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

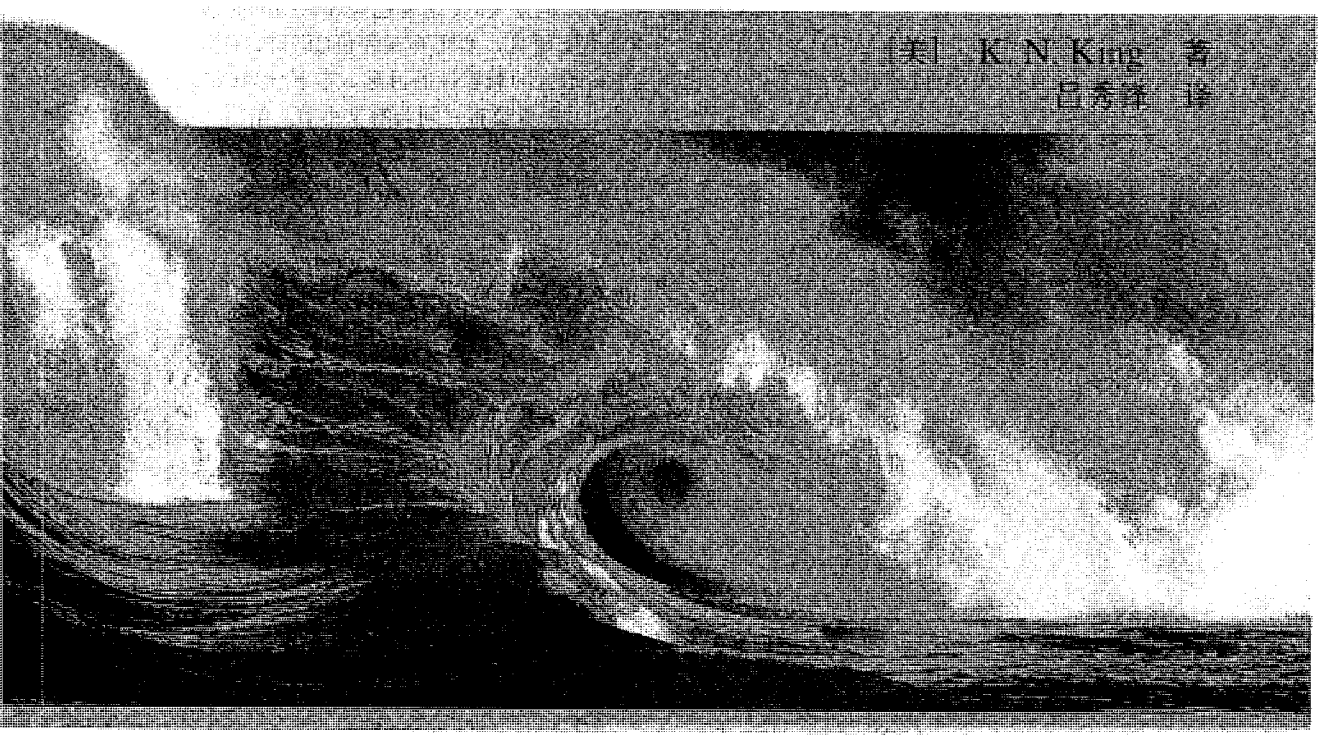
[美] K. N. King 著
吕秀锋 译

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
吕秀锋 译

人民邮电出版社
北 京

www.TopSage.com

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 | 软考历年真是解析与答案

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…

大家网
TopSage.com

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

www.TopSage.com

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

www.TopSage.com

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

www.TopSage.com

大家网
TopSage.com

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。

www.TopSage.com

- **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：

www.TopSage.com

- **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UINX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**（⚠）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。

www.TopSage.com

大家网 TopSage.com

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

www.TopSage.com

大家网
TopSage.com

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

www.TopSage.com

5.3 switch语句……53
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
练习……58

第6章 循环……61

6.1 while语句……61
6.1.1 无限循环……62
6.1.2 程序：显示平方值的表格……63
6.1.3 程序：数列求和……63
6.2 do语句……64
6.3 for语句……65
6.3.1 for语句的惯用法……66
6.3.2 在for语句中省略表达式……67
6.3.3 逗号运算符……67
6.3.4 程序：显示平方值的表格（改进版）……68
6.4 退出循环……69
6.4.1 break语句……69
6.4.2 continue语句……70
6.4.3 goto语句……71
6.4.4 程序：账目簿结算……71
6.5 空语句……73
问与答……74
练习……76

第7章 基本类型……78

7.1 整型……78
7.1.1 整型常量……79
7.1.2 读/写整数……80
7.1.3 程序：数列求和（改进版）……81
7.2 浮点型……81
7.2.1 浮点常量……82
7.2.2 读/写浮点数……83
7.3 字符型……83
7.3.1 转义序列……84
7.3.2 字符处理函数……85
7.3.3 读/写字符……86
7.3.4 程序：确定消息的长度……87
7.4 sizeof运算符……88
7.5 类型转换……89
7.5.1 常用算术转换……89
7.5.2 赋值中的转换……90
7.5.3 强制类型转换……91
7.6 类型定义……92
问与答……93
练习……95

第8章 数组……98

8.1 一维数组……98
8.1.1 数组下标……98
8.1.2 程序：数列反向……100
8.1.3 数组初始化……100
8.1.4 程序：检查数中重复出现的数字……101
8.1.5 对数组使用sizeof运算符……101
8.1.6 程序：计算利息……102
8.2 多维数组……103
8.2.1 多维数组初始化……104
8.2.2 常量数组……105
8.2.3 程序：发牌……105
问与答……106
练习……107

第9章 函数……110

9.1 函数的定义和调用……110
9.1.1 程序：计算平均值……110
9.1.2 程序：显示倒数计数……111
9.1.3 程序：显示双关语（改进版）……112
9.1.4 函数定义……113
9.1.5 函数调用……114
9.1.6 程序：判定素数……115
9.2 函数声明……116
9.3 实际参数……117
9.3.1 实际参数的转换……118
9.3.2 数组型实际参数……119
9.4 return语句……120
9.5 程序终止……121
9.6 递归函数……122
9.6.1 快速排序算法……123
9.6.2 程序：快速排序……124
问与答……125
练习……128

第10章 程序结构……131

10.1 局部变量……131
10.2 外部变量……132
10.2.1 程序：用外部变量实现栈……132
10.2.2 外部变量的利与弊……133
10.2.3 程序：猜数……134
10.3 程序块……137
10.4 作用域……137
10.5 构建C程序……138

www.TopSage.com

问与答 …… 144
练习 …… 144

第 11 章 指针 …… 146

11.1 指针变量 …… 146
11.2 取地址运算符和间接寻址运算符 …… 147
11.2.1 取地址运算符 …… 147
11.2.2 间接寻址运算符 …… 148
11.3 指针赋值 …… 148
11.4 指针作为实际参数 …… 149
11.4.1 程序：找出数组中的最大元素和最小元素 …… 151
11.4.2 用const保护实际参数 …… 152
11.5 指针作为返回值 …… 153
问与答 …… 153
练习 …… 155

第 12 章 指针和数组 …… 156

12.1 指针的算术运算 …… 156
12.1.1 指针加上整数 …… 157
12.1.2 指针减去整数 …… 157
12.1.3 指针相减 …… 158
12.1.4 指针比较 …… 158
12.2 指针用于数组处理 …… 158
12.3 用数组名作为指针 …… 160
12.3.1 程序：数列反向（改进版） …… 161
12.3.2 数组型实际参数（改进版） …… 161
12.3.3 用指针作为数组名 …… 162
12.4 指针和多维数组 …… 163
12.4.1 处理多维数组的元素 …… 163
12.4.2 处理多维数组的行 …… 163
12.4.3 用多维数组名作为指针 …… 164
问与答 …… 164
练习 …… 166

第 13 章 字符串 …… 168

13.1 字符串字面量 …… 168
13.1.1 字符串字面量中的转义序列 …… 168
13.1.2 延续字符串字面量 …… 169
13.1.3 如何存储字符串字面量 …… 169
13.1.4 字符串字面量的操作 …… 170
13.1.5 字符串字面量与字符常量 …… 170
13.2 字符串变量 …… 170
13.2.1 初始化字符串变量 …… 171
13.2.2 字符数组与字符指针 …… 172
13.3 字符串的读/写 …… 173
13.3.1 用printf函数和puts函数写字符串 …… 173
13.3.2 用scanf函数 和gets函数读字符串 …… 173
13.3.3 逐个字符读字符串 …… 174
13.4 访问字符串中的字符 …… 175
13.5 使用C语言的字符串库 …… 176
13.5.1 strcpy函数 …… 177
13.5.2 strcat函数 …… 177
13.5.3 strcmp函数 …… 178
13.5.4 strlen函数 …… 178
13.5.5 程序：显示一个月的提示列表 …… 179
13.6 字符串惯用法 …… 181
13.6.1 搜索字符串的结尾 …… 181
13.6.2 复制字符串 …… 182
13.7 字符串数组 …… 184
13.7.1 命令行参数 …… 185
13.7.2 程序：核对行星的名字 …… 186
问与答 …… 187
练习 …… 189

第 14 章 预处理器 …… 192

14.1 预处理器的工作方式 …… 192
14.2 预处理指令 …… 194
14.3 宏定义 …… 194
14.3.1 简单的宏 …… 194
14.3.2 带参数的宏 …… 196
14.3.3 #运算符 …… 197
14.3.4 ##运算符 …… 198
14.3.5 宏的通用属性 …… 199
14.3.6 宏定义中的圆括号 …… 199
14.3.7 创建较长的宏 …… 200
14.3.8 预定义宏 …… 201
14.4 条件编译 …… 202
14.4.1 #if指令和#endif指令 …… 202
14.4.2 defined运算符 …… 203
14.4.3 #ifdef指令和#ifndef指令 …… 203
14.4.4 #elif指令和#else指令 …… 203
14.4.5 使用条件编译 …… 204
14.5 其他指令 …… 205
14.5.1 #error指令 …… 205
14.5.2 #line指令 …… 205
14.5.3 #pragma指令 …… 206
问与答 …… 206
练习 …… 208

www.TopSage.com

大家网 TopSage.com

第 15 章 编写大规模程序……211
15.1 源文件……211
15.2 头文件……212
15.2.1 #include指令……212
15.2.2 共享宏定义和类型定义……213
15.2.3 共享函数原型……214
15.2.4 共享变量声明……215
15.2.5 嵌套包含……216
15.2.6 保护头文件……216
15.2.7 头文件中的#error指令……217
15.3 把程序划分成多个文件……217
15.4 构建多文件程序……223
15.4.1 makefile……223
15.4.2 链接期间的错误……224
15.4.3 重新构建程序……225
15.4.4 在程序外定义宏……226
问与答……227
练习……228

第 16 章 结构、联合和枚举……229
16.1 结构变量……229
16.1.1 结构变量的声明……229
16.1.2 结构变量的初始化……230
16.1.3 对结构的操作……231
16.2 结构类型……232
16.2.1 结构标记的声明……232
16.2.2 结构类型的定义……233
16.2.3 结构类型的实际参数和返回值……233
16.3 数组和结构的嵌套……234
16.3.1 嵌套的结构……234
16.3.2 结构数组……235
16.3.3 结构数组的初始化……235
16.3.4 程序：维护零件数据库……236
16.4 联合……241
16.4.1 使用联合来节省空间……242
16.4.2 使用联合来构造混合的数据结构……243
16.4.3 为联合添加“标记字段”……243
16.5 枚举……244
16.5.1 枚举标记和枚举类型……245
16.5.2 枚举作为整数……245
16.5.3 用枚举声明“标记字段”……246
问与答……246
练习……247

第 17 章 指针的高级应用……250
17.1 动态存储分配……250
17.1.1 内存分配函数……250
17.1.2 空指针……251
17.2 动态分配字符串……251
17.2.1 使用malloc函数为字符串分配内存……252
17.2.2 在字符串函数中使用动态存储分配……252
17.2.3 动态分配字符串的数组……253
17.2.4 程序：显示一个月的提示列表（改进版）……253
17.3 动态分配数组……254
17.3.1 使用malloc函数为数组分配存储空间……255
17.3.2 calloc函数……255
17.3.3 realloc函数……256
17.4 释放存储……256
17.4.1 free函数……257
17.4.2 “悬空指针”问题……257
17.5 链表……257
17.5.1 声明结点类型……258
17.5.2 创建结点……258
17.5.3 ->运算符……259
17.5.4 在链表的开始处插入结点……259
17.5.5 搜索链表……261
17.5.6 从链表中删除结点……262
17.5.7 链表排序……264
17.5.8 程序：维护零件数据库（改进版）……264
17.6 指向指针的指针……268
17.7 指向函数的指针……269
17.7.1 函数指针作为实际参数……269
17.7.2 qsort函数……270
17.7.3 函数指针的其他用途……271
17.7.4 程序：列三角函数表……272
问与答……273
练习……276

第 18 章 声明……278
18.1 声明的语法……278
18.2 存储类型……279
18.2.1 变量的特性……279
18.2.2 auto存储类型……280
18.2.3 static存储类型……280

www.TopSage.com

18.2.4 extern存储类型 …… 281
18.2.5 register存储类型 …… 282
18.2.6 函数的存储类型 …… 282
18.2.7 小结 …… 283
18.3 类型限定符 …… 284
18.4 声明符 …… 284
18.4.1 解释复杂声明 …… 285
18.4.2 使用类型定义来简化声明 …… 286
18.5 初始化式 …… 287
问与答 …… 288
练习 …… 289

第 19 章 程序设计 …… 291

19.1 模块 …… 291
19.1.1 内聚性与耦合性 …… 293
19.1.2 模块的类型 …… 293
19.2 信息隐藏 …… 293
19.3 抽象数据类型 …… 296
19.4 C++语言 …… 297
19.4.1 C语言与C++语言之间的差异 …… 297
19.4.2 类 …… 299
19.4.3 类定义 …… 300
19.4.4 成员函数 …… 300
19.4.5 构造函数 …… 302
19.4.6 构造函数和动态存储分配 …… 303
19.4.7 析构函数 …… 304
19.4.8 重载 …… 304
19.4.9 面向对象编程 …… 306
19.4.10 派生 …… 306
19.4.11 虚函数 …… 308
19.4.12 模板 …… 310
19.4.13 异常处理 …… 310
问与答 …… 311
练习 …… 312

第 20 章 低级程序设计 …… 314

20.1 按位运算符 …… 314
20.1.1 移位运算符 …… 314
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
20.1.3 用按位运算符访问位 …… 316
20.1.4 用按位运算符访问位域 …… 317
20.1.5 程序：XOR加密 …… 317
20.2 结构中的位域 …… 318
20.3 其他低级技术 …… 320
20.3.1 定义依赖机器的类型 …… 320
20.3.2 用联合从多个视角看待数据 …… 320
20.3.3 将指针作为地址使用 …… 321
20.3.4 程序：设置Num Lock键 …… 322
20.3.5 volatile类型限定符 …… 323
问与答 …… 323
练习 …… 323

第 21 章 标准库 …… 325

21.1 标准库的使用 …… 325
21.1.1 对标准库中使用的名字一些限制 …… 325
21.1.2 使用宏隐藏函数 …… 325
21.2 标准库概述 …… 326
21.3 <stddef.h>：常用定义 …… 327
问与答 …… 328
练习 …… 328

第 22 章 输入/输出 …… 329

22.1 流 …… 329
22.1.1 文件指针 …… 330
22.1.2 标准流和重定向 …… 330
22.1.3 文本文件与二进制文件 …… 330
22.2 文件操作 …… 331
22.2.1 打开文件 …… 332
22.2.2 模式 …… 332
22.2.3 关闭文件 …… 333
22.2.4 为流附加文件 …… 333
22.2.5 从命令行获取文件名 …… 334
22.2.6 程序：检查文件是否可以打开 …… 334
22.2.7 临时文件 …… 335
22.2.8 文件缓冲 …… 336
22.2.9 其他文件操作 …… 337
22.3 格式化的输入/输出 …… 337
22.3.1 ...printf类函数 …… 337
22.3.2 ...printf类函数的转换说明 …… 338
22.3.3 ...printf类函数的转换说明示例 …… 339
22.3.4 ...scanf类函数 …… 341
22.3.5 ...scanf类函数的格式化字符串 …… 342
22.3.6 ...scanf类函数的转换说明 …… 342
22.3.7 scanf函数的示例 …… 343
22.3.8 检测文件末尾和错误条件 …… 344
22.4 字符的输入/输出 …… 346
22.4.1 输出函数 …… 346

www.TopSage.com

22.4.2 输入函数……346
22.4.3 程序：复制文件……347
22.5 行的输入/输出……348
22.5.1 输出函数……348
22.5.2 输入函数……348
22.6 块的输入/输出……349
22.7 文件的定位……350
22.8 字符串的输入/输出……352
问与答……353
练习……356

第 23 章 库对数值和字符数据的支持……360

23.1 <float.h>：浮点型的特性……360
23.2 <limits.h>：整数类型的大小……361
23.3 <math.h>：数学计算……362
23.3.1 错误……362
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
练习……381

第 25 章 国际化特性……383

25.1 <locale.h>：本地化……383
25.1.1 类别……383
25.1.2 setlocale函数……384
25.1.3 localeconv函数……384
25.2 多字节字符和宽字符……386
25.2.1 多字节字符……387
25.2.2 宽字符……387
25.2.3 多字节字符函数……388
25.2.4 多字节字符串函数……389
25.3 三字符序列……389
问与答……390
练习……391

第 26 章 其他库函数……392

26.1 <stdarg.h>：可变长度实参……392
26.1.1 调用带有可变实参列表的函数……393
26.1.2 V…printf类函数……393
26.2 <stdlib.h>：通用的实用工具……394
26.2.1 字符串转换函数……394
26.2.2 程序：测试字符串转换函数……395
26.2.3 伪随机序列生成函数……397
26.2.4 程序：测试伪随机序列生成函数……397
26.2.5 与环境的通信……398
26.2.6 搜索和排序实用工具……399
26.2.7 程序：确定航空里程……399
26.2.8 整数算术运算函数……400
26.3 <time.h>：日期和时间……401
26.3.1 时间处理函数……401
26.3.2 时间转换函数……403
26.3.3 程序：显示日期和时间……405
问与答……406
练习……407

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）

www.TopSage.com

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 | 软考历年真是解析与答案

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…

大家网
TopSage.com

第1章 C语言概述

如果有人说“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）。

www.TopSage.com

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++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

www.TopSage.com

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

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

www.TopSage.com

大家网 TopSage.com

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?(
l^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 Osovlanski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。

---

www.TopSage.com

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以理解的。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。

最主要的目的是解决学生学习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

www.TopSage.com

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

8 漏”，即那些动态分配却从未被释放的内存块。

www.TopSage.com

大家网
TopSage.com

第2章

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，反倒是更愿意支持另一种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.
```


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。

www.TopSage.com

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
10 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 Sofware 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&A C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：

$$f(x)=x+1$$

$$g(y,z)=y^2-z^2$$

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;
```

www.TopSage.com

大家网 TopSage.com

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。

www.TopSage.com

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：

14

```
/* 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：

```
/*************************************************************
 *Namc:  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.
```

www.TopSage.com

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

15

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 ");      /* forgot to close this comment...
printf("car ");
printf("has ");     /* so it ends here */
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，Q&A 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。

www.TopSage.com

TopSage.com

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。

www.TopSage.com

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12×10×8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）。

www.TopSage.com

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**（fomat 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 */
```

20

%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>
```

www.TopSage.com

大家网 TopSage.com

```
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，因而在此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。

www.TopSage.com

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

www.TopSage.com

TopSage.com

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

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

或

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

26

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。

www.TopSage.com

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，而且程序包含了许多短小的注释。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1.5×23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。

28

***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C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？

www.TopSage.com

大家网 TopSage.com

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$v=4/3\pi r^3$。注意，分数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。

www.TopSage.com

第3章 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 ***/
```

www.TopSage.com

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
32 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%.1f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*.*p*X格式或%-*m*.*p*X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
33 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

www.TopSage.com

TopSage.com

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）

www.TopSage.com

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prinf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

www.TopSage.com

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：

37

```
••1¤-20•••.3¤•••-4.0e3¤
ssrsrrrsssrrssssrrrrrr
```

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

www.TopSage.com

大家网 TopSage.com

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-4.0×10^3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

www.TopSage.com

```
•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

股票价格通常是用美元数量表示的，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);
```

www.TopSage.com

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

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

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

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。

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。）

www.TopSage.com

TopSage.com

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);
```


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yymmdd）。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！

www.TopSage.com

```
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。

43　编写一个程序来分解用户录入的ISBN信息，格式如下：

```
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
```

44　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。）

www.TopSage.com

第4章

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。

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 */
```

www.TopSage.com

大家网 TopSage.com

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：

www.TopSage.com

```
i + j * k     等价于   i + ( j * k )
-i * -j       等价于   (-i) * (-j)
+i + j / k    等价于   (+i) + ( j / k )
```

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 47

```
i - j - k  等价于  (i - j ) - k
i * j / k  等价于  (i * j ) / k
```

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     等价于     -(+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（➤附录B），或者只是使用足够多的圆括号。

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]： 48

```
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 要计算的检验位分别为3（Jif）和6（Ocean Spray）。

www.TopSage.com

```
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));

  return 0;
}
```


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）。

www.TopSage.com

TopSage.com

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。

www.TopSage.com

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变化后的新值。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

大家网 TopSage.com

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。

www.TopSage.com

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。

58

***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

www.TopSage.com

TopSage.com

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。

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);
   ```

www.TopSage.com

(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。

(a)
```
i = 1;
printf("%d ", i++ - 1);
printf("%d", i);
```
60

(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++中只有一个是与表达式（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。）

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

www.TopSage.com

第5章

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。

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。

www.TopSage.com

大家网 TopSage.com

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

www.TopSage.com

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。

在所有其他情况中，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

www.TopSage.com

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（判等）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≤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。

TopSage.com

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;
```

68

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");
```

69

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：

www.TopSage.com

```
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语句作为了它自己的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：

```
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)
```

www.TopSage.com

```
    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：

71

```
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，

www.TopSage.com

大家网 TopSage.com

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）。

www.TopSage.com

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

73

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");
```

74

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");
```

www.TopSage.com

```
            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：

www.TopSage.com

TopSage.com

```
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。 76

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swithc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___. 77

www.TopSage.com

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

www.TopSage.com

```
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EricAllman（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：

79

```
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。

www.TopSage.com

大家网 TopSage.com

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。

www.TopSage.com

(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）

www.TopSage.com

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");
```

83 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");
}
```

84

www.TopSage.com

大家网
TopSage.com

第6章 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：

[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。

www.TopSage.com

- 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成单独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：

www.TopSage.com

[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。 87

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～*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，这样做的好处是使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： 88

```
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 */
```

www.TopSage.com

大家网 TopSage.com

```
#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
89 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。

www.TopSage.com

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);
```

www.TopSage.com

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
91 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。

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

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
92 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--) ...`

www.TopSage.com

TopSage.com

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代替了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。

www.TopSage.com

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

95 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

www.TopSage.com

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++)
```

www.TopSage.com

大家网 TopSage.com

```
    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;
    ...
  }
}
```


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sun += 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);
```

www.TopSage.com

```
    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
```


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: ");
```

www.TopSage.com

TopSage.com

```
    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;  ;  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

www.TopSage.com

会提高效率。但是在一些情况下，带空循环体的循环是比另一种更清楚。例如，这些带空循环体的循环更便于读字符（➤7.3.3节）数据。

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。

102

- 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)

www.TopSage.com

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？ 104

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：

```
{
  ...
```

www.TopSage.com

大家网 TopSage.com

```
    goto end_of_stmt;
    ...
    end_of_stmt: ;
  }
```

105

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。

106

```
Enter value of trade: 30000
Commission: $166.00
Enter value of trade: 20000
Commission: $144.00
Enter value of trade: 0
```

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：

www.TopSage.com

```
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);`

107

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");
```

108

www.TopSage.com

第7章 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01111111111111111111111111111111，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
```

www.TopSage.com

TopSage.com

```
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。(➤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

www.TopSage.com

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$2 \times 8^2+3 \times 8^1+7 \times 8^0$
111 $=128+24+7=159$。

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  0x7fffL
```

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  0x7fffU
```

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

www.TopSage.com

大家网 TopSage.com

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，
114 所以这里也用它做一个示例。

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
115 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。

www.TopSage.com

在某些极个别的情况下，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0110000~0111001码来表示，同时大写字母A~Z则用1000001~1011010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。 116

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。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：

www.TopSage.com

```
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
117 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小值整数的时候。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。

www.TopSage.com

TopSage.com

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。 118

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。 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

www.TopSage.com

赋值运算符把touuper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，touuper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。purchar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：

www.TopSage.com

```
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。我们既可以采用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;
```

www.TopSage.com

大家网 TopSage.com

```
  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，
123 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。计算机可能可以直接将两个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）。 124

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：

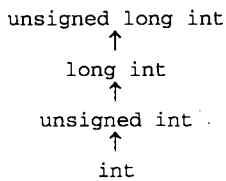
```
long double
     ↑
  double
     ↑
  float
```

125

www.TopSage.com

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：

```
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         */
```

126

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

www.TopSage.com

TopSage.com

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的类型转换：

```
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，

www.TopSage.com

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;
```

www.TopSage.com

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;
```

130

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tydef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”会发生什么？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

www.TopSage.com

大家网 TopSage.com

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。

131

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）。

132

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ʧa:(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）

www.TopSage.com

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??，那么编译器很可能会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类型一样好用吗？（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

www.TopSage.com

(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...
   ```

135　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-COL-LECT
   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
    ```

www.TopSage.com

TopSage.com

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touper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？ 136

```
d = i + f;
```

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

www.TopSage.com

第8章 数 组

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

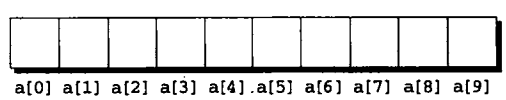

```
int a[10];
```

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];
```

www.TopSage.com

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&。

C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。

140

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++]的内存位置。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++]的内存位置，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];
```

141

www.TopSage.com

大家网 TopSage.com

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：

142

```
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;                                                    143


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;
}
```

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）。

www.TopSage.com

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: ");
```

145

www.TopSage.com

TopSage.com

```
    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;

}
```

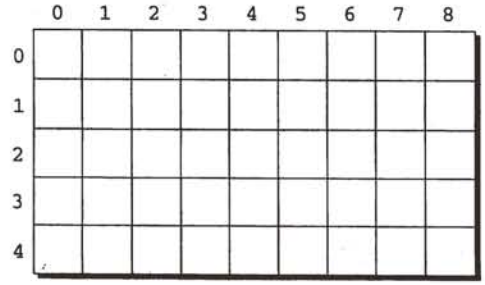
注意，这里使用NUM RATES控制两个for循环。如果后面改变数组value的大小，循环将会自动调整。

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：



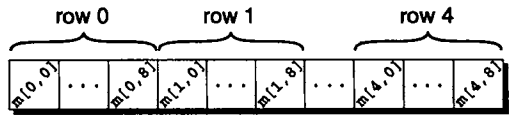
146

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：

www.TopSage.com



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}};
```

- 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},
                {0, 1, 0, 1, 1, 0, 0, 1},
                {1, 1, 0, 1, 0, 0, 0, 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。

www.TopSage.com

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”：

```
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。

为了避免两次都拿到同一张牌，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

www.TopSage.com

大家网 TopSage.com

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值0。

150

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？

www.TopSage.com

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

1. 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
```

2. 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
```

3. 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。
4. 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？
5. 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。
6. 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。
7. 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：

www.TopSage.com

```
H3Y DUD3, C 15 R1LLY C0OL!!!!!!!!!!
```

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 C0OL!!!!!!!!!!
```

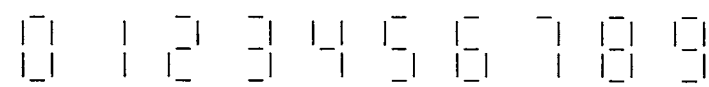
程序需要把消息转换成大写字母，用数字代替特定的字母（A→4, B→8, E→3, I→1, O→0,S→5），然后添加10个感叹号。提示：在字符数组中存储原始消息，然后从数组头开始逐个翻译并且显示字符。

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：

152



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srand函数和rand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

www.TopSage.com

TopSage.com

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

www.TopSage.com

第9章

函　数

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

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

www.TopSage.com

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);
```

156

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

www.TopSage.com

大家网 TopSage.com

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：

```
void print_count (int n)
{
  printf("T minus %d and counting\n",n);
}
```

157

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：

countdwn.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");
}
```

158

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：

[函数定义]

```
返回类型 函数名 (形式参数)
{
    声明
    语句
}
```

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。

159

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;
}
```

如果返回类型很冗长，比如unsigned long int类型，那么把返回类型单独放在一行是非常

www.TopSage.com

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：即使是有些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 if %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。

TopSage.com

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： 161

```
printf ("Hi, Mom!\n");     /* discards return value */
```

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

www.TopSage.com

```
  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>
```

www.TopSage.com

```
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：

165

```
int power(int x, int n)
{
  int result = 1;

  while (n--  > 0)
    result = result * x;
```

www.TopSage.com

大家网 TopSage.com

```
  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 if %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));
```

www.TopSage.com

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[])    /* no length specified */
{
  ...
}
```

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);
  ...
}
```

www.TopSage.com

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：

TopSage.com

```
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_SUCESS);    /* 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

www.TopSage.com

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2×1=2，从而导致
fact(3)返回3×2=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5×1=5，从而导致
  power(5, 2)返回5×5=25，从而导致
power(5, 3)返回5×25=125。
```

172

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。

www.TopSage.com

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。

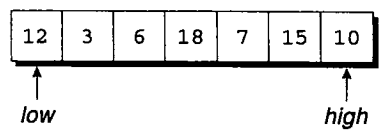
执行完第1步后，元素*e*处在正确的位置上。因为*e*左侧的元素全部都是小于或等于*e*的，所以一旦第2步对这些元素进行排序，那么这些小于或等于*e*的元素也将会处在正确的位置上；类似的理由也可以应用于*e*右侧的元素。

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。

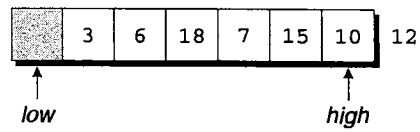
该算法依赖于两个名为*low*和*high*的标记，这两个标记用来跟踪数组内的位置。开始，*low*指向数组中的第一个元素，而*high*指向末尾元素。首先把第一个元素（分割元素）复制给其他地方的一个临时存储单元，从而在数组中留出一个“空位”。接下来，从右向左移动*high*，直到*high*指向小于分割元素的数时停止。然后把这个数复制给*low*指向的空位，这将产生一个新的空位（*high*指向的）。现在从左向右移动*low*，寻找大于分割元素的数。在找到时，把这个找到的数复制给*high*指向的空位。重复执行此过程，交替操作*low*和*high*直到两者在数组中间的某处相遇时停止。此时，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

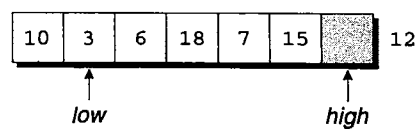
首先，假设数组包含7个元素。*low*指向第一个元素；*high*指向最后一个元素。



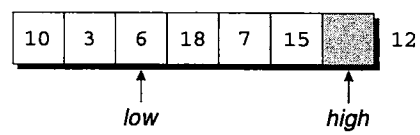
第一个元素12是分割元素。把它复制到某个位置，留出数组开始处的空位。



现在把12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。

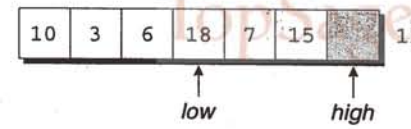


*low*指向的数3是小于12的，因此不需要进行移动。只是把*low*向右移动一位。

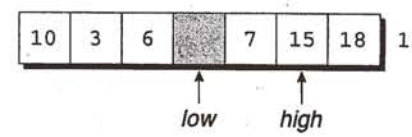


www.TopSage.com

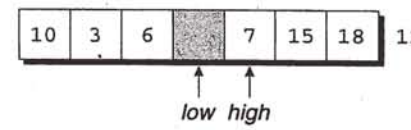
因为6也是小于12的，所以再把*low*向右移动一位。



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。

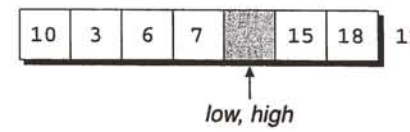


*high*指向的数15是大于12的，因此不需要进行移动。只是把*high*向左移动一位然后继续。

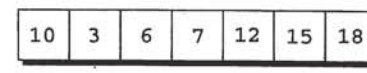


*high*指向的数7超出范围。在把7移动到空位后，把*low*向右移动一位。

174



*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。

qsort.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: ");
```

www.TopSage.com

```
  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;
}
```

175

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）。

176

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：

www.TopSage.com

	本 书	其 他 书
形式参数（形参）	parameter	formal argument, formal parameter
实际参数（实参）	argument	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，不就没有问题了吗？

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

www.TopSage.com

TopSage.com

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，是把指向数组第一个元素的指针给了函数。 179

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。

www.TopSage.com

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

1. 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);
}
```

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。
3. 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。）
4. 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）。
5. 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，

www.TopSage.com

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)
```

www.TopSage.com

大家网 TopSage.com

```
        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('0' + 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。

183 (b) 递归地调用函数本身来对前*n*-1个数组元素进行排序。

www.TopSage.com

第10章

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。

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。

www.TopSage.com

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来。
- **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

TopSage.com

```
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") ;
```

www.TopSage.com

```
    }
  }
```

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!

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
```

189

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");
```

www.TopSage.com

```
    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");
  }
}
```

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移入到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。

www.TopSage.com

大家网 TopSage.com

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 */

#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;
```

www.TopSage.com

```
    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严格局部于程序块。 193

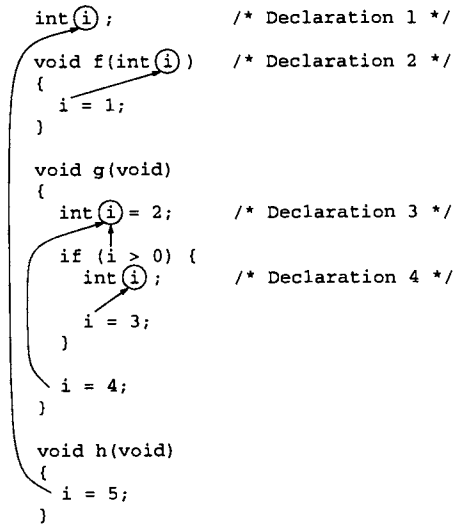
10.4 作用域

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：

www.TopSage.com



- 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

www.TopSage.com

TopSage.com

- 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。 195

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
```

196

www.TopSage.com

```
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_resul t();
   }
}

 /*************************************************************
  * read cards:  Reads the cards into external variables;     *
  *              checks for bad cards and duplicate cards.    *
  *************************************************************/
void read cards (void)
{
  ...
}
```

197

www.TopSage.com

```
/*************************************************************
 *  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);
```

www.TopSage.com

大家网 TopSage.com

```
/*************************************************************
 * 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.     *
 *************************************************************/
```

199

```
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;
```

www.TopSage.com

```
      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++;
  }
```

200

www.TopSage.com

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

1. 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 parenteses and/or braces:{(){}{()}}
Parenteses/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

2. 下面的程序框架只显示了函数定义和变量声明。

www.TopSage.com

TopSage.com

```
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5×2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）。

www.TopSage.com

第11章 指　针

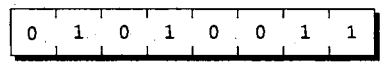
我忘记了第十一条戒律是“你应该计算”，还是“你不应该计算”。

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。

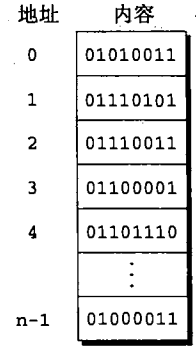
本章将从机器地址以及与其相关的指针变量的讨论开始（11.1节）；然后，11.2节介绍取地址运算符和间接寻址运算符；11.3节涵盖了指针赋值的内容；11.4节说明了给函数传递指针的方法，而11.5节则讨论从函数返回指针。

11.1 指针变量

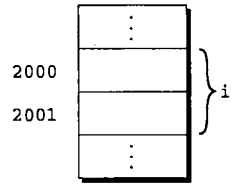
理解指针的第一步是在机器层上观察指针表示的内容。大多数现代计算机用**字节**（byte）来分割内存，每个字节可以存储8位的信息。



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。



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：

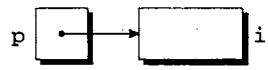


这就是指针的出处。虽然用数表示地址，但是其取值范围可能不同于整数的范围，所以一定不能用普通整型变量存储地址。但是，可以用特殊的**指针变量**（pointer variable）存储地址。

www.TopSage.com

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：

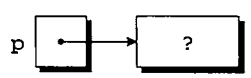
```
int *p;  /* points nowhere in particular */
```

(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;
```

大家网 TopSage.com

```
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);
```

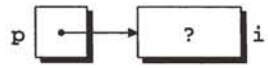
printf函数将会显示i的值，而不是i的地址。

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：

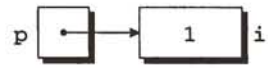
```
j = *&i;    /* same as j = i; */
```

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;
```

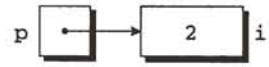



```
i = 1;
```




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

208




```
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、

www.TopSage.com

p和q有如下声明：

```
int i, j, *p, *q;
```

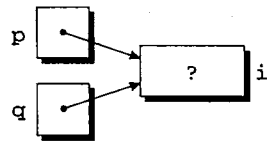
语句

```
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;
```

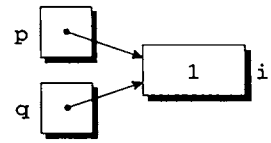
这条语句是把p的内容（即i的地址）复制给q，效果是把q指向了和p指向相同的地方：



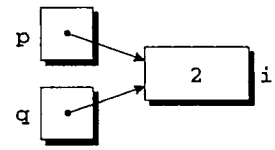
209

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

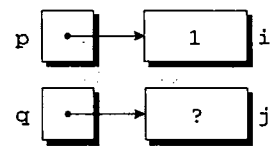
```
q = p;
```

和

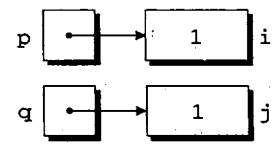
```
*q = *p;
```

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;
  *fract_part = x - *int_part;
}
```

decompose函数的原型既可以是

```
void decompose(float x, int *int_part, float *frac_part);
```

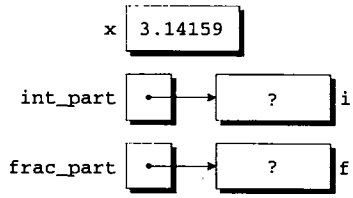
也可以是

```
void decompose(float, int *, float *);
```

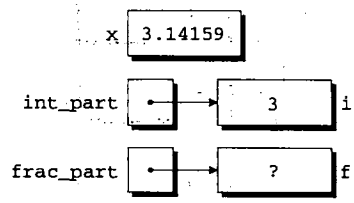
以下列方式调用decompose函数：

```
decompose(3.14159, &i, &f) ;
```

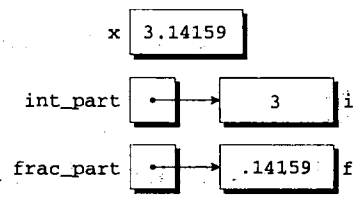
因为i和f前有地址运算符&，所以decompose函数的实际参数是指向i和f的指针，而不是i和f的值。调用decompose函数时，把值3.14159复制到x中，把指向i的指针存储在int_part
211 中，而把指向f的指针存储在frac_part中：



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：



www.TopSage.com

TopSage.com

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：

212

```
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。

213

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：

www.TopSage.com

```
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

214

```
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。

www.TopSage.com

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。 215

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)**

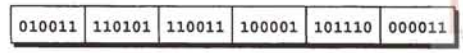
答：通常是，但不总是。考虑用字而不是字节划分内存的计算机。字可以包含36位、60位，或者更多位。如果假设36位的字，那么内存将有如下的显示：

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：

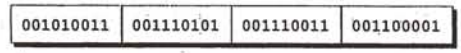
www.TopSage.com

大家网 TopSage.com



或者4个9位的字符：

216



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)**

217

```
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;
```

www.TopSage.com

```
    *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

www.TopSage.com

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

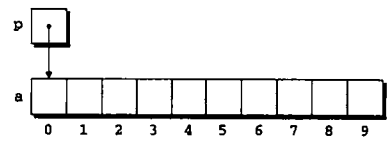
指针不仅可以指向普通变量，还可以指向数组元素。例如，假设已经声明a和p如下：

```
int a[10], *p;
```

通过下列写法可以使p指向a[0]：

```
p = &a[0];
```

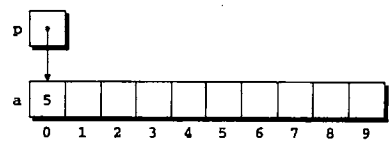
用图形方式表示，下面是我们刚刚做的：



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：

www.TopSage.com

TopSage.com

- 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：

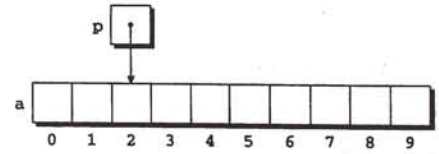
```
int a[10], *p, *q, i;
```

12.1.1 指针加上整数

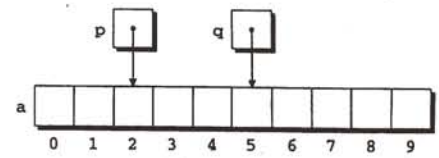
指针p加上整数j产生指向特定元素的指针，这个特定元素是p原先指向的元素后的j个位置。Q&A更确切些说，如果p指向数组元素a[i]，那么p+j指向a[i+j]（当然，前提是a[i+j]必须存在）。

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

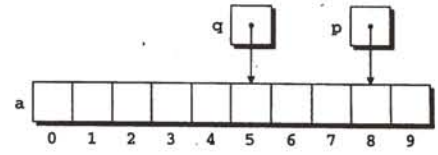
```
P = &a[2];
```




```
q = p + 3;
```



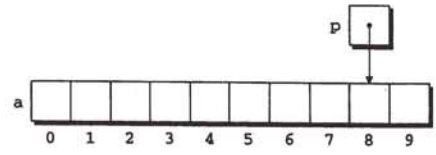

```
p += 6;
```



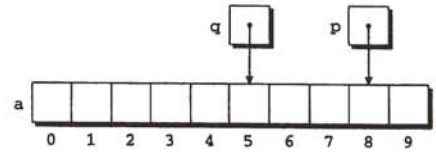
12.1.2 指针减去整数

如果p指向数组元素a[i]，那么p-j指向a[i-j]。例如：

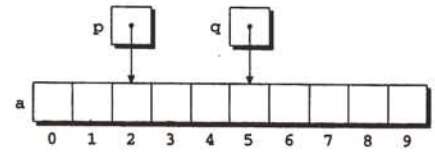
```
p = &a[8];
```




```
q = p-3;
```




```
p -= 6;
```

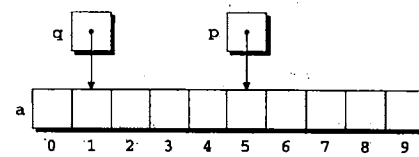


www.TopSage.com

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
223 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];
```

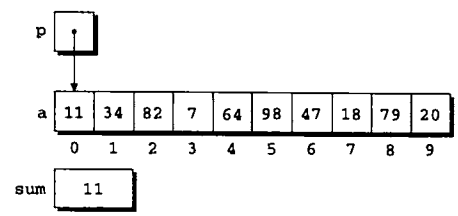
12.2 指针用于数组处理

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。

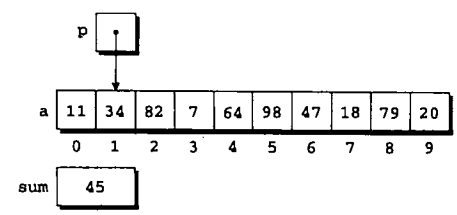
```
#define N 10
int a[N], sum, *p;
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

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：

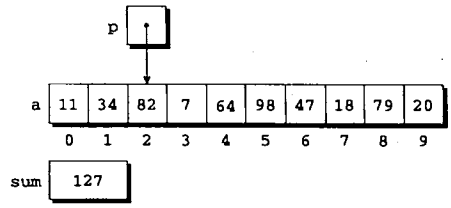


第2次重复的末尾：



www.TopSage.com

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本身是不变化的）。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
p++或(p++)	自增前表达式的值是*p，然后自增p
(*p)++	自增前表达式的值是*p，然后自增*p
++p或(++p)	先自增p，自增后表达式的值是*p
++*p或++(*p)	先自增*p，自增后表达式的值是*p

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。

www.TopSage.com

大家网 TopSage.com

```
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++;
```

www.TopSage.com

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。

229

- 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];
```

230

www.TopSage.com

TopSage.com

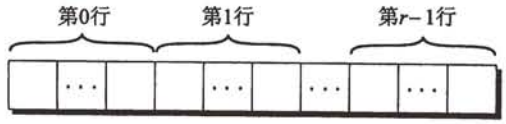
编译器对待p[i]就像对待*(p+i)一样，这是指针算术运算非常正规的用法。虽然我们对下标一个指针的能力有更多一些的好奇，但是要在17.3节才会看到它的实际用法。

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_ROW-][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

www.TopSage.com

```
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是很有意义的，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_RPOWS][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 × j加上p，也可以用4 × 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 × j加上p，因为double型的值通常都是8个字节长。

233　**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)**

www.TopSage.com

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) 234
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。

***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。

www.TopSage.com

大家网 TopSage.com

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？

235 (e) *p<*q的结果是真还是假？

*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]

www.TopSage.com

(c) `*p == a[0]`
(d) `p[0] == a[0]`

236

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。

237

www.TopSage.com

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语言书中称为之字串。——译者注

www.TopSage.com

TopSage.com

"\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 240
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");
```

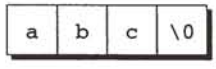
13.1.3 如何存储字符串字面量

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）。

www.TopSage.com

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
241 串字面量可以出现在赋值运算符的右边：

```
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：

www.TopSage.com

```
#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。

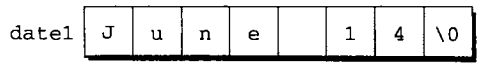
13.2.1 初始化字符串变量

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



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：

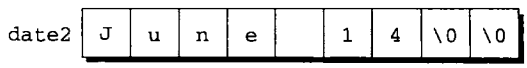
```
char date1[8] = {'J', 'u', 'n', 'e', ' ', '1', '4', '\0'};
```

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：

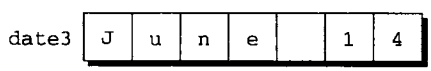


大体上来说，这种行为与C语言处理数组初始化式（➤8.1.3节）的方法一致。当数组的初始化式比数组本身短时，会把余下的数组元素初始化为0。在把字符数组额外的元素初始化为\0这点上，编译器对字符串和数组遵循相同的规则。

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：



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。

www.TopSage.com

大家网 TopSage.com

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。

245 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。

www.TopSage.com

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%.*p*s。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%*m*s会在大小为*m*的域内显示字符串。 246
（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*.*p*s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。

www.TopSage.com

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%*n*s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：

248

```
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且把它们存储起来。在读入换行符时循环终业。（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：

TopSage.com

```
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;
}
```

249

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。

www.TopSage.com

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(srt2, strcpy(srtl, "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：

www.TopSage.com

大家网 TopSage.com

```
strcpy(strl, "abc") ;
strcpy(str2, "def") ;
strcat(strl, strcat(str2, "ghi"));
  /* strl 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

253

```
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（➤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（➤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。

www.TopSage.com

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;

fen = 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 remlnder:  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ptintf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：

www.TopSage.com

```
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;
}
```


www.TopSage.com

大学网 TopSage.com

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;
```

www.TopSage.com

```
    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
258 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

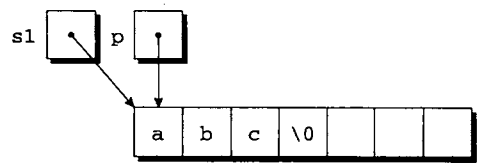
复制字符串是另一种常见操作。为了介绍C语言“字符串复制”这种惯用法，这里将开发
259 strcat函数的一种写法。就先从直接但有些冗长的strcat函数写法开始：

```
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 ;
}
```

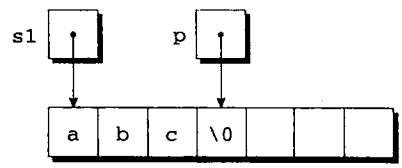
www.TopSage.com

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。

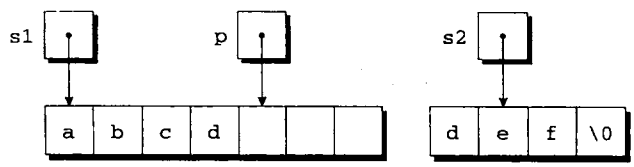
函数中的第一个while语句实现了第1步。把p设定为指向s1的第一个字符。参考下图，假设s1指向字符串"abc"：



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：

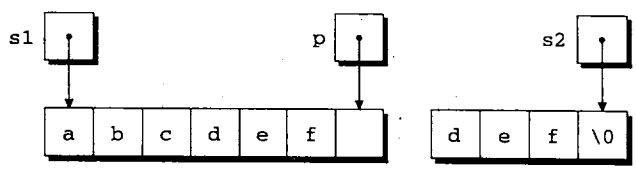


第二个while语句实现了第（2）步。循环体把s2指向的一个字符复制到p指向的地方，接着p和s2都进行自增。如果s2最初指向字符串"def"，下面显示了第一次循环后的样子：



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：

www.TopSage.com

大家网 TopSage.com

[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。

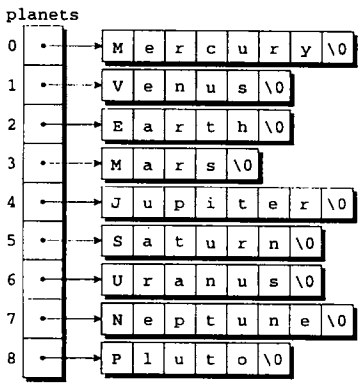
因为大部分字符串集都是长短字符串的混合，所以这些例子所暴露的低效性是在处理字符串时经常遇到的问题。我们需要的是**参差不齐的数组**（ragged array），即数组的每一行有不同的长度。C语言本身不提供这种“参差不齐的数组类型”，但它确实提供了模拟这种数组类型的工具。秘诀就是建立一个特殊的数组，这个数组的元素都是指向字符串的指针。

www.TopSage.com

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", "Jupi ter", "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： 262



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： 263

```
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]

www.TopSage.com

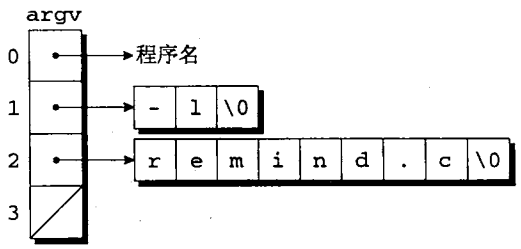
到argv[argc-1]则指向余下的命令行参数。

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）：

www.TopSage.com

TopSage.com

```
jupiter is planet 5
venus is not a planet
Earch 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。

266

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。

www.TopSage.com

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 Programing 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++ = *s2++)
  ;
```

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

www.TopSage.com

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。 268

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[]。这是否合法？**

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。 269

   (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。

www.TopSage.com

大家网 TopSage.com

(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;
}
```

270 (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;
```

www.TopSage.com

```
  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：

(a) 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。

(b) 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。）

(c) 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 spaded
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

www.TopSage.com

第14章 预处理器

在程序里我们始终有话要说，但是所有已知的语言都无法表述得很好。

在前章中我们用过#define与#include指令，但没有深入讨论它们。这些指令，以及我们还没有学到的指令，都是由**预处理器**处理的。预处理器是一个小软件，它可以在编译前编辑C程序。C语言（和C++语言）因为依赖预处理器而不同于其他的编程语言。

预处理器是一个强大的工具，但它同时也可能是许多难以发现的错误的根源。同时，预处理器也经常被错误地用来编写出一些几乎不可能读懂的程序。尽管有些C程序员十分依赖于预处理器，我依然建议适度地使用它，就像许多其他生活中的事物一样。现代的C语言编程风格呼吁减少对于处理器的依赖。在C++中，对语言的改变使得可以更进一步限制预处理器的使用。

本章首先会描述预处理器的工作过程（14.1节），并且给出一些会影响全部预处理指令的通用规则（14.2节）。14.3节和14.4节涵盖介绍预处理器最主要的两种能力：宏定义和条件编译。（而处理器另外一个主要功能（即文件包含）将留到第15章再进行详细介绍。）14.5节讨论较少用到的预处理指令：#error、#line和#pragma。

14.1 预处理器的工作方式

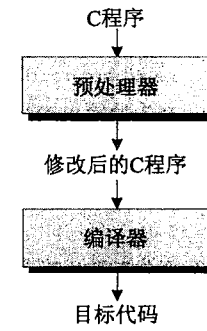
预处理器的行为是由**指令**控制的。这些指令是由#字符开头的一些命令。我们已经在前几章
273 中遇见过其中两种指令，#define和#include。

#define指令定义了一个**宏**——用来代表其他东西的一个名字，通常是某一类型的常量。预处理器会通过将宏的名字和它的定义存储在一起来响应#define指令。当这个宏在后面的程序中使用到时，预处理器“扩展”了宏，将宏替换为它所定义的值。

#include指令告诉预处理器打开一个特定的文件，将它的内容作为正在编译的文件的一部分“包含”进来。例如，下面这行指令

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

指示预处理器打开一个名字为stdio.h的文件，并将它的内容加到当前的程序中。（stdio.h包含了C语言标准输入/输出函数的原型。）



右图说明了预处理器在编译过程中的作用。预处理器的输入是一个C语言程序，程序可能包含指令。预处理器会执行这些指令，并在处理过程中删除这些指令。预处理器的输出是另一个程序：原程序的一个编辑后的版本，不再包含指令。预处理器的输出被直接交给编译器，编译器检查程序是否有错误，并经程序翻译为目标代码（机器指令）。

为了展现预处理器的作用，我们将它应用于2.6节的程序celsius.c。下面是原来的程序：

```
/* Converts a Fahrenheit temperature to Celsius */
```

www.TopSage.com

TopSage.com

```
#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 is: %.if\n", celsius);

   return 0;
}
```

274

预处理结束后，程序是下面的样子：

```
空行
空行
```

从stdio.h中引入的行

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

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

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

  printf("Celsius equivalent is: %.if\n", celsius);

  return 0;
}
```

预处理器通过把stdio.h的内容加入来响应#include指令。由于长度的原因，这里没有将stdio.h的内容显示出来。预处理器也删除了#define指令，并且替换了该文件中稍后出现在任何位置上的FREEZING_PT和SCALE_FACTOR。请注意预处理器并没有删除包含指令的行，而是简单地将它们替换为空。

正如这个例子所展示的那样，预处理器不仅仅是执行了指令，还做了一些其他的事情。特别值得注意的是，它将每一处注释都替换为一个空格字符。有一些预处理器还会进一步删除不必要的空白字符，包括在每一行开始用于缩进的空格符和制表符。

275

在C语言较早的时期，预处理器是一个单独的程序，并将它的输出提供给编译器。如今，预处理器通常和编译器集成在一起（为了提高编译的速度）。然而，我们仍然将它们认为是不同的程序。实际上，大部分C编译器提供一些方法，使用户可以看到预处理器的输出。一些编译器在打开特定的选项时（UNIX环境下通常是-P）仅产生预处理器的输出。其他一些编译器会提供一个独立的程序，这个程序的工作与集成的预处理器一致。如果需要更多的信息，可以查看你使用的编译器的文档。

注意，预处理器仅知道少量C语言的规则。因此，它在执行指令时非常有可能产生非法的程序。经常是源程序看起来没问题，使错误难以查找。对于较复杂的程序，检查预处理器的输出可能是找到这类错误的有效途径。

www.TopSage.com

14.2　预处理指令

大多数预处理指令属于下面3种类型：

- **宏定义**。#define指令定义一个宏，#undef指令删除一个宏定义。
- **文件包含**。#include指令导致一个指定文件的内容被包含到程序中。
- **条件编译**。#if、#ifdef、#ifndef、#elif、#else和#endif指令可以根据编译器可以测试的条件来将一段文本块包含到程序中或排除在程序之外。

剩下的#error、#line和#pragma指令是更特殊的指令，较少用到。本章将深入研究预处理指令。唯一一个我们不会在这里详细讨论的指令是#include，因为它会在15.2节介绍。

在进一步讨论之前，先来看几条应用于所有指令规则：

- **指令都以#开始**。#符号不需要在一行的行首，只要它之前只有空白字符就行。在#后是指令名，接着是指令所需要的其他信息。
- **在指令的符号之间可以插入任意数量的空格或横向制表符**。例如，下面的指令是合法的：

```
#    define    N    100
```

- **指令总是在第一个换行符处结束，除非明确地指明要继续**。如果想在下一行继续指令，我们必须在当前行的末尾使用\字符。例如，下面的指令定义了一个宏来表示硬盘的容量，按字节计算：

```
#define DISK_CAPACITY (SIDES *              \
                       TRACK_PER_SIDE *     \
                       SECTORS_PER_TRACK *  \
                       BYTES_PER_SECTOR)
```

276

- **指令可以出现在程序中任何地方**。我们通常将#define和#include指令放在文件的开始，其他指令则放在后面，甚至在函数定义的中间。
- **注释可以与指令放在同一行**。实际上，在一个宏定义的后面加一个注释来解释宏的意义是一种比较好的习惯：

```
#define FREEZING_PT 32.0     /* Freezing point of water */
```

14.3　宏定义

我们从第2章以来使用的宏被称为**简单的宏**，它们没有参数。预编译器也支持**带参数的宏**。本节会先讨论简单的宏，然后再讨论带参数的宏。在分别讨论它们之后，我们会研究一下二者共同的特性。

14.3.1　简单的宏

简单的宏定义有如下格式：

[#define指令（简单的宏）]　#define　标识符　替换列表

替换列表是一系列的C语言记号，包括标识符、关键字、数、字符常量、字符串字面量、运算符和标点符号。当预处理器遇到一个宏定义时，会做一个“标识符”代表“替换列表”的记录。在文件后面的内容中，不管标识符在任何位置出现，预处理器都会用替换列表代替它。

不要在宏定义中放置任何额外的符号，否则它们会被作为替换列表的一部分。一种常见的错误是在宏定义中使用 = ：

```
#define N = 100        /*** WRONG ***/
```

www.TopSage.com

```
int a[N];              /* 会成为 int a[= 100]; */
```

在上面的例子中，我们（错误地）把N定义成一对记号（= 和100）。

在宏定义的末尾使用分号结尾是另一个常见错误：

```
#define N 100;         /*** WRONG ***/
int a[N];              /*   become int a[100;]; */
```

这里N被定义为100和;两个记号。 277

在一个宏定义中，编译器可以检测到绝大多数由多余符号所导致的错误。但不幸的是，编译器会将每一处使用这个宏的地方标为错误，而不会直接找到错误的根源——宏定义本身，因为宏定义已经被预处理器删除了。

Q&A 简单的宏主要用来定义那些被Kernighan和Ritchie称为“明示常量”（manifest constant）的东西。使用宏，我们可以给数值、字符和字符串命名。

```
#define STE_LEN 80
#define TRUE    1
#define FALSE   0
#define PI      3.14159
#define CR      '\r'
#define EOS     '\0'
```

使用#define来为常量命名有许多显著的优点：

- **程序会更易读**。一个认真选择的名字可以帮助读者理解常量的意义。否则，程序将包含大量的“魔法数”，使读者难以理解。
- **程序会更易于修改**。我们仅需要改变一个宏定义，就可以改变整个程序中出现的所有该常量的值。“硬编码的”常量会更难于修改，特别是有时候当他们以稍微不同的形式出现时。（例如，如果一个程序包含一个长度为100的数组，它可能会包含一个从0到99的循环。如果我们只是试图找到所有程序中出现的100，那么就会漏掉99。）
- **可以帮助避免前后不一致或键盘输入错误**。假如数值常量3.14159在程序中大量出现，它可能会被意外地写成3.1416或3.14195。

虽然简单的宏常用于定义常量名，但是它们还有其他应用。

- **可以对C语法做小的修改**。实际上，我们可以通过定义宏的方式给C语言符号添加别名，从而改变C语言的语法。例如，对于习惯使用Pascal的begin和end（而不是C语言的{和}）的程序员，可以定义下面的宏：

  ```
  #define BEGIN  {
  #define END    }
  ```

  我们甚至可以发明自己的语言。例如，我们可以创建一个LOOP“语句”，来实现一个无限循环：

  ```
  #define LOOP   for (;;)
  ```

  当然，改变C语言的语法通常不是个好主意，因为它会使程序很难被其他程序员所理解。 278
- **对类型重命名**。在5.2节中，我们通过重命名int创建了一个Boolean类型：

  ```
  #define BOOL int
  ```

  虽然有些程序员会使用宏定义的方式来实现此目的，但类型定义（7.6节）仍然是定义新类型的最佳方法。
- **控制条件编译**。如将在14.4节中看到的那样，宏在控制条件编译中起重要的作用。例如，在程序中出现的宏定义可能表明需要将程序在“调试模式”下进行编译，来使用额外的语句输出调试信息：

  ```
  #define DEBUG
  ```

www.TopSage.com

大家网 TopSage.com

这里顺便提一下，如上面的例子所示，宏定义中的替换列表为空是合法的。

当宏作为常量使用时，C程序员习惯在名字中只使用大写字母。但是并没有如何将用于其他目的的宏大写的统一做法。由于宏（特别是带参数的宏）可能是程序中错误的来源，所以一些程序员更喜欢使用大写字母来引起注意。其他人则倾向于小写，即按照Kernighan和Ritchie编写的*The C Programming Language*一书中的样式。

14.3.2 带参数的宏

带参数的宏定义有如下格式：

[#define指令—带参数的宏]　#define 标识符（x_1, x_2, …, x_n）替换列表

其中x_1, x_2, …, x_n是标识符（宏的**参数**）。这些参数可以在替换列表中根据需要出现任意次。

> ⚠ 在宏的名字和左括号之间必须没有空格。如果有空格，预处理器会认为是在定义一个简单的宏，其中（$x_1, x_2, \cdots, x_n$）是替换列表的一部分。

当预处理器遇到一个带参数的宏，会将定义存储起来以便后面使用。在后面的程序中，如果任何地方出现了标识符（$y_1, y_2, \cdots, y_n$）格式的**宏调用**（其中$y_1, y_2, \cdots, y_n$是一系列标记），预处理器会使用替换列表替代，并使用y_1替换x_1，y_2替换x_2，依此类推。

例如，假定我们定义了如下的宏：

```
#define MAX(x,y)      ((x)>(y) ? (x) :(y))
#define IS_EVEN(n)    ((n)%2==0)
```

279

现在如果后面的程序中有如下语句：

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

预处理器会将这些行替换为

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

如这个例子所显示的，带参数的宏经常用来作为一些简单的函数使用。MAX类似一个从两个值中选取较大的值的函数。IS_EVEN则类似于另一种函数，该函数当参数为偶数时返回1，否则返回0。

下面的例子是一个更复杂的宏：

```
#define TOUPPER(c) ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
```

这个宏检测一个字符c是否在'a'与'z'之间。如果在的话，这个宏会用'c'减去'a'再加上'A'，来计算出c所对应的大写字母。如果c不在这个范围，就保留原来的c。像这样的字符处理的宏非常有用，所以C语言库在<ctype.h>（➤23.4节）中提供了大量的类似的宏。其中之一就是toupper，与我们上面的TOUPPER例子作用一致（但会更高效，可移植性也更好）。

带参数的宏可以包含空的参数列表，如下例所示：

```
#define getchar() getc(stdin)
```

空的参数列表不是一定确实需要，但可以使getchar更像一个函数。（没错，这就是<stdio.h>中的getchar，getchar的确就是个宏，不是函数——虽然它的功能像个函数。）

使用带参数的宏替代实际的函数有两个优点：

- **程序可能会稍微快些。**一个函数调用在执行时通常会有些额外开销——存储上下文信息、复制参数的值等。而一个宏的调用则没有这些运行开销。
- **宏会更“通用”。**与函数的参数不同，宏的参数没有类型。因此，只要预处理后的程序依然是合法的，宏可以接受任何类型的参数。例如，我们可以使用MAX宏从两个数中选出较大的一个，数的类型可以是int，long int，float，double等等。

www.TopSage.com

但是带参数的宏也有一些缺点。

- **编译后的代码通常会变大。**每一处宏调用都会导致插入宏的替换列表，由此导致程序的源代码增加（因此编译后的代码变大）。宏使用得越频繁，这种效果就越明显。当宏调用嵌套时，这个问题会相互叠加从而使程序更加复杂。思考一下，如果我们用MAX宏来找出3个数中最大的数会怎样？ 280

```
n = MAX(i, MAX(j, k));
```

下面是预处理后的这条语句：

```
n=((i)>(((j)>(k)?(j):(k)))?(i):(((j)>(k)?(j):(k))));
```

- **宏参数没有类型检查。**当一个函数被调用时，编译器会检查每一个参数来确认它们是否是正确的类型。如果不是，或者将参数转换成正确的类型，或者由编译器产生一个出错信息。预处理器不会检查宏参数的类型，也不会进行类型转换。
- **无法用一个指针来指向一个宏。**如在17.7节中将看到的，C语言允许指针指向函数。这一概念在特定的编程条件下非常有用。宏会在预处理过程中被删除，所以不存在类似的“指向宏的指针”。因此，宏不能用于处理这些情况。
- **宏可能会不止一次地计算它的参数。**函数对它的参数只会计算一次，而宏可能会计算两次甚至更多次。如果参数有副作用，多次计算参数的值可能会产生意外的结果。考虑下面的例子，其中MAX的一个参数有副作用：

```
n = MAX(i++, j);
```

下面是这条语句在预处理之后的结果：

```
n = ((i++)>(j)?(i++):(j));
```

如果i大于j，那么i可能会被（错误地）增加了两次，同时n可能被赋予了错误的值。

由于多次计算宏的参数而导致的错误可能非常难于发现，因为宏调用和函数调用看起来是一样的。更糟糕的是，这类宏可能在大多数情况下正常工作，仅在特定参数有副作用时失效。为了自保护，最好避免使用带有副作用的参数。

带参数的宏不仅适用于模拟函数调用。他们特别经常被作为模板，来处理我们经常要重复书写的代码段。如果我们已经写烦了语句

```
printf("%d"\n, x);
```

因为每次要显示一个整数x都要使用它。我们可以定义下面的宏，使显示整数变得简单些： 281

```
#define PRINT_INT(x)      printf("%d\n", x)
```

一旦定义了PRINT_INT，预处理器会将这行

```
PRINT_INT(i/j);
```

转换为

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

14.3.3 #运算符

宏定义可以包含两个运算符：#和##。编译器不会识别这两种运算符相反，它们会在预处理时被执行。

#运算符将一个宏的参数转换为字符串字面量。它仅允许出现在带参数的宏的替换列表中。Q&A（一些C程序员将#操作理解为“stringization（字符串化）”；其他人则认为这实在是对英语的滥用。）

www.TopSage.com

#运算符有大量的用途，这里只来讨论其中的一种。假设我们决定在调试过程中使用PRINT_INT宏作为一个便捷的方法，来输出一个整型变量或表达式的值。#运算符可以使PRINT_INT为每个输出的值添加标签。下面是改进后的PRINT_INT：

```
#define PRINT_INT(x) printf(#x " = %d\n", x)
```

x之前的#运算符通知预处理器根据PRINT_INT的参数创建一个字符串字面量。因此，调用

```
PRINT_INT(i/j);
```

会变为

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

在C语言中相邻的字符串字面量会被合并，因此上边的语句等价于：

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

当程序执行时，printf函数会同时显示表达式i/j和它的值。例如，如果i是11，j是2的话，输出为

```
i/j = 5
```

14.3.4　##运算符

##运算符可以将两个记号（例如标识符）“粘”在一起，成为一个记号。（无需惊讶，##运算符被称为“记号粘合”。）如果其中一个操作数是宏参数，“粘合”会在当形式参数被相应的实
282 际参数替换后发生。考虑下面的宏：

```
#define MK_ID(n) i##n
```

当MK_ID被调用时（比如MK_ID(1)），预处理器首先使用自变量（这个例子中是1）替换参数n。接着，预处理器将i和1连接成为一个记号（i1）。下面的声明使用MK_ID创建了3个标识符：

```
int MK_ID(1), MK_ID(2), MK_ID(3);
```

预处理后声明变为：

```
int i1, i2, i3;
```

##运算符不属于预处理器经常使用的特性。实际上，想找到一些使用它的情况是比较困难的。为了找到一个有实际意义的##的应用，我们来重新思考前面提到过的MAX宏。如我们所见，当MAX的参数有副作用时会无法正常工作。一种解决方法是用MAX宏来写一个max函数。遗憾的是，往往一个max函数是不够的。我们可能需要一个实际参数是int值的max函数，还需要参数为float值的max函数，等等。除了实际参数的类型和返回值的类型之外，这些函数都一样。因此，这样定义每一个函数似乎是个很蠢的做法。

解决的办法是定义一个宏，并使它展开后成为max函数的定义。宏会有唯一的参数type，它表示形式参数和返回值的类型。这里还有个问题，如果我们是用宏来创建多个max函数，程序将无法编译。（C语言不允许在同一文件中出现两个同名的函数。）为了解决这个问题，我们是用##运算符为每个版本的max函数构造不同的名字。下面是宏的显示形式：

```
#define GENERIC_MAX (type)          \
type type##_max(type x,  type y)    \
{                                   \
  return x > y ? x : y;             \
}
```

请注意宏的定义中是如何将type和_max相连来形成新函数名的。

现在，假如我们需要一个针对float值的max函数。下面是如何使用GENERIC_MAX宏来定义函数：

```
GENERIC_MAX(float)
```

预处理器会将这行展开为下面的代码：

www.TopSage.com

大家网 TopSage.com

```
float float_max(float x, float y) { return x > y ? x : y; }
```

14.3.5 宏的通用属性

现在我们已经讨论过简单的宏和带参数的宏了，我们来看一下它们都需要遵守的规则。 283

- **宏的替换列表可以包含对另一个宏的调用。**例如，我们可以用宏PI来定义宏TWO_PI：

```
#define PI     3.14159
#define TWO_PI (2*PI)
```

当预处理器在后面的程序中遇到TWO_PI时，会将它替换成（2*PI）。接着，预处理器会**重新检查**替换列表，看它是否包含其他宏的调用（在这个例子中，调用了宏PI）。Q&A 预处理器会不断重新检查替换列表，直到将所有的宏名字都替换掉为止。

- **预处理器只会替换完整的记号，而不会替换记号的片断。**因此，预处理器会忽略嵌在标识符名、字符常量、字符串字面量之中的宏名。例如，假设程序含有如下代码行：

```
#define SIZE 256

int BUFFER_SIZE;

if (BUFFER_SIZE > SIZE)
  puts ("Error : SIZE exceeded");
```

预处理后，这些代码行会变为：

```
int BUFFER_SIZE;

if (BUFFER_SIZE > 256)
  puts ("Error : SIZE exceeded");
```

标识符BUFFER_ZISE和字符串"Error: SIZE exceeded"没有被预处理影响，虽然它们都包含SIZE。

- **一个宏定义的作用范围通常到出现这个宏的文件末尾。**由于宏是由预处理器处理的，他们不遵从通常的范围规则。一个定义在函数中的宏并不是仅在函数内起作用，而是作用到文件末尾。
- **宏不可以被定义两遍，除非新的定义与旧的定义是一样的。**小的间隔上的差异是允许的，但是宏的替换列表（和参数，如果有的话）中的记号都必须一致。
- **宏可以使用#undef指令“取消定义”。**#undef指令有如下形式：

[#undef指令] `#undef 标识符`

其中标识符是一个宏名。例如，指令

```
#undef N
```

284

会删除宏N当前的定义。（如果N没有被定义成一个宏，#undef指令没有任何作用。）#undef指令的一个用途是取消一个宏的现有定义，以便于重新给出新的定义。

14.3.6 宏定义中的圆括号

在我们前面定义的宏的替换列表中有大量的圆括号。确实需要它们吗？答案是绝对需要。如果我们少用几个圆括号，宏可能有时会得到意料之外的——而且是不希望有的——结果。

对于在一个宏定义中哪里要加圆括号有两条规则要遵守。首先，如果宏的替换列表中有运算符，那么始终要将替换列表放在括号中：

```
#define TWO_PI (2*3.14159)
```

其次，如果宏有参数，每次参数在替换列表中出现时都要放在圆括号中：

```
#define SCALE(x) ((x)*10)
```

没有括号的话，我们将无法确保编译器会将替换列表和参数作为完整的表达式。编译器可能会

不按我们期望的方式应用运算符的优先级和结合性规则。

为了展示为替换列表添加圆括号的重要性，考虑下面的宏定义，其中的替换列表没有添加圆括号：

```
#define TWO_PI 2*3.14159
  /* 需要给替换列表加圆括号 */
```

在预处理时，语句

```
conversion_factor = 360/TWO_PI;
```

变为

```
conversion_factor = 360/2*3.14159;
```

除法会在乘法之前执行，产生的结果并不是期望的结果。

当宏有参数时，仅给替换列表添加圆括号是不够的。参数的每一次出现都要添加圆括号。例如，假设SCALE定义如下：

```
#define SCALE(x) (x*10)   /* 需要给x添加括号 */
```

在预处理过程中，语句

```
j = SCALE(i+1);
```

285 变为

```
j = (i+1*10);
```

由于乘法的优先级比加法高，这条语句等价于

```
j = i+10;
```

当然，我们希望的是

```
j = (i+1)*10;
```

在宏定义中缺少圆括号会导致C语言中最让人讨厌的错误。程序通常仍然可以编译通过，而且宏似乎也可以工作，仅在少数情况下会出错。

14.3.7 创建较长的宏

在创建较长的宏时，逗号运算符会十分有用。特别是可以使用逗号运算符来使替换列表包含一系列表达式。例如，下面的宏会读入一个字符串，再把字符串显示出来：

```
#define ECHO(s) (gets(s), puts(s))
```

gets函数和puts函数的调用都是表达式，因此使用逗号运算符连接它们是合法的。我们甚至可以把ECHO宏当作一个函数来使用：

```
ECHO(str);   /* 替换为 (gets(str), puts(str)); */
```

除了使用逗号运算符，我们也许还可以将gets函数和puts函数的调用放在大括号中形成复合语句：

```
#define ECHO(s)  { gets(s);  puts(s);  }
```

