于这个原因，PC编译器通常保留单词near和far用于指针变量的声明。

*问：如果指针可以指向程序中的数据，那么使指针指向程序代码是否可能？

答：可能。17.7节将会介绍指向函数的指针。

问：是否存在显示指针的值的方法？

答：调用printf函数，在格式串中采用转换%p；请参考22.3节获取更多细节。

问：下列声明使人糊涂：

```
void f(const int *p);
```

这是说我们不能修改p吗？

答：不是。说明不能改变指针p指向的整数，但是并不阻止改变p自身。

```
void f(const int *p);
{
    int j;

    p = &j; /* legal */
}
```

因为实际参数是按值传递，所以通过使指针指向其他地方的方法给p赋新值不会对函数外产生任何影响。

*问：如下例所示，当声明指针类型的形式参数时，在参数名前面放置单词const是否合法？(p.152)

```
void f(int * const p);
```

答：是合法的。然而效果不同于把const放在p的类型前面。在11.4节中已经见过在p的类型前面放置const可以保护p指向的对象。在p的类型后面放置const可以保护p本身：

```
void f(int * const p);
{
    int j;

    *p = 0; /* legal */
    p = &j; /* ***WRONG *** */
}
```

这一特性并不经常用到。因为p很少是另一个指针（调用函数时的实际参数）的副本，所以极少有什么理由保护它。

极少出现的情况是需要同时保护p和它所指向的对象，这可以通过在p类型的前和后都放置const来实现：

```
void f(const int * const p);
{
    int j;
```

```

*p = 0;      /*** WRONG ***/
p = &j;      /*** WRONG ***/
}

```

练习

11.2节

1. 如果i是变量，并且p指向i，那么下列哪个表达式是i的别名？

- (a) *p (c) *&p (e) *i (g) *&i
- (b) &p (d) &*p (f) &i (h) &*i

11.3节

2. 如果i是int型变量，而且p和q是指向int的指针，下列哪个赋值是合法的？

- (a) p = i; (d) p = &p; (g) p = *q;
- (b) *p + &i; (e) p = &p; (h) *p = q;
- (c) &p = q; (f) p = q; (i) *p = *q;

11.4节

3. 下列函数假设用来计算数组a中元素的和以及平均值，且数组a长度为n。avg和sum指向函数需要修改的变量。函数含有几个错误，请找出这些错误并且修改它们。

```

void avg_sum(float a[], int n, float *avg, float *sum)
{
    int i;

    sum = 0.0;
    for (i = 0; i < n; i++)
        sum += a[i];
    avg = sum / n;
}

```

218

4. 编写下列函数：

```
void swap(int *p, int *q);
```

当传递两个变量的地址时，swap函数应该交换两者的值：

```
swap(&x, &y); /* exchange values of x and y */
```

利用此函数修改第9章中的练习9中的程序，使它可以完成这项工作。

5. 编写下列函数：

```
void split_time(long int total_sec,
                int *hr, int *min, int *sec);
```

total_sec是以从午夜计算的秒数表示的时间。hr、min和sec都是指向变量的指针，这些变量在函数中将分别存储着按小时算（0~23）、按分钟算（0~59）和按秒算（0~59）的等价的时间。

6. 编写下列函数：

```
void find_two_largest(int a[], int n, int *largest,
                      int *second_largest);
```

当传递长度为n的数组a时，函数将在数组a中搜寻最大元素和第二大元素，把它们存储在分别largest和second_largest指向的变量中。

11.5节

7. 编写下列函数：

```
int *find_middle(int a[], int n);
```

当传递长度为n的数组a时，函数将返回指向数组的中间元素的指针。（如果n是偶数，选择较大下标的中间元素。例如，如果n=4，中间元素是a[2]，不是a[1]。）

219

第12章

指针和数组

优化阻碍发展。

第11章介绍了指针并且说明了如何把指针用作函数实际参数以及把指针用作函数的返回值。本章涵盖指针的另一种应用。当指针指向数组元素时，C语言允许对指针进行算术运算，即加法和减法，这种运算引出了一种对数组进行处理的替换方法，它可以使指针代替数组下标进行操作。

正如本章将看到的那样，C语言中指针和数组的关系是非常紧密的。后面的第13章（字符串）和第17章（指针的高级应用）还将利用到这种关系。理解指针和数组之间的关联对于熟练掌握C语言是非常关键的：它将会使人深入了解C语言的设计过程，并且帮助理解现有的程序。然而，需要知道的是，用指针处理数组的主要原因之一是效率，但是它已经不再像当初那样重要了，这些多亏了改进的编译器。

12.1节讨论指针的算术运算，并且说明如何使用关系运算符和判等运算符进行指针的比较；12.2节示范如何能用指针处理数组元素；12.3节显示了关于数组的关键事实，即可以用数组的名字作为指向数组中第一个元素的指针，并且利用这个事实说明数组型实际参数是如何真正工作的；12.4节说明前3节的主题如何应用于多维数组。

12.1 指针的算术运算

221

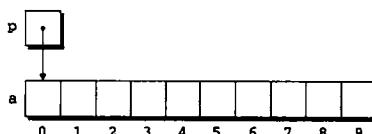
指针不仅可以指向普通变量，还可以指向数组元素。例如，假设已经声明a和p如下：

```
int a[10], *p;
```

通过下列写法可以使p指向a[0]：

```
p = &a[0];
```

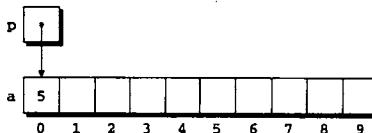
用图形方式表示，下面是我们刚刚做的：



现在可以通过p访问a[0]。例如，可以通过下列写法把值5存入a[0]中：

```
*p = 5;
```

下图显示的是现在的情况：



把指针p指向数组a的元素不是特别令人激动。但是，通过在p上执行指针算术运算（或者地址算术运算）可以访问到数组a的其他元素。C语言支持3种（而且只有3种）格式的指针算术运算：

- 指针加上整数。
- 指针减去整数。
- 两个指针相减。

一起来仔细看看每种运算。所有例子都假设有如下声明：

```
int a[10], *p, *q, i;
```

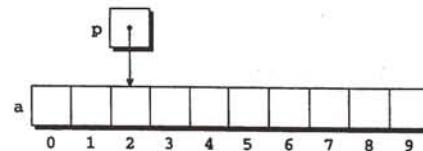
12.1.1 指针加上整数

指针p加上整数j产生指向特定元素的指针，这个特定元素是p原先指向的元素后的j个位置。**Q&A**更确切些说，如果p指向数组元素a[i]，那么p+j指向a[i+j]（当然，前提是a[i+j]必须存在）。

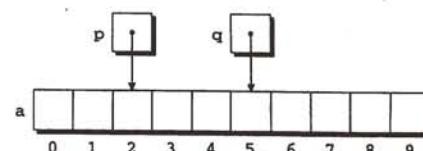
下面的示例说明指针的加法运算，插图说明计算中p和q在不同点的值。

222

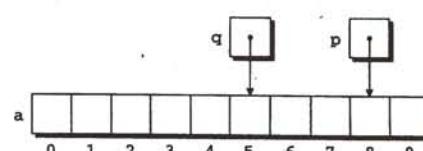
```
p = &a[2];
```



```
q = p + 3;
```



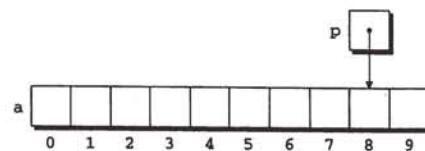
```
p += 6;
```



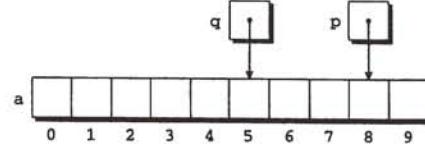
12.1.2 指针减去整数

如果p指向数组元素a[i]，那么p-j指向a[i-j]。例如：

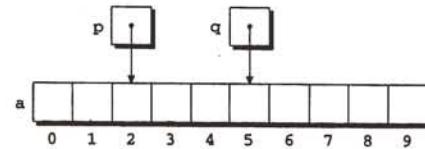
```
p = &a[8];
```



```
q = p-3;
```



```
p -= 6;
```

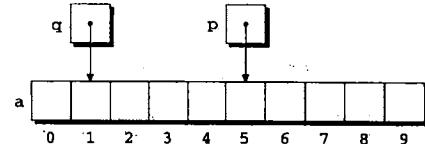


12.1.3 指针相减

当两个指针相减时，结果为指针之间的距离（用来计算数组中元素的个数）。因此，如果p指向a[i]且q指向a[j]，那么p-q就等于i-j。例如：

```
p = &a[5];
q = &a[1];
```

```
i = p-q;      /* i is 4 */
i = q-p;      /* i is -4 */
```



注意，Q&A只有在p指向数组元素时，指针p上的算术运算才会获得有意义的结果。此外，只有在两个指针指向同一个数组时，指针相减才有意义。

12.1.4 指针比较

可以用关系运算符(<、<=、>、>=)和判等运算符(==和!=)进行指针比较。只有在两个指针指向同一数组时，用关系运算符进行的指针比较才有意义。比较的结果依赖于数组中两个元素的相对位置。例如，在下面的赋值后p<=q的值是0，而p>=q的值是1。

```
p = &a[5];
q = &a[1];
```

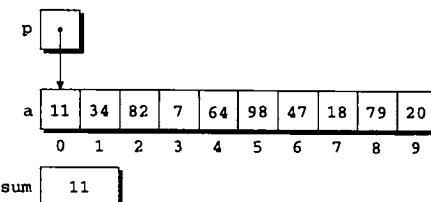
12.2 指针用于数组处理

指针的算术运算允许通过对指针变量进行重复自增来访问数组的元素。下面的程序段说明了这种方法。这段程序用来对数组a的元素进行求和。在示例中，指针变量p初始指向a[0]，每次执行循环，p进行自增；结果是p指向a[1]，然后指向a[2]，并且依次类推。在p执行到数组a的最后一个元素后循环终止。

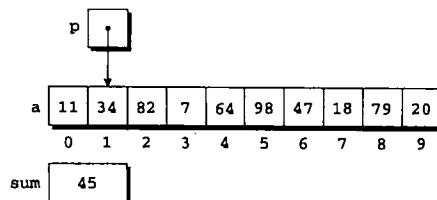
```
#define N 10
int a[N], sum, *p;
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
    sum += *p;
```

224 下图说明了在前3次循环重复的末尾（即p自增操作前）a、sum和p的内容。

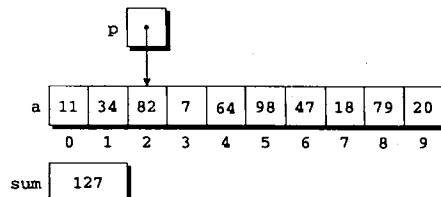
第1次重复的末尾：



第2次重复的末尾：



第3次重复的末尾:



*for*语句中的条件 *p < &a[N]* 值得特别说明一下。在标准C中，即使元素 *a[N]* 不存在（数组 *a* 的下标从 0 到 *N - 1*），但是对它使用取地址运算符是合法的。因为循环不会尝试检查 *a[N]* 的值，所以在上述方式下用 *a[N]* 是非常安全的。利用 *p* 等于 *&a[0]*、*&a[1]*……*&a[N-1]* 可以执行循环体，但是当 *p* 等于 *&a[N]* 时，循环终止。

当然，用下标代替可以很容易地写出不使用指针的循环。实际参数最经常的引用是支持指针的算术运算，这样做可以解决执行时间。**Q&A**但是，与依赖实现比起来，一些编译器依赖下标实际上会产生更好的循环代码。

*运算符和++运算符的组合

C程序员经常在处理数组元素的语句中组合*（间接寻址）运算符和++运算符。思考一个简单的例子，把值存入数组元素中，然后推进到下一个元素。利用数组下标可以写成

a[i++] = *j*;

