

计 算 机 科 学 丛 书

(原书第4版)

# C语言解析 教程

C by Dissection  
The Essentials of C  
Programming  
Fourth Edition

(美) Al Kelley 著  
Ira Pohl  
麻志毅 译



机械工业出版社  
China Machine Press



Addison-Wesley

本书通过细致地编写一些程序，对编程过程做了全面介绍，说明了C语言的关键特征。阅读本书不需要编程背景，想要学习C语言的计算机初学者和有编程经验的程序员都可以使用本书。

### 通过“解析”教学

本书利用“解析”方法介绍了编程概念。解析是作者们首次提出的一种独特的教学方法，用以指出程序代码的关键特征。它类似于结构化的代码逐步排演，解释新遇到的编程元素和程序代码中的惯用法。本版增加了新的解析，以强调这种方法，加强对编程要素的理解。

### 对编程的整体探讨

本书的每一章都给出了一些有详细注释的程序，通过对这些程序的学习，读者能从整体上不断改善编程技巧。从一开始，本书就向读者介绍完整的程序，并在前面向读者介绍如何编写作为结构化编程的主要特征的函数。本版对指针和内存管理部分做了扩充，并用更多的解析例子强化对递归的讨论。

### 传统编程和面向对象编程

本书作者了解到：很多程序员正开始学Java语言并想学习C语言或正开始C语言并想学习Java语言。本书讨论了从学习一种语言过渡到学习另一种语言时会出现困难的那些主题（例如输入/输出和数据类型转换），增加了Java和C++练习，引导读者对比传统编程和面向对象编程间的差别。通过学习最后一章“从C到C++”，你会感到转向C++编程并不是什么难事。

ISBN 7-111-09336-4



9 787111 093367



华章图书

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

北京市西城区百万庄南街1号 100037

购书热线：(010)68995259, 8006100280 (北京地区)

总编信箱：chiefeditor@hzbook.com

ISBN 7-111-09336-4/TP · 2107

定价：48.00 元

计算机科学丛书

# C语言解析教程

(原书第4版)

(美) Ai Kelley 著  
Ira Pohl

麻志毅 译



机械工业出版社  
China Machine Press

C语言在全世界的学术界和工业界都广泛地应用,同时它也是计算机科学教育编程课程的首选编程语言。本书通过应用作者首次提出的一种独特的教学方法——解析,对C语言的关键特征及编程过程做了细致全面的介绍。书中大量的练习和带有详细注释的工作程序会使读者从整体上提高编程能力、掌握编程技巧。本书的附录还给出了大量标准库函数供读者参考。

本书全面介绍了C语言的数组、指针、函数、串处理、文件处理和软件工具等方面的内容,本书是计算机专业本科生的极佳教材。

A1 Kelley, Ira Pohl: C by Dissection: The Essentials of C Programming, 4E.

Original edition copyright © 2001 by Addison Wesley Longman, Inc.

Chinese edition published by arrangement with Addison Wesley Longman, Inc.

All rights reserved.

本书中文简体字版由美国Addison Wesley公司授权机械工业出版社独家出版。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。

本书版权登记号:图字:01-2000-4101

### 图书在版编目(CIP)数据

C语言解析教程/(美)凯利(Kelley, A.), (美)波尔(Pohl, I.)著;麻志毅译.  
-北京:机械工业出版社,2002.1

(计算机科学丛书)

书名原文:C by Dissection: The Essentials of C Programming, 4E

ISBN 7-111-09336-4

I. C… II. ①凯… ②波… ③麻… III. C语言-程序设计 IV. TP312

中国版本图书馆CIP数据核字(2001)第065414号

机械工业出版社(北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑:杨海玲

北京第二外国语学院印刷厂印刷·新华书店北京发行所发行

2002年1月第1版第1次印刷

787mm×1092mm 1/16·27.25印张

印数:0 001-5 000册

定价:48.00元

凡购本书,如有倒页、脱页、缺页,由本社发行部调换

MS2PP/02

## 出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域中取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall, Addison-Wesley, McGraw-Hill, Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum, Stroustrup, Kernighan, Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及收藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：针对本科生的核心课程，剔抉外版菁华而成“国外经典教材”系列；对影印版的教材，则单独开辟出“经典原版书库”；定位在高级教程和专业参考的“计算机科学丛书”还将保持原来的风格，继续出版新的品种。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图

书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

电子邮件：[hzedu@hzbook.com](mailto:hzedu@hzbook.com)

联系电话：(010) 68995265

联系地址：北京市西城区百万庄南街1号

邮政编码：100037

# 专家指导委员会

(按姓名笔画顺序)

尤晋元  
张立昂  
周克定  
高传善  
石教英  
邵维忠  
郑国梁  
裘宗燕

王 珊  
李伟琴  
周傲英  
梅 宏  
吕 建  
陆丽娜  
施伯乐  
戴 葵

冯博琴  
李师贤  
孟小峰  
程 旭  
孙玉芳  
陆鑫达  
钟玉琢

史忠植  
李建中  
岳丽华  
程时端  
吴世忠  
陈向群  
唐世渭

史美林  
杨冬青  
范 明  
谢希仁  
吴时霖  
周伯生  
袁崇义

# 译者序

在过去的20年内，由于C语言具有丰富的运算符和数据类型、使用上灵活方便、生成的目标代码质量高和程序的运行效率高等特点，它受到了人们的普遍欢迎，全世界的学术界和工业界都在广泛地使用它，特别是大多数高校都把它作为编程的主要教学语言。C++在一些有意义的方面改善了C，与C是高度兼容的，特别是它增加了面向对象特征，而面向对象方法与技术在计算机科学技术领域已占据了主流地位。利用C语言进行程序设计与开发能锻炼程序员进行抽象程序设计的能力，也能为学习C++这种具有更为抽象的概念和技术的语言奠定基础，况且C程序设计特性在C++中得到了频繁的使用。

本书的英文版自问世以来很受欢迎，现已发行了第4版，这完全是由于作者对书中内容的独具匠心的编写与组织。作者们用他们首次提出的“解析”这种独特教学法解释了新遇到的编程元素和程序代码中的惯用法。本书给出了一些经过详细注释的完整程序，使读者能不断地改善编程技巧。此外，书中还注重于说明如何从C转向C++，通过学习，读者会感到转向C++编程并不是什么难事。

本书覆盖的内容较为广泛，译者希望读者能从中获益，同时恳请广大读者对翻译中可能出现的疏漏和错误予以批评指正。

译者

2000年5月



麻志毅，男，北京大学计算机科学技术系副教授。1999年在东北大学获得博士学位，同年到北京大学计算机科学技术系作博士后，出站后留校工作。现已发表学术论文30余篇，主持或参加政府科研项目十余项。主要研究领域为软件工程、面向对象技术和计算语言学。



# 前言

目前，全世界的学术界和工业界都在广泛使用ANSI C编程语言。在很多教育机构，它是编程课程的首选语言和用于计算机科学教育的语言。这主要是因为从高级的课程到介绍性的课程都可选择C。此外，C带有很多有用的库，并由复杂的集成环境所支持。ANSI C的改进弥补了一些在传统C中发现的不足，例如弱类型规则、这些改进以及C作为开发系统、图形和数据库的语言的广泛影响使得它成为讲授编程和计算机科学的首选。

通过详细地展开工作程序，本书对编程过程进行了全面的介绍，说明了C编程语言的关键特征。本书用一种循序渐进的方式对程序代码进行了全面说明。书中的代码已经在几个平台上测试过，你可从因特网站点aw.com获得这些代码。

本书中的代码能用于大多数C语言系统，例如MacOS、MS-DOS、OS/2、UNIX和Windows操作系统中的C系统。

## 解析 (Dissection)

通过用解析的方法详细地展开工作C程序，本书向读者提供了清晰而完整的编程过程。解析是作者于1984年研制的一种独特的教学法，用于说明工作代码的关键特征。解析类似于结构化的代码预排。其意图是向读者解释新遇到的编程元素和工作代码中含有的习惯语法。本书以一种按部就班的易于遵循的方式对程序和函数进行解释。在不同上下文中的应用充分强化了关键思想。

## 无背景假设

本书假设没有任何编程背景，学生和初学的计算机用户都可使用，不熟悉C的有经验的程序员也会受益于C语言细致的结构化表达。对于学生而言，本书可作为计算机科学或编程的第一门教程。

对于其他学科，本书也适合于CSI课程或初始编程课程。本书的每章都给出了一些带有详细注释的程序，以用整体的方式使学生不断地提高编程技巧。本书一开始就向学生介绍完整的程序，在本书的前面部分引导学生编写作为结构化编程主要特征的函数。函数与程序的关系就像段落和文章一样。能胜任编写函数是熟练程序员的标志，因而要强调这点。书中的例子和练习是富有内容和难度的，因而教师可精选适合于学生的作业。

## 专有特征

《C语言解析教程》(《C by Dissection: The Essentials of C programming》)的第4版增加了一些新的专有特征：

- 全部运用最新的ANSI C，例如函数原型。
- 每章末尾的“转向C++”一节有助于转换到面向对象的C++。
- 早期的对多文件程序的解释使得程序员能正确地编制模块代码和产品，并能正确地使

用库。

- 早期的对简单递归的解释反映出在早期阶段的计算机科学课程中对递归的提前介绍。
- 介绍了程序的正确性和输入的安全性。
- 对函数和指针进行深入的解释，这些概念通常是初学者的绊脚石。
- 对二维数组的讨论反应了科学家和工程师对C越来越多的使用。
- 对递归的深入研究反应了计算机科学家为了实现复杂算法对C的越来越多的使用。
- 可选的C++和Java练习。
- 附录中的位运算符改善了对主流语言特征表示。

## 章节特征

每章都含有如下的教学法元素：

**解析** 每章都有几个重要的示例程序。本书用解析方法解释这些程序的主要元素。这种新的编程思想的讨论，有助于读者在首次遇到这些思想时对它们的理解。

**编程风格和方法** 本书到处强调编程风格和方法。书中较早地提出了诸如结构化分支语句、嵌套控制流、自顶向下设计和结构化编程这样的重要概念。从一开始的详细解释就采用了一致和正确的编码风格，以强调编码风格的重要性和合理性。本书使用的编码风格是使用各种C语言编程的专业人员常用的。因为C支持函数原型和强类型检查，这种风格已被全面采用。

**工作代码** 从一开始，本书就向学生介绍完全可执行的程序。使用可执行的代码，学生能更好地理解和评价正在讨论中的编程思想。很多程序和函数都通过解析来解释。对编程思想的变化经常会体现在习题中。

**常见的编程错误** 书中描述了很多典型的编程错误以及避免这些错误的方法。学习编程语言遇到的大多数挫折都是由于遇到含糊的错误引起的。很多书讨论正确编码，而让读者用反复实验的方法去找出错误。本书说明了C中典型的错误是如何产生的以及应该怎样来改正错误。

**系统考虑** 在几乎任何的计算机和大多数操作系统中C都是可用的，但从一个系统到另一个系统在行为上偶尔会存在不同。本书描述了这些不同。在ANSI C和传统C之间也存在不同，这些不同在本书中也做了描述。所有的程序通常都在一些不同的环境下进行了测试。本书强调编写可移植的独立于系统的代码。

**转向C++** 每章的末尾有一个可选的节，它描述了转向C++所需的编程元素。本书对这些内容也给出了练习。就绝大部分而言，C是C++编程语言的子集，在转向C++之前很多学生先学习C，本书有助于这种自然过渡。对于希望完全精通C++的读者来说，可以将本书与Ira Pohl所著的《*C++ for C Programmers, Third Edition*》(Addison Wesley Longman, Inc., Reading, MA 1999, ISBN 0-201-39519-3)配合起来使用。

**小结** 在每章的末尾，我们给出了一个简洁的这一章的要点列表，对读者起复习的作用，以加深理解这一章提出的新思想。

**练习** 各章末的练习测试读者的语言知识。很多练习要与阅读正文交互式地进行，这样可以促进读者自我调整学习进度。还有一个练习特征：一些练习非常详细地着眼于一个主题，另外一些练习能扩充读者的知识以达到使用的高级境界。

## 作为学生的教材

本书作为教学生编程的教材，在第一学期的课程中可以使用第1章到第10章，其中包含了C编程语言的数组、指针和串的全部用法。第二学期课程致力于高级数据类型、文件处理和软件工具，这些内容包含在第11章到第15章。对于为已经具有一些编程知识（不一定是C语言的知识）的学生而设立的课程，教师也可以使用到本书的所有主题。本书也能用作需要学生使用C的其他计算机科学课程的教材。

## 交互式环境

本书可用于交互式环境。通过键盘和屏幕的实验完全受到鼓励。对PC机而言有很多厂商提供了交互式C/C++系统，例如Borland、IBM、Metroworks、Microsoft和Symantec。

## 作为专业程序员的教材

虽然本书是为初级程序员准备的，但对于有经验的程序员来说，本书也是对C语言较好的介绍。结合Al Kelley和Ira Pohl所著的《*A Book on C, Fourth Edition*》（Addison Wesley Longman, Inc., Reading, MA 1998, ISBN 0-201-18399-4），计算机专业人员会获得对该语言的全面理解，包括在MS-DOS和UNIX下有关使用它的关键之处。作为一种结合，这两本书提供了对C编程语言的完整分析以及在别处难以获得的用法。此外，本书与Ira Pohl所著的《*C++ for C Programmers, Third Edition*》和Ira Pohl所著的《*Object-Oriented Programming Using C++, Second Edition*》（Addison Wesley Longman, Inc., Reading, MA 1997, ISBN 0-201-89550-1）相结合，还向学生或专业人员提供了对面向对象语言C++的完整分析。

## ANSI C标准

ANSI 是美国国家标准协会（American National Standards Institute）的英文首写字母的缩写，该协会致力于包括编程语言在内的多种标准的制订。特别是ANSI委员会X3J11负责制订编程语言C的标准。在20世纪80年代后期，该委员会创建了称为“ANSI C”或“标准C”的标准草案。到1990年，该委员会完成了这项标准，国际标准化组织（International Standardization Organization, ISO）也批准了这项标准。因而，ANSI C或ANSI/ISO C标准在国际上获得承认。

ANSI C标准规定了用C编写的程序的格式，并确定了如何解释这些程序。该标准的目的是促进C语言程序在各种机器上的可移植性、可靠性、可维护性和有效执行。所有主要的C编译器都遵循ANSI C标准。

Al Kelley

Ira Pohl

University of California, Santa Cruz

# 目 录

出版者的话	
专家指导委员会	
译者序	
前言	
第1章 编写ANSI C程序	1
1.1 准备编程	1
1.2 第一个程序	2
1.3 变量、表达式和赋值	5
1.4 初始化	8
1.5 #define和#include的用法	8
1.6 printf()和scanf()的用法	9
1.6.1 printf()的用法	10
1.6.2 scanf()的用法	11
1.7 while语句	13
1.8 问题求解：计算总和	14
1.9 风格	15
1.10 常见的编程错误	16
1.11 系统考虑	17
1.11.1 编写和运行C程序	17
1.11.2 中断程序	18
1.11.3 输入文件尾标识	18
1.11.4 输入和输出的重定向	18
1.12 转向C++	19
小结	20
练习	20
第2章 词法元素、运算符和C系统	26
2.1 字符和词法元素	27
2.2 注释	28
2.3 关键字	29
2.4 标识符	29
2.5 常量	30
2.6 串常量	31
2.7 运算符和标点符号	31
2.8 运算符的优先级和结合性	32
2.9 增量运算符和减量运算符	33
2.10 赋值运算符	34
2.11 例子：计算2的幂	36
2.12 C系统	36
2.12.1 预处理器	37
2.12.2 标准库	37
2.13 风格	39
2.14 常见的编程错误	40
2.15 系统考虑	41
2.16 转向C++	42
小结	42
练习	43
第3章 控制流	48
3.1 关系、等式和逻辑运算符	48
3.2 关系运算符和表达式	49
3.3 等式运算符和表达式	50
3.4 逻辑运算符和表达式	50
3.5 复合语句	53
3.6 空语句	53
3.7 if和if-else语句	53
3.8 while语句	56
3.9 问题求解：找最大值	57
3.10 for语句	58
3.11 问题求解：组合数学	59
3.12 问题求解：布尔变量	61
3.13 逗号运算符	61
3.14 do语句	62
3.15 goto语句	63
3.16 break和continue语句	64
3.17 switch语句	65
3.18 嵌套的控制流	65
3.19 条件运算符	66
3.20 风格	67
3.21 常见的编程错误	68

3.22 系统考虑 .....	70	6.2 基本数据类型 .....	128
3.23 转向C++ .....	71	6.3 字符和数据类型char .....	129
小结 .....	71	6.4 数据类型int .....	130
练习 .....	72	6.5 整数类型short、long和unsigned .....	131
第4章 函数和结构化编程 .....	78	6.6 浮点类型 .....	132
4.1 函数调用 .....	78	6.7 sizeof运算符 .....	134
4.2 函数定义 .....	78	6.8 数学函数 .....	134
4.3 return语句 .....	80	6.9 转换和类型转换 .....	136
4.4 函数原型 .....	82	6.9.1 整型提升 .....	136
4.5 自顶向下设计 .....	83	6.9.2 常用的算术转换 .....	136
4.6 程序的正确性: assert()宏 .....	86	6.9.3 类型转换 .....	137
4.7 从编译器的角度来看函数声明 .....	87	6.10 问题求解: 计算利息 .....	138
4.8 问题求解: 随机数 .....	88	6.11 风格 .....	141
4.9 函数定义次序的可选风格 .....	89	6.12 常见的编程错误 .....	142
4.10 开发一个大程序 .....	90	6.13 系统考虑 .....	143
4.11 模拟: 正反面游戏 .....	93	6.14 转向C++ .....	143
4.12 调用和按值调用 .....	95	小结 .....	144
4.13 递归 .....	96	练习 .....	145
4.14 风格 .....	97	第7章 枚举类型和typedef .....	151
4.15 常见的编程错误 .....	97	7.1 枚举类型 .....	151
4.16 系统考虑 .....	99	7.2 typedef的用法 .....	152
4.17 转向C++ .....	100	7.3 例子: 石头、剪刀、布游戏 .....	154
小结 .....	102	7.4 风格 .....	158
练习 .....	103	7.5 常见的编程错误 .....	159
第5章 字符处理 .....	110	7.6 系统考虑 .....	160
5.1 数据类型char .....	110	7.7 转向C++ .....	160
5.2 getchar()和putchar()的用法 .....	112	小结 .....	160
5.3 例子: 大写 .....	115	练习 .....	161
5.4 ctype.h中的宏 .....	117	第8章 函数、指针和存储类型 .....	165
5.5 问题求解: 重复字符 .....	117	8.1 指针声明和赋值 .....	165
5.6 问题求解: 对单词计数 .....	118	8.2 地址和间接访问 .....	166
5.7 风格 .....	120	8.3 指向void的指针 .....	169
5.8 常见的编程错误 .....	121	8.4 引用调用 .....	170
5.9 系统考虑 .....	122	8.5 作用域规则 .....	171
5.10 转向C++ .....	123	8.6 存储类型 .....	172
小结 .....	125	8.6.1 存储类型auto .....	172
练习 .....	125	8.6.2 存储类型extern .....	172
第6章 基本数据类型 .....	128	8.6.3 存储类型register .....	174
6.1 声明和表达式 .....	128	8.6.4 存储类型static .....	174

8.7 静态外部变量 .....	175	10.10 系统考虑 .....	221
8.8 缺省的初始化 .....	176	10.11 转向C++ .....	221
8.9 例子: 字符处理 .....	176	小结 .....	221
8.10 函数声明和函数定义 .....	179	练习 .....	222
8.11 类型限定符const和volatile .....	179	第11章 递归 .....	225
8.12 风格 .....	180	11.1 递归问题求解 .....	225
8.13 常见的编程错误 .....	181	11.2 例子: 在屏幕上绘制图案 .....	228
8.14 系统考虑 .....	182	11.3 用递归处理串 .....	229
8.15 转向C++ .....	183	11.4 分而治之的方法 .....	230
小结 .....	184	11.5 例子: 汉诺塔 .....	231
练习 .....	185	11.6 风格 .....	237
第9章 数组和指针 .....	190	11.7 常见的编程错误 .....	238
9.1 一维数组 .....	190	11.8 系统考虑 .....	239
9.1.1 初始化 .....	191	11.9 转向C++ .....	239
9.1.2 下标 .....	192	小结 .....	240
9.2 例子: 分别对每个字母计数 .....	192	练习 .....	240
9.3 数组和指针间的关系 .....	194	第12章 结构和抽象数据类型 .....	245
9.4 指针运算和元素尺寸 .....	195	12.1 声明结构 .....	245
9.5 把数组传递给函数 .....	195	12.2 访问成员 .....	246
9.6 排序算法: 冒泡排序 .....	196	12.3 运算符的优先级和结合性: 总结 .....	248
9.7 二维数组 .....	197	12.4 结构、函数和赋值 .....	249
9.8 多维数组 .....	199	12.5 问题求解: 学生记录 .....	251
9.9 动态内存分配 .....	200	12.6 结构的初始化 .....	253
9.10 风格 .....	201	12.7 typedef的用法 .....	253
9.11 常见的编程错误 .....	203	12.8 自引用结构 .....	254
9.12 系统考虑 .....	203	12.9 线性链表 .....	255
9.13 转向C++ .....	204	12.10 对链表的操作 .....	256
小结 .....	205	12.11 计数和查找 .....	257
练习 .....	206	12.12 插入和删除 .....	258
第10章 串和指针 .....	211	12.13 风格 .....	259
10.1 串结束标志 .....	211	12.14 常见的编程错误 .....	260
10.2 串的初始化 .....	212	12.15 系统考虑 .....	260
10.3 例子: 心情愉快 .....	212	12.16 转向C++ .....	261
10.4 用指针处理串 .....	214	小结 .....	266
10.5 问题求解: 单词计数 .....	217	练习 .....	266
10.6 把参数传递给main() .....	218	第13章 输入/输出和文件 .....	269
10.7 标准库中的串处理函数 .....	218	13.1 输出函数printf() .....	269
10.8 风格 .....	220	13.2 输入函数scanf() .....	272
10.9 常见的编程错误 .....	220	13.2.1 控制串中的指示 .....	273

13.2.2 普通字符	273	14.10 其他有用的工具	309
13.2.3 空白字符	273	14.11 风格	310
13.2.4 转换说明	273	14.12 常见的编程错误	310
13.2.5 输入流中的浮点数	275	14.13 系统考虑	311
13.2.6 使用扫描集	275	小结	311
13.2.7 返回值	276	练习	312
13.2.8 一个scanf()的例子	276	第15章 从C到C++	315
13.3 函数sprintf()和sscanf()	277	15.1 为什么转到C++	315
13.4 函数fprintf()和fscanf()	277	15.2 类和抽象数据类型	317
13.5 访问文件	278	15.3 重载	318
13.6 例子:对文件行距加倍	279	15.4 构造器和析构器	320
13.7 使用临时文件和得体的函数	281	15.5 继承	321
13.8 随机地访问文件	283	15.6 多态性	322
13.9 风格	284	15.7 模板	324
13.10 常见的编程错误	285	15.8 C++中的异常	325
13.11 系统考虑	286	15.9 面向对象编程的益处	325
13.12 转向C++	287	15.10 风格	325
小结	289	15.11 常见的编程错误	326
练习	290	15.12 系统考虑	326
第14章 软件工具	295	小结	326
14.1 在C程序中执行命令	295	练习	327
14.2 环境变量	296	附录A 标准库	329
14.3 C编译器	297	附录B 预处理器	354
14.4 创建库	299	附录C 位运算符	363
14.5 使用profiler	300	附录D ANSI C与传统C的比较	375
14.6 关于时间的编码	303	附录E ASCII字符编码表	380
14.7 dbox的用法	305	附录F 运算符的优先级和结合性	381
14.8 make的用法	306	索引	382
14.9 touch的用法	309		

# 第1章 编写ANSI C程序

本章向读者介绍ANSI C编程世界。本章要讨论一些关于编程的一般思想，并要对一些基本的程序详尽地进行解释。本章提出的基本思想是后面各章中出现的更完整的说明的基础。C的基本输入/输出函数是本章的一个重点。把信息输入到机器和从机器输出信息是掌握任何编程语言的的第一项任务。

C广泛地把printf()和scanf()函数分别用于输入和输出。本章对这两个函数的用法进行了解释。本章中讨论的其他主题包括用于存储值的变量的用法以及用于改变变量值的表达式和赋值语句的用法。本章也包括对while语句的讨论。一个例子用于表明while语句如何提供重复的操作。

本章和本书给出了很多例子，包括很多要经常用于解析的完整程序，这使得读者能详细地理解每个结构如何工作。本章介绍的主题在后续章节中也会出现，在适当的地方会有更详细的解释。这种螺旋式的学习方法向程序员强调C编程的思想和技术本质。

每章以标题为“转向C++”的一节结束。C++大部分是C的超集。通过学习C++，你也可以学习到C的核心语言。Ira Pohl所著的本书的续集《C for C++ Programmers》第2版，讲述本书中没出现的C++的其余部分。大多数章有基于Java的练习。Java是部分基于C的，然而，与C++不同，一些C的概念不适合于Java或在Java中有不同的含义。正如在姊妹篇——Ira Pohl和Charlie McDowell所著的《Java by Dissection》——中所发现的那样，越来越多的有Java背景的人开始用C编程。现代的程序员在使用所有3种基于C的语言中必须是得心应手的。

## 1.1 准备编程

编写程序用于指示计算机完成特定的任务或解决特定的问题。步进式地完成所需任务的过程被称为算法(algorithm)。因而编程是沟通算法和计算机的活动。我们都习惯于用语言给别人下指令，让其完成该指令。编程的过程与此类似，只是机器不能容忍多义性，必须要用精确的语言不厌其烦地详述机器要执行的所有步骤。

### 编程过程

- 1) 描述任务。
- 2) 找出解决问题的算法。
- 3) 用C对算法编码。
- 4) 测试代码。

计算机是由处理器、内存和输入/输出设备这三个主要部件组成的数字电子机器。处理器也称作中央处理单元(central processing unit, CPU)。处理器执行存储在内存中的指令。数据和指令一样也存储在内存中，按一定的要求方式，通常处理器按指令操纵数据。输入/输出设备从机器的外部介质获取信息，并向外部介质提供信息。输入设备一般是终端键盘、磁盘驱动器和磁带机。输出设备一般是终端屏幕、打印机、磁盘驱动器和磁带机。机器的物理组成可



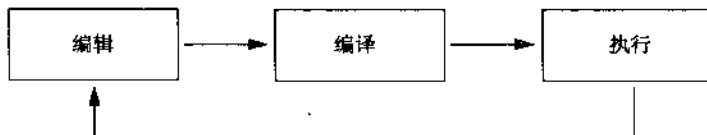
能是很复杂的，但用户不必关心这些细节。机器的操作系统负责协调机器的资源。

操作系统由一些专门的程序组成，有两个主要的用途。首先，操作系统在整体上监视和协调机器的资源。例如，当在磁盘上创建文件时，操作系统负责在磁盘的适当位置上定位文件，以及掌管文件名、文件的尺寸和创建的日期这样的细节。此外，操作系统向用户提供工具，其中的很多工具对于C程序员来说是很有用的。文本编辑器和C编译器就是其中的两个极为重要的工具。

我们假设读者能够用文本编辑器创建和修改含有C代码的文件。C代码也称作源代码(source code)，含有源代码的文件称为源文件(source file)。在创建了含有源代码的文件(程序)后，就可以调用C编译器。这个过程是与系统相关的(参见1.11.1节“编写和运行C程序”)。例如，在一些系统中，我们能用命令`cc pgm.c`调用编译器，其中`pgm.c`是含有程序的文件名。如果在`pgm.c`中没有错误，该命令产生一个能够运行或执行的可执行文件(executable file)。虽然我们把这看作是在编译程序，但实际上发生的事情比这要复杂得多。

在第14章“软件工具”中，我们将更详细地讨论编译过程。在这里我们仅想提及基础知识。当我们编译一个简单程序时，要发生三个独立的操作：首先调用预处理器，然后调用编译器，最后调用加载器。通过加进其他文件和做一些其他的改变，预处理器修改源代码的拷贝。(在1.5节“`#define`和`#include`的用法”中，我们要进一步讨论预处理器。)编译器把编辑的结果转换成目标代码(object code)，加载器产生最终可执行的文件。含有目标代码的文件称为目标文件(object file)。目标文件与源代码文件不同，它通常是不可读的。当我们说编译程序时，实际意味着是调用预处理器、编译器和加载器。对于简单程序而言，用简单的命令做这些就足够了。

在程序员编写一个程序后，他必须要编译和测试他的程序。如果需要修改，就必须再次编辑源代码。这样，编程过程由下述循环组成：



在程序员对程序的性能满意后，就结束循环。

## 1.2 第一个程序

对任何人而言，学习编程的第一项工作都是在屏幕上显示。让我们从编写在屏幕上显示短语“Hello, world!”的Kernighan和Ritchie程序开始。完整的程序如下：

```
/* The traditional first program in honor of
   Dennis Ritchie who invented C at Bell Labs in 1972 */

#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

用文本编辑器，程序员把这段程序键入到一个以`.c`结尾的文件。所选择的文件名应该有助于记忆。假设我们用`hello.c`作为已经编写的程序的文件名。当编译后执行该程序时，在屏幕

上会出现:

Hello, world!



### 对程序hello的解析

• /\*The traditional first program in honor of Dennis Ritchie who invented C at Bell Labs in 1972\*/

这行是注释。编译器忽略了由开始的符号对/\*和结束的符号对\*/括起来的文本。它是为阅读程序的人提供的文档。

• #include <stdio.h>

用#开始的行被称为预处理指令(preprocessing directive)。它们用于与预处理器通信。#include指令使得预处理器在代码的当前位置上加入标准头文件stdio.h的拷贝。这个头文件是C系统提供的。<stdio.h>的尖括号指明该文件能在“通常的位置”找到,这是与系统相关的。我们引入这个文件是因为它含有关于printf()函数的信息。

• int main(void)

每一个程序都有一个命名为main的函数,程序要从这里开始执行。main后面的圆括号告诉编译器这是一个函数,关键字int声明返回类型的值是整型值。关键字void表明该函数没有参数。

• {

各函数体以左花括号开始,相应的右花括号必须出现在函数末尾。我们的风格是每个括号单占一行,并靠左放置。这种括号也用于把语句组织在一起。

• printf()

C系统包含一个能用于程序中的标准函数库。该函数来自于在屏幕上显示的函数库。我们引入头文件stdio.h是因为它向编译器提供了一定的关于函数printf()的信息，即它的函数原型。(4.4节“函数原型”中讨论了关于函数原型的内容。)

• printf("hello,world!\n");

这是用一个单参数(即串"Hello,world!\n")对函数printf()的调用或引用。C中的串常量是用双引号括起来的一串字符。本串是函数printf()的一个参数，它控制显示内容。在串尾的两个字符\n代表一个被称为换行的单字符，这是一个非打印字符，其作用是把光标移到屏幕的下一行的开头。注意本行用一个分号结尾。C中的所有声明和语句都用分号结尾。

• return 0;

main()向操作系统返回整数值零。零意味着程序已经成功结束。非零值用于告诉操作系统main()没有成功。

• }

该右花括号与上面的左花括号相匹配，它结束了函数main()。 ■

函数printf()产生的效果是在屏幕上连续地显示，当读到一个换行字符时，会移动到一个新行。屏幕是一个从左到右和从上到下显示的二维显示器。为了读起来方便起见，程序员必须考虑输出在屏幕上的布局。

下面用两个printf()语句重写我们的程序。虽然程序看起来不同，但输出是一样的。

```
#include <stdio.h>

int main(void)
{
    printf("Hello, ");
    printf("world!\n");
    return 0;
}
```

注意作为第一个printf()语句的参数的串用一个空格字符结尾。如果此处没有这个空格字符，输出的Hello world!间将没有空格。

作为该程序的最后一个变型，让我们再增加一个短语“Hello,universe!”，并在两行显示。

```
#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n");
    printf("Hello, universe!\n");
    return 0;
}
```

执行这个程序时，如下信息会出现在屏幕上：

```
Hello, world!
Hello, universe!
```

注意在main()体内的两个printf()语句可以用下面的一个语句代替：

```
printf("Hello, world!\nHello, universe!\n");
```

在标准库中像printf()这样的有用函数的可用性是C的一个重要特征。虽然在技术上标准库不是C语言的一部分，但它是C系统的一部分，因为在标准库中的函数在C系统中随处可

用，程序员在程序中通常要使用它们。

### 1.3 变量、表达式和赋值

我们的第一个程序说明了printf()用于输出的用法。在下面的程序中，我们要说明操纵整数值的变量的用法。变量用于存储值。由于要用不同类型的变量存储不同的数据，所以必须要指明各变量的类型。为了说明我们的思想，我们基于Hesperus号（金星号）船的失事编写一个程序。这次海难于1839年发生在马萨诸塞州的格洛斯特附近的诺曼的悲哀（Norman's Woe）暗礁的海域，据此Henry Wadsworth Longfellow写了一首著名的词。暗礁在约7英寻（1英寻=6英尺）深的水下。在程序中我们要把这个深度转换为其他测量单位。如下是我们的程序使用的算法。

#### Hesperus的算法

- 1) 把英寻数赋值给一个变量。
- 2) 把英寻转换为英尺，并存到一个变量中。
- 3) 把英尺转换为英寸，并存到一个变量中。
- 4) 把不同的测量单位整齐地显示在屏幕上。

在编写C代码时，我们必须选择一个适当的变量集。在当前的情况下，我们自然要选择整数变量。我们必须保证转换表达式使用了正确的常量，并且输出也必须要便于阅读。

```
#include <stdio.h>

int main(void)
{
    int    inches, feet, fathoms;

    fathoms = 7;
    feet = 6 * fathoms;
    inches = 12 * feet;
    printf("Wreck of the Hesperus:\n");
    printf("Its depth at sea in different units:\n");
    printf("    %d fathoms\n", fathoms);
    printf("    %d feet\n", feet);
    printf("    %d inches\n", inches);
    return 0;
}
```

编译并运行该程序，在屏幕上会出现如下信息：

```
Wreck of the Hesperus:
Its depth at sea in different units:
    7 fathoms
    42 feet
    504 inches
```

#### 对程序depth的解析

- #include <stdio.h>

在使用printf()的任何程序中都包含标准头文件stdio.h。随后我们将看到编译器为什么需要这个文件。

- int main(void)
  - {
 int inches, feet, fathoms;

函数main()体内的第一行是一个声明。变量inches、feet和fathoms被声明为类型int，int是C的基本类型之一。类型为int的变量接纳整数值。在程序中的所有变量在使用

前都应该先声明。声明和语句都用分号结尾。

```
• fathoms = 7;
```

这是一个赋值语句。等号(=)是C的一个基本运算符。等号右边的表达式的值被赋值给它左边的变量，此处右边的常量表达式是7，该值被赋给变量fathoms。

```
• feet = 6 * fathoms;
  inches = 12 * feet;
```

这两个语句是赋值语句。因为1英寻等于6英尺，并要把给定的英寻数转换成等价的英尺数，所以我们必须要对其乘以6。星号(\*)是乘法运算符。表达式6 \* fathoms的值被赋给变量feet。因为变量fathoms的当前值是7，所以表达式6\*fathoms的值为42，该值被赋给变量feet。为了转换成英寸，必须对feet乘以12。表达式12 \* feet的值被赋给变量inches。

```
• printf("  %d fathoms\n", fathoms);
```

这个printf()语句有两个参数，分别是"%d fathoms\n"和fathoms。注意它们用逗号隔开。函数printf()中的第一个参数总是一个称为控制串的串。在本例中，控制串含有转换说明%d。转换说明也称作格式(format)。格式%d使得第二个参数中的表达式(本例为fathoms)的值按十进制整数的格式显示。控制串中的一般字符，即不用于构成格式的字符，被简单地显示在屏幕上。注意本例中的控制串用三个空格开始，这使得本行被缩进显示。该程序中的其他printf()语句都与此类似。 ■

在C中，所有的变量用于表达式和语句之前都必须先声明。简单程序的一般形式为：

```
preprocessing directives

int main(void)
{
    declarations

    statements
}
```

在这个文件的顶部可以有像#include这样的预处理指令行。在main()体内，声明必须出现在语句前。声明告诉编译器什么类型的数据可以存于那个变量中，这使得编译器能够留出适当量的用于保存数据的内存。我们已经了解了整型数据的用法，马上我们将讨论字符型数据和浮点型数据。程序中的语句执行所要求的计算，并在屏幕上显示信息。

变量名，也称为标识符(identifier)，由字母、数字和下划线组成，但不能用数字开头。所选择的标识符应该反应出它们在程序中的用途。按这种方式，标识符起文档的作用，使得程序更易读。在声明变量后，我们就可以对其赋值和在表达式中使用它们。

一些关键字，也称作保留字(reserved word)，不能被程序员用作变量名。例如，char、int和float等都是这样的关键字。在第2章“词法元素、运算符和C系统”中，我们将给出一个所有关键字的列表。C系统也使用了一些其他名字，程序员通常也不应该对它们进行重定义，名字printf就是一个例子，因为它是标准库中的一个函数名，通常不用作变量名。

通常表达式出现在赋值运算符的右边或作为函数的参数。最简单的表达式是像用在当前程序中的6和12那样的常量。单独的变量名可以被看作表达式，运算符与变量和常量的有意义的组合也是表达式。

C中的很多运算符是二元的数学运算符，+、-、\*、/和%分别用于加、减、乘、除和取模。称它们为二元运算符是因为它们作用在两个操作数上。例如，在表达式a+b中，运算符+

作用在操作数a和b上。像这样的表达式的值依赖于a和b的值。例如，若a的值为1，b的值为2，则a+b的值为3。

在C中，一个整数表达式被另一个整数表达式相除产生一个整数值，任何小数部分被省略。这样1/2的值为0、7/2的值为3、18/4的值为4、29/5的值为5。零作除数是不允许的。若a和b是int变量，其中的一个或两个是负数，则a/b的值是与系统相关的（请参见练习5）。

大多数程序初学者都不熟悉取模运算符%，我们将看到在编程中它有很多用处。表达式a%b在a被b除后产生余数。例如，由于5被3除的商为1余数为2，所以表达式5%3的值为2。以类似的方式，7%4的值为3、12%6的值为0、19%5的值为4、33%7的值为5。在表达式a%b中，b的值不能为0，因为这样会导致零作除数。模数运算符仅能作用于整数表达式，而其他算术运算符能作用于整数和浮点数。如同除法运算符一样，若取模运算符的任意一个操作数是负数，则运算的值是与系统相关的（请参见练习5）。

关键字char代表“字符”。类型为char的变量和常量用于操纵字符。类型为char的常量用单引号括起来，例如，'A'、'1'和'+'。下面是一个简单的例子。

```
#include <stdio.h>

int main(void)
{
    char    c;

    c = 'A';
    printf("%c\n", c);    /* the letter A is printed */
    return 0;
}
```

这个程序输出字母A，后跟一个换行符。首先声明变量c是char类型的，然后c被赋以值'A'，最后printf()语句致使输出发生。注意在printf()的参数表中的控制串含有格式%c，这使得在第二个参数中的变量c以字符的格式被显示。

在ANSI C中，有三种浮点类型：float、double和long double。它们用于操纵实数，也把它们称为浮点（floating或floating-point）数。像1.069、0.003和7.0这样的浮点常量都是double类型的，而不是float类型的。通过说C中的double是工作浮点类型(working floatingtype)来表达这个思想。类型为float的常量通过加一个后缀F来创建，例如1.069F。类似地，类型为long double的常量通过加一个后缀L来创建，例如-7.0L。注意浮点常量7.0和整型常量7是不同的。虽然它们的概念值是一样的，但它们的类型不同，这使得它们在机器中的存储有所不同。关于float和double的技术细节在第6章“基本数据类型”中讨论。

如下的程序中给出了浮点常量和变量用法的简单说明。

```
#include <stdio.h>

int main(void)
{
    float    x, y;

    x = 1.0;
    y = 2.0;
    printf("The sum of x and y is %f.\n", x + y);
    return 0;
}
```

这个程序的输出为

The sum of x and y is 3.000000

首先变量x和y被声明为float类型，然后分别被赋值为1.0和2.0。这些常量为double

类型，变量为float类型，这样不会出现问题。浮点类型在表达式和赋值能自由混合使用。printf()中的第一个参数含有格式%f，这使得第二个参数中的表达式x+y的值按在小数点的后边有6个数字的浮点数格式被显示。

浮点值的除法与人们的想象是一样的。例如，浮点表达式7.0/2.0的值为3.5。作为对照，类型为int的表达式7/2的值为3，这是因为整数除法中的余数被省略了。在浮点表达式中，不允许零做除数，若零做除数也可能产生不是数的值（请参见练习11）。

模数运算符%仅与整数表达式一起使用。如果x和y是类型为float或double的变量，则像x%y这样的表达式是非法的。

通常赋值语句是由等号、等号左边的变量和等号右边的表达式组成。表达式可能简单，也可能复杂，还可能包含函数调用。常量和普通的表达式不允许出现在等号的左边。我们可以写：

```
a = b + c;
```

但不能写：

```
a + b = c; /*assignment to this expression not allowed*/
2 = c;     /*assignment to constant is not allowed*/
```

## 1.4 初始化

在声明变量时，变量也可以被初始化。例如，考虑如下的声明：

```
char   c = 'A';
int    i = 1;
```

变量c被声明为类型char，其值被初始化为'A'。变量i被声明为类型int，其值被初始化为1。作为另一个初始化的例子，可把显示海难深度的程序depth重新编写如下：

```
#include <stdio.h>

int main(void)
{
    int   inches, feet, fathoms = 7;

    feet = 6 * fathoms;
    ....
```

变量是否被初始化依赖于在程序中使用它的目的。通常用常量或常量表达式初始化变量。我们可以写：

```
int   inches, feet, fathoms = 3 + 4;
```

但不能写：

```
int   inches, feet = 6 * fathoms, fathoms = 7;
```

变量fathoms在声明前不能使用，C语言没有超前的能力。在练习6中，我们要指出有意义的用常量表达式进行初始化的情形。

## 1.5 #define和#include的用法

在调用C编译器时，预处理器先工作。在进行编译之前，预处理器修改传递给编译器的源代码。例如，可能要引入文件，源代码中的指定字符串可能被转变成其他字符串。程序中向预处理器发命令的行被称为预处理指令，这样的命令用磅字符#开始。常见的编程风格是把#写在页的左边。

我们曾经使用过如下的预处理器指令：

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

使用这种模式可跨系统使用标准库。这样的代码是可复用的并且易于维护。

用如下形式的预处理指令：

```
#include <filename>
```

预处理器仅在标准的位置寻找文件。在UNIX系统中，像stdio.h、math.h、string.h和stdlib.h这样的头文件通常是在/user/include中。通常头文件的存储位置与系统有关。

#include功能的另一种形式如下：

```
#include "filename"
```

这使得预处理器用给定文件的内容的拷贝替换该行。首先在当前目录中搜索这个文件，然后在其他的与系统相关的位置搜索这个文件。对include文件所包含的内容没有任何限制。特别是它能包含由本预处理器展开的其他预处理指令。

一些#define指令的例子如下：

```
#define LIMIT 100
#define PI 3.14159
```

如果这些预处理指令出现在正在编译的文件的顶部，预处理器首先把出现的所有LIMIT变为100，PI变为3.14159。串常量中的内容保持不变。这样预处理器把printf("PI = %f\n", PI);变为printf("PI = %f\n", 3.14159)。标识符PI在所出现的地方都被变为3.14159，它被称为符号常数。

程序中符号常数的使用使程序更加易读。更重要的是，如果用#define功能把一个常量已经用符号定义，那么就可以在整个程序中使用它，以后对它进行修改也是容易的（如有必要的话）。例如，如果我们写如下语句：

```
#define LIMIT 100
```

那么就可以在数千行的代码中用LIMIT表示常量100，这样以后改变代码也容易。如果要重新定义符号常量LIMIT为1000，我们所做的仅是把预处理指令改为：

```
#define LIMIT 10000
```

这样就自动地修改了所有的源代码；为了修改由本程序产生的可执行文件，我们必须重新编译它。

#define行可出现在程序的任何地方。它仅影响文件中跟在它后边的那些行。通常#define指令放在文件的开始，但位于#include之后。按照惯例，要被预处理器改变的标识符按大写字母书写。

我们要在下节说明在计算面积的程序中符号常量的用法。

## 1.6 printf()和scanf()的用法

函数printf()用于显示格式化的输出。类似地，函数scanf()用于读取格式化的输入。这些函数是在标准库中，在C系统驻留的任何地方都可以使用它们。可以向printf()和scanf()传递能被看作是控制串和其他参数的参数列表。控制串是一个包含一些转换说明的串或格式。转换说明用%字符开始，并用一个转换字符结束。例如，在格式%d中，字母d是一



个转换字符。

### 1.6.1 printf() 的用法

像我们已经看到的那样, 格式`%d`用于按十进制整数显示表达式的值。类似地, `%c`用于按字符显示表达式的值, `%f`用于显示浮点表达式的值, `%s`用于显示串。控制串中的格式用于决定如何显示其他参数, 应该用适合于参数的格式。考虑如下的语句:

```
printf("Get set: %s %d %f %c%c\n", "one", 2, 3.33, 'G', 'O');
```

`printf()` 的参数用逗号分开。在这个例子中有6个参数:

```
"Get set: %s %d %f %c%c\n" "one" 2 3.33 'G' 'O'
```

第一个参数是控制串。控制串中的格式与其他参数相匹配。在本例中, `%s`与"one"对应, `%d`与2对应, `%f`与3.33对应, 第一个`%c`与'G'对应, 第二个`%c`与'O'对应。控制串中的各个格式说明了如何显示对应的参数的值。当执行时, 上述的`printf()`语句产生如下的输出:

```
Get set: one 2 3.330000 GO
```

有时把一个长的`printf()`语句写成几行是适宜的。如下是一个说明我们如何做到这点的例子:

```
printf("%s%s\n",
    "This statement will print ",
    "just one very long line of text on the screen.");
```

下表描述了格式中的转换字符如何影响对应的参数。

printf() 转换字符	
转换字符	如何显示对应的参数
c	作为字符
d	作为十进制整数
e	作为科学记数法的浮点数
f	作为浮点数
g	作为e格式或f格式, 取较短的一个
s	作为串

在显示一个参数时, 所显示的参数位置被称为参数的域 (field), 域中字符的个数被称为域宽 (field width)。格式中域宽能被表示为出现在%和转换字符间的整数。因此, 语句:

```
printf("%c%3c%7c\n", 'A', 'B', 'C');
```

显示的结果为:

```
A      B      C
```

首先显示A, 然后以3个字符的宽度显示B; 因为B仅需要1个位置, 所以其余的2个位置都是空格, 以7个字符的宽度显示C; 因为C仅需要1个位置, 所以其余的6个位置都是空格。

对于浮点值, 我们能控制精度和宽度。精度是显示在小数点右边的十进制数字的个数。在形式为`%m.nf`的格式中, `m`指定了域的宽度, `n`指定了精度。用形式为`%mf`的格式, 仅指定了域的宽度。用形式为`%.nf`的格式, 仅指定了精度。下面的语句说明了这些思想:

```
printf("Some numbers: %.1f %.2f %.3f\n", 1.0, 2.0, 3.0);
printf("More numbers: %7.1f%7.2f%7.3f\n", 4.0, 5.0, 6.0);
```

输出如下:

```
Some numbers: 1.0 2.00 3.000
More numbers: 4.0 5.00 6.000
```

为了说明输出,你必须仔细地数清空格的个数。printf()函数允许程序员在屏幕上整齐地显示,不过用心地去数可能是单调乏味的。

### 1.6.2 scanf()的用法

函数scanf()与函数printf()类似,但它用于输入而不是输出。它的第一个参数是一个带有格式的控制串,该格式和输入流中解释字符的各种方式相符合。在控制串后面的其他参数是地址(address)。一个变量的地址是该变量在内存中存储的位置。(第8章“函数、指针和存储类型”中详细地解释了地址和指针。)符号&表示地址运算符。在例子

```
scanf("%d", &x);
```

中格式%d使得在键盘上输入的字符被解释为十进制整数,并把此十进制整数值存储在x中。

在用键盘把值输入到程序中时,一个字符序列被敲入,并被程序接受。我们称这个序列为输入流(input stream)。如果输入“123”,输入它的人可能把它看作是十进制整数,但程序把它接受为字符序列。scanf()函数能用于把诸如123这样的十进制数字转换成整数值,并把它们存储到适当的地方。

下表描述了函数scanf()所使用的格式中的转换字符的作用。

scanf() 转换字符	
转换字符	如何转换输入流中的字符
c	字符
d	十进制整数
f	浮点数(float)
lf	浮点数(double)
Lf	浮点数(long double)
s	串

**注意** 在printf()中,%f格式用于显示float或double。在scanf()中,%f格式用于读float,%lf用于读double。(在1.10节“常见的编程错误”中你会再次看到这些内容。)

让我们编写一个提示用户输入他的名字缩写和年龄的程序。我们用scanf()函数读入在键盘上敲入的字符,把它们转换成适当的值,并存储在指定的地址中。

```
#include <stdio.h>

int main(void)
{
    char    first, middle, last;
    int     age;

    printf("Input your three initials and your age: ");
    scanf("%c%c%c%d", &first, &middle, &last, &age);
    printf("\nGreetings %c.%c.%c. %s %d.\n",
           first, middle, last,
```

```
    "Next year your age will be", age + 1);
    return 0;
}
```

注意对scanf()传递的参数是:

```
"%c%c%c%d"    &first    &middle    &last    &age
```

第一个参数是控制串。在控制串中的每个格式都与后面的一个参数相对应。很明显, 第一个格式是%c, 它与&first对应, &first是在控制串后的第一个参数; 第二个格式是%c, 它与&middle对应, &middle是在控制串后的第二个参数; 以此类推。在控制串后, 对scanf()传递的所有参数都必须是地址。地址运算符&应用到变量上以产生变量的地址。

假设我们执行前面的程序, 在出现提示时, 输入CBD和19。在屏幕上会出现如下信息:

```
Input your three initials and your age:  CBD  19 Greetings
C.B.D.  Next year your age will be 20.
```

在读入数字时, scanf()跳过空白字符(空格、换行符和跳格符), 但在读入字符时, 不会跳过空白字符。这样在用CB D作为输入时, 程序不能正确地运行。第三个字符被作为空格(它也确实是一个字符)读入, 随后scanf()试图把字符D解释为十进制整数, 这就使得程序出现了错误的行为。

前述的程序是不健壮的。毕竟当用户输入姓名首字母时, 程序应该接受夹杂的空白。在C中使用串变量解决这一问题很容易的, 在第10章“串和指针”中要涉及到这方面的内容。

在下一个程序中, 我们用#define预处理指令定义符号常量, 然后用scanf()从键盘读入值, 并显示在屏幕上。在这个程序中, 我们要特别注意%lf和%f格式。

```
#include <stdio.h>
#define  PI  3.141592653589793

int main(void)
{
    double  radius;

    printf("\n%s\n\n%s",
        "This program computes the area of a circle.",
        "Input the radius: ");
    scanf("%lf", &radius);
    printf("\n%s\n%s%.2f%s%.2f%s%.2f\n\n%s%.5f\n\n",
        "Area = PI * radius * radius",
        "    = ", PI, " * ", radius, " * ", radius,
        "    = ", PI * radius * radius);
    return 0;
}
```

假设我们执行这个程序, 当提示出现时输入2.333, 如下信息会出现在屏幕上。

```
This program computes the area of a circle.
Input the radius:  2.333
Area = PI * radius * radius
      = 3.14 * 2.33 * 2.33
      = 17.09934
```

手工计算表明,  $3.14 \times 2.33 \times 2.33$  等于17.046746, 这与程序显示的结果不符。其原因是PI和radius在显示时仅保留两位小数, 而在内存中它们的值是相当精确的。

要细心地注意到: 我们在控制串中用了个lf%格式, 即以double方式向scanf()读入。如果把变量radius的类型从double变为float, 我们必须把lf%改为f%; 对于printf()的控制串则不需要变化。格式lf%中的lf代表“long float”, 在传统的C中,

类型long float与double是同义的。在ANSI C中,不存在类型long float,尽管有一些实现仍然接受它。

在printf()和scanf()间的另一个不同是它们返回的类型为int的值。调用printf()返回的是显示的字符数,而调用scanf()返回的是已成功地转换的字符数。在1.8节“问题求解:计算总和”中,我们要说明一个关于scanf()返回值的典型应用。虽然程序员很少用到printf()的返回值,但要用也是容易的(请参见练习18)。

关于printf()和scanf()的完整细节以及相关的函数,请参见第13章“输入/输出和文件”。

## 1.7 while语句

正常情况下要一条接一条地执行程序中的语句,这被称为顺序的控制流(sequential flow of control)。为了执行重复的操作,C提供了while语句。

记数、累加、查找、排序和一些其他的工作经常涉及到重复地做一些事情。在本节中,我们将说明怎样用while语句完成重复的操作。同时我们也要反复地提到本章中已经讲过的很多思想。

下边的程序用while语句累加从1到10这些连续的整数。在下边的解析中,我们解释while语句如何工作。

```
#include <stdio.h>

int main(void)
{
    int    i = 1, sum = 0;

    while (i <= 10) {
        sum = sum + i;
        i = i + 1;
    }
    printf("Sum = %d\n", sum);
    return 0;
}
```

### 对程序add\_ten的解析

- int i = 1, sum = 0;

变量i和sum被声明为int类型,并分别被初始化为1和0。

- while (i <= 10){
  - sum = sum + i;
  - i = i + 1;

这个完整的结构是while语句即while循环。首先计算表达式i<=10。对该表达式的一个读法是“i小于或等于10”。由于i的当前值是1,该表达式为真,这使得将执行两个花括号间的语句。把sum的值加上i的值赋给变量sum,因为sum的旧值是0,i的值是1,所以赋给sum的值是1。把i的值加上1赋给变量i,因为i的旧值是1,所以赋给i的值是2。到此我们已经执行了一遍循环。现在程序重新计算表达式i<=10。因为i的值是2,表达式仍然为真,这使得该循环再次被执行。到第二次循环结束时,sum的值是1+2,i的值是3。因为i<=10仍然是真,循环再次被执行。到第三次循环结束时,sum的值是1+2+3,i的值是4。这样的过程持续执行,直到i的值为11,表达式i<=10为假为止。当表达式为假时,程序跳出循环体,执行

while语句后的下一条语句。

```
• printf("Sum = %d\n", sum)
```

该printf()语句显示Sum = 55。

while循环的一般格式是：

```
while (expression)
    statement
```

这里的statement可以是单语句，也可以是由花括号括起来的语句组。花括号括起来的语句组被称为复合语句(compound statement)。在C中，复合语句可以出现在单语句出现的任何地方。

## 1.8 问题求解：计算总和

编程就是在计算机的帮助下解决问题。很多问题需要使用特定的问题解决模式或技术，才能得到解决。在下面的程序中，我们用迭代(iteration)解决我们的问题。迭代是一种重复的行为。计算机最适合于做迭代工作，可快速地完成上千万次的循环。

我们想编写一个重复累加用户从键盘上输入的数字的程序。如下是实现该任务的算法。

**计算总和的算法**

- 1) 对变量cnt和sum初始化。
- 2) 提示用户输入。
- 3) 重复地读数据，对cnt加1，对sum进行累加。
- 4) 显示cnt和sum的值。

while语句是C中提供的用以完成重复操作的三种结构之一。这里我们用scanf()的返回值控制while语句的行为。这允许程序的用户随机地输入数据。在下边的解析中，我们要详细地解释这种机制。

```
/* Sums are computed. */
#include <stdio.h>

int main(void)
{
    int    cnt = 0;
    float  sum = 0.0, x;

    printf("The sum of your numbers will be computed\n\n");
    printf("Input some numbers: ");
    while (scanf("%f", &x) == 1) {
        cnt = cnt + 1;
        sum = sum + x;
    }
    printf("\n%s%5d\n%s%12f\n\n", "Count:", cnt, " Sum:", sum);
    return 0;
}
```

**对程序find\_sum的解析**

```
• scanf("%f", &x) == 1
```

符号==代表相等运算符。诸如a==b这样的表达式用于测试a的值是否等于b的值。如果是，则表达式为真；如果不是，则表达式为假。例如，1==1为真，2==3为假。scanf()函数用于读入用户键入的字符，把字符转换成类型为float的值，并存储到x中。若scanf()

成功地做到这些, 转换成功则该函数的返回值为1。若由于其他的原因, 转换过程失败则返回值为0; 若数据不再可用, 则返回值为-1。

表达式 `scanf("%f", &x) == 1` 用于测试 `scanf()` 是否成功地完成了任务。如果是, 那么表达式为真, 否则为假。

```
• while (scanf("%f", &x) == 1) {
    cnt = cnt + 1;
    sum = sum + x;
}
```

我们可以把该语句看作为:

```
while(scanf() succeeds in making a Conversion){
.....
```

只要表达式 `scanf("%f", &x) == 1` 为真, `while` 循环体就重复地执行。每次执行循环, `scanf()` 读入字符, 把字符转换成数字, 并把数字值存储到 `x` 中。把 `cnt` 的值加1赋给变量 `cnt`, 把 `sum` 的值加上 `x` 赋给变量 `sum`。 `cnt` 保存当前为止键入的数字个数, `sum` 保存数字的总和。什么时候这个过程结束? 这要等到如下的两件事情之一发生。一件事情是, 用户键入了不能被转换成 `float` 类型的事物。例如, 键入了一个不是数字的字符。另一件事情是, `scanf()` 没有成功地转换而返回值为0, 这会引起表达式 `scanf("%f", &x) == 1` 为假。如果用户告诉程序所有的数据都已输入, 该过程也会停止。为了做到这一点, 用户必须键入文件尾标志。在UNIX中, 回车后跟一个 `Ctrl+d` 就是一个典型的文件尾标志。在MS-DOS中, 必须输入回车, 并后跟一个 `Ctrl+z`。

```
• printf("\n%s%d\n%s%12f\n\n", "Count:", cnt, " Sum:", sum);
```

假设执行该程序输入的数字为

```
1.1 2.02 3.003 4.0004 5.00005
```

且后跟一个换行和文件尾标志, 则有如下信息出现在屏幕上:

```
The sum of your numbers will be computed
Input some numbers: 1.1 2.02 3.003 4.0004 5.00005
Count:    5
Sum:    15.123449
```

如果仔细数空格, 你会看到 `cnt` 的值按5个字符的宽度显示, `sum` 的值按12个字符的宽度显示, 这是由格式 `%5d` 和 `%12f` 产生的效果。注意显示出的 `sum` 的数字是错的, 它超出了第3个小数位 (请参见练习17)。 ■

## 1.9 风格

好的编码风格对编程艺术来说是必要的, 这样做易于阅读、编写和维护程序。好的编程风格使用空格和注释以使代码更容易阅读和理解, 并且在视觉上更吸引人。适当地缩进非常重要, 这样能向读者表明设计的控制流。请看如下的结构:

```
while (expression)
    statement
```

其中语句的缩进指明它是在 `while` 循环的控制下执行。另一个风格上的要点是对变量命名, 变量名应该表达出变量在程序中的用途, 以进一步辅助理解程序。好的风格应避免容易出错

的编码习惯。

在本书中，我们遵循贝尔实验室的行业编程风格。在第一列放置所有的#include、#define、main()和表示main()体开始与结尾的花括号。

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

#define GO "Let's get started."

int main(void)
{
    ....
}
```

main()体内的声明和语句要缩进三个空格，这在视觉上突出函数体的开头与结尾。在#include后有一空行，在#define后有一空行，main()体内的声明和语句间也有一个空行。

一般地可以缩进2个、3个、4个、5个或8个空格。我们采用缩进3个空格，无论选择缩进多少空格，在使用时要前后一致。为了增加可阅读性，我们在二元运算符的两边各加了一个空格。一些程序员不喜欢这样，但这是贝尔实验室风格的一部分。

没有惟一的所谓“好的风格”。像我们在本章所做的那样，我们经常指出可选的风格。既然你已经选择了一种风格，你就应该前后一致地遵循它。当心：初级程序员有时认为他们应该想象出一种与众不同的编码风格，应该避免这样做。首选的策略是选择已经被普遍采用的风格。

## 1.10 常见的编程错误

当首次编程时，你会犯很多令人沮丧的简单错误。一个这样的错误是丢失了标识串结束的一个双引号。当编译器遇到第一个双引号时，它开始把随后的所有字符当作是串。如果表示结束的双引号不出现，串会持续到下一行，这会引起编译器报警。不同的编译器给出的出错信息是不同的，下面是一个可能会出现的出错信息：

Unterminated string or character constant

另一个常见的错误是变量名拼写错误，或忘记了声明变量名。编译器能够容易地捕获这种错误，并向你给出适当的信息，即出现了什么错误。然而，如果你错误地拼写了函数名，例如写出了print()，而不是printf()，编译器会告诉你它不能找到该函数。如果你没有注意到用print代替了printf这样的错误信息，你会很迷惑（请参见练习4）。

即使是初级的错误，例如忘记了在语句的后面加分号或丢失了结束的括号，也能引起编译器给出令人迷惑的出错信息。随着你的经验的增加，你会理解编译器产生的一些出错信息。为了用出错信息实验编译器的能力，练习4给出了一些你可能会犯的编程错误。

printf()和scanf()都使用包含一定转换说明或格式的控制串。printf()使用的格式%f用于显示float或double型数据。但scanf()使用的格式%f用于读入float型数据，%lf用于读入double型数据。在使用scanf()读入double型数据而忘记了用%lf是一个常见的错误。大多数编译器都不能捕获这样的错误，因而程序能够运行，但会产生错误的结果。

另一个常见的错误是忘记了在printf()语句中的形式为%m.nf的格式中用m指明字段宽度。例如，为了指明在十进制的小数点的左边有两位数字，右边有三位数字，不要用%2.3f而要用%6.3说明包括小数点在内共有六位数字。

使用scanf()的最常见错误可能是忽略了地址运算符&。如果你写了scanf("%d%d", a, b); 而不是scanf("%d%d", &a, &b);

编译器可能不捕获这个错误,而你很可能会遇到难以调试的运行错误。

## 1.11 系统考虑

在本节中,我们要讨论一些与系统有关的主题。我们从编写和运行C程序的技巧开始讨论。

### 1.11.1 编写和运行C程序

你必须依靠操作系统用文本编辑器和编译器且遵循精确的步骤去创建含C代码的文件,并编译和运行它。不管怎样,在各种情况下大体过程都是相同的。我们首先详述在UNIX环境下该怎样做,然后讨论在MS-DOS环境下该怎样做。

在下面的讨论中,我们用cc命令调用C编译器。然而,实际上这个命令依赖于所使用的编译器。例如,如果我们使用命令行版本的Borland C编译器,就要用bcc而不是cc。请参见14.3节“C编译器”中的C编译器列表。

#### 编写和运行C程序的步骤

1) 用一个编辑器创建一个例如名为pgm.c含有C程序的文本文件。文件名必须用.c结尾,以指明这个文件含有C源代码。例如,用UNIX系统的vi编辑器,我们要给出命令

```
vi pgm.c
```

要使用一个编辑器,程序员必须知道插入和修改文本的适当命令。

2) 编译程序。用命令

```
cc pgm.c
```

做到这一点。cc命令依次调用预处理器、编译器和加载器。预处理器要按照预处理指令修改源代码的拷贝,并产生所谓的翻译单元(translation unit)。编译器把翻译单元翻译成目标代码。如果有错误,程序员必须再从步骤1开始编辑源代码。把发生在该阶段的错误称为语法错误(syntax error)或编译错误(compile error)。如果没有错误,加载器使用编译器产生的目标代码以及由系统提供的各种库中的源代码,以创建可执行的文件a.out。

3) 执行程序。用命令

```
a.out
```

做到这一点。通常程序完成执行,系统提示重新出现在屏幕上。把在执行期间发生的各种错误称为运行错误(run time error)。由于一些原因,如果需要改变程序,程序员必须再从步骤1开始。

如果我们要编译不同的程序,文件a.out要被重写并且它原有的内容也要丢失。如果已保存了可执行文件a.out,要对该文件进行移动或重命名。假设发出命令

```
cc hello.c
```

结果会把可执行代码自动地写进a.out。为了保存这个文件,可以使用命令

```
mv a.out hello
```



这会把a.out移到hello中。现在可用命令

```
hello
```

执行这个程序。在UNIX中，给可执行文件和对应的源代码文件以相同的名字是一种常见的做法，只是可执行文件没有后缀.c。如果愿意，我们也可以用-o选项对cc命令的输出定向。例如，命令

```
cc -o hello hello.c
```

会使得cc的可执行输出直接写进hello，而不会修改a.out。

在程序中可能会发生不同种类的错误。编译器捕获语法错误，而运行错误仅在程序执行时出现。例如，如果试图在程序中用零做除数，当程序执行时会发生运行错误（请参见练习10和11）。运行错误提供的信息对发现问题没有多大帮助。

现在考虑MS-DOS环境。这里可能用到一些其他编辑器。一些诸如Borland C这样的C程序既有命令行环境又有集成环境。集成环境包含文本编辑器和编译器。在MS-DOS中，由C编译器产生的可执行的输出通常被写入与源文件同名的文件，但用扩展名.exe代替.c。例如，假设我们用Borland C的命令行环境。如果我们发出命令**bcc hello.c**，那么可执行的代码被写入**hello.exe**。为了执行这个程序，我们发出命令**hello.exe**或等价的命令**hello**，调用这个可执行程序，可不写扩展名.exe。如果需要对这个文件重命名，我们可以使用**rename**（重命名）命令。

### 1.11.2 中断程序

用户可能想中断或取消正在运行的程序。例如，程序或许处于死循环中。（在交互式的环境中，在程序中使用死循环并非都是错误。）在本文中，我们假设用户知道怎样中断程序。在MS-DOS和UNIX中，通常用Ctrl+c中断程序。在一些系统中使用诸如删除(delete)或清除(rubout)这样的特殊键。你务必要知道如何中断你的系统中的程序。

### 1.11.3 输入文件尾标识

在程序从键盘接收输入时，为了程序工作正常可能有必要输入文件尾标识。在UNIX中，通过输入一个回车后跟一个Ctrl+d来实现。在MS-DOS中，则通过输入一个回车后跟一个Ctrl+z来实现（请参见练习19）。

### 1.11.4 输入和输出的重定向

包括MS-DOS和UNIX在内的很多操作系统都能对输入和输出进行重定向。为了理解此项工作，我们首先考虑UNIX命令ls。这个命令产生文件列表，并在屏幕上列目录（与MS-DOS的dir命令相似）。现在考虑命令ls>temp。符号>使得操作系统把该命令的输出重定向到文件temp，（在MS-DOS中，文件名需要一个扩展名。）以前写到屏幕的内容现在写到了文件temp。

下一个程序称为dbl\_out，它使用了输入和输出的重定向。该程序从标准输入文件（通常被连接到键盘）读入字符，并且两次把字符写入标准的输出文件（通常连接到屏幕）。

```
#include <stdio.h>

int main(void)
{
    char c;
```

```

    while (scanf("%c", &c) == 1) {
        printf("%c", c);
        printf("%c", c);
    }
    return 0;
}

```

如果我们编译这个程序，并把可执行的代码放进文件`dbl_out`，那么使用重定向我们能用于下述四种方式之一调用这个程序。

```

dbl_out
dbl_out < infile
dbl_out > outfile
dbl_out < infile > outfile

```

在这个上下文中，可以把符号`<`和`>`看作是箭头（请参见练习19）。

一些命令并不用于重定向。例如，`ls`命令不从键盘读入字符。说把输入重定向到`ls`命令是讲不通的，因为它没有从键盘取得输入，所以没有什么要重定向。

## 1.12 转向C++

大多数C程序在C++编译器中运行没有什么变化。因而，通过学习C，你就是已经在学习C++了。本节介绍C++风格的I/O。`stdio.h`用于C++程序界中，C++ I/O库是`iostream.h`。

```

/* hello program in C++, using iostream IO.
   Note the use of endl to create a newline.
*/

#include <iostream.h>
int main(void) {
    cout << "Hello, world!" << endl;
    return 0;          //This is optional
}

```

标识符`cout`代表屏幕。插入运算符`<<`用于把字符串`"Hello, world!"`放进输出流。I/O运算符`endl`用于刷新输出并移到新行。在C++程序中，可以省略`return 0`，因为它在程序中是被隐含插入的。

下述的程序计算两个整数的最大公约数。整数要从键盘上输入。

```

/* Greatest common divisor program.*/

#include <iostream.h>

int main(void)
{
    int m, n, r;

    cout << "\nPROGRAM Gcd C++";
    cout << "\nEnter two integers: ";
    cin >> m >> n;
    cout << "\nGCD(" << m << ", " << n << ") = ";
    while (n != 0) {
        r = m % n;
        m = n;
        n = r;
    }
    cout << m << endl;
}

```

标识符`cin`通常是与键盘输入相关的标准输入流。第一个输入的值被转换为整数值，并

被放在变量m中，第二个输入的值被放在变量n中。注意这个输入表达式是直观的，与对应的scanf()的用法相比更简单。这种用法的一个关键特征是不需要格式。用iostream.h进行输入和输出在类型上是安全的。

## 小结

- 算法是由一些基本步骤组成的计算过程。编程是对计算机交流算法的艺术。
- 简单程序由预处理指令和函数main()组成。函数体由写在花括号{ }之间的声明和语句组成。所有变量都必须声明。声明必须出现在语句之前。
- 最简单的表达式仅由常量、变量或函数调用组成。通常表达式由运算符和其他表达式组成。大多数表达式都有值。赋值运算符=用于把表达式的值赋给变量。
- 在声明变量时，也可以对变量进行初始化。通常把常量或常量表达式作为初始化值。
- 在编译源代码时，预处理器先工作。把用符号#开始的行称为预处理指令。程序员用预处理指令向预处理器发命令。通常要把#include和#define指令放在文件的头部。#define指令仅影响文件中出现在它后面的那些行。
- 预处理指令的形式为：

```
#define identifier replacement_string
```

它会在编译发生之前使得预处理器把出现的每个这样的identifier变为replacement\_string。

- 在传统的C中，标志着预处理指令开始的#必须出现在第一列。在ANSI C中，可以把空格或制表符放在#之前。
- 在标准库中的printf()函数用于输出。该函数的参数由后跟有其他参数的控制串组成。控制串由夹杂着转换说明或格式的一般文字组成。一般文字被简单显示，而格式会使相关参数的值按格式中的转换说明显示。格式由一个%开始，并由一个转换字符结束。
- 在标准库中的scanf()函数用于输入。它与printf()类似，这两个函数的参数个数是不定的，第一个参数都是控制串。对于printf()，其余的参数都是表达式；对于scanf()，其余的参数都是地址。表达式 &v 把v的地址作为它的值。
- 一般按顺序执行语句。像while语句这样的特殊控制流语句能改变程序的顺序执行。

## 练习

1. 以下述三种方式，在屏幕上写一句话

```
she sells sea shells by the seashore
```

(a) 写在一行，(b) 写在7行，(c) 写在一个框中。

2. 下边是一个由用户输入3个整数开始的程序片段：

```
#include <stdio.h>

int main(void)
{
    int    a, b, c, sum;

    printf("Input three integers: ");
    .....
```

请完成这个程序。在执行它时，输入2、3和7后，屏幕上应出现如下内容：

```
Input three integers: 2 3 7
Twice the sum of your integers plus 7 is 31 --- bye!
```

3. 下述的程序在屏幕上输出一个大字符I:

```
#include <stdio.h>

#define HEIGHT 17

int main(void)
{
    int i = 0;

    printf("\n\nIIIIIII\n");
    while (i < HEIGHT) {
        printf(" III\n");
        i = i + 1;
    }
    printf("IIIIIII\n\n\n");
    return 0;
}
```

执行这个程序,理解它的作用。编写一个在屏幕上显示一个大字符C的类似程序。

4. 本练习将帮助你熟悉一些由编译器产生的错误信息。你可以期待一些错误信息是有用的,而另一些则是无用的。首先核对,下面的程序经编译后没有错误信息:

```
#include <stdio.h>

int main(void)
{
    int a = 1, b = 2, c = 3;

    printf("Some output: %d %d %d\n", a, b, c, c)
    return 0;
}
```

现在依次地引入下述的程序错误、编译程序、记录产生的错误信息:

把声明中的第一个逗号改为分号。

把printf()改为print()。

去掉控制串中的第二个引号。

用a,b,c替换a,b, c,c。

去掉printf()语句后面的分号。

去掉用于结束的花括号。

5. 在本练习中,我们将研究一下运算符/和%怎样和负整数一起运算。如果类型为int的a和b有一个为负数,则a/b的值依赖于系统。例如,在一些机器上7/-2的值是-3,而在另一些机器上为-4。a%b的正负号也依赖于系统,但是在ANSI C系统中,能够保证(a/b) \* b + a % b的值为a。请看一下在你的系统中情况如何。编写一个含有下列代码的交互式程序:

```
int a, b;

printf("Input two nonzero integers: ");
scanf("%d%d", &a, &b);
printf("%s%4d\n%s%4d\n%s%4d\n%s%4d\n%s%4d\n",
    "a =", a,
    "b =", b,
    "a / b =", a / b,
    "a % b =", a % b,
    "ANSI check =", (a / b) * b + a % b - a);
```

6. 编写一个要求用户输入一个矩形场地的长和宽的交互式程序。长和宽的单位是码。程序应该计算出场地的平方码数、平方英尺数和平方英寸数(如果你愿意,还可计算出平方米数)在屏幕上整齐地输出所有结果。请使用如下的声明:

```
int cv_factor = 36 * 36; /*conversion: sq in per yrd */
```

一个等价的声明是：

```
int cv_factor = 1296; /* conversion: sq in per yrd */
```

由于大多数人都知道1码等于36英寸，这样第一个声明是较好的，人们能一眼便知所用的转换因子。警告：如果场地很大，而你的计算机很小，你就可能得不到正确的平方英寸数，即使你编写的程序很正确。所能存储的类型为int的整数的大小是有限制的（请参见第6章“基本数据类型”）。

7. 如下是计算一些币值的交互式程序的一部分。该程序要求用户输入半美元（50美分）、四分之一美元（25美分）和一角硬币（10美分）镍币（5美分）的数目。

```
#include <stdio.h>

int main(void)
{
    int    h,      /* number of half dollars */
          q,      /* number of quarters */
          d,      /* number of dimes */
          n,      /* number of nickels */
          p;      /* number of pennies */
    .....

    printf("Value of your change will be computed.\n\n");
    printf("How many half dollars do you have? ");
    scanf("%d", &h);
    printf("How many quarters do you have? ");
    scanf("%d", &q);
    .....
}
```

完成这个程序，让它输出相关的信息。例如，可以产生这样的输出：

```
You entered:    0 half dollars
                3 quarters
                2 dimes
               17 nickels
                 1 pennies
The value of your 23 coins is equivalent to 181 pennies.
```

注意pennies是复数，而不是应该是的单数。在你学会了第3章“控制流”中的if-else语句后，你就能修改这个程序，以使输出符合语法要求。

8. 修改在上一个练习中编写的程序，最后的一个输出行应为：

```
The value of your 23 coins is $1.81
```

提示：把value声明成类型为float的变量，在printf()语句中使用格式%.2f。

9. 函数scanf()的返回值是类型为int的已成功转换的数目。考虑如下语句：

```
printf("%d\n", scanf("%d%d%d", &a, &b, &c));
```

在执行这个语句时，会显示一个整数。这个整数值应该是多少？提示：编写一个测试程序，用重定向执行它。如果你不用重定向，输入和输出可能在屏幕上相混杂，使你分不清。当scanf()接收到文件结尾标志后，返回值是-1。

10. 本练习用于发现运行错误发生时系统会出现什么情况。试一下如下的代码：

```
int    a = 1, b = 0;
printf("Division by zero: %d\n", a / b);
```

在UNIX系统中，会得到信息转储（core dump），也就是说，系统可能创建一个名为core的文件，它含有程序在非正常结束前的状态信息，该文件是不可读的。调试器能用信息转储

告诉你关于程序在异常结束时所做的事情方面的信息。(不要丢下信息转储不管,因为它们相当大,会耗尽宝贵的硬盘空间。)

11. 在一些系统中,用浮点零做除数不会导致运行错误,而在另一些系统中则不然。试一下如下的代码:

```
double x = 1.0, y = -1.0, z = 0.0;
printf("Division by zero: %f %f\n", x / z, y / z);
```

在你的系统会发生什么情况? 如果显示出了Inf或NaN,你可以把这个值看作是“无穷大”或“不是数”。

12. 除了printf()外,下面的程序是完整的。它用整数除和取模运算符把秒转换成分和秒。

```
/* Convert seconds to minutes and seconds. */
#include <stdio.h>

int main(void)
{
    int input_value, minutes, seconds;

    printf("Input the number of seconds: ");
    scanf("%d", &input_value);
    minutes = input_value / 60;
    seconds = input_value % 60;
    printf("...");
    return 0;
}
```

通过编写一个适当的printf()语句完成这个程序。例如,如果在提示后输入123,程序应该显示出:

```
123 seconds is equivalent to 2 minutes and 3 seconds
```

13. 修改在上一个练习中完成的程序,把秒转换成小时、分和秒。例如,如果在提示后输入7348,程序应该显示出:

```
7348 seconds is equivalent to 2 hours, 3 minutes and 4 seconds
```

14. 重复行为对大多数程序来说是必要的,因此,程序员必须精确地掌握while循环是怎样工作的。详细地研究如下代码,写出显示结果。然后编写一个测试程序,检查你的答案。

```
int i = 1, sum = 0;

while (i < 10) {
    sum = sum + i;
    i = i + 1;
    printf("sum = %d i = %d\n", sum, i);
}
```

15. 对上一个练习中编写的程序做两次改变,分别查看运行结果。首先用

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

替换

```
sum = sum + i;
```

然后用

```
sum = (sum / 3) + (i * i);
```

替换

```
sum = sum + i;
```

16. 在你的系统中，如何输入文件结尾标志？用程序find\_sum进行实验，看一下当输入一个不适当的字符或文件结尾标识时程序的终止。当执行程序时，不输入数字会发生什么？

17. 与整数运算不同，浮点数运算不必是精确的。用浮点数计算会发生非常小的误差。此外，误差是与系统相关的，通常与用户无关。使用程序find\_sum的输入数据，计算总和的第6个小数点的位置上就有一个误差。修改这个程序，把变量sum的类型float改为double。由于通常double表示的实数比float更精确，所以使用同样的输入得到的结果会更精确。检查一下你的计算机是否如此。

18. 在ANSI C中，printf()函数返回类型为int的已显示的字符数。为了了解这点，编写一个含有如下代码的小程序：

```
int cnt;
cnt = printf("abc\n");
printf("%d\n", cnt);
```

显示出的整数是什么？用如下的串代替"abc\n"：

```
"\tMontana!\n\n\tIt really is big sky country!\n\n"
```

现在显示出的整数是什么？写出你的答案，然后运行你的程序进行验证。提示：不要忘记换行和制表符也被计数。

19. 如同很多新思想一样，通过实验能最好地理解重定向。编写一个我们在1.11.4节“输入和输出的重定向”中提到的程序dbl\_out，它在名为dbl\_out.c的文件中。编译、执行这个程序，在理解它的作用后，试一试如下的命令：

```
dbl_out < dbl_out.c
dbl_out < dbl_out.c > temp
```

下一个命令更有趣：

```
dbl_out > temp
```

该命令使得在键盘上输入的字符被写入文件temp中，当完成时，你要输入文件结尾标识。如果用取消程序的Ctrl+c代替输入的文件结尾标识会发生什么？

20. 本练习将使用上一个练习中的程序dbl\_out。首先发出命令

```
dbl_out
```

然后输入abc，后跟一个回车。在屏幕上显示的内容依赖于你的操作系统配置如何。正常地，在处理字符前操作系统要等待，直到输入一个完整行。如果是这种情况，你能在屏幕的下一行看到显示的aabbcc。如果使用UNIX，给出命令

```
stty cbreak
```

此时，操作系统随着输入读入各个字符。再次试用命令dbl\_out，输入abc，后跟一个回车。在屏幕上会出现什么？提示：在键盘上输入的字符在正常的情况下会在屏幕上回应。你可能想进一步实验，发出命令

```
stty -echo
```

这会关闭回应。当你完成这个实验后，你应该发出命令

```
stty -cbreak echo
```

这会使操作系统返回到正常状态。警告：如果你把回应关掉，就不会看到你正在做什么。

21. C++：用cout对1.3节“变量、表达式和赋值”中的程序Hesperus重编码。

22. C++: 编写一个通用程序, 取英寸作为输入的深度单位, 并把它转变成英尺和英寸。
23. C++: 改进上一个程序。编写一个循环, 连续地接受各种输入, 直到输入一个负数为止。
24. C++: 编写一个说明C++ I/O的具有输入保护功能的程序。对输入和输出类型为int和double的数值的程序进行编码。看当输入像2.99这样的值给类型为int的变量和类型为double的变量时会发生什么情况。输出这两个变量, 看显示出什么?



## 第2章 词法元素、运算符和C系统

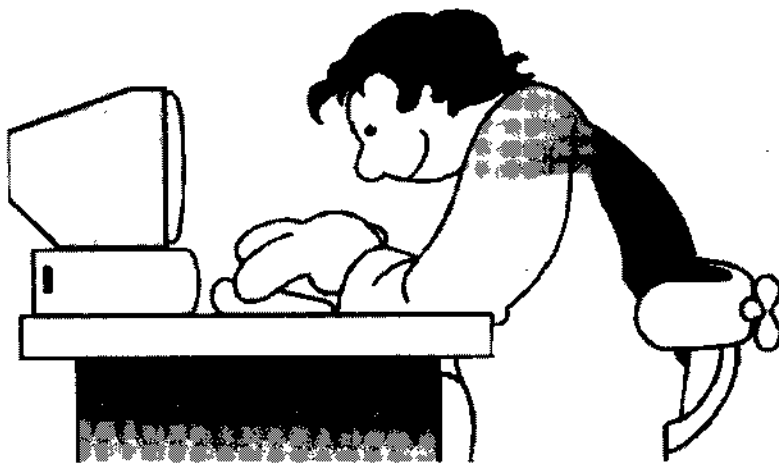
在本章中，我们要介绍C编程语言的词法元素。C是一种语言，像其他语言一样，它有字母表和把词以及标点符号组合在一起的规则，以保证语言正确、合法或符合要求。这些规则是语言的语法。把检查C代码合法性的程序称为编译器。如果有错误，编译器会给出错误信息并停止。如果没有错误，源代码就是合法的，编译器把它翻译成目标代码，然后加载器用目标代码产生可执行文件。

在调用编译器时，预处理器先工作，由于这个原因，我们可以认为预处理器已被嵌入到了编译器中。在一些系统中确实如此，但在一些系统中二者是分离的。本章不关心这些，然而我们必须要知道，我们从预处理器和编译器中都会得到错误信息（请参见练习24）。在本章中，我们在概念上认为预处理器已被嵌入到了编译器中。

C程序是一个字符系列，C编译器要把这些字符转换成目标代码，目标代码又被转换成特定机器上的目标语言。在大多数系统中，目标语言是一种能被运行或解释的机器语言。由于这个缘故，程序在语法上必须是正确的。编译器首先把程序的字符聚集成能作为语言基本词汇的标记(token)。

在ANSI C中，有六种标记：关键字、标识符、常量、串常量、运算符和标点符号。编译器按照语言的语法检查这些标记是否构成合法的串。大多数编译器的要求是非常精确的。英语的阅读者对于带有额外的标点符号或错误单词的句子也能够理解，编译器与此不同，C编译器不能对有语法错误的程序进行转换，无论错误是多么的小。因此，程序员必须学会精确地编码。

程序员应该争取编写其他程序员能够理解的代码。做到这一点的关键是用有意义的标识符名产生带有良好注释的代码。在本章中，我们将说明这些概念。



“计算机不能直接处理我的C程序，我必须先编译我的程序。  
编译先从C代码创建标记，然后把产生的标记转化成能在  
我的处理器上运行的目标代码。”

## 2.1 字符和词法元素

首先程序员要把一个字符序列构造成C程序，这些字符出现在下表中：

用在程序中的字符	
小写字母	a b c . . . . . z
大写字母	A B C . . . . . Z
数字	0 1 2 3 4 5 6 7 8 9 .
其他字符	+ - * / = ( ) { } [ ] < > ' " ! @ # \$ % & _   ^ ~ \ . , ; : ?
空白字符	空格, 换行符, 跳格符等

编译器把这些字符聚集成称之为标记的语法单元。在进行严格的语法定义之前，让我们来看一个简单的程序，先非正式地学习一些程序中的标记。

```
/* Read in two scores and print their sum. */
#include <stdio.h>

int main(void)
{
    int    score_1, score_2, sum;

    printf("Input two scores as integers: ");
    scanf("%d%d", &score_1, &score_2);
    sum = score_1 + score_2;
    printf("%d + %d = %d\n", score_1, score_2, sum);
    return 0;
}
```

### 对程序sum的语法解析

• /\* Read in two scores and print their sum. \*/

注释由/\*和\*/定界。首先编译器用一个空格替换每一个注释，随后编译器丢弃空白或用它作为分隔标记。

• #include <stdio.h>

这是一个预处理指令，它引入标准头文件stdio.h。我们引入它是因为它含有函数原型printf()和scanf()。函数原型是一种声明。编译器需要函数原型完成它的工作。

```
• int main(void)
{
    int    score_1, score_2, sum;
```

编译器把这些字符聚集成4种标记。函数main()是一个标识符，紧跟着main的括号()是运算符，这些告诉编译器main是一个函数。左花括号、逗号以及分号是标点符号；int是关键字；score\_1、score\_2和sum是标识符。

编译器用int和score\_1间的空格区别这两个标记。我们不能写如下的语句：

```
intscore_1, score_2, sum; /*错误：此处需要空一个空格*/
```

另一方面，逗号后的空格是多余的。我们可以写这样的语句：

```
int score_1, score_2, sum;
```

但不能写如下的语句：

```
int score_1score_2sum;
```

这样编译器会把score\_1score\_2sum看作是一个标识符。

```
• printf("Input two scores as integers: ");
  scanf("%d%d", &score_1, &score_2);
```

名字printf和scanf都是标识符，后跟的圆括号告诉编译器它们是函数。在编译器翻译了C代码后，加载器试图创建一个可执行文件。如果程序员没有为printf和scanf提供代码，就从标准库中提取它们。通常程序员不对这些标识符进行重定义。用双引号括起来的字符序列是串常量，例如，"Input two scores as integers: "。

编译器把串常量处理成单标记，并在内存中存储它。在&score\_1和&score\_2中的字符&是地址运算符，编译器把地址运算符处理成一个标记，即使运算符&和a是邻接的，编译器也把它们看作是分离的标记。我们可以写& score\_1, & score\_2或&score\_1, & score\_2，但不能写

```
&score_1 &score_2    /* the comma is missing */
score_1&, &score_2   /* & requires operand on the right */
```

逗号是一个标点。

```
• sum = score_1 + score_2;
```

字符=和+是运算符。这里的空格被忽略，因此我们可以写

```
sum=score_1+score_2;
```

或

```
sum = score_1 + score_2 ;
```

但不能写

```
s u m = score_1 + score_2;
```

如果这样做，编译器会把这行中的每一个字符处理为单独的标识符。由于这样的标识符并没有被声明，因而编译器工作会不正常；即使这样的标识符已经被声明，表达式su也是不合法的。 ■

编译器会忽略空格或把空格用作语言的分隔元素。程序员用空格编写更合法的代码。对于编译器，程序文本是一个语义含糊的单字符流，但对于阅读者来讲，它是一个二维表。

## 2.2 注释

注释是一个放于分界符/\*和\*/之间的任意符号串。注释不是标记。编译器把每个注释转换成一个空格字符，因而注释不是可执行程序的一部分。我们已经看到了如下这样的注释：

```
/* a comment */    /** another comment ***/    /*****/
```

另一个例子是：

```
/*
 * A comment can be written in this fashion
 * to set it off from the surrounding code.
 */
```

如下写法说明了一种着重描述注释的风格：

```
/******
 * If you wish, you can      *
 * put comments in a box.    *
 *****/
```

程序员用注释作为辅助文档，其目的是解释清楚程序怎样工作以及怎样使用程序。有时文档含有说明程序正确性的非正式论据。

注释与程序文本同时书写。一些程序员在最后一步才插入注释，但这样做存在两个问题。第一个问题是，一旦程序能运行了，程序员就会忽略或省略了注释。另一个问题是，理想情况下注释应该起注解作用，指明程序的结构，使得程序更加清楚和正确；如果在完成编码后才插入注释，就起不到这样的作用。

## 2.3 关键字

在C中作为个体标记，关键字是保留字，它有严格的含义。不能对关键字进行重定义，在其他上下文中也不能使用关键字。

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

一些C中可以有附加的关键字。在不同的C中，关键字是不同的。

作为一个例子，下面是Borland C中的一些附加关键字。

Borland C的附加关键字						
asm	cdecl	far	huge	interrupt	near	pascal

和其他的一些主要的语言相比，C仅有少量的关键字。例如，Ada有63个关键字。用相对少的特定符号和关键字做大量的事是C的一个特征。

## 2.4 标识符

标识符是由字符、数字和被称为下划线的特殊符号“\_”组成的标记。标识符的首字符必须是字母或下划线。在大多数C语言中，小写与大写字母是有区别的。选择有助于记忆的且具有一定含义的标识符是一种良好的编程习惯，这样可增强程序的可读性和程序的文档性。如下是一些标识符的例子：

```
k
_id
iamanidentifier2
so_am_i
```

但以下的例子不是标识符：

```
not#me      /* special character # not allowed */
101_south   /* must not start with a digit */
-plus       /* do not mistake - for _ */
```

创建标识符是为了对程序中的对象进行惟一地命名。可以把关键字看作是C语言中的具有特殊含义的保留标识符。我们已经知道像scanf和printf这样的标识符在C系统中已作为标准

库中的输入/输出函数。标识符main是专用的，在C中程序总是从称为main的函数处开始执行。

在一些操作系统和C编译器中的一个主要的不同是辨别标识符的长度。在一些较老的系统中，可以接受长度超过8个字符的标识符，但只使用前8个字符，而把其余的字符丢弃。例如，在这样的系统中，变量名i\_am\_an\_identifier和i\_am\_an\_elephant被认为是一样的。

在ANSI C中，至少可以辨别标识符的前31个字符。很多C系统可以辨别出更多的字符（请参见2.15节“系统考虑”）。

良好的编程风格要求程序员选择有意义的名字。如果你要编程计算税金，就可以使用像tax\_rate、price和tax这样的名字，这样语句

```
tax = price * tax_rate;
```

的含义就会显而易见。下划线用于制作单标识符，这样的标识符用于表示通常是由空格分隔的词串。遵循良好的编程风格是有意义的，但要防止把编程变成构造要点指南。

**警告** 用下划线开始的标识符可能和系统的名字相冲突，应该只有系统程序员才能使用这样的名字。作为一个例子，考虑标识符\_iob，经常把它定义为stdio.h中的结构数组的名字。如果程序员试图把它用于其他目的，编译器的工作就可能不正常或程序可能会出现错误的行为。这里忠告程序员最好不要用下划线开头的标识符。此外，在一个标识符的任何位置上使用两个连续的下划线的方式只限于系统使用。

## 2.5 常量

像我们在一些介绍性的简单程序中曾看到的那样，C能操纵各种值。像整数0和17，都是整形常量，像1.0和3.14159这样的小数都是浮点型常量。像大多数语言一样，C对整型和浮点型常量的处理是不同的。在第6章“基本数据类型”中，我们要详细地讨论C怎样理解数字。也有像'a'、'b'和'+'这样的字符型常量。要把字符型常量写在单引号之间，我们将在第6章“基本数据类型”中看到，它们与整型是密切相关的。像写为'\n'的回车符这样的一些字符型常量是特殊的。反斜线符号是转义符，我们把\n看作是“对通常的n的含意的转义”。即使把\n写为字符\和字符n，它仍表示称为换行符的单字符。

我们已经讨论了一些常量，C中的常量个数是有数的。在第7章“枚举类型和typedef”中，我们要把它们与关键字enum一起讨论。编译器把整型常量、浮点型常量、字符型常量和枚举型常量都看作是标记。由于实现上的限制，可在语法上表达的常量在特殊的机器上可能是不可用的。例如，一个整数可能过大，以致于不能存储在一个机器字中。

十进制整数是十进制数字的有穷串。由于C提供了八进制、十六进制和十进制整数，我们必须仔细地地区别这些不同种类的整数。例如，17是十进制整型常量，017是八进制整型常量，0x17是十六进制整型常量（关于进一步的讨论，请参见第6章“基本数据类型”）。也把像-33这样的负的常量整数看作是常量表达式。如下是一些十进制常量整数的例子：

```
0
77
123456789000      /* too large for the machine? */
```

但以下的例子不是十进制常量整数：

```
0123      /* an octal integer */
-49       /* a constant expression */
123.0     /* a floating constant */
```

虽然我们已经使用了像144这样的整型常量和像39.7这样的浮点型常量，但按照类型、对内存的需求和机器的精度来看，它们的含义过于复杂，需要全面的讨论。在第6章“基本数据类型”中，我们将做此项工作。

## 2.6 串常量

像"abc"这样的在双引号内的字符序列是串常量或串文字。编译器把它认做为单标记。在第10章“串和指针”中，我们会看到编译器把串常量存储为字符数组。串常量和字符型常量总是不同的。例如，"a"和'a'不是一回事。

注意双引号“仅是一个字符，而不是两个字符。如果字符”出现在串常量中，它必须紧跟在转义字符\后。如果字符\出现在串常量中，它也必须紧跟在转义字符\后。如下是串常量的一些例子：

```
"a string of text"
"" /* the null string */
" " /* a string of blank characters */
" a = b + c; " /* nothing is executed */
" /* this is not a comment */ "
"a string with double quotes \" within"
"a single backslash \\ is in this string"
```

但以下的例子不是串常量：

```
/* "this is not a string" */
"and
neither is this"
```

当一个有一定含义的字符序列用双引号括起来时，如果它位于串常量之内，则它仅是一个字符序列。在上述的例子中，一个字符串含有语句a = b + c;，但由于它出现在双引号之内，所以它显然是一个字符序列。

编译器把用空格分隔的两个串常量连接成一个单串。例如，"abc" "def"等价于"abcdef"。

编译器把串常量处理成标记。与其他常量一样，编译器在内存中存储串常量。在第10章“串和指针”中，我们还要强调这点。

## 2.7 运算符和标点符号

在C中，有很多具有特定含义的字符。如+、-、\*和/这样的算术运算符分别代表加、减、乘和除。回想一下，在数学中a对b取模的结果是a被b除后的余数。例如，5%3的值是2，7%2的值是1（关于对负操作数的取模运算请参见第1章“编写ANSI C程序”和练习5）。在程序中，用运算符分隔标识符。虽然通常我们把空格放在二元运算符的前后以强调可阅读性，但不强求这样做。

```
a+b /* this is the expression a plus b */
a_b /* this is a 3-character identifier */
```

一些符号的含义依赖于上下文。例如，考虑下边两个语句中的%号：

```
printf("%d", a); 和 a = b % 7;
```

第一个%是转换说明或格式的开始符，而第二个%代表取模运算符。

像圆括号、大括号、逗号和分号都是标点符号。考虑如下代码：

```
int main(void)
{
```

```
int a, b = 2, c = 3;
a = 17 * (b + c);
.....
```

把紧跟在main后的圆括号看作是运算符，圆括号告诉编译器main是一个函数的名字。随后的{, ; ( ) 符号都是标点符号。编译器把运算符和标点符号看作是标记，和空格一样，它们用来分隔语言元素。

一些特殊的字符可用在多个上下文中，上下文本身能决定特殊字符的用意。例如，有时圆括号由于指示函数名，有时用作标点符号。如下的表达式是这样的例子：

```
a + b      ++a      a += b
```

这些表达式都把+号作为字符，但是++和+=都是单操作符。依赖于上下文符号具有不同的含义有利于小符号集和简明语言。

## 2.8 运算符的优先级和结合性

运算符有精确地决定怎样计算表达式的优先级规则和结合性规则。由于要先计算圆括号里面的表达式，所以可用圆括号表明或改变执行运算的顺序。考虑如下的表达式：

```
1 + 2 * 3
```

在C中，运算符\*的优先级高于+，这使得先运算乘法，然后再运算加法。因此，该表达式的值是7。如下是一个等价的表达式：

```
1 + (2 * 3)
```

换句话说，由于先计算圆括号里的表达式，所以说它与下面的表达式是不同的：

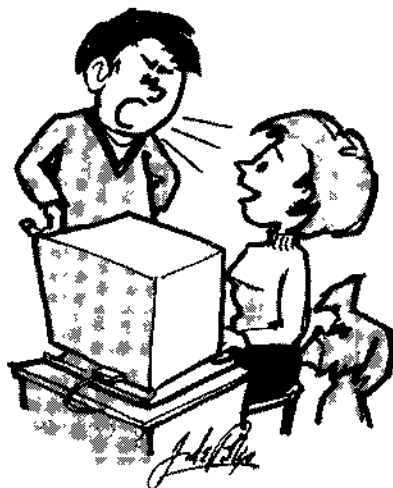
```
(1 + 2) * 3
```

该表达式的值是9。现在考虑表达式：

```
1 + 2 - 3 + 4 - 5
```

它与表达式(((1 + 2) - 3) + 4) - 5是等价的。

由于二元运算符+和-有相同的优先级，所以用从左到右的结合性规则决定怎样计算该表达式，这意味着按从左到右的次序完成该运算。这样，上述的两个表达式是等价的。



“阿什利，我不管手册说什么，我想有一些运算符是不相关的。”

下表给出了C中的一些运算符的优先级规则和结合性规则。表中的一些运算符我们见过，至于其他运算符随后要在本章中讨论。

运算符的优先级和结合性					结 合 性
运 算 符					
()	++ (后缀)	-- (后缀)			从左到右
+ (一元)	- (一元)	++ (前缀)	-- (前缀)		从右到左
	*	/	%		从左到右
	+	-			从左到右
=	+=	-=	*=	/=	等
					从右到左

在一个给定行上的所有像\*、/和%这样的运算符，就彼此之间而论，优先级相等，但越早出现的优先级就越高。在一个给定行上的所有运算符的结合性规则见表右侧。

无论我们什么时候引入新的运算符，我们都先给出优先级规则和结合性规则，程序员应该掌握这些基本规则。

除了二元加外，还有一元加，对于减号也是如此。ANSI C引入了一元加；传统C中没有一元加，只有一元减。

从优先级表中可以看出，一元运算符比二元加和减有较高的优先次序。在表达式  $-a * b - c$  中，第一个减号是一元的，第二个减号是二元的。使用优先次序规则，我们可以看出  $-a * b - c$  与  $((-a) * b) - c$  是等价的。

## 2.9 增量运算符和减量运算符

增量运算符++和减量运算符--是与一元加+和一元减-具有同样优先级的一元运算符，它们从左到右地关联。++和--用于变量，不能用于常量或普通表达式。此外，这两个运算符能出现在前缀位置和后缀位置，但结果不同。这样的例子有：

```
++i
cn--
```

但下述的例子是错误的：

```
777++ /* constants cannot be incremented */
++(a * b - 1) /* cannot increment ordinary expressions */
```

表达式++i和i++都有一个值；每个表达式都使得内存中存储的变量i加1。表达式++i使得内存中存储的变量i先加1，然后该表达式以内存中i的新值作为它的值。相反，表达式i++以内存中i的值作为它的值，然后内存中存储的变量i再加1。下边的代码说明了这种情形。

```
int a, b, c = 0;
a = ++c;
b = c++;
printf("%d %d %d\n", a, b, ++c); /* 1 1 3 is printed */
```

以类似的方式，--i使得内存中存储的变量i先减1，然后该表达式以内存中i的新值作为它的值。表达式i--以内存中i的值作为它的值，然后内存中存储的变量i再减1。

要仔细地注意到，++和--会使内存中变量的值发生变化。相反，运算符+则不如此。像



$a+b$ 这样的表达式在计算后有一个结果值，但变量 $a$ 和 $b$ 的值不发生变化。可以通过说运算符 $++$ 和 $--$ 有副作用来表达这个思想；这些运算符不仅产生值，而且它们也改变在内存中存储的变量的值（请参见练习20）。

在一些情况下，我们在前缀或后缀位置使用 $++$ 有相同的结果。例如，表达式 $++i$ ；和 $i++$ ；都与语句 $i = i + 1$ ；等价。

在简单的情形中，可以用 $++$ 和 $--$ 作为对变量进行加1和减1的简洁表示法。在其他的情况下，对把它们是用于前缀位置还是后缀位置必须要谨慎小心。因为如果出错少，前缀运算符是更受欢迎的风格。表2-5中给出了实例。

声明和初始化		
<code>int a = 1, b = 2, c = 3, d = 4;</code>		
表 达 式	等价表达式	值
<code>a * b / c</code>	<code>(a * b) / c</code>	0
<code>a * b % c + 1</code>	<code>((a * b) % c) + 1</code>	3
<code>++ a * b - c --</code>	<code>((++ a) * b) - (c --)</code>	1
<code>7 - - b * ++ d</code>	<code>7 - ((- b) * (++ d))</code>	17

## 2.10 赋值运算符

为了改变变量的值，我们曾经使用过像

```
a = b + c;
```

这样的赋值语句。与其他语言不一样，C把 $=$ 处理为运算符，它的优先次序低于到目前为止我们曾讨论过的所有运算符，它的结合性规则是从左到右的。在本节，我们要详细地解释其含义。

为了理解 $=$ 是运算符，作为比较让我们首先考虑 $+$ 。二元运算符需要两个操作数，例如， $a + b$ 。该表达式的值是 $a$ 的值与 $b$ 的值之和。通过比较，简单的赋值表达式的形式为：

```
variable = right_side
```

此处的`right_side`是表达式。注意表达式结尾的分号使得它成为一个赋值语句。赋值运算符 $=$ 有两个操作数，分别为`variable`和`right_side`。`right_side`的值被赋给`variable`，`variable`的值也是赋值表达式的值。为了说明这点，考虑如下语句：

```
b = 2;
c = 3;
a = b + c;
```

此处的变量都是`int`型的。通过使用赋值表达式，我们可以把它们压缩成：

```
a = (b = 2) + (c = 3);
```

赋值表达式`b = 2`把值2赋给变量`b`，这个赋值表达式本身也是这个值。类似地，赋值表达式`c = 3`把值3赋给变量`c`，这个赋值表达式本身也是这个值。最终，两个赋值表达式的值相加，结果值赋给`a`。

尽管这个例子是人工模拟的，确实存在着赋值作为表达式的一部分的很多情形。一个经

常发生的情形是多重赋值。考虑如下语句：

```
a = b = c = 0;
```

由于运算符=是从右到左结合的，一个等价的语句是：

```
a = (b = (c = 0));
```

首先把0赋给c，表达式 $c = 0$ 的值是0。然后把0赋给b，表达式 $b = (c = 0)$ 的值是0。最后把0赋给a，表达式 $a = (b = (c = 0))$ 的值是0。很多语言不用这种精致的方式赋值。在这方面C是不同的。

除了=以外，还有像+=和-=这样的赋值运算符。一个这样的例子是：

```
k = k + 2
```

把2与k的旧值相加，并把结果值赋给k，表达式的值就是这个结果值。表达式 $k += 2$ 也能完成同样的工作。下表包含了所有的赋值运算符。

赋值运算符						
=	+=	-=	*=	/		
=	%=	>>=	<<=	&=	^=	=

所有的这些运算符有同样的优先次序，并且它们都有从右到左的结合性。用如下的形式描述该语义：

```
variable op= expression
```

它等价于

```
variable = variable op (expression)
```

但有一个例外，如果variable本身是表达式，variable仅被计算一次。在处理数组时，这是一个重要的技术点（请参见练习15）。要细心地注意到，像 $j *= k + 3$ 这样的赋值表达式与 $j = j * (k + 3)$ 是等价的，而不与 $j = j * k + 3$ 等价。

下表说明了如何计算赋值语句。

声明和初始化			
int i = 1, j = 2, k = 3, m = 4;			
表达式	等价表达式	等价表达式	值
i += j + k	i += (j + k)	i = (i + (j + k))	6
j *= k = m + 5	j *= (k = (m + 5))	j = (j * (k = (m + 5)))	18

虽然赋值语句有时与数学等式相似，但这两种表示法是有区别的，不应混淆。在你输入

```
x + 2 = 0; /* wrong */
```

时，数学等式 $x + 2 = 0$ 不会变成赋值等式。等号=的左边是一个表达式，而不是变量，因而该表达式没有被赋值。现在考虑赋值语句

```
x = x + 1;
```

x的旧值加1后赋给了x。如果x的旧值是2，则执行了该语句后x的值是3。请观察一下数学等式 $x = x + 1$ ，在从等式两边减去x后，我们得到的是 $0 = 1$ ，这是无意义的。虽然它们看起

来相像，但是C中的赋值运算符和数学中的等号是不可比较的。

## 2.11 例子：计算2的幂

为了说明本节中提到的一些思想，我们编写一个程序，它在一行中显示一些2的幂，这一程序如下：

```
/* Some powers of 2 are printed. */
#include <stdio.h>

int main(void)
{
    int    exponent = 0, power_of_two = 1;

    while (++exponent <= 10)
        printf("%5d", power_of_two *= 2);
    printf("\n");
    return 0;
}
```

该程序输出的结果如下：

```
2    4    8   16   32   64   128   256   512   1024
```

对程序pow\_of\_2的解析

- /\* Some powers of 2 are printed. \*/

程序经常用解释其用意或用途的注释开始。如果程序较大，注释也可能扩展。编译器把注释处理为空格。

- #include <stdio.h>

头文件stdio.h含有函数printf()的函数原型，这是对printf()的一种声明，有了它编译器能正确地工作。

- int exponent = 0, power\_of\_two = 1;

变量exponent 和power\_of\_two被声明为int类型，它们的初始值分别为0和1。

- while (++exponent <= 10)

只要++exponent的值小于或等于10，就执行while的循环体。第一次执行完循环体，++exponent的值是1；第二次执行完循环体，++exponent的值是2，以此类推。这样执行循环体10次。

- printf("%5d", power\_of\_two \*= 2);

while的循环体只有这一个语句。串常量"%5d"是printf()传递的第一个参数，该串含有格式%5，它表明按域长为5的十进制显示表达式power\_of\_two\*=2的值。

- power\_of\_two \*= 2

这个赋值表达式与表达式power\_of\_two=power\_of\_two \*2是等价的。它把power\_of\_two的旧值乘2后的结果值赋给power\_of\_two。赋给power\_of\_two的结果值就是这个赋值表达式的值。第一次执行完循环体，power\_of\_two的旧值是1，新值是2；第二次执行完循环体，power\_of\_two的的旧值是2，新值是4，以此类推。 ■

## 2.12 C系统

C系统由C语言、预处理器、编译器、库和程序员使用的诸如编辑器和调试器这样的其他

工具组成。在本节中，我们要讨论预处理器和库。关于标准库中的函数细节，请参见附录A“标准库”。

### 2.12.1 预处理器

把用#开始的行称为预处理指令(preprocessing directive)，这样的行要经过预处理器处理。在传统的C中，要求预处理指令从第一列开始，但在ANSI C中，已经取消了这个限制。虽然在#前可以加一个空格，但从第一列开始预处理指令仍然是常用的风格。

我们已经使用了像#include <stdio.h>和#define PI 3.14159这样的预处理指令。使用#include的另一种形式是#include "filename"。这使得预处理器用指定filename的文件内容替换该预处理指令所在的行。首先在当前的目录下搜寻该文件，然后在与系统有关的位置搜寻。用形式为#include <filename>的预处理指令，预处理器仅在“其他位置”搜寻该文件，而不是在当前目录下搜寻。

由于#include指令通常出现在程序的开头，所以把#include指令中引入的文件称为头文件(header file)，.h用于终结文件名。这是一种惯例，不是预处理器所要求的。对引入文件能包含什么没有限制。特别是，它能包含由预处理器依次展开的其他预处理指令。虽然可以引入任何种类的文件，但引入含有函数定义代码的文件则是一种不良的编程风格（请参见第5章“字符处理”）。

在UNIX系统中，像stdio.h这样的头文件通常可以在目录/usr/include下找到。在Borland C中，可以在目录/bc/include下找到头文件。一般地，标准#include文件的位置是与系统相关的。这样的文件都是可读的，由于各种原因，程序员偶尔会阅读它们。

头文件的基本用途之一是提供函数原型。例如，stdio.h含有这样的程序行：

```
int printf(const char *format, ...);
int scanf(const char *format, ...);
```

这两行是标准库中函数printf()和scanf()的函数原型。粗略地讲，函数原型告诉编译器传递给函数的参数类型和函数返回值的类型。在我们明白printf()和scanf()的函数原型之前，我们需要学习函数的定义机制、指针和类型限定词，在随后的章节中要提出这样的思想。我们在这里强调的是经常要引入的头文件，这是因为头文件包含了现在使用的函数的函数原型。编译器需要函数原型来正确地进行它的工作。

### 2.12.2 标准库

标准库包含有很多有用的函数，它为C系统添加了相当大的能力和灵活性。其中的很多要被所有的C程序员广泛地使用，而一些被有选择性地使用。大多数程序员都熟悉标准库中所应必知的基本函数。

通常程序员不关心标准库在系统中的位置，因为库包含的是不可读的已编译的代码。标准库可能由多个文件组成。例如，数学库在概念上是标准库的一部分，但它通常在一个分离的文件中。无论什么情况，系统都知道在库中的何处找到程序员使用的诸如printf()和scanf()这样的函数的相应代码；然而，即使系统提供代码，提供函数原型仍是程序员的责任。这通常通过引入适当的头文件来完成。

**警告** 不要把头文件错认为库。标准库含有已经编译了的函数目标代码。标准头文件不

包含已编译了的代码。

为了说明标准库中函数的用法, 让我们看一下rand()函数如何用于生成一些随机分布的整数。在随后的章节中, 我们会用rand()填充用于测试目的的数组和串, 此处, 我们用它在屏幕上显示一些整数。

```
/*Printing random numbers. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int i, n;

    printf("\n%s\n%s",
        "Some randomly distributed integers will be printed.",
        "How many do you want to see? ");
    scanf("%d", &n);
    for (i = 0; i < n; ++i) {
        if (i % 6 == 0)
            printf("\n");
        printf("%9d", rand());
    }
    printf("\n");
    return 0;
}
```

假设我们执行这个程序, 当出现提示时输入11, 如下的内容会出现在屏幕上:

```
Some randomly distributed integers will be printed.
How many do you want to see? 11

    16838     5758    10113    17515    31051     5627
    23010     7419    16212     4086     2749
```

#### 对程序prn\_rand的解析

- #include <stdio.h>  
#include <stdlib.h>

引入这些头文件是因为它们包含有函数原型。特别是, 函数原型int rand(void);是在stdlib.h中。它告诉编译器rand()是一个没有参数且返回值的类型为int的函数。不引入stdlib.h, 我们可以在文件的头部书写该行, 也可以在main()内部把该行作为一个声明。

- printf("\n%s\n%s",  
"Some randomly distributed integers will be printed.",  
"How many do you want to see? ");  
scanf("%d", &n);

该段程序在屏幕上向用户给出一个提示。scanf()接受用户输入的字符, 把字符转换成十进制整数格式, 并放于n中。

- for (i = 0; i < n; ++i) {  
.....  
}

这是for循环。它与如下程序段等价:

```
i = 0;
while (i < n) {
    .....
    ++i;
}
```

编写本程序的另一种方法是把*i*初始化为0，然后使用结构

```
while (i++ < n) {
    .....
}
```

要仔细地注意到，*i*++ < *n*与++*i* < *n*是不同的（请参见练习10）。

```
• if (i % 6 == 0)
    printf("\n");
```

运算符==是“相等”运算符。如果*expr1*和*expr2*是具有相同值的表达式，那么表达式*expr1* == *expr2*为真；否则为假。在第4章“函数和结构化编程”中，我们会看到==的优先级低于%。这样，*i*%6==0等价于（*i*%6）==0。该语句的效果是，首次执行循环以及以后的每6的倍数次执行循环，整个表达式为真。只要表达式为真，就显示一个换行符。

```
• printf("%9d", rand());
```

每次执行循环，都按十进制整数格式显示调用rand()的返回值。显示的整数域宽为9个字符。 ■

## 2.13 风格

下面的两个语句都是内存中的变量*i*加1。

```
++i;
i++;
```

表达式本身的值没有什么用处，仅是要利用运算符++的副作用。在这个简单的例子中，用++*i*还是*i*++仅凭个人的感觉，我们更喜欢前一种。作为更复杂的表达式的一部分，适当的做法是仅采用其中之一。这两个表达式的值是不同的。

正确的风格应争取代码的可读性。虽然语句

```
x = (y = 2) + (z = 3);
```

是正确的和简明的，但它不如下述语句可读：

```
y = 2;
z = 3;
x = y + z;
```

为了少用空格而精简代码是不必要的。可读性是不可忽略的。

如果我们想对变量*a*加7，我们可以写语句

```
a+=7;
```

或

```
a = a + 7;
```

虽然在很大的程度上选择那种方式凭个人感觉，但是专业程序员无疑更喜欢第一种。

注释对程序的可阅读性是十分重要的。没有哪种风格是一贯正确的。适当使用的注释可让其他人能明白程序做什么以及程序如何工作。个人和组织应该采取并遵循一致的注释风格。风格是一种习惯，良好的习惯有助于编好程序。

注释应该出现在程序的顶部以及主要的具有某种结构的源代码块的头部。当语句的作用不明显时，短的注释应该出现在单语句的右边。

应该显式地写出开头的注释，注释也应该包含诸如组织名称、程序员姓名、日期和程序的用途这样的信息。

```
/*
 * Organization:  SOCRATIC SOLUTIONS (Trade Mark)
 * Programmer:    Constance B. Diligent
 * Date:          19 April 1993
 *
 * Purpose:       Birthday greetings
 */

#include <stdio.h>

int main(void)
{
    printf("\nHAPPY BIRTHDAY TO YOU!\n\n");
    return 0;
}
```

虽然在实践中对代码的过度注释几乎永远不会发生，然而注释不应使得程序混乱，注释应该说明程序做什么。例如，

```
tax = price * rate;    /* sales tax formula */
```

中的注释可让人深入地了解程序，但

```
tax = price * rate;    /* multiply price by rate */
```

中的注释是多余的，因而是无用的。选择能描述其用途的标识符是非常重要的，因此可以避免过多的注释。

## 2.14 常见的编程错误

本章中讨论的编程错误主要是语法方面的。这些错误由编译器捕捉，一般而言在有错误的情况下编译器不会产生可执行的输出文件。

考虑在文件exmpl\_1.c中包含的代码。因为代码在语法上是不正确的，当我们编译它时，会产生错误和警告消息。消息的具体形式随编译器的不同而不同，但一般而言消息的内容是类似的。

```
#include <stdio.h>

int main(void)
{
    int    a = 1, b = 2, c = 3;

    x = a + b;
    printf("x = %d\n", x);
    return 0;
}
```

假设我们在一个Borland C系统上用bcc命令编译这个程序。要产生如下的一些信息：

```
Error EXMPL_1.C 7: Undefined symbol 'x' in function main
Warning EXMPL_1.C 9: 'c' is assigned a value that is never used
Warning EXMPL_1.C 9: 'b' is assigned a value that is never used
.....
```

信息中列出了包含代码的文件名以及错误出现的行号。集成环境tc高亮显示错误出现的行，这样程序员能马上用编辑器改正有错误的行。错误是容易理解的，即使用了x但没有进行声明。第一个警告消息是正确的，第二个则不然，第二个警告消息是误报，其实是没有对x进行声明。

让我们考虑另一个例子。除非你是一个很有经验的程序员，否则你是不会一眼就能看出错误的。

```
#include <stdio.h>

int main(void)
{
    int    a, b = 2, c = 3;    /* a, b, and c will be used
                               to illustrate arithmetic */

    a = (4 * b + 5 * c) / 6;
    printf("a = %d    b = %d    c = %d\n", a, b, c);
    return 0;
}
```

编译器会产生大量的信息，其中的一些是误报的。

```
Error EXMPL_2 12: Unexpected end of file in comment started
                  on line 5 in function main
Error EXMPL_2 13: Compound statement missing } in function main
Warning EXMPL_2 13: 'c' is assigned a value that is never used
                  in function main
.....
```

所发生的是，在第5行开始的注释从未被关闭，第一个错误指出了这点。注意，由于编译器不能找到用以结束程序的}，消息中的行号是误报的，也产生了对c的误报。编译器经常产生此类的误导建议。编译器的自动错误检测不能代替编程时的精心设计。

在程序员改正一个错误后，编译器可能还揭示了其他错误。上述程序中就有这种现象，如在printf()语句的上一行的结尾少了一个分号。

大多数编译器有关于其产生警告种类的选项。通常，应该尽可能高地设置警告的级别。考虑下边的代码：

```
#include <stdio.h>

main(void)
{
    printf("Try me!\n");
    return 0;
}
```

用选项-w设置Borland编译器的最高警告级别。我们给出命令

```
bcc -w try_me.c
```

会产生如下的警告：

```
Warning TRY_ME.C 6: Function should return a value
                  in function main
```

如果我们把main()换为void main(void)则不会出现这个警告。这会告诉编译器main()是一个没有参数且没有返回值的函数。Borland编译器欢迎这样做，但由于技术原因，一些其他的编译器不欢迎这样做。随后我们要对这点做更详细的讨论（请参见4.16节“系统考虑”）。警告，在一些编译器中选项-w会关闭所有的警告！此类选项和对它们的激活随编译器的不同而不同。

## 2.15 系统考虑

ANSI C中有一元减和一元加，但传统C中只有一元减。如果你正编写必须运行在传统编译器和ANSI C编译器上的可移植代码，你就不应该使用一元加运算符。



浮点类型long double在传统C中是不可用的。因为对ANSI C来说该类型是新的，很多编译器以相同的方式处理double和long double。随着时间的变化，这种情况会发生变化。

在操作系统和C编译器中的一个主要不同是辨别内部的和外部的字符的长度。宏的名字和普通变量都是内部标识符的例子。在一些系统中，可以接受多于8个字符的标识符，但仅使用前8个字符，其余的字符被丢弃了。例如，在这样的系统中，变量名cafeteria\_1和cafeteria\_2被认为是相同的。

像printf和scanf这样的标准库中的函数名以及文件名都是典型的外部标识符。在MS-DOS中，把文件名限定为8个字符，且用另外的3个字符做扩展名。在一个给定的系统中，程序员要通过实验、阅读手册和请教他人来学习可辨别的字符长度，这不成问题。另一方面，如果程序员正在编写要运行在多种系统上的C代码，就要知道且遵循所有这样系统的限制。在ANSI C中，编译器必须至少要辨别内部标识符的31个字符，系统必须至少要辨别外部标识符的6个字符。很多系统能识别更多的字符。

## 2.16 转向C++

C++的注释行的注释符是//，这是对C的括起来的注释符的改进，因为这样能减少错误，并且也更方便。

```
//The circumference and area of a circle.
//      by
//      Cottage Consultants - LMP & DJD
//      V 2.3

const double pi = 3.14159; //pi to 6 significant digits
```

C++允许声明与可执行语句混合。下边的程序是上边的程序的继续。

```
#include <iostream.h>      //standard C++ IO library

int main()
{
    while (true) {          //infinite loop - control-C exits;
    true is a bool
        cout << "\nENTER radius: "; //prompt
        double r;              //C++ allows declarations inside block
        cin >> r;              //input
        double diam = 2 * r;    //declare and initialize
        cout << "\nDiameter = " << diam;
        cout << "\nArea = " << pi * r * r;
        cout << "\nCircumference = " << pi * diam<< endl;
    }
    return 0; //this may be omitted; return 0 is understood
}
```

依据C++允许声明与可执行语句混合，你可以在使用变量时再声明它，这能避免初始化错误。

C++的关键字数量约为C的两倍。尽管合法，但在C程序中用C++的关键字做标识符是不良的做法。C++中使用的一些关键字有：用于隐藏数据的private、public和protected；用于例外处理的catch、try和throw；用于内存管理的新和delete；用于定义数据类型的class和template。

## 小结

- 标记是C的基本语法单位。其中包括关键字、标识符、常量、串常量、运算符和标点。

空格、运算符和标点也能用作分隔标记。由于这个原因，把空格、运算符和标点统称为分隔符(separator)。当空格不被用作分隔标记时，编译器会忽略它。

- 编译器把用/\*和\*/括起来的注释处理为空格。对于良好的程序文档来说，注释是非常重要的，注释有助于读者使用和理解程序。
- 称为保留字的关键字有明确的含义。在C中有32个关键字，不能再对它们进行重定义。
- 标识符是程序员用于对变量和函数命名的标记。它们用字符或下划线开头，对于阅读者来讲，它们应有一定的含义。
- 系统已经使用了一些标识符作为标准库中的函数名，其中包括输入/输出函数scanf()和printf()，以及像sqrt()、sin()、cos()和tan()这样的数学函数。
- 常量包括各种类型的整型常量、浮点型常量、像'a'和'#'这样的字符常量以及像"abc"这样的串常量。编译器把所有的常量都作为标记。
- 像"deep blue sea"这样的串常量是包括空格在内的字符的随机序列，要把串常量放在双引号内。串常量被存储为字符数组，但编译器把它处理为单标记。编译器在内存中为串常量提供所需要的存储空间。字符常量和串常量在处理上是不同的。例如，'x'和"x"是不同的。
- 在C中，运算符和标点是可枚举的。在main()中的圆括号是运算符；这样的圆括号用于告诉编译器main是一个函数。在表达式a\*(b+c)中的圆括号是标点。应该先计算圆括号内的表达式。
- 在C中，运算符的优先级规则和结合性规则决定了怎样计算表达式。程序员应该知道这些。
- 增量标识符++和减量标识符--有副作用。此外它们它们都一个值，像++i这样的表达式会使得存储在内存中的i的值被加1。
- 标识符++和--可用在前缀和后缀的位置，但效果可能不同。表达式++i使内存中的i的值被加1，i的新值是表达式的值。表达式i++的值就是i的当前值，然后i的值在内存被加1。
- 在C中，赋值符号是等号。像a=b+c这样的表达式把b+c的值赋给a，表达式的值就是这个计算出的值。虽然C中的赋值运算符和数学中的等号看起来相像，但它们间不具有可比性。
- 在标准库中有很多有用的函数，程序员可以使用它们。当要用标准函数时，通过引入适当的标准头文件，就可以获得相应的函数原型。

## 练习

1. main是关键字吗？请解释。
2. 请列出5个关键字，并解释它们的用法。
3. 举例给出3种标记。
4. 下述的哪些不是标识符，为什么？

```
3id      o_no_o_no      00_go      star*it      __yes
1_i_am   one_i_aren't    me_to-2    xYshouldI    int
```

5. 设计一个介绍性注释的标准格式，其内容包括程序的作者及程序的用途。
6. 选一个像+这样的符号，说明在程序中使用它的不同方法。

7. 尽管很多编译器提供了嵌套注释, 但ANSI C没有。在你的编译器中试一下如下的行, 看会发生什么。

```
/* This is an attempt /* to nest */ a comment. */
```

8. 编写一个把磅和盎司转换成千克和克的交互式程序。请使用在main()前定义的符号常量。

9. 本练习要说明在一个运算符前后的空格位置是很重要的。因为+和++都是运算符, 表达式a+++b可被解释成a++ +b或a+ ++b, 这依赖于怎样对加号分组。通常要把头两个加号分为一组进行编译, 以分析语法是否正确。请编写一个短程序, 看你的编译器做出怎样的解释。

10. 对于2.11节“例子: 计算2的幂”中的程序pow\_of\_2, 请说明如果表达式++i变为i++后会产生什么结果。

11. 研究下列代码, 并写出你模拟出的显示结果。再编写一个测试程序, 检查你的答案。

```
int a, b = 0, c = 0;
a = ++b + ++c;
printf("%d %d %d\n", a, b, c);
a = b++ + c++;
printf("%d %d %d\n", a, b, c);
a = ++b + c++;
printf("%d %d %d\n", a, b, c);
a = b-- + --c;
printf("%d %d %d\n", a, b, c);
```

12. 如果去掉语句

```
x = (y = 2) + (z = 3);
```

中的一部分或所有的括号, 会产生什么影响? 请解释。

13. 完成下表。然后编写一个程序, 检查你写入的结果是否正确。

声明和初始化		
int a = 2, b = -3, c = 5, d = -7, e = 11, f = -3;		
表达式	等价表达式	值
a / b / c	(a / b) / c	0
7 + c * -- d / e	7 + ((c * (-- d)) / e)	
2 * a % - b + c + 1		
39 / - ++ e - + 29 % c		
a += b += c += 1 + 2		
7 - + ++ a % (3 + f)	错误, 为什么?	