遗憾的是，这种方式并不奏效。假如我们将ECHO宏用于下面的if语句：

```
if (echo_flag)
  ECHO(str);
else
  gets(str);
```

将ECHO宏替换会得到下面的结果：

```
if (echo_flag)
  { gets(str);  puts(str);  };
```

www.TopSage.com

```
else
  gets(str);
```

编译器会将头两行作为完整的if语句：

```
if (echo_flag)
  { gets(str); puts(str); }
```

编译器会将跟在后面的分号作为空语句，并且对else子句产生出错信息，因为它不属于任何if语句。我们可以通过记住永远不要在ECHO宏后面加分号来解决这个问题。但是这样做会使程序看起来有些怪异。 286

逗号运算符可以解决ECHO宏的问题，但并不能解决所有宏的问题。假如一个宏需要包含一系列的语句，而不仅仅是一系列的表达式，这时逗号运算符就起不到帮助的作用了。因为它只能连接表达式，不能连接语句。解决的方法是将语句放在do循环中，并将条件设置为假：

```
do { ... } while (0)
```

do循环必须始终随跟着一个分号，因此我们不会遇到在if语句中使用宏那样的问题了。为了看到这个技巧（嗯，应该说是技术）的实际作用，让我们将它用于ECHO宏中：

```
#define ECHO(s)          \
        do {             \
          gets (s) ;     \
          puts (s) ;     \
          } while  (0)
```

当使用ECHO宏时，一定要加分号：

```
ECHO(str);
  /* becomes do {  gets(str); puts(str);  } while (0);  */
```

14.3.8 预定义宏

在C语言中预定义了一些有用的宏，见表14.1。这些宏主要是提供当前编译的信息。宏__LINE__和__STDC__是整型常量，其他3个宏是字符串字面量。我们在本章的后面会用到__STDC__宏，因此这里将重点放在其他的宏上。

表14-1 预定义宏

名 字	描 述
__LINE__	被编译的文件的行数
__FILE__	被编译的文件的名字
__DATE__	编译的日期（格式"Mmm dd yyyy"）
__TIME__	编译的时间（格式"hh:mm:ss"）
__STDC__	如果编译器接受标准C，那么值为1

__DATE__宏和__TIME__宏指明程序编译的时间。例如，假设程序以下面的语句开始：

```
printf("Wacky Windows (c) 1996 Wacky Software, Inc.\n");
printf("Compiled on %s at %s\n", __DATE__, __TIME__);
```

每次程序开始执行，程序都会显示下面两行： 287

```
Wacky Windows (c) 1996 Wacky Software, Inc.
Compiled on Dec 23 1996 at 22:18:48
```

这样的信息可以帮助区分同一个程序的不同版本。

我们可以使用__LINE__宏和__FILE__宏来找到错误。考虑下面这个检测被零除的除法的发生位置的问题。当一个C程序因为被零除而导致中止时，通常没有信息指明哪条除法运算导致错误。下面的宏可以帮助我们查明错误的根源：

```
#define CHECK_ZERO(divisor)  \
  if (divisor == 0)  \
```

www.TopSage.com

大家网 TopSage.com

```
printf("*** Attempt to divide by zero on line %d "  \
       "of file %s  ***\n",__LINE__, __FILE__)
```

CHECK_ZERO宏应该在除法运算前被调用：

```
CHECK_ZERO(j);
k = i / j;
```

如果j是0，会显示出如下形式的信息：

```
*** Attempt to divide by zero on line 9 of file FOO.c ***
```

类似这样的错误检测的宏非常有用。实际上，C语言库提供了一个通用的、用于错误检测的宏——assert宏（➤24.1节）。

14.4　条件编译

C语言的预处理器可以识别大量用于支持条件编译的指令。条件编译是指根据预处理器所执行的测试结果来包含或排除程序的片断。

14.4.1　#if 指令和#endif 指令

假如我们正在调试一个程序。我们想要程序显示出特定变量的值，因此将printf函数调用添加到程序中重要的部分。一旦找到错误，经常需要保留这些printf函数调用，以备以后使用。条件编译允许我们保留这些调用，但是让编译器忽略它们。

下面是我们需要采取的方式。首先定义一个宏，并给它一个非0的值：

```
#define DEBUG 1
```

288 宏的名字并不重要。接下来，我们要在每组printf函数调用的前后加上#if和#endif：

```
#if DEBUG
printf("Value of i : %d\n", i);
printf("Value of j : %d\n", j);
#endif
```

在于处理过程中，#if指令会测试DEBUG的值。由于DEBUG的值非0，因此预处理器会将这两个printf函数调用保留在程序中（但#if和#endif行会消失）。如果我们将DEBUG的值改为0并重新编译程序，预处理器则会将这4行代码都消失。编译器将不会看到这些printf函数调用，所以这些调用就不会在目标代码中占用空间，也不会在程序运行时浪费时间。我们可以将#if-#endif保留在最终的程序中，这样如果程序在运行时出错，可以继续产生这些诊断信息（将DEBUG改为1并重新编译）。

一般来说，#if指令的格式如下：

[#if指令]　`#if 常量表达式`

#endif指令则更简单：

[#endif指令]　`#endif`

Q&A 当预处理器遇到#if指令时，会计算常量表达式。如果表达式的值为0，那么在#if与#endif之间的行将在预处理过程中从程序中删除。否则，这些在#if和#endif之间的行会被保留在程序中，并继续被编译器处理——这时#if和#endif对程序没有任何影响。

对于没有定义过的标识符，#if指令会把它当作是值为0的宏对待。因此，如果省略DEBUG的定义，测试

```
#if DEBUG
```

会失败（但不会产生出错消息），而测试

```
#if !DEBUG
```

www.TopSage.com

会成功。

14.4.2 defined 运算符

14.3节中介绍过运算符#和##，还有另外一个运算符——defined，它仅用于预处理器。当defined应用于标识符时，如果标识符是一个定义过的宏返回1，否则返回0。#defined运算符通常与#if指令结合使用，允许写成 289

```
#if defined(DEBUG)
...
#endif
```

仅当DEBUG被定义成宏时，#if和#endif之间的代码会被保留在程序中。DEBUG两侧的括号不是必需的，因此可以简单写成

```
#if defined DEBUG
```

由于defined运算符仅检测DEBUG是否被定义为宏，所以不需要给DEBUG一个值：

```
#define DEBUG
```

14.4.3 #ifdef 指令和#ifndef 指令

#ifdef指令测试一个标识符是否已经定义为宏：

[#ifdef指令] `#ifdef 标识符`

#ifdef指令的使用与#if指令类似：

```
#ifdef              标识符
当标识符被定义为宏时需要包含代码
#endif
```

严格地说，Q&A并不需要#ifdef，因为我们可以组合#if指令和defined运算符来达到相同的效果。换而言之，指令

```
#ifdef 标识符
```

等价于

```
#if defined(标识符)
```

#ifndef指令与#ifdef指令类似，但是测试的是标识符是否没有被定义为宏：

[#ifndef指令] `#ifndef 标识符`

写成

```
#ifndef 标识符
```

等价于写成

```
#if !defined(标识符)
```

14.4.4 #elif 指令和#else 指令 290

#if指令、#ifdef指令和#ifndef指令可以像普通的if语句那样嵌套使用。当发生嵌套时，最好随着嵌套层次的增加而增加缩进。一些程序员对每一个#endif都加注释，来指明是对应于哪个条件测试的#if指令：

```
#if DEBUG
...
#endif   /* DEBUG */
```

这种方法可以帮助读者更方便地找到起始的#if指令。

为了提供更多的便利，预处理器提供了#elif和#else指令：

www.TopSage.com

[#elif指令]　#elif 表达式

[#else指令]　#else

#elif指令和#else指令可以与#if指令、#ifdef指令和#ifndef指令组合使用，来测试一系列条件：

```
#if 表达式1
当表达式1非0时需要包含的代码
#if 表达式2
当表达式1为0但表达式2非0时需要包含的代码
#else
其他情况下需要包含的代码
#endif
```

虽然上面的例子使用了#if指令，但#ifdef指令或#ifndef指令也可以替代使用。在#if指令和#endif指令之间可以有多个#elif指令，但最多只能有一个#else指令。

14.4.5 使用条件编译

条件编译对于调试是非常方便的，但它并不仅限于此。下面是其他一些常见的应用：

- **编写在多台机器或多种操作系统之间可移植的程序。**下面的例子中会根据WINDOWS、DOS或OS2是否被定义为宏，而将三组代码之一包含到程序中：

```
#if defined(WINDOWS)
...
#elif defined(DOS)
...
#elif defined(OS2)
...
#endif
```

291

一个程序中可能包含许多这样的#if指令。在程序的开头会定义这些宏之一（而且只有一个），由此选择了一个特定的操作系统。例如，定义OS2宏可以指明程序将运行在OS/2操作系统下。

- **编写可以使用不同的编译器进行编译的程序。**不同的编译器经常用于不同的C语言版本，这些版本之间会有一些差异。一些会接受标准C，另外一些则不会。一些版本会包含针对特定机器的扩展，而其他的则或没有，或提供不同的扩展集。条件编译可以使程序适应于不同的编译器。思考编写这样一个程序，当使用标准C的编译器进行编译时，可不可能被编译成功。__STDC__宏允许预处理器检测编译器是否支持标准C，如果不支持，我们可能必须修改程序的某些方面，尤其是有可能必须使用经典C的函数声明替代标准C的函数原型。对于每一处函数的声明，我们都可以使用下面的代码：

```
#if __STDC__
标准C函数原型
#else
经典C函数声明
#endif
```

- **为宏提供默认定义。**条件编译使我们可以检测一个宏当前是否已经被定义了，如果没有，则提供一个默认的定义。例如，如果它还没有被定义的话，下面的代码会定义宏BUFFER_SIZE：

```
#ifndef BUFFER_SIZE
#define BUFFER_ZISE 256
#endif
```

- **临时屏蔽包含注释的代码。**由于在标准C中注释不能嵌套，因此不能直接“注释掉”包

www.TopSage.com

TopSage.com

含注释的代码。然而，可以使用#if指令替代：

```
#if 0
包含注释的代码行
#endif
```

Q&A 将代码以这种方式屏蔽掉经常称为“条件屏蔽”。 292

15.2节会讨论另外一种条件编译的常用用途：保护头文件以避免重复包含。

14.5 其他指令

在本章的最后，我们将简要地了解一下#error指令、#line指令和#pragma指令。这些指令有一个共同点：C语言的初学者不会经常使用它们。实际上，本书中的所有程序都不会使用它们，因此你可以放心地跳过本节。以后，当你准备成为一个C语言专家时，你会需要熟悉这些指令。

14.5.1 #error 指令

#error指令有如下格式：

[#error 指令] `#error 消息`

其中，消息是一个C语言标记序列。如果预处理器遇到一个#error指令，它会显示一个出错消息，这个出错消息一定会包含消息。对于不同的编译器，出错消息的具体形式也可能会不一样。下面是一个典型的示例：

```
Error directive: 记号序列
```

碰到#error指令预示着一个严重的程序错误，大多数编译器会立即终止编译而不去找出其他错误。

#error指令通常与条件编译指令一起用于检测正常编译过程中不应出现的情况。例如，假定我们需要确保，一个程序在一台无法使用int类型来存储大于100 000的数的机器上不能编译。最大允许的int值用INT_MAX宏（➤23.2节）表示，所以我们需要做的就是当INT_MAX宏不超过100 000时调用#error指令：

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

如果试图在一台整型以16位存储的机器上编译这个程序，将产生一条出错消息：

```
Error directive: int type is too small
```

#error指令通常会出现在#if-#elif-#else序列中的#else部分：

```
#if defined(WINDOWS)
...
#elif defined(DOS)
...
#elif defined(OS2)
...
#else
#error No operating system specified
#endif
```

293

14.5.2 #line 指令

#line指令是用来改变给程序行编号的方式的。（程序行通常是按1，2，3，…来编号的。）我们也可以使用这条指令使编译器认为它正在从一个有不同名字的文件中读取一个程序。

#line指令有两种形式。在一种形式中，指定一个行号：

[#line 指令（形式1）]　`#line n`

*n*必须是介于1到32 767之间的整数。这条指令导致程序后面的行被编号为*n*、*n*+1、*n*+2等。

在#line的第二种形式中，需要同时指定行号和文件名：

[#line指令（形式2）]　`#line n "文件名"`

指令后面的行会被认为是来自文件，行号由*n*开始。

#line指令的一种作用是改变__LINE__宏（可能还有__FILE__宏）的值。更重要的是，大多数编译器会使用来自#line指令的信息产生出错消息。例如，假设下列指令出现在文件foo.c的开头：

```
#line 10 "bar.c"
```

现在，假设编译器在foo.c的第5行监测到一个错误。出错消息会指向bar.c的第13行，而不是foo.c的第5行。（为什么是第13行呢？因为指令占据了foo.c的第1行，因此对foo.c的重新编号从第2行开始，并将这一行作为bar.c的第10行。）

乍一看，#line指令使人迷惑。为什么要使出错消息指向另一行，甚至是另一个文件呢？这样不是会使程序变得难以调试吗？

实际上，程序员并不经常使用#line指令。然而，它主要用于那些产生C代码作为输出的程序。程序yacc（Yet Another Compiler-Compiler）是其中著名的程序之一。yacc是一个UNIX工具，用于自动生成部分的编译器。在使用yacc之前，程序员需要准备一个包含yacc所需要的信息以及C代码段的文件。通过这个文件，yacc生成了一个C程序y.tab.c，并合并了程序员所提供的代码。程序员接着按照正常方法编译y.tab.c。通过在y.tab.c中插入#line指令，yacc会使编译器认为代码来自原始文件——也就是程序员写的那个文件。于是，任何编译y.tab.c时产生的出错消息会指向原始文件中的行，而不是y.tab.c中的行。其最终结果是：调试变得更容易，因为出错消息都指向程序员编写的文件，而不是（更复杂的）由yacc生成的文件。

294

14.5.3　#pragma 指令

#pragma指令为要求编译器执行某些特殊操作提供了一种方法。这条指令对非常大的程序或需要使用特定编译器的特殊功能的程序非常有用。

#pragma指令有如下形式：

[#pragma指令]　`#pragma 记号`

其中，记号是一般C语言的记号。#pragma指令通常只跟着一个记号，这个记号表示了一条编译器需要服从的命令。

一些编译器允许#pragma指令所包含的不仅是简单的命令。特别是有些编译器允许#pragam指令带参数：

```
#pragma data(heap_size => 1000, stack_size => 2000)
```

#pragma指令中出现的命令集在不同的编译器上是不一样的。你必须通过查阅你所使用的编译器的文档来了解哪些命令是可以使用的，以及这些命令的功能。顺便提一下，如果#pragma指令包含了无法识别的命令，编译器必须忽略这些#pragma指令，不允许产生出错消息。

问与答

问：我看到在有些程序中#单独占一行。这样是合法的吗？

答：是合法的。这就是所谓的**空指令**，它没有任何作用。一些程序员用空指令作为条件编译模块之间的间隔：

www.TopSage.com

```
#if INT_MAX < 100000
#
#error int type is too small
#
#endif
```

295

当然，空行也可以。不过#可以帮助读者看出模块的范围。

问：我不清楚程序中哪些常量需要定义成宏。有没有一些可以参照的规则？（p.195）

答：一条首要的规则是每一个数字常量，如果不是0和1，就需要定义成宏。字符常量和字符串常量有一点复杂，因为使用宏来替换字符或字符串常量并不总是提高程序的可读性。我个人建议在下面的条件下使用宏来替代字符或字符串常量：（1）常量被不止一次地使用；（2）存在常量某天被修改的可能。根据第二条规则，我不会像这样使用宏：

```
#define NUL '\0'
```

虽然有些程序员会使用。

***问：如果要被“字符串化”的参数包含 " 或 \ 字符，#运算符会如何处理？（p.197）**

答：它会将 " 转换为 \"，\ 转换为 \\。考虑下面的宏：

```
#define STRINGIZE(x) #x
```

预处理器会将STRINGIZE("foo")替换为"\"foo\""。

***问：我无法使下面的宏正常工作：**

```
#define CONCAT(x,y) x##y
```

如期望那样，CANCAT(a,b)会给出ab，但CONCAT(a,CONCAT(b,c))会给出一个怪异的结果。这是为什么？

答：感谢那些连Kernighan和Ritchie都承认“怪异”的规则，替换列表中依赖##的宏通常不能嵌套调用。这里的问题在于CONCAT(a,CONCAT(b,c))不会按照“正常”的方式扩展——CONCAT(b,c)首先得出bc，然后CONCAT(a,bc)给出abc。C标准指明，在替换列表中，位于##运算符之前和之后的宏参数在替换时不被扩展，结果，CONCAT(a,CONCAT(b,c))扩展成aCONCAT(b,c)，而不会进一步扩展，因为没有名为aCONCAT的宏。

有一种办法可以解决这个问题，但不太好看。技巧是定义第二个宏，只是简单地调用第一个宏：

```
#define CONCAT2(x,y) CONCAT(x,y)
```

写成CONCAT2(a,CONCAT2(b,c))就会得到我们希望的结果。在扩展外面的CONCAT2调用时，预处理器将会同时扩展CONCAT2(b,c)。这里的区别在于CONCAT2的扩展列表不包含##。如果这个也不行，那也不用担心，这种问题并不是经常遇到的。

296

顺便提一下，#运算符也有同样的问题。如果#x出现在替换列表中，其中x是一个宏参数，其对应的实际参数也不会被扩展。因此，假设N是一个代表10的宏，且STR(x)包含替换列表#x，STR(N)扩展的结果为"N"，而不是"10"。解决的方法与我们在CONCAT例子中类似：定义第二个宏来调用STR。

***问：如果预处理器再重新扫描时又发现了最初的宏名会如何处理呢？例如下面的例子：**

```
#define N (2*M)
#define M (N+1)

i = N;    /* infinite loop?*/
```

预处理器会将N替换为(2*M)，接着将M替换为(N+1)。预处理器还会再次替换N，成为一个无限循环吗？（p.199）

答：一些早期的预处理器确实会进入无限循环，但符合标准C的预处理器则不会。按照C语言标准，如果在扩展宏的过程中原先的宏名重复出现的话，宏名不会再次被替换。下面是预处理后i的赋值语句的形式：

```
i = (2*(N+1));
```

一些大胆的程序员会通过编写其名字与保留字或标准库中的函数名匹配的宏来利用这一行为。

www.TopSage.com

大家网 TopSage.com

以库函数sqrt为例。sqrt函数（➤23.3.5节）计算参数的平方根，如果参数为负数则返回一个由实现定义的值，我们可能希望当参数为负数时返回0。由于sqrt是标准库函数，我们无法很容易地修改它。但是我们可以定义一个宏，使它在参数为负数时返回0：

```
#define sqrt(x)  ((x)>0 ? sqrt(x) : 0)
```

预处理器会截获sqrt的调用，并将它替换成上面的条件表达式。在扫描宏的过程中条件表达式中的sqrt调用不会被替换，因此会被保留由编译器处理。

297

问：我觉得预处理器就是一个编辑器。它如何计算常量表达式呢？（p.202）

答：预处理器比你想的要复杂。它足够“了解”C语言，所以能够计算常量表达式。虽然它不会完全按照编译器的方式去做。（例如，预处理器认为所有未定义的名字的值为0。其他的差异太深奥，就不再深入了。）在实际使用中，预处理器常量表达式中的操作数通常为常量、表示常量的宏或defined运算符的应用。

问：为什么C提供#ifdef指令和#ifndef指令，既然我们可以使用#if指令和define运算符达到同样效果？（p.203）

答：#ifdef指令和#ifndef指令从20世纪70年代就在C语言中存在了，而defined运算符则是在80年代的标准化过程中加到C语言中的。因此，实际的问题是：为什么将defined运算符加到C语言中？答案就是defined增加了灵活性。我们现在可以使用#if和defined运算符来测试任意数量的宏，而不再是只能使用#ifdef和#ifndef对一个宏进行测试。例如，下面的指令检查是否FOO和BAR被定义了而BAZ没有被定义：

```
#if defined(FOO) && defined(BAR) && !defined(BAZ)
```

问：我想编译一个还没有写完的程序，因此我“条件屏蔽”未完成的部分。我在开始的地方加了一条信息来提示自己以后将程序写完：

```
#if 0
Haven't finished this part yet.
...
#endif
```

为什么我在编译时会得到一条出错消息呢？预处理器不是简单地忽略#if指令和#endif指令之间的所有行吗？（p.205）

答：在#if指令和#endif指令之间的行必须由预处理记号（➤附录A）组成。预处理记号类似于C语言记号（标识符、运算符、数等）。当预处理器试图将第一行分解为记号时，会遇到Haven（一个合法的标识符），接着是't（一个非法的字符常量）。一些预处理器会跳过#if指令和#endif指令之间所有的行而不会检查预处理记号，但这些预处理器并没有严格遵守C语言标准的规则。

练习

14.3节

1. 编写宏来计算下面的值。
   (a) x的立方。
   (b) x除以4的余数。
   (c) 如果x与y的乘积小于100值为1，否则值为0。
   你写的宏始终正常工作吗？如果不是，哪些参数会失败呢？
2. 编写一个宏NELEMS(a)来计算一个一维数组a中元素的个数。提示：使用sizeof运算符。
3. 假定DOUBLE是如下宏：

   ```
   #define DOUBLE(x) 2*x
   ```

   (a) DOUBLE(1+2)的值是多少？
   (b) 4/DOUBLE(2)的值是多少？

www.TopSage.com

(c) 改正DOUBLE的定义。 298

4. 针对下面每一个宏，举例说明宏的问题，并提出修改方法。

(a) `#define AVG(x,y)  (x+y)/2`

(b) `#define AREA(x,y)  (x)*(y)`

*5. 下面的宏定义有问题：

```
#define ABS(a)((a)<0?-(a):a)
```

举例说明为什么ABS不能正常工作，并提出修改方法。你可以假定ABS的参数没有副作用。

6. 假定TOUPPER定义成下面的宏：

```
#define TOUPPER(c)  ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
```

假设s是一个字符串，i是一个int型变量。给出下面每个代码段所产生的输出。

(a)
```
strcpy(s, "abcd");
i = 0;
putchar(TOUPPER(s[++i]));
```

(b)
```
strcpy(s, "0123");
i = 0;
putchar(TOUPPER(s[++i]));
```

7. (a) 编写宏DISP(f,x)，使其扩展后调用printf函数来显示函数f的参数为x时的值。例如：

```
DISP(sqrt, 3.0);
```

应该扩展为

```
printf("sqrt(%g) = %g\n", 3.0, sqrt(3.0));
```

(b) 编写宏DISP2(f,x,y)，类似DISP但应用于有两个参数的函数。

*8. 假定GENERIC_MAX是如下宏：

```
#define GENERIC_MAX(type)           \
type type##_max(type x, type y)     \
{                                   \
   return x > y ? x : y;            \
}
```

(a) 写出GENERIC_MAX(long)被预处理器扩展后的形式。

(b) 解释为什么GENERIC_MAX不能应用在像unsigned long这样的基本类型上？

(c) 如何使GENERIC_MAX对任何基本类型都可以正常工作？提示：不要改变GENERIC_MAX的定义。

*9. 如果需要一个宏，使它展开后包含当前行号和文件名。换而言之，我们会写

```
const char *str = LINE_FILE;
```

扩展后为

```
const char *str = "Line 10 of file foo.c";
```

其中foo.c是包含程序的文件，10是调用LINE_FILE行的行号。警告：这个练习仅针对高级程序员。尝试编写前请认真阅读“问与答”小节的内容！ 299

14.4节

10. 假定宏M有如下定义：

```
#define M 10
```

下面哪项测试会失败？

(a) `#if M`

(b) `#ifdef M`

(c) `#ifndef M`

(d) `#if defined(M)`

(e) `#if !defined(M)`

11. (a) 指出下面的程序预处理后的形式。

www.TopSage.com

```
#define N 100

void f(void);

main()
{
  f();
#ifdef N
#unde f N
#endif
  return 0;
}

void f(void)
{
#if defined(N)
  printf("N is %d\n", N);
#else
  printf("N is undefined\n");
#endif
}
```

(b) 这个程序的输出是什么？

12. 指出下面的程序预处理后的形式。其中有几行可能会导致编译错误，请找出这些错误。

```
#define N = 10
#define INC(x) x+1
#define SUB (x,y) x - y
#define SQR(x) ((x)*(x))
#define CUBE(x) (SQR(x)*(x))
#deflne M1(x,y) x##y
#define M2(x,y) #x #y

main ( )
{
  int a[N],  i,  j, k, m;
#ifdef N
  i = j;
#else
  j = i;
#endif
  i = 10 * INC(j);
  i = SUB(j, k);
  i = SQR(SQR(j++)) ;
  i = CUBE(j);
  i = M1(j, k);
  puts(M2(i, j)) ;
#undef SQR
  i = SQR(j);
#define SQR
  i = SQR(j);
  return 0 ;
)
```

300

301

www.TopSage.com

大家网
TopSage.com

第15章 编写大规模程序

> 很难找到正确的时间单位来衡量计算机的发展。有些大教堂用了一个世纪才建成。
> 你能想象一个壮丽辉煌的大程序也能花这么长的时间吗?

虽然某些C程序小得足够放入一个单独的文件中，但是大多数程序都不是这样的。程序由多个文件构成的原则更容易让人接受。本章将会看到一个由几个**源文件**（source file）以及一些通常的**头文件**（header file）组成的典型程序。源文件包含函数的定义和外部变量，而头文件包含可以在源文件之间共享的信息。15.1节讨论源文件，15.2节详细地介绍头文件，15.3节描述把程序分割成源文件和头文件的方法，15.4节说明如何“构建”（即编译和链接）由多个文件组成的程序，以及在改变程序的部分内容后如何进行“重新构建”。

15.1 源文件

到现在为止一直假设C程序是由单独一个文件组成的。事实上，可以把程序分割成一定数量的**源文件**。根据惯例，源文件的扩展名为.c。每个源文件包含程序的部分内容，主要是函数的定义和变量。一个源文件必须包含名为main的函数，此函数作为程序的起始点。

例如，假设打算编写一个简单计算器程序，用来计算按照逆波兰符号（Reverse Polish Notation，RPN）录入的整数表达式。所谓逆波兰符号是指运算符都跟在操作数的后边。如果用户录入表达式

```
30 5 - 7 *
```

我们希望程序可以显示出此表达式的值（此例中值为175）。如果可以使程序逐个读入操作数和运算符，那么利用栈跟踪中间结果这样的方式计算逆波兰表达式是很容易的。如果程序读取数，就把此数压入栈。如果程序读取运算符，那么将从栈顶弹出两个数进行相应的运算，然后把结果压入栈。当程序执行到用户输入的末尾时，表达式的值将在栈中。例如，程序将按照下列方式计算表达式30 5 - 7 *的值：

(1) 把30压入栈。
(2) 把5压入栈。
(3) 从栈顶弹出两个数，30减去5，结果为25，然后把此结果压回到栈中。
(4) 把7压入栈。
(5) 从栈顶弹出两个数，两数相乘，然后把结果压回到栈中。

在这些步骤后，栈将包含表达式的值（即175）。

把这种策略转换为程序并不困难。程序的main函数将用循环来执行下列动作：

- 读取“记号”（数或运算符）。
- 如果记号是数，那么把它压入栈。
- 如果记号是运算符，那么从栈顶弹出它的操作数进行运算，然后把结果压入栈中。

当像这样把程序分割成文件时，将相关的函数和变量放入同一文件中是很有意义的。读取

www.TopSage.com

记号的函数可能和任何需要用到记号的函数一起属于某个源文件（比如说token.c文件）。比如push函数、pop函数、make_empty函数、is_empty函数和is_full函数这些与栈相关的函数可能都属于一个不同的stack.c文件。表示栈的变量也属于stack.c文件，而main函数则可以在另一个calc.c文件中。

把程序分裂成多个源文件有许多显著的优点：

- 把相关的函数和变量集合在单独一个文件中可以帮助明了程序的结构。
- 可以单独对每一个源文件进行编译。如果程序规模很大而且需要频繁改变（这一点在程序开发过程中是非常普遍的）的话，这种方法可以极大地节约时间。
- 当把函数集合在单独的源文件中时，会更容易在其他程序中重新使用这些函数。在示例中，把stack.c和token.c从main函数中分离出来使得在今后更容易重新使用栈函数和记号函数。

304

15.2　头文件

当把程序分割为几个源文件时，问题也随之产生了：某文件中的函数如何能调用定义在其他文件中的函数呢？函数如何能访问其他文件中的外部变量呢？两个文件如何能共享同一个宏定义或类型定义呢？答案取决于#include指令，此指令使得在任意数量的源文件中共享信息成为可能，其中，这些信息可以是函数原型、宏定义、类型定义等。

#include指令告诉预处理器打开指定的文件，并且把此文件的内容插入到当前文件中。因此，如果打算几个源文件可以访问相同的信息，那么将把此信息放入文件中，然后利用#include指令把文件的内容带进每个源文件中。把按照此种方式包含的文件称为是**头文件**（或者有时称为**包含文件**）。本节后将会更详细地讨论头文件。根据惯例，头文件的扩展名为.h。

注意：C标准使用术语“源文件”来指示程序员编写的全部文件，包括.c文件和.h文件。而这里“源文件”只是指.c文件。

15.2.1　#include 指令

#include指令有两种书写格式。第一种格式用于属于C语言自身库的头文件：

[#include指令格式1]　`#include <文件名>`

第二种格式用于所有其他头文件，也包含任何自己编写的文件：

[#include指令格式2]　`#include "文件名"`

编译器定位头文件的方式是两种格式间的细微差异。Q&A下面是大多数编译器遵循的规则：

- #include <文件名>：搜寻系统头文件所在的目录（或多个目录）。例如，在UNIX系统中，通常把系统头文件保存在目录/usr/include中。
- #include "文件名"：搜寻当前目录，然后搜寻系统头文件所在的目录（或多个目录）。通常可以改变搜寻头文件的位置，这种改变经常利用诸如-I路径这样的命令行选项来实现。

305

不要在包含自行编写的头文件时用尖括号：

```
#include <myheader.h> /*** WRONG ***/
```

因为预处理器将可能在保存系统头文件的地方寻找 myheader.h（显然是找不到的）。

在#include指令中的文件名可以含有帮助定位文件的信息，比如目录的路径或驱动器号：

```
#include "C:\cprogs\utils.h"   /* DOS path*/
```

www.TopSage.com

```
#include "/cprogs/utils.h"     /* UNIX path*/
```

虽然#include指令中记号的双引号使得文件名看起来像字符串字面量，但是预处理器不会把它们作为字符串字面量来处理。(这是幸运的，因为在DOS例子中，字符串字面量中出现的\c和\u，将会被作为转义序列处理。)

可移植性技巧 通常最好的做法是在#include指令中不包含路径或驱动器的信息。当把程序转移到其他机器上，或者更糟的情况是转移到其他操作系统上时，这类信息会使编译程序变得很困难。

例如，下面的这些#include指令指定了驱动器或路径信息，而这些信息不可能一直是有效的：

```
#include "d-utils.h"
#include "\cprogs\utils.h"
#include "d:\cprogs\utils.h"
```

下列这些指令相对好一些。它们没有限制特殊的驱动器，而且指定的目录与当前目录相关：

```
#include <sys\stat.h>
#include "utils.h"
#include "..\include\utils.h"
```

15.2.2 共享宏定义和类型定义

大多数大规模的程序包含用于几个源文件（或者，最极端的情况是用于全部源文件）共享的宏定义和类型定义。这些定义应该放在头文件中。

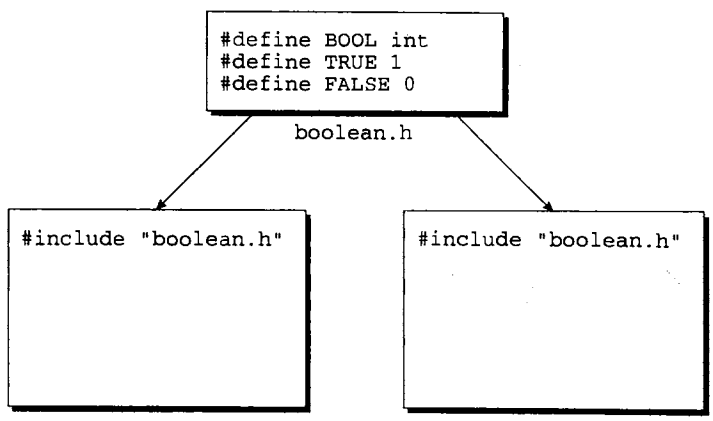
例如，假设正在编写的程序使用名为BOOL、TRUE和FALSE的宏。不用在每个需要的源文件中重复定义这些宏，而是把这些定义放在像名为boolean.h这样的头文件中，这样做是很有意义的： 306

```
#define BOOL int
#define TRUE 1
#define FALSE 0
```

任何需要这些宏的源文件只需简单包含下面这一行：

```
#include "Boolean.h"
```

在下面的图中，两个文件包含了boolean.h。



类型定义在头文件中也是很普遍的。例如，不用定义BOOL宏，而是可以用typedef产生一个Bool类型。如果这样做，boolean.h文件将有下列显示：

```
#define TRUE 1
#define FALSE 0
typedef int Bool;
```

www.TopSage.com

大家网 TopSage.com

把宏定义和类型定义放在头文件中有许多明显的好处。首先，不用把定义复制给需要的源文件可以节约时间。其次，程序变得更加容易修改。改变宏定义或类型定义只需要编辑单独的头文件，而不需要修改许多使用宏或类型的源文件。最后，不需要担心由于源文件包含相同宏或类型的不同定义而导致的矛盾。

15.2.3　共享函数原型

假设源文件包含函数f的调用，而函数f是定义在另一个foo.c文件中的。首先，调用没有声明的函数f是非常危险的。没有函数原型可依赖，迫使编译器假定函数f的返回类型是int型
307 的，而且假定形式参数的数量和函数f的调用中的实际参数数量是匹配的。通过默认的实际参数提升（➤9.3.1节），实际参数自身自动转化为“标准格式”。编译器的假设也可能是错误的，但是，因为一次只能编译一个文件，所以是没有办法进行检查的。如果假设是错误的，那么程序大概无法工作，而且没有任何作为原因的线索。

当调用定义在其他文件中的函数f时，要始终确保编译器在调用之前看到函数f的原型。

第一个冲动可能是在调用函数f的文件中声明它。这样可以解决问题，但是可能产生持续的“噩梦”。假设50个源文件要调用函数，如何能确保函数f的原型在所有文件中都一样呢？如何能保证这些原型和foo.c文件中函数f的定义相匹配呢？如果稍后函数f发生了改变，如何能找到所有用到此函数的文件呢？

解决办法是显而易见的：把函数f的原型放进头文件中，然后在所有调用函数f的地方包含头文件。Q&A既然在文件foo.c中定义了函数f，那么就让我们把头文件命名为foo.h。除了在调用函数f的源文件中包含foo.h，还将需要把它包含在foo.c中，从而使编译器检查foo.h中函数f的原型是否和foo.c中的函数定义相匹配成为可能。

在含有函数f定义的源文件中始终包含声明函数f的头文件。如果这样做失败可能导致难以发现的错误，因为在程序别处对函数f的调用可能会和函数f的定义不匹配。

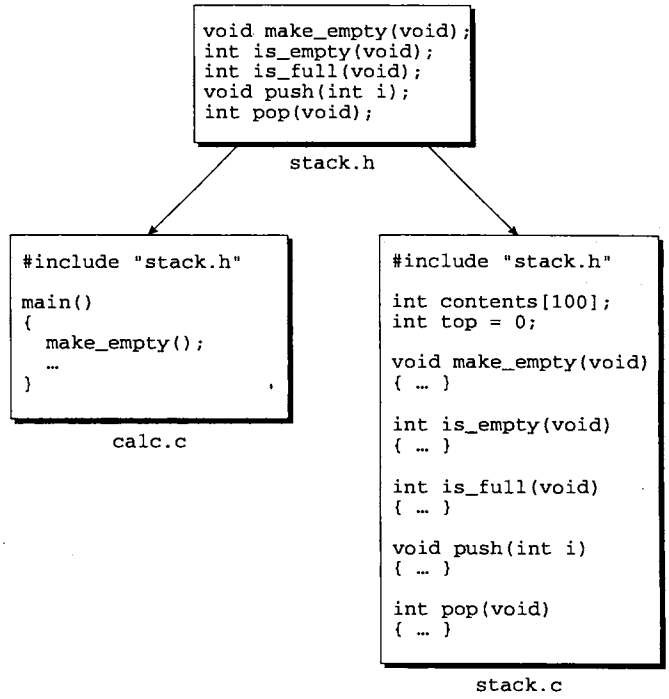
如果文件foo.c包含其他函数，那么应该像函数f一样在同一个头文件中声明大多数的函数。毕竟，文件foo.c中的其他函数大概会与函数f有关。任何含有函数f调用的文件可能会需要文件foo.c中的其他一些函数。然而，打算仅用于文件foo.c中的函数不需要在头文件中声明，如果声明了将会产生误解。

为了说明头文件中函数原型的使用，一起回到15.1节逆波兰计算器的示例。文件stack.c将包含函数make_empty、函数is_empty、函数is_full、函数push和函数pop的定义。这些函数的原型应该在头文件stack.h中：

```
void make_empty(void);
int is_empty (void);
int is_full (void);
void push(int i);
int pop (void);
```

（为了避免示例复杂化，函数is_empty和函数is_full将不再返回Bool型值而返回int型值。）文件calc.c中将包含stack.h以便于编译器将知道每个函数的返回类型，以及形式参数的数量和类型。文件stack.c中也将包含stack.h以便于编译器可以检查stack.h中的函数原型是否
308 和stack.c中的定义相匹配。下面这张图说明了stack.h、stack.c和calc.c。

www.TopSage.com



15.2.4 共享变量声明

就像在函数间共享变量的方式一样，变量可以在文件中共享。为了共享函数，要把函数的定义放在一个源文件中，然后在需要调用此函数的其他文件中放置声明。共享变量的方法和此方式非常类似。

在此之前，不需要区别变量的声明和它的定义。为了声明变量i，写成如下形式：

```
int i;   /* declares i and defines it as well */
```

这样不仅声明i是int型的变量，而且也对i进行了定义，从而使编译器为i留出了空间。为了声明没有定义的变量i，需要在变量声明的开始处放置关键字extern：

```
extern int i;/* declares i without defining it */
```

extern提示编译器变量i是在程序中的其他位置定义的（大多数可能是在不同的源文件中），因此不需要为i分配空间。

顺便说一句，extern可以用于所有类型的变量。在数组的声明中使用extern时，可以忽略数组的长度：

```
extern int a[];
```

309

Q&A 因为此刻编译器不用为数组a分配空间，所以也就不需要知道数组a的长度了。

为了在几个源文件中共享变量i，首先把变量i的定义放置在一个文件中：

```
int i;
```

如果需要对变量i初始化，那么可以在这里放初始值。在编译这个文件时，编译器将会为变量i分配内存空间，而其他文件将包含变量i的声明：

```
extern int i;
```

通过在每个文件中声明变量i，使得在这些文件中可以访问/或修改变量i。然而，由于关键字extern，使得编译器不会在每次编译其中某个文件时为变量i分配额外的内存空间。

当在文件中共享变量时，会面临和共享函数时相似的挑战：确保变量的所有声明和变量的定义一致。

www.TopSage.com

当同一个变量的声明出现在不同文件中时，编译器无法检查声明是否和变量定义相匹配。例如，一个文件可以包含如下定义：

```
int i;
```

同时另一个文件包含声明

```
extern long int i;
```

这类错误可能导致程序的行为异常。

为了避免矛盾，通常把共享变量的声明放置在头文件中。需要访问特殊变量的源文件可以稍后包含适当的头文件。此外，含有变量定义的源文件包含每一个含有变量声明的头文件，这样使编译器可以检查两者是否匹配。

虽然在文件中共享变量是C语言界中的长期惯例，但是它有重大的缺点。在19.2节中将会看到这个问题的内容，并且学习如何设计不需要共享变量的程序。

15.2.5　嵌套包含

头文件自身可以包含#include指令。虽然这种做法可能看上去有点奇怪，但实际上却是十
310 分有用的。思考含有下列原型的stack.h文件：

```
int is_empty(void);
int is_full (void);
```

既然这些函数只能返回0或1，那么声明它们的返回类型是Bool型而不是int型是一个很好的主意：

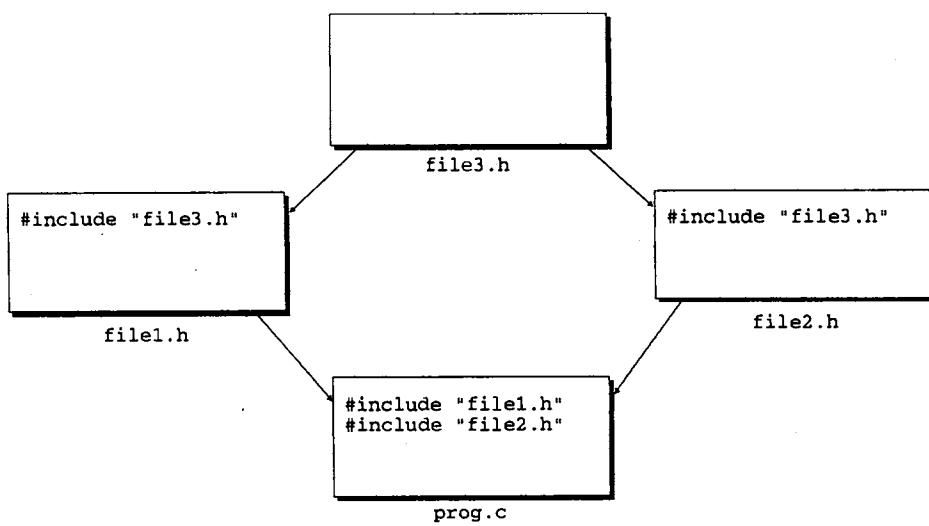
```
Bool is_empty (void);
Bool is_full (void);
```

当然，将需要在stack.h中包含文件boolean.h以便在编译stack.h时Bool的定义是有效的。

就传统而言，C程序员避免使用嵌套包含。(C语言的早期版本根本不允许嵌套包含。)但是，这种对嵌套包含的偏见正在逐渐减弱。C++一个原因就是嵌套包含在C++语言中的普遍应用。

15.2.6　保护头文件

如果源文件包含同一个头文件两次，那么可能产生编译错误的结果。当头文件包含其他头文件时，这种问题十分普遍。例如，假设file1.h包含file3.h，file2.h包含file3.h，而prog.c同时包含file1.h和file2.h：



www.TopSage.com

TopSage.com

在编译prog.c时，就将会编译两次file3.h。

包含同一个头文件两次不会总是造成编译错误。如果文件只包含宏定义、函数原型和/或变量声明，那么将不会有任何困难。然而，如果文件包含类型定义，则会带来编译错误。 311

就安全而言，保护全部头文件可能是个好主意。那样的话可以在稍候添加类型定义而不用冒可能因忘记保护文件而产生的风险。此外，在程序开发过程期间，避免相同头文件的不必要重复编译可以节约许多时间。

为了防止头文件多次包含，将用#ifndef和#endif两个指令来把文件的内容闭合起来。例如，可以用如下方式保护文件boolean.h：

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

#define TRUE 1
#define FALSE 0
typedef int Bool;

#endif
```

在首次包含这个文件时，将不定义宏BOOLEAN_H，所以预处理器允许保留在#ifndef和#endif之间的多行内容。但是如果再次包含此文件，那么预处理器将把#ifndef和#endif之间的多行内容删除。

宏（BOOLEAN_H）的名字不是真正的问题。但是，给它取类似于头文件名的名字是避免和其他的宏冲突的好方法。既然不能给宏命名为BOOLEAN.H（标识符不能含有句点），所以像BOOLEAN_H这样的名字是个很好的选择。

15.2.7 头文件中的#error 指令

经常把#error指令（➤14.5.1节）放置在头文件中是用来检查不应该包含头文件的条件。例如，假设头文件包含只能正常工作在DOS程序中的图形函数原型。为了保证只能在DOS程序中包含它，头文件可以包含#ifdef（或#if）指令用来检查宏，这个宏指示DOS是操作系统：

```
#ifndef DOS
#error Graphics supported only under DOS
#endif
```

如果非DOS的程序试图包含此头文件，那么编译将在#error指令处停止。 312

15.3 把程序划分成多个文件

现在应用我们知道的关于头文件和源文件的知识来开发一种把程序划分成多个文件的简单方法。这里将集中在函数上讨论，但是同样的规则也适用于外部变量。假设已经设计好程序，换句话说，已经决定程序需要什么函数以及如何把函数逻辑化地编排在相关的组中。（程序设计本身就是一个完整的主题，第19章将会讨论程序设计问题。）

下面是处理的方法。把每组函数集合放入单独的源文件中（比如用名字foo.c来表示一个这样的文件）。另外，创建和源文件同名的头文件，只是扩展名为.h（在此例中，头文件是foo.h）。还在foo.h文件中放置函数的原型，而函数的定义则是在foo.c中（在foo.h文件中不需要也不应该声明只为用于foo.c内部而设计的函数）。每个需要调用定义在foo.c文件中的函数的源文件都包含foo.h文件。而且，foo.c文件也包含foo.h文件，这是为了编译器可以检查foo.h文件中的函数原型是否与foo.c文件中的函数定义相一致。

main函数将出现在某个文件中，这个文件的名字与程序的名字相匹配。如果希望称程序为bar，那么main函数就应该在文件bar.c中。只要程序中的其他文件不调用其他函数，那么这

www.TopSage.com

些函数是可以和main函数在同一个文件中的。

程序：文本格式化

为了说明刚刚论述的方法，现在用它来做文本格式化的小程序。既然一些操作系统已经有了名为format的程序，那么就把要编写的程序命名为fmt。作为给fmt的输入样例，将采用文件quote，假设这个文件包含下列（不完全格式化的）引用语，这些引用语来自Dennis M. Ritchie写的“*The Development of the C language*”（*ACM SIGPLAN Notices*（March 1993）：207）：

```
    C    is quirky,   flawed,    and an
enormous success.       While accidents of    history
 surely helped,  it evidently     satisfied   a    need
       for a    system implementation     language     efficient
  enough    to displace          assembly    language,
     yet sufficiently    abstract    and fluent     to describe
    algorithms    and      interactions     in a    wide    variety
of    environments.
                          -- Dennis      M.          Ritchie
```

为了程序能在UNIX和DOS环境下运行，最好录入命令

```
fmt <quote
```

符号<提示操作系统程序fmt将用读入文件quote来代替键盘输入。把由UNIX、DOS和其他操作系统支持的这种特性称为是**输入重定向**（input redirection）（➤22.1.2节）。当把给定的文件quote作为输入时，程序fmt将产生下列输出：

```
C  is  quirky,   flawed,   and  an  enormous   success.   While
accidents of history surely helped,  it evidently satisfied a
need for a system implementation language  efficient   enough
to displace assembly language, yet sufficiently abstract and
fluent to describe algorithms and  interactions   in  a  wide
variety of environments.  -- Dennis M. Ritchie
```

程序fmt的输出通常将显示在屏幕上，但是可以利用**输出重定向**（output redirection）（➤22.1.2节）把结果保存在文件中：

313

```
fmt <quote >newquote
```

程序fmt的输出将放入到文件newquote中。

通常情况下，除了额外的空格和删除的空行，以及做过填充和调整的行，程序fmt的输出应该和输入一样。“填充”行意味着添加单词直到再多加一个单词就会导致行溢出时才停止。“调整”行意味着在单词间添加额外的空格以便于每行有精确的相同长度（60个字符）。必须进行调整，只有这样一行内单词间的空格才是相等的（或者几乎是相等的）。对输出的最后一行将不进行调整。

假设没有单词的长度超过20个字符。（把与单词相邻的标点符号看成是单词的一部分。）当然，这是一点点的限制，但是一旦编写和调试程序，就可以很容易的增加这种限制，事实上是从未超过这种限制的。如果程序遇到较长的单词，它需要忽略前20个字符后的所有字符，用一个单独的星号替换它们。例如，单词

```
antidisestablishmentarianism
```

将会显示成

```
antidisestablishment*
```

现在明白了程序应该完成的内容，应该考虑设计了。首先发现程序不能像读似的一个一个写单词，而必须把输入存储在一个“行缓冲区”中，直到有足够的空间填满一行。在进一步思考之后，我们决定程序的核心将是如下所示的循环：

www.TopSage.com

```
for (; ;) {
  读单词;
if (不能读单词) {
   不用调整地写行缓冲区的内容;
   终止程序;
}
if (单词不适合在行缓冲区中){
  调整地写行缓冲区的内容;
  清除行缓冲区;
}
往行缓冲区中添加单词;
}
```

314

因为我们需要函数处理单词，并且还需要函数处理行缓冲区，所以把程序划分为3个源文件。把所有和单词相关的函数放在一个文件中（word.c），而把所有和行缓冲区相关的函数放在另一个文件中（line.c）。第3个文件（fmt.c）将包含main函数。除了上述这些文件，还需要两个头文件word.h和line.h。头文件word.h将包含word.c文件中的函数原型，而头文件line.h将对line.c承担类似的工作。

通过检查主循环可以发现需要只和单词相关的函数是函数read_word。（如果read_word函数因为到了输入文件末尾而不读入单词，那么将通过假装读取“空”单词的方法给主循环发信号。）因此，文件word.h是一个短小的文件：

word.h

```
#ifndef WORD_H
#define WORD_H

/*************************************************************
 * read_word: Reads the next word from the input and         *
 *            stores it in word. Makes word empty if no      *
 *            word could be read because of end-of-file.     *
 *            Truncates the word if its length exceeds       *
 *            len.                                           *
 *************************************************************/
void read_word(char *word, int len);

#endif
```

注意宏WORD_H是如何保护多次包含的word.h文件的。虽然word.h文件不是真的需要保护，但是按照这种方式保护所有头文件是个很好的方法。

文件line.h将不会像word.h那样短小。主循环的轮廓显示了对执行下列操作的函数的需求：

- 不调整地写行缓冲区的内容。
- 检查单词是否适合在缓冲区中。
- 调整地写行缓冲区的内容。
- 清除行缓冲区。
- 往行缓冲区中添加单词。

我们将要调用下面这些函数：flush_line、space_remaining、write_line、clear_line和add_word。下面是头文件line.h。

315

line.h

```
#ifndef LINE_H
#define LINE_H

/*************************************************************
 * clear_line: Clears the current line.                      *
 *************************************************************/
```

www.TopSage.com

大家网 TopSage.com

```
void clear_line(void);

/*************************************************************
 * add_word: Adds word to the end of the current line.       *
 *           If this is not the first word on the line,      *
 *           puts one space before word.                     *
 *************************************************************/
void add_word(const char *word);

/*************************************************************
 * space_remaining: Returns the number of characters left    *
 *                  in the current line.                     *
 *************************************************************/
int space_remaining(void);

/*************************************************************
 * write_line: Writes the current line with                  *
 *             justification.                                *
 *************************************************************/
void write_line(void);

/*************************************************************
 * flush_line: Writes the current line without               *
 *             justification. If the line is empty, does     *
 *             nothing.                                      *
 *************************************************************/
void flush_line(void);

#endif
```

在编写文件word.c和文件line.c之前，可以用在头文件word.h和头文件line.h中声明的函数来编写主程序fmt.c。编写这个文件主要是把原始的循环设计翻译成C语言。

fmt.c

```
/* Formats a file of text */

#include <string.h>
#include "line.h"
#include "word.h"

#define MAX_WORD_LEN 20

main()
{
  char word[MAX_WORD_LEN+2];
316 int word_len;


  clear_line();
  for (;;) {
    read_word(word, MAX_WORD_LEN+1);
    word_len = strlen(word);
    if (word_len == 0) {
      flush_line();
      return 0;
    }
    if (word_len > MAX_WORD_LEN)
      word[MAX_WORD_LEN] = '*';
    if (word_len + 1 > space_remaining()) {
      write_line();
      clear_line();
    }
    add_word(word);
  }
}
```

www.TopSage.com

包含两个头文件line.h和word.h可以使编译器在编译fmt.c时访问到两个文件中的函数原型。

main函数利用一个技巧解决了单词超过20个字符的问题。在调用read_word函数时，main函数告诉read_word截短任何超过21个字符的单词。当read_word函数返回后，main函数检查word包含的字符串长度是否超过20个字符。如果超过了，那么读入的单词必须至少是21个字符长（在截短前），所以main函数会用星号来替换第21个字符。

现在开始编写word.c程序。虽然头文件word.h只有唯一一个read_word函数的原型，但是可以在word.c中放置额外需要的函数。在运行时，如果添加一个小的"帮助"函数read_char，这样可以比较容易地编写函数read_word。read_char函数的工作就是读单独一个字符，并且把遇到的换行符或制表符转换为空格。用read_char函数代替getchar函数进行调用，read_word函数将自动把换行符和制表符作为空格来处理。

下面是文件word.c：

word.c

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

int read_char(void)
{
  int ch = getchar();

  if (ch == '\n' || ch == '\t')
    return ' ';

  return ch;
}

void read_word(char *word, int len)
{
  int ch, pos = 0;
  while ((ch = read_char()) == ' ')
    ;

  while (ch != ' ' && ch != EOF) {
    if (pos < len)
      word[pos++] = ch;
    ch = read_char();
  }

  word[pos] = '\0';
}
```

317

在讨论read_word函数之前，有两个正好关于getchar函数的注释。第一，getchar函数实际上返回的是int型值而不是char型值，这是因为在read_char函数中把变量ch声明为（➤22.4节）int类型。第二，当不能连续读入时（通常因为读到了输入文件的末尾），getchar返回值EOF。

read_word函数由两个循环构成。第一个循环跳过空格，在遇到第一个非空字符时停止。（EOF不是空的，所以如果到达输入文件的末尾，循环停止。）第二个循环读字符直到遇到空格或EOF时停止。循环体把字符存储到word中直到达到len的限制时停止。在这之后，循环继续读入字符，但是不再存储这些字符。read_word函数中的最后的语句以空字符结束单词，从而构成字符串。如果read_word在找到非空字符前遇到EOF，pos将在末尾置为0，从而使得word为空字符串。

唯一剩下的文件是line.c。这个文件提供在文件line.h中声明的函数的定义。line.c文件也将需要变量来跟踪行缓冲区的状态。一个变量line将存储当前行的字符。严格地讲，line

www.TopSage.com

是我们需要的唯一变量。然而，出于对速度和便利的考虑，将用到另外两个变量：line_len（当前行的字符数量）和num_words（当前行的单词数量）。

下面是文件line.c：

line.c

```
#include <stdio.h>
#include <string.h>
#include "line.h"

#define MAX_LINE_LEN 60

char line[MAX_LINE_LEN+1];
int line_len = 0;
int num_words = 0;

void clear_line(void)
{
  line[0] = '\0';
  line_len = 0;
  num_words = 0;
}

void add_word(const char *word)
{
  if (num_words > 0) {
    line[line_len] = ' ';
    line[line_len+1] = '\0';
    line_len++;
  }
  strcat(line, word);
  line_len += strlen(word);
  num_words++;
}

int space_remaining(void)
{
  return MAX_LINE_LEN - line_len;
}

void write_line(void)
{
  int extra_spaces, spaces_to_insert, i, j;

  extra_spaces = MAX_LINE_LEN - line_len;
  for (i = 0; i < line_len; i++) {
    if (line[i] != ' ')
      putchar(line[i]);
    else {
      spaces_to_insert = extra_spaces / (num_words - 1);
      for (j = 1; j <= spaces_to_insert + 1; j++)
        putchar(' ');
      extra_spaces -= spaces_to_insert;
      num_words--;
    }
  }
  putchar('\n');
}

void flush_line(void)
{
  if (line_len > 0)
    puts(line);
}
```

318

www.TopSage.com

TopSage.com

文件line.c中大多数函数都很容易编写。唯一需要技巧的函数是write_line。这个函数用来调整地写一行内容。函数write_line在line中一个一个地写字符，如果需要添加额外的空格那么就在每对单词之间停顿。额外空格的数量存储在变量spaces_to_insert中，这个变量的值由extra_spaces / (num_words -1)获得，其中extra_spaces初始是最大行长度和当前行长度的差。因为在打印每个单词之后extra_spaces和num_words都发生变化，所以spcaes_to_insert也将变化。如果extra_spaces初始为10，并且num_words初始为5，那么将有2个额外的空格跟着第1个单词，有2个额外空格跟着第2个单词，有3个额外的空格跟着第3个单词，以及有3个额外的空格跟着第4个单词。 319

15.4 构建多文件程序

在2.1节中，我们验证了编译和链接程序的过程适用于单独一个文件。现在将把这种讨论扩展到由多个文件构成的程序中。构建大规模程序需要和构建小程序相同的基本步骤：

- **编译**。必须对程序中的每个源文件单独进行编译。（不需要编译头文件。当包含头文件的源文件编译时会自动编译头文件。）编译器产生一个文件，此文件包含来自每个源文件的目标代码。这些被称为**目标文件**（object file）的文件在UNIX系统中的扩展名为.o，而在DOS系统中的扩展名为.obj。
- **链接**。链接器把上一步产生的目标文件和库函数的代码结合在一起生成可执行的程序。在其他责任中，链接器还有责任要解决编译器遗留的外部参考问题。（外部参考发生在一个文件中的函数调用另一个文件中定义的函数时，或者访问另一个文件中定义的变量时。）

大多数编译器允许用单独一步来构建程序。例如，对于UNIX的cc编译器来说，最好使用下列命令行来构建15.3节的fmt程序：

```
% cc -o fmt fmt.c line.c word.c
```

（字符%是UNIX提示符。）首先把三个源文件编译成目标代码，并且分别用名为fmt.o、line.o和word.c的文件来存储这些代码。然后，会自动把目标文件传递给链接器，链接器会把这些文件结合成一个单独的文件。选项-o是告诉编译器需要的可执行文件的名字是fmt。

15.4.1 makefile

把所有源文件的名字放在命令行中的方法很快变得枯燥乏味。更糟糕的是，如果重新编译了所有源文件，不仅仅是最近变化的源文件会受到影响，而且重新构建程序可能会浪费大量的时间。

为了更易于构建大规模的程序，UNIX系统发明了makefile的概念，这个文件包含构建程序的必要信息。makefile不仅列出了作为程序部分的文件，而且还描述了文件之间的**依赖性**。假设文件foo.c包含文件bar.h，那么就说foo.c“依赖于”bra.h，因为bar.h的变化将会需要重新编译foo.c。

下面是针对程序fmt而设的UNIX系统的makefile： 320

```
fmt: fmt.o word.o line.o
        cc -o fmt fmt.o word.o line.o

fmt.o: fmt.c word.h line.h
        cc -c fmt. c

word.o: word.c word.h
        cc -c word.c

line.o: line.c line.h
        cc -c line.c
```

www.TopSage.com

这里有4组行。每组的第一行给出了目标文件，跟在后边的是它所依赖的文件。如果因为其中一个所依赖的文件的变化而使目标必须要重新构建，那么每组的第二行是用来执行的命令。首先来看前两组，而后两组相似。

在第一组中，fmt（可执行程序）是目标文件：

```
fmt: fmt.o word.o line.o
        cc -o fmt fmt.o word.o line.o
```

第一行说明fmt程序依赖于fmt.o文件、word.o文件和line.o文件。因为程序是最后构建，那么若3个文件中的任何一个发生改变，则都需要重新构建fmt程序。紧跟其后的一行命令说明是如何进行构建的（通过使用cc来链接3个目标文件）。

在第二组中，fmt.o是目标文件：

```
fmt.o: fmt.c word.h line.h
        cc -c fmt. c
```

第一行说明，如果文件fmt.c、word.h文件或line.h文件发生改变，那么fmt.o需要重新构建。（理由是提及fmt.c包含的word.h和line.h这两个文件，所以任意一个文件的改变都可能会对fmt.c产生影响。）下一行信息说明如何更新fmt.o（通过重新编译fmt.c）。选项-c通知编译器去编译fmt.c，但是不要试图链接它，因为它不是一个完整的程序。

用于其他操作系统的makefile是很类似的，但不是完全一样的。例如，如果使用Borland的bcc编译器，将使用略有不同的makefile：

```
fmt.exe: fmt.obj word.obj line.obj
        bcc fmt.obj word.obj line.obj
fmt.obj: fmt.c word.h line.h
        bcc -c fmt.c
word.obj: word.c word.h
        bcc -c word.c
line.obj: line.c line.h
        bcc -c line.c
```

321

编译器用bcc代替了cc，目标文件也由扩展名.obj代替了.o，同时可执行文件也由fmt.exe代替了fmt。而且，不再需要选项-o，因为第一个目标文件的名字fmt.obj就确定了可执行文件的名字。

一旦为程序创造了makefile，就能使用make工具来构建（或重新构建）程序了。通过检查与程序中每个文件相关的时间和日期，make可以确定哪个文件是过期的。然后，它因为需要重新构建程序而自动唤醒编译器和链接器。

make是如此复杂以致足够用一本书[①]来介绍，所以这里不会试图深入研究它的复杂性。这里只是说明真正的makefiles通常不是像例子显示的那样容易理解。这里有几种方法可以减少makefile中的冗余，并且使得它们更容易修改。但是，同时这些技术也极大地减少了它们的可读性。

顺便说一句，不是每个人都用makefile的。其他程序维护工具正在变得流行，包括一些集成开发环境支持的“工程文件”。检查所使用系统的说明文档，看看它是否支持makefile或工程文件，还是二者都支持。

15.4.2　链接期间的错误

一些在编译期间无法发现的错误将会在链接期间被发现。事实上，如果程序中丢失了函数定义或变量定义，那么链接器将无法解决外部引用，从而导致出现类似“Undefined symbol”或“Unresolved external reference”的信息。

① Tondo, Nathanson, and yount, *Mastering MAKE*, second Edition, Prentice-Hall 1994。

www.TopSage.com

通常很容易修改链接器检查到的错误。下面是一些最常见的错误起因：

- **拼写错误**。如果变量名或函数名拼写错误，那么链接器将作为丢失来进行报告。例如，在程序中定义了函数read_char，但是却把它写为read_cahr，那么链接器将报告说丢失了read_char函数。
- **丢失文件**。如果链接器不能找到文件foo.c中的函数，那么它可能不会知道此文件。这时就要检查makefile或工程文件来确保foo.c文件是列出了的。
- **丢失库**。链接器不可能找到程序中用到的全部库函数。发生在UNIX系统中的经典例子，在链接期间无法搜索到数学函数的地方，直到出现了选项-lm才办到。检查所使用的系统文档，看看可以用于链接器的选项有哪些。

15.4.3 重新构建程序

在开发程序期间，极少需要编译全部文件。大多数时候，将测试程序，发生变化，然后再次构建程序。为了节约时间，重新构建的过程应该只对那些可能受到最后一次变化影响的文件进行重新编译。 322

假设按照15.3节的框架方法设计了程序，对每一个源文件都使用了头文件。为了发现在变化后需要重新编译的文件的数量，我们需要考虑两种可能性。

第一个可能性是变化影响单独一个源文件。这种情况下，只有此文件需要重新编译。(当然，在此之后整个程序将需要重新链接。)思考程序fmt。假设决定精简word.c中的函数read_char(修改过的地方用粗体标注)：

```
int read_char (void)
{
  int ch = getchar();
  return (ch == '\n'  || ch == '\t' ? ' '  : ch);
}
```

这种改变没有影响read_char的调用方式，所以不需要修改word.h。在改变以后，只需要重新编译word.c并且重新链接程序就行了。

第二个可能性是变化影响头文件。这种情况下，应该重新编译包含此头文件的所有文件，因为它们可能潜在地受到这种变化的影响。(这些文件中的一部分可能受到影响，但是采取保守的方法。)

作为示例，思考一下程序fmt中的函数read_word。注意，为了确定刚读入的单词的长度，main函数在调用read_word函数后立刻调用strlen。因为read_word函数已经知道了单词的长度(read_word函数的变量pos负责跟踪长度)，所以使用strlen就显得有些没必要了。修改read_word函数来返回单词的长度是很容易的。首先，改变word.h文件中的read_word函数的原型：

```
/*************************************************************
 * read_word: Reads the next word from the input and         *
 *            stores it in word. Makes word empty if no      *
 *            word could be read because of end-of-file.     *
 *            Truncates the word if its length exceeds       *
 *            len. Returns the number of characters          *
 *            stored.                                        *
 *************************************************************/
int read_word(char *word, int len);
```

当然，一定要记住修改附属于read_word函数的注释。接下来，修改word.c文件中的read_word函数的定义：

```
int read_word(char *word, int len)
{
```

www.TopSage.com

大家网 TopSage.com

```
    int ch, pos = 0;

    while ((ch = read_char()) == ' ')
      ;
    while (ch != ' ' && ch != EOF)  {
      if (pos < len)
        word[pos++] = ch;
      ch = read_char();
  }

  word[pos] = '\0';
  return pos;
  }
```

323

最后，再来修改fmt.c，方法是删除对<string.h>的包含，以及按如下方式修改main函数：

```
main()
{
  char word[MAX_WORD_LEN+2];
  int word_len;

  clear_line ();
  for (;;) {
    word_fen = read_word (word, MAX_WORD_LEN+1);
    if (word_len == 0)  {
      flush_line ();
      return 0;
    }
    if (word_len > MAX_WORD_LEN)
      word[MAX WORD LEN] = '*';
    if (word_len + 1 > space_remaining())  {
      write_line();
      clear_line();
    }
    add_word(word);
  }
}
```

一旦做了上述这些修改，将需要重新构建程序fmt，方法是要重新编译word.c和fmt.c，然后再重新进行链接。不需要重新编译line.c，因为它不包含word.h，所以也就不会受到word.h改变的影响。在UNIX系统中，可以使用下列命令来重新构建程序：

```
% cc -o fmt fmt.c word.c line.o
```

注意，这里用line.o代替line.c。

使用makefile的好处之一就是可以自动进行重新构建。通过检查每个文件的日期，make实用程序可以确定从程序最后一次构建后哪些文件发生了改变。然后，它会把那些改变的文件和直接或间接依赖于它们的全部文件一起进行重新编译。

15.4.4　在程序外定义宏

在编译程序时，通常C语言编译器提供一些指定宏的值的方法。这种能力使我们不需要编辑任何程序文件就对宏的值进行改变变得非常容易。当利用makefile自动构建程序时这种能力尤其有价值。

324

大多数UNIX编译器（和某些非UNIX编译器）支持选项-D，此选项允许宏的值用命令行来指定：

```
% cc -DDEBUG=1 foo.c
```

在这个例子中，定义宏DEBUG在程序foo.c中的值为1，就如同在foo.c的开始处出现的下面这行信息：

```
#define DEBUG 1
```

www.TopSage.com

如果选项-D命名的宏是没有指定的值，那么这个值被设为1。

一些编译器也支持选项-U，这个选项“未定义”宏，就如同使用了#undef一样：

```
% cc -UDEBUG foo.c
```

问与答

问：这里没有任何例子是使用#include指令来包含源文件。如果这样做了会发生什么？

答：这不是个好方法，但是它是合法的。这里有一个这类问题的例子可以拿来讨论。假设foo.c定义的函数f在bar.c和baz.c中会需要，所以把下列指令放在bar.c和baz.c中：

```
#include "foo.c"
```

这些文件都会很好地被编译。稍后，当链接器发现函数f的两个目标代码的副本时，问题就出现了。当然，如果只是bar.c包含此函数，而baz.c没有，那么将会从包含的foo.c中拿走。为了避免出现问题，最好只在头文件中使用#include，而不要在源文件中使用。

问：针对#include指令的精确搜索规则是什么？（p.212）

答：这与所使用的编译器有关。C标准在#include的表述中故意模糊不清。如果文件名用尖括号括起来，那么预处理器把它看成是“实现定义的地方的序列”，作为倾斜的标准来放置。如果文件名用双引号引起来，那么文件就是“以实现定义的方式搜索”，而且如果没有发现，那么搜索就好像它的名字使用尖括号括起来的。原因很简单：不像DOS和UNIX系统，不是所有操作系统都有层次（像树型的）文件系统。

为了使这事更加有趣，标准根本不要求括在尖括号内的名字是文件名字，留下开放的可能，这种可能是#include指令在编译器中利用<>来完全进行操作。

325

问：我不理解为什么每个源文件都需要它自己的头文件。为什么没有一个大的头文件包含宏定义、类型定义和函数原型呢？通过包含这个文件，每个源文件都可以访问全部需要共享的信息。（p.214）

答：“一个大的头文件”的方法的确可以工作，许多程序员使用这种方法。而且，这种方法有一个好处：因为只有一个头文件，所以要管理的文件较少。然而，对于大规模的程序来说，这种方法的坏处大于它的好处。

使用单独一个头文件为稍后人们读程序提供无用的信息。通过多个头文件，读者可以迅速看到通过特殊源文件使用的程序的其他部分。

但是也不绝对。既然每个源文件都依赖于一个大的头文件，所以改变它会导致要对全部源文件重新编译，这是大规模程序中的一个显著缺陷。更糟的情况是，由于包含了大量信息，所以头文件可能会频繁地改变。

问：本章说到共享数组应该按照下列方式声明：

```
extern int a[];
```

既然数组和指针关系密切，那么用下列写法代替是否合法呢？（p.215）

```
extern int *a;
```

答：不合法。在用于表达式时，数组“衰退”成指针。（当数组名用作函数调用中的实际参数时我们已经注意到这种行为。）在变量声明中，数组和指针是截然不同的两种类型。

问：如果源文件包含了不是真正需要的头文件，会有损害吗？

答：不会，除非头文件的声明或定义与源文件中的冲突。否则，可能发生的最坏情况就是在编译源文件时镜像增加。

问：我需要调用文件foo.c中的函数，所以包含了匹配的头文件foo.h。但是程序编译后并不链接。为什么？

答：在C语言中编译和链接是完全独立的。头文件存在是为了给编译器而不是为链接器提供信息。如果希望调用文件foo.c中的函数，那么需要确保对foo.c进行了编译，还要确保为了找到函数使链接器意识到必须搜索到foo.c的目标文件。通常情况下，这就意味着在程序的makefile或工程文件中命

www.TopSage.com

名foo.c。

问：如果程序调用<stdio.h>中的函数，这是否意味着在<stdio.h>中的所有函数都将和程序链接吗？

答：不是的。包含<stdio.h>（或者任何其他头文件）对链接没有任何影响。在任何情况下，大多数链
326 接器都只会链接程序实际需要的函数。

练习

15.1节

1. 15.1节列出了把程序分割成多个源程序的几个优点。
   (a) 请描述几个其他的优点。
   (b) 请描述一些缺点。

15.2节

2. 下列哪个不应该放置在头文件中？为什么？
   (a) 函数原型。
   (b) 函数定义。
   (c) 宏定义。
   (d) 类型定义。
3. 如果文件是我们已经编写好的，那么已经看到用#include <文件>代替#include "文件"可能无法工作。如果文件是系统头文件，那么用#include "文件"代替#include <文件>是否有什么问题？
4. 假设文件foo.c定义了外部变量i如下：

   ```
   int i;
   ```

   而且文件bar.c以下列方式声明此变量：

   ```
   extern long int i;
   ```

   (a) 假设sizeof(int)和sizeof(long int)是完全一样的。如果文件bar.c中的一个函数给i赋值为0，请解释会发生什么？
   (b) 假设sizeof(int)小于sizeof(long int)。请重复(1)的问题。

15.3节

5. 程序fmt通过在单词间插入额外的空格来调整行。当前编写的函数writen_line的方法是，与开始处的单词间隔相比，靠近行末尾单词的间隔略微宽一些。（例如，靠近末尾的单词彼此之间可能有3个空格，而靠近开始的单词彼此之间可能只有2个空格。）请通过替换函数write_line来改进此程序，替换后的函数可以在靠近行末尾处的单词之间放置较大的空隙，而在行开始处的单词之间放置一般空隙。
6. 利用15.2节中的设计，编写实现逆波兰表达式计算器的程序。使计算器可以实现双目运算+、-、*和/。假设它们的含义和在C语言中一样。

15.4节

7. 假设程序有3个源文件构成：main.c、f1.c和f2.c，此外还包括两个头文件f1.h和f2.h。全部3个源文件都包含f1.h，但是只有f1.c和f2.c包含f2.h。为此程序编写UNIX makefile。假设需要可执行文件名为demo。
8. 下面的问题引用了练习7描述的程序。
   (a) 当程序第一次构建时，需要对哪些文件进行编译？
   (b) 如果在程序构建后对f1.c进行了修改，那么需要对哪个（些）文件进行重新编译？
327 (c) 如果在程序构建后对f1.h进行修改，那么需要对哪个（些）文件进行重新编译？
   (d) 如果在程序构建后对f2.h进行了修改，那么需要对哪个（些）文件进行重新编译？
9. (a) 修改程序fmt，使函数read_word（代替main函数）在被截短的单词的末尾存储*字符。
328 (b) 如果按照(a)进行了修改，那么需要对哪个（些）文件进行重新编译？

www.TopSage.com

大家网
TopSage.com

第16章 结构、联合和枚举

函数延迟绑定：数据结构导致绑定。
记住：在编程过程后期再结构化数据。

本章介绍3种新的类型：结构、联合和枚举。**结构**（structure）是可能具有不同类型的值（**成员**（member））的集合。**联合**（union）和结构很类似，不同之处在于联合的成员共享同一存储空间。这样的结果是，联合可以每次存储一个成员，但是无法同时存储全部成员。**枚举**（enumeration）是一种整数类型，它的值由程序员来命名。

在这3种类型中，结构是到目前为止最重要的一种类型，所以本章的大部分内容都是关于结构的。16.1节说明了如何声明结构变量，以及如何对其进行基本操作。随后，16.2节解释了定义结构类型的方法，借助结构类型，我们就可以编写函数，接受结构类型的参数或返回结构类型的值。16.3节探讨如何实现数组和结构的嵌套。本章的最后两节分别讨论了联合（16.4节）和枚举（16.5节）。

16.1 结构变量

到目前为止唯一介绍的数据结构就是数组。数组有两个重要特性：首先，数组的所有元素具有相同的类型；其次，为了选择数组元素需要指明元素的位置（作为整数下标）。

结构所具有的特性与数组很不相同。结构的元素（在C语言中的说法是**成员**）可能具有不同的类型。而且，每个结构成员都有名字，所以为了选择特殊的结构成员需要指明结构成员的名字而不是它的位置。

由于大多数编程语言都提供类似的特性，所以结构可能听起来很熟悉。在其他语言中，经常把结构称为**记录**（record），把结构的成员称为**字段**（field）。

16.1.1 结构变量的声明

当需要存储相关数据项的集合时，结构是一种合理的选择。例如，假设需要记录存储在仓库中的零件。用来存储每种零件的信息可能包括零件的编号（整数）、零件的名称（字符串）以及现有零件的数量（整数）。为了产生一个可以存储全部3种数据项的变量，可以使用类似下面这样的声明：