如果 *p* 指向数组元素，那么相应的语句将会是

**p++* = *j*;

225

因为后缀++在*前执行，所以编译器可以把上述语句看成是

**(p++)* = *j*;

p++ 的值是 *p*。（因为使用后缀++，所以 *p* 只有在表达式计算出来后才可以自增。）因此，**(p++)* 的值将是 **p*，即 *p* 当前指向的对象。

当然，**p++* 不是唯一合法的*和++的组合。例如，可以编写 *(*p)++*，这个表达式返回 *p* 指向的对象的值，然后对象进行自增（*p* 本身是不变的）。如果发现这样很混乱，那么下面的表格可以提供一些帮助：

| 表达式 | 含义 |
|-----------------------------------------|-------------------------------------|
| <i>*p++</i> 或 $\ast(\text{p}++)$ | 自增前表达式的值是 <i>*p</i> ，然后自增 <i>p</i> |
| $(\ast\text{p})++$ | 自增前表达式的值是 <i>*p</i> ，然后自增 <i>*p</i> |
| <i>++p</i> 或 $\text{++}(\text{p})$ | 先自增 <i>p</i> ，自增后表达式的值是 <i>*p</i> |
| <i>++*p</i> 或 $\text{++}(\ast\text{p})$ | 先自增 <i>*p</i> ，自增后表达式的值是 <i>*p</i> |

虽然这4种组合普及性互不相同，但是所有4种组合都可以出现在程序中。最频繁见到的就是 **p++*，它在循环中是很方便的。不再书写下列语句用来对数组 *a* 的元素求和：

```
for (p = &a[0]; p < &a[N]; p++)
    sum += *p;
```

而是可以写成

```
p = &a[0];
while (p < &a[N])
    sum += *p++;
```

*运算符和--运算符的混合方法类似于*和++的组合。为了应用*和--的组合，一起回到10.2节的栈例子。原始版本的栈依赖名为top的整型变量来跟踪contents数组中“栈顶”的位置。现在用指针变量来替换top，指针变量初始指向contents数组的第0个元素。

```
int *top_ptr = &contents[0];
```

下面是新的push函数和pop函数（把更新其他栈函数留作练习）：

```
void push (int i)
{
    if (is_full())
        stack_overflow();
    else
        *top_ptr++ = i;
}
int pop (void)
{
    if (is_empty())
        stack_underflow();
    else
        return *--top_ptr;
}
```

226

注意，因为希望pop函数在取回top_ptr指向的值之前对top_ptr进行自减，所以要写成`*--top_ptr`，而不是`*top_ptr--`。

12.3 用数组名作为指针

指针的算术运算是数组和指针之间相互关联的一种方法，但这不是两者之间唯一的联系。下面是另一种关键的关系：可以用数组的名字作为指向数组第一个元素的指针。这种关系简化了指针的算术运算，而且使得数组和指针都更加通用。

例如，假设用如下形式声明a：

```
int a[10];
```

用a作为指向数组第一个元素的指针，可以修改a[0]：

```
*a = 7; /* stores 7 in a[0] */
```

可以通过指针a + 1来修改a[1]：

```
* (a+1) = 12; /* store 12 in a[1] */
```

通常情况下，a + i就如同&a[i]（两者都是表示指向数组a中元素i的指针），而且*(a+i)就等于a[i]（两者都是表示元素i本身）。换句话说，可以把数组的下标看成是指针算术运算的格式。

数组名可以用作指针的事实使得编写从头到尾单步操作数组的循环更加容易。思考下面这个来自12.2节的循环：

```
for(p = &a[0]; p < &a[N]; p++)
    sum += *p;
```

为了简化这个循环，可以用a替换&a[0]，同时用a+N替换&a[N]：

```
[惯用法] for (p = a; p < a + N; p++)
    sum += *p;
```



虽然可以把数组名用作指针，但是不能给数组名赋新的值。试图使数组名指向其他地方是错误的：

```
while (*a != 0)
    a++; /* *** WRONG *** /
```

227

可以始终把a复制给指针变量，然后改变指针变量。这种方法是不会有巨大损失的：

```
p = a;
while (*p != 0)
    p++;
```

12.3.1 程序：数列反向（改进版）

8.1节的程序reverse.c读进10个数，然后反序写出这些数。程序读数时会把这些数存入数组。一旦读入所有的数，程序就会反向地从尾到头单步浏览数组，同时显示出浏览的数。

原来的程序利用下标来访问数组中的元素。下面是改进后的程序，利用指针的算术运算来代替数组的下标操作。

```
reverse2.c
/*
 * Reverses a series of numbers (pointer version)
 */

#include <stdio.h>

#define N 10

main()
{
    int a[N], *p;

    printf("Enter %d numbers: ", N);
    for (p = a; p < a + N; p++)
        scanf("%d", p);

    printf("In reverse order:");
    for (p = a + N - 1; p >= a; p--)
        printf(" %d", *p);
    printf("\n");

    return 0;
}
```

在原本的程序中，整型变量i用来跟踪数组内的当前位置。新改进的程序用p替换了i，p是指针变量。读入的数还是存储在数组中；我们简单地使用不同的方法来跟踪数组中的位置。

注意，scanf函数的第二个实际参数是p，不是&p。因为p指向数组的元素，所以它是满足scanf函数要求的参数。另一方面，&p则是指向另一个指针的指针，且另一个指针是指向数组元素的。

228

12.3.2 数组型实际参数（改进版）

在传递给函数时，数组名始终作为指针。思考下面的函数，这个函数会返回整型数组中最大的元素：

```
int find_largest(int a[], int n)
{
    int i, max;

    max = a[0];
    for (i = 1; i < n; i++)
        if (a[i] > max)
            max = a[i];
    return max;
}
```

假设调用find_largest函数如下：

```
largest = find_largest (b, N);
```

这个调用会导致把数组b的第一个元素赋值给a；数组本身并没有进行复制。

把数组型形式参数看作是指针的事实会产生许多重要的结果：

- 在给函数传递普通变量时，把变量的值进行复制；任何对相应的形式参数的改变都不会影响到变量。反之，因为没有使数组本身进行复制，所以数组作为实际参数不会防止改

变。例如，通过在数组的每个元素中存储零的方法，下列函数可以修改数组：

```
void store_zeros(int a[], int n)
{
    int i;

    for (i = 0; i < n; i++)
        a[i] = 0;
}
```

为了指明数组型形式参数不会改变，可以在它的声明中包含单词const：

```
int find_largest(const int a[], int n)
{
    ...
}
```

- 给函数传递数组所需的时间不依赖于数组的大小。因为没有对数组进行复制，所以传递大数组不会产生不利的结果。
- 如果想要指针，可以把数组型形式参数声明为指针。例如，可以按如下形式定义find_largest函数：

```
int find_largest(int *a, int n)
{
    ...
}
```

声明a是指针就相当于声明它是数组。**Q&A** 编译器处理这两类声明就好像它们是完全一样的。



虽然声明形式参数是数组就如同声明它是指针一样，但是这种一样不适用于变量。声明

```
int a[10];
```

会导致编译器为10个整数预留了空间，但声明

```
int *a;
```

会导致编译器为指针变量分配空间。在稍后的例子中，a不是数组。试图用a作为数组可能会导致极糟的后果。例如，赋值

```
*a = 0; // *** WRONG *** /
```

将在a指向的地方存储0。因为不知道a指向哪里，所以对程序的影响是无法预料的。

- 可以给形式参数为数组的函数传递数组的“片断”，所谓片断是连续元素的序列。假设希望用find_largest函数来定位数组b中某一部分内的最大元素，比如说元素b[5], ..., b[14]。调用find_largest函数时，将传递b[5]的地址和数10，这说明需要用find_largest函数检查从b[5]开始的10个数组元素：

```
largest = find_largest(&b[5], 10);
```

12.3.3 用指针作为数组名

如果可以用数组名作为指针，那么C语言是否允许把指针好像数组名一样进行标记呢？到目前为止，你可能更愿意期望答案是肯定的，而且相信自己是对的。下面是个例子：

```
#define N 100

int a[N], i, sum = 0, *p = a;

for (i = 0; i < N; i++)
    sum += p[i];
```

编译器对待 $p[i]$ 就像对待 $* (p+i)$ 一样，这是指针算术运算非常正规的用法。虽然我们对下标一个指针的能力有更多一些的好奇，但是要在17.3节才会看到它的实际用法。

12.4 指针和多维数组

就像指针可以指向一维数组的元素一样，指针还可以指向多维数组的元素。在本节内，将探讨用指针处理多维数组元素的最常用方法。为了简化内容，这里将专注于二维数组，但是每种应用都可以作用在更多维的数组中。

12.4.1 处理多维数组的元素

在8.2节看到，C语言始终按照行为主的顺序存储二维数组；换句话说，先是0行的元素，接着是1行的，并且以此类推下去。 r 行的数组将会有如下的表现形式：



在用指针工作时可以利用这个优势。如果使得指针 p 指向二维数组中的第一个元素（即第0行第0列的元素），就可以通过重复自增 p 的方法访问到数组中的每一个元素。

作为示例，一起来看看把二维数组的所有元素初始化为0的问题。假设数组具有如下的声明：

```
int a[NUM_ROWS][NUM_COLS];
```

显而易见的方法是用嵌套的for循环：

```
int row,col;  
  
for(row = 0; row < NUM_ROWS; row++)  
    for (col = 0; col < NUM_COLS; col++)  
        a[row][col] = 0;
```

但是，如果把 a 看成是整型的一维数组，那么就可以用单独一个循环来替换上述两个循环了：

```
int *p;  
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)  
    *p = 0;
```

231

循环从 p 指向的 $a[0][0]$ 处开始。对 p 的连续自增可以使指针 p 指向 $a[0][1]$ 、 $a[0][2]$ 等等。当 p 达到 $a[0][NUM_COLS-1]$ 时（即0行的最后一个元素），此时再次对 p 自增将使得它指向 $a[1][0]$ ，也就是1行的第一个元素。处理过程直到 p 达到 $a[NUM_ROWS-1][NUM_COLS-1]$ 为止，也就是到达数组中的最后一个元素。

虽然处理二维数组就像用一维数组来欺骗一样，但是它在C语言中是非常合法的。这样做是否是个好主意则要另当别论。像这类方法明显破坏了程序的可读性，但是至少对一些老的编译器来说这种方法在效率方面进行了补偿。然而，使用许多现代的编译器经常极少或没有速度的优势。

12.4.2 处理多维数组的行

只在二维数组的一行内处理元素，该怎么办呢？再次选择使用指针变量 p 。为了访问到 i 行的元素，最好初始化 p 指向的数组 a 中的 i 行的元素0：

```
p = &a[i][0];
```

对于任意的二维数组 a 来说，既然表达式 $a[i]$ 是指向 i 行中第一个元素的指针，那么还可以简化写成

```
p = a[i];
```

为了观察工作的过程，再次调用数组下标和指针算术运算间相互关联的神奇公式：对于任意数组a来说，表达式 $a[i]$ 等价于 $*(\&a + i)$ 。因此，因为 $\&$ 和 $*$ 运算符可以取消，所以 $\&a[i][0]$ 等同于 $\&(*(\&a[i] + 0))$ ，也就等同于 $\&*a[i]$ ，也就等同于 $a[i]$ 。下面把这种简化公式用于下列循环中，循环表明数组a的i行：

```
int a[NUM_ROWS][NUM_COLS], *p, i;
...
for (p = a[i]; p < a[i] + NUM_COLS; p++)
    *p = 0;
```

因为 $a[i]$ 是指向数组a的i行的指针，所以可以把 $a[i]$ 传递给期望一维数组作为实际参数的函数。换句话说，设计使用一维数组的函数也将可以用二维数组中的一行工作。结果是，诸如`find_largest`和`store_zeros`这类的函数会比预期的更加通用。思考最初设计用来找到一维数组中最大元素的`find_largest`函数。现在正好可以简单地用`find_largest`函数来确定二维数组一行内的最大元素：