14. 考虑下面的代码:

```
int a = 1, b = 2, c = 3;
a += b += c += 7;
```

用加括弧的方式, 写出一个等价的表达式。变量a、b和c的值是多少? 先写出你的答案, 然后编写一个测试程序, 检查你的答案。

15. 对于读者来说, 良好的编程风格是至关重要的, 尽管是编译器只关心字符流。考虑下面的程序:

```
int main(void)
{float qx,
  zz,
```

```

tt;printf("gimme 3"
);scanf
( "%f%f %f",&qx,&zz
,&tt);printf("average is=%f",(qx+tt+zz)/3.0);
return 0;}

```

虽然本段代码的可读性并不怎么好，但它是可编译和可执行的。请执行它，看它的运行情况。再编写一个完成的程序，用空格和注释使它更可读和文档化。提示：引入一个头文件，选择用于代替qx、zz和tt的新标识符。

16. 函数rand()产生的整数都落在区间[0,n]内，此处的n是与系统相关的。在ANSI C中，符号常量RAND\_MAX决定了给定的n的值，在标准头文件stdlib.h中定义了这个符号常量。当然，在不完善的ANSI C系统中这个符号常量可能会不可用。在你的C系统中该符号常量是可用的吗？编写一个程序进行检验。

17. 函数rand()返回的整数落在区间[0,RAND\_MAX]内（请参见上一个练习）。如果你声明一个变量median，并用值RAND\_MAX/2进行初始化，那么rand()的返回值有时大于median，有时小于median。然而平均来讲，比median大的值可能与比median小的值一样多。下面要验证这个假设。编写一个称为rand()的程序，for循环要进行500次，对于每一次循环，变量plus\_cnt加1，rand()返回值小于median，变量minus\_cnt减1，rand()返回值大于median。对于每次循环，显示出plus\_cnt和minus\_cnt的值差。值差的震荡应该接近于零。结果是这样吗？

18. 重新编写2.12.2节“标准库”中的程序prn\_rand，使得显示出的整数在区间[1,100]内。提示：使用取模运算符。在显示出100之前，你要显示出多少数？（如果没有输出100，程序肯定有错误）。

19. 利用如下的结构重新编写程序prn\_rand：

```

while (i++ < n) {
    .....
}

```

在运行程序并理解它的作用后，重新编写程序，把i++<n改为++i<n，此时的程序行为发生了变化。为了弥补这种变化，重新编写while循环体，恢复程序的原来行为。

20. 像++a和a++这样的表达式的值是依赖于系统的，这是因为增量运算符++的副作用会在不同的时间发生。这既是C的长处（因为编译器只能在机器的水平上做事情）又是C的弱点（因为这样的表达式是依赖于系统的，并且在不同的机器上其值是不同的）。有经验的C程序员认为这种表达式具有潜在的危险，而不使用它们。在你的机器上进行实验，看一下在a被初始化为0后，++a+a++会产生什么样的值。你的编译器给出该表达式危险的警告了吗？

21. 在UNIX中库通常以.a结尾，它是“archive”的助记符。在MS-DOS中库通常以.lib结尾。看一下你能否在你的系统中找到标准的C库。这些库是不可读的。在UNIX系统中，你能用命令ar t /lib/libc.a 来看到库中各对象的标题。

22. 在ANSI C和传统的C中，一个串常量中的行尾处的反斜杠表示下一行是本行的继续。例如，

```

"by using a backslash at the end of the line \
a string can be extended from one line to the next"

```

用这种结构编写一个程序。很多屏幕每行有80个字符。如果显示串的字符多于80个，会发生什么情况？

23. 在ANSI C中, 在一行尾处的反斜杠表示下一行是本行的继续。在任何C编译器(无论是ANSI C还是传统C)中, 都可把反斜杠用在串常量和宏定义中(请参见上一个练习)。除了用于宏定义外, 这种结构用得很少。你的编译器支持这个结构吗? 请用下述语句进行实验。

```
#\
include <stdio.h>
mai\
n()
{
    print\
f("Will this \
work?\n");
}
```

24. 当你调用编译器时, 系统首先调用预处理器。在本练习中, 我们将故意地设置一个预处理错误, 看会发生什么情况。试一下下面的程序:

```
#incl <stdixx.h>

int main(void)
{
    printf("Try me.\n");
    return 0;
}
```

如果把#incl改为#include会发生什么情况?

25. C++: 用C++对计算2的幂的程序重新编码。

26. C++: 重新编写程序prn\_rand, 使得显示出的整数位于区间[0,M]中。程序要提示输入整数M, 并要用cin实现输入。

27. Java: 如下是一个计算矩形面积的程序:

```
// SimpleInput.java-reading numbers from the keyboard
//Java by Dissection: page 27
import tio.*; // use the package tio

class SimpleInput {
    public static void main (String[] args) {
        int width, height, area;

        System.out.println("type two integers for" +
            " the width and height of a box");
        width = Console.in.readInt();
        height = Console.in.readInt();
        area = width * height;
        System.out.print("The area is ");
        System.out.println(area);
    }
}
```

把这个程序转换转换成C语言程序。请注意如何用scanf()代替包tio中的方法readInt()。为了《Java解析》开发了方法readInt(), 用以提供简单的终端输入/输出, 这样Java I/O在类型上是安全的。在C中, 当要为printf()和scanf()编写格式控制串时, 程序员要非常小心。

28. Java的更重要的优势是, 无论代码运行在什么机器上, 程序产生的结果都是一样的。Java的类型是与机器无关的, C的类型是与机器相关的, 随着机器的不同可能会发生变化。当用像double这样的浮点类型时, 这点特别重要。上一个练习中的代码改用了double类型后, 程序如下:

```
import tio.*; // use the package tio

class SimpleInputDouble {
    public static void main (String[] args) {
```

```
double width, height, area;

System.out.println("type two integers for" +
    " the width and height of a box");
width = Console.in.readDouble();
height = Console.in.readDouble();
area = width * height;
System.out.print("The area is ");
System.out.println(area);
    }
}
```

用C编写这个程序。在你的机器上，看用Java和C编写的程序是否给出了相同的结果。

## 第3章 控制流

程序中的语句通常是一个接着一个地执行的，这被称为顺序控制流(sequential flow of control)。我们经常要改变串行控制流，以进行行为选择或行为重复。我们可以用if、if-else和switch语句在可选择的行为中进行选择，用while、for和do语句完成重复行为。我们要在本章中解释这些控制流结构。我们将从关系、等式和逻辑运算符开始讨论，我们还将讨论用于把语句组合起来作为一个单元处理的复合语句。

### 3.1 关系、等式和逻辑运算符

下表中的运算符经常用于影响控制流。

关系、等式和逻辑运算符		
关系运算符	小于	<
	大于	>
	小于或等于	<=
	大于或等于	>=
等式运算符	相等	==
	不等	!=
逻辑运算符	(一元)非	!
	逻辑与	&&
	逻辑或	

我们在一开始要详尽地对这些运算符进行讨论。这些运算符用于表达式之中，我们认为表达式具有真(true)假(false)值。我们要解释怎样在C中实现true和false。

像其他运算符一样，关系、等式和逻辑运算符有优先级规则和结合性规则，这些规则精确地决定了涉及到这些运算符的表达式如何被计算。下表中列出了这些规则。

运算符优先级和结合性	
运算符	组合性
() ++ (后缀) -- (后缀)	从左到右
+(一元) -(一元) ++ (前缀) -- (前缀)	从右到左
* / %	从左到右
+ -	从左到右
< <= > >=	从左到右
== !=	从左到右
&&	从左到右
	从左到右
?:	从右到左
= += -= *= /= 等	从右到左
, (逗号运算符)	从左到右

！运算符是一元的，所有其他的关系、等式和逻辑运算符都是二元的。它们都作用于表达式，表达式的计算结果为int型的值0或1，其原因是在C语言中用0表示false，用非0值表示true。能用于表示false的一些表达式的例子有：具有0值的整型表达式、具有0.0值浮点表达式、空字符'\0'（参见第5章“字符处理”）和空指针（参见第8章“函数、指针和存储类型”）。类似地，具有非0值的表达式能用于表示true。在直觉上，像 $a < b$ 这样的表达式的值不是true就是false。在C中，如果该表达式是true，那么它产生int型值1；如果该表达式是false，那么它产生int型值0。

## 3.2 关系运算符和表达式

关系运算符

`< > <= >=`

都是二元的。它们都用两个表达式作为操作数，产生int型值0或int型值1。如下是一些合法的表达式：

```
a < 3
-1.1 >= (2.2 * x + 3.3)
a < b < c /* syntactically correct, but confusing */
```

下面的写法是不合法的：

```
a =< b /* out of order */
a < = b /* space not allowed */
a >> b /* this is a shift expression */
```

考虑像 $a < b$ 这样的表达式。如果 $a$ 小于 $b$ ，那么表达式的结果是int型值1。如果 $a$ 不小于 $b$ ，那么表达式的结果是int型值0，这被认为是假。观察一下， $a < b$ 的值和 $(a - b) < 0$ 的值是相同的。因为关系运算符的优先级低于数学运算符的优先次序，所以 $a - b < 0$ 与 $(a - b) < 0$ 是等价的。通常数学转换出现在关系表达式中（参见第6章“基本数据类型”）。

设 $a$ 和 $b$ 是任意的两个数学表达式。下表表明了 $a - b$ 的值如何决定了关系表达式的值。

关系表达式的值				
$a - b$	$a < b$	$a > b$	$a <= b$	$a >= b$
positive	0	1	0	1
zero	0	0	1	1
negative	1	0	1	0

下表说明了用于计算关系表达式的优先级规则和结合性规则的用法。

声明和初始化		
<pre>int    i = 1, j = 2, k = 3; double x = 5.5, y = 7.7;</pre>		
表达式	等价表达式	值
$i < j - k$	$i < (j - k)$	0
$-i + 5 * j >= k + 1$	$((-i) + (5 * j)) >= (k + 1)$	1
$x - y <= j - k - 1$	$(x - y) <= ((j - k) - 1)$	1
$x + k + 7 < y / k$	$((x + k) + 7) < (y / k)$	0



### 3.3 等式运算符和表达式

等式运算符`==`和`!=`是用于表达式的二元运算符，它们产生`int`型值0或`int`型值1。如下是一些合法的表达式的例子：

```
ch == 'A'
count != -2
x + y == 2 * z - 5
```

下面的写法是不合法的：

```
a = b          /* an assignment expression */
a == b - 1     /* space not allowed */
(x + y) != 44  /* syntax error: equivalent to (x+y) != 44 */
```

在直觉上，像`a == b`这样的表达式的结果不是`true`就是`false`。更精确地讲，如果`a`等于`b`，那么表达式的结果是`int`型值1 (`true`)，否则表达式的结果是`int`型值0 (`false`)。注意等价的表达式是`a - b == 0`，这是在机器水平上所实现的。

表达式`a != b`说明了“不等于”运算符的用法。除了用不等于代替等于外，计算它的方法与计算`a == b`类似。下表给出了该运算符的语义。

等式表达式的值		
<code>a - b</code>	<code>a == b</code>	<code>a != b</code>
zero	1	0
nonzero	0	1

下表说明了优先级规则和结合性规则如何用于计算带有等式运算符的表达式。

声明和初始化		
<code>int i = 1, j = 2, k = 3;</code>		
表达式	等价表达式	值
<code>i == j</code>	<code>j == i</code>	0
<code>i != j</code>	<code>j != i</code>	1
<code>i + j + k == - 2 * - k</code>	<code>((i + j) + k) == ((- 2) * (- k))</code>	1

### 3.4 逻辑运算符和表达式

逻辑运算符`!`是一元的，逻辑运算符`&&`和`||`是二元的。当应用到表达式时，这些运算符都产生`int`型值0或`int`型值1。

可以把逻辑非应用到数学或指针类型的表达式。如果表达式的值是0，对它做非运算，则产生的结果是`int`型值1，如果表达式的值不是0，对它做非运算，则产生的结果是`int`型值0。

一些例子如下：

```
!a
!(x + 7.7)
!(a < b || c < d)
```

但如下两个式子是非法的：

```
a!           /* out of order */
a != b       /* != is the token for "not equal" operator */
```

下表给出了! 运算符的语义。

非表达式的值	
a	!a
零	1
非零	0

虽然逻辑非是一个非常简单的运算符，但也有微妙之处。在C中，运算符!与一般逻辑中的否定(not)运算符不同。如果s是一个逻辑语句，那么 $\text{not}(\text{not } s)=s$ 。而在C中，!!5的值是1。因为! 从右到左进行结合，这与所有其他的一元操作一样，所以表达式!!5与!(!5)等价，!(!5)与其值是1的!(0)等价。下表说明了如何计算一些带有逻辑非的表达式。

声明和初始化		
int i = 7, j = 7; double x = 0.0, y = 999.9;		
表达式	等价表达式	值
!(i - j) + 1	(!(i - j)) + 1	2
! i - j + 1	((! i) - j) + 1	-6
!! (x + 3.3)	! (x + 3.3)	1
! x * !! y	(! x) * (!! y)	1

二元逻辑运算符&&和||也用于表达式，产生int型值0或int型值1。这样的一些表达式如下：

```
a && b
a || b
!(a < b) && c
3 && (-2 * a + 7)
```

但如下的式子是非法的：

```
a &&           /* one operand missing */
a || b        /* extra space not allowed */
a & b          /* this is a bitwise operation */
&b            /* the address of b */
```

下表给出了这两个二元运算符的语义。

位表达式的值			
a	b	a && b	a    b
零	零	0	0
零	非零	0	1
非零	零	0	1
非零	非零	1	1

虽然该表完全准确, 但没反应出程序员在处理逻辑表达式时的思维方法, 甚至有经验的程序员会按真值的术语把该表认作为下表。

&&和  的真值表			
a	b	a && b	a    b
F	F	F	F
F	T	F	T
T	F	F	T
T	T	T	T

&&的优先次序高于||, 但这两个运算符的优先次序都低于所有的一元运算符。下表说明了优先级规则和结合性规则如何用于计算一些逻辑表达式的值。

声明和初始化		
<pre>int      i = 3, j = 3, k = 3; double   x = 0.0, y = 2.3;</pre>		
表达式	等价表达式	值
<code>i &amp;&amp; j &amp;&amp; k</code>	<code>(i &amp;&amp; j) &amp;&amp; k</code>	1
<code>x    i &amp;&amp; j - 3</code>	<code>x    (i &amp;&amp; (j - 3))</code>	0
<code>i &lt; j &amp;&amp; x &lt; y</code>	<code>(i &lt; j) &amp;&amp; (x &lt; y)</code>	0
<code>i &lt; j    x &lt; y</code>	<code>(i &lt; j)    (x &lt; y)</code>	1

### 短路求值

计算含有&&和||的运算数的表达式时, 只要得到了结果真或假, 求值的过程就停止, 把这样的计算称为短路求值(short-circuit evaluation), 这是这些运算符的一个重要的性质。假设`expr1`和`expr2`是表达式, `expr1`的值是0。在对逻辑表达式

`expr1 && expr2`

的求值时, 不会对`expr2`进行求值, 这是因为从整体上来说, 已经决定了该逻辑表达式的值是0。类似地, 如果`expr1`的值不为0, 那么对

`expr1 || expr2`

求值时, 不会对`expr2`进行求值, 这是因为从整体上来说, 已经决定了该逻辑表达式的值是1。下边的代码说明了短路求值:

```
int  i, j;
i = 2 && (j = 2);
printf("%d %d\n", i, j);           /* 1 2 is printed */
(i = 0) && (j = 3);
printf("%d %d\n", i, j);           /* 0 2 is printed */
i = 0 || (j = 4);
printf("%d %d\n", i, j);           /* 1 4 is printed */
(i = 2) || (j = 5);
printf("%d %d\n", i, j);           /* 2 4 is printed */
```

下面是一个如何使用短路求值的简单例子。假设我们要依赖一定的条件做一个计算, 该

条件是:

```
if (x >= 0.0 && sqrt(x) <= 7.7) {
    .....                               /* do something */
}
```

如果x值为负, 不求x的平方根(参见练习20)。

### 3.5 复合语句

复合语句是由一系列的用花括号括起来的声明和语句。复合语句的主要用途是把多个语句组成一个可执行的单元。当声明出现在复合语句的头部时, 就把此部分称为块(block)(请参见第8章的“函数、指针和存储类型”)。在C中, 凡是在语法上能正确地出现语句的地方, 都能出现复合语句。复合语句本身是一个语句。

一个复合语句例子如下:

```
{
    a = 1;
    {
        b = 2;
        c = 3;
    }
}
```

注意本例中在复合语句中还有一个复合语句。复合语句的一个重要的用途是实现在if、if-else、while、for、do和switch语句中所需要的控制流。

### 3.6 空语句

用一个单分号表示空语句。在语法上需要出现一个语句但在语义上并不需要任何行为的情况是有用的。像我们已经看到的那样, 在像if-else和for语句这样的控制流结构中这种情况有时是很有用的。

紧跟一个分号的表达式被称为表达式语句(expression statement), 空语句是表达式语句的一种特殊情况。如下是一些表达式语句的例子:

```
a = b;                /* an assignment statement */
a + b + c;            /* legal, but no useful work done */
;                     /* an empty statement */
printf("%d\n", a);    /* a function call */
```

### 3.7 if和if-else语句

if语句的一般形式为:

```
if(expr)
    statement
```

如果expr是非零(true), 那么执行statement; 否则跳过statement, 控制转到下一个语句。在如下的例子中:

```
if (grade >= 90)
    printf("Congratulations!\n");
printf("Your grade is %d.\n", grade);
```

当grade的值大于或等于90时, 就显示一条祝贺信息。第二个显示语句总是要执行的。

通常在if语句中的表达式是关系、等式和逻辑表达式, 但用其他领域的表达式也是允许

的。下面是一些其他if语句的例子：

```
if (y != 0.0)
    x /= y;
if (a < b && b < c) {
    d = a + b + c;
    printf("Everything is in order.\n");
}
```

但下面的不是：

```
if b == a          /* parentheses missing */
    area = a * a;
```

在适当的位置，用复合语句把单if语句的控制下的一系列语句群聚成一组。如下代码由两个if语句组成：

```
if (j < k)
    min = j;
if (j < k)
    printf("j is smaller than k\n");
```

通过用带有复合语句作为语句体的单if语句，可以以更有效和更易理解的形式编写这段代码。

```
if (j < k) {
    min = j;
    printf("j is smaller than k\n");
}
```

if-else语句与if语句是紧密相关的。它的一般形式为：

```
if (expr)
    statement1
else
    statement2
```

如果expr的值不是零，那么就执行statement1，而跳过statement2；如果expr的值为零，那么就执行statement2，而跳过statement1。无论哪种情况，在if语句执行后控制都转到下一个语句。考虑下述代码：

```
if (x < y)
    min = x;
else
    min = y;
printf("Min value = %d\n", min);
```

如果x<y为true，那么把x的值赋给min；如果x<y为false，那么把y的值赋给min，然后控制转给printf()语句。下面是另一个if-else结构的例子：

```
if (c >= 'a' && c <= 'z')
    ++lc_cnt;
else {
    ++other_cnt;
    printf("%c is not a lowercase letter\n", c);
}
```

但下面代码不是if-else结构：

```
if (a != b) {
    a += 1;
    b += 2;
};
else          /* syntax error */
    c *= 3;
```

由于右花括号后跟的一个分号引起了一个空语句，后续的else不属于任何if语句，所以会产生一个语法错误。

因为if语句本身是一个语句，所以可以把它作为另一个if语句的一部分使用。考虑如下代码：

```
if (a == 1)
    if (b == 2)
        printf("***\n");
```

这段代码的形式为：

```
if (a == 1)
    statement
```

此处statement是如下的if语句：

```
if (b == 2)
    printf("***\n");
```

以类似的方式，可以把if-else作为另一个if-else语句的一部分使用。考虑下边的例子：

```
if (a == 1)
    if (b == 2)
        printf("***\n");
    else
        printf("###\n");
```

现在我们面临着一个语义难题。这段代码说明了一个“else的悬摆”问题：else部分属于哪个if是不清楚的。不要为这段代码的格式所愚弄，就机器关心的问题而言，这段代码与如下代码是等价的：

```
if (a == 1)
    if (b == 2)
        printf("***\n");
else
    printf("###\n");
```

规则是else属于离它最近的if，这样，第一次给出的代码格式是正确的。它的形式如下：

```
if (a == 1)
    statement
```

此处statement是如下的if-else语句：

```
if (b == 2)
    printf("***\n");
else
    printf("###\n");
```

为了说明if和if-else语句的用法，我们要编写一个求出由键盘输入的3个值中的最小值的程序。

```
/* Find the minimum of three values. */
#include <stdio.h>

int main(void)
{
    int    x, y, z, min;

    printf("Input three integers: ");
    scanf("%d%d%d", &x, &y, &z);
    if (x < y)
        min = x;
    else
        min = y;
```

```

    if (z < min)
        min = z;
    printf("The minimum value is %d\n", min);
    return 0;
}

```

### 对程序find\_min的解析

- #include <stdio.h>

头文件stdio.h由系统提供, 引入它的原因是它含有printf()和scanf()的函数原型。

- printf("Input three integers: ");

在交互式环境中, 程序必须提示用户输入。

- scanf("%d%d%d", &x, &y, &z);

库函数scanf()用于读入3个整数, 并分别存放到x、y和z中。

- if (x < y)
  - min = x;
  - else
    - min = y;

这个结构是单if-else语句, 它要比较x和y的值。如果x小于y, 那么把x的值赋给min; 如果x不小于y, 那么把y的值赋给min。

- if (z < min)
  - min = z;

这是一个if语句, 它检查z的值是否小于min的值。如果是, 那么把z的值赋给min; 否则min的值保持不变。 ■

## 3.8 while语句

实现行为的重复是我们使用计算机的一个原因。在处理大量的数据时, 使用控制重复地执行特定语句的机制是非常方便的。在C中, while、for和do都提供重复行为。

虽然我们已经在很多例子中使用了while语句即while循环, 但我们仍要精确地说明这种重复的机制是如何工作的。考虑如下形式的结构:

```

while (expr)
    statement
next statement

```

首先计算expr, 如果它的值不是0(ture), 那么执行statement, 并把控制回传给while循环体的开始处。这种机制的效果是让while的循环体 (即statement) 重复地执行, 直到expr是为止。当expr为0时, 控制传递到next statement。如下是一个例子:

```

while (i <= 10) {
    sum += i;
    ++i;
}

```

假设在循环前i的值是1, sum的值是0。这个循环的效果是通过变量i的值重复地对sum的值进行增量, 然后对i进行加1操作, 如下表所示。

在循环体执行10次后, i的值是11, 表达式i<=10的值是0(false), 因此, 循环体停止执行, 控制转给下一条语句。当退出while循环时, sum的值是55。再次注意, 复合语句用于把语句聚集成组, 复合语句在语法上本身表示单语句。

while循环执行期间的值	
循环次数	sum的值
第一次	0 + 1
第二次	0 + 1 + 2
第三次	0 + 1 + 2 + 3

### 3.9 问题求解：找最大值

程序员经常必须要在一个项集中寻找具有特殊意义的项。通过在键盘上交互式输入的一些实数中找最大值，我们要说明这样一项任务。我们的程序要使用if和while语句。

```
/* Find the maximum of n real values. */
#include <stdio.h>

int main(void)
{
    int    cnt = 0, n;
    float  max, x;

    printf("The maximum value will be computed.\n");
    printf("How many numbers do you wish to enter? ");
    scanf("%d", &n);
    while (n <= 0) {
        printf("\nERROR: Positive integer required.\n\n");
        printf("How many numbers do you wish to enter? ");
        scanf("%d", &n);
    }

    printf("\nEnter %d real numbers: ", n);
    scanf("%f", &x);
    max = x;
    while (++cnt < n) {
        scanf("%f", &x);
        if (max < x)
            max = x;
    }
    printf("\nMaximum value: %g\n", max);
    return 0;
}
```

假设我们执行这个程序，并在提示后输入5。在第二次提示后，如果输入1.01、-3、2.2、7.07000和5后在屏幕上会出现如下信息；

```
The maximum value will be computed.
How many numbers do you wish to enter? 5
Enter 5 real numbers: 1.01 -3 2.2 7.07000 5
Maximum value: 7.07
```

#### 对程序find\_max的解析

```
• int    cnt = 0, n;
  float  max, x;
```

变量cnt和n被声明为int型，变量max和x被声明为float型。我们用cnt作为计数器。

```
• printf("The maximum value will be computed.\n");
```

显示一行文本，解释程序的用意。这是一个重要的辅助文档，实际上，程序正对它的输出进行文档化，这是一种良好的编程风格。



```
• printf("How many numbers do you wish to enter? ");
  scanf("%d", &n);
```

提示用户输入一个整数。然后用函数scanf()把用户输入的整数值存储在n中。

```
• while (n <= 0) {
    printf("\nERROR: Positive integer required.\n\n");
    printf("How many numbers do you wish to enter? ");
    scanf("%d", &n);
}
```

如果n是负数或0,表达式n <= 0的值是1(true),这使得while循环体被执行。在显示错误消息和另一个提示后,在n中存储了一个新值。只要用户输入的不是负数,就重复地执行循环体。这个while循环体向程序提供了一些输入错误纠正功能,但像输入的是字符a而不是数字这样的输入错误仍然会导致程序运行失败。对于更健壮的错误纠正功能,我们需要考虑用户输入的实际字符。为了做到这一点,我们需要字符处理工具和串(请参见第5章“字符处理”和第10章“串和指针”)。

```
• printf("\nEnter %d real numbers: ", n);
  scanf("%f", &x);
  max = x;
```

提示用户输入n个实数。函数scanf()使用格式%f把输入流字符转换成浮点数,并把值存储到x中。把x的值赋给变量max。

```
• while (++cnt < n) {
    scanf("%f", &x);
    if (max < x)
        max = x;
}
```

在进入这个循环之前,我们已经为x输入了一个值。在循环内,通过对cnt增量统计x。在第1次循环中,表达式++cnt的值是1,变量n的值是5,表达式++cnt < n的值是1(true)。在每一次循环中,都为x输入一个值,并测试它是否大于当前的max,如果是,就把x赋值给max。然后把控制回传到循环体的开始处,在这里对cnt加1,并测试它以决定我们是否需要输入更多的值。while语句体是一个单语句,在本例中它是一个复合语句。通过把几个执行语句语句聚集成一个单元,复合语句有助于控制流。

```
• printf("\nMaximum value: %g\n", max);
```

我们用格式%g显示max的值。注意,在键盘上输入的是7.07000,而显示的是7.07。格式%g没有显示多余的0。 ■

### 3.10 for语句

像while语句一样,for语句用于重复地执行代码。我们能按照while语句的术语解释for语句的行为。其结构如下:

```
for (expr1; expr2; expr3)
    statement
next statement
```

它与下边的语句在语义上是等价的:

```
expr1;
while (expr2) {
    statement
    expr3;
```

```

}
next statement

```

假设`expr2`存在, 并且在`for`循环内没有任何`continue`语句。从我们对`while`语句的理解来看, 我们能看出`for`语句的语义是这样的: 首先计算`expr1`; 通常`expr1`用于初始化循环。然后计算`expr2`; 如果它的值不是0(true), 那么执行`statement`, 计算`expr3`, 并把控制回传到`for`循环的开始处, 但不再对`expr1`求值。通常`expr2`是一个控制重复的逻辑表达式。这个过程重复进行, 直到`expr2`的值是0(false)为止。当`expr2`的值是0时, 就把控制传递到`next statement`。

下面是`for`循环或`for`语句的一些例子:

```

for (i = 1; i <= n; ++i)
    factorial *= i;
for (j = 2; k % j == 0; ++j) {
    printf("%d is a divisor of %d\n", j, k);
    sum += j;
}

```

但以下语句是错误的;

```

for (i = 0, i < n, i += 3)    /* semicolons are needed */
    sum += i;

```

`for`语句内的任何或所有语句都可以遗漏, 但是两个分号必须保留。如果遗漏了`expr1`, 就没有执行作为`for`循环一部分的初始化步骤。

```

i = 1;
sum = 0;
for ( ; i <= 10; ++i)
    sum += i;

```

以上代码计算从1到10的整数之和, 下边的代码也能实现这个功能:

```

i = 1;
sum = 0;
for ( ; i <= 10 ; )
    sum += i++;

```

若遗漏了`expr2`, 则测试总为true。这样在下边代码中的`for`循环是一个无限循环:

```

i = 1;
sum = 0;
for ( ; ; ) {
    sum += i++;
    printf("%d\n", sum);
}

```

`for`语句能用作`if`、`if-else`、`while`和另一个`for`语句的一部分。考虑如下的结构:

```

for ( ... )
    for ( ..... )
        for ( ..... )
            statement

```

整体上这个结构是一个单`for`语句, 它的语句部分是另一个`for`语句, 第二个`for`语句也是如此。

在很多情形下, 通过用`while`或`for`语句能实现对程序的控制, 其选择经常靠感觉。`for`循环的一个主要优势是能在开始进行控制和定位。嵌套的`for`循环便于阅读代码, 下一节中的程序要说明这一点。

### 3.11 问题求解: 组合数学

我们要考虑一个组合数学领域中的问题, 即列举组合和排列。这个问题是列出所有的非

负整数的三元组，每组数的和应该是一个给定的数，例如7。如下的程序能满足这个要求。

```
/* Find triples of integers that add up to N. */
#include <stdio.h>

#define N 7

int main(void)
{
    int cnt = 0, i, j, k, n;

    for (i = 0; i <= N; ++i)
        for (j = 0; j <= N; ++j)
            for (k = 0; k <= N; ++k)
                if (i + j + k == N) {
                    ++cnt;
                    printf("%3d%3d%3d\n", i, j, k);
                }
    printf("\nCount: %d\n", cnt);
    return 0;
}
```

当执行这个程序时，在屏幕上会输出如下信息：

```
0 0 7
0 1 6
0 2 5
0 3 4
.....
6 0 1
6 1 0
7 0 0
```

Count: 36

#### 对程序add\_to\_n的解析

- #define N 7

为了能更容易地进行我们的实验，我们要使用符号常量N。

- for (i = 0; i <= N; ++i)  
.....

最外层的for循环的语句部分仍是一个for循环，同样第二个for循环的语句部分还是一个for循环。我们也可以把第一个for循环写为

```
for (i = 0; i <= N; ++i) {
    .....
```

然而由于这个for语句的体是另一个for语句，这个花括号不是必须的。

- for (j = 0; j <= N; ++j)
 for (k = 0; k <= N; ++k)
 .....

这是最外层的for循环的语句部分或体。对于每个最外层的i的值，在内层循环中的各个j值都循环一遍。对于每个j的值，在最内层循环中的各个k值都循环一遍。这与里程表是类似的，在里程表的单位较大的数字变化之前，单位较小的数字先循环一遍。

- if (i + j + k == N) {
 ++cnt;
 printf("%3d%3d%3d\n", i, j, k);
 }

这是最内部循环的体，它将检查 $i+j+k==N$ 是否成立。如果成立，对`cnt`加1，并显示一该整数三元组。 ■

### 3.12 问题求解：布尔变量

布尔代数在计算机电路中扮演了重要的角色。在布尔代数中，所有变量的值不是0就是1。晶体管和存储器技术用电流、电压和磁定位实现了0-1值模式。通常电路设计师计划功能，并检查所有可能的0-1输入/输出是否合乎要求。

我们用`int`型变量`b1`, `b2`, ..., `b5`表示5个布尔变量，它们的值仅能取0或1。这些变量的布尔函数的返回值只能是0或1。一个典型的布尔函数的例子是多数逻辑函数：如果多数变量的值都是1，则它的返回值是1，否则返回值是0。我们要为函数`b1||b3||b5`、函数`b1&&b2||b4&&b5`和多数逻辑函数创建一张表。要记住逻辑表达式的值总是`int`型的0或1。

```
/* Print a table of values for some boolean functions. */
#include <stdio.h>

int main(void)
{
    int b1, b2, b3, b4, b5;          /* boolean variables */
    int cnt = 0;
    printf("\n%5s%5s%5s%5s%5s%7s%7s%11s\n\n", /* headings */
        "Cnt", "b1", "b2", "b3", "b4", "b5",
        "fct1", "fct2", "majority");
    for (b1 = 0; b1 <= 1; ++b1)
        for (b2 = 0; b2 <= 1; ++b2)
            for (b3 = 0; b3 <= 1; ++b3)
                for (b4 = 0; b4 <= 1; ++b4)
                    for (b5 = 0; b5 <= 1; ++b5)
                        printf("%5d%5d%5d%5d%5d%6d%7d%9d\n",
                            ++cnt, b1, b2, b3, b4, b5,
                            b1 || b3 || b5, b1 && b2 || b4 && b5,
                            b1 + b2 + b3 + b4 + b5 >= 3);
    printf("\n");
    return 0;
}
```

这个程序说明了嵌套`for`循环的典型用法。它的输出是一张关于所有可能的输入及对应的输出的值表。电路设计师能用这张表检查布尔函数是否符合要求。如下是该程序的一些输出：

Cnt	b1	b2	b3	b4	b5	fct1	fct2	majority
1	0	0	0	0	0	0	0	0
2	0	0	0	0	1	1	0	0
3	0	0	0	1	0	0	0	0
....								

### 3.13 逗号运算符

在C的所有运算符中，逗号运算符的优先级最低。它是一个用表达式作为操作数的二元运算符，它遵循从左到右的结合性规则。在形式为

`expr1, expr2`

的逗号表达式中，先计算`expr1`，然后再计算`expr2`。逗号表达式作为一个整体有一个与其右操作数相同的值和类型。如下是一个例子：

`a = 0, b = 1`

如果声明了b的类型为int, 那么该表达式的值为1, 且类型为int。

有时在for语句中使用逗号运算符。这使得可进行多次初始化。例如, 代码

```
for (sum = 0, i = 1; i <= n; ++i)
    sum += i;
```

能用于计算从1到n的整数之和。进一步地运用这个思想, 我们可以把整个for循环体填充在for的括号内。可把上述的代码重新编写为

```
for (sum = 0, i = 1; i <= n; sum += i, ++i)
    ;
```

但如下的写法是错误的:

```
for (sum = 0, i = 1; i <= n; ++i, sum += i)
    ;
```

在逗号表达式

```
++i, sum += i
```

中, 先计算++i, 再把增量的值赋给sum。

下表给出了一些逗号表达式的例子。

声明和初始化		
<pre>int    i, j, k = 3; double x = 3.3;</pre>		
表达式	等价表达式	值
<code>i = 1, j = 2, ++k + 1</code>	<code>((i = 1), (j = 2)), ((++k) + 1)</code>	5
<code>k != 7, ++x * 2.0 + 1</code>	<code>(k != 7), (((++x) * 2.0) + 1)</code>	9.6

程序中的大多数逗号并不代表逗号运算符。例如, 用于分隔函数参数列表的逗号或初始化程序列表中的逗号都不是逗号运算符。如果在这些地方使用逗号运算符, 必须用括号把出现逗号运算符的逗号表达式括起来。

### 3.14 do语句

可以把do语句看作是while语句的变种。do语句不在循环的开始处进行测试, 而是在结尾处。如下是一个例子:

```
do {
    sum += i;
    scanf("%d", &i);
} while (i > 0);
```

考虑如下的结构形式:

```
do
    statement
while (expr);
next statement
```

首先, 执行statement和expr。如果expr的值不是0(true), 那么控制回递到do语句的开始处, 再重复执行自身; 如果expr的值是0(false), 那么控制传递到next statement。

作为一个例子, 假设我们要读入一个整数, 并要保证整数始终是正的, 下边的代码实现

了这一功能。

```
do {
    printf("Input a positive integer: ");
    scanf("%d", &n);
    if (error = (n <= 0))
        printf("\nERROR: Negative value not allowed!\n\n");
} while (error);
```

只要输入负整数，就通知用户应该输入正整数。只有在输入正整数后，才退出循环。

### 3.15 goto语句

大多数现代的编程方法都认为goto语句是有害的结构，它会引起一个无条件的跳转，跳转到的位置是当前函数中的一个带有标号的语句，这样它会破坏由其他控制流机制（if、if-else、for、while、do和switch）提供的有益结构。

由于goto跳转到带有标号的语句，我们需要先讨论这种结构。一个有标号的语句的形式为：

```
label: statement
```

这里label是一个标识符。如下是一些带有标号的语句的例子：

```
bye: exit(1);
L444: a = b + c;
bug1: bug2: bug3: printf("bug found\n"); /* multiple labels */
```

但下面不是带标号的语句：

```
333: a = b + c; /* 333 is not an identifier */
```

标号标识符有自己的命名空间，这意味着相同的标识符可用于标号和变量。然而，这种做法被认为是不良的编程风格，应该避免。

通过执行如下形式的goto语句，控制就无条件地转到带有标号的语句。

```
goto label;
```

下面是一个例子：

```
goto error;
.....
error: {
    printf("An error has occurred - bye!\n");
    exit(1);
}
```

goto语句与它对应的带有标号的语句必须在同一个函数内。下面是一个使用goto语句的更具体的代码片段：

```
while (scanf("%lf", &x) == 1) {
    if (x < 0.0)
        goto negative_alert;
    printf("%f %f %f\n", x, sqrt(x), sqrt(2 * x));
}

negative_alert:
    if (x < 0.0)
        printf("Negative value encountered!\n");
```

可以按很多方法而不用goto重新编写这段代码。下面是一种方法：

```
while (scanf("%lf", &x) == 1 && x >= 0.0)
    printf("%f %f %f\n", x, sqrt(x), sqrt(2 * x));
```

```
if (x < 0.0)
    printf("Negative value encountered!\n");
```

一般而言，应该避免使用goto。它是改变控制流的最初方法，在具有较丰富结构的语言中它是不需要的。使用带有标号的语句和goto是增量的拼凑式的程序设计的特点。如果程序员用goto附加代码碎片，以此来修改程序，马上会使程序的可理解性变差。

什么时候应该使用goto呢？回答是简单的：永远不。的确，遵循这个劝告，就不会犯错误。然而在一些极端的情况下，用goto能使程序更有效，但此时应该仔细地写好文档。在另一些情况，它能简化控制流。例如，如果程序控制要从深层嵌套的内部循环中跳转到函数的最外层，就有可能使用goto。

### 3.16 break和continue语句

break和continue这两个特殊的语句会中断正常的控制流。break语句退出最内层的循环或switch语句（在下一节讨论）。在下面的例子中，将测试一个数是否是负数，如果是负数，就用break语句把控制转给紧跟着循环的语句。

```
while (1) {
    scanf("%lf", &x);
    if (x < 0.0)
        break;          /* exit loop if x is negative */
    printf("%f\n", sqrt(x));
}

/* break jumps to here */
```

这是对break的一个典型应用。这是一个无限循环，它用if表达式测试给定的条件来终止循环。

continue语句使得当前循环停止，并立即进入下一次循环。

```
while (cnt < n) {
    scanf("%lf", &x);
    if (x > -0.01 && x < +0.01)
        continue;      /* disregard small values */
    ++cnt;
    sum += x;
    /* continue transfers control here to begin next iteration */
}
```

continue语句仅出现在for、while和do中。像例子所示的那样，continue把控制转到当前循环的尾部，而break终止循环。

在有continue参与的情况下，如下形式的for循环

```
for (expr1; expr2; expr3) {
    statements
    continue;
    more statements
}
```

与

```
expr1;
while (expr2) {
    statements
    goto next;
    more statements
next:
    expr3;
}
```

是等价的，但与

```
expr1;
while (expr2) {
    statements
    continue;
    more statements
    expr3;
}
```

是不同的。

关于测试它们的常规方法，请参见练习25。

### 3.17 switch语句

switch语句是一种概括了if-else语句的多路条件语句，下面是switch语句的一个典型示例：

```
switch (val) {
case 1:
    ++a_cnt;
    break;
case 2:
case 3:
    ++b_cnt;
    break;
default:
    ++other_cnt;
}
```

注意，例子中switch语句体是一个复合语句，在几乎大多数退化的情形中都是如此。紧跟在关键字后边的括号中的控制表达式必须是整数类型的（请参见第6章“基本数据类型”）。在本例中，变量val的类型正是int。在计算表达式后，控制跳到相应的case标号处。case标号后面的常量整型表达式必须都是惟一的。通常在case（第1个除外）或default标号前面的最后一个语句是break语句。如果没有任何break语句，就会误执行后续的case中的语句。使用switch语句常犯的一个错误是遗失了break语句。在一个switch语句中至少要有default标号。通常default标号最后出现，尽管它可出现在任何地方。关键字case和default不能出现在switch的外面。

#### switch的作用

- 1) 对switch表达式求值。
- 2) 跳到与第1步中的表达式的值相匹配的标号（标号是一个常数值）处。如果不能匹配，跳到default标号；如果没有default标号，则终止switch。
- 3) 当遇到break语句时，终止switch，或转到结束。

让我们回顾一下我们能使用的各种跳转语句，它们是goto、break、continue和return。对goto的使用没有什么限制，但作为一种危险结构，要避免使用它；break语句可用于循环中，对于switch语句的正确构成来说，它是很重要的；continue语句必须在循环中使用，但有时它不是必须的；return语句必须用于具有返回值的函数中。在第4章“函数和结构化编程”中将会讨论它。

### 3.18 嵌套的控制流

像if、for、while和switch语句这样的控制流语句经常要在自身内嵌套或相互嵌套。



虽然这样的嵌套控制结构可能是相当复杂的，但其中的一些有正规的结构，并且很容易理解。

最常见的嵌套控制流结构之一是if-else的重复使用。下面是它的一般形式：

```
if (expr1)
    statement1
else if (expr2)
    statement2
else if (expr3)
    statement3
...
else if (exprN)
    statementN
else
    default statement
next statement
```

这个巨大的结构（除了next statement）是一个单if-else语句。例如，假设expr1和expr2都是0(false)，expr3不是(true)。在这种情况下，跳过statement1和statement2，执行statement3；然后控制传递到next statement，其他的语句都没有被执行。如果我们假设所有的表达式都是0，就仅执行default statement。在一些情况下，不需要执行default statement。在这样的情况下，要用上述的结构，只是没有用到

```
else
    default statement
```

**编程提示** 如果你在这个巨大的if-else结构的顶部放置很合理的case语句，你的代码就会更有效，这是因为在控制转到结构的外部之前计算的表达式要少。

### 3.19 条件运算符

条件运算符?:是不常用的，因为它是一个三元运算符，它需要三个表达式作为运算数。条件运算符的一般形式是：

*expr1 ? expr2 : expr3*

首先计算expr1。如果它的值不是0(true)，那么就计算expr2，expr2的值就是该表达式的值，如果expr1的值是0(false)，那么就计算expr3，expr3的值就是该表达式的值。这样，条件表达式能用于完成if-else语句的工作。例如，考虑如下代码：

```
if (y < z)
    x = y;
else
    x = z;
```

该段代码的作用是把y和z的最小值赋给x。这也可以用下述语句完成：

*x = (y < z) ? y : z;*

这个语句中的括号是不必要的，这是因为条件运算符的优先级高于赋值运算符。尽管如此，用括号还是有助于弄清楚测试的是什么。

条件表达式

*expr1 ? expr2 : expr3*

的类型由expr2和expr3决定。如果它们的类型是不同的，通常使用转换规则（请参见第6章“基本数据类型”）。要仔细地注意到，条件表达式的类型不依赖对哪个表达式（expr2或expr3）

求值。

条件运算符?:的优先级高于赋值运算符,它遵从从右到左的结合性规则。下表说明了如何对条件表达式求值。

声明和初始化			
<pre>int    a = 1, b = 2; double x = 7.07;</pre>			
表达式	等价表达式	值	类型
<code>a == b ? a - 1 : b + 1</code>	<code>(a == b) ? (a - 1) : (b + 1)</code>	3	int
<code>a - b &lt; 0 ? x : a + b</code>	<code>((a - b) &lt; 0) ? x : (a + b)</code>	7.07	double
<code>a - b &gt; 0 ? x : a + b</code>	<code>((a - b) &gt; 0) ? x : (a + b)</code>	3.0	double

### 3.20 风格

本书中,我们采用贝尔实验室的工业编程风格,这种风格具体包含4个关键特征:

#### 贝尔实验室工业编程风格

- 1) 在可能之处遵从英语的标准规则,例如在逗号之后放一个空格。
- 2) 在每个二元运算符的两边各放一个空格,以便于阅读。
- 3) 要一致地缩进编写代码,以指明控制流。
- 4) 如下例所示的那样使用花括号:

```
for (i = 0; i < n; ++i) {
    .....
```

反应了控制流的花括号的位置对良好的编程风格来讲是非常重要的。在循环体内的语句要缩进三个空格,结束的花括号}要直接地放在for下。缩进在可视化方面使得组成循环体的语句更加美观。}和在其上边的for的位置分别用来可视化地标记循环的开始和结束。这种风格的一个变体是选择不同的缩进空格数,如2个、4个或8个空格。

另一种编程风格是在同一列放置开始和结束的花括号。下面是一个例子:

```
while (i < 10)
{
    .....
```

有时把这种风格称为学生风格(student style)。你使用哪一种风格是根据你的爱好,在一定的程度上也靠编辑器的能力。采用贝尔实验室的工业编程风格的程序员倾向于使用功能强的编辑器,这样的编辑器能找到相匹配的花括号或括号,能把多行代码作为一个单位进行缩进以及有自动缩进的功能等等。例如,UNIX中的emacs和vi编辑器就有这样的功能。

如果循环体是一个单语句,就不需要花括号。虽然如此,作为一种风格,一些程序员总是使用花括号。

```
while (i < 10) {
    a single statement
}
```

这是一种可接受的做法。无论是否有多余的花括号，编译器产生的可执行代码都同样有效。

在C中很少使用do循环。在使用do循环时，人们认为良好的编程风格应该使用花括号，即使是不需要的情况下。在下面结构中的花括号

```
do {
    a single statement
} while (.....);
```

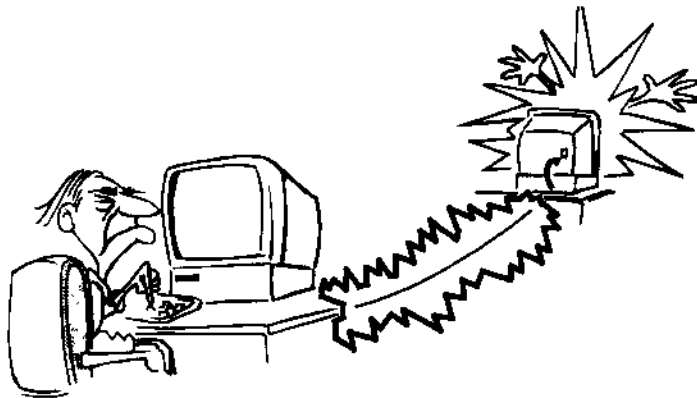
使得读者能够较容易地认出这是一个do语句，而不是后跟一个空语句的while语句。

在很多情形中，可以使用while循环，也可以使用for循环。然而，如果要在顶部使控制和索引可见，就应该用for循环，否则就靠个人感觉进行选择。

下面的if-else嵌套风格不受推荐，事实上也很少使用：

```
if (expr1) /* wrong style */
    statement
else if (expr2)
    statement2
else if (expr3)
    statement3
else if (expr3)
    statement3
.....
```

使得if-else嵌套满屏幕的风格任何变体都是不可接受的，因为这样的长链会超出屏幕空间。



“我告诉你，我们喜欢在程序中使用精致的缩进，对代码进行良好的注释，这样人们理解控制流就容易得多。我们不使用GOTO语句，也不编写意大利细面条式的代码。”

### 3.21 常见的编程错误

我们要讨论一些常见的错误。首先我们要注意表达式  $a == b$  和  $a = b$  间的可能混淆之处。虽然它们看起来相似，但它们在功能上是根本不同的，表达式  $a == b$  用于相等测试，而  $a = b$  是一个赋值表达式。最常见的一个错误是用

```
if (k = 1)
    .....
```

代替

```
if (k == 1)
    .....
```

由于k的值是1，赋值表达式 $k=1$ 总为true。一些编译器会对此给出警告，而有些编译器不会给出警告。如果没有警告，像这样的错误是很难发现的。请注意，如果我们写

```
if (1 = k)
    .....
```

那么编译器肯定要向我们发出关于这个错误的警告，把一个值赋给一个常量是非法的。由于这个原因，一些程序员通常编写如下形式的测试：

```
constant == expr
```

这能防止犯用=代替==的错误。这种风格的一个缺点是，它不遵从我们平常的思考方法，也就是说，如果k等于1，那么……。

第二个常见的编程错误是控制循环语句的表达式引起一个不希望的无限循环。应该留心，以避免这种事情的发生。例如，考虑如下代码：

```
printf("Input an integer: ");
scanf("%d", &n);
while (--n) {
    .....
}
```

该段程序的用意是输入一个正整数，把输入的值存储在n中，重复地执行while的循环体，直到表达式n的值最终变为0。然而，如果不注意把一个负数赋给n，循环就变成了无限循环。为了防止这种可能性，程序员应该使用如下的代码：

```
while (--n > 0) {
    .....
}
```

第三个常见的编程错误涉及到在if、while或for后使用多余的分号。例如，考虑

```
for (i = 1; i <= 10; ++i);
    sum += i;
```

这样的代码的行为会与所期望的不一致，因为在第一行末尾的分号会产生一个多余的空语句。此段代码与下述的代码是等价的：

```
for (i = 1; i <= 10; ++i)
    ;
sum += i;
```

很显然这不是程序员的用意。这种类型的缺陷是很难发现的。

下面我们要讨论误用关系运算符如何导致不期望的结果。回想一下，数学表达式 $2 < k < 7$ 意味着k大于2，小于7。依据k的值，我们也能把它看作是k为true还是为false的断言。例如，如果k是8，那么断言就为false。与此相对照，考虑如下代码：

```
int k = 8;
if (2 < k < 7)
    printf("true");          /* true is printed! */
else
    printf("false");
```

表达式是true的原因是显然的。因为关系运算符遵从从左到右的结合性规则， $2 < k < 7$ 与 $(2 < k) < 7$ 是等价的。由于 $2 < k$ 是true，它的值是1。这样， $(2 < k) < 7$ 等价于 $1 < 7$ ，它明显是true。编写 $2 < k$ 且 $k < 7$ 测试的正确方法是 $2 < k \ \&\& \ k < 7$ ，它与 $(2 < k) \ \&\& \ (k < 7)$ 等价，这是由

于<的优先级高于&&。这个表达式在整体上为true，当且仅当&&的两个运算数都为true。

最后一个常见的错误涉及到超出大多数机器精度的等式测试。如下是说明这点的一个程序：

```
/* An equality test that fails. */
#include <stdio.h>

int main(void)
{
    int    cnt = 0;
    double sum = 0.0, x;

    for (x = 0.0; x != 9.9; x += 0.1) {      /* trouble! */
        sum += x;
        printf("cnt = %5d\n", ++cnt);
    }
    printf("sum = %f\n", sum);
    return 0;
}
```

在数学上，如果x开始是0，并重复地加0.1，它最终的值是9.9。程序员的用意是，当x为9.9时，程序控制for循环退出。然而，如果测试 $x == 9.9$ 超出了机器的精度范围，那么表达式 $x != 9.9$ 总为true，程序陷入无限循环。程序员必须要记住，在任何一台机器上，浮点运算是不精确的，这与整数运算不一样。为了纠正这个问题，我们可以用 $x <= 9.9$ 代替 $x != 9.9$ （请参见练习27）。

如果合适，良好的编程风格使用关系表达式而不是等式表达式控制循环以及if或if-else语句。一般而言，用这种风格产生的代码更健壮。

### 3.22 系统考虑

硬件和操作系统决定了在机器内怎样表示数。我们要讨论由于浮点数不能用无限精度表示而引起的问题，有时，这能引起料想不到的结果。在数学中关系 $x < x + y$ 等价于 $0 < y$ 。在数学上，如果y是正的，那么这两个关系在逻辑上都是真的。在计算上，如果x是一个具有很大的浮点数，如 $7 \times 10^{33}$ ，y是一个具有很小值的浮点数，如0.0001，那么关系表达式 $x < x + y$ 可能是false，尽管在数学上是true。该式的一个等价的表达式是 $(x - (x + y)) < 0.0$ ，机器上执行的正是这个表达式。按照机器精度，如果x和 $x + y$ 等值，则该表达式产生的值是整数0（请参见练习6）。

下面我们讨论无限循环。有时一不注意就会出现无限循环，但在交互式环境中，程序员可能要故意地使用无限循环。如果是这样，用户必须中断程序以停止无限循环。要中断程序所用的输入是与系统相关的，在MS-DOS和UNIX中，通常用Ctrl+c终止程序，然而，在其他的操作系统中可能不是这样。无限循环的两个常规的格式如下：

```
while (1) {
    .....
}
```

和

```
for ( ; ; ) {
    .....
}
```

让我们假设程序员要做一个实验。与一遍又一遍地运行程序相比，把基本代码放入无限

循环中可能更方便。例如：

```
printf("Sums from 1 to n will be computed.\n\n");
for ( ; ; ) {
    printf("Input n:  ");
    scanf("%d", &n);
    sum = 0;
    for (i = 1; i <= n; ++i)
        sum += i;
    printf("sum = %d\n\n", sum);
}
```

### 3.23 转向C++

C中的语句类型和控制流在C++中保持不变。已经提到的一个扩展是，在C++中声明是语句，并可与可执行的语句相混合。在循环结构中这是很常见的。

```
for (int i = 1, sum = 0; i <= n; ++ i)
    sum += i;
```

在这个for语句中，给出了对sum和i的声明和初始化。变量sum和i在其余的块中继续存在，正像在main()的开头声明它们一样。

ANSI C现在允许布尔类型bool，这样可以计算出布尔表达式的true和false。类型bool在传统的C中是没有的。这几年中，曾经用了很多模式实现布尔类型，例如：

```
typedef int boolean;
#define true 1
#define false 0
```

或

```
enum boolean { false, true };
```

类型bool的使用排除了这些不一致。下面是声明的例子：

```
bool my_turn = false, your_true;
bool* p = &my_turn;
```

类型bool可以由关系、逻辑和等式表达式返回的类型。bool常量true和false分别用1和0表示。把非零值指派为true，0指派为false。

### 小结

- 关系表达式、等式表达式和逻辑表达式的值是类型为int的0或1，这些表达式主要用于测试影响控制流的数据。
- 除了非运算符!是一元的之外，关系、等式和逻辑运算符都是二元的。像!a这样的非表达式的值是类型为int的0或1。通常!!a的值和a不同。
- 成组结构{……}是一个复合语句。它把括起来的语句作为一个单元处理。
- if语句提供了选择是否执行一个语句的手段。if-else语句提供了从两个语句中选择执行哪一个语句的手段。if-else语句的else部分属于最靠近它的if，这解决了else归属不清的问题。
- 我们使用计算机的一个原因是要重复地执行一定的行为。在C中while、for和do语句提供了循环机制。while或for语句的循环体要被执行0次或多次。do语句的循环体要被执行1次或多次。

- 程序员经常必须要在是使用for语句还是使用while语句之间做出选择。若需要在循环的顶部使控制和索引可见，则自然地要选择for语句。
- 偶尔在for语句中使用逗号运算符。在C中的所有运算符中，逗号运算符的优先级最低。
- 如下四种类型的语句会引起控制流无条件地转换：

go      break      continue      return

除了在switch中使用break外，应该尽量少使用它们。

- 避免用goto。对于良好的编程来说，它是有害的。
- switch语句提供了多路条件分支。当有很多的特殊情况要处理时，它是有用的。通常，switch中需要break。

## 练习

1. 不用非运算符，给出等价的逻辑表达式。

```
!(a > b)
!(a <= b && c <= d)
!(a + 1 == b + 1)
!(a < 1 || b < 2 && c < 3)
```

2. 完成下表。

声明和初始化		
int      a = 1, b = 2, c = 3; double   x = 1.0;		
表达式	等价表达式	值
a > b && c < d		
a < ! b    ! ! a		
a + b < ! c + c		
a - x    b * c && b / a		

3. 编写一个含有如下循环的程序。

```
while (scanf("lf", &salary) == 1) {
    .....
}
```

在while循环体内，计算23%的联邦代扣税款和7%的州代扣税款，把它们与相应的工资一起显示出来，计算被显示的所有工资和税金的累加和。在程序退出while循环后显示这些累加和。

4. 当执行如下的代码时，会显示什么？

```
int      a = 1, b = 2, c = 3;
float   x = 3.3, y = 5.5;

printf("%d %d\n", ! a+b/c, ! a + b / c);
printf("%d %d\n", a == -b + c, a * b > c == a);
printf("%d %d\n", !!x < a + b + c, !!x + !!!y);
printf("%d %d\n", a || b == x && y, !(x || !y));
```

5. 假设程序员正在处理一个需要特殊行为的程序，程序中的变量k的值是类型为int的7。考虑如下的代码：

```

while (k = 7) {
    ..... /* do something */
    k = 0; /* finished, exit the loop */
}

```

把这段代码与下述的代码进行对照：

```

if (k = 7) {
    ..... /* do something */
}

```

这两段代码在逻辑上是错误的。其中的一个产生的运行效果很明显，据此能很容易地找到错误，而另一个产生的运行效果很微妙，很难找到错误。请解释原因。

6. 下面的代码是与系统相关的，尽管如此，大多数机器会产生在逻辑上是错误的结果。首先解释在逻辑上应该显示什么，然后在你的机器上测试代码，看实际显示出什么。

```

double x = 1e+33, y = 0.001;
printf("%d\n", x + y > x - y);

```

如果你把1000赋值给y，会发生什么？如何把1 000 000赋值给y，又会发生什么？本练习是要强调浮点运算不如数学运算那么精确。

7. 下面的程序显示的结果是什么？请解释。

```

int i = 7, j = 7;
if (i == 1)
    if (j == 2)
        printf("%d\n", i = i + j);
else
    printf("%d\n", i = i - j);
printf("%d\n", i);

```

8. 下面代码段中的语法错误实际上没有出现在注释所对应的行。运行一个包含该代码段的程序，找出那一行标记有语法错误。请解释为什么。

```

while (++i < LIMIT) do { /* syntax error */
    j = 2 * i + 3;
    printf("j = %d\n", j);
}

/* Many other languages require "do", but not C. */

```

9. 在下面的代码中，假设i和j的值在循环体内不变。这些代码能导致无限循环吗？请解释。

```

printf("Input two integers: ");
scanf("%d%d", &i, &j);
while (i * j < 0 && ++i != 7 && j++ != 9) {
    ..... /* do something */
}

```

10. 编写一个程序，把一个整数值读到n中，如果n不是负数，那么从n到2\*n做累加；如果n是负数，那么从2\*n到n做累加。以两个版本编写程序：一个版本仅用for循环，另一个版本仅用while循环。

11. 除非中断，否则下列的代码要在屏幕上重复地显示True forever!。(在MS-DOS和UNIX中，用Ctrl+c影响中断。)

```

while (1)
    printf(" True forever! ");

```

编写一个简单的程序实现它，但要用for语句代替while语句，并用一个空语句作为for语句的循环体。



12. 我们已经解释了

```
while (1) {
    ....
}
```

是一个无限循环。下列代码完成什么？如果你不是很确定，就请试一试它。也就是，编写一个包含该语句的程序并运行它。

```
while (-33.777)
    printf("run forever, if you can");
```

13.  $a$ 和 $b$ 是两个数学表达式。我们要证明 $a != b$ 等价于 $!(a == b)$ 。请通过完成下表做到这一点。

数学表达式的值			
$a+b$	$a!=b$	$a==b$	$!(a==b)$
零			
非零			

14. 运行程序find\_max，当出现提示时输入1。你会在屏幕上看到

Enter 1 real numbers:

当然，这是不正确的英文。改变该程序，如果 $n$ 的值是1，就显示number，否则显示numbers。

15. 假设你厌恶偶整数而喜欢奇整数。修改程序find\_max，使得它仅处理奇整数，其中的所有变量的类型都是int。当然，你必须用适当的printf()语句向用户解释这些。

16. 如果你在你的系统上运行下列代码，发生什么情况？如果运行不正确，就修改它。

```
double    sum = 0.0, x;

printf("%5s%15s\n", "Value", "Running sum");
printf("%5s%15s\n", "-----", "-----");
for (x = 0.0; x != 9.9; x += 0.1) { /*test not robust*/
    sum += x;
    printf("%5.1f%15.1f\n", x, sum);
}
```

17. 编程初学者有时会弄混用于控制for循环的表达式的次序。在下面的代码中，试图累加从1到5的整数。这样做的结果如何？首先进行手工模拟，看结果如何，然后编写一个测试程序，看你做得是否正确。

```
int    i, sum = 0;
for (i = 1; ++i; i <= 5)
    printf("i = %d    sum = %d\n", i, sum += i);
```

18. 编写一个交互式程序，请求用户提供3个整数 $k$ 、 $m$ 和 $n$ ， $k$ 要大于1。程序要计算在 $m$ 和 $n$ 之间的所有能被 $k$ 除的整数之和。

19. C++：用C++编写上一个练习。

20. 本练习是要演习短路求值。

```
int    a = 0, b = 0, x;

x = 0 && (a = b = 777);
printf("%d %d %d\n", a, b, x);
```

```
x = 777 || (a = ++b);
printf("%d %d %d\n", a, b, x);
```

这段程序将显示什么？首先写下你的答案，然后编写一个测试程序，进行检验。

21. C++：用C++编写上一个练习。

22. 完成下表。

声明和初始化		
int a = 1, b = 2, c = 3;		
表达式	等价表达式	值
a && b && c	(a && b) && c	
a && b    c		
a    b && c		
a    ! b && ! ! c + 4		
a += ! b && c == ! 5		

23. 逻辑表达式的语义暗示着在一些计算中求值的次序是至关重要的。下面哪一个表达式最可能是正确的？请解释。

(a) if ((x != 0.0) && ((z - x) / x \* x < 2.0))  
.....

(b) if (((z - x) / x \* x < 2.0) && (x != 0.0))  
.....

24. 用于布尔函数的真值表(truth table)是由其中的所有变量的值和相应的布尔函数本身的值组成的表。在前面，我们为多数函数和两个其他的函数创建了一个真值表（请参见3.12节“问题求解：布尔变量”）。为下面的函数各创建一个真值表：

b1 || b2 || b3 || b4                      !(b1 || b2) && (!b3 || b4)

在真值表中用T和F分别代表true和false。提示：用#define预处理指令定义BOOLEX。然后再编写一个处理任意的BOOLEX的程序。

25. 下面是测试for循环体中continue语句作用的一个简单方法。它将显示出什么？

```
for (putchar('1'); putchar('2'); putchar('3')) {
    putchar('4');
    continue;
    putchar('5');
}
```

26. 可以用下面的条件表达式表达数学运算min(x, y)：

(x < y) ? x : y

以类似的方式，仅用条件表达式描述数学运算：

min(x, y, z)    和    max(x, y, z, w)

27. 在3.21节“常见的编程错误”中给出的不健壮的等式测试程序会在你的机器上导致无限循环吗？如果会，像建议的那样修改它，再执行它。它产生的答案与数学计算接近吗？如果x在循环中增加，x == 9.9总不是true，那么或许答案的偏离量约是0.1。

28. 在形式为

*expr1* , *expr2*

的逗号表达式中, 首先计算*expr1*, 然后计算*expr2*, 作为一个整体逗号表达式具有一个与它的右操作数相同的值和类型。如下是逗号表达式的一个例子:

```
a = 1, b = 2
```

如果在声明中*b*的类型为*int*, 那么这个逗号表达式的值是2, 其类型为*int*。下面是一个显示偶数列和奇数列的*for*循环。哪一个逗号是逗号运算符? 哪一个不是逗号运算符? 编写一个不使用逗号运算符的程序。

```
int i, j;

for (i = 0, j = 1; i < LIMIT; i += 2, j += 2)
    printf("%12d%12d\n", i, j);
```

29. C++: C++现在允许使用布尔类型。这可以用整型常量在还没实现布尔类型的编译器上进行模拟。

```
const int true = 1;
const int false = 0;
```

输出语句

```
cout << (expression)? "true\t" : "false\t" ;
```

显示布尔值。用这种形式的I/O编写一个程序, 完成练习20。

30. 下面的Java程序计算未定数目的类型为*double*的数的平均值。该程序用标记值0停止用于累加类型为*double*的读人数的循环, 请把这个程序转换成用C编写的程序。

```
// Average.java - compute average of input values -Java by
Dissection page 74.
import tio.*;

public class Average {
    public static void main(String[] args) {
        double number;
        int count = 0;
        double runningTotal = 0;

        // initialization before first loop iteration
        System.out.println("Type some numbers, " +
            "the last one being 0");
        number = Console.in.readDouble();

        while (number != 0) {
            runningTotal = runningTotal + number;
            count = count + 1;
            // prepare for next iteration
            number = Console.in.readDouble();
        }
        System.out.print("The average of the ");
        System.out.print(count);
        System.out.print(" numbers is ");
        System.out.println(runningTotal/count);
    }
}
```

如同其他的像简单的C程序的Java程序一样, 在类中有一个*main()*方法。程序的主要区别在于对不同I/O模式的使用。

31. 下面的Java程序说明了在Java和C之间的一些其他的不同。请把它转换成用C编写的程序。

```
// BreakContinue.java - example of break and continue
//Java by Dissection page 80
import tio.*;

class BreakContinue {
    public static void main(String[] args) {
        int n;

        while (true) {           //seemingly an infinite loop
            System.out.print("Enter a positive integer ");
            System.out.print("or 0 to exit:");
            n = Console.in.readInt();
            if (n == 0)
                break;           // exit loop if n is 0
            if (n < 0)
                continue;        //wrong value
            System.out.print("squareroot of " + n);
            System.out.println(" = " + Math.sqrt(n));
            //continue land here at end of current iteration
        }
        //break lands here
        System.out.println("a zero was entered");
    }
}
```

Java和C之间的不同是Java有布尔类型。if或while语句的控制表达式是布尔表达式，这意味着像“if (n = 0)”这样的错误在语法上由Java编译器捕捉。Java编译器会发出警告：这是一个类型为int的表达式，不能用作if的测试表达式；在C中，由于该表达式意味着把0赋值给n，返回值是0(false)，所以这不是一个语法错误。另外，在Java中不允许使用goto。在C中是允许的，但这是一种不良做法，不鼓励使用。

## 第4章 函数和结构化编程

结构化编程是一种解决问题的策略和包括如下标准的一种编程方法：

### 结构化编程标准

- 1) 程序中的控制流应该尽可能简单。
- 2) 应该自顶向下地设计程序结构。

自顶向下设计(top-down design), 也称为逐步细化(stepwise refinement), 由把一个问题分解成几个小问题的重复分解组成。最终, 得到一个由容易编码的小问题或任务组成的集合。

在C中, 用函数编写解决由分解而来的小问题的代码。这些函数要和其他函数结合, 最终用在解决最初问题的main()中。C中提供的函数机制完成明确的编程任务。像printf()和scanf()这样的一些函数由系统提供, 其他函数可由程序员编写。

在本章中我们要说明结构化编程和自顶向下设计, 但首先我们要描述函数机制。

### 4.1 函数调用

程序由一个或多个函数组成, 其中的一个是main(), 程序的执行总是从main()开始。当程序控制遇到一个后跟有括号的函数名时, 就调用或请求(call或invoke)函数。这意味着要把程序控制传递给函数。在函数完成工作后, 程序控制又返回到调用环境, 程序接着继续执行。作为一个简单的例子, 考虑如下的显示消息的程序:

```
#include <stdio.h>

void prn_message(void);          /* fct prototype */

int main(void)
{
    prn_message();               /* fct invocation */
}

void prn_message(void)           /* fct definition */
{
    printf("A message for you: ");
    printf("Have a nice day!\n");
}
```

在文件的顶部有#include预处理指令, 紧接着是prn\_message()的函数原型, 该函数原型告诉编译器该函数没有参数, 且不向调用环境返回值。程序从main()开始执行, 当程序控制遇到prn\_message()时就调用它, 并把程序控制传递给它。在执行完prn\_message()中的printf()语句后, 程序控制又返回到调用环境, 在本例中返回到main()。由于在main()中没有更多的工作可做, 程序结束。虽然函数prn\_message()没有向调用环境返回值, 但像我们将会看到的那样, 很多函数是有返回值的。

### 4.2 函数定义

把描述函数做什么的C代码称为函数定义(function definition)。函数定义的一般形式为:

```

type function_name ( parameter type list )
{
    declarations
    statements
}

```

在第一个花括号前的文字构成了函数定义的头(header)，在花括号中间的文字构成了函数定义的体(body)。函数的类型(type)就是函数的返回值类型，如果没有返回值，就使用关键字void。参数类型表(parameter type list)描述了在调用函数时传递给函数的参数的个数和类型，如果不传递参数，就使用关键字void。

在参数类型表中的参数是标识符，这些参数能用在函数体中。有时把函数定义中的参数称为形式参数(formal parameter)，用以强调它作为占位符的角色，在调用函数时传递给函数的实际值要传递给形式参数。调用函数时，对应于形式参数的参数值就用于正在执行的函数体内。

为了说明这些思想，让我们重新编写4.1节“函数调用”中的程序，函数prn\_message()要有一个形式参数，这个参数用于说明要显示信息多少次。

```

#include <stdio.h>

void prn_message(int k);

int main(void)
{
    int how_many;

    printf("%s",
        "There is a message for you.\n"
        "How many times do you want to see it? ");
    scanf("%d", &how_many);
    prn_message(how_many);
    return 0;
}

void prn_message(int no_of_messages)
{
    int i;

    printf("Here is the message:\n");
    for (i = 0; i < no_of_messages; ++i)
        printf(" Have a nice day!\n");
}

```

### 对该信息程序的解析

- #include <stdio.h>

标准头文件stdio.h含有printf()和scanf()的函数原型，我们引入这个文件是因为我们要使用这些函数。函数原型是一种函数定义。为了正确地编译，编译器需要知道传递给函数的参数的个数和类型以及函数返回值的类型，函数原型能提供这些信息。

- void prn\_message(int k);

这是函数prn\_message()的函数原型，该函数的类型是void，它告诉编译器该函数没有返回值。参数列表是int no-of-messages，它告诉编译器这个函数有一个类型为int的参数。

- int main(void)
  - {
  - .....
  - }

这是main()的函数定义。第一行是该函数定义的头，在两个括号之间的行构成了该函数定义的体。通常程序员不为main()提供函数原型。

```
• printf("%s",
    "There is a message for you.\n"
    "How many times do you want to see it? ");
    scanf("%d", &how_many);
```

对printf()的调用提示用户输入一个整数。对scanf()的调用读入用户输入的字符，把字符转换成十进制数，并把值存储在how\_many中。

```
• prn_message(how_many);
```

这个语句调用函数prn\_message()，把how\_many的值作为参数传递给函数。

```
• void prn_message(int no_of_messages)
{
    .....
}
```

这是prn\_message()的函数定义。第一行是该函数定义的头，由于没有值返回到调用环境，所以函数的类型是void。标识符int no\_of\_message是被声明为int类型的参数，可以把参数int no\_of\_message看作是在调用函数时传递给函数的实参值的表示。在main()中调用该函数的语句是

```
prn_message(how_many);
```

在main()中，假设how\_many是2。当程序控制传递给prn\_message()时，变量no\_of\_message的值就是2。

```
• void prn_message(int no_of_messages)
{
    int i;

    printf("Here is the message:\n");
    for (i = 0; i < no_of_messages; ++i)
        printf(" Have a nice day!\n");
}
```

在两个花括号之间的行构成了prn\_message()的函数定义体。如果我们认为no\_of\_message的值是2，那么就显示消息两次。当程序控制达到函数尾时，控制返回到调用环境。■

注意，用在一个函数定义中的参数和局部变量与用在其他函数定义中的参数和局部变量无关。例如，如果变量i用在main()中，则它与用在prn\_message()中的变量i无关。

### 4.3 return语句

当执行return语句时，程序控制立即返回到调用环境；如果一个表达式跟在一个关键字return后面，表达式的值就返回到调用环境。如果需要，就把该值转换成像函数定义头描述的函数类型。return语句可为如下形式之一：

```
return;
return expression;
```

下面是一些例子：

```
return;
return 77;
return ++a;
return (a + b + c);
```

如果表达式复杂，你可以把被返回的表达式放在括号中，这样的编程习惯是很好的。

在一个函数中可以有多条return语句，也可以没有。如果没有return语句，当遇到函数体的结束花括号时，控制返回到调用环境，把这种情况称为离开结束(falling off the end)。为了说明return语句的用法，让我们编写一个计算两个整数中的最小值的程序。

```
#include <stdio.h>

int min(int a, int b);

int main(void)
{
    int j, k, minimum;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
    minimum = min(j, k);
    printf("\nOf the two values %d and %d, "
           "the minimum is %d.\n\n", j, k, minimum);
    return 0;
}

int min(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}
```

#### 对程序minimum的解析

- #include <stdio.h>
- int min(int a, int b);

我们引入系统头文件stdio.h是因为它含有printf()和scanf()的函数原型。除了printf()和scanf()外，我们还使用自己编写的函数min()，min()的函数原型有助于编译器完成工作。

- int main(void)
  - {
    - int j, k, minimum;
    - printf("Input two integers: ");
    - scanf("%d%d", &j, &k);

变量j、k和minimum被声明为int类型。用户要输入两个整数，函数scanf()用于将值存储于j和k中。

- minimum = min(j, k);

把j和k的值作为参数传递给min()。函数min()返回一个值赋给minimum。

- printf("\nOf the two values %d and %d, "
  - "the minimum is %d.\n\n", j, k, minimum);

显示出j、k和minimum的值。

- int min(int a, int b)

这是min()的函数定义的头，它的返回类型是int。这意味着，如果需要，该函数的返回值在返回到调用环境之前被转换成int。参数表int a, int b声明了a和b的类型是int。



这些参数要用在min()函数定义体中。

```
int min(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}
```

在括号内的代码构成了min()的函数定义体。如果a的值小于b的值，那么a的值要返回到调用环境，否则b的值返回到调用环境。 ■

即使是像min()这样短小的函数也为代码提供了有用的结构。如果我们想修改程序，计算出最大值，我们就可以使用函数max()来实现。我们必须把max()的函数原型放在文件的顶部，在main()中的适当位置调用它，并在文件的底部编写它的函数定义。下面是这个函数的定义：

```
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

我们已经设计了处理整数值的min()和max()，如果我们要使这两个函数能处理类型为double类型的值，就必须重新编写这两个函数。我们重新编写了min()，留下max()作为练习（请参见练习9）。不用a和b，而用x和y，这是计算float型和double型值的常见作法。

```
double min(double x, double y)
{
    if (x < y)
        return x;
    else
        return y;
}
```

即使函数返回了一个值，程序也不一定要使用它。

```
while (.....) {
    getchar(); /* get a char, do nothing with it */
    c = getchar(); /* c will be processed */
    .....
}
```

## 4.4 函数原型

函数应该先声明后使用。ANSI C提供了一种称为函数原型(function prototype)的新的函数声明语法。函数原型告诉编译器传递给函数的参数的个数和类型以及函数返回的值的类型。下面是一个例子：

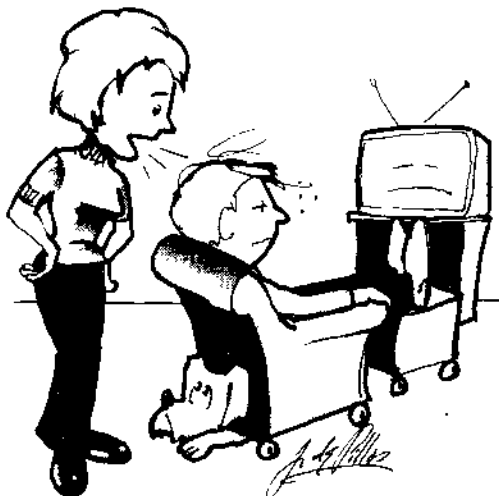
```
double sqrt(double);
```

它告诉编译器sqrt()是一个具有类型为double的单参数的函数，它的返回值的类型是double。函数原型的一般形式为

```
type function_name ( parameter type list );
```

参数类型表通常是用逗号分隔开来的类型列表。在表中可以没有标识符，标识符不会影响原型。例如，函数原型`void f(char c, int i);`等价于`void f(char, int);`。

编译器不使用函数原型中出现在参数类型列表中的像`c`和`i`这样的标识符，用这样标识符的目的是为程序员和阅读代码的读者提供文档。如果函数没有参数，就使用关键字`void`，如果函数没有返回值，也使用关键字`void`。



“噢，不要这样，哈罗德，你正建立一个无返回值的函数！”

如何函数的参数个数是不定的，那么就用省略符号(...)。请参见在标准头文件`stdio.h`中的`printf()`的函数原型。

函数原型可使编译器更彻底地检查代码。在需要之处，也可强制传递给函数的值。例如，如果已经描述了`sqrt()`的函数原型，那么函数调用`sqrt(4)`会产生正确的值，因为编译器知道`sqrt()`的参数类型是`double`，所以它把类型为`int`的值提升为`double`型的，并返回正确的值（请参见练习2）。

在传统的C中，不允许参数类型列表出现在函数声明中。下面是`sqrt()`的函数声明：

```
double sqrt();      /* traditional C style */
```

用这样的声明，函数调用`sqrt(4)`不会产生正确的值。尽管ANSI C编译器接受传统风格的函数定义，最好还是使用函数原型。

## 4.5 自顶向下设计

设想一下，我们必须分析由整数文件描述的一组数据。随着我们读入每个整数，我们要显示出计数、当前读入的整数、大于该整数的所有整数之和、小于该整数的最小整数以及大于该整数的最大整数。除此以外，假设必须要在显示页的头部显示一个标题，在标题部分的适当位置要分列整齐地显示所有信息。为了构造该程序，我们要用能把问题分解成如下子问题的自顶向下设计：

### 分解求和程序

#### 1) 显示标题。

- 2) 显示各列上部的标题部分。
- 3) 读入数据并分列整齐地显示它们。

其中的每个子问题都能作为函数直接被编码, 然后把这些函数用在main()中, 解决总的问题。注意, 通过这种方式设计代码, 我们可以增加用于分析数据的其他函数, 而不影响程序的结构。

```
#include <stdio.h>

void prn_banner(void);
void prn_headings(void);
void read_and_prn_data(void);
int min(int a, int b);
int max(int a, int b);

int main(void)
{
    prn_banner();
    prn_headings();
    read_and_prn_data();
    return 0;
}
```

该段程序以非常简单的方式说明了自顶向下设计的思想。程序员把要完成的任务和各项任务的编码看作是函数。如果一项任务很复杂, 那么就可以把它分解成一些较小的任务, 把每一项编制成函数。另一个益处是在整体上程序更易读, 程序的自编文档性也增强了。

对个体函数的编码是很简单的。我们在同一个文件中把所有的函数放在main()之后。第一个函数含有printf()语句。

```
void prn_banner(void)
{
    printf("\n%s%s\n",
        "*****\n",
        "    RUNNING SUMS, MINIMUMS, AND MAXIMUMS    *\n",
        "*****\n");
}
```

下一个函数显示各列上面的标题部分。用格式%5s显示5个间隔长的串, 用格式%12s分四次显示4个12个间隔长的串。

```
void prn_headings(void)
{
    printf("%5s%12s%12s%12s%12s\n\n",
        "Count", "Item", "Sum", "Minimum", "Maximum");
}
```

在read\_and\_prn\_data()中完成大多数工作。我们要解析这个函数, 以详细地表明它如何工作。在练习中, 要求你从键盘输入数据, 当你输入不正确的数据时, 看会发生什么情况(请参见练习3)。

```
void read_and_prn_data(void)
{
    int count = 0, item, sum, smallest, biggest;

    if (scanf("%d", &item) == 1) {
        ++count;
        sum = smallest = biggest = item;
        printf("%5d%12d%12d%12d%12d\n",
            count, item, sum, smallest, biggest);
        while (scanf("%d", &item) == 1) {
            ++count;
            sum += item;
        }
    }
}
```

```

        smallest = min(item, smallest);
        biggest = max(item, biggest);
        printf("%5d%12d%12d%12d%12d\n",
            count, item, sum, smallest, biggest);
    }
}
else
    printf("No data was input - bye!\n\n");
}

```

假设这个程序已被编译，并把可执行的代码已放进了名为run\_sums的文件中。如果我们执行这个程序，从键盘直接输入数据，我们就会在屏幕上看到输入字符的回显和程序的输出。为了防止这种问题的出现，我们创建一个含有如下整数的称为data的文件：

```
19 23 -7 29 -11 17
```

现在我们给出命令：

```
run_sums < data
```

这就把从键盘到程序的输入重定向到文件。下面是在屏幕上显示的信息：

```

*****
*   RUNNING SUMS, MINIMUMS, AND MAXIMUMS   *
*****

```

Count	Item	Sum	Minimum	Maximum
1	19	19	19	19
2	23	42	19	23
3	-7	35	-7	23
4	29	64	-7	29
5	-11	53	-11	29
6	17	70	-11	29

### 对函数read\_and\_prn\_data()的解析

```

• void read_and_prn_data(void)
{
    int    count = 0, item, sum, smallest, biggest;

```

该函数定义的头是花括号前的一行。由于函数没有返回值，所以它的类型是void。该函数没有参数。在函数定义的体中，把count、item、sum、smallest和biggest声明为int类型的局部变量。变量count被初始化为0，变量item的值从输入流中得到，要计算出变量sum、smallest和biggest的值。

```

• if (scanf("%d", &item) == 1) {
    ++count;
    sum = smallest = biggest = item;
    printf("%5d%12d%12d%12d%12d\n",
        count, item, sum, smallest, biggest);
    .....
}
else
    printf("No data was input - bye!\n\n");

```

函数scanf()返回已成功转换的数目。此处的scanf()从标准的输入流(键盘)读入字符，转换成十进制整数，并把结果存储在item中。如果转换处理成功，则表达式scanf("%d", &item)==1为真，就执行if语句体，也就是说，count加1，把item的值赋给变量sum、smallest和biggest，把这些值在适当的列上显示出来。注意在printf()语句中的格式类似于prn\_headings()中的格式。如果在转换中scanf()没有成功，那么就执行if-else语句的else部分。有两个原因会使得转换会失败。第一个原因是，

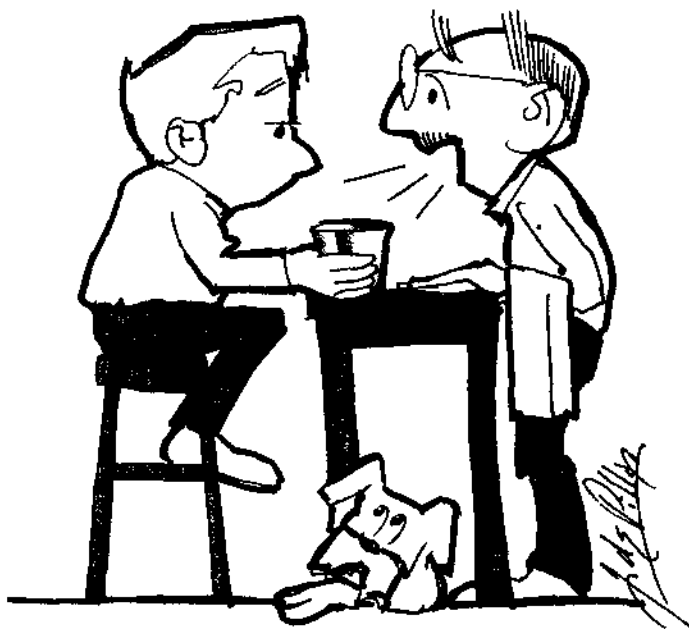
在输入流中数字出现之前出现了不适当的字符，例如字母x。由于scanf()不能把字符x转换成十进制整数，所以scanf()的返回值是0。另一个原因是，在输入流中没有字符，或只有空白字符，因为scanf()会跳过空白字符达到文件尾，当读到文件结束标志时，scanf()返回EOF，这个值是与系统相关的，通常是-1。

```

• while (scanf("%d", &item) == 1) {
    ++count;
    sum += item;
    smallest = min(item, smallest);
    biggest = max(item, biggest);
    printf("%5d%12d%12d%12d%12d\n",
        count, item, sum, smallest, biggest);
}

```

在从输入流中获得第一个字符后，我们在while循环中用scanf() 获得其他输入。scanf() 每做一次成功的转换，就执行一次while循环体。这使得count加1，用item的当前值对sum增量，把item当前值和smallest的最小值赋给smallest，把item当前值和biggest的最大值赋给biggest，并把这些值显示在适当的列上。最终，scanf()会遇到输入流中的一个不正确的字符，或达到文件尾，无论哪种情况，scanf()都返回一个不是1的值，使得程序控制从while循环中退出。 ■



“看，佛瑞德，又一个自底向上，你就不能对项目自顶向下编码！”

#### 4.6 程序的正确性：assert() 宏

ANSI C在标准库文件assert.h中提供了assert()宏，这个宏能用于保证表达式的值是程序员所希望的。

假设我们正编写一个重要函数f(int a, int b)，我们要保证传递给函数的参数满足一定的条件，例如，a必须是正的，b必须是在区间[7,11]内，还要假设函数的返回值必须要大于3。

```
#include <assert.h>

double f(int a, int b)
{
    double x;

    assert(a > 0);           /* precondition */
    assert(b >= 7 && b <= 11); /* precondition */
    .....
    assert(x >= 3.0);        /* postcondition */
    return x;
}
```

如果断言失败，系统就要显示出一个消息并终止程序。把断言`assert(a > 0)`称为前置条件(precondition)，这是因为它测试输入是否适合该函数以使该函数正确地工作。把断言`assert(x >= 3.0)`称为后置条件(postcondition)是因为它测试使该函数正确地工作必须成立的关系。

断言是容易编写的，它增强了代码的健壮性，并有助于代码的读者理解其用意，也有助于保证函数的行为满足要求，此外，断言有助于程序员考虑程序的正确性。这个原理是有益的。断言不必限制放在函数的开头和结尾，但其位置要自然。使用断言是一种良好的编程方法。

注意，如果在引入`assert.h`的地方定义宏`NDEBUG`，那么就会忽略所有的断言，这使得程序员在程序开发期间自由地使用断言，随后在定义宏`NDEBUG`后丢弃它们。

## 4.7 从编译器的角度来看函数声明

对于编译器而言，可利用函数调用、函数定义、显式的函数声明和函数原型的方式生成函数声明。如果函数调用`f(x)`在任何它的声明、定义或原型出现之前遇到，编译器就假设用如下形式的缺省声明：

```
int f();
```

对这个函数，没有给出任何参数表信息。现在假设下面的函数定义首先出现：

```
int f(x)           /* traditional C style */
double x;
{
    .....
```

这段代码为编译器提供了声明和定义。然而，还是没有给出任何参数表信息。把一个类型为`double`的单参数传递给`f()`是程序员的责任。可以想到像`f(1)`这样的函数调用是要失败的，因为`1`的类型是`int`而不是`double`。现在假设我们用ANSI C风格进行定义。

```
int f(double x)    /* ANSI C style */
{
    .....
```

编译器现在知道参数表。在这种情况下，可以想到`f(1)`这样的函数调用能正常地工作。当用`int`类型的值作参数时，它被转换成`double`型。

函数原型是函数声明的一种特殊情况。良好的编程风格在使用一个函数之前要给出它的函数定义(ANSI C风格)、函数原型或两者都给出。引入标准头文件的主要原因是它含有函数原型。

## 限制

对函数定义和函数原型有一定的限制。函数存储类型说明符可以是extern(外部的)也可以是static(静态的),但不能两者都是;不能使用auto(自动)和register(寄存器)(请参见8.6.4节“存储类型static”)。函数不能返回数组类型和函数类型的值,然而,表示数组或函数的指针可以作为函数的返回值(请参见第9章“数组和指针”)。能出现在参数类型表中的惟一存储类说明符是register。不能对参数初始化。

## 4.8 问题求解: 随机数

在计算机中,随机数有很多用途。一种用途是作为测试代码的数据;另一个用途是模拟现实世界中与概率有关的事件。模拟方法是一个重要的解决问题的技术,把用随机数函数产生概率的程序称为蒙特卡罗(Monte Carlo)模拟。用蒙特卡罗技术可以解决很多问题,否则这些问题是不可能得到解决的。

随机数发生器是一个返回整数的函数,这些整数随机分布在0到n区间,此处的n是与系统相关的。在标准库中的函数rand()提供了该项功能。让我们编写一个显示由rand()产生的一些随机数的程序,下面是这个程序的第一部分:

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

int    max(int a, int b);
int    min(int a, int b);
void   prn_random_numbers(int k);

int main(void)
{
    int    n;

    printf("Some random numbers will be printed.\n");
    printf("How many would you like to see? ");
    scanf("%d", &n);
    prn_random_numbers(n);
    return 0;
}
```

由于rand()的函数原型是在标准头文件stdlib.h中,我们要在文件的开始处引入它。函数scanf()用于把从键盘上输入的字符转换成十进制整数,并将值存储在n中,n的值被作为参数传递给函数prn\_random\_numbers()。

在程序的剩余部分中,我们要编写函数定义max()、min()和prn\_random\_numbers()。我们已经讨论了max()和min(),下面是函数prn\_random\_numbers():

```
void prn_random_numbers(int k)
{
    int    i, r, biggest, smallest;

    r = biggest = smallest = rand();
    printf("\n%7d", r);
    for (i = 1; i < k; ++i) {
        if (i % 7 == 0)
            printf("\n");
        r = rand();
        biggest = max(r, biggest);
        smallest = min(r, smallest);
        printf("%7d", r);
    }
    printf("\n\n%5d\n%5d\n%5d\n\n",
```

```

    "Count: ", k,
    "Maximum: ", biggest,
    "Minimum: ", smallest);
}

```

我们要对这个函数定义进行解析，但在解析之前，让我们看一下这个程序的输出是怎样的。假设我们运行这个程序，当出现提示时输入23，下面是在屏幕上显示的信息：

```

Some random numbers will be printed.
How many would you like to see? 23

16838  5758  10113  17515  31051  5627  23010
7419   16212 4086   2749  12767  9084  12060
32225  17543 25089  21183  25137  25566  26966
4978   20495

Count:    23
Maximum: 32225
Minimum:  2749

```

#### 对函数prn\_random\_numbers()的解析

```

• void prn_random_numbers(int k)
{
    int i, r, biggest, smallest;

```

变量k是一个被声明为类型是int的变量。局部变量i、r、smallest和biggest被声明为int类型的。

```

• r = biggest = smallest = rand();
  printf("\n%7d", r);

```

标准库中的函数rand()用于产生随机数。把产生的随机数赋值给变量r、smallest和biggest。函数printf()用于显示r的值，即把r作为十进制整数按长度为7显示。

```

• for (i = 1; i < k; ++i) {
    if (i % 7 == 0)
        printf("\n");
    r = rand();
    .....
}

```

这个for循环用于显示其余的随机数。由于已经显示了一个随机数，在循环头部的变量i被初始化为1而不是0。只要i能用7除（值为7，14，21，…），由表达式i%7==0控制的if语句就为真，这会引引起一个换行。这样，在每一行上至多显示7个随机数。 ■

### 4.9 函数定义次序的可选风格

如果我们要在一个文件中编写一个程序，我们通常把#include和#define放在文件的头部，枚举类型（请参见第7章“枚举类型和typedef”）和结构类型（请参见第12章“结构和抽象数据类型”）这样的程序元素放在下边，接着是函数原型的列表。在文件的底部，我们要编写main()的函数定义，它跟在所有的其他函数定义的后边。在4.8节“问题求解：随机数”的程序random就使用了这种次序。

```

#include <stdio.h>
#include <stdlib.h>

list of function prototypes

int main(void)

```



```

{
    .....
}

int max(int a, int b)
{
    .....
}

int min(int a, int b)
{
    .....
}

void prn_random_numbers(int k)
{
    .....
}

```

由于函数定义也起函数原型的作用，一种可选的风格是去掉函数原型的列表，在调用一个函数之前，给出要被调用的函数的函数定义。实际上，main()的函数放于文件的最后。让我们通过重新安排程序random的次序，来说明这种可选的风格。

```

#include <stdio.h>
#include <stdlib.h>

int max(int a, int b)
{
    .....
}

int min(int a, int b)
{
    .....
}

void prn_random_numbers(int k)
{
    .....
}

int main(void)
{
    .....
}

```

因为prn\_random\_numbers()调用max()和min()，所以max()和min()必须要先出现，由于main()调用prn\_random\_numbers()，所以main()要置后（请参见练习11）。

虽然我们喜欢先书写main()这种自顶向下的风格，但偶尔我们也用这种可选的风格。

#### 4.10 开发一个大程序

通常，要在一个单独的目录中编写一个大程序，形成一个.h和.c文件集，每一个.c文件含有一个或多个函数定义。按需要重新编译每个.c文件，这样可节省程序员和机器的时间。我们将在第14章“软件工具”中进一步讨论这个问题，在那里我们要讨论库和make的用法。

假设我们正在开发一个称为pgm的大程序。在每一个.c文件的头部含有一条

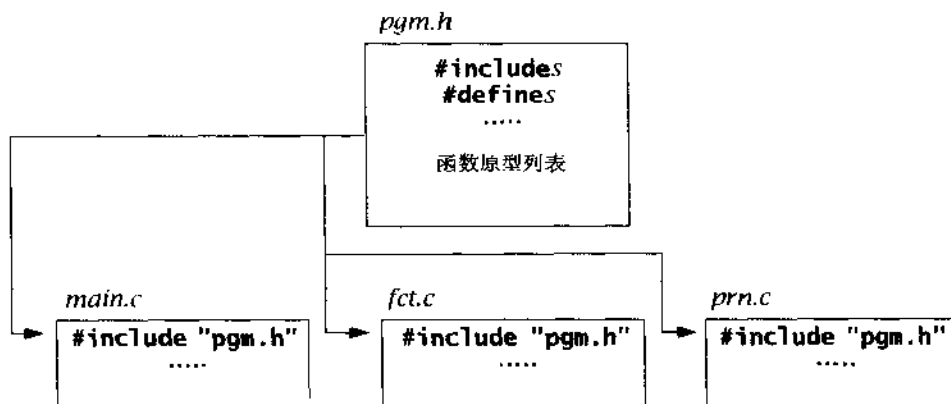
```
#include "pgm.h"
```

当预处理器遇到这条指令时，它首先在当前的目录中寻找文件pgm.h。如果存在这个文件，就引入它；如果不存在，预处理器就在与机器相关的目录中寻找它；如果没有找到pgm.h，预

处理器就会给出一个出错消息，并停止编译。

我们的头文件pgm.h可以包含#include和#define、枚举类型声明、结构类型声明、其他编程结构以及在底部的函数原型列表。这样，pgm.h包含了所需要的所有程序元素。由于pgm.h出现在每一个.c文件的头部，它是把程序结合在一起的“胶水”。

创建一个包含在所有.c文件中的.h文件



我们给出一个它如何工作的简单例子。我们在一个单独的目录中编写程序，它由一个.h和三个.c组成。通常，目录的名字和程序的名字是相同的，下面是我们的程序。

pgm.h中的内容：

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

#define N 3

void fct1(int k);
void fct2(void);
void prn_info(char *);
```

main.c中的内容：

```
#include "pgm.h"

int main(void)
{
    char ans;
    int k, n = N;

    printf("\n%s",
        "This program does not do very much.\n"
        "Do you want more information? ");
    scanf(" %c", &ans);
    putchar('\n');
    if (ans == 'y' || ans == 'Y')
        prn_info("pgm");
    for (k = 0; k < n; ++k)
        fct1(k);
    printf("Bye!\n\n");
    return 0;
}
```

fct.c中的内容：

```
#include "pgm.h"
```

```

void fct1(int n)
{
    int i;

    printf("Hello from fct1()\n");
    for (i = 0; i < n; ++i)
        fct2();
}

void fct2(void)
{
    printf("Hello from fct2()\n");
}

```

prn.c中的内容:

```

#include "pgm.h"

void prn_info(char *pgm_name)
{
    printf("%s\n",
        "Usage:  pgm\n"
        "\n"
        "This program illustrates how one can write\n"
        "a program in more than one file. In this\n"
        "example, we have a single .h file that is\n"
        "included at the top of our three .c files.\n"
        "Thus the .h file acts as the \"glue\"\n"
        "that binds the program together.\n"
        "\n"
        "Note that the functions fct1() and fct2()\n"
        "when called only say \"hello.\" When writing\n"
        "a serious program, the programmer sometimes\n"
        "does this in a first working version\n"
        "of the code.\n");
}

```

注意,我们在prn\_info()的函数原型中使用了类型char\* (指向char的指针)(在第10章“串和指针”中我们将讨论指针和串)。我们用命令