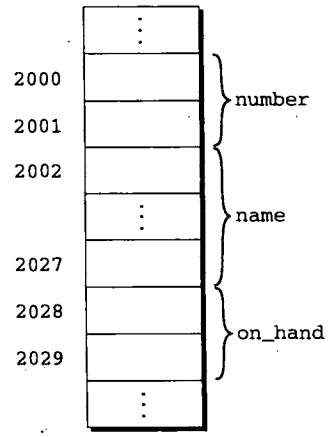
```
struct {
  int number;
  char name [NAME_LEN+1 ] ;
  int on_hand;
} part1, part2 ;
```

每个结构变量都有3个成员：number（零件的编号）、name（零件的名称）和on_hand（现有数量）。注意，这里的声明格式和C语言中其他变量的声明格式一样。struct{...}指明了类型，而part1和part2则是具有这种类型的变量。

结构的成员在内存中是按照声明的顺序存储的。为了说明part1在内存中存储的形式，现在假设：（1）part1存储在地址为2000的内存单元中，（2）每个整数在内存中占两个字节，（3）

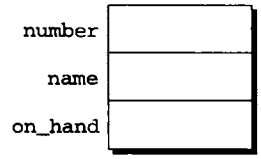
www.TopSage.com

NAME_LEN的值为25，(4)成员之间没有间隙。根据这些假设，part1在内存中的样子如下图所示：

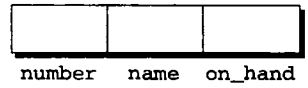


通常情况下不需要画出如此详细的结构。这里一般更加抽象地显示结构，用一系列的方框表示：

330



有时还可以用水平方向的盒子来代替垂直方向的方框：



结构成员的值稍后将放入盒子中。但是现在，这里保留为空。

每个结构表示一种新的范围。任何声明在此范围内的名字都不会和程序中的其他名字冲突。(C语言的术语表示每个结构都为它的成员设置了单独的**名字空间**（name space)。）例如，下列声明可能出现在同一程序中：

```
struct {
  int number;
  char name [NAME_LEN+1];
  int on_hand;
} part1, part2;

struct {
  char name[NAME_LEN+1];
  int number;
  char sex;
} employee1, employee2;
```

结构part1和part2中的成员number和成员name不会和结构employee1和employeet2中的成员number和成员name冲突。

16.1.2　结构变量的初始化

和数组一样，结构变量也可以在声明的同时进行初始化。为了初始化结构，要准备存储在结构中的值列表并用大括号把它括起来：

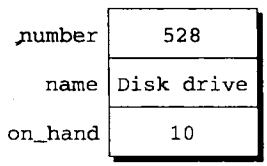
```
struct  {
  int number;
  char name[NAME_LEN+1];
  int on, hand;
```

www.TopSage.com

```
} part1 = { 528, "Disk drive", 10 },
  part2 = { 914, "Printer cable", 5};
```

初始化式中的值必须按照结构成员的顺序进行显示。在此例中，结构part1的成员number值为528，成员name则是"Disk driver"，以此类推。下面是结构part1初始化后的样子：

331



结构初始化式遵循的原则类似于数组初始化式的原则。用于结构初始化式的表达式必须是常量。例如，不能用变量来初始化结构part1的成员on_hand。初始化式可以短于它所初始化的结构。就像对数组那样，任何“剩余的”成员都用0作为它的初始值。

16.1.3 对结构的操作

既然数组最常见的操作是下标操作，也就是通过位置选择数组元素的操作，那么结构最常用的操作是选择成员也就无需惊讶了。但是，结构成员的访问是通过名字而不是通过位置。

为了访问结构内的成员，首先写出结构的名字，然后写出成员的名字。例如，下列语句将显示结构part1的成员的值：

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1 on_hand);
```

结构成员的值是左值（➤4.2节），所以它们可以出现在赋值运算的左侧，或者是作为自增或自减表达式的操作数：

```
part1.number = 258;       /* changes part1's part number */
part1.on_hand++;    /* increments part1's quantity on hand */
```

用于访问结构成员的句点实际上就是一个C语言的运算符。参考附录B的运算符表可知，它的运算优先级和后缀++和后缀--运算符一样，所以句点运算符的优先级几乎高于所有其他运算符。考虑下面的例子：

```
scanf("%d", &part1.on_hand);
```

表达式&part1.on_hand包含两个运算符（即&和.）。.运算符优先级高于&运算符，所以就像希望的那样，&计算的是part1.on_hand的地址。

另一种主要的结构操作是赋值运算：

```
part2 = part1;
```

现在part2.number包含了和part1.number一样的值，part2.name也将包含和part1.name一样的值，以此类推。

因为数组不能用=运算符进行复制，所以结构可以用=运算符复制应该是一个惊喜。当复制闭合的结构时，考虑把嵌在结构内的数组进行复制甚至会带来更大的惊喜。一些程序员利用这种性质来产生“空”结构，以封装稍候将进行复制的数组：

332

```
struct { int a [10]; } a1, a2 ;

a1 = a2;     /* legal, since a1 and a2 are structures */
```

运算符=仅仅用于类型一致的结构。两个同时声明的结构（比如part1和part2）是一致的。正如下一节将会看到的那样，使用同样的“结构标记”或同样的类型名声明的结构也是一致的。

除了赋值运算，C语言没有提供其他用于整个结构的操作。Q&A特别是不能使用运算符==和!=来判定两个结构是否相等或不等。

www.TopSage.com

大家网 TopSage.com

16.2　结构类型

虽然16.1节说明了声明结构变量的方法，但是它没有讨论一个重要的问题：命名结构类型。假设程序需要声明几个具有相同成员的结构变量。如果一次可以声明全部变量，那么没有什么问题。但是如果需要在程序中的不同位置声明变量，那么问题就复杂了。如果在某处编写了

```
struct {
  int number;
  char name [NAME_LEN+1];
  int on_hand;
} part1;
```

并且在另一处编写了

```
struct {
  int number;
  char name [NAME LEN+1];
  int on_hand;
} part2;
```

那么立刻就会出现问题。重复的结构信息会使程序膨胀。既然不容易确保声明会保持一致，那么稍后改变程序会很危险。

但是这些还都不是最大的问题。根据C语言的规则，part1和part2不具有一致的类型。这样的结果是不能把part1赋值给part2，反之亦然。而且，因为part1或part2的类型没有名字，所以也就不能用它们作为函数调用的参数了。

333　为了克服这些困难，需要为表示结构的类型定义名字，而不是为特定结构的变量命名。正如产生的那样，Q&A C语言提供了两种命名结构的方法：既可以声明“结构标记”，也可以使用typedef来定义类型名（类型定义（➤7.6节））。

16.2.1　结构标记的声明

结构标记（structure tag）是用于标识某种特定结构类型的名字。下面的例子声明了名为part的结构标记：

```
struct part {
  int number;
  char name [NAME_LEN+1];
  int on_hand;
};
```

注意，右大括号后的分号，必须用它来表示声明的结束。

如果无意间忽略了结构声明结尾处的分号，可能会导致意料之外的结果。考虑下面的例子：

```
struct part {
  int number;
  char name [NAME_LEN+1] ;
  int on_hand;
}               /*** WRONG--semicolon missing *** /

f(void)
{
  ...
  return 0;    /* error detected at this line */
}
```

因为丢失了函数f的返回值类型，所以通常将默认为int类型。然而，在此例中，由于前面的结构声明没有正常终止，所以编译器会假设struct part的返回类型。编译器直到执行函数中第一条return语句时才会发现错误。结果是：含义模糊的出错信息。

www.TopSage.com

一旦产生了标记part，就可以用它来声明变量了：

```
struct part part1, part2;
```

可惜的是，不能通过漏掉单词struct来缩写这个声明：

```
part part1, part2;    /*** WRONG ***/
```

part不是类型名。如果没有单词struct的话，它没有任何意义。

因为结构标记只有在前面放置了单词struct才会有意义，所以它们不会和程序中用到的其他名字发生冲突。程序拥有名为part的变量是非常合法的（虽然有点混淆）。 334

顺便说一句，结构标记的声明可以和结构变量的声明合并在一起：

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

在这里不仅声明了结构标记part（可能稍后会用part声明更多的变量），而且声明了变量part1和part2。

所有声明为struct part类型的结构彼此之间是一致的：

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;
part2 = part1;    /* legal; both parts have the same type */
```

16.2.2 结构类型的定义

作为声明结构标记的替换，还可以用typedef来定义真实的类型名。例如，可以按照如下方式定义名为part的类型：

```
typedef struct  {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

注意，类型Part的名字必须出现在定义的末尾，而不是在单词struct的后边。

可以像内置类型那样使用Part。例如，可以用它声明变量：

```
Part part1, part2;
```

因为类型Part是typedef的名字，所以不允许书写strcut Part。无论在哪里声明，所有的Part类型的变量都是一致的。

当需要命名结构时，Q&A通常既可以选择声明结构标记也可以使用typedef。但是，正如稍后将看到的，结构用于链表（➤17.5节）时，必须强制声明结构标记。

16.2.3 结构类型的实际参数和返回值

函数可以有结构类型的实际参数和返回值。下面来看两个例子。当把结构part用作实际参数时，第一个函数显示出结构的成员： 335

```
void print_part (struct part p)
{
  printf("part number: %d\n", p.number);
  printf("Part name: %s\n", p.name),
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

下面是print_part可能的调用方法：

```
print_part(part1);
```

www.TopSage.com

第二个函数返回结构part，因为此结构构成了函数的实际参数：

```
struct part build_part(int number, const char* name,
                       int on_hand)
{
  struct part p;
  p. number = number;
  strcpy (p.name, name);
  p. on_hand = on_hand;
  return p;
}
```

注意，函数build_part的形式参数名和结构part的成员名相同是合法的，因为结构拥有自己的名字空间。下面是build_part可能的调用方法：

```
part1 = build_part(528, "Disk driver", 10);
```

给函数传递结构和从函数返回结构都要求使用结构中所有成员的副本。这样的结果是，这些操作对程序强加了一定数量的系统开销，特别是结构很大的时候。为了避免这类系统开销，有时用传递指向结构的指针来代替传递结构本身是很明智的做法。类似地，可以使函数返回指向结构的指针来代替返回实际的结构。

有时，可能希望在函数内初始化结构变量来匹配其他结构，就可能应用结构变量作为函数的形式参数。在下面的例子中，part2的初始化式是传递给函数f的形式参数：

```
void f(struct part part1)
{
  struct part part2 = part1;
  ...
}
```

C语言允许这类初始化式，因为初始化的结构（此例中的part2）具有自动的存储期限（对函数而言它是局部的，而且没有声明为static）。初始化式可以是适当类型的任意表达式，包括返回结构的函数调用。

336

16.3　数组和结构的嵌套

结构和数组的组合没有限制。数组可以有结构作为元素，同时结构可以包含数组和结构作为成员。已经看过数组嵌套在结构内部的示例（结构part的成员name）。下面再来探讨其他的可能性：成员是结构的结构和元素是结构的数组。

16.3.1　嵌套的结构

一种结构嵌套在另一种结构中经常是非常有用的。例如，假设声明了如下的结构，此结构用来存储一个人的教名和姓：

```
struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NANE_LEN+1];
};
```

可以用结构person_name作为更大结构的一部分内容：

```
struct student {
  struct person_name name;
  int_id, age;
  char sex;
} student1, student2;
```

访问student1的名或姓要求两次应用运算符。

www.TopSage.com

TopSage.com

```
strcpy(student1.name.first, "Fred");
```

使name成为结构（替换了结构student中的成员first、middle_initial和last）的好处之一就是可以更容易地把名字作为数据单元来处理。例如，如果打算编写函数来显示名字，可以把结构person_name只作为一个实际参数代替三个实际参数来进行传递：

```
display_name(student1.name);
```

同样地，把结构person_name的信息复制给结构student的成员name将只需要一次而不是三次赋值操作：

```
struct person_name new_name;
...
student1.name = new_name;
```

337

16.3.2 结构数组

数组和结构最常见的组合之一就是具有结构元素的数组。这类数组可以用作简单的数据库。例如，下列具有结构part的数组能够存储100种零件的信息：

```
strcut part inventory[100];
```

为了访问数组中的某种零件，可以使用下标方式。例如，为了显示存储在位置i的零件，可以写成

```
print_part(inventory[i]);
```

访问结构part内的成员要求把下标和成员选择组合使用。为了给inventory[i]的成员number赋值为883，可以写成

```
inventory[i].number = 883;
```

访问零件名中的单独字符要求用下标方式（选择特定的零件），跟着是成员选择（选择成员name），再跟着是下标（选择零件名中的字符）。为了使存储在inventory[i]的名字变为空字符串，可以写成

```
inventory[i].name[0] = '\0';
```

16.3.3 结构数组的初始化

初始化结构数组的工作和初始化多维数组的方法非常相似。每个结构都拥有自己的大括号括起来的初始符，数组的初始符会在结构初始符的外围简单括上另一对大括号。

初始化结构数组的原因之一就是计划把它作为程序执行期间不改变的信息数据库。例如，假设用到的程序在打国际长途电话时会需要访问国家（地区）代码。首先，将设置结构用来存储国家（地区）名和相应代码：

```
struct dialing_code {
  char *country;
  int code;
};
```

注意，country是个指针而不是字符数组。如果计划用结构dialing_code作为变量可能有问题，但是这里不这样做。当初始化结构dialing_code时，country会结束指向字符串字面量。

接下来，声明这类结构的数组并且初始化它，从而使此数组包含一些世界上人口最多的国家的代码：

338

```
const struct dialing_code country_codes[] =
  { {"Argentina",          54},   { "Bangladesh",    880},
    {"Brazi 1",            55},   { "China",          86},
    {"Colombia",           57},   { "Egypt",          20},
    {"Ethiopia",          251},   { "France",         33},
    {"Germany",            49},   { "India ",         91},
```

www.TopSage.com

```
    {" Indonesia"          62},   { "Iran",          98},
    {"Italy",              39},   { "Japan",         81},
    {"Korea, Republic of" 82},   { "Mexico",        52},
    {"Nigeria",           234},   { "Pakistan",      92},
    {"Philippines",        63},   { "Poland",        48},
    {"Russia",              7},   { "South Africa",  27},
    {"Spain",              34},   { "Thailand",      66},
    {"Turkey",             90},   { "Ukraine",        7},
    {"United Kingdom",     44},   { "Vietnam",       84},
    {"Zaire",             243}};
```

每个结构值周围的内层大括号是可选项。然而，作为书写风格最好不要忽略它们。

16.3.4 程序：维护零件数据库

为了说明实际应用中数组和结构是如何嵌套的，现在将开发一个相对较大的程序，此程序用来维护仓库存储的零件的信息数据库。程序围绕一个结构数组建立，且每个结构包含以下信息：零件的编号、零件的名称以及某种零件的数量。程序将支持下列操作：

- **添加新零件编号、名称和现有的初始数量。**如果零件已经在数据库中，或者数据库已满，那么程序必须显示出错信息。
- **给定零件编号，显示出零件的名称和当前现有的数量。**如果零件编号不在数据库中，那么程序必须显示出错信息。
- **给定零件编号，改变现有的零件数量。**如果零件编号不在数据库中，那么程序必须显示出错信息。
- **显示表格列出数组库中的全部信息。**零件必须按照录入的顺序显示出来。
- **终止程序的执行。**

使用i（即插入）、s（即搜索）、u（即更新）、p（即显示）和q（即退出）分别表示这些操作。与程序的会话可能如下：

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10
Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10

Enter operation code: s
Enter part number: 914
Part not found.

Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8

Enter operation code: p
Part Number  Part Name           Quantity on Hand
    528      Disk drive                 8
```


www.TopSage.com

```
    914      Printer cable              5
Enter operation code: q
```

程序将在结构中存储每种零件的信息。这里会限制数据库的大小为100种零件，这使用数组来存储结构成为可能，这里称此数组为inventory。（如果这个数组的范围太小，稍后可以经常改变它。）为了记录当前存储在数组中的零件的编号，将使用名为num_parts的变量。

既然此程序是以菜单方式驱动的，那么十分容易勾划出主循环结构：

```
for (;;) {
  提示用户输入操作码
  读操作码
  Switch (操作码)
    case 'i': 执行插入操作; break;
    case 's': 执行搜索操作; break;
    case 'u': 执行更新操作; break;
    case 'p': 执行显示操作; break;
    case 'q': 终止程序;
    default: 打印出错信息;
}
}
```

为了方便起见，将分别设置不同的函数执行插入、搜索、更新和显示操作。因为这些函数将都需要访问inventory和num_parts，所以可以把这些变量外部化。或者把变量藏在main函数内，然后把它们作为实际参数传递给函数。从设计角度来说，通常把针对函数的变量局部化比把它们外部化更好（如果忘记了原因，请见10.2节）。然而，在此程序中，把inventory和num_parts隐藏在main函数中将只会使程序复杂化。 340

由于稍后会解释的原因，这里决定把程序分割为三个文件：invent.c文件，它包含程序的大部分内容；readline.h文件，它包含read_line函数的原型；readline.c文件，它包含read_line函数的定义。在本节的后半部分将讨论后两个文件。现在将集中讨论invent.c文件。

invent.c

```
/* Maintains a parts database (array version) */

#include <stdio.h>
#include "readline.h"

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} inventory[MAX_PARTS];

int num_parts = 0;   /* number of parts currently stored */

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);

/**********************************************************
 * main: Prompts the user to enter an operation code,     *
 *       then calls a function to perform the requested   *
 *       action. Repeats until the user enters the        *
 *       command 'q'. Prints an error message if the user *
 *       enters an illegal code.                          *
 **********************************************************/
```

www.TopSage.com

大家网 TopSage.com

```
main()
{
  char code;

  for (;;) {
    printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n')   /* skips to end of line *  /
      ;
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}

/**********************************************************
 * find_part: Looks up a part number in the inventory     *
 *            array. Returns the array index if the part  *
 *            number is found; otherwise, returns -1.     *
 **********************************************************/
int find_part(int number)
{
  int i;

  for (i = 0; i < num_parts; i++)
    if (inventory[i].number == number)
      return i;
  return -1;
}

/**********************************************************
 * insert: Prompts the user for information about a new   *
 *         part and then inserts the part into the        *
 *         database. Prints an error message and returns  *
 *         prematurely if the part already exists or the  *
 *         database is full.                              *
 **********************************************************/
void insert(void)
{
  int part_number;

  if (num_parts == MAX_PARTS) {
    printf("Database is full; can't add more parts.\n");
    return;
  }

  printf("Enter part number: ");
  scanf("%d", &part_number);
  if (find_part(part_number) >= 0) {
    printf("Part already exists.\n");
    return;
  }

  inventory[num_parts].number = part_number;
  printf("Enter part name: ");
```

341

www.TopSage.com

```
  read_line(inventory[num_parts].name, NAME_LEN);                    342
  printf("Enter quantity on hand: ");
  scanf("%d", &inventory[num_parts].on_hand);
  num_parts++;
}

/**********************************************************
 * search: Prompts the user to enter a part number, then  *
 *         looks up the part in the database. If the part *
 *         exists, prints the name and quantity on hand;  *
 *         if not, prints an error message.               *
 **********************************************************/
void search(void)
{
  int i, number;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Part name: %s\n", inventory[i].name);
    printf("Quantity on hand: %d\n", inventory[i].on_hand);
  } else
    printf("Part not found.\n");
}

/**********************************************************
 * update: Prompts the user to enter a part number.       *
 *         Prints an error message if the part doesn't    *
 *         exist; otherwise, prompts the user to enter    *
 *         change in quantity on hand and updates the     *
 *         database.                                      *
 **********************************************************/
void update(void)
{
  int i, number, change;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    inventory[i].on_hand += change;
  } else
    printf("Part not found.\n");
}

/**********************************************************
 * print: Prints a listing of all parts in the database,  *
 *        showing the part number, part name, and         *
 *        quantity on hand. Parts are printed in the      *
 *        order in which they were entered into the       *
 *        database.                                       *
 **********************************************************/                                                    343
void print(void)
{
  int i;

  printf("Part Number   Part Name                "
         "Quantity on Hand\n");
  for (i = 0; i < num_parts; i++)
    printf("%7d      %-25s%11d\n", inventory[i].number,
           inventory[i].name, inventory[i].on_hand);
}
```

www.TopSage.com

在main函数中，格式串" %c"允许scanf函数在读入操作代码之前跳过空白字符。格式串中的空格是至关重要的。如果没有它，那么scanf函数有时会读入使输入提前终止的换行符。

程序包含一个名为find_part的函数，main函数不调用此函数。这个“辅助的”函数用来避免多余的代码和简化更重要的函数。通过调用find_part，insert函数、search函数和update函数可以定位数据库中的零件（或者用来简单确定是否存在零件）。

这里只留了一个细节：read_line函数。这个函数用来读零件的名字。13.3节讨论了包含书写此类函数的问题。但是13.3节开发的read_line的写法无法正常工作在当前的程序中。请思考当用户插入零件时会发生什么：

```
Enter part number: 528
Enter part name: Disk drive
```

在录入完零件的编号后，用户按回车（或回退）键，然后再录入零件的名字后再按回车键，这样每次都给程序留下一个必须读取的无形的换行符。为了讨论的目的，现在假装这些字符都是可见的：

```
Enter part number: 528␤
Enter part name: Disk drive␤
```

当调用scanf函数来读零件编号时，函数吸收了5、2和8，但是留下了字符␤未读。如果试图用原始的read_line函数来读零件名称，那么函数将会立刻遇到字符␤，并且停止读入。当数字化输入后边跟有字符输入时，这种问题非常普遍。解决办法就是编写read_line函数，使它在开始往字符串中存储字符之前跳过空白字符。这不仅解决了换行符的问题，而且可以避免存储用户在零件名称的开始处录入的任何空格。

由于read_line函数与invent.c文件中的其他函数无关，而且由于它在其他程序中有复用的可能，所以决定把此函数从invent.c中独立出来。read_line函数的原型将来自头文件
344 read.h：

readline.h

```
#ifndef READLINE_H
#define READLINE_H

/**********************************************************
 * read_line: Skips leading white-space characters, then  *
 *            reads the remainder of the input line and   *
 *            stores it in str. Truncates the line if its *
 *            length exceeds n. Returns the number of     *
 *            characters stored.                          *
 **********************************************************/
int read_line(char str[], int n);

#endif
```

readline.c

```
#include <ctype.h>
#include <stdio.h>
#include "readline.h"

int read_line(char str[], int n)
{
  int ch, i = 0;

  while (isspace(ch = getchar()))
    ;

  while (ch != '\n' && ch != EOF) {
    if (i < n)
```

www.TopSage.com

TopSage.com

```
      str[i++] = ch;
    ch = getchar();
  }

  str[i] = '\0';
  return i;
}
```

isspace函数用来判定它的实际参数是否是空白字符。15.3节解释了ch用int代替char的原因，还解释了它便于判定EOF的原因。

16.4 联合

像结构一样，**联合**（union）也是由一个或多个成员构成的，而且这些成员可能具有不同的数据类型。但是，编译器只为联合中最大的成员分配足够的内存空间。联合的成员在这个空间内彼此覆盖。这样的结果是，给一个成员赋予新值也会改变所有其他成员的值。

为了说明联合的基本性质，现在声明一个联合变量u，且这个联合变量有两个成员： 345

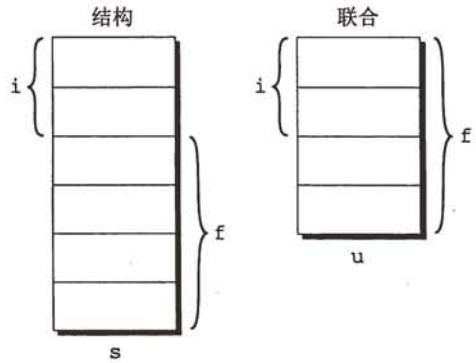
```
union {
  int i;
  float f;
} u;
```

注意联合的声明方式非常类似于结构的声明方式：

```
struct {
  int i;
  float f;
} s;
```

事实上，结构变量s和联合变量u只有一处不同：s的成员都是存储在不同的内存地址中，而u的成员都是存储在同一内存地址中。下面是s和u在内存中的存储情况（假设int类型的值要占用2个字节内存，而且float类型的值则占用4个字节内存）：

在变量s的结构中，成员i和f占有不同的内存单元。s总共占用了6个字节的内存单元。在变量u的联合中，成员i和f互相交迭（i实际上是占用了f的前两个字节内存），所以u只占用了4个字节的内存单元。如图所示，u.i和u.f具有相同的地址。



访问联合成员的方法和访问结构成员的方法相同。为了把数82存储到u的成员i中，可以写成

```
u.i = 82;
```

为了把值74.8存储到成员f中，可以写成

```
u.f = 74.8;
```

既然编译器把联合的成员重叠存储，那么改变一个成员就会使之前存储在任何其他成员中的任

www.TopSage.com

意值发生改变。因此，如果把一个值存储到u.f中，那么将会丢失先前存储在u.i中的任意值。（如果测试u.i的值，那么它会显示出无意义的内容。）类似地，改变u.i也会影响u.f。由于这个性质，所以可以把u想成是存储i或者存储f的地方，而不是同时存储二者的地方。（s的结构
346 允许存储i和f。）

联合的性质和结构的性质几乎一样。所以可以用声明结构标记和类型的方法来声明联合的标记和类型。像结构一样，联合可以使用运算符=进行复制操作，也可以在函数间进行传递，还可以作为函数的返回。

联合初始化的方式甚至都和结构的初始化很类似。但是，只有联合的第一个成员可以获得初始值。例如，可以用下列方式初始化联合u的成员i为0：

```
union {
  int i;
  float f ;
} u = {0} ;
```

注意，即使初始化式是单独一个表达式，也要保证大括号的存在。大括号内的表达式必须是常量。

联合有几个重要的应用。现在将讨论其中的两种。联合的另外一个应用是关于观察存储的不同方法，由于这个应用与机器相关，所以将推迟到20.3节再介绍。

16.4.1 使用联合来节省空间

在结构上经常使用联合作为节省空间的一种方法。假设打算设计的结构将包含礼品册中项目的售出信息。礼品册上只有三种商品：书籍、杯子和衬衫。每个商品项目都含有库存量、价格以及和项目类型相关的其他信息：

- 书籍：书名、作者、页数。
- 杯子：设计。
- 衬衫：设计、可选颜色、可选尺寸。

首要的设计是希望结果可以具有下列结构：

```
struct catalog_item {
  int stock_number;
  float price;
  int item_type;
  char title[TITLE_LEN+1];
  char author[AUTHOR_LEN+1];
  int num_pages;
  char design[DESIGN_LEN+1];
  int colors;
  int sizes;
};
```

成员item_type将具有值BOOK、MUG或SHIRT之一。成员colors和sizes将存储颜色和尺寸的组合代码。

347 虽然上述结构十分合用，但是它浪费了空间，因为对礼品册中的所有项目来说只有结构中的部分信息是常用到的。比如，如果项目是书籍，那么就不需要存储design、colors和sizes。通过在结构catalog_item内部放置联合的方法，可以减少结构所要求的内存空间。联合的成员将是一些特殊的结构，每种结构都包含特定类型的商品项目所需要的数据：

```
struct catalog_item {
  int stock_number;
  float price;
  int item_type;
  union {
    struct {
```

www.TopSage.com

```
      char title[TITLE_LEN+1];
      char author[AUTHOR_LEN+1];
      int num_pages;
    } book;
    struct {
      char design[DESIGN_LEN+1];
    } mug;
    struct {
      char design[DESIGN_LEN+1];
      int colors;
      int sizes;
    } shirt;
  } item;
};
```

注意联合（名为item）是结构catalog_item的成员，而结构book、mug和shirt则是联合item的成员。如果c是表示书籍的结构catalog_item，那么可以用下列方法显示书籍的名称：

```
printf("%s", c.item.book.title);
```

正如上边的例子显示的那样，访问嵌套在结构内部的联合是很困难的：为了定位书籍的名称，不得不指明结构的名字（c）、结构的联合成员的名字（item）、联合的结构成员的名字（book），以及此结构的成员名（title）。C++语言通过允许**匿名**（**anonymous**）的方式使联合更容易使用。C++在C++语言中，在创建结构时可以忽略名字item，然后书写c.book.title来代替c.item.book.title。

16.4.2 使用联合来构造混合的数据结构

联合另一个重要的应用：创建含有不同数据类型的混合的数据结构。现在假设需要数组的元素是int值和float值的混合。因为数组的元素必须是相同的类型，所以好像不可能产生如此类型的数组。但是，利用联合这件事就相对容易多了。首先，定义一种联合类型，此种联合所包含的成员分别表示要存储在数组中的不同数据类型：

348

```
typedef union {
  int i;
  float f;
} Number;
```

接下来，创建一个数组，使数组的元素是具有Number类型的值：

```
Number number_array[1000];
```

数组number_array的每个元素都是具有Number类型的联合。联合Number既可以存储int类型的值又可以存储float类型的值，所以使用此种类型值来存储数组number_array中的int和float的混合值是可以的。例如，假设需要用数组number_array的0号元素来存储5，而用1号元素来存储8.395。下列赋值语句可以达到期望的效果：

```
number_array[0].i = 5;
number_array[1].f = 8.395;
```

16.4.3 为联合添加“标记字段”

联合所面临的主要问题是：没有简便的方法可以表明联合最后改变的成员，并且因此会含有无意义的值。请思考下面这个问题。编写了一个函数，用来显示当前存储在联合Number中的值。这个函数可能有下列框架：

```
void print_number(Number n)
{
  if (n包含一个整数)
    printf("%d", n.i);
  else
```

www.TopSage.com

大家网 TopSage.com

```
    printf("%g", n.f);
}
```

可惜的是，没有方法可以帮助函数print_number来确定n包含的是整数还是浮点数。

为了记录此信息，可以把联合嵌入一个结构中，且此结构还含有另一个成员："标记字段"或者"判别式"，它是用来提示当前存储在联合中的内容的。在本节先前讨论的结构catalog_item中，item_type就是用于此目的的。

下面把Number类型转换成具有嵌入联合的结构类型：

```
#define INT_KIND    0
#define FLOAT_KIND 1

typedef struct {
  int kind;    /* tag field */
  union{
    int i;
    float f ;
   } u;
} Number;
```

349 Number有两个成员kind和u。成员kind有两种可能的值INT_KIND和FLOAT_KIND。

每次给u的成员赋值时，也会改变kind，从而提示出修改的是u的哪个成员。例如，如果n是Number类型的变量，对u的成员i进行赋值操作可以采用下列形式：

```
n.kind = INT_KIND;
n.u.i = 82;
```

注意对i赋值要求首先选择n的成员u，然后才是u的成员i。

当需要找回存储在Number型变量中的数时，kind将表明联合的哪个成员是最后进行赋值的。函数print_number可以利用这种能力：

```
void print_number(Number n)
{
  if (n.kind == INT_KIND)
    printf("%d",n.u.i);
  else
    printf("%g",n.u.f);
}
```

每次对联合的成员进行赋值时，程序负责改变标记字段的内容。如果标记字段维护数据失败，可能会导致奇怪的错误。一些编程语言采用更加安全的方法来处理标记字段，但是C语言没有做这样的尝试。

16.5 枚举

在一些程序中，我们会需要变量只具有少量有意义的值。例如，"布尔型"变量应该只有两种可能的值："真值"和"假值"。用来存储扑克牌花色的变量应该只有四种隐含的值："梅花"、"方片"、"红桃"和"黑桃"。显然可以用声明成整数的方法来处理此类变量，并且用一组代码来表示变量的可能值：

```
int s;     /* s will store a suit      */

s = 2;     /* 2 represents "hearts"    */
```

虽然这种方法可行，但是也遗留了许多问题。某些人读程序时可能不会意识到suit不是普通的
350 整型变量，而且不会知道2具有特殊含义。

使用宏来定义牌的花色"类型"和不同花色的名字是一种正确的措施：

```
#define SUIT        int
#define CLUBS       0
#define DIAMONDS    1
#define HEARTS      2
#define SPADES      3
```

那么前面的示例现在可以变得更加容易阅读：

```
SUIT s;

s = HEARTS;
```

这种方法是一种改进，但是它仍然不是最好的解决方案，因为这样做没有为阅读程序的人指出宏表示具有相同“类型”的值。如果可能值的数量多于这些现有的，那么为每种类型定义独立的宏是很麻烦的。而且，由于预处理器会删除已经定义的CLUBS、DIAMONDS、HEARTS和SPADES这些名字，所以在调试期间这些名字是无效的。

C语言为具有少量可能值的变量提供了特别设计的一种特殊类型。**枚举**（enumeration）是一种由程序员列出（“列举”）值的类型，而且程序员必须为每种值命名（**枚举常量**）。下列示例枚举的值（CLUBS、DIAMONDS、HEARTS和SPADES）可以赋值给变量s1和s2：

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

虽然枚举没有什么和结构、联合共同的地方，但是它们的声明方法很类似。然而，不像结构或联合的成员那样，枚举常量的名字必须不同于闭合作用域内声明的其他标识符。

枚举常量类似于用#define指令产生的常量，但是两者不完全一样。特别地，枚举常量是C语言的作用域规则的对象：如果枚举声明在函数体内，那么它的常量对外部函数来说是不可见的。

16.5.1 枚举标记和枚举类型

为了命名结构和联合的相同的原因，会常常需要创建枚举的名字。相对于结构和联合，这里有两种方法命名枚举：通过声明标记的方法，或者通过使用typedef来创建独一无二的类型名。

枚举标记类似于结构和联合的标记。例如，为了定义标记suit，可以写成

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

351

变量suit可以按照下列方法来进行声明：

```
enum suit s1, s2;
```

作为一种替换，还可以用typedef来给suit进行类型命名：

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
Suit s1, s2;
```

顺便说一下，利用typedef来创建布尔类型是非常好的一种方法：

```
typedef enum {FALSE, TRUE} Bool;
```

16.5.2 枚举作为整数

在系统内部，C语言会把枚举变量和常量作为整数来处理。例如，这里的枚举suit中，CLUBS、DIAMONDS、HEARTS和SPADES分别表示数0、1、2和3。

我们可以为枚举常量自由选择不同的值。现在假设希望CLUBS、DIAMONDS、HEARTS和SPADES分别表示1、2、3和4。在声明枚举时可以指明这些数：

```
enum suit {CLUBS = 1, DIAMONDS = 2, HEARTS = 3, SPADES = 4};
```

枚举常量的值可以是任意整数，列出也可以不用按照特定的顺序：

```
enum dept {RESEARCH = 20, PRODUCTION = 10, SALES = 25};
```

两个或多个枚举常量具有相同的值甚至也是合法的。

www.TopSage.com

当没有为枚举常量指定值时，它的值是一个大于前一个常量值的值。(默认第一个枚举常量的值为0。) 在下列枚举中，BLACK的值为0，LT_GRAY为7，DK_GRAY为8，而WHITE为15：

```
enum EGA_colors {BLACK, LT_GRAY = 7, DK_GRAY, WHITE = 15};
```

因为除了稀疏地掩饰整数外，枚举的值什么也不是，所以C语言允许把它们与普通整数进行混合：

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;
i = DIAMONDS;      /* i is now 1                    */
s = 0;             /* s is now 0  (CLUBS)           */
s++;               /* s is now 1  (DIAMONDS)        */
i = s + 2;         /* i is now 3                    */
```

编译器会把s作为整型变量来处理，而名字CLUBS、DIAMONDS、HEARTS和SPADES只是数0、1、2和3的同义词。

虽然把枚举的值作为整数使用非常方便，但是把整数用作枚举的值却是非常危险C++的
352 (例如，可能会在s中存储数4)。在这方面C++语言比C语言要求严格。在没有强制的情况下，C++语言不允许整数作为枚举的值来使用。

16.5.3 用枚举声明“标记字段”

枚举用来解决16.4节遇到的问题是非常合适的：用来确定联合中最后一个被赋值的成员。例如，在结构Number中，可以使枚举的成员kind来替代int：

```
typedef struct {
  enum {INT_KIND, FLOAT_KIND} kind;
  union {
    int i;
    float f;
  } u;
} Number ;
```

这种新结构和旧结构的用法完全一样。这样做的优势是不仅远离了宏INT_KIND和FLOAT_KIND(他们现在是枚举常量)，而且阐明了kind的含义，现在kind显然应该只有两种可能值：INT_KIND和FLOAT_KIND。

问与答

问：当试图使用sizeof来确定结构中的字节数量时，获得的数大于成员加在一起后的数。为什么会这样？

答：先来看看下面这个例子：

```
struct {
  char a;
  int b;
} s;
```

如果char型的值占有一个字节，而int型值占用4个字节，s会是多大呢？显然，答案是5个字节，但可能不是正确的答案。一些计算机要求数据项从某个数量字节（一般是4个字节）的倍数开始。为了满足计算机的要求，通过在邻近的成员之间留“空洞”（即无用的字节）的方法，编译器会把结构的成员“排列”起来。如果假设数据项必须从4个字节的倍数开始，那么结构s的成员a将跟着3个字节的空洞。结果是sizeof(s)将为8。

353 顺便说一句，就像在成员间有空洞一样，结构也可以在末尾有空洞。例如，

```
struct {
  int a;
```

TopSage.com

```
  char b;
} s;
```

此结构可以在成员b的后边有3个字节的空洞。

问：是否可能会在结构的开始处有“空洞”？

答：不会的。C标准说明只允许在成员之间或者最后一个成员的后边有空洞。这样的结果是结构第一个成员的地址保证和整个结构的地址完全一样。（但是，注意两个指针没有相同的类型。）

问：使用==来判定两个结构是否相等为什么是不合法的？（p.231）

答：这种操作超出了C语言的范围，因为无法实现它始终是和语言的体系相一致的。逐个比较结构成员将是极没有效率的。比较结构中的全部字节是相对较好的方法（许多计算机有特殊的指令可以用来快速执行此类比较）。然而，如果结构含有空洞，那么比较字节会产生不正确的结果。甚至是，假设对应的成员有同样的值，那么出现在空洞中的废弃值也可能会不同。这类问题可以通过下列方法解决，那就是编译器要确保空洞始终包含相同的值（比如零）。然而，初始化空洞会影响全部使用结构的程序的性能，所以它是不可行的。

问：为什么C语言提供两种命名结构类型的方法（标记命名和typedef命名）？（p.232）

答：C语言早期没有typedef，所以标记是结构类型命名的唯一有效方法。当加入typedef时，已经太晚了，以致无法删除标记了。此外，当结构的成员是指向结构自身的指针时，标记始终是非常必要的。

问：结构可否同时有标记名和类型名？（p.233）

答：可以。事实上，虽然不要求，但是标记名和类型名甚至可以是一样的：

```
typedef struct part {
  int number ;
  char name [NAME_LEN+1 ] ;
  int on_hand;
} part ;
```

C++ 在C++语言中，所有标记也是类型名；把part作为标记来声明也就是使part自动成为了类型名，而不再需要类型定义。 354

问：如何能在程序的几个文件中共享结构类型呢？

答：把结构标记（或者，如果喜欢也可以用typedef）的声明放在头文件中，然后在需要结构的地方包含此头文件就可以了。例如，为了共享结构part，可以在头文件中放入下列这行内容：

```
struct part {
  int number;
  char name [NAME_LEN+1];
  int on_hand;
};
```

注意，这里只是声明结构标记，不是声明具有这种类型的变量。

顺便提一句，含有结构标记声明或结构类型声明的头文件可能需要保护以避免多重包含（➤15.2.6节）。在同一文件中声明标记或类型两次是错误的。类似的说明也适用于联合和枚举。

问：如果在两个不同的文件中包含了结构part的声明，那么一个文件中的part型变量和另一个文件中的part型变量是否一样呢？

答：技术上来说，不一样。但是，C标准提到，一个文件中的part型变量所具有的类型和另一个文件中的part型变量所具有的类型是兼容的。具有兼容类型的变量可以互相赋值，所以是“兼容的”类型和是“相同的”类型之间几乎没有差异。

练习

16.1节

1. 在下列声明中，结构x和结构y都拥有名为x和y的成员：

```
struct { int x,  y; } x;
struct { int x,  y; } y;
```

www.TopSage.com

基于独立的基础而言，这些声明是否合法呢？两个声明是否可以同时出现在程序中呢？请证明你的想法是正确的。

2. (a) 声明名为c1、c2和c3的结构变量，每个结构变量都拥有double型的成员re和im。
   (b) 修改(a)中的声明，使c1的成员初始值为0.0和1.0，而c2的成员初始值为1.0和0.0。(c3无初始值。)
   (c) 编写语句用来把c2的成员复制给c1。这项操作是否可以在一条语句中完成，或者是需要两条？
   (d) 编写语句把c1和c2的相应成员进行相加，并且把结果存储在c3中。

16.2节

355 3. (a) 说明如何为具有两个double型的成员re和im的结构声明名为complex的标记。
   (b) 利用标记complex来声明名为c1、c2和c3的变量。
   (c) 编写名为make_complex的函数，此函数用来把两个实际参数（两个参数的类型都是double型）存储在complex型结构中，然后返回此结构。
   (d) 编写名为add_complex的函数，此函数用来把两个实际参数（都是complex型结构）的相应成员进行相加运算，然后返回结果（另一个complex型结构）。
4. 重做练习3的各种操作，这次要求使用类型来命名complex。

16.3节

5. 下列结构用来存储图形屏幕上的对象信息。结构point用来存储屏幕上点的*x*轴和*y*轴坐标，结构rectangle用来存储矩形的左上和右下坐标点。

```
struct point { int x, y; };
struct rectangle { struct point upper_left, lower_right; };
```

   编写函数，要求可以在rectangle型结构变量r上执行下列操作，且r作为实际参数传递。
   (a) 计算r的面积。
   (b) 计算r的中心，并且把此中心作为point型的值返回。
   (c) 移动r，方法是x单元按照X轴方向移动，y单元按照*y*轴移动，并且返回r修改后的内容。(x和y是函数额外的实际参数。)
   (d) 确定点p是否位于r内，返回TRUE或者FALSE。(p是具有struct point类型的额外的实际参数。)
6. 编写程序用来要求用户录入国家（地区）名称，然后可以在数组country_codes中查找到它。如果找到对应的国家（地区）名称，程序需要显示相应的国家（地区）电话代码。如果没有找到，程序应该显示出错消息。
7. 修改程序invent.c使p（显示）操作可以根据零件数显示存储的零件。
8. 修改程序invent.c使inventory和num_parts成为main函数的局部内容。
9. 通过为结构part添加成员price来修改程序invent.c。insert函数应该要求用户录入新的数据项的价格。serach函数和print函数应该显示价格。添加新的命令，从而允许用户改变零件的价格。

16.4节

10. 假设s具有下列结构：

```
struct {
  float a;
  union {
    char b[4];
    float c;
    int d;
  } e;
  char f [4];
} s;
```

   如果char型值占有1个字节，int型值占有2个字节，而float型值占有4个字节，那么编译器将为s
356 分配多大的内存空间？(假设编译器没有在成员之间留“空洞”。)

11. 假设s具有下列结构(point是在练习5中声明的结构标记)：

```
struct shape {
  int shape_kind;          /* RECTANGLE or CIRCLE */
```

www.TopSage.com

```
  struct point center;     /* coordinates of center    */
  union {
    struct {
      int length, width;
    } rectangle;
    struct {
      int radius;
    } circle;
  } u;
} s;
```

请指出下列哪些语句是合法的，并且说明如何修改不合法的语句。

(a) `s.shape_kind = RECTANGLE;`

(b) `s.center.x = 10;`

(c) `s.length = 25;`

(d) `s.u.rectangle.width = 8;`

(e) `s.u.circle = 5;`

(f) `s.u.radius = 5;`

12. 假设shape是练习11中声明的结构标记。编写函数用来在shape型结构变量s上完成下列操作，并且s作为实际参数在函数间传递：

(a) 计算s的面积。

(b) 计算s的中心，返回point型的中心值。

(c) 移动s，方法是x单元按照*x*轴方向移动，y单元按照*y*轴移动，并且返回s修改后的内容。（x和y是函数额外的实际参数。）

(d) 确定点p是否位于s内，返回TRUE或者FALSE。（p是具有struct point类型的额外的实际参数。）

13. 编写一个类似于invent.c的程序，利用catalog_item型结构来存储礼品册中数据项的信息。

16.5节

14. (a) 为枚举声明标记，此枚举的值表示一个星期的7天。

(b) 用typedef定义(a)中枚举的名字。

15. 下列关于枚举常量的叙述哪些是正确的？

(a) 枚举常量可以表示程序员指明的任何整数。

(b) 枚举常量具有的性质和用#define产生的常量的性质完全一样。

(c) 枚举常量的默认值为0，1，2，…。

(d) 枚举中的任何常量必须具有不同的值。

(e) 枚举常量在表达式中可以作为整数使用。

16. 假设b和i具有如下形式的声明：

```
enum {FALSE, TRUE} b;
int i;
```

下列哪些语句是合法的？哪些是“安全的”（始终产生有意义的结果）？ 357

(a) `b = FALSE;`

(b) `b = i;`

(c) `b++;`

(d) `i = b;`

(e) `i = 2 * b + i;`

17. (a) 棋盘的每个角上可能会有一个棋子，即兵、士、相、车、皇后或国王，也可能为空。每个棋子可能为黑色的也可能是白色的。请定义两个枚举类型Piece用来包含7种可能的值（其中一种为“空”），Color用来表示2种颜色。

(b) 利用(a)中的类型，定义名为Square的结构类型，使此类型可以存储棋子的类型和颜色。

(c) 利用(b)中的类型Square，声明一个名为board的8×8的数组，此数组可以用来存储棋盘上的全部内容。

(d) 给(c)中的声明添加初始值，使board的初始值对应日常国际象棋比赛开始时的棋子布置。 358

www.TopSage.com

大家网
TopSage.com

第17章 指针的高级应用

人们只会在脑海显示复杂的信息。比如看，静景无论多么生动，都不如景色的运动、流动和改变重要。

前面的两章描述了指针的两种重要应用。第11章说明了如何利用指向变量的指针作为函数的实际参数从而允许函数修改变量。第12章说明了如何对数组元素进行指针的算术运算来处理数组。本章则通过观察两种额外的应用来完善指针的内容：**动态存储分配**（dynamic storage allocation）和指向函数的指针。

通过使用动态存储分配，程序可以在执行期间获得需要的内存块。17.1节解释动态存储分配的基本概念。17.2节则讨论动态分配字符串，这种方法比C语言通常的字符数组固定长度的方式更加灵活。17.3节大概地介绍数组的动态存储分配。17.4节处理存储分配的问题，即不再需要内存单元时，动态地释放已分配的内存块。

因为动态分配的结构可以链接在一起形成表、树和其他高度灵活的数据结构，所以它们在C语言编程中扮演着重要的角色。17.5节重点讲述**链表**（linked list），它是最基础的链式数据结构。17.6节介绍指向指针的指针，这是来自于17.5节的主题。

17.7节介绍指向函数的指针，这是非常有用的内容。C语言中一些功能最强大的库函数都期望把指向函数的指针作为实际参数。这里将考察其中一个函数qsort，它是一个对任意数组进行排序的函数。

17.1 动态存储分配

359

C语言的数据结构通常是固定大小的。例如，数组拥有固定数量的元素，而且每个元素具有固定的大小。因为在编写程序时强制选择了大小，所以固定大小的数据结构可能会有问题。也就是说，在没有修改程序并且再次编译程序的情况下无法改变数据结构的大小。

请思考16.3节中允许用户添加部分数据库的invent程序。数据库存储在长度为100的数组中。为了扩大数据库的容量，可以增加数组的大小并且重新编译程序。但是，无论如何增大数组，始终有可能填满数组。幸运的是没有丢失全部数据。C语言支持**动态存储分配**，即在程序执行期间分配内存单元的能力。利用动态存储分配，可以根据需要设计扩大（和缩小）的数据结构。

虽然可以适用于所有类型的数据，但是动态存储分配更常用于字符串、数组和结构。动态地分配结构是特别有趣的，因为可以把它们链接形成表、树或其他数据结构。

17.1.1 内存分配函数

为了动态地分配存储空间，将需要调用3种内存分配函数中的一种，这些函数都是声明在<stdlib.h>中的：

- malloc函数——分配内存块，但是不对内存块进行初始化。
- calloc函数——分配内存块，并且对内存块进行清除。
- realloc函数——调整先前分配的内存块。

www.TopSage.com

在这3种函数中，malloc函数可能是最常用的一种。因为malloc函数不需要对分配的内存块进行清除，所以它比calloc函数更高效。

当为申请内存块而调用内存分配函数时，由于函数无法知道计划存储在内存块中的数据是什么类型的，所以它不能返回普通的int型指针或char型指针或其他。取而代之的，函数会返回void*型的值。void*型的值是“通用”指针，本质上它只是内存地址。

17.1.2 空指针

当调用内存分配函数时，无法定位满足我们需要的足够大的内存块，这种问题始终可能出现。如果真的发生了这类问题，函数会返回**空指针**（null pointer）。空指针是“指向为空的指针”，这是一个区别于所有有效指针的特殊值。在把返回值存储在指针变量p中以后，需要判断p是否为空指针。

程序员的责任是测试任意内存分配函数的返回值，并且在返回空指针时采取适当的操作。通过空指针试图访问内存的效果是未定义的，程序可能会崩溃或者出现不可预测的行为。

360

Q&A 由于用名为NULL的宏来表示空指针，所以可以用下列方式测试malloc函数的返回值：

```
p = malloc(10000);
if (p == NULL) {
  /* allocation failed;  take appropriate action  */
}
```

一些程序员把malloc函数的调用和NULL的测试组合在一起：

```
if ((p = malloc(10000))  == NULL) {
  /* allocation failed;  take appropriate action  */
}
```

名为NULL的宏在6个头文件中都有定义：<locale.h>、<stddef.h>、<stdio.h>、<stdlib.h>、<string.h>和<time.h>。只要把这些头文件中的一个包含在程序中，编译器就可以识别出NULL。当然，使用任意内存分配函数的程序都会包含<stdlib.h>，这使NULL必然有效。

在C语言中，指针测试真假的方法和数的测试一样。所有非空指针都为真，而只有空指针为假。因此，不用编写语句

```
if (p == NULL) ...
```

而是可以写成

```
if (!p) ...
```

而且，不用编写语句

```
if (p ! = NULL) ...
```

而是可以写成

```
if (p) ...
```

作为一种书写风格，这里喜欢与NULL进行明确的比较。

17.2 动态分配字符串

动态内存分配经常用于字符串操作。字符串始终存储在固定长度的数组中，而且可能很难预测这些数组需要的长度。通过动态地分配字符串，可以推迟到程序运行时才作决定。

www.TopSage.com

17.2.1　使用malloc函数为字符串分配内存

malloc函数具有如下原型：

361
```
void *malloc(size_t size);
```

malloc函数分配size字节的内存块，并且返回指向此内存块的指针。注意size的类型是size_t（➤21.3节），这是在C语言库中定义的无符号整数类型。除非正在分配一个非常巨大的内存块，否则可以只把size考虑成普通整数。

用malloc函数来为字符串分配内存是很容易的，因为C语言保证char型值确切需要一个字节的内存（换句话说，sizeof(char)的值为1。）。为给n个字符的字符串分配内存空间，可以写成

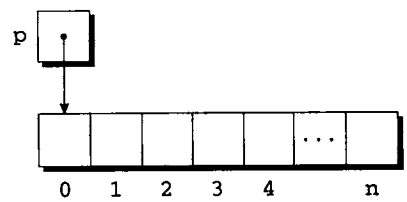
```
p = malloc(n+1);
```

这里的p是char*型变量。在执行复制操作时会把malloc函数返回的通用指针转化为char*类型，而不需要强制执行。（通常情况下，可以把void*型值赋给任何指针类型的变量。）Q&A然而，一些程序员喜欢强制转换malloc函数的返回值：

```
p = (char *) malloc(n+1);
```

当使用malloc函数为字符串分配内存空间时，不要忘记包含空字符的空间。

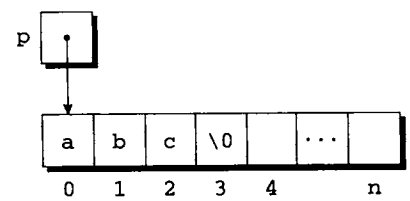
由于使用malloc函数分配内存不需要清除或者以任何方式的初始化，所以p指向带有n+1个字符的未初始化的数组：



调用strcpy函数是对上述数组进行初始化的一种方法：

```
strcpy(p, "abc");
```

数组中的前4个字符分别为a、b、c和\0：



17.2.2　在字符串函数中使用动态存储分配

362
动态存储分配使编写返回指向“新”字符串的指针的函数成为可能，所谓新字符串是指在调用此函数之前字符串并不存在。如果编写的函数不改变任何一个字符串而把两个字符串连接起来，请思考一下这样做会遇到什么问题。C标准库没有包含此类函数（因为strcat函数改变了传递过来的字符串中的一个，所以此函数并不是希望的函数），但是可以很容易的自行写出这样的函数。

自行编写的函数将测量用来连接的两个字符串的长度，然后调用malloc函数只为结果分配适当数量的内存空间。接下来函数会把第一个字符串复制到新的内存空间中，并且调用strcat函数来拼接第二个字符串。

www.TopSage.com

TopSage.com

```
char *concat(const char *s1, const char *s2)
{
  char *result ;

  result = malloc(strlen(s1) + strlen(s2) +1);
  if (result == NULL)  {
    printf("Error: malloc failed in concat\n");
    exit (EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat (result,s2);
  return result;
}
```

如果malloc函数返回空指针，那么concat函数显示出错信息并且终止程序。并不总是采取正确的措施，一些程序需要从内存分配失误中恢复并且继续运行。

下面是concat函数可能的调用方式：

```
p = concat("abc", "def");
```

调用函数之后，p将指向字符串"abcdef"，此字符串是存储在动态分配的数组中的。数组包括结尾的空字符一共有7个字符长。

像concat这样动态分配存储空间的函数必须小心使用。当不再需要concat函数返回的字符串时，需要调用free函数来释放它占用的空间。如果不这样做，程序可能会过早地运行越界。

17.2.3 动态分配字符串的数组

13.7节解决了在数组中存储字符串的问题。我们发现按行的方式在二维字符数组中存储字符串可能会浪费空间，所以试图为字符串字面量设置为指针数组。13.7节的方法正好和数组元素是指向动态分配字符串的指针方式一样。为了说明这个观点，先来重新编写13.5节的程序remind.c，此程序显示出一个日常提示月列表。

363

17.2.4 程序：显示一个月的提示列表（改进版）

原始程序remind.c把提示字符串存储在二维字符数组中，且数组的每行包含一个字符串。在程序读入一天和相关的提示后，搜索数组从而确定这一天所处的位置，利用strcmp函数可以进行比较工作。然后，程序使用函数strcpy把该位置下面的全部字符串向下移动一个位置。最后，程序把这一天复制到数组中，并且调用strcat函数来添加这一天的提示。

在新程序中（remind2.c），数组是一维的，且数组的元素是指向动态分配的字符串的指针。在此程序中换成动态分配的字符串会有两个主要好处。第一，与提示存储在固定数量的字符数组中（就如同原始程序所做的那样）相比，为要存储的提示分配确切字符数量的空间，这样做可以更有效地利用空间。第二，不需要为一个新提示留空间而调用函数strcpy来把字符串“降低”，而只是移动指向字符串的指针。

下面是新程序，程序中有改动的部分用粗体进行了标注。把二维数组换成指针数组显得异常容易：只需要改变程序的8行内容。

remind2.c

```
/* Prints a one-month reminder list */

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

www.TopSage.com

```
#define MAX_REMIND 50
#define MSG_LEN 60

int read_line(char str[], int n);

main()
{
  char *reminders[MAX_REMIND];
  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--)
      reminders[j] = reminders[j-1];

    reminders[i] = malloc(2 + strlen(msg_str) + 1);
    if (reminders[i] == NULL) {
      printf("-- No space left --\n");
      break;
    }

    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;
}
```

364

17.3　动态分配数组

动态分配数组会获得和动态分配字符串相同的好处（不用惊讶，因为字符串就是数组）。当

www.TopSage.com

编写程序时，常常很难为数组估计合适的大小。较方便的做法是等到程序运行时再来确定数组的实际大小。C语言解决了这个问题，方法是允许在程序执行期间为数组分配空间，然后通过指向数组第一个元素的指针访问数组。数组和指针之间的紧密关系已经在第12章中讨论过了，使用动态分配的数组就好像使用普通数组一样简单。 365

虽然malloc函数可以为数组分配内存空间，但calloc函数确实是最常使用的一种选择，因为calloc函数对分配的内存进行初始化。realloc函数允许根据需要对数组进行“扩展”或“缩减”。

17.3.1 使用 malloc 函数为数组分配存储空间

可以使用mallco函数为数组分配存储空间，这种方法和用它为字符串分配空间非常相像。主要区别就是任意数组的元素不需要像字符串那样是一个字节的长度。这样的结果是，会需要使用sizeof运算符（➤7.4节）来计算出每个元素所需要的空间数量。

假设正在编写的程序需要n个整数的数组，这里的n可以在程序执行期间计算出来。首先需要声明指针变量：

```
int *a;
```

一旦n的值已知了，就让程序调用malloc函数为数组分配存储空间：

```
a = malloc(n * sizeof(int));
```

当计算数组所需要的空间数量时始终要使用sizeof运算符。如果分配的内存空间数量不够会产生严重的后果。思考下面的语句，此语句试图为n个整数的数组分配空间：

```
a = malloc(n * 2);
```

如果int型值大于两个字节，那么malloc函数将无法分配足够大的内存块。当稍后往数组中存储值时，程序可能会崩溃或者行为异常。

一旦a指向动态分配的内存块，就可以忽略a是指针的事实，并且把它用作数组的名字。这都要感谢C语言中数组和指针的紧密关系。例如，可以使用下列循环对a指向的数组进行初始化：

```
for (i = 0; i < n; i++)
a[i] = 0;
```

当然，用指针的算术运算代替下标来访问数组元素的方法是可以选择的。

17.3.2 calloc 函数

虽然malloc函数可以用来为数组分配内存，但是C语言还提供了另外一种选择，即calloc函数，此函数有时会更好用一些。calloc函数在<stdlib.h>中具有如下所示的原型： 366

```
void *calloc(size_t nmemb, size_t size);
```

calloc函数为nmemb个元素的数组分配内存空间，其中每个元素的长度都是size个字节。如果要求的空间无效，那么此函数返回空指针。Q&A在分配了内存之后，calloc函数会通过对所有位设置为0的方式进行初始化。例如，下列calloc函数的调用为n个整数的数组分配存储空间，并且保证全部初始为零：

```
a = calloc(n, sizeof(int));
```

既然calloc函数会清除分配的内存，而malloc函数不会，那么可能有时需要使用calloc函数为非空数组分配空间。通过调用以1作为第一个实际参数的calloc函数，可以为任何类型的数据项分配空间：

```
struct point {int x' y;}*p;
p = calloc (1, sizeof (struct point));
```

在执行此语句之后，p将指向结构，且此结构的成员x和y都会被设为零。

www.TopSage.com

大家网 TopSage.com

17.3.3 realloc函数

一旦为数组分配完内存，稍后可能会发现数组过大或过小。realloc函数可以调整数组的大小使它更适合需要。下列realloc函数的原型出现在<stdlib.h>中：

```
void *realloc(void *ptr, size_t size);
```

当调用realloc函数时，ptr必须指向内存块，且此内存块一定是先前通过malloc函数、calloc函数或realloc函数的调用获得的。size表示内存块的新尺寸，新尺寸可能会大于或小于原有尺寸。虽然realloc函数不要求ptr指向正在用作数组的内存，但实际上通常是这样的。

要确定传递给realloc函数的指针来自于先前malloc函数、calloc函数或realloc函数的调用。如果不是这样的指针，那么程序可能会行为异常。

C标准列出几条关于realloc函数的规则：

- 当扩展内存块时，realloc函数不会对添加进内存块的字节进行初始化。
- 如果realloc函数不能按要求扩大内存块，那么它会返回空指针，并且在原有的内存块中的数据不会发生改变。
- 如果realloc函数调用时以空指针作为第一个实际参数，那么它的行为就将像malloc函数一样。

367

- 如果realloc函数调用时以0作为第二个实际参数，那么它会释放掉内存块。

C标准中明确地包含了一些关于realloc函数行为的规则。尽管如此，我们仍然希望它适当有效。在要求减少内存块大小时，realloc函数应该“在适当位置”缩减内存块，而不需要移动存储在内存块中的数据。由于同样的用意，realloc函数应该始终试图扩大内存块而不需要对其进行移动。如果无法扩大内存块（因为内存块后边的字节已经用于其他目的），realloc函数会在别处分配新的内存块，然后把旧块中的内容复制到新块中。

一旦realloc函数返回，请一定要对指向内存块的所有指针进行更新，因为可能realloc函数移动了其他地方的内存块。

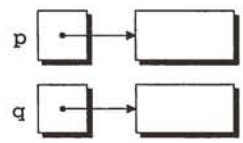
17.4 释放存储

malloc函数和其他内存分配函数所获得的内存块都来自一个称为**堆**（**heap**）的存储池。调用这些函数经常会耗尽堆，或者要求大的内存块也可能耗尽堆，这会导致函数返回空指针。

更糟的是，程序可能分配了内存块，然后又丢失了这些块的追踪路径，因而浪费了空间。请思考下面的例子：

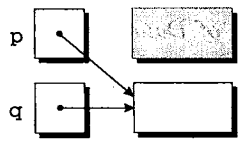
```
p = malloc(...)
q = malloc(...)
p = q;
```

在执行完前两条语句后，p指向了一个内存块，而q指向了另一个内存块：



在把q赋值给p之后，两个指针现在都指向了第二个内存块：

www.TopSage.com



368

因为没有指针指向第一个内存块（图上阴影部分），所以再也不能使用此内存块了。

对程序而言不再访问到的内存块被称为是**垃圾**（garbage）。在后边留有垃圾的程序有**内存泄漏**（memroy leak）。一些语言提供**垃圾收集器**（garbage collector）用于垃圾的自动定位和回收，但是C语言不提供。相反，每个C程序负责回收各自的垃圾，方法是调用free函数来释放不需要的内存。

17.4.1 free 函数

free函数在<stdlib.h>中有下列原型：

```
void free(void *ptr);
```

使用free函数很容易，只是简单地把指向不再需要内存块的指针传递给free函数就可以了：

```
p = malloc(...)
q = malloc(...)
free(p);
p = q;
```

调用free函数来释放p所指向的内存块。然后会把这个释放的内存返回给堆，也就是使此内存块可以被后续的malloc函数或其他内存分配函数的调用可以复用。

free函数的实际参数必须是指针，而且此指针一定是被先前内存分配函数返回的。调用不带其他实际参数（比如，指向变量或数组元素的指针）的free函数可能导致不可预测的行为。

17.4.2 “悬空指针”问题

虽然free函数允许收回不再需要的内存，但是使用此函数会导致一个新的问题：**悬空指针**（dangling pointer）。调用free(p)函数会释放p指向的内存块，但是不会改变p本身。如果忘记了p不再指向有效内存块，混乱可能随即而来：

```
char *p = malloc(4);
...
free(p);
...
strcpy(p, "abc");     /*** WRONG ***/
```

修改p指向的内存是严重的错误，因为程序不再对此内存有任何控制权了。

369

试图修改释放掉的内存块会有程序崩溃等损失惨重的后果。

悬空指针是很难发现的，因为几个指针可能指向相同的内存块。在释放内存块时，全部的指针都会留有悬空。

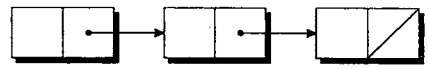
17.5 链表

动态存储分配对建立表、树、图和其他链接数据结构是特别有用的。本节将会介绍链表，而对其他链接数据结构的讨论超出了本书的范畴。为了获取更多的信息，可以参考Horowitz、

www.TopSage.com

Sahni和Anderson-Freed的*Fundamentals of Data Structure in C*这样的书（New York：Computer Science Press，1993）。

链表（Linked List）是由一连串的结构（称为**结点**）组成的，其中每个结点都包含指向下一个链中结点的指针：



表中的最后一个结点包含一个空指针，图上用斜线表示出来。

在前面几章中我们已经知道，无论何时需要存储数据项的集合时都可以使用数组，而现在链表为我们提供了另外一种选择。链表比数组更灵活，我们可以很容易地在链表中插入和删除结点，也就是说允许链表根据需要扩大和缩小。另一方面，也失去了“随机访问”数组的能力。我们可以用相同的时间访问到数组内的任何元素，而访问链表中的结点用时不同。如果结点距离链表的开始处很近，那么访问到它会很快。反之，若结点靠近链表结尾处，访问到它就很慢。

本节会描述在C语言中建立链表的方法，还将说明如何对链表执行几个常见的操作，即在链表开始处插入结点、搜索结点和删除结点。

17.5.1　声明结点类型

为了建立链表，第一件事就是需要一个表示表中单个结点的结构。为了简化，先假设结点只包含一个整数（即结点的数据）和指向表中下一个结点的指针。下面是结点结构的描述：

```
struct node {
  int value;              /* data stored in the node     * /
  struct node *next ;     /* pointer to the next node    * /
);
```

370 注意，成员next具有struct node*类型，这就意味着它能存储一个指向node型结构的指针。顺便说一下，关于名字node没有任何特殊含义，只是一个普通的结构标记。

node型结构的一个方面受到了特别的关注。正如16.2节说明的那样，通常可以选择使用标记或者用typedef来定义一种特殊的结构类型的名字。但是，在结构有一个Q&A指向相同结构类型的成员时，就像node做的那样，要求使用结构标记。没有node标记，就没有办法声明next的类型。

现在有了声明好的node型结构，就需要有跟踪表开始位置的方法。换句话说，需要变量始终指向表中的第一个结点。这里给此变量命名为first：

```
struct node *first = NULL;
```

把first初始化为NULL就说明链表初始为空。

17.5.2　创建结点

在构建链表时，需要逐个创建结点，并且把生成的每个结点加入到链表中。创建结点包括3个步骤：

(1) 为结点分配内存单元；
(2) 把数据存储到结点中；
(3) 把结点插入到链表中。

这里将集中介绍前两个步骤。

为了创建结点，需要一个变量临时指向该结点，并且直到该结点插入链表中为止。把此变量设为new_node：

```
strcut node *new_node;
```

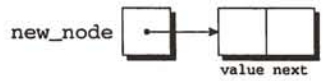
www.TopSage.com

TopSage.com

可以使用malloc函数为新结点分配内存空间，并且把返回值保存在new_node中：

```
new_node = malloc(sizeof(struct node));
```

现在new_node指向了一个内存块，且此内存块正好放下一个node型结构：



小心sizeof给出是分配的类型的名字，而不是指向此类型的指针的名字：

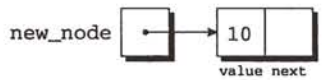
```
new_node = malloc(sizeof(new_node));    /*** WRONG ***/
```

371

程序会始终编译，但是malloc函数将只为指向node型结构的指针分配足够的内存单元，而不是为结构本身分配。当程序试图把数据存储到new_node可能指向的结点中时，结果是稍后可能会崩溃。

接下来，将把数据存储到新结点的成员value中：

```
(*new_node).value = 10;
```



为了访问到结点的成员value，可以采用间接寻址运算符*（为了引用new_node指向的结构），然后选择运算符.（为了选择此结构内的一个成员）。在*new_node周围的圆括号是强制要求的，因为运算符.的优先级高于运算符*（运算符表）（➤见附录B）。

17.5.3 ->运算符

在继续介绍往链表中插入新结点之前，先来讨论一种有用的捷径。利用指针访问结构中的成员是如此的普遍，以致C语言针对此目的还提供了一种特殊的运算符，此运算符被称为**右箭头选择**（right arrow selection），它是由一个减号－跟着一个>组成的。利用运算符->可以编写语句

```
new_node->value = 10;
```

来代替语句

```
(*new_node).value = 10;
```

运算符->是运算符*和运算符.的组合，即为了定位它指向的结构，先对new_node间接寻址，然后再选择结构的成员value。

由于运算符->产生左值（➤4.2.2节），所以可以在任何允许普通变量的地方使用它。恰好已经看到一个new_node->value出现在赋值运算的左侧的例子。这就像出现在scanf调用中那样容易：

```
scanf("%d", &new_node->value);
```

注意，即使new_node是一个指针，运算符&始终也是需要的。没有运算符&，就会把new_node->value的值传递给scanf函数，而这个值是int类型。

372

17.5.4 在链表的开始处插入结点

现在准备往链表中插入新结点了。链表的好处之一就是可以在表中的任何位置添加结点：在开始处、在结尾处或者中间的任何位置。然而，链表的开始处是最容易插入结点的地方，所以这里集中讨论这种情况。

如果new_node正指向要插入的结点，并且first正指向链表中的首结点，那么为了把结点插入链表将需要两条语句。首先，为了指向先前在链表开始处的结点，将要修改结点的成员next：

```
next_node->next = first;
```

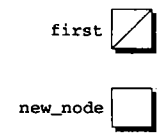
www.TopSage.com

接下来，要使first指向新结点：

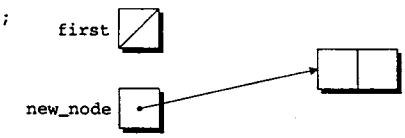
```
first = new_node;
```

如果在插入结点时链表为空，那么这些语句是否还能工作呢？幸运的是，可以。为了确信这是真的，一起来跟踪一下在空链表中插入两个结点的过程。首先，将插入的结点含有数10，跟着要插入的结点还有数20。在下图中空指针用斜线表示。

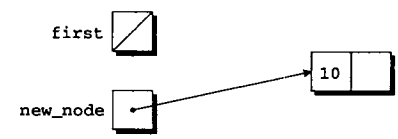
```
first = NULL;
```



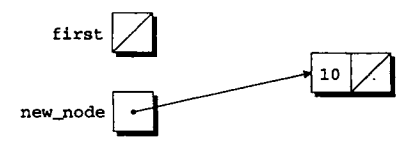

```
new_node = malloc(sizeof(struct node));
```



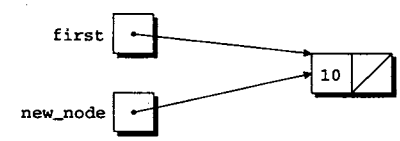

```
new_node->value = 10;
```




```
new_node->next = first;
```

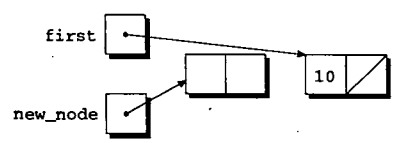



```
first = new_node;
```

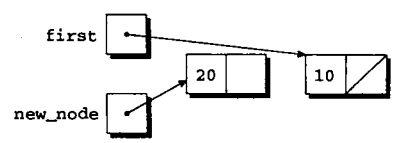


373

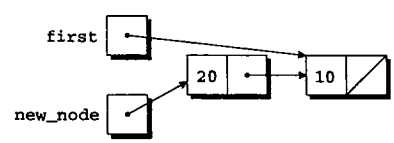
```
new_node = malloc(sizeof(struct node));
```



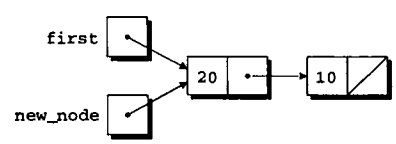

```
new_node->value = 20;
```




```
new_node->next = first;
```




```
first = new_node;
```



www.TopSage.com

往链表中插入结点是经常用到的操作，所以希望为此目的编写一个函数。现在把此函数命名为add_to_list。此函数有两个形式参数：list（指向旧链表中首结点的指针）和n（存储在新结点中的整数）。

```
struct node *add_to_list(struct node *list, int n)
{
  struct node *new_node;

  new_node = malloc(size,of(struct node));
  if (new node == NULL) {

    printf("Error: malloc failed in add_to list\n");
    exit(EXIT_FAILuRE);
  }
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
```

注意，add_to_list函数不会修改指针list，而是返回指向新产生的结点的指针（现在位于链表的开始处）。当调用add_to_list函数时，需要把它的返回值存储到first中：

```
first = add_to_list(first, 10);
first = add_to_list(first, 20);
```

上述语句为first指向的链表增加了含有10和20的结点。用add_to_list函数直接更新first，而不是为first返回新的值，这样做是个技巧。17.6节将回到这个问题。 374

下列函数用add_to_list来创建一个含有用户录入数的链表：

```
struct node *read_numbers(void)
{
  struct node * first = NULL;
  int n;

  printf("Enter a series of integers (0 to terminate):");
  for (;;) {
    scanf("%d", &n);
    if (n == 0)
      return first;
    first = add_to_list(first, n);
  }
}
```

链表内的数将会发生顺序倒置，因为first始终指向包含最后录入数的结点。

17.5.5 搜索链表

一旦创建了链表，可能就需要为某个特殊的数据段而搜索链表。虽然while循环可以用于搜索链表，但是for语句却常常是首选。在编写含有计数操作的循环时，我们习惯使用for语句，但是for语句的灵活性使它也适合其他工作，包括对链表的操作。下面是一种搜索链表的习惯方法，使用了指针变量p来跟踪“当前”结点：

[惯用法]

```
for (p = first; p != NULL; p = p->next)
  ...
```

赋值表达式p = p->next使指针p从一个结点移动到下一个结点。当编写遍历链表的循环时，在C语言中总是采用这种形式的赋值表达式。

现在编写名为search_list函数，此函数为找到整数n而搜索链表（由形式参数list指向）。如果找到n，那么search_list函数将返回指向含有n的结点的指针；否则，它会返回空指针。search_list函数的第一种形式依赖于惯用的“链表搜索”惯用法：

```
struct node *search_list(struct node *list, int n)
```

www.TopSage.com

大家网 TopSage.com

```
{
  struct node *p;

  for (p = list; p != NULL; p = p->next)
      if  (p->value == n)
        return p;
  return NULL;
}
```

375

当然，还有许多其他方法可以编写search_list函数。其中一种替换方式是除去变量p，而用list自身来代替进行当前结点的跟踪：

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL; list = list->next)
    if (list->value == n)
      return list;
  return NULL;
}
```

因为list是原始链表指针的副本，所以在函数内改变它不会有任何损害。

另一种替换方法是把判定list->value == n和判定list != NULL合并起来：

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL && list->value != n;
         list = list->next)
    ;
  return list;
}
```

因为到达链表末尾处list会为NULL，所以即使找不到n，返回的list也是正确的。如果使用while语句，那么search_list函数的此种形式可能会更加清楚：

```
struct node *search_list(struct node *list, int n)
{
  while (list != NULL && list->value != n)
    list = list->next;
  return list;
}
```