232

```
largest = find_largest(a[i], NUM_COLS);
```

12.4.3 用多维数组名作为指针

就像一维数组的名字可以用作指针一样，可以忽略数组维数而采用任意数组的名字作为指针。但是，需要小心。思考下列数组：

```
int a[10], b[10][10];
```

虽然可以使用a作为指针指向元素 $a[0]$ ，但是不是说b是指向 $b[0][0]$ 的指针，而是说b是指向 $b[0]$ 的指针。如果从C语言的观点来看待它很有意义的，C语言认为b不是二维数组而是作为一维数组，且这个一维数组的每个元素又是一维数组。在类型项中，a可以用作是`int *`型的指针，而b用作指针时则是具有`int **`型的指针（指向整数指针的指针）。

例如，思考如何使用`find_largest`函数找到下列二维数组的最大元素：

```
int a[NUM_ROWS][NUM_COLS];
```

计划是用`find_largest`函数巧妙地把a考虑成是一维数组。a（数组的地址）将作为`find_largest`函数的第一个实际参数进行传递。`NUM_ROWS * NUM_COLS`（数组a的元素总数量）作为第二个实际参数也将传递：

```
largest = find_largest(a, NUM_ROWS * NUM_COLS); /* WRONG */
```

这条语句不编译，因为a的类型为`int **`而`find_largest`函数期望的实际参数类型是`int *`。正确的调用是：

```
largest = find_largest(a[0], NUM_ROWS * NUM_COLS);
```

Q&A $a[0]$ 指向第0行的第0个元素，而且它的类型为`int *`，所以调用将正确地执行。

问与答

问：我不理解指针的算术运算。如果指针是地址，那么这是否意味着诸如 $p+j$ 这样的表达式是把加上 j 后的地址存储在 p 中呢？(p.157)

答：不是的。标量用于指针算术运算的整数要依赖于指针的类型。例如，如果p的类型是`int *`，那么 $p+j$ 通常既可以用 $2 \times j$ 加上p，也可以用 $4 \times j$ 加上p，依据就是`int`型的值要求的是2个字节还是4个字节。但是，如果p的类型为`double *`，那么 $p+j$ 可能是 $8 \times j$ 加上p，因为`double`型的值通常都是8个字节长。

问：文中提到指针运算只对指向数组元素的指针才有意义，这是什么意思？(p.158)

答：根据C语言的标准，对于并非指向数组的指针，指针的运算是“未定义的”。这并不意味着不能这样做，只是意味着不能保证会发生什么。

问：编写处理数组的循环时，使用数组下标和指针算术运算，哪种更好一些呢？(p.159)

答：因为它依赖于所使用的机器和编译器自身，所以这个问题没有简单的答案。在早期PDP-11机器上的C语言，指针算术运算会产生执行较快的程序。在当今的机器上，采用现今的编译器，数组下标方法常常是很好的，而且有时甚至会更好。底线是：学习这两种方法，然后采用对你正在编写的程序更自然的方法。

*问：从某些地方读到`i[a]`和`a[i]`是一样的，这是真的吗？

答：是的，这是真的，确实很奇怪。对于编译器而言`i[a]`等同于`* (i + a)`，也就是`* (a+i)`（像普通加法一样，指针加法也是可交换的）。而`* (a + i)`也就是`a[i]`。但是请不要在程序中使用`i[a]`，除非你正计划参加下一届“困惑C”比赛。

问：为什么在形式参数的声明中`*a`和`a[]`是一样的？(p.162)

答：上述这两种形式都说明期望实际参数是指针。在这两种情况下（特别是，指针算术运算和数组下标运算）可能在`a`上的操作是相同的。而且，在这两种情况下，可以在函数内给`a`本身赋予新的值。（虽然C语言允许数组变量的名只作为“常量指针”，但是对于作为形式参数的数组名没有这类限制。）

问：当函数有数组`a`作为形式参数时，`*a`或`a[]`哪种格式声明形式参数更好呢？

答：这个问题很棘手。从一种观点看，因为`*a`是不明确的（函数是需要多对象的数组还是指向单独对象的指针？），所以`a[]`是显而易见的选择。另一方面，因为`*a`提示只是传递指针而不是复制数组，所以许多程序员辩称用`*a`声明形式参数会更加精确。根据该函数是使用指针算术运算还是下标运算来访问数组的元素，其他函数在`*a`和`a[]`之间进行切换。（这也是这里将采用的方法。）在实践中，`*a`比`a[]`更通用，所以最好使用前者。不知道是真是假，听说现在Dennis Ritchie把符号`a[]`称为“活化石”，因为它“在使学习者困惑方面起的作用同它提醒程序阅读者方面的作用是相同的”。

问：已经看到C语言中数组和指针之间的紧密联系。是否可以精确地称它们是可互换的呢？

答：不可以。数组型形式参数和指针形式参数是可以互换的，但是作为数组变量不同于指针变量。从技术上说，数组的名字不是指针。更确切地说，需要时C语言编译器会把数组的名字转换为指针。为了更清楚地看出两者的区别，思考一下，当对数组`a`使用`sizeof`运算符时，会发生什么？`sizeof(a)`的值是数组中字节的总数，即每个元素的大小乘以元素的数量。但是，如果`p`是指针变量，那么`sizeof(p)`的值则是用来存储指针值所需的字节数量。

234

*问：说明了如何使用指针处理二维数组的行中的元素。用相似的方法处理列中的元素是否可行？

答：是可行的，但是不像行处理那样容易。因为数组是按行存储的，而不是按列。下面这个循环清楚地表明数组`a`中列`i`的元素：

```
int a [NUM_ROWS] [NUM_COLS], i (*p) [NUM_COLS];

for (p = a; p <= &a [NUM_ROWS-1]; p++)
    (*p) [i] = 0;
```

已经声明了`p`是指向长度为`NUM_COLS`的数组的指针，且此数组的元素为整型的。在表达式`(*p) [NUM_COLS]`中要求`*p`周围有圆括号。如果没有圆括号，那么编译器将会把`p`作为指针的数组而不是指向数组的指针来处理了。表达式`p++`在下一行开始时提前处理`p`。在表达式`(*p) [i]`中，`*p`表示数组`a`的完整一行，所以`(*p) [i]`选择了此行`i`列的元素。在`(*p) [i]`中的圆括号是必需的，因为编译器将把`(*p) [i]`解释为`* (p[i])`。

问：如果`a`是二维数组，为什么可以给`find_largest`函数传递`a[0]`而不是数组`a`本身呢？`a`和`a[0]`不是都指向同一位置（数组开始的位置）吗？(p.164)

答：它们是指向同一位置，作为实际情况，两者都指向元素`a[0][0]`。但是，编译器注意到`a`的类型为`int **`（这不是`find_largest`函数所希望的），而`a[0]`的类型为`int *`。关于类型的这种考虑实际是很好的。如果C语言不是如此挑剔，可能会使编译器注意不到所有的指针错误。

练习

12.1节

1. 假设下列声明是有效的：

```
int a[] = {5, 15, 34, 54, 14, 2, 52, 72};
int *p = &a[1], *q = &a[5];
```

- (a) *(p+3) 的值是多少？
- (b) *(q-3) 的值是多少？
- (c) p-q 的值是多少？
- (d) p<q 的结果是真还是假？
- (e) *p<*q 的结果是真还是假？

235

*2. 假设high、low和middle都是具有相同类型的指针，并且low和high指向数组元素。下面的语句为什么是不合法的，如何修改它？

```
middle = (low + high) / 2;
```

12.2节

3. 在下列语句执行后，数组a的内容会是什么？

```
#define N 10
```

```
int a[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *p = &a[0], *q = &a[N-1], temp;

while (p < q) {
    temp = *p;
    *p++ = *q;
    *q-- = temp;
}
```

4. (a) 编写程序，用来读一条消息，然后反向显示出这条消息。程序的输出格式如下：

```
Enter a message: Don't get mad, ger even.
Reversal is: .neve teg ,dam teg t'nod
```

提示：读消息一次读取一个字符（用getchar函数），并且把这些字符存储在数组中，当数组满了或者读到字符'\n'时停止读操作。

(b) 修改上述程序，用指针来代替整数来跟踪数组中的当前位置。

5. (a) 编写程序，用来读一条消息，然后检查这条消息是否是回文（信息中从左到右的字母和从右到左的字母完全一样）：

```
Enter a message: He lived as a devil, eh?
Palindrome
```

```
Enter a message: Madam, I am Adam.
Not a palindrome
```

忽略所有不是字母的字符。用整型变量来跟踪数组内的位置。

(b) 修改上述程序，使用指针来代替整数跟踪数组的位置。

6. 用指针变量top_ptr代替整型变量top来重新编写栈函数make_empty、is_empty和is_full（>10.2节）。

12.3节

7. 假设a是一维数组而p是指针变量。如果刚执行了赋值操作p = a，那么由于类型不匹配，下列哪些表达式是不合法的？正确的表达式中，哪些为真（即有非零值）？

- (a) p == a[0]
- (b) p == &a[0]

- (c) $*p == a[0]$
 (d) $p[0] == a[0]$

8. 请利用数组名可以用作指针的事实简化练习4中(b)的程序。
 9. 请利用数组名可以用作指针的事实简化练习5中(b)的程序。
 10. 用指针的算术运算代替数组的下标来重新编写下列函数。(换句话说, 消除变量i和所有用[]运算的地方。) 改动尽可能少。

```
int sum_array(int a [], int n)
{
    int i, sum;

    sum = 0;
    for (i = 0; i < n; i++)
        sum += a[i];
    return sum;
}
```

11. 编写下列函数:

```
Bool search(int a[], int n, int key);
```

a是要搜寻的数组, n是数组内元素的数量, 而且key是搜索键。如果key与数组a的某个元素匹配了, 那么search函数必须返回TRUE, 否则返回FALSE。要求使用指针算术运算而不是下标来访问数组元素。

12.4节

12. 8.2节有一个代码段是用两个嵌套的for循环初始化数组ident, 此数组是用作恒等矩阵。请重新编写这段代码, 采用一个指针来逐步访问数组中的元素, 且每次一个元素。提示: 因为不能用row和col来索引变量, 所以不会很容易知道哪里存储1。但是, 可以利用数组的第一个元素必须是1这个事实, 接着N个元素都必须是0, 再接下来的元素是1, 以此类推。用变量来跟踪连续的0的数量, 并把此变量存储起来。当计数达到N时, 就是存储1的时候了。

13. 假设下列数组含有一周24小时的温度读数, 数组的每一行是某一天的读数:

```
int temperatures[7][24];
```

编写语句, 使用search函数在整个temperatures数组中寻找值32。

14. 编写循环用来显示出(练习13中的)temperatures数组中行i存储的所有温度读数。利用指针来访问该行中的每个元素。

15. 编写循环用来显示出(练习13中的)temperatures数组一星期中每一天的最高温度。循环体应该调用find_largest函数, 且一次传递数组的一行。

第13章

字符串

很难从字符串中找到感觉，但它们却是我们能指望的唯一交流纽带。

虽然在前12章中已经使用过char型变量和char型数组，但是仍然缺乏更便捷的方法来处理字符序列（或者，C语言的术语是字符串）。本章将弥补这一点，并介绍字符串常量（在C标准中称为，字符串字面量）和字符串变量。其中，字符串变量可以在程序运行过程中发生改变。

13.1节介绍有关字符串字面量的规则，包括如何在字符串字面量中嵌入转义序列，以及如何分割较长的字符串字面量。13.2节说明声明字符串变量的方法，字符串变量并不等同于字符数组，字符串变量使用特殊的空字符来标示字符串的末尾。13.3节描述了读/写字符串的方法。13.4节讨论用来处理字符串的函数的编写方法。13.5节涵盖了一些C语言函数库中处理字符串的函数。13.6节介绍在处理字符串时经常会采用的惯用法。13.7节描述如何创建一个数组，使这个数组的元素都是指向不同长度字符串的指针，这一节还会说明C语言使用这种数组为程序提供命令行支持的方法。