```
cc -o pgm main.c fct.c pm.c
```

编译程序。

编译器创建了可执行文件pgm和三个对应于.c文件的.o文件。在MS-DOS中,它们是.obj文件。.o文件被称为目标文件。关于这些目标文件和编译器如何使用它们的进一步讨论,请参见14.3节“C编译器”。



在下节，我们要编写一个更有趣的多文件程序。在7.3节“例子：石头、剪刀、布游戏”中给出以多文件编写程序的另一个例子。如果你能上因特网，你可以用匿名的ftp得到更多的例子，用下述命令开始：

```
ftp aw.com
```

在你已经连接上后，你就可以注册和改变目录(cd)到cseng/authors，然后用cd到kelley\_pohl下进行查找。如果你获得了一个大程序的源代码，探索它是很容易的。你能显示感兴趣的部分，如果你愿意你还可以通过调试器的帮助来单步调试程序。

### 大程序由什么构成

作为一个个体，一个大程序可以由数百行代码组成。应该计算哪些行？通常在READ\_ME文件（应该仅有一个）、.h文件、.c文件和makefile中的行都应计算在内（请参见14.8节“make的用法”）。在UNIX中，可用词计算工具wc完成此事：

```
wc READ_ME *.h *.c makefile
```

在工业化生产中，通常一个程序是由一个程序员组编写的，一个大程序可能超过十万行。

在程序的各自目录中编写程序，形成由.h和.c文件组成的集合，这种风格对任何重要的问题都很适用，而且无论问题的大小，有经验的程序员都遵循这种风格。要想精通这种风格，程序员必须学习怎样使用make或一些类似的工具（请参见14.8节“make的用法”）。

### 4.11 模拟：正反面游戏

为了给出另一个使用函数的例子，我们要完成一个模拟孩子们称之为正反面的计算机游戏。在这个游戏中，第一个孩子投掷一枚硬币，第二个孩子猜是正面还是反面。如果第二个孩子猜得正确，他就赢了，否则他就输了。游戏可以重复地进行，并记录赢输的次数。

机器用rand()模拟投掷硬币。这是一种简单的蒙特卡罗模拟形式。如果rand()返回的整数是偶数，就认为它代表硬币的正面；如果是奇数，就认为它代表硬币的反面。通过向比赛者显示一些指令作为比赛的开始，这些指令包含有一些对程序设计方面的考虑。在每次投掷硬币后，要显示结果。作为程序的结果，要显示一份最终的报告。

我们要在程序自己的目录中编写它，该程序由.h文件和.c文件组成。自顶向下设计展现出了所需要的函数数目。每个函数都不长，这使得整个程序容易阅读。

heads\_or\_tails.h中的内容：

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

#define MAXWORD 100

int get_call_from_user(void);
void play(int how_many);
void prn_final_report(int win, int lose, int how_many);
void prn_instructions(void);
void report_a_win(int coin);
void report_a_loss(int coin);
int toss(void);
```

main.c中的内容：

```
#include "heads_or_tails.h"
```

```

int main(void)
{
    char    ans;
    int     no_of_plays;

    printf("\n"
           "THE GAME OF HEADS OR TAILS\n"
           "\n"
           "Do you want instructions? ");
    scanf("%c", &ans);
    putchar('\n');
    if (ans == 'y' || ans == 'Y')
        prn_instructions();
    printf("How many times do you want to play? ");
    scanf("%d", &no_of_plays);
    putchar('\n');
    play(no_of_plays);
    return 0;
}

```

get.c中的内容:

```

#include "heads_or_tails.h"

int get_call_from_user(void)
{
    int    guess;                /* 0 = heads, 1 = tails */

    do {
        printf("Call it: ");
        if (scanf("%d", &guess) != 1) {
            printf("\nSORRY: Severe input error - bye!\n\n");
            exit(1);
        }
        if (guess != 0 && guess != 1) {
            printf("\n%s\n\n",
                  "ERROR: Type 0 for heads, 1 for tails.");
        }
    } while (guess != 0 && guess != 1);
    return guess;
}

```

play.c中的内容:

```

#include "heads_or_tails.h"

void play(int how_many)    /* machine tosses, user calls */
{
    int    coin, i, lose = 0, win = 0;

    for (i = 0; i < how_many; ++i) {
        coin = toss();
        if (get_call_from_user() == coin) {
            ++win;
            report_a_win(coin);
        }
        else {
            ++lose;
            report_a_loss(coin);
        }
    }
    prn_final_report(win, lose, how_many);
}

int toss(void)
{
    return (rand() % 2);    /* 0 = heads, 1 = tails */
}

```

prn.c中的内容:

```
#include "heads_or_tails.h"

void prn_instructions(void)
{
    printf("%s\n",
        "This is the game of calling heads or tails.\n"
        "I will flip a coin; you call it. If you\n"
        "call it correctly, you win; otherwise,\n"
        "I win.\n"
        "\n"
        "As I toss the (simulated) coin, I will\n"
        "tell you to \"call it.\" To call heads,\n"
        "type 0; to call tails, type 1.\n"
        "\n");
}

void prn_final_report(int win, int lose, int how_many)
{
    printf("\n%s\n%s%3d\n%s%3d\n%s%3d\n\n",
        "FINAL REPORT:",
        "    Number of games that you won: ", win,
        "    Number of games that you lost: ", lose,
        "    Total number of games: ", how_many);
}
```

在这个程序中有为数不多的几个新想法。在prn\_instructions()的函数定义中,格式%s用在printf()语句中,用于显示一个由一些用空格隔开的串常量组成的串参数。编译器把这些串连接成一个单串。注意"call it"是串参数的一部分。若在一个串中包含一个双引号作为一个字符,必须用转义字符\作为它的前缀,以防止串过早地结束。在下一章中,我们会看到转义机制的其他例子。

toss()的函数体由一个单语句组成,返回值是表达式rand()%2的值。回想一下,形式为a%b的取模表达式的值是a用b除后的余数。例如,4%2的值是0,5%2的值是1。这样,如果rand()返回的整数是偶数,那么rand()%2的值是0;如果rand()返回的整数是奇数,那么rand()%2的值是1。

该程序的用户必须输入0表示正面,输入1表示反面。一个更好的策略是输入字母h表示正面,输入t表示反面,但这种表示我们要用到第5章“字符处理”中提出的思想。

## 4.12 调用和按值调用

通过写一个函数的名字来对此函数进行调用,同时要在括号内给出适当的参数。通常,这些参数在数目和类型上要与函数定义的参数列表中的参数相匹配。所有参数都是按值调用的,这意味着要计算每一个参数,参数的值局部地用于相应的形参出现的地方。这样,若把一个变量传递给一个函数,则在调用环境中该变量的存储值不会发生变化。

让我们编写一个基本程序,用它明确地说明按值调用的概念。

```
#include <stdio.h>

int compute_sum(int n);

int main(void)
{
    int n = 3, sum;

    printf("%d\n", n);                /* 3 is printed */
    sum = compute_sum(n);
    printf("%d\n", n);                /* 3 is printed */
    printf("%d\n", sum);              /* 6 is printed */
}
```

```

    return 0;
}

int compute_sum(int n)      /* sum ints from 1 to n */
{
    int    sum = 0;

    for ( ; n > 0; --n)    /* in main(), n is unchanged */
        sum += n;
    printf("%d\n", n);      /* 0 is printed */
    return sum;
}

```

即使把 $n$ 传递给`compute_sum()`，在函数体内 $n$ 的值会发生变化，但在调用环境中 $n$ 的值没有发生变化。

在第8章“函数、指针和存储类型”中，我们将解释如何实现传址（引用）调用。它是把变量的地址（引址）传递给函数允许函数体改变调用环境中的变量值的一种方法。

### 4.13 递归

像在本章开始的函数`prn_message()`中所表现的那样，重复计算通常用重复的语句来实现。C函数支持的一种可选的控制流模式是递归(recursion)，递归是一种函数自己直接或间接调用自己的能力。下面是一个说明递归用法的程序：

```

#include <stdio.h>

void r_prn_message(int k);

int main(void)
{
    int    how_many;

    printf("%s",
        "There is a message for you.\n"
        "How many times do you want to see it? ");
    scanf("%d", &how_many);
    printf("Here is the message:\n");
    r_prn_message(how_many);
    return 0;
}

void r_prn_message(int k)
{
    if (k > 0) {
        printf("    Have a nice day!\n");
        r_prn_message(k - 1);    /* recursive fct call */
    }
}

```

由于函数`r_prn_message()`调用自己，所以它是递归的。递归和循环中都进行重复的计算，直到满足终止条件为止。

在上面的程序中，终止条件是 $k \leq 0$ 。如果传递给函数`r_prn_message()`的整数参数 $k$ 是正的，就显示一行，并且该函数用参数 $k - 1$ 递归地调用自己。由于在每次递归调用中 $k$ 都减1，这就保证递归肯定能终止。

程序初学者使用递归很容易犯错误，我们在4.15节“常见的编程错误”中要说明一种错误类型。

关于间接递归的讨论，请参见练习7。在较高级的程序中，递归是一个很重要的概念。关于进一步的讨论，请参见第11章“递归”。

“我为你感到骄傲，哈罗德。你使用的几乎全是递归，却毫无怨言！”



#### 4.14 风格

把问题分解成能作为函数编码的子问题对于良好的编程风格来说是至关重要的。为了易于阅读，一个函数至多一页代码。从选择的标识符名来看函数的用途是不明显的，应该对函数进行注释。每一个参数的标识符也应该能表明其用途，否则应该对其加注释。

在文件中函数定义出现的次序并不重要。通常，在`main()`后写其他的函数定义还是先写其他的函数定义再写`main()`全凭个人的爱好。如果一个函数被调用，并且它的定义是在另一个文件中，或是在同一个文件中调用语句的后面，则它的原型应该放在函数调用之前。然而，如果正在进行自顶向下设计，自然要从`main()`开始。当然，对于大项目来说，首先可以在纸上勾画一些程序组织，即使在自顶向下设计中，函数编码也可以先出现。

在给定函数中仅有几个`return`语句是一种良好的编程风格。如果有很多`return`语句，代码的逻辑是难以理顺的。

名字`read`、`write`和`print`常被用于组成系统的函数名，例如`prnft()`就使用了`print`。为了把我们的命名与系统名清楚地区别开来，我们经常用`prn`和`wrt`来组成名字。我们可以写一个命名为`print()`的函数，但它很容易与系统的`prnft()`混淆。

在循环内部对一些事物尽可能地计数是一个好主意。在`read_and_prn_data()`的函数定义中对`count`增量就遵循了这样的规则。

#### 4.15 常见的编程错误

一个常见的编程错误是假设函数能改变变量的值。因为在C中函数机制是严格遵循按值调用的，所以在调用环境中不可能通过用变量作为参数调用函数来改变变量的值。如果`f()`是函数，`v`是变量，那么语句

```
f(v);
```

在调用环境中不能改变`v`的值。然而，如果`f()`返回一个值，那么语句



`v = f(v);`

能改变`v`的值。

在ANSI C中, 假设`main()`向宿主环境或操作系统返回一个整数值。通常程序员编写

```
int main(void)
{
    .....
    return 0;
}
```

一些编译器接受`void`作为函数类型, 也允许省略`return`语句。

```
void main(void)          /* non ANSI C style */
{
    .....               /* no return statement */
}
```

虽然有的编译器可能对此处理得很好, 但在技术上这是错误的, 其他的ANSI C编译器也不接受这种风格。

在`main()`中, 通过写以下语句程序员把一个整数值返回到宿主环境中:

`return expr;`或`exit( expr );`

在`main()`中, 这两个语句是等价的, 但在任何的其他函数中, 这两个语句的作用是不同的。从函数中调用`exit()`会终止程序, 并向宿主环境返回一个值, 把返回的值称为退出状态(exit status)或程序状态(program status)。根据惯例, 退出状态0表示程序成功地终止, 而非0的退出状态表示异常或不正常的状况(请参见A.13节“遗留问题”)。

缺少函数原型会引起难以检测到的运行错误。传统的C被认为是不适合于编程初学者的, 这是因为函数参数机制没有提供类型检查的安全机制。考虑下边的程序:

```
/* Print a table of square roots. */
#include <math.h>
#include <stdio.h>

double  sqrt();          /* traditional C style */

int main()
{
    int  i;

    for (i = 1; i < 20; ++i)
        printf("%5d:%7.3f\n", i, sqrt(i));
    return 0;
}
```

显示出的内容是与编译器相关的。一些编译器输出如下:

```
1:  0.000
2:  0.000
3:  0.000
.....
20: 0.000
```

在本例中, 类型为`int`的`i`的值被传递给函数`sqrt()`, 该函数没有把`i`转换成函数所希望的`double`型, 这引起了不正确的显示值。在ANSI C中, 在`math.h`中提供的`sqrt()`的函数原型是

```
double  sqrt(double);
```

如果我们用这个原型代替传统C的声明, 作为参数传递给`sqrt()`的任何整数表达式都被转换成`double`型的, 程序的输出也更有意义。很多C的实践者在要求类型安全的参数传递机

制中都考虑这种改善，这是ANSI C优于传统C的最重要之处。

编写递归函数的终止条件是很容易犯错误的。这种类型的错误经常导致死循环。当发生这种错误时，用户通常（但不总是）要用Ctrl+c终止程序。如下是一个作为例子的简单程序，它不会正常结束。当心：在一些MS-DOS系统中，如果你执行这个程序，你可能不得不重新启动机器，才能终止它。

```
#include <stdio.h>

int main(void)
{
    int    count = 0;

    if (++count < 77) {
        printf("    The universe is ever expanding!  ");
        main();
    }
    return 0;
}
```

每一次递归都要调用main()，一个新的局部变量count被初始化为0，因此，终止条件++count >= 7总也不能满足。改正这一问题的一种方法是使用声明：

```
static int    count = 0;
```

当再次调用函数main()时，不会再次对静态变量初始化，在函数调用之间该变量保持不变（请参见8.6.6节“存储类型static”）。

## 4.16 系统考虑

调用rand()产生一个位于区间[0, RAND\_MAX]中的值，此处RAND\_MAX是一个在stdlib.h中给出的符号常量。由于RAND\_MAX通常是一个相对小的值32767，函数rand()在很多科学计算中用处不大。在UNIX上的大多数系统都为程序员提供了随机数发生器族rand48，之所以这样叫是因为48位算术被用于产生数字。例如，函数drand48()能用于产生随机分布在[0, 1]内的双精度数，lrand48()能用于产生随机分布在[0, 2<sup>31</sup> - 1]内的整数。通常，这个函数族的函数原型在stdlib.h中。要找出更多的伪随机数发生器，请参看William Press等著述的《Numerical recipes in C》(Cambridge, England:Cambridge University Press,1992)的第274~328页。

为了开启所有的警告，配置你的编译器是很重要的。假设我们正在MS-DOS环境下使用Borland C的命令行版本。若我们发出命令bcc -w pgm.c，就开启了所有的警告。特别是，如果pgm.c丢失了一个或几个函数原型，编译器就会告诉我们。通过在bcc.exe所在的目录里的turboc.cfg中放置-w，可以配置bcc，以使警告能被自动地开启。类似地，可以配置Borland C的集成环境，以使所有的警告能自动地开启（参见Borland C手册）。为了理解警告，试一下下面的程序：

```
#include <stdio.h>

int main(void)
{
    printf("%d\n", f(2));          /* 7 is printed */
    printf("%d\n", f(2.0));       /* 1 is printed */
    return 0;
}

int f(int n)
```

```
{
    return (3 * n + 1);
}
```

该程序所显示的内容与系统相关。我们已经看到我们的Borland C系统显示出的消息。

ANSI C和传统C是兼容的。由于传统C没有函数原型，如果我们在ANSI C系统中编写一个没有函数原型的程序，传递给函数的实参就得不到保证，这意味着不进行类型检查或转换。在上一个程序中，若传递给f()的参数是double型的常量2.0，就不会进行类型检查。由于f()期望的是int型的值，这样就会引起一个错误。如果我们在main()上面放一个函数原型

```
int f(int);
```

那么编译器就把传递给f()的任何数学表达式转换为int型的。

在ANSI C中的函数原型的类型安全特征是很强大的，但程序员必须要一致地使用原型，以便获益。

## 4.17 转向C++

C++增加了大量扩展了函数的有效性和效率的概念和结构。我们要讨论三个概念：内联函数 (inline Function)、缺省参数和函数重载(function overloading)。

通过在函数返回类型的前面加关键字inline来修改函数。如果可能，这样的函数被内联编译，这样避免了函数调用和函数返回的系统开销。这类似于C编译器对带有参数的#define宏的使用 (请参见B.2节“#define的用法”)。下面是一个简单的宏：

```
#define CUBE(x) ((x) * (x) * (x)) /* C style macro */
```

在定义这个宏后，我们能像使用一个函数那样的方法使用它：

```
double x, y = 2;
x = CUBE(y); /* x has value 8 */
```

在现代编程方法中，使用带参数的宏被认为是不健壮的，因而要避免使用它或尽量少使用它 (关于更多的信息，请参见附录B“预处理器”)。相反，在C++中，使用内部函数被认为是良好的编程实践。下面是说明了如何把cube()函数写为内部函数：

```
inline double cube(double x) { return x * x * x; }
```

像我们已经做的那样，通常把短的内联函数写在一行中。用内部函数而不用宏有几个原因。一般地讲，内联是较安全的，这是因为编译器增强了类型和作用域规则；另一方面，编译器通过预处理器用文本置换实现宏，预处理器不“知道”C。

在C++中，函数可以有缺省参数。这种效果通过把缺省值赋给在函数原型中的形参来实现。然而，如果这样做，右面的其他参数也必须给定缺省值。通常，缺省值是一个在调用函数时经常出现的适当的常量。下述的程序说明了这个机制：

```
// Illustrate default values for a function.
#include <iostream.h>
int sum(int a, int b = 2, int c = 3); // default values
int main(void)
{
    cout << "sum(3)          = " << sum(3)          << "\n"
          << "sum(3, 5)       = " << sum(3, 5)       << "\n"
```

```

        "sum(3, 5, 7) = " << sum(3, 5, 7) << "\n";
    }

    int sum(int a, int b, int c)
    {
        return a + b + c;
    }

```

当调用`sum(3)`时, 控制传递给带有参数`a`、`b`和`c`的函数, 此时`a`的值是3, `b`的值是2, `c`的值是3, 这样值8被返回, 并被显示。类似地, 当调用`sum(3, 5)`时, `a`的值是3, `b`的值是5, `c`的缺省值是3, 这样值11被返回, 并被显示。如果我们愿意, 我们能重新编写这个程序, 使函数定义起函数原型的作用。

```

// Illustrate default arguments for a function.

#include <iostream.h>

int sum(int a = 1, int b = 2, int c = 3) // fct def
{
    return a + b + c;
}

int main(void)
{
    cout << "sum()          = " << sum()          << "\n"
         << "sum(3)         = " << sum(3)         << "\n"
         << "sum(3, 5)        = " << sum(3, 5)        << "\n"
         << "sum(3, 5, 7)      = " << sum(3, 5, 7) << "\n";
}

```

如果用一个缺省值为该函数原型中的一个参数赋值, 则要用缺省值给参数表中的所有剩余参数赋值, 参数本身不必出现。下面是一些例子:

```

int    f(int a = 1, int b = 2, char c = 'A'); // ok
char   g(char, float = 0.0, double d = 3.14); // ok
float  h(int i, int j, float, float, int = 0); // ok
float  k(int = 0, int, int, float f = 3.579); // wrong

```

在C++中, 两个或多个函数或运算符可以有相同的名字。若是这样, 则把这样的函数或运算符说是被重载(overload)。我们要在此讨论函数重载, 留下运算符重载在15.3节“重载”中讨论。在下面的程序中我们要重载函数`max()`:

find\_max.h中的内容:

```

#include <iostream.h>

int    max(int a, int b);
double max(double x, double y);

```

main.c中的内容:

```

#include "find_max.h"

int main(void)
{
    int    a, b;
    double x, y;

    cout << "Enter two integers a and b: ";
    cin >> a >> b;
    cout << "Enter two doubles x and y: ";
    cin >> x >> y;
    cout << "max(a, b) = " << max(a, b) << ", an int\n";
    cout << "max(x, y) = " << max(x, y) << ", a double\n";
}

```

fmax.c中的内容:

```
#include "find_max.h"

int max(int a, int b)          // int version
{
    return ((a > b) ? a : b);
}

double max(double x, double y) // double version
{
    return ((x > y) ? x : y);
}
```

通过特征标记(signature),我们能预定传递给函数的参数的个数和类型。编译器用函数名和特征标记选择一个重载函数,例如,函数max(a, b)的特征标记与

```
int max(int, int);
```

的函数定义匹配,它是要调用的函数。

### C++中的函数原型

在C++中,函数原型是必须的,在函数原型和函数定义里的参数类型表中的void的使用是可选的。例如,在C++中,int f()等价于int f(void)。注意,这个想法与用在传统C和ANSI C中的结构相矛盾,在传统C和ANSI C中,像int f();这样的函数声明意味着f()的参数个数是未知的。在传统C中,void不是关键字,不能在函数声明或函数定义的参数表中作为类型。在ANSI C中,建议程序员使用函数原型。

### 小结

- 结构化编程是一种解决问题的策略,也是一种尽量简化控制流和使用自顶向下设计的编程方法。
- 自顶向下设计,也被称为逐步细化,由把问题分解为较小问题的反复分解组成。
- 应该把大程序编写为一个函数集,每个函数不大于一页。每个函数应该完成解决整个问题的任务中的一些小任务。
- 在一个函数体中,编译器用后跟括号的名字(如prn\_message()或min(x, y))进行函数调用。
- 通过编写函数定义,程序员创建由头和体组成的函数。头由函数返回的类型、函数名和用逗号分隔的参数声明表组成,参数声明表由括号括起来。体由用花括号括起来的声明和语句组成。
- 当调用一个函数时,就把程序控制传递给该函数。当执行return语句时,或达到函数的结束标志时,控制就传回调用环境。如果return语句含有表达式,表达式的值也被传回调用环境。
- 在ANSI C中存在函数原型机制,但在传统C中则不存在。函数原型有如下的一般形式:  
*type function\_name ( parameter type list );*  
*type*是函数返回的类型。*parameter type list*通常是由逗号分隔的类型表。如果函数没有参数,就用关键字void。当调用函数时,函数原型允许编译器强制类型兼容。
- 在函数原型中,参数表中的标识符可以跟在一个类型的后边。例如, int f(int a,

`float b);` 和 `int f(int, float);` 是等价的。编译器不需要参数标识符, 仅需要参数类型。然而, 参数本身能为读者提供进一步的文档。

- 在C中, 所有参数都是按值传递的, 这意味着当把一个变量作为参数传递给函数时, 它的值在调用环境中保持不变。
- 虽然所有的C系统都提供了函数 `rand()`, 但对于重要的工作而言, 其他的随机数发生器可能更合适些。
- C中的所有函数都可递归使用。也就是说, 任何函数都可直接或间接地调用自身。

## 练习

1. 编写程序 `message`, 使得它的输出为:

```
Message for you: Have a nice day!
                  Have a nice day!
                  Have a nice day!
                  .....
```

2. 编写一个函数 `square()`, 输入一个整数, 返回它的平方; 再编写一个函数 `cube()`, 输入一个整数, 返回它的立方。用函数 `square()` 和 `cube()` 编写函数 `quartic()` 和 `quintic()`, 它们分别返回一个整数的四次方和五次方。用这些函数编写一个显示从1到25的整数幂的表的程序。程序的输出应该具有如下的形式:

A TABLE OF POWERS

```
-----
Integer      Square      Cube      Quartic      Quintic
-----
          1           1           1           1           1
          2           4           8          16          32
          3           9          27          81         243
          .....
```

3. 执行程序 `run_sums`, 从键盘上直接输入数据。当你完成输入数据后, 输入文件结束标志 (请参见1.11节“输入文件结束标志”)。如果你输入一个字母而不是一个数字, 会发生什么情况?

4. 当要求输入随机数的个数时, 如果用户输入0, 则程序 `prn_rand` 工作错误。修改 `prn_rand`, 使得发生上述情况时, 它能正确地工作。

5. 考虑函数 `prn_random_numbes()` 中的 `for` 循环, 它的开始部分是:

```
for (i = 1; i < k; ++i) {
    if (i % 7 == 0)
        printf("\n");
    .....
```

假设我们把第一行重写如下:

```
for (i = 2; i <= k; ++i) {
```

显示的随机数的个数会发生变化吗? 这样的修改会引起输出格式发生变化。试一下, 看情况如何。对程序的 `for` 循环体做进一步修改, 使得输出格式保持正确。

6. 运行程序 `prn_rand()` 三次, 显示出100个随机分布的整数。观察每次显示出相同的数表。对于很多应用来说, 都不希望如此。通过用 `srand()` 为随机数发生器播种, 修改 `prn_rand()`。你的程序的开始几行应该如下:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    int i, n, seed;

    seed = time(NULL);
    srand(seed);
    printf("\n%s",
        "Some randomly distributed "
        "integers will be printed.\n"
        "How many do you want to see? ");
    .....

```

该函数调用`time(NULL)`返回自1970年1月1日以来经过的秒数(请参见附录A“标准库”),我们把这个值存储在变量`seed`中,然后用该函数调用`srand(seed)`为随机数发生器播种。重复地调用`rand()`,最终产生所有的在 $[0, \text{RAND\_MAX}]$ 间的整数,但次序是混合的。用于为随机数发生器播种的值决定了在混合次序中函数`rand()`产生随机数的开始点。如果我们用`time()`产生的值作为种子值,那么无论何时运行程序,种子都是不同的,这会使得产生的数集是不同的。重复地运行这个程序,每次你应该看到不同的数集。是这样吗?

7. 在上面的练习中,我们使用了代码

```

seed = time(NULL);
srand(seed);

```

大多数程序员可能不使用这两行,而使用

```
srand(time(NULL));
```

像这样改编程序,编译并执行它,看它的行为是否还和从前一样。(应该一样。)

8. 研究一下,怎样编写你自己的随机数发生器,有很多方法能做到这点。如果你不关心统计上的可靠性,任务是容易完成的(但你也要进行检查)。理解随机数发生器的理论比它为它编码更难。

9. 在4.3节“return语句”中,我们曾经编写了一个计算两个整数中最小值的交互式程序,用类型`double`重新编写程序。在测试了你的程序并感到满意后,修改它,使它能计算四个数中的最大和最小值。

10. 设 $n_0$ 是一个给定的正整数。对于 $i=0,1,2,\dots$ 定义:

若 $n_i$ 是偶数,则 $n_{i+1} = n_i/2$ ;

若 $n_i$ 是奇数,则 $n_{i+1} = 3n_i + 1$ ;

若 $n_i$ 是1,则序列结束。

把用这种方法产生的数称为冰雹(hailstone)。编写产生一些冰雹的程序,你的程序应该用函数

```
void hailstones(int n);
```

计算并显示由 $n$ 产生的序列。你的程序的输出看起来可能如下:

Hailstones generated by 77:

77	232	116	58	29	88
44	22	11	34	17	52
26	13	40	20	10	5
16	8	4	2	1	

Number of hailstones generated: 23

你可能发现，你所产生的序列都是有限的。一般而言，这是否是真的还仍然是未决的问题。提示：如果程序的行为不端，就用类型为long的变量代替类型为int的变量（请参见第6章“基本数据类型”）。

11. 如果你在一个单文件中以main()开始编程，则函数原型的列表要出现在main()之前。如果你在多个文件中以main()开始编程，则函数原型的列表要出现在.h文件的底部。这些文件提供的函数原型应该是一致的，且可以重复。在上一个练习中你编写的程序中增加下列函数原型：

```
void hailstones(int);
void hailstones(int k);
void hailstones(int n);
void hailstones(int nn);
```

你的编译器工作得会很顺利，是这样吗？

12. 编写一个显示前n个素数的程序，n由用户输入。当你执行你的程序时，你应该在屏幕上看到如下的显示信息：

```
PRIMES WILL BE PRINTED.
How many do you want to see? 3000

1:      2
2:      3
3:      5
4:      7
5:     11
.....
25:     97
26:    101
.....
2998: 27431
2999: 27437
3000: 27449
```

在程序自己的目录里以文件primes.h、main.c和is\_prime.c编程。下面是程序的一部分：

文件primes.h的内容：

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

int is_prime(int n);
```

文件is\_prime.c的内容：

```
#include "primes.h"

int is_prime(int n)
{
    int k, limit;

    if (n == 2)
        return 1;
    if (n % 2 == 0)
        return 0;
    limit = n / 2;
    for (k = 3; k <= limit; k += 2)
        if (n % k == 0)
            return 0;
    return 1;
}
```

请解释is\_prime()是如何工作的。提示：用手工做一些模拟。通过在main.c中编写



main()完成这个程序。有168个小于1000的素数，你的程序的计算结果是这样吗？小于10000的素数有多少个？

13. 通过让用户输入要显示的素数个数和开始的索引，仿照上一个练习重新编写一个程序，你的程序的输出应该如下：

```
PRIMES WILL BE PRINTED.
```

```
How many do you want to see? 33
```

```
Beginning at what index? 700
```

```
700: 5279
701: 5281
702: 5297
.....
731: 5527
732: 5531
```

14. (高级) 在1972年，德国数学家和天文学家Karl Friedrich Gauss在15岁时推测出了著名的素数定理(Prime Number Theorem)：让 $\pi(x)$ 代表小于或等于 $x$ 的素数的个数，那么 $\pi(x)$ 渐进于 $x/\log(x)$ ，即随着 $x$ 变为无穷大，商 $\pi(x)/(x/\log(x))$ 趋于1。修改你在练习12编写的程序，研究这个商的极限值。你的程序的输出看起来应该如下：

```
PRIME NUMBER THEOREM:
```

```
lim (pi(x) / (x / log(x))) = 1
x -> inf
```

```
where pi(x) is the number of primes
less than or equal to x.
```

```
How many primes do you want to consider? 3000
```

x	pi(x)	pi(x) / (x / log(x))
1000	168	1.160502886868999
2000	303	1.151536722620625
3000	430	1.147579351363202
.....		
27000	2961	1.118993938566849
27449	3000	1.116989873919079

你能观察到，收敛是很慢的。由于这个原因，你应该让 $x$ 步增为500或1000而不是1（最后一行除外，其中你应该让 $x$ 的值为相应的所要求的素数的个数）。

15. 1742年哥德巴赫做了如下的猜想：每一个大于6的偶整数都是两个奇数之和。至今为止，这个猜想还没有被证明成立或不成立。计算机已经被广泛地用于证明这个猜想，但还没有发现反例。编写一个程序证明对于在符号常量BEGIN和END之间的偶整数这一猜想成立。例如，如果你写

```
#define BEGIN 700
#define END 1100
```

那么你的程序的输出看起来应该如下：

```
GOLDBACH'S CONJECTURE:
```

```
Every even number n > 6 is the sum of two odd p
```

```
700 = 17 + 683
702 = 11 + 691
704 = 3 + 701
.....
1098 = 5 + 1093
1100 = 3 + 1097
```

提示：使用练习12中给出的函数is\_prime()。

16. 在本练习中，我们要修改用随机数发生器rand()模拟反复投掷硬币的程序。程序如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>      /* for time() */

int main(void)
{
    int    count_heads = 0, count_tails = 0, i, n, val;

    srand(time(NULL)); /* seed the rand. no. generator */
    printf("\n%s",
        "SIMULATED COIN TOSSING:\n"
        "How many times? ");
    if (scanf("%d", &n) != 1) {
        printf("INPUT ERROR: Bye!\n\n");
        exit(1);
    }

    if (n < 1) {
        printf("Bye!\n\n");
        exit(1);
    }

    for (i = 0; i < n; ++i) {
        val = rand() % 2;
        (val == 1) ? ++count_heads : ++count_tails;
        if (i % 10 == 0)
            putchar('\n');
        printf("%7s", (val == 1) ? "heads" : "tails");
    }
    printf("\n\n%s%d\n%s%d\n\n",
        "Number of heads: ", count_heads,
        "Number of tails: ", count_tails);
    return 0;
}
```

注意条件运算符?:的用法（请参见3.19节“条件运算符”）。还要注意格式%7s用于显示长度为7个字符的串，如果你把它改成格式%15s会发生什么情况？不调用srand()，运行时每次程序产生相同的输出。如果你把对srand()的调用作为for循环体的第一条语句，会发生什么？请解释程序的行为为什么会有如此的显著不同。（你感到惊讶吗？）正确的编程实践是在一个程序中仅为随机数发生器播种一次。

17. 在下述程序中，因为main()调用f()，f()调用g()，g()调用main()，所以main()是递归的。

```
#include <stdio.h>

void    f(void);
void    g(void);

int main(void)
{
    static int    count = 0;

    printf("Hello from main()\n");
    if (++count <= 2)
        f();
    return 0;
}

void f(void)
{
    g();
}
```

```

    printf("Hello from f()\n");
    g();
}

void g(void)
{
    printf("Hello from g()\n");
    main();
}

```

这是一个间接递归的例子。首先，写出你认为应该显示出的信息，然后把main()中的if语句改变成如下的语句：

```

if (++count <= 2) {
    f();
    g();
}

```

先写出你认为应该显示出的信息，然后执行程序，进行检查。当心：即使对程序的改变很小，看显示出的信息也是非常困难的。

18. C++: 用cout代替printf()重新编写上一个练习中的程序。执行时，程序的行为应该和前面一样。

19. C++: 编写一个重载函数print()集，显示带有精细信息的数据。下面是一个例子：

```

void print(double x)
{
    cout << "double x = " << x << endl;
}

```

用至少三种类型，编写该程序，并执行它。

20. Java: 在Java中，把所有的函数看作是方法，方法必须驻留在类中。在C中，函数是全局的或外部的，驻留在文件中。在C中，通过编译文件的内容完成编译工作。在Java中，通过编译类完成编译工作（当然，类也驻留在文件中）。在一些方面，C函数类似于Java的静态方法。

```

// Message2.java: method parameter use - Java by Dissection
page 98.
class Message2 {
    public static void main(String[] args) {
        System.out.println("HELLO DEBRA!");
        printMessage(5); //actual argument is 5
        System.out.println("Goodbye.");
    }
    static void printMessage(int howManyTimes) {
        //formal parameter is howManyTimes
        System.out.println("A message for you: ");
        for (int i = 0; i < howManyTimes; i++)
            System.out.println("Have a nice day!\n");
    }
}

```

用C重新编写这个程序。改变代码，使得它能接受应该显示的信息的次数的输入。如果你知道怎样改变程序，还要修改程序使得它能请求要显示的名字，通过输入“LAURA”测试它。

21. Java: 如下是Java的一段简单的递归程序。它要计算什么函数？把它转换成递归的C函数。把它转换成循环的C函数。在你的机器上递归或循环哪一个运行得更有效？

```

class RecursionExercise {
    public static void main(String[] args) {

```

```
        for (int i = 0; i < 10; i++)
            System.out.println(recurse(i));
    }
    static long recurse(long n) {
        if (n <= 0)
            return 1;
        else
            return 2 * recurse(n - 1);
    }
}
```

注意C和Java都有用于特别大的整数的long类型。在你的机器上，C和Java的long类型的范围相同吗？要记住，对于基本类型，C的表示是与系统相关的。

## 第5章 字符处理

本章中，我们介绍一些涉及到字符处理的基本概念。我们在本章中讨论如何在机器中存储和操纵字符，如何把字符按小整数处理，以及如何通过一定的标准头文件对字符进行使用。为了说明这些想法，我们要给出一些简单的完成这些工作的字符处理程序，这些示例程序使用字符输入/输出宏`getchar()`和`putchar()`。对于试图掌握一种新语言的人来讲，把数据输入到机器或从机器输出数据是应尽早熟悉的技能。

本章中将涵盖一些重要的概念，并解释符号常量`EOF`的用法。当`getchar()`检测到文件尾标志时，它返回值`EOF`，程序员利用这个值能检测到何时达到文件尾。本章将详细地解释头文件`ctype.h`的用法，这个头文件向程序员提供了一组用于处理字符数据的宏。程序员可以像使用函数一样使用宏。像`ctype.h`这样的系统头文件的使用使得程序员可以编写可移植的代码。本章提出的像`caps`这样的程序是很简单的，但它能说明对字符处理的基本思想。

### 5.1 数据类型char

类型`char`是C语言的基本类型之一，该类型的常量和变量用于表示字符。每个字符在机器中存储占一个字节。我们假设一个字符由8个二进制位组成，它能存储 $2^8$ 即256个不同的值。

当一个字符存储在一个字节中时，可以把该字节中的内容看作是一个字符或一个小的整数。虽然在一个字节中能存储256个不同的值，但仅其中的一部分能代表实际可打印的字符。可打印的字符包括小写字母、大写字母、数字、标点符号和`+`、`*`和`%`这样的特殊字符。该字符集还包括间隔字符：空格、跳格和换行。非打印字符的例子有换行和报警字符或响铃。在本章中我们要说明响铃的用法。

要把字符常量写在单引号之间，例如，`'a'`、`'b'`和`'c'`。类型为`char`的变量的典型声明是

```
char c;
```

可以对字符变量进行初始化，如下是一个例子：

```
char c1 = 'A', c2 = 'B', c3 = '';
```

按照一定的编码，一个字符存储在内存的一个字节中。大多数机器使用ASCII或EBCDIC字符码，对于其他编码，编码数是不同的，但思想是类似的。在附录E“ASCII字符码”中给出了ASCII表。

在C中，一个字符有一个对应于ASCII编码的整数值，下表给出了一些例子。

观察一下，在表示数字的字符常量的值和内在的整数之间没有什么特殊的关系。也就是说，`'7'`的值不是7。像`'a'`、`'b'`和`'c'`等这样的值按次序出现是一个重要的性质，它便于对字符、字和行等按字典次序排序。

在函数`printf()`和`scanf()`中，`%c`用于指明字符的格式。例如，语句

```
printf( "%c", 'a' ); /*a is printed*/
```

按字符格式显示字符常量`'a'`。类似地，

一些字符常量及它们的整数ASCII值					
小写字母的 ASCII值	'a'	'b'	'c'	...	'z'
	97	98	99	...	112
大写字母的 ASCII值	'A'	'B'	'C'	...	'Z'
	65	66	67	...	90
数字的 ASCII值	'0'	'1'	'2'	...	'9'
	48	49	50	...	57
其他 ASCII值	'&'	'*'	'+'	...	
	38	42	43	...	

```
printf("%c%c%c", 'A', 'B', 'C'); /* ABC is printed */
```

显示ABC。

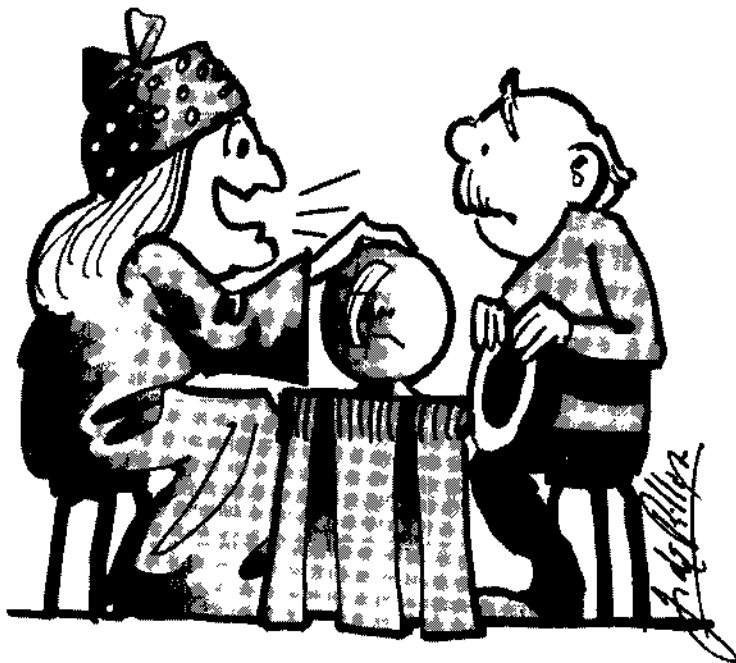
能把类型为char的常量或变量按小整数处理。语句

```
printf("%d", 'a'); /* 97 is printed */
```

按十进制整数的格式显示字符常量'a'的值，这样，就显示了97。另一方面，语句

```
printf("%c", 97); /* a is printed */
```

按字符格式显示十进制整数常量97的值，这样，就显示了a。



“我看你是个好人，再给5美元，我就给你它的ASCII码。”

一些非打印和难以打印的字符需要转义序列(escape sequence)。例如，在程序中把换行符写为'\n'，即使是它由两个字符\和n来描述，它也表示一个ASCII字符。把反斜杠符号\称为转义字符(escape character)，它用于“转义”跟在它后边的字符的通常含义。下表包含一些非打印字符和难以打印字符。

非打印字符和难以打印字符		
字符名	C中的写法	整数值
报警	\a	7
反斜杠	\\	92
退格	\b	8
回车	\r	13
双引号	\"	34
走纸	\f	12
横向跳格	\t	9
换行	\n	10
空字符	\0	0
单引号	\'	39
纵向跳格	\v	11

如果在一个串中把双引号"用作一个字符，必须要对它转义，否则它会终结该字符串。下边是一个例子：

```
printf("\'ABC\");    /*"ABC" is printed */
```

然而，在单引号中可以写"\'"，虽然\'也是可接受的。一般地，对普通字符转义没有什么效果。



“你所要使用的一些字符需要用反斜杠字符转义，‘A’与‘\A’是不一样的！”

在一个串中，单引号是一个普通的字符：

```
printf(" \'ABC\' ");    /* \'ABC\' is printed */
```

写一个字符常量的另一种方法是依靠1、2或3个八进制数转义序列，例如，'007'。这是一个警告字符\a或听得见的响铃，也可以把它写为'\07'或'\7'，但不能写为'7'。

## 5.2 getchar()和putchar()的用法

系统提供了getchar()和putchar()，用于输入和输出字符，它们是定义在stdio.h中的宏。要从键盘上读一个字符，就使用getchar()；要在屏幕上写一个字符，就使用putchar()。例如，在屏幕上显示

```
She sells sea shells by the seashore
```

的程序如下：

```
#include <stdio.h>

int main(void)
{
    putchar('S');
    putchar('h');
    putchar('e');
    putchar(' ');
    .....
    putchar('e');
    putchar('.');
    putchar('\n');
    return 0;
}
```

当然，这是一种另人乏味的完成该项工作的方法，用printf()语句会容易得多。

在下面的程序中，getchar()从输入流（键盘）中获得字符，并把它赋值给变量c，然后用putchar()在屏幕上显示该字符两次。

```
#include <stdio.h>

int main(void)
{
    char c;

    while (1) {
        c = getchar();
        putchar(c);
        putchar(c);
    }
    return 0;
}
```

注意，变量c的类型是char。在该程序的下一个版本中，我们要改变它。因为1是非零值，作为一个表达式它总是真。因此，结构

```
while (1) {
    .....
}
```

是一个无限循环。停止程序的惟一方法是用中断，在我们的系统中是用Ctrl+c中断程序。

由于一些原因，上一个程序实际上是不可接受的。让我们重新编写它，把它称为新版本的dbl\_out。

```
#include <stdio.h>

int main(void)
{
    int c;

    while ((c = getchar()) != EOF) {
        putchar(c);
        putchar(c);
    }
    return 0;
}
```

#### 对程序dbl\_out的解析

- #include <stdio.h>

用#开始的行是预处理指令。这样的行由预处理器处理，预处理指令的形式为：

```
#include <filename>
```

这使得预处理器在把代码传递到编译器之前，把命名的文件的拷贝引入到源代码的相应



位置上。stdio.h前后的三角括号告诉编译器在与系统相关的“通常位置”处寻找该文件。文件stdio.h是C系统提供的标准头文件，通常在使用一定的标准输入/输出结构的函数中引用它。该头文件的其中一行是

```
#define EOF (-1)
```

标识符EOF是“文件尾”的助记符号，实际用于文件尾的标记的是与系统相关的。虽然经常使用类型为int的值-1，但不同系统使用的值可能是不同的。通过引入文件stdio.h并使用符号常量EOF，我们可使得程序可移植：可把源文件移到不同的系统并且可以不做修改就运行。

```
int c;
```

在程序中已经把变量c的类型声明为int，而不是char。无论用什么作为文件尾的标记，都不能用表示字符的值。因为c的类型是int，它能保留所有的可能字符值，也能保留特殊值EOF。虽然人们通常可以把char看作是很短的int类型，但也可以把int看作是很长的char类型。

```
while ((c = getchar()) != EOF) {
```

表达式(c = getchar()) != EOF由两部分组成。子表达式c = getchar()从键盘上得到一个值，并赋值给变量c，该子表达式的值也是它，符号!=代表“不等于”运算符。只要子表达式c = getchar()的值不等于EOF，就执行while的循环体。要结束循环，我们必须在键盘上输入文件尾标记，然后操作系统告诉getchar()已经到达了文件尾，getchar()返回EOF。在键盘上如何输入文件尾的值是与系统相关的。在UNIX中，文件尾的值通常是回车后跟一个Ctrl+d。在MS-DOS中，用户键入是Ctrl+z。

```
(c = getchar()) != EOF
```

子表达式c = getchar()前后的括号是必须的。假设我们输入

```
c = getchar() != EOF
```

由于运算符的优先级，该表达式等价于

```
c = (getchar() != EOF)
```

该语句从输入流得到一个字符，测试该字符是否等于EOF，并把测试结果(0或1)赋值给变量c(请参见练习10)。 ■

也可以用简单递归来编写该程序。在大多数情况下，用简单递归代替基本的while循环是很容易的，其中while循环测试用于终止递归。

```
#include <stdio.h>

void dbl_out(int c)
{
    if ((c != EOF) {
        putchar(c);
        putchar(c);
        dbl_out(getchar())
    }
}

int main(void)
{
    dbl_out(getchar());
    return 0;
}
```

### 5.3 例子：大写

每个字符都有一个潜在的整数值，在大多数C系统中，它是7位ASCII表示的数字值。例如，字符常量'a'的值是97。如果把字符看作是小整数，那么对字符进行算术运算是有意义的。由于在小写和大写字母表中字符的值是有次序的，表达式'a'+1的值是'b'，'b'+1的值是'c'，'Z'-'A'的值是25。此外，'A'-'a'的值和'B'-'b'的值是相同的，与'C'-'c'的值也是相同的。由于这个原因，如果把一个小写字母的值赋给变量c，那么表达式c+'A'-'a'的值是该小写字母对应的大写字母的值。这些想法将出现在下一个程序中，它把所有的小写字母变为大写的，并显示两次换行符。

```
/* Capitalize lowercase letters and double space. */
#include <stdio.h>

int main(void)
{
    int c;

    while ((c = getchar()) != EOF)
        if ('a' <= c && c <= 'z')
            putchar(c + 'A' - 'a');
        else if (c == '\n') {
            putchar('\n');
            putchar('\n');
        }
        else
            putchar(c);
    return 0;
}
```

#### 对程序caps的解析

- while ((c = getchar()) != EOF)

宏getchar() 获取一个字符，并把它赋值给变量c。只要变量c的值不是EOF，就执行while循环体。

- if ('a' <= c && c <= 'z')  
    putchar(c + 'A' - 'a');

按照运算符的优先级，表达式'a'<=c && c<='z'和表达式('a'<=c)&& (c<='z')是等价的。符号<=代表运算符“小于或等于”。子表达式'a'<=c测试值'a'是否小于或等于c的值，子表达式c<='z'测试c的值是否小于或等于值'z'。符号&&代表“逻辑与”运算符。如果这两个子表达式都是真的，那么表达式'a'<=c && c<='z'就是真的，否则它是假的。这样，如果当且仅当c是一个小写字母，作为一个整体该表达式是真的。如果这个表达式是真的，那么就执行语句

```
putchar(c + 'A' - 'a');
```

该语句显示相应的大写字母。

- else if (c == '\n') {  
    putchar('\n');  
    putchar('\n');  
}

符号==代表“相等”运算符。如果c不是小写字母，就测试它是否是换行符。如果是，就显示两个换行符。

- else  
    putchar(c);

如果变量c不是小写字母也不是换行符，那么就显示c的值所对应的字符。else总是属于最靠近它的上一个if。

虽然程序caps可移植到任何ASCII机器上，但它在EBCDIC机器上工作是不正常的。这是因为按EBCDIC码大写字母不是都连续的。下面的程序caps在所有的机器上工作都正常：

```
/* Capitalize lowercase letters and double space. */
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int c;

    while ((c = getchar()) != EOF)
        if (islower(c))
            putchar(toupper(c));
        else if (c == '\n') {
            putchar('\n');
            putchar('\n');
        }
        else
            putchar(c);
    return 0;
}
```

#### 对可移植的程序caps的解析

- #include <stdio.h>  
  #include <ctype.h>

文件ctype.h和stdio.h都是C系统提供的标准头文件。ctype.h含有处理字符时经常使用的宏和函数原型。宏是由预处理器展开的代码。在附录B“预处理器”中，我们详细地解释了宏是如何工作的。按本章的目的，我们把ctype.h中的宏处理为函数。虽然宏和函数在技术上是不同的，但对它们的使用方式是类似的。程序caps使用了islower()和toupper()。由于ctype.h含有islower()的宏定义和toupper()的函数原型，所以引入了这个头文件。

- while ((c = getchar()) != EOF)  
    if (islower(c))  
        putchar(toupper(c));

从输入流读取一个字符并赋值给c。只要变量c的值不是EOF，就执行while循环体。在ctype.h中定义了宏islower()。如果c是小写字母，那么islower()就有非零值；否则它的值就是零。在ctype.h中给出了toupper()的函数原型。如果c是小写字母，那么toupper(c)的值就是对应的大写字母。因此，if语句能测试c的值是否是小写字母，如果是，就在屏幕上显示对应的大写字母。要注意，调用isupper(c)或toupper(c)不会改变c本身存储的值。

C语言的初学者不需要精确地知道怎样执行ctype.h中的宏。像标准库中的函数printf()和scanf()一样，可把这些宏看作是由系统提供的资源。要记住的关键点是，通过使用这些函数和宏，你可以编写运行在任何符合ANSI环境的可移植的代码。

为什么学习c + 'A' - 'a'这样的结构？一些C代码是为ASCII环境而写的，即使结构不符合良好编程惯例，人们一般也能理解它。因为程序员必须学会读代码和写代码，所以就应该掌握这种特殊的结构。要使代码可移植，在适宜之处要使用ctype.h中的宏，这是一种良好

的编程习惯。

## 5.4 ctype.h中的宏

系统提供了标准头文件ctype.h, 它含有一组用于测试字符的宏和一组用于转换字符的函数原型。通过预处理指令#include < ctype.h>使用它。下表中的宏用于测试字符, 这些宏都需要类型为int的参数, 并返回类型为int的非零值(true)或零(false)。

字符宏	
宏	返回非零(真)值, 如果
isalpha(c)	c是一个字母
isupper(c)	c是一个大写字母
islower(c)	c是一个小写字母
isdigit(c)	c是一个数字
isalnum(c)	c是一个字母或数字
isxdigit(c)	c是一个十六进制数字
isspace(c)	c是一个空白字符
ispunct(c)	c是一个标点字符
isprint(c)	c是一个可打印字符
isgraph(c)	c是可打印的, 但不是空格
iscntrl(c)	c是控制字符
isascii(c)	c是ASCII码

下表中, 我们列出了标准库中的函数toupper()和tolower(), 以及宏toascii(), 它们都在ctype.h中, 它们都需要类型为int的参数, 并且返回值也是int型的。要注意, 存储在内存中的c的值没有发生变化。

字符宏和函数	
函数或宏	作用
toupper(c)	把c从小写变大写
tolower(c)	把c从大写变小写
toascii(c)	把c变为ASCII码

## 5.5 问题求解: 重复字符

使用带有适当参数的函数解决问题是一个很好的想法, 这是综合(generalization)的一个方面。人们经常在简单的具体情况下解决特殊的问题。下面的例子把字母C显示三次。

```
printf("%c%c%c", 'C', 'C', 'C');
```

现在我们要用另一种不同的方法显示4个C和6个b。通过为显示的字符和显示次数设置两个参数, 我们能解决更一般的问题。我们编写一个函数来完成此项工作:

```
void repeat(char c, int how_many);
```

一旦编好这个函数, 并且代码正确, 我们就可以为很多目的复用这个函数。的确, 已经开发出的很多标准库函数都是常用的具有一般性的操作。

5.2节中的程序dbl\_out中, 我们说明了如何读入每个字符并各显示两次。在这里, 我们要

编写一个显示给定字符how\_many次的程序，以对上述想法通用化。

```
void repeat(char c, int how_many)
{
    int i;

    for (i = 0; i < how_many; ++i)
        putchar(c);
}
```

注意，变量c被声明为char型的，而不是int型的。由于在本函数中没有对EOF进行测试，因而不需要声明c是int型的。假设我们用语句

```
repeat('B' - 1, 2);
```

调用这个函数，这个函数调用的参数是'B' - 1和2，这个两个参数的值被传递给相应的函数的形参。这个函数调用的作用是显示字母'A'两次。下面是用于测试repeat()的main()函数：

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

void repeat(char, int);

int main(void)
{
    int i;
    const char alert = '\a', c = 'A';

    repeat('B' - 1, 2);
    putchar(' ');
    for (i = 0; i < 10; ++i) {
        repeat(c + i, i);
        putchar(' ');
    }
    repeat(alert, 100);
    putchar('\n');
    return 0;
}
```

注意，我们使用了类型说明符const来指明不能改变变量alert和c。编译并运行该程序，在屏幕上会出现如下信息：

```
AA B CC DDD EEEE FFFFF GGGGGG HHHHHH IIIIIIII JJJJJJJJ
```

函数repeat()能用于在屏幕上绘制简单的图。在练习8中，我们要说明怎样用repeat()绘制一个三角形，并把怎样绘制菱形留作练习。

## 5.6 问题求解：对单词计数

很多计算都是重复的，有时重复也基于计数。例如，一个食谱上说“搅动40秒”，我们就要数到40。有时重复等到一些条件发生变化。例如，一个食谱上说“搅动，直到变为橙色为止”。寻找在逻辑上终止重复的特征是解决问题的一个重要方法。在对字符的处理中，我们经常要寻找文件尾，作为终止的条件。

假设我们要对从键盘上输入的词进行计数。用自顶向下设计，我们把该问题分解成一些小问题。为了做到这一点，我们需要知道对词的定义，也需要知道何时结束任务。对于这个问题，我们假设词由空格隔开，这样任何词都是一个不含空格的连续的字符串。通常，当遇到文件尾标记时，我们结束对字符的处理。该程序的核心是一个函数found\_next\_word()，它检测词。我们要较详细地解释这个函数。

```

#include <stdio.h>
#include <ctype.h>

int found_next_word(void);

int main(void)
{
    int word_count = 0;

    while (found_next_word() == 1)
        ++word_count;
    printf("Number of words = %d\n\n", word_count);
    return 0;
}

int found_next_word(void)
{
    int c;

    while (isspace(c = getchar()))
        ; /* skip white space */
    if (c != EOF) { /* found a word */
        while ((c = getchar()) != EOF && !isspace(c))
            ; /* skip all except EOF and white space */
        return 1;
    }
    return 0;
}

```

#### 对程序word\_count的解析

- `int word_count = 0;`

把类型`int`的变量`word_count`初始化为0。

- `while (found_next_word() == 1)`  
    `++word_count;`

只要`found_next_word()`的返回值是1,就执行`while`循环体,对`word_count`加1。

- `printf("Number of words = %d\n\n", word_count);`

在退出程序之前,显示已找到的词的个数。

- `int found_next_word()`  
    {  
        `int c;`

这是`found_next_word()`的函数定义的开始部分。在该函数的参数列表部分没有参数,在函数体中,声明了类型为`int`的变量`c`。虽然我们要用`c`接收字符值,但我们把`c`声明为`int`型的,而不是`char`型的。最终`c`的值是`EOF`,在一些系统中,该值可能不适合于类型`char`。

- `while (isspace(c = getchar()))`  
    ;  
        `/* skip white space */`

从输入流读入字符,并把它赋值给`c`,子表达式`c=getchar()`的值也是这个值。只要这个值是空白字符,就执行`while`循环体,然而`while`循环体只是一个空语句。这样,`while`循环的作用就是跳过空格。注意,空语句本身也占一行,良好的编程习惯要求这样做。如果我们写

```
while (isspace(c = getchar()));
```

那么空语句的可视性就变差了。

- `if (c != EOF) {`      `/* found a word */`  
    `while ((c = getchar()) != EOF && !isspace(c))`  
        ;  
        `/* skip all except EOF and white space */`  
    `return 1;`  
}

在跳过空格后，c的值是EOF或词的首字母。如果c的值不是EOF，那么就找到了一个词。while循环中的测试表达式由三部分组成。首先，从输入流读入字符，并把它赋值给c，子表达式c = getchar()的值也是这个值；然后测试这个值是否为EOF。如果是，就不执行while循环体，控制传递给下一个语句，如果不是EOF，就测试这个值是否为空格，如果是，就不执行while循环体，控制传递给下一个语句。如果不是空格，就执行while循环体，而这个循环体只是一个空语句，因此，while循环体的作用就是跳过除了EOF和空格外的所有输入，也就是说，现在已经跳过了已经找到的词。

- return 1;

在找到并跳过了一个词后，返回值1。

- return 0;

如果没有找到词，就返回值0。 ■

## 5.7 风格

简单的字符变量常为标识符c或用c开头的标识符，例如，c1、c2和c3。专业的C程序员趋于使用简洁的表示，使用ch、chr或ch\_var作为类型为char的变量标识符通常更为清晰。处理字符的函数和宏经常用char作为其名字的一部分，或者用字母c作为名字的结尾。例如，getchar()和putchar()，以及我们在第13章“输入/输出和文件”看到的getc()和putc()。ctype.h中选择的宏标识符是有启发性的。像isalpha()和isupper()这样的具有真/假值的那些宏的名字都以is开头；像toupper()这样的在ctype.h中具有原型的那些能改变字符的值的函数，其名字都以to开头。选择适当的标识符名对于可读性和文档化来说是至关重要的。

对于字符处理任务，我们可以用getchar()和scanf()读入字符，类似地，我们用putchar()和printf()输出字符。在很多情况下，选择哪一个凭个人感觉，然而，如果要处理大量的字符，则使用getchar()和putchar()以及标准头文件stdio.h能加速编码，这是因为在stdio.h中getchar()和putchar()被作为宏来处理。像我们在附录B“预处理器”中看到的那样，宏是一种能用于避免函数调用的代码替换机制。

在putchar()和printf()间的一个不同是，putchar()返回类型为int的写到输出流的字符值，而printf()返回已显示的字符的个数，这规定了putchar()的用法。

一个常见的C编程惯例是在控制while或for循环的表达式中同时完成赋值和测试，在处理字符的代码中人们经常会见到这种作法。例如，代码

```
while ((c = getchar()) != EOF) {
    ....
}
```

就使用了这种习惯作法。与此相对照，我们可以写

```
c = getchar();
while (c != EOF) {
    ....
    c = getchar();
}
```

但是现在，如果循环体较长，那么影响循环控制的最后一个语句离测试表达式很远。另一方面，可以把结构

```
while (isspace(c = getchar()))
    ; /* skip white space */
```

编写为更好的结构：

```
c = getchar();
while (isspace(c))
    c = getchar();
```

此处的循环体很短，事实上，如果我们把所有的控制都放在顶部，循环体就是空的了。使用哪一种形式在很大程度上是凭你的喜好。

## 5.8 常见的编程错误

我们已经解释了如果程序用变量读入字符并测试值EOF，那么变量的类型应该是int而不是char，部分地讲这是基于对可移植性的考虑。如果把值EOF赋值给类型为char的变量，而不是类型为int的变量，一些C系统不会把EOF作为文件尾标志。试一下如下的代码：

```
char    c;                                /* wrong */

while ((c = getchar()) != EOF)
    putchar(c);
```

在你的机器上它工作得可能不正常。即使工作正常，当测试EOF时，它也不用char。现在你不可能移植这样的代码，但在将来你或许能移植这样的代码。

假设我们有一个格式为两倍间距或三倍间距的文本文件，我们的任务是复制该文件，但要把格式转换成单倍间距。例如，我们要把双倍间距的文件转换成单倍间距的文件。为此编制程序如下：

```
/* Copy stdin to stdout, except single space only. */

#include <stdio.h>

int main(void)
{
    int    c, last_c = '\0';

    while ((c = getchar()) != EOF) {
        if (c == '\n') {
            if (last_c != '\n')
                putchar('\n');
        }
        else
            putchar(c);
        last_c = c;
    }
    return 0;
}
```

在程序的开头，把变量last\_c初始化为空字符，但随后该变量保存从输入流（键盘）读入的字符，在此处对空字符的使用没有什么特殊之处，我们只是需要用一些字符（换行符除外）初始化它。在该程序中可能会出现两个常见的错误。假设我们输入了

```
if (c = '\n') {
```

此处使用了=而不是==。因为表达式c='\n'总是真，else部分总也执行不到，因而程序的问题就严重了。如果我们输入了

```
if (c == '\n')
    if (last_c != '\n')
        putchar(c);
else
    putchar(c);
```



就会出现另一个错误。缩排显示出了我们想要的逻辑，但实际上没有得到我们想要的结果，因为else语句总是属于离它最近的上一个if，这样上述代码等价于

```
if (c == '\n')
    if (last_c != '\n')
        putchar(c);
    else
        putchar(c);
```

这明显是错误的。程序员必须要牢记编译器看到的仅是字符流。

## 5.9 系统考虑

我们已经讨论了可重定向的标准输入与输出（请参见1.11.4节“对输入和输出的重定向”）。你应该复习一下相关的内容，并用程序caps试一下重定向。创建一个名为input的文件，并在其中放一些文本。试一下如下的命令：

```
caps
caps < input
caps > output
caps < input > output
```

大多数操作系统都有把一个文件拷贝到另一个文件的命令。在MS-DOS中该命令是copy，在UNIX中该命令是cp。可以把下面的程序认为是这样的命令的一个简本：

```
#include <stdio.h>

int main(void)
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
    return 0;
}
```

如果我们编译该程序，并把执行的代码放在my\_copy中，那么命令

```
my_copy < infile > outfile
```

就把infile的内容拷贝到outfile。

如果c中保存的是一个小写字母的值，那么toupper(c)就返回相应的大写字母的值。ANSI C标准表明：如果c中保存的不是一个小写字母的值，那么toupper(c)返回的值就是参数的值。因此，要把所有的小写字母变为大写字母，我们要使用如下的代码：

```
while ((c = getchar()) != EOF)
    putchar(toupper(c));
```

然而，很多较老的C系统把toupper()作为一个宏，这样上述代码会失败。在这样的系统中，必须对c进行测试：

```
while ((c = getchar()) != EOF) /* traditional C code */
    if (islower(c))
        putchar(toupper(c));
    else
        putchar(c);
```

对tolower()的处理与此类似。

为了更有效地对字符进行处理，很多C系统提供了宏\_toupper()和\_tolower()。如果存在这些宏，那么它们就在ctype.h中。下面是一个使用它们的例子：

```
while ((c = getchar()) != EOF)
    if (_islower(c))
        putchar(_toupper(c));
    else
        putchar(c);
```

然而要注意，包括传统C和ANSI C在内的很多C系统不支持\_toupper()和\_tolower()。当心：如果你正在编写可移植的代码，使用toupper()和tolower()要谨慎。一些组织使用传统C，一些组织使用ANSI C，还有的混合使用传统C和ANSI C。

所有的编译器都把char处理为小整数。在大多数系统中（但不是全部），char的值的缺省范围是从-128到127。很多编译器提供了一个选项，利用该选项可把该范围变为0到255。对于需要存储大量数据而使用尽量少的磁盘空间的情况而言，该选项是很有用的。如果所有数据都处于0到255之间，那么用这个选项编译后的程序能操纵这些数据，并把数据存储为char型的，而不是int型的。我们在第6章“基本数据类型”将看到，存储char类型的数据比存储int类型的数据所用的空间要少。

## 5.10 转向C++

C++的输出被插入到一个在头文件iostream.h中声明的类型为ostream的对象中。在这个类中重载了运算符<<，以完成输出转换。把重载的左移运算符称为插入或放置运算符，该运算符是左结合的，返回类型为ostream&的值。相当于stdout的标准输出ostream是cout，相当于stderr的标准输出ostream是cerr。

执行下面的简单输出语句：

```
cout << "x = " << x << "\n";
```

其效果是，先在屏幕上显示一个有四个字符的串，后跟x的输出的适当表示，然后换行。其中的表示依赖于调用的<<的重载版本。

类ostream包含一些公共成员，例如：

```
ostream& operator<<(int i);
ostream& operator<<(long i);
ostream& operator<<(double x);
ostream& operator<<(char c);
ostream& operator<<(const char* s);
ostream& put(char c);
ostream& write(const char* p, int n);
ostream& flush();
```

成员函数put输出c的字符表示；write输出p指向的长度为n的串；flush强制输出流被写。因为它们都是成员函数，所以下述写法也是正确的：

```
cout.put('A'); //output A

char* str = "ABCDEFGHI";
cout.write(str + 2, 3); //output CDE
cout.flush(); //write contents of buffer
```

运算符<<用缺省的最小的字符个数作为输出长度。其结果可造成输出混乱，请看下面的例子：

```
int i = 8, j = 9;
cout << i << j; //confused: prints 89
cout << i << " " << j; //better: prints 8 9
cout << "i = " << i << " j = " << j; //best: i = 8 j = 9
```

我们已经使用了两种适当地分隔输出的模式。一种是用串分隔输出的值；另一种是用\n产

生的换行符和用t产生的跳格。我们也在流输出中用控制符控制输出格式。

控制符是一个对它所操纵的流产生特殊效果的值或函数。例如，在*iostream.h*中定义的endl就是一个简单的控制符，它输出一个换行，并刷新ostream。

```
x = 1;
cout << "x = " << x << endl;
```

执行上述代码会显示行

```
x = 1
```

另一个控制符flush刷新ostream，请看下行；

```
cout << "x = " << x << flush;
```

该行的执行效果几乎与上一个例子相同，但不预先产生一个换行。

控制符dec、hex和oct能用于改变整数的进制。缺省的进制是十进制。在有明确的改变之前，进制保持不变。

```
//Using different bases in integer I/O.
#include <iostream.h>

int main(void)
{
    int i = 10, j = 16, k = 24;

    cout << i << '\t' << j << '\t' << k << endl;
    cout << oct << i << '\t' << j << '\t' << k << endl;
    cout << hex << i << '\t' << j << '\t' << k << endl;
    cout << "Enter 3 integers, e.g. 11 11 12a" << endl;
    cin >> i >> hex >> j >> k;
    cout << dec << i << '\t' << j << '\t' << k << endl;
}
```

输出的结果如下：

```
10      16      24
12      20      30
a       10      18
Enter 3 integers, e.g. 11 11 12a
11      17      298
```

输出的最后一行是11，后跟17。其原因是把输入的第二个11被解释为十六进制，转换成十进制后为16+1。

上述的控制符可在*iostream.h*中找到，其他的控制符可在*iomanip.h*中找到。例如，setw(int width)是一个把为下一个格式化的I/O操作的缺省域宽改变为其参数值的控制符。下表主要列出了标准控制符，并给出各控制符的功能及其定义的位置。

C++ I/O控制符		
控制符	功 能	文 件
<u>endl</u>	输出换行并刷新	<i>iostream.h</i>
<u>ends</u>	在串中输出空	<i>iostream.h</i>
<u>flush</u>	刷新输出	<i>iostream.h</i>
<u>dec</u>	使用十进制	<i>iostream.h</i>
<u>hex</u>	使用十六进制	<i>iostream.h</i>
<u>oct</u>	使用八进制	<i>iostream.h</i>
<u>ws</u>	跳过输入中的空白字符	<i>iostream.h</i>
<u>setw(int)</u>	设置域宽	<i>iomanip.h</i>
<u>setfill(int)</u>	设置填充字符	<i>iomanip.h</i>
<u>setbase(int)</u>	设置进制格式	<i>iomanip.h</i>
<u>setprecision(int)</u>	设置浮点精度	<i>iomanip.h</i>
<u>setiosflags(long)</u>	设置格式位	<i>iomanip.h</i>
<u>resetiosflags(long)</u>	重置格式位	<i>iomanip.h</i>

## 小结

- 把一个字符按照它的ASCII编码存储在一个字节中，并认为它有一个相应的整数值。例如，字符常量'a'的值是97。
- 有非打印字符。这样的例子有警告字符（发声铃）'\a'和换行符'\n'。换行符广泛地用于格式化输出。
- 用宏getchar()和putchar()能很容易地实现字符的基本输入/输出。在标准头文件stdio.h中定义了这些宏，在某些方面可以把宏看成函数。
- 在用某些输入/输出结构时，应该引入系统头文件stdio.h。用预处理指令#include<stdio.h>来做到这一点。
- 在输入字符时，经常要测试文件尾标记。在程序中用符号常量EOF来做到这一点。
- 在系统的头文件stdio.h中定义了符号常量EOF。在大多数系统中，EOF的值是-1。
- 一些系统提供的宏和函数测试或转换字符值。通过引入系统头文件ctype.h，就可使用宏和这些函数的原型。

## 练习

1. 用getchar()和putchar()编写一个程序，从键盘上读入字符，并在屏幕上输出。读入的每个字符都应在屏幕上输出三次，后面再跟一个换行。应该忽略读入的换行。把所有的字符复制到屏幕上。

2. 用getchar()编写一个程序，从标准输入流（键盘）读入字符，直到遇到警戒字符#为止。程序要计算字母a、b和c出现的次数。

3. 改写上一个练习中的程序，要求读入字符，直到遇到EOF为止。用重定向测试程序。

4. 编写一个程序，从键盘上读入字符，并在屏幕上输出。把所有的元音字母按大写输出，非元音字母按小写输出。提示：编写一个测试一个字符是否为元音字母的函数isvowel()。在练习12中可复用本程序。

5. 退格符、换行符、空格和跳格符是可打印的吗？用ctype.h中的宏isprint()作为判断依据。

6. 编写一个编排文本文件格式的程序，要求大多数行含有约N个字符。程序用预处理指令

```
#define N 30
```

开始。对写入文件的字符计数，只要计数小于N，把读入的换行都变为空格。当计数等于N或大于N，把空格写为换行，并使计数为0。如果我们假设大多数词含有的字符个数少于10，那么用该程序产生的文件所包含的行都只有N到N+10个字符。打字员通常都遵循这样的算法。编译你的程序，并把可执行的编译结果放在reformat中。通过给出命令

```
reformat < text
```

用重定向测试该程序。当然，通过改变符号常量N的值，你可以改变行的长度。

7. 编写一个程序，使一个文本文件的所有行都缩进N个字符，这里的N是一个符号常量。

8. 可用函数repeat()在屏幕上绘制一个简单的图。例如，下述的程序可绘制一个三角形：

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

#define N 33
```

```

void repeat(char, int);

int main(void)
{
    char    c = 'X';
    int     i;

    for (i = 1; i < N; i += 2) {
        repeat(c, i);
        putchar('\n');
    }
    return 0;
}

```

编译并运行这个程序，以理解该程序的作用。编写一个类似的程序，在你的屏幕的中间显示一个菱形。

9. `putchar()` 和 `printf()` 间的一个不同是 `putchar()` 返回写到输出流的字符值（以 `int`），而 `printf()` 返回已显示的字符数。下述的代码显示什么？

```

for (putchar('0'); putchar('1'); putchar('2'))
    putchar('3');

```

下述代码有什么意义？请解释。

```

printf("%c%c%c\n", putchar('A'), putchar('B'),
        putchar('C'));

```

10. 请用下述循环把标准输入文件拷贝到标准输出文件。

```

while ((c = getchar()) != EOF)
    putchar(c);

```

假设我们故意地把内部的括号去掉，该循环就变为：

```

while (c = getchar() != EOF)
    putchar(c);

```

编写一个测试程序，看产生的效果是什么？如果输入文件有  $n$  个字符，那么会有  $n$  个字符被写到输出文件中吗？

11. 投掷硬币的正反面游戏需要用户输入0代表正面，输入1代表反面（请参见4.11节“模拟：正反面游戏”）。重新编写原程序，用户输入h代表正面，输入t代表反面。提示：你可以用带有格式%c的 `scanf()` 读入用户输入的字符，但要对空白字符进行处理，因为用户输入h或t总要在其后跟一个换行。

12. 古埃及人使用象形文字，不表示元音而仅表示辅音。如果英语中没有元音，你还能理解英语吗？为了进行此项实验，编写一个测试一个字符是否为元音的函数 `isvowel()`，用这个函数编写一个程序，从标准输入文件读入，并写到标准输出文件，同时要删除元音。对一个含有一段文本的文件使用重定向，测试你的程序。

13. 大多数操作系统都提供了数据压缩工具，使用压缩工具会缩短文本文件，使之占据较少的磁盘空间。这样的工具以两种方式工作，被压缩的文件不能再被压缩。编写一个程序 `crunch`，通过去掉所有附加的空白符（包括换行符）减少C源代码文件的尺寸。用命令

```
crunch < pgm.c > try_me.c
```

测试你的程序。编译并执行 `try_me.c` 中的代码，其效果与 `pgm.c` 一样吗？用一些.c文件测试你的程序。平均来讲，`crunch` 在空间上的降低能用百分率表达吗？为了回答这个问题，编

写一个对文件中的字符计数的程序。

14. 大多数系统都有“美化显示”的工具，这样的工具能把布局拙劣的C程序转换得更加易读。当把这样的工具应用到你在上一个练习中编写的程序上，显示的间距会更好。编写你自己的“美化显示”工具。用一个C程序作输入，产生的结果将增加空白符和换行符，以使得程序更易读。用以前的C代码，测试你的“美化显示”工具。

15. 我们已经给出了程序repeat的输出（请参见5.5节“问题求解：重复字符”）。下面又一次给出了这个输出：

```
AAAAA B CC DDD EEEE FFFFF GGGGGG HHHHHHH IJJJJJJJ
```

注意，在B前有两个空格，而随后的输出中空格都只有一个。请解释这是为什么？

16. C++：用C++重新编写程序caps（请参见5.3节“例子：大写”），如果可能，用递归代替while循环。

17. C++：用hex控制符编写一个程序，显示所有用十六进制表示的字符。

18. Java：如下是一个简单的Java程序，它从键盘上读入10个字符。把这个程序转换成C程序。

```
import tio.*;
import java.io.IOException;

public class TestChar {
    public static void main (String args[]) throws
    IOException
    {
        System.out.println("Read a Char:");
        for(int i = 0; i < 10; i++) {
            int c = Console.in.readChar();
            System.out.println(c + " in char " + (char)c);
        }
    }
}
```

在Java中，用两个字节（能存储16位）表示字符，这样就能表示所有主要国家的字母表。C用一个字节的ASCII码，这仅表示了标准美国字母表和键盘字符。

# 第6章 基本数据类型

在本章的一开始我们要简要地讨论声明和表达式，然后我们要对各个基本的数据类型进行详细地解释，并特别注意C怎样把字符处理成小整数。在具有不同类型运算数的表达式中，会发生某些隐式的转换。我们要解释转换规则，并讨论强制进行显式转换的类型转换运算符。

## 6.1 声明和表达式

变量和常量是程序操纵的对象。在C中，使用一个变量前必须要先声明它。声明用于两个目的。一个目的是，通过声明，告诉编译器在内存中留出一定的空间存储变量的值，另一个目的是，通过声明，使得编译器向机器发指令，让机器正确地完成特定的操作。在表达式  $a+b$  中，运算符+被应用到两个变量上。机器做加法时，把运算符+用到类型为int的变量上的结果与用到类型为float的变量上的结果是不同的。当然，程序员不必关心这两种加法在机器上有什么不同，但是C编译器必须识别这种不同，并给出适当的机器指令。

表达式是常量、变量和函数调用的有意义的组合。像变量一样，大多数表达式也有值和类型。在很多情况下，所发生的依赖表达式的类型，表达式的类型又依赖组成表达式的常量、变量和函数调用的类型。在下面的各节中，我们要讨论与类型的概念有关的问题。

## 6.2 基本数据类型

C提供了几种基本数据类型，我们已经看到了一些。我们要讨论对各种基本类型的存储限制。

基本数据类型，长形式		
char	signed char	unsigned char
signed short int	signed int	signed long int
unsigned short int	unsigned int	unsigned long int
float	double	long double

表中所示的类型都是关键字，不能把它们用作变量的名字。像数组和指针这样的一些数据类型都派生于基本数据类型（请参见第9章“数组和指针”）。

通常，不使用关键字signed。例如，signed int等价于int，由于较短的名字容易输入，所以经常使用int。然而类型char在这点上特殊的（请参看下一节）。同样，关键字short int、long int和unsigned int或许经常分别被缩写为short、long和unsigned。关键字signed本身等价于int，但在本文中很少使用。按照所有的这些惯例，我们给出一个新的基本数据类型列表。

基本数据类型		
char	signed char	unsigned char
short	int	long
unsigned short	unsigned	unsigned long
float	double	long double

可以按照功能对基本类型分组。整数类型是能用于保存整数值的那些类型；浮点类型是能用于保存实数值的那些类型。它们都是算术类型。

按功能对基本类型分组			
整数类型	char	signed char	unsigned char
	short	int	long
	unsigned short	unsigned	unsigned long
浮点类型	float	double	long double
算术类型	整数类型 + 浮点类型		

这些分组的名字是有用的。例如，在第9章“数组和指针”中，讨论数组时，我们要解释只允许用整数表达式做下标，这意味着允许用与整数类型有关的表达式做下标。

### 6.3 字符和数据类型char

在C中，很多整数类型变量能用于表示字符。特别是，char和int型变量能用于此目的。正如我们在第5章“字符处理”中看到的那样，当用一个变量读入字符时，必须测试EOF，这个变量应该是int型的，而不是char型的。我们把像'a'和'+'这样的常量看作是类型为int的字符，而不是char类型的。没有类型为char的常量。

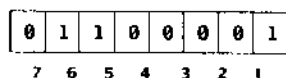
回忆一下，把字符处理为小整数，相反把小整数处理为字符。特别是，可以按字符格式或整数格式显示任何整数表达式。

```
char   c = 'a';           /* 'a' has ASCII encoding 97 */
int    i = 65;            /* 65 is ASCII encoding for 'A' */
printf("%c", c + 1);      /* b is printed */
printf("%d", c + 2);      /* 99 is printed */
printf("%c", i + 3);      /* D is printed */
```

在C中，把每个类型为char的变量都存储在内存的一个字节中。在几乎大多数机器上，一个字节由八位组成。让我们看一下一个类型为char的变量怎样按位存储在内存中。考虑声明

```
char   c = 'a';
```

我们认为c以如下方式存储在内存的一个字节中：



图中的每一个框代表一个位，从这些位的最小有效位开始编号。组成一个字节的位不是开就是关，分别用1和0表示它们。据此，我们把内存中的每个字节看作是由八个二进制数组成的串。二进制数组成的串也被称为位串(bit string)。我们可以把存储在内存中的变量c看作是位串01100001。更一般地，可以把每个机器字看作是一个形成若干个字节的二进制数组成的串。

把由二进制数组成的串解释为二进制数。在我们描述怎样做之前，先回想一下怎样把十进制数组成的串解释为十进制数。作为一个例子，考虑一下十进制数10 753。该数的值是：

$$1 \times 10^4 + 0 \times 10^3 + 7 \times 10^2 + 5 \times 10^1 + 3 \times 10^0$$

更一般地，把十进制按位记数写成如下的形式：

$$d_n d_{n-1} \cdots d_2 d_1 d_0$$

此处每个 $d_i$ 是一个十进制数。它的值是：



$$d_n \times 10^n + d_{n-1} \times 10^{n-1} + \cdots + d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0$$

把二进制按位记数写成如下的形式：

$$b_n b_{n-1} \cdots b_2 b_1 b_0$$

此处每个 $b_i$ 是一个二进制数，它不是1就是0。它的值是：

$$b_n \times 2^n + b_{n-1} \times 2^{n-1} + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

现在让我们再次考虑c的值。在一个字节中它被存储为01100001。这个二进制数的值是：

$$1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

它等于十进制的 $64+32+1$ ，即97。

ANSI C提供了三种数据类型，即char、signed char和unsigned char。通常依赖于编译器，类型char等价于signed char或unsigned char。这三种类型的变量或常量都存储在一个字节中，可有256个不同的值。对于signed char，值域为-128到127。对于unsigned char，值域为0到255。

## 6.4 数据类型int

数据类型int是C语言的主要工作类型。它和char、short和long这样的整型一样，是为了处理能在机器上表示的整数值而设计的。

数学中的自然数是0、1、2、3、…，这些与它们的负数一起组成了整数。在机器上，一个给定的整型仅能表示这些整数的一部分。

通常把一个int型的值存储在一个机器字中。一些计算机的机器字由两个字节组成（16位）；还有一些计算机的机器字由四个字节组成（32位）。还有其他的可能，但大多数机器的机器字是由两个或四个字节组成。例如，个人计算机使用两个字节的字；Apollo、Hewlett-Packard、Next、Silicon Graphics和Sun等制造的高档个人计算机和工作站使用四个字节的字。很多大型机也使用四个字节的机器字。由于各种机器的机器字的大小不一，所以int型的不同的值的个数是与机器相关的。假设我们正在使用的计算机的机器字为四个字节。这意味着int型的不同的值的个数为 $2^{32}$ 个，这是因为一个机器字有32位。其中的一半用于表示负整数，另一半用于表示正整数：

$$-2^{31}, -2^{31}+1, \cdots, -3, -2, -1, 0, 1, 2, 3, \cdots, 2^{31}-1$$

另一方面，如果我们使用有两个字节的机器字，那么int型的不同的值的个数为 $2^{16}$ 个。同样，其中的一半用于表示负整数，另一半用于表示正整数：

$$-2^{15}, -2^{15}+1, \cdots, -3, -2, -1, 0, 1, 2, 3, \cdots, 2^{15}-1$$

令 $N_{\min\_int}$ 表示int型值的最小数，令 $N_{\max\_int}$ 表示int型值的最大数。如果i是一个类型为int的变量，那么i的值的范围是：

$$N_{\min\_int} \leq i \leq N_{\max\_int}$$

范围的上下限与机器有关。通常的情形是：

在四个字节字的机器上：

$$\begin{aligned} N_{\min\_int} &= -2^{31} &= -2147483648 &\approx -20\text{亿} \\ N_{\max\_int} &= +2^{31} - 1 &= +2147483647 &\approx +20\text{亿} \end{aligned}$$

在两个字节字的机器上:

$$\begin{aligned} N_{\min\_int} &= -2^{15} = -32768 \approx -32\,000 \\ N_{\max\_int} &= +2^{15} - 1 = +32767 \approx +32\,000 \end{aligned}$$

在任何机器上, 下面的代码语法上都是正确的:

```
#define BIG 2000000000 /* 2 billion */

int main(void)
{
    int a, b = BIG, c = BIG;

    a = b + c; /* out of range? */
    .....
```

然而, 在运行时, 可能赋给变量a一个错误值。表达式b+c的逻辑值是40亿, 这可能大于 $N_{\max\_int}$ 。如果是这样, 该加法运算要引起所说的整数溢出(integer overflow)。通常当出现整数溢出时, 程序会继续执行, 但逻辑结果是错误的。由于这个原因, 程序员必须时时要保证整数表达式的值要在合理的范围内。

除了十进制整数常量外, 还有像0x1a这样的十六进制整型常量和像0377这样的八进制整型常量。很多C程序员可能不需要使用十六进制或八进制数, 但程序员必须知道用0开始的整数不是十进制整数。例如, 11和011是不同的。

## 6.5 整数类型short、long和unsigned

在C中, 把数据类型int认为是处理整数的“自然的”或“有用的”类型。把像char、short和long这样的整型认为是更专用的。例如, 当关心存储时, 用数据类型short, 编译器为short型的值提供的空间要比为int型的值提供的空间要少, 虽然不是必须这样做; 以类似的方式, 当需要较大的整数值时, 用类型long, 编译器为long型的值提供的空间要比为int的值提供的空间要多, 虽然不是必须这样做。通常, 用两个字节存储类型为short的值, 用四个字节存储类型为long的值。这样, 在机器字为四个字节的机器上, int的尺寸和long的尺寸是一样的, 在机器字为两个字节的机器上, int的尺寸和short的尺寸是一样的。如果small是short型的变量, 那么small值的范围是

$$N_{\min\_short} \leq \text{small} \leq N_{\max\_short}$$

通常,

$$\begin{aligned} N_{\min\_short} &= -2^{15} = -32768 \approx -32\,000 \\ N_{\max\_short} &= +2^{15} - 1 = +32767 \approx +32\,000 \end{aligned}$$

如果big是long型的变量, 那么big值的范围是

$$N_{\min\_long} \leq \text{big} \leq N_{\max\_long}$$

通常,

$$\begin{aligned} N_{\min\_long} &= -2^{31} = -2147483648 \approx -20\text{亿} \\ N_{\max\_long} &= +2^{31} - 1 = +2147483647 \approx +20\text{亿} \end{aligned}$$

存储一个类型为unsigned的变量的字节数与存储一个类型为int的变量的字节数相同。然而, 像名字暗示的那样, 存储的整数值没有符号。通常, 把类型为int和类型为unsigned的变量存储在一个机器字中。如果u是一个为unsigned的变量, 那么u的值的范围是

$$0 \leq u \leq 2^{\text{wordsize}} - 1$$

通常的情形是：

在机器字是四个字节的机器上：

$$N_{\text{max\_unsigned}} = +2^{32} - 1 = +4294967295 \approx +40 \text{ 亿}$$

在机器字是两个字节的机器上：

$$N_{\text{max\_unsigned}} = +2^{16} - 1 = +65535 \approx +65 \text{ 000}$$

完成对无符号变量的算术运算要用  $2^{\text{wordsize}}$  取模（请参见练习17）。

可以为一个整数常量加后缀来指明它的类型，无后缀整数常量的类型是 `int`、`long` 或 `unsigned`，系统在这些类型中选择能表达一个值的第一个类型。例如，在机器字是两个字节的机器上，常量32000的类型是 `int`，而常量33000的类型是 `long`。

long和unsigned的组合		
后 缀	类 型	例 子
u 或 U	unsigned	37U
l 或 L	long	37L
ul 或 UL	unsigned long	37UL

## 6.6 浮点类型

ANSI C提供了三种浮点类型：`float`、`double`和`long double`。浮点类型的变量能存储像0.001、2.0和3.14159这样的实数。在浮点常量的后面可以加后缀，用以表明其类型，无后缀的浮点常量是`double`类型的。与其他语言不同，C中的工作浮点类型是`double`，而不是`float`。

float和unsigned的组合		
后 缀	类 型	例 子
f or F	float	3.7F
l or L	long double	3.7L

可以把整数表示成浮点常量，但必须要用小数点。例如，常量1.0和2.0都是`double`类型的，而常量3是`int`型的。

除了用普通的十进制表示法表示浮点常量外，还可以用指数表示法。例如1.234567e5，这与科学计数法  $1.234567 \times 10^5$  是相对应的。回忆一下，

$$\begin{aligned} 1.234567 \times 10^5 &= 1.234567 \times 10 \times 10 \times 10 \times 10 \times 10 \\ &= 1.234567 \times 100000 \\ &= 123456.7 \text{ (小数点移5位)} \end{aligned}$$

以类似的方式，把1.234567e-3小数点向左移动三位，就得到了它的等价值0.001234567。

现在我们要仔细地描述指数表示法。在给出一些精确的规则后，我们还要给出一些例子。像333.77777e-22这样的浮点常量不含有任何空格或特殊字符，该常量的各部分都有一个名字。

浮点常量333.7777e-22的各个部分		
整 数	小 数	指 数
333	77777	e-22

浮点常量可以含有整数部分、小数点、小数部分和指数部分。浮点常量必须含有小数点或指数部分。如果含有小数点，就必须含有整数部分或小数部分。如果不含有小数点，就必须含有整数部分和指数部分。下面是浮点常量的一些例子：

```
3.14159
314.159e-2F    /* of type float */
0e0             /* equivalent to 0.0 */
1.             /* equivalent to 1.0, but harder to read */
```

下述的例子不是浮点常量：

```
3.14,159       /* comma not allowed */
314159         /* decimal point or exponent part needed */
.e0            /* integer or fractional part needed */
-3.14159       /* this is floating constant expression */
```

通常，C编译器为类型double提供的存储空间要多于为类型float提供的存储空间，虽然不是必须如此。在大多数机器上，用四个字节存储一个float型值，用八个字节存储一个double型值。这样，float存储了6个十进制位，double存储了15个十进制位。ANSI C编译器可能为类型long double的变量提供的存储空间要多于为类型double的变量提供的存储空间，虽然不是必须如此。很多编译器把long double处理为double（请参见练习16）。

用称为精度(precision)和范围(range)的术语来描述可以给浮点类型赋的值。精度描述了浮点值中的有意义的十进制位的个数。范围描述了浮点类型变量能表示的正的最大浮点值和最小浮点值。在很多机器上float的精度大约为6位，范围约为 $10^{-38}$ 到 $10^{+38}$ ，这意味着在机器中用如下的形式表示正的float值：

$$0.d_1d_2d_3d_4d_5d_6 \times 10^n$$

其中的 $d_i$ 是十进制数；第一个 $d_1$ 是正的； $-38 \leq n \leq +38$ 。在机器中实际上是用二进制表示float的值，而不是十进制，但上述的提法是正确的。

在很多机器上double的精度大约为15位，范围约为 $10^{-308}$ 到 $10^{+308}$ 。这意味着在机器中用如下的形式表示正的double值：

$$0.d_1d_2 \dots d_{15} \times 10^n$$

其中的 $d_i$ 是十进制数；第一个 $d_1$ 是正的； $-308 \leq n \leq +308$ 。假设x是类型为double的变量。则语句

```
x = 123.45123451234512345; /*20 significant digits*/
```

把用如下形式存储的值赋给了x（大约）：

$$0.123451234512345 \times 10^{+1} \quad (15个有意义的数字)$$

你必须要知道的要点是：（1）不是所有的实数都是可表示的；（2）浮点算术运算与整数算术运算不一样，它不需要多么精确。对于小的计算，实际上不需要关心这点，对于像解一大组微分方程这样的大的计算，可能需要很好地理解舍入和定标等。这属于数值分析领域。

## 6.7 sizeof运算符

C提供了一元运算符sizeof，用以给出存储一个对象所需的字节数。它优先级和结合性与其他的一元运算符相同。表达式的格式sizeof(object)返回一个表示在内存中存储对象所需要的字节数的整数。对象可以是像int或float这样的类型，或可以是像a+b这样的表达式，它还可以是数组或结构类型。下面的程序要使用这个运算符。具体的机器会提供基本类型所需要的存储空间方面的精度信息。

```
/* Compute the size of some fundamental types. */
#include <stdio.h>

int main(void)
{
    printf("Size of some fundamental types computed.\n\n");
    printf("    char:%3d byte \n", sizeof(char));
    printf("    short:%3d bytes\n", sizeof(short));
    printf("    int:%3d bytes\n", sizeof(int));
    printf("    long:%3d bytes\n", sizeof(long));
    printf("    unsigned:%3d bytes\n", sizeof(unsigned));
    printf("    float:%3d bytes\n", sizeof(float));
    printf("    double:%3d bytes\n", sizeof(double));
    printf("long double:%3d bytes\n", sizeof(long double));
    return 0;
}
```

因为C语言对基本类型所需要的存储空间方面的处理是很灵活的，但随着机器的不同情形也不同。但是，它可以保证以下几方面：

```
sizeof(char) = 1
sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
sizeof(signed) = sizeof(unsigned) = sizeof(int)
sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)
```

而且能保证所有的有符号和无符号的各整数类型版本都有相同的尺寸。

注意，我们写sizeof(……)就好像它是一个函数，然而它不是，它是一个运算符。如果把sizeof应用到类型，就需要括号；否则括号是可选择的。例如，sizeof(a+b+7.7)和sizeof a+b+7.7是等价表达式。该运算符返回的值的类型通常是unsigned，这与系统相关。

## 6.8 数学函数

在C中没有内置的数学函数。像sqrt()、pow()、exp()、log()、sin()、cos()和tan()这样的数学库中的函数是可用的，在概念上数学库是标准库的一部分。在传统的C系统中，通常认为数学库与C系统是分开的。有一些系统要用-lm选项或其他选项编译使用数学库的程序（请参见6.13节“系统考虑”）。

除了幂函数pow()外，上述的函数使用类型为double的单参数，返回值的类型也是double型的。幂函数使用类型为double的双参数，返回值的类型也是double型的。下一个程序要说明sqrt()和pow()的用法。该程序要求用户为x输入一个值，然后将其与x的平方根和x的x次幂一起显示。

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

int main(void)
{
    double x;
```

```

printf("\n%s\n%s\n%s\n\n",
    "The square root of x and x raised",
    "to the x power will be computed.",
    "---");

while (1) { /* do it forever */
    printf("Input x: ");
    scanf("%lf", &x);
    if (x >= 0.0)
        printf("\n%15s%22.15e\n%15s%22.15e\n%15s%22.15e\n\n",
            "x = ", x,
            "sqrt(x) = ", sqrt(x),
            "pow(x, x) = ", pow(x, x));
    else
        printf("\nSorry, number must be nonnegative.\n\n");
}
return 0;
}

```

如果我们执行这个程序，并在提示后输入2，那么如下的信息就出现在屏幕上：

```

The square root of x and x raised
to the x power will be computed.
---

Input x: 2

        x = 2.000000000000000e+000
    sqrt(x) = 1.414213562373095e+000
    pow(x, x) = 4.000000000000000e+000

Input x:

```

### 对程序sqrt\_pow的解析

- #include <stdio.h>  
#include <math.h>

这两个头文件含有函数原型。特别是，math.h含有数学库中的函数的原型。我们也可以自己编写函数原型而不引入math.h，例如，

```
double sqrt(double), pow(double, double);
```

可以在文件中把该声明放在main()的顶部或main()的内部。

- while (1) { /\* do it forever \*/  
.....

因为任何非零值都被认为是真，表达式1产生了一个无限while循环。希望用户重复地输入值并在完成时中断程序。

- scanf("%lf", &x);

由于x是double型的，在控制串中使用了格式%lf。一个常见的错误是用%f代替了%lf。注意，我们输入2是要说明程序的作用，我们也可输入2.0、2e0或0.2e1；函数调用scanf("%lf", &x)将会把它们都转换成double型的。在C中，源代码2和2.0是不同的，前一个的类型是int，后一个的类型是double。scanf()读入的输入流不是源代码，因而不能使用用于源代码的规则。当scanf()读入一个double值时，2和2.0是同样的（请参见13.2.4节“转换说明”）。

- if (x >= 0.0)  
.....

由于平方根函数仅能用于正数，所以必须要测试，以保证x的值不是负的。像sqrt(-1.0)这样的调用会引起运行错误（请参见练习19）。

```
printf("\n%15s%22.15e\n%15s%22.15e\n%15s%22.15e\n",
      "x = ", x,
      "sqrt(x) = ", sqrt(x),
      "pow(x, x) = ", pow(x, x));
```

注意，我们用格式%22.15e显示double值。这样，在小数点的左边有一位，在小数点的右边有16位，共有16个有效位。在我们的机器上，仅有n位是可用的，其中n是15或16。由于从二进制到十进制的转换，n是变化的。你可以要求显示大量的十进制的位数，但你对所读到的内容不要都相信。 ■

## 6.9 转换和类型转换

像x+y这样的数学表达式具有值和类型。例如，如果x和y的类型为int，那么表达式x+y的类型也是int，但是如果x和y的类型为short，那么x+y的类型是int，而不是short，这是因为在任何表达式中，都要把short提升或转换成int。在本节中，我们要给出精确的转换规则。

### 6.9.1 整型提升

char、short、signed、unsigned或枚举类型能用在int或unsigned int出现的任何表达式中（请参见第7章“枚举类型和typedef”）。如果原始类型的值都能用int表示，那么值就被转换成int型，把这称为整型提升(integral promotion)。下面是一个例子：

```
char c = 'A';
printf("%c\n", c);
```

char型的变量c本身作为printf()一个参数。然而，由于整型提升，所以表达式c的类型是int，而不是char。

### 6.9.2 常用的算术转换

当对二元运算符的运算数求值时，会发生算术转换。假设i是int型的，f是float型的。在表达式i+f中，操作数i被提升为float型的，整个表达式i+f是float型的。把这样的规则称为一般算术转换(usual arithmetic conversion)。

#### 一般算术转换

1) 如果一个运算数的类型为long double，则就把另一个运算数的类型转换为long double。

2) 否则，如果一个运算数的类型为double，则就把另一个运算数的类型转换为double。

3) 否则，如果一个运算数的类型为float，则就把另一个运算数的类型转换为float。

4) 否则，按下述规则，对两个运算数进行整型提升：

A. 如果一个运算数的类型为unsigned long，则就把另一个运算数的类型转换为unsigned long。

B. 否则，如果一个运算数的类型为long，另一个运算数的类型为unsigned，那么下

面情形之一将发生:

如果long能表示unsigned的所有值,那么把类型为unsigned的运算数转换成long型。

如果long不能表示unsigned的所有值,那么把两个运算数都转换成unsigned long型。

C. 否则,如果一个运算数的类型为long,那么把另一个运算数转换成long型。

D. 否则,如果一个运算数的类型为unsigned,那么把另一个运算数转换成unsigned型。

E. 否则,两个运算数都为int型。

这个过程有过各种名字:自动转换、隐式转换、强制、提升或延展。

下述的声明和混合表达式及其相应的类型说明了自动转换的思想:

声 明			
char c;                    short s;                    int i; unsigned u;                unsigned long ul;        float f; double d;                  long double ld;			
表 达 式	类 型	表 达 式	类 型
c - s / i	int	u * 7 - i	unsigned
u * 2.0 - i	double	f * 7 - i	float
c + 3	int	7 * s * ul	unsigned long
c + 5.0	double	ld + c	long double
d + s	double	u - ul	unsigned long
2 * i / 1	long	u - 1	system-dependent

除了在混合表达式中存在自动转换外,赋值也会发生自动转换,例如d=i把类型为int的i值转换成double型,并赋值给d,表达式的类型也是double型的。像d=i这样的提升或延展通常很守规矩,但像i=d这样的收缩或降格可能要丢失信息。此处,要丢失d的小数部分。在每种情况下确切地要发生什么情况是与系统相关的。

### 6.9.3 类型转换

在赋值和混合表达式中会出现隐式转换。除了隐式转换外,还有显式转换被称为类型转换,如果i是int型的,那么(double)i对i的值进行类型转换,该表达式的类型也是double型的,变量i本身保持不变。可对表达式应用类型转换。下面是一些例子:

```
(long) ('A' + 1.0)
x = (float) ((int) y + 1)
(double) (x = 77)
```

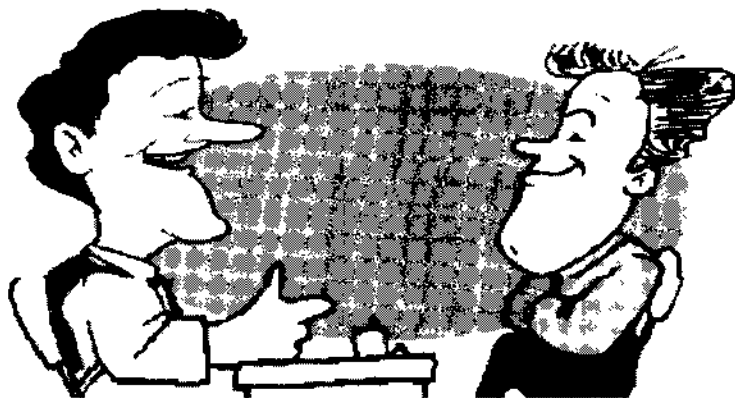
但下面的不是:

```
(double) x = 77    /* equivalent to    ((double) x) = 77, error */
```

类型转换运算符“(类型)”是一元运算符,与其他的一元运算符有同样的优先级和自右向左的结合性。因此,表达式(float)i+3等价于((float)i)+3。这是因为类型转换运算



符“(类型)”比+的优先级高。



“你认为'A'与97或97.0一样吗？不一样，三者都不一样，但是你可以用类型转换把它们从一种转换成另一种”

## 6.10 问题求解：计算利息

人们都熟悉存钱获得利息。在商业社会，很多合同和协定都涉及到金钱借入和借出，但要付利息。为了说明本章提出的一些思想，我们要说明如何计算年复合年息。在练习12中，我们说明如何计算复合季息和复合日息。

假设我们存 $P$ 美元，复合年息是7%。在第一年末，我们有原始本金 $P$ 美元和利息 $7\% \times P$ 。由于 $P$ 的7%是 $0.07 \times P$ ，第一年末我们的存款为 $P + 0.07 \times P$ ，也就是 $1.07 \times P$ 。在每年末，为了计算出我们的存款，我们要用本年年初的钱数乘以1.07，这样，我们的钱数为：

在第一年末 $1.07 \times P$

在第二年末 $1.07 \times (1.07 \times P)$  即 $1.07^2 \times P$

在第三年末 $1.07 \times (1.07^2 \times P)$  即 $1.07^3 \times P$

.....