17.5.6 从链表中删除结点

把数据存储到链表中一个很大的好处就是可以轻松删除不需要的结点。就像产生结点一样，删除结点也包含3个步骤：

(1) 定位要删除的结点；

(2) 改变前一个结点，从而使它“绕过”删除结点；

(3) 调用free函数从而收回删除结点占用的内存空间。

第1步看起来比较困难。如果按照显而易见的方式搜索链表，那么将在指针指向要删除的结点时终止搜索。但是，这样做就不能执行第2步了，因为第2步要求改变前一个结点。

针对这个问题有各种不同的解决办法。这里将使用“追踪指针”的方法：在第1步搜索链表

376

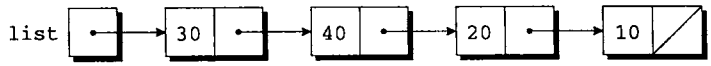
时，将保留一个指向前一个结点的指针（prev），还有指向当前结点的指针（cur）。如果list指向搜索的链表，并且n是要删除的整数，那么下列循环就可以实现第1步：

```
for (cur = list, prev = NULL;
     cur  != NULL && cur->value != n;
     prev = cur, cur = cur->next)
  ;
```

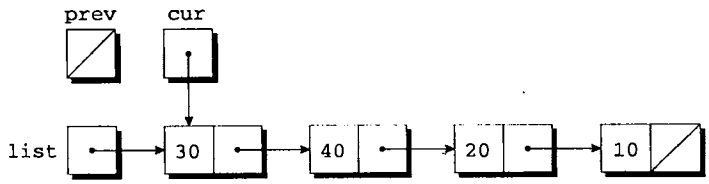
这里我们看到了C语言中for语句的威力。这是个很奇异的示例，它采用了空循环体和逗号运算符的丰富应用，执行的全部操作都是为了搜索到n。当循环终止时，cur指向要删除的结点，而prev指向前一个结点（如果这里有的话）。

www.TopSage.com

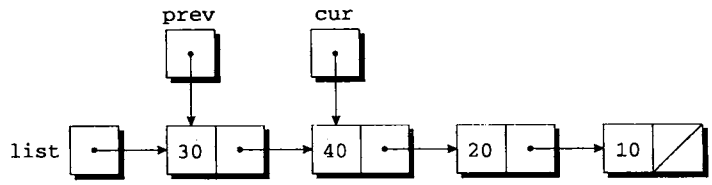
为了看清楚这个循环的工作过程，现在假设list指向依次含有30、40、20和10的链表：



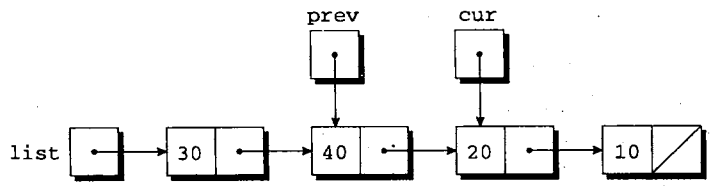
假设n为20，那么目标就是删除此链表中的第3个结点。在执行完cur = list, prev = NULL后，cur指向了链表中的第1个结点：



因为cur正指向一个结点，且此结点不含有20，所以判断表达式cur != NULL && cur->value != n为真。在执行完prev = cur, cur = cur->next后，开始看到指针prev跟踪在指针cur的后边：



再次判断条件表达式cur != NULL && cur->value != n为真，所以再次执行prev = cur, cur = cur->next：



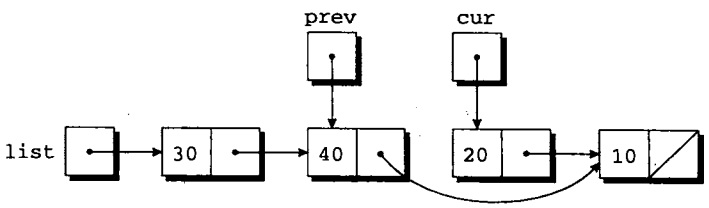
377

因为cur此时指向了含有20的结点，所以条件表达式cur != NULL && cur->value != n为假，从而循环终止。

接下来，将根据步骤2的要求执行绕过操作。语句

```
prev->next = cur->next;
```

使前一个结点中的指针指向了当前结点后面的结点：



现在准备完成步骤3，即释放当前结点占用的内存：

```
free(cur);
```

下列函数所使用的策略就是刚刚概述的操作。在给定链表和整数n时，delete_from_list函数就会删除含有n的第一个结点。如果没有含有n的结点，那么函数什么也不做。无论上述哪

www.TopSage.com

种情况，函数都返回指向链表的指针。

```
struct node *delete_from_list(struct node *list, int n)
{

  struct node *cur, *prev;

  for (cur = list, prev = NULL;
       cur != NULL && cur->value != n;
       prev = cur,  cur = cur->next)
    ;

  if (cur == NULL)
    return list;                  /* n was not found */
  if (prev :: NULL)
    list = list->next;            /* n is in the first node */
  else
    prev->next = cur->next;       /* n is in some other node */
  free (cur);
  return list;
}
```

删除链表中的首结点是一种特殊情况。判定表达式prev == NULL会检查这种情况，这需要一种不同的绕过步骤。

17.5.7　链表排序

378 当链表的结点保持顺序时，我们把对结点内存储的数据进行排序称为是链表的**排序**。对有序链表的操作和对无序链表的操作相似。虽然往有序列表中插入结点会更困难一些（不再始终把结点放置在链表的开始处），但是搜索会更快（在到达期望结点定位的地方之后，可以停止查看操作）。下面一节中的程序既说明了插入结点增加了难度，也说明了搜索更加快速。

17.5.8　程序：维护零件数据库（改进版）

下面重做16.3节的零件数据库程序，这次把数据库存储在链表中。用链表代替数组有两个主要的好处：（1）不需要事先限制数据库的大小，数据库可以扩大到没有更多内存空间存储零件为止；（2）可以很容易保持用零件编号排序的数据库，当往数据库中添加新零件时，只是简单把它插入链表中的适当位置就可以了。在原来的程序中，数据库不能排序。

在新程序中，结构part将包含一个额外的成员（指向链表中下一个结点的指针），而且变量inventory是指向链表首结点的指针：

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};

struct part *inventory = NULL;    /* points to first part */
```

新程序中的大多数函数非常类似于它们在原始程序中的副本。然而，函数find_part和函数insert变得更加复杂了，因为把结点保留在用零件编号排序的链表inventory中。

在原来的程序中，函数find_part返回数组inventory的索引。而在新程序中，函数find_part返回指针，此指针指向的结点含有需要的零件编号。如果没有找到零件编号，函数find_part会返回空指针。既然链表inventory是根据零件编号排序的，新版本的函数find_part就可以节约时间，方法是在找到含有特定零件编号的结点时停止搜索，其中此特定零件编号是大于或等于需要的零件编号的。函数find_part的搜索循环将有如下形式：

www.TopSage.com

TopSage.com

```
for (p = inventory;
     p != NULL && number > p->number;
     p = p->next)
     ;
```

当p变为NULL时（说明没有找到零件编号）或者当number > p->number为假时（说明找到的零件编号小于或等于已经存储在结点中的数），循环将终止。在后一种情况下，始终无法知 379
道需要的数组是否真的在链表中，所以将需要另一种判断：

```
if (p != NULL && number == p->number)
  return p;
```

原始版本的函数insert把新零件存储在下一个有效的数组元素中；新版本的函数需要确定新零件在链表中所处的位置，并且把它插入到那个位置。函数insert还要检查零件编号是否已经出现在链表中了。函数insert可以通过使用类似于函数find_part中的一个循环来完成这两项任务：

```
for (cur = inventory, prey = NULL;
     cur  != NULL && new_node->number > cur->number;
     prev= cur, cur = cur->next)
  ;
```

此循环依赖于两个指针：指向当前结点的指针cur和指向前一个结点的指针prev。一旦终止循环，函数insert将检查是否cur不为NULL，而且new_node->number是否等于cur->number。如果条件成立，那么零件的编号已经在链表中了。否则，函数insert将把新结点插入到prev和cur指向的结点之间，所使用的策略类似于删除结点所采用的方法。（即使新零件的编号大于链表中的任何编号，此策略仍然有效。这种情况下，cur将为NULL，而prev将指向链表中的最后一个结点。）

下面是新程序。和原始程序一样，此版本需要16.3节描述的函数read_line。假设realine.h含有此函数的原型。

invent2.c

```
/* Maintains a parts database (linked list version) */

#include <stdio.h>
#include <stdlib.h>
#include "readline.h"

#define NAME_LEN 25

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};

struct part *inventory = NULL;   /* points to first part   */

struct part *find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
/**********************************************************
 * main:  Prompts the user to enter an operation code,     *
 *        then calls a function to perform the requested   *
 *        action. Repeats until the user enters the        *
 *        command 'q'. Prints an error message if the user *
 *        enters an illegal code.                          *
 **********************************************************/
```

380

www.TopSage.com

```
main()
{
  char code;

  for (;;) {
    printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n')   /* skips to end of line     */
      ;
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}

/**********************************************************
 * find_part:  Looks up a part number in the inventory    *
 *             list. Returns a pointer to the node        *
 *             containing the part number; if the part    *
 *             number is not found, returns NULL.         *
 **********************************************************/
struct part *find_part(int number)
{
  struct part *p;

  for (p = inventory;
       p != NULL && number > p->number;
       p = p->next)
    ;
  if (p != NULL && number == p->number)
    return p;
381
  return NULL;
}

/**********************************************************
 * insert: Prompts the user for information about a new   *
 *         part and then inserts the part into the        *
 *         inventory list; the list remains sorted by     *
 *         part number. Prints an error message and       *
 *         returns prematurely if the part already exists *
 *         or space could not be allocated for the part.  *
 **********************************************************/
void insert(void)
{
  struct part *cur, *prev, *new_node;

  new_node = malloc(sizeof(struct part));
  if (new_node == NULL) {
    printf("Database is full; can't add more parts.\n");
    return;
  }

  printf("Enter part number: ");
  scanf("%d", &new_node->number);
```

www.TopSage.com

```
  for (cur = inventory, prev = NULL;
       cur != NULL && new_node->number > cur->number;
       prev = cur, cur = cur->next)
    ;
  if (cur != NULL && new_node->number == cur->number) {
    printf("Part already exists.\n");
    free(new_node);
    return;
  }

  printf("Enter part name: ");
  read_line(new_node->name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &new_node->on_hand);

  new_node->next = cur;
  if (prev == NULL)
    inventory = new_node;
  else
    prev->next = new_node;
}

/**********************************************************
 * search: Prompts the user to enter a part number, then  *
 *         looks up the part in the database. If the part *
 *         exists, prints the name and quantity on hand;  *
 *         if not, prints an error message.               *
 **********************************************************/
void search(void)
{
  int number;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL) {
    printf("Part name: %s\n", p->name);
    printf("Quantity on hand: %d\n", p->on_hand);
  } else
    printf("Part not found.\n");
}

/**********************************************************
 * update:  Prompts the user to enter a part number.      *
 *          Prints an error message if the part doesn't   *
 *          exist; otherwise, prompts the user to enter   *
 *          change in quantity on hand and updates the    *
 *          database.                                     *
 **********************************************************/
void update(void)
{
  int number, change;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    p->on_hand += change;
  } else
```

382

www.TopSage.com

大家网 TopSage.com

```
    printf("Part not found.\n");
}

/*************************************************************
 * print:  Prints a listing of all parts in the database,    *
 *         showing the part number, part name, and           *
 *         quantity on hand. Part numbers will appear in     *
 *         ascending order.                                  *
 *************************************************************/
void print(void)
{
  struct part *p;

  printf("Part Number   Part Name                "
         "Quantity on Hand\n");
  for (p = inventory; p != NULL; p = p->next)
    printf("%7d      %-25s%11d\n", p->number,
           p->name, p->on_hand);
}
```

注意函数insert中free的用法。函数insert在检查零件是否已经存在之前就为零件分配内存空间。如果真是这样，那么函数insert释放内存以便程序不会冒险运行过早越界。

383

17.6　指向指针的指针

在13.7节中，已经遇到过指向指针的指针。在那一节中，使用了元素类型为char *的数组，以及一个指向数组元素的指针，此指针的类型为char**。“指向指针的指针”概念也频繁出现在链式数据结构的内容中。特别是，当函数的实际参数是指针变量时，有时候会希望函数能通过指针指向别处的方式改变此变量。做这项工作就需要用到指向指针的指针。

请思考一下17.5节中的函数add_to_list，此函数用来在链表的开始处插入结点。当调用函数add_to_list时，我们会传递给它指向链表内首结点的指针，然后函数会返回指向新链表的指针：

```
struct node *add_to_list(struct node *list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if(new_node == NULL)  {
    printf("Error: malloc failed in add_to_list\n");
    exit (EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
```

假设修改了函数使它不再返回new_node，而是把new_node赋值给list。换句话说，把return语句从函数add_to_list中移走，同时用下列语句进行替换：

```
list = new_node;
```

可惜的是，这个想法无法实现。假设按照下列方式调用函数add_to_list：

```
add_to_list(first, 10);
```

在调用点，会把first复制给list。（像所有其他实际参数一样，指针进行了按值传递。）函数内的最后一行改变了list的值，使它指向了新的结点。但是，此赋值操作对first没有影响。

可以让函数add_to_list修改first，但是这就要求给函数add_to_list传递一个指向first的指针。下面是此函数的正确形式：

www.TopSage.com

```
void add_to_list(struct node **list, int n)
{
  struct node *new_node;
  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next : *list;
  *list = new_node;
}
```

384

当调用新版本的函数add_to_list时，第一个实际参数将会是first的地址：

```
add_to_list(&first, 10);
```

既然给list赋予了first的地址，那么可以使用*list作为first的别名。特别是，把new_node赋值给*list将会修改first的内容。

17.7 指向函数的指针

到目前为止，已经使用指针指向过各种类型的数据，包括变量、数组元素以及动态分配的内存块。但是C语言没有要求指针只能指向数据，它还允许指针指向函数。指向函数的指针（函数指针）不像人们想象的那样奇怪。毕竟函数占用内存单元，所以每个函数都有地址，就像每个变量都有地址一样。

17.7.1 函数指针作为实际参数

可以以使用数据指针相同的方式使用函数指针。在C语言中把函数指针作为实际参数进行传递是十分普遍的。假设编写了名为integrate的函数用来求a点和b点之间的函数f的积分。我们希望函数integrate通过传递f作为实际参数的方式变得更为通用。为了在C语言中达到这种效果将把f声明为指向函数的指针。假设希望求积分函数具有double型形式参数，并且返回double型的结果，函数integrate的原型如下所示：

```
double integrate(double (*f)(double), double a, double b);
```

在*f周围的圆括号说明f是个指向函数的指针，而不是函数的返回值为指针。把f声明成好像就是函数也是合法的：

```
double integrate(double f(double), double a, double b);
```

从编译器的标准来看，这种原型和前一种形式是完全一样的。

在调用函数integrate时，将把一个函数名作为第一个实际参数。例如，下列调用将计算sin函数（➤23.3.2节）从0到π/2之间的积分：

385

```
result = integrate(sin, 0.0, PI/2);
```

注意，在sin的后边没有圆括号。当函数名后边没跟着圆括号时，C语言编译器会产生指向函数的指针来代替产生函数调用的代码。在此例中，不是在调用函数sin，而是给函数integrate传递了一个指向函数sin的指针。如果这样看上去很混乱的话，可以想想C语言处理数组的过程。如果a是数组的名字，那么a[i]就表示数组的一个元素，而a本身则作为指向数组的指针。类似的方法，如果f是函数，那么C语言把f(x)看成是函数的调用来处理，而f本身则是指向函数的指针。

在integrate函数体内，可以调用f所指向的函数：

```
sum += (*f)(x);
```

www.TopSage.com

*f表示f所指向的函数，而x是函数调用的实际参数。因此，在函数integrate(sin, 0.0, PI/2)执行期间，*f的每次调用实际上都是sin函数的调用。作为(*f)(x)一种替换选择，C语言允许用f(x)来调用f所指向的函数。虽然f(x)看上去更自然一些，但是这里将坚持把(*f)(x)作为f是指向函数的指针而不是函数名的提示。

17.7.2 qsort函数

虽然指向函数的指针看似对日常编程没有什么用处，但是从事实来看这是没有远见的。实际上，C函数库中一些功能最强大的函数要求把函数指针作为实际参数。Q&A其中之一就是函数qsort，此函数的原型可以在<stdlib.h>中找到。函数qsort是给任意数组排序的通用函数。

因为数组的元素可能具有任何类型，甚至是结构或联合，所以必须告诉函数qsort如何确定两个数组元素哪一个“更小”。通过编写**比较函数**可以为函数qsort提供这些信息。当给定两个指向数组元素的指针p和q时，比较函数必须返回一个数。如果*p“小于”*q，那么返回的数为负数。如果*p“等于”*q，那么返回的数为零。如果*p“大于”*q，那么返回的数为正数。这里把“小于”、“等于”和“大于”放在双引号中是因为确定*p和*q如何比较是我们的责任。

函数qsort具有下列原型：

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar) (const void *, const void *));
```

base必须指向数组中的第一个元素。（如果只是对数组的一段区域进行排序，那么要使base指向这段区域的第一个元素。）在一般情况下，base就是数组的名字。nmemb是要排序元素的数量（不一定是数组中元素的数量）。size是每个数组元素的大小，用字节来衡量。compar是指向比较函数的指针。当调用函数qsort时，它会对数组进行升序排列，并且在任何需要比较数
386 组元素时调用比较函数。

Q&A为了对16.3节的数组inventory进行排序，可以采用函数qsort的下列调用方式：

```
qsort(inventory, num_parts, sizeof(struct part),
      compare_parts);
```

请注意，第二个实际参数是num_parts而不是MAX_PARTS。我们不希望对整个数组inventory进行排序，只是对当前存储的区域进行排序。

编写compare_parts函数并不像想象的那么容易，因为函数qsort要求它的形式参数类型为void型。可惜的是不能通过void *型的指针访问到零件编号。我们需要指向结构part的指针。为了解决这个问题，将用compare_parts把p和q赋值给struct part *型的变量，从而把它们转化成为希望的类型。现在compare_parts可以使用新指针访问到p和q指向的结构成员了。假设希望根据编号对零件排序，下面是函数compare_parts可能的形式：

```
int compare_parts(const void *p, const void *q)
{
  struct part *p1 = p;
  struct part *q1 = q;

  if (p1->number < q1->number)
    return -1;
  else if (p1->number == q1->number)
    return 0;
  else
    return 1;
}
```

虽然可以使用此版本的函数compare_parts，但是大多数C程序员愿意编写更加简明的函数。首先，注意到能用强制类型转换表达式替换p1和q1：

```
int compare_parts(const void *p, const void *q)
```

www.TopSage.com

TopSage.com

```
{
  if (((struct part *) p)->number <
      ((struct part *) q)->number)
    return -1;
  else if (((struct part *) p)->number ==
           ((struct part *) q)->number)
    return 0;
  else
    return 1;
}
```

在表达式((struct part *)p)周围的括号是必须的。如果没有这些圆括号，那么编译器会试图把p->number强制转换成struct part*型。

通过移除if语句甚至可以把函数compare_parts变得更短：

387

```
int compare_parts(const void *p, const void *q)
{
  return ((struct part *) p)->number -
         ((struct part *) q)->number;
}
```

如果p有较小的零件编号，那么p的零件编号减去q的零件编号会产生负值。如果两个零件编号相同，则减法结果为零。而如果p的零件编号较大，那么减法的结果是正数。

为了用零件的名字代替零件编号对数组inventory进行排序，可以使用下列写法的函数compare_parts：

```
int compare_parts(const void *p, const void *q)
{
  return strcmp(((struct part*) p)->name,
                ((struct part*) q)->name);
}
```

函数compare_parts全部需要做的就是调用函数strcmp，此函数会方便地返回负的、零或正的结果。

17.7.3 函数指针的其他用途

虽然已经强调函数指针作为实际参数对其他函数而言是无用的，但是并不是都没有好处的。C语言对待指向函数的指针就像对待指向数据的指针一样。我们可以把函数指针存储在变量中，或者用作数组的元素，再或者用作结构或联合的成员，甚至可以编写返回函数指针的函数。

下面例子中的变量就是存储指向函数的指针：

```
void (*pf)(int);
```

pf可以指向任何带有int型实际参数的函数，而且此函数返回void型的值。如果f是这样的一个函数，那么可以用下列方式把pf指向f：

```
pf = f;
```

注意，在f的前面没有取地址符号(&)。因为pf现在指向函数f，所以既可以用下面这种写法调用f：

```
(*pf)(i);
```

也可以用下面这种写法调用：

```
pf(i);
```

元素是函数指针的数组拥有相当广泛的应用。例如，假设正在编写的程序用来显示用户选择格式的命令菜单。可以编写函数实现这些命令，然后把指向这些函数的指针存储在数组中：

388

```
void (*file_cmd[])(void) = { new_cmd,
                             open_cmd,
                             close_cmd,
```

www.TopSage.com

```
    close_all_cmd,
    save_cmd,
    save_as_cmd,
    save_all_cmd,
    print_cmd,
    exit cmd
};
```

如果用户选择命令n，且n是在0到8之间的数，那么可以通过对数组file_cmd进行下标操作从而找到所调用的函数：

```
(*file_cmd[n])(); /* or file_cmd[n](); */
```

当然，通过使用switch语句可以获得类似的效果。然而，在数组中存储函数指针可以有更大的灵活性，因为数组元素可以在程序运行时发生改变。

17.7.4　程序：列三角函数表

下列函数用来显示含有cos函数、sin函数和tan函数（这3个函数都在<math.h>（➤23.3节）中进行了声明）值的表格。程序围绕名为tabulate的函数构建。当给此函数传递函数指针f时，此函数会显示出函数f的值。

tabulate.c

```
/* Tabulates values of trigonometric functions */

#include <math.h>
#include <stdio.h>

void tabulate(double (*f)(double), double first,
              double last, double incr);

main()
{
  double final, increment, initial;

  printf("Enter initial value: ");
  scanf("%lf", &initial);

  printf("Enter final value: ");
  scanf("%lf", &final);

  printf("Enter increment: ");
  scanf("%lf", &increment);

  printf("\n     x       cos(x)\n"
         "   -------    -------\n");
  tabulate(cos, initial, final, increment);

  printf("\n     x       sin(x)\n"
         "   -------    -------\n");
  tabulate(sin, initial, final, increment);

  printf("\n     x       tan(x)\n"
         "   -------    -------\n");
  tabulate(tan, initial, final, increment);

  return 0;
}

void tabulate(double (*f)(double), double first,
              double last, double incr)
{
  double x;
```


www.TopSage.com

```
  int i, num_intervals;

  num_intervals = ceil((last - first) / incr);
  for (i = 0; i <= num_intervals; i++) {
    x = first + i * incr;
    printf("%10.5f %10.5f\n", x, (*f)(x));
  }
}
```

函数tabulate使用了函数ceil（➤23.3.6节），此函数属于标准库函数。当给定double型的实际参数x时，函数ceil会返回大于或等于x的最小整数。

下面是使用tabulate.c程序可能的结果：

```
Enter initial value:  0
Enter final value:  .5
Enter increment:  .1

      x        cos(x)
-------     -------
  0.00000     1.00000
  0.10000     0.99500
  0.20000     0.98007
  0.30000     0.95534
  0.40000     0.92106
  0.50000     0.87758

      x        sin(x)
-------     -------
  0.00000     0.00000
  0.10000     0.09983
  0.20000     0.19867
  0.30000     0.29552
  0.40000     0.38942
  0.50000     0.47943
      x        tan(x)
-------     -------
  0.00000     0.00000
  0.10000     0.10033
  0.20000     0.20271
  0.30000     0.30934
  0.40000     0.42279
  0.50000     0.54630
```

390

问与答

问：宏NULL表示什么？（p.251）

答：NULL实际是表示0。当在要求指针的地方使用0时，会要求C语言编译器把它看成是空指针而不是整数0。提供宏NULL只是为了避免混乱。赋值表达式p = 0;既可以是给数值型变量赋值为0，也可以是给指针变量赋值为空指针。而我们无法简单地说明到底是哪一种。相反，赋值表达式p = NULL;却明确地说明p是指针。

C++ 在C++语言中，更通用的方式是使用0而不是NULL。原因过于技术化了，很难在这里讨论清楚。一些程序员在C程序中也喜欢使用0而不是NULL。

***问：在伴随编译器的头文件中，NULL按照如下所示进行定义：**

```
#define NULL (void *) 0
```

这样把0强制转化为void*型有什么好处吗？

答：这种技巧在标准C中是合法的。它可以帮助编译器检查到空指针的不正确使用。例如，假设试图把NULL赋值给整型变量：

www.TopSage.com

大家网 TopSage.com

```
i = NULL;
```

如果NULL定义为0，那么这个赋值绝对是合法的。但是，如果把NULL定义为(void *)0，那么编译器将提示我们把指针赋值给整型变量违反了标准C的规则。

把NULL定义为(void *)0还有第二点重要的好处。假设调用带有可变长度实际参数列表（➤26.1节）的函数，且把传递的NULL作为了其中一个实际参数。如果NULL定义为0，那么编译器将会传递
391 一个不正确的整数零。（在普通函数调用中，因为编译器从函数的原型可以知道它所希望的是指针，所以NULL可以正常工作。然而，当函数具有可变长度实际参数列表时，编译器不会获得这类信息，它会假设0就是表示整数。）如果NULL定义为(void *)0，那么编译器将会传递空指针。

甚至情况更混乱的是，一些头文件把NULL定义为0L（0的long int型版本）。就像把NULL定义为0一样，这种定义是C语言早期时代的延续，那时的指针和整数彼此兼容。但是，就大多数目的而言，NULL是如何定义的真的不是问题，只是把它想成是空指针的名字就可以了。

问：既然0用来表示空指针，那么我猜想空指针就是字节中各位都为零的地址，对吗？

答：不一定。每个C语言编译器都被允许用不同的方式来表示空指针，而且不是所有编译器都使用零地址的。例如，一些编译器为空指针使用不存在的内存地址。硬件会检查出这种试图通过空指针访问内存的方式。

我们不关心如何在计算机内存储空指针。这是编译器专家关注的细节。重要的是，当在指针环境中使用0时，编译器会把它转换为适当的内部形式。

问：把NULL用作空字符，这是否可以接受？

答：绝对不行。NULL是用来表示空指针的宏，不是空字符。把NULL用作为空字符对一些编译器可以适用，但不是全部都可以的（因为一些编译器把NULL定义为(void *)0）。在任何情况下，把NULL用作非指针的内容都会导致大量的混乱，如果希望给空字符一个名字，可以使用下列定义的宏：

```
#define NUL '\0'
```

问：程序终止时得到这样一条消息“Null pointer assignment”。这是什么意思呢？

答：此消息由一些DOS程序产生，它说明程序使用坏指针（并不一定是空指针）把数据存储到内存中了。可惜的是此消息直到程序终止才显示出来，所以没有线索可以表明是哪条语句导致的错误。消息“Null pointer assignment”可能是因为在scanf函数中丢失&导致的：

```
scanf ("%d", i); /* should have been scanf ("%d", &i); */
```

另一种可能是含有指针的赋值操作对指针未进行初始化或设为空：

```
*p = i;  /* p is uninitialized or null */
```

问：既然在获得消息“Null pointer assignment”时程序好像也可以工作，那么是否可以忽略此消息呢？

答：请重新阅读前一个问题的答案。如果得到消息“Null pointer assignment”，那么程序此时就存在错误。修正错误，或者不修正。虽然程序好像可以工作，但是却无法保证它始终会正确运行。如果使用了
392 未初始化的指针，那么程序有时可以工作，有时则可能失败。而且如果不同的编译器对程序进行了重新编译或者把程序转到其他计算机上，那么程序正常工作的几率几乎为零。

***问：程序如何知道发生了“Null pointer assignment”？**

答：此消息依赖于这样一个事实：数据在小型或中型存储模型中是存储在单个段中，且此段的地址起始为0。编译器会在数据段的开始处留出“空洞”，即初始化为0但是未被程序使用的一小块内存。当程序终止时，它会查看在“空洞”中的任何数据是否是非零的。如果是，那么一定是通过坏指针改变的。

问：强制类型转换malloc或者其他内存分配函数的返回值，是否有什么好处呢？（p.252）

答：虽然一些程序员都这样做，但是不是真的有什么好处。强制类型转换这些函数返回void*型的指针并不是标准C必需的，因为void*型的指针会在赋值操作时自动转换为任何指针类型。对返回值进行强制类型转换的习惯来自于经典C。在经典C中，内存分配函数返回char*型的值，用强制类型转换实现是必要的。

www.TopSage.com

问：函数calloc把内存块初始为“零位”，这是否意味着内存块中的全部数据项都变为零了？（p.255）

答：通常是，但不总是这样的。把整数设置成零位会始终使整数为零。把浮点数设置成零位通常会使数为零，但这是不能保证的，要依赖于浮点数的存储方式。此问题类似于指针的情况，指针各位置为零并不一定是空指针。

***问：我已经知道了结构标记机制是如何允许结构本身包含指针的。但是，如果两个结构都含有指向对方的成员，会怎么样呢？（p.258）**

答：下面是处理这种情况的方法：

```
struct s1;        /* incomplete declaration of s1 */
struct s2 {
  ...
  struct s1; *p;
};
struct s1; {
  ...
  struct s2; *q;
};
```

393

因为没有指明结构s1的成员，所以第一个s1结构的声明是“不完整的”。C语言允许不完整的结构声明，因为完整的声明稍后会出现在相同的作用域内。

问：为什么不把函数qsort简单命名为sort呢？（p.270）

答：函数qsort的名字来源于1962年C.A.R. Hoare出版的快速排序算法。反过来说，即使一些qsort函数的版本采用了快速排序算法， C标准也不要求函数qsort使用快速排序算法。

问：就像下例所示那样，把函数qsort的第一个实际参数强制转换为void*类型，不是必要的吧？（p.270）

```
qsort((void *) inventory, num_parts, sizeof(struct part),
      compare_parts);
```

答：不是必要的。任何类型的指针都可以自动转换为void *类型的。

***问：我打算使用函数qsort对整数数组进行排序，但是在编写比较函数时遇到了问题。编写的秘诀是什么？**

答：下面是可以使用的版本：

```
int compare_ints(const void *p, const void *q)
{
  return*(int *)p-*(int *)q;
}
```

很奇怪吗？表达式(int *)p把p强制转换为int*类型，所以*(int*)p将是p所指向的整数。

***问：我需要对字符串数组进行排序，所以计划只使用函数strcmp作为比较函数。然而，当把它传递给函数qsort时，编译器发出了出错警告。我试图通过把函数strcmp嵌入到比较函数中的方法来解决问题：**

```
int compare_strings(const void *p, const void *p)
{
  return strcmp(p,q);
}
```

现在对程序进行编译，但是函数qsort好像没有对数组进行排序。我做错什么了吗？

答：首先，不能把strcmp本身传递给函数qsort，因为qsort函数要求比较函数带有两个const void *型的形式参数。由于没有把p和q正确地假设为字符串（char *型指针），所以函数compare_strings无法工作。事实上， p和q指向的数组元素含有char*型指针。为了修改函数compare_strings，将把p和q强制转换为char **型的，然后用*运算符来移走间接寻址的一层操作：

394

```
int compare_strings(const void *p, const void *p)
{
  return strcmp(*(char **)p, *(char **)q);
}
```

www.TopSage.com

练习

17.2节

1. 每次调用时都检查函数malloc的返回值是件很烦人的事情。请编写一个名为my_malloc的函数，用来作为malloc函数的“包装器”。当调用函数my_malloc并且要求分配n个字节时，它会转到调用malloc函数，判断malloc函数确实没有返回空指针，然后返回来自malloc的指针。如果malloc返回空指针，那么函数my_malloc显示出错信息并且终止程序。
2. 请编写名为strdup的函数，此函数使用动态存储分配来产生字符串的副本。例如，调用

```
p = strdup(str);
```

将为和str长度相同的字符串分配内存空间，并且把字符串str的内容复制给新字符串，然后返回指向新字符串的指针。如果分配内存失败，那么函数strdup返回空指针。

3. 请编写一个程序把用户录入的一系列单词进行排序，并且显示删除的重复部分。提示：采用指针数组，且每个指针都指向动态分配的字符串。额外加分：使用qsort函数（17.7节）进行排序操作。

17.3节

4. 请修改程序invent.c（16.3节），使其可以对数组inventory进行动态内存分配，并且稍后在填满时再次进行内存分配。初始使用malloc为拥有10个part结构的数组分配足够的内存空间。当数组没有足够的空间给新的零件时，使用realloc函数来使内存数量加倍。在每次数组变满时重复加倍操作步骤。

17.5节

5. 假设下列声明有效：

```
struct point { int x, y; };
struct rectangle { struct point upper_left, lower_right };
struct rectangle *p;
```

假设希望p指向结构rectangle，其中此结构的左上角位于(0，1)的位置上，而右下角位于(1，0)的位置上。请编写一系列语句用来分配这样一个结构，并且像说明的那样进行初始化。

6. 假设f和p的声明如下所示：

```
struct {
  union {
    char a,b;
    int c;
  }d;
  int e[5];
} f, *p = &f;
```

395

那么下列哪些语句是合法的？

(a) `p->b = ' ';`

(b) `p->e[3] = 10;`

(c) `(*p).d.a = '*';`

(d) `p->d->c = 20;`

7. 请修改函数delete_from_list使它只使用一个指针变量而不是两个（即cur和prev）。
8. 假设下列循环删除了链表中的全部结点，并且释放了占用的内存。但是，此循环有错误。请解释错误是什么并且说明如何修正错误。

```
for (p = first; p!= NULL; p = p->next)
  free(p);
```

9. 请修改程序invent2.c，方法是增加允许用户把零件从数据库中删除的e（擦除）命令。
10. 15.2节描述的文件stack.c提供了在栈中进行整数排序的函数。在那一节中，栈是用数组实现的。请修改程序stack.c从而使栈现在可以作为链表来存储。使用单独一个指向链表首结点的指针变量（栈“顶”）来替换变量contents和变量top。在stack.c中编写的函数要使用此指针。删除函数

www.TopSage.com

TopSage.com

is_full，用返回TRUE（如果创建的结点可以获得内存）或FALSE（如果创建的结点无法获得内存）的函数push来代替。

17.6节

11. 请修改函数delete_from_list（17.5节），使函数的第一个实际参数是struct node **类型（即指向链表首结点的指针），并且返回类型是void。在删除了期望的结点后，函数delete_from_list必须修改第一个实际参数，使其指向该链表。

17.7节

12. 请说明下列程序的输出结果，并且说明理由。

```
#include <stdio.h>

int f1(int(*f)(int));
int f2(int i);

main()
{
  printf("Answer: %d\n", f1(f2));
  return 0;
}

int f1(int (*f) (int))
{
  int n = 0;
  while ((*f)(n)) n++;
  return n;
}

int f2(int i)
{
  return i * i +  i-.12;
}
```

396

13. 请编写下列函数。函数sum(g, i, j)的调用应该返回g(i) + …+ g(j)。

```
int sum(int (*f)(int), int start, int end);
```

14. 设a是有100个整数的数组。请编写函数qsort的调用，此调用只对数组a中的后50个元素进行排序。（不需要编写比较函数。）
15. 请修改函数compare_parts使零件根据编号进行降序排列。
16. 请修改程序invent.c（16.3节），使p（显示）命令显示零件之前调用函数qsort对数组inventory进行排序。
17. 请编写一个函数，要求在给定字符串作为实际参数时，此函数搜索下列所示的结构数组寻找匹配的命令名，然后调用和匹配名称相关的函数：

```
struct {
  char *cmd_name;
  void (*cmd_pointer)(void);
} file_cmd[] =
  { {"new",           new_cmd},
    {"open",          open_cmd},
    {"close",         close_cmd},
    {"close all",     close_all_cmd},
    {"save",          save_cmd},
    {"save  as",      save_as_cmd},
    {"save all",      save_all_cmd},
    {"print",         print_cmd},
    {"exi t" ,        exit_cmd}
 );
```

397

www.TopSage.com

第18章 声 明

让一些事情可变很容易，而掌控不变的期限则需要技巧。

声明在C语言编程中起到核心的作用。通过声明变量和函数，可以在检查程序潜在的错误以及把程序翻译成目标代码两方面为编译器提供至关重要的信息。

前几章已经提供了声明的示例，但是没有完整的描述，本章将会弥补这个缺憾。本章会探讨可以用于声明的复杂选项，并且显示变量声明和函数声明之间的几个共同点。此外，本章还为存储期限、作用域以及链接这些重要概念提供了坚实的基础。

18.1节介绍大多数声明格式的语法，这是之前我们一直回避的主题。然后，将集中讨论声明中出现的数据项：存储类型（18.2节）、类型限定符（18.3节）、声明符（18.4节）和初始化式（18.5节）。

了解声明需要一些时间，但它是需要掌握的至关重要的技能。本章可能不是全书中最重要的部分，但是在考虑成为C程序员之前需要熟练掌握它。

18.1 声明的语法

声明为编译器提供有关标识符含义的信息。当编写

```
int i;
```

399 时，是在告诉编译器：名字i表示当前作用域内数据类型为int的变量。声明

```
float f(float);
```

则是在告诉编译器：f是一个返回值为float型的函数，并且此函数有一个实际参数，此参数类型也为float型。

在大多数通用格式中，声明具有下列格式：

[声明的格式] 声明说明符 声明符;

声明说明符（declaration specifier）描述声明的数据项的性质。**声明符**（declarator）给出了数据项的名字，并且可以提供关于数据项性质的额外信息。

声明说明符分为以下3大类：

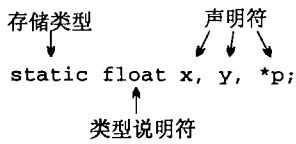
- **存储类型**。存储类型一共有4种：auto、static、extern和register。在声明中最多可以出现一种存储类型。如果表示存储类型，则必须把它放置在声明中的首要位置。
- **类型限定符**。只有两种类型限定符：const和volatile。声明可以指明一个限定符、两个都有或者一个也没有。
- **类型说明符**。关键字void、char、short、int、long、float、double、signed和unsigned全部都是类型说明符。第7章对这些单词组合进行了描述。这些单词出现的顺序不是问题（int unsigned long和long unsigned int完全一样）。类型说明符也包括结构、联合和枚举的说明（例如，struct point{int x, y;}、struct {int x, y}或者struct point）。用typedef创建的类型名也是类型说明符。

www.TopSage.com

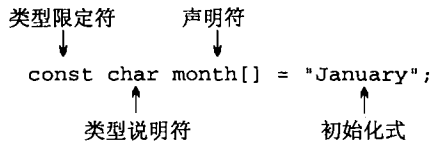
类型限定符和类型说明符必须跟随在存储类型的后边，但是两者的顺序没有严格的限制。由于书写风格，这里会将类型限定符放置在类型说明符的前面。

声明符包括标识符（简单变量的名字）、后边跟随[]的标识符（数组名）、前放置*的标识符（指针名）和后边跟随()的标识符（函数名）。声明符之间用逗号分割。表示变量的声明符后边可以跟随初始化式。

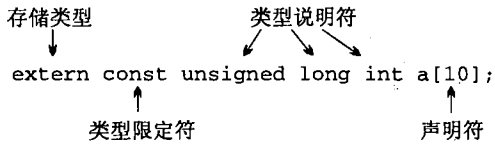
一起看些说明这些规则的例子。下面是一个带有存储类型和三个声明符的声明：



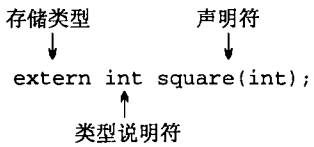
下列声明有类型限定符但是没有存储类型。此外，它还有初始化式：



下列声明既有存储类型也有类型限定符。此外，它还有三种类型说明符，当然它们的顺序并不重要：



和变量声明一样，函数声明也有存储类型、类型限定符和类型说明符。下列声明具有存储类型和类型说明符：



本章余下部分将详细介绍存储类型、类型限定符、声明符和初始化式。

18.2 存储类型

存储类型可以用于变量、较小范围的函数和形式参数的说明。现在将集中讨论变量的存储类型。

对于本节的余下部分，术语**块**（block）既表示函数体（大括号闭合的部分）也表示块语句（包含声明的复合语句）。

18.2.1 变量的特性

C程序中的每个变量都具有3个性质：

- **存储期限**。变量的存储期限决定了为变量预留和释放内存的时间。具有**自动存储期限**的变量在所属块被执行时获得内存单元，并在块终止时释放内存单元，从而会导致变量失去值。具有**静态存储期限**的变量在程序运行期间占有同样的存储单元，也就是可以允许变量无限期地保留它的值。

www.TopSage.com

大家网 TopSage.com

- **作用域**。变量的作用域是指引用变量的那部分程序文本。变量可以有**块作用域**（变量从声明的地方一直到闭合块的末尾都是可见的）或者**文件作用域**（变量从声明的地方一直到闭合文件的末尾都是可见的）。
- **链接**。Q&A变量的链接确定了程序的不同部分可以共享此变量的范围。具有**外部链接**的变量可以被程序中的几个（或许全部）文件共享。具有**内部链接**的变量只能属于单独一个文件，但是此文件中的函数可以共享这个变量。（如果具有相同名字的变量出现在另一个文件中，那么系统会把它作为不同的变量来处理。）**无链接**的变量属于单独一个函数，而且根本不能被共享。

变量的默认存储期限、作用域和链接都依赖于变量声明的位置：

- 在块内部（包括函数体）声明的变量具有自动存储期限、块作用域，并且无链接。
- 在程序的最外层，任意块外部声明的变量具有静态存储期限、文件作用域和外部链接。

下面的例子说明了变量i和变量j的默认性质：

```
int i;    静态存储期限
          文件作用域
          外部链接

void f(void)
{
  int j;  自动存储期限
          块作用域
          无链接
}
```

对许多变量而言，默认的存储期限、作用域和链接是可以符合要求的。当这些性质无法满足要求时，可以改变通过指定明确的存储类型来改变变量的性质：auto、static、extern和register。

18.2.2　auto 存储类型

auto存储类型只对属于块的变量有效。auto类型的变量具有自动存储期限（无需惊讶）、块的作用域，并且无链接。auto存储类型几乎从来不用明确地指明，因为对于在块内部声明的变量，它是默认的。

402

18.2.3　static 存储类型

static存储类型可以用于全部变量，而无需考虑变量声明所在的位置。但是，块外部声明的变量和块内部声明的变量会有不同的效果。当用在块外部时，单词static说明变量具有内部链接。当用在块内部时，static把变量的存储期限从自动的变成了静态的。下面的图说明把变量i和变量j声明为static所产生的效果：

```
static int i;    静态存储期限
                 文件作用域
                 内部链接

void f(void)
{
  static int j;  静态存储期限
                 块作用域
                 无链接
}
```

在用于外部块声明时，static本质上隐藏了它所在声明文件内的变量。只有出现在同一文件中的函数可以看到此变量。在下面的例子中，函数f1和函数f2都可以访问到变量i，但是其他文件中的函数无法做到：

```
static int i;

void f1(void)
```

www.TopSage.com

```
{
  /* has access to i */
}

void f2(void)
{
  /* has access to i */
}
```

static的此种用法可以用来实现一种称为**信息隐藏**（➤19.2节）的技术。

块内声明的static型变量在程序执行期间驻留在同一存储单元内。和每次程序离开闭合块就会丢失值的自动变量不同，static型变量会无限期的保留值。static型变量具有一些有趣的性质：

- 块内的static型变量只在程序执行前进行一次初始化，而auto型变量则会在每次变成有效时进行初始化（当然，需假设它有初始化式）。
- 每次函数进行递归调用时，它都会获得一组新的auto型变量的集合。另一方面，如果函数含有static型变量，那么此递归函数的全部调用都可以共享这个static型变量。 403
- 虽然函数不应该返回指向auto型变量的指针，但是函数返回指向static型变量的指针是没有错误的。

声明函数中的一个变量为static存储类型，这样做允许函数在“隐藏”区域内的调用之间保留信息。隐藏区域是程序其他部分无法访问到的地方。然而，更经常的做法是用static来使程序更加有效。思考下列函数：

```
char digit to hex_char(int digit)
{
  const char hex_chars[16] = "0123456789ABCDEF";
  return hex chars[digit];
}
```

每次调用digit_to_hex_char函数时，都会把字符0123456789ABCDEF复制给数组hex_chars来对其进行初始化。现在，把数组设为static类型的：

```
char digit to hex_char(int digit)
{
  static const char hex chars[16] = "0123456789ABCDEF";
  return hex chars[digit];
}
```

既然static型变量只进行一次初始化，那么这样做就改进了digit_to_hex_char函数的速度。

18.2.4 extern 存储类型

extern存储类型使几个源文件可以共享同一个变量。15.2节介绍了使用extern的基本概念，所以这里的讨论不会太多。回顾过去的内容可以知道，下列声明给编译器提供的信息是，i是int型变量：

```
extern int i;
```

但是这样不会导致编译器为变量i分配存储单元。在C语言的术语中，上述声明不是变量i的定义，它只是提示编译器需要访问定义在别处的变量（可能稍后在同一文件中，或者更经常是在另一个文件中）。变量在程序中可以有多次声明，但只能有一次定义。

对规则而言，变量的extern声明不是定义是一个例外。初始化变量的extern声明可以用作变量的定义。例如，声明

```
extern int i = 0;
```

等效于声明

```
int i = 0;
```

www.TopSage.com

404 这条规则防止用不同方法对初始化变量进行多次extern声明。

extern声明中的变量始终具有静态存储期限。变量的作用域依赖于声明的位置。Q&A如果声明在块内部，那么变量具有块作用域；否则，变量具有文件作用域：

```
extern int i;   静态存储期限 / 文件作用域 / 什么链接?

void f(void)
{
  extern int j;   静态存储期限 / 块作用域 / 什么链接?
}
```

确定exern型变量的链接有一定难度。如果变量在文件中较早的位置（任何函数定义的外部）声明为static，那么它具有内部链接；否则（通常情况下），变量具有外部连接。

18.2.5 register 存储类型

声明变量具有register存储类型就要求编译器把变量存储在寄存器中，而不是像其他变量一样保留在内存中。（寄存器是驻留在计算机CPU中的存储单元。在传统计算机架构中，存储在寄存器中的数据会比存储在普通内存中的数据访问和更新的速度更快。）指明变量的存储类型是register是一种要求，而不是命令。如果选择，编译器可以自由的把register型变量存储在内存中。

register存储类型只对声明在块内的变量有效。register型变量具有和auto型变量一样的存储期限、作用域和链接。但是，register型变量缺乏auto型变量所具有的一种性质：由于寄存器没有地址，所以对register型变量使用取地址运算符&是非法的。这种限制甚至使得编译器选择把变量存储在内存中。

register存储类型最好用于需要频繁进行访问和/或更新的变量。例如，在for语句中的循环控制变量用作register类型就是一个很好的选择：

```
int sum_array(int a[], int n)
{
  register int i;
  int sum = 0;
  for (i = 0; i < n; i++)
    sum += a[i] ;
  return sum;
}
```

405

现在register存储类型已经不像以前那样在C程序员中流行了。今天的编译器比早期的C语言编译器更加复杂了。一些编译器可以自动决定变量保留在寄存器中是否可以获得最大的好处。

18.2.6 函数的存储类型

和变量的声明一样，函数的声明（和定义）可以包含存储类型，但是选项只有extern和static。在函数声明开始处的单词extern说明函数具有外部链接，也就是允许其他文件调用此函数。static说明内部链接，也就是说只能在定义函数的文件内部调用此函数。如果不指明函数的存储类型，那么会假设函数具有外部连接。

思考下面的函数声明：

```
extern int f(int i);
static int g(int i);
int h(int i);
```

函数f具有外部链接，函数g具有内部链接，而函数h（默认情况下）具有外部链接。

www.TopSage.com

TopSage.com

声明函数是extern型的就如同声明变量是auto型一样，两者都没有使用的目的。基于这个原因，本书不在函数声明中使用extern。然而，需要意识到一些程序员广泛地使用extern也是无害的。

另一方面，声明函数是static型的确是十分的有用。事实上，当声明不打算被其他文件调用的任意函数时，建议使用static存储类型。这样做的好处包括有以下两点：

- **更容易维护**。把函数f声明为static存储类型保证在函数定义出现的文件之外函数f都是不可见的。这样的结果是，某些稍后修改程序的人可以知道对函数f的变化不会影响其他文件中的函数。（一个例外是：另一个文件中的函数如果传递了指向函数f的指针可能会受到函数f变化的影响。幸运的是，这种问题很容易通过检查函数f定义的文件来发现，因为传递f的函数一定也定义在此文件中。）
- **减少了“名字空间污染”**。由于声明为static的函数具有内部链接，所以可以在其他文件中重新使用这些函数的名字。虽然可能不会为一些其他目的故意重新使用函数名字，但是在大规模程序中是很难避免这种现象的。带有外部链接的大量函数名可能产生的结果就是C程序员所谓的“名字空间污染”，即在不同文件中的名字意外地发生互相冲突。使用static存储类型可以有效地预防此类问题。

函数的形式参数具有和auto型变量相同的性质：自动存储期限、块作用域和无链接。唯一能用于说明形式参数存储类型的就是register。

406

18.2.7 小结

已经介绍了各种存储类型，现在对已知内容进行一个总结。下面的代码段说明了变量和形式参数声明中包含或者忽略存储类型的所有可能的方法。

```
int a;
extern int b;
static int c;

void f(int d, register int e)
{
  auto int g;
  int h;
  static int i;
  extern int j;
  register int k;
}
```

表18-1说明了上述例子中每个变量和形式参数的性质。

表18-1 变量和形式参数的性质

名 字	存储期限	作用域	链 接
a	静态	文件	外部
b	静态	文件	①
c	静态	文件	内部
d	自动	块	无
e	自动	块	无
g	自动	块	无
h	自动	块	无
i	静态	块	无
j	静态	块	①
k	自动	块	无

① 由于这里没有显示出变量b和j的定义，所以不可能确定这些变量的链接。在大多数情况下，变量会定义在另一个文件中，并且具有外部链接。

www.TopSage.com

在4种存储类型之中，最重要的是extern和static。auto类型没有任何效果，且现代编译器已经使register类型变得废弃无用了。

18.3　类型限定符

C语言中一共有两种类型限定符：const和volatile。因为volatile只用在底层编程中，所以本书将此限定符的讨论推迟到20.3节再介绍。const用来声明一些类似于变量的对象，但这些变量是“只读”的。程序可以访问const型对象的值，但是无法改变它的值。例如，下面这个声明产生了名为n的const型对象，且此对象的值为10：

```
const int n = 10;
```

407 而下列声明产生了名为days_per_month的const型数组。

```
const int days_per_month[] =
  {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

用const来说明对象的值不会改变具有几个好处：

- const是文档格式。声明对象是const类型就是提示任何稍候阅读程序的人，对象的值不会改变。
- 编译器可以检查到程序不会特意地试图改变对象的值。
- 当为某种可能的应用类型编写程序时（特别是嵌入式系统），编译器可以用单词const来确定数据存储到ROM（只读内存）中。

乍一看，好像const违反了前几章中用过的产生常量名的#define指令。然而，实际上#define和const之间有明显的差异：

- 可以用#define指令产生数字常量、字符常量或字符串常量的名字。const可用于产生任何类型的只读对象，这包括常量数组、常量指针、常量结构和常量联合。
- const类型的对象遵循和变量相同的作用域规则，而用#define产生的常量不遵守这些规则。特别是，不能用#define产生具有块作用域的常量。
- 和宏的值不同，可以在调试器中看到const型对象的值。
- 不同于宏Q&A，不可以把const型对象用于常量表达式。例如，由于数组边界必须是常量表达式，所以不能写成下列形式：

```
const int n = 10;
int a[n];           /*** WRONG ***/
```

没有绝对的原则说明何时使用#define和何时使用const。这里建议对表示数字或字符的常量使用#define。另外，还能使用常量作为数组维数，并且在switch语句或其他要求常量表达式的地方使用常量。我们使用const主要是为了保护存储在数组中的常量数据。

18.4　声明符

声明符是由标识符（声明的变量或函数的名字）以及可能在前边的符号*或者跟随在后边的
408 []或()共同组成的。通过把*、[]和()组合在一起，可以创建复杂声明符。

在认识较为复杂的声明符之前，先来复习一下前面了解的声明符的知识。最简单的情况，声明符就是标识符，就如同下面例子中的i：

```
int i;
```

声明符还可以包含符号*、[]和()：

- 用*开头的声明符表示指针：

www.TopSage.com

```
int *p;
```

- 用[]结尾的声明符表示数组：

```
int a[10];
```

如果数组是形式参数，或者数组有初始化式，再或者数组的存储类型为extern，那么方括号内可以为空：

```
extern int a[];
```

因为a是在程序中的某处定义的，所以这里编译器不需要知道数组的长度。(在多维数组中，只有第一维方括号可以为空。)

- 用()结尾的声明符表示函数：

```
int abs(int i);
void swap(int *a, int *b);
int find_largest(int a[], int n);
```

C语言允许在函数声明中忽略形式参数的名字：

```
int abs(int);
void swap(int *, int *);
int find_largest(int [], int);
```

甚至括号内可以为空：

```
int abs();
void swap();
int find_largest();
```

这些声明指明了abs、swap和find_largest的返回类型，但是没有提供有关它们实际参数的信息。括号内置为空不等同于把单词void放置在圆括号内，后者说明没有实际参数。来自于经典C的这种函数声明的空括号形式正在迅速消失。这种格式比标准C的原型形式差，因为空括号形式不允许编译器检查函数调用是否有正确的实际参数。

如果全部的声明符都和前面讲的这些一样简单，那么C语言的编程将一蹴而就。可惜的是，实际程序中的声明符往往组合了符号*、[]和()。我们已经见过这类组合的示例了。我们知道下列语句声明了一个数组，此数组的元素是10个指向整数的指针：

```
int *ap[10];
```

我们还知道下列语句声明了一个函数，此函数有float型的实际参数，并且返回指向float型值的指针。 409

```
float *fp(float);
```

此外，17.7节学过这样一条语句，用来声明一个指向函数的指针，此函数有int型实际参数且返回void型的值：

```
void (*pf)(int);
```

18.4.1 解释复杂声明

到目前为止，在声明符的理解方面还没有遇到太多的麻烦。但是，下面这个声明符是什么呢？

```
int *(*x[10])(void);
```

这个声明符组合了*、[]和()，所以x是指针、数组还是函数并不明显。

幸运的是，无论多么费解，有两条简单的规则可以用来理解任何声明：

- **始终从内往外读声明符。**换句话说，定位用来声明的标识符，并且从此处的声明开始解释。
- **在作选择时，始终先是[]和()后是*。**如果*在标识符的前面，而标识符后边跟着[]，那么标识符表示数组而不是指针。同样地，如果*在标识符的前面，而标识符后边跟着

www.TopSage.com

大家网 TopSage.com

()，那么标识符表示函数而不是指针。(当然，可以一直使用圆括号来使超过*的[]和()的优先级无效。)

首先把这些规则应用于下面这些简单的示例。在下列声明中，

```
int *ap[10];
```

ap是标识符。由于*在ap的前面，且后边跟着[]，而[]优先级高，所以ap是指针数组。在下列声明中，

```
float *fp(float);
```

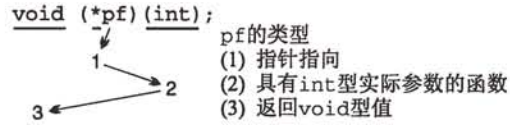
fp是标识符。由于*在标识符的前面，且后边跟着()，而()优先级高，所以fp是返回指针的函数。

下列声明是一个小陷阱：

```
void (*pf)(int);
```

由于*pf在闭合的括号内，所以pf应该是指针。但是(*pf)后边跟着(int)，所以pf必须指向
410 函数，且此函数带有int型的实际参数。单词void表明了此函数的返回类型。

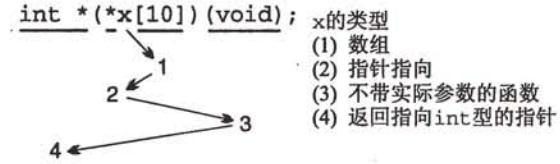
正如最后的示例说明的那样，对复杂声明符的理解经常会遇到从标识符的一边到另一边的曲折：



下面用这种曲折方法来解释早前给出的声明：

```
int *(*x[10])(void);
```

首先，定位的标识符是声明的(x)。在x前有*，而后边又跟着[]。因为[]优先级高于*，所以取右侧（x是数组)。接下来，从左侧找到数组中元素的类型（指针)。再接下来，到右侧找到指针所指向的数据类型(不带实际参数的函数)。最后，回到左侧看每个函数返回的内容(指向int型的指针)。图示过程如下所示：



要想熟练掌握C语言的声明需要花些时间并且要多练习。唯一的好消息是在C语言中有不能声明的特定内容。函数不能返回数组：

```
int f(int)[];      /*** WRONG ***/
```

函数不能返回函数：

```
int g(int)(int);  /*** WRONG ***/
```

函数型的数组也是不可能的：

```
int a[10](int);   /*** WRONG ***/
```

18.4.2　使用类型定义来简化声明

一些程序员利用类型定义来简化复杂的声明。考虑一下前面检查过的x的声明：

```
int *(*x[10])(void);
```

www.TopSage.com

为了使x的类型更容易理解，可以使用下列一系列的类型定义：

411

```
typedef int *Fcn(void);
typedef Fcn *Fcn_ptr;
typedef Fcn_ptr Fcn_ptr_array[10];
Fcn_ptr_array x;
```

反向阅读，发现x具有Fcn_ptr_array类型，Fcn_ptr_array是有Fcn_ptr值的数组，Fcn_ptr是指向Fcn类型的指针，而且Fcn是不带实际参数的函数，且此函数返回指向int型值的指针。

18.5 初始化式

为了方便，C语言允许在声明变量时为它们指定初始值。为了初始化变量，可以在声明符的后边书写符号=，然后在其后再跟上初始化式。（不要把声明中的符号=和赋值运算符相混淆；初始化和赋值不一样。）

在前几章中已经见过各种各样的初始化式了。简单变量的初始化式就是一个与变量类型一样的表达式：

```
int i = 5 /2 ;    /* i is initially 2 */
```

如果类型不匹配，C语言会用和赋值运算相同的规则对初始化式进行类型转换（➤7.5节）：

```
int j = 5.5;      /* converted to 5 */
```

针对指针变量的初始化式，必须是具有和变量相同类型或void*类型的指针表达式：

```
int *p = &i;
```

数组、结构或联合的初始化式通常是一串封闭在大括号内的值：

```
int a[5] = {1, 2, 3, 4, 5};
```

为了全面覆盖声明的范围，现在来看看一些控制初始化式的额外规则：

- 具有静态存储期限的变量的初始化式必须是常量：

412

```
#define FIRST 1
#define LAST 100

static int i = LAST - FIRST + 1;
```

由于LAST和FIRST都是宏，所以编译器可以计算出i（100-1+1=100）的初始值。如果LAST和FIRST是变量，那么初始化式就是非法的。

- 如果变量具有自动存储期限，那么它的初始化式不需要是常量：

```
int f(int n)
{
  int last = n - 1;
  ...
}
```

- 用大括号闭合的数组、结构或联合的初始化式必须只能包含常量表达式，不允许有变量或函数调用：

```
#define N 2
int powers[5] = {1, N, N*N, N*N*N, N*N*N*N};
```

因为N是常量，所以powers的初始化式是合法的。如果N是变量，那么程序将无法进行编译。

- 针对自动类型的结构或联合，它们的初始化式可以是另外一个结构或联合：

```
void g(struct complex c1)
{
```

www.TopSage.com

```
  struct complex c2 = c1;
  ...
}
```

虽然初始化式需要是具有适当类型的表达式，但是它们不需要一定是变量或形式参数名。例如，c2的初始化式可以是*p，这里的p具有struct complex*类型，或f(c1)类型，f是返回complex结构类型的函数。

未初始化的变量

在前面几章中已经暗示了，未初始化变量有未定义的值。但并不总是这样的，变量的初始化值依赖于变量的存储期限：

- 具有自动存储期限的变量没有默认的初始值。不能预测自动变量的初始值，而且每次变量变为有效时可以对值进行改变。
- 具有静态存储期限的变量默认情况下的值为零。用calloc分配的内存是简单的给字节的位置零，而静态变量不同于此，它是基于类型的正确初始化：即整型变量初始化为0，浮点变量初始化为0.0，而指针则初始化为空指针。

作为一种书写风格，最好为静态类型的变量提供初始化式，而不是依赖事实上保证的零。如果程序访问到没有明确初始化的变量，那么稍后阅读程序的人可能不容易确定出是否变量设为零，或者很难确定出变量是否在程序中的某处进行了赋值初始化。

413

问与答

问："作用域"和"链接"之间的差异到底是什么？ (p.280)

答：作用域得益于编译器，而链接得益于链接器。编译器用标识符的作用域来确定在文件定义处提到的标识符是否是合法的。当编译器把源文件翻译成目标代码时，它会注意到有外部链接的名字，并最终把这些名字存储到目标文件内的表中。因此，链接器可以访问到具有外部链接的名字，而内部链接的名字或无链接的名字对链接器而言是不可见的。

问：我无法理解具有块作用域但是外部链接的名字。可否详细解释一下？ (p.282)

答：当然可以。假设某个源文件定义了变量i：

```
int i;
```

现在假设变量i的定义放在了任意函数之外，所以默认情况下它具有外部链接。在另一个文件中，有一个函数f需要访问变量i，所以f的函数体把i声明为extern类型：

```
void f(void)
{
  extern int i;
  ...
}
```

在第一个文件中，变量i具有文件作用域。但是，在函数f内，i是块作用域。如果除函数f以外的其他函数需要访问变量i，那么它们将需要单独进行声明。（或者简单地把变量i的声明移到函数f外，从而使其变成文件作用域。）在整个事情中会混淆的就是每次声明或定义i会建立不同的作用域。有时建立的是文件作用域，有时建立的是块作用域。

***问：为什么不能把const型的对象用于常量表达式呢？constant不就是常量吗？ (p.284)**

答：不一定。const型对象只是保证在它的生命期内保留常量，而不是在程序的整个执行期内。假设是在函数体内声明的const型对象：

```
void f(int n)
{
```

www.TopSage.com

TopSage.com

```
  const int m = n;
}
```

414

当调用函数f时，m将会被初始化为函数f的实际参数的值。然后m将在f返回之前保留常量。当再次调用函数f时，m可能会得到不同的值。这就是出现问题的地方。假设用m来制定数组的长度：

```
void f(int n)
{
  const int m = n;
  int a[m];     /*** WRONG ***/
}
```

那么直到函数f调用之前数组a的长度都是未知的，这显然违反了C语言的规则。C语言规定对编译器而言每个数组的长度都必须是已知的。

但是这还不是const唯一的问题。在块外部声明的const型对象具有外部链接，并且可以在文件之间对其进行共享。如果C语言允许在常量表达式中使用const型对象，那么很容易会自行发现下列情况：

```
extern const int n;
  int a[n];   /*** WRONG ***/
```

可能在其他文件中对n进行了定义，这使编译器无法确定数组a的长度。

C++ C语言在限制const型对象方面没有问题是很令人苦恼的。C++语言通过允许const型对象出现在常量表达式中改进了这种现象，因为C++允许：（1）它是整数，（2）它的初始化式是常量：

```
const int n = 10;
  int a[n];  /* legal in C++, but not in C */
```

默认情况下，C++语言还指定const型对象具有内部链接，这样就使此类型对象的定义可以放置到头文件中。

问：为什么声明符的语法如此古怪？

答：声明试图进行模拟使用。指针声明符的格式为*p，这种格式和稍后将用于p的间接寻址运算符方式相匹配。数组声明符的格式为a[...]，这种格式和数组稍后的下标方式相匹配。函数声明符的格式为f(...)，这种格式和函数调用的语法相匹配。这种原因甚至可以扩展到最复杂的声明符上。请思考一下17.7节中的数组file_cmd，此数组的元素都是指向函数的指针。数组file_cmd的声明符格式为

```
(*file_cmd[])(void)
```

而且调用此种函数的格式为

```
(*file_cmd[n])();
```

其中圆括号、方括号和*都在同样的位置上。

415

练习

18.1节

1. 请指出下列声明的存储类型、类型限定符、类型说明符、声明符和初始化式。
   (a) `static char **lookup(int level);`
   (b) `volatile unsigned long io_flags;`
   (c) `extern char *file name[MAX FILES], path[];`
   (d) `static const char token_buf[]  : "";`

18.2节

2. 用auto、extern、register和/或static来回答下列问题。
   (a) 哪种存储类型可以用于说明能被几个文件共享的变量或函数？
   (b) 假设变量x可以被一个文件中的几个函数共享，但是对其他文件中的函数却是隐藏的。那么变量x

www.TopSage.com

应该被声明为哪种存储类型呢？

(c) 哪些存储类型会影响变量的存储期限？

3. 请列出下列文件中每个变量和形式参数的存储期限、作用域和链接：

```
extern float a;
void f(register double b)
{
  static int c;
  auto char d;
}
```

4. 假设f是下列函数。如果在此之前f从来没有被调用过，那么f(10)的值是多少呢？如果在此之前f已经被调用过5次了，那么f(10)的值又是多少呢？

```
int f(int i)
{
  static int j = 0;
  return i * j++;
}
```

18.3节

5. 假设声明x为const型对象，那么下列关于x的语句哪条是假的呢？

(a) 如果x的类型是int型，那么可以用它来声明数组的长度。

(b) 编译器将查到没有对x进行赋值。

(c) x和变量遵循同样的作用域规则。

(d) x可以是任意类型。

18.4节

6. 请编写下列每个声明指定的x类型的完整描述。

(a) `char (*x[10])(int);`　　(b) `int (*x(int))[5];`

416 (c) `float *(*x(void))(int)[10];`　　(d) `void (*x(int, void (*y) (int)))(int);`

7. 请利用一系列的类型定义来简化练习6中的每个声明。

8. 请为下列变量和函数编写声明：

(a) p是指向函数的指针，并且此函数带有字符型指针作为实际参数，函数返回的也是字符型指针。

(b) f是带有两个实际参数的函数：一个参数是指向结构的指针p，且此结构标记为t；另一参数是长整数n。f返回指向函数的指针，且指向的函数没有实际参数也无返回值。

(c) a是含有4个元素的数组，且每个元素都是指向函数的指针，而这些函数都是没有实际参数且无返回值的。a的元素初始指向的函数名分别是insert、serach、update和print。

(d) b是含有10个元素的数组，且每个元素都是指向函数的指针，而这些函数都有两个int型实际参数且返回标记为t的结构。

9. 在18.4节看到了下列非法的声明：

```
int f(int)[] ;     /* Functions can't return arrays     */
int g(int)(int);   /* Functions can't return functions  */
int a[10](int);    /* Array elements can't be functions */
```

然而，可以通过使用指针获得相似的效果：函数可以返回指向数组第一个元素的指针，也可以返回指向函数的指针，而且数组的元素可以是指向函数的指针。请根据这些描述重新修订上述每个声明。

18.5节

10. 下列哪些声明是合法的？（假设PI是表示3.14159的宏。）

(a) `char c = 65;`　　(b) `static int i = 5, j = i * i;`

(c) `float f = 2 * PI;`　　(d) `double angles[] = {0, PI/2, PI, 3*PI/2};`

11. 下列哪些类型的变量不能进行初始化？

(a) 数组变量。 (b) 枚举变量。 (c) 结构变量。 (d) 联合变量。 (e) 不是上述类型的变量

12. 变量的哪种性质用来确定是否具有默认的初始值？

417 (a) 存储期限。 (b) 作用域。 (c) 链接。 (d) 类型。

www.TopSage.com

第19章 程序设计

只要有模块化就有可能发生误解：隐藏信息意味着需要检查沟通。

实际应用的程序显然比本书中的例子要大，但你可能还没意识到会大多少。更快的CPU和更大的主存已经使我们可以编写一些几年前还完全不可行的程序。图形界面的流行同样大大增加了程序的平均长度。如今，大多数功能完整的程序至少有100 000行代码，百万行级的程序也已经很常见，甚至上千万行的程序都听说过。

Q&A 虽然C语言不是专门用来编写大规模程序的，但许多大规模程序的确是用C语言编写的。这会很复杂，需要很多的耐心和细心，但确实可以做到。本章将讨论那些有助于编写大规模程序的技术，并且会展示哪些C语言的特性（例如static存储类）特别有用。

编写一个大型程序（通常称为“大规模程序设计”）与编写小型程序有很大的不同——就如同写一篇学期论文（当然是双倍的行间距）与写一本500页的书不同一样。一个大规模程序需要更加注意编写风格，因为会有许多人一起工作。需要有仔细的文档，同时还需要对维护进行规划，因为程序可能会多次修改。

尤其是，相对于小型程序，编写一个大规模的程序需要更仔细的设计和更详细的计划。正如Smalltalk程序设计语言的设计者Alan Kay所言，“You can build a doghouse out of anything.”建造一个狗舍可以不需要经过任何特别设计，也可以使用任何原材料。然而对于住人的房屋就不能这么干了，因为这要复杂得多。

在第15章曾经讨论过用C语言编写大规模的程序，但更多地侧重于语言的细节。本章会再次讨论这个主题，并着重讨论好的程序设计所需要的技术。当然，要全面地讨论程序设计会超出了本书的范围。但会尽量简要地涵盖一些在程序设计中的重要观念，以及如何使用它们来编写出更易读、更易于维护的C程序。

19.1节讨论如何将C程序看作是一组相互服务的模块。随后，我们会看到如何使用信息隐藏（19.2节）和抽象数据类型（19.3节）来改进程序模块。19.4节会介绍C++，一种C语言的扩展版本，更好地支持了信息隐藏、抽象数据类型以及大规模程序设计的其他方面。

19.1 模块

当设计一个C程序（或其他任何语言的程序）时，最好将它看作是一些独立的**模块**。模块是一组功能（服务）的集合，其中一些功能可以被程序的其他部分（称为**客户**）使用。每个模块都有一个**接口**来描述所提供的功能。模块的细节，包括这些功能自身的源代码，都包含在模块的**实现**中。

在C语言环境下，这些“功能”就是函数。模块的接口就是头文件，头文件中包含那些可以被程序中其他文件调用的函数的原型。模块的实现就是包含该模块中函数的定义的源文件。

为了解释这个术语，我们来看一下第15章中的计算器程序。这个程序由calc.c文件和一个栈模块组成。calc.c文件包含main函数，而栈模块则存储在stack.h和stack.c中（见后面的图）。文件calc.c是栈模块的客户，文件stack.h是栈模块的接口，stack.c文件是栈模块

www.TopSage.com

大家网 TopSage.com

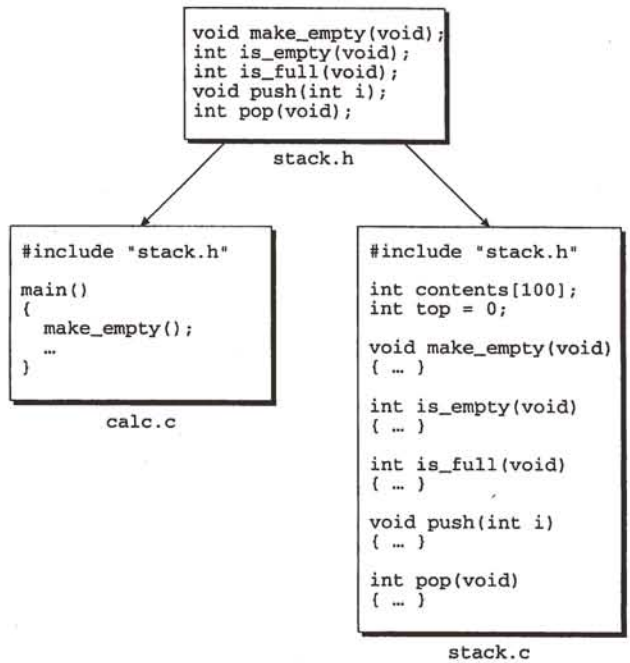
的实现，其中包括操作栈的函数的定义以及组成栈的变量的定义。

C函数库本身就是一些模块的集合。库中每个头文件都会作为一个模块的接口。以stdio.h为例，它就是包含字符串处理函数的模块的接口。

将程序分割成模块有一系列好处：

- **抽象**。如果一些模块是合理设计出来的，我们可以把它们作为**抽象**对待。我们知道模块会做什么，但不需要知道这些功能是如何被实现的细节。因为抽象的存在，使我们不必为了修改部分程序而了解整个程序是如何工作的。同时，通过抽象，我们可以更容易让一个团队的多个程序员共同开发一个程序。一旦对模块的接口达成一致，实现每一个模块的责任可以被分摊到各个成员身上。团队成员可以更大程度上相互独立地工作。

420



- **可复用性**。每一个提供一定功能的模块，都有可能在另一个程序中复用。例如我们的栈模块，就是可复用的。由于很难预测模块在将来是否需要，因此最好将模块设计成可复用的。
- **可维护性**。将程序模块化后，程序中的错误通常只会影响一个模块，因而更容易找到并解决错误。在解决了错误之后，重新编译程序只需要将该模块的实现进行编译即可（然后重新链接整个程序）。更广泛地说为了提高性能或将程序移植到另一个平台上，我们甚至可以替换一个完整的模块的实现。

上面这些问题都很重要，但其中可维护性是最重要的。现实中许多程序会使用许多年，在使用过程中会发现问题，并做一些改进和修改以适应需求的更新。将程序按模块进行设计会使维护更容易。维护一个程序就像维护一辆汽车一样，修理轮胎应该不需要同时检修引擎。

我们可以就近以第16章和第17章中的inventory程序为例。最初的程序（16.3节）将零件记录在一个数组中。假设在程序使用了一段时间后，客户不同意对存储的零件的数量有一个固

421

定的上限。为了满足客户的需求，我们可能会考虑改成链表（正如在17.5节所做的）。为了做这个修改，需要仔细检查整个程序，找到所有依赖于零件存储方式的地方。如果我们一开始就采用不同的方式来设计程序——使用一个独立的模块来处理零件的存储，可能只需要重写这一个模块的实现，而不需要检查整个程序。

www.TopSage.com

一旦我们已经认同了模块化程序设计是正确的方向，接下来的问题就是设计程序的过程中究竟应该定义哪些模块，每个模块应该提供哪些功能，各个模块之间的相互关系是什么？我们现在就来简要地看看这个问题。如果需要了解程序设计的更多信息，可以参考软件工程方面的书籍，像Ghezzi、Jazayeri、和Mandrioli的*Fundamental of Software Engineer*（Englewood Cliffs, N.J.:Prentice-Hall, 1991）一书就是一个很好的选择。本书将采用和该书一样的术语。

19.1.1 内聚性与耦合性

一个好的模块接口并不是随意的一组声明。对于一个认真设计的程序，模块应该具有下面两个性质：

- **高内聚性。**模块中的元素应该相互紧密相关。我们可以认为它们是为了同一目标而相互合作的。高内聚性会使模块更易于使用，同时使程序更容易理解。
- **低耦合性。**模块之间应该尽可能相互独立。低耦合性可以使程序更便于修改，并方便以后复用模块。

我们的计算器的程序有这些性质吗？实现栈的模块是明显具有内聚性的，它的功能是实现与栈相关的操作。整个程序的耦合性也很低，文件`calc.c`依赖于`stack.h`（当然还有`stack.c`依赖于`stack.h`），除此之外就没有其他的明显的依赖关系了。

19.1.2 模块的类型

由于需要具备高内聚性、低耦合性，模块通常会属于下面几类：

- **数据池。**数据池是一些相关的变量或常量的集合。在C语言中，这类模块通常只是一个头文件。从程序设计的角度说，通常不建议将变量放在头文件中。在C语言库中，`<float.h>`（➤23.1节）和`<limits.h>`（➤23.2节）都属于这类模块。
- **库。**库是一组相关函数的集合。例如`<string.h>`就是字符串处理函数库的接口。
- **抽象对象。**一个抽象对象是指对于隐藏的数据结构进行操作的一组函数的集合。（“对象”就是一组数据以及针对这些数据的操作的集合。如果数据是隐藏起来的，那么这个对象是“抽象的”。）
- **抽象数据类型。**将具体数据实现方式隐藏起来的数据类型称为抽象数据类型。作为客户的模块可以使用该类型来声明变量，但不会知道这些变量的具体数据结构。如果客户模块需要对变量进行操作，则必须调用抽象数据类型所提供的函数。抽象数据类型在当今的程序设计中起着非常重要的作用。我们会在19.3节做更详细的讨论。

422

19.2 信息隐藏

一个设计良好的模块经常会对它的客户隐藏一些信息。例如我们的栈模块的使用者就不需要知道究竟栈是用数组实现的，还是用链表或其他方式实现的。这种谨慎地对客户隐藏信息的方法称为**信息隐藏**。信息隐藏有两大优点：

- **安全性。**如果客户不知道栈是如何存储的，就不可能通过栈的内部机制擅自修改栈的数据。它们必须通过模块自身提供的函数来操作栈，而这些函数都是我们编写并测试过的。
- **灵活性。**无论对模块的内部机制进行多大的改动，都不会很复杂。例如，我们可以首先将栈用数组实现，然后又改成是用链表或其他方式。我们当然需要重写这个模块，但是只要模块是按正确的方式设计的，就不需要改变模块的接口。

在C语言中，可以用于强行信息隐藏的主要工具是`static`存储类型（➤18.2.3节）。将一个函数声明成`static`类型可以使函数内部链接，从而阻止其他文件（包括模块的客户）调用这个

www.TopSage.com

函数。将一个带文件作用域的变量声明成static类型可以达到类似的效果，使该变量只能被同一文件中的其他函数访问。

栈模块

为了清楚地看到信息隐藏所带来的好处，下面来看看栈模块的两种实现。一种是使用数组，另一种使用链表。我们假设模块的头文件如下所示：

stack.h