13.1 字符串字面量

字符串字面量（string literal）^①是用一对双引号括起来的字符序列：

"Put a disk in drive A, then press any key to continue\n"

我们是在第2章中首次遇到字符串字面量的。字符串字面量作为格式串常常出现在printf函数和scanf函数的调用中。

239

13.1.1 字符串字面量中的转义序列

字符串字面量可以像字符常量一样包含转义序列（>7.3.1节）。某些时候，我们在printf函数和scanf函数的格式串中已经使用过转义字符。例如，已经知道字符串中每一个字符\n都会导致光标移到下一行：

"Candy\nIs dandy\nBut liquor\nIs quicker.\n --Ogden Nash\n"

所以输出为

```
Candy
Is dandy
But liquor
Is quicker.
--Ogden Nash
```

虽然字符串字面量中的八进制数和十六进制数的转义序列也都是合法的，但是它们不像字符转义序列那样常见。



请在字符串字面量中小心使用八进制数和十六进制数的转义序列。八进制数的转义序列在3个数字之后结束，或者在第一个非八进制数字符处结束。例如，字符串

① 在C++语言中常称为字符串字面值，或称为常值，或称为字面量。其含义是在程序执行过程中保持不变的数据，在有些C语言书中称之为字串。——译者注

"\1234"包含2个字符 (\123和4)，而字符串"\189"包含3个字符 (\1,8和9)。而另一方面，十六进制数的转义序列则不限制为3个数字，而是直到第一个非十六进制数字符截止。思考一下，如果字符串包含转义序列\x81，那么会出现什么情况。 \x81这个字符在IBM兼容机上对应的字符为ü。字符串"z\x81rich"（“Zürich”）有6个字符(z, \81, r, i, c, 和h)，但是字符串"\x81ber"却只有2个字符 (\x81be和r)。大部分编译器将拒绝接收后面那种字符串，因为计算机通常把十六进制数的转义序列限制在\x0-\x7f(或可能为\x0-\xff)范围之内。

13.1.2 延续字符串字面量

如果发现字符串字面量太长而无法放置在单独一行以内，只要把第一行用字符\结尾，那么C语言就允许在下一行延续字符串字面量。除了（看不到的）末尾的换行符，在同一行不可以有其他字符跟在\后面：

```
printf("Put a disk in drive A, then \
press any key to continue\n");
```

顺便说一下，不只是字符串（虽然通常只用在字符串中），字符\还可以用来分割任何长的符号。

使用\有一个缺陷：字符串字面量必须从下一行的起始位置继续。因此，这就破坏了程序的缩进结构。C语言标准化时引入了更好的处理长字符串字面量的方法。根据C语言的标准，当两条或更多条字符串字面量相连时（仅用空白字符分割），编译器必须把它们合并成单独一条字符串。这条规则允许把字符串分割放在两行或者更多行中：

```
printf("Put a disk in drive A, then"
      "press any key to continue\n");
```

13.1.3 如何存储字符串字面量

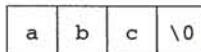
我们经常在printf函数调用和scanf函数调用中用到字符串字面量。但是，当调用printf函数并且用字符串字面量作为参数时，究竟传递了什么呢？为了回答这个问题，需要明白字符串字面量是如何存储的。

从本质上而言，C语言把字符串字面量作为字符数组来处理。当C语言编译器在程序中遇到长度为n的字符串字面量时，它会为字符串字面量分配长度为n+1的内存空间。这块内存空间将用来存储字符串字面量中的字符，以及一个额外的字符——空字符。空字符用来标志字符串的末尾。空字符是ASCII字符集（>附录E）中真正的第一个字符，因此它用转义序列\0来表示。



不要混淆空字符（'\0'）和零字符（'0'）。空字符的ASCII码值为0，而零字符的ASCII码值为48。

例如，字符串字面量"abc"是作为有4个字符的数组来存储的(a, b, c和\0)：



字符串字面量可以为空。字符串""作为单独一个空字符来存储：



既然字符串字面量是作为数组来存储的，那么编译器会把它看作是char *类型的指针。例如，printf函数和scanf函数都接收char *类型的值作为它们的第一个参数。思考下面的例子：

```
printf("abc");
```

当调用printf函数时，会传递"abc"的地址（即指向字母a存储单元的指针）。

13.1.4 字符串字面量的操作

通常情况下可以在任何C语言允许使用`char *`指针的地方使用字符串字面量。例如，字符串字面量可以出现在赋值运算符的右边：

```
241 char *p;
      p = "abc";
```

这个赋值操作不是复制"abc"中的字符，而仅仅是使`p`指向字符串的第一个字符。

C语言允许对指针添加下标，因此可以给字符串字面量添加下标：

```
char ch;
      ch = "abc"[1];
```

`ch`的新值将是字母b。其他可能的下标是0（这将选择字母a），2（字母c），和3（空字符）。字符串字面量的这种特性并不是经常使用，但偶尔也会发现它作用。思考下面的函数，这个函数把0~15的数转换成等价的十六进制的字符形式：

```
char digit_to_hex_char(int digit)
{
    return "0123456789ABCDEF"[digit];
}
```



允许改变字符串字面量中的字符，但是不推荐这么做：

```
char *p = "abc";
      *p = 'b'; /* string literal is now "bbc" */
```

Q&A对于一些编译器而言，改变字符串字面量可能会导致程序运行异常。

13.1.5 字符串字面量与字符常量

只包含一个字符的字符串字面量不同于字符常量。字符串字面量" a"是用指针来表示的，这个指针指向存放字符" a"（以及随后的空字符）的内存单元。字符常量'a'是用整数（字符的ASCII码）来表示的。



不要在需要字符串的时候使用字符（或者反之亦然）。下面的函数调用是合法的：

```
printf("\n");
```

因为`printf`函数期望指针作为它的第一个参数。然而，下面的调用却是非法的：

```
242 printf('\'\n'); /* *** WRONG *** /
```

13.2 字符串变量

一些编程语言为声明字符串变量提供了特殊的`string`类型。C语言采取了不同的方式：只要保证字符串是以空字符结尾的，任何一维的字符数组都可以用来存储字符串。这种方法很简单，但使用起来有很大难度。有时很难辨别是否把字符数组作为字符串来使用。如果编写自己的字符串处理函数，请千万注意要正确地处理空字符。而且，没有比逐个字符地搜索空字符更快捷的方法来确定字符串长度了。

假设需要变量用来存储最多80个字符的字符串。既然字符串会在末尾处需要空字符，那么要声明的变量是含有81个字符的数组：

```
#define STR_LEN 80
```

[惯用法] char str[STR_LEN+1];

注意这里把STR_LEN定义为80而不是81，因此强调的事实是str可以存储最多80个字符的字符串。对宏加一的这种方法是C程序员常用的方式。



当声明用于存放字符串的字符数组时，始终要保证数组的长度比字符串的长度多一个字符。这是因为C语言规定每个字符串都要以空字符结尾。如果没有给空字符预留位置，可能会导致程序运行时出现不可预知的结果，因为C函数库中的函数假设字符串都是以空字符结束的。

声明长度为STR_LEN+1的字符数组并不是意味着它始终会包含长度为STR_LEN的字符串。字符串的长度取决于空字符的位置，而不是取决于用于存放字符串的字符数组的长度。有STR_LEN+1个字符的数组可以存放多种长度的字符串，范围是从空字符串到长度为STR_LEN的字符串。

13.2.1 初始化字符串变量

字符串变量可以在声明时进行初始化：

```
char date1[8] = "June 14";
```

编译器将把字符串"June 14"中的字符复制到数组date1中，然后追加一个空字符从而使date1可以作为字符串使用。date1将如下图所示：

243

| | | | | | | | | |
|-------|---|---|---|---|--|---|---|----|
| date1 | J | u | n | e | | 1 | 4 | \0 |
|-------|---|---|---|---|--|---|---|----|

虽然"June 14"看起来是字符串字面量，但其实不然。C编译器会把它看成是数组初始化式的缩写形式。实际上，我们可以写成：

```
char date1[8] = {'J', 'u', 'n', 'e', ' ', '1', '4', '\0'};
```

相信大家都会认同原来的方式看起来更便于阅读。

如果初始化式太短以致于不能填满字符串变量时将会如何呢？在这种情况下，编译器会添加空字符。因此，在下列声明后：

```
char date2[9] = "June 14";
```

date2将如下显示：

| | | | | | | | | | |
|-------|---|---|---|---|--|---|---|----|----|
| date2 | J | u | n | e | | 1 | 4 | \0 | \0 |
|-------|---|---|---|---|--|---|---|----|----|

大体上来说，这种行为与C语言处理数组初始化式（>8.1.3节）的方法一致。当数组的初始化式比数组本身短时，会把余下的数组元素初始化为0。在把字符数组额外的元素初始化为\0这点上，编译器对字符串和数组遵循相同的规则。

如果初始化式比字符串变量长时又会怎样呢？这对字符串而言是非法的，就如同对数组是非法的一样。然而，C语言允许初始化式（不包括空字符）与变量有完全相同的长度：

```
char date3[7] = "June 14";
```

编译器把初始化式中的字符简单地复制到date3中：

| | | | | | | | |
|-------|---|---|---|---|--|---|---|
| date3 | J | u | n | e | | 1 | 4 |
|-------|---|---|---|---|--|---|---|

没有空间给空字符，所以编译器也不会试图存储一个空字符。



如果计划初始化用来放置字符串的字符数组，一定要确保数组的长度要长于初始化式的长度。否则，编译器将忽略空字符，这将使得数组无法作为字符串使用。

字符串变量的声明可以忽略它的长度。这种情况下，编译器会自动计算长度：

```
char date4[] = "June 14";
```

244 编译器为date4分配8个字符的空间，这足够存储"June 14"中的字符和一个空字符。（事实是date4的长度没有指明不意味着稍候可以改变数组的长度。一旦编译了程序，那么date4的长度就固定是8了。）如果初始化式很长，那么忽略字符串变量的长度是特别有效的，因为手工计算长度很容易出错。

13.2.2 字符数组与字符指针

一起来比较一下下面两个声明：

```
char date[] = "June 14";
```

它声明date是个字符数组。和这个声明相似的是下面这个声明：

```
char *date = "June 14";
```

它声明date是个指向字符串字面量的指针。正因为有了数组和指针之间的紧密关系，才使上面两个声明中的date都可以作为字符串。尤其是，任何期望传递字符数组或字符指针的函数都将接收这两种声明的date作为参数。

然而，需要注意，不能错误地认为上面两种date可以互换。两者之间有着显著的差异：

- 在声明为数组时，就像任意数组元素一样，可以修改存储在date中的字符。在声明为指针时，date指向字符串字面量。而且在13.1节已经看到是不可以修改字符串字面量的。
- 在声明为数组时，date是数组名。在声明为指针时，date是变量，这个变量可以在程序执行期间指向其他字符串。

如果需要可以修改的字符串，那么就要建立字符数组用来存储字符串。这时声明指针变量是不够的。下面的声明使编译器为指针变量分配了足够的内存空间：

```
char *p;
```

可惜的是，它不为字符串分配空间。（这怎么可能呢？因为我们没有指明字符串的长度。）在使用p作为字符串之前，必须把p指向字符数组。一种可能是把p指向已经存在的字符串变量：

```
char str[STR_LEN+1], *p;
```

```
p = str;
```

现在p指向了str的第一个字符，所以可以把p作为字符串使用了。



使用未初始化的指针变量作为字符串是非常严重的错误。考虑下面的例子，它试图创建字符串"abc"：

```
char *p;
p[0]='a';    /*** WRONG ***
p[1]='b';    /*** WRONG ***
p[2]='c';    /*** WRONG ***
p[3]='\0';   /*** WRONG ***/
```

因为p没有初始化，所以我们不知道它指向哪里。把字符a、b、c和\0写入p所指向的内存将会对程序产生无法预期的影响。程序可能没有错误继续运行，或者可能崩溃，或者可能行为异常。