在第 $n$ 年末 $1.07 \times (1.07^{n-1} \times P)$  即 $1.07^n \times P$

我们要用这样的思想编写一个计算复合年息的程序。用户提供本金、利率和年数的值，按照自顶向下设计，本程序要显示提示、读入值、计算利息和显示结果。下面就是这样的结构化设计，我们编写的程序由`main()`、`prn_instructions()`、`computs()`和`prn_results()`组成。让我们假设这些函数都放在文件`interest.c`中。在文件的顶部是`#include`行、函数原型和`main()`。

```
/* Compute interest compounded yearly. */
#include <stdio.h>

double compute(double p, double r, int n);
void prn_instructions(void);
void prn_results(double a, double p, double r, int n);

int main(void)
{
    double amount; /* principal + interest */
    double principal; /* beginning amount in dollars */
    double rate; /* example: 7% corresponds to 0.07 */
```

```

int      nyears;      /* number of years */

prn_instructions();
for ( ; ; ) {
    printf("Input three items: ");
    scanf("%lf%lf%d", &principal, &rate, &nyears);
    amount = compute(principal, rate, nyears);
    prn_results(amount, principal, rate, nyears);
}
return 0;
}

```

注意，我们把函数原型写在了main()的外边并靠近文件的顶部。虽然我们可以把它们写在main()的内部，但通常要把它们写在外边或写在要引入的.h文件中。在同一个文件中把其余的函数定义写在main()的后面。

```

void prn_instructions(void)
{
    printf("%s",
        "This program computes interest compounded yearly.\n"
        "Input principal, interest, and no. of years.\n"
        "For $1000 at 5.5% for 17 years here is example:\n\n"
        "Example input: 1000 5.5 17\n\n");
}

```

在我们说明本程序的其余代码之前，先看一下程序做了什么。假设我们执行它，并在出现提示后输入了1000.00、7和20。下面是在屏幕上出现的信息：

```

This program computes interest compounded yearly.
Input principal, interest, and no. of years.
For $1000 at 5.5% for 17 years here is example:

Example input: 1000 5.5 17

Input three items: 1000.00 7 20

Interest rate: 7%
Time period: 20 years

Beginning principal: 1000.00
Interest accrued: 2869.68
Total amount: 3869.68

```

在main()中，函数调用compute()的返回值被赋给了变量amount，这个值表示年限末的本金和利息。在程序的核心之处计算出了此值。

```

double compute(double principal, double rate, int nyears)
{
    int      i;
    double   amount = principal;

    rate *= 0.01;      /* example: convert 7% to 0.07 */
    for (i = 0; i < nyears; ++i)
        amount *= 1.0 + rate;
    return amount;
}

```

### 对函数compute()的解析

```

• double compute(double principal, double rate, int nyears)
{
    int      i;
    double   amount = principal;

```

第一个double告诉编译器函数返回的类型值。参数表含有三个标识符：两个double型的，一个int型的。在函数体中，把用principal对amount进行初始化，我们从这个钱数开始计算。

```
• rate *= 0.01;          /* example: convert 7% to 0.07 */
```

用户输入了三个值，中间的值是7，它就是计算中要用到的7%，这个值存储在main()中的类型为double的变量rate中，然后把rate作为参数传递给compute()。这条语句把rate的值转换成程序中所需要的0.07。注意，这条语句没有改变main()中rate。compute()中的rate仅是main()中rate的拷贝。

```
• for (i = 0; i < nyears; ++i)
    amount *= 1.0 + rate;
```

如果rate的值是0.07，那么1.0+rate的值是1.07，在每次循环中，都用1.07乘amount。由于amount被初始化为principal，在第一次循环中，amount的值为1.07乘principal的值；在第二次循环中，它的值为1.07<sup>2</sup>乘principal的值，以此类推。在最后的循环中，它的值为1.07<sup>nyears</sup>乘principal的值。

```
• return amount;
```

把amount的值返回到调用环境。 ■

最终返回到main()，我们要在屏幕上显示计算结果。我们把相关的变量作为参数传递给函数prn\_results()，并调用它。

```
void prn_results(double a, double p, double r, int n)
{
    double interest = a - p;    /* amount - principal */

    printf("\n%s%g%c\n%s%d%s\n\n",
        "Interest rate: ", r, '%',
        "Time period: ", n, " years");
    printf("%s%9.2f\n%s%9.2f\n%s%9.2f\n\n",
        "Beginning principal:", p,
        "Interest accrued:", interest,
        "Total amount:", a);
}
```

注意，我们用格式%g抑制对多余的0的显示，用格式%9.2f把屏幕上的数对齐。

下面我们要说明如何用数学库中的pow()简化程序。这个函数需要两个类型为double的表达式作参数，它的返回值是double型的。像pow(x,y)这样的函数调用计算x的y次幂。现在我们要修改程序，因为要用pow()取代compute()，所以我们要丢弃构成compute()函数定义的代码，以及靠近文件顶部的它的函数原型。要得到pow()的函数原型，我们可以增加一行

```
#include<math.h>
```

我们也可以自己编写这个函数原型：

```
double pow(double, double);
```

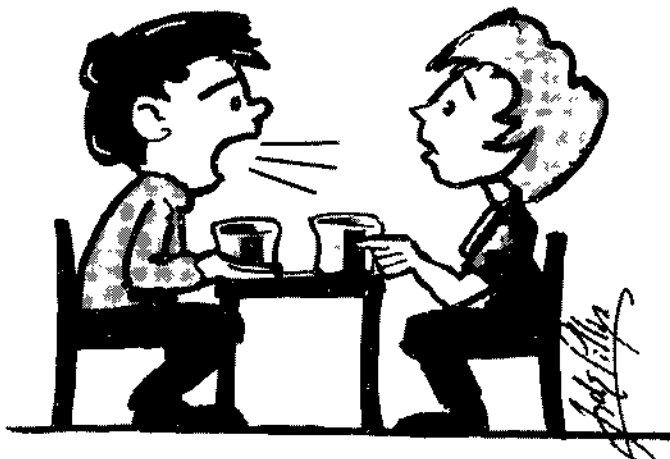
最后我们还要用语句

```
rate *= 0.01;
amount = pow(1.0 + rate, (double) nyears) * principal;
```

代替

```
amount = compute(principal, rate, nyears);
```

注意，已经把nyears类型转换为double。由于使用了函数原型，这个类型转换是不必要的；传递给pow()的任何int型的值都被自动地转化成了double型的。然而，如果我们使用的是不能用函数原型的传统C编译器，那么就需要用类型转换。



“银行说，不能向我们浮动(float)贷款，因为我们长期(long)负债，缺乏(short)资金。这样申请被拒签(unsigned)，全都落空(void)了。你能相信吗？银行用C语言拒绝了我们！”

## 6.11 风格

普通的编程风格是用像*i*, *j*, *k*, *m*和*n*这样的标识符作为类型为*int*的变量，用像*x*, *y*和*z*这样的标识符作为浮点变量。这种命名惯例在数学上是不严格的，历史上，早期的编程语言假设用从*i*到*n*的字母开始的变量名在缺省上是整型的。在一些简单的情形中还采用这种风格，例如用*i*作为*for*或*while*的计数器。然而在较复杂的情形中，你应该使用能描述其用途或用意的变量名。

一些程序员不喜欢随意地使用无限循环，但是，如果只打算把程序用于交互式的应用，这种作法是可以接收的。在我们的计算应计利息的程序中（请参见6.10节“问题求解：计算利息”），我们使用如下结构：

```
for ( ; ; ) {
    printf("Input three items: ");
    scanf("%lf%lf%d", &principal, &rate, &years);
    ....
}
```

此处的意图是用户要交互式地输入数据。用户重复地进行输入，直到工作完成为止，要通过输入一个终止信号（在我们的系统上是Ctrl+c）来中断程序。这种编程风格的优点是简单，读者能马上理解在中断发生之前正在重复做的事情；这种编程风格的缺点是，在一些系统上不能对程序的输入进行重定向（请参见练习10）。采用另一种更保守的风格是很容易的。下面是我们重写的程序：

```
printf("Input three items: ");
while (scanf("%lf%lf%d", &principal,
    &rate, &years) == 3) {
    ....
    printf("Input three items: ");
}
```

现在进行交互的用户随时可以通过在键盘上输入文件尾信号（在我们的系统上是Ctrl+d）来终止程序。另外，现在可以对程序使用重定向。

另一个格式上的问题关系到在浮点表达式中对浮点常量的使用。假设*x*是一个浮点变量，

由于自动转换,  $x \geq 0.0$  与  $x \geq 0$  是等价的。虽然如此, 我们还是认为使用第一个表达式较好。使用浮点常量  $0.0$  会提醒读者  $x$  属于浮点类型。类似地,  $1.0/3.0$  等价于  $1/3.0$ , 还是由于自动转换, 这两个表达式的值是相同的。然而, 由于在第一个表达式中分子和分母都是 `double` 型的, 读者能更容易地看出表达式是 `double` 型的。

## 6.12 常见的编程错误

在机器字为两个字节的机器上, 容易发生整数溢出的问题。程序员有责任把值保持在适当的范围内。类型 `long` 用于整数值大于 32 000 的情况。

在 `printf()` 和 `scanf()` 语句中, 像 `%d` 这样的适合于 `int` 的格式, 用于 `long` 可能会引起意外的结果。应当对 `long` 使用 `%ld`。在本文中, 转换符 `d` 前的修饰符 `l` 代表 `long`。类似地, 当使用 `short` 时, 应该用个数 `%hd`。在本文中, 转换符前的修饰符 `h` 代表 `short`。

让我们看一个例子, 它说明什么能出错。假设我们使用的机器的机器字为两个字节。

```
int  a = 1, b = 1776, c = 32000;
printf("%d\n", a + b + c); /* error: -31759 is printed */
```

表达式 `a+b+c` 的类型是 `int`, 在机器字为两个字节的机器上, 逻辑值 33 777 过大, 作为 `int` 型的值不能被存储。一个解决的办法是把上述语句改写为:

```
int  a = 1, b = 1776, c = 32000;
printf("%ld\n", (long) a + b + c); /* 33777 is printed */
```

由于运算符的优先级, `(long)a + b + c` 与 `((long)a) + b) + c` 是等价的。首先 `a` 被类型转换为 `long` 型的, 这把其他的被加数提升为 `long` 型的, 表达式 `(long)a + b + c` 也是 `long` 型的。最后, 用格式 `%ld` 显示表达式的值。注意, 用格式 `%d` 会产生错误的显示值。类似地, 当用 `scanf()` 向 `long` 型变量读入值时, 应该用格式 `%ld`。

程序员在 `printf()` 中可以用 `%f` 输出 `float` 或 `double`。然而, 在 `scanf()` 中必须用格式 `%f` 输入 `float`, 用格式 `%lf` 输入 `double`。程序初学者对这样的两分法容易混淆。如果使用错误的格式, 会产生意外的结果。通常 C 系统不提供任何错误指示 (请参见练习 2)。

在浮点表达式中使用整数常量可能会引起意外的结果。虽然表达式  $1/2$  和  $1.0/2.0$  看起来相似, 但第一个表达式是 `int` 型的, 其值是 0, 而第二个是 `double` 型的, 其值是 0.5。要减少在应该使用第二个表达式的地方使用第一个表达式的机会, 就要养成在浮点表达式中使用浮点常量的习惯。例如, 如果变量 `x` 是 `float` 型的, 并且你想把 0 赋给它, 那么最好用 `x = 0.0`, 而不用 `x = 0`。

另一个常见的编程错误是把错误类型的参数传递给函数。例如, 考虑一下函数调用 `sqrt(4)`。如果程序员已经为函数 `sqrt()` 提供了原型, 那么编译器会自动地把 `int` 型值 4 转换成 `double` 型的, 函数调用不会有什么问题。如果程序员没有为函数 `sqrt()` 提供了原型, 程序也能运行, 只是逻辑结果错误。传统 C 系统没有函数原型机制, 在这些系统中, 程序员应该写 `sqrt(4.0)` 或 `sqrt((double)4)`。

良好的编程风格要为所有的函数提供原型, 这有助于编译器防止用不恰当的参数调用函数的错误的发生。

在机器字为两个字节的机器上, 有时需要类型转换来保证常量表达式的行为适当。如下

是一个例子：

```
long product = (long) 2 * 3 * 5 * 7 * 11 * 13 * 17;
```

若没有该类型转换，这个初始化会产生意外的行为。

### 6.13 系统考虑

在本章中，我们已经解释了在机器字为两个字节和四个字节的机器上通常会发生什么情况。然而，一个机器的机器字是两个字节还是四个字节能有时也是难以断定的。Intel生产的Pentium芯片经常用作个人计算机的CPU，经常把这样的计算机称为“奔腾机”，在任何情况下该芯片的机器字都是四个字节的。然而，如果在机器上的操作系统是MS-DOS，那么该芯片的机器字只起到两个字节的机器字的作用。如果发生了这种情况，那么该操作系统没有发挥出CPU的全部作用。如果同样的机器使用的是UNIX操作系统，那么芯片的机器字就起到四个字节的机器字的作用。

计算领域已经达成了一个共识，即在机器上应该怎样表示浮点数。在这一点上，ANSI C委员会建议要遵循IEEE的二元浮点算术标准(IEEE Standard for Binary Floating-Point Arithmetic)(ANSI/IEEE Std 754-1985)。在大多数使用该标准的机器上，double的精度大约为15个有效位，大约范围是 $10^{-308}$ 到 $10^{+308}$ 。该标准的另一个作用是零做除数或数出界都不会导致运行错，而是产生一个被称为非数的值(请参见练习24)。

如果你在Cray巨型计算机上检查基本类型的大小，你会发现：char的大小是1，其他的都是8，但long double除外，它的大小是16。这意味着Cray能操纵非常大的整数值。

一些C系统仍然把数学库与标准库分离对待。在这些系统上，加载器可能找不到对应于sqrt()这样的数学函数的目标代码，除非你告诉加载器到哪儿去找。在很多旧式的UNIX系统上，需要选项-lm。例如，

```
cc pgm.v -lm
```

注意，选项的位置是不常见的。这是因为选项是提供给加载器的，而不是给编译器的。回想一下，cc命令先调用预处理器，然后调用编译器，最后调用加载器。

在所有的C系统中，类型short、int和long分别等价于signed short、signed int和signed long。然而，对于类型char，情形是与系统相关的。char可与signed或unsigned char等价。

C语言增加了新类型long double。一些C系统把它处理成double；还有一些系统提供了扩展精度。在Turbo C中，用十个字节存储long double。在Sun工作站上，用十六个字节存储long double，其精度约为33个有效位。

在标准库中提供了绝对值函数abs()。然而，它需要int型参数，并返回int型的值。这意味着它与对应于通常的获取实数绝对值的数学运算的函数不同。获取实数绝对值的函数是fabs()，它是浮点绝对值函数。fabs()在标准库中，其函数原型在math.h中。

C是一种通用语言，但它也适合于编写操作系统。系统程序员经常要处理以字节或字存储的值的显式的表示。由于十六进制和八进制整数可用于这样的目的，所以C语言也把它们引入，作为其中的一部分。然而，在本文中我们没有明确地使用它们。

### 6.14 转向C++

C++允许用户使用自定义类型，就好像它们是原有的类型一样。complex.h提供的类型

complex就是一个很好的实例:

```
//Roots of quadratic equations.
#include <iostream.h>
#include <complex.h>
int main(void)
{
    complex z1, z2;
    double a, b, c, discriminant;

    cout << "ENTER A B C from Ax*x + Bx + C :";
    cin >> a >> b >> c;
    discriminant = b * b - 4 * a * c;
    if (discriminant > 0.0) {
        z1 = (b + sqrt(discriminant)) / (2 * a);
        z2 = (b - sqrt(discriminant)) / (2 * a);
    }
    else if (discriminant == 0.0) {
        z1 = z2 = b / (2 * a);
    }
    else {
        z1 = (b + sqrt(-discriminant)) / (2 * a);
        z2 = (b - sqrt(-discriminant)) / (2 * a);
    }
    cout << "ROOTS in complex terms are :" << z1 << " "
        << z2 << endl;
}
```

在这个程序中,无缝地混用了complex和double变量。通过这样的混用,人们可把C程序很容易地扩展到很多新领域,而不用再学习或使用各种附加类型。

## 小结

- 编译器需要知道常量、变量和表达式的值和类型;编译器用这些信息在内存中正确地留出保存这些值的空间,并向机器发布正确的指令,以执行特定的操作。
- 基本数据类型有char、short、int和long,以及这些类型的无符号版本,此外还有三种浮点类型。类型char是一个字节的整数类型,主要用于表示字符。
- 类型int是“自然的”或“工作的”整型。像short、long和unsigned这样的其他整数类型用于较特殊的情形。
- 类型float、double和long double用于表示实数。通常用四个字节存储float,用八个字节存储double。与整数运算不同,浮点运算不总是精确的。类型double是“工作”类型,而float不是。
- 在ANSI C中类型long double是可用的,但在传统C中是不可用的。在一些C系统中, long double被实现为double;在另一些系统中,类型long double提供的精度比double高。
- 一元运算符sizeof()能用于计算出存储一个类型或一个表达式的值所需要的字节数。在机器字为两个字节的机器上, sizeof(int)的值是2,在机器字为四个字节的机器上, sizeof(int)的值是4。
- 在ANSI C中,在概念上数学库是标准库的一部分,它含有通常的像sin()、cos()和tan()这样的数学函数。系统提供了头文件math.h,该头文件含有数学函数的函数原型。
- 在数学库中的大多数函数都有一个类型为double的单参数,都返回类型为double的值。
- 函数pow()是一个例外,它有两个类型为double的参数,返回类型为double的值。
- 在混合表达式和等号两边中会发生自动转换。类型转换能用于显式的强制转换。

- 用0x和0开始的整数常量分别表明了它们是十六进制和八进制的。
- 可以用后缀显式地描述常量的类型。例如，3U的类型是unsigned，7.0F的类型是float。

## 练习

1. 不是所有的实数都能在机器上表示；这样的实数有很多。在一台机器上可使用的数有“粒度”的限制。例如，代码：

```
double x = 123.45123451234512345;
double y = 123.45123451234512300;
/* last two digits different */

printf("%.17f %.17f\n", x, y);
```

显示的两个数是相同的。在对y的初始化中，其尾部必须有多少个0才能使显示的值不同？请解释你的答案。

2. 如果把数3.777舍位，保留两位小数，那么它就变成了3.77，但如果把它舍入，保留两位小数，那么它就变成了3.78。编写一个测试程序，当显示带有小数部分的float或double值时，看printf()是舍位还是舍入。

3. 如果你使用一个库函数，但不声明它，那么编译器就假设函数缺省返回的值是int型的（在第4章“函数和结构化编程”中讨论了这个想法）。考虑如下代码：

```
double cos(double), x, y; /* sin() not declared */

while (1) {
    printf("Input a number: ");
    scanf("%lf", &x);
    y = sin(x) * sin(x) + cos(x) * cos(x);
    printf("\n%s%.15g\n%s%.15e\n\n",
        "x = ", x,
        "sin(x) * sin(x) + cos(x) * cos(x) = ", y);
}
```

这段代码说明了一个数学上的事实：

对于所有的x，有 $\sin^2(x) + \cos^2(x) = 1$ 。

注意，代码中没有sin()的函数原型。用正确的声明执行这段代码，以理解它的作用。然后不声明sin()的函数原型，实验一下会发生什么情况。你的编译器工作得顺利吗？它应该正常。

4. 在scanf()中用格式%f读入double型数据是一种常见的编程错误。在你的系统上试一试下面的代码。注意，你的编译器会顺利地工作，发生的惟一的事情是打印的逻辑值是错误的。

```
double x;

printf("Input a number: ");
scanf("%f", &x); /* error: wrong format */
printf("\nHere it is: %f\n", x);
```

5. 编写一个显示sin()、cos()和tan()的三角函数值的程序。表中的角度应该是从0到 $\pi$ ，共分20步。

6. 如果你的机器用两个字节存储int型的值，运行如下程序，并解释它的输出。如果你的机器用四个字节存储int型的值，在运行程序之前把符号常量BIG的值改为2000000000。

```
#define BIG 32000 /* 32 thousand is big? */

int main(void)
{
```



```

int      i = 8IG + BIG;
unsigned u = BIG + BIG;

printf("\n%s%d\n%s%d\n%su\n%su\n\n",
      "i with %d format is ", i,
      "u with %d format is ", u,
      "i with %u format is ", i,
      "u with %u format is ", u);
return 0;
}

```

7. 下表列出了在大多数机器上存储一些基本类型所需要的字节数。在你的机器上值是多少？执行6.7节“sizeof运算符”中的程序，完成此表。

基本类型	机器字为四个字节的机器所需的内存	机器字为两个字节的机器所需的内存	你的机器所需的内存
char	1字节	1字节	
short	2字节	2字节	
int	4字节	2字节	
unsigned	4字节	2字节	
long	4字节	4字节	
float	4字节	4字节	
double	8字节	8字节	

8. 在计算机时代以前，经常要制作平方根表、正弦表和余弦表等。编写一个程序，显示从1到10的整数的平方根和四次方根表。显示的表应该如下：

Integer	Square root	Fourth root
1	1.000000000e+00	1.000000000e+00
2	1.414213562e+00	1.189207115e+00
3	1.732050808e+00	1.316074013e+00
4	2.000000000e+00	1.414213562e+00
.....		

提示：一个数的四次方根是该数的平方根的平方根。

9. 本练习中的程序需要仔细地研究。

```

#include <stdio.h>

int main(void)      /* mystery? */
{
    printf("Why is 21 + 31 equal to %d?\n", 21 + 31);
    return 0;
}

```

下面是程序的输出结果：

Why is 21 + 31 equal to 5?

你能演绎出行为准则吗？

10. 对带有无限循环的程序使用重定向，程序可能会不正常。把一些实数放到名为data的文件中，试一下命令：

```
sq_roots < data
```

现在修改sq\_roots程序，用如下形式的while循环代替for循环，再试一下重定向。

```
while (scanf("%lf", &x) == 1) {
    .....
}
```

11. 利用数学库中的pow()函数重新编写程序interest（请参见6.10节“问题求解：计算利息”）。修改后的程序和以前的程序产生的结果相同吗？在调用pow()中，如果不把变量nyears类型转换为double会发生什么？

12. 设Constance B.DeMogul有百万美元要投资。她正在考虑要以9%的复合年息、8.75%的复合季息或8.7%的复合日息进行为期10或20年的投资。对于每种投资策略和期限，她各获利多少？编写一个程序，为她提供参考。提示：假设以8.75%的复合季息投资P美元，本金和利息数是：

$(1 + 0.0875/4)^4 \times P$       第一年末

$(1 + 0.0875/4)^{4 \times 2} \times P$       第二年末

$(1 + 0.0875/4)^{4 \times 3} \times P$       第三年末

.....

以8.7%的复合日息投资P美元，公式也是类似的，只是要用0.087代替0.0875，用365代替4。通过编写一个如下形式的函数可完成这个计算：

```
double find_accrued_interest(
    double principal,
    double rate,          /* interest rate */
    double c_rate,        /* compounding rate: example:
                           with daily compounding,
                           c_rate = 365.0 */
    double period         /* in years */
)
{
    .....
}
```

13. 试一下如下代码：

```
unsigned long a = -1;

printf("The biggest integer: %lu\n", a);
```

显示出了什么？解释原因。显示值应该与标准头文件limits.h中的一个或多个数相匹配，是这样吗？

14. 偶尔程序员需要对整数使用幂函数。由于这样的函数很容易写，通常就不再去数学库中找它。编写一个power()函数定义，如果m和n是整数，并且n是正的，那么调用power(m,n)返回m的n次幂。提示：使用如下的代码：

```
product = 1;
for (i = 1; i ≤ n; ++i)
    product *= m;
```

15. 类型为char的变量能用于存储小整数值。如果把的大值赋给类型为char的变量会发生什么情况？考虑下面的代码：

```
char c = 256;          /* too big! */

printf("c = %d\n", c);
```

一些编译器会警告你数太大了，而有的编译器不给出警告。你的机器如何？你能猜测出会显

示出什么吗?

16. 考虑如下代码:

```
char    a = 1, b = 2, c = 3;

printf("sizeof(c)           = %d\n", sizeof(c));
printf("sizeof('a')         = %d\n", sizeof('a'));
printf("sizeof(c = 'a')     = %d\n", sizeof(c = 'a'));
printf("sizeof(a + b + 7.7) = %d\n", sizeof(a + b + 7.7));
```

首先写出你认为应该显示的结果, 然后执行代码, 检验你的答案。

17. 在刻度为24小时的时钟上, 0点是午夜, 23点是晚上11点 (离午夜还有一个小时)。在这样的钟上, 23点再加1小时, 就是0点, 而不是24点 (没有24点)。以类似的方式,  $22 + 5$  是 3, 因为  $22 + 2 = 0$ , 多出的3就是结果。这是取模运算的一个例子, 更精确地说, 这是用24取模运算的例子。大多数机器都能对整数进行取模运算。用无符号整数可很容易地说明一点。运行下面的程序, 并解释显示结果。

```
#include <stdio.h>
#include <limits.h>      /* for UINT_MAX */

int main(void)
{
    int      i;
    unsigned u = UINT_MAX;

    printf("The largest unsigned int is %u\n\n", u);
    for (i = 0; i < 10; ++i)
        printf("%u + %d = %u\n", u, i, u + i);
    for (i = 0; i < 10; ++i)
        printf("%u * %d = %u\n", u, i, u * i);
    return 0;
}
```

18. 用  $\text{UINT\_LONG}$  和  $\text{MAX\_ULONG}$  表示在你的系统上能存储的 unsigned 型的最小值和最大值。这些值是什么? 提示: 读标准头文件 limits.h。

19. 函数调用 `sqrt(-1.0)` 会引起运行错误。大多数编译器都能对此类错误进行诊断。你的编译器能提供什么样的诊断?

20. 如果  $x$  的值过大, 那么函数调用 `pow(x, x)` 会引起运行错误或表现出一些异常行为。在你的系统上这样的最大整数  $x$  是多少? 编写一个交互式的程序, 用户输入  $x$ , 在屏幕上显示  $x$  和 `pow(x, x)`。

21. 在你的机器上, 可用的最大 double 型值是多少? 如果超出这个值会发生什么情况? 通过运行如下的程序进行实验:

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

int main(void)
{
    double  x = 1.0, y = -1.0, z = 0.0;

    printf("The largest double: %.3e\n", DBL_MAX);
    printf("  Add a little bit: %.3e\n", DBL_MAX + 1.0);
    printf("      Add a lot: %.3e\n",
           DBL_MAX + DBL_MAX);
    printf("  Division by zero: %.3e  %.3e\n",
           x / z, y / z);
    return 0;
}
```

修改程序，使它也能显示出你的机器上的最大float型值。该值比最大的double型值小吗？提示：读你的系统中的标准头文件float.h。

22. 这个问题是为那些熟悉十六进制和八进制的读者准备的。在程序中，用0开头的整数是八进制，用0x或0X开头的整数是十六进制。例如，

```
int    i = 077, j = 0x77, k = 0xcdb;
printf("Some numbers:%7d%7d%7d\n", i, j, k);
```

的输出为：

```
Some numbers:    63    119   3261
```

用在printf()的格式中的转换字符d用于显示十进制整数，转换字符x用于显示十六进制数，转换字符o用于显示八进制数。定义一个符号常量LIMIT，从1到LIMIT显示包含相应十进制、十六进制和八进制整数的值表。提示：先显示一个头，再使用如下语句：

```
for (i = 0; i <= LIMIT; ++i)
    printf("%12d%12x%12o\n", i, i, i);
```

23. 在数学中，e和 $\pi$ 都是为人熟知的；e是自然对数的底数， $\pi$ 是圆周率。e<sup>\*</sup>和 $\pi^*$ 哪个大？这是微积分学中的基本问题。然而，即使你从没有听说过e和 $\pi$ ，也不知道微积分，你也应该能回答这个问题。提示：e $\approx$ 2.71828182845904524， $\pi\approx$ 3.14159265358979324。

24. 在6.13节“系统考虑”中，我们讨论了IEEE浮点算术标准。你的机器遵循这样的标准吗？试着执行如下的代码：

```
double    x = 1e+308;           /* on the edge */
printf("x * x = %e\n", x * x); /* too big? */
```

如果显示出了Inf和NaN，你可以把它们认作是“无穷大”或“不是数”。如果这样的情况发生，就表明（但不是证明）你的机器遵循IEEE标准。

25. C++：把下述代码转换成C++函数：

```
void roots(double a,      //a*x*x +
           double b,      //b*x +
           double c,      //c
           complex r1,     //root one
           complex r2)     //root two
{
}
```

26. C++：在roots()的尾部加一个assert，检查计算的正确性。要记住浮点运算会舍入。

27. Java：C中的转换在Java中仅部分可用。Java是较安全的，因为它不允许自动的降格转换。在Java中，如果n是int型的，x是double型的，那么，

```
n = x; //okay in Java and C
x = n; //okay in C but illegal in Java
x = (double)n; //okay in both C and Java
```

你能列出允许的自动的C转换，并指明哪些在Java中也能使用吗？

28. Java：Java有一个Math类，它是一个标准函数库，很像C的math.h。该库含有一个范围在0.0~1.0间的伪随机数发生器。把下面的Java程序转换成C程序。

```
//RandomPrint.java: Print Random numbers in the
//    range (0.0 - 1.0). Java by Dissection page 107.
class RandomPrint {
    public static void main(String[] args) {
```

```
int n = 10;

System.out.println("We will print " + n +
    " random numbers");
printRandomNumbers(n);
}
static void printRandomNumbers(int k) {
    for (int i = 0; i < k; i++)
        System.out.println(Math.random());
}
}
```

## 第7章 枚举类型和typedef

在本章中我们要讨论枚举类型。枚举类型是用户自定义类型，它允许用户命名一个元素有限集，其中的元素称为枚举元(enumerator)。程序员定义这样的类型，并在需要的时候使用。下面，我们要讨论typedef，它允许你给一个类型新的名字，这种作法为枚举和结构类型提供更短的名字。

### 7.1 枚举类型

关键字enum被用于声明枚举类型。它允许你对一个有限集进行命名并声明集合中的元素标识符（称为枚举元）。例如，考虑声明

```
enum day {sun, mon, tue, wed, thu, fri, sat};
```

该声明创建了一个自定义类型enum day。在关键字enum的后面跟有标记名day。枚举元是标识符sun, mon, ... sat。它们是类型为int的常量。第一个枚举的缺省值是0，后继的枚举元也都有相继的整数值。该声明是一个类型说明符的示例。现在还没有声明类型enum day的变量。要声明它们，我们可以写

```
enum day d1, d2;
```

这声明了类型为enum day的变量d1和d2。它们的值只能是集合中的元素（枚举元）。因此，

```
d1 = fri;
```

是把值fri赋给d1，

```
if (d1 == d2)
    .... /* do something */
```

测试d1是否等于d2。注意，类型是enum day，关键字enum本身并不是一个类型。

可对枚举元进行初始化。同样，如果我们愿意，我们能够直接在枚举元声明后声明变量。下面是一个例子：

```
enum suit {clubs = 1, diamonds, hearts, spades} a, b, c;
```

由于已经把clubs初始化为1，diamonds、hearts和spades的整数值就分别为2、3和4。该声明由两部分组成：

类型说明符：

```
enum suit {clubs = 1, diamonds, hearts, spades}
```

该类型的变量：

```
a, b, c;
```

下面是初始化的另一个例子：

```
enum fruit {apple = 7, pear, orange = 3, lemon} frt;
```

由于已经把apple枚举初始化为7，pear的整数值就为8。类似地，由于已经把orange初始化为3，lemon的整数值就为4。可以有更多的值，但标识符要惟一。

```
enum veg {beet = 17, corn = 17} vege1, vege2;
```

标记名也可以不出现。例如，

```
enum {fir, pine} tree;
```

由于没出现标记名，所以不能再声明类型enum {fir, pine}的其他变量。

一般来讲，要把枚举元作为程序员指定的常量来对待，使用枚举元是为了使程序更清楚。如果需要，用类型转换能获得枚举元的基础值。在函数中的变量和枚举元的标识符必须是不一样的。

## 7.2 typedef的用法

C提供了typedef，它能把标识符和特定的类型联系在一起。例如，

```
typedef int color;
```

就使得color和int是同义的，color也是一种类型，像其他类型一样能用在声明中。例如，

```
color red, blue, green;
```

typedef使得程序员可以用更适宜于具体应用的类型名。同样，当程序员建立像枚举类型或结构类型这样的复杂或冗长的用户自定义类型时，使用它会有助于控制复杂性（请参见第12章“结构和抽象数据类型”）。

我们通过编写一个计算下一日的函数说明枚举类型的用法。按常规，我们先typedef我们的枚举类型。

```
/* Compute the next day. */
enum day {sun, mon, tue, wed, thu, fri, sat};
typedef enum day day;

day find_next_day(day d)
{
    day next_day;
    switch (d) {
        case sun:
            next_day = mon;
            break;
        case mon:
            next_day = tue;
            break;
        .....
        case sat:
            next_day = sun;
            break;
    }
    return next_day;
}
```

### 对函数find\_next\_day()解析

- enum day {sun, mon, tue, wed, thu, fri, sat};

这个声明是类型说明符的一个示例。它告诉编译器enum day是类型名，sun, mon, ..., sat是该类型的变量可取的值。没有对该类型声明任何变量，该声明仅提供了随后要在程序中使用的类型。

- typedef enum day day;

我们用typedef为类型enum day创建了一个新名。按常规，我们选择了类型enum day的标记名作为类型名。在ANSI C中，标记名的命名空间与其他名字的命名空间是分开的。这样，编译器就能理解在标记名day和类型day间的不同。如果把typedef移出程序，那么在整个代码中凡是用标识符day的地方，都必须要写enum day。

```
• day find_next_day(day d)
{
    day    next_day;
```

find\_next\_day的函数定义的头告诉编译器这个函数仅需要一个类型为day的单参数，并向调用环境返回一个类型为day的值。在函数体内有一个声明。它告诉编译器next\_day是类型为day的变量。

```
• switch (d) {
    case sun:
        next_day = mon;
        break;
```

回忆一下，仅常量整型表达式能用作做case的标号。由于枚举元是常量，所以能在这样的情况下使用枚举元。注意，赋给next\_day的是一个枚举元。

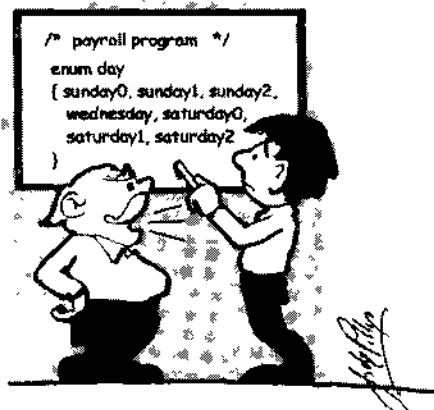
```
• return next_day;
```

类型为day的next\_day的值被返回到了调用环境。

下面是该函数的另一个版本，它用类型转换完成同样的结尾：

```
/* Compute the next day with a cast. */
enum day {sun, mon, tue, wed, thu, fri, sat};
typedef enum day day;
day find_next_day(day d)
{
    return ((day) (((int) d + 1) % 7));
}
```

可把枚举类型用于普通的表达式中，只要类型兼容即可。然而，如果把枚举元用作整型，并要经常地访问这种不清晰的表示，最好还是用整型变量。枚举类型的重要性在于它具有文档性，枚举元本身有助于记忆，此外，枚举元能强制编译器提供自定义类型检查。这样就不会不注意地把苹果和钻石相混淆。



“哈罗德，不要和我开玩笑，你不能仅在周末编程！”



### 7.3 例子：石头、剪刀、布游戏

我们通过编写一个传统的小孩玩的游戏的程序“石头、剪刀、布”说明枚举类型的用法。在这个游戏中，孩子们用手表示布、石头或剪刀中的一个。伸手表示布，出拳头表示石头，伸两根手指表示剪刀。孩子们面对面地数到三时，给出他们的选择。如果选择是一样的，就形成平局，否则就用如下的规则决定胜负：

石头、剪刀、布的规则

- 布覆盖石头
- 石头砸坏剪刀
- 剪刀剪碎布

我们要在一个程序目录中编写一个多文件程序，就像在4.11节“模拟：正反面游戏”中讨论的那样。程序由一个.h和几个.c文件组成，每个.c文件都在顶部引入头文件。在头文件中，我们要放入#include指令、四个对枚举类型的声明、类型定义和函数原型。

文件p\_r\_s.h的内容：

```
/* The game of paper, rock, scissors. */

#include <ctype.h>          /* for isspace() */
#include <stdio.h>          /* for printf(), etc */
#include <stdlib.h>         /* for rand() and srand() */
#include <time.h>           /* for time() */

enum p_r_s {paper, rock, scissors,
            game, help, instructions, quit};

enum outcome {win, lose, tie, error};

typedef enum p_r_s    p_r_s;
typedef enum outcome outcome;

outcome compare(p_r_s player_choice, p_r_s machine_choice);
void prn_final_status(int win_cnt, int lose_cnt);
void prn_game_status(int win_cnt,
                    int lose_cnt, int tie_cnt);

void prn_help(void);
void prn_instructions(void);
void report(outcome result, int *win_cnt_ptr,
            int *lose_cnt_ptr, int *tie_cnt_ptr);
p_r_s selection_by_machine(void);
p_r_s selection_by_player(void);
```

我们通常在#include行不加注释，但在此处我们要试着为程序初学者提供一个交叉参考。

文件main.c的内容：

```
#include "p_r_s.h"

int main(void)
{
    int win_cnt = 0, lose_cnt = 0, tie_cnt = 0;
    outcome result;
    p_r_s player_choice, machine_choice;

    srand(time(NULL)); /* seed the random number generator */
    prn_instructions();
```

```

while ((player_choice = selection_by_player()) != quit)
    switch (player_choice) {
        case paper:
        case rock:
        case scissors:
            machine_choice = selection_by_machine();
            result = compare(player_choice, machine_choice);
            report(result, &win_cnt, &lose_cnt, &tie_cnt);
            break;
        case game:
            prn_game_status(win_cnt, lose_cnt, tie_cnt);
            break;
        case instructions:
            prn_instructions();
            break;
        case help:
            prn_help();
            break;
        default:
            printf("PROGRAMMER ERROR: Cannot get to here!\n\n");
            exit(1);
    }
    prn_game_status(win_cnt, lose_cnt, tie_cnt);
    prn_final_status(win_cnt, lose_cnt);
    return 0;
}

```

main() 中的第一个可执行的语句是

```
srand(time(NULL));
```

它为随机数发生器rand()播种，它使得每次执行该程序时产生的整数序列都不同。更明确地讲，传递给srand()一个整数，以决定rand()从何处开始。函数调用time(NULL)返回一个自1970年1月1日（大约是UNIX的生日）以来经过的秒数。标准C库提供了srand()和time()，srand()的函数原型是在stdio.h中，time()的函数原型是在time.h中。系统提供了这两个头文件。注意，我们在p\_r\_s.h中引入了它们。

main() 中的下一个可执行的语句是调用prn\_instructions()，它向用户提供指示，嵌入在指令中的是一些对游戏编程方面的设计上考虑。我们在prn.c中和另一个显示函数一起编写这个函数。

**文件prn.c中的内容：**

```

#include "p_r_s.h"

void prn_final_status(int win_cnt, int lose_cnt)
{
    if (win_cnt > lose_cnt)
        printf("CONGRATULATIONS - You won!\n\n");
    else if (win_cnt == lose_cnt)
        printf("A DRAW - You tied!\n\n");
    else
        printf("SORRY - You lost!\n\n");
}

void prn_game_status(int win_cnt, int lose_cnt, int tie_cnt)
{
    printf("\n%s\n%s%4d\n%s%4d\n%s%4d\n%s%4d\n\n",
        "GAME STATUS:",
        " Win: ", win_cnt,
        " Lose: ", lose_cnt,
        " Tie: ", tie_cnt,
        " Total: ", win_cnt + lose_cnt + tie_cnt);
}

```

```

void prn_help(void)
{
    printf("\n%s\n",
        "The following characters can be used for input:\n"
        "  p   for paper\n"
        "  r   for rock\n"
        "  s   for scissors\n"
        "  g   print the game status\n"
        "  h   help, print this list\n"
        "  i   reprint the instructions\n"
        "  q   quit this game\n");
}

void prn_instructions(void)
{
    printf("\n%s\n",
        "PAPER, ROCK, SCISSORS:\n"
        "\n"
        "In this game\n"
        "\n"
        "  p is for \"paper\"\n"
        "  r is for \"rock\"\n"
        "  s is for \"scissors\"\n"
        "\n"
        "Both the player and the machine will choose one\n"
        "of p, r, or s. If the two choices are the same,\n"
        "then the game is a tie. Otherwise:\n"
        "\n"
        "  \"paper covers the rock\"      (a win for paper)\n"
        "  \"rock breaks the scissors\"    (a win for rock)\n"
        "  \"scissors cut the paper\"      (a win for scissors)\n"
        "\n"
        "There are other allowable inputs:\n"
        "\n"
        "  g for game status   (print number of wins)\n"
        "  h for help          (print short instructions)\n"
        "  i for instructions  (print these instructions)\n"
        "  q for quit          (quit the game)\n"
        "\n"
        "This game is played repeatedly until q is entered.\n"
        "\n"
        "Good luck!\n");
}

```

要玩游戏，机器和游戏者（用户）必须在布、石头和剪刀之间做出选择。我们在select.c中编写这些程序。

文件select.c中的内容：

```

#include "p_r_s.h"

p_r_s selection_by_machine(void)
{
    return ((p_r_s) (rand() % 3));
}

p_r_s selection_by_player(void)
{
    char    c;
    p_r_s   player_choice;

    printf("Input p, r, or s: ");
    scanf("%c", &c);
    switch (c) {
        case 'p':
            player_choice = paper;
            break;
        case 'r':
            player_choice = rock;
            break;
        case 's':

```

```

        player_choice = scissors;
        break;
    case 'g':
        player_choice = game;
        break;
    case 'i':
        player_choice = instructions;
        break;
    case 'q':
        player_choice = quit;
        break;
    default:
        player_choice = help;
        break;
    }
    return player_choice;
}

```

通过用表达式`rand()%3`产生分布在0到2之间的随机整数来计算出机器的选择。因为该函数的类型是`p_r_s`，所以如果需要，就把它的返回值转换成`p_r_s`型。我们提供了一个显式的类型转换使得代码更文档化。在`selection_by_player()`中，我们用

```
scanf(" %c", &c);
```

从输入流中读入数据，并把数据放在`c`中。在`%`前的空格是有意义的，它使得`scanf()`与可选的空白相匹配（请参见13.2.3节“空白字符”）。观察一下对从键盘上输入的所有非空白字符的处理，其中的大多数要经过`switch`语句的`default`。

`selection_by_player()`返回的值依赖游戏者输入的信息。例如，如果游戏者输入字符`g`，那么把枚举元`game`返回到`main()`中的调用环境，这使得`selection_by_player()`被调用。在`selection_by_player()`中，如果输入了除空白符、`p`、`r`、`s`、`g`、`i`和`q`之外的任何字符，就把枚举元`help`返回到`main()`中的调用环境，这使得`prn_help()`被调用。

一旦游戏者和机器都做出了选择，我们需要比较它们的选择，以决定游戏的结果。下面的函数完成此项工作：

文件`compare.c`中的内容：

```

#include "p_r_s.h"

outcome compare(p_r_s player_choice, p_r_s machine_choice)
{
    outcome result;

    if (player_choice == machine_choice)
        return tie;
    switch (player_choice) {
    case paper:
        result = (machine_choice == rock) ? win : lose;
        break;
    case rock:
        result = (machine_choice == scissors) ? win : lose;
        break;
    case scissors:
        result = (machine_choice == paper) ? win : lose;
        break;
    default:
        printf("PROGRAMMER ERROR: Unexpected choice!\n\n");
        exit(1);
    }
    return result;
}

```

`main()`中的`compare()`的返回值被传递给了函数`report()`，`report()`向用户报告

此轮比赛的结果，并记录相应的获胜、失败和平局的次数。

文件report.c中的内容：

```
#include "p_r_s.h"

void report(outcome result, int *win_cnt_ptr,
            int *lose_cnt_ptr, int *tie_cnt_ptr)
{
    switch (result) {
        case win:
            ++*win_cnt_ptr;
            printf("%27sYou win.\n", "");
            break;
        case lose:
            ++*lose_cnt_ptr;
            printf("%27sYou lose.\n", "");
            break;
        case tie:
            ++*tie_cnt_ptr;
            printf("%27sA tie.\n", "");
            break;
        default:
            printf("PROGRAMMER ERROR: Unexpected result!\n\n");
            exit(1);
    }
}
```

我们现在准备编译程序。用下述命令之一完成此项工作：

```
cc -o p_r_s main.c compare.c prn.c report.c select.c
cc -o p_r_s *.c
```

只有本程序的.c文件都在一个目录中，才用第二个命令。在我们学习14.8节“make的用法”之后，我们就能用适当的make程序的描述文件简化程序的开发。

## 7.4 风格

由于枚举类型是有助于记忆的，它的效用趋于文档化，因此，对枚举类型的使用被认为是一种良好的编程风格。

与用0和1来区别一些交替的选择相比，程序员使用枚举类型更有效。下面是一些典型的例子：

```
enum bool {false, true};
enum off_on {off, on};
enum no_yes {no, yes};
enum speed {slow, fast};
```

这种结构的使用使得代码更加易读。像

```
enum no_yes {no, yes};
```

这样的声明告诉编译器enum no\_yes是自定义类型。把这样的声明放在头文件中被认为是一种良好的编程风格。（不应该把在内存中分配空间的结构放入头文件。）

由于标记名有自己的命名空间，我们可以把标记名重用作为变量或枚举元。例如，

```
enum veg {beet, carrot, corn} veg; /* poor style */
```

虽然这样做是合法的，但这种编程风格是不好的。另一方面，用标记名作为类型定义却是一种良好的编程风格（在C++中，这一点是自动完成的）。例如：

```
typedef enum veg {beet, carrot, corn} veg; /* good style */
```

该声明告诉编译器veg是自定义类型。要声明这种类型的变量，我们可以写

```
veg    v1, v2;
```

这种作法能减少混乱，特别是在有很多枚举类型的情况下。

若要用typedef为枚举类型创建新的名字，可用几种方法。我们可以先声明再创建：

```
enum lo_hi {lo, hi};
typedef    enum lo_hi    lo_hi;
```

也可以把两步合为一步：

```
typedef    enum lo_hi {lo, hi}    lo_hi;
```

如果不想用标记名，我们也可以写：

```
typedef    enum {lo, hi}    lo_hi;
```

每种编程风格都是可以接受的，使用哪一种凭个人的喜好。

## 7.5 常见的编程错误

在同一个文件中相同的类型定义不能重复出现。假设下行既出现在你的.c文件中，也出现在你的头文件中：

```
typedef    enum {off, on}    off_on;
```

如果你在.c文件的顶部引入头文件，编译器会报警。在这点上，typedef不像#define。在同一个文件中#define可出现多次。

定义枚举类型时的枚举元的次序可能导致逻辑错误。考虑如下代码：

```
#include <stdio.h>

typedef    enum {yes, no}    yes_no;    /* poor */

yes_no    is_good(void);

int main(void)
{
    ....
    if (is_good())
        printf("Hooray!  A good condition exists.\n");
    else
        printf("Rats!  Another bummer.\n");
    ....
}
```

此处的想法是is\_good()函数要检查一些事情，并依据情况返回yes或no。让我们假设在上述的代码要返回yes，由于yes的值是int型的0(false)，所以显示的串是

```
Rats!  Another bummer.
```

这样，意外发生了。通过显式地检查返回值，我们能防止此类错误的发生。

```
if (is_good() == yes)
    ....    /* do something */
```

作为附加的警戒，最好把typedef写作

```
typedef    enum {no, yes}    no_yes;
```

现在no的值是0(false)，yes的值是1(true)，这种事态更合法。

## 7.6 系统考虑

系统程序员广泛地使用枚举类型。下面的例子来自于Sun机器上的一个系统头文件：

```
enum fp_precision_type {    /* extended precision */
    fp_extended    = 0,
    fp_single      = 1,
    fp_double      = 2,
    fp_precision_3 = 3
};
```

注意，对枚举类型的初始化是多余的，但这样会使得代码更易读。

在C中，可以把枚举类型和int自由混合使用，但这不是良好的编程风格。下面是一个例子：

```
#include <stdio.h>

typedef enum {no, yes} no_yes;

int main(void)
{
    int i;
    no_yes val = yes;

    i = val;    /* acceptable programming style */
    val = 777;  /* poor programming style */
    ....
```

在C++中，不允许把int型的值赋给枚举类型变量，虽然很多编译器仅给出警告。如果我们把对val的赋值变为

```
val = (no_yes) 777;    /* use a cast */
```

则编译器不会给出警告。虽然赋值给val的值既不与no对应，也不与yes对应，但这不被看作错误。

## 7.7 转向C++

C++中的枚举类型比C中的更强。每个类型都是不同的。用一些其他的值向枚举类型赋值是错误的（虽然一些编译器仅给出警告）。没有适当的类型转换，程序不能编译。

在C++中枚举标记名是自动键入的，因此在C++中很少需要typedef。让我们编写一个简单程序来说明这个想法。

```
#include <iostream.h>

enum color {black, red, blue, green, white, other};

int main(void)
{
    color val = blue;

    if (val == red)
        cout << "Value of red = " << red << "\n";
    else if (val == blue)
        cout << "Value of blue = " << blue << "\n";
    else
        cout << "The color is neither red nor blue.\n";
}
```

## 小结

- 程序员用关键字enum定义枚举类型。考虑定义声明

```
enum no_yes {no, yes};
```

它告诉编译器enum no\_yes是用户自定义类型。词enum本身不是类型。像

```
enum no_yes answer;
```

这样的声明定义了一个类型为enum no\_yes 的变量answer。它用集合{no, yes}中的成员作值，这些成员称为枚举元(enumerator)。枚举元是常量，且值是int型的。在缺省情况下，编译器把0赋给第一个枚举，把1赋给第二个枚举，以此类推。

- 枚举都是有助于记忆的各不相同的标识符。它们的使用为程序员提供了类型检查约束，也增强了程序的文档性。
- 若要在表达式中把枚举元或枚举类型变量与整数混合使用，可用类型转换解决类型冲突。
- 由于枚举元是int型的常量，可以把枚举元用作switch的case标号。
- 一个类型定义可用在另一个类型定义中（请参见练习8）。在随后的篇章中我们将会看到，这是一种重要的思想。
- 系统程序员经常使用枚举类型。在一些情况下，这会和一些应用程序发生冲突（请参见练习7和练习9）。
- 在C++中，作为类型的可选名，枚举类型的标记名是自动可用的。
- 在C++中，用文字false和true把bool类型内置到系统中。

## 练习

1. 编写一个名为previous\_month()的函数，它返回上一个月份。程序以如下代码开始：

```
enum month {jan = 1, feb, ..., dec};
typedef enum month month;
```

如果把dec作为参数传递给该函数，那么应该返回jan。编写另一个显示月份的函数，更明确地，如果把jan作为参数传递给该函数，那么应该返回January。编写调用函数的main()，产生一个各月相邻的十二个月的表。当心：如果使用printf()，那么显示枚举类型变量的结果是整数值。也就是说，

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

显示出的值是1，而不是jan。

2. 对于一个给定的年份，编写一个计算下一日的程序。程序用两个数做输入，例如输入17和5，代表5月17日，程序应该显示出18 May，也就是5月17日的下一日。在程序中使用枚举类型。特别要注意跨月份的问题。

3. 编写一个轮盘赌程序。轮盘赌机随机地在0到35之间选取一个数。比赛者可以选奇/偶赌或特殊数赌。一次赢得奇/偶赌，获利是2:1，但如果轮盘赌机选择的是0，则所有的奇/偶赌都算输。如果比赛者进行特殊数赌，并且轮盘赌机也选择了该特殊数，比赛者获利35:1。如果你用每次用1\$下赌，在你输10\$之前，你能玩几次？

4. 在一个营养平衡程序中，用枚举类型定义5个基本食物组：鱼（fish）、水果（fruit）、谷物（grain）、肉（meat）和蔬菜（vegetable）。例如：

```
enum fish {bass, salmon, shrimp, trout}
```



```
enum fruit {apple, peach, pear};
```

```
.....
```

用随机数发生器从每组中各选一项。编写一个函数meal(), 从每组中各选一项, 显示一个菜单, 显示20个随机产生的菜单。有多少种不同的可用菜单?

5. 在石头、剪刀、布的游戏, 用向游戏者显示

You win. 或 You lose.

来表示不是平局的结果。重新编写该程序, 显示如下信息:

```
You chose paper and I chose rock. You win.
```

6. 在Pascal语言中, 布尔类型是自动可用的, 而在C语言中, 程序员必须明确地提供该类型。下面是做到这一点的两种方法:

```
typedef enum {true, false} boolean;
typedef enum { false, true } boolean;
```

这两种typedef哪一个更好些? 请解释。

7. 一些标识符会引起意想不到的困难。考虑下面使用了枚举元sun的程序:

```
#include <stdio.h>

enum day {sun, mon, tue, wed, thu, fri, sat};

typedef enum day day;

int main(void)
{
    day val = sun;

    if (val == sun)
        printf("Today is Sunday.\n");
    else
        printf("Today is a working day.\n");
    return 0;
}
```

通过一种等价于使用#define预处理指令的机制, 在Sun机器上的一些C系统把sun定义为符号常量, 这与会与程序中作为标识符的sun相冲突。如果可能, 你在Sun机器上和一些其他的机器上试一下这个程序, 看会发生什么情况。如果程序不能编译, 你会观察到错误信息是完全无用的。要克服这个难题, 你可以用在文件的头部使用如下的预处理指令:

```
#undef sun
```

通过把该行放在文件的头部, 修改程序。如果你使用的是Sun机, 这行就能解决问题。如果你使用的不是Sun机, 该预处理指令对程序不会产生影响。

8. 一个类型定义能用于另一个类型定义中(在第11章“递归”中, 我们会看到这是一种重要的思想)。下面是一个说明这一点的程序:

```
#include <stdio.h>

typedef enum no_yes {no, yes, maybe} no_yes;
typedef no_yes chk_it_out;

int main(void)
{
    char c;
    chk_it_out ans;

    printf("Do you like vegetables? ");
    scanf("%c", &c);
}
```

```

if (c == 'n' || c == 'N')
    ans = no;
else if (c == 'y' || c == 'Y')
    ans = yes;
else
    ans = maybe;
switch (ans) {
case yes:
    printf("The answer was yes.\n");
case no:
    printf("The answer was no.\n");
case maybe:
    printf("You probably typed a nonsense reply.\n");
default:
    printf("Impossible to get here.\n");
}
return 0;
}

```

在你的系统上编译并执行这个程序。编译应该很顺利，但在运行程序时，你可能对所发生的感到惊讶。修改程序，使它的行为更适当。还要注意scanf()中的控制串%c。在百分号前的空格使得可选的空白被匹配。在修改程序后，重复地执行它，在回答是与否之前，试一试各种空白。重复地进行，直到你明白程序的作用为止。然后修改控制串，把空格去掉，程序的作用会发生变化吗？

9. 在前一个练习中，用枚举作为switch的case标号。由于在case表达式中的标号必须是常量整型表达式，你能对枚举元断定出什么？你能在main()体内使用

```
no = yes = 3;
```

这样的语句吗？请解释。

10. C++: 用C++重新编写石头、剪刀、布的游戏。如果你的系统有在类型上是安全的类型转换，就使用它。如果你的系统有内置的枚举类型bool，使用它。

11. C++: 用下述语句实现一个三态逻辑值程序：

```
enum ternary_logic {false, maybe, true};
ternary_logic tern_not(ternary_logic v);
```

若v是true，则函数应返回false；若v是false，则函数应返回true，若v是maybe，则函数应返回maybe。想法是要模拟普通的会话，此处“Maybe it will rain today”和“Maybe it will not rain today”的含义是等价的。要以一致的方式对tern\_and()、tern\_or()和tern\_not()编码。

12. 下边的Java程序模拟了硬币投掷，把该程序转换成C程序。你可以用枚举类型模拟Java的boolean类型。Java没有typedef或enum类型，Java用类构造类型，不需要typedef。一些Java程序员对Java被扩展以包含枚举类型会感到高兴。

```

//CoinToss.java - Compute the approximate probability
// of n heads in a row by simulating coin tosses. Java by
// Dissection pages 110-111.

class CoinToss {
    public static void main(String[] args) {
        //Input the number of tosses in a row to try for.
        int numTosses = 4; //Just use 4 for testing

        //Input the number of trials to run.
        int numTrials = 10000; //Use 10000 for testing

        //Perform the specified number of trials
    }
}

```

```
int numSuccesses = performTrials(numTosses,numTrials);

//Print the results
double probability = numSuccesses / (double)numTrials;
System.out.println("Probability found in "
    + numTrials + " is " + probability);
}

// return true if numTosses heads are tossed
// before a tail
static boolean isAllHeads(int numTosses) {
    double outcome;
    for (int numHeads = 0; numHeads < numTosses;
        numHeads++) {
        outcome = Math.random(); // toss the coin
        if ( outcome < 0.5)
            return false; // tossed a tail
    }
    return true; // tossed all heads
}

// perform numTrials simulated coin tosses
// and return the number of successes
static int performTrials(int numTosses,
    int numTrials) {
    System.out.println("Monte Carlo " + numTosses +
        " in a row heads");
    int numSuccesses = 0;
    for (int trials= 0 ; trials < numTrials; trials++)
        // perform one trial
        if ( isAllHeads(numTosses))
            numSuccesses++; // trial was a success
    return numSuccesses;
}
}
```

## 第8章 函数、指针和存储类型

当把表达式作为参数传递给函数时，要复制表达式的值，把复制的值传递给函数，而不是用表达式本身进行传递，这种机制被称为按值调用，在C中，这是严格的规定。假设 $v$ 是变量， $f()$ 是函数，如果写 $v = f(v)$ ，那么函数的返回值能改变调用环境中的 $v$ 。除此以外，函数调用 $f(v)$ 本身不能改变 $v$ ，这是因为仅把 $v$ 的值的拷贝传递给了 $f()$ 。然而在不同的编程语言中，函数调用本身也能改变调用环境中 $v$ 的值，把这样的机制称为引用调用(call-by-reference)。

让函数修改在参数表中引用的变量的值是很便利的。为了在C中进行引用调用，我们必须在函数定义体的参数表中使用指针，在函数调用时把变量的地址作为参数传递。然而在详细解释之前，我们需要明白指针是如何工作的。

在C中，指针有很多用途。在本章中，我们要解释如何把指针作为函数的参数。在下面的两章中，我们要解释指针用于数组和串的方式。在第12章“结构和抽象数据类型”中，我们将说明指针如何用于结构，在第13章“输入/输出和文件”中，我们将说明指针如何用于文件。

在本章中，我们还将讨论作用域规则和存储类型。全局变量在整个程序中都能用，除非在一个函数或块中对它进行了重新声明。

### 8.1 指针声明和赋值

在程序中使用指针访问内存和操纵地址。我们已经看到了把地址作为`scanf()`的参数的用法。像`scanf("%d", &v)`这样的函数调用会把适当的值存储在内存中的一个特定的地址。

如果 $v$ 是一个变量，那么 $\&v$ 就是在内存中存储 $v$ 的值的地址或位置。地址运算符 $\&$ 是一元的，与其他的一元运算符有同样的优先级和从右到左的结合性。可以在程序中声明指针变量，然后用地址作为值。声明

```
int i, *p;
```

声明 $i$ 是`int`型的， $p$ 是“指向`int`的指针”型的。任何指针的值的合法范围包括特殊的地址0和一组正整数，在具体C系统中会把它解释为机器地址。通常在`stdio.h`中把符号常量`NULL`定义为0。下面是一些对指针 $p$ 赋值的例子：

```
p = &i;
p = 0;
p = NULL;           /* equivalent to p = 0; */
p = (int *) 1307;    /* an absolute address in memory */
```

我们把第一个例子中的 $p$ 看作是“引用 $i$ ”、“指向 $i$ ”或“包含 $i$ 的地址”。编译器决定用什么地址存储变量 $i$ 的值。这随着机器的不同而不同，甚至在同一台机器上会随着执行的不同而不同。第二个和第三个例子说明了怎样把特殊值0赋值给变量 $p$ 。在最后的例子中，需要用类型转换`(int*)`来避免编译错误。在类型转换中的类型是“指向`int`的指针”。最后一个例子是不寻常的，因为程序员一般不把绝对地址赋值给指针，只有特殊值0是例外，不需要对这个特殊值类型转换。

## 8.2 地址和间接访问

我们要检查一些基本代码，并展示一些说明在内存中要发生什么的图。让我们从下述声明开始：

```
int a, b;
int *p;
```

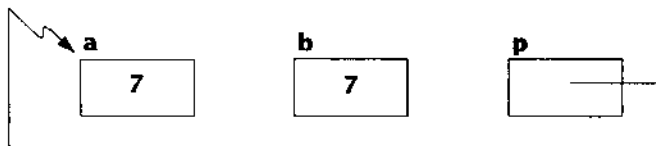
第一个声明为“a是一个int型的变量，b是一个int型的变量”。第二个声明为“p是一个指向int的指针变量”。只要声明了一个指针变量，该变量就有一个像int这样的它能引用的类型。对a和b的声明使得编译器在内存中分配了两个用于存储int型值的空间；对p的声明使得编译器在内存中分配了一个用于存储指向int的指针的空间，这里，变量的内容是无用的，因为还没有向它们赋值。



我们在图中使用了问号，这是因为我们不知道在三个变量中存储的值是什么。在执行了赋值语句

```
a = b = 7;
p = &a;
```

后，我们有：



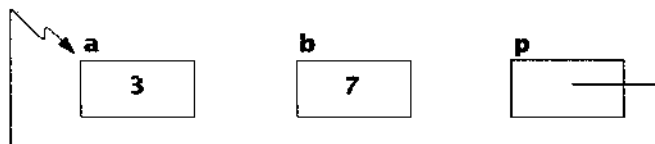
现在，我们能用指针p访问存储在a中的值。通过间接访问运算符\*来做到这一点，这个运算符是一元的，与其他的一元运算符有同样的优先级和从右到左的结合性。由于p是指针，表达式\*p的值就是p指向的变量的值。指针p的值是整型变量a的地址。术语间接访问(dereference)或间接(indirection)取自机器语言。p的直接值是内存地址，而\*p是p的间接值，即存储在p中的内存位置的值。考虑下面的语句：

```
printf("p = %d\n", *p); /* 7 is printed */
```

由于p指向a，a的值是7，p的间接引用值是7，所以该语句显示7。现在考虑

```
*p = 3;
printf("a = %d\n", a); /* 3 is printed */
```

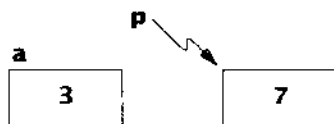
第一个语句为“把值3赋给p指向的对象”。由于p指向a，所以用3重写在内存中存储的a的值。这样，显示a的值时，显示出的是3。这时，内存中的情况如下：



下一个语句使得p指向b：

```
p = &b;
```

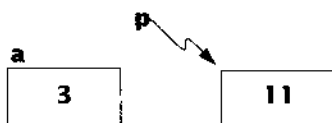
这时，我们绘制的下面的图与上图有所不同：



我们实际上不关心p在内存中存储的位置，我们仅关心p指向的对象。现在考虑代码

```
*p = 2 * *p - a;
printf("b = %d\n", b);    /* 11 is printed */
```

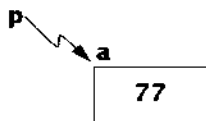
第一个语句为“把值2乘上p所指的位置中的值再减去a”。注意，赋值右边的表达式\*p先被求值，这是因为间接访问运算符\*的优先级高于二元算术运算符的优先级。



现在假设我们要从键盘上读入a的值。虽然如下的作法并没有什么优点，只是我们要用指针p完成读入。

```
p = &a;
printf("Input an integer: ");
scanf("%d", p);    /* put it at the address of a */
```

当调用scanf()时，从键盘上输入的字符被转换成十进制整数值，并把该值放于地址p处。我们可以把指针看作是地址，把地址看作是指针。如果在提示后我们输入了77，那么内存中的情况如下：



简单地讲，间接访问运算符\*是地址运算符&的逆。考虑下面的代码：

```
double x, y, *p;
```

```
p = &x;
y = *p;
```

先把x的地址赋给p，然后把p指向的对象的值赋给y。后两个语句与

```
y = *&x;
```

是等价的，它又与下一个语句是等价的：

```
y = x;
```

在声明中能对指针变量进行初始化，但程序初学者容易弄错这种表示法。下面是一个例子：

```
int a, *p = &a;
```

这个声明告诉编译器p是一个指向int的指针，其初始值是&a。当心：不要把该语句读作“p指向的对象被初始化为a的地址”。也要注意。

```
int *p = &a, a;          /* wrong */
```

是不对的。编译器必须在内存中为a分配空间，但在这之前要用a的地址对p进行初始化。编译器不允许这种超前。

在下面的函数定义中，参数p是指向int的指针，我们用它初始化声明中的变量。

```
void f(int *p)
{
    int a = *p, *q = p;
    .....
}
```

该段声明说明a是int型的，并把它初始化为p指向的对象；q是指向int的指针，并把它初始化为p。

下表说明了如何对一些表达式求值。要用心地正确理解指针的初始化部分。

声明和初始化		
int i = 3, j = 5, *p = &i, *q = &j, *r; double x;		
表达式	等价表达式	值
p == &i	p == (&i)	1
p = i + 7	p = (i + 7)	illegal
* * & p	* (* (& p))	3
r = & x	r = (& x)	illegal
7 * * p / * q + 7	((7 * (* p))) / (* q) + 7	11
* (r = & j) * = * p	(* (r = (& j))) * = (* p)	15

在ANSI C中，惟一能用于赋给指针的整数值是0。为了能为指针赋其他的值，就需要使用类型转换。与此相反，在传统C中，指针和整数可以混用。

在上表中，我们试图把&x赋给r，由于r是指向int的指针，而表达式&x是指向double的指针型，所以这是非法的。也要注意，如果我们用表达式

```
7 * * p / * q + 7
```

代替

```
7 * * p /* q + 7          /* trouble? */
```

我们会发现，编译器要把/\*作为注释的开头，这会导致一个难以对付的错误。

让我们编写一个简短的程序，用以说明在指针值和对它的间接访问值间的区别。我们还将说明怎样用%p格式输出指针值或地址。

```
#include <stdio.h>

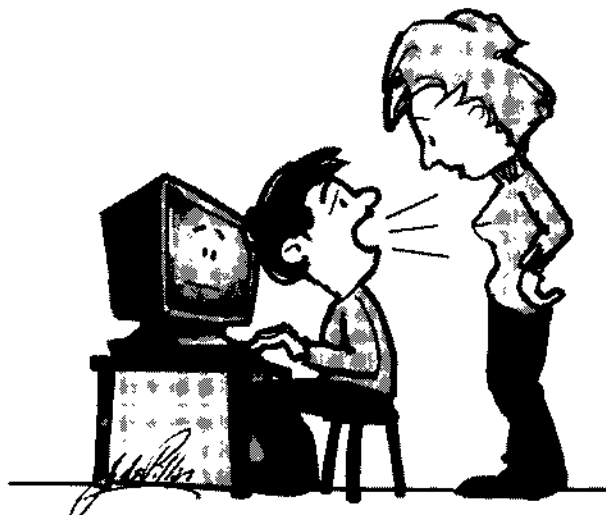
int main(void)
{
    int i = 777, *p = &i;

    printf(" Value of i: %d\n", *p);
    printf("Address of i: %u or %p\n", p, p);
    return 0;
}
```

下面是在我们的系统上该程序的输出：

```
Value of i: 777
Address of i: 234880252 or dfffcfc
```

一个变量在内存中的实际位置是与系统相关的。运算符\*用p的值作为内存的位置，返回存储在这个位置的值，并按照p的类型声明进行适当的解释。我们用%u格式按无符号十进制整数显示i的地址；用%p显示i的系统自然地址，在我们的系统中，这是一个十六进制整数。在一些传统的C系统中，%p格式是不可用的。在MS-DOS中，%u格式也并不总是能用的（请参见练习13）。



“你是对的，阿什利，C是不难的，但是别人教我说所有指针都是很难学的！”

### 8.3 指向void的指针

在传统C中，把不同类型的指针看作是赋值兼容(assignment-compatible)的，然而在ANSI C中，仅当两个指针的类型相同或其一是指向void的指针时，才能把一个指针赋值给另一个指针。因此，我们可以把void\*看作是通配指针类型。下表给出了一些合法和非法指针赋值的例子。

声 明	
<pre>int    *p; double *q; void   *v;</pre>	
合法赋值	非法赋值
<code>p = 0;</code>	<code>p = 1;</code>
<code>p = (int *) 1;</code>	<code>v = 1;</code>
<code>p = v = q;</code>	<code>p = q;</code>
<code>p = (int *) q;</code>	

在9.9节“动态内存分配”中，我们讨论了标准库函数calloc()和malloc()，这两个函数为数组和结构提供了动态内存分配。由于它们返回指向void的指针，我们可以写



```
int *a;
a = calloc(.....);
```

在传统C中，我们要用类型转换。

```
a = (int *) calloc(.....); /* traditional C */
```

在传统C中，把类型char \*用作（一种）通配指针类型，但要求使用类型转换。在传统C中不存在void \*。

## 8.4 引用调用

无论何时把变量作为参数传递给函数，都把它们值复制给对应的函数参数，在调用环境中，变量本身不发生变化。在C中是要严格遵守这种按值调用的机制的。在本节中，我们要描述怎样把变量的地址(address)作为参数传递给函数的参数，以实现引用调用。

对于要实现引用调用的函数，必须要在其定义的参数表中使指针。在调用函数时，必须把变量的地址作为参数传递。下面程序的函数swap()说明了这些思想。

```
#include <stdio.h>

void swap(int *, int *);

int main(void)
{
    int a = 3, b = 7;

    printf("%d %d\n", a, b); /* 3 7 is printed */
    swap(&a, &b);
    printf("%d %d\n", a, b); /* 7 3 is printed */
    return 0;
}
```

由于把a的地址和b地址作为参数传递给了swap()，在调用环境中swap()能交换a和b的值。

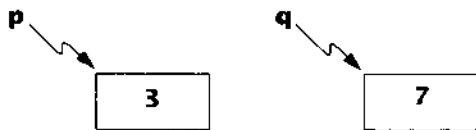
```
void swap(int *p, int *q)
{
    int tmp;

    tmp = *p;
    *p = *q;
    *q = tmp;
}
```

### 对函数swap()的解析

```
• void swap(int *p, int *q)
{
    int tmp;
```

函数的类型是void，这意味着没有返回值。参数p和q的类型是指向int的指针。变量tmp对于该函数是局部的，我们把它看作是临时存储。当我们在main()中用&a和&b作为参数调用这个函数时，此时内存中的情况如下：



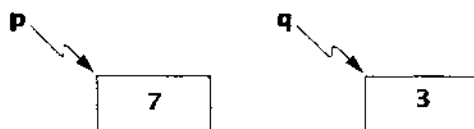
```

• tmp = *p;
  *p = *q;
  *q = tmp;

```

我们把它们读作：把p指向的对象的值赋给变量tmp；把q指向的对象的值赋给p指向的对象；把tmp的值赋给q指向的对象。

在执行三个语句后，此时内存中的情况如下：



如果对你来说指针是新概念，那么在执行每一个语句后你应该绘制出内存情况图（请参见练习2）。 ■

按下列步骤完成引用调用：

- 1) 把函数参数声明为指针。
- 2) 在函数体内使用间接访问指针。
- 3) 当调用函数时，把地址作为参数传递。

## 8.5 作用域规则

标识符的作用域(scope)是标识符在程序正文内能被分辨或访问的那部分程序段。这个概念与块(block)的概念有关，块是带有声明的复合语句。

作用域的基本规则是仅在声明标识符的块的内部能访问已定义的标识符，在块的外部标识符不能被识别。这条规则是容易遵循的，但若程序员由于各种原因在不同的声明中使用了相同的标识符，情况就不一样了，这就存在一个标识符指的是哪一个对象的问题。下边的例子说明了这种情况：

```

/* The Block Scope program. */
#include <stdio.h>

int main(void)
{
    int a = 2;           /* outer block a */
    int *p = &a;
    printf("%d\n", a);    /* 2 is printed */
    {
        int a = 7;       /* inner block a */
        printf("%d\n", a); /* 7 is printed */
        printf("%d\n", *p); /* 2 is printed */
    } /* back to the outer block */
    printf("%d\n", ++a);  /* 3 is printed */
    return 0;
}

```

对程序Block Scope的解析

```

• {
    int a = 2;           /* outer block a */
    int *p = &a;

```

一个函数是一个块。如果你混淆了不同的块中的a，考虑把代码重写为：

```

{
    int a_outer = 2;
    int *p = &a_outer;
    printf("%d\n", a_outer);

```

很显然，显示出2，p指向变量a\_outer。

```

• {
    int a = 7;          /* inner block a */
    printf("%d\n", a);  /* 7 is printed */
    printf("%d\n", *p); /* 2 is printed */
}                       /* back to the outer block */

```

花括号内引入了内部块。标识符a现在被声明为内部块中的a，它与外部块的a是有区别的，并且对外部块来说它是不可见的。如果我们把这段代码重写为

```

{
    int a_inner = 7;
    printf("%d\n", a_inner);
    printf("%d\n", *p); /* 2 is printed */
}

```

我们就能更好地理解这一点。

指针变量p正指向变量a\_outer，因此，第二个printf()显示的是2。 ■

每个块都引入自己的术语表。外部块中的名在内部块中是有效的，除非在内部块中又对外部块中的名进行了重定义。如果进行重定义，外部块中的名对内部块来说就是不可见的，或是被屏蔽了。内部块可以多重嵌套，其深度由系统限定。

## 8.6 存储类型

在C中的每一个变量和函数都有两个属性：类型(type)和存储类型(storage class)。四个存储类型分别是自动、外部、寄存器和静态，对应的关键字分别是：

```
auto extern register static
```

到目前为止，用于变量的大多数常用的存储类型是auto。然而，程序员需要知道所有的存储类型，它们都有重要的用途。

### 8.6.1 存储类型auto

在函数体中定义的变量缺省是auto。在四种存储类型中auto是最常用的。如果混合语句用变量声明开始，那么这些变量能在整个复合语句体范围内起作用。把带有声明的复合语句称为块(block)，以区别不用声明开始的复合语句。

在块内的变量声明暗示着存储类型是auto。关键字auto能用于显式地指定存储类型，下面是一个例子：

```

auto int    a, b, c;
auto double f;

```

由于缺省的存储类型是自动的，所以很少使用auto。

当进入块时，系统为自动变量分配内存。在块内，这些变量被定义，并被认为它们是局部于本块的。当退出块时，系统释放分配给自动变量的内存，因此，变量的值就丢失了。如果重新进入块，系统会为自动变量再次分配内存，但原先的值已经没有了。如果一个函数定义体含有声明，那么它就构成一个块。如果是这样，每一次函数调用都会建立一个新的环境。

### 8.6.2 存储类型extern

在块和函数间传送信息的一种方法是使用外部变量。若在函数的外部声明变量，就为变

量永久地分配存储，它的存储类型是`extern`。可以把对外部变量的声明与对函数或块内的变量声明看作是一样的。可以认为这种变量对于在其后声明的所有函数来说是全局的，在退出块或函数后，外部变量依然存在。下面的程序说明了这样的思想：

```
#include <stdio.h>

int a = 1, b = 2, c = 3;      /* global variables */
int f(void);                 /* function prototype */

int main(void)
{
    printf("%3d\n", f());      /* 12 is printed */
    printf("%3d%3d%3d\n", a, b, c); /* 4 2 3 printed */
    return 0;
}

int f(void)
{
    int b, c;                  /* b and c are local */
                                /* global b, c are masked */
    a = b = c = 4;
    return (a + b + c);
}
```

注意，我们可以写

```
extern int a = 1, b = 2, c = 3; /* global variables */
```

这种`extern`的用法在一些传统的C编译器上可能会有问题。虽然在ANSI C中允许这样使用`extern`，但不必须这样使用`extern`。定义在函数外部的变量的存储类型都是`extern`，即使不使用关键字`extern`，这样的变量的存储类型不会是`auto`或`register`。可以用关键字`static`，但那是特殊的用法，就像在8.7节“静态外部变量”中解释的那样。

用关键字`extern`告诉编译器“在本文件或其他文件中寻找它”。让我们重新编写上述程序，以说明关键字`extern`的典型用法。

```
#include <stdio.h>

int a = 1, b = 2, c = 3;      /* external variables */
int f(void);

int main(void)
{
    printf("%3d\n", f());
    printf("%3d%3d%3d\n", a, b, c);
    return 0;
}
```

文件`file2.c`的内容：

```
int f(void)
{
    extern int a;              /* look for it elsewhere */
    int b, c;

    a = b = c = 4;
    return (a + b + c);
}
```

可分别对这两个文件进行编译。第二个文件中的`extern`告诉编译器变量`a`被定义在别处，可能在本文件中，也可能在另一个文件中。当编写大程序时，这种分别编译文件的能力是很重要的。

外部变量从不会消失。因为外部变量在整个程序执行期间都是存在的，所以可以用外部

变量在函数间传递值。然而，如果对标识符进行重定义，也会隐藏外部变量。表达外部变量的另一种方法是认为外部变量被定义在包含整个程序的块中。

可以用两种方式向函数传递信息：一种是用外部变量，另一种是用参数机制。虽然存在例外情况，但还是使用参数机制为好。这样有助于改善代码的模块性，降低不期望的副作用。

当函数从其内部而不是通过参数表改变全局变量时，会产生副作用。这样的构造容易引发错误。正确的作法是通过参数和返回机制影响全局变量。坚持这种作法会改善模块性和可读性，因为变化是局部性的，这样就容易编写和维护程序。

所有的函数的存储类型都具有外部存储类型，这意味着我们在函数体中和函数原型中都可以使用关键字extern。例如，

```
extern double sin(double);
```

是sin()的有效函数原型，我们可以编写其函数定义为：

```
extern double sin(double x)
{
    ....
}
```

### 8.6.3 存储类型register

存储类型register告诉编译器应该把有关的变量存储在高速的内存寄存器中，在物理上和语义上要尽可能地提供存储类型register。由于资源限制和语义约束，有时做不到这点，当编译器不能分配合适的物理存储器时，就把这种存储器缺省为自动的。通常编译器仅有为数不多的这样的寄存器可供使用；其中的很多为系统所用，而不能用于别处。

基本上使用存储类型register是要试图改善执行速度。当关心速度时，程序员可能选择一些最经常访问的变量，并把它们的存储类型声明为register，这样的变量可以是循环变量和函数参数。下而是一个例子：

```
{
    register int i;
    for (i = 0; i < LIMIT; ++i) {
        ....
    }
} /* block exit will free the register */
```

声明register i;等价于register int i;如果在声明中说明了存储类型而没有说明类型，那么缺省类型就是int。

注意，在我们的例子中，把寄存器变量i在尽可能接近使用它的位置处进行声明。这使得物理寄存器得到了最大的利用，仅在需要时才使用寄存器。始终要记住，仅把寄存器声明作为向编译器提的建议。

### 8.6.4 存储类型 static

静态声明具有两个重要但截然不同的用途。最基本的用途是允许局部变量保存它的原有值，以便再进入块时使用。与此相反，当退出块后，普通的自动变量保存的值都丢失了，再进入时必须重新初始化。第二种用法更巧妙，与外部声明有关，在8.7节要对此讨论。

作为用static保存值的一个例子，我们编写一个看起来有所不同的函数框架，这依赖于它被调用的次数。

```
void f(void)
{
    static int    cnt = 0;

    ++cnt;
    if (cnt % 2 == 0)
        .....          /* do something */
    else
        .....          /* do something different */
}
```

在第一次调用函数时，把变量cnt初始化为0，在退出函数时，cnt的值被保存在内存中。当再次调用函数时，就不用再对cnt进行初始化。cnt仍保留着上次调用函数时的值。在f()内把cnt声明为static int型，使得cnt为f()私有的。如果把cnt声明在函数外边，那么其他函数也能访问它。

## 8.7 静态外部变量

第二种用法更巧妙，与外部声明有关。用外部结构，能提供对于程序的模块性来说非常重要的私有(privacy)机制。私有机制的含义是在别处可以访问变量或函数的可见性或作用域限制。

乍一看静态外部变量是不必要的，外部变量已经能够在块间和退出函数后保存其值。不同的是静态外部变量是有作用域限制的外部变量。作用域是声明变量的源文件的残余。因此，对于在文件中早期定义的函数或在其他文件中定义的函数来说，这样的变量是不可用的，即使这些函数试图用存储类型关键字extern也是如此。

```
void f(void)
{
    .....          /* v is not available here */
}

static int    v;    /* static external variable */

void g(void)
{
    .....          /* v can be used here */
}
```

让我们用该方式提供一个变量，该变量对一组函数来说是全局的，但同时对本文件来说是私有的。我们编写两个伪随机数发生器，对于它们都使用相同的种子。(该算法基于线性同余方法；请参见Donald Ervin Knuth著述的Addison-Wesley于1981年出版的《*The Art of Computer Programming*》第2版的第二卷《*Seminumerical Algorithms*》)。

```
/* A family of pseudo random number generators. */

#define INITIAL_SEED      17
#define MULTIPLIER        25173
#define INCREMENT         13849
#define MODULUS           65536
#define FLOATING_MODULUS  65536.0

static unsigned    seed = INITIAL_SEED; /* external, but
                                         private to this file */

unsigned random(void)
```

```

{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
    return seed;
}

double probability(void)
{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
    return (seed / FLOATING_MODULUS);
}

```

函数`random()`产生在0和MODULUS间的整数随机序列。函数`probability()`产生在0和1间的浮点数随机序列。

注意，依赖变量`seed`的旧值，对`random()`和`probability()`的调用为`seed`产生一个新值。由于`seed`是静态外部变量，对此文件来说它是私有的，它的值在函数调用之间是被保存的。我们现在能在其他文件中创建调用这些随机数发生器而不担心副作用的函数。

对`static`的最后一个应用是把它作为函数定义和原型的存储类型说明符，这能限定函数的作用域。静态函数仅在定义它的文件中是可见的。与普通的函数不同，普通的函数能由其他文件访问，`static`函数在它自己的文件内是可用的，在其他文件内则是不可用的。再次强调，该功能对开发函数定义的私有模块是很有用的。

```

void f(int a)
{
    ..... /* g() available here, but not in other files */
}

static int g(void)
{
    .....
}

```

## 8.8 缺省的初始化

在C中，未被程序员显式初始化的外部变量和静态变量都由系统初始化为0，对数组、串、指针、结构和联合的初始化都是这样。对数组和串来讲，这意味着把每个元素都初始化为0；对于结构和联合来讲，这意味着把每个成员都初始化为0。与此相反，系统通常不对自动和寄存器变量进行初始化，这意味着这样的变量一开始含有无用的值。虽然一些C系统把自动变量初始化为0，但这不可依靠，因为这样会使代码不可移植。

## 8.9 例子：字符处理

使用`return`的函数能向调用环境传回一个值。如果调用环境需要多个传回的值，必须把地址作为参数传递给函数。为了说明这个想法，让我们编写一个用特殊的方法处理字符的程序。下面是我们要完成的任务：

### 处理字符的目标

- 从输入流读入字符，直到遇见EOF为止。
- 把所有的小写字符转换成大写字符。
- 一行显示三个词，把各词用单空格分开。
- 对字符和显示的字母计数。

在这个程序中，我们把词看作是最长的不包含空白字符的字符序列，下面是main()：

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

#define  NWORDS  3          /* number of words per line */

int  process(int *, int *, int *);

int main(void)
{
    int  c, nchars = 0, nletters = 0;

    while ((c = getchar()) != EOF)
        if (process(&c, &nchars, &nletters) == 1)
            putchar(c);
    printf("\n%s%5d\n%s%5d\n\n",
           "Number of characters:", nchars,
           "Number of letters:  ", nletters);
    return 0;
}
```

在函数process()中对各字符进行处理。由于在调用环境中要改变变量c、nchars、和nletters，所以要把这些变量的地址作为参数传递给process()。注意，c是int型的，而不是char型的，这是因为c最终必须是EOF，而EOF不是char型的。还要注意，仅当process()的返回值是1才显示一个字符。在本文中，我们把1看作是一个字符已经被适当处理并准备显示的标记，我们用0作为一个字符不被显示的标记。当出现连续的空白字符时，会发生这种情况。让我们看一下process()是如何工作的。

```
int process(int *p, int *nchars_ptr, int *nletters_ptr)
{
    static int  cnt = 0, last_char = ' ';

    if (isspace(last_char) && isspace(*p))
        return 0;

    if (isalpha(*p)) {
        ++*nletters_ptr;
        if (islower(*p))
            *p = toupper(*p);
    }

    else if (isspace(*p))
        if (++cnt % NWORDS == 0)
            *p = '\n';
        else
            *p = ' ';
    ++*nchars_ptr;
    last_char = *p;
    return 1;
}
```

在我们解释process()之前，我们要给出程序的一些输出。首先我们要编译程序，把可执行的代码放于文件process中，然后我们创建名为data的包含如下行的文件：

```
she sells sea shells
by the seashore
```

注意，我们是故意在文件中放入了连续的空格。现在，如果我们发出了命令

```
process < data
```

在屏幕上会出现如下的信息：

```
SHE SELLS SEA
SHELLS BY THE
```



```
SEASHORE
Number of characters:  37
Number of letters:    30
```

### 对函数process()的解析

```
• int process(int *p, int *nchars_ptr, int *nletters_ptr)
{
    static int  cnt = 0, last_char = ' ';
```

函数的类型是int，这意味着函数的返回值的类型是int。函数的参数是三个指向int的指针。虽然我们把p看作是指向字符的指针，此处的声明应该与它在main()中的用法是一致的。局部cnt和last\_char的存储类型是static，因此它们仅被初始化一次，并且它们的值在函数调用间被保存，如果这些变量的存储类型是auto，在每次调用函数时都要对它们进行初始化。

```
• isspace(last_char)
```

在ctype.h中定义了宏isspace()。如果参数是空格、跳格或换行，那么它返回一个非零值(true)。

```
• if (isspace(last_char) && isspace(*p))
    return 0;
```

如果读入的字符是空白字符，并且p指向的字符也是空白字符，那么返回到调用环境的值是0。当接收到该值时，返回到调用环境（即返回到main()），不显示当前的字符。

```
• if (isalpha(*p)) {
    ++*nletters_ptr;
    if (islower(*p))
        *p = toupper(*p);
}
```

如果p指向的字符是字母，那么由nletters\_ptr指向的对象的值要加1。此外，如果由p指向的对象的值是小写字母，那么对由p指向的对象赋予相应的大写字母。

```
++*nletters_ptr;
```

让我们在一些细节上深究这个语句。增量运算符++和间接运算符\*都是一元运算符，都遵从自右到左的运算。这样，上述语句等价于

```
++(*nletters_ptr);
```

被增量的是在调用环境中被间接引用的值。要注意到++\*nletters\_ptr与\*nletters\_ptr++是不等价的，后一个表达式与\* (nletters\_ptr++)等价，这使得当前的指针值被间接引用，并且指针本身也被增量。这是指针运算的一个例子。

```
• else if (isspace(*p))
    if (++cnt % NWORDS == 0)
        *p = '\n';
    else
        *p = ' ';
```

如果p指向的字符是空白字符，那么last\_char就不能是空白字符；我们已经处理过这种情况。无论这个字符是什么，我们都要显示换行或空格，这依赖于cnt的增量值。由于符号常量NWORDS的值是3，我们每逢3的倍数次就显示换行，其余两次显示空格。这样的效果是每行至少显示3个词，并用空格分隔。

```
• ++*nchars_ptr;
  last_char = *p;
  return 1;
```

我们首先对由`nchars_ptr`指向的对象的值增量；然后我们把由`p`指向的对象的值赋给`last_char`；最后，向调用环境返回值1，以指明要显示一个字符。 ■

## 8.10 函数声明和函数定义

对于编译器而言，通过函数调用、函数定义、显式的声明和原型都可以生成函数声明。如果在对函数进行声明、定义或建立原型前就遇到了像`f(x)`这样的函数调用，那么编译器就为函数假设一种缺省的声明形式：

```
int f();
```

对于函数的参数表不做任何假设。现在假设先出现的是如下的函数定义：

```
int f(x)          /* traditional C style */
double x;
{
    .....
```

这为编译器提供了声明和定义。此处对于函数的参数表同样不做任何假设。仅传递一个类型为`double`的参数是程序员的责任。像`f(1)`这样的函数调用是要失败的。假如我们用以下的新风格进行定义：

```
int f(double x)   /* ANSI C style */
{
    .....
```

现在编译器知道参数表。在这种情况下，如果把`int`型的值作为参数传递，那么会把该值正确地转换成`double`型。

函数原型是函数定义的一种特殊情况。良好的编程风格在使用函数前给出了函数定义（新风格）或函数原型。加进标准头文件的主要原因是它包含有函数原型。

对函数定义和函数原型有一定的限制。如果存在函数存储类型标识符，那么它可能是`extern`或`static`，但不能两者都是；不能是`auto`和`register`。函数不能返回数组类型和函数返回类型，但能返回表示的数组或函数的指针。在参数类型表中惟一能出现的存储类型标识符是`register`。不能对参数进行初始化。

## 8.11 类型限定符`const`和`volatile`

ANSI C委员会已经在C语言中增加了关键字`const`和`volatile`。在传统C中它们是不可用的。因为它们用于声明中，告诉编译器怎样使用标识符，所以把它们称为类型限定符(type qualifier)。

让我们先讨论怎样使用`const`。通常在声明中，`const`出现在存储类型之后类型之前，考虑声明

```
static const int k = 3;
```

我们把它读作“`k`是一个静态存储类型的整型常量”。由于已经用`const`限定了`k`的类型，我们可以对`k`进行初始化，但以后不能再对`k`赋值、增量或减量。即使用`const`对变量作了限制，也不能在另一个声明中用变量描述数组的大小。

```
const int n = 3;
int v[n];          /* the compiler will complain */
```

因此，由const限定的变量与符号常量是不等价的。

不能把由const限定的变量的地址赋值给非限定的指针。下面的代码会引起编译器报警：

```
const int    a = 7;
int         *p = &a;    /* the compiler will complain */
```

其原因是，p是一个普通的指向int的指针，随后我们可以在像++\*p这样的表达式中使用它，但那将会改变a的存储值，这与a是常量的概念相违背。另一方面，如果我们写

```
const int    a = 7;
const int    *p = &a;
```

那么编译器会正常。把后一个声明读作“p是一个指向int的常量的指针，其初始值是a的地址”。注意，p本身不是一个常量，我们可以用一些其他的地址向它赋值，但我们不能向\*p赋值。不应该修改p指向的对象。

假设我们要使p本身是常量，而不是a，这可以用如下的声明来完成：

```
int          a;
int * const  p = &a;
```

我们把后一个声明读作“p是一个指向int的常量指针，其初始值是a的地址”。此后我们不能向p赋值，但可以向\*p赋值。现在考虑

```
const int    a = 7;
const int * const  p = &a;
```

后一个声明告诉编译器p是一个指向常量int的常量指针。对p和\*p都不能进行赋值、增量或减量。

与const相比，类型限定词volatile很少使用。volatile对象是这样的一个对象：硬件可以用一些意想不到的方式修改它。考虑声明

```
extern const volatile int  real_time_clock;
```

extern意味着“在本文件或其他文件中寻找它”。限定符volatile指明可以用硬件影响该对象。由于const也是限定符，在程序中不能向该对象进行赋值、增量或减量。

## 8.12 风格

我们在程序中经常会发现把p、q和r用作指针变量的标识符。用p代表指针(pointer)，在字母表中q和r是p的后继字母，这已是一种自然习惯。以类似的方式，也把p1，p2，……用作指针变量的标识符。其他的常用方式还有：用p\_作为指针标识符的前缀名，如p\_hi和p\_lo；用\_ptr作为指针标识符的后缀名，如nchars\_ptr。

一种用于指针的可选的声明风格是：

```
char* p;
```

它等价于

```
char *p;
```

一些程序员更喜欢这种风格，这是因为\*现在更靠近所指向的相关类型，但必须注意到，

```
char* p, q, r;
```

与

```
char *p, *q, *r;
```

不等价，而是与

```
char *p, q, r;
```

等价。

我们下面要讨论函数的副作用(side effect)。当函数调用改变调用环境中的变量值时，会产生副作用。良好的编程风格要用return或参数机制影响这样的变化。为了说明这一点，让我们重新编写程序swap。

```
#include <stdio.h>

int    a = 3, b = 7;          /* global variables */
void   swap(void);           /* function prototype */

int main(void)
{
    printf("%d %d\n", a, b);  /* 3 7 is printed */
    swap();
    printf("%d %d\n", a, b);  /* 7 3 is printed */
    return 0;
}
```

我们已经把对a和b的声明移出main()，使得它们成为全局变量。现在我们重写swap()函数。

```
void swap(void)               /* very bad programming style */
{
    extern int    a, b;
    int          tmp;

    tmp = a;
    a = b;
    b = tmp;
}
```

对全局变量a和b的修改是函数调用swap()产生的副作用。用这种编程风格编写的大程序，是难以让人读懂和维护的。理想的情况是，人们编写的代码在局部上是可理解的。

### 8.13 常见的编程错误

在学习使用指针时，程序初学者常犯概念性错误。一个典型的例子是

```
int *p = 3;
```

该声明企图对由p指向的对象的值进行初始化，但这是对p本身进行初始化，这很可能不是想要做的。假设我们试着把它改写成

```
int *p = &i, i = 3;
```

现在出现了更微妙的错误。C没有提供超前的能力。在p被初始化为i的地址时，还没有为变量i分配空间。我们可以用下述语句来改正这个错误；

```
int i = 3, *p = &i;
```

首先为i分配了空间，把i初始化为3；然后为p分配了空间，把p初始化为i的地址。

在处理指针时，程序员必须要学会区别指针p和对它的引用\*p。为了尽量减少混乱，应该用表明指针用途的名字对指针命名。下面是一个不能效仿的例子。假设v1和v2是浮点变量，我们要在调用环境中通过名为swap(&v1, &v2)的函数交换这两个变量的值。为了对swap()编码，可以写：

```
void swap(double *v1, double *v2)      /* poor style */
{
    .....
```

但是现在出现了混乱。在main()中，把标识符v1和v2用作类型为double的变量名，但在swap()中，把它们又用作类型为指向double的指针的变量名，更好的办法是使用p\_v1和p\_v2。用名字来清楚地表明指针的用法有助于程序员少犯错误，也有助于其他人阅读代码，以理解其含义。

当然，并不是所有的值都存储在内存中可以访问到的位置，记住下述的禁律是有用的。

不被指向的结构		
不能指向常量	&3	非法
不能指向普通表达式	&(k + 99)	非法
不能指向寄存器变量	register v; &v	非法

可以把地址运算符用于变量和数组元素。如果a是一个数组，那么像&a[0]和&a[i+j+7]这样的表达式是有意义的（请参见第9章“数组和指针”）。

对块的一种常见用法是用它进行调试。想象一下，我们处于包含变量v的一段有毛病的代码中，通过向代码中插入一个临时块，我们可以用局部变量，而不涉及程序的其余部分。

```
{
    static int    cnt = 0;      /* debugging starts here */
    printf("*** debug: cnt = %d    v = %d\n", ++cnt, v);
}
```

变量cnt是局部于块的，它不会干涉外部块中的同名变量。由于它的存储类型是static，当再次进入该块时，它还保持原有值。此处用cnt记录执行该块的次数（或许我们可以在一个for循环中）。我们假设在外部块中声明了变量v，因而在本块内也知道它。为了调试，我们要显示它的值。在我们找到毛病以后，该块就是多余的了，我们要删掉它。

## 8.14 系统考虑

在传统C中，可以把整数赋给指针，但在ANSI C中，仅可把特殊的整数值0赋给指针，而不论指针是什么类型。如果把非零值赋给指针，一些编译器会给出出错信息；还有些编译器仅给出一个警告。假设它们都发出错误信息，但很多ANSI C编译器能用于对新旧C代码都必须编译的环境中。

显式地使用绝对地址的程序经常是不可移植的。不同的系统有不同的地址空间，它们使用它们的地址空间的方法可能是相互不兼容的。如果我们必须要用绝对地址编程，最好用#define机制，以局部化任何可能的与系统相关的代码。

在指针表达式中不应该混用类型。作为一个例子，假设p是指向char的指针，q是指向int的指针，那么赋值表达式p = q就混用了类型。虽然在ANSI C中把这样的表达式认为是非法的，但很多编译器仅给出了一个警告。例如，在Turbo C中我们可以得到：

Suspicious pointer conversion in.....

一个正确的格式是使用类型转换。例如，我们可以写

```
p = (char *) q;
```

当编写可移植的代码时，程序员必须要留意所有的编译器警告，因为在一个系统上所警告的在另一个系统上可能就是被禁止的。

在ANSI C中，%p格式能用在显示指针值的printf()语句中，通常显示出的是十六进制数。在一些传统的C编译器中，这个格式是不可用的。

最好把函数原型放在要引入的头文件中，或放在文件的顶部（在#include、#define和typedef之后），这使得函数原型是可见的。在ANSI C中，放在一个块（例如函数的体）中的函数原型被假设其范围就是该块。在传统C中，无论函数声明放在文件的什么地方，它在文件中经常都是可见的。一些ANSI C编译器仍然这样处理函数原型和函数声明，虽然在技术上这样做是错误的。在一些情况下，这会引起移植上的困难。

## 8.15 转向C++

引用声明是C++的一个新特征。对于在一个引用的初始化中描述的对象，引用声明把它的标识符声明为可选择名或别名，这使得引用调用的参数得以简化。例如，

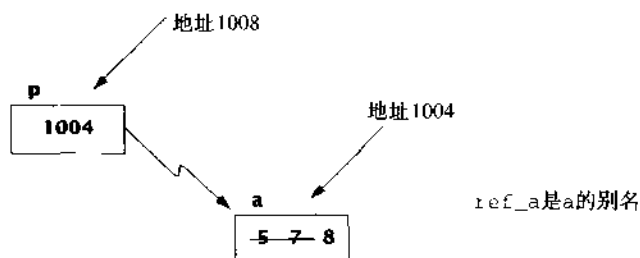
```
int      n;
int&     nn = n      //nn is alternative name for n
double  a[10];
double&  last = a[9]; //last is an alias for a[9]
```

必须要对定义的引用声明进行初始化，通常用于引用声明的是简单变量。被初始化的是左值表达式，它给出了变量在内存中的位置。在上述例子中，名字n和nn互为别名；也就是说，它们指的是同一个对象。修改nn等价于修改n，反之亦然。名字last是单数组元素a[9]的可选择名。一旦对这些进行了初始化，就不能改变它们。

声明变量i后，在内存中它就有了一个地址。当声明一个指针变量p并用&i对它初始化后，它就和i有本质的区别；当声明引用变量r并用i对它初始化后，它与i是一致的。这没有把同一对象的其他名字分开。

下述定义说明了指针、间接访问和别名的用法。假设整型变量a的内存地址是1004，指针变量p的内存地址是1008。

```
int  a = 5;      //declaration of a
int* p = &a;     //p points to a
int& ref_a = a;  //alias for a
*p = 7;         //lvalue in effect a is assigned 7
a = *p + 1;     //rvalue 7 added to 1 and a assigned 8
```



注意，在上图中，对a的值的任何改变都等价于改变ref\_a，这样的改变影响对p的值的间接访问。可以向指针p赋另一个值，p就与a无关了。然而，在这个作用域中互为别名的a和ref\_a必须指的是同一个对象。

可以把这些声明用作引用调用的参数，这使得C++拥有了直接的引用调用的参数。在C中，该特征是不可用的；它与Pascal中的var类似。

用这种机制，把函数order()重新编码为：

```
void order(int& p, int& q)
{
    int temp;

    if (p > q) {
        temp = p;
        p = q;
        q = temp;
    }
}
```

在main()中要对其进行原型化，并调用它：

```
void order(int&, int&);
int main(void)
{
    int i, j;
    .....
    order(i, j);
    .....
}
```

让我们用这个机制编写函数greater()，如果第一个值大于第二个值，则它会交换这两个值。

```
int greater(int& a, int& b)
{
    if (a > b) {           //exchange
        int temp = a;
        a = b;
        b = temp;
        return 1;
    }
    else
        return 0;
}
```

现在，如果i和j是类型为int的变量，则

```
greater(i, j)
```

使用对i的引用和对j的引用交换它们的值（如果需要）。

## 小结

- 指针变量通常用NULL或其他变量的地址作为值。
- 地址运算符&和间接引用运算符\*是一元运算符，它们和其他的一元运算符一样，都遵循同样的优先次序和从右到左的结合性。如果v是变量，那么表达式\*&v等价于v。然而要记住，如果v的存储类型是register，那么操作&v是不允许的。
- 在C中要严格遵守按值调用机制。这意味着当把表达式作为参数传递给函数时，传递的是表达式的拷贝。像f(v)这样的函数调用的本身不能改变调用环境中v的值。
- 通过使用指针、地址运算符&和间接引用运算符\*能实现引用调用。
- 为了实现引用调用，程序员必须用指针作为函数定义头中的形式参数，然后把一个值赋给函数体内的间接引用指针能改变调用环境中的变量的值。当调用这样的函数时，用地址作为实际参数。

- `auto`、`extern`、`register`和`static`是四种最常用的存储类型。进入块自动变量会出现，退出块自动变量会消失。若在内部块中对外部块的标识符重定义，则这样的变量会被隐藏。
- 关键字`extern`意味着“在本文件或其他文件中寻找它”。
- 在函数外部声明的所有变量的存储类型都是`extern`，不必再用关键字`extern`。可以在整个程序中使用这些变量。通过重新声明能隐藏它们，但不能毁坏它们的值。
- 所有的函数的存储类型都可以是`extern`。函数的类型说明符是`int`，除非显式地进行另外的声明。在`return`语句中的表达式类型必须与函数的类型兼容。
- 存储类型`register`能用于改善执行速度。在语义上与自动等价。
- 存储类型`static`用于保存变量退出值，也用于限定外部标识符的作用域。
- 作用域规则是与标识符有关的可视性约束。带有外部标识符的关键字`static`提供了对程序模块化来说非常重要的私有的形式。考虑如下代码：

```
static void f(int k)
{
    .....
}

static int  a, b, c;
    .....
```

在整个文件中都知道`f()`，但在其他文件中则不然。仅在本文件且仅在声明它们后的那些地方知道`a`、`b`和`c`。

- 系统把程序员没有显式地进行初始化的外部和静态变量初始化为0。

## 练习

1. 下面的代码会显示出什么结果？

```
int    i = 5, *p = &i;

printf("%p %d %d %d %d\n",
       p, *p + 2, **&p, 3**p, **& p+4);
```

注意，我们用`%p`格式显示指针值。所有的ANSI C编译器都理解这种格式。如果你把`%p`变为`%d`会发生什么情况？在UNIX中，它可能工作，但在MS-DOS中可能会失败。如果你可以使用MS-DOS，也试一下`%u`格式。

2. 考虑如下代码：

```
char    c1 = 'A', c2 = 'B', tmp;
char    *p = &c1, *q = &c2;

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

绘制一张反映在各声明和语句执行后内存情况的图。

3. 考虑代码

```
double  *p, *q;

p = 3;
q = 7 - 5 - 2;
```

你的编译器会对这些行中的一行给出警告，但对其他行则不给出警告。请解释为什么。



如果你使用指向int的指针，而不使用指向double的指针，你的编译器会给出警告吗？

4. 如果i和j是int型的，p和q是指向int的指针，下述的哪一个赋值表达式是非法的？

```
p = &i      p = &*i      i = (int) p      q = &p
*q = &j      i = (*&)j      i = *&j      i = (*p)++ + *q
```

5. 用以下的声明编写一个程序：

```
char  a, b, c, *p, *q, *r;
```

显示出编译器赋给这些变量的位置。从显示出的值来看，你能说出分配给各变量的字节数是多少吗？

6. 以循环的方式编写一个把5个字符变量存储的值循环移位的程序。该函数应该按如下的方法工作。假设c1, c2, ..., c5是char型的变量，它们的值分别是'A', 'B', ..., 'E'。函数调用shift(&c1, &c2, &c3, &c4, &c5)使得变量c1、c2、...和c5的值分别为'B', 'C', 'D', 'E', 'A'。应该用如下的代码开始函数的定义：

```
void shift(char *p1, char *p2, char *p3, char *p4, char *p5)
{
    ....
}
```

调用它5次，显示结果依次是BCDEA、CDEAB、DEABC、EABCD和ABCDE，据此来测试你的程序。

7. 编写一个对三个变量中存储的值进行排序的函数。例如，c1、c2和c3是三个字符变量，其值分别是'C'、'B'和'D'。函数调用order\_chars(&c1, &c2, &c3)应该使c1、c2和c3的存储值分别为'B'、'C'和'D'。编写一个程序，测试你的函数。

8. 我们在8.9节“例子：字符处理”中编写的处理字符的程序，可以不使用函数process()，而把所有的代码放入main()中。当然，我们使用process()是为了说明指针的用法。不使用指针，重新编写该程序。

9. 在C.4“压缩和解压”中，我们编写了函数unpack()。它一次能解压int的一个字节。重新编写一个程序，使它一次能解压所有的字节。在机器字为四个字节的机器上，以下面的代码开始你的函数：

```
/* Unpack the packed int p into 4 characters. */
void unpack(int p, char *pa, char *pb, char *pc,
            char *pd)
{
    unsigned mask = 255; /* turn on low-order byte */
    ....
}
```

编写一个程序，检查你的函数。你的程序应该使用bit\_print()函数（请参见C.3“按位显示int”）。

10. 你的C系统用几个字节存储指针变量？存储指向char的指针比存储long double的指针所用的空间少吗？用sizeof运算符找出答案。编写一个程序，显示一个存储指向各种基本类型的指针所需要的字节数的表。

11. 因为运算符\*既表示间接运算符又表示乘法运算符，有时不能立即分清。考虑下面的代码：

```
int  i = 2, j = 4, k = 6;
int  *p = &i, *q = &j, *r = &k;
```

```
printf("%d\n", * p * * q * * r);
printf("%d\n", ++ * p * -- * q * ++ * r);
```

它会显示出什么？写下你的答案，然后执行这段代码，检查你的答案。按两种方式重新编写printf()中的表达式。第一种，移去两个表达式中的所有空格。你的编译器对此产生了混乱了吗？第二种，留下二元运算符左右的空格，移去一元运算符与其操作对象之间的空格。此时编译器产生的结果有变化吗？如果你认为这样的格式编排是针对无能之辈的，就把所有的乘法运算符改为除法运算符，重做这个练习。结果可能会令你惊奇。

12. 下面的程序有一个概念上的错误，试一下看你能否找到它。

```
#include <stdio.h>

#define LUCKY_NUMBER 777

int main(void)
{
    int *p = LUCKY_NUMBER;

    printf("Is this my lucky number? %d\n", *p);
    return 0;
}
```

在我们的系统上，该程序的输出为：

```
Is this my lucky number? 24864
```

你能解释这个输出吗？

13. 正像存在着指向int的指针那样，也存在着指向int的指针的指针。用以下声明编写一个测试程序：

```
int v = 7, *p = &v, **q = &p;
```

标识符q是指向int的指针的指针，它的初始值是p的地址。用语句

```
printf("%d\n%d\n%d\n", q, *q, **q);
```

显示一些值来测试这个概念。像q==&p这样的表达式有意义吗？在你的程序中使用这样的表达式。当心：如果你使用的是MS-DOS操作系统，当显示指针的值时必须要小心。可以料到%p格式能工作，当然在UNIX中它也工作。%u格式在小内存模式下能工作，但在大内存模式下则不能工作。试一下如下代码：

```
printf("%p\n%p\n%d\n", q, *q, **q);
printf("---\n");
printf("%u\n%u\n%u\n", q, *q, **q);
```

在你检查输出时，你能说出代码是否有毛病吗？

14. 在本练习中，我们继续讨论上一个练习中的思想。考虑代码

```
int v = 7, *p = &v, **q = &p;

printf("%d\n%d\n%d\n%d\n%d\n%d\n%d\n%d\n",
    &v, *&v, &p, *&p, **&p, &q, *&q, **&q, ***&q);
```

输出的图案是很有意思的。你能解释一些数为什么重复吗？注意，我们使用了组合\*&，而不是&\*。存在&\*在语义上是正确的情形吗？在MS-DOS中，试一下用%u代替所有的%d格式。如果看起来不正确，用%p代替所有的%d格式，但第二、第五和最后一个不变。

15. 编写一个测试程序，把r声明为指向int的指针的指针的指针，扩展上一个练习中的思想。

16. 在ANSI C中, 在一个声明中至多能说明一个存储类型。如果在你的系统上试一下下面的代码, 会发生什么情况?