```
#ifndef STACK_H
#define STACK_H

void make_empty(void);
int is_empty(void);
void push(int i);
int pop(void);

#endif
```

423

注意，stack_full的原型并没有放在头文件stack.h中。stack_full函数在使用数组存储栈时是有意义的，但在使用链表来存储栈时就没有意义了。

首先，用数组实现这个栈：

stack1.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

#define STACK_SIZE 100

static int contents[STACK_SIZE];
static int top = 0;

void make_empty(void)
{
  top = 0;
}

int is_empty(void)
{
  return top == 0;
}

static int is_full(void)
{
  return top == STACK_SIZE;
}

void push(int i)
{
  if (is_full()) {
    printf("Error in push: stack is full.\n");
    exit(EXIT_FAILURE);
  }
  contents[top++] = i;
}

int pop(void)
{
  if (is_empty()) {
    printf("Error in pop: stack is empty.\n");
    exit(EXIT_FAILURE);
  }
```

www.TopSage.com

TopSage.com

```
  return contents[--top];
}
```

用于实现栈的变量（contents和top）都被声明成static类型了，因为没有理由让程序的其他部分直接访问它们。这里在stack.c中包含了一个is_full函数，而且也被声明成static类型。因此对于程序的其他部分，它也被隐藏了。

424

在格式上，一些程序员使用宏来指明哪些函数和变量是“公有的”（即可以被程序的其他部分访问），哪些是“私有的”（即仅限该文件内访问）：

```
#define  PUBLIC   /* empty */
#define  PRIVATE  static
```

将static写成PRIVATE是因为static在C语言中有很多的用法，使用PRIVATE可以更清晰地指明这里它是被用来强化信息隐藏的。下面是使用PUBLIC和PRIVATE后程序的样子：

```
PRIVATE int contents[STACK_SIZE];
PRIVATE int top : 0;

PUBLIC void make_empty(void)  { ... }

PUBLIC int is_empty(void)  { ... }

PRIVATE int is_full(void)  { ... }

PUBLIC void push(int i)  { ... }

PUBLIC int pop(void)  { ... }
```

现在我们换成使用链表实现：

stack2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

struct node {
  int data;
  struct node *next;
};

static struct node *top = NULL;

void make_empty(void)
{
  top = NULL;
}

int is_empty(void)
{
  return top == NULL;
}

void push(int i)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error in push: stack is full.\n");
    exit(EXIT_FAILURE);
  }

  new_node->data = i;
  new_node->next = top;
  top = new_node;
}
```

425

```
int pop(void)
{
  struct node *old_top;
  int i;

  if (is_empty()) {
    printf("Error in pop: stack is empty.\n");
    exit(EXIT_FAILURE);
  }

  old_top = top;
  i = top->data;
  top = top->next;
  free(old_top);
  return i;
}
```

这里不再需要is_full函数，因为栈已经没有固定大小了。然而，push函数需要测试malloc函数是否返回空指针。如果malloc返回空指针，则说明已经没有足够的内存来压入下一个元素了。幸运的是，is_full函数在stack1.c中被声明成static，因此其他文件无法调用is_full函数。由于这些文件并不知道有is_full函数，删除这个函数也就不会影响到它们。

我们的栈示例清晰地展示了信息隐藏带来的好处：使用stack1.c还是使用stack2.c来实现模块无关紧要。两个版本具有同样的接口定义，因此可以相互替换，而不会影响程序的其他部分。

19.3　抽象数据类型

对于作为抽象对象的模块，像上一节中的栈模块，有一个缺点：不可能对同一对象有多个实例（在本例中，可以理解为有多个栈）。为了达到这个目的，我们需要进一步创建一个新的类型。

一旦定义了Stack类型，就可以有任意个栈了。下面的程序段显示了如何在同一个程序中
426 有两个栈：

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

main()
{
  Stack s1, s2;

  make_empty(&s1) ;
  make_empty(&s2);
  push(&s1, 1);
  push(&s2, 2);
  if (!is_empty(&s1))
    printf("%d\n", pop(&s1)) ;     /* prints "1" */
  ...
}
```

我们并不知道s1和s2究竟是什么（结构？指针?），但这并不重要。对于栈模块的客户，s1和s2是抽象的对象，它只响应特定的操作（make_empty、is_empty、push以及pop）。

我们来将头文件stack.h改成提供Stack类型的方式。这需要给每个函数增加一个Stack类型（或Stack *）的形式参数：

```
#define STACK_SIZE 100

typedef struct {
  int contents[STACK_SIZE];
  int top;
```

www.TopSage.com

```
) Stack;
void make_empty(Stack *s);
int is_empty(const Stack *s);
void push(Stack *s, int i);
int pop(Stack *s);
```

作为函数make_empty、push和pop参数的栈变量需要定义为指针，因为这些函数会改变栈的内容。is_empty函数的参数并不需要定义为指针，但这里我们仍然使用了指针。给is_empty函数传递一个Stack指针比传递一个Stack值更有效，因为传递值会导致整个数据结构被复制。

封装

遗憾的是，上面的Stack不是抽象数据类型，因为stack.h暴露了Stack类型的具体实现方式，因此无法阻止客户将Stack变量作为结构直接使用：

```
Stack s1;

s1.top =0;
s1.contents[top++] = 1;
```

427

通过提供对top和contents成员的访问，模块的客户可以破坏栈的数据。更糟糕的是，在没有检查是否需要修改客户之前，我们不能改变栈的存储方式。

我们真正需要的是一种阻止客户知道Stack类型的具体实现的方式。遗憾的是，C语言没有设计专门用于**封装**类型的特性。虽然确实有技巧可以达到类似目的，但使用起来相当笨拙，而且依赖性很强。C++实现封装的最佳方法是使用C++语言。C++语言允许我们隐藏数据类型的细节。实际上，C++语言产生的原因之一就是因为C语言不能很好地支持抽象数据类型。

19.4 C++语言

C++对于讨论程序设计的章中，如果没有谈及C++语言的话就是不完整的。C++语言是由AT&T贝尔实验室的Bjarne Stroustrup在20世纪80年代开发出来的C语言的扩展版。在现代程序设计理念上，包括抽象数据类型，C++语言比C提供了更好的支持。（当然，C语言是一种比较老的语言，所以我们并不能责怪它不支持新的程序设计技术。）C++语言中最重要的特性是支持**类**（class）。类使我们可以达到在19.3节中寻求的封装的效果。除此以外，C++语言提供了大量针对大规模程序设计的新特性，包括：

- 支持**面对对象的程序设计**，通过允许从已经存在的类“派生”出新的类，而不是从头编写新的类，从而确保了更高的代码复用率。
- **运算符重载**，可以给传统的C语言的运算符赋予新的含义。运算符重载使我们可以定义新的数据类型，这些新类型甚至与基本类型毫无差别，从而扩展编程语言本身。
- **模板**，可以使我们写出通用的、高度可复用的类和函数。
- **异常处理**，一种统一的方式用来检测并响应错误。

C++语言的一个目标是尽可能保持与C语言的兼容。因此，C++语言中包含了标准C的全部特性。然而，这并不意味着所有C语言的程序都可以在C++的环境下编译。两种语言之间仍然存在一些小的差异，其中一些是由于C++语言增加了更多强制性限制，从而比C语言更加安全。

本节余下部分将提供C++语言的概述。注意，这里只会包含C++语言中的一些新特性，而且介绍也不会很完整。当然，你仍然可以从中了解C++语言是一种怎样的语言。

428

19.4.1 C 语言与 C++语言之间的差异

相对于C语言，C++语言增加的主要特性包括类、重载、派生、虚函数、模板以及异常处理。

www.TopSage.com

大家网 TopSage.com

但在进一步讨论这些新特性之前，我们需要讨论这两种语言之间的一些小差异。

1. 注释

C++语言支持单行注释。单行注释由//开始，在之后的第一个换行符处结束：

```
// This is a comment.
// So is this.
```

单行注释比C语言的注释（C语言的注释仍然是合法的）更安全，因为它们不会意外丢掉注释结束的标记。

2. 标记与类型名

在C++语言中，标记（用于标识特定的结构、联合或枚举的名字）会自动被认为是类型名。因此，我们将

```
typedef struct { double re, im; } Complex;
```

简单写成

```
struct Complex { double re, im; } ;
```

3. 不带参数的函数

在声明或定义一个不带参数的C++函数时，可以不使用void：

```
void draw(void);  // no arguments
void draw();      // no arguments either
```

4. 默认实际参数

C++语言允许函数的实际参数有默认值。例如，下面的函数可以显示任意个数的换行符。如果调用时没有提供任何实际参数，函数会显示一个换行符。

```
void new_line(int n = 1)   // default argument
{
  while (n-- > 0)
   putchar('\n');
}
```

调用new_line函数时，可以提供一个实际参数，也可以不提供实际参数：

```
new_line(3);      // print 3 blank lines
new_line();       // prints 1 blank line by default
```

5. 引用参数

C语言规定实际参数是按值传递的，这使编写那些需要修改作为实际参数提供的变量（除了数组）的函数非常困难。为了回避这个限制，我们只能传递指向该变量的指针。对于一个将
429 两个变量的内容交换的函数，其C语言的实现大致如下：

```
void swap(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

当swap函数被调用时，其参数应该是指向变量的指针：

```
swap(&i, &j);
```

虽然这种方式可以正常工作，但它使用起来并不方便，代码也不易理解，还容易出错。C++语言在这方面做了一些改进，允许实际参数被声明成引用，而不是指针。下面是将a和b声明为引用后swap函数的样子：

```
void swap(int& a,  int& b)    // a and b are references
{
  int temp;
```

www.TopSage.com

```
    temp = a;
    a = b;
    b = temp;
}
```

当调用swap函数时，不需要在实际参数前加&运算符：

```
swap(i, j);
```

在swap的函数体中，a和b被分别理解为i和j的别名。语句temp = a确实会将i的值复制到temp中。语句a = b也确实将j的值复制到i中。同样，b = temp 将temp的值复制给j。

6. 动态存储分配

C程序可以使用函数malloc、calloc、realloc和free来动态分配和释放内存。虽然C++程序仍然可以使用这些函数，但更好的做法是使用new和delete。new和delete是运算符，不是函数。new用来分配空间，delete用来释放分配的空间。new的操作数是一个类型说明符：

```
int *int_ptr, *int_array;
int_ptr = new int;          // allocates memory for an int
int_array = new int[10] ;   // allocates memory for an array
                            //    of ten integers
```

当无法分配所要求的内存时，new会返回空指针。delete需要用一个指针作为它的操作数：

```
delete int_ptr;             // releases memory pointed to by
                            //   int_ptr
delete [] int_array;        // [] required when deallocating
                            //   an array
```

430

19.4.2 类

C语言与C++语言之间最重要的区别在于C++语言支持**类**（class）。（类对于C++的重要性体现在C++语言最初的名字中："C with Classes"。）一个类根本上说就是一个抽象数据类型：一组数据以及操作这些数据的函数。通过编写一个类，可以产生一个新的数据类型。这个新数据类型的功能可以同基本数据类型同样强大。

假设我们需要以分数形式存储数，例如1/4、3/7等。如果我们编写一个类Fraction，就可以很方便地操作这些分数。可以按下面的方式声明Fraction类的变量：

```
Fraction f1, f2, f3;
```

可以使用=运算符复制分数，将分数传递给函数，或编写函数返回分数。

不仅如此，通过使用"运算符重载"的特性，我们可以将C++运算符用作对Fraction对象的操作的名字。通过重载*运算符，可以使它实现分数间的乘法。我们将可以使用下面的代码将f1与f2相乘：

```
f3 = f1 * f2;
```

假设f1的值为1/2，f2的值为2/3，f3会被赋值为1/3。允许重载运算符使它们可以用于分数，这是使Fraction类可以与int和float一样简单易用的重要进步。

类允许我们构造任何需要的数据类型。如果我们需要一种C++语言通常不支持的数值类型（如分数、复杂的数、有无限个数的整数等），我们可以通过设计一个合适的类将这种类型添加到语言中。如果对C++语言的类型的通常行为不满意，可以构造自己的类型。例如，自己定义的Array类中，可以有一个变量对下标进行越界检查。而我们自己的String变量可以根据需要扩展或缩短。类同样适合构造没有作为C++类型提供的复杂数据结构，如队列、集合、栈等。

然而，真正使类最有意义的是它可以用来对现实世界中的对象建模，而不仅仅是作为程序通常使用的数据结构。假设我们在开发一个银行的程序，我们可能会定义诸如Account这样的类。Account类能提供类似deposit和withdraw的操作。类似这样的类会使程序更易读也更易编写，因为这样的操作更接近实际。

www.TopSage.com

使用类的不足之处在于，类的设计和实现比较复杂。这是我们为易用性必须付出的代价，而这也同样是计算机领域近几年内的妥协：随着程序使用起来越来越方便，程序的内部实现也
431 越来越复杂。

19.4.3 类定义

在C++语言中定义一个类非常像在C语言中定义一个结构。在最简单的情况下，类的定义几乎与结构的定义一模一样，只是将struct替换为class：

```
class Fraction {
  int numerator;
  int denominator;
};
```

numerator和denominator称为Fraction类的**数据成员**（data member）。顺便提一下，C++语言并不要求类名以大写字母开始，这只是许多C++程序员所遵循的规范。

一旦一个类被定义了，我们可以使用这个名字来声明变量，声明的方式与使用结构名一样：

```
Fraction f1, f2;
```

（类标记（class tag）在C++语言中可以直接作为类型名使用，因此不需要写成class Fraction。）编译器会构造两个变量f1和f2。每个变量都有自己的成员numerator和denominator。C++语言对f1和f2这样的变量有一个特殊的称呼，它们被称作Fraction类的**实例**（instance）。任何类的实例就是**对象**（object）。

结构的成员可以用运算符.和->访问。然而在类中，成员默认是隐藏的。因此，下面的语句是非法的：

```
f1.numerator = 0;          // illegal
denom = f2.denominator;    // illegal
```

我们称numerator和denominator是Fraction类的**私有**（private）成员。

我们可以通过将成员声明为public（公有）使这些成员可以被访问：

```
class Fraction {
public:
  int numeratior;
  int denominator;
};
```

甚至可以混合使用公有的和私有的成员：

```
class Fraction {
public:
  int numeratior;      // accessible outside the class
private:
  int denominator;     // hidden within the class
};
```

432 注意private的用法：指明denominator是一个私有成员。

19.4.4 成员函数

既然类的私有成员不能从类的外面进行访问，那么怎么修改它们或是检查它们的值呢？对这个问题的回答是十分巧妙的：那些需要访问类的数据成员的函数必须声明在类里面。属于类的函数称为**成员函数**（member function）。

让我们将numerator和denominator设置为Fraction类的私有成员，并给类添加两个成员函数：

```
class Fraction {
public:
  void create(int num, int denom);
```

www.TopSage.com

TopSage.com

```
  void print();
private:
  int numerator;
  int denominator;
};
```

create和print是Fraction类的公有成员，因此它们可以在类以外的地方调用。

成员函数通过对象调用，所使用的运算符“点”和访问结构的成员时所用的是一样的：

```
f1.create(1, 2);  // f1 now stores 1/2
f1.print();       // prints "1/2"
```

无可否认，这看起来确实有点怪，但你会很快习惯的。下面是对create调用的理解：

> “f1是Fraction类的对象，因此我们调用的是Fraction类中的create函数。create函数会将1存到f1的numerator成员中，将2存到f1的denominator成员中。”

下面是print函数调用的含义：

> “f1是Fraction类的对象，因此我们调用的是Fraction类中的print函数。print函数会显示f1的numerator成员，接着显示一个/字符，然后显示f1的denominator成员。”

注意成员函数会知道是哪个对象在调用它，即使对象本身并不作为函数的实际参数。我们可以想象f1是一种放在函数名前面的实际参数，而不是放在实际参数列表中。

成员函数并不一定需要是公有的。在Fraction的例子中，我们可以增加reduce（用于简化分数）作为私有成员函数：

```
class Fraction {
public:
  void create(int num, int denom);
  void print();
private:
  void reduce();
  int numerator;
  int denominator;
};
```

433

在实际使用中，数据成员通常被声明成私有的。成员函数通常被声明成公有的，除非它们仅为了类的内部实现使用。

到目前为止，我们仅仅是声明了create、print和reduce这些函数，那么它们的定义在哪儿？一个可行的做法是稍后在类定义之外定义每一个成员函数。例如，函数create的定义可能会如下所示：

```
void Fraction::create(int num, int denom)
{
  numerator = num;
  denominator = denom;
  reduce();
}
```

注意，函数名前面添加了Fraction::前缀。这是必需的，否则C++编译器会将create作为一个普通的函数，而不是Fraction类的成员。还要注意create是直接访问numerator和denominator的。通常来说，成员函数可以访问类的所有成员，包括公有的和私有的。最后，请注意reduce的调用。它看起来有些奇怪，因为它似乎没有指明简化哪个对象。实际上，当一个成员函数调用另一个成员函数时，后者会默认为是从同一个对象中调用的。换言之，下面的调用：

```
f1.create(1,2);
```

就如同执行了下面的语句：

```
f1.numerator = num;
f1.denominator = denom;
f1.reduce();
```

除了在类之外定义成员函数以外，我们还可以选择将整个函数放在类的定义之中：

```
class Fraction {
public:
  void create(int num, int denom)
    { numerator = hum; denominator = denom; reduce(); }
...
};
```

将成员函数的定义放在类定义之中只在函数的实现非常短小时才可以考虑。

434 现在，来给Fraction类添加乘法函数。首先，我们需要在类的定义中声明成员函数：

```
class Fraction {
public:
  void create(int num, int denom);
  void print();
  Fraction mul(Fraction f);
private:
  void reduce();
  int numerator;
  int denominator;
};
```

接下来，需要编写函数mul的定义：

```
Fraction Fraction::mul(Fraction f)
{
  Fraction result;

  result.numerator = numerator * f.numerator;
  result.denominator = denominator * f.denominator;
  result.reduce();
  return result;
}
```

乍一看，mul函数令人不解：f绝对是参与乘法的分数之一，但另一个分数在哪儿呢？这个问题的答案在于调用mul的方法：

```
f3 = f1.mul(f2);
```

下面是这条语句的含义：

> “f1是一个Fraction对象，因此我们调用的是Fraction类的mul函数。mul函数会将f1的分子与f2的分子相乘，然后将产生的值存放在result的分子中。接着，mul函数会将f1的分母与f2的分母相乘，然后将产生的值存放在result的分母中。接下来，mul会调用reduce来化简result的分数。最后，mul返回result，而reslut将被复制到f3中。”

19.4.5 构造函数

为了确保正确地初始化类的实例，类可以包含一个特殊的函数，称为**构造函数**（constructor）。类还可以提供一个**析构函数**（destructor）——当释放类的实例时进行清理。构造函数和析构函数最方便的（当然也是危险的）地方在于它们通常是被自动调用的，不需要明确的函数调用。换而言之，我们需要给我们的类编写构造函数和析构函数，编译器会安排在需要的时候自动调用它们。

Fraction类已经有一个初始化用的函数create了。我们来将create替换为构造函数。构

造函数看起来像一个与类同名的函数： 435

```
class Fraction {
public:
  Fractioni(int num, int denom)
    { numerator =num; denominator = denom; reduce(); }
  ...
};
```

与其他的成员函数不同，构造函数没有指定的返回类型。注意构造函数被放在类的public成员部分中。

构造函数可以像其他函数一样调用，但它们通常是在声明实例时隐式调用：

```
Fraction f(3, 4);    // declares and initialize f
```

在上面f的声明中，Fraction类的构造函数会被调用，调用时的实际参数是3和4。结果，f的初始值为3/4。

构造函数通常会有默认实际参数：

```
class Fraction {
public:
  Fraction(int hum = 0, int denom = i)
    { numerator =num; denominator = denom; reduce(); }
  ...
};
```

由于num和denom有默认值，Fraction构造函数在调用时可以有两个实际参数：

```
Fraction f(3,4);
```

一个参数：

```
Fraction f(3);    // same as Fraction f(3,1);
```

或没有参数：

```
Fraction f;       // same as Fraction f(0,1);
```

19.4.6 构造函数和动态存储分配

构造函数和析构函数对那些需要动态分配存储空间（使用new和delete运算符）的函数特别有用。例如，假如我们已经受够了普通的C字符串的限制。创建自己的String类可以带来几大好处：

- String对象可以包含任意长度的字符串。在C语言中，字符串的大小受限于数组的长度。
- String对象的长度可以迅速地确定。要得到C语言字符串的长度，需要调用strlen函数，而strlen函数则需要遍历整个字符串来找到标志字符串末尾的空字符。
- 需要时可以给String类添加操作。在C语言中，我们不能方便地修改<string.h>来增加函数。 436

下面是可以用来声明String对象的方式：

```
String s1("abc"), s2("def");
```

初始化后，s1会包含"abc"，s2会包含"def"。当然，这些变量的值都可以随后修改。

由于对字符串的长度没有限制，String对象需要包含一个指针来动态地分配内存（我们将这个指针命名为text）。出于运行速度的考虑，我们还需要一个成员来保存字符串的长度：

```
class String {
  ...
private:
  char *text;     // pointer to string
  int len;        // length of string
};
```

www.TopSage.com

大家网 TopSage.com

接下来，需要一个构造函数来将普通的字符串转换成String对象：

```
class String {
public:
  String(const char *s);   // constructor
  ...
private:
  char *text;
  int len;
};
```

下面是构造函数大概的实现：

```
String::String(const char *s)
{
  len = strlen(s);
  text = new char[len+1];
  strcpy(text, s) ;
}
```

在计算了s所指向的字符串的长度后，构造函数使用new运算符分配足够的内存来复制字符串。最后，构造函数将字符串复制到刚分配的内存中。

19.4.7　析构函数

与动态存储分配有关类有一个很难对付的问题。思考在一个函数内部使用的String对象会发生什么：

```
void f()
{
  String s1("abc");
  ...
}
```

437

当f被调用时，s1对象开始存在。s1的构造函数分配了一个4个字符的数组，并将字符串"abc"复制到数组中。当函数f返回时，s1将不再存在，因为s1使用的是自动存储期限。不幸的是，释放String对象所占的内存时，仅释放了成员text和len使用的内存，而不会释放text所指向的内存。结果，程序会出现内存泄漏。

释放动态分配内存的问题是C++语言提供**析构函数**的原因之一。析构函数会在对象被释放时自动被调用。构造函数和析构函数关系密切。构造函数在对象诞生时对它进行初始化；析构函数在对象消亡时进行清理。如果一个类的构造函数动态分配了内存，析构函数很可能要释放这块内存。

与构造函数一样，析构函数也是一个成员函数。析构函数的名字与类名一致，只是在开头带了一个~（波浪）字符。析构函数没有返回类型，也没有实际参数。下面是添加了析构函数后String类的样子：

```
class String {
public:
  String(const char *s);
  ~String() { delete [] text; }   // destructor
  ...
private:
  char *text;
  int len;
};
```

析构函数~String会释放text所指向的字符数组。

19.4.8　重载

在C++语言中，同一作用域中的两个或更多的函数可以有同样的函数名。当函数以这种方

式重载时，编译器会根据检测函数的实际参数来决定哪个函数被调用。例如，假如在同一作用域中有两个版本的函数f：

```
void f(int);
void f(double);   // overloading
```

下面显示了f调用是如何被转变的：

```
f(1);       // a call of f(int)
f(1.0);     // a call of f(double)
```

重载有一个主要的优点：对于执行相同操作，但操作数的类型不同的函数，可以有相同的函数名。因此，会需要记住更少的函数名。例如，下面的函数同样是计算x的y次方，但实际参数的类型不同：

438

```
int pow(int x, int y) ;
double pow(double x, double y);
```

类中的成员函数可以被重载。（实际上，这可能是函数重载在C++语言中最常见的使用方式了。）例如，通过重载我们可以给String类添加另一个构造函数：

```
class String {
public:
  String(const char *s);
  String() {text = 0; len = 0; }     // overloading
  ~String() { delete [] text; }
  ...
private:
  char *text;
  int len;
};
```

如果你还在考虑为什么text被赋值为0，那么请回忆一下我们讲过0代表空指针。C++程序员通常更喜欢用0而不是NULL，其具体原因就不在这里讨论了。

这个新string的构造函数称为**默认的构造函数**，因为它不带实际参数。它会在声明String对象而没有指定初始值时被调用：

```
String s;   // default constructor is invoked
```

除了函数的重载，C++语言还支持运算符的重载：根据操作数类型的不同，同样的运算符号可以代表不同的操作。运算符重载使我们可以重新定义C++语言的运算符，来使用在类的实例上。这样产生的程序看起来更自然，也更易读。而且类的客户可以使用运算符来执行操作，而不需要调用那些名字难以记住的函数了。

例如，如果将mul函数替换为*运算符，Fraction类就会更好用。做起来很简单，只需要在声明函数时用operator*替换mul就行了：

```
class Fractioin {
public:
  ...
  Fraction operator*(Fraction f);
private:
  ...
};
Fraction Fraction::operator*(Fraction f)
{
  // same body as mul function
}
```

函数的内部实现是一样的，只有名字被改变了。

439

当一个运算符被定义为类的成员时，它其中的一个操作数始终是隐含的。因此，我们所定义的运算符*是一个二元运算符，而不是一元运算符。当我们写如下语句时：

```
f3 = f1 * f2;
```

www.TopSage.com

编译器会与意识到f1是一个Fraction对象，所以会查看Fraction类，找到名字为operator*的函数，然后将语句转换成：

```
f3 = f1.operator*(f2);
```

随后，接下来的操作与执行一般的成员函数一样。

C++语言中的输入/输出

虽然C++程序可以使用<stdio.h>，但C++提供了额外的I/O库。<iostream.h>是新函数库中最重要的头文件。它定义了几个类，包括istream（输入流）和ostream（输出流）。I/O通过对istream和ostream操作来进行。那些从键盘获得输入和在屏幕上显示输出的简单程序可以使用cin对象来进行输入，用cout对象来进行输出。cin是istream类的一个实例，cout是ostream类的一个实例。

istream类和ostream类都与运算符重载关系密切。特别是，C运算符<<和>>（向左或向右移位）（➤20.1.1节）用于绝大多数读和写操作中。istream类重载了>>，使它可以从输入流中读取数据。ostream类重载了<<，使它可以从输出流中读取数据。使用<<和>>进行交互的形式大致如下：

```
cout << "Enter a number: ";
cin >> n;
cout << "The square is ";
cout << n * n;
cout << "\n";
```

第一条语句有这样的含义："cout是一个ostream对象，所以调用的是ostream类的operator<<函数。operator<<函数会将字符串"Enter a number"作为它的实际参数。"

使用新I/O库的一个好处是可以将它们扩展，用来读/写我们编写的类的实例。例如，我们可以使用重载运算符<<来写一个Fraction对象：

```
Fraction f(3, 4);
cout << f;    // prints 3/4
```

能够使用<<来输出分数是我们又向着"使Fraction类和基本类型一样好用"的目标迈进了一步。

440

19.4.9 面向对象编程

虽然专家们还在讨论对"面向对象"的编程语言的确切要求，但对于语言必须具有下述能力的还是达成了广泛的认同：

- **封装**——具有能够定义新的类型以及一组对这个类型的操作的能力，而不会暴露类型的具体实现。（在"面向对象"的环境中，类型的值就是"对象"。）C++语言通过限制对私有数据成员的访问，从而支持了封装。
- **继承**——具有能够定义新的类型，并在新的类型中继承已经存在的类型的属性的能力。C++语言通过**派生类**来实现继承。
- **多态性**——对于同样的操作，对象能够根据它所属于的类采取不同的响应。在C++语言中，虚函数支持了多态性。

我们已经讨论过类，因此下面来看看类的派生和虚函数。

19.4.10 派生

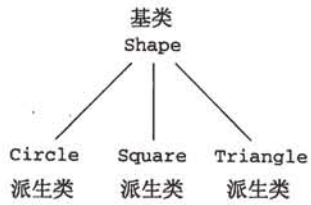
需要一个新类时，C++语言允许从一个已经定义的类**派生**出这个新类，而不需要重新写这

TopSage.com

个新类。例如，一个用来生成图形的程序可能需要叫作Cricle、Square以及Triangle的类。这些类可以都从一个更通用的类Shape派生出来。对于3个类共同的属性，可以只在Shape类中定义一次；同时对于所有形状都适用的操作也可以定义在Shape类中。如果每个形状都有颜色和屏幕上的*x-y*坐标，而且每个形状都可以改变它的位置和颜色，那么Shape类可能如下所示：

```
class Shape {
public:
  void change color(int new_color);
  void move(int x change,  int y change);
  ...
private:
  int x, y;        // coordinates of center
  int color;       // current color
  ...
};
```

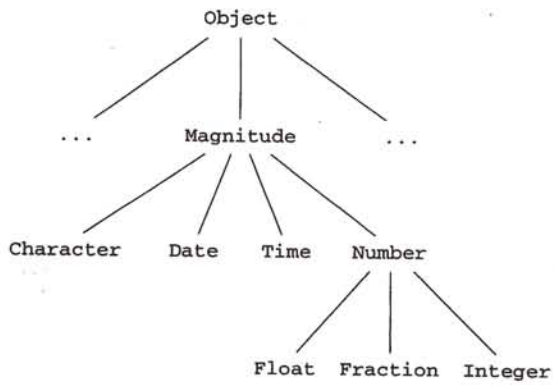
Shape类称为**基类**。Circle、Square和Triangle称为**派生类**。下面的图显示了这些类之间的关系。



派生的一大优势是可以帮助大量地复用代码。假如稍后需要添加一个Pentagon（五角形）的类，我们可以从Shape类派生出来。这比重新写Pentagon类容易得多，因为Pentagon可以从Shape中继承大多数它所需要的属性。我们只需要编写那些Pentagon与Shape不共有的属性代码就可以了。 441

派生也可以使维护更简单。假如Shape类有错误，我们只需要在Shape中修改就可以，不需要改动派生类。同样，如果我们要给所有形状添加一个新属性（或操作），也只要加到Shape类供派生类继承就可以了。

派生经常被用来开发可以扩展的相关类的库。在**Smalltalk**（一种影响了C++语言的面向对象编程语言）中，所有的类都会直接或间接地从一个单一的Object类派生。例如其中Magnitude类从Object类派生出来，然后Character、Date、Time和Number类再从Magnitude类派生，而像Float，Fraction和Integer类则都是从Number类派生出来的：



每个类都会包含它的基类中的所有属性，以及它自身所特有的属性。Magnitude类的对象有一个共同的特性，就是可以使用关系运算符（大于、小于等）进行比较。Number类的对象继

www.TopSage.com

承了这一功能，同时提供额外的支持对算术运算的功能。而算术运算不会提供给其他Magnitude对象。(对日期进行比较是有意义的，但将日期相加就不合理了。)

我们已经学习了派生的概念。下面我们回到Shape的例子说明派生的具体实现。为了指明
442 Circle类从Shape类派生的，在Circle的定义中需要包含一个**派生列表**：

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

Circle**继承**了Shape类的成员(除了构造函数和析构函数)。换而言之，Shape类中的成员也是Circle类的成员。

派生类可以声明额外的在基类中不存在的数据成员和成员函数。例如，Circle类会需要一个数据成员来保存圆的半径。Shape类并没有这样一个radius成员，因此需要添加到Circle中：

```
class Circle: public Shape {
public:
  ...
private:
  int radius;      // radius of circle
};
```

除了radius数据成员，Circle类的对象还会包含成员x，y和color(从Shape继承的)。

当一个类从另一个类派生时，C++语言允许基类的指针指向派生类的实例。例如，Shape*类型的变量可以指向Circle、Square或Triangle对象：

```
Circle c;
Shape *p = &c;    // Shape pointer points to a Circle
```

类似地，C++语言允许Shape*类型的形式参数，与指向Circle、Square或Triangle对象的实际参数匹配。Shape&类型的参数也可以匹配任何Circle、Square或Triangle对象。下面的函数表面上要求一个Shape类型的实际参数，但实际上也可以使用Circle，Square或Triangle类型的对象：

```
void add_to_list(Shape& s)
{
  ...
}
```

add_to_list是一个可以灵活处理各种形状的函数。

19.4.11 虚函数

当类派生与虚函数结合使用时会更有价值。虚函数可以在基类中声明，然后在各个派生类中提供不同的实现。以Shape类为例，每个Shape对象都可以增大它的大小，因此需要一个grow
443 函数：

```
class Shape {
public:
  void grow();
  ...
};
```

但是，grow函数对每个从Shape派生的类都不一样。为了解决这个问题，可以在Shape类中将grow声明成virtual：

```
class Shape {
public:
  virtual void grow();
  ...
};
```

www.TopSage.com

现在，Circle、Square和Triangle类可以各自提供自己的grow版本了。Circle类的版本大概如下所示：

```
class Circle: public Shape {
public:
  void grow() { radius++; }
  ...
private:
  int radius;    // radius of circle
  ...
};
```

大多数情况下，虚函数与普通的成员函数有相同的行为。假设c是一个Circle对象，调用c.grow()会增加c的半径。然而当通过基类的指针（或引用）调用该函数时，虚函数会有特殊的处理：根据指针当前所指向的对象的实际类型，来决定所调用的函数的版本。

假设p是一个指向Shape对象的指针。由于Circle和Square都是从Shape类派生的，p可以指向其中一个类的实例。当使用p调用grow函数时，如果p指向的是一个Circle对象，则调用Circle::grow()，而如果p指向的是一个Square对象，那么Square::grow()会被调用：

```
Shape *p;
Circle c;
Square s;

p = &c;           // p points to a Circle
p->grow() ;       // calls Circle::grow
p = &s;           // p points to a Square
p->grow();        // calls Square::grow
```

注意这里使用->来调用grow函数。通过指向对象的指针调用成员函数时，需要使用->运算符而不是.运算符。

虚函数依赖于一种称为“动态绑定”的技术，这是由于编译器并不总是能判断应该调用函数的哪个版本。（对于普通函数调用，甚至对于重载函数的调用，编译器都可以判断出正确的函数版本。）考虑下面的例子： 444

```
if (...)
  p = &c;
else
  p = &s;
p->grow();    // calls either Circle::grow or Square::grow
```

编译器无法知道应该调用grow的哪个版本，这一点直到程序运行时才可以知道。

动态绑定技术使我们可以构造数据结构来包含不同类的对象（只要这些类都是从同一个基类派生出来的），然后对每个对象执行相同的操作。每一个对象会根据它自己所属的类，采取不同的响应。例如，假设我们通过把Shape对象存储在一个列表中来跟踪当前在屏幕上显示的对象。通过逐个访问每个对象，并调用它的grow函数，我们可以改变它的大小，而不需要知道它具体是什么形状：

```
while (不在列表结尾) {
  让p指向当前形状;
  p->grow();    // calls either Circle::grow, Square::grow,
                // or Triangle::grow
  向前至列表中下一项;
}
```

动态绑定可以简化我们的代码，因为我们将不需要在执行操作前使用switch语句来检测每个对象了。而且我们也可以增加新类或删除旧类，而不必改动操作数据结构的代码。例如，如果我们增加了一个Pentagon类（由Shape类派生），不需要改动扩大每个形状的循环。

www.TopSage.com

大家网 TopSage.com

19.4.12 模板

模板是构造类所使用的“模式”。(C++语言也支持函数模板，不过本书不讨论。)除了对类定义的部分内容未加具体说明以外，模板看起来很像一个普通的类。忽略类定义的部分内容可以使类更通用，也更易复用。

考虑19.3节的Stack类型。如果将它转换成一个C++的类，可能会得到如下定义：

```
class Stack {
public:
  void make_empty();
  int is_empty();
  void push(int);
  int pop();
  ...
};
```

445

遗憾的是，这个Stack对象只能存储整数。如果以后需要一个可以存储其他类型数据（例如float值）的栈时，可以将类定义复制一份，改变类名，将分散各处的int改为float。但构造一个Stack模板才是个更好的办法：

```
template <class T>
class Stack {
public :
  void make_empty ();
  int is_empty();
  void push(T);
  T pop();
  ...
};
```

除了下面这行代码，Stack模板看起来非常像Stack类：

```
template <class T>
```

这行代码指明Stack是一个模板，直到将缺少的类T补充进来才会完整。T（一个“模板实际参数”）只是一个假名，任何名字都可以。注意push现在需要参数是T类型的，同时pop会返回一个T类型的值。Stack的成员函数会如下所示：

```
template <class T>
void Stack<T>::push(T, x)
{
  ...
}
```

模板类在提供了模板参数后才会被“实例化”。虽然写成<class T>，Stack的实际参数不需要一定是个类，任何C++类型都可以。例如，下面是Stack类的3种不同的实例化方式，分别使用了int、float和char作为模板实际参数：

```
Stack<int> int_stack;          // stack of int values
Stack<float> float_stack;      // stack of float values
Stack<char> char_stack;        // stack of char values
```

下面对push的调用解释了如何使用这三种栈：

```
int_stack.push(10);            // pushes i0 onto int_stack
float_stack.push(1.2);         // pushes 1.2 onto float_stack
char_stack.push('a') ;         // pushes  'a'  onto char_stack
```

19.4.13 异常处理

异常是指程序运行时可能产生的情况，通常作为出错的结果。例如：使用Stack时可能产生两种错误：在栈已满时试图向栈中压入数据，或栈为空时试图弹出数据。我们可以使用名字

为StackFull和StackEmpty的异常来表示这两种错误。 446

当错误发生时，函数可以“抛出”一个异常。在Stack类的例子中，push函数和pop函数可以相应地抛出StackFull和StackEmpty异常：

```
void Stack::push(int i)
{
  if (无空间)
throw StackFull();
  ...
}

int Stack::pop()
{
  if (栈空)
throw StackEmpty();
  ...
}
```

有可能发生异常的代码会被放在“try程序块”中。异常会被“处理程序”(“catch程序块”)捕获。在下面的例子中，一个包含push和pop调用的try程序块跟着两个catch程序块。第一个catch程序块处理StackFull异常，显示Error: Stack full；第二个catch程序块处理StackEmpty异常，显示Error: Stack empty。

```
try {
  ...
  s.push(y);
  ...
  z = s.pop();
  ...
}
catch (StackFull) {
  cout << "Error: Stack full\n";
}
catch (StackEmpty) {
  cout << "Error: Stack empty\n";
}
```

在异常处理之后，程序会从最后的catch程序块后面的语句继续执行。

对于一个异常，如果在当前的try程序块末尾没有相应的catch来捕获，C++语言并不会放弃。实际上，它会检查被包围的try程序块来寻找相应的处理者。如果这次查找失败了，则会中止当前函数，并将当前的异常传播给调用函数处理。如果需要会继续传播给再上层的调用者，以次类推。最坏的情况下，异常会一直传播到main函数。如果main函数也无法处理这个异常，整个程序会被终止。异常的这种性质可以帮助避免错误被意外地忽略。 447

问与答

问：本节中提到C语言不是为开发大型程序设计的。UNIX不是大型程序吗？ (p.291)

答：在C语言被设计出来时还不是。在1978年的一篇论文中，Ken Thompson估计UNIX内核大约是10 000行C代码（加上一小部分汇编代码）。UNIX的其他部分也有类似的大小。在另一篇1978年的论文中，Dennis Ritchie和他的同事将PDP-11的C编译器的大小设定为9660行。按现在的标准，这绝对只是小型程序。

问：C语言库中有什么抽象数据类型吗？

答：从技术上说，没有。但有一些很接近，包括FILE类型（定义在<stdio.h>中）。在对一个文件进行操作之前，必须声明一个FILE *类型的变量：

www.TopSage.com

```
FILE *fp;
```

这个fp变量随后会被传递给不同的文件处理函数。

程序员需要把FILE作为一个抽象的类型，在使用时不需要知道FILE具体是怎样的。假设FILE是一个结构类型，但C标准并不保证这一点。实际上，最好不要管FILE值究竟是如何存储的，因为FILE类型的定义对不同的编译器可能（也确实经常）是不一样的。

当然，我们总是可以查看stdio.h找到FILE到底是什么。如果这么做，那么就没什么可以阻止我们编写代码来访问FILE的内部机制。例如，我们可能发现FILE结构中有一个叫bsize（文件的缓冲区大小）的成员：

```
typedef struct {
  ...
  int bsize;      /* buffer size */
  ...
} FILE;
```

一旦我们知道了bsize成员，就无法阻止我们直接访问特定文件的缓冲区大小：

```
printf("buffer size: %d\n", fp->bsize);
```

然而，这样做并不是个好主意，因为其他的C编译器可能将缓冲区大小存在其他名字中，或者是用其他方式跟踪这个值。试图修改bsize的值则是个更糟糕的做法：

448

```
fp->bsize = 1024;
```

这是一件非常危险的事，除非我们知道文件存储的全部细节。即使我们的确知道相关的细节，这样做对于不同的编译器或是同一编译器的更新版本也同样是非常危险的。

问：如果C++语言真的那么好，为什么还有人使用C语言呢？

答：以某种角度来说，这个问题是没有意义的。C++语言包含了C语言的全部特性，所以所有使用C++语言的人就在“使用”C语言。我们来换个角度问这个问题：“如果C++语言真的那么好，为什么不是所有人都使用C++语言呢？”

其一，C++比C复杂得多。由于C++实际上继承了所有C语言的特性，同时增加了大量新特性，C++显然是一个庞大的语言。C++不同功能之间的多种组合也进一步增加了语言的复杂程度。对于编写小程序，C语言更简单而且使用起来与C++不相上下。

C++所提倡的新特性需要编译器做更多的工作。因此，C++程序编译起来会比C程序慢一些。此外，使用C++的新特性会对程序的运行性能带来一些负面影响，这种影响虽然较小，但是可以察觉的。这对于一部分程序来说可能是不可接受的。

虽然C++解决了C语言的一些著名的隐患，但仍有一些没有涉及。当然，C++的新特性也会带来一些新的陷阱，而这些陷阱是C语言没有的。正如Stroustrup自己评价的那样：“C语言使你很容易击中自己的脚。C++使这变得困难了，一旦击中，它会炸飞你整条腿。”

不要忘记C语言存在的时间比C++长许多。虽然经历了几年的改动后，C++已经开始逐渐稳定下来了。但C++编译器仍需要一段时间来达到C编译器已经提供的能力。除此以外，与C++相比，C语言有更多种类的编译器，尤其在对一些不太流行的平台上。

总之，对于“简而达意”的程序，以及需要更广泛的移植性的程序，C语言更适合。对于大型的、功能齐全的程序（包括那些有复杂的图形用户界面的程序）来说，C++更强一些。

练习

19.1节

1. **队列**类似于栈，两者的差异是队列的数据从一端添加，而从另一端按FIFO（先进先出）的方式删除。对于队列的操作可以包括：
   - 向队列的末端加入一个数据。

www.TopSage.com

TopSage.com

- 从队列的开始删除一个数据。
- 返回队列第一个数据（不改变对列）。
- 返回队列的末尾数据（不改变对列）。
- 检查队为是否为空。

以头文件queue.h的形式给队列定义一个接口。

19.2节

2. 修改文件stack2.c，以使用PUBLIC宏和PRIVATE宏。 449
3. (a) 按照练习1中的描述用数组实现一个队列模块。
   (b) 按照练习1中的描述用链表实现一个队列模块。

19.3节

4. (a) 编写一个基于数组的stack类型的实现。
   (b) 使用链表替换数组，重写上面的stack类型。（给出stack.h和stack.c。）
5. (a) 将练习1中的queue.h头文件加以修改，使之定义一个Queue类型。同时修改queue.h头文件中的函数，用Queue（或Queue*）作为形式参数。
   (b) 使用数组实现Queue类型。
   (c) 使用链表实现Queue类型。

19.4节

6. Fraction类需要一个析构函数吗？验证你的答案。
7. 给Fraction类添加重载运算符+、-和/。写出这些运算符的定义以及print函数和reduce函数的实现。
8. 与C语言的printf和scanf相比，C++语言的<<和>>运算符有什么优点？
9. 将练习5的Queue类型改成Queue类。
10. 将练习9的Queue类改成Queue模板。 450

www.TopSage.com

第20章 低级程序设计

当程序要求关注不相干的内容时，所用的编程语言就是低级的。

前面几章中讨论的是C语言中高级的、与机器无关的特性。虽然这些特性对不少程序都够用了，但仍有一些程序需要进行位级别的操作。位操作和其他一些低级运算在编写系统程序（包括编译器和操作系统）、加密程序、图形程序以及其他一些需要高执行速度或高效地使用空间的程序时非常有用。

20.1节介绍C语言的按位运算符。按位运算符提供了对单个位或位域的访问。20.2节介绍如何声明包含位域的结构。最后，20.3节描述如何使用一些普通的C语言特性（类型定义、联合和指针）来帮助编写低级程序。为了方便说明，本节中的所有例子都是16位的，而且可以方便地将这些示例扩展到32位。这里的讨论也不会依赖于特定的操作系统，20.3节的部分内容针对DOS编程除外。

本章中描述的一些技术需要用到数据在内存中如何存储的知识，这对不同的机器和编译器可能会不同。依赖于这些技术很可能会使程序丧失可移植性，因此除非必要，否则最好尽量避免使用它们。如果确实需要，尽量将使用限制在特定的模块中，不要分散在各处。同时，最重要的是确保使用文档记录所做的事！

20.1 按位运算符

C语言提供了6个按位运算符。这些运算符可以用于在整数和字符上进行按位运算。这里先
451 讨论移位运算符。

20.1.1 移位运算符

移位运算符可以改变数的二进制形式，将它的位向左或向右移动。C语言提供了两个移位运算符。参见表20-1。

表20-1 移位运算符

符 号	含 义
<<	左移位
>>	右移位

运算符<<和运算符>>的操作数可以是任意整型或字符型的。对两个操作数都会进行整型提升，返回值的类型是左边操作数提升后的类型。

i<<j的值是将i中的位左移j位后的结果。每次从i的最左端溢出一位，在i的最右端补一个0位。i>>j的值是将i中的位右移j位后的结果。如果i是无符号数或非负值，则需要在i的左端补0。如果i是负值，其结果是由实现定义的。一些实现会在左端补0，其他一些实现会保留符号位而补1。

可移植性技巧 为了更好地保留可移植性，最好仅对无符号数进行移位运算。

www.TopSage.com

下面的例子展示了对数13应用移位运算的效果：

```
unsigned int i, j;

i = 13;       /* i is now 13   (binary 0000000000001101) */
j = i << 2;   /* j is now 52   (binary 0000000000110100) */
j = i >> 2;   /* j is now  3   (binary 0000000000000011) */
```

如上面的例子所示，两个运算符都不会改变它的操作数。如果要对一个变量进行移位，需要使用复合赋值运算符<<=和>>=：

```
i = 13;       /* i is now 13   (binary 0000000000001101) */
i <<= 2;      /* i is now 52   (binary 0000000000110100) */
i >>= 2;      /* i is now 13   (binary 0000000000001101) */
```

移位运算符的优先级比算术运算符的优先级低，因此可能产生意料之外的结果。例如，i<<2+1等同于i<<(2+1)，而不是(i<<2)+1。

452

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

表20-2列出了余下的按位运算符。

表20-2 其他按位运算符

符 号	含 义
~	按位求反
&	按位与
^	按位异或
|	按位或

运算符~是一元运算符，对其操作数会进行整型提升。其他运算符都是二元运算符，对其操作数进行常用的算术转换。

运算符~、&、^和|对操作数的每一位执行布尔运算。~运算符会产生对操作数求反的结果，即将每一个0替换成1，将每一个1替换成0。运算符&对两个操作数相应的位执行逻辑与运算。运算符^和|相似（都是对两个操作数执行逻辑或运算），差异是当两个操作数的位都是1时，^产生0而|产生1。

不要将按位运算符&和|与逻辑运算符&&和||相混淆。Q&A有时候按位运算会得到与逻辑运算相同的结果，但它们绝不等同。

下面的例子说明了运算符~、&、^、|的作用：

```
i = 21;       /* i is now     21  (binary 0000000000010101)  */
j = 56;       /* j is now     56  (binary 0000000000111000)  */
k = ~i;       /* k is now 65514  (binary 1111111111101010)  */
k = i & j;    /* k is now     16  (binary 0000000000010000)  */
k = i ^ j;    /* k is now     45  (binary 0000000000101101)  */
k = i | J;    /* k is now     61  (binary 0000000000111101)  */
```

其中对~i所显示的值是基于unsigned int类型的值占有16位的假设。

对运算符~需要一些额外的说明，因为它可以帮助我们使低级程序可移植性更好。假设我们需要一个整数，它的所有位都为1。最好的方法是使用~0，因为它不会依赖于整数所包含的位的个数。类似地，如果我们需要一个整数，除了最后5位其他的位全都为1，我们可以写成~0x001f。

运算符~、&、^和|有不同的优先级：

大家网 TopSage.com

最高级：~
　　&
　　^

453 最低级：|

因此，可以在表达式中组合使用这些运算符，而不必添加括号。例如，可以写i & ~j|k而不需要写成(i & (~j))|k，同样，可以写i ^ j & ~k而不需要写成i ^ (j & (~k))。当然，仍然可以使用括号来避免混淆。

运算符&、^和|的优先级比关系运算符和判等运算符低。因此，下面的语句不会得到期望的结果：

```
if (status & 0x4000 != 0) ...
```

语句会先计算0x4000 != 0（结果是1），接着判断status & 1是否非0，而不是判断status & 0x4000是否非0。

组合赋值运算符&=、^=和|=分别对应于按位运算符&、^和|：

```
i = 21;     /* i is now 21  (binary 0000000000010101) */
j = 56;     /* j is now 56  (binary 0000000000111000) */
i &= j;     /* i is now 16  (binary 0000000000010000) */
i ^= j;     /* i is now 40  (binary 0000000000101000) */
i |= j;     /* i is now 56  (binary 0000000000111000) */
```

20.1.3　用按位运算符访问位

在进行低级编程时，经常会需要将信息存储为单个位或一组位。例如，在编写图形程序时，我们可能会需要将两个或更多的像素挤在一个字节中。通过按位运算，我们可以提取或修改存储在少数几个位中的数据。

假设i是一个16位整型变量。以i为例，我们来看看如何使用最常用的单个位运算：

- **设置位。**假设我们需要将设置i的第4位。（我们假定最高位为第15位，最低位为第0位。）设置第4位的最简单方法是将i的值与常量0x0010（一个在第4位上为1的“掩码”）进行或运算：

  [惯用法]
  ```
  i = 0x0000;        /* i is now 0000000000000000 */
  i |= 0x0010;       /* i is now 0000000000010000 */
  ```

  更通用的做法是，如果需要设置的位的位置存储在变量j中，可以使用移位运算符来构造掩码：

  ```
  i |= 1 << j;       /* set bit j */
  ```

  例如，如果j的值为3，1<<j是0x0008。

454 - **将位清0。**要清除i的第4位，可以使用第4位为0、其他位为1的掩码：

  [惯用法]
  ```
  i = 0x00ff;        /* i is now 0000000011111111 */
  i &= ~0x0010;      /* i is now 0000000011101111 */
  ```

  按照类似的思路，我们可以很容易编写语句来清除一个特定的位，这个位的位置存储在一个变量中：

  ```
  i &= ~(1 << j);    /* clears bit j */
  ```

- **测试位。**下面的if语句测试i的第4位是否被设置：

  [惯用法]
  ```
  if (i & 0x0010) ...   /* test bit 4 */
  ```

  如果要测试第j位是否被设置，可以使用下面的语句：

www.TopSage.com

```
if (i & 1 << j)...    /* test bit j */
```

为了使对于位的操作更容易，经常会给它们起名字。例如，如果我们想要使用一个数的第0、1和2位对应于相应的颜色蓝、绿和红。首先，定义名字分别代表三个位：

```
enum {BLUE = 1, GREEN = 2, RED = 4};
```

当然，可以将BLUE、GREEN和RED定义成宏。设置、清除或测试BLUE位可以如下进行：

```
i |= BLUE;        /* sets BLUE bit   */
i &= ~BLUE;       /* clears BLUE bit */
if (i & BLUE)...  /* tests BLUE bit  */
```

同时设置、清除或测试几个位也一样简单：

```
i |= BLUE I GREEN;              /* sets BLUE and GREEN bits   */
i &= ~(BLUE | GREEN);           /* clears BLUE and GREEN bits */
if (i & (BLUE | GREEN))...      /* tests BLUE and GREEN bits  */
```

其中if语句测试BLUE位和GREEN位中是否其中一位被设置了。

20.1.4 用按位运算符访问位域

处理一组连续的位（位域）比处理单个位要复杂一点。下面是两种最常见的位域操作的例子：

- **修改位域。**修改位域需要使用按位与（用来清除位域），接着使用按位或（用来将新的位存入位域）。下面的语句显示了如何将二进制的值101存入变量i的第4位—第6位：

  ```
  i = i & ~0x0070 | 0x0050;   /* stores 101 in bits 4-6 */
  ```

  运算符&清除了i的第4位至第6位，接着运算符|设置了第6位和第4位。注意，使用i |= 0x0050并不总是可行，这只会设置第6位和第4位，但不会改变第5位。为了使上面的例子更通用，我们假设变量j包含了需要存储到i的第4位—第6位的值。我们需要在执行按位或操作之前将j移位至相应的位置：

  455

  ```
  i = (i & ~0x0070) | (j << 4);    /* store j in bits 4-6 */
  ```

  运算符<<的优先级比运算符&和|的优先级高，所以可以去掉圆括号：

  ```
  i = i & ~0x0070 | j << 4;
  ```

- **获取位域。**当位域处在数的末尾时（在低序号的位时），获得它的值非常方便。例如，下面的语句获取了变量i的第0位—第2位：

  ```
  j = i & 0x0007;     /* retrieves bits 0-2 */
  ```

  掩码0x0007在每个需要的位置都为1。如果位域位于i的中间，那首先需要将位域移位至数的最右端，再使用运算符&来获取位域。例如要获取i的第4位—第6位，可以使用下面的语句：

  ```
  j = (i >> 4) & 0x0007;  /* retrieves bits 4-6 */
  ```

20.1.5 程序：XOR 加密

将数据加密的一种最简单的方法就是，将每一个字符与一个密匙进行异或（XOR）运算。假设密匙是一个&字符。（➤附录E）如果将它与字符z异或，我们会得到字符\（假定使用ASCII字符集）。具体计算如下：

```
        00100110   (&的ASCII码)
XOR     01111010   (z的ASCII码)
        01011100   (\的ASCII码)
```

要将信息解码，只需采用相同的算法。换而言之，只需将加密后的信息再次加密，即可得到原始的信息。例如，如果我们将&字符与\字符异或，就可以得到原来的字符z：

www.TopSage.com

```
        00100110   (&的ASCII码)
XOR     01011100   (\的ASCII码)
        01111010   (z的ASCII码)
```

下面的程序xor.c通过将每个字符与&字符进行异或来加密信息。原始信息可以由用户输入，或者使用输入重定向从文件中读入。加密后的信息可以在屏幕上显示，也可以通过输出重
456 定向来存入文件中（➤22.1节）。例如，假设文件msg包含下面的内容：

```
Trust not him with your secrets, who, when left
alone in your room, turns over your papers.
            Johann Kaspar Lavater (1741-1801)
```

为了将文件msg加密，并将加密后的信息存储在文件newmsg中，需要使用下面的命令：

```
xor <msg >newmsg
```

文件newmsg将包含下面的内容：

```
rTSUR HIR NOK QORN _IST UCETCRU, QNI, QNCH JC@R
GJIHC OH _IST TIIK, RSTHU IPCT _IST VGVCTU.
            --IINGHH mGUVGT jGPGRCT (1741-1801)
```

要获取原始的信息，需要使用命令

```
xor <newmsg
```

将原始信息显示在屏幕上。

正如例子中看到的，程序不会改变一些字符，包括数字。将这些字符与&异或会产生不可见的控制字符，这在一些操作系统中会引发错误。在第22章中，我们会看到在读和写包含控制字符的文件时，如何避免问题的发生。而这里，为了安全我们将使用iscntrl函数（➤23.4.1节）来检查原始字符或新字符（加密后的字符）是否为控制字符。如果是的话，让程序使用原始的字符，而不用新的字符。

下面是完成后的程序，相当短小：

xor.c

```
/* Performs XOR encryption */

#include <ctype.h>
#include <stdio.h>

#define KEY '&'

main()
{
  int orig_char, new_char;

  while ((orig_char = getchar()) != EOF) {
    new_char = orig_char ^ KEY;
    if (iscntrl(orig_char) || iscntrl(new_char))
      putchar(orig_char);
    else
      putchar(new_char);
  }

  return 0;
}
```

457

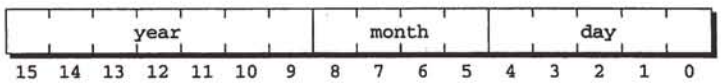
20.2 结构中的位域

虽然在20.1节中的方法可以操作位域，但这些方法不易使用，而且可能会引起一些误会。幸运的是，C语言提供了另一种选择——声明其成员表示位域的结构。

例如，我们来看看DOS是如何存储在文件创建和最后修改的日期的。由于日期、月和年都

大学网 TopSage.com

是很小的数，将它们按整数存储会很浪费空间。然而，DOS只使用了16位来存储日期。其中5位用于日、4位用于月、7位用于年：



利用位域，我们可以定义相同形式的C结构：

```
struct file_date {
   unsigned int day: 5;
   unsigned int month: 4;
   unsigned int year: 7;
};
```

在每个成员后面的数指定了它所占用位的长度。由于所有的成员的类型都一样，如果需要，我们可以简化声明：

```
struct file_date {
  unsigned int day: 5, month: 4, year: 7;
};
```

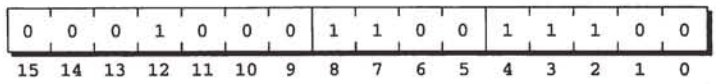
位域的类型必须是int、unsigned int或signed int。使用int会引起二义性，因为一些编译器将位域的最高位作为符号位，而其他一些编译器则不会。

可移植性技巧 将所有的位域声明为unsigned int或signed int。

我们可以将位域如同结构的其他成员一样使用。如下面的例子：

```
struct file_date fd;

fd.day = 28;
fd.month = 12;
fd.year = 8;     /* represents 1988 */
```

在这些赋值语句之后，变量fd的形式如下图所示。



458

使用按位运算符可以达到同样效果，而且使用按位运算符甚至可能使程序更快些。然而，使程序更易读通常比获得几个微秒更重要一些。

使用位域有一个限制，这个限制对结构的其他成员不起作用。由于通常意义上讲位域没有地址，C语言不允许将&运算符用于位域。由于这条规则，像scanf这样的函数无法直接向位域中存储数据：

```
scanf("%d", &fd.day);    /*** WRONG ***/
```

当然，我们可以用scanf函数将输入读入到一个普通的整型变量中，然后再赋值给fd.day。

位域是如何存储的

C语言标准在如何存储位域方面给编译器保留了一定的自由度。我们再仔细看一下编译器是如何处理包含位域成员的结构声明的。

有关编译器处理位域的规则基于“存储单元”的概念。一个存储单元的大小是由实现定义的，通常为8位、16位或32位。当编译器处理结构的声明时，会将位域逐个放入存储单元，位域之间没有间隙，直到剩下的空间不够用来放下一个位域了。这时，一些编译器会跳到下一个存储单元继续存放位域，而另一些则会将位域拆开跨存储单元存放。（具体哪种情况会发生是由实现定义的。）位域存放的顺序（从左至右，还是从右至左）也是由实现定义的。

前面file_date例子假设是基于16位长的存储单元的（8位的存储单元也可以，只要编译器

www.TopSage.com

将month字段拆开跨两个存储单元存放。）我们也可以假设位域是从右至左存储的（第一个位域会占据低序号的位），这是DOS系统上编译器常用的方式。

为了提供对位域存储的更多控制，C语言允许忽略位域的名字。未命名的位域经常用来作为字段间的“填充”，以保证其他位域存储在适当的位置。例如，思考与DOS文件关联的时间以下列方式存储：

```
struct file_time {
  unsigned int seconds: 5;
  unsigned int minutes: 6;
  unsigned int hours: 5;
};
```

（可能你在奇怪怎么可能将一个0~59的数只用5位存储呢。实际上，DOS将秒数除以2，因此
459 seconds成员实际存储的是0~29的数。）如果我们并不关心seconds字段，可以不给它命名：

```
struct file_time {
  unsigned int: 5;          /* not used */
  unsigned int minute: 6;
  unsigned int hours: 5;
};
```

其他的位域仍会正常放置，如同second字段存在时一样。

另一个用来控制位域存储的技巧是指定未命名的字段长度为0：

```
struct s {
  unsigned int a: 4;
  unsigned int : 0;         /* 0-length bit-field */
  unsigned int b: 8;
};
```

长度为0的位域是给编译器的一个信号，告诉编译器将下一个位域放在一个存储单元的起始位置。假设存储单元是8位长的，编译器会给成员a分配4位，接着跳过余下的4位到下一个存储单元，然后给成员b分配8位。如果存储单元是16位，编译器会给a分配4位，接着跳过12位，然后给成员b分配8位。

20.3　其他低级技术

前面几章中讲过的一些C语言的特性也同样经常用于编写低级程序。作为本章的结尾，我们来看几个重要的例子：定义代表存储单元的类型，使用联合来回避通常的类型检查，以及将指针作为地址使用。我们还将介绍18.3节中没有讨论的volatile类型限定符。

20.3.1　定义依赖机器的类型

依据定义，char类型占据一个字节，所以我们有时将字符当作是字节来存储一些并不一定是字符形式的数据。但这样做时，最好定义一个BYTE类型：

```
typedef unsigned char BYTE;
```

对于不同的机器，我们还可能需要定义其他类型，例如：

```
typedef unsigned int WORD;
```

460 在稍后的例子中，我们会使用到BYTE和WORD类型。

20.3.2　用联合从多个视角看待数据

虽然在16.4节的例子中已经介绍了有关联合的便捷的使用方式，但是在C语言中，联合经常是被用于一个完全不同的目的：将一块内存看成是两种或更多不同的数据形式。

这里根据20.2节中描述的file_date结构给出一个简单的例子。由于一个file_date结构正好放入两个字节中，我们可以将任何两个字节的数据当作是一个file_date结构。特别是可

以将一个unsigned int值看作是一个file_date结构（假设整数是16位长）。下面定义的联合可以使我们方便地将一个整数与文件日期相互转换：

```
union int_date {
  unsigned int i;
  struct file_date fd;
};
```

通过这个联合，我们可以以两个字节的形式获取磁盘中文件的日期，然后得到其中的month、day和year字段的值。相反地，我们也可以按照file_date结构构造一个日期，然后作为两个字节写入磁盘中。

作为一个使用int_date联合的例子，下面的函数将整型参数以文件日期的形式显示出来：

```
void print_date(unsigned int n)
{
  union int_date u;

  u.i = n;
  printf("%d/%d/%.2d\n)", u.fd.month, u.fd.day,
         (u.fd.year+1980)%100;
}
```

为了得到年的最后两位，我们将u.fd.year加上1980（因为年是以相对于1980年计算的——根据微软的说法，1980年就是世界的开始），然后计算其结果与100相除的余数。

在使用寄存器时，这种使用联合从多种角度看待数据的方法会非常有用，因为寄存器通常划分为较小的单元。以Intel 80x86处理器为例，它包含16位的寄存器——AX、BX、CX和DX。每一个寄存器都可以看作是两个8位的寄存器。例如AX可以被划分为AH和AL两个寄存器。

当针对使用Intel处理器的计算机编程时，可能会需要用到表示寄存器AX、BX、CX和DX中的值的变量。我们需要对16位寄存器和8位寄存器都进行访问，同时保留它们之间的关系（改变AX的值会同时改变AH和AL的值，改变AH也会同时改变AX）。为了解决这一问题，可以构造两个结构，一个包含对应于16位寄存器的成员，另一个包含对应于8位寄存器的成员。然后构造一个包含这两个结构的联合：

461

```
union {
  struct {
    WORD ax, bx, cx,dx;
  } word;
  struct {
    BYTE al, ah, bl, bh, cl, ch, dl, dh;
  } byte;
} regs;
```

word结构的成员会和byte结构的成员相互重叠。例如，ax会使用与al和ah同样的内存空间。当然，这恰恰就是我们所需要的。下面是一个使用regs联合的例子：

```
regs.byte.ah = 0x12;
regs.byte.al = 0x34;
printf("AX: %x\n", regs.word.ax);
```

对ah和al的改变也会影响ax，其输出是：

```
AX: 1234
```

20.3.3 将指针作为地址使用

在11.1节中我们已经看到了，指针实际上就是一种内存地址。虽然我们通常不需要知道其细节内容，但是编写低级程序时，这些细节内容就很重要了。

在一些计算机中，地址所包含的位的个数与整型或长整型一致。因此，构造一个指针来表示某个特定的地址是十分方便的：只需要将一个整数强制转换成指针就行。例如，下面的例子将地址1000（十六进制）存入一个指针变量：

www.TopSage.com

大家网 TopSage.com

```
BYTE *p;
p = (BYTE *) 0x1000;       /* p contains address 0x1000 */
```

对于其他一些计算机则比较麻烦。当一台使用Intel CPU的计算机运行在“实时模式”下时（DOS使用的模式），地址由两个16位数组成：**段地址**和**偏移量**。构造一个包含特定地址的指针通常需要调用由非标准头提供的宏。例如，MK_FP宏（make far pointer）可以根据一对段地址/偏移量来构造指针，这个宏通常可以在<dos.h>中找到：

```
BYTE far *p;
p = MK_FP(segment, offset);
```

其中，far（不属于标准C）指明p是一个“远指针”。换而言之，它由段地址和偏移量构成。（“近
462 指针”只包含偏移量。）

20.3.4　程序：设置 Num Lock 键

在IBM PC机及其兼容机上，Num Lock切换用来确定数字键盘上的按键是作为数字键使用，还是作为移动光标的方向健使用。用户可以通过按Num Lock键，打开或关闭Num Lock功能。

下面的两个程序nlockon.c和nlockoff.c可以在不按Num Lock键的条件下设置Num Lock状态。这种功能在批处理文件（一种包含一系列DOS命令的文件）中十分有用。例如，我们可以将nlockoff命令放在DOS的autoexec.bat文件中，使Num Lock在机器启动时被关闭。

这些程序其实很容易编写，因为Num Lock的状态就保存在内存中，而且在每一台计算机中保存的地址都一样。在第40（十六进制）地址段、偏移量为17（十六进制）的字节的第5位（右数第6位）用来控制Num Lock状态。设置这一位会打开Num Lock，而清除这一位则关闭Num Lock。nlockon.c和nlockoff.c只需要将这个字节的地址存在一个指针中，然后使用按位运算符修改该字节中的Num Lock位即可。

程序nlockon.c和nlockoff.c是专门针对支持far关键字，并且提供MK_FP宏的DOS编译器编写的程序。程序nlockon.c使用|=运算符来设置Num Lock位：

nlockon.c

```
/* Turns Num Lock on */

#include <dos.h>

typedef unsigned char BYTE;

main()
{
  BYTE far *p = MK_FP(0x0040, 0x0017);

  *p |= 0x20;    /* sets Num Lock bit */
  return 0;
}
```

nlockoff.c使用&=运算符来清除Num lock位：

nlockoff.c

```
/* Turns Num Lock off */

#include <dos.h>

typedef unsigned char BYTE;

main()
{
  BYTE far *p = MK_FP(0x0040, 0x0017);

  *p &= ~0x20;   /* clears Num Lock bit */
  return 0;
}
```

463

www.TopSage.com

20.3.5 volatile 类型限定符

在一些计算机中，一部分内存空间是“易变”的，保存在这种内存空间的数据可能会在程序运行期间发生改变，即使程序自身并未试图存放新值。例如，一些内存空间可能被用于保存直接来自输入设备的数据。

使用volatile类型限定符，我们可以通知编译器程序中使用了这类易变的数据。volatile限定符通常使用在用于指向易变内存空间的指针的声明中：

```
volatile BYTE *p;      /* p will point to a volatile byte */
```

为了了解为什么要使用volatile，我们假设指针p指向的内存空间用于存放用户通过键盘输入的最近一个字符。这个内存空间是易变的：每次用户输入一个新字符，这里的值都会发生改变。我们可能使用下面的循环获取键盘输入的字符，并将它们存入一个缓冲数组中：

```
while (缓冲区未满) {
等待输入;
  buffer[i] = *p;
  if (buffer[i++] == '\n')
break;
}
```

一个完整的编译器可能会注意到这个循环既没有改变p，也没有改变*p。因此可能会对程序进行优化，使*p只被读取一次：

```
在寄存器中存储*p;
while (缓冲区未满) {
  等待输入;
  buffer[i] = 存储在寄存器中的值;
  if (buffer[i++] == '\n')
    break;
}
```

优化后的程序会不断复制同一个字符来填满缓冲区，这并不是我们想要的程序。将p声明成指向不稳定的数据可以避免这一问题的发生，因为volatile限定符会通知编译器*p每一次都必须从内存中重新读取。

对于volatile的其他用法，请见第24章的“问与答”小节。

问与答

问：为什么说&和|运算符产生的结果有时会跟&&和||一样，但又不总是如此呢？ (p.315)

答：我们来比较一下i&j与i&&j（对|与||是类似的），只要i和j的值是0或1（任何组合都可以），两个表达式的值是一样的。然而，一旦i和j是其他的值，两个表达式的值不会始终一致。例如，如果i的值是1，j的值是2，那么i&j的值是0（i和j之间没有哪一位同为1），而i&&j的值是1。如果i的值是3，而j的值是2，那么i&j的值是2，i&&j的值则是1。

464

另一个问题是副作用。计算i&j++始终会先增加j，而计算i&&j++有时会先增加j。

练习

20.1节

*1. 指出下面每一个代码段的输出。假定i、j和k都是unsigned int类型的变量。

(a)
```
i = 8; j = 9;
printf("%d", i >> 1 + j >> 1);
```

(b)
```
i = 1;
```

www.TopSage.com

```
printf("%d", i & ~i);
```

(c)
```
i = 2; j = 1; k = 0;
printf("%d", ~i & j ^ k);
```

(d)
```
i = 7; j = 8; k = 9;
printf("%d", i ^ j & k);
```

2. 请说出如何“切换”一个位（从0改为1或从1改为0）。通过编写一条语句切换变量i的第4位来说明这种方法。

*3. 请解释下面的宏对它的实际参数起什么作用。假设参数具有相同类型。

```
#define M(x,y) ((x)^=(y),(y)^=(x),(x)^=(y))
```

4. 在计算机图形处理中，颜色通常是用3个数存储的，分别代表红、绿和蓝3种颜色。假定每个颜色需要8位来存储，而且我们希望将三种颜色一起存放在一个长整型数据中。请编写一个名为MK_COLOR的宏，包含3个参数（红、绿、蓝的强度）。MK_COLOR宏需要返回一个long int值，其中后3个字节分别包含红、绿和蓝，红作为最后一个字节。

5. 编写名字为GET_RED、GET_GREEN和GET_BLUE3个宏，并以一个给定的颜色值作为参数（见练习4）。宏会返回一个8位的值表示给定颜色中红、绿、或蓝。

6. (a) 使用按位运算符编写如下函数：

```
unsigned short int swap_byte(unsigned short int i);
```

函数swap_byte的返回值是将i的两个字节调换后产生的结果。（在大多数计算机中，短整型数据占两个字节。）例如，假设i的值是0x1234（二进制形式为00010010 00110100），那么swap_byte的返回值应该为0x3412（二进制形式00110100 00010010）。编写一个程序来测试你的函数。程序以十六进制读入数，然后交换两个字节并显示出来：

```
Enter a hexadecimal number: 1234
Number with byte swapped:   3412
```

提示：使用%hx转换来读入和输出十六进制数。

465 (b) 将swap_byte函数的函数体化简为一条语句。

7. 编写如下函数：