13.3 字符串的读/写

使用printf函数或puts函数来编写字符串是很容易的。读入字符串却有点麻烦，主要是因为输入的字符串可能比用来存储的字符串变量更长。为了一次性读入字符串，可以使用scanf函数或gets函数，也可以每次一个字符的方式读入字符串。

13.3.1 用printf函数和puts函数写字符串

转换说明%s允许printf函数写字符串。参考下面的例子：

```
char str[] = "Are we having fun yet?"

printf("Value of str: %s\n", str);
```

输出会是

```
Value of str: Are we having fun yet?
```

printf函数会逐个写字符串中的字符直到遇到空字符才停止。（如果空字符丢失，printf函数会越过字符串的末尾继续写，直到最终在内存的某个地方找到空字符为止。）

如果只显示字符串的一部分，可以使用转换说明%.ps。这里的p是要显示的字符数量。语句

```
printf("%.6s\n", str);
```

会显示出

```
Are we
```

就像数一样，字符串可以在指定域内显示。转换说明%ms会在大小为m的域内显示字符串。（对于超过m个字符的字符串，printf函数会显示出整个字符串，而不会截断。）如果字符串少于m个字符，则会在域内右对齐输出。为了强制左对齐，可以在m前加一个减号。m值和p值可以组合使用：转换说明%m.ps会使字符串的前p个字符在大小为m的域内显示。

printf函数不是唯一一个字符串输出函数。C函数库还提供puts函数。此函数可以按如下方式使用：

```
puts(str);
```

puts函数只有一个参数，此参数就是需要显示的字符串，参数中没有格式串。在写完字符串后，puts函数总会添加一个额外的换行符，因此显示会移至下一输出行的开始处。

13.3.2 用scanf函数和gets函数读字符串

转换说明%s允许scanf函数读入字符串：

```
scanf("%s", str);
```

在scanf函数调用中，不需要在str前添加运算符&。因为str是数组名，编译器会自动把它当作指针来处理。

调用时，scanf函数会跳过空白字符，然后读入字符，并且把读入的字符存储到str中，直到遇到空白字符为止。scanf函数始终会在字符串末尾存储一个空字符。

用scanf函数读入字符串永远不会包含空白字符。因此，scanf函数通常不会读入一整行输入。换行符会使scanf函数停止读入，空格符或制表符也会产生同样的结果。为了每次读入一整行输入，可以使用gets函数。类似于scanf函数，gets函数把读入的字符放到数组中，然后存储一个空字符。然而，在其他方面gets函数有些不同于scanf函数：

- gets函数不会在开始读字符串之前跳过空白字符（scanf函数会跳过）。
- gets函数会持续读入直到找到换行符才停止（scanf函数会在任意空白字符处停止）。

此外，gets函数会忽略掉换行符，而不会把它存储到数组中，用空字符代替换行符。

为了领会scanf函数与puts函数之间的差异，请考虑下面的程序段：

```
char sentence[SENT_LEN+1];

printf("Enter a sentence: \n");
scanf("%s", sentence);
```

247 假定在提示信息后

Enter a sentence:

用户输入信息

To C, or not to C: that is the question.

scanf函数会把字符串 "To" 存储到sentence中。下一次scanf函数调用将从单词To后面的空格处继续读入这行。

现在假设用gets函数替换掉scanf函数：

```
gets(sentence);
```

当用户输入和先前相同的信息时，gets函数会把字符串

"To C, or not to C: that is the question."

存储到sentence中。



在把字符读入数组时，scanf函数和gets函数都无法检测何时填满数组。因此，它们可能越过数组的边界存储字符，这会导致程序行为异常。通过用转换说明%ns代替%s可以使scanf函数更安全。这里的数字n指出可以存储的最大字符的数量。可惜的是，gets函数天生就是不安全的。fgets函数（>22.5.2节）则是一种更安全的选择。

关于gets函数和puts函数的最后一点提示：既然这些函数比scanf函数和printf函数简单，因此通常运行也就更快。

13.3.3 逐个字符读字符串

对许多程序而言，因为scanf函数和gets函数都有风险且不够灵活，C程序员经常会编写自己的输入函数。通过每次一个字符的方式来读入字符串，这类函数可以提供比标准输入函数更大程度的控制。

如果决定设计自己的输入函数，那么就需要考虑下面这些问题：

- 在开始存储字符串之前，函数应该跳过空白字符吗？
- 什么字符会导致函数停止读取：换行符、任意空白字符、还是其他一些字符？需要存储这类字符还是忽略掉？
- 如果输入的字符串太长以致无法存储，那么程序应该做些什么：忽略额外的字符，还是把它们留给下一次的输入操作？

假定需要的函数不会跳过空白字符，在第一个换行符处（不把换行符存储到字符串中）停止读取，并且忽略额外的字符。函数将有如下原型：

```
int read_line(char str[], int n);
```

str表示用来存储输入的数组，而且n是最大读入字符的数量。如果输入行包含多于n个的字符，read_line函数将忽略多余的字符。read_line函数会返回实际存储在str中的字符数量（在0到n之间的任意数）。不可能总是需要read_line函数的返回值，但是有这个返回值也没问题。

read_line函数主要由一个循环构成，只要str留有空间，那么此循环就逐个读入字符并把它们存储起来。在读入换行符时循环终止。（严格地说，如果getchar函数读入字符失败，也应该终止循环，但是这里暂时忽略这种复杂情况。）Q&A下面是read_line函数的完整定义：

```

int read_line(char str[], int n)
{
    char ch;
    int i = 0;

    while ((ch = getchar()) != '\n')
        if (i < n)
            str[i++] = ch;
    str[i] = '\0'           /* terminates string */
    return i;               /* number of characters stored */
}

```

返回之前，`read_line`函数在字符串的末尾放置了一个空字符。就像`scanf`函数和`gets`函数一样，标准函数会自动在输入字符串的末尾放置一个空字符。然而，如果你正在写自己的输入函数，那么必须要考虑这一点。

13.4 访问字符串中的字符

既然字符串是以数组的方式存储的，那么可以使用下标来访问字符串中的字符。例如，为了对字符串s中的每个字符进行处理，可以设定一个循环来对计数器i进行自增操作，并且通过表达式`s[i]`来选择字符。

假定需要一个函数来统计字符串中空格的数量。利用数组下标可以写出如下函数：

```

int count_spaces(const char s[])
{
    int count = 0, i;

    for (i = 0; s[i] != '\0'; i++)
        if (s[i] == ' ')
            count++;
    return count;
}

```

`s`的声明中包含`const`用来表明`count_spaces`函数不会改变数组。如果`s`不是字符串，`count_spaces`将需要第2个参数来指明数组的长度。然而，因为`s`是字符串，所以`count_spaces`可以通过测试字符是否为空字符来定位`s`的末尾。

许多C程序员不会像例子中那样编写`count_spaces`函数。相反，他们更愿意使用指针来跟踪字符串中的当前位置。就像在12.2节中见到的那样，这种方法对于处理数组来说一直有效，而且在处理字符串方面尤其方便。

下面用指针运算代替数组下标来重新编写`count_spaces`函数。这次不再需要变量`i`，而是利用`s`自身来跟踪字符串中的位置。通过对`s`反复进行自增操作，`count_spaces`函数可以逐次访问字符串中每个字符。下面是`count_spaces`函数的新写法：

```

count_spaces:

int count_spaces(const char *s)
{
    int count = 0;

    for (*s != '\0'; s++)
        if (*s == ' ')
            count++;
    return count;
}

```

注意，`const`没有阻止`count_spaces`函数对`s`的修改，它的作用是阻止函数改变`s`所指向的字符。而且，因为`s`是传递给`count_spaces`函数的参数的副本，所以对`s`进行自增操作不会影响参数。

`count_spaces`函数提出了一些关于如何编写字符串函数的问题：

- 用数组操作或指针操作访问字符串中的字符，哪种方法会更好一些呢？只要使用方便，可以随意使用任意一种，甚至可以混合使用两种方法。在`count_spaces`函数的第2种写法中，不再需要变量*i*，而是把*s*作为指针来对函数进行一些简化。从传统意义上来说，C程序员更倾向于使用指针来处理字符串。
- 字符串形式参数是应该声明为数组还是指针呢？`count_spaces`函数的两种写法说明了这两种选择：第1种写法把*s*声明为数组，而第2种写法则把*s*声明为指针。实际上，这两种声明之间没有任何差异。回顾12.3节的内容就知道，编译器把数组型形式参数当作声明为指针的方式来对待。
- 形式参数的形式（*s[]*或者`*s`）是否会对实际参数的应用产生影响呢？不会的。当调用`count_spaces`函数时，实际参数可以是数组名、指针变量或者字符串字面量。`count_spaces`函数无法说明差异。

250

13.5 使用 C 语言的字符串库

一些编程语言提供的运算符可以对字符串进行复制、比较、合并、选择子串等类似的操作。相反，C语言的运算符根本无法操作字符串。在C语言中把字符串当作数组来处理，因此，对字符串的限制方式和对数组的一样，特别是，它们都不能用C语言的运算符进行复制和比较操作。



直接尝试对字符串进行复制或比较操作会失败。例如，假定`str1`和`str2`有如下声明：

```
char str1[10], str2[10];
```

利用=运算符来把字符串复制到字符数组中是不可能的：

```
str1 = "abc";      /* WRONG */
str2 = str1;        /* WRONG */
```

C语言把这些语句解释为一个指针与另一个指针之间的（非法的）赋值运算。但是，使用=初始化字符数组是合法的：

```
char str1[10] = "abc";
```

这是因为在声明中，=不是赋值运算符。

试图使用关系运算符或判等运算符来比较字符串是合法的，但不会产生预期的结果：

```
if (str1==str2) ... /* WRONG */
```

这条语句把`str1`和`str2`作为指针来进行比较，而不是比较两个数组的内容。因为`str1`和`str2`有不同的地址，所以表达式`str1 == str2`的值一定为0。

幸运的是，所有字符串的操作功能并没有丢失：C语言的函数库为字符串的操作提供了丰富的函数集。这些函数的原型驻留在`<string.h>`（>23.5节）中，所以需要字符串操作的程序应该包含下列内容：

```
#include <string.h>
```

声明在`<string.h>`中的每个函数至少需要一个字符串作为实际参数。把字符串形式参数声明为`char *`类型，同时允许实际参数可以是字符数组、`char *`类型变量或者字符串字面量，上述这些都适合作为字符串。然而，要注意那些没有声明为`const`的字符串形式参数，因为会在调用函数时修改这类形式参数，所以对应的实际参数不应该是字符串字面量。

251

<string.h>中有许多函数。这里将介绍4种应用最广泛的函数。在后续的例子中，假设str1和str2都是用作字符串的字符数组。

13.5.1 strcpy 函数

strcpy（字符串复制）函数在<string.h>中的原型如下：

```
char *strcpy(char *s1, const char *s2);
```

strcpy函数把字符串s2复制给字符串s1。（准确地讲，应该说成是“strcpy函数把s2指向的字符串复制到s1指向的数组中”，但是这种说法太绕弯了。）也就是说，strcpy函数把s2中的字符复制到s1中直到（并且包括）遇到s2中的一个空字符为止。strcpy函数返回s1（即指向目的字符串的指针）。因为不会改变s2指向的字符串，因此声明为const。

strcpy函数的存在弥补了不能使用赋值运算符复制字符串的不足。例如，假设想把字符串"abcd"存储到str1中，就不能使用下面的赋值：

```
str1 = "abcd";           /* *** WRONG *** /
```

因为str1是数组名，而且不能出现在赋值运算的左侧。但是，现在可以用调用strcpy函数：

```
strcpy(str1, "abcd");    /* str1 now contains "abcd" */
```

类似地，不能直接把str1赋值给str2，但是可以调用strcpy：

```
strcpy(str2, str1);      /* str2 now contains "abcd" */
```