```
#include <stdio.h>

static extern int a = 1;

int main(void)
{
    printf("a = %d\n", a);
    return 0;
}
```

本练习的要点是你要用静态外部变量, 但编译器不允许你做这样的声明。你应该怎样做?

17. 如果你在声明中使用const, 然后试图做一个不适当的初始化或赋值, 你的编译器会报警, 所报警的是错误还是警告, 是与系统相关的。在你的系统上试一下下面的代码:

```
const double x = 7.7;
double *p = &x; /* compiler error or warning? */

printf("x = %g\n", *p);
```

重新编写代码, 继续使用const, 但要使编译器不报警。

18. C++: 重做练习6, 要求循环移位, 并使用引用声明。

19. C++: 把8.7节“静态外部变量”中的随机数发生器代码编写成一个内联C++函数, 并把它在C中作为普通的函数与在C++中作为内部函数产生100000个随机数所需的时间做比较。依据这些结果, 人们经常说在很多情况下C++实际上比C更有效。

20. C++: 编写一个通用例程mem\_swap(void\* x, void\* y, size\_t, n), 这个例程把从内存地址x处开始的n个字节与从内存地址y处开始的n个字节进行交换。该函数应该能断定n的值是否合适, 若有重叠, 则不能进行交换。

21. Java: 在Java中, 为了在同一个类中一个方法能调用另一个方法, 我们要写被调用的方法名和括号内的适当的参数表, 这些参数必须在数目上和类型上与方法定义的参数表中的参数匹配。也就是说, 对各参数求值, 把值用于被调用的方法中, 以初始化相应的形参。这样, 如果把一个变量传递给一个方法, 在调用环境中变量存储的值不会被改变。在下面的例子中, 我们试图用类似于8.4节“引用调用”中的方法, 交换两个局部变量的值。

```
//FailedSwap.java - Call-By-Value test   Java by Dissection
page 113.
class FailedSwap {
    public static void main(String[] args) {
        int numOne = 1, numTwo = 2;
        swap(numOne, numTwo);
        System.out.println("numOne = " + numOne);
        System.out.println("numTwo = " + numTwo);
    }

    static void swap(int x, int y) {
        int temp;
        System.out.println("x = " + x);
        System.out.println("y = " + y);
        temp = x;
        x = y;
        y = temp;
        System.out.println("x = " + x);
        System.out.println("y = " + y);
    }
}
```

该程序的输出是：

```
x = 1
y = 2
x = 2
y = 1
numOne = 1
numTwo = 2
```

注意，虽然我们成功地交换了形参x和y的值，但这样做对实参numOne和numTwo没有什么影响。在方法swap()中形参实际上是局部变量，用相应的实参的值初始化形参。在本程序中标识了五个内存位置，用于存储整数numOne、numTwo、x、y和temp。

在Java中，我们不能用类似于已经提到的那种方法——实际上交换的是作为参数传递的两个基本类型变量的值——来编写方法swap()。如果swap()实际上已经引起了main()中的值的变化，那么我们要说它达到了它的结果，但是用副作用代替了返回一个值，该副作用对两个实参作了改变。使用返回类型为void的任何方法一定会引起一些副作用，否则它完全是没用的。编写一个C程序，它要以与这个Java程序一样的方式失败。

## 第9章 数组和指针

数组是具有相同类型的数据项的序列，可以对它进行标引和连续地存储。通常数组是一种用于表示大量同类值的数据类型。通过下标访问数组的元素。数组可以是各种类型的，包括数组的数组。串就是字符数组，但有必要对它分别处理，下一章讨论串。

典型的数组声明要从一个基地址开始分配内存空间。数组名实际上是一个指向基地址的指针常量，本章要详细地解释数组和地址间的关系。本章要讨论的另一个关键点是怎样把数组作为参数传递给函数。我们要用一些细致的例子来说明这些概念。

### 9.1 一维数组

程序经常使用同类的数据，例如，如果我们要处理一些成绩，我们可以声明

```
int grade0, grade1, grade2;
```

然而，如果成绩量很大，就要使用大量的标识符进行表示，且标识符还都要惟一，进而进行处理，这样做是很麻烦的。我们应该使用一种派生类型，即数组。用下标访问数组的各个元素，也把下标称为标引(index)，用方括号括住数组的下标。为了在程序中使用`grade[0]`、`grade[1]`和`grade[2]`，我们声明

```
int grade[3];
```

声明中的整数3表示数组的尺寸即数组中元素的个数。对数组元素的下标总是从0开始。这是C语言的典型特征。

一维数组声明是一个类型后跟一个带有用方括号括起来的常量整数表达式的标识符。常量表达式指定了数组的尺寸，但它的值必须是正的；它指定了数组中元素的个数。为了存储数组的元素，编译器会分配从一个基地址开始的适当大小的内存。

为了说明这样的一些思想，让我们编写一个小程序，它填充数组、显示数组的值并对数组的元素进行累加。

```
/*Fill and Print an array. */
#include <stdio.h>

#define N 5

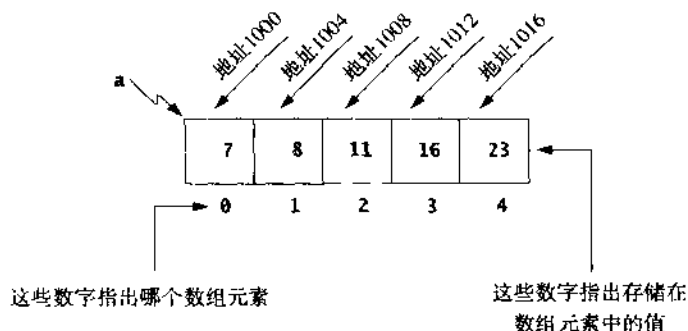
int main(void)
{
    int a[N]; /* allocate space for a[0] to a[4] */
    int i, sum = 0;

    for (i = 0; i < N; ++i) /* fill the array */
        a[i] = 7 + i * i;
    for (i = 0; i < N; ++i) /* print the array */
        printf("a[%d] = %d ", i, a[i]);
    for (i = 0; i < N; ++i) /* sum the elements */
        sum += a[i];
    printf("\nsum = %d\n", sum); /* print the sum */
    return 0;
}
```

该程序的输出是

```
a[0] = 7    a[1] = 8    a[2] = 11   a[3] = 16    a[4] = 23
sum = 65
```

数组a需要存储5个整数值的内存空间。假设我们的机器用4个字节存储一个int型的值。如果a[0]存储在地址1000,那么其余的数组元素连续地存储在地址1004、1008、1012和1016。



### 对程序Fill\_and\_Print的解析

```
• #define N 5

int main(void)
{
    int a[N]; /* allocate space for a[0] to a[4] */
    int i, sum = 0;
```

把数组的尺寸定义为符号常量是一种良好的编程作法。因为很多代码要依赖这个值,要改变数组的大小,可在#define中很方便地改变该值。注意,a[N]怎样为5个int型元素分配内存。通常把变量i用作标引数组的int型变量。

```
• for (i = 0; i < N; ++i) /* fill the array */
    a[i] = 7 + i * i;
```

这是一种处理全部数组元素的关键性惯用法。下标变量从0开始,一直到N-1。注意,如果用i <= N,就错了,这样会导致访问a[5],而a[5]不是该数组的元素。

```
• for (i = 0; i < N; ++i) /* print the array */
    printf("a[%d] = %d ", i, a[i]);
for (i = 0; i < N; ++i) /* sum the elements */
    sum += a[i];
```

这里我们再次用for循环的这种惯用法处理各个元素。 ■

#### 9.1.1 初始化

数组的存储类型可以是自动、外部、静态或常量,但不能是寄存器。像简单变量一样,可以在声明中对数组初始化。数组的初始化部分(array initializer)是用花括号括起来的用逗号分隔开的一系列初始值,下面是一个例子:

```
float x[7] = {-1.1, 0.2, 33.0, 4.4, 5.05, 0.0, 7.7};
```

该语句把x[0]初始化为-1.1, x[1]初始化为0.2,以此类推。若初始化表的长度短于要被初始化的数组元素的数目,那么就把剩余的元素初始化为0。如果外部或静态数组没被初始化,

那么系统就自动地把所有元素初始化为0。未被初始化的自动或常量数组含有的值是无用的，即用随意的值填写为数组分配的内存。在传统C中，数组的初始化部分仅能对外部和静态数组初始化，ANSI C也允许对自动和常量数组初始化。

如果声明的数组没有尺寸，并且用了一系列的值对它进行初始化，那么就暗示着数组的大小为初始化部分中的值的数目。这样，声明

```
int a[] = {3, 4, 5, 6};
```

与

```
int a[4] = {3, 4, 5, 6};
```

是等价的。

### 9.1.2 下标

假设已经进行了如下声明：

```
int i, a[size];
```

我们可以用`a[i]`来访问数组的元素。更一般地，我们可以用`a[expr]`来访问数组的元素，此处`expr`是一个整数表达式。我们把`expr`称为`a`的下标(subscript)或标引(index)。下标的值必须位于范围0到`size - 1`之间。若数组的下标值超出了这个范围，就会引起运行错误。把这种常见的编程错误称为超出了数组边界即下标越界。这种错误是与系统相关的，能引起混乱。经常产生的一个后果是返回或修改一些不相关的变量的值。这样，程序员必须要确保所有的下标不越界。



“哎呀，菲尼小姐，如果我在每页都犯有同样类型的拼写错误，那么会使其成为合法的拼写错误数组吗？”

## 9.2 例子：分别对每个字母计数

在上一章中，我们说明了如何对数和字母计数等。通过使用数组，我们能较容易地计算出大写字母出现的次数，下面的程序能完成此项工作：

```

/* Count each uppercase letter separately. */
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    int c, i, letter[26];

    for (i = 0; i < 26; ++i) /* init array to zero*/
        letter[i] = 0;
    while ((c = getchar()) != EOF) /* count the letters */
        if (isupper(c))
            ++letter[c - 'A'];
    for (i = 0; i < 26; ++i) { /* print the results */
        if (i % 6 == 0)
            printf("\n");
        printf("%4c:%3d", 'A' + i, letter[i]);
    }
    printf("\n\n");
    return 0;
}

```

把本章的英文版内容作为输入。如果我们把这个程序编译成cnt\_abc, 那么发出下面的命令:

```
cnt_abc < chapter9
```

在屏幕上就会出现以下信息:

```

A: 75  B: 52  C:219  D: 14  E:121  F: 13
G:  9  H: 13  I:121  J:  1  K:  1  L: 39
M: 25  N: 44  O: 38  P:243  Q:  1  R: 37
S: 73  T: 96  U:  7  V:  3  W: 17  X:  9
Y: 11  Z: 27

```

#### 对程序cnt\_abc的解析

```
* int c, i, letter[26];
```

用数组letter存储对26个大写字母的计数。要记住, 数组的元素是letter[0], letter[1], ..., letter[25], 如果你忘记了数组的下标是从0开始的, 你就可能会犯很多错误。把变量i用作下标。

```

* for (i = 0; i < 26; ++i) /* init array to zero*/
    letter[i] = 0;

```

必须要显式地对自动数组进行初始化。处理一个数组的所有元素都采用这样的for循环。这是用C编程的一个版式。把下标变量初始化为0。终止测试是看是否超出了上界。

```

* while ((c = getchar()) != EOF) /* count the letters */
    if (isupper(c))
        ++letter[c - 'A'];

```

用库函数getchar()重复地从输入流中读入字符, 并赋值给c。当检测到文件尾标记时, 退出while循环。用<ctype.h>中的宏isupper()测试c是否为大写字母。如果是, 那么数组letter的相应的元素就加1。

```
* ++letter[c - 'A'];
```

这行代码是与系统相关的。在ASCII机器上, 如果c的值是'A', 那么表达式c - 'A'的值是0; 如果c的值是'B', 那么表达式c - 'A'的值是1; 以此类推。这样, c的大写字母值就映射在值域0到25上。由于方括号的优先级高于++, 下面的表达式与该表达式等价:

```
++(letter[c - 'A']);
```



这样我们看到，如果c的值是'A'，那么就对letter[0]加1；如果c的值是'B'，那么就对letter[1]加1；以此类推。

```
* for (i = 0; i < 26; ++i) {          /* print the results */
    if (i % 6 == 0)
        printf("\n");
    printf("%4c:%3d", 'A' + i, letter[i]);
}
```

用同样的for循环版式处理数组array。每循环6次，就显示一个换行。随着i从0变到25，显示出的表达式'A' + i的值就是A到Z，每个值后跟一个冒号及相应的计数。 ■

### 9.3 数组和指针间的关系

数组名本身是一个地址即指针值，在访问内存方面，指针和数组几乎是相同的。然而，也有不同，这些不同是微妙且重要的。指针是以地址作为值的变量，数组名是一个特殊的固定地址，可以把它看作是常量指针。在声明数组时，编译器必须分配基地址和足够的存储空间，以存储数组的所有元素。数组的基地址是在内存中存储数组的起始位置；它是数组的第一个元素（标引为0）的地址。假设我们给出如下声明：

```
#define N 100
int a[N], *p;
```

系统分别把编号为300, 304, 308, ..., 696的内存字节作为a[0], a[1], a[2], ..., a[99]的地址，其中位置300是a的基地址。语句

```
p = a;
```

与

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

是等价的，都是把300赋给p。指针运算为对数组标引提供了一个选择。语句

```
p = a + 1;
```

与

```
p = &a[1];
```

是等价的，都是把304赋给了p。假设已经对a的元素进行了赋值，我们能如下语句对数组求和：

```
sum = 0;
for (p = a; p < &a[N]; ++p)
    sum += *p;
```

在这个循环中，用数组a的基地址对指针变量p初始化。p的连续值是&a[0], &a[1], ..., &a[N-1]。一般而言，如果i是类型为int的变量，那么p+i就是距地址p的第i个偏移。以类似的方式，a+i是距数组a的基地址的第i个偏移。下面是对数组求和的另一种方法：

```
sum = 0;
for (i = 0; i < N; ++i)
    sum += *(a + i);
```

正像表达式\*(a + i)与a[i]等价一样，表达式\*(p + i)与p[i]等价。下面是对数组求和的第三种方法：

```
p = a;
```

```
sum = 0;
for (i = 0; i < N; ++i)
    sum += p[i];
```

虽然在很多方面都能同样处理数组和指针，但有一个本质的不同。因为数组a是常量指针，不是变量，所以像

```
a = p      ++a      a += 2
```

这样的表达式是非法的。我们不能改变a的地址。

## 9.4 指针运算和元素尺寸

指针运算是C的重要特征之一。如果变量p是指向一种特殊类型的指针，那么表达式p + 1会产生存储或访问该类型的下一个变量的正确机器地址。以类似的方式，像p+i、++p和p+=i这样的表达式都是有意义的。如果p和q是指向数组元素的指针，那么p - q产生一个int型的值，该值表示在p和q间的数组元素的个数。即使指针表达式和算术表达式看起来相似，对两种类型表达式的解释也有着本质的不同，下面的代码说明了这种不同。

```
#include <stdio.h>

int main(void)
{
    double  a[2], *p, *q;

    p = &a[0];           /* points at base of array */
    q = p + 1;           /* equivalent to q = &a[1]; */

    printf("%d\n", q - p); /* 1 is printed */
    printf("%d\n", (int) q - (int) p); /* 8 is printed */
    return 0;
}
```

最后一个语句所显示的内容是与系统相关的。很多系统以8个字节存储一个double型值，因此，在被解释为整数的两个机器地址间的差是8。

## 9.5 把数组传递给函数

在函数定义中，被声明为数组的形参实际上是一个指针。当传递数组时，按值调用传递它的基地址，数组元素本身不被复制。作为一种表示习惯，编译器允许在作为参数声明的指针中使用数组方括号。为了说明这点，我们编写一个对类型为int的数组的元素进行累加的程序。

```
int sum(int a[], int n) /* n is the size of the array */
{
    int  i, s = 0;

    for (i = 0; i < n; ++i)
        s += a[i];
    return s;
}
```

作为函数定义头的一部分，声明

```
int a[];
```

等价于

```
int *a;
```

但另一方面，作为函数体内的声明，它们是不等价的。第一个声明创建一个常量指针，而第

二个创建一个指针变量。

假设把`v`声明为具有100个类型为`int`的元素的数组。在对元素赋值后，我们能用上一个函数`sum()`对`v`的各元素作加法。下表说明了一些可能性。

调用 <code>sum()</code> 的各种方法	
调用	被计算和被返回的内容
<code>sum(v, 100)</code>	<code>v[0] + v[1] + ..... + v[99]</code>
<code>sum(v, 88)</code>	<code>v[0] + v[1] + ..... + v[87]</code>
<code>sum(&amp;v[7], k - 7)</code>	<code>v[7] + v[8] + ..... + v[k - 1]</code>
<code>sum(v + 7, 2 * k)</code>	<code>v[7] + v[8] + ..... + v[2 * k + 6]</code>

最后一个函数调用再次说明了对指针运算的用法。把`v`的地址偏移7，`sum()`把局部指针变量`a`初始化为这个地址，这使得在函数调用中的所有地址计算结果都被类似地偏移。

## 9.6 排序算法：冒泡排序

对于搜索大型数据库来说，对信息进行排序的算法是至关重要的。想象一下词典或电话号码本，用它们来查找信息都是相对容易和方便的，这是因为其中的信息都按字母表或词典顺序排了序。排序是一种非常有助于解决问题的技术，此外，如何有效地排序的问题本身是一个重要的研究领域。

要对有 $n$ 个元素的数组排序，有效的排序算法通常需要进行 $n\log n$ 次比较。冒泡排序效率不高，这是因为它需要 $n^2$ 次比较；然而对一些小程序来说，它的性能通常还是可以接受的。在我们给出`bubble()`的代码后，我们将详细说明该函数如何处理一个特殊的整数数组。我们使用8.4节“引用调用”中的函数`swap()`。

```
void swap(int *, int *);

void bubble(int a[], int n)    /* n is the size of a[] */
{
    int i, j;
    for (i = 0; i < n - 1; ++i)
        for (j = n - 1; i < j; --j)
            if (a[j-1] > a[j])
                swap(&a[j-1], &a[j]);
}
```

假设我们声明

```
int a[] = {7, 3, 66, 3, -5, 22, -77, 2};
```

然后调用`bubble(a, 8)`。下表显示出了在每次外部循环后数组`a[]`中的元素。

在第一次循环的开始处，把`a[6]`与`a[7]`比较，由于它们符合次序要求，对它们没做交换；然后把`a[5]`与`a[6]`比较，由于它们不符合次序要求，对它们要做交换。再把`a[4]`与`a[5]`比较，以此类推。若邻接元素不符合次序要求，则要对它们进行交换。第一次循环的效果是把数组中的最小元素“冒泡”到`a[0]`。在第二次循环中，不再检查`a[0]`，即不再改变它，把`a[6]`再与`a[7]`比较，如此等等。在第二次循环后，第二小的数存放在`a[1]`中。由于每次循环都把当前最小的元素放在数组的合适位置中，在 $n-1$ 次循环后，算法就把所有的元素排了序。注意，在本例中循环5次就把所有的元素排了序。也可以修改该算法，通过增加

一个变量进行检测，若在一次循环中元素的次序没有发生变化，则终止循环（请参见练习7）。

未排序的数据	7	3	6	3	-	2	-	2
			6		5	2	77	
第一遍	-	7	3	6	3	-	22	2
	77			6		5		
第二遍	-	-	7	3	6	3	2	2
	77	5			6			2
第三遍	-	-	2	7	3	6	3	2
	77	5				6		2
第四遍	-	-	2	3	7	3	66	2
	77	5						2
第五遍	-	-	2	3	3	7	22	6
	77	5						6
第六遍	-	-	2	3	3	7	22	6
	77	5						6
和七遍	-	-	2	3	3	7	22	6
	77	5						6

9.7 二维数组

C语言允许任何类型的数组，甚至允许数组的数组。使用两对方括号，我们就能得到二维数组。要得到高维数组，只要简单地继续增加方括号即可。每使用一对方括号，我们就对数组增加了一维。

数 组 维	
数组的声明	注 释
int a[100];	一维数组
int b[2][7];	二维数组
int c[5][3][2];	三维数组

k维数组的尺寸与各个维的尺寸有关。如果用 $s_i$ 代表数组的第*i*维尺寸，那么数组声明为 $s_1 \times s_2 \times \dots \times s_k$ 个元素分配的空间。在上表中，b有 $2 \times 7$ 个元素，c有 $5 \times 3 \times 2$ 个元素。从数组的基地址开始，所有的数组元素都存储在连续的内存中。

即使把数组元素一个接一个地连续存储，经常把二维数组看作是由行和列组成的矩阵更为方便。例如，如果我们声明

```
int a[3][5]
```

那么我们数组元素的排列如下：

	第1列	第2列	第3列	第4列	第5列
第1行	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
第2行	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
第3行	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

为了说明这些思想，让我们编写一个填充二维数组的程序，它要显示数组的值，并要对

数组的元素进行累加。

```
#include <stdio.h>

#define M 3      /* number of rows */
#define N 4      /* number of columns */

int main(void)
{
    int a[M][N], i, j, sum = 0;

    for (i = 0; i < M; ++i)      /* fill the array */
        for (j = 0; j < N; ++j)
            a[i][j] = i + j;
    for (i = 0; i < M; ++i) {      /* print array values */
        for (j = 0; j < N; ++j)
            printf("a[%d][%d] = %d", i, j, a[i][j]);
        printf("\n");
    }
    for (i = 0; i < M; ++i)      /* sum the array */
        for (j = 0; j < N; ++j)
            sum += a[i][j];
    printf("\nsum = %d\n", sum);
    return 0;
}
```

程序的输出为:

```
a[0][0] = 0   a[0][1] = 1   a[0][2] = 2   a[0][3] = 3
a[1][0] = 1   a[1][1] = 2   a[1][2] = 3   a[1][3] = 4
a[2][0] = 2   a[2][1] = 3   a[2][2] = 4   a[2][3] = 5

sum = 30
```

在处理多维数组中的各个元素时,每维都需要一个单for循环。

由于数组和指针间的关系,存在有很多访问二维数组元素的方法。

等价于a[i][j]的表达式
*(a[i] + j)
(* (a + i))[j]
*((*(a + i)) + j)
*(&a[0][0] + 5*i + j)

由于方括号[]比间接运算符\*有较高的优先次序,圆括号是必须的。我们可以把a[i]看作是(从0开始计)的第i行,把a[i][j]看作是a(从0开始计)的第i行第j列。数组名a本身等价于&a[0];它是指向有5个int型值的数组的指针。该数组的基地址是&a[0][0],而不是a。从数组的基地址开始,编译器为15个int型的值分配连续的空间。对于任何数组而言,把指针值和数组标引间的映射称为存储映射函数。用等价于\*(&a[0][0]+5\*i+j)的a[i][j]描述数组a的存储映射函数。若多维数组是函数定义的一个形参,则必须要描述所有的尺寸(第一个除外),以使得编译器能正确地计算出存储映射函数。在对给定的数组a的元素赋值后,下面的函数能用于对数组a的元素求值,当心不要忘记描述列的尺寸。

```
int sum(int a[][5])
{
    int i, j, sum = 0;

    for (i = 0; i < 3; ++i)
        for (j = 0; j < 5; ++j)
            sum += a[i][j];
    return sum;
}
```

在函数定义的头部有参数声明`int a[][5]`，它与如下两个声明是等价的：

```
int a[][5]    int (*a)[5]    int a[3][5]
```

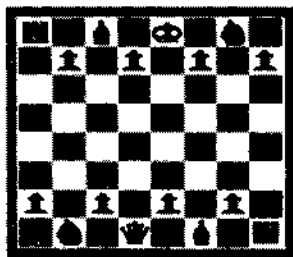
由于运算符有优先次序，上边的圆括号是必须的。常量3对代码的读者来说有提醒作用，但编译器忽略它。

有一些对二维数组进行初始化的方法，下面的三个初始化是等价的。

```
int a[2][3] = {1, 2, 3, 4, 5, 6};
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
int a[ ][3] = {{1, 2, 3}, {4, 5, 6}};
```

如果没有内部花括号，那么就依次地对数组元素`a[0][0]`，`a[0][1]`， $\dots$ ，`a[1][2]`进行初始化。注意对行的标引。如果初始值的个数少于数组元素的个数，那么就用0对剩余的元素初始化。如果第一对方括号为空，那么编译器就用内部圆括号中值的个数作为尺寸。除了第一个尺寸外，其余的必须要明确地给出。二维数组的一个典型的用法是作为 $n \times n$ 的矩阵。

可以很容易地把棋盘表示为二维数组，用typedef对各棋子命名



这种数据结构是一种重要的对数据类型的科学抽象，是所有线性代数的核心。我们在12.7节“typedef的用法”中说明了怎样对该类型使用typedef，并说明了如何完成两个矩阵的基本乘法运算。

## 9.8 多维数组

高于二维的数组的工作方式是类似的，让我们描述一下三维数组是如何工作的。如果我们声明

```
int a[7][9][2];
```

编译器分配了 $7 \times 9 \times 2$ 个int型的连续存储空间。数组的基地址是`&a[0][0][0]`，用等价于`*(&a[0][0][0] + 9*2*i + 2*j + k)`的`a[i][j][k]`描述存储映射函数。

如果在程序中使用像`a[i][j][k]`这样的表达式，那么编译器用存储映射函数产生正确的访问内存中的数组元素的代码。虽然通常不必这样做，但程序员可以直接使用存储映射函数。下面是一个对数组a的元素进行累加的函数：

```
int sum(int a[][9][2])
{
    int i, j, k, sum = 0;
    for (i = 0; i < 7; ++i)
        for (j = 0; j < 9; ++j)
            for (k = 0; k < 2; ++k)
                sum += a[i][j][k];
    return sum;
}
```

在函数定义的头中的声明`int a[][9][2]`与如下声明是等价的：

```
int a[][9][2]      int a[7][9][2]      int (*a)[9][2]
```

常量7 对代码的阅读者来说有提醒作用,但对编译器来说它是无用的。编译器需要另两个常量,用它们产生正确的存储映射函数。

有很多对多维数组进行初始化的方法。考虑如下的初始化:

```
int    a[2][2][3] = {
    {{1, 1, 0}, {2, 0, 0}},
    {{3, 0, 0}, {4, 4, 0}}
};
```

它等价于

```
int    a[][2][3] = {{{1, 1}, {2}}, {{3}, {4, 4}}};
```

如果初始化部分是完整的且都被一致地括着,那么凡是出现了初始值不足的地方,就把剩余元素用0初始化。

一般而言,如果没有对存储类型为自动的数组显式地进行初始化,那么数组元素的初始值都是无用的,但对静态和外部数组用缺省的0进行初始化。下面是一个把数组的所有元素都初始化为0的简单方法:

```
int    a[2][2][3] = {0};
```

## 9.9 动态内存分配

C在标准库中提供了用于动态内存分配的`calloc()`(连续分配)和`malloc()`(内存分配),它们的函数原型是在`stdlib.h`中。与其在程序中用特定常量指定数组的尺寸,不如让用户输入数组的尺寸或通过计算获得数组的尺寸。下面是一种函数调用的形式:

```
calloc( n, object_size)
```

它返回一个指向足以存储 $n$ 个对象的内存空间的指针,每个对象需要 $object\_size$ 个字节。 $n$ 和 $object\_size$ 都是正的。如果系统不能分配出所需要的空间,返回的指针值为`NULL`。

在ANSI C中,在`stdlib.h`中用`typedef`给定了类型`size_t`。通常该类型是无符号的,但它随着系统的不同而不同。在`calloc()`和`malloc()`的函数原型中要使用这个类型定义:

```
void    *calloc(size_t, size_t);
void    *malloc(size_t);
```

因为返回的指针的类型是`void*`,所以不用类型转换就可把它赋值给其他指针。由`calloc()`分配的存储空间被自动地初始化为0,而由`malloc()`分配的存储空间没有被初始化,因而其开始的值是无用的。为了说明`calloc()`的用法,让我们编写一个小程序,它交互式地提示用户输入一个数组的尺寸。

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

int main(void)
{
    int    *a, i, n, sum = 0;
    printf("\n%s",
        "An array will be created dynamically.\n\n");
    printf("Input an array size n followed by n integers: ");
    scanf("%d", &n);
    a = calloc(n, sizeof(int)); /* get space for n ints */
    for (i = 0; i < n; ++i)
        scanf("%d", &a[i]);
```

```

    for (i = 0; i < n; ++i)
        sum += a[i];
    free(a); /* free the space */
    printf("\n%s%7d\n%s%7d\n\n",
        "Number of elements:", n,
        "Sum of the elements:", sum);
    return 0;
}

```

注意，我们调用`free(a)`释放由`calloc()`分配的空间，从而把释放的空间还给系统。在这个小程序中这是不必要的，因为当程序退出时会释放空间。在`stdio.h`中给出了`free()`的函数原型：

```
void free(void *ptr);
```

在程序运行期间，由`calloc()`和`malloc()`分配的空间一直是可用的，直到程序员显式地释放它们为止。在函数退出时，是不会释放空间的。

在上一个程序中，我们用`calloc()`为数组`a`动态地分配空间。这里我们写出语句：

```
a = calloc(n, sizeof(int)); /* get space for n ints */
```

不用`calloc()`，我们要用`malloc()`。为此我们写出语句：

```
a = malloc(n * sizeof(int)); /* get space for n ints */
```

两个语句的惟一不同是`malloc()`没有把所分配的空间初始化为0。由于我们的程序不需要初始化，我们无妨就使用`malloc()`。然而，对于数组而言，我们还是愿意使用`calloc()`。

## 9.10 风格

像本章的例子所表明的那样，人们经常用符号常量定义数组的尺寸。如果需要用代码处理不同尺寸的数组，程序员利用常量仅修改一处就能达到目的。

下面是一个重复做10次的for循环：

```
for (i = 1; i <= 10; ++i)
    .....
```

我们也可以把它写成如下形式：

```
for (i = 0; i < 10; ++i)
    .....
```

具体使用哪种形式仅依赖循环体要做什么。然而，在一些情况两种形式都可以用。在处理数组时，C程序员一般偏好使用第二种形式，这是因为第二种形式符合正确的编程习惯。由于在编程中要普遍地使用数组，大多数有经验的C程序员从0而不是从1开始计算数组。

一种常用的重要风格是对程序进行结构化，用一个函数来完成一个基本任务，这种作法是结构化编程的核心；然而，当处理数组时这也会导致代码低效，让我们看一个具体的例子。

```

/* Compute various statistics. */
#include <stdio.h>

#define N 10 /* size of the array */

double average(double *, int);
double maximum(double *, int);
double sum(double *, int);

int main(void)

```



```

{
    int    i;
    double a[N];

    printf("Input %d numbers: ", N);
    for (i = 0; i < N; ++i)
        scanf("%lf", &a[i]);
    printf("\n%s%5d\n%s%7.1f\n%s%7.1f\n%s%7.1f\n\n",
        "Array size:", N,
        "Maximum element:", maximum(a, N),
        "Average:", average(a, N),
        "Sum:", sum(a, N));

    return 0;
}

```

为了调用三个函数，我们编写了main()，用这三个函数计算所要求的值。把这三个函数编写如下：

```

double maximum(double a[], int n)
{
    int    i;
    double max = a[0];

    for (i = 0; i < n; ++i)
        if (max < a[i])
            max = a[i];
    return max;
}

double average(double a[], int n)
{
    return (sum(a, n) / (double) n);
}

double sum(double a[], int n)
{
    int    i;
    double s = 0.0;

    for (i = 0; i < n; ++i)
        s += a[i];
    return s;
}

```

其中的两个函数用for循环处理数组元素，average()调用sum()来完成它的工作。由于效率上的缘故，我们要重新编写程序，首先重新编写main()。

```

/* Compute various statistics more efficiently. */
#include <stdio.h>

#define N 10

void stats(double *, int, double *, double *, double *);

int main(void)
{
    int    i;
    double a[N], average, max, sum;

    printf("Input %d numbers: ", N);
    for (i = 0; i < N; ++i)
        scanf("%lf", &a[i]);
    stats(a, N, &average, &max, &sum);
    printf("\n%s%5d\n%s%7.1f\n%s%7.1f\n%s%7.1f\n\n",
        "Array size:", N,
        "Maximum element:", max,
        "Average:", average,
        "Sum:", sum);

    return 0;
}

```

现在我们编写函数stats(), 用单for循环计算所有的所要求的值。

```
void stats(double a[], int n,
           double *p_average,
           double *p_max,
           double *p_sum)
{
    int i;

    *p_max = *p_sum = a[0];
    for (i = 1; i < n; ++i) {
        *p_sum += a[i];
        if (*p_max < a[i])
            *p_max = a[i];
    }
    *p_average = *p_sum / (double) n;
}
```

第二个版本避免了额外的重复函数调用。在第一个版本中, 进行 $3 \times N$ 次函数调用, 而在第二个版本中仅调用了stats()一次。本例是太小了, 以至于没体现出效率, 但当把代码用于各种工作环境时, 我们已经给出的思想就时常地会体现出来。一般而言, 像程序的清晰性和正确性一样, 有效性也是编程时要考虑的一个重要因素。

### 9.11 常见的编程错误

使用数组所犯的大多数常见编程错误都是下标越界。假设10是整数数组a[]的尺寸。如果我们写

```
sum = 0;
for (i = 1; i <= 10; ++i)
    sum += a[i];
```

我们就会犯错误。这样的语句要访问a[10]中值, 但是在内存中那个地址处的值是无法预见的。

在很多编程语言中, 当声明尺寸为n的数组时, 就确定了相应的下标范围是从0到n-1。C用0作为数组下标的下界, 用n-1作为下标的上界, 记住这点是很重要的。边界检查是一种需要培养的重要的编程技巧。在处理很大的数组前, 手工模拟处理小数组的程序经常是很有帮助的。

当程序员使用动态内存分配时, 一种常见的编程错误是忘记释放内存。在小程序中, 这样做通常不存在问题。然而, 如果在循环中重复地分配内存、使用内存而忘记了释放它, 程序会出乎意料地失败, 这是因为已没有更多的内存可供使用。

### 9.12 系统考虑

如果效率是一个问题, 那么我们最好用malloc(n \* sizeof(something))代替calloc(n, sizeof(something)), 除非要把所分配的空间初始化为0。对于大多数程序, 在执行时间上的不同是不值得注意的。

内存空间是一种有限的资源。当调用calloc()或malloc()动态地分配空间时, 程序员应该检查调用是否成功, 下面的代码表明了怎样用calloc()做到这一点:

```
int *a, n;

..... /* get n from somewhere */
if ((a = calloc(n, sizeof(int))) == NULL) {
```

```
    printf("\nERROR: calloc() failed - bye!\n\n");
    exit(1);
}
```

如果我们正在编写一些代码，并需要重复地调用`calloc()`和`malloc()`，那么我们就需要得体地编写这些函数，下面就是这样的函数：

```
/* graceful function */
void *gcalloc(size_t n, size_t sizeof_something)
{
    void *p;

    if ((p = calloc(n, sizeof_something)) == NULL) {
        printf("\nERROR: calloc() failed - bye!\n\n");
        exit(1);
    }
    return p;
}
```

在各种编码工作中，使函数优雅是很必要的。在13.7节“使用临时文件和得体的函数”中，我们会再次遇到这样的思想。

### 9.13 转向C++

在C++中，用`new`和`delete`代替`calloc()`和`malloc()`。它们是该语言中的运算符，在类型上是安全的，使用起来也很方便。一元运算符`new`和`delete`可用于操纵自由存储(free store)。自由存储是系统为程序员直接管理其生存期的对象提供的内存工具。程序员用`new`创建对象，用`delete`撤消对象。对于链表和树这样的数据结构来说，它们是很重要的。

在C++中，以如下的形式使用运算符`new`：

```
new type-name
new type-name initializer
new (type-name)
```

在每一种情况下，至少有两种作用。第一种，从自由存储中分配出适当数量的内存，以容纳指定的类型，第二种，把对象的基地址作为`new`表达式的值返回。该表达式的类型是`void*`，能被赋值给任何类型的指针变量。当内存不够时，运算符`new`返回值0。

下面是使用`new`的例子：

```
int* ptr_i;
ptr_i = new int(5); //allocation and initialization
```

在上述的代码中，把所分配的类型为`int`的对象的存储地址赋值给指向类型为`int`的指针变量`ptr_i`。用值5初始化`ptr_i`所指向的位置。对于像`int`这样的简单类型，这种用法是少见的，自动地分配栈中的或全局的整型变量是更常规和自然的。

运算符`delete`撤消由`new`创建的对象，实际上`delete`返回所分配的存储空间给自由存储，以便重新使用。以下述形式使用运算符`delete`：

```
delete 表达式
delete [ ] 表达式
```

在相应的`new`表达式分配的不是数组时，就用第一种形式。第二种形式使用了一对方括号，用以指明所分配的是对象数组。运算符`delete`不返回值，可以说它的返回类型是`void`。

下面的例子使用了这些动态地分配数组的结构：

```
//Use of new to dynamically allocate an array.
#include <iostream.h>
```

```

int main(void)
{
    int* data;
    int size;

    cout << "\nEnter array size: ";
    cin >> size;
    data = new int[size];    //allocate an array of ints

    for (int j = 0; j < size; ++j)
        cout << (data[j] = j) << '\t';

    cout << "\n\n";
    delete[] data;          //deallocate an array
    data = new int[size];
    for (j = 0; j < size; ++j)
        cout << data[j] << '\t';
    return 0;
}

```

### 对程序dyn\_array的解析

```

• int* data;
  int size;

  cout << "\nEnter array size: ";
  cin >> size;
  data = new int[size];    //allocate an array of ints

```

指针变量data用于存放动态分配的数组的基地址，该数组的元素个数是size的值。程序提示用户输入整数给size。new运算符用于从自由存储中分配存储空间，以存储类型为int[size]的对象。在需要用两个字节存储一个整数的系统中，这将分配2 × size个字节。这里，把该存储的基地址赋给data。

```

• for (int j = 0; j < size; ++j)
    cout << (data[j] = j) << '\t';

```

这个语句对data数组的值进行初始化，并进行显示。

```

• delete[] data;          //deallocate an array

```

运算符delete把与指针变量data相关的存储空间归还给自由存储，该存储空间只能是由new分配的。使用方括号是因为相应的分配是一个数组。

```

• data = new int[size];
  for (j = 0; j < size; ++j)
    cout << data[j] << '\t';

```

我们再次访问自由存储，但这次不是对data数组进行初始化。在一般的系统中，这会又使用刚返回到自由存储的内存，原有的值重新出现。但是，不保证从自由存储分配的对象中的值重现。在你的系统上测试一下这种情况。程序员应负责对这样的对象进行正确地初始化。 ■

### 小结

- 数组是一系列具有相同类型的数据项，可以对数组进行标引，并连续地存储它。可以用数组处理大量的同类值。像

```
int a[100];
```

这样的声明生成了一个int型的数组。编译器在内存中分配了100个连续的int型的单元，并用0到99对这些元素编号。

- 用像 `a[i]` 这样的表达式访问数组元素。更一般地，我们可以用 `a[expr]`，`expr` 是下标即标引，它是一个具有非负值的表达式，该表达式的值不能超过 `a` 的下标上界。程序员有责任确保数组的下标不越界。
- 当把数组名作为参数传递给函数时，实际上传递的是数组基地址的拷贝。在函数定义的头文件中，声明

```
int a[];
```

等价于

```
int *a;
```

在函数定义的头中，对多维数组的声明必须给出所有的尺寸，只是第一个除外（请参见练习9）。

- 可以通过在花括号中放一个适当长度的值的列表来对数组初始化。如果声明的是外部或静态数组但未对其初始化，那么数组的所有元素都被自动地初始化为0。不对自动数组进行初始化，它的开始值是无用的。
- 可以创建任何类型的数组，甚至可以创建数组的数组。例如，

```
double a[30][50];
```

声明 `a` 是一个“有50个 `double` 型单元的数组”的数组。用像 `a[i][j]` 这样的表达式访问 `a` 的元素。

## 练习

1. 请解释下列术语：

- 下界
- 下标
- 越界

2. 下面的数组声明中有几处错误，请找出它们。

```
#define SIZE 4

int main(void)
{
    int a[SIZE] = {0, 2, 2, 3, 4};
    int b[SIZE - 5];
    int c[3.0];
}
```

3. 编写一个函数，分别对 `double` 型数组的下标为偶数的元素和下标为奇数的元素求和。依据元素下标是奇数还是偶数，把各元素计算到相应的和中。你的函数定义看起来应如下：

```
void sum(double a[],
        int n, /* n is the size of a[] */
        double *even_index_sum_ptr,
        double *odd_index_sum_ptr)
{
    .....
```

4. 编写一个函数，计算一个整数数组的元素的两个和。依据元素本身是奇数还是偶数，把各元素计算到相应的和中。你的函数定义看起来应如下：

```
void sum(int a[],
        int n,
        int *even_element_sum_ptr,
```

```

        int *odd_element_sum_ptr)
{
    .....

```

5. 修改9.2节“例子：分别对每个字母计数”中的程序cnt\_abc，使它也能分别对每个小写字母计数。

6. 本练习要测试你对指针运算的理解。假设SIZE是值为100的符号常量。如果一个声明为：

```

char    a[SIZE], *p = a;
int     i;

```

那么编译器会在内存中分配100个字节的连续存储空间，数组名a指向该存储空间的基地址。我们故意使用了char型数组，这是因为每个char单元用1个字节存储。把指针p初始化后，它的值就同a一样。现在我们要以一种非常简单形式填充这个数组。

```

for (i = 0; i < SIZE; ++i)
    a[i] = i;

```

该语句用从0到99的连续整数值对这个数组赋了值。现在考虑

```

printf("%d\n", *(p + 3));
printf("%d\n", *(char *)((int *) p + 3));
printf("%d\n", *(char *)((double *) p + 3));

```

这段代码会显示什么？对这个问题的答案是与系统相关的。请解释为什么是这样，并解释在另一台不同的机器会显示什么。提示：考虑表达式

```
(int *) p + 3
```

其中有两个运算符在起作用，哪一个优先次序较高？用此信息决定该指针表达式正指向哪一个数组元素。现在考虑

```
(char *) pointer_expression
```

它把指针表达式类型转换为指向char的指针。现在考虑

```
*(char *) pointer_expression
```

这两个一元运算符如何相关联？

7. 编写一个程序，用bubble()函数对一个整数数组排序。修改bubble()，使得在首次没发生交换的循环后及时退出。这两个程序哪一个运行得更快？

8. 编写一个程序，找出一个二维数组中的最大和最小元素。在main()中完成所有的工作。

9. 重新编写上一个练习中的程序，要求使用一个在参数表中用二维数组作参数的函数。提示：在用多维数组在函数定义的头中作参数时，必须要说明各维的尺寸（第一维除外）。其作用是增强函数，使它仅能应用到某些数组。考虑

```

double sum(double a[][5], int m)
/* m is the number of rows */
{
    .....

```

在这个例子中，我们把5作为a的列尺寸。编译器需要这一信息处理函数中形式为a[i][j]的表达式。如果在调用环境中a是3×5的数组，我们用sum(a, 3)调用该函数；如果a是7×5的数组，我们用sum(a, 7)调用该函数。一般而言，我们向sum()传递给任何n×5的数组。从上述来看，你可能推断出C对多维数组的处理不是很得体，但并非如此。还有更多的故事，但

由于这涉及到更复杂的指针应用，我们就不再多述。

10. 编写一个程序，按月保持10年的销售数据。数组的标引尺寸应该是12，以处理各月份。给定这一数据，对当年的各月销售数据进行排序，看哪一个月的销售最好。

11. 有很多为人熟知的排序方法。下面一个简单的转置排序的核心部分：

```
for (i = 0; i < SIZE; ++i)
    for (j = i + 1; j < SIZE; ++j)
        if (a[i] > a[j])
            swap(&a[i], &a[j]);
```

编写一个程序，完成该排序。在程序能工作后修改它，使得在每次外部循环后，都显示数组的所有元素，例如，假设数组的尺寸是8，它的开始值是

```
7  3  66  3  -5  22  -77  2
```

程序应该在屏幕上显示如下信息：

```
Unordered data:  7  3  66  3  -5  22  -77  2
After pass 1:  -77  7  66  3  3  22  -5  2
After pass 2:  -77  -5  66  7  3  22  3  2
.....
```

12. 上一个练习中的程序输出说明了作用在特殊数组上的特殊排序算法的作用。在本练习中，我们要做一些更深的思考。修改上一个练习中的程序，使得每次交换两个元素，显示数组，并对被交换的元素打上下划线。使用上一个练习中的尺寸为8的数组，程序在屏幕上应该显示如下信息：

```
 3      7      66      3      -5      22      -77      2
--      --
-5      7      66      3      3      22      -77      2
--      --
.....
```

13. 编写一个程序，把n个整数读入一个数组，然后分行显示各不同元素的值及其出现的次数。例如，假设你输入了

```
-7 3 3 -7 5 5 3
```

作为数组元素的值，那么程序应该显示

```
5 occurs 2 times
3 occurs 3 times
-7 occurs 2 times
```

用你的程序研究rand()的输出。先用rand()创建一个文件rand\_out，它含有100个从1到10的随机数。回想一下，用如下形式的for循环可以做到这一点：

```
for (i = 1; i <= 100; ++i) {
    printf("%7d", rand() % 10 + 1);
    if (i % 10 == 0)
        printf("\n");
}
```

由于我们还没有说明如何写到文件，请参见第13章“输入/输出和文件”。编写一个创建随机数的小程序cr\_rand，然后给出如下的命令，以重定向输出：

```
cr_rand > rand_out
```

14. 利用calloc()重新编写上一个程序。假设文件rand\_out的第一项是它所包含的随机

数的数目。程序要把第一项读到变量size中。假设已经把变量rand\_array声明为指向int的指针。通过利用下面的calloc(), 你能动态地分配存储空间。

```
rand_array = calloc(size, sizeof(int));
```

在分配空间后, 可以把指针rand\_array看作一个数组。例如, 为了填充数组, 你可以写

```
for (i = 0; i < size; ++i)
    scanf("%d", &rand_array[i]);
```

15. 回想一下, 赋值运算符+=的语义是用如下的规则说明的:

```
variable op expression
```

等价于

```
variable = variable op (expression)
```

以下情况除外: *variable*本身是表达式, 这样仅对它求值一次。这意味着a[expr]+=2与a[expr]=a[expr]+2不须要有相同的作用。这是一个重要的技术点。试一下下面的代码:

```
int a[] = {3, 3, 3}, i = 0;
int b[] = {3, 3, 3}, j = 0;

a[++i] += 2;           /* perfectly acceptable */
b[++j] = b[++j] + 2;   /* legal, but unacceptable */
for (i = 0; i < 3; ++i)
    printf("a[%d] = %d    b[%d] = %d\n",
           i, a[i], i, b[i]);
```

在大多数系统中, 数组a[]和b[]的值不同。在你的系统中会显示出什么?

16. C++: 用由new分配的存储空间。编写一个完成串反转的程序。

```
char* strrev(char*& s1, const char* s2);
//s1 ends up with the reverse of the string s2
//use new to allocate s1 adequate store strlen(s2) + 1
```

17. C++: 编写一个程序, 以用户提供的上界和下界从自由存储分配一个一维数组。程序应该检查上界是否超越了下界。如果没有超越, 用如下的assert.h包完成一个出错退出:

```
#include <assert.h>
.....//input lower bound and upper bound
assert(ub - lb > 0);
.....
```

18. C++: 数组的尺寸是: 上界 - 下界 + 1。给定一个标准的C++数组, 编写一个用标准数组初始化动态数组的函数。通过以精细的格式写出初始化前后的数组, 对该函数进行测试。

19. Java: 我们要找出2到100之间的素数。我们要基于埃拉托色尼筛选编程来完成这项工作。我们分配一个有100个元素的布尔数组isPrime, 并把每个元素置为真。元素从isPrime[2]开始, 其下标值是2, 其余的数组元素是isPrime[4], isPrime[6], ....., isPrime[98], 把这些元素都置为假。从isPrime[3]开始每隔3个元素置为假。这样做直到isPrime[10]为止, 因为10是100的平方根, 这足以在从2到100的范围内进行检查。当我们完成此项工作时, 只有为真的项是素数。

```
//Primes.java-Sieve of Eratosthenes for Primes up to 100.
// see Java by Dissection page 160
class Primes {
    public static void main(String[] args) {
        boolean[] sieve = new boolean[100];
        int i;
        System.out.println(" Table of primes to 100.");
        for (i = 0; i < 100; i++)
```



```

        sieve[i] = true;
    for (int j = 2; j < Math.sqrt(100); j++)
        if (sieve[j])
            crossOut(sieve, j, j + j);
    for (i = 0; i < 100; i++) //print primes
        if (sieve[i])
            System.out.print(" " + i);
}

public static void crossOut(boolean[] s,
                             int interval, int start)
{
    for (int i = start; i < s.length; i += interval)
        s[i] = false;
}
}

```

我们可以很容易地把这个程序推广到随意大的数 $n$ ，关键是用 $n$ 代替固定值100。把这个程序转换成C，要求用typedef创建一个boolean类型，并把程序推广到 $n$ 。显示一个300以内的素数表。

20. Java: 使用二维数组，用C重新编写下面的Java程序。

```

// TwoD.java - simple example of two-dimensional array
//Java by Dissection page 164.
class TwoD {
    public static void main(String[] args) {
        int[][] data = new int[3][5];
        for (int i = 0; i < data.length; i++) {
            System.out.print("Row " + i + ": ");
            for (int j = 0; j < data[i].length; j++) {
                data[i][j] = i * j;
                System.out.print(data[i][j] + ", ");
            }
            System.out.println();
        }
    }
}

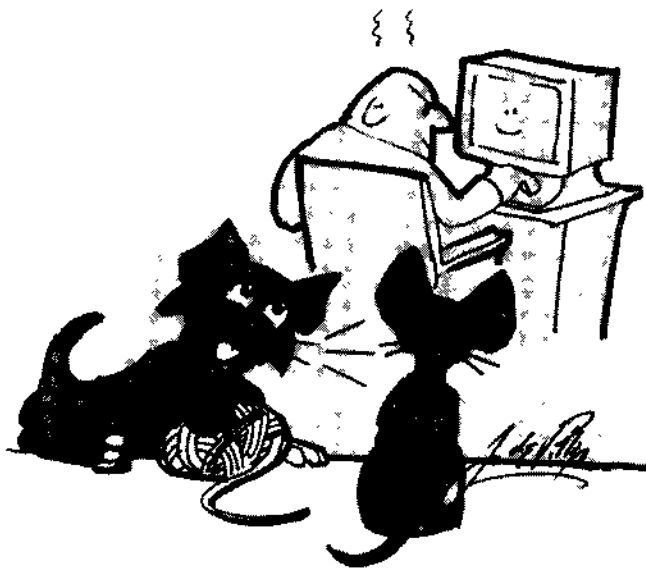
```

Java用new分配内存，而C用calloc()或malloc()分配内存。同样，Java自动地用成员length跟踪各维的尺寸，这些使得Java比C更安全与易用。

## 第10章 串 和 指 针

在C中，串是char型的一维数组。可以把串中的字符作为数组中的元素访问，或利用指向char的指针对其访问，这种灵活性使得C在编写串处理程序时特别有用。标准库提供了很多有用的串处理函数。

虽然可以把对串的处理看作是对数组的处理的一种特殊情况，但是对串的处理也有它为人们所喜爱的特征。重要的是用字符值\0终止串的用法。本章中包含了一些示例程序，用以说明串处理的思想。在第9章“数组和指针”中，我们已经看到了数组和指针是紧密相关的。类型指向char的指针在概念上是串。我们的例子说明了适当处理串数据所需要的指针运算和间接引用。



“喔，C监视器，米滕，如果人类认为串很难，那么人类还能算聪明吗？”

## 10.1 串结束标志

按照惯例，用串结束标志\0或空字符终结串。在内存中把"abc"这样的常量串以4个字节存储，最后一个字节是\0。因此串"abc"的长度是3，但该串的尺寸是4。要为一个串分配内存，我们可以写

```
#define MAXWORD 100

int main(void)
{
    char w[MAXWORD];
```

在分配存储空间后，有很多把字符值放进串w中的方法。第一种方法，把字符一个接一个

放进串中：

```
w[0] = 'A';
w[1] = 'B';
w[2] = 'C';
w[3] = '\\0';
```

注意，我们用空字符结束串。

第二种方法，利用scanf()把字符值放进w中，用格式%s读入串。可以认为这个过程分三步。声明

```
scanf("%s", w);
```

首先定位输入流中的最初非空白字符，然后把各非空白字符放到以w的基地址开始的内存中，最后当遇到空白字符或EOF时停止读入。此时，用一个空格字符结束串。由于数组名是指向数组基地址的指针，所以表达式w等价于&w[0]。由于w的尺寸是100，我们可以从键盘上输入99个字符，最后用\\0结束。如果输入的过多，那么就超出了w的界限。

标志\\0也称为定界符(delimiter)，用它能很简单地检测出串是否结束。串的长度由空字符限定，其长度可变有助于我们的工作，但最大的长度由数组的尺寸决定。串的尺寸必须包括存储空字符所需的空空间。像所有的数组一样，程序员必须要保证不越界。

注意，'a'和"a"是不同的。前一个是字符常量，后一个是串常量。串"a"是一个有两个元素的数组，前一个的值是'a'，后一个的值是'\\0'。

## 10.2 串的初始化

回想一下，对数组（当然包括字符数组）可以初始化。

```
char s[] = {'a', 'b', 'c', '\\0'};
```

它声明了char型的数组s，s有4个元素，其串值是abc。还有一种等价的初始化字符数组的语法。

```
char s[] = "abc";
```

这与上一个例子的含义是完全一致的，但它比较简洁，更为大多数C程序员所喜爱。在两种情况下，数组的尺寸比串长大1，本例中数组的尺寸是4。

还可以用常量串初始化指向char的指针，但解释是不同的，下面是一个例子：

```
char *p = "abc";
```

回想一下，把数组名本身看作是指向数组基地址的指针。像所有的其他常量一样，编译器把串常量存储在内存中（程序员不为常量提供空间），因此编译器把"abc"存储在内存中，用这个指针值对变量p初始化。

这样，用常量串初始化数组和用常量串初始化指针之间的区别是，数组包含最后跟一个空字符的若干个体字符，而指针被赋予了内存中的常量串的地址。

## 10.3 例子：心情愉快

因为串是一个字符数组，处理串的一个方法是用带有下标的数组。我们要编写一个说明这点的交互式程序，我们的程序要把用户输入的一行字符读入到一个串中，再按反序显示串，并对数组中的字母求和。

```

/* Have a nice day! */
#include <stdio.h>
#include <ctype.h>

#define MAXSTRING 100

int main(void)
{
    char c, name[MAXSTRING];
    int i, sum = 0;

    printf("\nHi! What is your name? ");
    for (i = 0; (c = getchar()) != '\n'; ++i) {
        name[i] = c;
        if (isalpha(c)) /* sum the letters */
            sum += c;
    }
    name[i] = '\0';
    printf("\n%s%s\n",
        "Nice to meet you ", name, ".",
        "Your name spelled backward is ");
    for (--i; i >= 0; --i)
        putchar(name[i]);
    printf("\n%s%d\n",
        "and the letters in your name sum to ", sum, ".",
        "Have a nice day!");
    return 0;
}

```

假设我们执行这个程序，在提示时输入“C.B.Diligent”，在屏幕上出现的信息如下：

```

Hi! What is your name? C. B. Diligent

Nice to meet you C. B. Diligent.
Your name spelled backward is tnegiliD .B .C
and the letters in your name sum to 949.

Have a nice day!

```

### 对程序nice\_day的解析

- #include <stdio.h>  
#include <ctype.h>

标准头文件stdio.h含有printf()的函数原型，它还含有getchar()和putchar()的宏定义。标准头文件ctype.h含有isalpha()的宏定义。

- #define MAXSTRING 100

用符号常量MAXSTRING设置字符数组name的尺寸。我们现在假设程序的用户输入的字符合不多于100个。

- char c, name[MAXSTRING];  
int i, sum = 0;

变量c的类型是char。标识符name的类型是“char型数组”，它的尺寸是MAXSTRING。在C中，所有的数组下标都从0开始，因此name[0]，name[1]，……，name[MAXSTRING-1]都是该数组的元素。变量i和sum的类型是int；sum被初始化为0。

- printf("\nHi! What is your name? ");

该语句给用户一个提示。现在程序希望输入一个后跟一个回车的名。

- (c = getchar()) != '\n'

这个表达式由两部分组成。左边是(`c=getchar()`)，与其他语言不同，在C中赋值是一个运算符（请参见2.10节“赋值运算符”）。这里，用`getchar()`从键盘读入一个字符，并把该字符赋给`c`。表达式的值就是赋给`c`的值。由于运算符`=`的优先级小于运算符`!=`的优先级，所以括号是必要的。因此，`c=getchar() != '\n'`等价于`c=(getchar() != '\n')`。

```

• for (i = 0; (c = getchar()) != '\n'; ++i) {
    name[i] = c;
    if (isalpha(c))          /* sum the letters */
        sum += c;
}

```

开始变量`i`的值是0。`getchar()`从键盘上取得一个字符赋给`c`，并测试它是否是换行符。如果不是，就执行for循环体。首先把`c`的值赋给数组元素`name[i]`，然后用宏`isalpha()`决定`c`是大写字母还是小写字母。如果`c`是一个字母，就用`c`的值对`sum`增量。正如我们在第5章“字符处理”中看到的那样，在C中，字符有与相应的ASCII码对应的整数值。例如，字符‘a’的ASCII码值是97，‘b’的是98；等等。最后，在for循环的结束处变量`i`加1。重复地执行for循环，直到接收到一个换行符为止。

```

• name[i] = '\0';

```

在完成for循环后，把空字符`\0`赋值给元素`name[i]`。根据惯例，所有的串用空字符结束。像`printf()`这样的处理串的函数用空字符作为串结束的标记。现在我们把内存中的数组`name`看作是：

C	.		B	.		D	i	l	i	g	e	n	t	\0	*	...	*
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		99

注意，\*表示数组中在`\0`后的未知内容。

```

• printf("\n%s%s\n",
    "Nice to meet you ", name, " ",
    "Your name spelled backward is ");

```

用格式`%s`显示串。这里，数组`name`是四个正被显示的串参数之一。一个接一个地显示数组元素，直到遇到串结束标记`\0`为止。这个语句的效果是在屏幕上显示：

```

Nice to meet you C. B. Diligent.
Your name spelled backward is

```

```

• for (--i; i >= 0; --i)
    putchar(name[i]);

```

如果我们假设“C.B.Diligent”后跟一个输入的换行符，那么在for循环的开始处`i`的值是14（不要忘记是从0而不是1开始算起）。在对`i`减量后，下标对应着已输入的`name`的上一个字符。因此，这个for循环的效果是在屏幕按反序显示`name`。

```

• printf("\n%s%d\n\n%s\n",
    "and the letters in your name sum to ", sum, ".",
    "Have a nice day!");

```

我们显示用户已输入的`name`中的字母的总和，并显示最终信息。 ■

## 10.4 用指针处理串

在上一节中，我们说明了用下标处理串；在本节中，我们要用指针处理串，我们还要表

明如何用串作为函数的参数。

让我们编写一个交互式小程序，它读入用户输入的一行字符作为一个串。然后程序要创建一个新串，并要打印它。

```
/* Character processing: change a line. */
#include <stdio.h>

#define MAXLINE 100
void read_in(char *);

int main(void)
{
    char line[MAXLINE], *change(char *);

    printf("\nWhat is your favorite line? ");
    read_in(line);
    printf("\n%s\n\n%s\n\n",
        "Here it is after being changed:", change(line));
    return 0;
}
```

在对用户进行提示后，该程序用read\_in()把字符放进line。把line作为参数传递给change()，该函数返回一个指向char的指针。以串格式的printf()显示返回的指针值。下面是函数read\_in()：

```
void read_in(char s[])
{
    int c, i = 0;

    while ((c = getchar()) != EOF && c != '\n')
        s[i++] = c;
    s[i] = '\0';
}
```

参数s的类型是指向char的指针。不妨写为

```
void read_in(char *s)
{
    .....
```

在while循环中，从输入流中得到连续字符，并一个接一个地把它放进以基地址s开始的数组中。当接收到换行符时，就终止循环，并把空字符放进数组中作为串结束标记。注意，这个函数没有分配存储空间。在main()中，用line的声明分配存储空间。我们假设用户输入的字符数小于MAXLINE。我们把这个检查留做练习（请参见练习6）。当把line作为参数传递给read\_in()时，传递的是数组的基地址的拷贝，参数s接收这个值，数组元素本身不被拷贝，但可在read\_in()中通过基地址访问它们。

```
char *change(const char *s)
{
    static char new_string[MAXLINE];
    char *p = new_string;

    *p++ = '\t';
    for (; *s != '\0'; ++s)
        if (*s == 'e')
            *p++ = 'E';
        else if (*s == ' ') {
            *p++ = '\n';
            *p++ = '\t';
        }
    else
        *p++ = *s;
}
```

```

    *p = '\0';
    return new_string;
}

```

这个函数接收一个串并进行复制，同时要把每个e变为E，并用换行和跳格符代替每个空格。假设我们运行程序，在提示后输入

```
she sells sea shells
```

屏幕上出现的信息如下：

```
What is your favorite line? she sells sea shells
```

变化后信息为：

```
shE
sElls
sEa
shElls
```

我们要较详细地解释函数change()是如何工作的。

### 对函数change()的解析

```

• char *change(const char *s)
{
    static char    new_string[MAXLINE];
    char          *p = new_string;

```

第一个char是函数的返回类型，它告诉编译器change()返回一个指向char的指针。把参数s和局部变量p声明为指向char的指针型的。因为s是函数头中的一个参数，我们无妨写

```

char *change(const char s[])
{
    .....

```

然而，由于p是局部变量而不是参数，对p做类似的声明是错误的。把数组new\_string的存储类型声明为静态的，为MAXLINE个字符分配存储空间。本解析的后续部分要解释使用静态而不使用自动的原因。把指针p初始化为new\_string的基地址。

```

• *p++ = '\t';

```

该代码行等价于

```

*p = '\t';
++p;

```

现对此情形分析如下。因为运算符\*和++是一元的，并从右到左结合，所以表达式\*p++等价于\*(p++)。这样++运算符使得p被增量，相反(\*p)++使得p指向的值被增量，可以看出二者是截然不同的。

由于++运算符出现在p的右侧，而不是左侧，所以在对整个表达式\*p++ = '\t'求值后，才对p增量。赋值是求值过程的一部分，这使得把一个跳格符赋值给p指向的位置。因为p指向new\_string的基地址，所以把跳格符赋值给new\_string[0]。在对p增量后，p指向new\_string[1]。

```

• for ( ; *s != '\0'; ++s)

```

每次进行for循环，都必须测试s所指位置的值是否是串结束标志，如果不是，那么就执行for循环体，并且对s进行增量。对指向char的指针增量的作用是使指针指向数组的下一个字符。

```
• if (*s == 'e')
    *p++ = 'E';
```

在for循环体中,要测试s是否指向字符e,如果是,那么把E赋值给p所指向的对象,并对p增量。

```
• else if (*s == ' ') {
    *p++ = '\n';
    *p++ = '\t';
}
```

否则,要测试s是否指向空格,如果是,那么把换行符赋给p所指向的对象,并对p增量,再把跳格符赋值给p所指向的对象,并再次对p增量。

```
• else
    *p++ = *s;
```

如果s指向的字符既不是e也不是空格,那么把s指向的对象的值赋给p所指向的对象,并对p增量。这个for循环的作用是把作为参数传递给change()的串复制成从new\_line[1]开始的串,只是用E代替了e,用换行和跳格符代替了空格。

```
• *p = '\0';
```

当退出for循环时,把串结束标志赋值给p指向的对象。

```
• return new_string;
```

返回数组名new\_string。把数组名本身作为指向内存中数组基地址的指针。由于new\_string的存储类型是静态的,所以在函数退出时在内存中要保存它。 ■

如果new\_string的存储类型是自动的而不是静态的,分配给它的内存在函数退出时将被不被保存。在很多系统中,这会导致难以诊断的错误。一种可能是,内存会被重写,并且main()中的最后一个printf()语句的工作会不正常。

## 10.5 问题求解: 单词计数

上一节的例子说明了指针的用法和处理串的指针运算,在本节中我们要对此给出另一个说明。我们编写一个记录串中词的个数的函数,就本函数而言,词由一系列非空白字符组成。

```
/* Count the number of words in a string. */
#include <ctype.h>

int word_cnt(const char *s)
{
    int cnt = 0;

    while (*s != '\0') {
        while (isspace(*s)) /* skip white space */
            ++s;
        if (*s != '\0') { /* found a word */
            ++cnt;
            while (!isspace(*s) && *s != '\0') /* skip word */
                ++s;
        }
    }
    return cnt;
}
```

这是一个典型的串处理函数。用指针运算和间接引用搜索各种模式或字符。



## 10.6 把参数传递给main()

C提供任何类型的数组,也包括指针数组。虽然这是一个高级的主题(我们不想详细讨论),但我们需要使用指向char的指针数组来编写使用命令行参数的程序。按照惯例被称为argc和argv的两个参数能和main()一起使用,以和操作系统通信。下面是一个显示其命令行参数的程序:

```
/* Echo the command line arguments. */
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;

    printf("argc = %d\n", argc);
    for (i = 0; i < argc; ++i)
        printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```

变量argc提供了对命令行参数的计数,数组argv是指向char的指针,可把它看作是一个字符串数组。因为元素argv[0]总是含有命令本身的名字,所以argc的值总是1或更大。假设我们对该程序进行编译,并把可执行代码放进了my\_echo中。如果我们发出命令

```
my_echo
```

则在屏幕上显示如下信息:

```
argc = 1
argv[0] = my_echo
```

现在假设我们给出命令

```
my_echo try this
```

在屏幕上会出现如下信息:

```
argc = 3
argv[0] = my_echo
argv[1] = try
argv[2] = this
```

可以按下述形式声明参数argv:

```
char **argv;
```

正是指向char的指针的指针能被看作是指向char的指针数组,又可以把它看作是字符串数组。注意,我们没有为命令行上的串分配任何空间。系统会做这些的,并通过两个参数argc和argv把信息传递给main()。

## 10.7 标准库中的串处理函数

标准库中含有很多有用的串处理函数。它们都要求作为参数传递的串是以空结尾,并且它们都返回整数值或指向char的指针。下面描述了一些函数(在附录A“标准库”中描述了所有串处理函数)。在头文件string.h中给出了函数原型,在使用这些串处理函数时要引入这个文件。

### 标准库中一些串处理函数

- `char *strcat(char *s1, const char *s2);`

这个函数用两个串做参数，对两个串做拼接，并把拼接的结果放入s1。程序员必须要保证s1指向的空间能容纳结果值。函数返回的是串s1。

- `char *strcmp(const char *s1, const char *s2);`

这个函数用两个串做参数。按照字典次序看s1是小于、等于或大于s2，返回的整数就相应地小于、等于或大于0。

- `char *strcpy(char *s1, const char *s2);`

把串s2复制到s1，直到遇到s2中的\0为止。s1中的内容被覆盖掉。s1要有足够的空间容纳s2。函数返回的是s1。

- `unsigned strlen(const char *s);`

函数返回\0前的字符个数。

这些函数没有什么特殊之处。它们都是用C编写的，而且都相当短。为了执行较快，把这些函数中的变量都声明为存储类型register。下面是编写函数strlen()的一个方法。

```
unsigned strlen(const char *s)
{
    register int    n = 0;

    for ( ; *s != '\0'; ++s)
        ++n;
    return n;
}
```

下表说明了串处理函数。注意，为作为参数传递给函数的串分配足够大的空间是程序员的责任。串越界是一个常见的编程错误。

声明和初始化	
char s1[] = "beautiful big sky country", s2[] = "how now brown cow";	
表达式	值
strlen(s1)	25
strlen(s2 + 8)	9
strcmp(s1, s2)	<i>negative integer</i>
语句	打印内容
printf("%s", s1 + 10);	big sky country
strcpy(s1 + 10, s2 + 8);	
strcat(s1, "s!");	
printf("%s", s1);	beautiful brown cows!

在应用标准库中的任何函数之前，你必须要提供函数原型。通常用#include <string.h>来引入头文件string.h，这个头文件含有标准库中的所有串处理函数的原型。

## 10.8 风格

有两种用于处理串的编程风格，它们是带下标的数组表示法和指针及指针运算。虽然这两种都是常用的，但有经验的程序员偏爱于使用指针。

由于空字符总是用于划定串的界限，所以在处理串时显式地测试\0是一种常见的编程风格。然而，这样做不是必须的，使用串的长度也是一种作法。作为一个示例，我们可以写

```
n = strlen(s);
for (i = 0; i <= n; ++i)
    if (islower(s[i]))
        s[i] = toupper(s[i]);
```

来把小写的字母都变成大写字母。这种处理串的风格当然是可以接收的。然而，要注意形式为

```
for (i = 0; i <= strlen(s); ++i)
    .....
```

的for循环是低效的。这段代码会使得在每次循环时都计算s的长度。

有时习惯于用指向char的指针指向一个常量串。作为一个例子，考虑

```
char *p;

p = "RICH";
printf("C. B. DeMogul is %s %s %s!\n", p, p, p);
```

该段代码的另一种写法是

```
printf("C. B. DeMogul is %s %s %s!\n",
      "RICH", "RICH", "RICH");
```

在本例中，对p的重复使用节省了一些内存空间。编译器为各常量串分别地分配存储空间，即使常量串是相同的也如此。

在可能之处，程序员应该使用标准库中的函数，而不要再编写相同的函数，即使是专门编写的函数在效率上能有其他的收获也不应该这样做。标准库中的函数都被设计为跨系统可移植的。

虽然这样做是一种不良的编程作法，但是在大多数系统中指向一个常量串的指针能改变串的内容（请参见练习9）。

## 10.9 常见的编程错误

一种常见的编程错误是串越界。像其他的数组一样，程序员有责任确保为串分配足够的空间。考虑

```
char s[17], *strcpy();

strcpy(s, "Have a nice day!\n");
```

此处，程序员仔细地数了要复制到s中的串中的字符个数，但忘记了为空字符分配空间。在使用像strcpy(s1, s2)这样的函数调用时容易发生函数越界，两个串拼接的结果必须要放在为s1分配的空间里。

另一种常见的编程错误是忘记了用空字符终止串，在大多数系统中，编译器不能捕捉此类错误，这样的错误的影响可能是时有时无的，有时程序能正确地运行，有时不能正确地运

行, 此类错误可能是难以发现的。另一种常见的编程错误是把'a'写作"a", 或把"a写作'a', 通常编译器能发现这样的错误。使用像scanf("%s", &w)这样的函数调用把一个串读入字符数组w也是一个错误, 因为w本身是一个指针, 正确的函数调用是scanf("%s", w)。

## 10.10 系统考虑

在ANSI C中, 编译器把用0个或多个空白字符分隔的串常量连接成一个长串, 传统C编译器不支持这样的作法, 下面是一个例子:

```
char *long_string;
long_string = "A list of words:\n"
              "\n"
              " 1 abacus\n"
              " 2 bracelet\n"
              " 3 cafeteria\n";
```

本例的优点是它包含的串看起来与显示的结果一样。传统C编译器对此并不支持。

标准库含有17个串处理函数(请参见附录A“标准库”), 使用这些函数而不重新再编写它们是一种良好的编程作法。这些函数增强了到其他系统的可移植性, 虽然老的编译器并不支持所有的串处理函数。新的ANSI C标准已经增加了很多函数。

## 10.11 转向C++

C++也把以空字符结尾的字符序列作为串, 但是越来越多的人愿意使用C++标准模板库(STL)中数据类型string。把类型string变为\*char是容易的。在现在的C++中, C库用c做前缀, 因此cstring与C库中的string.h是一样的。通过引入文件string定义这种新的串类型。

类型string的函数和运算的范围更广。下面是一些例子;

```
string a, b("ABC"), c("DEF"); // a is empty, b is "ABC",
                               // c is "DEF"
b + c // + stands for concatenation
b == c // == is an equality test
a = b // = is assignment
a.to_lower() //changes a's value to lowercase
```

## 小结

- 串是char型的一维数组。空字符\0用于限定串。像printf()这样的系统函数仅能处理以空字符结尾的串。
- 像scanf("%s", w)这样的函数调用能用于把非空白字符读入串w。在读入所有的字符后, scanf()自动地用空字符结尾。
- 可以对串进行初始化。一种标准化的形式是

```
char *s[] = "cbd";
```

编译器把它与如下的初始化看作是等价的:

```
char *s[] = {'c', 'b', 'd', '\0'};
```

- 通过用带下标的数组和用指针及指针运算能对串进行处理。由于这种灵活性, C被广泛地用于处理串。
- 在C中通过在main()的函数定义中使用参数argc和argv, 可以访问命令行参数。参数

argc是int型的；它的值是命令行参数的个数。参数argv是指向char的指针数组；可以把它认为是串数组。系统在内存中像放置串那样放置命令行参数，并用数组argv的元素指向命令行参数。argc的值总是1或更大，argv[0]指向的串总是命令的名字。

- 标准库含有很多有用的串处理函数。例如，可用像strcmp(s1,s2)这样的函数调用按字典次序比较串s1和串s2。

## 练习

1. 用指针和指针运算重新编写10.3节“例子：心情愉快”中的程序nice\_day。

2. 使用带下标的数组重新编写10.5节“问题求解：对词计数”中的函数word\_cnt()。编写一个交互式的程序，它读入用户输入的行，算出行中词的个数。程序应该能处理长行。通过实验，看如果你输入了超过屏幕的一行会发生什么情况。

3. 编写一个搜索一个串中的字母的函数。从在串中出现的字母中，函数要找出出现次数最少但至少出现一次的字母，以及出现次数最多的字母。把这些信息和相应的字母的出现次数返回到调用环境。该函数定义的开始部分如下：

```
void search(char s[], char *p_least, char *p_most,
            int *p_least_cnt, int *p_most_cnt)
{
    .....
```

分别处理小写和大写字母。要保证对没有字母的串也可以进行得体的处理。编写一个程序测试你编写的函数。

4. 编写一个函数bubble\_string()，当执行bubble\_string(s)时，它对串s中的字符进行冒泡排序。如果串s是"xylophone"，那么下面的语句会显示ehlnoopyx：

```
printf("%s\n", bubble_string(s));
```

5. 修改10.6节“把参数传递给main()”中的程序my\_echo，使得它有如下作用。如果发出命令行

```
my_echo pacific sea
```

那么应该显示如下的结果：

```
pacific
sea
```

进一步修改它，使得如果使用选项-c就以大写的形式显示参数，但不显示包含该选项的参数。

6. 通过增加一个检查不超出MAXLINE个字符的函数assert()，完善10.4节“用指针处理串”中的void read\_in(char s[])。

7. 自己编写一个库函数strcmp()。附录A“标准库”中给出了该函数的原型和行为描述。

8. 在本练习中，我们要使用指向char的指针多维数组。请完成下表。

9. 在很多系统中，可以改变常量串的值。这是一种很坏的编码作法，下面是一个为什么不这样做的例子。请说出显示的结果，并解释原因。

```
#include <stdio.h>

int main(void)
{
    char    *p, *q;
```

```

p = q = " RICH";
printf("C. B. DeMogul is%s%s%s!\n", p, p, p);
*++q = 'p';
*++q = 'o';
*++q = 'o';
*++q = 'r';
printf("C. B. DeMogul is%s%s%s!\n", p, p, p);
return 0;
}

```

声明和初始化		
char *p[2][3] = { "abc", "defg", "hi", "jklmno", "pqrstuvw", "xyz" };		
表达式	等价表达式	值
***p	p[0][0][0]	'a'
**p[1]		
**(p[1] + 2)		
*(*(p + 1) + 1)[7]		错误
(*(*(p + 1) + 1))[7]		
*(p[1][2] + 2)		

注意 一些系统能检测到并禁止这样的对常量串的改变。

10. 编写一个交互式的程序，它用scanf()读入用户输入的7个串。程序把7个串显示成一张表，然后按字典次序排序并显示一个新表。对串排序时可使用函数strcmp()。编写程序时也要使用预处理指令

```
#define N_STRINGS 7
```

以这样的方式，当需要对不同数目的串排序时仅需改变这一行。

11. (高级) 编写一个与上个练习相似的程序，它对各命令行参数排序并显示。

12. C++: 用cstring中的string编写一个对输入文件test.txt中的串"in"的出现个数进行计数的程序。

13. Java: 在下述的Java程序中，我们有一个Java串数组。

```

// StringArray.java - uses a string array initializer
// Java by Dissection page 176.
class StringArray {
    public static void main(String[] args) {
        String[] myStringArray = { "zero", "one", "two",
            "three", "four", "five", "six", "seven",
            "eight", "nine" };
        for (int i = 0; i < myStringArray.length; i++)
            System.out.println(myStringArray[i]);
    }
}

```

把它重新编写为C程序。Java程序员习惯了Java的处理存储管理的方法，这在C中更复杂。在C中，你需要使用一个数组，其元素是指向char的指针。

14. Java: 在Java中，我们用String和StringBuffer作为不可修改串的类型和可修改串的类型。用C重新编写下述程序：

```

// StringTest.java - demo some String methods
// Java by Dissection pages 193-194
public class StringTest {

```

```

    public static void main(String[] args) {
        String str1 = "aBcD", str2 = "abcd", str3;
        System.out.println(str1.equals(str2));
        System.out.println(str1.length());
        System.out.println(str1.charAt(1));
        System.out.println(str1.compareTo("aBcE"));
        System.out.println(str1.compareTo("aBcC"));
        System.out.println(str1.compareTo("aBcD"));
        System.out.println(str1.indexOf('D'));
        System.out.println(str1.indexOf("Bc"));
        System.out.println(str1.indexOf("zz"));
        System.out.println(str1.concat("efg"));

        str3 = str1.toLowerCase();
        System.out.println(str3);
        str3 = str1.toUpperCase();
        System.out.println(str3);
        System.out.println(str1);
        str3 = String.valueOf(123);
        System.out.println(str3.equals("123"));
    }
}

```

该程序要显示：

```

false
4
B
-1
1
0
3
1
-1
aBcDefg
abcd
ABCD
aBcD
true

```

要记住在Java中，用方法`equals()`比较两个串；如果按字典顺序`str1`是在参数之前，那么对方法`compareTo()`的调用的返回值是`-1`；对方法`indexOf()`进行重载，以接收`String`或`char`作为参数。当参数是`char`时，方法返回字符出现的串的第一个位置的标引。当参数是`String`时，方法返回在被操作的`String`中（本例是`str1`）参数`String`的位置标引。如果参数`String`不是被操作`String`的一个子串，那么方法的返回值是`-1`。最好把`str1+"efg"`看作是`str1.concat("efg")`的缩写。记住，对`toLowerCase()`和`toUpperCase()`的调用不影响`String str1`，这些调用会返回包含适当字符的新串对象。方法`valueOf()`是类方法的一个例子；它不是实例方法，也不处理串对象。它与串类相关是因为它能用于创建串对象，因而把它定义在串类中。对方法`valueOf()`进行重载是为了能接收任何基本类型的参数。在各种情况下，该方法都返回一个描述基本值的`String`表示。

# 第11章 递 归

递归(recursion)是函数直接或间接地对自己进行的调用。自然地,一些编程任务需要用递归来解决,可以把递归看作是控制流的高级形式。可选用递归代替迭代。

我们要用一些简单的示例程序解释递归。一个特别好的例子是在屏幕上绘制图案的函数,另一个例子是递归地计算串的长度。递归是一种自然地实现分而治之解决问题的编程技术,我们要用一个排序算法作为例子来解释一个强有力的策略。

## 11.1 递归问题求解

如果函数直接或间接地对自己进行调用,就说函数是递归的。在C中,所有的函数都可以递归地使用。直接递归是最简单的形式。

```
#include <stdio.h>

void count_down(int n);

int main(void)
{
    count_down(10);
    return 0;
}

void count_down(int n)
{
    if (n) {
        printf("%d ! ", n);
        count_down(n - 1);
    }
    else
        printf("\nBLAST OFF\n");
}
```

该程序的显示结果是:

```
10 ! 9 ! 8 ! 7 ! 6 ! 5 ! 4 ! 3 ! 2 ! 1 !
BLAST OFF
```

这也可以用一个包含printf()的循环语句来完成。这里的新颖之处在于count\_down()调用自己。

下面是另一个递归函数的简单示例,它要计算前n个正整数之和:

```
int sum(int n)
{
    if (n <= 1)
        return n;
    else
        return (n + sum(n - 1));
}
```

下表是对递归函数sum()的分析。首先要考虑基本情况,然后由基本情况进行推算再考虑其他情况。

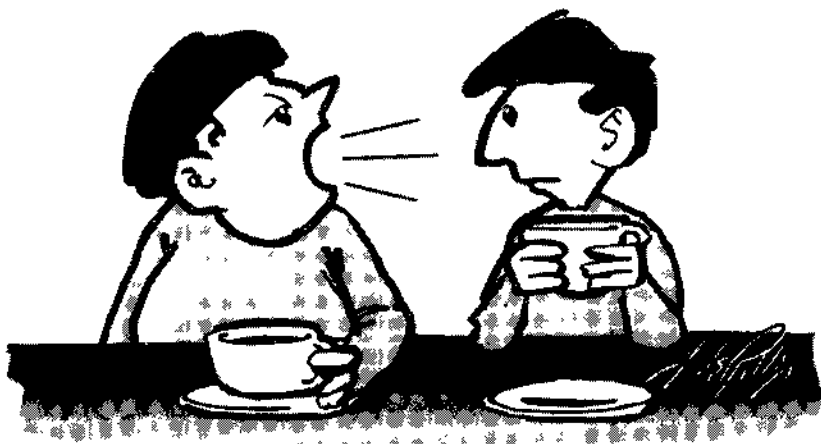


sum() 返回值	
函数调用	返回值
sum(1)	1
sum(2)	2 + sum(1) 即 2 + 1
sum(3)	3 + sum(2) 即 3 + 2 + 1
sum(4)	4 + sum(3) 即 4 + 3 + 2 + 1

简单的递归例程有一个标准模式。通常，有可对函数的入口进行测试的基本情况。然后把函数中的一个变量（一般为整型）作为参数，考虑一般的递归情况，最终递归情况能演化到基本情况。在sum()中，每次变量n都减1，直到达到n等于1这个基本情况。

递归是一种强大的解决问题的技术，其关键是识别出一般情况。一个易犯的错误是忘记了考虑达到终止递归的基本情况。

在把整数n传递给sum()时，在返回前递归一层一层地使用嵌套函数中的n的拷贝，这意味着在计算中进行了n次函数调用。大多数简单的递归函数都能改写为等价的迭代形式。由于函数调用通常比迭代要求更多的计算，为什么还使用递归呢？如果使用递归能容易编写和维护代码，且运行效率并不至关重要，那么就使用递归。



“这是她第13次拒绝我了，但是我无法确定这是迭代还是递归。”

在本章的其余部分，我们要给出几个说明递归的思想和能力的例子。下面的程序首先提示用户输入一行，然后使用递归函数按逆序重新显示该行。

```
/* Write a line backward. */
#include <stdio.h>
void prn_it(void);
int main(void)
{
    printf("Input a line: ");
    prn_it();
    printf("\n\n");
    return 0;
}
void prn_it(void)
{
    char c;
```

```

    if ((c = getchar()) != '\n')
        prn_it();
    putchar(c);
}

```

如果在提示后用户输入的行是sing a song of sixpence, 那么在屏幕上出现的结果如下:

Input a line: sing a song of sixpence

ecnepxis fo gnos a gnis

### 对程序wrt\_bkwd的解析

- printf("Input a line: ");  
prn\_it();

在向用户提示后, 开始调用递归函数prn\_it()。用户用回车(字符\n)终止输入的行。

- if ((c = getchar()) != '\n')  
prn\_it();

随着每个字符的读入, 都启动一次新的对prn\_it()的调用。每次调用时变量c都有自己的局部存储空间, 用于存入输入行中的字符。每次调用都被堆叠, 直到读入换行符为止。

- putchar(c);

仅在读入换行符后, 才开始显示。现在prn\_it()的每次调用显示存储在局部变量c中的值。首先输出换行符, 然后输出换行符前的第一个字符, 这样下去, 直到输出第一个字符为止。这样, 就实现了输入行的反转。 ■

该程序的一个有趣的变体是读入一行词, 然后按词把整行倒置输出, 下面的程序能做到这一点:

```

#include <ctype.h>
#include <stdio.h>

#define MAXWORD 100

void prn_it_by_word(void);
void get_word(char *);

int main(void)
{
    printf("Input a line: ");
    prn_it_by_word();
    printf("\n\n");
    return 0;
}

void prn_it_by_word(void)
{
    char w[MAXWORD];

    get_word(w);
    if (w[0] != '\n')
        prn_it_by_word();
    printf("%s ", w);
}

void get_word(char *s)
{
    static char c = '\0';

    if (c == '\n')
        *s++ = c;
    else
        while (!isspace(c = getchar()))

```

```

        *s++ = c;
    *s = '\0';
}

```

如果在提示后用户输入的是deep in the heart of texas, 那么在屏幕上出现的结果如下:

```
Input a line: deep in the heart of texas
```

```
texas of heart the in deep
```

注意, 变量c的存储类型是static, 这样当读入换行符并赋给c后, 在退出函数后标记值不会丢失。仅当第一次调用函数get\_word()时, 对c进行了初始化, 这是因为c的存储类型是static, 在后续的函数调用中, c的值是前次调用退出时c的值。因为静态存储保证被初始化为0, 对c的显式初始化实际上是不必要的。然而, 这样能提示读者变量c的初始化值是一个空字符。

下一个例子说明了一个对串中的字符进行处理的函数, 也可以很容易地把该程序改编成一个等价的迭代函数。

```

/* Recursively reverse characters from s[j] to s[k]. */
#include <stdio.h>

void reverse(char *s, int j, int k);
void swap(char *, char *);

int main(void)
{
    char phrase[] = "by the sea, by the beautiful sea";

    reverse(phrase, 3, 17);
    printf("%s\n", phrase);
    return 0;
}

void reverse(char *s, int j, int k)
{
    if (j < k) {
        swap(&s[j], &s[k]);
        reverse(s, ++j, --k);
    }
}

void swap(char *p, char *q)
{
    char tmp;

    tmp = *p;
    *p = *q;
    *q = tmp;
}

```

下面是程序的输出:

```
by eht yb ,aes eht beautiful sea
```

注意, 从0开始计数已经把从位置3到位置17的字符反转。

## 11.2 例子: 在屏幕上绘制图案

使用递归函数可在屏幕上绘制精致的图案, 我们用一个简单的例子来说明这一点。

```

#include <stdio.h>

#define SYMBOL '*'
#define OFFSET 0
#define LENGTH 19

```

```

void display(char, int, int);
void draw(char, int);

int main(void)
{
    display(SYMBOL, OFFSET, LENGTH);
    return 0;
}

void display(char c, int m, int n)
{
    if (n > 0) {
        draw(' ', m);
        draw(c, n);
        putchar('\n');
        display(c, m + 2, n - 4);
    }
}

void draw(char c, int k)
{
    if (k > 0) {
        putchar(c);
        draw(c, k - 1);
    }
}

```

函数main()调用display(), display()调用draw()和display(), display()是递归的。函数draw()显示k个字符c。我们也把这个函数写成递归的了。当执行该程序时, 屏幕上会出现如下的图案:

```

*****
*****
*****
*****
*****

```

### 11.3 用递归处理串

串由内存中的连续的字符组成, 并由一个空字符\0结尾。在概念上, 我们把串看成仅由一个空字符组成的空串, 或后跟一个串的一个字符。这样的串定义描述了串是一种递归的数据结构, 我们能用这样的定义对一些基本的处理串的函数进行递归地编码。

在第10章“串和指针”中, 我们说明了怎样用迭代对标准库函数strlen()进行编码。下面的代码说明了如何用递归实现它。

```

/* Recursive string length. */
int r_strlen(const char *s)
{
    if (*s == '\0')
        return 0;
    else
        return (1 + r_strlen(s + 1));
}

```

这里的基本情况是测试空串, 如果是空串就返回。用r\_strlen(s + 1)进行递归调用, 此处的s + 1是指针表达式, 该表达式指向串中的下一个字符。

这种优雅的递归方式有损于运行效率。如果一个串的长度是k, 那么就要进行k+1次对r\_strlen()的调用。优化的编译器能避免这点。

有时串比较是更为复杂的。下面是标准库函数strcmp()的递归版本, 它按字典次序比

较两个串的至多前 $n$ 个字符。

```
/* Recursive string n compare. */
int r_strncmp(const char *s1, const char *s2, int n)
{
    if (*s1 != *s2 || *s1 == '\0' || n == 1)
        return (*s1 - *s2);
    else
        return (r_strncmp(++s1, ++s2, --n));
}
```

这个函数先着眼于由 $s1$ 和 $s2$ 指向的两个串的第一个字符。如果两个字符不同，或它们都是空字符，或者 $n$ 的值是1，那么返回值就是两个字符的差；否则就对两个串指针都增量，对 $n$ 减量，对函数进行递归调用。递归可能在两个串的第一个不同之处终止，或在两个字符为空处终止，还可能在进行了第 $n-1$ 次递归后终止。

## 11.4 分而治之方法

通常在分而治之算法中使用递归。这一算法把大问题划分成小问题，直接地解决小问题，或使用递归解决小问题，再把对各个部分的解决组合成对整个问题的解决。

让我们用分而治之方法找一个整数数组中的最大数和最小数。在1972年，作为作者之一，Ira Pohl发表了“A Sorting Problem and Its Complexity”（《Communications of ACM》，15,no.6），该篇文章中的算法对解决我们的问题来讲可能是最好的。“最好”的准则是所需的比较次数为最少。为了简化起见，这里仅处理数组中的元素个数是2的次幂的情况。在练习11中，我们要继续讨论这个算法，对其进行修改，使其不受2的次幂这个限制。

```
/* best possible minmax algorithm - Pohl, 1972 */
/* size of the array a is n; it must be a power of 2. */
/* code can be rewritten to remove this restriction. */

void minmax(int a[], int n, int *min_ptr, int *max_ptr)
{
    int min1, max1, min2, max2;

    if (n == 2)
        if (a[0] < a[1]) {
            *min_ptr = a[0];
            *max_ptr = a[1];
        }
        else {
            *min_ptr = a[1];
            *max_ptr = a[0];
        }

    else {
        minmax(a, n/2, &min1, &max1);
        minmax(a + n/2, n/2, &min2, &max2);
        if (min1 < min2)
            *min_ptr = min1;
        else
            *min_ptr = min2;
        if (max1 < max2)
            *max_ptr = max2;
        else
            *max_ptr = max1;
    }
}
```

对函数minmax()的解析

```

• if (n == 2)
    if (a[0] < a[1]) {
        *min_ptr = a[0];
        *max_ptr = a[1];
    }
    else {
        *min_ptr = a[1];
        *max_ptr = a[0];
    }

```

这是基本情况。把两个元素 $a[0]$ 和 $a[1]$ 的较小者赋给由 $\text{min\_ptr}$ 指向的值，把较大者赋给由 $\text{max\_ptr}$ 指向的值。

```

• else {
    minmax(a, n/2, &min1, &max1);
    minmax(a + n/2, n/2, &min2, &max2);

```

这是分而治之的步骤。把数组 $a$ 分为两部分。第一个调用寻找在元素 $a[0], \dots, a[n/2 - 1]$ 中的最小值和最大值。第二个调用找在元素 $a[n/2], \dots, a[n - 1]$ 中的最小值和最大的值。注意， $a$ 是一个值为 $\&a[0]$ 的指针表达式， $a + n/2$ 是一个值为 $\&a[n/2]$ 的指针表达式。

```

• if (min1 < min2)
    *min_ptr = min1;
else
    *min_ptr = min2;

```

比较两部分中的最小值，把较小者赋给由 $\text{min\_ptr}$ 指向的对象。

```

• if (max1 < max2)
    *max_ptr = max2;
else
    *max_ptr = max1;

```

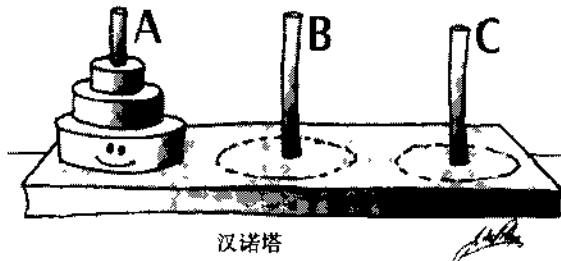
类似地，比较两部分中的最大值，把较大者赋值给由 $\text{max\_ptr}$ 指向的对象。 ■

这个算法有理论上和实践上的含意。在练习10中要进一步对其进行讨论。练习11将去掉对数组 $a[]$ 的尺寸为2的次幂的限制。

很多排序算法使用分而治之技术。其中非常重要的一个是快速排序算法，关于这个算法，请参看由Al Kelley和Ira Pohl所著的《A Book on C》(Reading, Mass.: Addison-Wesley, 1998)。

### 11.5 例子：汉诺塔

在名为汉诺塔的游戏，有3个标号为A、B和C的塔。一开始在塔A上有 $n$ 个圆盘。为了简化，假设 $n$ 是3。把圆盘标号为1、2和3。为了不失一般性，我们假设每个圆盘的直径与其标号相同，即圆盘1的直径是1（按某种测量单位），圆盘2的直径是2，圆盘3的直径是3。按1、2和3的次序把3个圆盘放在A塔上。游戏的目标是把A塔上的3个圆盘都移到C塔上。一次只允许移动一个圆盘。可以把圆盘临时放在塔A、B和C上。在每次移动后，每个塔上的圆盘必须遵守一定的次序，即大的圆盘不能放在比它小的圆盘的上面。



我们用递归编写一个程序，它产生一个反映如何完成把 $n$ 个圆盘从塔A移到塔C的移动过程的列表。此外，还要显示每一步中在每个塔上的圆盘情况。下面是程序的头文件。

文件hanoi.h的内容：

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

extern int    cnt;                                /* # of moves */

enum no_yes {no, yes};

// The towers will be tagged with 'A', 'B', and 'C'.
typedef char   tower_tag;
typedef int    disk;
typedef int *  tower;
typedef const char cchr;
typedef enum no_yes no_yes;

void    chk(tower a, tower b, tower c, int sz);
void    error_exit_cannot_get_here(cchr* place);
void    error_exit_wrong_order(tower a, int sz);
int     get_n_from_user(void);
no_yes  is_out_of_order(tower a, int sz);

void    move(int n, tower a, tower b, tower c, int sz);
int     ndisks(tower a, int sz);
int     top_index(tower a, int sz);
void    wrt(tower a, tower b, tower c, disk d, int sz);
void    wrt_tower(tower a, int sz);
```

全局变量cnt用于保留移动数。注意，我们用typedef定义了tower\_tag、disk和tower等类型，以方便使用。类型为tower的变量是指向int的指针，在分配空间后，可以把塔处理为数组。

文件main.c中内容：

```
#include "hanoi.h"

int    cnt = 0;                                /* # of moves */

int main(void)
{
    int    i;
    int    n;
    int    sz;
    tower  a, b, c;

    /* number of disks */
    /* size of the arrays */
    /* the towers of Hanoi */

    n = get_n_from_user();
    assert(n > 0);
    sz = n + 1;

    // Create space for towers a, b, and c.
    // Each tower has space for a tag and n disks.
    a = calloc(sz, sizeof(int));
    b = calloc(sz, sizeof(int));
    c = calloc(sz, sizeof(int));

    // Tag towers a, b, and c with 'A', 'B', and 'C', respectively.
    a[0] = 'A';
    b[0] = 'B';
    c[0] = 'C';

    // Start with n disks on tower a.
    for (i = 1; i < sz; ++i)
        a[i] = i;
    // Write the starting picture on the screen.
```

```

wrt(a, b, c, 0, sz);

// Move n disks from tower A to tower C,
// using tower B as an intermediate tower.
move(n, a, b, c, sz);          /* recur*/
return 0;
}

```

main()中, 用户先输入n。假设n的值是3。数组a[]、b[]和c[]的尺寸是4, a[]含有'A'、1、2和3; 由于calloc()把所有的元素赋值为0, b[]含有'B'、0、0和0; c[]含有'C'、0、0和0。至此函数调用move(n, a, b, c, sz)中的所有参数都准备完毕。我们把该调用读作“从塔A到塔C移动n个圆盘, 用塔B作为中间塔”, 我们也可以把它看作是“通过塔B把n个圆盘从塔A移动到塔C”。我们要详细地解释函数move()是怎样工作的, 但现在先使用它。如果用户在提示后输入4, 那么程序的输出如下:

TOWERS OF HANOI:

There are three towers: A, B, and C.

The disks on tower A must be moved to tower C. Only one disk can be moved at a time, and the order on each tower must be preserved at each step. Any of the towers A, B, or C can be used for intermediate placement of a disk.

The problem starts with n disks on Tower A.

Input n: 4

```

0: Start:           1 2 3 4
1: Move disk 1 from A to B: 2 3 4      1
2: Move disk 2 from A to C:   3 4      1 2
3: Move disk 1 from B to C:   3 4      1 2
4: Move disk 3 from A to B:   4        3 1 2
5: Move disk 1 from C to A:   1 4        3 2
6: Move disk 2 from C to B:   1 4      2 3
7: Move disk 1 from A to B:   4      1 2 3
8: Move disk 4 from A to C:       1 2 3      4
9: Move disk 1 from B to C:       2 3      1 4
10: Move disk 2 from B to A:      2        3 1 4
11: Move disk 1 from C to A:      1 2      3 4
12: Move disk 3 from B to C:      1 2      3 4
13: Move disk 1 from A to B:      2        1 3 4
14: Move disk 2 from A to C:      1        2 3 4
15: Move disk 1 from B to C:      1 2 3 4

```

为了节省空间, 把每个塔上的圆盘从左至右地进行输出, 而没有垂直地输出。从这个输出中直接可以看出, 大圆盘下面都没有比它小的圆盘。注意输出的对称性, 这表明后一半的移动能从前一半的移动中演绎出来。如果存在对称性, 有时用它就可以改善程序的执行时间, 幸运的是汉诺塔问题不关心这点。

让我们看一下本程序中的其他函数。

文件get.c中的内容:

```

#include "hanoi.h"

int get_n_from_user(void)
{
    int n;

    printf("%s",
        "TOWERS OF HANOI:\n\n"
        "There are three towers: A, B, and C.\n\n"
        "The disks on tower A must be moved to tower C. Only one\n"
        "disk can be moved at a time, and the order on each tower\n");
}

```



```

        "must be preserved at each step. Any of the towers A, B,\n"
        "or C can be used for intermediate placement of a disk.\n"
        "\n"
        "The problem starts with n disks on Tower A.\n\n"
        "Input n: ");
if (scanf("%d", &n) != 1 || n < 1) {
    printf("\nERROR: Positive integer not found - bye!\n\n");
    exit(1);
}
printf("\n");
return n;
}

```

文件move.c中的内容:

```

/*
// Move n disks from tower a to tower c,
// using tower b for temporary storage.//
// CAUTION:
// The tags for the towers a, b, and c
// need not be A, B, and C, respectively.
*/

#include "hanoi.h"

void move(int n, tower a, tower b, tower c, int sz)
{
    int i;
    int j;
    disk d;

    chk(a, b, c, sz);
    if (n == 1) {
        ++cnt;
        i = top_index(a, sz);
        j = top_index(c, sz);
        assert(i >= 1 && i <= sz - 1);
        assert(j >= 2 && j <= sz);
        d = a[i];
        a[i] = 0;
        c[j - 1] = d;
        wrt(a, b, c, d, sz);
    }
    else {
        move(n - 1, a, c, b, sz);
        move(1, a, b, c, sz);
        move(n - 1, b, a, c, sz);
    }
}

```

对递归函数move()的解析

- void move(int n, tower a, tower b, tower c, int sz)

这是函数move()的函数定义的头。回想一下,在头文件中我们创建了与“指向int的指针”同义的类型tower,因而我们可以把塔a、b和c看成是int型的数组。在main()中,我们先调用move(),塔a、b和c的标记分别是'A'、'B'、和'C'。但是要注意,当move()在它的函数体中调用自己时,不需要保持这种关系。

- disk d;

在头文件中,我们已用typedef机制使得类型disk与类型int同义。这里,我们要把d作为塔上的圆盘。

- chk(a, b, c, sz)

这个函数在程序构造阶段为程序员提供帮助。它检查在每个塔上的圆盘次序,即不允许

大的圆盘在比它小的圆盘上面。如果出现了这样的错误，就在屏幕上显示一个信息，然后程序终止。

```
• if (n == 1) {
    ++cnt;
    i = top_index(a, sz);
    j = top_index(c, sz);
    assert(i >= 1 && i <= sz - 1);
    assert(j >= 2 && j <= sz);
}
```

如果移动圆盘数是1，那么我们对计数器加1，寻找a[]和c[]的顶部标引，用assert()保证标引是在所希望的范围内。假设sz的值是5，并且a[]和c[]的值如下：

```
in a[]: 'C'  0  0  1  2
in c[]: 'A'  0  0  0  4
```

要找到顶部标引，我们忽略标记，找数组中的第一个非零元素的标引（记住要从0开始）。这样a[]的顶部标引是3，c[]的顶部标引是4。用这些标引或下标值去移动圆盘。

```
• d = a[i];
  a[i] = 0;
  c[j - 1] = d;
  wrt(a, b, c, d, sz);
}
```

我们把塔a[]中的顶部圆盘的值赋给d，通过把0赋给a[i]把圆盘从塔a[]上移走。通过把d赋值给c[j - 1]把圆盘放在塔c[]上，并且在屏幕上显示一行，以说明我们所做的移动。

```
• else {
    move(n - 1, a, c, b, sz);
    move(1, a, b, c, sz);
    move(n - 1, b, a, c, sz);
}
```

这是算法的递归部分。如果n大于1，我们就先通过c从a到b移动n - 1个圆盘，然后通过b从a到c移动1个圆盘，最终通过a把当前已经移到b上的n - 1个圆盘移动到c。反复进行这样的3种移动的效果是通过b从a到c移动了n个圆盘。 ■

把移动一摞圆盘归结到反复地移动一个圆盘。只要我们得到能正确地移动一个圆盘的基本情况，递归就关心其余的部分。

下面是其余的函数。

文件error.c中的内容：

```
#include "hanoi.h"

void error_exit_cannot_get_here(cchr* place)
{
    printf("%s%s\n",
        "----\n",
        "PROGRAMMER ERROR: Cannot get here:\n",
        "\n",
        "  In ", place, ":\n",
        "\n",
        "Bye!\n");
    exit(1);
}

void error_exit_wrong_order(tower a, int sz)
{
    printf("%s%c%s\n",
        "----\n",
        "ERROR: Tower ", a[0], " is out of order:\n");
    wrt_tower(a, sz);
}
```

```
    printf("\n\n");
    exit(1);
}
```

文件fct.c中的内容:

```
#include "hanoi.h"

void chk(tower a, tower b, tower c, int sz)
{
    int n1 = ndisks(a, sz);
    int n2 = ndisks(b, sz);
    int n3 = ndisks(c, sz);
    assert(n1 >= 0 && n2 >= 0 && n3 >= 0);
    assert(n1 + n2 + n3 == sz - 1);
    if (is_out_of_order(a, sz) == yes)
        error_exit_wrong_order(a, sz);
    if (is_out_of_order(b, sz) == yes)
        error_exit_wrong_order(b, sz);
    if (is_out_of_order(c, sz) == yes)
        error_exit_wrong_order(c, sz);
}

no_yes is_out_of_order(tower a, int sz)
{
    int i;

    for (i = top_index(a, sz); i < sz - 1; ++i) {
        if (a[i] >= a[i + 1])
            return yes;
    }
    return no;
}

int ndisks(tower a, int sz)
{
    return (sz - top_index(a, sz));
}

int top_index(tower a, int sz)
{
    int i;

    for (i = 1; i < sz && a[i] == 0; ++i)
        ;
    return i;
}
```

文件wrt.c中的内容:

```
#include "hanoi.h"

void wrt(tower a, tower b, tower c, disk d, int sz)
{
    cchr* fct_name = "wrt()";
    int fw = 26; /* field width */
    int i;
    tower_tag t; /* 'A', 'B', or 'C' */

    if (cnt == 0)
        printf("%5d%-*s", cnt, fw, ": Start:");
    else
        printf("%5d%-*s%-*s%-*s%-*s", cnt,
            ": Move disk ", d, " from ", a[0], " to ", c[0], ":");

    // CAUTION:
    // Towers a, b, c need not be named A, B, C, respectively.
    // Find A and print it, then find B and print it, .....

    for (t = 'A'; t <= 'C'; ++t) {
```

```

        if (a[0] == t)
            wrt_tower(a, sz);
        else if (b[0] == t)
            wrt_tower(b, sz);
        else if (c[0] == t)
            wrt_tower(c, sz);
        else
            error_exit_cannot_get_here(fct_name);
    }
    putchar('\n');
}

void wrt_tower(tower a, int sz)
{
    int    fw;                /* field width */
    int    i;

    if (a[0] == 'C')
        printf(" ");
    for (i = 1; i < sz; ++i) {
        fw = (a[i] < 10) ? 2 : 3;
        if (a[i] == 0)
            printf("%*s", fw, "");
        else
            printf("%*d", fw, a[i]);
    }
}

```

在wrt()中,我们使用了调用printf("%5d%-\*s",cnt,fw,":Start:").对格式%-\*s需要进行解释。减号要对域进行左调整,\*意味着应该从相应的参数表中得到整数值。在本例中\*从fw中得到它的值和域宽。有关printf()的细节,请参见第13章“输入/输出和文件”。

## 11.6 风格

对于大多数常用的递归都有简单、等价的迭代程序。通过取消记录不同标引的局部变量,递归简化了编码;迭代经常是更有效的解决问题的方法。究竟使用哪一种,凭你的经验选择。

让我们编写一个计算一个数组的平均值的递归程序。

```

double average(double a[], int n) /* n is size of a[] */
{
    if (n == 1)
        return a[0];
    else
        return ((a[n - 1] + (n - 1) * average(a, n - 1))/n);
}

```

在这样的情况下,递归是初步的,使用while或for循环可把它简单地转换成迭代形式。下面是函数average()的迭代形式:

```

double average(double a[], int n) /* n is size of a[] */
{
    double    sum = 0.0;
    int       i;

    for (i = 0; i < n; ++i)
        sum += a[i];
    return (sum / n);
}

```

在这种情况下,该函数的迭代形式是较为简单的。它也避免了 $n-1$ 次函数调用,这里的 $n$ 是被求平均数的数组的尺寸。如果用迭代和用递归编写一个函数都很简单,就倾向于用迭代,这是一种常用的编程风格,但,很多算法通常都是用递归编写的。这样的一个是“快速排序”,另一个是最大公约数算法(请参见练习8)。

## 11.7 常见的编程错误

使用递归函数最常犯的编程错误是导致死循环。我们用阶乘的递归定义来说明几种常见的错误。

对于一个非负的整数 $n$ ，把 $n$ 的阶乘写为 $n!$ ，它的定义为：

对于 $n > 0$ ， $0! = 1$ ， $n! = n(n-1) \cdots 3 \cdot 2 \cdot 1$

或为：

对于 $n > 0$ ， $0! = 1$ ， $n! = n((n-1)!)!$

例如， $5 \times 4 \times 3 \times 2 \times 1 = 120$ 。用阶乘的递归定义编写递归的阶乘函数是很容易的。

```
int factorial(int n)    /* recursive version */
{
    if (n <= 1)
        return 1;
    else
        return (n * factorial(n - 1));
}
```

这段代码是正确的，在给定系统所允许整数精度的范围内它能正常地工作。然而，由于 $n!$ 增长得很快，只对有限的 $n$ ，函数调用`factorial(n)`能产生的有效结果。在我们的系统中，函数调用`factorial(12)`返回的值是正确的，但如果函数的参数大于12，返回的值就是错误的。这种类型的编程错误是常见的。如果函数体中的逻辑操作超过了系统允许的整数精度，那么尽管它在逻辑上是正确的，也会返回错误的结果。

如果程序员忽略了基本情况而错误地编写了阶乘函数，就会导致死循环。

```
long factorial_forever(long n)
{
    return (n * factorial_forever(n - 1));
}
```

假设考虑了基本情况，但仅考虑 $n$ 的值为1的情况。在参数值是在1到12这个范围内的情况下，这个函数能正常地工作，但是如果用值为0或负数的参数调用函数，就会产生死循环。

```
long factorial_positive(long n)
{
    if (n == 1)
        return 1;
    else
        return (n * factorial_positive(n - 1));
}
```

另一种不怎么与递归相关的常见的错误是对减量运算符的错用。在很多递归中，把变量作为参数传递给递归步中函数，假设变量是 $n$ ，它逐渐减小。对于一些算法，`--n`是正确的参数，而对于另一些算法则是不正确的。请考虑如下的代码：

```
long factorial_decrement(long n)
{
    if (n <= 1)
        return 1;
    else
        return (n * factorial_decrement(--n));
}
```

在第二个`return`语句中，我们使用了表达式

```
n * factorial_decrement(--n)
```

它使用了变量n两次。由于减量运算符有副作用，第二个n可能影响第一个n的值。这种类型的编程错误是常见的，特别是对递归函数编码。

## 11.8 系统考虑

对递归函数的每次调用都需要内存空间。由于很多调用的活动都是同时进行的，操作系统可能会耗尽可用的内存。与大系统相比，明显地小系统更容易出问题。当然，如果你要编写一个要在很多系统上运行的程序，你必须要知道并遵从各系统在这方面的限制。

让我们编写一个程序，它显示它在Borland C系统上运行时的递归深度。深度随系统不同的而不同，并与所用的递归函数相关。不过我们的实验是要给出我们所用的机器在这方面的一些限制。我们使用11.1节“递归问题求解”中给出的递归函数sum()，并用long型整数对它进行修改。修改的目的是避免在处理过大的n时产生溢出问题。函数调用sum(n)要递归n次，因此n就是递归深度。

```
/* Test the depth of recursion for sum() */
#include <stdio.h>

long sum(long);
int main(void)
{
    long n = 0;

    for ( ; ; n += 100)
        printf("recursion test: n = %ld sum = %ld\n",
               n, sum(n));
    return 0;
}

long sum(long n)
{
    if (n <= 1)
        return n;
    else
        return (n + sum(n - 1));
}
```

**警告** 这个程序会产生灾难性的后果，你需要重新启动机器。下面是该程序显示的最后几行：

```
.....
recursion test: n = 7900 sum = 31208950
recursion test: n = 8000 sum = 32004000
recursion test: n = %ld sum = %ld
```

程序在此处失败，没有把控制返回给操作系统。这表明对sum()调用超过8000次后，系统失败。在该系统达到极限之前，它允许很深的递归。注意，本质上并不是每秒调用的次数引起失败，而是超出了8000次的递归深度引起了问题。

## 11.9 转向C++

递归在C++中与C中没有什么不同。然而，C++库经常有扩展精度包。类型big\_int是一个无界的精度整数包，像factorial()这样的递归可以用它返回一个正确的、任意大的阶乘计算值，当然系统对此也有一定限制。

很多C++库有一个有序的数组类型，可以用它说明二分查找的效率。虽然下面的程序与在C中的表达有所不同，但它也说明了分而治之算法的重要性，使用了现成的递归算法。

```
//Binary lookup in a sorted integer array
int bin_lookup(const sorted& keys, int size, int d)
{
    int index ;
    if ( size == 0) //Failure
        return - 1;
    index = size - 1 / 2;
    if (keys[index] == d)
        return index;
    else if (keys[index] < d)
        return bin_lookup(keys, size- 1/2, d)
    else
        return bin_lookup(&keys[index + 1], size - 1/2, d)
}
```

函数bin\_lookup()以对数级的时间工作。每次递归都处理剩余的一半。结果是含有值d的元素的标引，如果没有找到这样的元素，则返回的值是-1。

## 小结

- 如果函数直接或间接地调用自己，就说它是递归的。递归是控制流的高级形式。
- 通常递归由一个或几个基本情况以及一般情况组成。确保函数终止是很重要的。
- 任何递归函数都能写成等价的迭代形式。由于系统过度地调用函数，递归函数可能比等价的迭代函数的效率低，但差别经常是很小的。在递归函数比等价的迭代函数较容易编写和维护且损失的效率也很小的的情况下，使用递归形式较好。

## 练习

1. 下面的程序先输出BLAST OFF，请解释它的行为。

```
#include <stdio.h>

void count_down(int n)
{
    if (n) {
        count_down(n - 1);
        printf("%d ! ", n);
    }
    else
        printf("\nBLAST OFF\n");
}

int main(void)
{
    count_down(10);
    return 0;
}
```

2. 编写一个测试一个串是否为回文的递归函数。回文是正读和反读都一样的串，例如"abcba"和"otto"就是回文。编写一个程序测试你的函数。再用迭代编写一个程序，与递归程序进行比较。

3. 考虑下面的递归函数。在一些系统上，该函数返回正确的结果，而在另一些系统上返回的结果不正确。请解释这是为什么。编写一个测试程序，看在你的系统上会发生什么情况。

```
int sum(int n)
{
    if (n <= 1)
        return n;
    else
        return (n + sum(--n));
}
```

4. 仔细地检查递归函数 `r_strcmp()` 的基本情况。下面的基本情况能工作吗？哪一个更有效？请给出解释。

```
if (*s1 != *s2 || *s1 == '\0' || *s2 == '\0' || n == 1)
    return (*s1 - *s2);
```

编写标准库函数 `strcmp()` 的递归版本。如果 `s1` 和 `s2` 是串，函数调用 `r_strcmp(s1, s2)` 应该返回一个整数值，按字典顺序根据 `s1` 是小于、等于还是大于 `s2`，该整数值是负的、0 或正的。使用下面的 `main()` 测试你的函数。编译你的程序，并把可执行的代码放入文件 `test`，如果文件 `infile` 中含有很多词，那么用命令 `test < infile` 测试你的程序。

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

#define MAXWORD 30 /* max characters in a word */
#define N 50 /* number of words in array */

int r_strcmp(char *, char *);

int main(void)
{
    int i, j;
    char word[N][MAXWORD], /* an array of N words */
        temp[MAXWORD];

    for (i = 0; i < N; ++i)
        scanf("%s", word[i]);
    for (i = 0; i < N - 1; ++i)
        for (j = i + 1; j < N; ++j)
            if (r_strcmp(word[i], word[j]) > 0) {
                strcpy(temp, word[i]);
                strcpy(word[i], word[j]);
                strcpy(word[j], temp);
            }
    for (i = 0; i < N; ++i)
        printf("%s\n", word[i]);
    return 0;
}
```

5. 编写标准库函数 `strcpy()` 的递归版本。如果 `s1` 和 `s2` 是串，函数调用 `r_strcpy(s1, s2)` 应该用 `s2` 的内容重写 `s1`，并返回第一个参数 `s1` 的指针值。请参见附录A“标准库”中对 `strcpy()` 的描述。

6. 尽管把对常量串进行重写的作法认为是不良的编程习惯，我们在本练习中这样做是为了强调一个要点。考虑如下的两个语句：

```
printf("%s\n", strcpy("try this", "and this"));
printf("%s\n", strcpy("and", "also this"));
```

其中的一个语句工作正常，而另一个会出问题。请解释为什么。重新编写上一个练习中的函数 `r_strcpy()`，如果出现了禁用的错误类型，那么就显示错误信息，程序停止运行。即使函数是递归的，也要保证仅对错误检查一次。还要注意，一些系统不允许对常量串进行重写。提示：使用静态变量。

7. 一个函数调用另一个函数，而第二个函数又调用第一个函数，把这种调用称为相互递归。注意相互递归函数成对出现。编写一个对一个串中的字母进行计数并对串中的数字进行求和的程序。例如，串 `"Aois444apple7"` 有8个字母，数字和是19。编写两个相互递归函数，完成此项工作。用 `count_alph()` 对字母计数，用 `sum_digit()` 对数字求和。这两个函数应该相互递归。编写一个能完成此项工作的更自然的非相互递归函数，与使用相互递归函数



的情况进行比较。提示：如果需要，使用静态变量。

8. 两个正整数的最大公约数是两个正整数的公约数中的最大的那一个。例如，6和15的最大公约数是3，15和22最大公约数是1。下面的递归函数计算两个正整数的最大公约数。首先编写一个测试该函数的程序，然后用迭代编写一个等价的函数，并进行测试。

```
int gcd(int a, int b)
{
    int r;

    if ((r = a % b) == 0)
        return b;
    else
        return gcd(b, r);
}
```

9. 编写一个递归查找函数，用于在一个已经排序的数组中搜寻。编写一个程序测试你的函数。用random()填写尺寸为n的数组a[]，用bubble\_sort()对a[]排序。用函数调用look\_up(v, a, n)在数组a[]中查找变量v。如果在数组中仅有一个元素的值是v，假设是a[i]，那么返回的值是i。如果在数组中有两个或多个元素的值是v，那么返回其中哪个标引都可以。如果在数组中没有值是v的元素，那么返回的值是-1。基本情况是查找“中间”的元素a[n/2]。如果这个元素的值是v，那么就返回n/2；否则就对下面合适的部分数组递归地查找v：a[0], ..., a[n/2 - 1]或a[n/2 + 1], ..., a[n - 1]。例如，如果v的值小于a[n/2]，那么应该在数组的前半部分查找v，递归的形式是

```
return(look_up(v, a, n/2));
```

10. 可以用下面的代码测试11.4节“分而治之方法”中的递归函数minmax()。n的值是交互式地输入的；用函数calloc()动态地创建尺寸为n的数组a[]；用随机分布的整数值填写数组a[]；用递归调用ninmax()和迭代这两种方法计算数组的最大值和最小值。因为所有的这些都在循环的内部的发生，所以在循环结束后我们用标准库函数free()把由a指向的存储空间返还给系统。

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

void minmax(int *, int, int *, int *);

int main(void)
{
    int *a, i,
        n,
        r_min, r_max, /* n is the size of a[] */
        i_min, i_max; /* recursive min and max */
                    /* iterative min and max */

    printf("Input a power of 2: ");
    while (scanf("%d", &n) == 1 && n > 0) {
        a = calloc(n, sizeof(int));
        for (i = 0; i < n; ++i)
            a[i] = rand();
        minmax(a, n, &r_min, &r_max);
        printf("\n%s%d%s%d\n",
            "recursion: min = ", r_min, "max = ", r_max);
        i_min = i_max = a[0];

        for (i = 1; i < n; ++i) {
            i_min = (i_min < a[i]) ? i_min : a[i];
            i_max = (i_max > a[i]) ? i_max : a[i];
        }
    }
}
```

```

        printf("%s%d%s%d\n",
               "iteration: min = ", i_min, "max = ", i_max);
        free(a);
        printf("Input a power of 2: ");
    }
    return 0;
}

```

$n$ 的值是数组 $a[]$ 的尺寸。每个 $n$ 是2的次幂，以递归的和迭代的方式计算数组的最大值和最小值。在这样的计算中各用了多少次比较？提示：在 $n$ 的值（2的次幂）不大的情况下用手工模拟，找出答案。

11. 递归算法 $\text{minmax}()$ 在比较次数为最少方面的优势一点也不明显。在11.4节“分而治之方法”中引用的Ira Pohl所发表论文已经证明：不存在更好的算法。在本练习中我们要表明如何修改该算法，以使得它能处理任何尺寸的数组。修改后的算法在效率上可能不如以前，但它仍然还是很有效的。

```

void minmax(int a[], int n, /* n is a the size of a[] */
            int *min_ptr, int *max_ptr)
{
    int min1, max1, min2, max2;

    if (n == 1)
        *min_ptr = *max_ptr = a[0];
    else {
        minmax(a, n/2, &min1, &max1);
        minmax(a + n/2, n - n/2, &min2, &max2);
        if (min1 < min2)
            *min_ptr = min1;
        else
            *min_ptr = min2;
        if (max1 < max2)
            *max_ptr = max2;
        else
            *max_ptr = max1;
    }
}

```

编写一个程序测试该函数。注意，它仍然是分而治之算法，并能正确地处理任何尺寸的数组。现在 $n$ 的值为1是一个基本情况。在11.4节“分而治之方法”中给出的可能是最好的算法中， $n$ 的值为2是一个基本情况。当然，2是2的幂，1也是2的幂。用 $n$ 的值为1作为基本情况对原 $\text{minmax}()$ 进行修改后，算法还能是最好的吗？或许用 $n$ 的值为2作为基本情况强调算法的“2的性质”。毕竟若 $n$ 的值为2，算法也不复杂。还有一些小的细节需要考虑。在本练习中使用参数 $n - n/2$ ，而在原算法中使 $n/2$ ，请解释为什么。

12. 用汉诺塔程序进行实验。如果在塔A上有 $n$ 个圆盘，需要移动多少次才能完成游戏？如果 $n$ 是64，一天移动一个圆盘，需要多少年才能把全部的圆盘移到C上？

13. (高级) 不用递归编写汉诺塔程序。

14. (高级) 在国际象棋中马是一个走L步的棋子。棋盘有64个格；如果在棋盘的角上马可以有两种合法的走法，如果在棋盘的中间马可以有八种合法的走法。编写一个计算马在一个具体的格上能有多少种合法的走法的程序。合法的走法的数目与格数有关。这被称作是马的格连通性。编写一个程序，计算并显示马在各个格上合法的走法的数目。对应棋盘的64个格，显示出的数字格式应该是 $8 \times 8$ 阵列，在每一个位置上的数字表示马的格连通性。这个阵列就是在棋盘上的马的格连通性。

15. (高级，请参见Ira Pohl于1967年发表在《Communications of the ACM》第7期上的文

章“A Method for Finding Hamiltonian Paths and Knight's Tours”)。马的漫游是马可走的要覆盖64个格但不能重复任何格的路径。Warnsdorf规则指出,要找马的漫游,要从一个角开始,走到还没有到达的格,并且连通性要最小。邻近格是马能立即走到的格。在格被访问后,邻近格的所有连通数都要减少。用Warnsdorf规则找出马的漫游。显示一个与棋盘对应的 $8 \times 8$ 阵列,在每一个位置上显示马走到该格的移动数目。

16. (高级)Pohl改进了Warnsdorf规则,建议打破递归联系(请参见上一个练习)。Warnsdorf规则是启发性的。不能保证它能工作,但用它解决组合的难题还是很有有效的。有时两个格有相同的最小连通性。为了打破这种联系,递归地计算能有最小的连通性的那个格,并选择该格。在一般的 $8 \times 8$ 棋盘上,从任何格开始,Warnsdorf-Pohl规则都能工作。完成这个启发算法,从不同的格开始执行它5次,显示各次的漫游。