```
unsigned in rotate_left(unsigned int i, int n);
unsigned in rotate_right(unsigned int i, int n);
```

函数rotate_left（i,n）的值应是将i左移n位并将从左侧移出的位移入i右端而产生的结果。（例如，假定整型占16位，rotate_left(0x1234, 4)将返回0x2341。）类似地，函数rotate_right也类似，只是将数字中的位向右循环移位。

8. 假定函数f的实现如下：

```
unsigned int f(unsigned int i, int m, int n)
{
  return (i >> (m+1-n) & ~(~0 << n);
}
```

(a) ~(~0 << n)的结果是什么？

(b) 函数f的作用是什么？

20.2节

9. 当按照IEEE浮点标准存储浮点数时，一个float类型的值由1个符号位（最左边的位或最重要的位），8个指数位以及23个小数位依次组成。请设计一个32位的结构类型，包含与符号位、指数位和小数位相对应的位域成员。声明的位域类型为unsigned int。请参考你的用户手册来决定位域的顺序。警告：一些编译器会限制位域在16位以内，因此当你编译这个结构时可能会有出错信息。

20.3节

10. 请设计一个联合类型，使一个32位的值既可以看作是一个float型的值，也可以看作是练习9中定义的结构。写一个程序将1存储在结构的符号位，将128存储在指数位，0存储在小数位。然后按float

466 值的形式显示存储在联合中的值。（如果你的位域设置正确的话，结果应该是-2.0。）

www.TopSage.com

大家网
TopSage.com

第21章 标 准 库

每个程序都是某些其他程序的一部分，但很少是正合适的。

前面几章中零碎地介绍了一些C语言标准库的相关知识。在本章中，我们将完整地讨论标准库。在21.1节中，会列举使用库的一些通用的指导原则，还会介绍在一些库的头中发现的技巧：使用宏来“隐藏”函数。21.2节会对标准库的15个头分别做概述性的介绍。

在随后几章中，将深入讨论标准库的头，并将相关联的头放在一起讨论。其中<stddef.h>明显不同于其他的，因此会在21.3节中介绍。

21.1 标准库的使用

C语言的标准库总共划分成15个部分，每个部分用一个头描述。许多编译器都会使用扩展后的库，因此包含的头通常会多于15个。额外添加的头当然不属于标准库的范畴，所以我们不能假设其他的编译器也可以支持这些头。这类头通常提供一些针对特定机型或特定操作系统的函数(这也解释了为什么它们不属于标准库)，它们可能会提供对屏幕或键盘更多的操作的函数。用于支持图形或窗口界面的头也是很常见的。

标准头主要由函数原型、类型定义、以及宏定义组成。如果我们的文件中调用了头中的函数，或是使用了头中定义的类型或宏，那么我们就需要在文件开头将相应的头包含进来。当一个文件包含了多个头时，#include指令的顺序无关紧要。

21.1.1 对标准库中使用的名字一些限制

任何包含了标准头的文件都必须遵守两条规则。第一，该文件不能以任何目的再使用在头文件中定义过的宏的名字。例如，如果文件包含了<stdio.h>，就不能再使用NULL了，因为使用这个名字的宏已经在<stdio.h>中定义过了；第二，具有文件作用域的库名（尤其是类型名）也不可以在文件层次重定义。因此，一旦文件包含了<stdio.h>，由于<stdio.h>中已经将size_t定义为类型名，那么就不允许在文件作用域内将size_t重定义为任何标识符。

虽然上述这些限制似乎是显而易见的，但C语言还有一些其他的限制，可能是你想不到的：

- **由一个下划线和一个大写字母开头或由两个下划线开头的标识符**，属于标准库中保留的标识符。程序不允许因任何目的使用这种形式的标识符。
- **由一个下划线开头的标识符**被保留，用于文件作用域内的标识符和标记。除非仅声明在函数内部，否则不应该使用这类标识符。
- **在标准库中所有外部链接的标识符**被保留，用于作为需要外部链接的标识符。特别是所有标准库函数的名字都被保留。因此，即使文件不需要包含<stdio.h>，也不应该声明一个外部函数叫printf，因为在标准库中已经有一个同名的函数了。

这些规则对程序的所有文件都起作用，不论文件包含了哪个头。虽然这些规则并不总是强制性的，但不遵守这些规则可能会导致程序的可移植性下降。

21.1.2 使用宏隐藏函数

C程序员经常会用宏来替代小的函数，这在标准库中同样很常见。C语言标准允许在头中定

www.TopSage.com

义与库函数同名的宏，为了起到保护作用，还要求有实际的函数存在。因此，对于库的头，声明一个函数并同时定义一个有相同名字的宏的情况并不少见。

在<ctype.h>（➤23.4节）中有大量这样成对的函数或宏定义的例子，例如用来检测一个其字符是否可以显示的函数isprint。通常的做法是在<ctype.h>中定义isprint作为一个函数：

```
int isprint(int c);
```

并同时把它定义为一个宏：

```
#define isprint(c)     ((c) >= 0x20 && (c) <= 0x7e)
```

468 在默认情况下，对isprint的调用会被作为宏调用（因为宏名会在预处理时被替换）。

在大多数情况下，我们对于使用宏来替代实际的函数还是满意的，因为这样可能会提高程序的运行速度。然而在某些情况下，我们可能需要的是一个真实的函数。这可能是由于需要尽量缩小可执行代码的大小，也可能是因为需要一个指向这个函数的指针（➤17.7节）。

如果确实存在这种需求，我们可以使用#undef指令（➤14.3.5节）来删除宏定义，从而可以访问到真实的函数。例如，下面的代码通过取消了isprint的宏定义来使用isprint函数：

```
#include <ctype.h>
#undef isprint
```

即使isprint不是宏，这样的做法也不会带来任何负面影响，因为当所提供的名字没有被定义成宏时，#undef指令不会起任何作用。

此外，我们也可以通过给名字加圆括号来屏蔽个别宏调用：

```
(isprint)(c)
```

预处理器无法分辨出带圆括号的宏，除非宏名后跟着一个左圆括号；而编译器则不会这么容易被欺骗，它仍可以认出isprint函数。

21.2　标准库概述

现在简单讨论一下标准库中的15个头。本节可以帮助你分辨出你所需要的是C标准库的哪部分。在本章及随后几章中会对每个头有更详细的介绍。如果需要了解某个库函数的具体信息，请参考附录D。

1. <assert.h>：诊断

<assert.h>（➤24.1节）仅包含assert宏。我们可以在程序中插入该宏，从而检查程序状态。一旦任何检查失败，程序会被终止。

2. <ctype.h>：字符处理

<ctype.h>（➤23.4节）包括用于字符分类及大小写转换的函数。

3. <errno.h>：错误

<errno.h>（➤24.2节）提供了errno（“error number”）。errno是一个左值（lvalue），可
469 以在调用特定库函数后进行检测，来判断调用过程中是否有错误发生。

4. <float.h>：浮点型的特性

<float.h>（➤23.1节）提供了用于描述浮点类型特性的宏，包括值的范围及精度。

5. <limits.h>：整型的大小

<limits.h>（➤23.2节）提供了用于描述整数类型和字符类型特性的宏，包括它们的最大值和最小值。

6. <locale.h>：本地化

<locale.h>（➤25.1节）提供一些函数来帮助程序适应针对一个国家或地区的特定行为方式。这些与本地化的相关的行为包括数显示的方式（包括用于小数点的字符）、货币的格式（例

www.TopSage.com

如货币符号）、字符集以及日期和时间的表示形式。

7. <math.h>：数学计算

<math.h>（➤23.3节）提供了大量用于数学计算的函数、包括三角函数，双曲函数、指数函数、对数函数、幂函数、相近取整（四舍五入）函数及绝对值运算函数。其中大部分函数使用double类型的实际参数，并返回一个double类型的值。

8. <setjmp.h>：非本地跳转

<setjmp.h>（➤24.4节）提供了setjmp函数和longjmp函数。setjmp函数会“标记”程序中的一个位置，随后可以用longjmp返回被标记的位置。这些函数可以用来从一个函数跳转到另一个（仍然活动中的）函数中，绕过正常的函数返回机制。setjmp函数和longjmp函数主要用来处理程序执行过程中的重大问题。

9. <signal.h>：信号处理

<signal.h>（➤24.3节）提供了用于异常情况（信号）处理的函数，包括中断和运行时错误。signal函数可以设置一个函数，使系统会在给定信号发生后自动调用该函数；raise函数用来产生一个信号。

10. <stdarg.h>：可变实际参数

<stdarg.h>（➤26.1节）提供给函数可以处理不定个数个参数的工具，就像printf和scanf函数。 470

11. <stddef.h>：常用定义

<stddef.h>（➤21.3节）提供了经常使用的类型和宏的定义。

12. <stdio.h>：输入/输出

<stddef.h>（22.1节至22.8节）提供了大量用于输入/输出的函数。包括对顺序读写和随机读写文件的操作。

13. <stdlib.h>：常用实用程序

<stdlib.h>（➤26.2节）包含了大量无法划归于其他头的函数。包含在<stdlib.h>中的函数可以将字符串转换成数、产生伪随机值、执行内存管理任务、与操作系统通信、执行搜索与排序以及对多字节字符及字符串进行操作。

14. <string.h>：字符串处理

<string.h>（➤23.5节）提供了用于进行字符串操作的函数，包括复制、拼接、比较及搜索。

15. <time.h>：日期和时间

<time.h>（➤26.3节）提供相应的函数来获取日期和时间、操纵时间和以多种方式显示时间等。

21.3 <stddef.h>：常用定义

<stddef.h>提供了常用的类型和宏的定义，但没有声明任何函数。定义的类型包括：

- ptrdiff_t。当进行指针相减运算时，其结果的类型。
- size_t。运算符sizeof的返回值类型。
- wchar_t。一种足够大的、可以用于表示所有支持的地区的所有字符的类型。

所有这3种类型都是整数类型。其中ptrdiff_t必须是带符号的类型，而size_t则必须是无符号的类型。关于wchar_t得更多细节，见25.2节。 471

<stddef.h>中还定义了两个宏。一个是NULL，用来表示空指针。另一个宏offsetof需要两个、参数：类型（一个结构类型）和指定成员（结构的一个成员）。offsetof宏会计算结构的起点到指定成员间的字节数。

www.TopSage.com

大家网 TopSage.com

考虑下面的结构：

```
struct s {
  char a;
  in b[2];
  float c;
};
```

offsetof(struct s, a)的值一定是0，C语言确保结构的第一个成员的地址与结构自身地址相同。我们无法确定地说出b和c的偏移量是多少。一种可能是offsetof(struct s, b)是1（因为a的长度是一个字节），offsetof(struct s, c)是5（假设整数是16位）。然而，一些编译器会在结构中留下一些空洞（无效字节）（见在第16章结尾的“问与答”小节），从而会影响到offsetof产生的值。例如，对于在a后面留下一个无效字节的编译器，b和c的偏移量相应会是2和6。这就是offsetof宏的优点：对任意编译器，它都会返回正确的偏移量，使我们可以编写移植性更好的程序。

offsetof有很多用途。例如，假如我们需要将结构s的前两个成员写入文件，但忽略成员c。我们无法使用fwrite函数（➤22.6节）来写sizeof(struct s)个字节，因为这样会将整个结构写入文件。然而，我们可以只写offsetof(struct s, c)个字节。

最后一点：一些在<stddef.h>中定义的类型和宏在其他头中也会出现。（例如，NULL宏在<locale.h>、<stdio.h>、<stdlib.h>、<string.h>和<time.h>中也有定义。）因此，只有少数程序真的需要包含<stddef.h>。

问与答

问：我注意到书中使用“标准头”，而不是“标准头文件”。不使用“文件”有什么具体原因吗？

答：是的。依据C标准，一个“标准头”不需要一定是文件。虽然绝大部分编译器确实将标准头以文件形式存储，但标准实际上允许将标准头直接内置在编译器自身中。

472

练习

21.1节

1. 在你的系统中找到存放头文件的位置。找出那些非标准头，并指明每一个的用途。
2. 在存放头文件的目录中（见练习1），找到一个使用宏来隐藏函数的标准头。
3. 当使用宏隐藏函数时，在头文件中哪一个必须放在前面：宏定义还是函数原型？验证你的结论。

21.2节

4. 你期望在哪个标准头中可以分别找到下面描述的函数或宏？
   (a) 可以得到当前是星期几的函数。
   (b) 判断一个字符是否是数字的函数。
   (c) 给出最大的unsigned int的值的宏定义。
   (d) 对一个浮点数，得到比它大的最近的整数的函数。
   (e) 指定一个字符包含多少位的宏。
   (f) 指定在一个double类型的值中，有效位个数的宏。
   (g) 在字符串中查找特定字符的函数。
   (h) 用来以读的方式打开一个文件的函数。

21.3节

5. 编写一个程序，声明书中的结构s，并显示出成员a、b和c的大小和偏移量。（使用sizeof来得到大小，使用offsetof来得到偏移量。）同时使程序显示出整个结构的大小。根据这些信息，判断结构中是否包含空洞（无效字节）。如果包含，指出每一个的位置和大小。

473

www.TopSage.com

第22章

输入/输出

在人机共生的世界中，必须调整的是人：机器是无法调整的。

C语言的输入/输出库是用<stdio.h>进行表示的。它是标准库中最大且最重要的部分。由于输入/输出是C语言的高级应用，因此这里将用一整章（在本书中，本章也许不是最重要的一章，但却是最大的一章）来讨论<stdio.h>。

从第2章开始，我们已经在使用<stdio.h>了，而且已经对printf函数、scanf函数、putchar函数、getchar函数、puts函数以及gets函数的使用有了一定的了解。本章会提供更多有关上述这些函数的信息，还会介绍一些新的用于文件处理的函数。而且值得高兴的是许多新函数和我们已经熟知的函数有着紧密的联系。例如，fprintf函数就是printf函数的“文件版”。

本章的开始将会讨论一些基本问题：流（stream）的概念、FILE类型、输入和输出重定向以及文本文件和二进制文件的差异（22.1节）。随后将转入讨论特别为文件使用而设计的函数，其中包括有打开和关闭文件的函数（22.2节）。

在讨论完printf函数、scanf函数以及与“格式化”输入/输出相关的函数以后（22.3节），将会看到读/写非格式化数据的函数：

- getc函数、putc函数以及相关的函数，它们每次读写一个字符（22.4节）。
- gets函数、puts函数以及相关的函数，它们每次读写一行字符（22.5节）。
- 读/写数据块的fread函数和fwrite函数（22.5节）。

然后，22.7节会说明如何在文件上执行随机的访问操作。最后，22.8节会描述sprintf函数和sscanf函数，这两个函数不同于读和写一个字符串的printf函数和scanf函数。

本章涵盖了<stdio.h>中的绝大部分函数，只是忽略了其中4个函数。它们分别是perror函数（➤24.2节）、vfprintf函数、vprintf函数和vsprintf函数（➤26.1.2节）。这些函数和C语言库中的其他内容关系紧密。

22.1 流

在C语言中，术语流意味着任意输入的源或任意输出的目的地。许多小型程序，就像前面介绍的那些，它们都是通过一个流（通常和键盘相关）获得全部的输入，并且通过另一个流（通常和屏幕相关）写出全部的输出。

而较大规模的程序可能会需要额外的流。这些流常常表示为磁盘上的文件，但却可以和其他类型的设备相关联：调制解调器、网络端口、打印机、光盘驱动器等。这里将集中讨论磁盘上的文件，因为这类文件通用且容易理解。（在应该说流的时候，这里可能偶尔会使用术语文件。）但是，请千万记住一点，<stdio.h>中的许多函数不仅可以处理表示成文件的流，还可以处理所有其他形式的流。

www.TopSage.com

22.1.1　文件指针

C程序中流的访问是通过**文件指针**（file pointer）实现的。此指针拥有的类型为FILE *（在<stdio.h>中定义了FILE的类型）。用文件指针表示的特定流具有标准化的名字。如果需要，则可以声明一些额外的文件指针。例如，除了标准流，如果程序还需要两个流，那么可以在程序中包含下列形式的声明：

```
FILE *fp1, *fp2;
```

虽然操作系统通常会限制在任意某时刻可以打开的流的数量，但是一个程序可以声明任意数量的FILE *型变量。

22.1.2　标准流和重定向

Q&A <stdio.h>提供了3种标准流（表22-1）。这3个标准流是备用的，也就是说不能声明它们，也无法打开或关闭它们。

表22-1　标准流

文件指针	流	默认的含义
stdin	标准输入	键盘
stdout	标准输出	屏幕
stderr	标准错误	屏幕

前面使用过的printf、scanf、putchar、getchar、puts和gets这些函数都是通过
476 stdin获得输入，并且用stdout进行输出。默认情况下，stdin表示键盘，而stdout和stderr则表示屏幕。然而，某些操作系统允许通过所谓的**重定向**（redirection）机制来改变这些默认的含义。

例如，在UNIX和DOS操作系统中可以迫使程序从文件中而不是键盘上获得输入。方法是在命令行中在字符<的后边放上文件的名字：

```
demo <in.dat
```

这种方法被称为是**输入重定向**（input redirection），它本质上是使stdin流表示为文件（此例中为文件in.dat）而非键盘。重定向的绝妙之处在于demo程序不会意识到正在从文件in.dat中读取数据，而只是知道从stdin获得的任何数据是从键盘上录入的。

输出重定向（output redirection）和此很类似。在UNIX和DOS系统中对stdout流进行重定向是通过在命令行中在字符>的后边放置文件名实现的：

```
demo >out.dat
```

现在所有写入stdout的数据将进入out.dat文件中，而不是出现在屏幕了。顺便说一下，我们还可以把输出重定向和输入重定向进行合并：

```
demo <in.dat >out.dat
```

输出重定向的一个问题是会把写给stdout的每样内容都放入到文件中。如果程序运行失常并且开始发出错误信息，那么我们只能在看到文件的那一刻才会知道。而这些应该是出现在stderr中的。把错误信息写到stderr而不是stdout中，这样做可以保证在stdout发生重定向时这些错误信息将仍会出现在屏幕上。

22.1.3　文本文件与二进制文件

<stdio.h>支持两种类型的文件：文本文件和二进制文件。在**文本文件**（text file）中，字节表示字符，这使人们可以检查或编辑文件。例如，C程序的源代码是存储在文本文件中的。

www.TopSage.com

TopSage.com

另一方面，在**二进制文件**（binary file）中，字节不一定就表示字符，字节组还可以表示其他类型的数据，比如整数和浮点数。如果看看某个二进制文件，就会立刻意识到可执行的C程序是存储在二进制文件中的。

为了明白文本文件和二进制文件之间的区别，可以思考一下在文件中存储数32 767的方法。一种选择是以文本的形式把3、2、7、6、7作为字符存储起来。假设字符集为ASCII，那么就可以得到下列5个字节：

00110011	00110010	00110111	00110110	00110111
'3'	'2'	'7'	'6'	'7'

477

另一种选择是以二进制的形式存储此数，这种方法只会占用两个字节：

01111111	11111111

就像上述示例显示的那样，用二进制的形式存储数可以节省相当大的空间。

为什么一定要区分文本文件和二进制文件呢？毕竟，无论用哪种形式，一个文件就是一个字节的序列。文本文件按行进行划分，所以必须用一些方法来标记每行的末尾，比如采用特殊的字符。而且，操作系统还可能用特殊的字符来说明文本文件的结束。另一方面，二进制文件不是按行进行划分的，而且由于二进制文件可以合法地包含任何字符，所以不可能留出文件结束字符。

在DOS操作系统中，文本文件和二进制文件之间存在两方面的差异：

- **行的结尾。**当文本文件中写入换行符时，此换行符会扩展成一对字符，即回行符和跟随的回车符。与之对应的转换发生在输入过程中。然而，把换行符写入二进制文件时，它就是单独一个字符（换行符）。
- **文件末尾。**在文本文件中把字符Ctrl+Z（\x1a）设定为文件的结束标记。（不一定要在文本文件末尾有字符Ctrl+Z，但是某些编辑器会把它放上。）而在二进制文件中字符Ctrl+Z没有特别的含义，处理它就像其他任何字符一样。

与此相反的，UNIX操作系统对文本文件和二进制文件不进行区分。两者会以相同的方式进行存储。一个UNIX文本文件在每行的结尾只有单独一个换行符，而且没有特殊字符用来标记文件末尾。

当编写用来读或写文件的程序时，需要考虑是文本文件还是二进制文件。要在屏幕上显示文件的内容，程序就可能要把文件设定为文本文件。而另一方面，一个文件复制程序就不能把要复制的文件设定为文本文件。如果那样做，那么就不能完全复制含有文件结束字符的二进制文件了。在无法确定文件是文本形式还是二进制形式时，安全的做法是把文件设定为二进制文件。

22.2 文件操作

简单正是输入和输出重定向吸引人的地方之一。它不需要打开文件、关闭文件或者执行任何其他明确的文件操作。可惜的是，重定向在许多应用中受到限制。当程序依赖重定向时，它无法控制自己的文件，甚至无法知道这些文件的名字。更糟糕的是，如果程序需要在同一时间读入两个文件或者写出两个文件，重定向都无法做到。

478

当重定向无法满足需要时，将终止使用<stdio.h>提供的文件操作。本节将探讨这些文件操作，包括打开文件、关闭文件、改变缓冲文件的方式、删除文件以及重命名文件。

www.TopSage.com

22.2.1　打开文件

```
FILE *fopen(const char *filename, const char *mode);
```

用流的方式打开文件，这要求调用fopen函数。fopen函数的第一个实际参数是含有要打开文件名的字符串。（“文件名”可能包含关于文件位置的信息，例如驱动器号或路径。）第二个实际参数是“模式字符串”，它用来说明打算对文件执行的操作内容。例如，字符串"r"说明将从文件读入数据，但是不能写数据。

DOS程序员：在fopen函数调用的文件名中含有字符\时，请一定小心。因为C语言会把字符\看成是转义序列（➤7.3.1节）的开始标志。

```
fopen("c:\project\test1.dat", "r");
```

这个调用将始终无法执行，因为编译器会把\t看成是转义字符（\p的含义是未定义的）。为了避免这类问题，可以用\\代替\：

```
fopen("c:\\project\\test1.dat", "r");
```

fopen函数返回一个文件指针。程序可以（且通常将）把此指针存储在一个变量中，稍后在需要对文件进行操作时使用它。fopen函数的典型调用形式如下所示：

```
fp = fopen("in.dat", "r");      /* opens in.dat for reading */
```

当程序为了稍后从文件in.dat中读数据而调用输入函数时，将会把fp作为一个实际参数。

当无法打开文件时，fopen函数会返回空指针。也许文件不存在，或者文件在错误的地方，再或者是未获得打开文件的许可。

永远不能假设可以打开文件。为了确保不会返回空指针，需要始终测试fopen函数的返回值。

22.2.2　模式

打算传递给fopen函数的模式字符串的内容不仅依赖于稍后将要对文件采取的操作内容，还取决于文件上的数据是文本形式还是二进制形式。为了打开一个文本文件，可以采用表22-2
479 中的一种模式字符串。

表22-2　用于文本文件的模式字符串

字符串	含义
"r"	打开文件用于读
"w"	打开文件用于写（文件不需要存在）
"a"	打开文件用于追加（文件不需要存在）
"r+"	打开文件用于读和写，从文件头开始
"w+"	打开文件用于读和写（如果文件存在就截去）
"a+"	打开文件用于读和写（如果文件存在就追加）

Q&A 当使用fopen打开二进制文件时，会需要在模式字符串中包含字母b。表22-3列出了用于二进制文件的模式字符串。

www.TopSage.com

表22-3 用于二进制文件的模式字符串

字符串	含义
"rb"	打开文件用于读
"wb"	打开文件用于写（文件不需要存在）
"ab"	打开文件用于追加（文件不需要存在）
"r+b"或者"rb+"	打开文件用于读和写，从文件头开始
"w+b"或者"wb+"	打开文件用于读和写（如果文件存在就截去）
"a+b"或者"ab+"	打开文件用于读和写（如果文件存在就追加）

从表22-2和表22-3可以看出<stdio.h>对写数据和追加数据进行了区分。当给文件写数据时，通常会对先前的内容覆盖写。然而，当为追加打开文件时，试图给文件写数据实际是在文件末尾进行添加，因而会留存文件的原始内容。

顺便说一下,应用于既读又写一个已打开的文件时的特殊规则（模式字符串包含字符+）。如果不是先调用一个文件定位函数（➤22.7节），那么就不能把读转换成写。而且，如果既没有调用fflush函数（本节稍后会有介绍）也没有调用文件定位函数，那么就不能把写转化为读。

22.2.3 关闭文件

```
int fclose(FILE *stream);
```

fclose函数允许程序关闭不再使用的文件。fclose函数的实际参数必须是文件指针，此指针来自fopen函数或freopen函数（本节稍后会有介绍）的调用。如果成功关闭了文件，那么fclose函数会返回零。否则，它将会返回错误代码EOF（在<stdio.h>中定义的宏）。

为了说明如何在实际中使用fopen函数和fclose函数，这里给出了一个程序的框架。此程序打开文件example.dat进行读数据操作，并要检查打开是否成功，然后在程序终止前再把文件关闭：

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

#define FILE_NAME "example.dat"
main()
{
  FILE *fp

  fp = fopen(FILE_NAME, "r")
  if (fp == NULL) {
    printf("Can't open %s\n",FILE_NAME);
    exit(EXIT_FILURE);
  }
  …
  fclose(fp);
  return 0
}
```

480

当然，按照C程序员的编写习惯，通常不会把fopen函数的调用和fp的声明组合在一起使用：

```
FILE *fp = fopen(FILE_NAME, "r");
```

或判定是否为NULL：

```
if ((fp = fopen(FILE_NAME, "r")) == NULL) ...
```

22.2.4 为流附加文件

```
FILE *freopen( const char *filename, const char *mode,
               FILE *stream );
```

www.TopSage.com

大家网 TopSage.com

freopen函数为已经打开的流附加上一个不同的文件。freopen函数最常见的应用是把文件和其中一个标准流相关联，这些标准流包括：stdin、stdout或stderr。例如，为了使程序开始往文件foo中写数据，可以使用下列形式的freopen函数调用：

```
if (freopen("foo","w", stdout) == NULL) {
  /* error; foo can't be opened */
}
```

在关闭了任何先前与stdout相关联的文件之后（通过命令行重定向或者前一个freopen函数调用），freopen函数将打开文件foo，并且使此文件和stdout相关联。

freopen函数通常返回的值是它的第三个实际参数（一个文件指针）。如果无法打开新的文件，那么freopen函数会返回空指针。（如果无法关闭旧的文件，那么freopen函数会忽略掉错误。）

22.2.5　从命令行获取文件名

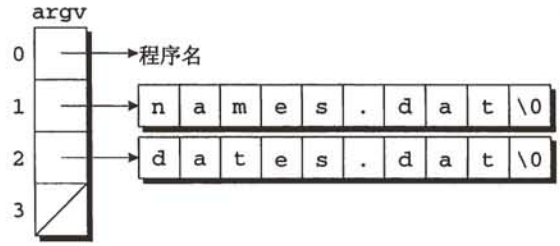
当正在编写的程序需要打开文件时，一个问题立刻变得清晰起来：如何为程序提供文件名
481 呢？把文件名嵌入程序自身的做法不会提供更多的灵活性，Q&A而且提示用户输入文件名可能也是笨拙的做法。最好的解决方案常常是通过用户在程序运行时录入的命令行来获取文件的名字。例如，当执行命名为demo的程序时，可以通过把文件名放入命令行的方法为程序提供文件名：

```
demo name.dat dates.dat
```

在13.7节中，看到了通过定义带有两个形式参数main函数的方法访问到命令行实际参数的过程：

```
main(int argc, char *argv[])
{
...
}
```

argc是命令行实际参数的数量，而argv是一个指针数组，数组中的指针都指向实际参数字符串。argv[0]指向程序的名字，从argv[1]到argv[argc-1]都指向剩余的实际参数，而argv[argc]是空指针。在上述例子中，argc是3，argv[0]指向含有程序名的字符串，argv[1]指向字符串"names.dat"，而argv[2]则指向字符串"dates.dat"：



22.2.6　程序：检查文件是否可以打开

下面的程序用来确定，若文件存在就可以打开进行读入。在运行程序时，用户将给出要检测的文件的名字：

```
canopen f1.dat
```

然后程序将显示出f1.dat can be opened或者显示出f1.dat can't be opened。如果在命令行中录入了错误的实数参数数量，那么程序将显示出信息usage: canopen filename来提醒用户canopen需要单独一个文件名。

canopen.c

```
/* Checks whether a file can be opened for reading */
```

www.TopSage.com

```
#include <stdio.h>
main(int argc,char *argv[])
{
  FILE *fp                                                    482
  if (argc != 2) {
    Printf("usage: canopen filename\n")
    Return 2
  }
  if ((fp = fopen(argv[1],"r")) == NULL) {
     printf("%s can't be opened\n", argv[1]);
     return 1;
  }

  printf("%s can be opened\n", argv[1]);
  fclose(fp);
  return 0;
}
```

注意，为了丢弃canopen的输出，也为了简单地测试返回的状态值（如果可以打开文件，则值为0，否则为1），可以使用重定向。

22.2.7 临时文件

```
FILE *tmpfile(void);
char *tmpnam(char *s);
```

现实世界中的程序经常需要产生临时文件，即只在程序运行时存在的文件。例如，C语言编译器就常常产生临时文件。编译器可能先把C程序翻译成一些存储在文件中的中间形式。然后，在稍后把程序翻译成目标代码时编译器会读取这些文件。一旦完全编译通过了程序，就不再需要保留那些含有程序中间形式的文件了。<stdio.h>提供了两个函数用来处理临时文件，即tmpfile函数和tmpnam函数。

tmpfile函数产生临时文件，这些临时文件将存到文件关闭时或程序终止时。tmpfile函数的调用会返回文件指针，此指针可以用于稍后访问文件：

```
FILE *tempptr;

Tempptr = tmpfile();   /* creates a temporary file */
```

如果产生文件失败，tmpfile函数就会返回空指针。

虽然tmpfile函数很易于使用，但是它还是有两个缺点：（1）无法知道tmpfile函数产生的文件名是什么；（2）无法决定稍后是否要使文件成为永久性的。如果这些限制产生成了问题，可以替换的解决方案就是用fopen函数产生临时文件。显然不会希望此文件拥有和前面已经存在的文件相同的名字，所以就需要一些方法产生新的文件名。这也就是tmpnam函数出现的原因。

tmpnam函数为临时文件产生名字。如果它的实际参数是空指针，那么tmpnam函数会把文 483
件名存储到静态变量中，并且返回指向此变量的指针：

```
char *filename;

filename = tmpnam(NULL);   /* creates a temporary file name */
```

否则的话，tmpnam函数会把文件名复制到程序员提供的字符数组中：

```
char filename[L_tmpnam];

tmpnam(filename);   /* creates a temporary file name */
```

在后一种情况下，tmpnam函数也会返回指向临时文件名的指针。L_tmpnam在<stdio.h>中是一个宏，它说明了保存临时文件名的字符数组有多长。

www.TopSage.com

当传递指向tmpnam函数的指针时，一定要确信指针指向至少有L_tmpnam个字符的数组。而且，还要当心不能过于频繁的调用tmpnam函数。宏TMP_MAX（在<stdio.h>中定义的）说明程序执行期间由tmpnam函数产生的临时文件名的最大数量。

22.2.8 文件缓冲

```
int fflush(FILE *stream);
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode,
            size_t size);
```

从磁盘驱动器传出或者传入信息是相对较慢的操作。这样的结果是每次程序想读或写字符时无法直接访问磁盘文件来进行。获得有效性能的诀窍就是**缓冲**（**buffering**）：写入流的数据实际是存储在内存的缓冲区域内。当缓冲区满了（或者关闭流）时，缓冲区会“清洗”（写入实际的输出设备）。输入流可以用类似的方法进行缓冲：缓冲区包含来自输入设备的数据。从缓冲区读数据代替了从设备本身读数据。缓冲在效率上可以取得巨大的收益，因为从缓冲区读字符或者在缓冲区内存储字符几乎不花什么时间。当然，需要花时间把缓冲区的内容传递给磁盘，或者从磁盘传递给缓冲区，但是一个大的“块移动”比任何微小的字符移动要快很多。

当看似有用时，<stdio.h>中的函数会自动完成缓冲操作。缓冲发生在屏幕的后台，而且通常不用担心它的操作。然而，极少的情况下可能需要我们承担更主动的作用。如果真是如此，可以使用fflush函数、setbuf函数和setvbuf函数。

当程序向文件中写输出时，数据通常是放在缓冲区中而不是文件内。当缓冲区满了或者关
484 闭文件时，缓冲区会自动清洗。然而，通过调用fflush函数，程序可以像希望的那样频繁地清洗文件的缓冲区。调用

```
fflush(fp);  /* flushes buffer for fp */
```

为和文件相关联的fp清洗了缓冲区。调用

```
fflush(NULL);     /* flushes all buffers */
```

清洗了全部输出流。如果调用成功，那么fflush函数会返回零，而如果发生错误，则返回EOF。

setvbuf函数允许改变缓冲流的方法，并且允许控制缓冲区的大小和位置。函数的第三个实际参数说明了期望缓冲区的类型：

- _IOFBF（满缓冲）。当缓冲区为空时，从流读入数据。或者当缓冲区满时，向流写入数据。
- _IOLBF（行缓冲）。每次从流读入一行数据或者向流写入一行数据。
- _IONBF（无缓冲）。直接从流读入数据或者直接向流写入数据，而没有缓冲区。

（所有这3种宏都在<stdio.h>中进行了定义。）

setvbuf函数的第二个实际参数（如果它不是空指针的话）是期望缓冲区的地址。缓冲区可以有静态存储期限、自动存储期限或是可以动态分配的。使缓冲区自动化将允许它的空间在块退出时可以被自动的重声明。动态分配缓冲区使在不需要时可以释放缓冲区。setvbuf函数的最后一个实际参数是缓冲区内字节的数量。较大的缓冲区可以提供更好的性能，而较小的缓冲区可以节约空间。

例如，下列这个setvbuf函数的调用利用buffer数组中的N个字节作为缓冲区，而把strream的缓冲变成了满缓冲：

```
char buffer[N];
setvbuf(stream, buffer, _IOFBF, N);
```

www.TopSage.com

大学网 TopSage.com

必须在打开stream之后，而且执行任何其他在stream上的操作之前调用setvbuf函数。

如果调用成功，setvbuf函数返回零。如果要求的缓冲模式是无效的或者无法提供，那么setvbuf函数会返回非零值。

setbuf函数是一个较早期的函数，它用来设定缓冲模式的默认值和缓冲区的大小。如果buf是空指针，那么setbuf(stream, buf)函数的调用就等价于

```
(void) setvbuf(stream, NULL, _IONBF, 0);
```

否则的话，它就等价于

```
(void) setvbuf(stream, buf, _IOFBF, BUFSIZ);
```

485

这里的BUFSIZ是在<stdio.h>中定义的宏。我们把setbuf函数看成是陈旧的内容，不建议大家在新程序中使用。

当使用setvbuf函数或者setbuf函数时，一定要确保在释放缓冲区之前已经关闭了流。

22.2.9 其他文件操作

```
int remove(const char *filename);
int rename(const char *old, const char *new);
```

remove函数和rename函数允许程序执行基本的文件管理操作。不同于本节中大多数其他函数，remove函数和rename函数对文件名而不是文件指针进行处理。如果调用成功，两个函数都返回零。否则，都返回非零值。

remove函数删除文件：

```
remove("foo");        /* deletes the file named "foo" */
```

如果程序使用fopen函数来产生临时文件，那么它可以使用remove函数在程序终止前删除此文件。要确信已经关闭了要移除的文件。移除文件的效果就是当前打开的是由实现定义的。

rename函数改变文件的名字：

```
rename("foo", "bar");      /* renames "foo" to "bar" */
```

如果程序需要决定使文件变为永久的，那么rename函数是很便于对用fopen函数产生的临时文件进行换名的。如果具有新的名字的文件已经存在了，那么效果会是由实现定义的。

如果打开了要换名的文件，那么一定要确保在调用rename函数之前此文件是关闭的。如果文件是打开的，则无法对文件进行换名。

22.3 格式化的输入/输出

在本节中，我们将介绍用来控制格式串读/写的库函数。这些库函数包括已经知道的printf函数和scanf函数。这类库函数可以在输入时把字符格式的数据转化为数字格式的数据，并且可以在输出时把数字格式的数据再转化成字符格式的数据。其他的输入/输出函数都不能完成这样的转化。

486

22.3.1 ...printf 类函数

```
int fprintf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
```

www.TopSage.com

fprintf函数和printf函数为输出流写可变的数据项，并且利用格式串来控制输出的形式。这两个函数的原型都是以省略号（➤26.1节）结尾的，省略号说明额外的实际参数的可变数量，这两个函数的返回值是写入的字符数。若出错则返回一个负值。

fprintf函数和printf函数唯一的不同就是printf函数始终向标准输出流stdout中写输出，而fprintf函数则向它自己的第一个实际参数说明的流中写输出：

```
printf("Total: %d\n", total);          /* writes to stdout */
fprintf(fp, "Total: %d\n", total);     /* writes to fp */
```

printf函数的调用等价于fprintf函数把stdout作为第一个实际参数而进行的调用。

但是，不要以为fprintf函数只是把数据写入磁盘文件的函数。和<stdio.h>中的许多函数一样，fprintf函数用于任何输出流都是非常好的。事实上，fprintf函数最普通的应用之一就是向标准错误stderr写出错信息，而标准错误流和磁盘文件是没有任何关系的。下面就是这类调用的一个示例：

```
fprintf(stderr, "Error: data file can't be opened.\n");
```

即使用户重定向stdout，向stderr写入的信息也保证会出现在屏幕上。

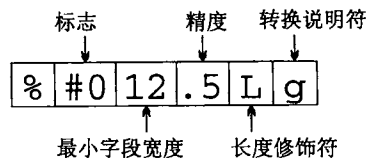
在<stdio.h>中还有其他两种函数也可以向流写入格式化的输出。（➤26.1节）这两个不常见的函数一个是vfprintf函数，另一个是vprintf函数。这两个函数都依赖于<stdarg.h>中定义的va_list类型，因此这两个函数将和<stdarg.h>一起进行讨论。

22.3.2 ...printf 类函数的转换说明

fprintf函数和printf函数都要求格式串包含普通字符或**转换说明**。普通字符将会原样输出，而转换说明则描述了如何把剩余的实参转换为字符格式显示出来。3.1节简要介绍了转换说明，而且在其后续章中还添加了不少这方面的细节。现在，我们将对已知的转换说明内容进行回顾，并且把剩余的内容补充完整。

487

...printf这类函数的转换说明由字符%和跟随其后的最多5个不同的选项构成：



下面对上述这些选项进行了详细的描述，并且显示的顺序是按照要求的次序排列的：

- **标志**。（可选项，允许多于一个）。标志-使转换参数在其字段内左对齐。而其他标志则会影响显示数的形式。表22-4给出了标志的一个完整列表。

表22-4　用于...printf类函数的标志

标　志	含　义
-	在字段域内左对齐
+	以+开头的正符号数
空格	用空格作为正符号数的前缀（+标志取代空格标志）
#	以0开头的八进制数，以0x或0X开头的十六进制数。浮点数始终是十进制形式。不能删除由g或G转换的输出数的尾部零
0（零）	用前导零在数的字段宽度内进行填充。如果转换是d、i、o、u、x或x，而且指定了精度，那么可以忽略标志0

- **最小字段宽度**（可选项）。如果字符的数量太少以致于无法达到字段宽度的最小值时，就会对字符数量进行扩充。（默认情况下会在数据项的左侧添加空格，因此要在其字段宽度内进行右对齐。）如果字符数量过多超过了字段宽度的最小值，那么会始终完整地显示出来。字

www.TopSage.com

段宽度既可以是整数也可以是字符*。如果是字符*，那么字段宽度由下一个参数决定。

- **精度**（可选项）。精度的含义依赖于转换说明符：
  如果转换说明符是d、i、o、u、x、X，那么精度表示最少数字位数（如果数字位数少于精度值，则添加前导零）；如果转换说明符是e、E、f，那么精度表示小数点后的数字位数；如果转换说明符是g、G，那么精度表示最大有效数字位；如果转换说明符是s，那么精度表示最大字符数。
  精度是由一个小数点（.）跟随一个整数或字符*构成的。如果出现字符*，那么精度由下一个参数决定。如果只有小数点，那么精度就为零。 488
- **h、l或L这些字母中的一个**（可选项）。当把这些字母用于显示整数时，字母h说明整数是short型的；字母l说明整数是long型的。当和e、E、f、g或G一起使用时，字母L说明的是long double型的参数。
- **转换说明符**。转换说明符必须是表22-5列出的某一种字符。注意f、e、E、g和G全部设计用来描述double型的值，但是把它们用于float型的值效果一样很好。这都是因为有了默认的实际参数提升（➤9.3.1节），所以当对带有可变实参的函数进行传递时，float型实参会自动转化为double型数据。与此类似的，把字符传递给...printf类函数时也会自动转换为int型，所以也可以正常使用转换说明符c。

表22-5 ...printf类函数的转换说明符

转换说明符	含 义
d、i	有符号整数转换为十进制形式
o、u、x、X	无符号整数转换为8进制(o)、10进制（u）或16进制（x、X）形式。x表示用小写字母a-f来显示十六进制数；而X表示用大写字母A-F来显示十六进制数
f	double型值转换为十进制形式，并且把小数点放置在正确的位置上。如果没有说明精度，那么在小数点后面显示6个数字
e、E	转换为科学计数法形式表示的double型值。如果没有说明精度，那么在小数点后面显示6个数字。如果选择e，那么要把字母e放在指数前面。如果选择E，那么要把字母E放在指数前面
g、G	g会把double型值转化为f形式或者e形式。仅当数值的指数部分小于-4，或者指数部分大于或等于精度值时，会选择e形式显示。不显示尾部零，且小数点仅在后边跟有数字时才显示出来。而G会在f形式和E形式之间进行选择
c	显示无符号字符的int型值
s	写出由实参指向的字符串。当达到精度值（如果存在）或者遇到空字符时，才停止写操作
p	转化为可显示格式的void *型值
n	匹配的实参必须是指向int型（如果h放在n前面，表示为short int型数；如果l放在n前面，则表示long int型数）数的指针。到目前为止，...printf类函数调用输出的字符的数量会存储到指向的整数中，不产生输出
%	写字符%

请认真遵守这里描述的规则。使用无效转换说明的结果是无法定义的。

许多看似可能的转换说明（比如说%le、%lf和%lg）实际是无效的。 489

22.3.3 ...printf 类函数的转换说明示例

现在来看一些示例。在前面我们已经看过大量日常转换说明的例子了，所以下面将集中说明一些更高级的应用示例。正像前几章那样，这里将用·表示空格字符。

我们首先来看看标志在转换%d上的效果（它们在其他转换上具有类似的效果）。表22-6的第一行显示了%8d没有任何标志的效果。接下来的四行分别显示了带有标志-、+、空格以及0所产

大家网 TopSage.com

生的效果（标志#永远不能用于%d中）。剩下的几行显示了标志组合所产生的效果。

表22-6　标志在转换%d上所产生的效果

转换说明	对123应用转换说明的结果	对-123应用转换说明的结果
%8d	·····123	····-123
%-8d	123·····	-123····
%+8d	····+123	····-123
% 8d	·····123	····-123
%08d	00000123	-0000123
%-+8d	+123····	-123····
%- 8d	·123····	-123····
%+08d	+0000123	-0000123
% 08d	·0000123	-0000123

表22-7说明了在转换o、x、X、g和G上标志#所产生的效果。（标志#也可用于e、E和f，但是这种用法非常少见。）

表22-7　标志#的效果

转换说明	对123应用转换说明的结果	对123.0应用转换说明的结果
%8o	·····173	
%#8o	····0173	
%8x	·····7b	
%#8x	····0x7b	
%8X	·····7B	
%#8X	····0X7B	
%8g		·····123
%#8g		·123.000
%8G		·····123
%#8G		·123.000

在前面的章中已经在数表示上使用过最小字段宽度和精度了，所以这里不再给出更多的示例了。而表22-8显示了在转换%s上最小字段宽度和精度所产生的效果。

490

表22-8　最小字段宽度和精度在转换%s上所产生的效果

转换说明	对"bogus"应用转换说明的结果	对"buzzword"应用转换说明的结果
%6s	·bogus	buzzword
%-6s	bogus·	buzzword
%.4s	bogu	buzz
%6.4s	··bogu	··buzz
%-6.4s	bogu··	buzz··

表22-9说明了转换%g如何以%e格式显示一些数而以%f格式显示其他数。表中的全部数都用转换说明%.4g进行了书写。前两个数所具有的指数至少为4，因此它们是按照%e的格式显示的。接下来的8个数是按照%f的格式显示的。最后两个数所具有的指数少于-4，所以也用%e的格式进行显示。

表22-9　转换%g的示例

数	对数应用转换%.4g所产生的结果
123456.	1.235e+05
12345.6	1.235e+04
1234.56	1235
123.456	123.5

www.TopSage.com

（续）

数	对数应用转换%.4g所产生的结果
12.3456	12.35
1.23456	1.235
0.123456	0.1235
0.0123456	0.01235
0.00123456	0.001235
0.000123456	0.0001235
0.0000123456	1.235e-05
0.00000123456	1.235e-06

过去，我们假设最小字段宽度和精度都是嵌在格式串中的常量。在数中放置字符*通常会允许把此字符说明是在格式串后的实际参数。例如，下列printf函数的调用都产生相同的输出：

```
printf("%6.4d", i);
printf("%*.4d", 6, i);
printf("%6.*d", 4, i);
printf("%*.*d", 6, 4, i);
```

注意，为字符*而填充的值只出现在显示值之前。顺便说一句，字符*的主要优势就是它允许使用宏来说明字段宽度或精度：

```
printf("%*d*", WIDTH, I);
```

在程序执行期间我们甚至可以计算出字段宽度或精度：

```
printf("%*d", page_width/num_cols, I);
```

最不常见的转换说明是%p和%n。转换%p允许显示出指针的值：

```
printf("%p\n", (void* )ptr);   /* displays value of ptr */
```

虽然在调试时%p偶尔有用，但它不是大多数程序员在日常基本会用到的特性。当用%p进行显示时，C标准不会指定指针显示的形式，但它可能会以八进制或十六进制的形式显示出来。 491

转换%n用来找出到目前为止由...printf函数调用所显示出的字符数量。例如，

```
printf("%d%n\n", 123, &len);
```

在下列调用后len的值将为3，因为在执行转换%n的时候printf函数已经显示出3个字符(123)。注意在len前面必须要有&，这样才不会显示出len自身的值。

22.3.4 ...scanf 类函数

```
int fscanf(FILE *stream, const char *format, ...);
int scanf(const char *format, ...);
```

fscanf函数和scanf函数从输入流读入数据，并且使用格式串来说明输入的格式。任意数量的指针跟在格式串的后边作为额外的实际参数。把输入的数据项进行转换（根据格式串中的转换说明）并且存储在指针指定的位置上。

Scanf函数始终从标准输入流stdin中读入内容，而fscanf函数则从它自己的第一个实参所指定的流中读入内容：

```
scanf("%d%d", &I, &j);              /* reads from stdin */
fscanf(fp, "%d%d", &I, &j);    /* reads from fp */
```

scanf函数的调用等价于fscanf函数把stdin作为第一个实际参数而进行的调用。

如果发生**输入失败**（即没有可以读入的字符），或者发生**匹配失败**（即输入字符无法和格式串相匹配），那么...scanf类函数都会提前返回。这两个函数都返回读入并且赋值给实参的数据

www.TopSage.com

项数量。如果在能读取任何数据项之前发生输入失败，那么会返回EOF。

在C程序中测试scanf函数的返回值的循环很普遍。例如，下列循环逐个读取一串整数，在首个问题符号处停止：

[惯用法]

```
while (scanf ("%d", &I) == 1) {
  ...
}
```

22.3.5 ...scanf 类函数的格式化字符串

...scanf类函数的调用类似于...printf类函数的调用。然而，这样的相似可能会产生误解，
492 实际上...scanf类函数的工作是不同于...printf类函数的。把scanf函数和fscanf函数看成是“模式匹配”函数是很合适的。格式串表示的模式就是...scanf类函数试图匹配的输入方式。如果输入和格式串不匹配，那么一旦发现不匹配函数就会返回。将来为了读取不匹配的输入字符将对其进行“回退”操作。

...scanf类函数的格式串可能含有三种信息类型：

- **转换说明**。在...scanf类函数格式串中的转换说明类似于...printf类函数格式串中的转换说明。大多数转换说明会在输入项的开始处跳过空白字符（%[、%c和%n例外]（➤3.2.2节）。但是，转换说明从来不会跳过尾部的空白字符。如果输入含有·123¤，那么转换说明%d会把·、1、2和3吸收进来，但是留下¤不读取。（这里使用·表示空格符，而用¤表示换行符。）
- **空白字符**。在...scanf类函数格式串中的一个或多个连续的空白字符匹配零个或多个输入流中的空白字符。
- **非空白字符**。除了%，其他非空白字符和输入流中的相同字符进行匹配。

例如，格式串"ISBN %d-%d-%ld-%d"说明输入由下列这些内容构成：多个字母ISBN，一些可能的空白字符，一个整数，字符-，一个整数（可能前面放置空白字符），字符-，一个长整数（可能前面放置空白字符），字符-和一个整数（可能前面放置空白字符）。

22.3.6 ...scanf 类函数的转换说明

实际上用于...scanf类函数的转换说明比用于...printf类函数的转换说明简单了许多。...scanf类函数的转换说明由字符%和跟随其后的选项构成。下面按照出现的顺序列出了可能的选项。

- **字符*（可选项）**。字符*的出现意味着**赋值屏蔽**（assignment suppression）：读入此数据项，但是不会把它赋值给变量。用*匹配的数据项不会包含在...scanf类函数返回的计数中。
- **最大字段宽度（可选项）**。最大字段宽度限制了输入项的字符数量。如果达到了这个最
493 大值，那么此数据项的转换将结束。对转换开始处跳过的空白字符不会进行统计。
- **h、l或L这些字母中的一个（可选项）**。当用于读取整数时，字母h说明相匹配的实参是指向short型整数的指针；而字母l则说明是指向long型整数的指针。当和e、E、f、g或者G一起使用时，字母l说明实参是指向double型值的指针；而字母L则说明实参是指向long double型值的指针。
- **转换说明符**。转换说明符必须是表22-10中列出的某一种字符。

表22-10　用于...scanf类函数的转换说明符

转换说明符	含　义
d	匹配十进制整数
i	匹配整数。假定数是十进制形式的，除非它以0（说明是八进制形式）开头，或者是以0x或0X（说明是十六进制形式）开头

www.TopSage.com

TopSage.com

（续）

转换说明符	含 义
o	匹配八进制整数。设定相应的实参是指向unsigned int型值的指针
u	匹配十进制整数。设定相应的实参是指向unsigned int型值的指针
x,X	匹配十六进制整数。设定相应的实参是指向unsigned int型值的指针
e,E,f,g,G	匹配float型值
s	匹配一序列非空白字符，然后在末尾添加空字符
[	匹配来自*扫描集合*（稍后解释）的非空的字符序列，然后在末尾添加空字符
c	匹配*n*个字符，这里的*n*是最大字段宽度值。如果没有指定字段宽度，那么就匹配一个字符。不在末尾添加空字符
p	匹配以...printf类函数可以写出的格式的指针值
n	相应的实参必须指向int型的变量（如果在n前有h，则需要是short int型变量。如果在n前有l，则需要是long int型变量）。把到目前为止读入的字符数量存储到此变量中。没有输入会被吸收进去，而且...scanf类函数的返回值也不会受到影响
%	匹配字符%

数值型数据项可能始终用符号（+或-）作为开头。然而，说明符o、u、x和X把数据项转换成无符号的形式，所以通常不用这些说明符来读取负数。

说明符[是说明符s更加复杂（且更加灵活）的版本。使用[的完整转换说明格式是%[集合]或者%[^集合]，这里的集合可以是任意字符集。（但是，如果]是集合中的一个字符，那么它必须要首先出现。）%[集合]匹配集合（即**扫描集合**）中的任意字符序列。%[^集合]匹配非集合（换句话说，构成扫描集合的全部字符不在集合中）中的任意字符序列。例如，%[abc]匹配的是只含有字母a、b和c的任何字符串，而%[^abc]匹配的是不含有字母a、b或c的任何字符串。

许多...scanf类函数的转换说明符和<stdlib.h>中的字符串转换函数（➤26.2.1节）有着紧密的联系。这些函数把字符串（比如 "-297"）转换成与其等价的数值型值（-297）。例如，说明符d寻找可选择的符号+或-，后边跟着一串十进制的数字。这样就与要把字符串转换成十进制数时strol函数所要求的格式完全一样了。表22-11说明了转换说明符和字符串转换函数之间的对应关系。 494

表22-11 ...scanf类函数转换说明符和字符串转换函数之间的对应关系

转换说明符	字符串转换函数
d	10作为基数的strol函数
i	0作为基数的strol函数
o	8作为基数的stroul函数
u	10作为基数的stroul函数
x,X	16作为基数的stroul函数
e,E,f,g,G	strod函数

当编写scanf函数的调用时是需要十分小心的。在scanf函数格式串中无效的转换说明就像在printf函数格式串中的一样糟糕。这两种情况都会导致未定义的行为出现。

22.3.7 scanf 函数的示例

下面出现的每个示例都将把scanf函数应用在它右侧显示的输入字符上。调用会吸收用删除线显示出的字符。调用后变量的值会出现在输入的右侧。

表22-12中的示例说明了把转换说明、空白字符以及非空白字符组合在一起的效果。表22-13中的示例显示了赋值屏蔽和指定字段宽度的效果。表22-14中的示例描述了更加深奥的转换说明符（即i、[和n）。

www.TopSage.com

表22-12　scanf函数示例（第一组）

scanf函数的调用	输　入	变　量
n = scanf("%d%d", &i, &j);	12·, ·34¤	n:1 i:12 j:?
n = scanf("%d,%d", &i, &j);	12·, ·34¤	n:1 i:12 j:?
n = scanf("%d ,%d", &i, &j);	12·, ·34¤	n:2 i:12 j:34
n = scanf("%d, %d", &i, &j);	12·, ·34¤	n:1 i:12 j:?

495

表22-13　scanf函数示例（第二组）

scanf函数的调用	输　入	变　量
n = scanf("%*d%d", &i);	12·34¤	n:1 i:34
n = scanf("%*s%s", str);	My·Fair·Lady¤	n:1 str: "Fair"
n = scanf("%1d%2d%3d", &i, &j, &k);	12345¤	n:3 i:1 j:23 k:45
n = scanf("%2d%2s%2d", &i, str, &j);	123456¤	n:3 i:12 str:"34" j:56

表22-14　scanf函数示例（第三组）

scanf函数的调用	输　入	变　量
n = scanf("%i%i%i", &i, &j, &k);	12·012·0x12¤	n:3 i:12 j:10 k:18
n = scanf("%[0123456789]", str);	123abc¤	n:1 Str: "123"
n = scanf("%[0123456789]", str);	abc123¤	n:0 Str: ?
n = scanf("%[^0123456789]", str);	abe123¤	n:1 Str: "abc"
n = scanf("%*d%d%n", &i, &j);	10·20·30¤	n:1 i:20 j:5

22.3.8　检测文件末尾和错误条件

```
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
```

www.TopSage.com

如果要求...scanf类函数读入并存储*n*个数据项，那么希望它的返回值就是*n*。如果返回值小于*n*，那么一定是出错了。一共有三种可能情况：

- **文件末尾**。函数在完全匹配格式串之前遇到了文件末尾。
- **错误**。错误的发生超出了函数控制的范围。
- **匹配失败**。数据项的格式是错误的。例如，函数可能在搜索整数的第一个数字期间遇到了一个字母。

496

但是如何可以告知发生的是哪类问题呢？在许多情况下，这都不是问题；程序出问题了，可以把它舍弃掉。然而，当需要明确失败的原因时就可能会花费很多时间。

每个流都有与之相关的两个指示器：**错误指示器**（error indicator）和**文件末尾指示器**（end-of-file indicator）。当打开流时会清除这些指示器，而当流上的操作失败时就会设置某个指示器。遇到文件末尾就设置文件末尾指示器，而遇到错误就设置错误指示器。但是，匹配失败不会改变任何一个指示器。

一旦设置了错误指示器或者文件末尾指示器，它就会保持这种状态，直到可能由clearerr函数的调用引发的明确清除操作为止。clearerr函数可以清除文件末尾指示器和错误指示器：

```
clearerr(fp);     /* clears eof and error indicators for fp */
```

Q&A 某些其他的库函数因为副作用可以清除某种指示器或两种都可以清除，所以不会需要经常使用clearerr函数。

虽然没有直接访问错误指示器和文件末尾指示器，但是我们可以调用feof函数和ferror函数来判断一个流的指示器，从而检测出先前在流上的操作失败的原因。如果为和流相关的fp设置了文件末尾指示器，那么feof(fp)函数调用就会返回非零值。如果设置了错误指示器，那么ferror(fp)函数的调用也会返回非零值。而其他情况下，这两个函数都会返回零。

当scanf函数返回小于预期的值时，可以使用feof函数和ferror函数来检测问题。如果feof函数返回了非零的值，那么就说明已经到达了输入文件的末尾。如果ferror函数返回了非零的值，那么就表示在输入过程中产生了错误。如果两个函数都没有返回非零值，那么一定是发生了匹配失败。不管问题是什么，scanf函数的返回值都会告诉我们在问题产生前所读入的数据项的数量。

为了明白feof函数和ferror函数可能的使用方法，现在来编写一个函数。此函数用来搜索文件中以某个整数起始的行。下面是预计的函数调用方式：

```
n = find_int("foo", &i);
```

其中，"foo"是要搜索的文件的名字，i用来存放要搜索的整数的值，而给n赋的值就是找到的整数所在行的序号。如果出现问题（文件无法打开或者输入错误，再或者没有一此整数起始的行），find_int函数将返回一个错误代码（分别是-1、-2或-3）。

```
int find_int(const char *filename, int *ptr)
{
  FILE *fp  fopen (filename, "r");
  int line = 1;

  if (fp == NULL)
    return -1;     /* can't open file */
  While (fscanf(fp, "%d", ptr) != 1) {
    if (ferror(fp)) {
      fclose(fp);
      return -2;   /* input error */
    }
    if (feof(fp)) {
      fclose(fp)
```

497

www.TopSage.com

大家网 TopSage.com

```
      return -3    /* integer not found */
    }
    fscanf(fp, "%*[^\n]")  /* skips rest of line */
    line++;
  }

  fclose(fp);
  return line;
}
```

在while表达式中，find_int函数调用fscanf函数是打算从文件中读取整数。如果尝试失败了（fscanf函数返回了不为1的值），那么find_int函数就会调用ferror函数和feof函数来了解问题是输入错误还是遇到了文件末尾。如果都不是，那么fscanf函数一定是由于匹配错误产生的问题。由此，find_int函数会跳过当前行剩余部分的字符，并且行计数器进行自增，然后继续在下一行寻找。请注意转换说明%*[^\n]的用法是跳过全部字符直到下一个换行符为止。

22.4　字符的输入/输出

在本节中，我们将讨论用于读和写单一字符的库函数。这些函数用于文本流和二进制流是等效的。

请注意，本节中的函数把字符作为int型而非char型的值来处理。这样做的原因之一就是由于输入函数是通过返回EOF来说明一个文件末尾（或错误）情况的，而EOF又是一个负的整型常量。

22.4.1　输出函数

```
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

putchar函数向标准输出流stdout写一个字符：

```
putchar(ch);  /* writes ch to stdout */
```

498 fputc函数和putc函数是putchar函数向任意流写字符的更通用的版本：

```
fputc(ch, fp);    /* writes ch to fp */
putc(ch, fp);     /* writes ch to fp */
```

虽然putc函数和fputc函数做的工作相同，但是putc函数经常作为宏来实现，而fputc函数则作为函数使用。putchar函数通常也作为宏来使用：

```
#define putchar(c) putc*((c), stdout)
```

既提供putc函数又提供fputc函数的库看起来很奇怪。但是，正如14.3节看到的那样，宏本身有几个潜在的问题。Q&A 虽然程序员通常偏好使用putc函数，因为此函数可以提高程序的运行速度，但是fputc函数作为备选也是可用的。

如果出现了错误，那么上述这三种函数都会为流设置错误指示器并且返回EOF。否则，它们都会返回写入的字符。

22.4.2　输入函数

```
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
```

getchar函数从标准输入流stdin中读入一个字符：

```
ch = getchar();   /* reads a character from stdin */
```

www.TopSage.com

fgetc函数和getc函数从任意流中读入一个字符：

```
ch = fgetc(fp);   /* reads a character from fp */
ch = getc(fp);    /* reads a character from fp */
```

虽然getc函数和fgetc函数做的工作相同，但是getc函数通常作为宏来实现，而fgetc函数则作为函数来使用。getchar函数本身常常是按下列方式定义的宏：

```
#define getchar() getc(stdin)
```

对于从文件中读取字符来说，程序员通常喜欢用getc函数胜过用fgetc函数。因为getc函数是宏，所以它执行起来的速快较快。如果getc函数不合适，那么fgetc函数作为备选是可以使用的。

如果出现问题，那么这三个函数的行为是一样的。如果是遇到了文件末尾的问题，那么这三个函数都会设置流的文件末尾指示器，并且返回EOF。如果产生了错误，它们则都会设置流的错误指示器，并且返回EOF。为了区分这两种情况，可以调用foef函数或者ferror函数。

fgetc函数、getc函数和getchar函数最常见的用法之一就是从文件读入字符直到遇到文件末尾。一般习惯使用下列while循环来实现此目的：

499

```
[惯用法] while((ch = getc(fp)) != EOF){
           ...
         }
```

在从与fp相关的文件中读入字符并且把它存储到变量ch（它必须是int类型的）之中后，判定条件会把ch与EOF进行比较。如果ch不等于EOF，这表示还未到达文件末尾，那么就可以执行循环体。如果ch等于EOF，则循环终止。

当对文件进行读取时，始终要把fgetc函数、getc函数或getchar函数的返回值存储在int型的变量中，而不是char型的变量中。Q&A把char型变量与EOF进行比较可能会产生错误的结果。

还有另外一种字符输入函数，即ungetc函数。此函数把从流中读入的字符进行“回退”，并且清除掉流的文件末尾指示器。如果在输入过程中需要“回看”字符，那么这种能力可能会非常有效。比如，为了读入一系列数字，并且在遇到首个非数字时停止操作，可以写成

```
while (isdigit(ch = getc(fp))) {
  ...
}
ungetc(ch, fp);   /* puts back last value of ch */
```

通过持续调用ungetc函数而无需干涉读入操作就可以回退字符的数量，此数量依赖于实现和所含的流类型。这里只会确保第一次的ungetc函数调用是成功的。调用文件定位函数（即fseek函数、fsetpos函数或rewind函数）（➤22.7节）会导致回退的字符丢失了。

如果要求ungetc函数回退，那么它会返回字符。如果试图在另一次读入操作或者文件定位操作之前回退过多的字符，那么ungetc函数会返回EOF。

22.4.3 程序：复制文件

下列程序用来进行文件的复制操作。当程序执行时，会在命令行上指定原始文件名和新文件名。例如，为了把文件f1.c复制给文件f2.c，可以使用命令行

```
fcopy f1.c f2.c
```

如果命令行上没有两个正确的文件名，或者至少有一个文件无法打开，那么程序fcopy都将产生出错信息。

www.TopSage.com

fcopy.c

```
/* Copies a file */

#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
  FILE *source_fp, *dest_fp;
  int ch;

  if (argc != 3) {
    fprintf(stderr, "usage: fcopy source dest\n");
    exit(EXIT_FAILURE);
  }

  if ((source_fp = fopen(argv[1], "rb")) == NULL) {
    fprintf (stderr, "Can't open %s\n", argv[1]);
    exit(EXIT_FAILURE)
  }

  if ((dest_fp = fopen(argv[2],"wb")) == NULL) {
    fprintf (stderr, "Can't open %s\n", argv[2]);
    fclose(source_fp);
    exit(EXIT_FAILURE)
  }

  while ((ch = getc (source_fp)) != EOF)
    putc (ch, dest_fp)

  fclose(source_fp);
  fclose(dest_fp);
  return 0;
}
```

500

采用"rb"和"wb"作为文件的模式使fcopy程序既可以复制文本文件也可以复制二进制文件。如果用"r"和"w"来代替，那么程序将无法复制二进制文件。

22.5　行的输入/输出

下面将要介绍读和写行的库函数。虽然这些函数也可有效的用于二进制文本流，但是它们多数用于文本流。

22.5.1　输出函数

```
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

在13.3节已经见过puts函数，它是用来向标准输出流stdout写入一串字符的：

```
puts("Hi, there!");           /* writes to stdout */
```

501

在写入字符串中的字符以后，puts函数总会添加一个换行符。

fputs函数是puts函数的更通用版。此函数的第二个实参指明了输出要写入的流：

```
fputs("Hi, there!", fp);  /* writes to fp */
```

不同于puts函数，fputs函数不会自己写入换行符，除非字符串中本身含有换行符。

当出现错误时，上面这两种函数都会返回EOF。否则，它们都会返回一个非负的数。

22.5.2　输入函数

```
char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```

www.TopSage.com

大学网 TopSage.com

在13.3节中已经见过gets函数了，它是用来从标准输入流stdin中读取一串字符的：

```
gets(str);                          /* reads a line from stdin */
```

gets函数逐个读取字符，并且把它们存储在字符串中，直到它读到换行符时停止，且会丢弃此换行符。

fgets函数是gets函数的更通用版。它可以从任意流中读取信息。fgets函数也比gets函数更安全，因为它会限制将要存储的字符的数量。下面是使用fgets函数的方法，假设str是字符数组的名字：

```
fgets(str, sizeof(str), fp);   /* read sa line from fp */
```

在响应此调用时，fgets函数将逐个读入字符，在遇到首个换行符时停止操作，或者当已经读入了sizeof(str)-1个字符时结束操作，且无论这两种情况哪种先发生都可以。如果fgets函数读入了换行符，那么它会把换行符和其他字符一起存储。（因此，gets函数从来不存储换行符，而fgets函数有时会存储换行符。）

如果出现了错误，或者是在存储任何字符之前达到了输入流的末尾，那么gets函数和fgets函数都会返回空指针。（通常，可以使用feof函数或ferror函数来检测错误的类型。）否则的话，两个函数都会返回指向读入字符串的指针。就像希望的那样，两个函数都会在字符串的末尾存储空字符。

现在既然已经学习了fgets函数，那么建议大家在大多数情况下用fgets函数来代替gets函数来使用。对于gets函数而言，始终会有跟踪超出接收数组的范畴的可能，所以只有在确保读入字符正好适合数组大小时使用gets函数才是安全的。在没有保证的时候（而且通常是没有的），更安全的做法是使用fgets函数。注意如果把stdin作为第三个实参进行传递，那么fgets函数就会从标准输入流中进行读入：

```
fgets(str, sizeof(str), stdin);
```

502

22.6 块的输入/输出

```
size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

fread函数和fwrite函数允许程序在单步中读和写大的数据块。Q&A 虽然小心使用fread函数和fwrite函数可以用于文本流，但是它们主要还是用于二进制的流。

fwrite函数设计用来把内存中的数组复制给流。fwrite函数调用中首个实参就是数组的地址，第二个实参是每个数组元素的大小（按字节衡量），而第三个实参则是要写的元素数量。第四个实参是文件指针，此指针说明了要写入的数据位置。例如，为了写入整个数组a的内容，就可以使用下列fwirte函数调用：

```
fwrite(a, sizeof(a[0]), sizeof(a)/sizeof(a[0]), fp);
```

没有规定必须写入整个数组，可以很容易的写入数组任何区间的内容。fwrite函数返回实际写入的元素（不是字节）的数量。如果出现写入错误，那么此数就会小于第三个实参。

fread函数将从流读入数组的元素。fread函数的实参类似于fwrite函数的实参：数组的地址、每个元素的大小（按字节衡量）、读入的元素数量以及文件指针。为了把文件的内容读入数组a，可以使用下列fread函数调用：

```
n = fread(a, sizeof(a[0]), sizeof(a)/sizeof(a[0]), fp);
```

检查fread函数的返回值是非常重要的。此返回值说明了实际读入的元素（不是字节）的数量。

www.TopSage.com

此数应该等于第三个实参，除非达到了输入文件末尾或者出现了错误。feof函数和ferror函数可以用于检测任何缺陷的缘由。

请注意不要把fread函数的第二个实参和第三个实参搞混了。思考下面这个fread函数的调用：

```
fread(a, 1, 100, fp);
```

这里要求fread函数读入100个元素，且每个元素占有一个字节，所以它返回0~100的值。而下面的调用则要求fread函数读入一个有100个字节的块：

```
fread(a, 100, 1, fp);
```

503 此情况中fread函数的返回值不是0就是1。

当程序需要在终止之前把数据存储到文件中时使用fwrite函数是非常方便的。稍后，程序（或者由于其他原因是另外的程序）可以使用fread函数从内存中把数据读回来。不考虑显示，数据不一定要是数组格式的。fread函数和fwrite函数都可以用于全部类型的变量。特别是可以用fread函数读入结构，或者用fwrite函数写出结构。例如，为了把结构变量s写入文件，可以使用下列形式的fwrite函数调用：

```
fwrite(&s, sizeof(s), 1, fp);
```

22.7　文件的定位

```
int fgetpos(FILE *stream, fpos_t *pos);
int fseek(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos);
long int ftell(FILE *stream);
void rewind(FILE *stream);
```

每个流都有相关联的**文件位置**（file position）。在打开文件时，根据模式可以在文件的起始处或者末尾处设置文件位置。然后，在执行读或写操作时，文件位置会自动推进，并且允许按照顺序贯穿整个文件。

虽然对许多应用来说顺序访问是很好的，但是某些程序需要具有在文件中跳跃的能力，即可以在这里访问一些数据又可以到那里访问其他数据。例如，如果文件包含一系列记录，我们可能希望直接跳到特殊的记录处，并对其进行读入或更新。<stdio.h>通过提供5个函数来支持这种形式的访问，这些函数允许程序确定当前的文件位置或者允许改变文件的位置。

fseek函数改变与首个实参（即文件指针）相关的文件位置。第三个实参说明计算出的新位置是否和文件的起始处、当前位置或文件末尾有关。<stdio.h>为此定义了3种宏：

- SEEK_SET：文件的起始处。
- SEEK_CUR：文件的当前位置。
- SEEK_END：文件的末尾处。

第二个实参是个（可能为负的）字节计数器。例如，为了移动到文件的开始处，搜索的方向将为SEEK_SET，而且字节计数器为零：

```
fseek(fp, 0L, SEEk_SET);  /* moves to beginning of file */
```

504 为了移动到文件的末尾处，搜索的方向则应该是SEEK_END：

```
fseek(fp, 0L, SEEk_END);  /* moves to end of file */
```

为了向后移动10个字节，搜索的方向应该为SEEK_CUR，并且字节计数器要为-10：

```
fseek(fp, -10L, SEEk_CUR);/* moves back 10 bytes */
```

www.TopSage.com

注意，字节计数器的类型是long int型的，所以这里用0L和-10L作为实参。（当然，用0和-10也可以工作，因为实参会自动转化为正确的类型。）

通常情况下，fseek函数返回零。如果产生错误（例如，要求的位置不存在），那么fseek函数就会返回非零值。

顺便提一句，文件定位函数最适合用于二进制的流。C语言不禁止程序对文本流使用这些定位函数，但是对于操作系统的差异要小心。由于这些差异，fseek函数对流是文本型的还是二进制型的很敏感。对于文本流而言，或者（1）offset（fseek的第二个实参）必须为零；或者（2）whence（fseek的第三个实参）必须是SEEK_SET，且通过前面的ftell函数调用获得offset的值。（换句话说，我们只可以利用fseek函数移动到文件的起始处或者文件的末尾处，在或者返回前一次访问到的位置。）对于二进制流而言，fseek函数不要求支持whence是SEEK_END的调用。

ftell函数以长整型返回当前文件位置。（如果发生错误，ftell函数会返回-1L，并且把错误码存储到errno中。）ftell可能会存储返回的值并且稍后将其提供给fseek函数的调用，这也使返回前一个文件位置成为可能：

```
long int file_pos;
…
file_pos = ftell (fp);     /* saves current position */
…
fseek (fp, file_pos, SEEk_SET);     /* returns to old position */
```

如果fp是二进制流，那么ftell(fp)调用会以字节计数来返回当前文件位置，其中零表示文件开始。（但是，如果fp是文本流，ftell(fp)返回的值不一定是字节计数，结果，最好不要对ftell函数返回的值进行算术运算。例如，为了查看两个文件位置的差距而把ftell返回的值相减不是个好做法。）

rewind函数会把文件位置设置在起始处。调用rewind(fp)函数几乎等价于fseek(fp, 0L, SEEK_SET)，两者的差异是rewind函数不返回值，但是会为fp清除掉错误指示器。

fseek函数和ftell函数都有一个问题：它们都限制文件的位置必须是存储在长整型数中。Q&A为了用于大型文件，标准C提供了两种额外函数，即fgetpos函数和fsetpos函数。这两个函数都可以用于大型文件，因为它们都用fpos_t型值来表示文件位置。fpos_t型值不一定就是整数，比如，它可以是结构。 505

调用fgetpos(fp, &file_pos)会把与fp相关的文件位置存储到file_pos型变量中。调用fsetpos(fp, &file_pos)会为fp设置文件的位置，且此位置是存储在file_pos中的值。（此值必须是通过前一个fgetpos调用获得的。）如果fgetpos函数调用失败或者fsetpos函数调用失败，那么都会把错误代码存储到errno中。当调用成功时，这两个函数都会返回零；否则，都会返回非零值。

下面是使用fgetpos函数和fsetpos函数保存文件位置并且稍后返回的方法：

```
fops_t file_pos;
…
fgetpos (fp, &file_pos);  /* saves current position */
…
fsetpos (fp, &file_pos);  /* returns to old position */
```

程序：修改零件记录文件

下面这个程序把part结构的二进制文件读入到数组中，且给每个结构的成员on_hand置为0，然后再把此结构写回到文件中。注意，打开的文件是既可读又可写的（"rb+"）。

invclear.c

```
/* Modifies a file of part records by setting the quantity
```

www.TopSage.com

大家网 TopSage.com

```
  on hand to zero */

#include <stdio.h>
#include <stdlib.h>

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
  int number;
  char name [NAME_LEN+1];
  int on_hand;
} inventory [MAX_PARTS];

int num_parts;

main()
{
  FILE *fp;
  int i;

  if ((fp = fopen ("invent.dat", "rb+")) == NULL) {
     fprintf (stderr,"Can't open inventory file\n");
     exit(EXIT_FAILURE);
  }

506 num_parts = fread (inventory, sizeof (struct part),MAX_PARTS, fp);
  for (i = 0; i < num_parts; i++)
     inventory[i].on_hand = 0;

  rewind (fp);
  fwrite (inventory, sizeof(struct part), num_parts, fp);
  fclose (fp);

  return 0;
}
```