在strcpy (str2, str1) 的调用中，strcpy函数无法检查str1指向的字符串的大小是否真地适合str2指向的数组。假设str2指向的字符串长度为n，如果str1指向的字符串有不超过n-1个的字符，那么复制操作可以完成。但是，如果str1指向更长的字符串，那么结果就无法预测了。（由于strcpy函数会一直复制到str1的第一个空字符为止，所以它会越过str2指向的数组的边界继续复制。无论原来存放在数组后面内存中的是什么，都将被覆盖。）尽管执行会慢一点，但是调用strncpy函数（>23.5.1节）仍是一种更安全的复制字符串的方法。

大多数情况下会忽略掉strcpy函数的返回值。但是，有些时候，返回值会比较有用。比如，strcpy函数的调用作为较大表达式的一部分时，使用函数的返回值就非常有用。例如，为了达到和多重赋值同样的效果，可以把一系列strcpy函数调用连起来：

```
strcpy(str2, strcpy(str1, "abcd"));
/* both str1 and str2 now contain "abcd" */
```

252

13.5.2 strcat 函数

strcat（字符串拼接）函数有下面的原型：

```
char *strcat(char *s1, const char *s2);
```

strcat函数把字符串s2的内容追加到字符串s1的末尾，并且返回字符串s1（指向结果字符串的指针）。

下面列举了一些使用strcat函数的例子：

```
strcpy(str1, "abc");
strcat(str1, "def");    /* str1 now contains "abcdef" */
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);    /* str1 now contains "abcdef" */
```

同使用strcpy函数一样，通常忽略strcat函数的返回值。下面的例子说明了可能使用返回值的方法：

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, strcat(str2, "ghi"));
/* str1 now contains "abcdefghi"; str2 contains "defghi" */
```



如果str1指向的数组不是足够大到可以容纳str2指向的字符串中的字符，那么调用strcat(str1, str2)的结果将是不可预测的。考虑下面的例子：

```
char str1[6] = "abc";
strcat(str1, "def");    /*** WRONG ***/
```

strcat函数会试图把字符d、e、f和\0添加到str1中已存储的字符串的末尾。不幸的是，str1仅限于6个字符，这导致strcat函数写到了数组末尾的后面。

13.5.3 strcmp 函数

strcmp（字符串比较）函数有下面的原型：

```
int strcmp(const char *s1, const char *s2);
```

strcmp函数比较字符串s1和字符串s2，然后根据s1是否小于、等于、或大于s2，**Q&A**函数会返回小于、等于、或大于0的值。例如，为了检查str1是否小于str2，可以写成

```
if (strcmp(str1, str2) < 0) /* is str1 < str2? */
...
```

为了检查str1是否小于或等于str2，可以写成

```
if (strcmp(str1, str2) <= 0) /* is str1 <= str2? */
...
```

通过选择适当的关系运算符(<、<=、>、>=)或判等运算符(==、!=)，可以测试str1与str2之间任何可能的关系。

类似于字典中单词的编排方式，strcmp函数利用字典顺序进行字符串比较。更精确地说，如果满足下列两个条件之一，那么strcmp函数就认为s1是小于s2的：

- s1与s2的前*i*个字符一致，但是s1的第(*i*+1)个字符小于s2的第(*i*+1)个字符。例如，“abc”小于“bcd”，而“abc”小于“abd”。
- s1的所有字符与s2的字符一致，但是s1比s2短。例如，“abc”小于“abcd”。

当比较两个字符串中的字符时，strcmp函数会查看表示字符的数字码。一些底层字符集的知识可以帮助预测strcmp函数的结果。假定我们的机器使用的是ASCII字符集(见附录E)，下面是strcmp函数会遵守的一些规则：

- 所有的大写字母都小于所有的小写字母。(在ASCII码中，65~90的编码表示大写字母；97~122的编码表示小写字母。)
- 数字小于字母。(48~57的编码表示数字。)
- 空格符小于所有打印字符。(ASCII码中空格符的值是32。)

13.5.4 strlen 函数

strlen（求字符串长度）函数有下面的原型：

```
size_t strlen (const char *s);
```

定义在C函数库中的size_t函数(见21.3节)是无符号整数类型(通常是unsigned int或unsigned long int)。除非在处理极长的字符串，否则不需要关心这种技术细节。我们可以简单地把返回值作为整数处理。

`strlen`函数返回字符串s的长度。更精确地说，`strlen`函数返回s中第一个空字符前的字符的个数，但不包括第一个空字符。例如：

```
int len;

len = strlen("abc");      /* len is now 3 */
len = strlen("");         /* len is now 0 */
strcpy(strl, "abc");
len = strlen(strl);       /* len is now 3 */
```

254

最后的例子说明了一个重点：当用数组作为函数的实际参数时，`strlen`函数不会测量数组本身的长度，而是返回存储在数组中的字符串的长度。

13.5.5 程序：显示一个月的提示列表

为了说明C语言字符串函数库的用法，现在来看一个程序。这个程序会显示一个月的每日提示列表。用户需要输入一系列提示，每条提示都要有一个前缀来说明是一个月中的哪一天。当用户用0代替有效的天录入时，程序会显示出录入的全部提示的列表，并且这些提示是按天排序的。下面是与这个程序的对话信息：

```
Enter day and reminder: 24 Susan's birthday
Enter day and reminder: 5 6:00 - Dinner with Marge and Russ
Enter day and reminder: 26 Movie - "Chinatown"
Enter day and reminder: 7 10:30 - Dental appointment
Enter day and reminder: 12 Movie - "Dazed and Confused"
Enter day and reminder: 5 Saturday class
Enter day and reminder: 12 Saturday class
Enter day and reminder: 0

Day Reminder
5 Saturday class
5 6:00 - Dinner with Marge and Russ
7 10:30 - Dental appointment
12 Saturday class
12 Movie - "Dazed and Confused"
24 Susan's birthday
26 Movie - "Chinatown"
```

总体策略不是很复杂：程序需要读入一系列天和提示的组合，并且按照顺序进行存储（按日期排序），然后显示出来。为了读入天，将使用`scanf`函数。为了读入提示，将使用`read_line`函数（13.3节）。

把字符串存储在二维的字符数组中，数组的每一行包含一条字符串。在程序读入某天以及相关的提示后，通过使用`strcmp`函数进行比较来查找数组从而确定这一天所在的位置。然后，程序会使用`strcpy`函数把此位置之后的所有字符串往后移动一个位置。最后，程序会把这一天复制到数组中，并且调用`strcat`函数来把提示附加到这一天后面。（天和提示在此之前是分开存放的。）

当然，总会有少量略微复杂的地方。例如，希望天在两个字符的域中右对齐以便它们的个位可以对齐。有很多种方法可以解决这个问题。这里选择用`scanf`函数把日期读入到整型变量中，然后调用`sprintf`函数（>22.8节）把天转换成字符串格式。除了把输出写到字符串中之外，`sprintf`是个类似于`printf`的库函数。调用

```
sprintf(day_str, "%2d", day);
```

255

是把`day`的值写到`day_str`中。因为`sprintf`在写完后会自动添加一个空字符，所以`day_str`会包含一个由空字符结尾的合法字符串。

另一个复杂的地方是确保用户没有输入两位以上的数字，为此将使用下列`scanf`函数调用：

```
scanf("%2d", &day);
```

即使输入有更多的数字，在%与d之间的数2也会通知scanf函数在读入两个数字后停止。

考虑到上述细节，程序如下所示：

remind.c

```
/* Prints a one-month reminder list */

#include <stdio.h>
#include <string.h>

#define MAX_REMIND 50
#define MSG_LEN 60

int read_line(char str[], int n);

main()
{
    char reminders[MAX_REMIND][MSG_LEN+3];
    char day_str[3], msg_str[MSG_LEN+1];
    int day, i, j, num_remind = 0;

    for (;;) {
        if (num_remind == MAX_REMIND) {
            printf("-- No space left --\n");
            break;
        }

        printf("Enter day and reminder: ");
        scanf("%2d", &day);
        if (day == 0)
            break;
        sprintf(day_str, "%2d", day);
        read_line(msg_str, MSG_LEN);

        for (i = 0; i < num_remind; i++)
            if (strcmp(day_str, reminders[i]) < 0)
                break;
        for (j = num_remind; j > i; j--)
            strcpy(reminders[j], reminders[j-1]);

        strcpy(reminders[i], day_str);
        strcat(reminders[i], msg_str);

        num_remind++;
    }

    printf("\nDay Reminder\n");
    for (i = 0; i < num_remind; i++)
        printf(" %s\n", reminders[i]);
}

return 0;
}

int read_line(char str[], int n)
{
    char ch;
    int i = 0;

    while ((ch = getchar()) != '\n')
        if (i < n)
            str[i++] = ch;

    str[i] = '\0';
    return i;
}
```

256

虽然程序remind.c很好地说明了strcpy函数、strcat函数和strcmp函数，但是作为实际的提示程序，它还缺少一些东西。显然有许多需要完善的地方，完善的范围是从次要改进调整到主要改进（例如，当程序终止时在文件中保存数据库）。本章末尾的练习和后续各章将会讨论这些改进。

13.6 字符串惯用法

处理字符串的函数是特别丰富的惯用法资源。本节将会探索其中两种最著名的惯用法，并利用它们编写strlen函数和strcat函数。（当然，永远都不需要编写这两个函数，因为它们已经是标准函数库的一部分内容了。但是有可能要编写类似的函数。）

本节使用的简洁风格是在许多C程序员中流行的风格。即使不准备在自己的程序中使用，也应该掌握这种风格。因为很可能会在其他程序员编写的程序中遇到。

13.6.1 搜索字符串的结尾

许多字符串操作需要搜索字符串的结尾。strlen函数就是一个重要的例子。下面的strlen函数搜索字符串参数的结尾，并且使用一个变量来跟踪字符串的长度：

257

```
size_t strlen(const char *s)
{
    size_t n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

随着指针s从左至右扫描整条字符串，变量n始终跟踪当前已经扫描的字符数量。当s最终指向一个空字符时，n所包含的值就是字符串的长度。

现在看看是否能精简strlen函数的定义。首先，把n的初始化移到它的声明中：

```
size_t strlen(const char *s)
{
    size_t n = 0;

    for (; *s != '\0'; s++)
        n++;
    return n;
}
```

接下来注意到条件`*s != '\0'`与`*s != 0`是一样的，因为空字符的ASCII码值就是0。但是测试`*s != 0`与测试`*s`是一样的，两者都在`*s`不为0时结果为真。这些发现引出strlen函数的又一个版本：

```
size_t strlen(const char *s)
{
    size_t n = 0;

    for ( ; *s; s++)
        n++;
    return n;
}
```

然而，就如同在12.2节中见到的那样，在同一个表达式中对s进行自增操作并且测试`*s`是可行的：

```
size_t strlen(const char *s)
{
    size_t n = 0;
```

```

    for ( ; *s++ ; )
        n++ ;
    return n ;
}

```

用while语句替换for语句，可以得到如下版本的strlen函数：

```

size_t strlen(const char *s)
{
    size_t n = 0;

    while (*s++)
        n++;
    return n;
}

```

虽然已经对strlen函数进行了相当大地精简，但是可能仍没有提高它的运行速度。至少对于一些编译器来说下面的版本确实会运行更快一些：

```

size_t strlen(const char *s)
{
    const char *p = s;

    while (*s)
        s++;
    return s - p;
}

```

此版本的strlen函数通过定位空字符位置的方式来计算字符串的长度，然后用空字符的地址减去字符串中第一个字符的地址。提高运行速度不需要在while循环中增加n。请注意在p的声明中出现了单词const，如果没有它，编译器会注意到把s赋值给p会给s指向的字符串造成一定风险。

语句

[惯用法] while (*s)
 s++;

和相关的

[惯用法] while (*s++)
 ;