17. C++: 编写一个计算斐波纳契(Fibonacci)数的递归函数,斐波纳契数的定义如下:

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

进行斐波纳契递归很快会引起机器的栈溢出。比较使用迭代与使用递归进行计算所需的运行时间。如果可能,编写big\_int产生前100个斐波纳契数。

18. Java: 如果一个方法直接或间接地调用自己,就说它是递归的。最简单的递归是直接递归。用C重新编写下面的Java程序。

```
// Recur.java - recursive goodbye
//Java by Dissection page 124.
public class Recur {
    public static void main(String[] args) {
        sayGoodBye(5);
    }

    static void sayGoodBye(int n) {
        if (n < 1) //base case
            System.out.println("#####");
        else {
            System.out.println("Say goodbye Gracie.");
            sayGoodBye(n - 1); //recursion
        }
    }
}
```

## 第12章 结构和抽象数据类型

结构类型允许程序员把一些分量聚合成一个有名字的变量。一个结构的各个分量都有名字，把这些分量称为成员(member)。由于结构的成员可以是各种类型的，程序员能创建适合于问题的数据聚合。像数组和指针一样，结构是一种派生类型。

在本章中，我们要说明怎样声明结构，以及怎样用它们表达各种熟悉的例子（如纸牌和学生记录）。处理结构的关键点是用成员运算符，或结构指针运算符  $\rightarrow$  访问其成员。这些运算符以及  $()$  和  $[]$  都有最高的优先级。在介绍这些运算符后，我们要提供一张关于C的所有运算符的优先级和结合性的表。

本章给出了很多说明如何处理结构的例子。用一个处理学生记录的系统作为例子说明结构的用法及如何对其成员进行访问。本章还解释用于创建链表的自我引用结构的用法，并提出了处理链表的编码需求。这些用户定义的纸牌、学生记录和链表都是用户定义类型或抽象数据类型(ADT)的例子。

### 12.1 声明结构

结构是把类型可能不同的数据项聚合成一个集合的手段。作为一个简单的例子，让我们定义一个描述纸牌的结构。一张牌上的点描述了纸牌的数值，把这样数值称为纸牌的牌点(pips)。例如，如果一张纸牌上有3个黑桃，那么就说这张牌的牌点是3，花色(suit)是黑桃。我们用结构把这样的变量组织在一起。我们可以声明结构类型

```
struct card {
    int    pips;
    char   suit;
};
```

用它来描述所需要的纸牌信息。在这个声明中struct是关键字，card是结构的标记名，变量pips和suit是结构的成员。pips的取值范围是从1到13，分别表示从A到K的纸牌，suit可取的值是'c'、'd'、'h'和's'，分别表示梅花、方块、红桃和黑桃。

这个声明创建了一个派生的数据类型struct card。可以把这个声明看作是一个蓝图；它创建了类型struct card，但没有为它分配存储空间。现在可以用标记名声明该类型的变量。声明

```
struct card c1, c2;
```

为类型是struct card的标识符c1和c2分配存储空间。要访问c1和c2的成员，我们要使用结构成员运算符“.”。假设我们用描述一张有5个方块的纸牌的信息向c1赋值，用描述一张黑桃Q的纸牌的信息向c2赋值，我们就可以写：

```
c1.pips = 5;
c1.suit = 'd';
c2.pips = 12;
c2.suit = 's';
```

可把形式为

*structure\_variable . member\_name*

的结构用作变量，其用法与简单变量或数组元素一样。在所声明的结构中成员名必须是惟一的。由于总是用惟一的结构变量标识符作为成员的前导，或通过结构变量标识符访问成员，所以在不同的结构中可以有相同的成员名，而不会发生混淆。下面是一个例子：

```
struct fruit {
    char   name[15];
    int    calories;
};
struct vegetable {
    char   name[15];
    int    calories;
};

struct fruit    a;
struct vegetable b;
```

在做了这些声明后，我们可以无歧义地访问a.calories和b.calories。

在一个单声明中，可以同时创建结构类型并声明该类型的变量，示例如下：

```
struct card {
    int    pips;
    char   suit;
} c, deck[52];
```

标识符card是类型标记名。把标识符c声明为struct card型的变量，把标识符deck声明为struct card型的数组。下面是另一个例子：

```
struct {
    char   *last_name;
    int    student_id;
    char   grade;
} s1, s2, s3;
```

它声明了s1、s2和s3，用这3个变量代表3个学生记录，但该声明没有标记名。假设我们把它写为：

```
struct student {
    char   *last_name;
    int    student_id;
    char   grade;
};
```

现在的这个声明与前边的声明不一样，它用student作为结构标记名，但没有为该类型声明变量。可以把它看作是一个蓝图。现在我们写

```
struct student temp, class[100];
```

把temp和class声明为struct student型的，仅在这时才为变量分配了存储空间。结构类型声明本身不会引起存储空间的分配。

## 12.2 访问成员

我们已经看到了成员运算符. 的用法。在本节中我们要给出其他的例子，以说明该运算符的用法，并介绍结构指针运算符->。

假设我们正在编写一个称为class\_info的程序，它生成一个关于有100个学生的班级的信息。我们从创建一个头文件开始。注意，我们使用的名字cl\_info.h遵从了MS-DOS关于文件名长度的限制。在没有这种限制的系统中，使用class\_info.h可能会更有助于记忆。

```
#define CLASS_SIZE 100

struct student {
    char *last_name;
    int student_id;
    char grade;
};
```

现在组成程序的模块用这个头文件共享信息。假设我们在另一个文件中写

```
#include "cl_info.h"

int main(void)
{
    struct student temp, class[CLASS_SIZE];
    .....
```

我们用下面的语句对结构变量的成员temp赋值:

```
temp.grade = 'A';
temp.last_name = "Bushker";
temp.student_id = 590017;
```

现在假设我们要对一个给定的班级中不及格的学生计数。我们编写一个访问grade成员的函数来完成此项工作。下面的函数fail()对数组class[]中成绩为F的学生进行计数。

```
/* Count the failing grades. */
#include "cl_info.h"

int fail(struct student class[])
{
    int i, cnt = 0;

    for (i = 0; i < CLASS_SIZE; ++i)
        cnt += class[i].grade == 'F';
    return cnt;
}
```

### 对函数fail()的解析

```
• int fail(struct student class[])
{
    int i, cnt = 0;
```

参数class的类型是指向struct student的指针,我们可以把它看成是结构型的一维数组。任何类型(当然包括结构类型)的参数都可以用在函数定义的头中。

```
• for (i = 0; i < CLASS_SIZE; ++i)
```

我们正在假设在调用函数时类型为struct student的数组及其尺寸被作为参数传递。

```
• cnt += class[i].grade == 'F';
```

这样的表达式说明了C是如何的简练。C中的运算符是丰富的;为了在使用时不出现问题,程序员必须注意运算符的优先级和结合性。这个语句与下面的语句等价:

```
cnt += (((class[i]).grade) == 'F');
```

选择结构数组class的第i个元素(从0开始计数)的成员grade,并做一个测试,看是否等于'F'。如果等于,那么表达式

```
class[i].grade == 'F'
```

的值是1;如果不等于,表达式的值为0,且cnt的值保持不变。该表达式的一个更清晰但较冗长的等价表达形式是:

```
if (class[i].grade == 'F')
    ++cnt;
```

```
• return cnt;
```

把不及格的人数返回到调用环境。

C提供了结构指针运算符 $\rightarrow$ ，用它可以访问结构的成员。用一个减号后跟一个大于号表示该运算符。如果用一个结构的地址对指针变量赋值，那么就可以用下面的形式访问结构的成员：

```
pointer_to_structure -> member_name
```

一个等价的形式是：

```
(*pointer_to_structure).member_name
```

这里的括号是必须的。运算符 $\rightarrow$ 和“.”，以及()和[]都有最高的优先级和自左到右的结合性。由于这个原因，没有圆括号的结构与下面的表达式是等价的：

```
*(pointer_to_structure.member_name)
```

在复杂的情形中可能以复杂的方式结合这两种访问模式。下表以直观的方式说明了它们的使用。

声明和赋值		
<pre>struct student temp, *p = &amp;temp; temp.grade = 'A'; temp.last_name = "Bushker"; temp.student_id = 590017</pre>		
表 达 式	等价表达式	概念值
temp.grade	p -> grade	A
temp.last_name	p -> last_name	Bushker
temp.student_id	p -> student_id	590017
(*p).student_id	p -> student_id	590017

## 12.3 运算符的优先级和结合性：总结

我们现在要展示C的所有运算符的全部优先级和结合性。在本章中已经介绍了运算符 $\rightarrow$ 和“.”。运算符 $\rightarrow$ 和“.”以及()和[]都有最高的优先级。

在C中逗号运算符的优先级最低。在声明中和在函数的参数表中的逗号不是逗号运算符。

像我们在第6章“基本数据类型”中看到的那样，可以用一元运算符sizeof确定在内存中存储一个对象所需要的字节数。例如，表达式sizeof(struct card)的值是系统存储一个类型为struct card的变量所需要的字节数。在大多数系统中，表达式的类型是unsigned。在本章的后续部分，我们在创建链表时会看到sizeof运算符的扩展用法。

虽然整个运算符表的内容很多，但可以使用一些简单的规则。最重要的运算符是函数的圆括号、下标的方括号和两个访问结构成员的地址运算符，这四个运算符具有最高的优先级。接下来的是一元运算符，再下面的是算术运算符。算术运算符遵从常规，也即乘法运算符有比加法运算符高的优先级。各种赋值的优先级都很低，仅高于逗号运算符。如果程序员在某

种情形下不知道优先级和结合性规则，就要去查找相应的规则或使用圆括号。

运算符					结合性
()	[]	.	->	++ (后缀)    -- (后缀)	从左到右
++ (前缀)	-- (前缀)	!	~	sizeof (类型)	从右到左
+(一元)	-(一元)	& (地址)		*(间接引用)	
	*	/	%		从左到右
	+	-			从左到右
	<<	>>			从左到右
	<	<=	>	>=	从左到右
	==	!=			从左到右
	&				从左到右
	^				从左到右
					从左到右
	&&				从左到右
					从左到右
	?:				从右到左
=	+=	-=	*=	/=	从右到左
	%=	>>=	<<=	&=	
	^=	=			从右到左
	,	(逗号运算符)			

## 12.4 结构、函数和赋值

传统的C系统允许把指向结构类型的指针作为参数传递给函数，也可作为返回值。ANSI C允许把结构本身作为参数传递给函数，也可作为返回值。在这个环境中，如果a和b是同一结构的变量，那么a=b是允许的，该表达式把b的各个成员的值赋给相应的a的各个成员（关于进一步的评论，请参见12.15节“系统考虑”）。

为了说明结构在函数中的用法，我们使用结构类型struct card。在本章的其余部分，假设头文件card.h含有本结构的声明。

```
struct card {
    int    pips;
    char   suit;
};
```

让我们编写一个函数，它向card赋值，提取card的成员值，并显示card的值。我们假设在所需要之处已经引入了头文件card.h。

```
void assign_values(struct card *c_ptr, int p, char s)
{
    c_ptr -> pips = p;
    c_ptr -> suit = s;
}

void extract_values(struct card *c_ptr, int *p_ptr, char *s_ptr)
{
    *p_ptr = c_ptr -> pips;
    *s_ptr = c_ptr -> suit;
}
```

这些函数用指向一个类型为struct card的变量的指针访问card。在程序中用结构指针运算符->访问需要的成员。然后，让我们编写一个显示纸牌的例程，它使用一个指向



struct card的指针，并用extract\_values()显示它的值。

```
void prn_values(struct card *c_ptr)
{
    int    p;           /* pips value */
    char   s;           /* suit value */
    char   *suit_name;

    extract_values(c_ptr, &p, &s);
    switch (s) {
    case 'c':
        suit_name = "clubs";
        break;
    case 'd':
        suit_name = "diamonds";
        break;
    case 'h':
        suit_name = "hearts";
        break;
    case 's':
        suit_name = "spades";
        break;
    default:
        suit_name = "error";
    }
    printf("card:  %d of %s\n", p, suit_name);
}
```

最后，我们说明怎样使用这些函数。首先，我们向一副纸牌赋值，然后作为监测显示出红桃。

```
void assign_values(struct card *, int, char );
void prn_values(struct card *);
void extract_values(struct card *, int *, char *);

int main(void)
{
    int    i;
    struct card  deck[52];

    for (i = 0; i < 52; ++i) {
        assign_values(deck + i, i + 1, 'c');
        assign_values(deck + i + 13, i + 1, 'd');
        assign_values(deck + i + 26, i + 1, 'h');
        assign_values(deck + i + 39, i + 1, 's');
    }
    for (i = 0; i < 13; ++i) /* print out the hearts */
        prn_values(deck + i + 26);
    return 0;
}
```

注意，这段代码怎样用地址运算进行赋值，以及怎样显示各种花色的值。用下面的语句显示花色为红桃的牌：

```
prn_values(deck + i + 26);
```

它等价于

```
prn_values(&deck[i + 26]);
```

函数使用结构而不是指针作为参数来完成赋值工作。让我们重写函数extract\_values()。

```
void extract_values(struct card  c,
                    int *p_ptr, char *s_ptr)
{
    *s_ptr = c.suit;
    *p_ptr = c.pips;
}
```

在C中，进行函数调用时，要复制传递给函数的参数的值。在4.12节“调用和按值调用”

中和第8章“函数、指针和存储类型”中讨论了这种按值调用的机制。由于这种原因，当把结构作为参数传递给函数时，函数调用时要复制结构。因此，传递结构的地址比传递结构本身更有效。

## 12.5 问题求解：学生记录

在C中，可以把结构、数组和指针进行组合，以创建复杂的数据结构。在很大程度上对问题的解决是要为处理的信息选定适当的数据结构。通常像学生这样的现实世界中的对象最好是用一组特征来描述。在C中，struct是一种用于数据聚合的封装机制。聚合后，在概念上把对象处理为学生，而不是一组具体的特征。

我们从12.2节“访问成员”中的struct student开始，把它完善成一个更容易理解的用于学生记录的数据结构。我们从定义这个数据类型开始。

```
#define CLASS_SIZE 50
#define NCOURSES 10 /* number of courses */

struct student {
    char *last_name;
    int student_id;
    char grade;
};

struct date {
    short day;
    char month[10];
    short year;
};

struct personal {
    char name[20];
    struct date birthday;
};

struct student_data {
    struct personal p;
    int student_id;
    char grade[NCOURSES];
};
```

注意，用嵌套的结构构造结构student\_data，其成员之一是结构p，p的成员之一是结构birthday。在声明

```
struct student_data temp;
```

后，表达式

```
temp.p.birthday.month[0]
```

把temp中的学生的birthday的month的第一个字母作为它的值。像data和personal这样的结构经常用在数据库应用中。

让我们编写一个函数read\_date()，它把数据输入到一个类型为struct date的变量中。在调用函数时，必须把变量的地址作为参数传递给函数。

```
#include "student.h"

void read_date(struct date *d)
{
    printf("Enter day(int) month(string) year(int): ");
    scanf("%hd%s%hd", &d -> day, d -> month, &d -> year);
}
```

### 对函数read\_date()的解析

```
• void read_date(struct date *d)
```

```
{
    printf("Enter day(int) month(string) year(int): ");
```

参数d的类型是指向struct date的指针。printf()语句用于向用户提示信息。

```
• &d -> day
```

这是一个地址。由于&的优先级小于->, 所以这个表达式等价于

```
&(d -> day)
```

首先用指针d访问成员day, 然后对成员day应用地址运算符&, 以获得它的地址。

```
• d -> month
```

这是一个地址。用指针d访问本身是一个数组的成员。数组名本身是一个指针或地址; 它指向数组的基地址。

```
• scanf("%hd%s%hd", &d -> day, d -> month, &d -> year);
```

用函数scanf()读入3个值, 并存放在适当的地方。回想一下, 在头文件student.h中, struct date的两个成员day和year被声明为short型的。用格式%hd把标准输入流中的字符转换成类型为short的值。(在格式中修饰转换字符d的h来自于词short的第二个字母。) ■

可以用函数read\_date()把信息读入类型为struct student\_data的变量。例如, 可以用代码

```
struct student_data temp;
read_date(&temp.p.birthday);
```

把信息放到temp的适当成员中。

下面是输入成绩的函数:

```
void read_grades(char g[])
{
    int i;

    printf("Enter %d grades: ", NCOURSES);
    for (i = 0; i < NCOURSES; ++i)
        scanf(" %c", &g[i]);
}
```

用控制"%c"一个非空白字符。%前的空格与输入流中的空白字符相匹配。可以按如下形式调用这个函数:

```
read_grades(temp.grade);
```

把一系列成绩读入到temp。参数temp.grade是一个地址, 因为它引用结构的一个自身是数组的成员, 而且数组名本身就是数组的基地址。因此, 调用函数会在调用环境中改变temp.grade的值。

基本上, 理解了结构就理解了怎样访问它的成员。作为更深入的例子, 现在让我们编写一个函数, 它把在struct student\_data中以长型存储的数据转变成在struct student中以短型存储的数据。

```
#include "student.h"
```

```
void extract( struct student_data *s_data,
              int n,
              struct student *undergrad)
```

```

{
    undergrad -> student_id = s_data -> student_id;
    undergrad -> last_name = s_data -> p.name;
    undergrad -> grade = s_data -> grade[n];
}

```

## 12.6 结构的初始化

系统把没被显式地初始化的所有外部和静态变量（包括结构变量）都自动地初始化为0。在传统C中，程序员可以对外部和静态结构进行初始化。在ANSI C中，我们也可以初始化自动的结构，其语法与初始化数组类似。结构变量的后面可以跟一个等号=和用括号括起来的常量表。如果没有足够的值用以给结构的成员赋值，那么缺省就用0对剩余的成员赋值，下面是一些例子：

```

struct card   c = {12, 's'}; /* the queen of spades */

struct fruit  frt = {"plum", 150};

struct complex {
    double  real;
    double  imaginary;
} m[3][3] = {
    {{1.0, -0.5}, {2.5, 1.0}, {0.7, 0.7}},
    {{7.0, -6.5}, {-0.5, 1.0}, {45.7, 8.0}},
}; /* m[2][] is assigned zeroes */

```

## 12.7 typedef的用法

在实践中，经常用typedef机制对结构类型重新命名。在12.9节“线性链表”中我们会看到这一点。

让我们定义几个处理向量和矩阵的函数来说明typedef的用法。

```

#define   N   3 /* size of all vectors and matrices */

typedef   double   scalar;
typedef   scalar   vector[N];
typedef   scalar   matrix[N][N];

```

我们已经用typedef机制创建了类型scalar、vector和matrix，这在文档和概念上是合适的。以自然的方式已经对我们的编程语言进行了扩展，以把这些类型和领域相结合。注意怎样用typedef建造类型的层次。例如，我们可以用

```
typedef   vector   maxtrix[N];
```

代替

```
typedef   scalar   maxtrix[N][N];
```

使用typedef创建scalar、vector和matrix这样的类型名，使得程序员能用应用领域中的术语进行思考。现在我们准备创建一个函数，它提供的操作超出了我们的领域的需要。

```

void add(vector x, vector y, vector z) /* x = y + z */
{
    int i;

    for (i = 0; i < N; ++i)
        x[i] = y[i] + z[i];
}

```

```

scalar dot_product(vector x, vector y)
{
    int    i;
    scalar sum = 0.0;

    for (i = 0; i < N; ++i)
        sum += x[i] * y[i];
    return sum;
}

void multiply(matrix a, matrix b, matrix c)    /* a=b*c */
{
    int    i, j, k;

    for (i = 0; i < N; ++i)
        for (j = 0; j < N; ++j) {
            a[i][j] = 0.0;
            for (k = 0; k < N; ++k)
                a[i][j] += b[i][k] * c[k][j];
        }
}

```

## 12.8 自引用结构

在本节中，我们用指针成员定义结构，且指针成员引用包含其自身的结构类型，把这样的结构称为自引用(self-referential)结构。自引用结构经常需要存储管理例程来显式地获取或释放内存。

让我们用一个成员域定义一个结构，该成员域指向相同的结构类型。我们这样做是希望把一些数目未定的结构链接在一起。

```

struct list {
    int    data;
    struct list *next;
};

```

类型为struct list的每个变量都有两个成员，即data和next。指针变量next被称为链(link)。用成员next把结构一个接一个地链接起来。可用表示成箭头的链把这样结构绘制出来，如下图所示：



指针变量next含有后继的struct list元素的内存地址或特定的值NULL，通常在stdio.h中把NULL定义为符号常量0，用值NULL指明链的结尾。为了理解这些都是如何工作的，让我们从如下声明开始：

```
struct list a, b, c
```

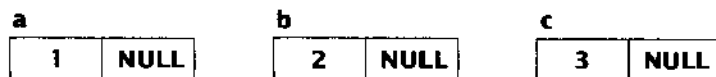
我们要操纵结构变量a、b和c，以创建一个链表。先对这些结构进行赋值：

```

a.data = 1;
b.data = 2;
c.data = 3;
a.next = b.next = c.next = NULL;

```

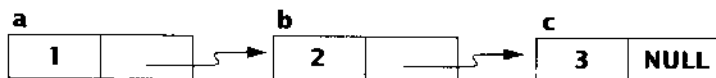
以图形的方式把这段代码的结果描述如下：



下面把这3个结构链接在一起。

```
a.next = &b;
b.next = &c;
```

用这两个语句就把a连接到b，b连接到c。



现在，我们从相继的元素中检索数据。例如，`a.next-> data`的值是2，`a.next->next-> data`的值是3。

## 12.9 线性链表

线性链表就像把数据结构连续地悬挂在其上的晒衣绳。有一个头指针指向链表的第一个元素的地址，表中的每个元素都有一个指向后继元素的指针，最后一个元素的指针值是NULL。通常要动态地创建链表，在本节中我们要说明怎样做这件事。

让我们从创建一个头文件开始，随后我们要编写的一个链表处理函数使用它。该头文件引入了文件`stdio.h`，这是因为在该文件中定义了NULL。

文件`list.h`的内容：

```
#include <stdio.h>

typedef char DATA; /* use char in examples */

struct linked_list {
    DATA d;
    struct linked_list *next;
};

typedef struct linked_list ELEMENT;
typedef ELEMENT * LINK;
```

在头文件`list.h`中，我们使用`typedef`创建较复杂的类型的名字。注意，虽然`DATA`是简单的数据类型`char`，但在概念上它是较复杂的类型，数组和结构也是这样（请参见练习11）。

### 动态存储分配

在`list.h`中的声明`struct linked_list`时并没有为其分配存储空间。只有在声明该类型的变量时，系统才为变量分配存储空间，所以说声明的结构只起到存储蓝图的作用。我们用`typedef`机制把该类型命名为`ELEMENT`，是因为我们希望把结构作为链表上的元素。像`malloc()`这样的标准库提供的用以动态地分配存储空间的工具函数，使得像`ELEMENT`这样的自引用类型特别有用。如果`head`是一个`LINK`型变量，那么

```
head = malloc(sizeof(ELEMENT));
```

就从系统中获得一块用以存储`ELEMENT`的内存空间，并把其基地址赋值给指针`head`。像上一个例子那样，调用`malloc()`的函数经常用`sizeof`运算符，该运算符用于计算存储特定对象所需要的字节数。

假设我们要动态地创建一个线性链表，用以存储字符`n`、`e`和`w`，下面的代码能完成此项

工作：

```
head = malloc(sizeof(ELEMENT));
head -> d = 'n';
head -> next = NULL;
```

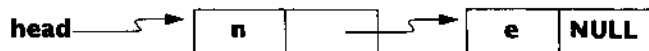
这几条语句创建了如下链表：



用下述语句添加第2个元素：

```
head -> next = malloc(sizeof(ELEMENT));
head -> next -> d = 'e';
head -> next -> next = NULL;
```

下面是含有两个元素的链表：



最后我们用下述的语句添加第3个元素：

```
head -> next -> next = malloc(sizeof(ELEMENT));
head -> next -> next -> d = 'w';
head -> next -> next -> next = NULL;
```

现在已建立了一个由head指向的有3个元素的链表，该链表用标记值NULL结尾。



## 12.10 对链表的操作

对线性链表的基本操作包括以下内容：

### 基本的链表操作

- 创建一个链表。
- 对元素计数。
- 查找元素。
- 插入元素。
- 删除元素。

我们要说明对链表的这些操作进行编程的技术。因为链表是一种递归定义的结构，所以自然要使用递归函数。每一个例程都需要在头文件list.h中说明。观察一下，在这些例子中的d被重定义为较复杂的数据结构。

在第一个例子中，我们编写了一个由串生成链表的函数。函数返回一个指向结果链表的指针。该函数的核心是通过分配存储空间并向成员赋值来创建链表元素。

```
/* List creation by recursion. */

#include "list.h"

LINK string_to_list(char s[])
{
    LINK head;

    if (s[0] == '\0') /* base case */
        return NULL;
```

```

    else {
        head = malloc(sizeof(ELEMENT));
        head -> d = s[0];
        head -> next = string_to_list(s + 1);
        return head;
    }
}

```

### 对函数string\_to\_list()的解析

```

• LINK string_to_list(char s[])
{
    LINK    head;

```

把串作为参数传递时，就创建一个字符链表。由于返回的是指向链表头的指针，所以在函数定义的头中的类型标识符是LINK。

```

• if (s[0] == '\0')    /* base case */
    return NULL;

```

当检测到串结束标志时，接返回NULL，像我们看到的那样，递归结束。NULL也被用于标记链表的结束。

```

• else {
    head = malloc(sizeof(ELEMENT));

```

如果串s[]不是空串，那么用malloc()分配足够数量的存储空间存储类型为ELEMENT的对象。指针变量head现在指向由malloc()提供的存储块。

```

• head -> d = s[0];

```

用串s[]中的第一个字符向已分配的类型为ELEMENT的结构的成员d赋值。

```

• head -> next = string_to_list(s + 1);

```

指针表达式s+1指向该串的其余部分。用s+1作参数递归地调用本函数。用string\_to\_list(s+1)返回的指针值对指针成员next赋值。这个递归调用返回作为LINK，或等价地作为指向ELEMENT的指针，它指向剩余的子链表。

```

• return head;

```

函数退出时返回链表头的地址。 ■

注意，递归是怎样处理像空链表的创建这样的基本情况和像子链表的创建这样的一般情况的。

## 12.11 计数和查找

本节中我们要编写两个对链表进行操作的递归程序。第一个是count()，它用来对链表中的元素计数。从头至尾对链表进行递归，直到发现NULL为止。如果链表是空的，那么返回值0；否则返回链表中的元素个数。

```

/* Count a list recursively. */

#include "list.h"

int count(LINK head)
{
    if (head == NULL)
        return 0;
    else
        return (1 + count(head -> next));
}

```



第二个函数用于在链表中搜索指定的元素。如果找到了指定的元素，那么返回的是指向该元素的指针；否则返回NULL指针。

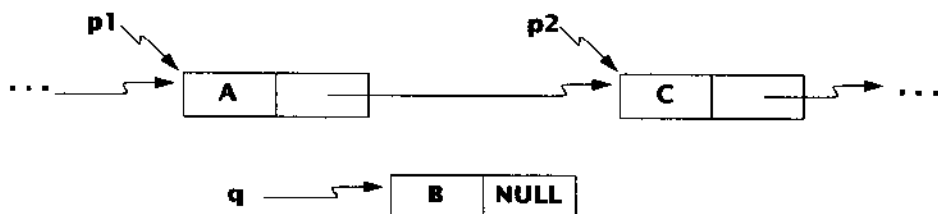
```
/* Lookup c in the list pointed to by head. */
#include "list.h"
LINK lookup(DATA c, LINK head)
{
    if (head == NULL)
        return NULL;
    else if (c == head -> d)
        return head;
    else
        return (lookup(c, head -> next));
}
```

## 12.12 插入和删除

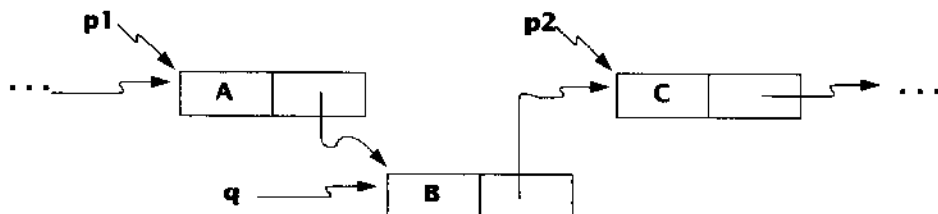
链表的最有用的性质之一是在链表的各个位置上插入元素所需的时间是固定的。对比而言，如果在一个大数组中插入一个值，并按原次序保留其他的数组值，平均的插入时间与数组的长度成正比。新插入值后面的数组的所有元素的值都必须移动一个位置。

通过让p1和p2指向两个相邻元素和在它们中间插入一个由q指向的元素来说明对表的插入。

插入前



插入后

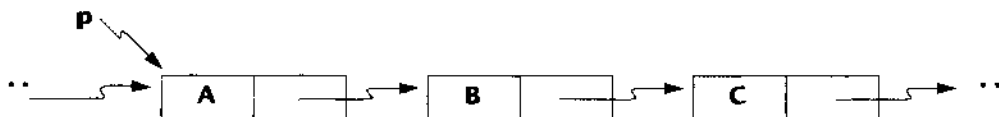


下面是完成此项工作的函数：

```
/* Inserting an element in a linked list. */
#include "list.h"
void insert(LINK p1, LINK p2, LINK q)
{
    p1 -> next = q; /* insertion */
    q -> next = p2;
}
```

在线性链表中删除元素与此类似。要被删除的元素的前驱让链接成员被赋予了指向被删

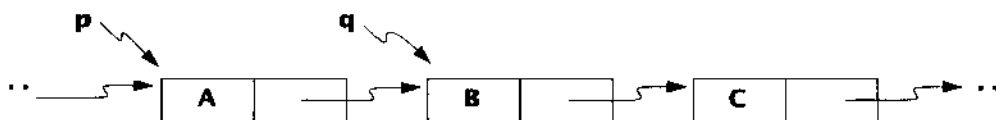
除的元素的后继的地址。让我们先说明删除前的情况。



下行代码

```
q = p -> next;
```

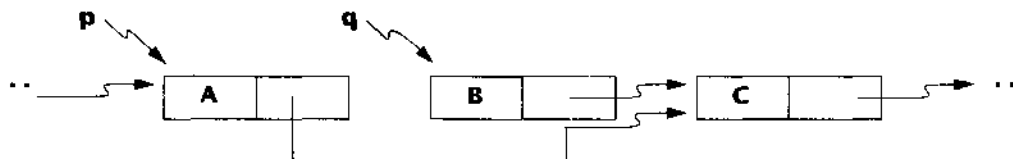
使得q指向我们要删除的元素。



现在考虑语句

```
p -> next = q -> next;
```

在执行该语句后，情况如下：



像图中所显示的那样，包含B的元素就不再可以访问了，即不能再使用它了。把这样的不能被访问的元素称为垃圾(garbage)。因为内存常常是关键性资源，所以要把这种存储区归还给系统，以备再用。可以用标准库函数free()来完成此工作。用free()编写一个把动态地分配给链表的存储返回给系统的删除例程。

```
/* Recursive deletion of a list. */
#include "list.h"

void delete_list(LINK head)
{
    if (head != NULL) {
        delete_list(head -> next);
        free(head); /* release storage */
    }
}
```

### 12.13 风格

把相关的数据聚合成结构是一种良好的编程风格。通过声明一个结构，程序员可以创建一个适合于问题的数据结构。声明应该按一定的意图在每一行上列出各个成员。

```
struct automobile {
    char    name[15];    /* example: buick */
    int     year;        /* example: 1983 */
    double  cost;        /* example: 2390.95 */
};
```

一个结构类型拥有一个标记名通常是一种良好的编程作法，这有利于进一步地进行声明和产生文档。

如果进一步的工作要使用结构声明，用typedef机制为类型创建一个新的名字是有益的。对于struct automobile，我们可以使用

```
typedef struct automobile CAR;
```

新的类型名不必大写，也可以使用Car或car。在一些风格中，只有预处理器标识符才完全大写。如何选择凭你的爱好。

把结构声明放在要引入的头文件中是一种常见的编程风格。如果在以后需要改变结构类型的声明，那么只在头文件中改变即可。或许随后发现我们要在CAR中加入描述汽车马力的成员。这可通过在头文件的结构声明中加入像

```
int horsepower; /* example: 225*/
```

这样的行来完成，可见这样做是很容易的。

## 12.14 常见的编程错误

在使用自引用结构时，一种常见的编程错误是访问错内存位置。例如，如果没有用NULL正确地终止一个线性链表，就会出现一些不可预料的运行错误。

另一种常见的编程错误是在使用typedef时把位置颠倒。例如，

```
typedef ELEMENT struct linked_list; /* wrong*/
```

是不正确的，这是因为标识符ELEMENT必须跟在struct linked\_list的后面，而不是出现在struct linked\_list的前面。还要注意，typedef构造用分号结尾。

还有一种编程错误是对具有相同结构类型的变量进行比较。假设这样的变量是a和b，虽然赋值表达式

```
a = b
```

是合法的，但用表达式

```
a == b /*wrong*/
```

测试两个结构相等则是不允许的。由于运算符=和==看起来很相似使得编程新手有时会犯这样的错误。

## 12.15 系统考虑

C随着时间在演化着。传统C系统允许把指向结构的指针作为参数传递给函数，还允许返回这样的值。ANSI C编译器也允许把指向结构的指针作为参数传递给函数，还允许返回这样的值。此外，这些编译器还允许对结构进行赋值。如果程序员编写的程序要向下兼容，就必须遵守老编译器可能有的限制。

传统的C不允许对存储类型为自动的结构变量进行初始化。要保持与老系统的兼容性，就要仅对静态和外部结构变量进行初始化。

在数据库应用中，结构经常有数十甚至数百个成员。如果把结构作为参数传递给函数，在调用函数时就要对结构进行复制。如果结构很大，并且确实不需要本地复制，那么由于效率的缘故最好传递指向结构的指针，而不传递结构本身。

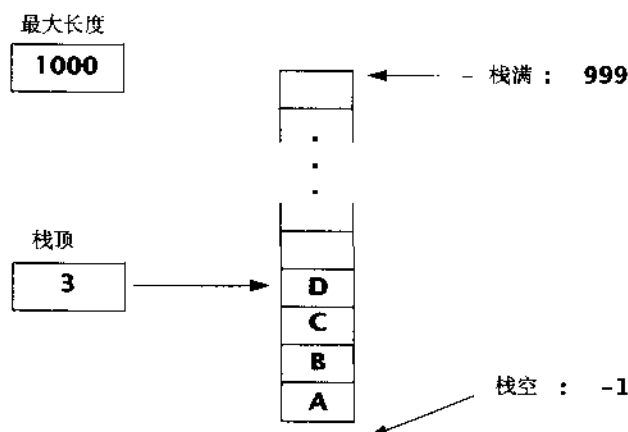
不同操作系统对存储区定位边界的要求是不同的。例如，一些操作系统可能要求int定位在一个字的边界，而有的操作系统不是这样。由于这个原因，在为结构分配存储空间时，

可能会发生作为整体结构所需要的空间要大于其各个成员所需空间之和的情况。此外，结构所需的存储空间可能与其内部的成员的排列次序有关（请参见练习18）。只有在缺乏内存资源的情况下，程序员才关心这种情况。

## 12.16 转向C++

在C++中，把struct这个概念参数化，允许函数作为成员。在结构声明中含有函数声明，用访问结构成员的方法调用函数。在struct声明中应该直接引入struct数据结构所要求的函数性。

我们用对栈ADT的编码来说明这些思想。栈(stack)是最有用的标准数据结构之一。仅允许在栈顶一次插入或删除一个元素。这就是后进先出规则。在概念上，它的行为像一摞托盘，当把托盘移走或放入时，托盘就弹出或叠加。下面是一个已经压入了ABCD的栈：



通常，栈允许的操作有压入(push)、弹出(pop)、栈顶(top)、栈空(empty)和栈满(full)。压入操作在栈中放入一个元素；弹出操作从栈中检索一个值，并从中删除；栈顶操作返回栈的顶部值；栈空操作测试栈是否为空；栈满操作测试栈是否已满。栈是一种典型的ADT。

我们希望以C的形式使用struct实现一个栈，作为C++的数据类型。实现的方法是用固定长度的字符数组存储栈的内容。栈顶是一个名为top的具有整数值的成员。用函数实现这种栈操作，每一个函数的参数表都包括指向stack的指针。这样就允许对栈进行修改，从而避免按值调用时的栈复制。

```
//A C-like implementation of type stack.
```

```
const int max_len = 1000;
enum boolean { false, true };
enum { EMPTY = -1, FULL = max_len - 1 };
```

```
struct stack {
    char s[max_len];
    int top;
};
```

现在我们要编写一组函数，以实现对栈的标准操作。

```
//A standard set of stack operations.
```

```
void reset(stack* stk)
{
    stk -> top = EMPTY;
```

```

}

void push(char c, stack* stk)
{
    stk -> s[++stk -> top] = c;
}

char pop(stack* stk)
{
    return (stk -> s[stk -> top--]);
}

char top(stack* stk)
{
    return (stk -> s[stk -> top]);
}

boolean empty(const stack* stk)
{
    return (boolean(stk -> top == EMPTY));
}

boolean full(const stack* stk)
{
    return (boolean(stk -> top == FULL));
}

```

### 对函数stack的解析

```

• const int max_len = 1000;
  enum boolean { false, true };
  enum { EMPTY = -1, FULL = max_len - 1 };

  struct stack {
      char s[max_len];
      int top;
  };

```

我们声明了一个新的类型boolean，在C++中，enum类型的标记名是一种新类型。用0对常量false初始化，用1对常量true初始化。struct声明创建了新类型stack。stack有两个成员，分别是数组成员s和int型成员top。

```

• void reset(stack* stk)
  {
      stk -> top = EMPTY;
  }

```

这个函数用于初始化。用值EMPTY对成员top赋值。这个被初始化了的栈是作为地址被传递的参数。

```

• void push(char c, stack* stk)
  {
      stk -> s[++stk -> top] = c;
  }

• char pop(stack* stk)
  {
      return (stk -> s[stk -> top--]);
  }

```

用一个有两个参数的函数实现压入操作，成员top进行增量，把c的值放在栈顶。这个函数假设栈不是满的。对弹出操作的实现与此类似，函数pop()也假设栈不是空的，pop()返回栈顶的值，对成员top进行减量。

```

• boolean empty(const stack* stk)
  {

```

```

    return (boolean(stk -> top == EMPTY));
}

boolean full(const stack* stk)
{
    return (boolean(stk -> top == FULL));
}

```

这两个函数都返回一个被枚举的boolean类型值。每一个函数都对stack的成员top进行适当的条件测试。例如，在调用push()前，程序员必须测试栈不是满的，以保证push()能正确地工作。这两个函数不修改所指向的栈。因此，我们能把指针参数声明为const。在所有的函数中，栈参数都作为地址被传递，用结构指针运算符->访问成员。 ■

让我们通过把与栈相关的各种函数作为成员进行声明来重新编写本例。

```

struct stack {
    //data representation
    char s[max_len];
    int top;
    enum { EMPTY = -1, FULL = max_len - 1 };

    //operations represented as member functions
    void reset() { top = EMPTY; }
    void push(char c) { top++; s[top] = c; }
    char pop() { return s[top--]; }
    char top_of() { return s[top]; }
    boolean empty() { return (boolean(top == EMPTY)); }
    boolean full() { return (boolean(top == FULL)); }
};

```

所写出的成员函数与其他函数很相似，不同点是它们能使用数据成员名，这样在stack中的成员函数就可以使用top和s。当被调用的函数使用类型为struct的具体对象时，它们就对该对象的指定成员起作用。

下面的例子说明了这些思想。如果对两个类型为stack的变量

```
stack data, operands;
```

做了声明，那么

```
data.reset();
operands.reset();
```

调用成员函数reset，这两个语句把data.top和operands.top设置为EMPTY。如果指向stack的指针

```
stack* ptr_operands = &operands;
```

被声明，那么

```
ptr_operands -> push('A');
```

调用成员push，这会对operands.top进行增量，并把operands.s[top]设置为'A'。观察一下，由于命名冲突，成员top\_of()是改后的名字。

定义在struct中的成员函数是隐式内联的。通常仅在struct中定义短的常用的成员函数，就像本例这样。要在struct外定义成员函数，就要用域区分符。让我们通过把push的定义改变为struct stack中的函数原型来说明这一点。我们用域区分符写出它。在本例中该函数不是隐含内联的。

```

struct stack {
    //data representation

```

```

    char s[max_len];
    int top;
    enum { EMPTY = -1, FULL = max_len - 1 };

    //operations represented as member functions
    void reset() { top = EMPTY; } //implicitly inline
    void push(char c);           //function prototype
    ....
};

void stack::push(char c)          //definition not inline
{
    top++;
    s[top] = c;
}

```

域区分符::允许不同struct类型的成员函数有相同的名字。在这样的情况下，调用哪一个成员函数依赖所操作的对象类型。声明

```
stack s, t, u;
```

分别创建了3个sizeof(stack)字节的栈对象。每一个变量都有自己的数据成员：

```
char s[max_len];
int top;
```

在概念上成员函数是类型的一部分。对于这三个栈变量的任意一个都没有不同的成员函数。

在C++中把struct这个概念参数化，允许函数有public和private成员。在struct内部，关键字private后跟一个冒号的用法限制了遵从本结构的成员的作用域。仅有为数不多种类的函数可以使用private成员，而且这些函数要有访问这些成员的权限。这些函数包括struct的成员函数。其他种类的可以访问这些成员的函数要在15.3节“重载”中讨论。

我们现在修改stack的例子，以隐藏它的数据表示。

```

struct stack {
public:
    void reset() { top = EMPTY; }
    void push(char c) { top++; s[top] = c; }
    char pop() { return s[top--]; }
    char top_of() { return s[top]; }
    boolean empty() { return (boolean)(top == EMPTY); }
    boolean full() { return (boolean)(top == FULL); }
private:
    char s[max_len];
    int top;
    enum { EMPTY = -1, FULL = max_len - 1 };
};

```

我们现在要编写测试这些操作的main()。

```

//Reverse a string with a stack.
int main(void)
{
    stack s;
    char str[40] = {"My name is Don Knuth!"};
    int i = 0;
    cout << str << endl;
    s.reset();           //s.top = EMPTY; would be illegal
    while (str[i])
        if (!s.full())
            s.push(str[i++]);
    while (!s.empty()) //print the reverse
        cout << s.pop();
    cout << endl;
}

```

本测试程序的输出如下：

```
My name is Don Knuth!
!htuK noD si eman yM
```

像在main()的注释中所说的那样，要控制对隐藏变量top的访问。成员函数reset()可以改变它，但是不能直接访问它。还要注意，怎样用结构成员运算符形式把变量s传递给各个成员函数。

struct stack拥有包含数据描述的私有(private)部分和包含执行栈操作的成员函数的公有(public)部分。把私有部分看成是对实现程序使用的限制，把公有部分看成是对客户程序可以使用的接口的描述，这样做是有用的。实现程序能改变私有部分而不影响客户程序对栈类型使用的正确性。

隐藏数据是面向对象编程的一个重要组成部分，它使得对代码的调试和维护更加容易，这是因为错误和修改被局部化。客户程序只需要知道类型的接口描述。

## 类

在C++中用关键字class引入类。类是struct的一种形式，其缺省的私有性描述是private。这样，用适当的属性描述可以交换使用struct和class。

很多科学计算要求复杂的数值。让我们为复杂的数值编写一个ADT。

```
struct complex {
public:
    void assign(double r, double i);
    void print() {cout << real << " + " << imag << "i ";}
private:
    double real, imag;
};

inline void complex::assign(double r, double i = 0.0)
{
    real = r;
    imag = i;
}
```

下面是等价的class描述：

```
class complex {
public:
    void assign(double r, double i);
    void print() {cout << real << " + " << imag << "i ";}
private:
    double real, imag;
};

inline void complex::assign(double r, double i = 0.0)
{
    real = r;
    imag = i;
}
```

也可能是

```
class complex {
    double real, imag;
public:
    void assign(double r, double i);
    void print() {cout << real << " + " << imag << "i ";}
};
```

此处我们认为对class成员的缺省访问是私有的。愿意用class而不愿意用struct是



C++的风格，除非所有的成员都是公有访问的数据成员。

## 小结

- 结构是对作为单个变量的元件的聚合。把结构的元件称为成员。
- 用成员运算符 `.` 和结构指针运算符 `->` 访问结构成员。如果 `s` 是一个拥有名为 `m` 的成员的变量，那么 `s.m` 引用 `s` 中的成员 `m` 的值。如果 `p` 是一个指向 `s` 的指针，那么 `p->m` 和 `s.m` 是等价的。在 C 的运算符中 `.` 和 `->` 的优先级最高。
- 结构可以是其他结构的成员。在结构中嵌套结构、指针和数组时，可能要考虑复杂性。应该注意，访问变量要正确。
- 自引用结构使用指向同样结构类型的指针成员。可以用自引用结构创建线性链表。除了最后一个元素外，每个元素都指向下一个元素。最后一个元素的指针成员的值是 `NULL`。
- 用函数 `malloc()` 动态地分配内存。它需要一个类型为 `size_t` 的参数，并返回一个类型为 `void` 的指针，该指针指向所分配空间的基地址。类型 `size_t` 是无符号整型，在 `stdlib.h` 中有它的定义。
- 用于处理链表的标准算法自然要递归地实现。经常地，基本情况是对 `NULL` 链的检测，在链表中移动一个元素会重现一般情况。

## 练习

1. 假设用下面的结构编写一个食物程序：

```

structure food {
    char   name[15];
    int    portion_weight;
    int    calories;
};

```

该结构的标记名是什么？怎样声明该类型的一个数组 `meal[10]`？假设4盎司 (ounce) 的苹果含有200卡路里 (calorie)，你怎样向 `meal[0]` 的3个成员赋值，以表示这样一个苹果？

2. 编写一个计算给定食物的卡路里的程序。要把食物存储在数组 `meal[ ]` 中。程序要对每道菜进行显示。

3. 创建一个能描述一个餐馆的结构。它的成员应该包括名称、地址、平均价格和食物的种类。假设已经创建了一个描述餐馆的结构数组。编写一个函数，对于给定的食物，按着最小花费显示出所有的餐馆。

4. 下面的函数用于向纸牌赋值，但不能正常工作。描述一下它的问题出在哪儿。

```

#include "card.h"

struct card *assign_values (int    p,      /* pips value */
                           char   s      /* suit value */
)
{
    card   *c_ptr;

    c_ptr -> pips = p;
    c_ptr -> suit = s;
    return c_ptr;
}

```

5. 在玩扑克或其他卡片游戏时，通常要对一手牌进行排列，以反映出该手牌的牌值。

编写一个程序，对5张牌按牌点值排列并显示。假定A的值最高，K的值次之，以此类推。

6. 用12.2节“访问成员”中的学生记录的例子编写一个函数，显示一个班级中的各个学生的平均分。等级A的值是4，等级B的值是3，以此类推。

7. 编写一个函数，按生日从大到小的次序显示各个学生的数据。最初的学生记录集合是无序的。

8. 编写一个函数`prn_student_data()`，按优美的格式显示一个类型为`struct student_data`的变量中的所有信息。

9. 定义一个含有食物名称、每份的卡路里、食物组（如肉食或水果）和价格的结构。编写一个能生成均衡食谱的程序。把食物存储在结构数组中。程序构造的每种食谱都应该含有4组食品，并有卡路里和价格的限制。它应该能生成大量的不同的菜单。

10. 下面的声明在编译时会出错。请解释错在那里，并重新编写它。

```
struct husband {
    char    name[10];
    int     age;
    struct wife spouse;
} a;
struct wife {
    char    name[10];
    int     age;
    struct husband spouse;
} b;
```

11. 修改12.9节的头文件`list.h`，用如下的声明替换`typedef`行：

```
struct data {
    char    name[10];
    int     age;
    int     weight;
};
typedef struct data DATA;
```

编写一个函数`create_list()`，把类型为`DATA`的数组转换成线性链表。编写另一个函数，对超过给定年龄和体重的人计数。

12. 给定一个类型同上一个练习的线性链表，编写一个按年龄对链表排序的函数`sort_by_age()`。编写另一个对名字按字典次序对链表排序的函数`sort_by_name()`。

13. 对在12.10节“对链表的操作”中提到的函数`count()`编写一个迭代版本。

14. 编写一个插入函数，该函数把一个元素插在存储特定`DATA`项的元素之前的第一个位置。你应该用`lookup()`去找这个元素，如果没有找到，就把元素插在链表的尾部。

15. 请解释在下面的结构中为什么需要圆括号：

```
(pointer_to_structure).member_name
```

用这个结构编写一个测试程序。如果把圆括号去掉，你的程序还能通过编译吗？

16. 在简单的情况下，可以用`#define`替换`typedef`，但是有时这会导致不可预料的错误。把头文件`list.h`重新编写如下：

```
#include <stdio.h>

#define DATA char /* use char in examples */

struct linked_list {
    DATA d;
    struct linked_list *next;
};
```

```
#define ELEMENT struct linked_list
#define LINK ELEMENT *
```

在做完这项工作后，检查一下，看函数string\_to\_list()、count()和lookup()是否还像以前那样能通过编译。函数insert()不能通过编译，请解释为什么？对它进行修改，使它能通过编译。

17. 在12.12节“插入和删除”中我们编写了函数insert()，其中假定p1和p2正指向一个链表中的两个相邻的元素。如果p1和p2正指向一个链表中的两个不相邻的元素，会发生什么情况？

18. 在一些系统中，对下述两个结构存储的字节数是不同的：

```
struct s1 {
    char   c1;
    char   c2;
    int    i;
};

struct s2 {
    char   c1;
    int    i;
    char   c2;
};
```

语句

```
printf("%d\n%d\n", sizeof(struct s1),
        sizeof(struct s2));
```

显示的值是与系统相关的。在你的C系统上这两个结构的存储空间的需求是怎样的？在一些系统上，例如Borland C，就存在对字节和字的定界选项。在这样的系统上试一下这两个结构。

19. C++：用线性链表代替数组对栈的示例重新编码。在理论上操作应该保持不变。

20. Java：像C++一样，Java也使用class，而不用C的struct。在Java中，每一个事物都是一个class。由于这个原因，人们认为Java是纯面向对象语言。Java使用无用集，Java程序员不用malloc()和free()。此外，Java没有指针。类型class是可引用类型，用new创建类的实例。下面是一个实现ADT计数器的示例。Counter是这样的一种类型，它从0到99计数，像汽车的里程表一样，计到99后又返回到0。把Java的类转换成相应的C中的struct，成员函数转换成作用于指向struct的指针的函数。

```
// CounterTest.java - demonstration of class Counter
//Java by Dissection pages 200-201
class CounterTest {
    public static void main(String[] args) {
        Counter c1 = new Counter(); //create a Counter
        Counter c2 = new Counter(); //create another
        c1.click(); // increment Counter c1
        c2.click(); // increment Counter c2
        c2.click(); // increment Counter c2 again
        System.out.println("Counter1 value is " +
                           c1.get());
        System.out.println("Counter2 value is " +
                           c2.get());
        c1.reset();
        System.out.println("Counter1 value is " +
                           c1.get());
    }
}

class Counter {
    int value; //0 to 99
    void reset() { value = 0; }
    int get() { return value; } //current value
    void click() { value = (value + 1) % 100; }
}
```

## 第13章 输入/输出和文件

在本章中，我们要解释如何使用标准库中的一些输入/输出函数，其中包括函数 `printf()` 和 `scanf()`，虽然我们已经在本书中使用过这些函数，但是对很多细节仍需要加以解释。我们给出了一些说明各种格式的作用的表。标准库提供了一些与 `printf()` 和 `scanf()` 相关的函数，用这些函数可以处理文件和串；这里要解释它们的用法。

对于处理驻留在磁盘和磁带上的文件中数据的应用来说，一般的文件输入/输出是很重要的。我们要表明怎样打开要处理的文件，以及怎样用指针指向文件。一些应用需要临时文件。本章用一些例子来说明这些用法。

### 13.1 输出函数 `printf()`

`printf()` 函数有两个很好的性质，这使得 `printf()` 具有高度的灵活性。第一，`printf()` 的参数表的长度可以是任意的，第二，用简单的转换说明或格式控制显示。函数 `printf()` 把它的字符流交付给标准输出文件 `stdout()` (通常它与屏幕相连)。`printf()` 的参数表有两部分：

*control\_string* 和 *other\_arguments*

在下例中：

```
printf("she sells %d %s for $%f", 99, "sea shells", 3.77);
```

我们有：

```
control_string:    "she sells %d %s for $%f"
other_arguments:  99, "sea shells", 3.77
```

对 *other\_arguments* 中的表达式求值并按照控制串中的格式进行转换，然后把结果放进输出流。控制串中的不为格式的部分的字符直接被放进输出流。符号 `%` 引入转换说明或格式。单个转换说明用 `%` 开头并用一个转换字符结尾。

printf() 转换字符	
转换字符	怎样显示相应的参数
c	作为字符
d, i	作为十进制整数
u	作为无符号十进制整数
o	作为无符号八进制整数
x, X	作为无符号十六进制整数
e	作为浮点数，例如 7.123000e+00
E	作为浮点数，例如 7.123000E+00
f	作为浮点数，例如 7.123000
g	以 e 格式或 f 格式，都是较短的
G	以 E 格式或 F 格式，都是较短的
s	作为串
p	相应的参数是指向 void 的指针；按十六进制显示它的值
n	相应的参数是指向一个整数的指针，该整数是至今成功写到流或缓冲区的字符个数；对参数不做转换
%	使用 %% 把 % 写入输出流；没有相应的参数被转换

函数printf()返回显示的字符数。在下例中:

```
printf("she sells %d %s for $%f", 99, "sea shells", 3.77);
```

我们可以把控制串中的格式与参数表中相应的参数相匹配。

格 式	相 应 参 数
%d	9 99
%s	" "sea shells"
%f	3 3.77

可以明确地把格式信息引入到转换说明中。如果不这样,就使用缺省格式。例如,格式%f的对应参数是3.77,显示的结果是3.770000。按缺省,显示的十进制值的小数点后保留6位。

在开始进行转换说明的%和终止转换说明的转换字符之间,可能会出现如下内容:

#### 在转换说明中可以出现的内容

- 修改转换说明的零个或多个标志字符。标志字符在下边讨论。
- 说明被转换参数的最小域宽(field width)的正整数,它是可选的。把显示参数的位置称为域(field),把显示参数的格数称为域宽。如果被转换的参数的字符数小于域宽,那么就依赖对被转换参数是左调整还是右调整,在左边或右边填补空格。如果被转换的参数的字符数大于域宽,那么就把域宽调整到所需的大小。如果定义域宽的整数用0开始,并且对被显示的参数在它的域中是右调整的,那么就用0而不是空格进行填充。
- 精度(precision)是可选的,用一个后跟非负整数的句点描述它。对于d, i, o, u, x和X转换,描述了被显示的数字的最小个数。
- e, E和f转换描述了小数点右侧的数字个数。g和G转换描述了有效数字的最大位数。s转换描述了显示一个串的最大字符个数,可选项h或l分别是short或long修饰符。如果h后跟转换字符d, i, o, u, x或X,那么转换描述适用于short int或unsigned short int参数;如果h后跟转换字符n,那么相应的参数是指向short int或unsigned short int的指针。如果l后跟转换字符d, i, o, u, x或X,那么转换描述适用于long int或unsigned long int参数;如果l后跟转换字符n,那么相应的参数是指向long int或unsigned long int的指针。



“孩子,噢,孩子,你注意到自从梅尔文变成双精度后他是多么傲慢了吗?”

- L是可选的，它是一个long修饰符。如果L后跟转换字符e, E, f, g或G，那么转换说明适用于long double参数。

#### 在转换说明中的标志字符

- 减号，其含义是在它的域中被转换的参数是左对齐的。如果没有减号，那么被转换的参数是右对齐的。
- 加号，其含义是，如果输出值是非负的整数且有符号，就在其前加一个+。它与d, i, e, E, f, g和G一起使用。所有的负数都用减号开头。
- 空格，其含义是，如果输出值是非负的整数且有符号的，就在其前加一个空格。它与d, i, e, E, f, g和G一起使用。如果+和空格同时出现，就把空格标记忽略掉。
- #，其含义是依赖转换字符把结果转换成“可选择的形式”。与转换字符o一起使用，#使得在显示的八进制数前加0。在x或X转换中，它使得在显示的十六进制数前加0x或0X。在g或G转换中，它使得尾部的0被显示。在e, E, f, g或G转换中，它使得小数点被显示，甚至精度为0也是如此。其他的转换没有定义这样的行为。
- 0，其含义是用0代替空格来填充域。与d, i, o, u, x, X, e, E, f, g和G转换符一起使用，会导致用0作前导。被显示的数的前任何标记和任何0x或0X都优先于前导0。

在一个格式中，可以用\*代替整数来说明域宽或精度，这表明要从参数表中获取值。下面是如何使用该机制的一个例子：

```
int      m, n;
double   x = 333.7777777;
.....
printf("x = %*. *f\n", m, n, x); /* get m and n from somewhere */
```

如果对应于域宽的参数的值为负，那么就把该参数看作是减号后跟一个正的域宽。如果对应于精度的参数的值为负，那么就将它丢弃。

在输出流中可以用转换描述%%显示单百分号。这是一种特殊情况，因为没有相应的参数被转换。对于所有的其他格式，应该有相应的参数。如果没有足够的参数，那么行为是不定的。如果参数过多，那么对多余的求值而忽略其他。

域宽是用于显示参数的格数。缺省是正确地显示参数的值所需的格数。这样，对于十进制转换d或八进制转换o，整数值255（十进制）需要3个格，但是对于十六进制转换x，则需要2个格。

在显示参数时，把适合于转换说明的字符放在域中。除非有减号作为标记，否则对字符都是右调整的。如果所描述的域宽太短，以致于不适合显示相应变量的值，就把域宽增加到缺省值。如果整个域宽多于要显示的被转换参数的需要，那么就在左边或右边填补空格，这依赖于对被转换的参数是进行左调整还是右调整。通过用前导0描述域宽，可在字符左边填补0。

浮点格式用句点右边非负的数可以指明精度。对于串转换，这是能显示的字符的最大数目。对于e, E, 和f转换，它指定了出现在小数点右边的数字个数。

下表给出了字符和串格式的例子。我们用双引号把字符括起来以界定显示内容，实际上不显示双引号。

声明和初始化			
char c = 'A', s[] = "Blue moon!";			
格 式	相应参数	在其域中的打印结果	说 明
%c	c	"A"	缺省域宽为1
%2c	c	" A"	域宽为2, 右调整
%-3c	c	"A "	域宽为3, 左调整
%s	s	"Blue moon!"	缺省域宽为10
%3s	s	"Blue moon!"	需要更多的格
%.6s	s	"Blue m"	精度6
%-11.8s	s	"Blue moo "	精度8, 左调整

下表给出了用于显示数字的格式的例子。我们也用双引号把字符括起来以界定显示内容, 实际上不显示双引号。

声明和初始化			
int i = 123; double x = 0.123456789;			
格 式	相应参数	在其域中的打印结果	说 明
%d	i	"123"	缺省域宽为3
%05d	i	"00123"	填补0
%7o	i	" 173"	右调整, 八进制
%-9x	i	"7b "	左调整, 十六进制
%-#9x	i	"0x7b "	左调整, 十六进制
%10.5f	x	" 0.12346"	域宽为10, 精度5
%-12.5e	x	"1.23457e-01 "	左调整, e格式

### 13.2 输入函数scanf()

scanf()函数有两个很好的性质, 这使得scanf()具有高度的灵活性。第一, scanf()的参数表的长度可以是任意的; 第二, 是用简单的转换说明或格式控制输入。函数scanf()从标准输入文件stdin中读字符。scanf()的参数表分两部分: control\_string 和 other\_arguments。

在下面的例子中:

```
char    a, b, c, s[100];
int     n;
double  x;

scanf("%c%c%c%d%s%lf", &a, &b, &c, &n, s, &x);
```

我们可以说

```
control_string:    "%c%c%c%d%s%lf"
other_arguments:  &a, &b, &c, &n, s, &x
```

控制串后的参数由用逗号分隔开的指针或地址表达式组成。注意，在本例中，如果写&s，则是一个错误，表达式s本身就是地址。

### 13.2.1 控制串中的指示

scanf()的控制串由3种指示(directive)组成：普通字符、空白字符和转换说明。我们详细地对它们进行讨论。

### 13.2.2 普通字符

把控制串中除空白字符和转换说明中的字符以外的字符称为普通字符(ordinary character)。普通字符必须要与输入流中的字符相匹配。下面是一个例子：

```
float amount;
scanf("$%f", &amount);
```

字符\$是一个普通字符。要在输入流中匹配\$。如果匹配成功，那么就跳过空白（如果有），并且匹配能转换成浮点值的字符。把转换的值放在内存中amount的地址中。

在下一个例子中，三个字符a、b和c中每一个都是普通字符指示。

```
scanf("abc");
```

首先匹配字符a，然后匹配字符b，最后匹配字符c。如果在某处scanf()不能进行匹配，就在输入流中留下非法字符，scanf()返回。如果调用scanf()成功，那么马上就可以对输入流中跟在a，b，c后面的字符准备进行处理。

### 13.2.3 空白字符

在控制串中但不在转换说明中的空白字符可以与输入流的空白字符匹配，也可以不匹配。考虑下面的例子：

```
char c1, c2, c3;
scanf(" %c %c %c", &c1, &c2, &c3);
```

如果输入流含有字母a、b和c，无论它们有无前导空格，也无论是否用空白把它们间隔开，都会分别把a、b和c读入到c1、c2和c3中去。空白指示使得输入流中的空白（如果有）被跳过。由于这个原因，下面两个语句是等价的：

```
scanf(" %c %c %c", &c1, &c2, &c3);
scanf("\t%c \t %c \n%c", &c1, &c2, &c3);
```

### 13.2.4 转换说明

在scanf()的控制串中，转换说明指示由%开始，由一个转换字符结束。它决定了怎样与匹配转换输入流中的字符。



scanf() 的转换字符		
未加修饰的转换字符	在输入流中被匹配的字符	相应参数的类型
c	任何字符, 包括空白	char *
d	可选的有符号十进制整数	int *
i	可选的有符号十进制、八进制或十六进制整数, 例如 77, 077或0x77	int *
u	可选的有符号十进制整数	unsigned *
o	可选的有符号八进制整数, 不需要前导0	unsigned *
x, X	可选的有符号十六进制整数, 不允许前导0x或0X	unsigned *
e, E, f, g, G	可选的有符号浮点数	float *
s	非空白字符序列	char *
p	printf()中的%p所产生的, 通常是无符号十六进制整数	void **
n, %, [...]	(请看下表)	

三个转换字符具有特殊的性质, 其一是[], 它不是一个字符, 尽管把它被处理为字符。我们在下表中讨论这种情况。

特殊的scanf()转换字符	
未加修饰的转换字符	说 明
n	在输入流中没有任何字符被匹配。相应的参数是指向int的指针, 它存储的是已读入的字符数
%	转换字符%%使得输入流中的%被匹配。没有任何相应的参数
[...]	把在[]中的字符集称为扫描集(scan set)。它决定了匹配什么和读入什么(请看下面的解释)。相应的参数是指向字符数组的基地址的指针, 该数组是足够大的, 能容纳被匹配的字符, 并用自动加入的空字符\0结尾

#### %和转换字符间可能有的字符

- 可选的\*, 它表示了赋值抑制, 后跟一个定义最大扫描宽度的可选整数, 此外还跟有修改转换符的h、l或L。
- 修饰符h, 它可在转换符d, i, o, u, x或X的前面。它表示把被转换的值以short int或unsigned short int型存储。
- 修饰符l, 它可在转换符d, i, o, u, x或X的前面, 或在e, E, f, g或G的前面。在第一种情况下, 它表示把被转换的值以long int或unsigned long int型存储, 在第二种情况下, 它表示把被转换的值以double型存储。

- 修饰符L，它可在e，E，f，g或G转换符的前面。它表示把被转换的值以long double型存储。

按照控制串(control string)中的转换说明把输入流中的字符转换成值，并把值存储在由参数表中的相应指针表达式指向的地址。除了字符输入之外，扫描域由连续的适合于指定转换的非空白字符组成。在遇到不适宜的字符时，就结束扫描域；如果超过了扫描宽度(如果指定了)或遇到了文件结束标记，也结束扫描域。

扫描宽度是被扫描的字符数。缺省是输入流的长度。描述%s跳过空白字符，然后读入非空白字符，直到遇见空白字符或文件结束标记为止。与此相对照，描述%5s跳过空白字符，然后读入非空白字符，直到读入了5个字符、遇见空白字符或文件结束标记为止。在读入串时，假定在内存中已经分配足够的空间，它能容纳读入的串和自动加上串结束标记\0。

可以用格式%nc读入下n个字符，其中包括空白字符。在读入串时，假定在内存中已经分配足够的空间，它能容纳读入的串。空字符\0并没有加上。

### 13.2.5 输入流中的浮点数

把输入流中的浮点数格式化为一个后跟带有可选小数点的数字串的可选符号，其后跟有一个可选的指数部分。指数部分由后跟有一个可选符号的e或E组成，其中可选的符号的后面还要有一个数字串。下面是一些例子：

```
77          /* is converted to a floating value */
+7.7e1      /* equivalent to 77 */
770.0E-1    /* equivalent to 77 */
+0.003
```

记住，输入流不是C代码，应用的规则不同。

### 13.2.6 使用扫描集

形式为%[string]的转换说明表示读入具体的串。把[]中字符集称为扫描集。如果扫描集中的第一个字符不是抑扬字符^，那么输入的串仅由扫描集中的字符组成。这样格式%[abc]仅输入含有字母a、b和c的串，如果在输入流中出现其他字符(包括空白)，那么输入就停止。与此相对照，如果扫描集中的第一个字符是抑扬字符^，那么输入的串由除了扫描集中的字符之外的所有的字符组成，而不是由扫描集中的字符组成。这样，格式%^[abc]输入一个由a、b或c终结的串，而不是由空白终结的串。考虑下面的代码：

```
char store[30];
scanf("%29[AB \t\n]", store);
```

它向字符数组store读入一个至多有29个字符的串。该串由A、B、空格、制表符和换行组成，并以\0结尾。

程序员通常把行看成是字符串，这样的字符串中可包含空格和制表符，并用换行符结尾。把一行读入内存的一个方法是使用一个适当的扫描集。考虑下面的代码：

```
char line[300];
while (scanf("%[^\n]", line) == 1)
    printf("%s\n", line);
```

这段代码的作用是去掉空行，并去掉任何行的前导空白。下面是一个更明显的例子。我们先

创建一个含有上述行的程序，编译后把可执行的代码放进文件pgm中。然后把下面的行放入infile:

```
A is for
    apple and
        alphabet pie.
```

在发出命令

```
pgm < infile > outfile
```

在文件outfile中，我们会发现:

```
A is for
apple and
alphabet pie.
```

除了scanf()函数族外，C还提供了函数gets()和fgets()，用以从文件按行读入(请参见练习9)。

### 13.2.7 返回值

在调用scanf()时，可能会发生输入失败或匹配失败。如果在输入流中没有字符，就会发生输入失败，返回的值是EOF(通常是-1)。在发生匹配失败时，非法的字符被留在输入流中，返回已成功转换的字符数。如果没有进行转换，返回的数就是0。如果scanf()成功，就返回成功转换的字符数，这个数也可能是0。

### 13.2.8 一个scanf()的例子

我们要给出一个说明scanf()函数的某些功能的例子。我们要详细地描述处理一个特定的输入流的匹配过程，该例子如下:

```
char    c, *cntrl_string, save[7], store[15];
int     a, cnt;

cntrl_string = "%d, %s %% %c %[abc] %s %5s %s";
cnt = scanf(cntrl_string, &a, &c, save, store, &store[5]);
```

输入流中的字符如下:

```
23, ignore_this % C abacus read_in_this**
```

把值23放进a中;逗号也被匹配;忽略了串“ignore\_this”;%被匹配;把字符C放进c中;把串“abac”放进数组save的前4个元素中,并用\0结尾,“us”被忽略;把“read\_”放进store的前5个元素中,并用\0结尾;从store[5]到store[14]存放的是“in\_this\*\*”,其中store[14]存放的是\0。由于已经成功地进行了5个转换,scanf()返回的值是5。

scanf()示例			
控制串中的指示	相应参数的类型	输入流中的内容	说 明
ab%2c	char *	abacus	ab被匹配, ac被转换
%3hd	short *	-7733	-77被转换
%4li	long *	+0x66	+0x6被转换(十六进制)
-%2u	unsigned *	-123	-被匹配, 12被转换
+ %lu	unsigned long *	+-123	+被匹配, -123被转换

(续)

scanf() 示例			
控制串的指示	相应参数的类型	输入流中的内容	说 明
+ %lu	unsigned long *	+ -123	+被匹配, -123被转换
+ %lu	unsigned long *	+ - 123	+被匹配, 错误, -不能变转换
%3e	float *	+7e-2	+7e被转换
%4f	float *	7e+22	7e+2被转换
%5lf	double *	-1.2345	-1.23被转换
%4Lf	long double *	12345	1234被转换
%p	void **	与系统相关	可以读入带有%p的printf()的输出

### 13.3 函数sprintf()和sscanf()

函数sprintf()和sscanf()分别是printf()和scanf()函数的串版本。在stdio.h中有它们的函数原型, 它们的原型如下:

```
int  sprintf(char *s, const char *format, ...);
int  sscanf(const char *s, const char *format, ...);
```

省略号向编译器指出, 函数需要参数的个数不定。形式为

```
sprintf( string, control_string, other_arguments);
```

的语句向字符数组string写入字符。control\_string和other\_arguments遵从printf()的规定。以类似的方式, 形式为

```
sscanf( string, control_string, other_arguments);
```

的语句从字符数组string读字符。

让我们看一个例子。

```
char  in_string[] = "1 2 3 go";
char  out_string[100], tmp[100];
int   a, b, c;

sscanf(in_string, "%d%d%d%s", &a, &b, &c, tmp);
sprintf(out_string, "%s %s %d%d%d\n", tmp, tmp, a, b, c);
printf("%s", out_string);
```

该段程序显示:

```
go go 123
```

首先函数sscanf()从in\_string读入3个整数和一个串, 并分别放入a、b、c和tmp中。然后函数sprintf()把它们写进out\_string。最后, 我们用printf()在屏幕上显示out\_string。当心: 提供足够的用于sprintf()输出的内存空间是程序员的责任。

### 13.4 函数fprintf()和fscanf()

函数fprintf()和fscanf()分别是printf()和scanf()函数的文件版本。在讨论它们的用法之前, 我们需要知道C是怎样处理文件的。

头文件stdio.h含有一些关于文件的结构。其中的一个是标识符FILE, 它是一个其成员描述文件当前状态的结构类型。要使用文件, 程序员不必知道FILE结构类型的任何细节。

在stdio.h中还定义了3个文件指针：stdin、stdout和stderr。有时我们把它们看成是文件，尽管实际上它们是指针。

在stdio.h中的标准C文件		
C中书写的 文件指针	名 字	说 明
stdin	标准输入文件	与键盘相关
stdout	标准输出文件	与屏幕相关
stderr	标准错误文件	与屏幕相关

头文件stdio.h含有很多处理文件的函数的原型。下面是fprintf()的fscanf()原型：

```
int fprintf(FILE *ofp, const char *format, ...);
int fscanf(FILE *ifp, const char *format, ...);
```

省略号向编译器指出函数需要的参数个数不定。形式为

```
fprintf( file_ptr, control_string, other_arguments);
```

的语句向由file\_ptr指向的文件中写入数据。control\_string和other\_arguments遵从printf()的规定。特别是，fprintf(stdout,...);等价于printf(...);。以类似的方式，形式为

```
fscanf(file_ptr, control_string, other_arguments);
```

的语句从由file\_ptr指向的文件读出数据。特别是，fscanf(stdout,...);等价于scanf(...);。

在下节中，我们要表明怎样用fopen()打开文件，怎样用fprintf()和fscanf()访问文件，以及怎样用fclose()关闭文件。

### 13.5 访问文件

文件有几个重要的性质。文件有名字，文件必须要被打开和关闭，文件被写入数据、或从文件读出数据或加入数据。在概念上，除非已打开文件，否则不能对它做任何操作。它就像一本合着的书，只有在它被打开的时候，我们才能从头到尾访问它。为了防止误用，我们要告诉系统我们处理文件的活动（读出、写入和添加）。在使用完文件后，我们要关闭它。

可以把文件抽象地看作是字符流。在打开文件后，用标准库中处理文件的函数可以访问流。考虑下面的代码：

```
#include <stdio.h>

int main(void)
{
    int    sum = 0, val;
    FILE   *ifp, *ofp;

    ifp = fopen("my_file", "r");    /* open for reading */
    ofp = fopen("outfile", "w");    /* open for writing */
    ....
```

这段程序打开当前目录中用于读的文件my\_file和用于写的文件outfile（标识符ifp是写入文件指针的英文单词“infile pointer”的缩写，ofp是输出文件指针的英文单词“outfile pointer”的缩写）。在打开文件后，文件指针被用于对该文件的所有引用。假设my\_file含有整

数，如果我们想对其进行累加并把结果放入outfile中，我们可以写：

```
while (fscanf(ifp, "%d", &val) == 1)
    sum += val;
fprintf(ofp, "The sum is %d.\n", sum);
```

注意fscanf()像scanf()一样返回成功转换的个数。如果我们已使用完它，我们可以用下面的语句关闭它：

```
fclose(ifp);
fclose(ofp);
```

形式为fopen(file\_name, mode)的函数调用以特定的模式打开指定的文件，返回文件指针。可用的模式有多种。

打开文件的模式	
"r"	打开文本文件，用于读
"w"	打开文本文件，用于写
"a"	打开文本文件，用于添加
"rb"	打开二进制文件，用于读
"wb"	打开二进制文件，用于写
"ab"	打开二进制文件，用于添加

这些模式的每一种都可以带有+字符，这意味着打开文件用于读和写；例如，r+的含义是打开文本文件用于读和写。

试图打开不能打开或不存在的文件以进行读，会产生错误。在这样的情况下，fopen()返回NULL指针。如果打开一个不存在的文件用于写，就会创建该文件；如果存在，就对它重写。如果以添加模式打开一个不存在的文件，就会创建该文件；如果存在，就在文件尾进行添加。

如果+出现在模式中，被打开的文件就处于修改模式，用于读和写。考虑下面的代码：

```
FILE *fp;

fp = fopen("my_file", "r+"); /* open for read and write*/
```

该段程序打开文件my\_file用于输入，但对它可以进行输入和输出。然而，输入后不能直接进行输出，除非已到达文件尾或插入了对fseek()、fsetpos()或rewind()的调用。类似地，输出后不能直接进行输入，除非已到达文件尾或插入了对fflush()、fseek()、fsetpos()或rewind()的调用。

在UNIX操作系统中，除了内容之外二进制文件和文本文件间没有区别；二者的文件机制是相同的。在MS-DOS和一些其他的操作系统中，对这些文件类型的每一种都有不同的文件机制（请参见练习22）。

在附录A“标准库”中找到对fopen()和fclose()这样的文件处理函数的详细描述。为了理解怎样使用各种函数，有必要参考附录。

### 13.6 例子：对文件行距加倍

让我们通过编写一个对某一文件进行两倍行距处理的程序来说明一些处理文件的函数的用法。在main()中，我们把打开用于读和写的文件作为传递的命令行参数。在打开文件后，

我们调用double\_space()来完成两倍行距的任务。

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

void double_space(FILE *, FILE *);
void prn_info(char *);

int main(int argc, char **argv)
{
    FILE *ifp, *ofp;

    if (argc != 3) {
        prn_info(argv[0]);
        exit(1);
    }
    ifp = fopen(argv[1], "r"); /* open for reading */
    ofp = fopen(argv[2], "w"); /* open for writing */
    double_space(ifp, ofp);
    fclose(ifp);
    fclose(ofp);
    return 0;
}

void double_space(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF) {
        putc(c, ofp);
        if (c == '\n')
            putc('\n', ofp); /* found newline - duplicate it */
    }
}

void prn_info(char *pgm_name)
{
    printf("\n%s%s\n\n%s%s\n\n",
        "Usage: ", pgm_name, " infile outfile",
        "The contents of infile will be double-spaced ",
        "and written to outfile.");
}
```

假设我们已经编译了这个程序，并把可执行的代码放入文件dbl\_space中。在我们发出命令

```
dbl_space file1, file2
```

后，程序从file1读数据，并向file2写数据。file2的内容与file1的内容相同，只是对各换行符进行了复制。

### 对程序dbl\_space的解析

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

  void double_space(FILE *, FILE *);
  void prn_info(char *);
```

因为stdlib.h中含有prn\_info()要使用的exit()的函数原型，所以我们引入了这个头文件。标识符FILE是一个在stdio.h中定义的结构。在使用这些文件时，我们不必知道系统实现这些文件机制的细节。double\_space()的函数原型表明它需要用两个文件指针做参数。

```
• int main(int argc, char **argv)
  {
      FILE *ifp, *ofp;

      if (argc != 3) {
```

```

    prn_info(argv[0]);
    exit(1);
}

```

标识符ifp和ofp是文件指针。标识符ifp是写入文件指针的英文单词的缩写，ofp是输出文件指针的英文单词的缩写。程序访问的两个文件是由命令行参数传进来的。如果命令行参数过多或过少，就调用prn\_info()显示程序的信息，并调用exit()退出程序。根据惯例，若程序错误exit()就返回非零值。

```

• ifp = fopen(argv[1], "r");    /* open for reading */
  ofp = fopen(argv[2], "w");    /* open for writing */

```

我们可以把argv看作是串数组。用函数fopen()打开argv[1]中命名的文件用于读。把该函数返回的指针赋给ifp。以类似的方式，打开argv[2]中命名的文件用于写。

```

• double_space(ifp, ofp);

```

把两个文件指针作为参数传递给函数double\_space()，然后由它完成两倍行距的工作。可以编写其他这种形式的函数，以对文件进行某种所需的操作。

```

• fclose(ifp);
  fclose(ofp);

```

用标准库中的函数fclose()关闭由ifp和ofp指向的文件。

```

• void double_space(FILE *ifp, FILE *ofp)
{
    int c;
    .....
}

```

这是double\_space()函数定义的开始部分。标识符c是int型的。虽然用它存储从文件获得的字符，但最终赋给它的值是EOF，而这不是一个字符。

```

• while ((c = getc(ifp)) != EOF) {
    putc(c, ofp);
    if (c == '\n')
        putc('\n', ofp); /* found newline - duplicate it */
}

```

宏getc()从由ifp指向的文件中读出字符，并赋给变量c。如果变量c的值不是EOF，那么putc()把c写进由ofp指向的文件。如果变量c的值是一个换行符，那么就把另一个换行符也写进由ofp指向的文件，这样输出文件就是两倍行距的文件。不断地重复这个过程，直到遇见EOF为止。在stdio.h定义了宏getc()和putc()。 ■

在打开文件的函数中显式地关闭文件是一种良好的编程风格。程序员没有显式地关闭文件，在程序退出时系统会自动地关闭它。

## 13.7 使用临时文件和得体的函数

在ANSI C中，程序员可以调用库函数tmpfile()来创建临时二进制文件，当关闭该文件后或程序退出后，就删除该文件。用模式wb+打开要修改的文件，意味着打开二进制文件用于读和写。在MS-DOS中，也可以把二进制文件用做文本文件。在UNIX中，对一个文件仅存在一种机制；除了内容外，二进制文件和文本文件是相同的。

在本节中，我们要编写一个基本的程序，用以说明tmpfile()的用法。我们的程序也给出了fopen()的一个得体版本。我们把程序命名为dbl\_with\_caps。下面是我们要完成的工作：



程序dbl\_with\_caps要完成的工作

- 以修改模式打开一个临时文件，用于读和写。
- 把第一个文件的内容复制到临时文件，把所有的小写字母变为大写。
- 在第一个文件的底部写一个标记行，这样我们能容易把该文件已有的内容与增加的内容区分开来。
- 把临时文件的内容复制到第一个文件的底部，因而重复了该文件的内容。

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

FILE *gfopen(char *filename, char *mode);

int main(int argc, char **argv)
{
    int c;
    FILE *fp, *tmp_fp;

    if (argc != 2) {
        fprintf(stderr, "\n%s%s\n\n%s%s\n\n",
            "Usage: ", argv[0], " filename",
            "The file will be doubled and some ",
            "letters capitalized.");
        exit(1);
    }
    fp = gfopen(argv[1], "r+"); /* open for read write */
    tmp_fp = tmpfile(); /* open for write read */
    while ((c = getc(fp)) != EOF)
        putc(toupper(c), tmp_fp); /* capitalize lowercase */
    fprintf(fp, "---\n"); /* print marker at bottom */
    rewind(tmp_fp); /* mv file pos to top */
    while ((c = getc(tmp_fp)) != EOF) /* copy tmp file */
        putc(c, fp); /* at bottom */
    return 0;
}

/* A graceful version of fopen(). */
FILE *g fopen(char *fn, char *mode)
{
    FILE *fp;

    if ((fp = fopen(fn, mode)) == NULL) {
        fprintf(stderr, "Cannot open %s - bye!\n", fn);
        exit(1);
    }
    return fp;
}
```

在我们解释这个程序之前，让我们看一下它的作用。假定文件apple的内容如下：

A is for apple and alphabet pie.

在我们发出命令

*dbl\_with\_caps apple*

后，文件apple的内容为

```
A is for apple and alphabet pie.
---
A IS FOR APPLE AND ALPHABET PIE.
```

对程序dbl\_with\_caps的解析

- `fp = gfopen(argv[1], "r+");` /\* open for read write \*/

我们正用fopen()的一个得体版本打开用于读和写的文件。由于一些原因，如果不能打开

文件，就输出一个信息，程序停止运行。与此相对照，考虑语句

```
fp = fopen(argv[1], "r+");    /* dangerous! */
```

由于一些原因，如果不能打开`argv[1]`，那么`fopen()`就返回空指针。这样在屏幕上没有向用户输出是什么出错的任何警告信息。

```
• tmp_fp = tmpfile();          /* open for write read*/
```

ANSI C提供打开临时文件的函数`tmpfile()`。该文件的模式是`wb+`；回想一下，正是二进制文件被打开用于读和写。也可以把二进制文件用作文本文件。在程序退出时，系统会撤消临时文件。关于该函数原型和其他细节，请参看附录A“标准库”。

```
• while ((c = getc(fp)) != EOF)
    putc(toupper(c), tmp_fp);    /* capitalize lowercase */
```

在`stdio.h`中定义了宏`getc()`和`putc()`；用它们从一个文件读数据，并写到另一个文件。在`ctype.h`中给出了`toupper()`的函数原型；如果`c`是一个小写字母，那么`toupper(c)`返回的是相应的大写字母，否则`toupper(c)`返回`c`。当心：一些ANSI C编译器不这样做（它们有望被改善）。你可能要写

```
while ((c = getc(fp)) != EOF)
    if (islower(c))
        putc(toupper(c), tmp_fp);
    else
        putc(c, tmp_fp);
```

```
• fprintf(fp, "---\n");
```

由于我们已遇到了由`fp`指向的流的文件尾标记，我们能从读转向写。此处，我们写了一个标记行，用以划分文件中的两部分内容。

```
• rewind(tmp_fp);
```

这使得由`tmp_fp`指向的流的文件指示器重新指向文件的开始。这个语句等价于

```
fseek(tmp_fp, 0, 0);
```

关于`fseek()`的原型和示例，请参见附录A“标准库”。

```
• while ((c = getc(tmp_fp)) != EOF)    /* copy TMP file */
    putc(c, fp);                        /* at bottom */
```

现在正从由`tmp_fp`指向的流读数据，并写到由`fp`指向的流。注意，在我们从写转向读由`tmp_fp`指向的流之前，要调用`rewind()`。

```
• FILE *gfopen(char *fn, char *mode)
{
    .....
```

这是`fopen()`的得体版本。如果有错误，在屏幕上会显示信息，程序退出。 ■

注意，我们写到`stderr`。在这个程序中，我们也可以把它写到`stdout`。然而，在使用这个函数的其他的程序中，写到`stderr`会更好（请参见练习13）。

## 13.8 随机地访问文件

除了可以在文件中一个接一个地访问字符（顺序访问）外，我们还可以在文件的不同位置上访问字符（随机访问）。库函数`fseek()`和`ftell()`用于随机地访问文件。形式为

`ftell(file_ptr)`

的表达式返回文件位置指示器的当前值。这个值表示距离文件开始的字节数（从0开始数）。只要从文件中读出一个数，系统就把文件位置指示器加1。在技术上，文件位置指示器是由 *file\_ptr* 指向的结构的成员；文件指针本身不指向流中的个体元素。这是很多程序初学者犯的概念性错误。

函数 `fseek()` 需要3个参数：文件指针、整数偏移量和表示在文件中计算偏移量的开始位置的整数。形式为

`fseek( file_ptr, offset, place );`

的语句，把文件位置指示器设置到从 *place* 开始的 *offset* 个字节。*place* 的值可以是0、1或2，其含义分别是文件的开始、当前位置或文件结尾。当心：函数 `fseek()` 和 `ftell()` 仅能用于二进制文件。在MS-DOS中，如果你要用这些函数，就应该用二进制模式打开文件。在UNIX中，由于只有一种文件机制，所以在任何模式下均可使用它们。

一个常见的练习是写一个倒置文件。让我们编写一个程序，通过访问随机文件完成这项工作。

```
#include <stdio.h>

#define MAXSTRING 100

int main(void)
{
    char    file_name[MAXSTRING];
    int     c;
    FILE    *ifp;

    fprintf(stderr, "\nInput a file name: ");
    scanf("%s", file_name);
    ifp = fopen(file_name, "rb"); /* ms-dos binary mode */
    fseek(ifp, 0, 2);             /* move to end of file */
    fseek(ifp, -1, 1);            /* back one character */
    while (ftell(ifp) >= 0) {
        c = getc(ifp);            /* ahead one character */
        putchar(c);
        fseek(ifp, -2, 1);        /* back two characters */
    }
    return 0;
}
```

把给用户的提示写到 `stderr`，这样程序使用重定向能正常地工作（请参见练习13）。我们用模式 `rb` 打开文件，这样程序在MS-DOS和UNIX下都能工作。

## 13.9 风格

在ANSI C中，不同的转换字符可能有相同的作用。在可能的情况下，我们用小写的转换字符代替大写转换字符。同样，考虑到移植，在处理十进制整数时，我们更喜欢用 `d` 而不用 `i`。在很多传统的C系统中后者是不可用的。

良好的编程风格要检查 `fopen()` 能否正常地工作。在很多重要的程序中，这样的检查是必要的。假设我们要打开 `my_file` 用于读，常见的作法是

```
if ((ifp = fopen("my_file", "r")) == NULL) {
    printf("\nCannot open my_file - bye!\n\n");
    exit(1);
}
```

由于某些原因,如果`fopen()`不能打开指定的文件,返回的指针值就是`NULL`。我们测试返回值,如果该值是`NULL`,我们就显示一条消息并退出程序。

另一个风格上的问题是不加选择地打开文件用于写。如果用`fopen()`打开一个已经存在的文件用于写,那么就会毁坏文件的原有内容。由于文件可能还有用,在这样的情况下应该警告用户。解决该问题的一种方法是检查文件能否被打开,如果能打开,就说明文件已经存在。在这样的情况下,应该警告用户(请参见练习10)。

大多数操作系统对同时打开的文件数目有限制。在编写大程序时,程序员有必要知道打开了哪些文件。在打开文件的函数中关闭文件是一种良好的编程风格。

### 13.10 常见的编程错误

程序初学者有时会忘记文件模式是串,而不是字符。如果你写

```
ifp = fopen("my_file", 'r');    /* wrong */
```

你的编译器至少会给你一个警告。建议程序员要注意所有的关于指针的警告。

程序员在使用`printf()`和`scanf()`时,或使用这些函数的文件版本和串版本时,经常会造成控制串中的格式与参数不匹配,下面是一个这样的例子:

```
double    x;
FILE      *ifp, *ofp;

.....
fscanf(ifp, "%f", &x);          /* wrong, use %lf instead */
fprintf(ofp, "x = %lf\n", x);    /* wrong, use %f instead */
```

在`printf()`函数族中,不存在格式`%lf`。

很多常见的错误与文件的使用有关。程序初学者常犯的一个错误是用文件名代替文件指针。在打开文件后,应该用文件指针访问文件,而不是用文件名访问文件。像

```
fprintf( file_name, ... );      /* wrong */
fclose( file_name);            /* wrong */
```

这样的结构会引起无法预料的错误。大多数编译器对值得怀疑的指针转换都给出警告,应该注意所有这样的警告。

其他的一些常犯的错误是打开已经打开的文件,关闭已经关闭的文件,对打开用于读的文件写,或从打开用于写的文件读。在这些情况下会发生什么是与系统相关的。编译器不能捕捉此类错误;正确地打开文件、使用文件和关闭文件是程序员的责任。

在使用`fprintf()`、`sprintf()`、`fscanf()`和`sscanf()`时,一种常见的编程错误是忘记了用指向`FILE`的指针或指向串的指针做为第一个参数。

```
fprintf("k = %d\n", k);        /* wrong, file ptr missing */
```

犯这样的错误是很自然的,因为`fprintf()`的用法与`printf()`的用法是很相似的。你的编译器给出了一个警告,而你不能解释警告的内容是什么,这样的情况是存在的。

下面是一个很难发现的错误。困难就在于编译器根本不能为你提供帮助。

```
char    msg[100], wrk[100];
int     val;

.....
sprintf("%s %d\n", msg, val);   /* input msg and val */
/* wrong arguments */
```

程序员用两个串作为sprintf()的前两个参数,该语句能通过编译,即使是使用了错误的串也能通过。程序员的意图是用sprintf()向wrk写数据,这样wrk应该是sprintf()的第一个参数。

若把代码移植到MS-DOS,会出现一些难以琢磨的问题。在UNIX中,由于所有的文件都是相同的,在打开文件时通常不用二进制模式,然而,函数fseek()和ftell()仅能正确地操作二进制文件。

程序员经常把文件指针看成是指向字符流的指针,这种想法无疑是可接受的。但程序员不应该把文件指针看成是指向流中字符的指针,这是不允许的。在技术上,文件指针指向类型为FILE的结构,该结构的成员知道在文件中发生什么(程序员不必知道结构本身的细节)。

#### 函数sscanf()是不同的

我们先讨论fscanf(),然后再说明sscanf()是如何不同的。下面的代码用于一个接一个地读非空白字符:

```
char    c;
FILE    *ifp;

.....
while (fscanf(ifp, "%c", &c) == 1) {
    .....
} /* do something */
```

文件机制用文件位置指针掌握文件的当前位置。由于这个原因,每次调用fscanf()时,我们都能得到文件的下一个字符。

函数sscanf()则不同。假设line是一个串,我们要一个接一个地访问line中的非空格字符,下面的代码是错误的!

```
while (sscanf(line, "%c", &c) == 1) {
    .....
} /* do something */
```

每次调用sscanf()时,都从line的行首开始访问。没有任何指示我们串中位置的串位置指示器机制(请参见练习5)。C针对于这个问题的正确处理方法是使用指针p,方法如下:

```
for (p = line; *p != '\0'; ++p) {
    if (!isspace(*p))
        .....
} /* do something */
```

### 13.11 系统考虑

在任何机器上,如果内存不足,那么使用calloc()或malloc()动态地分配内存都会失败,在这种环境下,这些函数的得体的版本是特别有用的。

文件是一种有限的资源。同时仅能打开FOPEN\_MAX个文件,其中包括stdin、stdout和stderr。在stdio.h中定义了符号常量FOPEN\_MAX。通常FOPEN\_MAX的值是20,但在一些新系统上它是64或更多。

在UNIX中,仅有一种文件机制,而在MS-DOS中有二进制文件和文本文件(可把二者都存储为文本)(请参见练习22)。

在以添加方式打开文件时,通过调用fseek()和rewind()对文件位置指示器重定位。在MS-DOS中,不能对文件中已有的内容重写,但在UNIX却可以重写。

对各种大型文件的处理的需求或要求似乎是越来越多。ANSI C增加了标准库函数fgetpos()和fsetpos()用以访问很大的文件。对于由于过大以至于传统的函数ftell()和fseek()不能访

问的文件，`fgetpos()`和`fsetpos()`能访问。关于更多的细节，请参见附录A“标准库”。

### 13.12 转向C++

C系统把`stdin`、`stdout`和`stderr`作为标准文件，此外系统可以定义像`stdprn`和`stdaux`这样的其他标准文件。可以把文件抽象地看作是连续处理的字符流。

C中的标准文件		
C中的写法	名 称	相关的设备
<code>stdin</code>	标准的输入文件	键盘
<code>stdout</code>	标准的输出文件	屏幕
<code>stderr</code>	标准的错误文件	屏幕
<code>stdprn</code>	标准的打印文件	打印机
<code>stdaux</code>	标准的辅助文件	辅助端口

C++流输入/输出把前3个标准文件分别与`cin`、`cout`和`cerr`相结合。通常C++把`cprn`和`caux`与相应的标准文件`stdprn`和`stdaux`相结合。还有`clog`是`cerr`的缓冲版本。程序员可以打开或创建其他的文件。我们要编写一个把一个已经存在的文件变成两倍间距的文件，并存放在一个已经存在的文件中新文件中的程序，来说明如何做到这一点。在命令行中指定文件名，并传递给`argv`。

通过引入`fstream.h`处理文件I/O。这包含用于创建与操纵输出和输入文件流的类`ofstream`和`ifstream`。要正确地打开和管理与系统文件相关的`ofstream`或`ifstream`，你首先要用一个适当的构造器声明它。我们先研究`ifstream`的行为。

```
ifstream();
ifstream(const char*, int = ios::in, int prot = filebuf::openprot);
```

无参数的构造器创建一个随后与一个输入文件相关的变量。有3个参数的构造器的第一个参数是文件名，第二个参数指明模式，第三个参数用于文件保护。

把文件模式参数定义为下面的`ios`类中的枚举元。

文件模式参数	
<code>ios::in</code>	输入模式
<code>ios::app</code>	添加模式
<code>ios::out</code>	输出模式
<code>ios::ate</code>	打开并寻找文件尾
<code>ios::nocreate</code>	打开但不创建模式
<code>ios::trunc</code>	丢弃内容并打开
<code>ios::noreplace</code>	如果文件存在，打开失败

`ifstream`的缺省模式是输入模式，`ofstream`的缺省模式是输出模式。如果打开文件失败，那么向流插入一个坏状态，它可以用`operator`进行测试。

让我们用这种模式编写一个简单的处理文件的程序。

```

//A program to double-space a file.
//Usage: executable f1 f2
//f1 must be present and readable
//f2 must be writable if it exists

#include <fstream.h>      //includes istream.h
#include <stdlib.h>

void double_space(istream& f, ostream& t)
{
    char c;

    while (f.get(c)) {
        t.put(c);
        if (c == '\n')
            t.put(c);
    }
}

int main(int argc, char** argv)
{
    if (argc != 3) {
        cout << "\nUsage: " << argv[0]
              << " infile outfile" << endl;
        exit(1);
    }

    ifstream f_in(argv[1]);
    ofstream f_out(argv[2]);

    if (!f_in) {
        cerr << "cannot open " << argv[1] << endl;
        exit(1);
    }
    if (!f_out) {
        cerr << "cannot open " << argv[2] << endl;
        exit(1);
    }
    double_space(f_in, f_out);
}

```

### 对程序dbl\_sp的解析

- void double\_space(istream& f, ostream& t)
 

```

      {
          char c;

          while (f.get(c)) {
              t.put(c);
              if (c == '\n')
                  t.put(c);
          }
      }
      
```

成员函数get从istream中获取一个字符，成员函数put向ostream放入一个字符。这两个成员函数都不忽略空白字符。输出换行字符两次，在输出文件中创建所需的两倍间距。

- ifstream f\_in(argv[1]);
   
ofstream f\_out(argv[2]);

变量f\_in用于输入，f\_out用于输出。用它们创建相应的istream和ostream变量。按通过命令行传递到argv[]中的名字调用相应的构造器。如果输入文件打开，那么就构造ifstream f\_in，与argv[1]中命名的文件相连接。如果输出文件能打开，那么就构造ofstream f\_out，与argv[2]中命名的文件相连接。

- if (!f\_in) {
   
    cerr << "cannot open " << argv[1] << endl;
   
    exit(1);

```

    }
    if (!f_out) {
        cerr << "cannot open " << argv[2] << endl;
        exit(1);
    }

```

如果用于f\_in或f\_out的构造器失败，那么构造器就返回一个由operator检测的坏状态，并执行一个错误退出。在这一点上，f\_in类似地用于cin，f\_out类似地用于cout。

- double\_space(f\_in, f\_out);

完成从输入文件到输出文件的两倍行距工作。 ■

fstream.h中的其他重要成员函数包括：

```

//opens ifstream file
void open(const char*, int = ios::in,
          int prot = filebuf::openprot);

//opens ofstream file
void open(const char*, int = ios::out,
          int prot = filebuf::openprot);

void close();

```

可以用这些函数打开和关闭适当的文件。如果你用缺省的构造器创建一个文件流，正常地你要用open()把它与文件关联起来。你可以用close()关闭文件，然后再打开要使用同一个流的另一个文件。可以用其他I/O类中的成员函数对文件进行各种操作。

## 小结

- 函数printf()、函数scanf()以及相关的这些函数的文件版本和串版本使用控制串中的转换字符处理可变长度的参数表。
- 如果要使用文件，就要引入标准头文件stdio.h。它含有标识符FILE和文件指针stdin、stdout和stderr的定义，它还含有很多处理文件的函数的原型和宏getc()和putc()的定义。
- 可以把文件看作是字符流。可以连续地或随机地访问流。当从文件读字符或向文件写字符时，对文件位置指示器加1。
- 在各程序的开头，系统打开3个标准文件：stdin、stdout和stderr。函数printf()向stdout中写，scanf()从stdin中读。stdout和stderr通常与屏幕连接；stdin通常与键盘连接。重定向使得操作系统要做其他的连接。
- 程序员可以分别用fopen()和fclose()打开和关闭文件。在打开文件后，用文件指针引用文件。
- 宏调用getc(ifp)从由ifp指向的文件读出下一个字符。类似地，宏调用putc(c, ofp)向由ofp指向的文件写入c的值。
- 编译器不能匹配的处理文件的错误有很多种。一个例子是试图向已经打开用于读的文件中写数据。正确地打开文件、使用文件和关闭文件是程序员的责任。
- 文件是一种缺乏的资源。能同时打开的文件的最大数目由stdio.h中的符号常量FOPEN\_MAX指定。在很多系统中这个数是20，在一些较新的系统中这个数是64或更多。知道哪些文件被打开是程序员的责任。在程序退出时，系统会自动地关闭已打开的文件。



- 标准库提供了一组通过文件指针访问文件的函数。例如，函数调用

```
fgets(line, MAXLINE, ifp);
```

从由ifp指向的文件中读出下一行字符。

- 在ANSI C中，可以用标准库中的函数tmpfile () 打开临时文件。在关闭临时文件时，系统会删除它。

## 练习

1. 用scanf() 语句中的控制串里的指示%[\n] 编写一个程序，从命令行给出的文件中按行读出，并每隔一行写入stdout。提示：使用计数器，测试它是奇数还是偶数。

2. 在ANSI C中转换说明%n是可用的，但在传统的C中是不可用的。你的编译器能正确地处理它吗？试一下下面的代码：

```
int    n1, n2;

printf("try %n me %n \n", &n1, &n2);
printf("n1 = %d  n2 = %d\n", n1, n2);
```

3. 我们可以按任意次序在转换说明中写标志字符吗？ANSI C文档对这方面的描述不太具体，似乎是允许这样做。试一下下面的代码，看你的编译器会发生什么情况。

```
printf("%0+17d\n", 1);
printf("%+017d\n", 1);
```

4. 下面代码的作用是什么？

```
char    s[300];

while (scanf("%*[\n]%*[\n]%*[\n]%*[\n]", s) == 1)
    printf("%s\n", s);
```

把这些行写在程序中，编译后把可执行的代码放进pgm，然后发出命令

```
pgm < my_file
```

此处的my\_file含有一些文本行。my\_file中的空行会引出问题吗？

5. 访问串与访问文件不一样。在打开文件时，文件位置指示器知道你在文件的什么位置上，串没有这样的机制。编写一个含有下列行的程序，并说明tmp1和tmp2中的内容。

```
char    c, s[] = "abc", *p = s;
int      i;
FILE    *ofp1, *ofp2;

ofp1 = fopen("tmp1", "w");
ofp2 = fopen("tmp2", "w");
for (i = 0; i < 3; ++i) {
    sscanf(s, "%c", &c);
    fprintf(ofp1, "%c", c);
}
for (i = 0; i < 3; ++i) {
    sscanf(p++, "%c", &c);
    fprintf(ofp2, "%c", c);
}
```

6. 编译下面的程序，并把可执行的代码放进文件try\_me中：

```
#include <stdio.h>

int main(void)
{
    fprintf(stdout, "A is for apple\n");
}
```

```

    fprintf(stderr, "and alphabet pie!\n");
    return 0;
}

```

执行该程序，理解它的作用。当你进行重定向输出时，会发生什么情况？试一下命令：

```
try_me > temp
```

在执行完后，你读一下文件temp。如果你使用UNIX，那么你就试一下命令：

```
try_me > & temp
```

这也会把写到stderr中的输出重定向。在你看到temp文件的内容后，你可能会感到惊讶！

7. 编写一个计算文件中行数的程序。应该把输入文件名作为命令行参数传递给程序，程序输出到stdout。把输入文件中的每行标上行号，并在行号后面加一个空格，写到输出文件中。

8. 修改在上一个练习中你编写的程序，使行号是右对齐的。下面的输出是不可接受的：

```

.....
9 This is line nine.
10 This is line ten.

```

如果多于10行但少于100行，那么应该用格式%2d显示行号；如果多于100行但少于1000行，那么应该用格式%3d显示行号；以此类推。如果你使用UNIX，那么你就试一下命令：

```
nlines /usr/dict/words > outfile
```

此处nlines是你的程序的名字。请检查一下outfile的头和尾，以及中间的一些位置，看你的程序工作得是否正确。（这个文件很大，不要弃置不管。）

9. 阅读附录A中关于fgets()和fputs()的部分。用fgets()从命令行给出的程序文件中按行读。如果在行中的第一个非空白字符是//，那么把这样的行写入stdout之前，先把//以及紧挨着//前和后的空格和跳格符删除掉。应该把所有的其他行不做任何改变地写到stdout中。提示：使用下述代码：

```

char   line[MAXLINE], store[MAXLINE];
FILE   *ifp = stdin;

.....
while (fgets(line, MAXLINE, ifp) != NULL)
    if (sscanf(line, " // %[^\n]", store) == 1) {
        fputs(store, stdout);
        fputs("\n", stdout);    /* restore the newline */
    }
    else
        fputs(line, stdout);
}

```

10. 编写一个名为wrt\_rand的程序，它创建一个随机数文件。交互式地输入该文件名。你的程序应该使用3个函数。下面给出的是第一个：

```

void get_info(char *filename, int *n_ptr)
{
    printf("\n%s\n\n%s",
        "This program creates a file of random numbers.",
        "How many random numbers would you like? ");
    scanf("%d", n_ptr);
    printf("\nIn what file would you like them? ");
    scanf("%s", filename);
}

```

程序中使用的第二个函数是fopen()的“精细”版本。其目的是：如果输出文件已经存在，就警告用户。

```
FILE *cfopen(char *filename, char *mode)
{
    char    reply[2];
    FILE    *fp;

    if (strcmp(mode, "w") == 0
        && (fp = fopen(filename, "r")) != NULL) {
        fclose(fp);
        printf("\nFile exists. Overwrite it? ");
        scanf("%1s", reply);
        if (*reply != 'y' && *reply != 'Y') {
            printf("\nBye!\n\n");
            exit(1);
        }
    }
    fp = fopen(filename, mode);
    return fp;
}
```

第三个函数是`gopen()`，它是`fopen()`的得体版本。我们在13.7节“使用临时文件和得体的函数”中讨论过这个函数。提示：为了整齐地写入随机数，要使用下面的代码：

```
for (i = 1; i <= n; ++i) {
    fprintf(ofp, "%12d", rand());
    if (i % 6 == 0 || i == n)
        fprintf(ofp, "\n");
}
```

11. 在本练习中，我们要检验`sscanf()`的典型用法。假设我们正在编写一个重要的交互式程序，它要求用户输入正整数。为了防止错误出现，我们可以把用户输入的行作为串输入。下面是处理串的一种方法：

```
char    line[MAXLINE];
int     error, n;

do {
    printf("Input a positive integer: ");
    fgets(line, MAXLINE, stdin);
    error = sscanf(line, "%d", &n) != 1 || n <= 0;
    if (error)
        printf("\nERROR: Do it again.\n");
} while (error);
```

这段程序能捕捉一些（不是全部）输入错误。例如，如果输入的是23e而不是233，那么这样的错误就会被漏掉。修改这段代码，如果除了输入的是数字串且其前后可有的紧挨着的空白字符外，还输入了其他字符，就认为输入是错误的。用这些想法重新编写你在上一个练习中的编写的程序`wrt_rand`。

12. 可以用命令

```
dbl_space infile outfile
```

调用产生两倍行距文件的程序。但是如果存在`outfile`，就会对`outfile`重写；这存在潜在的危险。重新编写该程序，使得它向`stdout`写入。这样可以用命令

```
dbl_space infile > outfile
```

调用该程序。这样的程序设计是较安全的。在所有的系统命令中，仅有一些可以对文件进行重写。毕竟没有谁愿意意外地丢失文件。

13. 对前一个练习中的程序`dbl_space`做进一步的修改。由于现在打算对程序使用重定向，所以在`prn_info()`的函数定义中使用`fprintf(stderr, ...)`是有意义的，但使用`printf(...)`则不然。因为`printf()`向可重定向的`stdout`中写，所以用户不会看到屏幕

上的信息。用符号>重定向写到stdout中的内容，它不会影响stderr。用两种方法重新编写你的程序：把错误信息写到stderr；把错误信息写到stdout。使用重定向执行程序，再不使用重定向执行程序，以理解不同的效果。

14. 对前一个练习中的程序dbl\_space做进一步的修改。它能接受命令行选项-n，这里的n可以是1、2或3。如果n是1，那么输出文件应该是单倍行距的，即应该把输入文件中的两个或多个连续的换行符在输出文件中写成一个换行符。如果n是2，那么输出文件应该是两倍行距的，即应该把输入文件中的一个或多个连续的换行符在输出文件中的写成两个换行符。如果n是3，那么输出文件应该是三倍行距。

15. 编写一个函数getwords(ifp, k, words)，使它从由ifp指向的文件中读出k个词，放在串words中，并用换行符分隔。函数应该返回成功转换的字符数，并放在words中。编写一个程序测试你的函数。

16. 编写一个函数putstring(s, ofp)，使它把串s写进由ofp指向的文件中。用宏putc()完成该项任务。编写一个程序测试你的函数。

17. 从一个文件中读出3个字符后，可以用ungetc()把它们放回到原文件（请参见A.12.4节“字符输入/输出”）。编写一个程序对此进行测试。

18. 编写一个在屏幕上一次显示20行文件内容的程序。输入文件作为命令行参数。在键入回车后，程序应该显示下20行。（这是UNIX中的more工具的基本版本。）

19. 修改上一个练习中的程序。程序应该显示作为命令行参数给出的一个或多个文件的内容。同样，使用命令行选项-n，这里的n是指定一次显示行数的正整数。

20. 可以用库函数fgets()一次从文件中读出一行（请参见A.12.4节“字符输入/输出”）。编写一个名为search的程序，搜索指定的模式。如果发出下面的命令：

```
search hello my_file
```

那么就在文件my\_file中搜索模式hello。显示所有的含有指定模式的行。（这个程序是grep的基本版本）提示：使用下面的代码：

```
char    line[MAXLINE], *pattern;
FILE    *ifp;

if (argc != 3) {
    .....
}

if ((ifp = fopen(argv[2], "r")) == NULL) {
    fprintf(stderr, "\nCannot open %s\n\n", argv[2]);
    exit(1);
}

pattern = argv[1];
while (fgets(line, MAXLINE, ifp) != NULL) {
    if (strstr(line, pattern) != NULL)
        .....
}
```

21. 修改上一个练习中的程序。如果给出命令行参数选项-n，还应该显示出行号。

22. 在MS-DOS的早期，把嵌入在文件中的Ctrl+z作为文件结束标记。但现在不这样做了，如果文件中有Ctrl+z，那么就把它作为文本文件打开用于读，Ctrl+z后的字符是不可以访问的。用下述行编写一个程序：

```
char    cntrl_z = 26;    /* decimal value for control-z */
int      c;
FILE    *ifp, *ofp;
```

```

ofp = fopen("tmp", "w");
fprintf(ofp, "%s%c%s\n",
        "A is for apple", cntrl_z, " and alphabet pie.");
fclose(ofp);
ifp = fopen("tmp", "r");          /* open as text file */
while ((c = getc(ifp)) != EOF)    /* print the file */
    putchar(c);
fclose(ifp);
printf("\n---\n");                /* serves as marker */
ifp = fopen("tmp", "rb");          /* open as binary file */
while ((c = getc(ifp)) != EOF)    /* print the file */
    putchar(c);

```

程序会显示什么？（程序在UNIX系统上会有所不同吗？）试一下MS-DOS命令

```
type tmp
```

它仅显示出了Ctrl+z之前的那些字符。你怎样知道在文件中还有字符？提示：试一下dir命令。通常在文本文件中没有Ctrl+z，但在二进制文件中肯定有。如果你用模式r代替rb打开二进制文件，就会发生难以琢磨的问题。

23. 仔细地检查下面的程序。它会显示出什么？

```

#include <stdio.h>

int main(void)
{
    printf("Hello!\n");
    fclose(stdout);
    printf("Goodbye!\n");
    return 0;
}

```

24. C++：用C++文件I/O，重做练习18。

25. Java：在下面的Java例子中，我们要说明怎样用标准Java类BufferedReader检测EOF。这个程序打开由命令行参数指定的文件，并向控制台回显文件的内容。用C重新编写这段代码。

```

// Echo.java - echo file contents to the screen
//Java by Dissection page 365.
import java.io.*;
class Echo {
    public static void main(String[] args) throws IOException
    {
        if (args.length < 1){
            System.out.println("Usage: java Echo filename");
            System.exit(0);
        }

        BufferedReader input =
            new BufferedReader(new FileReader(args[0]));
        String line = input.readLine();
        while (line != null) {
            System.out.println(line);
            line = input.readLine();
        }
    }
}

```

## 第14章 软件工具

软件工具分两种，一种是操作系统为用户提供的普通工具，另一种是明确地用于帮助程序员进行软件开发的专用工具。由于在C程序中能执行操作系统的命令，程序员可以用这些命令作为完成特定任务的软件工具。

一些软件工具在一种操作系统能使用，而在另一种操作系统中可能不能使用。例如，`make`在UNIX中可以使用，而在MS-DOS中却是附加的功能部件。虽然如此，在很多MS-DOS系统中是可以使用`make`的。

软件工具随时间而演化。新系统中的调试器越来越比以前的好用。C编译器本身被看作是软件工具。现在的大多数编译器都遵循ANSI C标准。

本章中，我们先讨论怎样在程序中执行操作系统命令，然后讨论一些供程序员使用的更重要的工具，其中包括编译器、`make`、`touch`、`grep`、修饰器和调试器。

### 14.1 在C程序中执行命令

库函数`system()`提供了对操作系统命令的访问。C程序员也可以使用操作系统提供的工具。在MS-DOS和UNIX中，命令`date`把当前的日期显示在屏幕上。如果我们在程序中向屏幕上显示该信息，我们可以写

```
system("date");
```

把传递给`system()`的串看作是操作系统命令。当执行这个语句时，就把控制交给操作系统，操作系统执行该命令，然后把控制又交还给程序。在`stdlib.h`中给出了`system()`的函数原型。

在UNIX中，`vi`是一种常用的文本编辑器。假设我们在程序中要用`vi`编辑由命令行参数指定的文件。我们可以写

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

int main(int argc, char **argv)
{
    char    cmd[1024];

    printf("Opening %s for editing ...\n", argv[1]);
    sprintf(cmd, "vi %s", argv[1]);
    system(cmd);
    ...
}
```

如果我们希望看到按字典次序对文件的行排序后的文件内容，我们可以写

```
sprintf(cmd, "sort %s", argv[1]);
system(cmd);
```

在MS-DOS有类似的例子，只是要用在MS-DOS上可用的编辑器代替`vi`。

在本节的最后一个例子中，假设我们厌烦了看MS-DOS系统上的`dir`显示的各种大写字母。我们可以编写一个结合这个命令的程序，该程序在屏幕上仅显示小写字母。

```

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

#define MAXSTRING 100

int main(void)
{
    char    cmd[MAXSTRING];
    char    tfn[MAXSTRING];    /* tmp filename */
    int     c;
    FILE    *ifp;

    tmpnam(tfn);
    sprintf(cmd, "dir > %s", tfn);
    system(cmd);
    ifp = fopen(tfn, "r");
    while ((c = getc(ifp)) != EOF)
        putchar(tolower(c));
    remove(tfn);
    return 0;
}

```

首先我们用库函数`tmpnam()`创建一个临时文件名,然后调用`system()`把`dir`命令的输出重定向到这个临时文件。随后我们在屏幕上显示文件的内容,并把大写的字母变为小写字母。最后,在使用完临时文件后,我们调用库函数`remove()`删除它。关于这些函数的细节,请参见附录A“标准库”。

## 14.2 环境变量

在UNIX和MS-DOS中都可以使用环境变量。下面的程序会在屏幕上显示环境变量:

```

#include <stdio.h>

int main(int argc, char *argv[], char *env[])
{
    int     i;

    for (i = 0; env[i] != NULL; ++i)
        printf("%s\n", env[i]);
    return 0;
}

```

参数`argv`和`env`的类型是指向`char`的指针的指针。我们可以把它们看作是指向`char`的指针数组或串数组。系统提供串和存储串的空间。每个数组的最后一个元素是`NULL`指针。在这个程序中,我们仅使用了`env`。在我们的UNIX系统上,这个程序显示结果如下:

```

BASE=/c/c/blufox/base
HOME=/c/c/blufox
SHELL=/bin/csh
TERM=xterm
USER=blufox
.....