顺便说一下，这里调用rewind函数是很关键的。在调用完fread函数之后，文件位置是在文件的末尾。如果打算不先调用rewind函数就调用fwrite函数，那么fwrite函数将会在文件末尾添加新的数据，而不是在旧的数据上覆写。

22.8　字符串的输入/输出

```
int sprintf(char *s, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

sprintf函数和sscanf函数允许用字符串作为流读/写数据。

sprintf函数与printf函数和fprintf函数都很类似，唯一的不同就是sprintf函数把输出写入字符数组（利用sprintf函数的第一个实参指向的数组）而不是流中。sprintf函数的第二个实参是格式串，这与printf函数和fprintf函数所用的一样。例如，

```
sprintf(str, "%d/%d/%d", 9, 20, 94);
```

此调用将会把9/20/94复制到str中。当完成向字符串写入的时候，sprintf函数会添加一个空字符，并且返回所存储字符的数量（不计空字符）。

sprintf函数有着广泛的应用。例如，有些时候可能希望为输出而对数据进行格式化，但不是真的要把数据写出。这时就可以使用sprintf函数来实现格式化，然后把结果存储在字符串中直到需要产生输出的时候再写出。sprintf函数还可以用于数的格式向字符格式的转化。

sscanf函数与scanf函数和fscanf函数都很类似，唯一的不同就是sscanf函数是从字符串（利用sscanf函数的第一个实参指向的字符串）而不是流中读取数据。sscanf函数的第二

个实参是格式串，这与scanf函数和fscanf函数所用的一样。

sscanf函数对于从由其他输入函数读入的字符串中提取数据非常方便。例如，可以使用fgets函数来获取一行输入，然后把此行数据传递给sscanf函数进一步处理：

```
fgets(str, sizeof(str), stdin);     /* reads a line of input */
sscanf(str, "%d%d", &i, &j);        /* extracts two integers */
```

507

用sscanf函数代替scanf函数或者fscanf函数的好处之一就是，可以按需要多次检测输入行，而不再只是一次，这样使识别替换的输入格式和从错误中恢复都变得更加容易了。下面思考一下读取日期的问题。读取的日期既可以是月/日/年的格式，也可以是月-日-年的格式。假设str含有一行输入，那么可以强制月、日和年按照如下形式显示：

```
if (sscanf (str, "%d /%d/%d", &month,&day, &year ) == 3)
  printf ("Month: %d, day: %d, year: %d\n", month, day, year);
else if (sscanf (str, "%d -%d -%d", &month, &day, &year) == 3)
  printf ("Month: %d, day: %d, year: %d\n", month, day, year);
else
  printf ("Date not in the proper form\n");
```

像scanf函数和fscanf函数一样，sscanf函数也返回成功读入并存储的数据项的数量。如果在找到第一个数据项之前到达了字符串的末尾，那么sscanf函数会返回EOF。

还有另一个字符串输入/输出函数，即vsprintf函数（➤26.1.2节）。因为vsprintf函数依赖于在<stdarg.h>中定义的va_list类型，所以把它和<stdarg.h>一起讨论。

问与答

问：你这里只列出了3种标准流，即stdin、stdout和stderr，但是我的编译器还提供了stdaux和stdprn。这些是什么呢？ (p.330)

答：虽然大多数DOS编译器都支持stdaux和stdprn，但是它们不是标准C的内容。stdaux表示COM（串行）端口，而stdprn表示PRN（并行）端口。通过在stdaux上执行输入/输出操作，程序可以和通过串行端口连接的设备（比如调制解调器）进行交流。通过向stdprn进行写操作，程序可以直接向打印机发送输出。

问：如果我使用输入重定向或输出重定向，那么重定向的文件名会作为命令行参数显示出来吗？

答：不会。操作系统会把这些文件名从命令行中移走。假设用下列录入运行程序：

```
demo foo <in_file bar >out_file baz
```

argc的值将为4，argv[0]将会指向程序名，argv[1]会指向"foo"，argv[2]会指向"bar"，而argv[3]则会指向"baz"。

问：我正打算编写一个需要在文件中存储数据的程序，以便稍候其他程序可以读取某些数据。对数据的存储格式而言，文本格式和二进制格式哪种更好呢？

答：这是有依赖关系的。如果数据全部以文本开始，那么用哪种格式存储没有太大的差异。然而，如果数据包含数，那么决定就非常困难。

508

通常二进制格式比较受欢迎，因为此种格式的读和写都非常快。当存储到内存中时，数已经是二进制格式了，所以将它们复制给文件是非常容易的。用文本格式写数据就会相对慢许多，因为每个数必须要转化成（通常用fprintf函数）字符格式。而稍候读取文件也将花费更多的时间，因为必须要把数从文本格式转换回二进制格式。此外，就像在22.1节看到的那样，以二进制格式存储数据常常会节省空间。

然而，二进制文件有两个缺点。很难阅读，这也就妨碍了调试过程。而且，二进制文件通常无法从一个系统移植到另一个系统，因为不同类型的计算机存储数据的方式是不同的。比如，某些机器按照2个字节的方式存储整数，而其他一些机器则可能用4个字节进行存储。一些机器期望首先存储数

www.TopSage.com

的高字节部分，而其他机器则可能希望先存储低字节部分。

问：用于UNIX系统的C程序好像从不在模式字符串中使用字母b，即使在打开的文件是二进制的时候也是如此。这意味着什么？（p.332）

答：在UNIX系统中，文本文件和二进制文件具有完全相同的格式，所以不需要使用字母b。但是，UNIX的程序员也应该包含字母b，这样他们的程序将会更加适合移植到其他操作系统上。

问：我已经看过调用fopen函数并且把字母t放在模式字符串中的程序了。字母t意味着什么呢？

答：C标准允许额外的字符在模式字符串中出现，但是它们要跟在r、w、a、b或+的后边。DOS编译器经常允许使用t来说明打开的文件是文本模式而不是二进制模式的。当然，无论如何文本模式都是默认的，所以字母t不增加任何内容。任何可能的情况下，最好避免使用字母t和其他不可移植的特性。

问：为什么调用fclose函数来关闭文件呢？当程序终止时所有打开的文件都会自动关闭难道不是真的吗？

答：通常情况下是真的，但如果调用abort函数（➤26.2.5节）来终止程序就不是了。即使在不用abort函数的时候，调用fclose函数始终还是有许多好的理由。首先，这样会减少打开文件的数量。操作系统对程序每次可以打开的文件数量都有限制，而大规模的程序可能会与此种限制相冲突。（定义在<stdio.h>中的宏FOPEN_MAX指定了实现可以同时打开的文件的最少数量。）其次，这样做程序会变得更易于阅读和修改。通过寻找fclose函数，读者很容易确定不再使用此文件的位置。最后，这样做很安全。关闭文件确保可以正确地更新文件的内容和目录。如果稍候程序崩溃了，至少文件不会受到影响。

问：我正在编写的程序会提示用户录入文件的名字。我要设置多长的字符数组才可以存储这个文件名字呢？（p.334）

509

答：这与使用的操作系统有关。幸运的是，你可以使用宏FILENAME_MAX（定义在<stdio.h>中的）来指定数组的大小。FILENAME_MAX是字符串的长度，这个字符串将会保存实现保证可以打开的最长的文件名。

问：fflush可以清除为读和写而打开的流吗？

答：根据C标准，调用fflush函数的结果是把流定义为（1）为输出打开，或者（2）为更新打开并且流的最后操作不是读。在其他全部情况下，调用fflush函数的结果是未定义的。当传递给fflush函数空指针时，它会清除所有满足（1）或（2）的流。

问：在...printf类函数或...scanf类函数调用中，格式串可以是变量吗？

答：当然。它可以是char *类型的任意表达式。这个特性使...printf类函数和...scanf类函数甚至比猜想的缘由更加多样。请看下面这个来自Kernighan和Ritchie所著的*The C Programming Language*一书的经典案例。此案例显示出程序的命令行参数，以空格分隔：

```
while (--argc > 0)
    printf((argc > 1) ? "%s" : "%s", *++argv);
```

这里的格式串是表达式(argc > 1) ? "%s " : "%s"，其结果是除了最后一个参数以外，其他所有命令行参数都会使用"%s"。

问：除了clearerr函数，哪个库函数可以清除流的错误指示器和文件末尾指示器？（p.345）

答：调用rewind函数可以清除这两种指示器，就好像打开或重新打开流一样；而调用ungetc函数、fseek函数或者fsetpos函数仅可以清除文件末尾指示器。

问：我无法使feof函数工作。因为即使到了文件末尾，它好像还是返回0。我做错了什么吗？

答：当前面的读操作失败时，feof函数只会返回1。在尝试读之前，你无法使用feof函数来检查文件末尾。相反，你应该首先尝试读，然后检查来自输入函数的返回值。如果返回的值表明操作不成功，那么你可以随后使用feof函数来确定失败是否是因为到达了文件末尾。换句话说，最好不要认为调用feof函数是检测文件末尾的方法；相反，把它想成是确认文件尾方式的想法正是读取操作失败的原因的方法。

问：我始终不明白为什么输入/输出库除了提供名为fputc和fgetc的函数还提供名为putc和getc的宏呢？依据21.2节的介绍，putc和getc已经有两种版本了（宏和函数）。如果需要真正的函数来代

TopSage.com

替宏，我们可以通过未定义的宏来显示putc函数或getc函数。所以，为什么要有fputc和fgetc存在呢？ (p.346)

510

答：这是历史原因。早在标准化以前，C语言没有规则要求真正的函数在库中备份每种参数化的宏。putc函数和getc函数传统上只作为宏来实现，而fputc函数和fgetc函数则只作为函数来实现。

***问：把fgetc函数、getc函数或者getchar函数的返回值存储到char型变量中会有什么问题？我不明白为什么判断char型变量的值是否为EOF会得到错误的结果呢？ (p.347)**

答：在这个判定中有两种情况可能导致错误的结果。为了使下面的讨论更具体，这里假设使用二进制补码存储方式。

首先，假定char型是无符号类型。（回想到某些编译器把char型变量作为有符号类型来处理，而其他编译器则会把它看成是无符号类型的。）现在假设getc函数返回EOF，这里把用来存储EOF的char型变量命名为ch。既然EOF是-1的代名词，所以ch将用值255作为结束。ch无符号字符与EOF有符号整数进行比较就要求把ch转化成有符号整数（在这个例子中是255）。因为255不等于-1，所以与EOF的比较失败了。

反之，现在假设char是有符号类型。如果getc函数从二进制流中读取了含有值255的字节，请想想这样会产生什么情况呢？把255存储在char型变量中将会为它带来值-1，因为ch是有符号字符。如果判断ch是否等于EOF，将会（错误地）产生真的结果。

问：为什么22.4节（字符的输入/输出）不介绍任何关于getch函数和getche函数的内容呢？

答：简单说就是因为getch函数和getche函数都不是标准输入/输出函数库的内容。这些允许程序捕捉单个按键的函数通常是由DOS编译器在非标准<conio.h>（控制台的输入输出）中提供的。

标准输入函数getc、fgetc和getchar都是分配缓冲区的。也就是说，这些函数是在用户按下回车（返回）键时才开始读取输入的。而另一方面， getch函数和getche函数在按回车时返回字符。这两个函数的差异在于getch不回送输入的字符而getche回送。换句话说，如果使用函数getch，用户不会看见自己正在敲击的字符是什么。

当用户按下了功能键、光标键或者计算机键盘上任何其他的特殊键时，getch函数和getche函数都可以检测到。当用户按下这类键时，这两个函数都会返回0。在下一次调用函数时，它们会返回“扫描码”用以说明按下的是哪个键。如果你使用的是DOS编译器，那么编译器手册应该会提供一个扫描码的列表，或者可以参考有关PC系列编程的书籍。

像getch和getche这类函数对于编写某些类型的程序是非常有用的。首先，它们允许交互式程序的构造，比如编辑器。这样做应该能即刻响应用户的输入。其次，它们允许程序告知什么时候用户按下特殊键。最后，函数getch允许程序读取输入而不用回送，这在某些情况下确实是很有好处的（比如说，读入密码的情况）。

511

当然，getch函数和getche函数也有它们的问题。这些函数不会给用户回退和纠正错误的机会。而且，由于这两种函数都是非标准的，所以调用它们的程序可能无法移植到UNIX或其他操作系统。

问：当正在读取用户输入时，如何能跳过当前输入行左侧的全部字符呢？

答：一种可能是编写一个小函数来读入并且忽略掉（并且包含）第一个换行符之前的所有字符：

```
void skip_line(void)
{
  while (getchar() != '\n')
  ;
}
```

另外一种可能是要求scanf函数跳过第一个换行符前的所有字符：

```
scanf("%*[^\n]");  /* skips characters up to new-line */
```

scanf函数将读取第一个换行符之前的所有字符，但是不会把它们存储下来（*说明会抑制赋值操作）。使用scanf函数的唯一问题是它会留下换行符不读，所以可能需要单独丢弃换行符。

无论做什么，都不要调用fflush函数：

www.TopSage.com

```
fflush(stdin); /* effect is undefined */
```

虽然某些实现允许使用fflush函数来“清洗”未读取的输入，但是这样做并不是一个好主意。fflush函数是设计用来清洗输出流的。C标准规定fflush函数对输入流的效果是未定义的。

问：为什么把fread函数和fwrite函数用于文本流不是一个好主意呢？ (p.349)

答：困难之一是，在某些操作系统中当对文本文件写操作时会把换行符变成一对字符（详细内容见22.1节）。这就需要考虑这种扩展，否则就很有可能丢失数据的跟踪。例如，如果使用fwrite函数来写含有80个字符的块，所以有些块可能在文件结束时会占用多于80个字节因为换行符可能被扩展。

问：为什么有两套文件定位函数（即fseek/ftell和fsetpos/fgetpos）呢？一套函数难道不够吗？(p.351)

答：fseek函数和ftell函数作为C库的一部分已有些年头了，所以它们必须包含在C标准里。可惜的是这两个函数无法用于超大规模的文件（这类文件在设计C语言的年代并不普遍），因此在C语言标准化期间又加入了fsetpos函数和fgetpos函数。

512

问：为什么本章不讨论屏幕控制，即移动光标、改变屏幕上字符颜色等呢？

答：标准C没有提供用于屏幕控制的函数。标准只发布那些通过广泛的计算机和操作系统可以合理标准化的问题，而屏幕控制超出了这个范畴。如果在DOS系统中工作，对于屏幕控制可以有几种选择，其中包括可以调用<conio.h>中的函数，大多数DOS编译器都提供<conio.h>。UNIX系统程序员面临的问题则是程序需要在多种不同的终端上工作。解决这个问题的传统做法是使用UNIX的curses库，这个库支持不依赖终端方式的屏幕控制。

问：图形的标准函数是怎样的呢？

答：没有图形的标准函数，可参见前一个问题的答案。如果你的程序需要图形功能，那么有几种选择。你所在的编译器可能带有图形库，或者也可以获得由第三方编写的图形库，再或者是你编写自己的库。

练习

22.1节

1. 指出下列每个文件可能是包含文本数据还是二进制数据。
   (a) 由C语言编译器产生的目标代码文件。
   (b) 由C语言编译器产生的程序列表。
   (c) 从一台计算机发送到另一台计算机的电子邮件。
   (d) 含有图形映象的文件。

22.2节

2. 指出在下列每种情况下会把哪种模式字符串传递给fopen函数：
   (a) 数据库管理系统打开含有将被更新的记录的文件。
   (b) 邮件程序打开存有消息的文件以便可以在文件末尾添加额外的信息。
   (c) 图形程序打开含有将被显示在屏幕上的图片的文件。
   (d) 操作系统命令解释器打开含有将被执行的命令的“批文件”（或者“壳脚本”）。
3. 扩充canopen程序，以便用户可以把任意数量的文件名放置在命令行中：

```
canopen foo bar baz
```

这个程序应该为每个文件分别显示出can be opended消息或者can't be opened消息。如果命令行中没有参数，那么程序应该返回2；如果无法打开任何文件，那么程序返回1；如果可以打开所有文件，程序应返回0。

22.3节

513

4. 请指出如果printf函数用%#012.5g作为转换说明来执行显示操作，下列数据显示的形式：
   (a) 83.7361
   (b) 29748.6607

www.TopSage.com

(c) 1054932234.0

(d) 0.0000235218

5. printf函数的转换说明%.4d和%04d有区别吗？如果有，请说明区别是什么。

*6. 编写printf函数的调用，要求如果变量widget（类型为int型）的值为1，则显示1 widget；如果值为*n*，则显示出*n* widgets。不允许使用if语句或任何其他语句；答案必须是单独的一个printf调用。

*7. 假设按照下列形式调用scanf函数：

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

（其中，i、j和n都是int型变量，而x是float型变量。）假设输入流含有下面所示的字符，请指出这个调用后i、j、n和x的值。此外，请说明一下调用会消耗掉哪些字符。

(a) 10·20·30¤

(b) 1.0·2.0·3.0¤

(c) 0.1·0.2·0.3¤

(d) .1·.2·.3¤

8. 在前面几章中，当希望跳过空白字符而读取非空字符时，已经使用过scanf函数的" %c"格式串。而一些程序员用"%1s"来代替。这两种方法等效吗？如果不等效，区别是什么？

22.4节

9. 要想从标准输入流中读取一个字符，下列调用方式哪种是无效的？

(a) getch()

(b) getchar()

(c) getc(stdin)

(d) fgetc(stdin)

10. 程序fcopy有一个小缺陷：当它向目的文件写时无法检查错误。虽然在写操作过程中错误是极少的，但是偶尔会发生（比如，磁盘可能会变满）。假设希望一旦发生错误，程序可以写出消息并且立刻终止，请说明如何为fcopy.c添加遗漏的错误检查。

11. 在程序fcopy中出现了下列循环：

```
while ((ch = getc (source_fp)) != EOF )
putc(ch, dest_fp);
```

假设忽略表达式ch = getc (source_fp)两边的圆括号：

```
while (ch = getc (source_fp) != EOF )
  putc(ch, dest_fp);
```

程序可以无错通过编译？如果可以，那么运行时程序会做些什么呢？ 514

12. 编写一个名为toupper的程序，用来把文件中的所有字母转化成大写形式。（其他非字母字符不改变。）用户将采用命令行上的输入文件名：

```
toupper test.doc
```

让程序toupper把输出写到stdout中。

13. 编写一个名为fact的程序，通过把任意数量的文件写到标准输出中而把这些文件一个接一个的“拼接”起来，而且文件之间没有间隙。例如，下列命令将在屏幕上显示文件f1.c、f2.c和f3.c：

```
fcat f1.c f2.c f3.c
```

如果任何文件都无法打开，那么程序fcat应该发出错误信息。提示：因为每次只可以打开一个文件，所以程序fcat只需要一个文件指针变量。一旦对一个文件完成操作，程序fcat在打开下一个文件时可以使用同一个文件指针变量。

14. 让下列每一个程序都通过命令行获得文件名，并都把输出写到stdout中。

(a) 编写一个名为cntchar的程序，用来统计文本文件中字符的数量。

(b) 编写一个名为cntword的程序，用来统计文本文件中单词的数量（所谓“单词”指的是不含空白字符的任意序列）。

www.TopSage.com

大家网 TopSage.com

(c) 编写一个名为cntline的程序，用来统计文本文件中行的数量。

15. 20.1节中的程序xor拒绝对原始格式或加密格式中是控制字符的字节进行加密。现在可以摆脱这种限制了。修改此程序使输入文件名和输出文件名都是命令行的参数。以二进制形式打开这两个文件，并且把用来检查原始字符或加密字符是否是控制字符的判断删除。

16. 编写一个名为hexdump的程序，以十六进制代码序列的形式显示文件中的字节，且每行显示20个代码（如下所示）。

```
43 68 61 69 72 6d 61 6e 20 42 69 6c 6c 20 6c 65 61 64 73 20
74 68 65 20 68 61 70 70 79 20 77 6f 72 6b 65 72 73 20 69 6e
20 73 6f 6e 67 21 0d 0a
```

用户需要在命令行中指定文件名。请确保文件以"rb"模式打开。

17. 在进行文件内容压缩的众多方法中，最简单快捷的方法之一是**行程长度编码**方式。这种方法通过一对字节替换相同的字节序列来进行文件的压缩：重复计数后面跟着重复的字节。例如，假设文件以下列字节序列开始进行压缩（以十六进制形式显示）：

```
46 6f 6f 20 62 61 72 21 21 21 20 20 20 20 20
```

压缩后的文件将包含下列字节：

```
01 46 02 6f 01 20 01 62 01 61 01 72 03 21 05 20
```

如果原始文件包含许多相同字节的长序列，那么行程长度编码的方法非常适用。最差的情况是行程长度编码可能实际上是文件的长度的两倍。

(a) 请编写名为comp的程序，此程序使用行程长度编码方法来压缩文件。为了运行程序comp，将使用下列格式的命令行：

515

```
comp 原始文件 压缩后的文件
```

如果压缩后的文件没有扩大，那么程序comp将添加扩展名.rle。例如，命令

```
comp foo bar
```

将会使程序comp创建名为bar.rle的文件，并且把文件foo的压缩版写到此文件中。（程序comp将在文件bar.rle开始处保存文件foo的名字。）提示：练习16中的程序hexdump可以用来调试。

(b) 请编写名为uncomp的程序，此程序是程序comp的反向操作。程序uncomp的命令行格式为：

```
uncomp 压缩后的文件
```

如果压缩后的文件没有扩充，那么程序uncomp将添加扩展名.rle。例如，命令

```
uncomp bar
```

将会使程序uncomp打开文件bar.rle，并且写出文件内容的未压缩版，同时把文件的名字保存在bar.rle的开始处。

22.5节

18.

(a) 请编写自己版本的fgets函数，使此函数的操作尽可能和实际的fgets函数相同。特别是一定要确保函数具有正确的返回值。为了避免和标准库发生冲突，请不要把自己编写的函数也命名为fgets。

(b) 请编写自己版本的fputs函数，规则和(a)要求的一样。

22.6节

19. 通过添加两个新的操作的方式修改16.3节中的invent程序：

- 在指定文件中保存数据库。
- 从指定文件中装载数据库。

分别使用代码d（转储）和r（恢复）来表示这两种操作。与用户的交互应该按照下列显示进行：

```
Enter operation code: d
Enter name of output file: invent.dat

Enter operation code: r
```

www.TopSage.com

```
Enter name of input file: invent.bat
```

20. 编写程序对由invent程序存储的含有零件记录的两个文件进行合并（见练习19）。假设每个文件中的记录都是根据零件编号进行排序的，而且希望结果文件也应是排好序的。如果两个文件都拥有相同编号的零件，那么要对记录中存储的数量进行合并。（作为连贯的检查，程序要比较零件的名称，并且在不匹配时显示出错信息。）程序在命令行上要包含输入文件名以及合并后的文件名。

*21. 修改17.5节中的程序invent2，方法是添加练习19中描述的d（转储）操作和r（恢复）操作。因为零件的结构不存储在数组中，所以d操作无法通过单独一个fwrite调用来保存所有内容。因而，它需要访问链表中的每个节点，保存零件的编号、名称以及文件中现有零件的数量。（不保存指针next，因为一旦程序终止，此指针将不会有效。）当程序从文件中读取零件时，r操作将每次一个节点地重新构建列表。

22.7节

22. 编写fseek函数的调用来在二进制文件中执行下列文件定位操作，其中，二进制文件的数据以64字节“记录”的形式进行排列。采用fp作为下列每种情况中的文件指针。

    (a) 移动到记录n的开始处（假设文件中的首记录记为0）。 516

    (b) 移动到文件中最后一条记录的开始处。

    (c) 向前移动一条记录。

    (d) 向后移动两条记录。

22.8节

23. 编写一个名为dispdate的程序，用来从命令行读取数据，并且按照下列格式显示出来：

    ```
    September 13, 1995
    ```

    允许用户以9-13-95或者9/13/95的形式录入日期，并假设日期中没有空格。如果没有按照指定格式录入日期，那么程序显示出错信息。提示：使用sscanf函数从命令行参数中截取出年、月、日。 517

www.TopSage.com

第23章 库对数值和字符数据的支持

与计算机过长时间的接触把数学家变成了书记员，反之亦然。

本章会介绍5个函数库的头，这5个头提供了对数值、字符和字符串的支持。23.1节和23.2节介绍了<float.h>和<limits.h>，它们包含了用于描述数值和字符类型特性的宏。23.3节～23.5节讨论余下的标准头：<math.h>（数学函数）、<ctype.h>（字符函数）以及<string.h>（字符串函数）。

23.1 <float.h>：浮点型的特性

<float.h>中提供了用来定义浮点型的范围及精度的宏。在<float.h>中没有类型和函数的定义。

有两个宏对所有浮点型适用。FLT_ROUND说明了浮点加法的舍入模式。表23-1列出了FLT_ROUNDS的可能值。FLT_RADIX指定了指数基数的形式，最小值是2（二进制）。

表23-1 舍入模式

取 值	含 义
-1	不确定
0	趋于零
1	趋于最近有效值
2	趋于正无穷
3	趋于负无穷

519

其他宏用来描述特定类型的特性，这里会用一系列的表格来描述。根据宏是针对float、double还是long double类型，每个宏都会以FLT、DBL或LDBL开头。C标准对这些宏给出了相当具体的定义，因此这里的介绍会更注重易于理解，而不会十分精确。依据C标准，表中列出了部分宏的最大值和最小值。

表23-2列出了与定义有效数字个数相关的宏。

表23-2 <float.h>中的有效数字宏

宏 名	取 值	宏的描述
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG		有效数字的个数（基数FLT_RADIX）
FLT_DIG	≥6	有效数字的个数（十进制）
DBL_DIG	≥10	
LDBL_DIG	≥10	

表23-3列出了与指数相关的宏。

www.TopSage.com

大学网 TopSage.com

表23-3 <float.h>中的指数宏

宏 名	取 值	宏的描述
FLT_MIN_EXP		FLT_RADIX能表示的最小（负的次幂）
DBL_ MIN_EXP		
LDBL_ MIN_EXP		
FLT_MIN_10_EXP	≤-37	10能表示的最小（负的次幂）
DBL_ MIN_10_EXP	≤-37	
LDBL_ MIN_10_EXP	≤-37	
FLT_MAX_EXP		FLT_RADIX能表示的最大次幂
DBL_ MAX_EXP		
LDBL_ MAX_EXP		
FLT_ MAX_10_EXP	≥+37	10能表示的最大次幂
DBL_ MAX _10_EXP	≥+37	
LDBL_ MAX _10_EXP	≥+37	

表23-4列出了其他宏，描述了最大值、最接近0的值（最小正数），两个数之间的最小差值。

表23-4 <float.h>中的最大值、最小值和差值宏

宏 名	取 值	宏的描述
FLT_MAX	$\geq 10^{+37}$	最大的浮点数
DBL_MAX	$\geq 10^{+37}$	
LDBL_MAX	$\geq 10^{+37}$	
FLT_MIN	$\leq 10^{-37}$	最小的规格化浮点数
DBL_MIN	$\leq 10^{-37}$	
LDBL_MIN	$\leq 10^{-37}$	
FLT_EPSILON	$\leq 10^{-5}$	最小的正数ε，ε满足：1.0+ε不等于1.0
DBL_EPSILON	$\leq 10^{-9}$	
LDBL_EPSILON	$\leq 10^{-9}$	

由于只有进行数值分析的专家才会对上述<float.h>中定义的宏感兴趣，这可能是标准库中最不常用的头。 520

23.2 <limits.h>：整值类型的大小

<limits.h>中提供了用于定义每种整型和字符型取值范围的宏。在<limits.h>中没有定义类型或函数。

在<limits.h>中，一组宏用于字符型：char、signed char和unsigned char。表23-5列举了这些宏以及它们的最大值或最小值。

表23-5 <limits.h>中的字符型宏

宏 名	取 值	宏的描述
CHAR_BIT	≥8	每个字符包含位的个数
SCHAR_MIN	≤-127	最小带符号字符
SCHAR_MAX	≥+127	最大带符号字符
UCHAR_MAX	≥255	最大无符号字符
CHAR_MIN	①	最小字符
CHAR_MAX	②	最大字符
MB_LEN_MAX	≥1	多字节字符最多包含的字节数

① 如果char类型被当作signed char类型，CHAR_MIN与SCHAR_MIN相等；否则CHAR_MIN为0。

② 根据char类型被作为signed char或unsigned char，CHAR_MAX分别与SCHAR_MAX或UCHAR_MAX相等。

www.TopSage.com

其他在<limits.h>中定义的宏针对整型：short int、unsigned short int、int、unsigned int、long int以及unsigned long int。表23-6列举了这些宏以及它们的最大值或最小值。

表23-6　<limits.h>中整型的宏

宏　名	取　值	宏的描述
SHRT_MIN	≤-32767	最小短整型数
SHRT_MAX	≥+32767	最大短整型数
USHRT_MAX	≥65535	最大无符号短整型数
INT_MIN	≤-32767	最小整型数
INT_MAX	≥+32767	最大整型数
UINT_MAX	≥65535	最大无符号整型数
LONG_MIN	≤-2147483647	最小长整型数
LONG_MAX	≥+2147483647	最大长整型数
ULONG_MAX	≥4292967295	最大无符号长整型数

<limits.h>中定义的宏在查看编译器是否支持特定大小的整数时十分方便。例如，如果要判断int类型是否可以用来存储像100 000一样大的数，可以使用下面的预处理指令：

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

如果int类型不适用，#error指令（➤14.5.1节）会中止编译。

521 进一步讲，可以使用<limits.h>中的宏来帮助程序选择正确的类型定义。假设Quantity类型的变量必须可以存储像100 000一样大的整数。那么如果INT_MAX大于100 000，我们就可以将Quantity定义为int；否则，则需要定义为long int：

```
#if INT_MAX >= 100000
typedef int Quantity;
#else
typedef long int Quantity;
#endif
```

23.3　<math.h>：数学计算

<math.h>中定义的函数包含下面5种类型：

- 三角函数。
- 双曲函数。
- 指数和对数函数。
- 幂函数。
- 就近取整函数绝对值函数和取余函数

在深入讨论这些类型之前，先来简单了解一下<math.h>中的这些函数是如何处理错误的。

23.3.1　错误

<math.h>中的函数对错误的处理方式与其他库函数不同。当发生错误时，<math.h>中的大多数函数会将一个错误代码存储到一个名字为errno的特定变量中（在<errno.h>（➤24.2节）中）。此外，一旦函数的返回值大于double类型的最大取值，<math.h>中的函数会返回一个特殊的值，这个值由HUGE_VAL宏定义（这个宏在<math.h>中定义）。HUGE_VAL是double类型，

www.TopSage.com

但不一定是一个普通的数。(IEEE浮点运算标准(➤7.2节)定义了一个值叫“无穷”—— 这个值是HUGE_VAL的一个合理的选择。)

- **定义域错误**:函数的实参超出了函数的定义域。当定义域错误发生时,函数的返回值是由实现定义的,同时EDOM(“定义域错误”)会被存储到errno中。在一些<math.h>的实现中,当定义域错误发生时,函数会返回值**NAN**(“非数”)。NAN是在IEEE标准中定义的另一个特殊的值(与“无穷”类似)。
- **取值范围错误**:函数的返回值超出了double类型的取值范围。如果返回值的绝对值过大(溢出),函数会根据结果的符号返回正的或负的HUGE_VAL。此外,值ERANGE(“取值范围错误”)会被存储到errno中。如果返回值的绝对值太小(下溢出),函数返回零;一些实现可能也会将ERANGE存到errno中。

522

本节不讨论取余时可能发生的错误。在附录D中描述的函数会解释导致每种错误的情况。

23.3.2 三角函数

```
double acos(double x);
double asin(double x);
double atan(double x);
double atan2(double y, double x);
double cos(double x);
double sin(double x);
double tan(double x);
```

cos函数、sin函数和tan函数分别用来计算余弦、正弦和正切。假定PI被定义为3.14159265,那么以PI/4为参数调用cos函数、sin函数和tan函数会产生如下的结果:

```
cos(PI/4) ⇒ 0.707107
sin(PI/4) ⇒ 0.707107
tan(PI/4) ⇒ 1.0
```

注意,传递给cos函数、sin函数和tan函数的参数是以弧度表示的,而不是以角度表示的。

acos函数、asin函数和atan函数分别用来计算反余弦、反正弦和反正切:

```
acos(1.0) ⇒ 0.0
asin(1.0) ⇒ 1.5708
atan(1.0) ⇒ 0.785398
```

对cos函数、的计算结果直接调用acos函数不一定会得到最初传递给cos函数的值,因为acos函数始终返回一个0~π的值。asin函数与atan函数会返回–π/2~π/2的值。

atan2函数用来计算y/x的反正切值,其中y是函数的第一个参数,x是第二个参数。atan2函数的返回值在–π~π。调用atan(x)与调用atan2(x, 1.0)等价。

23.3.3 双曲函数

```
double cosh(double x);
double sinh(double x);
double tanh(double x);
```

523

cosh函数、sinh函数和tanh函数分别用来计算双曲余弦、双曲正弦和双曲正切:

```
cosh(0.5) ⇒ 1.12763
sinh(0.5) ⇒ 0.521095
tanh(0.5) ⇒ 0.462117
```

传递给cosh函数、sinh函数和tanh函数的实参必须以弧度表示,而不能以角度表示。

23.3.4 指数函数和对数函数

```
double exp(double x);
```

大家网 TopSage.com

```
double frexp(double value, int *exp);
double ldexp(double x, int exp);
double log(double x);
double log10(double x);
double modf(double value, double *iptr);
```

exp函数返回e的a次幂。

```
exp(3.0) ⇒ 20.0855
```

log函数与exp函数相反，它计算参数以e为底取对数的结果。log10计算“常用”对数（以10为底）的结果：

```
log(20.0855)  ⇒ 3.0
log10(1000)   ⇒ 3.0
```

对于不以e为底或不以10为底的对数，计算起来也不复杂。例如，下面的函数对任意的x和b，计算以b为底x的对数：

```
double logb(double x, double b)
{
  return log(x) / log(b);
}
```

modf函数和frexp函数将一个double类型的值拆解为两部分。modf将它的第一个参数分为整数和小数部分，返回其中的小数部分，并将整数部分存入第二个参数所指向的变量中：

```
modf(3.14159, &int_part) ⇒ 0.14159 (int_part被赋值为3.0)
```

虽然int_part的类型必须为double，但我们始终都可以随后将它强制转换成int或longint。

frexp函数将浮点数拆成小数部分f和指数部分n，使得原始值等于$f \times 2^n$，其中$0.5 \leqslant f \leqslant 1$ 524
或$f=0$。函数返回f，并将n存入第二个参数所指向的（整数）变量中：

```
frexp(12.0, &exp) ⇒ .75 (exp被赋值为4)
frexp(0.25, &exp) ⇒ 0.5 (exp被赋值为-1)
```

ldexp函数会复原frexp产生的结果，将小数部分和指数部分组合成一个数：

```
ldexp(.75, 4) ⇒ 12.0
ldexp(0.5, -1) ⇒ 0.25
```

一般而言，调用ldexp(x, exp)将返回$x \times 2^{exp}$。

23.3.5　幂函数

```
double pow(double x, double y);
double sqrt(double x);
```

pow函数计算第一个参数的幂，幂的次数由第二个参数指定：

```
pow(3.0, 2.0)      ⇒ 9.0
pow(3.0, 0.5)      ⇒ 1.73205
pow(3.0, -3.0)     ⇒ 0.037037
```

sqrt函数计算平方根：

```
sqrt(3.0) ⇒ 1.73205
```

顺便提一下，由于通常sqrt函数的运行速度非常快，因此使用sqrt计算平方根比使用pow更好。

23.3.6　就近取整函数、绝对值函数和取余函数

```
double ceil(double x);
double fabs(double x);
double floor(double x);
double fmod(double x, double y);
```

www.TopSage.com

ceil函数（ceilling）返回一个double类型的值，这个值是大于或等于其参数的最小整数。floor函数则返回小于或等于其参数的最大整数：

```
ceil(7.1)   ⇒ 8.0
ceil(7.9)   ⇒ 8.0
ceil(-7.1)  ⇒ -7.0
ceil(-7.9)  ⇒ -7.0
floor(7.1)  ⇒ 7.0
floor(7.9)  ⇒ 7.0
floor(-7.1) ⇒ -8.0
floor(-7.9) ⇒ -8.0
```

525

换言之，ceil“向上舍入”到最近的整数，floor“向下舍入”到最近的整数。没有一个标准库函数用来就近舍入到最近的整数，但我们可以简单地使用ceil函数和floor函数来实现：

```
double round(double x)
{
  return x< 0.0 ? ceil(x-0.5) : floor(x+0.5);
}
```

fabs函数计算参数的绝对值：

```
fabs(7.1)   ⇒ 7.1
fabs(-7.1)  ⇒ 7.1
```

fmod函数返回第一个参数除以第二个参数所得的余数：

```
fmod(5.5, 2.2) ⇒ 1.1
```

C语言不允许对%运算符使用浮点操作数，不过fmod函数足以用来替代%运算符。

23.4 <ctype.h>: 字符处理

<ctype.h>提供了两类函数：字符测试函数（如isdigit函数，用来检测一个字符是否是数字）和字符大小写转换函数（如toupper函数，用来将一个小写字母转换成大写字母）。

虽然C语言并不要求我们使用<ctype.h>中的函数来测试字符或进行大小写转换，但我们仍建议使用<ctype.h>中定义的函数来进行这类操作。第一，这些函数已经针对运行速度进行过优化（实际上，大多数都是用宏实现的）；第二，使用这些函数会使程序的可移植性更好，因为这些函数可以在任何字符集上运行；第三，当地点改变时（➤25.1节），<ctype.h>中的函数会相应地调整其行为，使我们编写的程序可以正确地运行在世界上不同的地点。

<ctype.h>中定义的函数都以int类型作为参数，并返回一个int类型的值。当然通常都可以忽略这种细节，因为需要时C语言可以自动将char类型的参数转换为int类型，或将int类型的返回值转换成char类型。

526

23.4.1 字符测试函数

```
int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
```

根据参数是否符合某种特性，每个字符测试函数都会返回0或1。表23-7列出了每个函数所测试的属性：

www.TopSage.com

表23-7　舍入模式

取　值	取值对应的舍入模式
isalnum(c)	c是否是字母或数字
isalpha(c)	c是否是字母
iscntrl(c)	c是否是控制字符①
isdigit(c)	c是否是十进制数字
isgraph(c)	c是否是可显示字符（除空格外）
islower(c)	c是否是小写字母
isprint(c)	c是否是可显示字符（包括空格）
ispunct(c)	c是否是标点符号②
isspace(c)	c是否是空白字符③
isupper(c)	c是否是大写字母
isxdigit(c)	c是否是十六进制数字

① 在ASCII字符集中，控制字符包括\0x00至\0x1f，以及\0x7f。
② 标点符号包括所有可显示字符中除掉空格，字母数字以外的其他字符。
③ 空白字符包括空格、换页符（\f）、换行符（\n）、回车符（\r）、横向制表符（\t）和纵向制表符（\v）。

23.4.2　程序：测试字符测试函数

下面的程序通过将字符测试函数应用于字符串"azAZ0 !\t"中的字符，来展示这些函数的作用。

tchrtest.c

```
/* Tests the character-testing functions */

#include <ctype.h>
#include <stdio.h>

527  #define TEST (f) printf (" %c ", f(*p) ? 'x' : ' ');
     main ( )
     {
        char *p ;
        printf ("      alnum     cntrl      graph      print"
                "      space     xdigit\n"
                "            alpha      digit      lower      punct"
                "      upper \ n" ) ;

        for  (p = "azAZ0  !\t";  *p  !=  '\0' ; p++){
           if  (iscntrl(*p))
              printf("\\x%02x:",  *p) ;
           else
              printf("    %c:",  *p) ;
           TEST (isalnum) ;
           TEST (isalpha) ;
           TEST (iscntrl) ;
           TEST (isdigit) ;
           TEST (isgraph) ;
           TEST (islower ) ;
           TEST (isprint) ;
           TEST (ispunct) ;
           TEST (isspace) ;
           TEST (isupper) ;
           TEST (isxdigit)
           printf("in") ;
        }
        return 0 ;
     }
```

程序产生的输出如下：

```
alnum     cntrl      graph      print      space     xdigit
     alpha      digit      lower      punct      upper
 a:  x    x            x      x    x                    x
```

www.TopSage.com

大学网 TopSage.com

```
   z:  x   x           x     x   x                          
   A:  x   x           x         x                  x    x
   Z:  x   x           x         x                  x
   0:  x           x   x         x                       x
    :                            x         x
   !:                x           x    x
\x09:          x                              x
```

23.4.3 字符大小写转换函数

```
int tolower(int c);
int toupper(int c);
```

tolower函数返回与作为参数的字母相对应的小写字母，而toupper函数返回与作为参数的字母相对应的大写字母。对于这两个函数，如果所传参数不是字母，那么将返回原始字符，不加任何改变。

528

23.4.4 程序：测试大小写转换函数

下面的程序对字符串"aA0!"中的字符进行大小写转换。

tcasemap.c

```
/* Tests the case-mapping functions */

#include <ctype.h>
#include <stdio.h>

main()
{
  char *p;

  for (p = "aA0!"; *p != '\0'; p++) {
    printf("tolower('%c') is '%c'; ", *p, tolower(*p));
    printf("toupper('%c') is '%c'\n", *p, toupper(*p));
  }
  return 0;
}
```

程序产生的输出如下：

```
tolower('a') is 'a'; toupper('a') is 'A'
tolower('A') is 'a'; toupper('A') is 'A'
tolower('0') is '0'; toupper('0') is '0'
tolower('!') is '!'; toupper('!') is '!'
```

23.5 <string.h>：字符串处理

第一次见到<string.h>是在13.5节，那一节讨论了最基本的字符串操作：strcpy（字符串复制）、strcat（字符串拼接）、strcmp（字符串比较）以及strlen（字符串长度计算）。接下来我们将看到，在<string.h>中还有许多其他字符串处理函数，以及一些对字符数组进行操作的函数。这些针对字符数组的函数不要求它们以空字符结尾。

<string.h>提供了5类函数：

- **复制函数**，将字符从内存中的一处复制到另一处。
- **拼接函数**，向字符串末尾追加字符。
- **比较函数**，用于比较字符数组的函数。
- **搜索函数**，在字符数组中搜索特定字符、字符组或字符串。
- **其他函数**，初始化字符数组或计算字符串的长度。

下面我们来逐一讨论每一类函数。

529

www.TopSage.com

23.5.1　复制函数

```
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
```

Q&A 这4个复制函数将字符（字节）从内存的一处（源）移动到另一处（目的）。每个函数都要求第一个参数指向目的，第二个参数指向源。所有的复制函数都会返回第一个参数（即指向目的的指针）。

memcpy函数从源向目的复制n个字符，其中n是函数的第三个参数。如果源和目的之间有重叠，memcpy函数的行为是未定义的。memmove函数与memcpy函数类似，只是当源和目的重叠时也可以正常工作。

strcpy函数将一个以空字符结尾的字符串从源复制到目的。strncpy与strcpy类似，只是它不会复制多于n个字符，其中n是函数的第三参数。（如果n太小，strncpy可能无法复制结尾的空字符。）如果strncpy遇到源字符串中的空字符，strncpy会向目的字符串不断追加空字符，直到写满n个字符为止。与memcpy类似，strcpy和strncpy不保证当源和目的相重叠时可以正常工作。

下面的例子展示了所有的复制函数。注释中给出了哪些字符会被复制。

```
char source[]= {'h',  'o',  't',  '\0',  't',  'e',  'a'  };
char dest[7];

memcpy(dest, source, 3);      /* h, o, t                        */
memcpy(dest, source, 4);      /* h, o, t, \0                    */
memcpy(dest, source, 7);      /* h, o, t, \0, t, e, a           */

memmove(dest, source, 3);     /* h, o, t                        */
memmove(dest, source, 4);     /* h, o, t, \0                    */
memmove(dest, source, 7);     /* h, o, t, \0, t, e, a           */

strcpy(dest, source);         /* h, o, t, \0                    */

strncpy(dest, source, 3);     /* h, o, t                        */
strncpy(dest, source, 4);     /* h, o, t, \0                    */
strncpy(dest, source, 7);     /* h, o, t, \0, \0, \0, \0        */
```

注意，memcpy、memmove和strncpy都不要求使用空字符结尾的字符串，它们对任意内存块都可以正常工作；然而strcpy函数则会持续复制字符，直到遇到一个空字符为止，因此strcpy仅用于以空字符结尾的字符串。

530

23.5.2　拼接函数

```
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
```

strcat函数将它的第二个参数追加到第一个参数的末尾。两个参数都必须是以空字符结尾的字符串。strcat函数会在拼接后的字符串末尾添加空字符。考虑下面的例子：

```
char str[7] = "tea";

strcat(str, "bag");  /* adds b, a, g, \0 to end of str */
```

字母b会覆盖字符a后面的空字符，因此现在str包含字符串"teabag"。strcat函数会返回它的第一个参数（指针）。

strncat函数与strcat函数基本一致，只是它的第三个参数会限制复制的字符的个数：

```
char str[7 = "tea"

strncat(str, "bag", 2);    /* adds b, a, \0 to str */
```

www.TopSage.com

```
strncat(str, "bag", 3);    /* adds b, a, g, \0 to str */
strncat(str, "bag", 4);    /* adds b, a, g, \0 to str */
```

正如上面例子所示，strnact函数会保证其结果字符串始终以空字符结尾。

23.5.3 比较函数

```
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
size_t strxfrm(char *s1, const char *s2, size_t n);
```

我们会将比较函数分为两组讨论。第一组中的函数（memcmp函数、strcmp和strncmp函数）比较两个字符数组。比较是按照计算机自身的排序顺序（通常为ASCII）对每个字符逐一进行的。第二组中的函数（strcoll函数和strxfrm函数）在需要考虑本地化（➤25.1节）时使用。

memcmp函数、strcmp函数和strncmp函数有许多共同的特性。这三个函数都需要指向字符数组的指针作为参数，然后用第一个字符数组中的字符逐一地与第二个字符数组中的字符进行比较。这三个函数都是在遇到第一个不匹配的字符时返回。另外，这三个函数都是根据比较结束时第一个字符数组中的字符是小于、等于或大于第二个字符数组中的字符，而相应地返回一个负整数、0或正整数。

531

这三个函数之间的差异在于如果数组相同，何时停止比较。memcmp函数包含第三个参数n，n会用来限制参与比较的字符个数，但memcmp函数不会关心空字符。strcmp函数没有对字符数设定限制，因此会在其中任意一个字符数组中遇到空字符时停止比较。（因此，strcmp函数只能用于以空字符结尾的字符串。）strncmp结合了memcmp和strcmp，当比较的字符数达到n个或在其中任意一个字符数组中遇到空字符时停止比较。

下面的例子解释了memcmp函数、strcmp函数和strncmp函数的上述特性：

```
char s1[] == {'b',  'i',  'g',  '\0',  'c',  'a',  'r'};
char s2[] == {'b',  'i',  'g',  '\0',  'c',  'a',  't'};

if  (memcmp(s1,  s2,  3) == 0)  ...      /*  true     */
if  (memcmp(s1,  s2,  4) == 0)  ...      /*  true     */
if  (memcmp(s1,  s2,  7) == 0)  ...      /*  false    */

if  (strcmp(s1,  s2) == 0) ...           /*  true     */

if  (strncmp(s1,  s2,  3)  == 0)  ...    /*  true     */
if  (strncmp(s1,  s2,  4)  == 0)  ...    /*  true     */
if  (strncmp(s1,  s2,  7)  == 0)  ...    /*  true     */
```

strcoll函数与strcmp函数类似，但比较的结果依赖于当前的本地化设置（通过调用setlocale函数（➤25.1.2节）设定）。对于那些根据程序运行的地点不同而可能按不同方式比较的程序，strcoll函数会比较有用。

大多数情况下，strcoll都足够用来处理依赖本地化设置情况下的字符串比较。但有些时候，我们可能需要多次进行比较（strcoll的一个潜在问题是，它不是很快），或者需要改变本地化设置但不希望影响比较的结果。在这些情况下，strxfrm函数（“字符串转换”）可以用来代替strcoll使用。

strxfrm函数会对它的第二个参数（一个字符串）进行转换，将转换的结果放在第一个参数所指向的字符串中。第三个参数用来限制向数组输出结果的字符个数。对带有两个转换后的字符串调用strcmp函数所产生的结果（负、0或正）与使用原始字符串调用strcoll函数的结果相同。

strxfrm函数返回转换后字符串的长度。因此strxfm函数通常会被调用两次：一次用于判断转换后字符串的长度，一次用来进行转换。下面是一个例子：

www.TopSage.com

大家网 TopSage.com

```
size_t len;
char *transformed;

len = strxfrm(NULL, original, 0);
transformed = malloc(len+1);
strxfrm(transformed, original, len);
```

532

23.5.4 搜索函数

```
void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
```

strchr函数在字符串中搜索指定字符。下面的例子说明了如何使用strchr函数在字符串中搜索字母f：

```
char *p, str[] = "Form follows function.";

p = strchr(str, 'f');   /* finds first 'f' */
```

strchr函数会返回一个指针，这个指针指向str中出现的第一个f（单词follows中的f）。如果需要多次搜索字符也很简单，例如，可以使用下面的调用搜索第二个f（即单词function中的f）：

```
p = strchr(p+1, 'f');   /* find next 'f' */
```

memchr函数与strchr函数类似，但memchr函数会在搜索了指定数量的字符后停止搜索，而不是当遇到首个空字符时才停止。memchr函数的第三个参数用来限制搜索时需要检测的字符总数。当不希望对整个字符串进行搜索或搜索的内存块不是以空字符结尾时，memchr函数会十分有用。下面的例子用memchr函数在一个没有以空字符结尾的字符数组中进行搜索：

```
char *p, str[22] = "Form follows function.";

p = memchr(str, 'f', sizeof(str));
```

与strchr函数类似，memchr函数也会返回一个指针，指向该字符第一次出现的位置。如果没有找到所查找的字符，两个函数都会返回空指针。

strrchr函数与strchr类似，但会反向搜索字符：

```
char *p, str[] = "Form follows function.";

p = strrchr(str, 'f');   /* finds last 'f' */
```

在此例中，strrchr函数会首先找到字符串末尾的空字符，然后反向查找字母f（单词function中的f）。如果找不到指定的字符，strrchr函数也会返回空指针。

strpbrk函数比strchr函数更通用。它返回一个指针，该指针指向第一个实际参数中与第二个实参中任意一个字符匹配的最左边一个字符：

533

```
char *p, str[] = "Form follows function.";

p = strpbrk(str, "mn");   /* finds first 'm' or 'n' */
```

在此例中，p最终会指向单词Form中的字母m。与其他搜索函数一样，当找不到指定的字符时，函数会返回空指针。

strspn函数和strcspn函数与其他的搜索函数不同，它们会返回一个表示字符串中特定位置的整数（size_t类型）。Q&A 当给定一个需要搜索的字符串以及需要搜索的字符集时，strspn函数返回字符串中第一个不属于给定字符集中的字符的下标。对于同样的参数，strcspn函数返回第一个属于给定字符集中的字符的下标。下面是使用两个函数的例子：

www.TopSage.com

```
size_t len;
char str[] = "Form follows function.";

len = strspn(str, "morF");     /* len = 4 */
len = strspn(str, " \t\n");    /* len = 0 */
len = strcspn(str, "morF");    /* len = 0 */
len = strcspn(str, " \t\n");   /* len = 4 */
```

strstr函数在第一个参数（字符串）中搜索第二个参数（也是字符串）。在下面的例子中，strstr函数搜索单词fun：

```
char *p, str[] = "Form follows function.";

p = strstr(str, "fun");  /* locates "fun" in str */
```

strstr函数返回指向在字符串中找到的第一处匹配子串的指针。如果找不到，则返回空指针。在上例的调用后，p会指向function中的字母f。

strtok函数是最复杂的搜索函数。它的目的是在一个字符串中搜索一个“记号”——就是一系列不包含特定分割符的字符。调用strtok(s1, s2)会在s1中搜索，从而查找一系列非空字符，这些字符中不包含s2中所指定的字符。在找到的记号后面的那个字符的位置，strtok函数会写入一个空字符来标记这个记号的末尾，然后返回一个指向这个记号的首字符的指针。

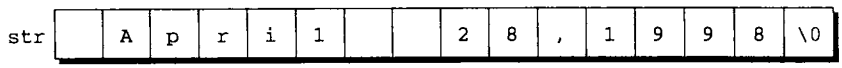
对于strtok函数最有用的特点是可以稍后调用strtok函数在同一字符串中搜索其他的记号。调用strtok(NULL, s2)就可以继续上一次的strtok函数调用。和上一次调用一样，strtok函数会用一个空字符来标记新的记号的末尾，然后返回然后返回一个指向新记号的首字符的指针。这个过程可以持续进行，直到strtok函数返回一个空指针，表明找不到符合要求的记号。

为了更清楚地解释strtok函数工作的原理，我们来使用它从如下书写的日期中：

month day' year

534

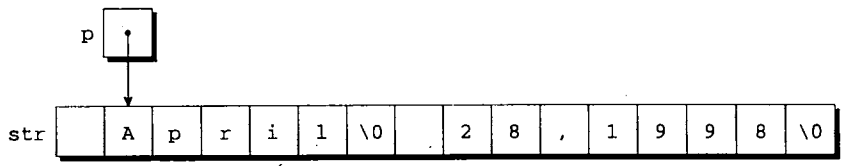
抽取出月、日和年。其中月与日之间，及日与年之间以空格或制表符分割。此外，空格或制表符前可能有逗号字符。假定开始时，字符串str有如下形式：



在调用后

```
p = strtok(str, " \t");
```

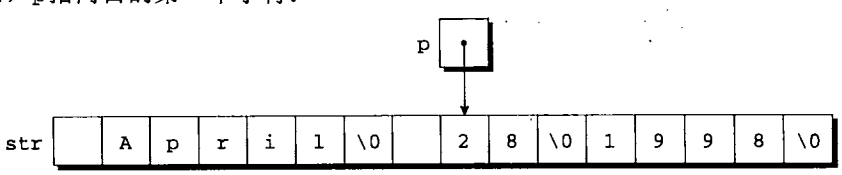
字符串str的形式如下：



p指向月的第一个字符，同时一个空字符标志着月的末尾。使用空指针作为strtok函数的第一个参数调用，会从上次结束的位置继续查找：

```
p = strtok(NULL, " \t");
```

这个调用后，p指向日的第一个字符：

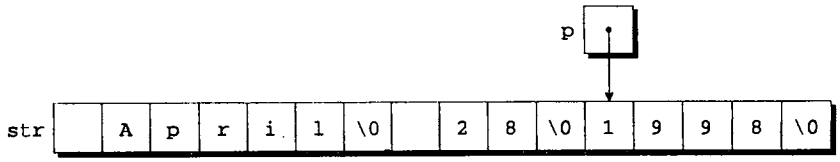


strtok函数的最后一个调用用来找到年：

www.TopSage.com

```
p = strtok(NULL, " \t,");
```

这次调用后，str的形式如下所示：



当重复调用strtok函数将一个字符串分割成记号时，对每次调用第二个参数并不需要保持一致。在我们的例子中，strtok函数的最后一次调用使用" \t,"替代了" \t"。

535

23.5.5 其他函数

```
void *memset(void *s, int c, size_t n);
size_t strlen(const char *s);
```

memset函数会将一个字符的多个副本存储到指定的内存区域。假设p指向一块N个字节的内存，例如，调用

```
memset(p, ' ', N);
```

会在这块内存的每个字节中存储空格。memset函数的一个用途是将数组全部初始化为0：

```
memset(a, 0, sizeof(a));
```

memset函数会返回它的第一个参数（指针）。

strlen函数返回字符串的长度，字符串末尾的空字符不计算在内。请见13.5节中使用strlen函数调用的例子。

此外还有一个与字符串相关的函数——strerror函数（➤24.2节），我们会和<errno.h>一起讨论。

问与答

问：为什么<string.h>中提供了那么多方法来做同一件事呢？我们真的需要4个复制函数（memcpy、memmove、strcpy和strncpy）吗？（p.368）

答：我们先看memcpy函数和strcpy函数，使用这两个函数的目的是不同的：strcpy函数只会复制一个以空字符结尾的字符数组（也就是字符串）；memcpy函数可以复制任意内存区域，而不需要这样的终止符（例如整数数组）。

另外两个函数可以使我们在运行速度和安全性中做出选择。strncpy函数比strcpy函数更安全，因为它限制了复制字符的个数。当然安全也是有代价的，因此strncpy函数也会比strcpy函数稍慢一点。使用memmove函数也需要做出类似的抉择。memmove函数可以将字符从一块内存区域复制到另一块可能会重叠的内存区域中。在同样情况下，memcpy函数无法保证能够正常工作。然而如果我们可以确保没有重叠，memcpy函数很可能会比memmove函数要快一些。

问：为什么strspn函数有这么一个奇怪的名字？（p.70）

答：不将strspn函数的返回值理解为第一个不属于指定字符集合中的字符的下标，而是将它的返回值理解为是其中所有属于指定字符集合的字符的最长“跨度”长度。

536

练习

23.3节

1. 编写一个程序，使用下面的公式求方程$ax^2+bx+c=0$的根：

TopSage.com

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

程序需要提示a、b、和c的值，然后显示出x的两个解。（如果b^2-4ac的值小于0，那么程序需要显示一条信息，指出根是虚数。）

2. 扩展round函数，使它可以将x舍入成小数点后n位。例如，调用round(3.14159, 3)会返回3.142。*提示：*将x乘以10^n，舍入成最接近的整数，再除以10^n。确保你的函数对正数x和负数x都可以正常工作。

23.4节

3. 使用isalpha函数和isalnum函数编写一个函数，用来检查一个字符串是否符合C语言标识符的语法（也就是说，它由字母、数字和下划线组成，并以字母或下划线开始）。
4. 编写一个程序，将文件从标准输入复制到标准输出，删除所有空行（仅包含空白字符的行）。
5. 编写一个程序，将文件从标准输入复制到标准输出，将每个单词的首字母大写。

23.5节

6. 对于下面列举的每种情况，指出使用memcpy，memmove，strcpy和strncpy中哪一个函数最好。假定所列举的行为都是由一个函数调用完成的。
   (a) 将数组中的每个元素都“下移”一个位置，以便将第0个位置空出给新的元素。
   (b) 通过将后面的所有字符都前移一个字节来删除以空字符结尾的字符串中的第一个字符。
   (c) 将一个字符串复制到一个字符数组中，这个字符数组的大小可能不够存放整个字符串。如果数组太小，就将字符串截断，而且在字符数组的末尾不需要空字符作为结尾。
   (d) 将一个数组变量的内容复制到另一个数组变量中。
7. 在23.5节中阐述了如何反复调用strchr函数在字符串中找到所有出现的指定字符。那么能否通过反复调用strrchr函数反向找到所有出现的指定字符呢？
8. 使用strchr函数编写如下函数：

```
int numchar(const char *s, char ch);
```

函数numchar返回字符ch在字符串s中出现的次数。

9. 使用一个strchr函数调用来替换下面if语句中的测试条件：

```
if (ch == 'a' || ch == 'b' || ch == 'c') …
```

537

10. 使用一个strstr函数调用来替换下面if语句中的测试条件：

```
if (strcmp(str, "foo") == 0 || strcmp(str, "bar") == 0 ||
    strcmp(str, "baz") == 0) …
```

*提示：*将字符串字面量合并到一个字符串中，并使用一个特殊的字符分割它们。你的解决方案是否对str的内容有所依赖呢？

11. 编写一个程序，提示用户输入一系列单词，然后按相反的顺序显示出来。将输入按字符串的形式读入，然后使用strtok函数将它们重新分割成单词。
12. 编写一个memset函数的调用，将一个以空字符结尾的字符串s的最后n个字符替换为'!'字符。
13. 许多<string.h>的版本提供了额外的（非标准的）函数，例如下面列出的一些函数。使用标准C的特性给出每一个函数的实现。
    (a) strdup(s)——返回一个指针，该指针指向通过调用malloc函数获得的内存中保存的s的一个副本。如果没有足够的内存可分配，则返回空指针。
    (b) stricmp(s1, s2) ——与strcmp函数类似，但不考虑字母的大小写。
    (c) strlwr(s) ——将s中的大写字母转换为小写字母，其他字符不变；返回s。
    (d) strrev(s) ——反转字符串s中的字符顺序（空字符除外）；返回s。
    (e) strset(s, ch) ——将s用ch的复本填充；返回s。

538

www.TopSage.com

第24章

错误处理

编写无错程序的方法有两种，但只有用第三种方法写的程序才行得通。

虽然学习C语言的学生所编写的程序在遇到异常输入时经常无法正常运行，但真正商业用途的程序却必须“非常强壮”——能够从错误中恢复正常而不致于崩溃。为了使程序非常强壮，需要我们能够预见程序执行时可能遇到的错误，包括对每个错误进行检测，并为错误一旦发生时提供一种合适的行为。

本章讲述了两种在程序中检测错误的方法：通过调用assert函数（24.1节）以及通过查询errno变量（24.2节）。24.3节讲解如何使程序检测并处理称为**信号（signal）**的条件，一些信号用于表示错误。最后，24.4节探讨了setjmp和longjmp机制，它们在相应处理错误时经常用到。

错误的检测和处理并不是C语言的强项。C语言对运行时错误以多种形式表示，而没有提供一种统一的方式。而且，在C语言程序中，必须由程序员将检测错误的代码编写在程序代码中。因此，很容易忽略一些可能发生的错误。一旦这些被略掉的错误中有某个错误发生，程序经常可以继续运行，虽然不是很好。C++C++语言对C语言的这一弱点进行了改进，提供了一种新的处理错误的方式——**异常处理**（exception handling）。

24.1 <assert.h>：诊断

```
void assert(int expression);
```

assert函数声明在<assert.h>中。它使程序可以监控自己的行为，并提早检测可能会发
539 生的错误。

虽然assert函数实际上是一个宏，但它是按照函数的使用方式设计的。assert函数有一个参数，这个参数必须是一种“断言”——一个我们认为在正常情况下一定为真的表达式。每次执行assert函数时，都会检查其参数的值。如果参数的值不为0，assert函数会显示一条信息（显示到stderr（标准错误流）（➤22.1.1节）），并调用abort函数（➤26.2.5节）终止程序执行。

例如，假定文件demo.c声明了一个长度为N的数组a。而我们担心demo.c程序中的语句

```
a[i] = 0;
```

可能会由于i不在0~N-1而导致程序失败。我们可以使用assert宏在给a[i]赋值前检查这种情况：

```
a[i]:
assert(0 <= i && i < N) ;      /* checks subscript first */
a[i] = 0;                          /* now does the assignment */
```

如果i的值小于0或者大于等于N，程序在输出类似下面的消息后会终止：

```
Assertion failed: 0 <= i && i < N, file DEMO.C, line 109
```

标准C并不要求显示的消息和上面的格式完全一样。但是，标准C要求在显示的消息中指明传递给assert函数的参数（以文本格式）、包含assert调用的文件名以及assert调用所在的行号。

www.TopSage.com

assert有一个缺点：因为它引入了额外的检查，因此会增加程序的运行时间。偶尔使用一次assert可能对程序的运行速度没有很大影响；但在实时程序中，这种对运行时间的增加可能是无法接受的。因此，许多程序在测试过程中会使用assert调用，但当程序最终完成时就会禁止assert调用。要禁止assert调用很容易，只需要在包含<assert.h>之前定义宏NDEBUG即可：

```
#define NDEBUG
#include <assert.h>
```

NDEBUG宏的值不重要，只要定义了NDEBUG宏即可。一旦之后程序又有错误发生，可以去掉NDEBUG宏的定义来重新起用assert调用。

不要在assert调用中使用有副作用的表达式，或有副作用的函数调用。一旦禁止了assert调用，这些表达式将不再会被计算。考虑下面的例子：

```
assert((p = malloc(n+1)) != NULL);
```

一旦定义了NDEBUG，assert调用会被忽略并且malloc不会被调用。 540

24.2 <errno.h>: 错误

标准库中的一些函数通过向<errno..h>中声明的errno变量存储一个错误代码(一个正整数）来表示有错误发生。(error可能实际上是个宏。如果确实是宏，C语言标准要求它表示左值（➤4.2.2节），以便和变量一样使用。）大部分使用errno变量的函数集中在<math.h>，但也有一些在标准库的其他部分。

假设我们需要使用一个库函数，该库函数通过给errno赋值来产生程序运行出错的信号。在调用这个函数之后，我们可以检查errno的值是否为零。如果不为零，则表示在函数调用过程中有错误发生。举例来说，假如需要检查sqrt函数（求平方根）（➤23.3.5节）的调用是否出错，可以使用类似下面的代码：

```
errno = 0;
y = sqrt(x);
if (errno != 0 ) {
  fprintf(stderr, "sqrt error; program terminated. \n");
  exit(EXIT_FAILURE);
}
```

对于像sqrt这类可能会改变errno值的函数，在调用前将errno置零非常重要。虽然在程序刚开始运行时errno的值为零，但有可能在随后的函数调用中已经被改动了。库函数不会将errno清零，这是程序责任。

Q&A 当错误发生时，向errno中存储的值通常是EDOM或ERANGE。(这两个宏都定义在<errno.h>中。）这两个值分别代表两种在一数学函数调用时可能发生的错误：

- **定义域错误**(EDOM)：传递给函数的一个参数不属于函数的定义域。例如用负数作为sqrt的参数就会导致一个定义域错误。
- **取值范围错误**（ERANGE)：函数的返回值太大，无法用double类型的值表示。例如，用1000作为exp函数（➤23.3.4节）的参数就经常会导致一个取值范围错误，因为e^{1000}太大以致无法在大多数计算机上用double类型表示。

一些函数可能会导致这两种错误，我们可以用errno分别与EDOM和ERANGE比较后，确定究竟发生了哪种错误。

perror 函数和 strerror 函数

```
void perror(const char *s);
char *strerror(int errnum);
```

541

www.TopSage.com

大家网 TopSage.com

当库函数向errno存储了一个非零值时，我们可能会希望显示一条描述这种错误的信息。一种实现方式是调用perror函数（声明在<stdio.h>中），它会按如下顺序显示以下信息：（1）调用perror的参数，（2）一个分号，（3）一个空格，（4）一条出错消息，消息的内容根据errno的值决定，（5）一个换行符。perror函数会输出到stderr（➤22.1.1节），而不是标准输出。

下面的代码是一个使用perror的例子：

```
errno = 0;
y = sqrt(x);
if (errno != 0) {
  perror("sqrt error");
  exit(EXIT_FAILURE);
}
```

perror函数在sqrt error后所显示的出错消息是由实现定义的。下面是一种可能的形式：

```
sqrt error: Math argument
```

这里我们假定Math argument是与EDOM错误相对应的消息。而ERANGE错误则通常会对应于另一条消息，例如Result too large。

strerror函数定义在<string.h>中，它与perror关系紧密。当以错误代码调用strerror时，函数会返回一个指针，它指向一个描述这种错误的字符串。例如，调用

```
puts(strerror(EDOM));
```

可能会显示

```
Math argument
```

如果给strerror函数传递errno作为它的参数，那么函数perror所显示的出错消息与strerror所返回的信息是相同的。

24.3 <signal.h>：信号处理

<signal.h>提供了处理异常情况工具，即信号（signal）。信号有两种类型：运行时错误（例如除以0）和程序以外导致的事件。例如，许多操作系统都允许用户中断或终止运行的程序，在C语言中这些事件作为信号。当有错误或外部事件发生时，我们称**产生**了一个信号。大多数信号是**异步的**：它们可以在程序执行过程中的任意时刻发生，而不仅是在程序员所知道的特定时刻发生。
542 由于信号可能会在任何意想不到的时刻发生，因此必须用一种唯一的方式来统一处理它们。

24.3.1 信号宏

<signal.h>定义了一系列的宏，用于表示不同的信号。Q&A表24-1列出这些宏以及它们的含义。C语言的实现可以提供更多的信号宏，只要宏的名字以SIG开头并随其后使用大写字母组成。

表24-1 信号

宏 名	含 义
SIGABRT	异常终止（可能由于调用abort导致）
SIGFPE	在数学运算中发生错误（可能是除以0或溢出）
SIGILL	非法指令
SIGINT	中断
SIGSEGV	非法存储访问
SIGTERM	终止请求

www.TopSage.com

C标准并不要求表24-1中列出的信号都自动发生，因为对于某个特定的计算机或操作系统，不是所有的信号都有意义。大多数C语言的实现都至少支持其中的一部分。

24.3.2 signal 函数

```
void (*signal(int sig, void (*func)(int)))(int);
```

在<signal.h>中最重要的函数就是signal函数，它会安装一个信号处理函数，以便将来给定的信号发生时使用。signal函数的使用比它的原型看起来要简单得多。第一个参数是特定信号的代码，第二个参数是一个指向函数的指针，这个函数就是当信号发生时用来处理信号的函数。例如，下面的signal函数调用对SIGINT信号安装了一个处理函数：

```
signal(SIGINT, handler) ;
```

handler就是信号处理函数的函数名。一旦随后在程序执行过程中出现了SIGINT信号，handler函数就会自动被调用。

每个信号处理函数都必须有一个int类型的参数。当一个特定的信号出现并调用相应的处理函数时，信号的代码会作为参数传递给处理函数。知道是哪种信号导致了处理函数被调用是十分有用的，尤其是，它允许我们对多个信号使用同一个处理函数。

信号处理函数几乎可以做所有它想做的事。这可能包含忽略该信号、执行一些错误修复或终止程序。然而，除非信号是由调用abort函数或raise函数引发的，否则信号处理函数不应该调用任何库函数，或试图使用一个静态存储期限的变量。

一旦信号处理函数返回，程序会从信号发生点恢复并继续执行。但是，有一些特殊情况。如果信号是SIGABRT，当处理函数返回时程序会终止（异常地）。如果信号是SIGFPE，那么处理函数返回的结果是未定义的。（也就是说，不要用它。） 543

虽然signal函数有返回值，但经常被忽略。返回值是指向对于指定信号的前一个处理函数的指针。如果需要，可以将它保存在变量中。特别是，如果我们打算恢复原来的处理函数，那么就需要保留signal函数的返回值：

```
void (*orig_handler)(int);     /* function pointer */
orig_handler = signal(SIGINT, handler);
```

上面的调用将handler函数设置为SIGINT的处理函数，并将指向原始的处理函数的指针保存在变量orig_handler中。如果要恢复原来的处理函数，我们需要使用下面的代码：

```
signal(SIGINT, orig_handler);  /* restores original handler */
```

24.3.3 预定义的信号处理函数

除了编写我们自己的信号处理函数，我们还可以选择使用<signal.h>提供的预定义的处理函数。有两个这样的函数，每个都是用宏表示的：

- **SIG_DFL**。函数SIG_DFL按“默认”方式处理信号。可以使用这样的调用安装SIG_DEF

  ```
  signal(SIGINT, SIG_DFL);     /* use default handler */
  ```

  调用SIG_DFT的结果是由实现定义的，但大多数情况下会导致程序终止。

- **SIG_IGN**。调用

  ```
  signal(SIGINT, SIG_IGN);     /* ignore SIGINT signal */
  ```

  指明随后当信号SIGINT发生时，忽略该信号。

除了SIG_DFL和SIG_IGN，<signal.h>可能还会提供其他的信号处理函数其函数名必须是以SIG_开头并随其后使用大写字母组成。当程序刚开始执行时，根据不同的实现，每个信号的处理函数都会被初始化为SIG_DFL或SIG_IGN。

<signal.h>还定义了另一个宏——SIG_ERR它看起来像是个信号处理函数。实际上，

www.TopSage.com

SIG_ERR根本不是处理函数，它是用来在安装处理函数时检测是否发生错误的宏。如果一个signal调用失败（即能不对所指定的信号安装处理函数），就会返回SIG_ERR并在errno中存入一个正值。因此，为了测试signal调用是否失败，可以使用如下代码：

```
if (signal(SIGINT, handler) == SIG_ERR) {
  /* error; can't install handler for SIGINT */
}
```

544

在整个信号处理机制中，有一个特殊的技巧：如果信号是由处理这个信号的函数引发的会怎样呢？为了避免无限递归，C语言要求——除了SIGILL以外，当一个信号的处理函数被调用时，该信号对应的处理函数要被重置为SIG_DFL（默认处理函数）或以其他方式加以封锁。（我们无法控制这一过程，因为一切都是在后台执行的。）

信号处理完之后，除非处理函数被重新安装，否则该信号不会被同一个函数处理两遍。Q&A当然，一种实现方法是在处理函数返回前调用signal函数。

24.3.4 raise函数

```
int raise(int sig);
```

虽然通常信号都是自然产生的，但有时候如果程序可以触发信号，就会非常方便。raise函数就可以实现这一目的，而且它也是包含在<signal.h>中的函数。raise函数的参数指定所描述信号的代码：

```
raise(SIGABRT);   /* raises the SIGABRT signal */
```

raise函数的返回值可以用来测试调用是否成功：0代表成功，非0则代表失败。

24.3.5 程序：测试信号

下面的程序说明了如何使用信号。首先，给SIGILL信号安装了一个惯用的处理函数（并小心地保存了原先的处理函数），然后调用raise_sig产生一个信号；其次，程序将SIG_IGN设置为SIGILL的处理函数并再次调用raise_sig；最后，它将信号SIGILL原先的处理函数重新安装，并最后调用一次raise_sig。

tsignal.c

```
/* Tests signals */

#include <signal.h>
#include <stdio.h>

void handler(int sig);
void raise_sig(void);

main()
{
  void (*orig_handler)(int);

  printf("Installing handler for signal %d\n", SIGILL);
  orig_handler = signal(SIGILL, handler);
  raise_sig();

  printf("Changing handler to SIG_IGN\n");
  signal(SIGILL, SIG_IGN);
  raise_sig();

  printf("Restoring original handler\n");
  signal(SIGILL, orig_handler);
  raise_sig();
```

545

www.TopSage.com

TopSage.com

```
  printf("Program terminates normally\n");
  return 0;
}

void handler(int sig)
{
  printf("Handler called for signal %d\n", sig);
}

void raise_sig(void)
{
  raise(SIGILL);
}
```

当然，调用raise并不需要在单独的函数中。这里定义raise_sig函数只是为了说明一点：无论信号是从哪里发出的（无论是在main函数中还是在其他函数中）它都会被最近安装的处理函数捕获。

由于在C标准中有一大部分信号处理机制是未定义的，因此上述程序的输出可能会有所不同。下面是一种可能的输出形式：