都是意味着“查找字符串结尾的空字符”的惯用法。第一个版本最终使s指向了空字符。第二个版本更加简洁，但是最后使s正好指向空字符后面的位置。

13.6.2 复制字符串

复制字符串是另一种常见操作。为了介绍C语言“字符串复制”这种惯用法，这里将开发strcat函数的一种写法。就先从直接但有些冗长的strcat函数写法开始：

```

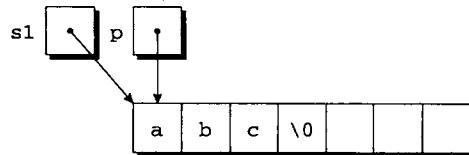
char *strcat(char *s1, const char *s2)
{
    char *p;

    p = s1;
    while (*p != '\0')
        p++;
    while (*s2 != '\0'){
        *p = *s2;
        p++;
        s2++;
    }
    *p = '\0';
    return s1;
}

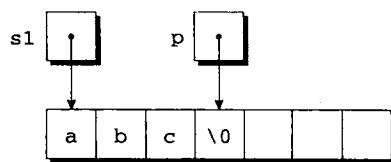
```

`strcat`函数的这种写法采用了两步算法：(1) 查找字符串`s1`末尾空字符的位置，并且使指针`p`指向它；(2) 把字符串`s2`中的字符逐个复制到`p`所指向的位置。

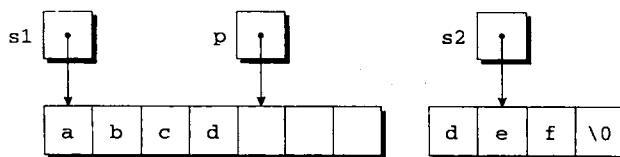
函数中的第一个`while`语句实现了第1步。把`p`设定为指向`s1`的第一个字符。参考下图，假设`s1`指向字符串“abc”：



接着`p`开始自增直到指向空字符为止。循环终止时，`p`必须指向空字符：

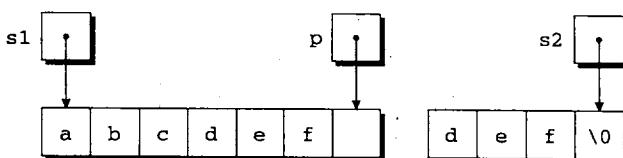


第二个`while`语句实现了第(2)步。循环体把`s2`指向的一个字符复制到`p`指向的地方，接着`p`和`s2`都进行自增。如果`s2`最初指向字符串“def”，下面显示了第一次循环后的样子：



260

当`s2`指向空字符时循环终止：



在`p`指向的位置放置空字符之后，`strcat`函数返回。

通过类似于对`strlen`函数所采用的方法，可以简化`strcat`函数的定义，得到下列写法：

```
char *strcat (char *s1, const char *s2)
{
    char *p = s1;

    while (*p)
        p++;
    while (*p++ = *s2++)
        ;
    return s1;
}
```

改进的`strcat`函数核心是“字符串复制”的习惯方法：

[惯用法] `while (*p++ = *s2++)`

如果忽略了两个`++`运算符，那么圆括号中的表达式会简化为普通的赋值表达式：

`*p = *s2`

这个表达式把`s2`指向的字符复制到`p`所指向的地方。正是由于这两个`++`运算符，所以赋值之后`p`和`s2`都进行了自增。重复执行此表达式所产生的效果就是把`s2`指向的一系列字符复制到`p`所指向的地方。

但是什么会促使循环终止呢？由于圆括号中的主要运算符是赋值运算符，所以`while`语句会测试赋值表达式的值，也就是测试复制的字符。除空字符以外的所有字符的测试结果都为真，因此，循环只有在复制空字符后才会终止。而且由于循环是在赋值之后终止，所以不需要单独一条语句用来在新字符串的末尾添加空字符。

13.7 字符串数组

261 现在来看一个在使用字符串时经常遇到的问题：存储字符串数组的最佳方式是什么？最明显的解决方案是创建二维的字符数组，然后按照每行一个字符串的方式把字符串存储到数组中。考虑下面的例子：

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto");
```

虽然允许省略`planets`数组中行的个数（因为这个数很容易从初始化式中元素数量求出），但是C语言要求说明列的个数。下面是数组可能的形式：

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|----|----|----|----|
| 0 | M | e | r | c | u | r | y | \0 |
| 1 | V | e | n | u | s | \0 | \0 | \0 |
| 2 | E | a | r | t | h | \0 | \0 | \0 |
| 3 | M | a | r | s | \0 | \0 | \0 | \0 |
| 4 | J | u | p | i | t | e | r | \0 |
| 5 | S | a | t | u | r | n | \0 | \0 |
| 6 | U | r | a | n | .u | s | \0 | \0 |
| 7 | N | e | p | t | u | n | e | \0 |
| 8 | P | l | u | t | o | \0 | \0 | \0 |

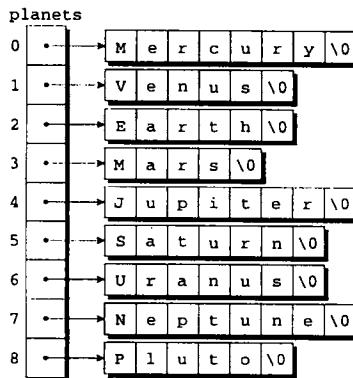
不是所有的字符串都足够填满数组的一整行，所以C语言用空字符来填补。因为只有3个行星名字达到8个字符的长度要求（包括末尾的空字符），所以这样的数组有一点浪费空间。`remind.c`程序（13.5节）就是这种浪费的代表，它把提醒信息按行存储到二维字符数组中，且为每条提醒信息都分配了60个字符的空间。在示例中，提醒信息的长度范围是在14个字符到33个字符之间，所以浪费的空间数量相当可观。

因为大部分字符串集都是长短字符串的混合，所以这些例子所暴露的低效性是在处理字符串时经常遇到的问题。我们需要的是参差不齐的数组（ragged array），即数组的每一行有不同的长度。C语言本身不提供这种“参差不齐的数组类型”，但它确实提供了模拟这种数组类型的工具。秘诀就是建立一个特殊的数组，这个数组的元素都是指向字符串的指针。

下面是planets数组的另外一种写法，这次看成是指向字符串的指针的数组：

```
char *planets[] = {"Mercury", "Venus", "Earth",
    "Mars", "Jupiter", "Saturn",
    "Uranus", "Neptune", "Pluto");
```

改动不是很大，只是简单去掉了一对方括号，并且在planets前加了一个星号。但是这对planets存储方式产生的影响却很大：



planets的每一个元素都指向以空字符结尾的字符串的指针。虽然必须为planets数组中的指针分配空间，但是字符串中不再有任何浪费的字符。

为了访问其中一个行星名字，只需要给出planets数组的下标。访问行星名字中的字符的方式和访问二维数组元素的方式相同，这都要感谢指针和数组之间的紧密关系。例如，为了在planets数组中搜寻以字母M开头的字符串，可以使用下面的循环：

```
for (i = 0; i < 9; i++)
    if (planets[i][0] == 'M')
        printf("%s begins with M\n", planets[i]);
```

13.7.1 命令行参数

在运行程序时，会经常需要提供一些信息——文件名或者是改变程序行为的开关。考虑UNIX的ls命令。如果我们按如下显示方式运行ls

```
ls
```

将会显示当前目录中的文件名。（对应的DOS命令是dir。）但是如果替换成

```
ls -l
```

那么ls会显示一个“很长的”（详细的）文件列表，包括显示每个文件的大小、文件的所有者、文件最后改动的日期和时间等。为了进一步改变ls的行为，可以指定只显示一个文件的详细信息：

```
ls -l remind.c
```

ls将会显示文件名为remind.c的详细信息。

不仅是操作系统命令，所有程序都有命令行信息。**Q&A**为了能够访问这些命令行参数(command-line argument)(标准中称为程序参数)，必须把main函数定义为含有两个参数的函数，**Q&A**这两个参数通常命名为argc和argv：

```
main(int argc, char *argv[])
{
    ...
}
```

argc（“参数计数”）是命令行参数的数量（包括程序名本身）。argv（“参数向量”）是指向命令行参数的指针数组，这些命令行参数以字符串的形式存储。argv[0]指向程序名，而从argv[1]

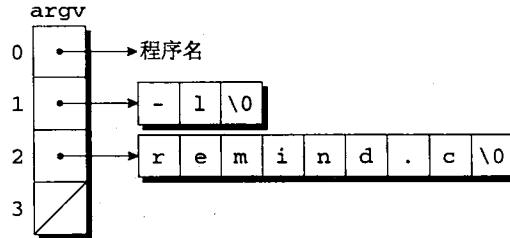
到`argv[argc-1]`则指向余下的命令行参数。

`argv`有一个附加元素，即`argv[argc]`，这个元素始终是一个空指针。空指针是一种不指向任何内容的特殊指针。后面的章中会讨论空指针（>17.1.2节）。但是目前只需要知道宏NULL代表空指针就够了。

如果用户输入了下列命令行：

```
ls -l remind.c
```

那么`argc`将为3，`argv[0]`将指向含有程序名的字符串，`argv[1]`将指向字符串“-l”，`argv[2]`将指向字符串“remind.c”，而`argv[3]`将为空指针：



这幅图没有详细说明程序名，因为在不同的操作系统上程序名可能会包括路径或其他信息。如果程序名不可用，那么`argv[0]`会指向空字符串。

因为`argv`是指针数组，所以已经知道访问命令行参数的方法了。典型做法是，期望有命令行参数的程序将会设置循环来按顺序检查每一个参数。设定这种循环的方法之一就是使用整型变量作为`argv`数组的索引。例如，下面的循环每行一条地显示命令行参数：

```
int i;
for (i=1; i<argc; i++)
    printf("%s\n", argv[i]);
```

264

另一种方法是构造一个指向`argv[1]`的指针，然后对指针重复进行自增操作来逐个访问数组余下的元素。因为数组的最后一个元素始终是空指针，所以循环可以在找到数组中第一个空指针时停止：

```
char **p;
for (p = &argv[1]; *p != NULL; p++)
    printf("%s\n", *p);
```

因为`p`是指向字符指针的指针，所以必须小心使用。把`p`设为`&argv[1]`是很有意义的，因为`argv[1]`是一个指向字符的指针，所以`&argv[1]`就是指向指针的指针了；因为`*p`和`NULL`都是指针，所以测试`*p != NULL`是没有问题的；对`p`进行自增操作看起来也是对的因为`p`指向数组元素，所以对它进行自增操作将使`p`指向下一个元素；显示`*p`的语句也是合理的，因为`*p`是一个指向字符的指针。

13.7.2 程序：核对行星的名字

下一个程序`planet.c`举例说明了访问命令行参数的方法。设计此程序的目的是为了测试一系列字符串，从而找出哪些字符串是行星的名字。程序执行时，用户将把测试的字符串放置在命令行中：

```
planet Jupiter venus Earth fred
```

程序会指出每个字符串是否是行星的名字。如果是，程序还将显示行星的编号（把最靠近太阳的行星编号为1）：

```
jupiter is planet 5
venus is not a planet
Earth is planet 3
fred is not a planet
```

注意，除非字符串的首字母大写并且其余字母小写，否则程序不会认为字符串是行星的名字。

planet.c

```
/* Checks planet names */

#include <stdio.h>
#include <string.h>

#define NUM_PLANETS 9

main(int argc, char *argv[])
{
    char *planets[] = {"Mercury", "Venus", "Earth",
                       "Mars", "Jupiter", "Saturn",
                       "Uranus", "Neptune", "Pluto"};
    int i, j;
    for (i = 1; i < argc; i++) {
        for (j = 0; j < NUM_PLANETS; j++) {
            if (strcmp(argv[i], planets[j]) == 0) {
                printf("%s is planet %d\n", argv[i], j+1);
                break;
            }
        }
        if (j == NUM_PLANETS)
            printf("%s is not a planet\n", argv[i]);
    }
    return 0;
}
```

265

程序会依次访问每个命令行参数，把它与planets数组中的字符串进行比较，直到找到匹配的名字或者到了数组的末尾才停止。程序中最有趣的部分是调用strcmp函数，此函数的参数是argv[i]（指向命令行参数的指针）和planets[j]（指向行星名的指针）。

问与答

问：字符串字面量可以有多长？

答：按照C语言的标准，编译器必须最少支持509个字符长的字符串字面量。（没错，就是509。不要怀疑。）许多编译器会允许更长的字符串字面量。

问：为什么不把字符串字面量称为“字符串常量”？

答：因为它们并不需要一定是常量。由于通过指针访问字符串字面量，所以没有限制程序修改字符串字面量中的字符。

问：改变字符串字面量似乎没有什么危险。为什么标准C不建议这么做？(p.170)

答：一些编译器试图通过存储相同字符串字面量的单独一份副本来自节约内存。考虑下面的例子：