```

等号左边的是环境变量,等号右边的是环境变量的值,把它们看作是一个串。在我们的MS-DOS系统上,这个程序显示结果如下:

```

COMSPEC=C:\COMMAND.COM
BASE=d:\base
INCLUDE=d:\msc\include
.....

```

UNIX和MS-DOS都提供了显示环境变量的命令。在UNIX中,该命令是`env`或`printenv`;在

MS-DOS中, 该命令是`set`。`set`命令的输出与我们的程序的输出是一样的。

按照惯例, 通常把环境变量大写。在C程序中, 我们能通过`main()`的第三个参数访问环境变量, 就像我们在上一个程序中做的那样; 我们也可以使用标准库中的函数`getenv()`访问环境变量。在`stdlib.h`中给出`getenv()`的函数原型:

```
char *getenv(const char *name);
```

如果作为参数传递的串是一个环境变量, 那么该函数返回由系统提供的串(指向`char`的指针), 该串是环境变量的值。如果作为参数传递的串不是一个环境变量, 那么该函数返回`NULL`。下面是说明`getenv()`用法的一些代码:

```
printf("%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n",
    "    User's name: ", getenv("NAME"),
    "    Login name: ", getenv("LOGNAME"),
    "    Shell: ", getenv("SHELL"),
    "    Base: ", getenv("BASE"),
    "Home directory: ", getenv("HOME"));
```

在我们的UNIX系统上, 这段代码打印如下信息:

```
User's name: Al Kelley
Login name: blufox
Shell: /bin/csh
Base: /c/c/blufox/base
Home directory: /c/c/blufox/center
```

在UNIX中, 系统提供一定的环境变量, 例如, `LOGNAME`、`SHELL`和`HOME`。要得到其他环境变量, 我们可以在`.login`文件中放进下面的行:

```
setenv BASE /c/c/blufox/base
setenv NAME "Al Kelley"
```

### 14.3 C编译器

有很多种C编译器, 一种操作系统可以提供其中的很多种, 下表仅显示了一部分。

编译器及其用法	
命 令	被调用的C编译器
<code>CC</code>	Bell实验室的C编译器
<code>CC</code>	运行在UNICOS下的CrayResearch的C编译器
<code>CC</code>	运行在HP-UX下的Hewlett-Packard的C编译器
<code>CC</code>	运行在IRIX下的Silicon Graphics的C编译器
<code>CC</code>	运行在Solaris下的Sun Microsystem的C编译器
<code>gcc</code>	Free Software Foundation的GNU C编译器
<code>hc</code>	Metaware的High C编译器
<code>occ</code>	Oregon Software的Oregon C编译器
<code>qc</code>	Microsoft的QuickC编译器
<code>bc</code>	Borland C编译器, 集成环境
<code>bcc</code>	Borland C编译器, 命令行版本

在本节中, 我们要解释编译器的用法, 并讨论在UNIX中可用于编译器的一些选项。很多这样的选项也适用于MS-DOS编译器。

如果一个完整的程序都在一个名为`pgm.c`的文件中, 那么命令

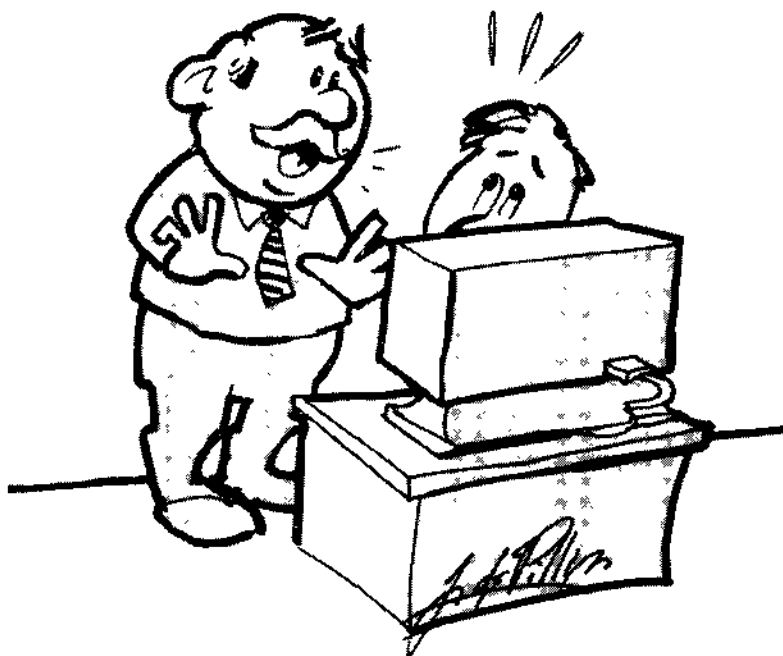


```
cc pgm.c
```

把pgm.c中的C代码转化成可执行的目标代码，并写进文件a.out。(在MS-DOS中，可执行的文件是pgm.exe。)命令a.out执行这个程序。现在考虑命令

```
cc -o pgm pgm.c
```

这使得可执行的代码被直接写进文件pgm，如果pgm已经存在，就重写它。a.out的内容会保持原样。(在MS-DOS中，该选项是-e，其后没有空格)



“不，不，达德利。我说我要执行你的程序，那是一件好事！”

实际上cc命令分三个阶段进行工作：首先调用预处理器，然后调用编译器，再调用加载器。加载器或连接器把所需要的各个部分生成可执行的文件，选项-c只可用于编译，也就是说，只可用于调用预处理器和编译器但不能调用加载器：如果我们把程序写在不只一个文件中，这个选项是很重要的，考虑命令：

```
cc -c main.c file1.c file2.c
```

如果没有错误；相应的以.o结尾的目标文件被建立（在MS-DOS中目标文件以.obj结尾），要建立一个可执行文件，我们可以混合编译.c和.o文件。例如，假定在main.c中有一个错误。在改正这个错误后，我们发出命令

```
cc -o pgm main.c file1.o file2.o
```

用.o文件代替.c文件可降低编译时间。除了.c和.o文件外，我们也可以用由带有-S选项的汇编程序或编译器生成的.s文件，还可以使用归档程序生成的库（请参见14.4节“创建库”）。通常库文件名用.a结尾（在MS-DOS中通常库文件名用.lib结尾）。

你的编译器可能不支持所有这些选项，也可能提供其他的选项，还可能提供等价的但名字（标记字符）不同的选项。MS-DOS中的编译器通常支持不同的内存模型。关于你的编译

器的选项表的细节，请参见相应的文档。

用于编译器的一些有用的选项	
<code>-C</code>	仅编译；生成相应的.o文件
<code>-g</code>	生成适合于调试器的代码
<code>-o name</code>	把可执行的输出代码放进name
<code>-p</code>	生成适合于profiler的代码
<code>-D name=def</code>	在每个.c文件的顶部放 <code>#define name def</code>
<code>-E</code>	调用预处理器，而不调用编译器
<code>-I dir</code>	在目录中dir寻找 <code>#include</code> 文件
<code>-M</code>	产生make程序的描述文件
<code>-MM</code>	产生make程序的描述文件，但不包括对任何标准头文件的依赖
<code>-O</code>	进行代码优化
<code>-S</code>	在相应的.s文件中生成汇编程序代码

## 14.4 创建库

很多操作系统都提供了创建和维护库的工具。在UNIX中，把这样的工具称为归档器（archiver），用`ar`命令调用它。在MS-DOS中，把这样的工具称为库管理程序（librarian），这是一个附加的功能部件。例如，Microsoft库是`lib`，而Borland C库是`stlib`。按常规来讲，在UNIX中库文件名用`.a`结尾，在MS-DOS中库文件名用`.lib`结尾。我们讨论的情形是与UNIX相关的，但一般的概念也适用于各种库。

在UNIX中，可以用归档器`ar`把一组文件编译成一个称为库的文件。标准的C库就是这样的一个例子。在大多数UNIX系统中，它是`/usr/lib/libc.a`文件，它可以作为整体存在，也可以分成若干部分存在多个文件中。如果你使用UNIX，试一下命令

```
ar t /usr/lib/libc.a
```

用关键字`t`显示库中的文件的标题或名字，标题比你看到的要多。要对它们计数，你可以发出命令

```
ar t /usr/lib/libc.a | wc
```

该命令把`ar`的输出传送到`wc`命令（对词计数）的输入，对行、词和字符进行计数。不要对标准库随时间而增长感到惊讶。在DEC VAX 11/780上，自20世纪80年代以来，标准库含有311个目标文件。在Sun机器上，据1990年的统计，标准库含有498个目标文件，在我们现在使用的Sun机器上，标准库含有822个目标文件。

下面我们要说明程序员怎样创建和使用他们自己的库。我们用对一个“得体的库”的创建来进行说明。在13.7节“使用临时文件和得体的函数”中，我们给出了`fopen()`的得体版本`gfopen()`。在名为`g_lib`的目录中有17个这样的得体函数，我们不时地要向它增加函数。它们是像`gfopen()`、`gfclose()`、`gcalloc()`和`gmcalloc()`这样的函数。每个函数都是在一个单独的文件中，但为了建造库，也可以把它们放在一个文件中。为了强化这些函数的思想，下面给出`gcalloc()`的代码：

```

#include <stdio.h>
#include <stdlib.h>

void *gcalloc(size_t n, size_t sizeof_something)
{
    void *p;

    if ((p = calloc(n, sizeof_something)) == NULL) {
        fprintf(stderr, "\n%s\n%s%u%s%u\n",
            "ERROR: calloc(n, sizeof_something) has failed",
            "with n = ", n, " and sizeof_something = ",
            sizeof_something);
        exit(1);
    }
    return p;
}

```

要创建库，我们必须首先编译.c文件以获得.o文件。在这之后，我们发出两个命令

```

ar ruv g_lib.a gfpopen.o gfclose.o gcalloc.o ...
ranlib g_lib.a

```

第一个命令中的ruv代表更换、修改和详述。如果库g\_lib.a不存在，这个命令就创建它；如果存在，就用.o文件替换库中同名的文件；如果在库中没有.o文件，就在库中增加它。在早期的UNIX系统上使用的ranlib命令，以有益于加载器的形式随机化库。较新的UNIX系统不需要randlib，不再把它作为工具提供。

假定我们要编写一个由main.c和两个.c文件组成的程序。如果程序调用gfpopen()，那么我们就需要让编译器（加载器）使用我们的库，可使用下面的命令做到这一点：

```
cc -o pgm main.c file1.c file2.c g_lib.a
```

假设程序调用一个函数，但没有提供它的函数定义。加载器首先在g\_lib.a中搜寻该函数，再在标准库中搜寻。加载器仅把所需要的那些函数放进最终的可执行文件中，它不会装入整个库。

如果我们写了一些程序，每一个都包含很多文件，那么就应该把每个程序放进它自己的目录中。同样，把像g\_lib.a这样的库都应该放进一个单独的目录，把像g\_lib.h这样的相关的头文件放进另一个目录。把g\_lib.a中的每一个函数的原型放在g\_lib.h中。在需要的时候引入这个头文件。为了管理这些，我们要使用make工具（请参见14.8节“make的用法”）。

## 14.5 使用profiler

在UNIX中，如果使用编译器时带有-p选项，就会产生额外的代码，并被放进编译器生成的目标文件和可执行文件。在调用程序时，额外代码产生能用于生成执行档案。被自动地写进文件mon.out，这个文件是不可阅读的。要获得mon.out中的信息，程序员必须发出命令

```
prof pgm
```

这里pgm是程序的名字。

下面给出一个例子，用以说明上述的工作情况。我们先编写一个用排序例程对两个同样的整数数组进行排序的程序。在编译程序时使用-p选项，我们会获得一个执行档案，它说明了两个排序例程的相关效率。

我们使用两个排序例程：一个是我们编写的置换排序，另一个是标准库中的qsort()。

文件cmp\_sorts.h中的内容:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include "g_lib.h"

void  chk_arrays(int *a, int *b, int n);
int   compare(const void *p, const void *q);
void  prn_array(int *a, int n);
void  slow_sort(int *a, int n);
```

文件main.c中的内容:

```
#include "cmp_sorts.h"

int main(void)
{
    int  *a, *b, i, n;

    printf("\n%s\n\n%s",
        "Two identical arrays of integers will be sorted.",
        "Input the array size: ");
    scanf("%d", &n);
    a = gcalloc(n, sizeof(int)); /* graceful calloc() */
    b = gcalloc(n, sizeof(int));
    for (i = 0; i < n; ++i)
        a[i] = b[i] = rand() % 1000;
    qsort(a, n, sizeof(int), compare);
    slow_sort(b, n);
    chk_arrays(a, b, n);
    if (n < 100)
        prn_array(a, n);
    return 0;
}
```

我们提示用户输入数组的尺寸, 然后用calloc()的得体版本动态地分配空间(请参见14.4节“创建库”)。用在0到999范围内的随机整数对数组a和b填充后, 我们用slow\_sort()和qsort()对a和b分别排序。在chk\_arrays()中, 我们确定已经按升序对数组进行了排序, 并且两个数组仍是相同的。如果数组尺寸小于100, 那么我们要在屏幕上显示a的元素。在开发程序期间, 这使得我们能知道每件事情是否按预期的那样工作。如果数组的尺寸很大, 我们就不必很麻烦地显示结果。我们仅想得到执行的简档。

文件compare.c中的内容:

```
#include "cmp_sorts.h"

int compare(const void *vp, const void *vq)
{
    const int  *p = vp, *q = vq;

    return (*p - *q);
}
```

因为把compare.c作为参数传递给了qsort(), 所以compare()的函数定义必须要与qsort()的函数原型中的相应参数相一致。特别是, 我们需要类型限定符const(请参见8.11节“类型限定符const和volatile”)。

文件slow\_sort.c中的内容:

```
#include "cmp_sorts.h"

void slow_sort(int *a, int n)
```

```

{
    int i, j, tmp;

    for (i = 0; i < n; ++i)
        for (j = i + 1; j < n; ++j)
            if (a[i] > a[j]) {
                tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
}

```

文件chk\_arrays.c中的内容:

```

#include "cmp_sorts.h"

void chk_arrays(int *a, int *b, int n)
{
    int i;

    for (i = 0; i < n - 1; ++i) {
        assert(a[i] == b[i]);
        assert(a[i] <= a[i+1]);
    }
    assert(a[i] == b[i]);
}

```

文件prn\_array.c中的内容:

```

#include "cmp_sorts.h"

void prn_array(int *a, int n)
{
    int i;

    for (i = 0; i < n; ++i) {
        if (i % 12 == 0)
            putchar('\n');
        printf("%5d", a[i]);
    }
    putchar('\n');
}

```

现在我们要得到程序的执行简档。为此,我们必须先使用-p选项编译程序。

```

gcc -p -o compare_sorts main.c chk_arrays.c compare.c \
    prn_array.c slow_sort.c g_lib.a

```

因为库含有gcalloc(),所以加载器需要g\_lib.a。下面,我们发出命令

```
compare_sorts
```

并在提示后输入1000,程序就会创建文件mon.out。最后,为了得到执行的简档,我们发出命令

```
prof compare_sorts
```

该命令在屏幕上显示下面的程序的执行档案:

%time	cumsecs	#call	ms/call	name
96.6	2.84	1	2839.90	_slow_sort
1.4	2.88	11291	0.00	_compare
1.4	2.92	1	40.00	_qsort
0.7	2.94	1	20.00	_main
0.0	2.94	1	0.00	__doprnt
0.0	2.94	2	0.00	_calloc
0.0	2.94	1	0.00	_chk_arrays
0.0	2.94	1	0.00	_exit

注意,对compare()的调用超过11 000次,这是因为qsort()是递归的。如果用-p选项编

译`qsort()`，我们就能看到对它调用多少次。表中的函数不都是用户定义的；其中的一些，例如`_doprnt`，是系统例程。简档中显示出执行`slow_sort()`需要96.6%的时间，执行`qsort()`仅需要1.4%的时间。在进行改善执行的时间效率的工作时，这样的执行简档可能是很有用的。如果你知道了哪些函数使用的时间多，就可以改善它们，以减少执行时间。同样，如果你知道了一个函数所用的执行时间的比例很小，那么你知道的有必要改善它的信息也很少。

## 14.6 关于时间的编码

大多数操作系统都提供了对机器的内部时钟的访问。在本节中，我们要说明怎样使用一些定时的函数。因为我们打算要在很多程序中使用定时函数，所以把它们都放在了库`u_lib.a`中，`u_lib.a`是我们的工具库。

在ANSI C中，通过使用一些其原型是在`time.h`中的函数访问机器的时钟。这个头文件也含有一些其他的结构，其中包括`clock_t`和`time_t`的类型定义，在处理时间上它们都是有用的。通常，`clock_t`和`time_t`的类型定义由：

```
typedef long clock_t
typedef long time_t
```

给出，这些类型被用在函数原型中。下面是我们在定时例程中用到的三个函数的原型：

```
clock_t clock(void);
time_t time (time_t *p);
double difftime(time_t time1, time_t time0);
```

在执行程序时，操作系统知道所用的处理器时间。在调用`clock()`时，返回的值是程序执行到该点时所用的最佳近似时间。时钟单位随机器的不同而不同。宏

```
#define CLOCKS_PER_SEC 1000000 /* machine-dependent */
```

由`time.h`提供，可以用它把由`clock()`返回的值变成秒。

通常函数`time()`返回自1970年1月1日起以来的秒数，`time()`也可能使用其他的时间单位或起始时间。如果传递给`time()`的指针参数不是`NULL`，那么也把返回值赋给参数指向的对象。这个函数的一个典型用法是

```
srand(time(NULL));
```

该语句为随机数发生器播种。如果把由`time()`产生的两个值传递给`difftime()`，那么返回值是按秒表达的时间差，其类型是`double`。

下面是有很多用途（包括有效码的开发）的一组定时例程。我们把这些函数放在文件`time_keeper.c`中。

文件`time_keeper.c`中的内容：

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

#define MAXSTRING 100

typedef struct {
    clock_t    begin_clock, save_clock;
    time_t     begin_time, save_time;
} time_keeper;
```

```

static time_keeper tk;    /* known only to this file */

void start_time(void)
{
    tk.begin_clock = tk.save_clock = clock();
    tk.begin_time = tk.save_time = time(NULL);
}

double prn_time(void)
{
    char    s1[MAXSTRING], s2[MAXSTRING];
    int     field_width, n1, n2;
    double  clocks_per_sec = (double) CLOCKS_PER_SEC,
            user_time, real_time;

    user_time = (clock() - tk.save_clock) / clocks_per_sec;
    real_time = difftime(time(NULL), tk.save_time);
    tk.save_clock = clock();
    tk.save_time = time(NULL);

    /* Print the values found, and do it neatly. */

    sprintf(s1, "%.1f", user_time);
    n1 = strlen(s1);
    sprintf(s2, "%.1f", real_time);
    n2 = strlen(s2);
    field_width = (n1 > n2) ? n1 : n2;
    printf("%s%.1f%s\n%s%.1f%s\n\n",
        "User time: ", field_width, user_time, " seconds",
        "Real time: ", field_width, real_time, " seconds");
    return user_time;
}

```

注意，结构tk在各函数的外部，仅可在本文件中使用，它用于在本文件中的函数间通信，本文件外的函数不能访问它。虽然我们可以设计把tk作为参数传递给这些函数，但这样是不适当的。如果我们这样做，用户就必须为tk分配空间，然后编写调用这些函数的start\_time(tk)和prn\_time(tk)，但是用户对start\_time()和prn\_time()的内部操作不感兴趣。这样最好向用户隐藏tk。

在调用start\_time()时，把clock()和time()的返回值存储在tk中。在调用prn\_time()时，用clock()和time()返回的新值计算并显示已经过的用户时间和已经过的实时时间，并把新值存储在tk中。用户时间是系统分配给程序运行的时间；实时时间是挂钟走的时间。在分时系统中，它们不需要是一致的。

函数prn\_total\_time()也在这个文件中，但我们没说明它。它与prn\_time()类似，只是相对于start\_time()的最后调用计算经过时间而不是相对于对三个函数之一调用计算经过时间。

因为我们打算在很多程序中使用定时例程，所以把它们都放在了库u\_lib.a中，u\_lib.a是我们的工具库。下面的命令完成这项工作：

```
cc -c time_keeper.c; ar ru v u_lib.a time_keeper.o
```

在头文件u\_lib.h中，u\_lib.a含有函数的原型。在需要该头文件的其他地方也要引入它。

现在我们要说明怎样使用定时例程。在某些应用中要求快速浮点乘法。我们使用的变量类型应该是float还是double？用下面的程序测试就能找到答案：

```

/* Compare multiplication times. */

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

#define N 1000000

```

```

int main(void)
{
    long    i;
    float    a, b = 3.333, c = 5.555; /*arbitrary values*/
    double   x, y = 3.333, z = 5.555;

    printf("\nNumber of multiplies: %d\n\n", N);
    printf("Type float:\n");
    start_time();
    for (i = 0; i < N; ++i)
        a = b * c;
    prn_time();
    printf("Type double:\n");
    for (i = 0; i < N; ++i)
        x = y * z;
    prn_time();
    return 0;
}

```

在使用传统C编译器的较老的机器上,我们很惊讶地发现单精度浮点乘法比双精度浮点乘法要慢!在传统的C中,要把任何float自动地提升到double,或许这一事实可以说明这一点。在使用ANSI C编译器的较新的机器上,我们发现单精度乘法比双精度浮点乘法要快30%。

## 14.7 dbx的用法

程序员可以用调试器单步地一次调试一个代码行,在每步可以看到变量和表达式的值。这对发现程序为什么出现意外是极有帮助的。在编程世界中,人们经常要使用调试器,但dbx并不是特别好的调试器,它通常用在UNIX系统上。在本节中,我们要描述dbx的基本用法。

### 使用dbx的步骤

- 1) 转移到含有C程序的目录。
- 2) 用-g选项(调试)编译程序。
- 3) 给出命令dbx pgm,此处pgm是可执行的文件的名称,该命令调用dbx。从现在开始,我们发出的命令都是dbx命令。
- 4) 发出file main.c这样的命令,它为dbx设置当前文件。
- 5) 用命令list查看文件的前10行。再次使用该命令查看文件的第二个10行。
- 6) 发出命令stop at n,此处n是可执行文件的行号。
- 7) 发出命令run(通常发出的是调用pgm的命令),例如run 3。
- 8) 在此处你可以用step和next命令单步调试程序。用step单步地进入函数,用next跳过函数。用命令print var显示变量var的当前值。
- 9) 用命令quit停止程序的执行。

这是怎样使用dbx的概述。其他有用的命令是alias、continue和help。alias用于查看别名,continue用于继续执行程序,help用于查看命令列表。

为了减少使用dbx所需要的输入量,程序员可以设置别名,有时系统能自动地进行这样的设置。(随着系统的不同,dbx会有略微的变化)程序员把别名放在文件.dbxinit中,该文件可能当前目录中,如果可使用别名,那么它应该放在主目录中。下面是我们要用到的别名:

```

alias a alias

a c cont;  a d delete;  a e edit;  a h help;  a l list;
a n next;  a p print;   a q quit;   a r run;   a j status;
a s step;  a w where;   a st stop;

```



在MS-DOS中, 调试器是一个附加产品。Microsoft和UNIX都提供了非常好的此类产品。UNIX工作站也支持这样的附加产品。程序员要花费一些时间学习使用调试器, 这样的努力是值得的。

## 14.8 make的用法

可把一个中等的或更大的程序放在一个文件中, 反复地对它多次编译。这样做对于程序员和机器来讲, 效率低且代价大。一个更好的策略是, 把一个程序写在多个.c文件中, 按需要分别编译它们。可以用make工具记录源文件, 它提供了对库和相关的头文件的便利访问。在UNIX中, 这个工具总是可用的, 在MS-DOS中, 它是附加功能部件, 有时是可用的。使用make工具有利于对程序进行构造和维护。

假设我们正在编写一个由一些.h和.c文件组成的程序。通常我们在一个单独的目录中进行编写。*make*命令读一个其缺省名是makefile的文件, 这个文件含有构成程序的各种模块或文件的依赖以及要采取的适当操作, 特别是它含有用于编译或再编译程序的指令。把这样的文件称为make程序的描述文件(makefile)。

为了简化, 假定我们的程序由main.c和sum.c组成, 头文件sum.h在两个文件中都出现, 可执行的程序代码在文件sum中。下面是一个能用于程序开发和维护的简单makefile:

```
sum: main.o sum.o
    cc -o sum main.o sum.o

main.o: main.c sum.h
    cc -c main.c

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

第一行指出文件sum依赖目标文件main.o和sum.o, 这是一个依赖(dependency)行的示例, 该行必须从第一列开始。第二行指出, 如果一个或几个.o文件发生了变化, 怎样对程序进行编译, 把这样的行称为操作行(action line)或命令(command)。在一个依赖行的后面可能有多多个操作。把依赖行和跟在它后面的操作行称为规则(rule)。当心: 每一个操作行必须用一个跳格符开始(在屏幕上或在纸上, 跳格符看起来像一连串的空格)。

缺省上, *make*命令处理它在makefile中发现的第一条规则。然而, 在规则中的依赖文件本身可能还依赖在其他规则中说明的其他文件, 这会使得先处理其他规则。这些文件可能还会引起更多的规则被处理。

在makefile中的第二条规则说明main.o依赖main.c和sum.h这两个文件。如果其中的一个发生变化, 就按操作行显示的那样进行处理, 修改main.o。在创建这个makefile后, 程序员可以用命令

```
make
```

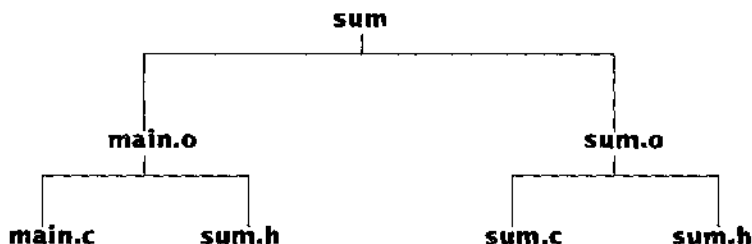
对程序sum进行编译和重编译。用这个命令, *make*读文件makefile, 创建用于自身的依赖树(dependency tree)并采取所需要的操作。

某些规则被内置到*make*中, 其中包括.o文件依赖相应的.c文件这样的规则。由于这个原因, 可以给出一个等价的makefile:

```
sum: main.o sum.o
    cc -o sum main.o sum.o
```

```
main.o: sum.h
    cc -c main.c

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



make工具识别一些内置的宏。使用其中一个宏，我们可以得到另一个等价的makefile：

```
sum: main.o sum.o
    cc -o sum main.o sum.o

main.o sum.o: sum.h
    cc -c $*.c
```

第二个规则指明两个.o文件依赖sum.h。如果我们编辑sum.h，就必须重制main.o sum.o。在重制main.o时，把宏\$\*.c扩展到main.c；在重制sum.o时，把宏\$\*.c扩展到sum.c。

总的来讲，makefile由一系列的被称为规则的项组成，规则描述了依赖和操作。带有一系列的用空格分开的目标文件的规则从第一列开始，在一个冒号后是一系列用空格分开的必备文件，也被称为源(source)文件。后边的所有的用跳格符开始的行是系统修改目标文件所使用的操作，例如编辑。目标文件在某些方面依赖必备文件，如果修改了必备文件，就必须修改目标文件。

在14.5节“使用profiler”中，我们编写一个用两个例程对相同的数组进行排序的程序。在那里我们说明了怎样用-p选项编译程序，以得到一个执行档案。实际上，我们把makefile用于程序开发。下面是我们要用到的一个makefile：

```
# Makefile for compare_sorts.
# After execution, use prof to get a profile.

BASE    = /c/c/blufox
CC       = gcc
CFLAGS  = -p
EFILE   = $(BASE)/bin/compare_sorts
INCLS   = -I$(BASE)/include
LIBS    = $(BASE)/lib/g_lib.a

OBJS = main.o  chk_arrays.o  compare.o \
        prn_array.o  slow_sort.o

$(EFILE): $(OBJS)
    @echo "linking ....."
    $(CC) $(CFLAGS) -o $(EFILE) $(OBJS) $(LIBS)

$(OBJS): compare_sorts.h
    $(CC) $(CFLAGS) $(INCLS) -c $*.c
```

#### 对用于程序compare\_sorts的makefile的解析

- # Makefile for compare\_sorts.  
# After execution, use prof to get a profile.

可以在makefile中放注释。注释用#开始，其范围是一行。

- `BASE = /c/c/blufox`

这是宏定义的一个示例。宏定义的一般形式如下：

```
macro_name = replacement_string
```

根据惯例，通常把宏的名字（macro name）大写，但不是必须这样。置换串（replacement string）可以含有空白字符。如果在行尾出现了反斜杠字符，置换串就持续到下一行。宏BASE表示在特定的机器上的操作基（在我们的主目录中用不着它）。

- `CC = gcc`  
`CFLAGS = -p`

第一个宏指明我们用的编译器，在本例中是GNU C编译器。第二个宏指明了和gcc命令一起使用的选项。

- `EFILE = $(BASE)/bin/compare_sorts`  
`INCLS = -I$(BASE)/include`  
`LIBS = $(BASE)/lib/g_lib.a`

第一个宏指明了可执行文件；第二个宏指明了包含文件的目录，其前放了一个-I选项；第三个指明了我们的得库（请参见14.4节“创建库”）。我们需要这个库，因为它含有gcalloc()的目标代码，即gcalloc()的得库版本。由于我们的程序经常要调用g\_lib.a中的函数，我们通常制作这个库，以供编译器（加载器）使用。g\_lib.a中的函数的原型在g\_lib.h中，我们必须告诉编译器在哪儿寻找这个头文件。

- `OBJS = main.o chk_arrays.o compare.o \`  
`prn_array.o slow_sort.o`

在这个宏定义中，替换串是出现在等号右边的目标文件列表。注意，我们用反斜杠继续本行。我们先列出了main.o，然后再按字典顺序列出了其他文件，其实顺序是不重要的。

- `$(EFILE): $(OBJS)`  
`@echo "linking ...."`  
`@$(CC) $(CFLAGS) -o $(EFILE) $(OBJS) $(LIBS)`

第一行是依赖行，其他的两行指明了要采取的操作。注意，后两行都用一个跳格符开始。（它看起来像屏幕上的8个空格。）@符号表示操作行本身不在屏幕上回显。宏调用的形式是

```
$( macro_name )
```

因此可以用

```
$(BASE)/bin/compare_sorts
```

替换\$(EFILE)，还可以用以下的形式替换：

```
/c/c/blufox/bin/compare_sorts
```

类似地，用目标文件的行替换\$(OBJS)，以此类推。因此，依赖行说明可执行文件依赖目标文件。如果对一个或多个目标文件做了修改，就会发生指定的行为。第二个操作行被扩展为

```
@gcc -p -o /c/c/blufox/bin/compare_sorts main.o \
chk_arrays.o compare.o prn_array.o slow_sort.o \
/c/c/blufox/lib/g_lib.a
```

由于打印页的空间限制我们把该操作行写为3行，实际上仍按一个操作行处理。选项-p使得编译器产生适合档案的额外代码。建议：如果你的make是新的，就先不要用@符号建造你

的makefile。在你理解了它的作用后，你就能用它防止回显。

```
• $(OBJ): compare_sorts.h
  $(CC) $(CFLAGS) $(INCLS) -c $.c
```

依赖行表明所有的目标文件依赖头文件compare\_sorts.h。如果已对头文件进行了修改，那么对所有的目标文件也必须修改。通过操作行可做到这一点。在操作行尾部的结构\$\*是一个预定义宏，把这个宏称为基文件名宏(base file name macro)。它扩展到正在建造的文件名，扩展名除外。例如，如果main.o正被建造，那么\$\*扩展到main.c，操作行变为：

```
gcc -p -I/c/c/blufox/include -c main.c
```

一定的依赖被内建到make工具中。例如，每个.o文件依赖相关的.c文件。这意味着如果.c文件被修改，那么为相应的.o文件指定的操作被采取。对于我们此处正在讨论的makefile，这意味着把.c文件重新编译，以产生新的.o文件，这又引起了对所有的目标文件的重新连接。

```
• -I/c/c/blufox/include
```

选项-Idir的含义是在目录dir中寻找#include的文件，这个选项对库的使用做了补充。在构成这个程序的每个.c文件的顶部，我们都写有行

```
#include "compare_sorts.h"
```

在靠近compare\_sorts.h的顶部，我们有

```
#include "g_lib.h"
```

这个头文件含有在库g\_lib.a中的函数的函数原型。-I选项告诉编译器，如果头文件不在当前目录中，就应该去那里找它。 ■

make工具不是C专有的，可以用它维护用任何语言编写的程序。更一般地，它可以用于任何种类的由带有依赖和相关的操作的文件组成的项目。

## 14.9 touch的用法

在UNIX中，touch工具总是可用的，在MS-DOS中有时也是可用的。通常make是通用的。touch工具用于把一个新的时间放进文件中。在用make时这是必须的，它比较文件的时间以决定必须做什么。

为了说明touch的用法，假设我们已经有了一个在上一节中讨论的makefile，以及相关的.h和.c文件。要把当前的日期放进文件compare\_sorts.h中，我们可以发出命令

```
touch compare_sorts.h
```

这使得compare\_sorts.h比依赖它的所有目标文件的时间更新。为了重编译所有的.c文件并连接目标文件，以创建新的可执行的文件，我们发出命令

```
make
```

## 14.10 其他有用的工具

操作系统为程序员提供了很多有用的工具。下面我们列出了一些在UNIX系统中发现的工具以及一些注释。有时在MS-DOS中可使用类似的工具。

有用的工具	
<i>cb</i>	C修饰器, 可用于“优质输出”C代码
<i>diff</i>	显示在不同的两个文件中的行
<i>grep</i>	在一个或多个文件中搜寻一个模式
<i>indent</i>	带有一些选项的C代码“优质打印机”
<i>wc</i>	对一个或多个文件中的行、词和字符计数

*cb*工具从stdin中读, 向stdout中写, 它的功能不是很强。为了看它能做什么, 试一下命令

```
cb < pgm.c
```

此处, *pgm.c*的格式不怎么好。工具*indent*的功能较强, 在Berkeley和Sun版本的UNIX中能发现它。要认真地使用它, 你就要阅读联机手册。

像*diff*、*grep*和*wc*这样的工具具有一般的性质, 除了程序员外, 别人也可使用它。虽然它们是UNIX的工具, 在MS-DOS中它们经常也是可用的, 特别是*grep*, 它是程序员的一个有力的工具。

最后, 我们要说一下C可与其他高级工具协作, 其中的一些是语言。特别重要的工具有*lex*和*yacc*。可用的更新的版本分别是*flex*和*bison*, 它们是Free Software Foundation的产品。在MS-DOS和UNIX中都可以使用它们。

更多的工具	
<i>awk</i>	模式扫描和处理语言
<i>csh</i>	像 <i>sh</i> 和 <i>ksh</i> 一样, 其外壳是可编程的
<i>lex</i>	为词法分析生成C代码
<i>sed</i>	从一个文件中取其命令的流编辑器
<i>yacc</i>	另一种编译程序的编译程序, 用于产生C代码

## 14.11 风格

一种良好的编程风格是使用系统命令, 而不是再编写与系统命令等同的工具。例如, 假设你要在名为*data*的文件中找所有的含有词*beautiful*的行。你可以用语句

```
system("grep beautiful data > tmp");
```

来完成这项工作。在你的程序中, 可以打开文件*tmp*以做进一步处理。

程序初学者经常对编译器生成的错误和警告信息在理解上感到困难, 由于这个原因, 有抛弃警告的趋势。一种良好的编程风格是反复地重写代码, 直到编译器不再给出警告为止。

一种有效的编程风格是程序员使用复杂的工具。像调试器、*profiler*、*make*和*touch*几乎是通用的。除了非常小的程序外, 对于所有的开发, 这些工具都是极其有用的。当然, 学习使用这些工具是要费些力的, 但回报是大的。

## 14.12 常见的编程错误

对于程序员来说, 编译器本身是主要的软件工具。一个常见的错误是没能把编译器的警告设置在最高的级别上, 应该注意所有的警告。

程序初学者经常犯有试图把一个程序全写在一个文件中的错误。较好的策略是把程序分成几个模块，用make跟踪依赖。

使用make时常犯的一个错误是忘记了必须要用一个跳格符作为操作行的开头。这是一个容易犯的错误，因为跳格符是不可见的。

在程序员编写makefile时，可能必须要调试它。用下面命令能调用“无操作”选项。

```
make -n
```

这个选项使得make在屏幕上显示它的操作会是什么，只是实际上它们并没有执行。

在程序中的控制流没有按程序员预期的那样工作时，可以用“优质打印机”作为调试工具。使用“优质打印机”进行调试的想法还没有被普遍地接受，但在一些情况下它能给人们一些帮助（请参见练习9）。

### 14.13 系统考虑

在如今的社会，程序员经常要在网络上工作，不像以往那样只在单机上工作。在我们的环境中，我们日常要在校园、家庭和社会中使用各种各样的计算机。管理这种应用的复杂性的一个方法是编写的代码要依赖环境变量。下面的行来自我们的.login文件：

```
setenv BASE /c/c/blufox
```

这使得环境变量BASE存有一个串值，它表示特定机器上的确定的目录。在其他机器上作为“基”的目录是不同的，但是由于我们编写的软件访问的环境变量是BASE，这样这个问题就不存在了。

很多编译器都支持“详细”选项-v，它使得编译器在屏幕上显示更多的信息。程序员应该试一下这个选项，看一下它的作用。

在UNIX中，make工具从名字为makefile或Makefile的文件中读取指令。一些程序员喜欢用makefile命名，是因为在使用ls命令时它就出现了。由于MS-DOS操作系统是不区分大小写的，所以两种写法的区别就不存在了。

在UNIX中，makefile中的每个操作行必须用一个跳格符开始。在MS-DOS中，它可以用一个或多个空格或跳格符开始。

在很多操作系统中，可把写到stdout中的命令或程序的输出“输送”到从stdin读的另一个命令或程序中去，使第一个命令的输出成为第二个命令的输入。例如在MS-DOS中，命令dir把当前的目录写到stdout中，不带有其他命令行参数的命令sort从stdin中读并写到stdout中去。通过发命令

```
dir | sort
```

把dir的输出输送到sort的输入。符号|代表“管道”。在UNIX中，用ls替换dir，就可以使用命令ls | sort。把一个命令的输出作为另一个命令的输入是把软件工具结合在一起工作的一种方法。

在本书中，我们用bc和bcc作为调用Borland C编译器的命令。然而，在1991年以前，用tc和tcc（t代表Turbo）命令调用Borland编译器。

### 小结

- 通过调用system()，可以在程序内部执行操作系统的命令。在MS-DOS中，语句

```
system("dir");
```

把目录表和文件列在屏幕上。在UNIX中，只是把dir换成了ls。

- 程序员可以用getenv()访问系统的环境变量。
- 大多数C编译器都有很多的选项。如果你正在做一项重要的工作，你就要学会你所用的编译器中的各种可用的选项。要在编译器中把警告的级别设置为最高，并关注所有的警告。
- 很多操作系统都提供了创建和管理库的工具。在UNIX中，这样工具是archiver（归档程序），用ar命名调用它。在MS-DOS中，这样的工具是librarian库管理程序，它是附加功能部件。
- 使用库能节省程序员的时间和精力，但是，编译过程会变得复杂。可以用make工具来帮助管理编译过程。
- UNIX系统提供了prof工具，给出程序执行的档案。也可以经常使用其他的profiler。例如，在很多的UNIX系统中可找到Berkeley的gprof工具。在MS-DOS系统上经常可以使用profiler，尽管它是附加功能部件。
- 在很多系统中可以使用调试器。较新的调试器为用户提供了图形接口。调试器dbx的技术陈旧，但一般地它在UNIX系统上是可用的。对于程序员来说，这个工具可能是很有用的，尽管它并不出色。
- 可以用make工具记录源文件，并为访问库及相关的头文件提供便利。
- touch工具把新的时间放在文件上。
- 像diff和grep这样的工具具有一般的特征；不仅仅是程序员，每个人都可以使用它。在所有的UNIX中grep都是可用的，在很多的MS-DOS系统中它也是可用的。程序员通常把它用于各种任务。

## 练习

1. 编写一个能列出你的机器中的所有环境变量的程序。如果你使用的是UNIX系统，就发出命令

```
setenv ABC "Try me!"
```

该命令设置环境变量ABC。在MS-DOS中，这样的命令是

```
set ABC="Try mw!"
```

再次运行你的程序。列出的新环境变量是按字母次序排列的吗？或者是在尾部吗？你在作为赋给环境变量的值的串中怎样得到双引号？

2. 如果你使用UNIX，就发出命令

```
man sort
```

阅读sort工具。然后做一个实验，看下面的程序能做什么：

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

#define MAXSTRING 100

int main(int argc, char **argv)
{
    char command[MAXSTRING];

    sprintf(command, "sort -r %s", argv[1]);
```

```

    system(command);
    return 0;
}

```

实际上,需要对程序进行改善。重新编写它,使得它能给用户一个提示。

3. 你能使用Turbo C编译器吗? 如果能,试一下命令

```
tcc
```

这个命令本身创建一个所有选项的列表,并显示在屏幕上。这是一个非常好的功能部件。你知道这样的选项有多少吗?

4. 你已经知道了如何编写冒泡排序程序。现在学习如何编写快速排序程序,请参见Al Kelley和Ira Pohl编著的《*A Book on C*》(Menlo Park, Calif.: Benjamin/Cummings, 1990)。用profiler查明这样的每一个排序例程的效率。你可能发现prof提供的信息是很有趣的。就效率而论, qsort()和置换排序差不多,但是qsort()和quicksort()差别却很明显。

5. 在MS-DOS中,清除屏幕的命令是cls;在UNIX中是clear。试一下清除屏幕的命令,以理解它的作用。修改我们在4.11节“模拟:正反面游戏”中提到的程序heads\_or\_tails,使得它在程序的开始要清除屏幕。用函数调用system("cls")和system("clear")完成此项工作。如果把输出重定向会发生什么情况? 仍会清除屏幕吗?

6. 在本章最后提到的makefile是真实的。尽管我们仔细地对它进行了分析,但是对于使用make没有经验的人来说,可能感到其概念是难以掌握的。如果你有该工具的新版本,在创建下面的文件后,试一下make命令。

```

# Experiment with the make command!

go: hello date list
    @echo Goodbye!
hello:
    @echo Hello!
date:
    @date; date; date
list:
    @pwd; ls

```

如果把@字符去掉,会发生什么情况? 如果存在名字为hello或date的文件,会发生什么情况?

7. 创建下面的文件,然后发出命令make,会显示什么结果? 先写出你的答案,然后做实验检查你的答案。

```

# Experiment with the make command!

Start: A1 A2
    @echo Start
A1: A3
    @echo A1
A2: A3
    @echo A2
A3:
    @echo A3

```

8. (高级)像很多工具一样,UNIX操作系统上也有一些游戏。由于磁盘空间是珍贵资源的缘故,很多计算中心都把游戏删除了。然而在系统中留一个提供随机财富的游戏是一种常见的做法。有时系统管理员要安排所有用户在退出前得到财富。(如果你把财富命令(fortune)放在你的.logout文件中,你就能安排这一点。)如果你用UNIX,你就用一些时间试一下下面



的命令:

```
fortune
```

如果没有把这个命令从系统中删掉,那么每次调用这个命令,它都会在屏幕上显示不同的财富。如果你发出了这个命令,并看到了你喜欢的财富,你能捕获它吗?(或许你想把它送给你的母亲,或通过电子邮件把它发送给在巴黎的朋友。)如果你再次发出`fortune`命令,你会得到一个新的与以前不同的财富。在本练习中,你要编写一个能捕获特定财富的程序。

先阅读联机手册上的`fortune`命令。你可以通过命令

```
man for tune
```

来实现这一点。在尾部,你能看到存储所有财富的文件名。在大多数系统中,这个文件对外是不可读的。我们从经验中得知,一个财富用以下行开始:

```
There are three possible parts to date, of which
```

用如下行结束

```
-Miss Manners' Guide to Excruciatingly Correct Behaviour
```

你的程序应该交互式地要求用户输入要寻找的短语。假定在提示后用户输入“three possible parts to a date”。你的程序应该重复地做以下的工作:

- 1) 用函数调用`system("fortune > tmp")`捕获文件中的财富。
- 2) 搜索文件,看是否含有所要求的短语。
- 3) 如果有,保存这个文件,退出循环,发送一个电子邮件,通知用户。

程序可能要持续一会,因而它应该运行在后台。这是为什么要用电子邮件发通知的原因。下述的代码可用于通过电子邮件向用户发送信息:

```
char    command[MAXSTRING], file_name[MAXSTRING],
        message[MAXLINE], *user = getenv("USER");

sprintf(message, "%s\n%s%s\n\n",
        "Found a fortune!",
        "It was saved in the file ", file_name);
sprintf(command, "echo \"%s\" | mail %s",
        message, user);
system(command);
```

9. 用“优质打印机”作为调试工具这个事实还没有被人们普遍地接受。把下面的代码放在文件`try_cb.c`中:

```
#include <stdio.h>

int main(void)
{
    int    a = 1, b = 2;

    if (a == 1)
        if (b == 2)
            printf("***\n");
    else
        printf("###\n");
    return 0;
}
```

然后发出命令

```
cb < try_cb.c
```

你能看到“优质打印机”`cb`按不同的方式排列了这段代码。请解释这是为什么。

## 第15章 从C到C++

我们用术语抽象数据类型(abstract data type, ADT)和面向对象的编程(object-oriented programming, OOP)来谈论新的功能更强大的编程方法。ADT是用户定义的对语言中存在的数据类型的扩展,它由一组值和一组作用在这些值上的操作组成。例如,Pascal语言和C语言没有复杂的数据类型,但是C++提供了类结构来增加这样一个复杂类型并且对已存在的类型集成。

对象(object)是类的变量。在OOP中创建和使用ADT是很容易的。OOP用继承(inheritance)机制从已存在的用户定义类型派生出新类型是很方便的,程序员在对问题域中的对象建模后,使用OOP对类的对象的内容和行为进行编程。

在C++中的新类结构提供了实现ADT的封装(encapsulation)机制。封装把类型的内部实现细节、外部可用的操作和能作用到该类型的对象上的函数包装起来。一个类的实现细节对使用该类的代码段是不可访问的。例如,可以把栈实现为固定长度的数组,而外部可用的操作包括入栈(push)和出栈(pop)。改变内部实现中的链表不会影响外部使用的入栈和出栈。把使用ADT的编码称为ADT的客户(client)代码。栈的实现对其客户是隐藏的。

### 15.1 为什么转到C++

十多年来在计算机工业界C一直是一种可选的语言。因为C++是基于C的,它保留了C的很多内容,包括丰富的运算符集、几近正交的设计、精练性和可扩展性。C++是一种高度可移植的语言,在很多机器和系统中都存在转换它的程序。C++编译器和已有的C程序是高度兼容的,因为维护这种兼容性是设计的目标。与其他的面向对象语言不同(例如Smalltalk)C++是一种对广泛使用在众多机器上的C语言的扩展。

在面向对象语言中,C++与Smalltalk相比是相对便宜的。用C++编程不需要图形环境,C++程序不会招致类型检查或无用单元收集方面的运行代价。

C++在一些有意义的方面改善了C,特别是在支持强类型方面。像现在标准C(ANSI C)所要求的那样,函数原型语法是C++的创新。与C相比,C++拥有较强的类型规则,这使得它是一种较安全的语言。

C++在水平上兼顾了高低。C是一种系统实现语言,更接近于机器。C++增加了面向对象特征,这允许程序员创建或导入适合于问题域的库。用户可以在针对问题的层次上编写代码,还能与机器层次上的实现细节保持联系。基于运算符参数的类型可以对运算符进行新的定义。运算符重载支持新类型的实现,像运算符一样,正规的函数可以被重载。

C++对预处理器依赖得较少。C程序经常使用预处理器处理常量和有用的宏。然而,参数化的#define宏引入了不安全性。在C++中,关键字inline要求编译器产生内联代码,在系统开销上,函数不会多于宏。内部函数有精确的语义和静态函数的语法。可对其进行类型检查,与标准类型比较,这会增加运行性能。在C++中,可用const类型修饰符说明对象是不可修改的,这样就减少了对预处理器的依赖。

还有大量的其他改善。程序员可以用符号//表示注释行。C++提供的新I/O库iostream.h比stdio.h更有用。用new和delete运算符访问自由内存更加便利。

在C++中通过class机制实现抽象类型。类允许程序员控制实现的可见性,即公共的是可访问的,私有的是隐藏的。数据隐藏是面向对象编程的一个组成部分。类有成员函数,包括那些重载的运算符。使用成员函数,程序员能对ADT的适当功能编码。可以通过继承机制定义类,这样提高了代码的共享程度,有利于开发库。继承是面向对象编程的另一个特点。

人们经常批评C在输入方面较弱,是一种不安全的语言,然而C++在输入方面却是很强的。只要类型被很好地定义,C++允许类型间的转换。事实上,它允许程序员创建任意类型间的转换函数,但在两个非类的类型间是不允许的。

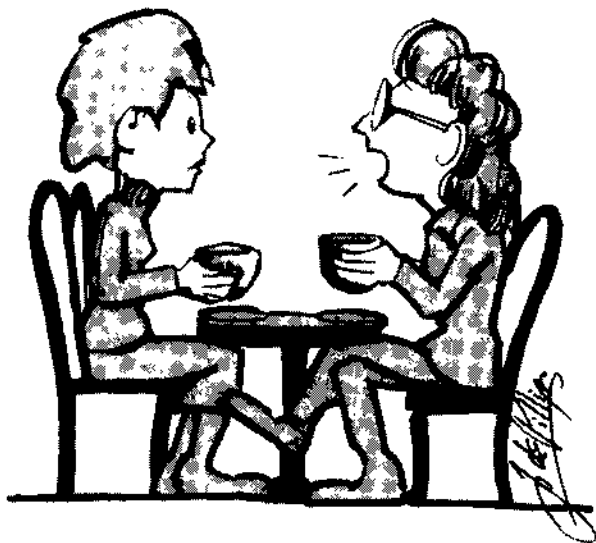
大体上,在语义的定义上C++比C严格得多。例如,类型转换和输入的实现是非常谨慎的。在C中对预处理器可扩展性的需求被缩减,可以用函数重载和内联替换带有参数的宏,const修饰符对于大多数被命名的常量来说是足够的了。C++主要保留了预处理器的文件引入用法和条件编译用法。

C++支持面向对象的编程风格。它与结构化编程风格相比有了较大的进步,但增加了语言的复杂性,这是付出的主要代价。按照C++的动机,这是正常的。虽然从C的成就中受了益,并且在增加改进的接口模式方面超过了要付出的代价,但是C++在C上增加的复杂性也是C++的最大缺陷之一。虽然增加的内容反映了大量的必要的新思想,但是这样会增加人们掌握它的难度。

C程序员要学习C++有几个原因。

C程序员为什么要学习C++

- C++和C一起都越来越多地得到了使用。
- 用较少的努力,C程序员就能利用几个增加的功能。
- 在大学和工业界,C++得到了越来越广泛地使用,特别是在先进的应用中,它正在取代C。



“我不知道,阿什利,但是即使就C程序员来说,我认为达德利在面向对象方面还差很多。”

本章不能给出太多的OOP概念的含义。关于更多的介绍, 请参看Ira Pohl编著的《*Object-Oriented Programming Using C++*》(Redwood City, Calif.: Benjamin/Cummings, 1993)。

## 15.2 类和抽象数据类型

C++的新颖之处在于它的聚集类型类(class)。类是对传统C中的结构思想的扩展。这与Pascal语言中的record类型相似。类提供了实现用户定义数据类型和相关的函数及操作码的手段。因此, 可以用类实现ADT。让我们编写一个称为string的类, 它实现了串的有限形成。

```
//An elementary implementation of type string.
#include <string.h>
#include <iostream.h>

const int max_len = 255;

class string {
public:           //universal access
    void assign(const char* st)
        { strcpy(s, st); len = strlen(st); }
    int length() const { return len; }
    void print() const
        { cout << s << "\nLength: " << len << endl; }
private:       //restricted access
    char s[max_len]; //implement as a character array
    int len;
};
```

在C++中, 类比C中的struct多了两个成分, 像在本例中看到的那样, 一个是本身为函数的成员, 如assign(), 另一个公共的和私有的成员。关键字public指出了对外的接口。没有这个关键字, 就表明成员是本类私有的。仅在同一个类中的其他成员可以访问私有成员。在类的声明范围内的任何函数都可以访问公共的成员。私有允许类隐藏其部分实现, 这能防止对其数据结构的任意修改。对访问加限制或隐藏数据是面向对象编程的一个特征。

通过成员函数声明, ADT可拥有能操作其私有成分的函数。例如, 成员函数length()返回串长(定义串的字符数, 但不包含第一个0值字符), 成员函数print()输出串及其长度, 成员函数assign()把一个字符串存储到隐藏变量s中, 并计算串的长度, 把值存储在隐藏变量len中。把不修改成员变量值的成员函数声明为const的。

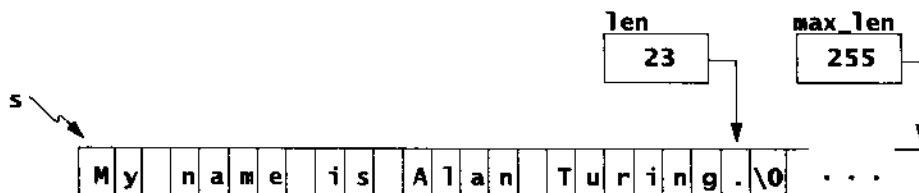
现在我们可以用数据类型string作为语言的基本类型。使用这个类型的代码是该类型的客户(client)。客户仅能使用对string类型的变量进行操作的公共成员。

```
//Test of the class string.

int main(void)
{
    string one, two;
    char three[40] = {"My name is Charles Babbage."};

    one.assign("My name is Alan Turing.");
    two.assign(three);
    cout << three;
    cout << "\nLength: " << strlen(three) << endl;
    //print the shorter of one and two
    if (one.length() <= two.length())
        one.print();
    else
        two.print();
    return 0;
}
```

变量one和two的类型是string, 变量three的类型是指向char的指针并且与string不兼容。下图显示出了two的len、max\_len和s的内容以及它们之间的关系。



本例的输出为:

```

My name is Charles Babbage.
Length: 27
My name is Alan Turing.
Length: 23
  
```

## 15.3 重载

重载(overloading)使得一个运算符或函数拥有几个含义。选择的含义要依赖运算符或函数使用的参数类型。让我们对上一个例子进行重载。下面是print函数的第二个定义:

```

class string {
public:
    .....
    void print() const
    { cout << s << "Length: " << len << endl; }
    void print(int n) const
    { for (int i = 0; i < n; ++i)
        cout << s << endl; }
    .....
}
  
```

这个print()版本有一个类型为int的参数, 它显示串n次。

```

three.print(2);      //print string three twice
three.print(-1);     //string three is not printed
  
```

对C++的大多数运算符都可进行重载。例如, 我们重载+, 用以拼接两个串。要做到这点, 我们需要两个新关键字: friend和operator。关键字operator放于运算符标记之前, 占据函数声明中的函数名位置。关键字friend使得一个函数可以访问一个类变量的私有成员。friend函数不是类的成员, 但它与一个类中声明的成员函数拥有同样的权限。

```

//Overloading the operator+.

#include <string.h>
#include <iostream.h>
const int max_len = 255;

class string {
public:
    void assign(const char* st){ strcpy(s, st); len = strlen(st); }
    int length() const { return len; }
    void print() const { cout << s << "\nLength: " << len << endl; }
    friend string operator+(const string& a, const string& b);
private:
    char s[max_len];
    int len;
};

//overload +
string operator+(const string& a, const string& b)
  
```

```

{
    string temp;
    temp.assign(a.s);
    temp.len = a.len + b.len;
    if (temp.len < max_len)
        strcat(temp.s, b.s);
    else
        cerr << "Max length exceeded in concatenation."
              << endl;
    return temp;
}

void print(const char* c) //file scope print definition
{
    cout << c << "\nLength: " << strlen(c) << endl;
}

int main(void)
{
    string one, two, both;
    char three[40] = {"My name is Charles Babbage."};

    one.assign("My name is Alan Turing.");
    two.assign(three);
    print(three);           //file scope print called
                            //Print shorter of one and two
    if (one.length() <= two.length())
        one.print();       //member function print called
    else
        two.print();
    both = one + two;      //plus overloaded as concatenate
    both.print();
    return 0;
}

```

### 对函数operator+(())的解析

- string operator+(const string& a, const string& b);

该语句对加进行了重载。它的两个参数是串，对参数的调用是引用调用。*type& identifier* 声明了一个被引用变量的标识符。使用const说明对参数不能修改。像Pascal这样的语言都支持引用调用。

- string temp

这个函数要返回类型为string的值。局部变量temp用于存储并返回拼接的字符串。

- temp.assign(a.s);  
temp.len = a.len + b.len;  
if (temp.len < max\_len)  
 strcat(temp.s, b.s);

通过调用string::assign(), 把串a.s复制到temp.s, 这要用到库函数strcpy()。测试拼接后的串的长度, 看它是否超过了串的最大长度, 如果没有超过, 就用隐藏的串成员temp.s和b.s调用标准库函数strcat()。因为这个函数是类string的友员, 所以对temp.s、a.s和b.s的引用是允许的。

- cerr << "Max length exceeded in concatenation."  
 << endl;

用标准错误流cerr显示错误信息, 不进行拼接。返回的仅是第一个串。

- return temp;

该运算符的返回类型是string, 把temp赋值给适当的被连接的串。 ■

对三元条件运算符`?:`、作用域解析运算符`::`、成员运算符`."`和成员运算符`.*`不能重载。

## 15.4 构造器和析构器

按OOP术语,把变量称为对象(object)。构造器(constructor)是一个用于初始化类的对象的成员函数。在很多情况下这会涉及到动态存储分配。在创建类的对象时,要调用构造器。析构器(destructor)是一个成员函数,它的工作是终结类的变量。在很多情况下这会涉及到对动态存储的解除。在自动对象超出作用域时,系统会隐式地调用析构器。

我们要对上一节的例子进行改造,为每个string变量自动地分配存储空间。用指针替换私有的数组变量,利用new运算符,改造后的类用构造器动态地分配适当的存储空间。

```
//An implementation of dynamically allocated strings.

class string {
public:
    //constructor
    string(int n) { s = new char[n + 1]; len = n; }
    void assign(const char* st)
        { strcpy(s, st); len = strlen(st); }
    int length() const { return len; }
    void print() const
        { cout << s << "\nLength: " << len << endl; }
    friend string operator+(const string& a,
                           const string& b);

private:
    char* s;
    int len;
};
```

构造器名与类名相同。通常构造器使用new为各初始对象分配存储空间。关键字new是一元运算符,它用一个数据类型做参数,它为存储这种类型分配适当数量的内存,并返回相应的指针值。在上面的例子中,要从自由存储中分配n+1个字节。这样,声明

```
string a(40), b(100);
```

要为a.s指向的变量a分配41个字节,为b.s指向的变量b分配101个字节。我们增加了一个字节,用于存放串尾值0。由new获得的存储空间是永久的,在块退出后不会自动地归还。若希望归还存储空间,类中必须要有析构函数。析构器与普通函数一样,只是它的名字与类的名字一样且其前带有~。析构器通常使用一元运算符delete自动地释放与指针表达式相关的存储空间。

```
//Add as a member function to class string.
~string() { delete []s; } //destructor
```

通常要对构造器重载,编写各种各样的适应多种初始化风格的函数。例如,用指向char值的指针初始化串。这样的构造器应该是:

```
string(const char* p)
{
    len = strlen(p);
    s = new char[len + 1];
    strcpy(s, p);
}
```

通常应该用下面声明调用这个版本的构造器:

```
char* str = "I came on foot.";
```

```
string a("I came by bus."), b(str);
```

也可能要使用无参数的构造器, 例如,

```
string() { len = 255; s = new char[255]; }
```

它将由不带括号参数的声明调用, 并按缺省要分配255个字节的内存。在下面的声明中要调用这三个构造器:

```
string a, b(10), c("I came by horse.");
```

通过声明的形式选择要重载的构造器。变量a没有参数, 因而要分配255个字节的内存。变量b的参数是整型的, 对构造器的调用要分配11个字节的内存。变量c的参数是串"I came by horse", 对构造器的调用要分配17个字节的内存, 该串要被复制到它的私有成员s中去。

## 15.5 继承

OOP的核心是对一组适当的数据类型及其操作的封装。带有成员函数和数据成员的类结构提供了适当的编码工具。类变量是要被操作的对象。Pascal语言用类型record封装变量。

类也提供了数据隐藏机制。可以对需要访问实现细节的函数的访问权限进行管理和限制, 这样可以促进模块性和健壮性。

OOP的另一个重要概念是通过继承机制促进代码复用。这是一种从已存在的类派生新类的机制, 已存在的类也被称为基类(base class)。为了创建新类, 可对基类进行添加或修改。这样就会创建共享代码的相关数据类型的层次。

很多类型彼此间是不同的, 为一种类型开发新代码经常是冗长乏味的, 并且是易出错的。派生类继承了基类的描述。层次是处理复杂性的一种方法。例如, 元素周期表中有气体元素, 这些气体元素具有共同的性质。惰性气体是一种重要的分类。像氩这样的惰性气体是气体, 这就是一个层次。这为理解惰性气体的行为提供了一个便利的途径。我们知道它们都是由质子和电子组成的, 这样的性质所有的元素都具有。我们还知道在室温下它们都处于气态, 所有的气体也都具有这样的性质。在一般的化学反应中它们不会与其他的元素相结合, 这是所有惰性元素所共同具有的性质。

我们设计一个用于学校管理的数据库。注册人员必须记录不同种类的学生。其中主要两类是毕业的和未毕业的大学生。我们要开发一个用于描述学生的基类。

### OOP设计方法

- 1) 确定一组正确的类型。
- 2) 以代码的形式, 用继承设计它们间的关系。

一个派生类的例子如下:

```
enum support { ta, ra, fellowship, other };
enum year { fresh, soph, junior, senior, grad };

class student {
public:
    student(char* nm, int id, double g, year x);
    void print() const;
private:
    int    student_id;
    double gpa;
    year   y;
```



```

    char    name[30];
};

class grad_student : public student {
public:
    grad_student(char* nm, int id, double g,
                 year x, support t, char* d, char* th);
    void print() const;
private:
    support  s;
    char     dept[10];
    char     thesis[80];
};

```

在这个例子中, grad\_student是派生类, student是基类。在派生类标题的冒号后用关键字public意味着grad\_student继承student的公共成员作为自己的公共成员。在派生类中不能访问基类的私有成员。公共继承也意味着派生类grad\_student是student的子类。

继承结构为整个系统提供一种设计。例如, 含有一个学校中的所有成员的数据库可以从基类person中派生出来。基类student能用于派生出法学系学生。类似地, person可以是各种雇员类别的基类。

## 15.6 多态性

C++中有很多种形式的多态(polymorphic)函数或运算符。例如, 在C++中, 除法运算符是多态的。如果除运算符的参数是整数, 就用整数除法; 如果其中的一个或两个参数是浮点数, 就用浮点数除法。

在C++中, 函数名或运算符是可重载的。基于函数的参数表中的各个参数类型(称为特征标记(signature))调用函数。

例如, 在除法表达式

```
a / b    //type is determined by native coercions
```

表达式的结果依赖自动转换后的参数。如果两个参数是整数, 结果就是整数。如果其中的一个或两个参数是浮点数, 结果就是浮点数。

另一个多态的例子是输出语句

```
cout << a; //polymorphism via function overloading
```

这里的移位运算符<<调用的函数能输出类型为a的对象。如果a是一个整数, 输出的就是整数。如果a是一个浮点数, 输出的就是浮点数。

多态性局部化行为责任。在用ADT提供的对代码的改善措施对系统增加功能时, 经常不需对客户代码做任何改动。

为了提供一种ADT形状(shape)而实现的例程包的技术依赖对形状的结构综合描述。例如,

```

struct shape {
    enum { CIRCLE, RECTANGLE, ..... } e_val;
    double center, radius;
    .....
};

```

就有为系统中当前可绘制的任何形状所需的所有成员。它还含有枚举元值, 因而可以识别它。可把面积例程编写为:

```
double area(shape* s)
{
    switch(s -> e_val) {
        case CIRCLE: return (PI * s -> radius * s -> radius);
        case RECTANGLE: return (s -> height * s -> width);
        ....
    }
}
```

在这段修改过的代码中要引入新的形状会涉及到什么？要在这段代码中增加一个case，并在结构中增加一个成员。不幸的是，这样会发生影响到整个代码的连锁反应，因为每个被结构化的例程都必须增加一个case。从而，在概念上的局部改善引起了全局的变化。

在C++中用于处理该问题的一种OOP编码技术是使用形状层次。层次是明显的，圆(circle)和矩形(rectangle)从形状(shape)派生。这样只在新的派生类中修改代码，即在局部增加新的描述。在这种情况下，程序员对含义发生改变的例程（即对新面积的计算）进行重载。不使用新类型的客户代码不受影响；由于新类型而被改变的客户代码受的影响是很小的。

遵循这样设计的C++代码用shape作为抽象基类(abstract base class)。抽象类是含有一个或多个纯虚函数的类的。

```
//shape is an abstract base class.
class shape {
public:
    virtual double area() = 0;           //pure virtual function
};

class rectangle : public shape {
public:
    rectangle(double h, double w) : height(h), width(w) { }
    double

area() { return (height * width); }     //override
private:
    double height, width;
};

class circle : public shape {
public:
    circle(double r) : radius(r) { }
    double area() { return ( 3.14159 * radius * radius); }
private:
    double radius;
};
```

计算任意面积的客户代码是多态的。在运行时决定选适当的area()函数。

```
shape* ptr_shape;
....
cout << " area = " << ptr_shape -> area();
....
```

现在想象一下，开发一个square类改善类型的层次。

```
class square : public rectangle {
public:
    square(double h) : rectangle(h,h) { }
    double area() { return (rectangle::area()); }
};
```

客户代码保持不变。使用非面向对象编程，情况就不是这样了。

## 15.7 模板

C++用关键字`template`提供参数多态性(parametric polymorphism)。参数多态性使得相同的代码可用于不同的类型，其中的类型是代码体的参数。所编写的代码具有一定的普遍性。这种技术的一个重要的用途是编写容器类(container class)。容器类用于包含特殊类型的数据。栈、向量、树和表都是标准容器类的例子。下面我们开发一个容器类`stack`作为参数化类型。

```
//template stack implementation
template <class TYPE>
class stack {
public:
    stack(int size = 1000) : max_len(size)
    { s = new TYPE[size]; top = EMPTY; }
    ~stack() { delete []s; }
    void
    reset() { top = EMPTY; }
    void push(TYPE c) { s[++top] = c; }
    TYPE pop() { return s[top--]; }
    TYPE top_of() { return s[top]; }
    boolean empty() { return boolean(top == EMPTY); }
    boolean full() { return boolean(top == max_len - 1); }
private:
    enum {EMPTY = -1};
    TYPE* s;
    int max_len;
    int top;
};
```

这个类声明的语法的开始部分是

```
template <class identifier>
```

上边的`identifier` (标识符)是模板的参数，在本质上它可以是任何类型。在整个类的定义中，可以把模板参数用作类型名。在实际声明中把这个参数实例化。下面对`stack`的声明就是这样的：

```
stack<char>      stk_ch;           // 1000 char stack
stack<char*>     stk_str(200);     // 200 char* stack
stack<complex>   stk_cplx(100);    // 100 complex stack
```

这种机制节省了我们重新编写类声明的时间，在这里变化的仅是类型声明。

在处理这样的类型时，代码总是把尖括号作为声明的一部分。下面给出使用`stack`模板的两个函数：

```
//Reversing a series of char* represented strings
void reverse(char* str[], int n)
{
    stack<char*> stk(n);           //this stack holds char*
    for (int i = 0; i < n; ++i)
        stk.push(str[i]);
    for (i = 0; i < n; ++i)
        str[i] = stk.pop();
}
```

在函数`reverse()`中，`stack<char*>`用于插入`n`个串，然后按相反的次序出栈。

```
//Initializing a stack of complex numbers from an array
void init(complex c[], stack<complex>& stk, n)
{
    for (int i = 0; i < n; ++i)
        stk.push(c[i]);
}
```

在函数`init()`中，通过引用传递`stack<complex>`变量，把`n`个复数入栈。

## 15.8 C++中的异常

C++中的异常处理机制对上下文是敏感的。引起异常的上下文是测试块。在测试块结束的地方可找到用关键字catch声明的处理程序。

使用throw表达式引发异常。通过调用在测试块后的处理程序列表中选取的一个适当的处理程序处理异常。下面是一个例子：

```
//stack constructor with exceptions
stack::stack(int n)
{
    if (n < 1)
        throw (n);           //want a positive value
    p = new char[n];          //create a stack of characters
    if (p == 0)               //new returns 0 when it fails
        throw ("FREE STORE EXHAUSTED");
    else
        top = EMPTY;
        max_len = n;
}

void g()
{
    try {
        stack a(n), b(n);
        .....
    }
    catch (int n) {.....}    //an incorrect size
    catch (char* error) {.....} //free store exhaustion
}
```

第一个throw()有一个整数参数，与catch(int n)匹配。这个处理程序应该执行一个适当的行为，此处把一个不正确的数组尺寸传递给了构造器。例如，显示错误信息并异常终止就是适当的行为。第二个throw()有一个指向char型参数的指针，与catch(char\* error)匹配。

## 15.9 面向对象编程的益处

经常用OOP编程比用像C或Pascal那样的一般过程化语言编程要难。在我们开始编写代码之前，至少还需要多做一步设计，即要设计适合要解决的问题的类型。

经常对问题的解决比实际需要更具有概括性。想法是以几种方式分解概括性的解决方法，该解决方法提高了封装性、健壮性并且对代码也更易于维护和修改，此外还提高了可复用性。例如，如果代码中需要一个栈，该栈就可以从已存在的代码中借用。在一般的过程性语言中，这样的数据结构经常固定在算法中，不能被导出。

对于很多人来说，试图定义OOP就像盲人摸象一样。下面的等式对OOP做了较综合的描述：

*OOP = 类型的可扩展性 + 多态性*

## 15.10 风格

C++中有两种注释风格：新的一种是用//作为注释的开始，注释的范围是本行；旧的风格是/\*old style\*/。无疑C++程序员喜欢新风格。与其他风格的注释相比，新风格的注释易写易读。

在本章中，我们大都遵循了把\*和&紧靠着类型书写的风格，这种风格在C++中是很常见

的。例如，写`char* c`，而不写`char *c`；写`float& x`，而不写`float &x`。

在C++中，无参数的函数说明与用`void`作参数的说明是等价的。例如，`int f();`等价于`int f(void);`

## 15.11 常见的编程错误

像使用其他新语言一样，可能会有一些错误。这里我们要提及两种可能是难以调试的错误。第一种错误由`#define`中的注释引起：

```
#define LIMIT 77          // danger - watch out!

for (i = 0; i < LIMIT; ++i)
    .....                // do something
```

这个错误是与系统相关的。一些预处理器会把注释去掉，而有些预处理器则不这样。如果预处理器不把这样的注释去掉，就会把出现的`LIMIT`都替换成`77 //danger-watch out!`。这样，`for`循环就变成

```
for (i = 0; i < 77          // danger - watch out! ; ++i)
    .....
```

自然编译器会报告有错误出现。不幸的是，由于此类错误的性质，编译器给出的错误信息对程序员不会有多大的帮助。

在计算中，程序员经常重载`^`运算符，用`m^n`计算`m`的`n`次幂。例如，

```
cout << "2 raised to the power 3 is " << 2^3 << "\n";
```

程序员常常忘记`<<`的优先级高于`^`的优先级。同样，由于此类错误的性质，编译器给出的错误信息对程序员也不会有多大的帮助。正确的写法应该是

```
cout << "2 raised to the power 3 is " << (2^3) << "\n";
```

## 15.12 系统考虑

ANSI C借用了C++的一些结构，包括函数原型和`const`限定符。考虑下面的代码：

```
const int n = 7;

int a[n];
```

在C++中这样写是合法的，但在C中是非法的。在C中数组的大小必须是常量。在C中，即使用`const`限定了`c`，它也是一个不可修改的变量，而不是常量。

ANSI C或许正在继承C++的另一个特征，即//注释风格。Borland和Microsoft的最新的编译器都支持新的注释风格。这种用法的很明显的例子是在支持视窗的Microsoft C代码中。这样的代码中有大量的注释，其中的大多数注释遵循了新的注释风格。

## 小结

- C++真正的新内容是它引入的聚集类型`class`。`class`是对传统C的`struct`思想的扩展。使用类是实现数据类型及其相关函数和运算符的一种方法。因此，类是对抽象数据类型(ADT)的实现。在`struct`思想上，C++还增加了两个新的内容：(1) 它引入了本

身是函数的成员；(2)它使用了新的关键字public(公共的)，这个关键字指定了它后面的成员的可见性，没有这个关键字，成员就是类私有的，仅类中的其他成员函数可使用私有成员。在类声明的范围内的函数都可以使用公共的成员。私有性允许类类型的部分实现隐藏起来。

- 术语重载(overloading)指的是给一个运算符或函数几个含义。所选择的含义依靠运算符或函数使用的参数的类型。
- 关键字operator引入运算符标记，替换函数声明中的函数名。用关键字friend赋予一个函数访问一个类变量的私有成员的权限。friend函数不是该类的成员，但具有声明它的类中的成员的权限。
- 构造器是成员函数，它的作用是对类的变量进行初始化。在很多情况下这要涉及到动态存储分配。在创建类的一个对象时，就调用该类的构造器。通常是在声明变量时调用构造器。
- 析构器是成员函数，它的作用是回收类变量所占用的资源。析构器回收动态分配给对象的存储空间，将其归还给内存。在块退出时，系统隐式地调用析构器回收在块内定义的所有类变量。
- 面向对象编程的核心是封装一组数据类型及其操作。这些用户定义的类型是ADT。带有成员函数和数据成员的结构提供了一个适当的编码工具。类变量是被操作的对象。
- OOP的另一个重要概念是通过继承机制促进代码复用。这是一种从已存在的类派生(deriving)新类的机制，已存在类也被称为基类(base class)。要创建派生类，可以对基类进行增改。按这种方式，可以创建相关数据类型的层次，以共享代码。通过虚(virtual)函数可以动态地使用这种分类层次。派生类重载基类中的虚成员函数。可以动态地使用虚成员函数。指向基类的指针也能指向派生类的对象，在用这样的指针指向重载的虚函数时，系统动态地选择调用哪一个成员函数。

## 练习

1. 采用下面的程序，轮流地删除每一行，每次都通过编译器运行它。记录每行删除引起的出错信息。

```
#include <iostream.h>

int main(void)
{
    int m, n, k;

    cout << "\nEnter two integers: ";
    cin >> m >> n;
    k = m + n;
    cout << "\nTheir sum is " << k << ".\n";
    return 0;
}
```

2. 编写一个交互式地输入用户的名字和年龄的程序，并输出

Hello name, next year you will be next\_age.

作为响应，这里的next\_age是输入的年龄加1。

3. 编写一个显示平方、平方根和立方的表。像下面那样，用跳格符或空格串把表排列整齐。

Integer	Square	Square root	Cube
1	1	1.00000	1
2	4	1.41421	8
.....			

4. 在ANSI C中, 传统的交换函数是

```
void swap(int *p, int *q)
{
    int tmp;

    tmp = *p;
    *p = *q;
    *q = tmp;
}
```

用C++的引用参数重新编写该函数, 然后编写一个程序测试它。注意, 在C中我们用p和q作为指针的标识符。使用引用参数, 用其他标识符更适当。例如,

```
void swap(int& i, int& j)
{
    .....
```

5. 为类string增加一个成员函数reverse() (请参见15.2节“类和抽象数据类型”)。该函数倒置私有成员s中的字符。

6. 为类string增加一个成员函数void print(int k)。这个函数重载print(), 它显示该串的前k个字符。

7. 重载类string中的运算符\*。其成员声明应该为:

```
string operator*(string& a, int n);
```

其要求是把a表示的string向a复制n次。要检查不要溢出内存。

## 附录A 标准库

标准库提供了一些程序员可用的函数。与标准库相关的是系统提供的标准头文件，这些头文件含有标准库中的函数的原型、宏定义和其他编程元素。如果程序员想用库中的特定函数，就必须引入相应的头文件。下面是一个包含全部头文件的列表。

C头文件			
<assert.h>	<limits.h>	<signal.h>	<stdlib.h>
<ctype.h>	<locale.h>	<stdarg.h>	<string.h>
<errno.h>	<math.h>	<stddef.h>	<time.h>
<float.h>	<setjmp.h>	<stdio.h>	

可以按任意次序引入这些头文件。多次引入它们的效果如同只引入一次。在本附录中我们将讨论头文件。

### A.1 诊断: <assert.h>

这个头文件定义了`assert()`宏。如果在引入<assert.h>的地方定义了`NDEBUG`，实际上就抛弃了所有的断言。

```
• void assert(int expr);
```

如果`expr`是零(false)，它就显示诊断信息，程序终止。诊断信息包括表达式、文件名和在文件中的行号。

### A.2 字符处理: <ctype.h>

这个头文件定义了几个用于测试字符参数的宏。它还含有两个用于映射字符参数的函数的函数原型。

#### A.2.1 测试字符

```
• int isalnum(int c);      /* is alphanumeric */
  int isalpha(int c);      /* is alphabetic */
  int iscntrl(int c);      /* is control */
  int isdigit(int c);      /* is digit: 0-9 */
  int isgraph(int c);      /* is graphic */
  int islower(int c);      /* is lowercase */
  int isprint(int c);      /* is printable */
  int ispunct(int c);      /* is punctuation */
  int isspace(int c);      /* is white space */
  int isupper(int c);      /* is uppercase */
  int isxdigit(int c);     /* is hex digit 0-9, a-f, A-F */
```

通常用宏实现这些字符测试，有关细节请参见相关资料中`ctype.h`。如果参数`c`满足测试要求，宏就返回非零值(true)，否则返回零值(false)。可以像使用函数那样使用这些宏。

打印字符是实现定义的，每个打印字符在屏幕上只占一个显示位置。除空格外，一个图形



字符可以是任意的打印字符，这样图形字符是在屏幕上占一个单显示位置的可视标记。标点符号是除了空格或使`isalnum(c)`为真的字符`c`以外的打印字符。标准空白字符有空格、换页(''\f'')、换行(''\n'')、回车(''\r'')、横向跳格(''\t'')和纵向跳格(''\v'')。控制字符有响铃(''\a'')和退格(''\b'')以及任何使`isspace(c)`为真的字符`c`（空格、Ctrl+c和Ctrl+h等除外）。

### A.2.2 映射字符

函数`tolower()`和`toupper()`用于映射字符参数。当心：很多较早版本的ANSI C编译器执行这两个函数会发生错误。

• `int tolower(int c);`

如果`c`是一个大写字符，就返回相应的小写字符，否则就返回`c`。

• `int toupper(int c);`

如果`c`是一个小写字符，就返回相应的大写字符，否则就返回`c`。

在ASCII机器上经常使用下面的三个宏。前两个宏与`tolower()`和`toupper()`有关，但并不一样。

```
#define _tolower(c) ((c) + 'a' - 'A')
#define _toupper(c) ((c) + 'A' - 'a')
#define toascii(c) ((c) & 0x7f)
```

十六进制常量`0x7f`是低7位的掩码。

### A.3 错误：<errno.h>

本节定义标识符`errno`以及几个用于报告错误状态的宏。

`extern int errno;`

通常在`errno.h`中有很多宏。在`errno.h`中究竟的那些宏是与系统相关的，但其中所有的宏必须要用`E`开头。各种库函数用这些宏来报告错误。

有两个宏是各种系统共有的，在库中的数学函数使用它们。

```
#define EDOM 33 /* domain error */
#define ERANGE 34 /* range error */
```

这两个宏也可以不用33和34这两个，不过通常都使用这两个值。

一个数学函数的域(domain)是定义该函数的参数集。例如，平方根函数的域是所有的非负整数。如果用不在域中的参数调用数学函数，就会发生域错误(domain error)。如果有这样的错误发生，系统就把值`EDOM`赋值给`errno`。程序员可以用`perror()`和`strerror()`显示与存储在`errno`的值相关的信息。

如果按数学定义函数的返回值，但不能表示成`double`，就会发生值域错误(range error)。如果有这样的错误发生，系统就把值`ERANGE`赋值给`errno`。

### A.4 对浮点数的限制：<float.h>

本节定义了用于定义各种浮点特征以及浮点界限的宏。这样的宏有很多，下面给出其中的一部分：

```

#define DBL_MAX      1.7976931348623157e+308
#define FLT_MAX      3.40282347e+38F
#define LDBL_MAX     1.7976931348623157e+308

#define DBL_MIN      2.2250738585072014e-308
#define FLT_MIN      1.17549435e-38F
#define LDBL_MIN     2.2250738585072014e-308

#define DBL_EPSILON  2.2204460492503131e-16
#define FLT_EPSILON  1.19209290e-07F
#define LDBL_EPSILON 2.2204460492503131e-16

```

常量是与系统相关的。我们假设把long double实现为double,但对一些系统是不能做这样的假设的。一些系统提供的精度和值域更大,请参见你的系统中的float.h。

## A.5 对整数的限制: <limits.h>

本节定义了用于定义各种整数特征以及整数界限的宏。这样的宏有很多,下面给出其中的一部分:

```

#define CHAR_BIT      8      /* number of bits in a byte */
#define CHAR_MAX      127
#define CHAR_MIN      (-128)
#define SHRT_MAX      32767
#define SHRT_MIN      (-32768)
#define INT_MAX        2147483647
#define INT_MIN        (-2147483648)

```

常量是与系统相关的。

## A.6 局部化: <locale.h>

这个头文件含有能用于设置或访问适合于当前场所的特征的编程结构。下面定义了这样的一个结构类型:

```

struct lconv {
    char    *decimal_point;
    char    *thousands_sep;
    char    *currency_symbol;
    .....
};

```

这些成员虑及到局部变量,例如,用逗号代替十进制的小数点。该头文件中至少定义了6个符号常量。

```

#define LC_ALL        1 /* all categories */
#define LC_COLLATE    2 /* strcoll() and strxfrm */
#define LC_CTYPE      3 /* character handling functions */
#define LC_MONETARY    4 /* monetary info in localeconv() */
#define LC_NUMERIC     5 /* decimal point in lib fcts */
#define LC_TIME       6 /* strftime() */

```

符号常量的值是与系统相关的。规定用LC\_开头的其他宏。这些宏可以用做setlocale()函数的第一个参数。

- char \*setlocale(int category, const char \*locale);

通常第一个参数是上述的符号常量之一。第二个参数是"C"、" "或其他一些串。如果函数可用,函数返回一个指向系统提供的静态串的指针,它描述了一个新场所;否则就返回一个指向NULL的指针。在程序启动时,系统的行为就好像是执行了

```
setlocale(LC_ALL, "C");
```

似的。这指定了C转换的最小环境。语句

```
setlocale(LC_ALL, "");
```

指定了本地的环境，这是与系统相关的。使用除了LC\_ALL之外的宏仅影响场所的一部分。例如，LC\_MONETARY仅影响处理金钱信息的场所的一部分。

- `struct lconv *localeconv(void);`

它返回一个指向由系统提供的结构的指针。它是静态的，并含有当前位置的数字信息。进一步调用函数`setlocale()`会改变存储在结构中的值。

## A.7 数学: <math.h>

这个头文件含有库中的数学函数的原型，它还含有一个宏定义：

```
#define HUGE_VAL 1.7976931348623157e+308
```

这个宏的值是与系统相关的。

数学函数的域是定义函数的参数集。在用不是域中的参数调用数学函数时，会发生域错误。如果这样的情况发生，函数返回的值是与系统相关的，系统把值EDOM赋给`errno`。

如果按数学定义了函数的返回值，但不能被表示成`double`，会发生值域错误。如果返回值太大（溢出），就返回HUGE\_VAL或-HUGE\_VAL。如果值太小（下溢），就返回0。在溢出的情况下，系统把宏ERANGE的值存储在`errno`中。在下溢的情况下会发生什么是与系统相关的。一些系统把宏ERANGE的值存储在`errno`中，但也有例外。

- `double cos(double x);`  
`double sin(double x);`  
`double tan(double x);`

它们分别是余弦、正弦和正切函数。

- `double acos(double x);` /\* arccosine of x \*/  
`double asin(double x);` /\* arcsine of x \*/  
`double atan(double x);` /\* arctangent of x \*/  
`double atan2(double y, double x);` /\* arctangent of y/x \*/

这些是反三角函数。它们返回的角度 $\theta$ 是弧度。函数`acos()`的值域是 $[0, \pi]$ 。函数`asin()`和`atan()`的值域是 $[-\pi/2, \pi/2]$ 。函数`atan2()`的值域是 $[-\pi, \pi]$ ，它的主要用途是把直角坐标变为极坐标。对于函数`acos()`和`asin()`，如果参数不在 $[-1, 1]$ 内，就会发生域错误。对于函数`atan2()`，如果两个参数都是0，不能表达 $y/x$ ，就会发生域错误。

- `double cosh(double x);`  
`double sinh(double x);`  
`double tanh(double x);`

它们分别是双曲线余弦、双曲线正弦和双曲线正切函数。

- `double exp(double x);`  
`double log(double x);`  
`double log10(double x);`

函数`exp()`返回 $e^x$ 。函数`log()`返回 $x$ 的自然对数（基为 $e$ ）。`log10()`返回基为10的 $x$ 的对数。对于后两个对数函数，如果 $x$ 是负数，就会发生域错误。如果 $x$ 是0，且系统不能表示0的对数（一些系统能表示），就会发生值域错误。

- `double ceil(double x);`  
`double floor(double x);`

`ceil`函数返回不小于x的最小整数。`floor`函数返回不大于x的最大整数。

- `double fabs(double x);` /\* floating absolute value \*/

它返回x的绝对值。当心：与它相关的函数是`abs()`，`abs()`的参数是整数，而不是浮点数，不要把二者混淆。

- `double fmod(double x, double y);` /\* floating modulus \*/

这个函数返回用y对x取模后的值。更明确地讲，如果y不是0，就返回 $x - i * y$ 的值，此处i是整数，它或者是0，或者是一个小于y并与x有相同符号的整数。如果y是0，那么返回值是与系统相关的，但通常的返回值是0。如果发生了y是0这样的情况，在一些系统中会发生域错误。

- `double pow(double x, double y);` /\* power function \*/

它返回x的y次幂。如果x是负数且y不是整数，就会发生域错误。

- `double sqrt(double x);` /\* square root \*/

它返回x的平方根，x应该是正数。如果x是负数，就会发生域错误。

- `double frexp(double value, int *exp_ptr);` /\* free exponent \*/

库中的其他函数要使用它。它把一个值分为尾数和指数。语句

```
x = frexp(value, &exp);
```

使得关系 $value = x * 2^{exp}$ 成立，这里x位于区间 $[1/2, 1]$ 中，或者x为0。

- `double ldexp(double x, int exp);` /\* load exponent \*/

它返回值 $x * 2^{exp}$ 。

- `double modf(double value, double *i_ptr);`

这个函数把value分为整数和小数部分。函数调用`modf(value, &i)`返回值f和间接的值i，使得 $value = i + f$ 。

## A.8 非局部跳转：<setjmp.h>

这个头文件提供了一个类型定义和两个原型。这些声明允许程序员做非局部跳转。非局部跳转与`goto`语句相似，但控制流要转移出当前函数之外。类型是与系统相关的。下面是一个例子：

```
typedef long jmp_buf[16];
```

用一个类型为`jmp_buf`的数组保持恢复调用环境的系统信息。

- `int setjmp(jmp_buf env);`

它把当前的调用环境保存在数组env中，以供`longjmp()`使用；它的返回值是0。虽然在很多系统中把它作为函数实现，但在ANSI C中，把它作为宏实现。

- `void longjmp(jmp_buf env, int value);`

函数调用`longjmp(env, value)`恢复由`setjmp(env)`最近存储的环境。如果没有对

setjmp(env)调用过,或如果调用过它的函数不再有效,那么函数调用longjmp(env, value)的行为是无定义的。成功的调用使得程序控制跳转到前次对setjmp(env)调用的后边。如果value不是0,该函数的效果就好像是再次调用setjmp(env)并返回value。如果value是0,该函数的效果就好像是再次调用setjmp(env)并返回1。

## A.9 信号处理<signal.h>

这个头文件包含程序员用于处理例外条件或信号的结构。在这个头文件中定义了以下宏:

```
#define SIGINT    2    /* interrupt */
#define SIGILL    4    /* illegal instruction */
#define SIGFPE    8    /* floating-point exception */
#define SIGSEGV   11   /* segment violation */
#define SIGTERM   15   /* asynchronous termination */
#define SIGABRT   22   /* abort */
```

常量是与系统相关的,很常用。通常C++还支持其他信号,请参见你的系统中的signal.h。可以把下面的宏用做函数signal()的第二参数。

```
#define SIG_DFL  ((void (*)(int)) 0)    /* default */
#define SIG_ERR  ((void (*)(int)) -1)    /* error */
#define SIG_IGN  ((void (*)(int)) 1)    /* ignore */
```

系统也可能提供其他这样的宏。宏的名字都用SIG\_开头,后续以大写字母。

- void (\*signal(int sig, void(\*func)(int)))(int);

函数调用signal(sig, func)把信号sig和信号处理器func()联系起来。如果调用成功,就返回前次带有第一个参数sig的调用的指针值func;如果没有调用过,返回值是NULL。如果调用不成功,返回指针值SIG\_ERR。

当信号sig出现时,函数调用signal(sig, func)指示系统调用func(sig)。如果signal()的第二个参数是SIG\_DFL,就会出现缺省的行为;如果第二个参数是SIG\_IGN,就忽略信号。在程序控制从func()返回时,程序控制返回到sig产生之处。

- int raise(int sig)

该函数产生信号sig。如果调用成功,就返回0,否则返回非0值。这个函数也用于测试目的。

## A.10 可变参数: <stdarg.h>

程序员使用这个头文件可编写像printf()这样的具有可变数目的参数的函数,以便于移植。这个头文件含有一个typedef和三个宏。怎样实现它们是与系统相关的,下面是一种实现方法:

```
typedef char * va_list;

#define va_start(ap, v) \
    ((void) (ap = (va_list) &v + sizeof(v)))
#define va_arg(ap, type)  (*((type *) (ap))++)
#define va_end(ap)       ((void) (ap = 0))
```

在宏va\_start()中,变量v是定义可变参数函数的头文件中的最后的参数。这个变量的存储类型不能是register,它也不能是数组类型或像char这样的被自动转换加宽的类型。宏va\_start()初始化参数指针ap。宏va\_arg()访问链表中的下一个参数。宏va\_end()在函数退出前完成所需要的清除工作。下面的程序说明了这些结构的用法:

```

#include <stdio.h>
#include <stdarg.h>

int va_sum(int cnt, ...);

int main(void)
{
    int a = 1, b = 2, c = 3;

    printf("First call: sum = %d\n", va_sum(2, a, b));
    printf("Second call: sum = %d\n", va_sum(3, a, b, c));
    return 0;
}

int va_sum(int cnt, ...) /* sum the arguments */
{
    int i, sum = 0;
    va_list ap;

    va_start(ap, cnt); /* startup */
    for (i = 0; i < cnt; ++i)
        sum += va_arg(ap, int); /* get next argument */
    va_end(ap); /* cleanup */
    return sum;
}

```

### A.11 常见的定义: <stddef.h>

这个头文件含有一些类型定义和常用于其他位置的宏。怎样实现它们是与系统相关的，下面是一种实现方法：

```

typedef char      wchar_t;
typedef int       ptrdiff_t;
typedef unsigned  size_t;

#define NULL      ((void *) 0)
#define offsetof(s_type, m) \
    ((size_t) &(((s_type *) 0) -> m))

```

此处，我们要把宽位字符类型wchar\_t定义为无格式char。系统可以把它定义为任意的整型。它必须能容纳所支持的各种场合下的最大的扩展字符集。类型ptrdiff\_t是在两个指针相减时获得的类型。类型size\_t是在使用sizeof运算符时获得的类型。形式为offsetof(s\_type, m)的宏调用计算成员m距结构s\_type开头的偏移字节数。下面的程序说明了它的用法：

```

#include <stdio.h>
#include <stddef.h>

typedef struct {
    double a, b, c;
} data;

int main(void)
{
    printf("%d %d\n",
        offsetof(data, a), offsetof(data, b));
    return 0;
}

```

在大多数系统中，这个程序显示0和8。

### A.12 输入/输出: <stdio.h>

这个头文件含有程序员用以访问文件的宏、类型定义和函数原型。下面是一些宏和类型

定义的例子：

```
#define BUFSIZ      1024    /*buf size for all I/O*/
#define EOF        (-1)    /*returned on EOF*/
#define FILENAME_MAX 255    /*max filename chars */
#define FOPEN_MAX   20     /*max open files*/
#define L_tmpnam     16     /*size tmp filename*/
#define NULL        0      /*null pointer value */
#define TMP_MAX      65535  /*max unique filenames*/

typedef long      pos_t;    /*used with fsetpos() */
typedef unsigned  size_t;   /*type from sizeof op*/
typedef char *    va_list;  /*used with vfprintf()*/
```

结构类型FILE中的成员描述了文件的当前状态。它的成员的名字和数目是与系统相关的。

下面是一个例子：

```
typedef struct {
    int      cnt;          /* size of unused part of buf */
    unsigned char *b_ptr;  /* next buffer loc to access */
    unsigned char *base;   /* start of buffer */
    int      bufsize;      /* buffer size */
    short    flag;         /* info stored bitwise */
    char     fd;           /* file descriptor */
} FILE;

extern FILE _iob[];
```

一个类型为FILE的对象应该能记录控制一个流的所有信息，其中包括文件位置指示器、指向相关缓冲区的指针、记录读/写错误是否发生的错误指示器以及记录是否达到文件尾标记的文件尾指示器。怎样实现这些是与系统相关的。例如，可以在结构成员flag中按位对错误指示器和文件尾指示器编码。

通常用

```
typedef long fpos_t;
```

给出类型fpos\_t。这个类型的对象应该能记录所需要的所有信息，以确切地指定文件中的各个位置。

宏用于定义stdin、stdout和stderr。虽然我们把它们看作是文件，实际上它们是指针。

```
#define stdin (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
```

与其他的文件不同，程序员不必显式地打开stdin、stdout和stderr。

一些宏要和函数一起使用。

```
#define _IOFBF 0    /* setvbuf(): full buffering */
#define _IOLBF 0x80 /* setvbuf(): line buffering */
#define _IONBF 0x04 /* setvbuf(): no buffering */
#define SEEK_SET 0   /* fseek(): file beginning */
#define SEEK_CUR 1   /* fseek(): current file pos */
#define SEEK_END 2   /* fseek(): EOF */
```

当打开一个文件时，操作系统把它与一个流(stream)联系起来，并在一个类型为FILE的对象中保存流的信息。可以认为指向FILE的指针是与文件或流相联系的。

### A.12.1 打开、关闭和调节文件

- FILE \*fopen(const char \*filename, const char \*mode);

它完成打开缓冲文件所需要的内务处理。成功的调用创建一个流，并返回与流相联系的指向FILE的指针。如果filename不可访问，就返回NULL。基本文件模式"r"、"w"和"a"分别对应于读、写和添加。如果文件模式是"w"或"r"，那么在文件的开始位置设置文件位置指示器；如果文件模式是"a"，那么在文件尾设置文件位置指示器。如果文件模式是"w"或"a"，并且文件不存在，那么就创建它。用"a+"指明修改模式（读和写）。用"b"指明二进制文件。例如，用模式"r+"打开文本文件，用于读和写。用模式"rb"打开二进制文件，用于读。用模式"rb"和"r+b"打开二进制文件，用于读和写。应用"w"和"a"也是类似的（请参见13.5节“访问文件”）。在修改模式下，输出不可以直接跟随着输入，除非达到了文件尾标记或对文件位置函数fseek()、fsetpos()或rewind()之一进行了调用。以类似的方式，输入不可以直接跟随着输出，除非对fflush()进行了调用，或对文件位置函数fseek()、fsetpos()或rewind()之一进行了调用。

- int fclose(FILE \*fp);

本函数清空缓冲区，并断开与fp相关的所有连接。如果文件被成功地关闭，就返回0，如果出现了错误，或文件已经被关闭，就返回EOF。打开的文件是有限制的资源，至多能同时打开FOPEN\_MAX个文件。如果仅在需要的时候打开文件，系统的效率会得到提高。

- int fflush(FILE \*fp)

刷新一个流。如果调用成功，返回0，否则返回EOF。

- FILE \*freopen(const char \*filename,  
const char \*mode, FILE \*fp);

该函数关闭与fp相关联的文件，按照mode打开文件filename，并把fp与新文件相关联。如果函数调用成功，就返回fp；否则返回NULL。对于改变与stdin、stdout和stderr相关的文件而言，这个函数是很有用的。

- void setbuf(FILE \*fp, char \*buf);

如果fp不是NULL，则函数调用setbuf(fp, buf)等价于

```
setvbuf(fp, buf, _IOFBF, BUFSIZ)
```

只是后者不返回任何信息。如果fp是NULL，那么模式是\_IONBF。

- int setvbuf(FILE \*fp, char \*buf, int mode, size\_t n);

该函数决定与fp相关的文件怎样被缓存。在文件被打开后且被访问前必须调用该函数。模式\_IOFBF、\_IOLBF和\_IONBF分别使文件为全缓存、线性缓存和无缓存。如果buf不是NULL，buf指向的尺寸为n的数组被用做缓存。如果buf是NULL，系统就提供缓存。成功的调用返回0。当心：如果把存储类型为自动的数组用做缓存，在函数退出之前应该关闭文件。

- FILE \*tempfile(void);

该函数用模式"wb+"打开一个临时文件，并返回一个与该文件相关的指针。如果要求不能满足，就返回NULL。在关闭文件或程序退出后，系统会删除该文件。

- char \*tmpnam(char \*s);

该函数创建一个惟一的临时名，通常该名被用做文件名。如果s不为NULL，该名就存储在c中，s的尺寸必须为L\_tmpnam或更大。如果s为NULL，系统提供一个静态的数组，以存储该名。进一步调用tmpnam()能重写这个空间。无论哪种情况，函数都返回存储名字的数



组的基地址。重复地调用`tmpnam()`会产生至少`TMP_MAX`个不同的名字。

### A.12.2 访问文件位置指示器

程序员使用本节中的函数随机地访问文件。与此类似的传统函数是`fseek()`、`ftell()`和`rewind()`。ANSI C增加了两个函数`fgetpos()`和`fsetpos()`。有些文件过大，以至于传统的函数不能访问，对于这样的情况一种实现方法可以设计这些函数来访问这样的文件。然而，很多较早版本的ANSI C编译器并没有这样做。

- `int fseek(FILE *fp, long offset, int place);`

本函数为下一个输入或输出操作设置文件位置指示器，这个位置是从`place`开始的`offset`字节。`place`的值可以是`SEEK_SET`、`SEEK_CUR`和`SEEK_END`，它们分别对应于文件的开始、文件的当前位置或文件的结尾。如果函数调用成功，就清除文件尾指示器并返回0。

- `long ftell(FILE *fp);`

该函数返回与`fp`相关的文件位置指示器的当前值。在二进制文件中，这个值是从文件开始的字节数；对于一些系统上的文本文件，这个值是“魔术甜饼”。在任何情况下，通过存储返回值都能用`fseek()`重新设置文件位置指示器。不成功的调用返回-1，并在`erron`中存储与系统相关的值。

- `void rewind(FILE *fp);`

本函数设置文件位置指示器在文件开始的位置，并清除文件尾指示器和错误指示器。函数调用`rewind()`等价于

```
(void) fseek(fp, 0L, SEEK_SET)
```

但`fseek()`仅清除文件尾指示器除外。

- `int fgetpos(FILE *fp, fpos_t *pos);`

该函数取得和`fp`相关的文件位置指示器的当前值，并把它存储在由`pos`指向的对象，随后`fsetpos()`用这个存储值重新设置文件位置指示器。成功的调用返回0，否则把一个与系统相关的值存储在`erron`中，并返回一个非零值。

### A.12.3 对错误的处理

- `void clearerr(FILE *fp);`

该函数清除与`fp`相关的文件的错误指示器和文件尾指示器。

- `int feof(FILE *fp);`

如果已经设置了与`fp`相关的文件尾指示器，该函数返回非零值。

- `int ferror(FILE *fp);`

如果已经设置了与`fp`相关的错误指示器，该函数返回非零值。

- `void perror(const char *s)`

该函数打印与`errno`相关的错误信息到`stderr`。首先打印串`s`，随后打印一个冒号和一个空格，然后打印相关的错误信息，后跟一个换行。(函数调用`strerror(errno)`仅打印相

关的错误信息。)

#### A.12.4 字符输入/输出

- `int getc(FILE *fp);`

该函数等价于 `fgetc()`，只是 `fgetc()` 作为宏。由于在宏定义中可能要对 `fp` 多次求值，这样带有副作用的参数的调用的工作可能不正确，例如 `fgetc(*p++)`。

- `int getchar(void);`

这个函数调用 `getchar()` 等价于 `getc(stdin)`。

- `char *gets(char *s);`

该函数从 `stdin` 读字符，并存储在由 `s` 指向的数组中，直到读入换行符或文件尾标记为止。此时写入的是空字符，而换行符被忽略。( `fgets()` 是要存储换行符的。) 如果写入了字符，就返回 `s`；否则 就返回 `NULL`。

- `int fgetc(FILE *fp);`

该函数从与 `fp` 相关的文件中取得下一个字符，并返回所读的字符值。如果遇到文件尾标记，就设置文件尾指示器，并返回 `EOF`。如果有错误出现，就设置错误指示器，并返回 `EOF`。

- `char *fgets(char *line, int n, FILE *fp);`

该函数从与 `fp` 相关的文件中至多读 `n - 1` 个字符，并写进由 `line` 指向的数组中。只要读到换行符或遇到文件尾，就不能从文件中读额外的字符。在这个过程的最后，把空字符写进数组。如果在开始就遇到文件尾，那么 `line` 的内容不受影响，返回 `NULL`；否则返回 `line`。

- `int fputc(int c, FILE *fp);`

该函数把参数 `c` 转换成 `unsigned char`，并写到与 `fp` 相关的文件中。如果调用 `fputc(c)` 成功，它返回

`(int) (unsigned char) c`

否则它设置错误指示器，并返回 `EOF`。

- `int fputs(const char *s, FILE *fp);`

该函数把以空字符结尾的串 `s` 复制到与 `fp` 相关的文件，但空字符除外。(函数 `puts()` 要把空字符写入。) 成功的调用会返回非负值，否则返回 `EOF`。

- `int putc(int c, FILE *fp);`

它等价于 `fputc()`，只是它作为宏实现。由于在宏定义中可能要对 `fp` 求值多次，这样带有副作用的参数的调用的工作可能不正确，例如 `putc(*p++)`。

- `int putchar(int c);`

调用 `putchar(c)` 等价于 `putc(c, stdout)`。

- `int puts(const char *s);`

该函数把以空字符结尾的串 `s` 复制到标准输出文件中，但不复制空字符，而是写入一个换行符。(与该函数相关的函数是 `fput()`，它不添加换行符。) 如果调用成功，函数返回一个非

负的值，否则返回EOF。

- `int ungetc(int c, FILE *fp);`

该函数把字符(`unsigned char`)`c`回放回到与`fp`相关的流中，`c`的值不能是EOF。至少要回放一个字符(大多数系统允许回放多个)。从流中读字符的次序与回放正好相反。字符一从流中读出，就会被丢弃；字符在文件中的放置不是永久的。当心：调用`fseek()`、`fsetpos()`或`rewind()`都会引起被回放字符的丢失。同样，在读取被回放的字符前，`ftell()`可能是不可靠的。

### A.12.5 格式化输入/输出

- `int fprintf(FILE *fp, const char *cntrl_string, ...);`

该函数把格式化文本写进由`fp`指示的文件中，并返回所写进的字节数。如果出现了错误，就设置错误指示器，并返回一个负值。转换说明或格式可以出现在`cntrl_string`中，转换说明或格式以`%`开始并以转换字符结束。格式决定了怎样显示其他参数(请参见13.1节“输出函数`printf()`”)。

- `int printf(const char *cntrl_string, ...);`

对函数`printf(cntrl_string, other_arguments)`的调用等价于调用`fprintf(stdout, cntrl_string, other_arguments)`。

- `int sprintf(char *s, const char *cntrl_string, ...);`

该函数是`printf()`的串版本，只是写到`s`指向的串代替了写到`stdout`。

- `int vfprintf(FILE *fp, const char *cntrl_string, va_list ap);`  
`int vprintf(const char *cntrl_string, va_list ap);`  
`int vsprintf(char s*, const char *cntrl_string, va_list ap);`

这些函数分别对应于`fprintf()`、`printf()`和`sprintf()`。它们没有使用长度可变的参数表，而是使用了在`stdarg.h`中定义的指向参数数组的指针。

- `int fscanf(FILE *fp, const char *cntrl_string, ...);`

该函数从与`fp`相关的文件流中读文本，并按控制流中的指示进行处理。有三种这样的指示：普通字符、空白字符和转换说明。对普通字符要匹配，用可选的空白字符匹配空白字符。转换说明的开始和结束分别是`%`和转换字符，函数用它计算从输入流中读出的字符的相关值，并存储到相应的参数中。如果在调用这个函数时输入流为空，就返回EOF；否则就返回已成功转换的字符数(请参见13.2.1节“返回值”)。

- `int scanf(const char *cntrl_string, ...);`

形式为`scanf(cntrl_string, other_arguments)`的函数调用等价于调用`fscanf(stdin, cntrl_string, other_arguments)`。

- `int sscanf(const char *s, const char *cntrl_string, ...);`

它是`scanf()`的串版本，只是用从`s`指向的串读代替了从`stdin`读。从串中读与从文件中读是不一样的，如果我们用`sscanf()`从`s`中再次读，那么输入开始于串首，而不是先前离开的位置。

### A.12.6 直接的输入/输出

函数 `fread()` 和 `fwrite()` 分别用于读和写二进制文件，并不做什么转换。在一些应用中，使用这些函数可以节省很多时间。

```
• size_t fread(void *a_ptr, size_t el_size, size_t n, FILE *fp);
```

该函数从由 `fp` 指示的文件中至多读 `n*el_size` 个字节(字符)，并放进由 `a_ptr` 指向的数组中。函数返回已成功写入的数组元素的个数。如果遇到文件尾标记，就设置文件尾指示器，返回短计数。如果 `el_size` 或 `n` 是 0，就不从输入流中读，返回 0。

```
• size_t fwrite(const void *a_ptr, size_t el_size, size_t n, FILE *fp);
```

该函数从由 `a_ptr` 指向的数组中读 `n*el_size` 个字节(字符)，并放进由 `fp` 指示的文件中。函数返回已成功写入的数组元素的个数。如果有错误发生，就返回短计数。如果 `el_size` 或 `n` 是 0，就不访问数组，返回 0。

### A.12.7 删除文件或对文件重命名