```
Installing handler for signal 4
Handler called for signal 4
Changing handler to SIG_IGN
Restoring original handler
```

从这个输出的结果中，我们看到SIGILL的值为4，而且最初SIGILL的处理函数一定是SIG_DFL。（如果是SIG_IGN，我们应该会看到信息Program terminates normally）最后，那么可以注意到SIG_DFL会导致程序终止，但不会显示出错误消息。

24.4 <setjmp.h>：非局部跳转

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

546

通常情况下，函数会返回到它被调用的位置。我们无法使用goto语句是它转到其他地方，因为goto只能跳转到同一函数内的标号处。但是<setjmp.h>可以使一个函数直接跳转到另一个函数，而不需要返回。

在<setjmp.h>中最重要的内容就是setjmp宏和longjmp函数。setjmp宏“标记”程序中的一个位置；随后可以使用longjmp跳转到该位置。虽然这一强大的机制可以有多种潜在的用途，它主要被用于错误处理。

如果要为将来的跳转标记一个位置，可以调用setjmp宏，调用的参数是一个jmp_buf类型的变量（同样定义在<setjmp.h>中）。Q&A setjmp宏会将当前“环境”（包括一个，指向setjmp宏自身被调用的位置的指针）保存到变量中以便随后可以在调用longjmp函数时使用，然后返回0。

如果要返回setjmp宏所标记的位置可以使用longjmp函数，调用的参数是我们在调用setjmp宏时使用的同一个jmp_buf类型的变量。longjmp函数会首先根据jmp_buf变量的内容恢复当前环境，然后从setjmp宏调用中返回——这是最难以理解的。这次setjmp宏的返回值是val，就是调用longjmp函数时的第二个参数。（但是如果val的值为0，那么setjmp宏会返回1。）

一定要确保作为longjmp函数的参数已经被setjmp初始化了。否则，调用longjmp会导致未定义的行为。（程序很可能会崩溃。）

总而言之，setjmp会在第一次调用时返回0；随后，longjmp将控制权重新转给最初的setjmp

www.TopSage.com

宏调用，而setjmp在这次调用时会返回一个非零值。明白了吗？看来我们可能需要个例子...

程序：测试 setjmp 和 longjmp

下面的程序使用setjmp宏在main函数中标记一个位置；然后函数f2通过调用longjmp函数返回到这个位置。

tsetjmp.c

```
/* Tests setjmp/longjmp */

#include <setjmp.h>
#include <stdio.h>

static jmp_buf env;

void f1(void);
void f2(void);

main()
{
  int ret;

  ret = setjmp(env);
  printf("setjmp returned %d\n", ret);
  if (ret != 0) {
    printf("Program terminates: longjmp called\n");
    return 0;
  }
  f1();
  printf("Program terminates normally\n");
  return 0;
}

void f1(void)
{
  printf("f1 begins\n");
  f2();
  printf("f1 returns\n");
}

void f2(void)
{
  printf("f2 begins\n");
  longjmp(env, 1);
  printf("f2 returns\n");
}
```

547

程序的输出如下：

```
setjmp returned 0
f1 begins
f2 begins
setjmp returned 1
Program terminates:longjmp called
```

setjmp宏的最初调用返回0，因此main函数会调用f1。接着，f1调用f2，f2使用longjmp函数将控制权重新转移给main函数，而不是返回到f1。当longjmp函数被执行时，控制权重新回到setjmp宏调用。这时，setjmp宏返回1（就是在longjmp函数调用时所指定的值）。

问与答

问：我使用的<errno.h>版本中除了EDOM和ERANGE以外，还定义了其他的宏。这是合法的吗？(p.375)

答：是合法的。C标准允许使用宏表示其他错误条件，只要宏的名字以字母E开头并随其后使用数字或大

www.TopSage.com

写字母组成。

问：一些表示信号的宏的名字含义比较模糊，比如SIGFPE和SIGSEGV。这些名字是如何得来的呢？(p.376)

答：信号的名字可以追溯到早期的C语言编译器，它们运行在DEC PDP-11计算机上。PDP-11的硬件可以检测一些错误，诸如“Floating Point Exception”和“Segmentation Violation”。 548

***问：我注意到在<signal.h>中有一个叫sig_atomic_t的类型。它的作用是什么？**

答：sig_atomic_t是一个整数类型。按照C标准，它可以“作为一个基本元素”使用。换言之，CPU可以使用一条机器指令从内存中获取它的值或存储它的值，而不用两条甚至更多的机器指令。sig_atomic_t通常被定义为int，因为大多数CPU都可以只用一条指令装载或存储一个整数。

通常，一个信号处理函数不应该访问有静态存储期限的变量。但是C标准允许一种例外的情况：信号处理函数可以向sig_atomic_t类型的变量存入一个值，只要该变量被声明为volatile（类型限定符）。（➤20.3.5节）想要理解这条奇特规则的原因，可以考虑一下如果信号处理函数试图修改一个比sig_atomic_t类型大的变量会发生什么情况？如果程序在信号发生前从内存中获取了这个变量的一部分，然后在信号处理后获取余下的部分，那么程序可能以一个无用的值终止。sig_atomic_t类型的变量只需要用一条语句获取，而且volatile类型的变量每次使用时必须重新获取，因此上述问题就不会发生了。

问：如果信号处理函数不支持调用库函数，那么信号处理函数又怎么能调用signal函数来重新安装它自己呢？(p.378)

答：C标准允许这一例外发生。信号处理函数可以合法地调用singal函数，只要第一个参数是当前正在被处理的信号就可以。

问：程序tsignal在信号处理函数内调用了printf函数。这不是非法的吗？

答：这是C标准所允许的另一个例外：如果信号处理函数是由raise或abort调用的，那么就可以调用库函数。

问：setjmp会如何修改传递给它的参数呢？C语言不是始终以值的形式传递参数吗？(p.379)

答：C标准要求jmp_buf必须是一个数组类型，因此传递给setjmp的实际上是一个指针。

问：我在使用setjmp时，程序有时无法通过编译。这是什么问题？

答：按照标准C，只有两种使用setjmp的方式是合法的：

- 作为表达式语句（可能会强制转换成void）。
- 作为if、swtich、while、do或for语句中控制表达式的一部分。整个控制表达式必须符合下面的形式之一。（*constexp*是一个计算结果为整数的常量表达式，并且*op*是关系或判等运算符。）

```
setjmp(…)
!setjmp(…)
constexp op setjmp(…)
setjmp(…) op constexpop
```

549

一些编译器允许不符合这些规则的 setjmp 调用。但是如果不遵守这些规则，程序就不是可移植的。

问：调用longjmp函数后，程序中变量的值是什么？

答：大部分变量的值保留了longjmp函数被调用时的值。然而，包含setjmp宏的函数中的自动变量的值是不确定的，除非该变量被声明为volatile或者在执行setjmp后没有被修改过。

问：在信号处理函数里调用longjmp函数合法吗？

答：是合法的，只要信号处理函数的调用不是由于在信号处理函数执行过程中触发的信号。

练习

24.1节

1. （a）断言可以用来检测两种问题：（1）如果程序正确执行就不应该发生的问题；（2）超出程序控制范围之外的问题。请解释为什么assert更适用于第一类问题？

大家网 TopSage.com

(b) 请举出3个超出程序控制范围之外的问题的例子。

24.2节

2. (a) 编写一个名为try_math_fcn的“包装”函数，用来调用数学函数（假定有一个double类型的参数，并返回一个double类型的值），然后检测调用是否成功。下面是使用try_math_fnc函数的例子：

```
y = try_math_fnc(sqrt, x, "Error in call of sqrt");
```

如果调用sqrt(x)成功，try_math_fnc返回sqrt函数计算的结果。如果调用失败，try_math_ fnc需要调用perror显示消息Error in call of sqrt，然后调用exit函数终止程序。

(b) 编写一个与try_math_fnc具有相同的效果的宏，但是要求使用函数的名字来构造出错消息：

```
y = TRY_MATH_FNC(sqrt, x);
```

如果调用sqrt失败，显示的出错消息应该是“Error in call of sqrt”。提示：让TRY_MATH_FNC调用try_math_fnc。

24.3节

3. 给SIGINT编写一个信号处理函数，用来记录它被调用了多少次。要求处理函数必须忽略前两次发生的信号，并在第三次发生时终止程序（通过调用exit）。

24.4节

4. 在invent程序中（16.3节），main函数中用一个for循环来提示用户输入一个操作代码，读入代码，然后根据代码调用insert、search、update或print。以这种方法在main函数中加入一个setjmp调用，要求使随后的longjmp调用会返回到for循环。（在调用longjmp函数后，用户会被提示输入一个操作码，随后程序正常执行。）setjmp宏需要一个jmp_buf类型的变量，这个变量应该在哪儿声明呢？

www.TopSage.com

第25章

国际化特性

如果您的计算机说英语，那么它可能产自日本。

在最初设计时，C语言并不十分适合在多个国家使用。经典C假定字符都是单字节的，并且所有计算机都识别字符#、[、\、]、^、{、|、}和~，因为这些字符都需要在C程序中用到。遗憾的是这些假定并不是在世界的任何地方都适用。在20世纪80年代创造标准C的专家意识到了将C语言国际化的重要性。本章描述他们给C语言添加的特性和函数库，而这些给全世界的程序员带来了帮助。

<locale.h>（25.1节）提供了允许程序员针对特定的“地区”（可能是国家、洲或省或者一种特定的文化）删减程序行为的函数。多字节字符和宽字符（25.2节）使程序可以工作在更大的字符集上，例如亚洲国家的字符集。三字符序列（25.3节）使我们可以在一些不支持某些C语言编程中常用字符的机器上编写程序。

在1994年C语言对整个国际社会的重要性得到了强调，在这一年针对ISO C标准的修正草案1批准通过。这一提案提出了为编写国际化程序增加的额外库，包括<iso646.h>、<wctype.h>以及<wchar.h>。由于这些内容还没有被广泛使用，所以本书不准备再讨论修正草案1的细节了。如果想了解更多细节，可以参考Harbison和Steele的*C: A Reference Manual*（第4版）（Englewood Cliffs，N.J.：Prentice-Hall，1995）。

25.1 <locale.h>：本地化

<locale.h>提供的函数用于控制标准库中对于不同的地点会不一样的部分。**地区**通常是一个国家，但并不需要一定如此。例如，一个国家的不同区域也可能被作为单独的地区来对待。地区甚至可以代表同一区域的不同文化。 551

在标准库中，依赖地区的部分包括：

- **数值的格式**。例如在一些地区，小数点是一个圆点（297.48），而在另一些地方则是逗号（297,48）。
- **货币的格式**。例如，不同国家的货币符号不同。
- **字符集**。字符集通常依赖于特定地区的语言。亚洲国家通常比西方国家需要更大的字符集。
- **日期和时间的表示形式**。例如，一些地方习惯在写日期时先写月（8/24/97），而另一些地方习惯先写日（24/8/97）。

25.1.1 类别

通过修改地区，程序可以改变它的行为来适应世界的不同区域。但地区改动可能会影响库的许多方面，其中一部分可能是我们不希望改变的。幸好，我们不需要同时对库的所有部分进行改变。实际上，可以使用下列宏中的一种来指定一个**类型**：

- LC_COLLATE。影响两个字符串比较函数（strcoll和strxfrm）的行为。（两个函数都

www.TopSage.com

声明在<string.h>（➤23.5节）中。）

- LC_CTYPE。影响<ctype.h>（➤23.4节）中函数（除了isdigit和isxdigit）的行为。同时还影响<stdlib.h>中的多字节函数（➤25.2.1节）。
- LC_MONETARY。影响由localeconv函数返回的货币格式信息。不影响任何库函数的行为。
- LC_NUMERIC。影响格式化输入/输出函数（例如printf和scanf）使用的小数点字符以及<stdlib.h>中的字符串转换函数（atof和strtod）（➤26.2.1节），还会影响localeconv函数返回的非货币格式信息。
- LC_TIME。影响strftime函数（在<time.h>中声明）（➤26.3.2节）的行为，该函数将时间转换成字符串。

C语言的实现提供了其他类型并且定义了上面未列出的以LC_开头的宏。

25.1.2 setlocale函数

552
```
char *setlocale(int category, const char *locale);
```

setlocale函数修改当前的地点，可以是针对一个类型的，也可以是针对所有类型的。如果setlocale调用的第一个参数是LC_COLLATE、LC_CTYPE、LC_MONETARY、LC_NUMERIC或LC_TIME之一，那么改变就只影响一个类型。如果第一个参数是LC_ALL，调用就会影响所有类型。C语言标准对第二个参数仅定义了两种可能值："C"和" "。其他的地区可以针对不同的C实现定义。

在任意程序执行开始时，都会隐含执行调用

```
setlocale(LC_ALL, "C");
```

当地点设置为"C"时，库函数按正常方式执行，小数点是一个句点。

如果在程序运行起来后想改变地点，就需要显式调用setlocale函数。用" "作为第二个参数调用setlocale函数可以切换到**本地模式**（native locale）。这种模式下程序会适应本地的环境。C语言标准并没有定义切换到本地模式的具体影响。一些setlocale函数实现的会检查当前的运行环境（与getenv函数（➤26.2.5节）的方式一样），查找特定名字（可能是与表示类型的宏同名）的环境变量。而另一些实现根本什么都不做。（C语言标准并没有要求setlocale有什么特定的作用。当然，如果库中的setlocale什么都不做，那么这个库在世界的一些地区它可能不会卖得很好。）

对于除"C"和" "以外的其他地点我们可能无法提供更多介绍，因为它们在不同的编译器之间可能有很大的差异。一些编译器可能不会提供任何其他地点。另一些编译器则可能会提供名为"Germany"的设置。一种常用的编译器使用类似"en_GB.WIN1252"的复杂字符串作为地点。其中en指定语言（English），GB是国家（Great Britain），WIN1252是字符集（Windows多语言字符集）。

当setlocale函数调用成功时，它会返回一个指向字符串的指针，这个字符串与新地点的类型相关联。（例如，这个字符串可能就是地点名字自身。）如果调用失败，setlocale函数返回空指针。

setlocale函数也可以当作搜索函数使用。如果第二个参数是空指针，setlocale函数会返回一个指向字符串的指针，这个字符串与当前地区类型的设置相关联。这一特性在将第一个参数设为LC_ALL时特别有用，因为这时可以获取对应于所有类型的当前设置。Q&A setlocale函数返回的字符串可以（通过复制到变量中）被保存起来以便以后调用setlocale函数时使用。

25.1.3 localeconv函数

```
struct lconv *localeconv(void);
```

虽然可以通过调用setlocale函数来获取当前地区的信息，但是setlocale函数可能不是
553 以最有效的形式返回信息的。为了找到关于当前地区的有效说明信息（小数点字符是什么？货

www.TopSage.com

币符号是什么？），就需要声明在<locale.h>中的另一个唯一的函数——localeconv函数。

localeconv函数返回指向struct lconv类型结构的指针，且指向的结构包含当前地区的详细信息。此结构具有静态存储期限，而且稍后通过localeconv函数或者setlocale函数调用还可以对此结构进行修改。请一定要确信在上述函数之一擦除结构信息之前，已经从lconv结构中摘取了需要的信息。

lconv结构中的一些成员具有char*类型，而另一些成员则具有char类型。表25-1列出了结构中所有char*类型的成员，其中前3个成员用来处理非货币型数值的格式，而其他成员则处理货币型数值。此表还说明了每个成员在"C"地区中（默认情况下）的值，其中""意味着“无效的”。

表25-1 lconv结构的char*类型的成员

	名 称	在"C"地区中的值	描 述
非货币类的	decimal_point	"."	十进制小数点字符
	thousands_sep	""	在十进制小数点前，用来分隔数字组的字符
	grouping	""	数字组的大小尺寸
货币类的	int_curr_symbol	""	国际货币符号①
	currency_symbol	""	区域货币符号
	mon_decimal_point	""	十进制小数点字符
	mon_thousands_sep	""	在十进制小数点前，用来分隔数字组的字符
	mon_grouping	""	数字组的大小尺寸
	positive_sign	""	用来说明非负值的字符串
	negative_sign	""	用来说明负值的字符串

① 分隔符（常常是空格或者句点）后边跟着3个字母的缩写。例如，意大利、荷兰、挪威以及瑞士的国际货币符号分别是"ITL."、"NLG "、"NOK "和"CHF "。

这里需要特别说明一下成员grouping和成员mon_grouping。在这两个字符串中的每个字符都说明了每组数字的大小。（分组工作是从十进制小数点开始自右向左进行的。）CHAR_MAX的值说明没有进一步要执行的分组操作了；0说明前面的元素应该用于其余的数字。例如，字符串"\3"（\3的后边跟着\0）说明第一组应该有3个数字，然后所有其他数字也应该以3分组中。

表25-2列出了lconv结构中的char类型成员，并且说明了在"C"地区中每个成员的值；其中CHAR_MAX的值意味着“无效”。表25-2中的所有成员都必须处理的是货币型值的格式。表25-3说明了如何解释成员p_sign_posn和成员n_sign_posn的值。 554

表25-2 lconv结构的char类型的成员

名 称	在"C"地区中的值	描 述
int_frac_digits	CHAR_MAX	十进制小数点后的数字个数（国际格式）
frac_digits	CHAR_MAX	十进制小数点后的数字个数（区域格式）
p_cs_precedes	CHAR_MAX	如果currency_symbol先于非负值，则为1；如果currency_symbol继数值之后，则为0
p_sep_by_space	CHAR_MAX	如果currency_symbol是用空格来分隔非负值，则为1；否则为0
n_cs_precedes	CHAR_MAX	如果currency_symbol先于负值，则为1；如果currency_symbol继数值之后，则为0
n_sep_by_space	CHAR_MAX	如果currency_symbol是用空格来分隔负值，则为1；否则为0
p_sign_posn	CHAR_MAX	positive_sign的位置表明非负值（见表25-3）
n_sign_posn	CHAR_MAX	negative_sign的位置表明负值（见表25-3）

www.TopSage.com

表25-3　p_sign_posn和n_sign_posn的值

值	含　义
0	围绕在数量和currency_symbol周围的圆括号
1	在数量和currency_symbol之前的符号
2	继数量和currency_symbol之后的符号
3	直接在currency_symbol之前的符号
4	直接继currency_symbol之后的符号

为了说明lconv结构的成员如何随着地区的不同而不同，下面来比较两个假想的示例。表25-4显示了用于美国和意大利两国的lconv结构成员的货币型常用值（稍后的示例是来自C标准自身的）。

表25-4　用于美国和意大利两国的lconv结构成员的货币型常用值

成　员	美　国	意大利
int_curr_symbol	"USD "	"ITL."
currency_symbol	"$"	"L."
mon_decimal_point	"."	""
mon_thousands_sep	","	"."
mon_grouping	"\3"	"\3"
positive_sign	""	""
negative_sign	"-"	"-"
int_frac_digits	2	0
frac_digits	2	0
p_cs_precedes	1	1
p_sep_by_space	0	0
n_cs_precedes	1	1
n_sep_by_space	0	0
p_sign_posn	4	1
n_sign_posn	4	1

555

下面是7593格式化成上述两个区域货币型值的情况：

	美　国	意大利
正数格式	$7,593.00	L.7.593
负数格式	$-7,593.00	-L.7.593
国际化格式	USD 7,593.00	ITL.7.593

请记住C语言的库函数不能自动格式化货币型值，直到每个程序都使用lconv结构中的信息才可以完成格式化。

25.2　多字节字符和宽字符

程序在适应不同地区的过程中最大的难题之一就是字符集的问题。在美国，主流计算机使用ASCII字符集，而其他的多数使用EBCDIC。在美国以外的地方，情况变得更加复杂。在一些国家，计算机采用类似于ASCII的字符集，但是缺少了某些字符。25.3节将会进一步讨论这个问题。在亚洲的其他国家则面临不同的问题：书写的语言要求巨大的字符集，通常是以千计的。

因为定义已经把char型值的大小限制为一个字节，所以通过改变char类型的含义来处理更

www.TopSage.com

大的字符集显然是不可能的。取而代之的是，C语言允许编译器提供一种**可扩展的**字符集。这种字符集可以用于编写C程序（例如，在注释和字符串中），也可以用于程序运行的环境中，或者两种地方都有。Q&A C语言提供了两种用于可扩展字符集的编码：**多字节字符**（multibyte character）和**宽字符**（wide character）。C语言还提供了把一种编码转换成另外一种编码的函数。

25.2.1 多字节字符

在**多字节字符**编码中，一个或多个字节表示一个可扩展的字符。任何可扩展的字符集必须包含C语言要求的基本字符（即字母、数字、运算符、标点符号和空白字符）。这些字符都要求是单字节的。可以把其他字节解释为多字节字符的开始。

日文字符集

日文采用不同的写入系统。最复杂的是日文中的汉字（kanji），它由上千个符号组成，因为符号实在是太多了，以致于不能用单个字节编码表示，（日文中的汉字符号实际上源自中国的汉字，汉字也有一个和大字符集类似的问题。）没有统一的方法对日文中的汉字编码，常用的编码包括JIS（日本工业标准）、Shift-JIS和EUC（可扩展的UNIX编码）。 556

一些多字节字符集依靠**依赖状态编码**（state-dependent encoding）。在这类编码中，每个多字节字符序列都以**初始移位状态**（initial shift state）开始。序列中稍后遇到的一些多字节字符会改变移位状态，并且会影响后续字节的含义。例如，日本的JIS编码把单字节码与双字节码进行混合，而嵌入在字符串中的“转义序列”则说明了单字节模式和双字节模式互相切换的时间。（反之，Shift-JIS编码不是依赖状态的。每个字符要求一个或者两个字节，但是双字节字符的第一个字节可以始终区别于单字节字符。）

在任何编码中，无论移位状态如何，C标准都要求零字节始终用来表示空字符。而且，零字节不能是多字节字符的第二个（或者后一个）字节。

C语言库提供了两种与多字节字符相关的宏MB_LEN_MAX和MB_CUR_MAX，这两种宏说明了多字节字符中字节的最大数量。宏MB_LEN_MAX（定义在<limits.h>中）给出了任意支持区域的最大值，而宏MB_CUR_MAX（定义在<stdlib.h>中）则给出了当前区域的最大值。（改变地区可能会影响多字节字符的解释。）显然，宏MB_CUR_MAX不可能大过宏MB_LEN_MAX。

25.2.2 宽字符

另外一种对可扩展字符集进行编码的方法是使用**宽字符**（wide characters）。宽字符是一种其值表示字符的整数。不同于长度不同的多字节字符，采用特殊实现支持的所有宽字符都要求相同的字节数。

宽字符具有wchar_t类型（定义在<stddef.h>和<stdlib.h>中），且它必须是整数类型才可以表示任何支持地区的可扩展字符集。例如，如果两个字节足够表示任何可扩展字符集，那么将会把wchar_t定义成unsigned short int类型。

使用宽字符的一个好处是C语言支持宽字符常量和宽字符串字面量。宽字符常量类似于普通的字符常量，只是前者需要有字母L作为前缀：

```
L 'a'
```

而宽字符串字面量也需要用字母L作为前缀：

```
L"abc"
```

www.TopSage.com

大家网 TopSage.com

557 此字符串表示一个含有宽字符L'a'、L'b'和L'c'并且后边跟着代码为零的宽字符的数组。

Unicode

宽字符非常适合于固定长度编码的字符集。一个重要的示例就是Unicode。作为一种通用字符集的尝试，Unicode是一种每个国家都可以采用的编码。Windows NT操作系统当前支持Unicode，而且它很可能会及时成为其他操作系统的特性。每个Unicode字符占用两个字节，所以Unicode可以表示65 536个字符，并且为所有现代语言以及一些古老语言（例如，梵语[①]）要求的字母表预留了足够的空间。Unicode还包含了一定数量的特殊符号，比如用于数学运算的符号等。

25.2.3 多字节字符函数

```
int mblen(const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
int wctomb(char *s, wchar_t wchar);
```

mblen函数检测第一个参数是否指向形成有效多字节字符的字节序列。如果是，函数返回字符中的字节数。如果不是，函数则返回-1。作为一种特殊情况，如果函数的第一个参数指向空字符，则mblen函数返回0。函数的第二个参数限制了mblen函数将检测的字节的数量。通常情况下会传递MB_CUR_MAX。

下面的函数使用mblen函数来确定字符串是否由有效的多字节字符构成。(此示例和稍后本节中的wccheck示例都来自于P . J . Plauger写的*The Standard C Library*）如果s指向有效字符串，则函数返回零。

```
int mbcheck(const char *s)
{
  int n;

  for(mblen(NULL, 0) ; ; s += n)
    if ((n = mblen(s, MB_CUR_MAX)) <= 0)
      return n;
}
```

mbcheck函数的两个概念需要特别说明一下。首先是mblen(NULL, 0)的神秘调用。此调用使mblen函数跟踪静态变量中的移位状态。mblen(NULL, 0)的调用设置了此变量的初始状态，以便于可以正确解释字符串中稍后的字符。(把空指针传递给mbtowc函数或者wctomb函数具有相似的效果。顺便说一下，每个函数都有自己的移位状态。）mblen函数的调用可以改变移位状态。其次是有关终止的问题。请记住s指向的是以空字符结尾的普通字符串。当mblen函数
558 遇到这个空字符时将返回零，这样的结果会导致mbcheck函数返回。

mbtowc函数把（由函数的第二个参数指向的）多字节字符转换为宽字符。第一个参数指向函数将存储结果的变量，第三个参数限制了mbtowc函数将检测的字节的数量。mbtowc函数返回和mblen函数一样的值：如果有效，则返回字符中字节的数量；如果无效，则返回-1；如果第二个参数指向空字符，则返回零。

wctomb函数把宽字符（第二个参数）转换为多字节字符，并且把多字节字符存储到第一个参数指向的数组中。wctomb函数可以存储和MB_LEN_MAX一样多的字符到数组中，但是不附加空字符。转换会考虑当前的移位状态，如果需要还会更新移位状态。如果有效，wctomb

① 一种古印度语，为印度及吠陀经所用文字，也是印度的古典文学语言。——编者注

www.TopSage.com

函数会返回字符中字节的数量；如果无效，则返回-1。（注意，如果要求转换空的宽字符，则返回1。）

下面这个函数使用wctomb函数来确定是否可以把宽字符的字符串转换为有效的多字节字符：

```
int wccheck(wchar_t *wcs)
{
  char buf [MB_LEN_MAX];
  int n;

  for (wctomb (NULL,0); ; ++wcs)
     if ((n = wctomb (buf, *wcs)) <= 0)
      retur -1;                        /* invalid character */
     else if (buf [n-1] == '\0')
      return 0;                        /* all characters are valid */
}
```

顺便说一下， mblen函数、mbtowc函数和wctomb函数都可以用来监测多字节编码是否是依赖状态的。当传递空指针作为char*类型的参数时，如果多字节字符是依赖状态的，那么上述每种函数都会返回非零值；否则返回零。

25.2.4 多字节字符串函数

```
size_t mbstowcs(wchar_t *pwcs, const char *s,
                size_t n);
size_t wcstombs(char *s, const wchar_t *pwcs,
                size_t n);
```

mbstowcs函数把多字节字符串转换为宽字符序列。函数的第二个参数指向多字节字符串，而第一个参数则指向宽字符的数组,第三个参数限制了可以存储在数组中的宽字符数量。当达到上限或者遇到空字符（存储在宽字符数组中）时，mbstowcs函数就停止。函数会返回修改的数组元素的数量，但是无论如何不会包括用来终止的零代码。如果遇到无效的多字节字符，mbstowcs函数则返回-1。

559

wcstombs函数和mbstowcs函数正好相反：它把宽字符串转换为多字节字符。函数的第二个参数指向宽字符串，第一个参数指向存储多字节字符的数组，第三个参数限制了存储在数组中的字节的数量。当达到上限或者遇到空字符（函数存储的）时，wcstombs函数就停止。函数会返回存储的字节的数量，但是无论如何不会包含用来终止的空字符。如果遇到一个宽字符无法对应任何多字节字符，则wcstombs函数返回-1。

mbstowcs函数假设要转换的字符串以初始移位状态开始,而由wcstombs函数产生的字符串则始终是以初始移位状态开始。

25.3 三字符序列

三字符序列（trigraph sequence）（或者简称为“三字符”）是一种三个字符码，它可以用作ASCII字符的替代品。三字符寻址的问题很简单：C程序需要字符#、[、\、]、^、{、|、}和~。许多欧洲国家使用缺少这样一些字符的ASCII的替换形式。例如，在德国，把#、[、\、]、^、{、|、}和~分别替换成了Ä、Ö、Ü、ä、ö、ü和ß。三字符提供了一种编写有效C程序而不使用任何缺少字符的方法。

表25-5给出了三字符序列的完整列表。所有三字符都以??开始，这样做虽然不能十分吸引人，但至少可以便于发现三字符。

www.TopSage.com

表25-5 三字符序列

三字符序列	等价的ASCII码
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

三字符可以自由地替换成等价的ASCII码。例如，程序

```
#include <stdio.h>

main()
{
  printf("hello, world\n");
  return 0;
}
```

560

可以写成

```
??=include <stdio.h>

main()
??<
  printf("hello, word??/n")
  return 0;
??>
```

尽管不是一直需要，但是要求所有的标准C编译器都接受三字符序列。偶尔，这个特性可能会导致问题。

在字符串中请小心放置??，因为编译器可能会把它作为三字符序列的开始标志。如果发生这种情况，那么通过在第二个？字符的前面放置字符\来把第二个字符?变成转义序列。?\?这样组合的结果就不会被看作是三字符的开始了。

问与答

问：setlocale函数可以返回多长的地区信息字符串？ (p.384)

答：不存在最大长度。这就引发了一个问题：如果不知道字符串的长度，如何为字符串设置空间呢？当然，答案就是动态存储分配。下面这个例子（基于Harbison和Steele写的*C: A Reference Manual*一书的类似示例）说明了如何确定需要的空间数量，然后再把地区信息复制到此空间中：

```
char *temp,*old_locale;

temp = setlocale(LC_ALL, NULL);
if (temp  NULL) {
  /* locale information not available */
}
old_locale = malloc (strlen (temp)+1);
if (old_locale == NULL) {
  /* memory allocation failed */
}
strcpy (old_locale, temp);
```

www.TopSage.com

TopSage.com

为了恢复旧的地区信息，最好首先切换成本地模式，然后再恢复旧的地区：

```
setlocale(LC_ALL, "")          /* switch to native locale */
setlocale(LU_ALL, old_locale);/* restore old locale */
```

561

问：为什么C语言提供多字节字符和宽字符呢？两者选其一难道不够吗？ (p.387)

答：两种编码用于不同的目的。多字节字符用于输入/输出目的很方便，因为输入/输出设备经常是面向字节的。但是宽字符更适用于程序内部，因为每个宽字符占有相同的空间。因此，程序可以读入多字节字符输入，把它转换为便于程序内部操作的宽字符格式，然后再把宽字符转换回用于输出的多字节格式。

练习

25.1节

1. 请确定你用的编译器支持哪种地区。
2. 编写一个程序，用来测试你用的编译器的""（本地）地区是否和"C"地区一样。

25.2节

3. 用于kanji（日文中的汉字）的Shift-JIS编码要求每个字符是单字节或者是双字节的。如果字符的第一个字节位于0x81和0x9f之间，或者位于0xe0和0xef之间，那么就需要第二个字节。（把任何其他字节看成是整个字符。）第二个字节必须在0x40和0x7e之间，或者在0x80和0xfc之间。（所有的范围都包含边界值。）对于下面的每个字符串，当传递其作为参数时，请指出25.2节的mbcheck函数将会返回的值。

   (a) `"\x05\x87\x80\x36\xed\xaa"`

   (b) `"\x20\xe4\x50\x88\x3f"`

   (c) `"\xde\xad\xbe\xef"`

   (d) `"\x8a\x60\x92\x74\x41"`

25.3节

4. 请通过尽可能多地用三字符替换字符的方法来修改下面的程序段。

```
While ((orig_char = getchar()) != EOF) {
  new_char = orig_char ^ KEY;
  if (iscntrl(orig_char) || iscntrl(new_char))
    putchar(orig_char);
  else
    putchar(new_char);
}
```

562

第26章

其他库函数

确定程序参数的应该是用户，而不应该是它们的创造者。

本章讨论标准库中剩下的三个头：<stdarg.h>、<stdlib.h>和<time.h>。这三个头不同于库函数中的其他头文件，所以把它们留到最后来介绍。<stdarg.h>（26.1节）可使编写的函数带有可变数量的实参；<stdlib.h>（26.2节）是函数的分类，但是此分类不适合其他库函数头的某一种。<time.h>允许程序处理日期和时间。

26.1 <stdarg.h>：可变长度实参

```
void va_start(va_list ap, parmN);
类型 va_arg(va_list ap, 类型);
void va_end(va_list ap);
```

我们已经见过类似printf函数和scanf函数这样的函数，它们在接收的参数数量上没有固定的限制。然而，这种能力是对库函数而言不限制处理可变数量的参数。现在<stdarg.h>将提供一种工具使我们自行编写的函数也具有可变长度的参数列表。<stdarg.h>定义了一种va_list类型和三种宏，这三种宏名为va_satrt、va_arg和va_end。可以把这些宏看成是带有上述原型的函数。

563

为了了解这些宏的工作过程，这里将用它们来编写一个名为max_int的函数。此函数用来在任意数量的整型参数中找出最大数。下面是此函数的调用过程：

```
max_int(3, 10, 30, 20)
```

函数的第一个实参说明了跟随其后的其他参数的数量。这里的max_int函数调用将会返回30（即10、30和20中的最大数）。

下面是max_int函数的定义：

```
int max_int(int n, ...)     /* n must be at least 1 */
{
  va_list ap;
  int i, current, largest;

  va_start (ap, n) ;
  largest = va_arg(ap, int) ;

  for (i : i; i < n; i++) {
     current = va_arg (ap, int) ;
     if (current > largest)
        largest = current;
  }

  va_end ( ap ) ;
  return largest;
}
```

在形式参数列表中的...符号（省略号）表示参数n后边跟随着其他可变数量的参数。

www.TopSage.com

max_int函数体从声明va_list类型的变量开始：

```
va_list ap;
```

声明这样的变量是为了强制max_int函数可以访问到跟在n后边的实参。

语句va_start(ap, n);指出了实参列表中可变长度部分开始的位置（这里的情况是从n后边开始）。带有可变数量参数的函数必须至少有一个“正常的”形式参数，在最后一个正常参数的后边始终会有省略号出现在参数列表的末尾。 564

语句largest = va_arg(ap, int);把获取的max_int函数第2个参数（n后面的一个）赋值给变量largest，并且自动前进到下一个参数处。语句中的单词int指明希望的max_int函数的第2个实参是int类型的。当程序执行内部循环时，语句current = va_arg(ap, int);会逐个获取max_int函数余下的参数。

不要忘记在获取当前参数后，宏va_arg始终会前进到下一个参数的位置上。正是由于这个特点，所以这里不能用如下方式编写max_int函数的循环：

```
for (I = 1; I < n; I++)
  if (va_arg (ap, int) > largest)   /*** WRONG ***/
    largest = va_arg(ap, int);
```

在函数返回之前，要求用语句va_end(ap);进行“清扫”。（如果不返回，函数可能会再次调用va_start并且遍历参数列表。）

当调用带有可变实参列表的函数时，编译器会在匹配省略号的全部参数上执行默认的实参提升（➤9.3.1节）：把字符型值提升为整数，并且把float型值提升为double型值。这样的结果是va_arg不会感觉到传递过来的是字符类型或float类型，因为提升后的参数将永远不会具有这些类型。

26.1.1 调用带有可变实参列表的函数

调用带有可变实参列表的函数是一个固有的风险提议。回溯到第3章就会发现给printf函数和scanf函数传递错误参数是多么危险。其他带有可变实参列表的函数也同样容易出问题。主要的难点就是带有可变实参列表的函数很难确定传递过来的参数的数量或类型。所以必须要把这个信息传递给函数，并且/或者函数假设知道了这个信息。示例中的max_int函数依靠第一个实参来指明跟随其后的其他参数的数量，并且它还假定参数是int类型的。而像printf函数和scanf函数这样的函数则是依靠格式化字串来描述其他的参数的数量和每种参数的类型。

另外一个问题就是不得不处理NULL作为参数传递的情况。通常都把NULL定义成表示0。但是，当把0传递给带有可变实参列表的函数时，编译器会假定它表示的是一个整数，因为这里没有办法可以说明希望它表示的是一个空指针。解决这种问题的方法就是添加一个强制类型转换，用（void*） NULL来代替NULL，这样就可以表明传递的是一个空指针了。（请见第17章的“问与答”小节关于此点的更多详细讨论。） 565

26.1.2 v...printf 类函数

```
int vfprintf(FILE *stream, const char *format,
             va_list arg);
int vprintf(const char *format, va_list arg);
int vsprintf(char *s, const char *format,
             va_list arg);
```

vfprintf函数、vprintf函数和vsprintf函数（即v...printf类函数）都属于<stdio.h>。本节讨论这些函数是因为它们总是和宏联合用于<stdarg.h>中。

v...printf类函数和fprintf函数、printf函数以及sprinf函数有着紧密的联系。但是，不同于这些函数的是v...printf类函数具有固定数量的实参。每个v...printf类函数的最后一个

www.TopSage.com

大家网 TopSage.com

实参都是一个va_list型值。这个类型的值意味着此函数将可以由带有可变实参列表的函数调用。实际上，v...printf类函数主要用于编写“包装”函数。包装函数接收可变数量的实参，并且稍后把这些参数传递给v...printf类函数。

举一个例子，假设正在工作的程序需要不时地显示出错消息，而且我们希望每条消息都以下列格式的前缀开始：

```
** Error n:
```

这里的*n*在显示第一条出错消息时是1，并且*n*会随着下一条错误信息而逐次加一。为了使产生出错信息更加容易，我们将编写一个名为errorf的函数，此函数类似于printf函数，但是它在输出的开始处添加了** Error n:，而且此函数不向stdout输出而是向stderr写输出。errorf函数将调用vfprintf函数来完成实际输出的大部分内容。下面是errorf函数可能的写法：

```
int errorf(const shar *format, ...)
{
  static int num_errors = 0;
  int n;
  va_list ap;

  num_errors++;
  fprintf(stderr, "** Error %d: ", num_errors);
  va_start(ap, format);
  n = vfprintf(stderr, format, ap);
  va_end (ap);
  fprintf(stderr, "\n");
  return n;
}
```

566

26.2 <stdlib.h>：通用的实用工具

<stdlib.h>涵盖了全部不适合于任何其他头的函数。<stdlib.h>中的函数可以大致分为7种各不相关的组：

- 字符串转换函数。
- 伪随机序列生成函数。
- 内存管理函数。
- 与外部环境的通信。
- 搜索和排序实用工具。
- 整数算术运算函数。
- 多字节字符和字符串函数。

这里将逐个介绍每组函数，但是有两组例外：内存管理函数以及多字节字符和字符串函数。

内存管理函数（即malloc函数、calloc函数、realloc函数和free函数）允许程序分配内存块，而且稍后它们还允许程序释放或者改变内存块的大小。第17章已经详细描述了这4种内存管理函数。

多字节字符和字符串函数允许程序对多于一个字节长度的字符进行操作。25.2节已经介绍了多字节字符，并且解释说明了多字节函数的工作原理。

26.2.1 字符串转换函数

```
double atof (const char *nptr);
int atoi(const char *nptr) ;
long int atol(const char *nptr) ;
double strtod(const char *nptr, char **endptr) ;
```

www.TopSage.com

```
long int strtol(const char *nptr, char **endptr,
                int base);
unsigned long int strtoul(const char *nptr,
                          char **endptr, int base);
```

这一组的函数会把含有字符格式的数字符串转换成它的等价数值。这些函数中有3个函数是非常旧的，而其他3个函数则是在C语言标准化过程中添加进来的。

旧函数（atof函数、atoi函数和atol函数）把字符串分别转换成double、int或者long int型值。每个函数都会在字符串的开始处跳过空白字符，并且把后续字符作为数的一部分，同时每个函数还会在第一个不是数的部分的字符处停止。

新函数（strtod函数、strtol函数和strtoul函数）更加复杂精妙。举个例子来说，这三个函数会通过修改endptr指向的变量来指出转换停止的位置。（如果不在乎转换结束的位置，那么函数的第二个参数可以为空指针。）为了检测函数是否可以使用到整个字符串，只需检测此变量是否可以指向空字符。更厉害的是strtol函数和strtoul函数还有一个base参数用来说明要转换数的基数。函数支持的基数是2到36之间的所有数（包括2和36）。 567

除了比原来的旧函数更通用以外，新函数还更善于处理错误。旧函数没有方法可以指明转换期间用到的字符串的数量。此外，如果旧函数无法定位要转换的数，它会返回零。如果数过大，那么旧函数则返回未定义的值。这样的结果是，上述这些都无法通过检测函数的返回值来发现任何问题。而如果转换产生的用来表示的值过大（或者过小），那么新函数就会把ERANGE存储在errno中（<errno.h>➤24.2节）。

由于strtod函数、strtol函数和stroul函数的添加，atof函数、atoi函数和atol函数就显得多余了。为了早期的C程序方便使用，这些旧函数仍保留在函数库中，，但是这里推荐新程序使用strtod函数、strtol函数和stroul函数。

26.2.2 程序：测试字符串转换函数

下面这个程序通过应用6种字符串转换函数中的每一种来把字符串转换为数值格式。在调用了strtod函数、strtol函数和stroul函数之后，程序还会显示出是否每种版本都产生了有效的结果，以及是否每种版本可以用到整个字符串。程序将从命令行中获得输入字符串。

tstrconv.c

```
/* Tests string conversion functions */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

#define CHK_VALID  printf("    %s          %s\n",        \
                          errno != ERANGE ? "Yes" : "No ", \
                          *ptr == '\0' ? "Yes" : "No");

main(int argc, char *argv[])
{
  char *ptr;

  if (argc != 2) {
    printf("usage: tstrconv string\n");
    exit(EXIT_FAILURE);
  }

  printf("Function   Return Value\n");
  printf("--------   ------------\n");
  printf("atof       %g\n", atof(argv[1]));
  printf("atoi       %d\n", atoi(argv[1]));
  printf("atol       %ld\n\n", atol(argv[1]));
```

568

```
    printf("Function   Return Value   Valid?   "
           "String Consumed?\n"
           "--------   ------------   ------   "
           "----------------\n");

    errno = 0;
    printf("strtod     %-12g", strtod(argv[1], &ptr));
    CHK_VALID;

    errno = 0;
    printf("strtol     %-12ld", strtol(argv[1], &ptr, 10));
    CHK_VALID;

    errno = 0;
    printf("strtoul    %-12lu", strtoul(argv[1], &ptr, 10));
    CHK_VALID;

    return 0;
}
```

如果3000000000是命令行参数，那么tstrconv的输出可能会有如下显示：

```
Function     Return Value
--------     ------------
atof         3e+09
atoi         24064
atol         -1294967296

Function     Return Value       Valid?     String Consumed?
--------     ------------       ------     ----------------
strtod       3e+09              Yes        Yes
strtol       2147483647         No         Yes
strtoul      3000000000         Yes        Yes
```

虽然3000000000这个数作为无符号长整数是有效的，但是因为它太长了对许多机器而言都很难表示为长整数。atoi函数和atol函数就无法发现这个问题，而且它们会以返回奇怪的值结束。stroul函数会执行正确地转换，而strtol函数则会返回2147483647（最大的长整数）并且把ERANGE存储到errno中。

如果123.456是命令行参数，那么输出将是：

```
Function     Return Value
--------     ------------
atof         123.456
atoi         123
atol         123

Function     Return Value      Valid?     String Consumed?
--------     ------------      ------     ----------------
strtod       123.456             Yes            Yes
strtol       123                 Yes            No
strtoul      123                 Yes            No
```

569

所有的函数都会把这个输入作为有效的数来处理，但是整型函数会在小数点处停止。stroul函数和strtoul函数无法完全用到整个输入，就是因为这个问题。

如果foo是命令行参数，那么输出将是：

```
Function   Return Value
--------     ------------
atof       0
atoi       0
atol       0
Function   Return Value   Valid?   String Consumed?
--------   ------------   ------   ----------------
strtod     0               Yes           No
```

www.TopSage.com

TopSage.com

```
strtol      0              Yes          No
strtoul     0              Yes          No
```

全部函数看到字母f立刻返回零。str… 类函数不会改变errno，但是我们会明白出现这问题实际是函数没有使用到字符串。

26.2.3 伪随机序列生成函数

```
int rand(void);
void srand(unsigned int seed);
```

rand函数和srand函数都可以用来生成伪随机数。这两个函数用于模拟程序和玩游戏程序（例如，在纸牌游戏中用来模拟骰子滚动或者发牌。）。

每次调用rand函数时，它都会返回一个0~RAND_MAX（定义在<stdlib.h>中的宏）的数。rand函数返回的数事实上不是随机的。这些数是由“种子”值产生的。但是，对于偶然的观察者而言，rand函数表现出来的是产生了不相关的数序列。

调用srand函数可以提供用于rand函数的种子值。如果在srand函数之前调用rand函数，那么会把种子值设定为1。每个种子值确定了一个特殊的“随机”数序列。srand函数允许用户选择他们自己想要的序列。

始终使用同一个种子值的程序将总会从rand函数得到相同的数序列。这个特点有时是非常有用的：程序在每次运行时按照相同的方式运行，这样会使得测试更加容易。但是，用户通常是希望每次程序运行时rand函数能产生不同的序列的。（玩纸牌的程序如果总是发同样的牌是不会受欢迎的。）使种子值“随机化”的最简单方法就是调用time函数（time函数➤26.3.1节），它会返回一个对当前日期和时间进行编码的数。把time函数的返回值传递给srand函数，这样可以使rand函数在每次运行时的行为都不相同。这种方法可以见10.2节中的示例guess.c程序和guess2.c程序。

570

26.2.4 程序：测试伪随机序列生成函数

下面这个程序首先显示由rand函数返回的前10个值，然后允许用户选择新的种子值。此过程会反复执行直到用户输入零作为种子值为止。

trand.c

```
/* Tests the pseudo-random sequence generation functions */

#include <stdio.h>
#include <stdlib.h>

main()
{
  int i, seed;

  printf("This program displays the first ten values of "
         "rand.\n");

  for (;;) {
    for (i = 0; i < 10; i++)
      printf("%d ", rand());
    printf("\n\n");
    printf("Enter new seed value (0 to terminate): ");
    scanf("%d", &seed);
    if (seed == 0)
      break;
    srand(seed);
  }

  return 0;
}
```

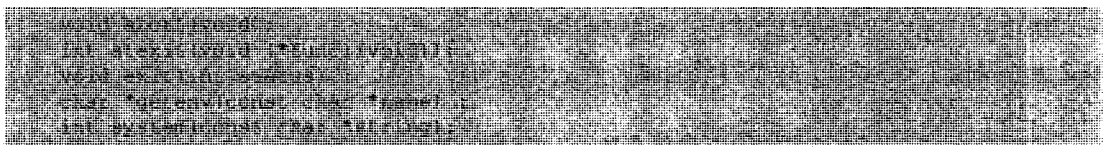
www.TopSage.com

下面是程序运行时可能的交互情况：

```
This program displays the first ten values of rand.
346 130 10982 1090 11656 7117 17595 6415 22948 31126

Enter new seed value (0 to terminate): 100
1862 11548 3973 4846 9095 16503 6335 13684 21357 21505

Enter new seed value  (0 to terminate): 1
346 130 10982 1090 11656 7117 17595 6415 22948 31126

Enter new seed value (0 to terminate):  0
```

编写rand函数的方法有很多，所以这里不保证每种rand函数的版本都可以生成上述这样的数。
571 请注意，选择1作为种子值与根本没有指明种子值会得到相同的数列。

26.2.5　与环境的通信



这一组函数为操作系统提供了简单的界面。它们允许程序：正常的或不正常的终止，并且为操作系统返回一个状态码；从用户的外部环境获取信息；执行操作系统的命令。

在程序中的任何位置执行exit(*n*)调用等价于在main函数中执行return *n*；语句：即程序终止，并且把*n*作为状态码返回给操作系统。<stdlib.h>定义了宏EXIT_FAILURE和宏EXIT_SUCCESS，这些宏可以用作exit函数的参数。exit函数另一个唯一的可移植参数就是0，它和宏EXIT_SUCCESS意义相同。返回其他不是上述这些的状态码也是合法的，但是它们对所有操作系统而言都是不可移植的。

当终止程序时，（atexit函数）通常还会在屏幕后台执行一些最后的动作，包括清洗输出缓冲区，关闭打开的流，以及删除临时文件。当然可能还有其他希望程序终止时执行的“清扫”操作。atexit函数允许用户“注册”一个临近程序终止时要调用的函数。例如，为了注册名为cleanup的函数，可以用如下方式调用atexit函数：

```
atexit(cleanup);
```

当把函数指针传递给atexit函数时，它会为将来的引用而把指针保存起来。稍后，程序终止时都将自动调用任何由atexit函数注册的函数。（如果注册了几个函数，那么将首先调用最新注册的函数）。

abort函数类似于exit函数，但是前者会导致异常的程序终止。还不能调用由atexit函数注册的函数。依靠实现，abort函数可以是这样一种情况：不能清洗文件缓冲区、不能关闭流和不能删除临时文件。Q&A abort函数返回一个由实现定义的状态码来说明“不成功的终止”。

许多操作系统都会提供“外部环境”：即一套描述用户特征的字符串。这些字符串通常包含用户运行程序时要搜索的路径、用户终端的类型（比如多用户系统的情况）等等。例如，一条UNIX系统的搜索路径可能如下所示：

```
PATH=~:~/bin:/bin:/usr/bin:.
```

572 而DOS系统的路径可能具有类似的表示：

```
PATH=C:\;C:DOS;C:\WINDOWS
```

getenv函数提供了在用户的外部环境中访问任意字符串的功能。例如，为了找到PATH字符串的当前值，可以写成

```
p = getenv("PATH");
```

www.TopSage.com

在执行了此条语句之后，p会指向诸如"~:~/bin:/bin:/usr/bin:."或者"C:\;C:DOS;C:\WINDOWS"这样的字符串。由getenv函数返回的字符串会被静态分配，并且由稍后的函数调用进行改变。

system函数允许C程序运行另一个程序（可能是一个操作系统命令）。system函数的参数是命令行，这类似于一个在操作系统提示下的录入内容。例如，假设正在编写的程序需要当前目录中的文件列表。UNIX程序将按照下列方式调用system函数：

```
system("ls >myfiles");
```

而DOS程序会使用略有不同的调用方式：

```
system("dir >myfiles");
```

无论哪种调用之后，myfiles都将包含目录列表。system函数返回的值是由实现定义的。通常情况下，system函数会返回来自程序的终止状态码，此程序是system函数要求运行的。测试这个返回值可以检测程序是否正常工作。调用带有空指针的system函数有特别的含义：如果命令处理程序是有效的，那么函数会返回非零值。

26.2.6 搜索和排序实用工具

```
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *,
                            const void *));
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

bsearch函数在有序的数组中搜索特殊值（**关键字**）。当调用bsearch函数时，形式参数key指向关键字，base指向数组，nmemb是数组中元素的数量，size是每个元素的大小（按字节计算），而compar是指向比较函数的指针。比较函数类似于qsort函数要求的函数：当把指针传递给关键字和数组元素（按顺序）时，函数必须依赖于关键字是小于、等于还是大于数组元素，而返回负整数、零或正整数。bsearch函数返回的指针要指向与关键字匹配的元素，如果没有找到匹配的元素，那么bsearch函数会返回一个空指针。 573

虽然标准C不要求，但是bsearch函数通常会使用“二分检索”算法来搜索数组。bsearch函数首先把关键字与数组的中间元素进行比较。如果比较结果匹配，那么函数就返回。如果关键字小于数组的中间元素，那么bsearch函数将把搜索限制在数组的前半部分。如果关键字大于数组的中间元素，那么bsearch函数则只搜索数组后半部分。bsearch函数会重复这种方法直到它找到关键字或者是超出了搜索元素范围。正是由于这种方法，使bsearch函数可以迅速进行搜索。比如，搜索有1000个元素的数组最多只需进行10次比较。搜索有1 000 000个元素的数组需要不多于20次的比较。

17.7节讨论了可以对任何数组进行排序的qsort函数。虽然bsearch函数只能用于有序的数组，但是可以在要求bsearch函数搜索某个数组之前，总是用qsort函数来对此数组进行排序。

26.2.7 程序：确定航空里程

下面的程序是用来计算从纽约到不同的国际城市之间的航空里程。程序首先要求用户录入城市的名称，然后显示从纽约到此城市的里程：

```
Enter city name: Frankfurt
Frankfurt is 3851 miles from New York City.
```

程序将把城市/里程数据对存储在数组中。通过使用bsearch函数在数组中搜索城市名，然后程序就可以很容易地找到相应的里程数了。（里程来自*The New York Library Desk Reference*第2版（New York：Prentice-Hall，1993）。）

www.TopSage.com

大家网 TopSage.com

airmiles.c

```
/* Determines air mileage from New York to other cities */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct city_info {
  char *city;
  int miles;
};

int compare_cities(const void *key_ptr,
                   const void *element_ptr);

main()
{
  char city_name[81];
  struct city_info *ptr;
  const struct city_info mileage[] =
    {{"Acapulco",       2260},   {"Amsterdam",          3639},
     {"Antigua",        1783},   {"Aruba",              1963},
     {"Athens",         4927},   {"Barbados",           2100},
     {"Bermuda",         771},   {"Bogota",             2487},
     {"Brussels",       3662},   {"Buenos Aires",       5302},
     {"Caracas",        2123},   {"Copenhagen",         3849},
     {"Curacao",        1993},   {"Frankfurt",          3851},
     {"Geneva",         3859},   {"Glasgow",            3211},
     {"Hamburg",        3806},   {"Kingston",           1583},
     {"Lima",           3651},   {"Lisbon",             3366},
     {"London",         3456},   {"Madrid",             3588},
     {"Manchester",     3336},   {"Mexico City",        2086},
     {"Milan",          4004},   {"Nassau",             1101},
     {"Oslo",           3671},   {"Paris",              3628},
     {"Reykjavik",      2600},   {"Rio de Janeiro",     4816},
     {"Rome",           4280},   {"San Juan",           1609},
     {"Santo Domingo",  1560},   {"St. Croix",          1680},
     {"Tel Aviv",       5672},   {"Zurich",             3926}};

  printf("Enter city name: ");
  scanf("%80[^\n]", city_name);
  ptr = bsearch(city_name, mileage,
                sizeof(mileage)/sizeof(mileage[0]),
                sizeof(mileage[0]), compare_cities);
  if (ptr != NULL)
    printf("%s is %d miles from New York City.\n",
           city_name, ptr->miles);
  else
    printf("%s wasn't found.\n", city_name);

  return 0;
}

int compare_cities(const void *key_ptr,
                   const void *element_ptr)
{
  return strcmp((char *) key_ptr,
                ((struct city_info *) element_ptr)->city);
}
```

574

26.2.8 整数算术运算函数

```
int abs(int j);
div_t div(int numer, int denom);
long int labs(long int j);
ldiv_t ldiv(long int numer, long int denom);
```

abs函数返回int型值的绝对值，labs函数返回long int型值的绝对值。

div函数用第一个实参除以第二个实参，并且返回一个div_t型值。div_t是一个含有商成

www.TopSage.com

员（命名为quot）和余数成员（命名为rem）的结构。例如，如果ans是一个div_t型变量，那么可以写成 575

```
ans = div(5,2);
printf("Quotient: %d Remainder: %d\n", ans.quot, ans.rem);
```

Q&A ldiv函数和div函数很类似，不同之处在于ldiv函数用于处理长整数。ldiv函数返回的ldiv_t型值也包含quot和rem两个成员。（在<stdlib.h>中定义了div_t类型和ldiv_t类型。）

26.3 <time.h>：日期和时间

<time.h>提供的函数可以确定时间（和日期），可以在时间值上进行算术操作，还可以显示格式化的时间。然而，在介绍这些函数之前，需要讨论一下时间是如何存储的。<time.h>提供了三种类型，每种类型表示一种不同的存储时间的方法：

- clock_t：按照“时钟嘀嗒”进行测量的时间值。
- time_t：紧凑的编码时间和日期（**日历时间**）。
- struct tm：把时间分解成秒、分、时等。经常把struct tm类型值称为**分解时间**。表26-1说明了tm结构的成员。全部成员都是int类型的。

表26-1 tm结构的成员

名 称	描 述	最小值	最大值
tm_sec	分后边的秒	0	61②
tm_min	时后边的分	0	59
tm_hour	从午夜以后的时	0	23
tm_mday	每月的天	1	31
tm_mon	从一月以后的月份	0	11
tm_year	从1900以后的年份	0	—
tm_wday	从星期日以后的天	0	6
tm_yday	从一月一日以后的天	0	365
tm_isdst	白天省时标记	①	①

① 如果白天省时有效，就为正数；如果无效，就为零；如果信息是未知的，就为负数。
② 允许两个额外的“闰秒”。

上述这些类型有不同的用途。clock_t值只善于表示时间区间。而time_t值和struct tm值则可以存储完整的日期和时间。time_t值是紧密编码，所以他们占用很少的空间。而struct tm值却要求较大的空间，但是这类值常常易于使用。C标准规定了clock_t和time_t必须是“算术运算类型”，姑且这样理解。结果是无法知道clock_t值和time_t值是要作为整数存储还是浮点数存储。

现在来看看<time.h>中的函数。这些函数分为两组：时间处理函数和时间转换函数。 576

26.3.1 时间处理函数

clock函数返回表示处理器时间的clock_t值，程序从执行开始后就使用这个处理器时间了。为了把这个值转换为秒，将把值除以定义在<time.h>中的宏CLOCKS_PER_SEC。

www.TopSage.com

当用clock函数来确定程序运行多长时间时，习惯做法是调用两次clock函数：一次在main函数开始处，一次在程序就要终止之前：

```
#include <time.h>

main()
{
  clock_t start_clock = clock();
  …
  printf("Processor time used: %g sec.\n"),
         (clock() - start_clock) / (double) CLOCKS_PER_SEC);
  return 0;
}
```

初始调用clock函数的理由是，由于隐藏的“启动”代码，程序会在到达main函数之前使用一些处理器时间。在main函数开始处调用clock函数可以确定启动代码需要多长时间，以便于稍候可以减去这部分时间。

C标准没有说明clock_t是整数类型还是浮点类型，也不知道宏CLOCKS_PER_SEC的类型。结果是我们无法知道下列表达式的类型：

```
(clock*() - start_time) / CLOCKS_PER_SEC
```

这样就很难用printf函数来显示内容。为了解决这个问题，这里的示例把宏CLOCKS_PER_SEC强制成double型，从而迫使整个表达式具有了double类型。

time函数返回当前的日历时间。如果实参不是空指针，那么time函数也会把日历时间存储在实参指向的对象中。time函数返回两种不同方式的时间的能力是历史遗留问题，但是它为用户提供了两种书写的选择，既可以是

```
curtime = time(NULL);
```

也可以是

```
time(&cur_time);
```

577 这里的cur_time是具有time_t类型的变量。

difftime函数返回time0（较早的时间）和time1之间按秒衡量的差值。因此，为了计算程序的实际运行时间（不是处理器时间），可以采用下列代码：

```
#include <time.h>

main()
{
  time_t start_time = time(NULL);
  …
  printf("Running time: %g sec.\n"),
         difftime(time(NULL), start_time));
  return 0;
}
```

mktime函数把分解时间（存储在函数参数指向的结构中）转换为日历时间，然后返回。作为副作用，mktime函数会根据下列规则调整结构的成员：

- mktime函数会改变其值不在合法范围内的任何成员（表26-1），这样的改变可能会依次要求改变其他成员。例如，如果tm_sec过大，那么mktime函数会把它减少到合适的范围内（0~59），并且会为tm_min增加额外的分钟数。如果现在tm_min过大，那么mktime函数会减少tm_min，同时为tm_hour增加额外的小时数。如果必要，此过程还将继续对成员tm_mday、成员tm_mon和成员tm_year进行操作。
- 在调整完结构的其他成员后，（如果必要）mktime函数会给tm_wday（按星期算的天）和tm_yday（按年算的天）设置正确的值。在调用mktime函数之前，从来不需要对tm_wday

www.TopSage.com

大学网 TopSage.com

和tm_yday的值进行任何初始化，因为mktime函数会忽略这些成员的初始值。

mktime函数调整tm结构成员的能力对于和时间相关的算术计算非常有用。例如，现在用mktime函数来回答下列这个问题：如果1996年的奥林匹克运动会从7月19日开始，并且历时16天，那么请说出结束的日期是哪天？这里将把开始日期1996年7月19日存储在tm结构中：

```
struct tm t;

t.tm_mday = 19;
t.tm_mon = 6;     /* July */
t, tm_year  96;   /* 1996 */
```

为了确保结构的其他成员不包含可能影响结果的垃圾值，这里还要对这些成员也进行初始化（成员tm_wday和tm_yday除外）：

```
t.tm_sec = 0;
t.tm_min = 0;
t.tm_hour = 0;
t.tm_isdst = -1;
```

578

接下来，将把16添加给成员tm_mday：

```
t.tm_mday += 16;
```

这样操作的结果使在成员tm_mday中产生了35，它超出了成员的取值范围。调用mktime函数将会使结构的成员恢复到正确的取值范围内：

```
mktime(&t);
```

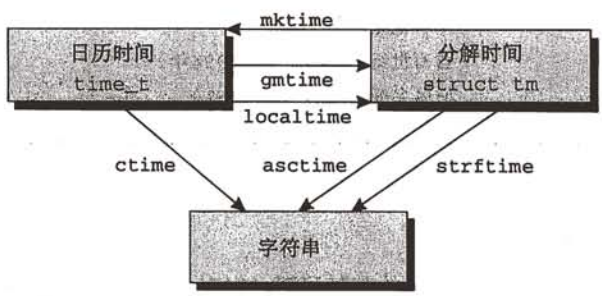
这里将舍弃mktime函数的返回值，因为我们只对在t上的函数效果感兴趣。现在，和t相关的成员具有下列值：

成　员	值	含　义
tm_mday	4	4
tm_mon	7	August
tm_year	96	1996
tm_wday	0	Sunday
tm_yday	216	217th day of the year

26.3.2 时间转换函数

```
char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timer);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
size_t strftime(char *s, size_t maxsize,
                const char *format,
                const struct tm *timeptr) ;
```

时间转换函数可以把日历时间转换成分解时间，还可以把时间（日历时间或分解时间）转换成字符串格式。下图说明了这些函数之间的关联关系。



www.TopSage.com

579 此图包含了mktime函数。C标准把此函数划分为“处理”函数而不是“转换”函数。

gmtime函数和localtime函数很类似。当程序把指针传递给日历时间时，这两种函数都会返回一个指向结构的指针，且指向的结构含有等价的分解时间。Q&A localtime函数会产生本地时间，而gmtime函数的返回值则是用UTC（协调世界时间）表示的。

asctime（ASCII时间）函数返回指向类似Tue Aug 30 17:07:12 1994\n格式串的指针，且此字符串的构造来源于函数参数所指向的分解时间。字符串存储在静态变量中，每次调用asctime函数都可以修改此静态变量。

ctime函数返回指向字符串的指针，且该指针所指向的字符串是描述本地时间的。调用ctime(&t)就等价于调用asctime(localtime(&t))。

strftime函数和asctime函数一样，也把分解时间转换成字符串格式。然而，不同于asctime函数的是，strftime函数可以提供大量对时间格式化方法的控制。事实上，strftime函数类似于sprintf函数（➤22.8节），因为strftime函数会根据格式串（函数的第三个参数）把字符“写入”到字符串s（函数的第一个参数）中。格式串可能含有普通字符和表26-2显示的转换说明符。其中普通字符不会改变原样地复制给字符串s。函数的最后一个参数指向tm结构，此结构用来作为数据和时间信息源。函数的第二个参数是对存储在字符串s中的字符数量的限制。

表26-2　用于strftime函数的转换说明符

转换说明符	替换的内容
%a	缩写的星期名（比如，Tue）
%A	完整的星期名（比如，Tuesday）
%b	缩写的月份名（比如，Aug）
%B	完整的月份名（比如，August）
%c	完全的日期和时间（比如，Aug 30 17:07:12 1994）
%d	月内的天（01～31）
%H	24小时制的小时（00～23）
%I	12小时制的小时（01～12）
%j	年内的天（001～366）
%m	月份（1～12）
%M	分钟（00～59）
%p	AM/PM指示符（AM或PM）
%S	秒（00～61）①
%U	星期数（00-53）②
%w	星期几（0-6）
%W	星期数量（00-53）③
%x	完全的日期（比如，Aug 30 1994）
%X	完全的时间（比如，17:07:12）
%y	不含世纪的年（00～99）
%Y	含有世纪的年（比如，1994年）
%Z	时区名或缩写（比如，EST）
%%	%

① 允许两个额外的“闰秒”。
② 把第一个星期日看作是第一个星期的开始。
③ 把第一个星期看作是第一个星期的开始。

strftime函数不同于<time.h>中的其他函数，它对当前地区（地区➤25.1节）是很敏感的。

www.TopSage.com

改变LC_TIME类型可能会影响转换说明符的行为。表26-2中的例子严格地设为"C"地区。在德国地区内，%A可能会产生Dienstag而不是Tuesday。

26.3.3 程序：显示日期和时间

现在需要一个显示当前日期和时间的程序。当然，程序的第1步是要调用time函数来获得日历时间。第2步是把时间转换成字符串格式并显示出来。最简单的做法就是第二步调用ctime函数，它会返回一个指向含有日期和时间的字符串的指针，然后把此指针传递给puts函数或printf函数。

到目前为止，一切方法都没问题。可是，我们希望程序按照特定方式显示的日期和时间究竟是什么呢？假设这里需要如下形式的显示格式：

```
08-30-94 5:07p
```

ctime函数一直采用相同的日期和时间格式，所以对此无能为力。strftime函数相对好一些。使用strftime函数可以基本达到希望的显示要求，但是strftime函数无法显示没有零开头的单数字小时数。而且，strftime函数使用AM和PM而不是a和p来表示时间。 580

既然strftime函数不够好，那么这里还有另外一种选择：把日历时间转换为分解时间，然后从结构中抽取相关的信息，同时自己用printf函数或类似的函数把信息进行格式化。我们甚至可以使用strftime函数来实现某些格式化，然后用其他函数来完成整个工作。

下面这个程序举例说明了选择方案。程序用三种方法显示了当前日期和时间：一种格式是由ctime函数格式化的，一种格式是接近于我们需求的（由strftime函数产生的），还有一种则是正确的格式（由printf函数产生的）。采用ctime函数版本容易实现，采用strftime函数版本则有点困难，而采用printf函数版本更困难。

datetime.c

```
/* Displays the current date and time in three formats */
#include <stdio.h>
#include <time.h>

main()
{
  time_t current = time(NULL);
  struct tm *ptr;
  char date_time[19];
  int hour;
  char am_or_pm;
    /* print date and time in default format */                    581
  puts(ctime(&current));

    /* print date and time using strftime to format */
  strftime(date_time, sizeof(date_time),
         "%m-%d-%y  %I:%M%p\n", localtime(&current));
  puts(date_time);

    /* print date and time using custom formatting */
  ptr = localtime(&current);
  hour = ptr->tm_hour;
  if (hour <= 11)
    am_or_pm = 'a';
  else {
    hour -= 12;
    am_or_pm = 'p';
  }
  if (hour == 0)
    hour = 12;
  printf("%.2d-%.2d-%.2d  %2d:%.2d%c\n", ptr->tm_mon+1,
```

www.TopSage.com

大家网 TopSage.com

```
        ptr->tm_mday, ptr->tm_year, hour, ptr->tm_min,
        am_or_pm);

  return 0;
}
```

此程序的输出如下所示：

```
Tue Aug 30 17:07:12 1994
08-30-94 05:07PM
08-30-94 5:07p
```

问与答

问：虽然<stdlib.h>提供了6种把字符串转换成数的函数，但是它没有出现任何把数转换成字符串的函数。提供什么了呢？

答：某些C的库提供名为itoa的函数可以把数转换成为字符串。但是，使用这类函数不是一个好主意，因为它们不是C标准的内容且无法移植。把数转换成为字符串的最好做法就是调用sprintf函数（➤22.8节）：

```
char s[10];
int i;

sprintf(s, "%d", i);  /* stores i in the string s */
```

582

sprintf函数不但可以移植，而且可以对数的显示提供大量的控制。

***问：abort函数和SIGABRT信号之间是否存在联系呢？ (p.398)**

答：存在。调用时，abort函数实际产生SIGABRT信号。如果没有SIGABRT的处理函数，那么程序会像26.2节描述的那样异常终止。如果为SIGABRT安装了处理函数（通过调用signal函数➤24.3.2节），那么就调用处理函数。如果处理函数返回，那么随后程序会异常终止。但是，如果处理函数不返回（比如它调用了longjmp函数（➤24.4节）），那么程序就无法终止。

问：难道就没有办法为bsearch函数或qsort函数避免比较函数中所有讨厌的强制类型转换吗？

答：有，虽然在程序的其他地方会包含强制类型转换。一起来看看用于airmiles.c程序中的比较函数：

```
int compare_cities(const void *key_ptr),
                   const void *element_ptr
{
  return strcmp((char *) key_ptr,
                ((struct city_info *)element_ptr)->city);
}
```

我们可以用更自然的方式编写此函数，且不会有强制类型转换：

```
int compare_cities(const void *key_ptr),
                   const struct city_info *element_ptr)
{
  return strcmp(key_ptr, element_ptr->city);
}
```

但是，我们无法把新版的compare_cities函数传递给bsearch函数，后者需要指向函数的指针，且此函数必须带有两个void*类型的参数。解决方案是为bsearch函数调用添加一个强制类型转换：

```
ptr = bsearch(city_name, mileage,
              sizeof(mileage)/sizeof(mileage[0]),
              sizeof(mileage[0]),
              (int (*) (const void *, const void *))
                compare_cities);
```

这种方法是否可以使程序更易读？这需要你自己来评判。

问：为什么存在div函数和ldiv函数呢？难道只用/和%运算符不行吗？ (p.401)

答：div函数和ldiv函数同/运算符和%运算符不完全一样。回顾4.1节就会知道把/运算符和%运算用于负的运算数无法得到可移植的结果。如果i和j为负数，那么i/j的值是向上舍入还是向下舍入是由实现定义的，就像i%j结果的符号一样。但是，由div函数和ldiv函数计算的答案是不依赖于实现的。商向零舍入，余数则根据公式$n=q\times d+r$计算得出。公式里的n是原始数，q是商，d是除数，而r是余数。下面是几个例子：

583

```
 n    d    q    r
 7    3    2    1
-7    3    2    1
 7   -3   -2   -1
-7   -3    2   -1
```

效率则是div函数和ldiv函数存在的另一个原因。许多机器可以在一条指令里计算出商和余数，所以调用div函数或ldiv函数比分别使用/运算符和%运算符要快得多。

问：gmtime函数名字的出处在哪里？ (p.404)

答：编程语言不是唯一被标准化的内容。国际时间在1883年标准化时诞生了24个时区。因为许多人（特别是航海家和天文学家）需要一种方法来规定绝对时间而不是对某个时区而言的相对时间，所以建立格林威治标准时间（Greenwich Mean Time）就是基于子午线横穿英国的格林威治。最近，虽然格林威治标准时间又被重新命名为协调世界时（Goordinated Universal Time），但是人们早已经广泛使用gmtime函数了。

练习

26.1节

1. 重新编写max_int函数，要求不再把传递整数的个数作为第一个参数，我们必须采用0作为最后一个参数。**提示**：max_int函数必须至少有一个“正常”参数，所以不能把参数n移走，相反假设它是要比较的数之一。
2. 编写printf函数的简写版，要求新函数只有一种转换说明%d，并且在第一个参数后边的所有参数都必须是int类型的。
3. 编写下列函数：

```
char *vstrcat(const char *first, ...);
```

假设vstrcat函数除最后一个参数必须是个空指针（强制成char *类型）外，全部参数都是字符串。函数返回的指针指向动态分配的且含有参数拼接的字符串。如果没有足够的内存，那么vstrcat函数应该返回空指针。**提示**：vstrcat函数必须遍历参数两次：一次用来确定返回字符串需要的内存数量，另一次用来把参数复制到字符串中。

26.2节

4. 解释说明下列语句的含义。假设value是long int型的变量，p是char*型的变量。

584

```
Value = strtol(p, &p, 10);
```

5. 编写一条可以随机从7、11、15或19中取一个数分配给变量n的语句。
6. 编写一个可以随机返回double型值d（$0.0\leq d<1.0$）的函数。
7. 编写一个程序，用来模拟称为“掷双骰”游戏。程序要通过随机选择1到6之间的两个数来“滚动”一对模拟的骰子。如果两个数的和是7或11，那么程序显示信息Player wins。如果和为2、3或12，则显示Player loses。否则，程序要重复滚动骰子直到再一次达到原始和（Player wins）或者骰子合计为7（Player loses）为止。程序需要在每次模拟滚动后显示一下骰子的值。
8. (a) 编写一个程序，使它可以调用rand函数1000次并且显示函数返回的每个值的最低位（如果返回值是偶数，则为0；如果返回值为奇数，则为1。）你看到过什么模式吗？（rand的返回值的最后几位往往不是特别随机的。）

www.TopSage.com

(b) 如何改进rand函数的随机性，使它可以在一个小范围内产生数？

9. 编写两个函数来测试atexit函数。一个函数显示That's all,，另一个显示folks!。在程序终止时用atexit函数来注册这两个要调用的函数。请一定确保这两个函数按照正确顺序进行调用，只有这样才可以在屏幕上看到That's all, folks!。

26.3节

10. 编写一个程序，用clock函数来测算qsort函数对有100个整数的数组进行排序所用的时间，其中此数组初始时是反序排列元素的。确保该程序还可以用于有1000个整数的数组或10 000个整数的数组。
11. 编写一个函数，要求当向此函数传递年（比如，1996）时，函数返回一个表示在年份开始处的time_t值（即……第1小时的第1分钟的第1秒）。
12. 编写一个程序，提示用户录入一个日期（月、日和年）和一个整数n，然后显示n天后的日期。
13. 编写一个程序，提示用户录入两个日期，然后显示两个日期之间相差的天数。提示：请使用mktime函数和difftime函数。
14. 编写一个可以按照下列每种格式显示当前日期和时间的程序。请使用strftime函数来完成全部或大部分格式化的工作。

    (a) Tuesday, August 30, 1994　　05:07p

    (b) Tue, 30 Aug 94　　17:07

585 (c) 08/30/94　5:07:12 PM

www.TopSage.com

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 | 软考历年真是解析与答案

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…