```
char *p = "abc", *q = "abc";
```

一些编译器会只存储"abc"一次，并且把p和q都指向此字符串字面量。如果试图通过指针p改变"abc"，那么q所指向的字符串也会受到影响。毫无疑问，这可能会导致一些非常讨厌的错误。

尽管标准C禁止修改字符串字面量，有些程序员也明知道编译器只存储唯一一份字符串字面量，却仍然会这么做。不过，我建议避免这样的用法，因为它降低了程序的可移植性。

问：是否每个字符数组都应该包含空字符的空间呢？

答：不是必需的。因为不是所有的字符数组都作为字符串使用。为空字符预留的空间（并实际在数组中存储一个空字符）只针对于计划调用以空字符结尾的字符串的函数情况。

266

如果只对独立的字符进行处理，那么就不需要空字符。例如，可能有一个作为翻译表的字符数组：

```
char translation_table[128];
```

对这个数组唯一可以执行的操作就是使用下标。这里不会把translation_table看成是字符串，而且也不会对它执行任何字符串操作。

问：如果printf函数和scanf函数需要char *类型的变量作为它们的第一个实际参数，那么是否意味着可以用字符串变量代替字符串字面量作为实际参数呢？

答：可以。如同下面例子说明的那样：

```
char fmt[] = "%d\n";
int i;
```

```
printf(fmt, i);
```

这种能力为一些有趣的实现提供了可能。例如，把格式串作为输入读取。

问：如果想让printf函数输出字符串str，是否可以如下例所示那样不仅仅把str用作格式串？

```
printf(str);
```

答：可以，但是很危险。如果str包含字符%，那么就不会获得预期的结果。因为printf函数会把%认定为转换说明的开始。

*问：read_line函数如何检测getchar函数读入字符是否失败？(p.174)

答：如果不能读入字符，可能是因为错误，也可能是因为文件尾。getchar函数返回int型的值EOF(>22.4节)。下面是改进后的read_line函数，此函数用来检测getchar函数的返回值是否为EOF。改动部分用粗体标记：

```
int read_line(char str[], int n)
{
    int ch;
    int i = 0;

    while ((ch = getchar()) != '\n' && ch != EOF)
        if (i < n)
            str[i++] = ch;
        str[i] = '\0';
        return i;
}
```

267

问：为什么strcmp函数会返回一个小于、等于或大于0的数？返回值有什么意义吗？(p.178)

答：strcmp函数的返回值可能是源于函数的传统编写方式。思考Kernighan和Ritchie的The C Programming Language一书中的写法：

```
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i]==t[i]; i++)
        if (s[i] == '\0')
            return 0;
        return s[i] - t[i];
}
```

函数的返回值是字符串s和字符串t中第一个“不匹配”字符的差。如果s指向的字符串“小于”t指向的，那么结果为负数。而如果s指向的字符串“大于”t指向的，则结果为正数。但是，不保证strcmp函数就是按照这种方法编写的，所以最好不要假设返回值有什么特殊的意义。

问：在尝试编译strcmp函数中的while语句时，编译器给出警告“possibly incorrect assignment”。是做错了什么吗？

```
while (*p++ = *s++)
```

答：没有。如果在通常需要采用==的地方使用了=，许多编译器都会给出警告，但不是所有的编译器都会这样做。这条警告消息95%的情况下是正确的，而且如果留意到它会节约大量的调试时间。可惜

的是，此消息在这个特殊的示例中是无效的。我们确实真的打算使用=，而不是==。为了除去警告，可以按如下方式重写while语句：

```
while ((*p++ = *s2++) != 0)
;
```

因为while语句通常测试*p++ = *s2++是否不为0，所以这样做没有改变while语句的意思。但是警告消息却没有了，原因是while语句现在测试的是条件，而不是赋值了。

问：**strlen函数和strcat函数是否真的像13.6节显示的那样编写？**

答：尽管对编译器供应商来说，用汇编语言代替C语言来编写这些函数和许多其他字符串函数是很普遍的做法，但是像13.6节显示的那样编写这些函数是有可能的。因为经常使用并且必须能处理任意长度的字符串，所以字符串函数的处理速度是越快越好。利用CPU可能提供的任何特别的字符串处理指令，用汇编语言编写的这些函数能够获得很高的效率。

问：为什么C标准采用术语“程序参数”而不是“命令行参数”？(p.185)

答：程序不总是在命令行中运行的。例如，在窗口环境下，程序是通过点击鼠标来启动的。在这类环境中，虽然可能有给程序传递信息的其他方式，但是没有传统意义上的命令行了。术语“程序参数”可以适用于这样的环境。

问：是否必须使用argc和argv作为main函数的参数名？(p.185)

答：不是的。使用argc和argv作为名字仅仅是一种习惯，而不是语言本身的要求。

问：看到argv的声明用**argv代替了*argv[]。这是合法的？

答：当然合法。在声明形式参数时，不管a的元素类型是什么，*a的写法和a[]的写法总是一样的。

问：已经见过如何创建用来指向字符串字面量的指针数组。指针数组是否还有其他应用？

答：有的。虽然到目前为止仅用指针数组指向字符串，但这不是指针数组的唯一应用。无论数据是否是数组的形式，都可以同样简单地创建指向任何其他数据类型的指针数组。指针数组在动态存储分配(>17.1节)联合上是特别有用的。

练习

13.3节

1. 下面的函数调用应该是写出单独一个换行符，但是其中有一些是错误的。请指出哪些调用是错误的，并说明理由。

- | | | |
|-------------------------|--------------------|-----------------|
| (a) printf("%c", '\n'); | (e) printf('\n'); | (i) puts('\n'); |
| (b) printf("%c", "\n"); | (f) printf("\n"); | (j) puts("\n"); |
| (c) printf("%s", '\n'); | (g) putchar('\n'); | (k) puts(""); |
| (d) printf("%s", '\n'); | (h) putchar("\n"); | |

2. 假设p的定义如下所示：

```
char *p = "abc";
```

下列哪些函数调用是合法的？请说明每个合法的函数调用的输出，并解释为什么其他的是非法的。

- | | |
|------------------|---------------|
| (a) putchar(p); | (c) puts(p); |
| (b) putchar(*p); | (d) puts(*p); |

*3. 假设按如下方式调用scanf函数：

```
scanf("%d%s%d", &i, s, &j);
```

如果用户输入12abc34 56def78，那么调用后i、s和j的值分别是多少？(假设i和j是int型变量，s是字符数组。)

4. 按照下述要求分别实现read_line函数：

- (a) 在开始存储输入字符前跳过空白字符。
- (b) 在读入第一个空白字符时停止。提示：调用isspace函数(>23.4.1节)来检查字符是否为空白字符。
- (c) 在读入第一个换行符时停止，然后把换行符存储到字符串中。

(d) 把没有空间存储的字符留下以备后用。

13.4节

5. (a) 编写名为strcap的函数用来把参数中的字母都改为大写字母。参数是空字符结尾的字符串，且此字符串包含任意的ASCII字符，不仅是字母。使用数组下标的方式访问字符串中的字符。提示：使用toupper函数（>23.4.3节）把每个字符转换成大写。
- (b) 重写strcap函数，这次使用指针来访问字符串中的字符。
6. 编写名为censor的函数，用来把字符串中出现的每一处字母"foo"替换为"xxx"。例如，字符串"food fool"会变为"xxxd xxxl"。在不失清晰性的前提下程序越短越好。
- *7. 下面程序的输出是什么？

```
#include <stdio.h>

main ()
{
    char s[] = "Hsjodi", *p;

    for (p = &s[5]; p >= s; p--) --*p;
    puts (s);
    return 0 ;
}
```

*8. 函数f如下所示：

```
int f(char *s, char *t)
{
    char *p1, *p2;

    for (p1 = s; *p1; p1++) {
        for (p2 = t; *p2; p2++)
            if (*p1 == *p2) break;
            if (*p2 == '\0') break;
    }
    return p1 - s;
}
```

(a) f("abcd", "babc")的值是多少？

(b) f("abcd", "bcd")的值是多少？

(c) 通常情况下，当传递两个字符串s和t时，函数的返回值是什么？

13.5节

9. 假设str是字符数组，下面哪条语句与其他3条语句不等价？

(a) *str = 0; (c) strcpy(str, "");
 (b) str[0] = '\0'; (d) strcat(str, "");

- *10. 在执行下列语句后，字符串str的值是什么？

```
strcpy(str, "tire-bouchon");
strcpy(&str[4], "d-or-wi";
strcat(str, "red?");
```

11. 在执行下列语句后，字符串s1与s2的值各是什么？

```
strcpy(s1, "computer");
strcpy(s2, "science");
if (strcmp(s1, s2) < 0)
    strcat(s1, s2);
else
    strcat(s2, s1);
s2[strlen(s2)-6] = '\0';
```

12. 下面的函数假设用来创建字符串的相同副本。请指出这个函数中的错误？

```
char *strdup(const char *p)
{
    char *q;
```

```
strcpy (q, p)
return q;
}
```

13. 在本章的末尾“问与答”小节说明了利用数组下标的方式来编写strcmp函数的方法。请用指针算术运算的方法来修改此函数。
14. 编写程序用来找到一组单词中“最大”单词和“最小”单词。当用户输入单词后，程序根据字典的排序顺序决定排在最前面和最后面的单词。当用户输入了4个字母的单词时，程序必须停止读入。假设所有单词都不超过20个字母。程序与用户的交互显示如下所示：

```
Enter word: dog
Enter word: zebra
Enter word: rabbit
Enter word: catfish
Enter word: walrus
Enter word: cat
Enter word: fish
Smallest word: cat
Largest word: zebra
```

271

提示： 使用两个名为smallest_word和largest_word的字符串来记录当前输入的“最小”单词和“最大”单词。每次用户输入新单词，就用strcmp函数把它与smallest_word进行比较。如果新的单词比smallest_word“小”，就用strcpy函数把新单词保存到smallest_word中。用类似的方式与larges_word也进行比较。用strlen函数来判断用户输入4个字母的单词的时候。

15. 按如下方式改进remind.c程序：

- 如果对应的天为负数或大于31，那么先是显示出错误信息，然后忽略提示。
提示： 使用continue语句。
- 允许用户输入天、24小时格式的时间（可能空白）和提示。显示的提示列表必须先按天排序存储，然后再根据时间排序存储。（原始的remind.c程序允许用户输入时间，但是它把时间作为提示的一部分来处理。）
- 程序需显示一年的提示列表。要求用户按照月/日的格式输入日期。

13.6节

16. 利用13.6节的方法来精简count_space函数（见13.4节）。特别是要用while循环替换for语句。

13.7节

17. 修改8.2节的deal.c程序，使它显示出牌的全名：

```
Enter number of cards in hand: 5
Your hand:
Seven of clubs
Two of spades
Five of diamonds
Ace of spades
Two of hearts
```

提示： 用指向字符串的指针数组来替换数组rank_code和数组suit_code。

18. 编写名为reverse.c的程序，用来把命令行参数按反序输出。如果按下述方式执行程序：

```
reverse void and null
产生的输出应为
null and void
```

19. 编写名为sum.c的程序，用来对命令行参数求和。假设参数都是整数。如果按下述方式执行程序：

```
sum 8 24 62
```

产生的输出应为

```
Total: 94
```

提示： 用atoi函数（►26.2.1节）把每个命令行参数从字符串格式转换为整数格式。

20. 改进程序planets.c，使它在比较命令行参数与planets数组中的字符串时忽略大小写。

272