```
• int remove(const char *filename);
```

该函数从文件系统中删除名字为 `filename` 的文件。如果调用成功，函数就返回 0；否则就返回 -1。(在传统 C 中，这样的函数是 `unlink()`。)

```
• int rename(const char *from, const char *to);
```

该函数改变文件名。旧的文件名在由 `from` 指向的串中，新的文件名在由 `to` 指向的串中。如果为文件指定的新文件名已经存在，会发生什么是与系统相关的；在 UNIX 中通常会对文件重写。在大多数系统中，新旧文件名都是文件名或目录名。如果其中的一个参数是目录名，另一个也必须是目录名。如果调用成功，函数就返回 0；否则返回 -1，并把与系统相关的值写进 `errno`。

## A.13 一般的实用程序：<stdlib.h>

这个头文件中含有通用函数的原型以及相关的宏和类型定义。下面是这样的宏和类型定义的一些示例：

```
#include <stddef.h>                /* for size_t and wchar_t */

#define EXIT_SUCCESS 0             /* for use with exit() */
#define EXIT_FAILURE 1            /* for use with exit() */
#define NULL 0                    /* null pointer value */
#define RAND_MAX 32767            /* 2^15 - 1 */

typedef struct {
    int quot;                      /* quotient */
    int rem;                       /* remainder */
} div_t;

typedef struct {
    long quot;                     /* quotient */
    long rem;                      /* remainder */
} ldiv_t;
```

### A.13.1 动态内存分配

• `void *calloc(size_t n, size_t el_size);`

该函数在内存中为一个具有n个元素的数组分配连续的空间，每一个元素需要el\_size个字节。用0按位对这个空间进行初始化。如果调用成功，函数就返回所分配空间的基地址；否则就返回NULL。

• `void *malloc(size_t size);`

该函数在内存中分配由size个字节组成的空间块，对所分配的空间不进行初始化。如果调用成功，函数返回所分配空间的基地址；否则返回NULL。

• `void *realloc(void *ptr, size_t size);`

该函数把由ptr指向的块的大小变为size个字节，但对块中的内容不做改变。这个函数对新的空间不做初始化。它先尝试着使用原有的块的基地址，如果不能做到这一点，它就重新分配一块空间，并在复制原有块的内容后，释放原有块。如果ptr是NULL，其作用同调用malloc()一样。如果ptr不是NULL，ptr就一定是先前通过调用calloc()、malloc()或realloc()分配的但尚未通过调用free()或realloc()释放的空间的基地址。如果调用成功，函数就返回尺寸调整后的空间的基地址；否则就返回NULL。

• `void free(void *ptr)`

该函数释放由ptr指向的内存空间。如果ptr不是NULL，ptr就一定是先前通过调用calloc()、malloc()或realloc()分配的但尚未通过调用free()或realloc()释放的空间的基地址；否则调用就是错误的。发生的错误是与系统相关的。

### A.13.2 搜索和排序

• `void *bsearch(const void *key_ptr, const void *a_ptr,  
size_t n_els, size_t el_size,  
int compare(const void *, const void *));`

该函数在由a\_ptr指向的排序数组中搜索和由key\_ptr指向的对象相匹配的元素。如果找到，函数就返回该元素的地址；否则就返回NULL。数组中元素的个数是n\_els，每个元素用el\_size个字节存储。就比较函数compare()而言，数组中的元素必须以升序排列。compare()需要两个参数，每个参数都是数组元素的地址。依靠比较函数的第一个参数指向的元素是小于、等于或大于第二个参数指向的元素，比较函数返回小于、等于或大于0的整数（函数bsearch()使用折半查找算法）。

• `void qsort(void *a_ptr, size_t n_els, size_t el_size,  
int compare(const void *, const void *));`

该函数对由a\_ptr指向的数组以升序进行排序。数组中的元素个数是n\_els，每个元素用el\_size个字节存储。compare()需要两个参数，每个参数都是数组元素的地址。依靠比较函数的第一个参数指向的元素是小于、等于或大于第二个参数指向的元素，比较函数返回小于、等于或大于0的整数。（传统的qsort()使用快速排序。）

### A.13.3 伪随机数发生器

- `int rand(void);`

对该函数的每次调用都会生成一个整数，并以此作为返回值。对它重复调用，它产生在区间  $[0, \text{RAND\_MAX}]$  内的随机分布的整数序列。

- `void srand(unsigned seed);`

它为随机数发生器播种，这使得重复对 `rand()` 的调用产生的序列每次都从不同的位置开始。在程序开始时，随机数发生器就像调用了 `srand(1)` 一样，语句

```
srand(time(NULL));
```

每次调用程序时用不同的值为随机数发生器播种。

### A.13.4 与环境通信

- `char *getenv(const char *name);`

该函数搜索操作系统提供的环境变量表。如果 `name` 是表中的元素，函数就返回相应串值的基地址；否则就返回 `NULL`（请参见14.2节“环境变量”）。

- `int system(const char *s);`

该函数把串 `s` 作为命令传递给操作系统的命令解释器。如果 `s` 不是 `NULL`，该函数返回命令解释器所返回的值。如果 `s` 是 `NULL`，并且命令解释器可用，该函数返回非零值；否则返回0。

### A.13.5 整数运算

- `int abs(int i);`  
`long labs(long i);`

这两个函数返回 `i` 的绝对值。

- `div_t div(int numer, int denom);`  
`ldiv_t ldiv(long numer, long denom);`

这两个函数用 `denom` 除 `numer`，并返回一个用商和余数作为成员的结构。下面是一个例子：

```
div_t d;  
d = div(17, 5);  
printf("quotient = %d, remainder = %d\n", d.quot, d.rem);
```

执行这段代码会显示出

```
quotient = 3, remainder = 2
```

### A.13.6 串转换

两个函数族 `ato...()` 和 `strto...()` 的成员都能用于把串转换成值。转换是概念性的；这样的函数解释串中的字符，但不改变串本身。串可以以空白字符开头。串在遇到第一

个不正确的字符处会停止。例如，函数调用`strtod("123x456", NULL)`和`strod("\n 123 456", NULL)`都返回类型为`double`的值123.0。函数组`strto...`()对转换过程提供了更多的控制，并提供了错误检查机制。

- `double atof(const char *s); /* ascii to floating number */`

该函数把串`s`转换成`double`，并以此作为返回值。除了错误行为外，对`atof(s)`的调用等价于调用`strtod(s, NULL)`。如果没有发生转换，函数就返回0。

- `int atoi(const char *s); /* ascii to integer */`

该函数把串`s`转换成`int`，并以此作为返回值。除了错误行为外，调用`atoi(s)`等价于调用`(int) strtol(s, NULL, 10)`。如果没有发生转换，函数就返回0。

- `long atol(const char *s); /* ascii to long */`

该函数把串`s`转换成`long`，并以此作为返回值。除了错误行为外，调用`atol(s)`等价于调用`strtol(s, NULL, 10)`。如果没有发生转换，函数就返回0。

- `double strtod(const char *s, char **end_ptr);`

该函数把串`s`转换成`double`，并以此作为返回值。如果没有发生转换，函数就返回0。如果`end_ptr`不是`NULL`，并进行了转换，就把终止转换过程的字符的地址存储在由`end_ptr`指向的对象中。如果`end_ptr`不是`NULL`，没有进行转换，就把`s`的值存储在由`end_ptr`指向的对象中。在溢出的情况下，返回的是`HUGE_VAL`或`-HUGE_VAL`，并把`ERANGE`存储在`erron`中。在下溢的情况下，返回的是0，并把`ERANGE`存储在`erron`中。

- `long strtol(const char *s, char **end_ptr, int base);`

该函数把串`s`转换成`long`，并以此作为返回值。如果`base`（基数）的值是在2到36之间，那么`s`中的数字和字母按此基数解释。如果基数是36，就分别把从`a`到`z`和从`A`到`Z`的字母解释为从10到35。如果基数较小，就仅解释小于基数的那些数字和字母。如果`end_ptr`不是`NULL`，并进行了转换，就把终止转换过程的字符的地址存储在由`end_ptr`指向的对象中。下面是一个例子：

```
char *p;
long value;

value = strtol("12345", &p, 3);
printf("value = %ld, end string = \"%s\"\n", value, p);
```

执行这段代码会显示出：

```
value = 5, end string = "345"
```

由于基数是3，所以串`"12345"`中的3终止了转换过程，仅串中的前两个字符被转换。在基数为3的情况下，把字符12转换成十进制值5。以类似的方式，下面的代码

```
value = strtol("abcde", &p, 12);
printf("value = %ld, end string = \"%s\"\n", value, p);
```

显示出

```
value = 131, end string = "cde"
```

由于基数是12，所以在串`"abcde"`中的`c`终止转换过程，仅串中的前两个字符被转换。在基数为12的情况下，把字符`ab`转换成十进制值131。

如果base是0, 依靠s中前导非空白字符, 把s解释成十六进制、八进制或十进制整数。如果可选符是0x或0X, 就把串解释成十六进制整数(基数16); 如果可选符是0, 但不是0x或0X, 就把串解释成八进制整数(基数8); 否则把串解释成十进制整数。

如果没有发生转换, 函数返回0。如果end\_ptr不是NULL, 并且没有发生转换, 函数就把s的值存放在由end\_ptr指向的对象中。在溢出的情况下, 返回的是LONG\_VAL或-LONG\_VAL, 并把ERANGE存储在erron中。

- unsigned long strtoul(const char \*s,  
char \*\*end\_ptr, int base);

该函数类似于strtoul(), 只是它返回类型为unsigned long的值。在溢出的情况下, 返回的是ULONG\_VAL或-ULONG\_VAL。

### A.13.7 多字节字符函数

多字节字符用于表示扩展字符集的成员。怎样定义扩展字符集的成员是与场所相关的。

- int mblen(const char \*s, size\_t n);

如果s是NULL, 该函数返回零或非零值, 这依赖多字节字符是否有与状态相关的编码。如果s不是NULL, 该函数至多检查s中的n个字符, 并返回构成下一个多字节字符的字节数。如果s指向空字符, 函数就返回0。如果s不指向多字节字符, 函数就返回-1。

- int mbtowc(wchar\_t \*p, const char \*s, size\_t n);

该函数行为同mblen()一样, 只是增加了如下的功能: 如果p不是NULL, 函数把s中的下一个多字节字符转换成相应宽度的字符类型, 并存储在p指向的对象中。

- int wctomb(char \*s, wchar\_t wc);

如果s是NULL, 该函数返回零或非零值, 这依赖多字节字符是否有与状态相关的编码。如果s不是NULL, 并且wc是相应于多字节的宽位字符, 函数就在s中存储多字节字符, 并返回表示它所要求的字节数。如果s不是NULL, 并且wc不与多字节字符对应, 就返回-1。

### A.13.8 多字节串函数

- size\_t mbstowcs(wchar\_t \*wcs, const char \*mbs, size\_t n);

该函数读取由mbs指向的多字节串, 并把相应宽位字符写进wcs。至多写入n个宽位字符, 后跟一个宽位空字符。如果转换成功, 函数返回所写进的宽位字符的数目, 但最后的宽位空字符不算在内; 否则返回-1。

- int wcstombs(char \*mbs, const wchar\_t \*wcs, size\_t n);

该函数读取由wcs指向的宽位字符串, 并把相应的多字节串写进mbs。在写入n个宽位字符或一个空字符后, 转换过程停止。如果转换成功, 函数返回写入的字符数, 但空字符不算在内; 否则返回-1。

### A.13.9 退出程序

- void abort(void);



该函数非正常地终止程序，除非信号处理器捕捉到SIGABRT并且不返回。是否正确地关闭打开的文件，以及是否删除临时文件都依赖于实现方法。

- `int atexit(void (*func)(void));`

该函数注册在正常终止程序时由func指向的函数。如果调用成功，函数返回0；否则返回非零。至少能注册32个这样的函数。按与注册相反的次序执行被注册的函数。仅全局变量可用于在这些函数。

- `void exit(int status);`

该函数正常地终止程序。按与注册的相反次序调用由atexit()注册的函数，还要清除缓冲区中的流，关闭文件，删除由tempfile()创建的临时文件。status的值及其控制要被返回到主环境。如果status的值是0或EXIT\_SUCCESS，主环境就假设程序成功执行；如果status的值是EXIT\_FAILURE，主环境就假设程序没有成功执行；主环境还能识别status的其他值。

## A.14 内存和串的处理: <string.h>

这个头文件含有两个函数族中的函数的原型。函数mem...()用于操纵特定尺寸的内存块，我们可以把这些块看作是字节（字符）数组。它们像串，只是不用NULL结尾。函数str...()用于操纵以NULL结尾的串。通常下面行要出现在头文件的顶部：

```
#include <stddef.h>          /* for NULL and size_t */
```

### A.14.1 内存处理函数

- `void *memchr(const void *p, int c, size_t n);`

从内存地址p开始，该函数搜索与c相匹配的第一个无符号字符，但至多搜索n个字节。如果成功，函数返回指向所找到的字符的指针；否则返回NULL。

- `int memcmp(const void *p, const void *q, size_t n);`

该函数比较内存中尺寸为n的两个块，并把字节处理为无符号字符。依靠由p指向的块按字典顺序小于、等于或大于由q指向的块，这个函数返回小于、等于或大于0的值。

- `void *memcpy(void *to, void *from, size_t n);`

该函数把由from指向的块复制到由to指向的块，并返回to的值。如果块有重叠，该函数的行为是不定的。

- `void *memmove(void *to, void *from, size_t n);`

该函数把由from指向的块复制到由to指向的块，并返回to的值。如果块有重叠，在把新值写进由from指向的块中的字节之前，要对这些字节进行访问，即把重叠区中最初源字节在被覆盖之前进行复制。

- `void *memset(void *p, int c, size_t n);`

该函数用c的值（无符号字符）对由p指向的尺寸为n的块中的每个字节进行设置，并返回p的值。

## A.14.2 串处理函数

- `char *strcat(char *s1, const char *s2);`

该函数拼接串s1和s2，即它把s2复制到s1的尾部。程序员要确保s1指向的空间能容纳拼接后的结果。函数返回串s1。

- `char *strchr(const char *s, int c);`

该函数在s中搜索与c的值相匹配的第一个字符。如果匹配成功，就返回所找到的字符的地址；否则就返回NULL。调用`strchr(s, '\0')`要返回一个指向s中的结尾空字符的指针。

- `int strcmp(const char *s1, const char *s2);`

该函数按字典顺序比较串s1和s2。它把串元素处理成无符号字符。依靠按字典顺序s1小于、等于或大于s2，这个函数返回小于、等于或大于0的值。

- `int strcoll(const char *s1, const char *s2);`

该函数用依靠当前场所的比较规则，比较串s1和s2。依靠s1小于、等于或大于s2，这个函数返回小于、等于或大于0的值。

- `char *strcpy(char *s1, const char *s2);`

该函数把串s2复制到s1，其中包括结尾空字符，s1中已存在的内容被覆盖。程序员要确保s1指向的空间能容纳复制后的结果。函数返回s1的值。

- `size_t strcspn(const char *s1, const char *s2);`

该函数计算s1中不出现在s2中的最大初始子串的长度。例如，函数调用

```
strcspn("April is the cruellest month", "abc");
```

的返回值是13，这是因为“April is the”是第一个参数中与“abc”无共同之处的最大初始子串。（在`strcspn`中字符c是“complememnt”的缩写，spn是“span”的缩写。）

- `char *strerror(int error_number);`

该函数返回一个指向由系统提供的错误串的指针。程序不必改变这个串的内容。如果错误使得系统在`errno`中写入了值，程序员可以调用`strerror(errno)`显示相关的错误信息。（也可以用相关的函数`perror()`显示错误信息。）

- `size_t strlen(const char *s);`

该函数返回串s的长度，该长度是串s中的字节数，但结尾空字符并不计算在内。

- `char *strncat(char *s1, const char *s2, size_t n);`

该函数把s2中的至多n个字符（不包括终结空字符）添加到s1中，然后再向s1写入一个空字符。程序员要确保s1指向的空间能容纳添加后的结果。函数返回串s1。

- `int strncmp(const char *s1, const char *s2, size_t n);`

该函数按字典顺序比较串s1和s2中的至多n个字符，比较在第n个字符或结尾空字符处停止。该函数把串元素处理为无符号字符。依靠按字典顺序s1的比较部分小于、等于或大于s2的比较部分，这个函数返回小于、等于或大于0的值。

- `char *strncpy(char *s1, const char *s2, size_t n);`

该函数把n个字符写进s1，覆盖原有的内容。函数从s2中读取字符，直到复制了n个字符

或复制了空字符为止。该函数把s1中剩余的字符全变为'\0'。如果s2的长度为n或更大，s1就不是以空字符结束的。程序员要确保s1指向的空间能容纳结果。函数返回s1的值。

- `char *strpbrk(const char *s1, const char *s2);`

该函数在s1中搜索与s2中的任何字符匹配的第一个字符。如果搜索成功，就返回在s1中发现的字符的地址；否则返回NULL。例如，函数调用

```
strpbrk("April is the cruelest month", "abc");
```

返回cruelest中的c的地址。(函数名strpbrk中的pbrk是“pointer to break”的缩写。)

- `char *strchr(const char *s, int c);`

该函数从右开始搜索s中的与c的值(char)相匹配的第一个字符。如果找到这样的字符，就返回它的地址。函数调用strchr(s, '\0')返回指向s中的结尾空字符的指针。

- `size_t strspn(const char *s1, const char *s2);`

该函数计算s1中出现在s2中的最大初始子串的长度。例如，函数调用

```
strspn("April is the cruelest month", "A is for apple");
```

的返回值是9，这是因为第一个参数中t前的所有字符都出现在第二个参数中，t并没有出现在第二个参数中。(在strcspn中的spn是“span”的缩写。)

- `char *strstr(const char *s1, const char *s2);`

该函数在s1中搜索子串s2的第一次出现。如果搜索成功，函数就返回指向s1中这样的子串的基地址；否则就返回NULL。

- `char *strtok(char *s1, const char *s2);`

该函数用s2中的字符作为标记分隔符在s1中搜索。如果s1含有一个或多个标记，并发现了s1中的第一个标记，马上就用空字符重写标记后的字符，系统把s1中的剩余部分存储在别处，函数返回标记中的第一个字符的地址。使用s1等于NULL的连续调用会返回由系统提供的含有下一个标记的串的基地址。如果已经无标记可用，函数就返回NULL。如果s1中没有标记，对strtok(s1, s2)的初始调用返回NULL。下面是一些例子：

```
char s1[] = " this is,an example ; ";
char s2[] = ",; ";
char *p;

printf("\n%s\\", strtok(s1, s2));
while ((p = strtok(NULL, s2)) != NULL)
    printf(" \\n%s\\", p);
putchar('\\n');
```

执行这段程序会显示

```
"this" "is" "an" "example"
```

- `size_t strxfrm(char *s1, const char *s2, size_t n);`

该函数转换串s2，并把结果放在s1中。至多把包括终结空字符在内的n个字符写进s1中。函数返回s1的长度。转换是这样的：在把两个被转换串用作strcmp()的参数时，依靠strcoll()应用到未被转换的串返回的值小于、等于或大于0，strcmp()返回小于、等于或大于0的值。(strxfrm中的xfrm代表“transform”。)

## A.15 日期和时间: <time.h>

这个头文件含有处理日期、时间和内部时钟的函数的原型。下面是一些宏和类型定义的例子:

```
#include <stddef.h>          /* for NULL and size_t */

#define CLOCKS_PER_SEC 1000000 /* machine-dependent */

typedef long clock_t;
typedef long time_t;
```

类型为struct tm的对象用于存储日期和时间。

```
struct tm {
    int    tm_sec;    /* seconds after the minute: [0, 60] */
    int    tm_min;    /* minutes after the hour: [0, 59] */
    int    tm_hour;    /* hours since midnight: [0, 23] */
    int    tm_mday;    /* day of the month: [1, 31] */
    int    tm_mon;    /* months since January: [0, 11] */
    int    tm_year;    /* years since 1900 */
    int    tm_wday;    /* days since Sunday: [0, 6] */
    int    tm_yday;    /* days since 1 January: [0, 365] */
    int    tm_isdst;    /* Daylight Savings Time flag */
};
```

注意, tm\_sec的值的范围必须适应“闰秒”, 虽然这只是偶尔能出现的。如果在实行夏令时, 标志tm\_isdst就是正的; 否则是0。如果信息不可用, 标志tm\_isdst就是负的。

### A.15.1 访问时钟

在大多数系统中, clock()函数提供了对机器时钟的访问机制。时钟的速率是与机器相关的。

- clock\_t clock(void)

该函数返回程序启动后经过的CPU“时钟滴答”的大概数。要把它转换成秒, 返回值能被CLOCKS\_PER\_SEC相除。如果CPU时钟不可用, 函数返回-1(请参见第14.6节“关于时间的编码”)。

### A.15.2 访问时间

在ANSI C中有两个时间版本, 即日历时间和分解时间。日历时间被表示为整数, 在大多数系统中它表示自从1970年1月1日以来的秒数, 分解时间被表示为类型为struct tm的结构。按照国际时间坐标(UTC), 用日历时间进行编码。程序员可以用库函数转换这两种时间版本, 也可以用函数把时间作为串显示。

- time\_t time(time\_t \*tp);

该函数返回当前的日历时间, 它表示自从1970年1月1日以来的秒数。也可以使用其他的时间单位和开始时间, 但这样的表示是常用的。如果tp不是NULL, 也把该值存储在由tp指向的对象中。请考虑下面的代码:

```
time_t    now;

now = time(NULL);
printf("\n%s%Td\n", now, now, now);
```

```

    "          ctime(&now) = ", ctime(&now).
    "asctime(localtime(&now)) = ", asctime(localtime(&now)));

```

在我们的系统上这些这段代码会显示:

```

    now = 685136007
    ctime(&now) = Tue Sep 17 12:33:27 1991
    asctime(localtime(&now)) = Tue Sep 17 12:33:27 1991

```

- `char *asctime(const struct tm *tp);`

该函数把由`tp`指向的分解时间转换成系统提供的串。这个函数返回该串的基地址。以后调用`asctime()`和`ctime()`会重写这个串。

- `char *ctime(const time_t *t_ptr);`

该函数把由`t_ptr`指向的日历时间转换成系统提供的串。这个函数返回该串的基地址。以后调用`asctime()`和`ctime()`会重写这个串。函数调用`ctime(&now)`与`asctime(localtime(&now))`是等价的。

- `double difftime(time_t t0, time_t t1);`

该函数计算`t1 - t0`的差, 如果需要, 就把结果值转换成在日历时间`t1`与`t0`间已经经过的秒数。函数把这个值作为`double`型的返回。

- `struct tm *gmtime(const time_t *t_ptr);`

该函数把由`t_ptr`指向的日历时间转换成分解时间, 并把它存储在系统提供的类型为`struct tm`的对象中。函数返回该结构的地址。该函数按照国际时间坐标计算分解时间。通常把它称为格林威治时间; 该函数的名字也正是这个含义。以后调用`gmtime()`和`localtime()`会重写这个结构。

- `struct tm *localtime(const time_t *t_ptr);`

该函数把由`t_ptr`指向的日历时间转换成当地的分解时间, 并把它存储在系统提供的类型为`struct tm`的对象中。函数返回该结构的地址。以后调用`gmtime()`和`localtime()`会重写这个结构。

- `time_t mktime(struct tm *tp);`

该函数把由`tp`指向的结构中的当地的分解时间转换成相应的日历时间。如果转换成功, 函数就返回日历时间; 否则就返回`-1`。由于计算的缘故, 函数忽略了结构成员`tm_wday`和`tm_tday`。在计算前, 其他成员的值可能超出了它们的通常范围; 在计算后, 可能要用各成员在标准范围内的等价值集对其进行重写。用其他成员的值计算`tm_wday`和`tm_tday`的值。例如, 可以用如下代码找距离现在的第1000天:

```

struct tm  *tp;
time_t     now, later;

now = time(NULL);
tp = localtime(&now);
tp->tm_mday += 1000;
later = mktime(tp);
printf("\n1000 days from now: %s\n", ctime(&later));

```

- `size_t strftime(char *s, size_t n, const char *cntrl_str, const struct tm *tp);`

在由`cntrl_str`指向的控制串的指导下, 该函数把字符写入由`s`指向的串中, 至多写入`n`

个字符，其中包括空字符。如果需要n个以上的字符，函数就返回0，而s的内容是不确定的；否则就返回s的长度。控制串由普通字符、转换说明或格式组成，它决定了如何写由tp指向的结构中的分解时间的值。每个转换说明由一个%及后跟的一个转换字符组成。

strftime() 的用法		
转换说明	显示的内容	例子
%a	星期名的缩写	Fri
%A	星期名的全称	Friday
%b	月名的缩写	Sep
%B	月名的全称	September
%c	日期和时间	Sep 01 02:17:23 1993
%d	月的天	01
%H	昼夜的小时	02
%h	以12小时计的 天中的小时	02
%j	年中的天	243
%m	年中的月	9
%M	小时中的分	17
%p	AM或PM	AM
%s	小时中的秒	23
%U	年中的星期	34
%w	星期中的天 (0-6)	5
%x	日期	Sep 01 1993
%X	时间	02:17:23
%y	世纪中的年	93
%Y	年	1993
%Z	时区	PDT
%%	百分比符号	%

考虑下面的代码：

```
char    s[100];
time_t  now;

now = time(NULL);
strftime(s, 100, "%H:%M:%S on %A, %d %B %Y",
        localtime(&now));
printf("%s\n\n", s);
```

执行含有这些代码的程序会显示出：

```
13:01:15 on Tuesday, 17 September 1991
```

## A.16 其他

除了ANSI C说明的函数外，系统还可能在库中提供其他函数。在本节中我们要描述一些已经在广泛使用的非ANSI C函数。像exec1()这样的一些函数在大多数系统上都是常见的，像fork()和spawnl()这样的函数仅在某些操作系统中可用。相关的头文件名是与系统相关的。

### A.16.1 使用文件描述符

- `int open(const char *filename, int flag, ...);`

该函数打开指定名的文件，按flag中存储的逐位信息规定进行读写。如果文件正被创建，就还需要一个类型为unsigned的参数，用以设置新文件的权限。如果调用成功，函数返回一个称为文件描述符(file descriptor)的非负整数；否则设置errno，并返回-1。在含有open()原型的头文件中给出了用于flag的值。这些值是与系统相关的。

- `int close(int fd);`

该函数关闭由文件描述符fd指示的文件。如果调用成功，函数返回0；否则设置errno，并返回-1。

- `int read(int fd, char *buf, int n);`

该函数从由文件描述符fd指示的文件中读取至多n个字符，并放到由buf指向的对象中。如果调用成功，函数就返回放到buf中的字符数；否则就返回-1。如果遇到文件尾字符，函数就返回短计数。

- `int write(int fd, const char *buf, int n);`

该函数从由buf指向的对象中读取至多n个字符，并放到由文件描述符fd指示的文件中。如果调用成功，函数就返回放到文件中的字符数；否则就返回-1。短计数指明磁盘已满。

### A.16.2 创建并发进程

- `int fork(void);`

该函数复制当前进程，并开始并行地执行它。子进程有自己的进程标识号。在调用fork()时，它把0返回给子进程，把子进程的ID给父进程。如果调用失败，就设置errno，并返回-1。在MS-DOS中这个函数不可用。

- `int vfork(void);`

该函数在虚拟空间中产生一个新进程，这是一种有效的方式。子进程有自己的进程标识号。该函数不完全复制父进程的地址空间，在按页存储的环境中这不怎么有效。子进程借用父进程的内存和控制线程，直到调用exec...()发生或子进程退出。在子进程使用父进程的资源时，父进程被挂起。在调用vfork()时，它向子进程返回0，把子进程的ID给父进程。如果调用失败，就设置errno，并返回-1。在MS-DOS中这个函数不可用。

### A.16.3 覆盖进程

在本节中，我们描述函数族exec...()和spawn...()。前一个一般用于MS-DOS和UNIX系统，后一个仅用在MS-DOS系统。在UNIX系统中，可以使用exec...()的fork()，以达到spawn...()的效果。

- `int execl(char *name, char *arg0, ..., char *argN);`  
`int execl(char *name, char *arg0, ..., char *argN,`  
`char **envp);`  
`int execlp(char *name, char *arg0, ..., char *argN);`  
`int execlpe(char *name, char *arg0, ..., char *argN,`  
`char **envp);`  
`int execv(char *name, char **argv);`

```
int execve(char *name, char **argv, char **envp);
int execvp(char *name, char **argv);
int execvpe(char *name, char **argv, char **envp);
```

这些函数用指定的程序覆盖当前的进程。对父进程没有什么返回值。缺省情况下，子进程继承父进程的环境。用`execl`开始的名称的函数族的成员需要一个参数表，用做子进程的命令行参数，参数表中的最后一个参数必须是`NULL`指针。用`execv`开始的名称的函数族的成员使用数组`argv`向子进程提供命令行参数，`argv`的最后一个参数必须是`NULL`指针。用`e`结尾的名称的函数族的成员使用数组`envp`向子进程提供环境变量，`envp`的最后一个参数必须是`NULL`指针。在名字中含有`p`的函数族的成员使用在环境中描述的路径变量，来决定在哪个目录中搜索程序。

```
• int spawnl(int mode, char *name, char *arg0,..., char *argN);
  ....
```

这个函数与`exec...()`函数族相对应，只是它的每一个成员有一个初始的整数参数。`mode`的值是0、1和2。`mode`为0使得父进程要在子进程完成后才继续执行；`mode`为1使得父进程和子进程应该并发执行，只是至今还没有实现，使用这个值会引发错误；`mode`为2使得子进程覆盖父进程。

#### A.16.4 进程间通讯

```
• int pipe(int pd[2]);
```

该函数创建一个称为管道(`pipe`)的输入/输出机制，它把相关的文件描述符（管道描述符）放到数组`pd`中。如果调用成功，函数返回0；否则设置`errno`，并返回-1。在创建管道后，系统假设通过连续调用`fork()`创建的两个或更多的协作进程使用`read()`和`write()`通过管道传递数据。描述符`pd[0]`用于读出，描述符`pd[1]`用于写入。管道的容积是与系统相关的，但至多为4096个字节。如果写满了管道，它就会堵塞，直到有数据读出为止。像其他的文件描述符一样，可以用`close()`显式地关闭`pd[0]`和`pd[1]`。在MS-DOS中这个函数是不可用的。

#### A.16.5 挂起程序的执行

```
• void sleep(unsigned seconds);
```

该函数挂起当前的执行进程，持续到所需要的秒数为止。这里指定的时间只是一个大约值。



## 附录B 预处理器

C语言用预处理器扩展它的能力和表示法。在本附录中，我们要对预处理器做详细地讨论，其中包括ANSI C委员会增加的新特征。我们先说明#include的用法，然后全面地讨论#define宏机制的用法。宏可以用于产生代替函数调用的内联代码。使用宏可以减少程序的执行时间。

用#开始的行被称为预处理指令(preprocessing director)，这样的行要和预处理器通信。在ANSI C中#前可以加空白符，但在传统C中#必须在第一列。预处理指令的语法独立于C语言的其余部分。预处理指令的作用起始于它在文件中的位置，持续到文件尾或被另一个指令否定为止。始终要记住，预处理器不“懂得”C。

### B.1 #include的用法

预处理指令

```
#include "filename"
```

使得预处理器用指定文件的内容替代该预处理指令行。在该方式下，编译器首先在当前目录下搜索指定的文件，如没找到，再在与系统相关的目录下进行寻找。使用

```
#include <filename>
```

预处理器不在当前目录下寻找，而仅在与系统相关的目录下进行寻找。在UNIX系统中，通常可以在/usr/include下找到像stdio.h和stdlib.h这样的标准头文件。一般而言，标准头文件存储在哪里是与系统相关的。

对于#include文件能包含的内容并没有做限制。特别是，它还能包含预处理器能扩展的其他预处理指令。当心：程序初学者有时会把标准头文件和标准库相混淆，虽然引入文件为标准库提供了接口，但标准头文件不是标准库。

### B.2 #define的用法

使用带#define的预处理指令有两种形式：

```
#define identifier token_stringopt
#define identifier(identifier, ..., identifier) token_stringopt
```

token\_string是可选的。如果定义较长，可以在当前行尾放一个\来把本行延续到下一行。如果在文件中使用了第一种形式，预处理器用token\_string替代文件中的每一个identifier，只有引用串除外。考虑下面的例子：

```
#define SECONDS_PER_DAY (60 * 60 * 24)
```

在本例中，标记串是(60\*60\*24)，预处理器用它替换文件中的每一个SECONDS\_PER\_DAY。

对#define的简单应用能改善程序的清晰度和可移植性。例如，如果要在程序中使用像 $\pi$ 或光速c这样的特定常量，程序员就应该定义它们。

```
#define PI 3.14159
#define C 299792.458 /* speed of light in km/sec */
```

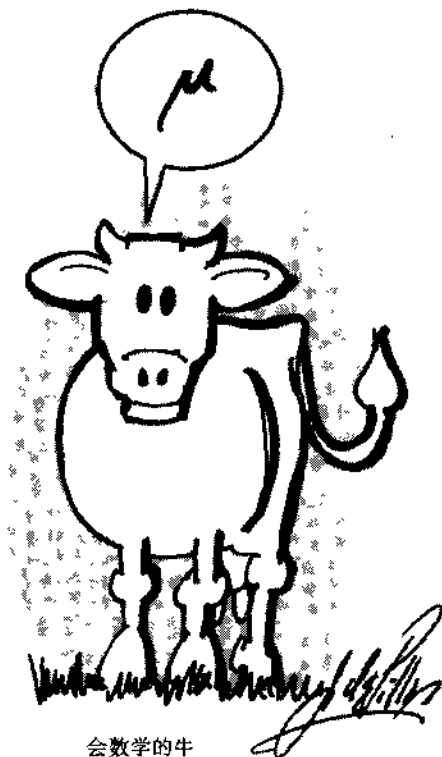
把在程序中使用的其他常量也最好定义成符号常量。

```
#define EOF (-1) /* typical end-of-file value */
#define MAXINT 2147483647 /* largest 4-byte integer */
```

也可以把程序员决定的对程序的限制指定为符号常量。

```
#define ITERS 50 /* number of iterations */
#define SIZE 250 /* array size */
#define EPS 1.0e-9 /* a numerical limit */
```

一般用有助于记忆的标示符替换以别的方式表示的可能是神秘的常量有助于文档化。与系统相关的常量随系统不同可能会变化, 把其定义为符号常量有助于移植。把易变的常量限制到一个位置, 也易于检查常量的实际值, 这样有助于提高程序的可靠性。



## 语法修饰

虽然并没有把使用预处理器看作是一种良好的编程习惯, 但是程序员使用预处理器可以改变C的语法。例如, 由于把==错用为=是一种常见的编程错误, 程序员应该用

```
#define EQ ==
```

来防范这样的错误。人们把这种对编程语法的表面上的改变称为语法修饰(syntactic sugar)。另一个这样的例子是通过引入do改变while语句的语法形式, 这是种ALGOL风格的结构。

```
#define do /*blank*/
```

在文件的顶部放这两个#define行，代码

```
while (i EQ 1) do {
    .....
```

在通过预处理器后，就变为

```
while (i == 1) {
    .....
```

记住，由于do不会在文件中出现，do-while语句就不能再用了。

### B.3 带参数的宏

在现代编程技术中，宏本质上被认为是不安全的，这是因为预处理器执行宏，而预处理器并不“懂得”C，带有参数的宏是特别危险的。虽然如此，由于宏节省了函数调用的开销，带有参数的宏在系统级上已经被广泛地使用。（在C++中，可以用内联函数代替带有参数的宏。）带有参数的宏的一般形式为：

```
#define identifier( identifier, ..., identifier) token_stringopt
```

在第一个identifier和左括号之间没有空格。在参数表中可以出现多个标识符，也可以不出现。例如，

```
#define SQ(x) ((x) * (x))
```

在#define中的标识符x是下文中要被替换的参数。在替换时，用的是不经过语法错误检查的串。例如，用参数7+w调用宏，SQ(7 + w)被扩充为((7 + w) \* (7 + w))，以类似的方式，SQ(SQ(\*p))被扩充为((( (\*p) \* (\*p))) \* (((\*p) \* (\*p))))。

这种表面上看对括号的过多使用是要保护对宏的求值次序。我们要解释为什么需要这些括号。首先假设我们已经定义了宏

```
#define SQ(x) x * x
```

用这个定义，SQ(a + b)被扩展为a + b \* a + b，由于运算符的优先级，扩展后的表达式与(a + b) \* (a + b)是不一样的。现在假设我们已经定义了宏

```
#define SQ(x) (x) * (x)
```

用这个定义，4 / SQ(2)被扩展为4 / (2) \* (2)，由于运算符的优先级，扩展后的表达式与4 / ((2) \* (2))是不一样的。

最后假设我们已经定义了宏

```
#define SQ (x) ((x) * (x))
```

用这个定义，SQ(7)被扩展为(x) ((x) \* (x)) (7)，这与我们的意图差得更远。在宏定义中，如果在宏的名字和后面的左括号间有一个空格，那么这行的其余部分就都被作为替换文本。

如果按一种健壮的方式书写宏，就像下例一样：

```
#define SQ(x) ((x)*(x))
```

就会出现错误。例如，如果我们写SQ(++k)，该宏调用将被扩展成(++k) \* (++k)，这肯定不是我们的意图。如果SQ(++k)是函数调用，而不是宏调用，将不会有问题。表达式++k将被求值，然后将该值传递给函数。

另一个常见的编程错误是用分号作#define的结尾，这使得分号也作为替换串的一部分，而事实上不想这样。下面是一个这样的例子：

```
#define SQ(x) ((x)*(x));
```

这里的分号是输入错误。由于程序员经常用分号作为代码行的结尾，这种错误是经常发生的。在函数体中使用这样的宏时，`x = SQ(y);`被扩展成 `x = ((y) * (y));`；最后的分号产生了一个多余的空语句。如果我们写

```
if (x == 2)
    x = SQ(y);
else
    ++x;
```

我们将会得到一个由多余的空语句引起的语法错误，这个多余的分号使得else语句不附属于该if语句。

虽然在很多情况下带参数的宏的行为像函数，但它们却是不相同的，这是因为预处理器把它们扩展成内联代码。由于这个原因，程序员可能会犯错误。考虑下面的宏：

```
#define PRN(array, size) printf("\n");
                          for (i = 0; i < size; ++i) \
                          printf("%5d", array[i])
```

假设我们要用这个宏显示一个数组5次。我们可以写代码如下：

```
for (j = 0; j < 5; ++j)
    PRN(a, n);
```

然而，这不会产生预期的结果。预处理器把这两行代码扩展成等价的如下代码：

```
for (j = 0; j < 5; ++j)
    printf("\n");
for (i = 0; i < n; ++i)
    printf("%5d", a[i]);
```

我们得到了这样的代码行和一个显示输出行。如果像这样的错误发生，可能是很不可思议的。要防止这样的错误发生，程序员应该在宏定义中使用括号。

调试含有带有参数的宏的代码是很困难的。大多数C编译器提供 `-E` 选项，以表明预处理器的输出。

```
cc -E pgm.c > tmp
```

由于输出趋向于越来越庞大，通常程序员要把输出重定向到一个临时文件，然后用文本编辑器看临时文件的内容。

宏经常用于替换具有内联代码的函数调用，这是更有效的。例如，程序员可以编写一个宏，代替编写一个找两个值中最小者的函数。这样的宏如下：

```
#define min(x, y) (((x) < (y)) ? (x) : (y))
```

在这样定义后，编译器把

```
m = min(u, v)
```

这样的表达式扩展成

```
m = (((u) < (v)) ? (u) : (v))
```

`min()` 的参数可以是任意的表达式，只要在类型上兼容就行。我们也可以用 `min()` 构造另一个宏。例如，如果我们要找4个值中的最小者，我们可以写

```
#define min4(a, b, c, d) min(min (a < b) , min (c < d))
```

在宏定义的体内既可以使用函数，也可以使用宏。下面是一些例子：

```
#define SQ(x)      ((x) * (x))
#define CUBE(x)    (SQ(x) * (x))
#define F_POW(x)   sqrt(sqrt(CUBE(x)))
/* fractional power: 3/4 */
```

## B.4 #undef的用法

形式为

```
#undef identifier
```

的预处理器指令取消宏。它使得前面的宏定义被取消。例如，下述的代码

```
#undef PI
#define PI "I like apple."
```

如果PI已经被定义，#undef预处理器指令在文件的其余部分取消PI的定义，或直到PI被再次定义为止。如果PI在先前没有做过定义，那么这个预处理器指令是不起作用的。

程序员有时要用出乎意料的#undef机制，下面是一个例子。在很多Sun机器上的C系统中，下面的程序会失败：

```
#include <stdio.h>

int main(void)
{
    int  earth = 1, moon = 2, sun = 3;

    printf("sum = %d\n", earth + moon + sun);
    return 0;
}
```

这是由于系统把标识符sun处理为#define符号常量，因而导致了这样情形的发生。如果在你的编译器上通不过这样的代码，你可以看到所产生的错误信息完全是无用的。你可以用编译器的-E选项去看编译器的输出。为了确定问题，你应该把

```
#undef sun
```

放在文件的顶部。当心：在其他机器上会发生这种问题。（在DEC机器上用dec作为标识符会出问题吗？在Cray机器上使用cray，又怎么样？）这样的问题是与编译器和机器相关的。

## B.5 条件编译

预处理器有条件编译指令，这样的指令用于开发程序和编写更容易移植的代码。各种预处理指令的形式如下：

```
#if      constant_integral_expression
#ifdef   identifier
#ifndef   identifier
```

这些预处理指令规定对在预处理指令#endif之前的代码进行条件编译。要编译在#if之后的插入代码，常量表达式必须是非零(true)，在#ifdef或#if defined之后，#define行中指定的标识符必须在先前定义。对于无插入的

`#undef identifier`

已经用于取消该宏。在`#ifndef`后，指定的标识符必须取消。

用在预处理指令中的整型常量表达式不能含有`sizeof`运算符或类型转换，但可以使用`defined`预处理运算符。在ANSI C中这个运算符是可用的，但在传统的C中则不一定。表达式`defined identifier` 等价于 `defined(identifier)` 如果标识符当前已定义，它的求值是1，否则是0。下面是怎样使用它的一个例子：

```
#if defined(HP9000) || defined(SUN4) && !defined(VAX)
..... /* machine-dependent code */
#endif
```

有时`printf()` 语句可用于调试程序。假设在文件的顶部我们写了

```
#define DEBUG 1
```

在文件的其余部分我们写了很多像：

```
#if DEBUG
    printf("debug: a = %d\n", a);
#endif
```

这样的行，因为符号常量`DEBUG`的值不是0，所以`printf()` 语句被编译。随后，通过把符号常量`DEBUG`的值变为0，编译就忽略这些行。

另一个可选的方案是定义无值的符号常量。假设在文件的顶部我们写了

```
#define DEBUG
```

那么我们可以用条件编译的`#ifdef`或`#if defined`形式。例如，如果我们写了

```
#ifdef DEBUG
.....
#endif
```

就会编译插入的代码行；如果我们删除在文件顶部定义`DEBUG`的`#define`行，就不会编译插入的代码行。

假设我们正在为一个大型的软件项目编码，我们可能想在所有的代码前引入一些别处提供的头文件。我们的代码可能要使用这些头文件中的一些函数原型和宏，但由于头文件是用于整个项目的，我们的代码可能只用了其中的一部分，此外，我们可能不知道在最后的头文件中的所有事情。为了防止宏名的冲突，我们可以使用`#undef`机制。

```
#include "everything.h"

#undef PIE
#define PIE "I like apple."
.....
```

如果恰巧在`everything.h`中定义了`PIE`，那么我们就能取消它。如果没有在`everything.h`中定义`PIE`，那么`#undef`就不起作用。

下面是条件编译的一般用法。想象一下，你正处于程序的开发阶段，你的代码形式为：

```
statements
more statements
and still more statements
```

由于调试或测试的缘故，你可能希望临时忽略或封闭一些代码。要做到这一点，你可以把代码放在注释中，

```
statements
/*
more statements
*/
and still more statements
```

然而，如果封闭的代码中也含有注释，这个方法就会导致语法错误；使用条件编译能解决这个问题。

```
statements
#if 0
more statements
#endif
and still more statements
```

预处理器拥有类似于C中的if-else语句这样的控制结构。每种#if形式都可以跟任意行，可能还含有如下形式的预处理指令：

```
#elif constant_integral_expression
```

它后跟有预处理指令

```
#else
```

它还跟有预处理指令

```
#endif
```

注意，#elif是“else-if”的缩写。条件编译的控制流与if-else语句提供的控制流类似。

## B.6 预定义宏

在ANSI C中，有5个预定义宏，它们总是可用的，程序员不能取消它们。每个这样的宏的名字都用两个下划线字符开头和结尾。

预定义宏	值
__DATE__	含有当前日期的串
__FILE__	含有文件名的串
__LINE__	表示当前行号的整数
__STDC__	如果实现遵循ANSI 标准C，那么值就是非零整数
__TIME__	含有当前时间的串

## B.7 运算符#和##

在ANSI C中预处理运算符#和##是可用的，但在传统C中是不可用的。二元运算符#使得在宏定义中的形式参数串化。下面是关于这种用法的一个例子：

```
#define message_for(a, b) \
    printf("#a " and " #b ": We love you!\n")

int main(void)
{
    message_for(Carole, Debra);
    return 0;
}
```

在调用宏时，宏定义中的每一个参数都被相应的参数替换，带有#使得参数被用双引号括上。这样，在通过预处理器后，我们获得的是

```
int main(void)
{
    printf("Carole" " and " "Debra" ": We love you!\n");
    return 0;
}
```

由于用空白字符分隔的串常量会被拼接，这个printf()语句等价于

```
printf("Carole and Debra: We love you!\n");
```

B.8节要讨论在断言中怎样使用串化运算符#。

二元运算符##用于合并标记。下面是一个例子，说明怎样使用这个运算符：

```
#define X(i) x ## i

X(1) = X(2) = X(3);
```

在通过预处理器后，我们得到的是

```
x1 = x2 = x3;
```

## B.8 assert() 宏

ANSI C在标准头文件assert.h中提供了宏assert()，你可以用这个宏保证一个表达式的值符合你的要求。假设我们正在编写一个重要的函数，你要保证参数满足一定的条件。下面是一个例子，说明怎样用assert()实现这一点：

```
#include <assert.h>

void f(char *p, int n)
{
    ....
    assert(p != NULL);
    assert(n > 0 && n < 5);
    ....
}
```

如果断言失败，系统就显示出消息，程序终止。虽然assert()宏在不同系统上的执行是不同的，但在总体上的它的行为是相同的。下面是编写宏的一种方式：

```
#if defined(NDEBUG)
#define assert(ignore) ((void) 0) /* ignore it */
#else
#define assert(expr) \
    if (!(expr)) { \
        printf("\n%s%s\n%s%s\n%s%d\n\n", \
            "Assertion failed: ", #expr, \
            "in file ", __FILE__, \
            "at line ", __LINE__); \
        abort(); \
    } \
#endif
```

注意，如果已经定义了宏NDEBUG，就忽略所有的断言。程序员在开发程序期间可自由地使用宏，以后通过定义宏NDEBUG可有效地放弃所使用的宏。函数abort()是在标准库中（请参见附录A“标准库”）。



## B.9 #error和#pragma的用法

ANSI C中增加了#error和#pragma预处理指令。下面的代码说明了怎样使用#error:

```
#if A_SIZE < B_SIZE
    #error "Incompatible sizes"
#endif
```

在编译期间,如果预处理器遇到了#error指令,就会发生编译时错误,并在屏幕上显示跟随在#error后的串。在这个例子中,我们用#error宏保持两个符号常量的一致性。以类似的方式,可以用这个指令强化其他的条件。

pragma指令用于实现特定的用途。它的一般形式为:

```
#pragma tokens
```

它导致的行为与具体的C编译器相关。编译器不识别的任何#pragma都被忽略。

## B.10 行号

形式为

```
#line integral_constant "filename"
```

的预处理指令使得编译器对源文本重编号,为下一行设定一个特定的常量,并认为当前的源文件名是filename。如果没有出现文件名,那么只发生重编号。通常程序员是看不见行号的,仅在给出警告或语法错误时行号才出现。

## B.11 相应的函数

在ANSI C中,假设标准头文件中给出的很多带有参数的宏在标准库中都有相应的函数。作为一个例子,假设我们要访问函数isalpha()而不访问宏,实现这一点的一种方法是在调用isalpha()前在文件中的某处给出

```
#undef isalpha
```

这样就放弃了宏isalpha的定义,使得编译器使用函数。我们在文件的顶部仍然要引用头文件ctype.h,这是因为这个文件除了包含宏外,还含有函数原型。

不用宏而用函数的另一个方法是写

```
(isalpha)(c)
```

预处理器不把这个结构看作是宏,编译器把它看作是函数调用。

# 附录C 位运算符

在本附录中，我们要讨论位运算符，使用位运算符程序员可以访问字节中的各个位。这种底层数据结构经常是不可移植的。对系统编程来说这是重要的，对要优化存储区利用率和效率的编程来说这也是很重要的。

## C.1 位运算符和表达式

位运算符仅对被表示为二进制数串的整型表达式起作用。显然这些运算符是与系统相关的。我们仅就一个字节为8位、一个机器字为4个字节的机器，以及整数的二进制补码表示和ASCII字符码进行讨论。

位运算符		
逻辑运算符	(一元) 求反 ~	~
	按位与	&
	按位异或	^
	按位或	
移位运算符	左移 <<	<<
	右移	>>

运算符~是一元的；所有其他的位运算符是二元的。这些运算符都用于整型表达式。像其他的运算符一样，位运算符也有精确地决定怎样计算包含它们的表达式的优先级和结合性规则（请参见附录F“运算符的优先级和结合性”）。

### C.1.1 按位求反

运算符~被称为求反运算符(complement operator)或按位求反运算符(bitwise complement operator)，它把其参数中的位串表示求反，即0变1、1变0。例如，一个声明为

```
int a = 70707;
```

a的二进制表示是

```
00000000 00000001 00010100 00110011
```

~a表达式是对a按位求反，求反后的二进制表示是

```
11111111 11111110 11101011 11001100
```

~a的值为int值 - 70708。

### C.1.2 二进制补码

非负整数n的二进制补码表示(complement representation)是以2为基写n而得到的位串。如

果我们对位串按位求反后再加1,我们就得到 $-n$ 的二进制补码表示。下表给出了一些例子,为了节省空间,我们仅显示了两个低位字节。

$n$ 的值	二进制表示	按位取反	$-n$ 的补码表示	$-n$ 的值
7	00000000 00000111	11111111 11111000	11111111 11111001	-7
8	00000000 00001000	11111111 11110111	11111111 11111000	-8
9	00000000 00001001	11111111 11110110	11111111 11110111	-9
-7	11111111 11111001	00000000 00000110	00000000 00000111	7

对上表从左到右读。如果我们先把正整数 $n$ 表示成二进制,然后对其求反加1,我们就得到 $-n$ 的二进制补码表示。我们把使用二进制补码表示作为其内存中的整数值的二进制表示的机器称为二进制补码机器(two's complement machine)。

在二进制补码机器中,如果我们先把负数 $-n$ 表示成二进制,然后对其求反加1,我们就得到 $n$ 的二进制补码表示即二进制表示,上表的最后一行说明了这一点。

对0和 $-1$ 的二进制补码表示是特殊的。0的各位都是0, $-1$ 的各位都是1。注意,如果把一个二进制串加到对它的反码上,那么结果的各个位都为1,也就是 $-1$ 的二进制补码表示。负数的特征是它的高位都是1。

在二进制补码机器上,做加和求反的硬件可以用来实现减法。操作 $a - b$ 与 $a + (-b)$ 是相同的,通过对 $b$ 求反加1得到 $-b$ 。

### C.1.3 按位二进制逻辑运算符

运算符 $\&$ (与)、 $\wedge$ (异或)和 $|$ (或)都是二元运算符。它们都用整型表达式作为运算数,两个运算数是按位运算。下表说明了作用在1位上的位运算符。

$a$	$b$	$a \& b$	$a \wedge b$	$a   b$
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

下表给出了一些作用在整型变量上的位运算符的例子:

声明和初始化		
int a = 33333, b = -77777;		
表达式	表示	值
a	00000000 00000000 10000010 00110101	33333
b	11111111 11111110 11010000 00101111	-77777
a & b	00000000 00000000 10000000 00100101	32805
a ^ b	11111111 11111110 01010010 00011010	-110054
a   b	11111111 11111110 11010010 00111111	-77249
~(a   b)	00000000 00000001 00101101 11000000	77248
(~a & ~b)	00000000 00000001 00101101 11000000	77248

## C.1.4 左移运算符和右移运算符

移位运算符的两个运算数必须是整型表达式，在运算数的两边完成整型提升，表达式整体的类型与提升后的左运算数相同。形式为

$expr1 \ll expr2$

的表达式使得 $expr1$ 的位表示被按 $expr2$ 指明的数目进行左移，并在低位端移入0。

声明和初始化		
char c = 'Z';		
表 达 式	表 示	行 为
c	00000000 00000000 00000000 01011010	不移位
c << 1	00000000 00000000 00000000 10110100	左移1位
c << 4	00000000 00000000 00000101 10100000	左移4位
c << 31	00000000 00000000 00000000 00000000	左移31位

即使c仅存储在1个字节中，在表达式中也把它提升为int。在对移位表达式求值时，分别在运算符的两边对运算数进行整型提升，表达式的类型与提升后的左运算数相同。这样，像c <<1这样的表达式的值按4个字节存储。

右移运算符>>与左移运算符是不对称的。对于无符号整型表达式，在右端移入0；对于有符号整型表达式，一些机器移入0，而另一些机器移入符号位（请参见练习9）。符号位是最高位，对于非负整数它是0，对于负整数它是1。

声明和初始化		
int a = 1 << 31; /* shift 1 to the high bit */ unsigned b = 1 << 31;		
表 达 式	表 示	行 为
a	10000000 00000000 00000000 00000000	不移位
a >> 3	11110000 00000000 00000000 00000000	右移3位
b	10000000 00000000 00000000 00000000	不移位
b >> 3	00010000 00000000 00000000 00000000	右移3位

注意，在我们的机器上对int移入符号位，而在其他台机器上移入的可能是0。要避免这一难题，程序员在使用位运算符时应经常使用无符号类型。

如果移位运算符的右运算数是负数，或它的值等于或超过了用于表示左运算数的位数，那么它的行为是无定义的。程序员有责任保证右运算数的值处于正确的范围内。

下表说明了移位运算符的优先级规则和结合性规则。为了节省空间，我们仅给出了两个低位字节。

声明和初始化			
unsigned a = 1, b = 2;			
表 达 式	等价表达式	表 示	值
a << b >> 1	(a << b) >> 1	00000000 00000010	128
a << 1 + 2 << 3	(a << (1 + 2)) << 3	00000000 01000000	64
a + b << 12 * a >> b	((a + b) << (12 * a)) >> b	00001100 00000000	3072

## C.2 掩码

掩码是用于从其他变量或表达式抽取所需要位的常量或变量。因为int型的常量1的位表示是

```
00000000 00000000 00000000 00000001
```

可以用它决定int型表达式的低位。下面的代码使用这个掩码，并显示0和1的交替序列：

```
int i, mask = 1;
for (i = 0; i < 10; ++i)
    printf("%d", i & mask);
```

如果要找出一个表达式中的一个特殊位的值，我们就可以使用在该位为1而在其他位为0的掩码。例如，我们可以用表达式1<<2作为从右数第3位的掩码。表达式

```
(v & (1 << 2)) ? 1 : 0
```

的值是为0还是1，就依靠v的第三位。

另一个掩码是255，它等于 $2^8 - 1$ ，它的位表示为

```
00000000 00000000 00000000 11111111
```

因为开通了一个低位字节，表达式

```
v & 255
```

产生的值的位表示为：高位字节是0，低位字节与v的低位字节相同。我们把它表达为“255是低位字节的掩码”。

## C.3 按位显示int

我们在本节中讨论的bit\_print()函数是一个典型的系统软件程序示例。对于任何编写涉及到机器底层的软件的人来讲，bit\_print()函数都是必须的，因为使用它程序员能明白正在发生什么。对于程序初学者而言，用bit\_print()作为帮助来提供概念框架是很有用的。

我们的bit\_print()函数用掩码显示int型的位表示，程序员可用这个函数查明在内存中如何表示表达式的值。

```
/* Bit print an int expression. */
#include <limits.h>
void bit_print(int a)
{
    int i;
```

```

int    n = sizeof(int) * CHAR_BIT;    /* in limits.h */
int    mask = 1 << (n - 1);           /* mask = 100...0 */

for (i = 1; i <= n; ++i) {
    putchar(((a & mask) == 0) ? '0' : '1');
    a <<= 1;
    if (i % CHAR_BIT == 0 && i < n)
        putchar(' ');
}

```

### 对函数bit\_print()的解析

- #include <limits.h>

在ANSI C的limits.h中定义了符号常量CHAR\_BIT。在传统C中，通常limits.h是不可用的。在大多数系统中CHAR\_BIT是8，CHAR\_BIT表示在一个char中的位数，等价于一个字节的位数。ANSI C要求一个字节至少要有8位。

- int n = sizeof(int) \* CHAR\_BIT; /\* in limits.h \*/

因为我们要在机器字为2个或4个字节的机器上使用这个函数，所以我们用变量n表示机器字的位数。我们希望表达式sizeof(int)的值是2或4，在limits.h中定义的符号常量CHAR\_BIT是8。这样，我们希望把n初始化为16或32，具体值依赖于机器。

- int mask = 1 << (n - 1); /\* mask = 100...0 \*/

考虑到运算符的优先级，初始化中的括号是不必要的，我们在这里加括号只是为了易读。由于<<的优先级高于=，所以要先对表达式1<<(n-1)求值。假设n的值是32，常量1仅使低位开通，表达式1<<31把该位移到高位。这样mask把所有位关闭，只有最高位（符号位）除外。

- for (i = 1; i <= n; ++i) {
 putchar(((a & mask) == 0) ? '0' : '1');
 a <<= 1;
 .....
 }

首先考虑一下表达式

(a & mask) == 0

如果a的最高位是关闭的，表达式a & mask就使所有的位关闭，表达式(a & mask) == 0为真(true)。相反地，如果a的最高位是打开的，表达式a & mask就打开使所有的位，表达式(a & mask) == 0为假(false)。现在考虑一下表达式

((a & mask) == 0) ? '0' : '1');

如果a的最高位是关闭的，条件表达式的值就为'0'；否则它的值为'1'，因此，如果最高位是关闭的，putchar()显示0；否则显示1。

- putchar(((a & mask) == 0) ? '0' : '1');
 a <<= 1;

在显示a的高位字节后，我们对其左移一位，并把结果放于a中。回想一下，a<<=1;等价于a = a<<1;。表达式a<<1的值的位模式与a相同，只是左移了一位。这个表达式本身没有改变内存中a的值。与此相反，表达式a<<=1改变了内存中a的值。其作用是把下一位移向高位，为下次显示做准备。

```
• if (i % CHAR_BIT == 0 && i < n)
    putchar(' ');
```

如果我们假设符号常量CHAR\_BIT是8，这段代码在显示每8位组后显示一个空格。 ■

bit\_print() 函数在各CHAR\_BIT位组间显示一个空格，这样做不是必须的，但使得输出更易读。

## C.4 压缩和解压

可用位表达式进行数据压缩，这样不但能节省空间，而且也能节省时间。在机器字为4个字节的机器上，每个指令周期并行地处理32位。下面的函数可用于把4个字符压缩成一个int，该函数用移位运算符按字节压缩。

```
/* Pack 4 characters into an int. */
#include <limits.h>

int pack(char a, char b, char c, char d)
{
    int p = a;    /* p will be packed with a, b, c, d */

    p = (p << CHAR_BIT) | b;
    p = (p << CHAR_BIT) | c;
    p = (p << CHAR_BIT) | d;
    return p;
}
```

为了测试这个函数，我们编写一个main()中含有以下行的程序：

```
printf("abcd = ");
bit_print(pack('a', 'b', 'c', 'd'));
putchar('\n');
```

下面是我们的测试程序的输出：

```
abcd = 01100001 01100010 01100011 01100100
```

观察一下，输出的高位字节的值是97，即'a'，其余字节的值是98、99和100。因此pack()的工作是正确的。

写完了pack() 现在我们要从32位int中抽取字符，我们再次用掩码完成这项工作。

```
/* Unpack a byte from an int. */
#include <limits.h>

char unpack(int p, int k)    /* k = 0, 1, 2, or 3 */
{
    int n = k * CHAR_BIT;    /* n = 0, 8, 16, or 24 */
    unsigned mask = 255;     /* low-order byte */

    mask <<= n;
    return ((p & mask) >> n);
}
```

### 对函数unpack()的解析

```
• #include <limits.h>
```

我们引入这个头文件是因为它含有符号常量CHAR\_BIT的定义。CHAR\_BIT表示了一个字节中的位数，在大多数机器上它是8。

```

• char unpack(int p, int k)      /* k = 0, 1, 2, or 3 */
{
    .....

```

我们把参数`p`看作是被压缩的`int`，其字节的号分别为0、1、2和3。参数`k`表示我们要处理的字节：如果`k`的值是0，就表示我们要处理低位字节；如果`k`的值是1，就表示我们要处理次低位字节；以此类推。

```

• int      n = k * CHAR_BIT; /* n = 0, 8, 16, or 24 */

```

如果我们假设`CHAR_BIT`是8，`k`为0、1、2或3，那么就用0、8、16或24初始化`n`。

```

• unsigned mask = 255;      /* low-order byte */

```

常量255是很特别的，要理解它首先考虑256。因为256=2<sup>8</sup>，所以256的位表示为除了从低位开始数的第9位是1外，其余各位都是0。255小于256，它的位表示是前8位都是1，其余各位全是0（请参见练习1）。因此`mask`的二进制表示是

```
00000000 00000000 00000000 11111111
```

```
• mask <<= n;
```

我们假设`CHAR_BIT`是8。如果`n`的值是0，那么在`mask`中的位左移8位。在这样的情况下，我们把在内存中存储的`mask`看作是

```
00000000 00000000 11111111 00000000
```

如果`n`的值是16，那么在`mask`中的位左移16位。在这样的情况下，我们把在内存中存储的`mask`看作是

```
00000000 11111111 00000000 00000000
```

以类似的方式，如果`n`的值是24，那么仅`mask`的高位字节全为1。

```
• #(p & mask) >> n
```

因为`&`的优先级低于`>>`，所以这对括号是有必要的。假设`p`的值是-3579753（选择它是因为它的位模式适合我们的需要），`n`的值是16。下表说明了所要发生的情况。

表达式	二进制	十进制
<code>p</code>	11111111 11001001 01100000 10010111	-3579753
<code>mask</code>	00000000 11111111 00000000 00000000	16711680
<code>p &amp; mask</code>	00000000 11001001 00000000 00000000	13172736
<code>(p &amp; mask) &gt;&gt; n</code>	00000000 00000000 00000000 11001001	201

```
• # return ((p & mask) >> n);
```

因为`unpack()`的类型为`char`，在把该表达式回传到环境前，`int`型的表达式`(p & mask) >> n`被转换成`char`型的。在把`int`转换为`char`时，仅保留低位字节，而把其他字节丢弃。 ■

想象一下怎样在一个整数中记录一个雇员的简写记录。我们假设可以用9个位存储“雇员标识号”，用6个位存储“职业类型”（这能存储64种职业），用1位存储雇员的“性别”。这3个域需要16位，在机器字为4个字节的机器上它是一个`short`型整数。我们可以把这3个位域看作是：



标 识	职业类型	性 别
bbbbbbbbb	bbbbbb	b

下面这个函数可以用在用来把雇员数据输入到short型值的程序中。相反的问题是从short型值读数据，利用mask可解决该问题。

```
/* Create employee data in a short int. */
short create_employee_data(int id_no, int job_type,
                           char gender)
{
    short    employee = 0;    /* start with all bits off */

    employee |= (gender == 'm' || gender == 'M') ? 0 : 1;
    employee |= job_type << 1;
    employee |= id_no << 7;
    return employee;
}
```

## C.5 常见的编程错误

程序员把&与&&、|与||相混淆是一种常见的错误。像应该用=而用了==这样的错误也是一种常犯的错误。在大多数情况下，结果表达式不会导致语法错误，这是因为编译器不会告诉你已经犯了什么错误。为了说明这些想法，我们假设a、b和c是int型变量。下边的表达式都是合法的：

```
a == b && c      a == b & c      a = b & c
```

它们是可以使用的。例如，它们可用于控制if语句或循环。

在使用位运算符时，程序员必须考虑表达式的类型。下面是一个看起来无害的初始化，其实它会导致难以调试的错误：

```
unsigned    lo_byte = ~0 >> 24; /* turn the low byte on */
```

程序员试图为低位字节创建一个掩码。因为0使所有的位都是0，所以~0使所有的位都是1，这样位被右移24次，以打算在高位端移入0。程序员要记住，对于unsigned字节，移进的总是0。但是这个初始化是失败的，在用bit\_print()看发生了什么时，程序员会发现所有的位都是1！问题是0的类型是int，操作~0 >> 24并没有改变该类型；最终的结果被赋给lo\_type，它的类型为unsigned，但这是毫无帮助的。正确的初始化如下：

```
unsigned    lo_byte = ~(unsigned) 0 >> 24;
```

外层的括号不是必须的，使用它们是为了易读。

在任何机器上，类型为long的掩码都是可接受的。然而，我们在机器字为2个字节的机器上进行下面的初始化时，我们的代码就不能如想象的那样工作：

```
long    mask = 1 << 31;    /* turn the high bit on */
```

我们犯了一个异乎寻常的错误。表达式1的类型为int，因此要以2个字节存储它。在1被左移31次后，所有的为都变成了0，这样mask实际上被赋值为0，这不是我们所想要的。下面是正确的初始化：

```
long    mask = (long) 1 << 31;    /* turn the high bit on */
```

## C.6 系统考虑

一些机器右移进的是0，而一些机器右移进的是符号位，这样会引起麻烦。假设你所用的机器移进的总是0，那么用int与用unsigned所产生的效果是相同的。如果以后你把这样的代码转移到移进的是符号位的机器上，所使用的int就会引起意想不到的结果。

在C中，任何类型为char或short的表达式都被提升到int，系统以保存值的方式实现这一点。任何unsigned char或unsigned short在左边添0，而符号扩充发生在有符号量上。这意味着符号位被扩散。下表说明了这一点：

表达式	二进制表示	值
128	00000000 00000000 00000000 10000000	128
(char) 128	11111111 11111111 11111111 10000000	-128
-1 ^ 128	11111111 11111111 11111111 01111111	-129
(char) (-1 ^ 128)	00000000 00000000 00000000 01111111	127
255	00000000 00000000 00000000 11111111	255
(unsigned char) 255	00000000 00000000 00000000 11111111	255

我们的系统把简单的char看作是signed char。在把int类型转换为char时，仅保留低位字节，把其余的字节都丢弃，但把表达式提升为int型。对于有符号量，char中的第8位决定把什么填充到左边。与此相反，用0填充unsigned char，而不管符号位。

提升和符号扩散的思想解释了在一些机器上为什么用char测试EOF会失败（请参见练习19）。

在使用多字符的字符常量时，程序员必须要小心。在大多数系统中，程序员可以使用像'ab'这样的常量，但实际上如何存储字节是与系统相关的。

### 小结

- 位运算符为程序员提供了访问整型表达式中的位的手段。通常我们把这些运算符的运算数看作是位串。
- 使用位表达式可跨越字节界对数据进行压缩，这样不但能节省空间，而且还能节省时间。在机器字为4个字节的机器上，每个指令周期并行地处理32个位。
- 大多数机器对整数采用二进制补码表示，这种表示的高位是符号位，对于负整数它是1，对于非负整数它是0。
- 明显地，位运算符是与系统相关的。左移移进的是0。对于右移，情况比较复杂。如果整型表达式是unsigned，那么移进的是0。如果表达式是有符号类型，所移进的是与系统相关的。一些机器移进的是符号位，这意味着，如果符号位是0就移进0；如果符号位是1，就移进1。一些机器在任何情况下移进的都是0。
- 掩码是常用的特殊值，它与1一起设置一系列的位，与&一起抽取一系列的位。
- 压缩把一些明确的值放进一个给定变量的子域，解压抽取这些值。
- 函数bit\_print()是一个软件工具，程序员可以用它在位的层次上看内存中所发生的情况。

## 练习

1. 使用`bit_print()`函数创建一个含有 $n$ 、 $2^n$ 的二进制表示和 $2^n - 1$ 的二进制表示的表，这里 $n=0, 1, 2, \dots, 32$ 。如果你的机器的机器字是两个字节，那么你的程序的输出看起来应该如下所示：

```
0: 00000000 00000001 00000000 00000000
1: 00000000 00000010 00000000 00000001
2: 00000000 00000100 00000000 00000011
....
15: 10000000 00000000 01111111 11111111
....
```

在完成此项工作后，手工写出一个类似的表，它含有 $n$ 、 $10^n - 1$ 和 $10^{n-1}$ ，这里 $n=0, 1, 2, \dots, 7$ 。在该表中写出以10为基的数。你看这两个表相像吗？提示：使用下面的代码：

```
int i, power = 1;
for (i = 0; i < 32; ++i) {
    printf("%2d: ", i);
    bit_print(power);
    printf(" ");
    bit_print(power - 1);
    putchar('\n');
    power *= 2;
}
```

2. 编写一个以十进制整数串作为输入的函数。可以把在串中的每个字符看作是十进制数字。应该把数字转换成4位的二进制串，并压缩到`int`中。如果`int`有32位，就可以在其中压缩8个数字。在进行测试时，你在屏幕上能看到：

```
Input a string of decimal digits: 12345678
12345678 = 0001 0010 0011 0100 0101 0110 0111 1000
```

再编写一个逆向函数，它对`int`解压，返回原串。提示：下面是一个逆向函数的开头部分：

```
int convert(char *s)
{
    char *p;
    int a = 0; /* turn all bits off */

    for (p = s; *p != '\0'; ++p) {
        a <<= 4;
        switch (*p) {
            case '1':
                a |= 1;
                break;
            case '2':
                ....
        }
    }
}
```

3. 在本章中，对一些数的二进制表示是容易检验正确性的，但对有些数则不这样。使用`bit_print()`检验一些较为难以检验的表示。例如，试一下70707及其按位求反，本书中的值正确吗？

4. 假设整数有16位的二进制补码表示。写出-1、-1、-101和-1023的二进制表示。提示：回忆一下，负整数的二进制补码表示是通过对其正整数求反加1得到的。

5. Carole、Barbara和Debra都对16个公民投票进行表决。假设用16个整数的位存储每个人的投票。编写一个用下述代码开始的函数：

```
short majority(short a, short b, short c)
{
    ....
}
```

该函数应该用分别存储在a、b和c中的Carole、Barbara和Debra的投票作为输入。它应该返回a、b和c中的半数以上数位。

6. 编写一个用下述代码开始的函数：

```
short majority(short a, short b, short c)
{
    ....
}
```

该函数把a左移n位，同时把高位又引入到低位。下面是两个为char（不是为int）定义的循环移位操作的例子：

对10000001 循环移位1次产生00000011

对01101011 循环移位3次产生01011011

用bit\_print()编写一个程序，测试你的函数。

7. 编写一个倒置一个int的位表示的函数。下面是两个为char（不是为int）定义的倒置操作的例子：

对01110101倒置产生10101110

对10101111倒置产生11110101

8. 编写一个从32位表达式每隔一位进行抽取的函数，抽取的结果是一个16位表达式，函数应返回这个结果。你的函数在机器字为2或4个字节的机器上都应该能工作。

9. 你的机器移入符号位吗？下面的代码能帮你做这项检查：

```
int      i = -1;      /* turn all bits on */
unsigned u = -1;

if (i >> 1 == u >> 1)
    printf("Zeros are shifted in.\n");
else
    printf("Sign bits are shifted in.\n");
```

请解释这段代码为什么能工作。

10. 我们可以用整数以日/月/年的形式写出20世纪的日期。例如，1/7/33表示1933年7月1日。编写一个压缩存储日、月和年的函数。因为日有31个值，月有12个值，年有100个值，所以我们用5个位表示日，4个位表示月，7个位表示年。函数应该把作为整数的日、月和年作为输入，返回压缩成16位的整数，作为日期。编写另一个解压函数，并编写一个测试这两个函数的程序。

11. 编写一个直接作用于被压缩日期的函数（参见上一个练习），它产生日历中的下一个被压缩日期。把这个程序与练习2编写的程序做对照。

12. 重新编写3.12节“问题求解：布尔变量”中的程序。用5个在char型变量b中的低位表示5个布尔变量b1, ..., b5。

13. 利用机器运算重新编写前一个练习中的程序。手工模拟表明，对b的位表示加1的效果等同于嵌套的for语句的效果。在本练习中，你的程序应该用单的非嵌套的for语句产生表。

14. 编写一个要求用户输入整数n的交互式程序。用bit\_print()写出n、2×n、4×n和8×n的二进制表示。你能解释所发生的情况吗？提示：以10为基，在你用一个数乘10时会发生什么？此处的k=1, 2, 3。

15. 假设你工作在一个二进制补码的机器上，它不提供对位求反运算。编写一个函数bit\_complement()，它仅用算术运算实现这个运算。提示：若a是一个类型为int的变量，

且二进制求反可用, 则表达式 $a$ 和  $\sim a - 1$ 的值相同。

16. 考虑函数`pack()` (请参见C.4节“压缩和解压”), 该函数体由4个语句组成。重新编写这个函数, 把这4个语句压缩成一个单`return`语句。

17. 重新编写函数`pack()`, 仅使用算术运算。

18. 大多数机器把简单的`char`实现为`signed char`或`unsigned char`。在你的机器上是怎样实现简单的`char`的? 试一下下面的代码:

```
char      c = 128;      /* turn the high bit on */
signed char  sc = 128;
unsigned char uc = 128;

printf("c = %d  sc = %d  uc = %d\n", c, sc, uc);
```

虽然在内存中用单字节存储类型为`char`的变量, 作为表达式, 要把它提升成`int`。把`%d`格式变为`%u`, 你看到了提升的效果了吗?

19. 假设系统把简单的`char`实现为`unsigned char`。请解释下述代码的作用:

```
char  c = EOF;

if (c == EOF)
    printf("Truth!\n");
else
    printf("This needs to be explained!\n");
```

我们现在假设已经引入了`stdio.h`。如果你的系统把简单的`char`实现为`signed char`, 那么要了解这段代码怎样在不同的系统上运行, 仅需要把声明改为

```
unsigned char c = EOF;
```

提示: 在`stdio.h`中查出`EOF`的值, 然后仔细阅读C.6节“系统考虑”。如果你仍感到迷惑, 就使用`bit_print()`看会发生什么情况。

## 附录D ANSI C与传统C的比较

在本附录中，我们要列出ANSI C与传统C间的主要区别。在适当之处，我们将给出例子。这个列表是不完整的，仅列出了要点。

### D.1 类型

- 增加了关键字signed。
- 规定了三种字符类型：简单char、signed char和unsigned char。在实现中可能把简单char表示为signed char或unsigned char。
- 关键字signed可用在对有符号整型的声明中和类型转换中。char有些例外，对它的使用总是可选的。
- 在传统C中，类型long float等价于double。由于long float很少用，ANSI C没有采用它。
- ANSI C增加了类型long double。这种类型的常量带有后缀L。long double提供的精度和宽度都多于double，但不必如此。
- 关键字void用于指明一个函数没有参数或无返回值。
- 类型void \*用于普通指针。例如，malloc()的函数原型为

```
void *mloc(size_t size);
```

可以对普通指针赋任何类型的指针值，可以用普通指针值向任何指针类型的变量赋值。类型转换不需要。与此相反，传统C中的普通指针类型是char\*，其中需要类型转换。

- 支持枚举类型。一个例子是

```
enum day{sum, mon, tue, wed, thu, fri, sat};
```

这个例子中的枚举元是sum, mon, ..., sat。枚举元是类型为int的常量，可用作switch语句中的case标号。

### D.2 常量

- 由空白字符分隔的串被拼接，因此

```
"abc"
```

```
"def" "ghi"
```

等价于

```
"abcdefghi"
```

- 串常量是不可修改的。（不是所有的编译器都强制如此。）
- 可用字母后缀指明数字常量的类型。下面是一些例子：

```
123L      /* long          */
123U      /* unsigned       */
123UL     /* unsigned long  */
1.23F     /* float          */
1.23L     /* long double    */
```

后缀可以小写，也可以大写。无后缀的数字常量是足以包含该常量值的类型。

- 数字8和9不再被认为是八进制数字，在八进制常量中不能使用它们。
- 引入了用\x开始的十六进制转义序列。像用\0开始的八进制转义序列一样，它们用在字符和串常量中。

### D.3 声明

- 增加了类型限定符const。其含义是用它声明的变量是不可修改的。（编译器并不总是强制如此。）
- 增加了类型限定符volatile。其含义是程序外部的客体可以修改用它声明的变量。例如，在一些系统中可以在头文件 `errno.h` 中做如下声明：

```
extern volatile int errno;
```

### D.4 初始化

- 在ANSI C中，可以对数组和结构这样的聚集进行初始化。在传统的C中，它们的存储类型可以是外部的或static。
- 可以对联合进行初始化。初始化涉及到联合的第一个成员。
- 可以用只有n个字符的串常量对尺寸为n的字符数组进行初始化。例如，

```
char today[3] = "Fri";
```

在"Fri"中的串结束标记\0并没有被复制到today。

### D.5 表达式

- 由于对称的原因，增加了一元加运算。
- 在传统C中，编译器对包含像+或\*这样的可交换的二元运算符的表达式可进行重新排布，即使在程序中使用了括号也是如此。例如，对语句

```
x = (a + b) + c;
```

中的变量，编译器可能以想不到的次序进行相加。在ANSI C中，并不如此，必须遵循括号的优先级。

- 可以显式地或隐式地间接访问指向函数的指针。例如，如果f是指向一个有3个参数的函数的指针，那么表达式f(a, b, c) 等价于 (\*f)(a, b, c)。
- sizeof运算符产生类型为size\_t的值。在stddef.h中给出了size\_t的类型定义。
- 在没有把类型为void\*的指针类型转换为适当的类型前，不能间接访问它，但可以在逻辑表达式中使用它，可与另一个指针相比较。

### D.6 函数

- ANSI C为函数定义提供了新的语法。在函数名后要写出参数声明列表，并用括号括起来。例如，

```
int f(int a, float b)
{
    .....
```

而在传统C中应写出：

```
int f(a, b)
int    a;
float  b;
{
    .....
```

• ANSI C提供了函数原型，这是一种新的对函数进行声明的风格。要在函数名后写出参数类型列表，并用括号括起来，而标识符是可选的。例如，

```
int f(int, float); 和  int f(int a, float b);
```

是等价的，而在传统C中，应写为：

```
int f();
```

如果函数没有参数，就把void用作函数原型中的参数类型；如果函数的参数个数是不确定的，就在函数原型中用省略号作为最右侧的参数。

• 在函数定义块的外部对参数标识符重新声明是非法的。下述的代码说明了此类错误：

```
void f(int a, int b, int c)
{
    int    a;    /* error: a cannot be redefined here */
    .....
```

虽然在传统C中这是合法的，但在ANSI C中这几乎就是编程错误。实际上这是一个难以发现的错误。

• 可以把结构和联合作为传递给函数的参数，函数也能把它们作为返回值，这种传递机制是按值调用，这意味着要进行局部复制。

## D.7 转换

• 类型为float的表达式并不自动地转换成double型。

• 在对给函数的参数求值时，结果值被转换成由函数原型指定的类型。所进行的转换应是兼容的，否则会发生语法错误。

• 要非常仔细地说明算术转换（请参见6.9节“转换和类型转换”）。在ANSI C中，转换的基本思想是尽可能地保留值。根据这点，就需要一些从机器字为两个字节的机器到机器字为四个字节的机器的转换规则。

• 移位运算符的结果类型与右运算数无关。在ANSI C中，对各运算数执行整型提升，结果类型就是被提升的左运算数的类型。

## D.8 数组指针

• 很多传统C编译器不允许地址运算符&的运算数是数组。在ANSI C中，这是合法的，可以使用指向多维数组的指针。下面是一个例子：

```
int    a[2][3] = {2, 3, 5, 7, 11, 13};
int    (*p)[3];          /* the first dimension
                           need not be specified */
p = &a;
printf("%d\n", (*p)[1][2]); /* 13 is printed */
```



## D.9 结构和联合

- 可以对结构和联合进行赋值。如果s1和s2是类型相同的两个变量，那么表达式s1=s2是合法的，它把s2中的各成员值复制到s1中的相应的各成员中。

- 可以把结构和联合作为参数传递给函数，函数也能把它们作为返回值。包括结构和联合在内的所有传递给函数的参数都是以按值调用被传递的。

- 如果m是结构或联合的成员，并且函数调用f()返回类型相同的结构或联合，那么表达式f().m是正确的。

- 结构和联合能和逗号运算符和条件表达式一起使用。下面是一些例子：

```
int      a, b;
struct s  s1, s2, s3;

.....
(a, s1)      /* comma expression having structure type */
a < b ? s1 : s2 /* conditional expression struct type */
```

- 如果expr是结构或联合，m是成员，那么形式为expr.m的表达式是正确的。然而，能对expr.m赋值仅当能对expr赋值。即使像

```
(s1 = s2).m    (a, s1).m    (a < b ? s1 : s2).m    f().m
```

这样的表达式是正确的，它们也不能出现在赋值运算符的左边。

## D.10 预处理器

- 预处理指令不必从第一列开始。

- 增加了以下预定义宏：

```
__DATE__ __FILE__ __LINE__ __STDC__ __TIME__
```

不可再对它们进行重定义或取消定义（请参见B.6节“预定义宏”）。

- 在取消宏前不能对它重定义。多重定义是允许的，只要它们是相同的。

- 增加了预处理器运算符#和##。一元运算符#使得宏定义中的形式参数“串化”。二元运算符##合并标记（请参见B.7节“运算符#和##”）。

- 增加了预处理器运算符defined（请参见B.5节“条件编译”）。

- 增加了预处理指令#elif、#error和#pragma（请参见B.5节“条件编译”和见B.9节“#error和#pragma的用法”）。

- 在传统C中，在ctype.h中把toupper()和tolower()定义为宏：

```
#define toupper(c) ((c)-'a'+'A')
#define tolower(c) ((c)-'A'+'a')
```

仅在c的值是小写字符时，宏toupper(c)才能正常工作。类似地，仅在c的值是大写字符时，宏tolower(c)才能正常工作。在ANSI C中，把toupper()和tolower()实现为函数或宏，但行为是不同的。如果c的值是小写字符，那么toupper(c)返回相应的大写字符值；如果c的值不是小写字符，就返回c的值。tolower(c)也与此类似。

- 在ANSI C中，也可以把宏用作函数。假设已经引入了stdio.h，那么putchar(c)是宏调用，而(putchar)(c)是函数调用。

## D.11 头文件

- ANSI C中增加了头文件。头文件`stdlib.h`含有标准库中的很多函数的原型。
- 头文件`float.h`和`limits.h`含有描述执行特征的宏定义。ANSI C要求每种算术类型都要支持一定的最小值和值域。

## D.12 其他

- 在传统C中，运算符`+=`和`=+`是同义的，只是`=+`方式较旧。在ANSI C中不允许使用`+=`、`=*`这样的运算符。
- ANSI C编译器把像`+=`这样的赋值运算符处理为单标记。在传统C中，把它处理为两个标记，这样就可以在`+`和`=`间插入空格。在传统C中`a + = 2`是合法的，而在ANSI C中它是非法的。
- 标号标识符、变量标识符、标记名以及结构和联合的成员名都有独特的命名空间。`enum`、`struct`和`union`的所有标记构成一个单命名空间。
- 如果两个标识符的前 $n$ 个字符是不同的（ $n$ 必须不小于31），就认为它们是不同的。
- 控制`switch`语句的表达式可以是整型的，但不可以是浮点型的。在`case`标号中的常量整型表达式可以是整型的，其中也包括枚举类型。
- 指针和`int`是不可交换的。不用类型转换，仅能把整数0赋给指针。
- 指针表达式能指向一个已分配了的数组以外的元素。
- 更谨慎地定义了外部声明和连接规则。
- 对标准库及其相关的头文件做了很多改变。

## 附录E ASCII字符编码表

美国信息交换标准码										
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	np	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

### 怎样读该表

- 观察一下，字符A在第6行第5列，这意味着字符A的值是65。

### 一些观察

- 从0到31的字符码和字符码127是不可打印的。
- 字符码32显示一个空格。
- 数字0到9的字符码是连续的，字母A到Z的字符码是连续的，字母a到z的字符码是连续的。
- 大写字母与相应的小写字母的差是32。

一些缩写的含义			
bel	响铃	ht	横向跳格
bs	退格	nl	换行
cr	回车	nul	空
esc	转义	vt	纵向跳格

**注意** 在大多数UNIX系统上，使用命令`man ascii`，可在屏幕上按十进制、八进制和十六进制显示ASCII表。

## 附录F 运算符的优先级和结合性

下表给出了C++运算符的优先级和结合性，并在可能出现歧义的地方使用了括号。

运算符	结合性
() [] -> ++ (后缀) -- (后缀)	从左到右
++ (前缀) -- (前缀) ! ~ sizeof (类型) + (一元) - (一元) & (地址) *(间接)	从右到左
* / %	从左到右
+ -	从左到右
<< >>	从左到右
< <= > >=	从左到右
== !=	从左到右
&	从左到右
^	从左到右
	从左到右
&&	从左到右
	从左到右
?:	从右到左
= += -= *= /= *= >>= <<= &= ^=  =	从右到左
, (逗号运算符)	从左到右

# 索引

## 符号开头的索引项

- = 减等于运算符, 35
- 减运算符, 31
- 自动减量运算符, 33 ~ 34
- != 不等于运算符, 48, 50
- ! 非运算符, 48 ~ 49, 52
- # 预处理器运算符, 360, 375
- ## 预处理器运算符, 360, 375
- #define, 8 ~ 9, 12, 89, 159, 354
- #elif, 378
- #endif, 358
- #error, 378
- #if, 358
- #ifdef, 358
- #ifndef, 358
- #include, 8, 37, 89, 354
- #line, 362
- #pragma, 362, 375
- #underscore, 358 ~ 359
- % 格式, 269, 274
- % 格式, 31
- % 取模运算符, 7, 31
- %% 格式, 269, 271, 274
- %= 取模等于运算符, 35
- %c 格式, 110, 156, 162, 269, 272
- %E 编译器选项, 299
- %e 格式, 269, 274
- %E 格式, 274
- %f 格式, 270
- %g 格式, 140, 269, 274
- %G 格式, 269, 274
- %hd 格式, 142, 252
- %l format, 269, 274
- %n 格式, 269, 274
- %o 格式, 269, 274
- %p 个数, 168, 183, 269, 274
- %s 格式, 269, 274
- %u 格式, 169, 269, 274
- %x 格式, 269, 274
- %X 格式, 274
- & 地址运算符, 11, 166, 167, 182
- & 与 (位) 运算符, 363, 364
- && 与 (逻辑) 运算符, 48, 50 ~ 52, 69
- &= 与等于运算符, 35
  - (函数) 体, 79
  - (函数的) 头, 79
- \* 乘运算符, 6, 31
- \* 间接引用或间接运算符, 166 ~ 167, 169, 180
- \*= 乘等于运算符, 35
  - , 逗号标点, 31
  - , 逗号运算符, 61, 378
- .h 文件, 90, 93, 308
- .a 文件, 299
- .c 文件, 90, 93, 308
- .lib 文件, 299
- .obj 文件, 92
- .o 文件, 92, 306
- .成员运算符, 246
- / 除运算符, 31
- /\*\*/ 注释对, 28
- /= 除等于运算符, 35
- ; 分号, 31, 53, 69
- ? 异或 (位) 运算符, 363, 364
- ? 抑扬字符, 275
- ? 条件表达式运算符, 66 ~ 67

?= 异或等于运算符, 35  
 [], 190, 197  
 []方括号, 190, 197  
 [...] 扫描集, 274  
 \ 串继续, 45  
 \ 反斜线, 30~31, 46  
 \ 转义符, 30, 111  
 \0 串结束标志, 211  
 \0 空字符, 112, 211, 220~221  
 \a 警告, 112  
 \b 退格, 112  
 \f 走纸, 112  
 \n 换行, 4, 30, 111  
 \r 回车符, 112  
 \t 制表符, 112  
 \_ 下划线, 29  
 —TIME—, 360, 375  
 {} 花括号, 31, 67~68, 191  
 {}, 31, 67~68, 191  
 | 或(位)运算符, 363, 364  
 || 或(逻辑)运算符, 48, 50~52  
 |= 或等于运算符, 35  
 ~ 求反(位)运算符, 363~364  
 ‘ 单引号, 111, 112  
 “w” 文件打开模式, 279  
 “wb” 文件打开模式, 279  
 《A Book On C》, Kelley/Pohl, 231, 313  
 《Numerical Recipes in C》, 印刷, 99  
 《The Art of Computer Programming》, Knuth, 175  
 + 加(二元)运算符, 31  
 + 一元加运算, 376  
 ++ 自动增量运算符, 33~34, 39, 45  
 += 加等于运算符, 35  
 << 输出运算符, 19  
 << 左移运算符, 363, 365  
 <<= 左移等于运算符, 35  
 <= 小于等于运算符, 48~49

<> 尖括号, 3  
 <> 三角括号, 3  
 < 小于运算符, 48~49  
 = 等于运算符, 6, 8, 34~35, 68~69, 260  
 == 相等运算符, 14, 39, 48, 50, 68~69, 260  
 > 大于运算符, 48~49  
 -> 结构指针运算符, 246, 248  
 >= 大于等于运算符, 48~49  
 >> 右移运算符, 363, 365  
 >>= 右移等于运算符, 35  
 ... 省略号, 83  
 0 (NULL常量), 165, 254  
 -lm编译器选项, 143  
 字母开头的索引项

## A

A Sorting Problem and Its Complexity, Pohl, 230  
 a.out命令, 17, 297  
 abort(), 345, 361  
 abs(), 143, 333, 343  
 ab文件打开模式, 279  
 acos(), 332  
 add(), 253  
 add\_ten()程序, 13  
 add\_ten()解析, 13  
 add\_to\_n程序, 60  
 add\_to\_n解析, 60  
 ADT, 245~260, 315  
 alias命令, 305  
 ANSI C, 375~379  
   #error, 362  
   #pragma, 362  
   %p 格式183  
   数组初始化, 191  
   assert(), 361  
   自动初始化, 253

- 字符类型, 129
  - 与传统C比较, 100, 375
  - Const, 179
  - 转换字符, 284
  - 外部变量, 173
  - 浮点类型, 132
  - 函数原型, 98, 183
  - 标识符辨别, 29, 42
  - 大文件访问, 286
  - 命名空间, 153
  - 指针赋值, 168, 182
  - 指向void的指针, 169
  - 预定义宏, 360
  - size\_t, 200
  - 串常量, 221
  - 结构参数, 249, 260
  - 临时文件, 281
  - 定时函数, 303
  - 类型安全, 100
  - 一元加, 33
  - Volatile, 179
  - argc, 218
  - argv, 218
  - ar命令, 45, 299
  - ASCII字符码, 110, 115, 380
  - asctime(), 349
  - asin(), 332
  - asm(Borland), 29
  - assert(), 87, 329, 361
  - assert.h, 86, 209, 329, 361
  - assign\_values(), 249
  - assignment赋值, 5, 34
  - atan(), 332
  - atan2(), 332
  - atexit(), 345 ~ 346
  - atol(), 344
  - atof(), 344
  - atoi(), 344
  - average(), 201, 237
  - awk命令, 310
  - a文件打开模式, 279
- B**
- backward程序, 284
  - bases程序, 124
  - bcc命令, 18, 297
  - bc命令, 297
  - birthday程序, 40
  - bison命令, 310
  - bit\_print(), 366
  - bit\_print()解析, 367
  - bool\_tbl程序, 61
  - Borland, 29, 37, 40 ~ 42, 99
  - break, 29, 64, 65
  - bsearch(), 342
  - bubble(), 196
- C**
- C, 19
  - C++, 315
    - <<输出, 123
    - ADT, 317
    - Alias, 183 ~ 184
    - base class 基类, 321
    - 抽象基类, 323
    - 抽象数据类型, 317
    - 访问, 264, 321
    - 布尔表达式, 71
    - 布尔类型, 71
    - 引用调用, 183 ~ 184, 319
    - 类, 265
      - big\_int, 239
      - circle, 323
      - complex, 265
      - grad\_student, 322
      - rectangle, 323
      - shape, 322 ~ 323
      - square, 323

- stack, 261, 262 ~ 264, 324
- string, 317, 318, 320
- student, 322
- 客体, 315, 317, 322
- 代码复用, 321
- 注释 //, 42
- 构造器, 320 ~ 321
- 构造器重载, 321
- 容器类, 324
- 转换, 315
- 数据隐藏, 316, 321
- 解除, 320
- 声明, 71
  - const, 316
- 缺省参数, 100
- 删除, 204
- 派生类, 321
- 派生, 321 ~ 322
- 析构器, 320 ~ 321
- 动态存储分配, 320
- 封装, 315, 321
- 枚举类型, 160
- 错误, 326
- 异常, 42, 325
- 扩展精度包, 239
- 终结, 320
- 函数
  - bin\_lookup(), 240
  - friend, 318
  - greater(), 184
  - order(), 184
  - 内联, 100, 263, 356
  - 原型, 102, 315
  - 重载, 100, 102
- 层次, 321
- I/O库, 316
- 继承, 315 ~ 316, 321
- 初始化
  - 对象, 320
- 内联函数, 100, 263, 356
- 插入, 123
- 实例化, 324
- 关键字
  - bool, 71
  - catch, 42, 325
  - class, 42, 315, 317
  - const, 315
  - delete, 42, 315, 320
  - false, 71
  - friend, 318
  - inline, 315
  - new, 316, 320 ~ 321
  - private, 42, 265, 308
  - protected, 42
  - public, 42, 265, 315
  - template, 42, 324
  - throw, 42, 325
  - true, 71
  - try, 42, 325
- 库, 240
  - complex.h, 144
  - fstream.h, 287
  - iomanip.h, 124
  - iostream.h, 123 ~ 124, 316
- 控制符, 123 ~ 124
- 成员函数, 263, 315, 318
- 内存管理, 204
- 创建, 204
- 对象, 320, 321
- 面向对象编程, 315
- 运算符
  - 域区分符, 263
- 输出, 123
- 重载, 101, 318
- 参数多态性, 324
- 多态, 322, 324
- 程序
  - circle, 42



- complex, 144
- dbl\_sp, 288
- dyn\_array, 205
- dyn\_string, 320
- gcd, 19
- reverse, 264
- stack, 261
- string\_ovl, 318
- string, 317 ~ 318
- 公共继承, 321
- 纯虚函数, 323
- 引用声明, 183
- 域, 320
- 特征标记, 322
- 栈ADT, 261
- 流
  - caux, 287
  - cerr, 287
  - cin, 287
  - clog, 287
  - cout, 123, 287
  - cprn, 287
- 串, 221
- 结构成员, 261
- 风格, 325
- 标记名, 160
- 用户定义类型, 144
- 为什么转到, 315
- call\_val程序, 95
- calloc(), 169, 200 ~ 201, 255 ~ 256, 286, 342 ~ 343
- caps程序, 115, 116
- caps解析, 115
- card struct, 245 ~ 246, 249, 253, 267
- card.h, 249
- card\_tst程序, 250
- case, 29, 65, 153, 375
- cast\_day程序, 153
- cb命令, 310, 314
- cc命令, 2, 17, 297
- cdecl(Borland), 29
- ceil(), 333
- cfopen(), 291
- change(), 215
- change()解析, 216
- Char, 7, 29, 108, 128, 131, 375
- char, 7, 29, 108, 128 ~ 129, 143, 211, 218
- char\_bk程序, 228
- chk\_arrays(), 302
- chk\_arr程序, 302
- cin(C++), 19
- circle程序, 12
- class程序, 247
- clearerr(), 339
- clock(), 303, 304, 349
- close(), 352
- cls命令, 313
- cnt\_abc程序, 193
- cnt\_abc解析, 193
- cnt\_char程序, 177
- coins程序, 22
- colors程序, 160
- comma程序, 76
- compare(), 157, 301
- complex struct, 253
- compute(), 139
- compute()解析, 139
- compute\_sum(), 95
- compute程序, 138
- const, 29, 179, 327, 376
- continue, 29, 64, 65
- continue命令, 305
- control-c, 18, 70, 98
- control-d, 15, 18, 114, 293 ~ 294
- copy命令, 122
- cos(), 134, 332
- cosh(), 332

count(), 257  
 count-dn程序, 225  
 count-down(), 225, 240  
 count程序, 257  
 cout (C++), 19  
 CPU, 1  
 cp命令, 122  
 Cray, 143  
 Create\_employee\_data(), 370  
 csh壳程序, 310  
 ctime(), 349  
 ctype.h, 110, 117, 213, 329 ~ 330  
 -c编译器选项, 299

## D

data结构, 251  
 \_\_DATE\_\_, 360, 375  
 day程序, 162  
 dbl\_out(), 114  
 dbl\_out程序, 19, 113, 114  
 dbl\_out解析, 113  
 dbl\_sp程序, 279  
 dbl\_with\_caps程序, 282  
 dbl\_with\_caps解析, 282  
 dbx命令, 305  
 def\_arg程序, 101  
 delete\_list(), 259  
 depth程序, 5  
 depth解析, 5  
 difftime(), 350  
 diff命令, 310  
 dir命令, 18, 295  
 display(), 229  
 div(), 344  
 do, 62  
 dot-product(), 254  
 double, 29, 128, 132, 375  
 double, 7 ~ 8, 29, 42, 128, 132  
 double\_space(), 280

doubles程序, 148  
 draw(), 229  
 -D编译器选项, 299

## E

EBCDIC字符码, 111, 116  
 else, 29, 54  
 else, 悬挂, 55  
 else的悬挂, 55  
 enum, 151  
 enum, 29, 151, 375  
 env\_var程序, 296  
 env命令, 297  
 EOF, 110, 121, 355  
 EOF, 121  
 EOF程序, 113  
 errno.h, 376  
 exec...(), 352  
 execl(), 351 ~ 352  
 execlp(), 352  
 execlpe(), 352  
 execv(), 352  
 execve(), 352  
 execvp(), 352  
 execvpe(), 352  
 exit(), 98, 281, 346  
 example\_2程序, 41  
 example\_1程序, 40  
 exp(), 134, 332  
 extern, 29, 88, 172 ~ 173, 175  
 extern程序, 173  
 extract(), 252  
 extract-values(), 250

## F

f(), 87, 100, 108, 173, 361  
 fabs(), 143, 332  
 factorial(), 238

- factorial\_decrement(), 238
- factorial\_positive(), 238
- fail(), 247
- fail()解析, 247
- far (Borland), 29
- fclose(), 278, 281, 337
- fct1(), 91
- fct2(), 91
- feof(), 339
- ferror(), 339
- fflush(), 279
- fgetc(), 339
- fgetpos(), 287
- fgets(), 276
- \_\_FILE\_\_, 360, 375
- find-max程序, 57
- find-max解析, 57
- find-min程序, 55
- find-min解析, 56
- find-next-day(), 153 ~ 154
- find-next-day()解析, 152
- find-sum程序, 14
- flex命令, 310
- float, 29, 128, 129
- float.h, 148, 329, 330, 379
- floor(), 333
- fmod(), 333
- fopen(), 336
- for, 29, 38
- fork(), 352
- found-next-word(), 119
- fprintf(), 277, 285, 340
- fps-arr程序, 190 ~ 203
- fps-marr程序, 198
- fputc(), 339 ~ 340
- fputs(), 339
- fread(), 341
- free(), 201, 259, 342
- free程序, 259
- freopen(), 337
- frexp(), 333
- fruit struct, 246, 253
- fscanf(), 277, 285, 340
- fseek(), 280, 281, 287, 336, 339
- fsetpos(), 279, 287, 337, 340
- ftell(), 283, 285, 283, 340
- func(), 334
- func()函数, 78, 82, 97 ~ 100, 165, 376
  - 并发过程, 352
  - 布尔, 61
  - 参数, 79, 80
  - 常见定义, 335
  - 程序挂起, 353
  - 程序退出, 345
  - 传递数组, 195
  - 串处理, 218, 347
  - 串多字节, 345
  - 串转换, 343
  - 错误, 330
  - 递归, 225
  - 调用, 78
  - 调用, 78, 87, 95
  - 定义, 78, 87, 89, 179, 376
  - 定义, 87
  - 定义次序, 89
  - 定义限制, 179
  - 多字节串, 345
  - 多字节字符, 345
  - 覆盖进程, 352
  - 环境, 343
  - 结构参数, 249
    - 变量参数, 334
    - 类型, 79
    - 时间, 349 ~ 350
  - 进程间通讯, 353
  - 局部化, 331
  - 块, 349
  - 内存处理, 346

内存分配, 342  
 排序, 342  
 日期, 349  
 声明, 87  
 输入/输出t, 335  
 数学的, 332  
 搜索, 342  
 体, 79  
 跳转, 333  
 头, 79  
 伪随机数, 343  
 信号处理, 334  
 一般的实用程序, 341  
 原型(ANSI C), 99, 183  
 原型, 82, 87, 89, 100, 143  
 原型性质, 179  
 诊断, 329  
 整数算法, 343  
 字符, 329  
 fwrite(), 341

## G

g(), 107  
 gcalloc(), 204, 299  
 gcc命令, 297  
 gcd(), 241  
 getc(), 280, 339  
 get-call-from-user(), 93  
 getchar(), 112, 339  
 getenv(), 297, 343  
 get-info(), 291  
 gets(), 339  
 get-word(), 227  
 gfopen(), 282, 300  
 g-lib.h, 300  
 gmtime(), 350 ~ 351  
 Goldbach, 106  
 goto, 29, 63, 65, 333  
 graceful open, 282

grep命令, 310  
 -g编译器选项, 299

## H

haracter, 110  
 hc命令, 297  
 headtail程序, 93  
 hello\_r程序, 107  
 hello程序, 1, 3 ~ 4, 19, 294  
 hello解析, 3  
 help命令, 305  
 huge (Borland), 29  
 husband struct, 267

## I

I, 19  
 I/O控制符, 124  
 if, 29, 53  
 if-else, 53 ~ 54, 66  
 indent命令, 310  
 inlink(), 341  
 input 输入, 19  
     cin(C++), 19  
     重定向, 276  
     返回值, 274, 275  
     浮点数, 10, 112, 118  
     键盘, 287  
     流, 269  
     模式, 18  
     扫描集[...], 277, 337  
 insert(), 258  
 insert程序, 258  
 int, 29, 128, 129  
 int, 29, 128, 129  
 iostream.h(C++), 19 ~ 20  
 is\_good程序, 159  
 is\_orime(), 105  
 isalnum(), 117, 329  
 isalpha(), 117, 213, 329, 362

isascii(), 117  
 isentrl(), 117, 329  
 isdigit(), 117, 329  
 isgraph(), 117, 329  
 islower(), 114 ~ 117, 330  
 isprint(), 117, 329  
 ispunct(), 117, 329  
 isspace(), 117, 329  
 isupper(), 117, 193, 329  
 isxdigit(), 117, 329  
 -I编译器选项, 299

## J

## Java

数组分配, 210  
 布尔, 77  
 按值调用, 188  
 转换, 149  
 无用集, 268  
 输入, 127  
 数学库, 149  
 程序  
   Average, 76  
   BreakContinue, 77  
   Class TwoD, 210  
   CoinToss, 163  
   CounterTest, 268  
   Echo, 294  
   FailedSwap, 188  
   Message, 108  
   Primes, 210  
   RandomPrint, 149  
   Recur, 244  
   RecursionExercise, 108  
   SimpleInput, 46  
   SimpleInputDouble, 46  
   StringArray, 223  
   StringTset, 223  
   TestChar, 127

方法, 108, 188  
   crossOut(), 210  
   equals(), 224  
   indexOf(), 224  
   sayGoodBye(), 244  
   swap(), 189  
 跳转, 63, 65, 333

## K

Kelley, A, 231, 313  
 Kernighan, 2  
 Knuth, D, 175

## L

labs(), 343  
 large\_I程序, 21  
 large程序, 90 ~ 91, 93  
 lcas\_dir程序, 295  
 lconv struct, 332  
 ldexp(), 333  
 ldiv(), 344  
 lex命令, 310  
 limits.h, 147, 329, 331, 367, 379  
 limits程序, 148  
 \_\_LINE\_\_, 360, 375  
 linked\_list struct, 255, 267  
 list struct, 254  
 list.h, 256  
 list程序, 255  
 locale.h, 329, 331  
 localtime(), 350  
 log(), 134, 332  
 log10(), 332  
 long double, 7, 42, 128, 132, 136, 375  
 Long double, 7, 42, 128, 132, 136, 375  
 long float, 12, 375  
 Long float, 12, 375  
 long, 29, 128, 131 ~ 132, 142

long\_orl程序, 146  
 longjmp(), 333  
 lookup(), 258  
 lookup程序, 258  
 ls命令, 18  
 lucky程序, 187  
 lvalue, 183

## M

main(), 78, 89  
 majority(), 372  
 makefile, 310  
 makefile for the compare\_sorts  
 makefile, 306  
   操作行, 306  
   规则, 306, 307  
   命令, 306  
   依赖行, 306  
 makefile中的操作行, 306  
 makefile中的命令, 306  
 makefile中的依赖行, 306  
 makefile中规则, 306, 307  
 make命令, 306, 311  
 malloc(), 169, 200 ~ 201, 203, 255 ~  
   256, 257, 286, 342 ~ 343  
 man命令, 314  
 math.h, 9, 98, 135, 143, 329, 332  
 max(), 82  
 max\_ovld程序, 101  
 maximum(), 202  
 mblen(), 345  
 mbstowcs(), 345  
 mbtowc(), 345  
 memchr(), 346  
 memcmp(), 346  
 memcpy(), 346  
 memmove(), 347  
 memset(), 347  
 message2, 96

message程序, 79 ~ 80  
 message解析, 79  
 min(), 81, 82  
 min\_sec程序, 23  
 min\_sec解析, 81  
 minimum程序, 81  
 minmax(), 230  
 minmax()解析, 231  
 misspelled函数, 16  
 mktime(), 350  
 --MM, 299  
 modf(), 333  
 Monte Carlo, 88  
 moon\_sun程序, 358  
 MS\_DOS, 92  
   % 格式, 169  
   control-d, 15, 18, 114, 293 ~ 294  
   control-z, 293  
   EOF, 114  
   make实用程序, 311  
   打开文件, 284  
   二进制文件, 281  
   二进制文件打开文件模式, 286  
   拷贝命令, 122  
   库, 45  
   库管理程序, 299  
   两个字节的机器字, 143  
   内存分配, 286  
   死循环, 99  
   文件机制, 286  
   文件名限制, 42, 246  
   文件尾标志, 15, 18, 114, 293 ~ 294  
   终止程序, 70  
 mv命令, 17  
 my\_copy程序, 122  
 my\_echo程序, 218  
 -M编译器选项, 299

## N

NDEBUG, 87, 329, 361

near (borland), 29  
new-del程序, 205  
next-day程序, 152  
nice-day程序, 213  
nice-day解析, 213  
NULL常量, 165, 254

## O

Occ命令, 297  
offset程序, 335  
old\_func程序, 100  
OOP, 265, 325  
open(), 352  
-o编译器选项, 299

## P

pack(), 368  
pack-unp程序, 368  
parentheses(), 31 ~ 32  
pascal (Borland), 29  
pattern程序, 229  
perror(), 330, 338, 346  
personal struct, 251  
pipe(), 353  
play(), 94  
Pohl, I, 230, 231, 242 ~ 243, 313, 317  
pointer程序, 168, 190  
pow(), 134, 333  
pow\_of\_2程序, 36  
pow\_of\_2解析, 36  
p-print程序, 314  
pref命令, 300  
printenv命令, 297  
printf(), 4, 9, 13, 16, 38, 269, 339  
prn\_age程序, 12  
prn\_array(), 302  
prn\_art程序, 302  
prn\_banner(), 84

prn\_char程序, 7  
prn\_final\_report(), 94  
prn\_final\_status(), 155  
prn\_game\_status(), 155  
prn\_headings(), 84  
prn\_help(), 155  
prn\_info(), 92, 280  
prn\_instructions(), 95, 139, 156  
prn\_it(), 227  
prn\_it\_by\_word(), 227  
prn\_message(), 79 ~ 80  
prn\_random\_numbers(), 89, 90  
prn\_random\_numbers()解析, 89  
prn\_rand程序, 38  
prn\_rand解析, 38  
prn\_results(), 140  
prn\_time(), 304  
prn\_values(), 250  
probability(), 176  
process(), 177  
process()解析, 178  
profiler程序, 301  
p-r-s程序, 154  
put(), 280, 339  
putchar(), 112, 339  
puts(), 339 ~ 340  
p编译器选项, 299

## Q

qc命令, 297  
qsort(), 342

## R

r\_prn\_message(), 96  
r\_strcmp程序, 241  
r\_strlen(), 229  
r\_strncmp(), 230  
raise(), 334  
rand(), 38, 45, 88, 93, 154, 343

rand\_num程序, 88  
 random(), 175  
 ranlib命令, 300  
 rb文件打开模式, 279  
 read(), 352  
 read\_and\_prn\_data(), 84  
 read\_and\_prn\_data()解析, 85  
 read\_date(), 251  
 read\_date()解析, 252  
 read\_grades(), 252  
 read\_in(), 215  
 realloc(), 341 ~ 342  
 rec\_deep程序, 239  
 redirection程序, 84  
 remove(), 296, 341  
 rename(), 341  
 repeat(), 118  
 repeat程序, 118  
 report(), 158  
 reverse(), 228  
 rewind(), 280, 281, 287, 336, 339  
 Ritchie, D., 2  
 rubout键, 18  
 run\_sums程序, 84  
 r文件打开模式, 279

## S

s2 struct, 268  
 scanf(), 11, 12, 16, 272, 340  
   空白字符, 273  
   控制串, 273  
   控制串空白, 273  
   控制串普通字符, 273  
   转换说明, 273  
 scanf() and printf(), 16  
 sed命令, 310  
 selection\_by\_machine(), 156  
 selection\_by\_player(), 156  
 setbuf(), 337  
 setjmp(), 333  
 setjmp.h, 329, 333  
 setlocale(), 332  
 setvbuf(), 337  
 set命令, 297  
 short, 29, 128, 131, 142  
 sieve of Erastophenes, 209  
 signal(), 334  
 signal.h, 329, 334  
 signed char, 128, 130  
 signed char, 29, 128, 131, 142  
 signed char, 29, 128, 375  
 signed int, 128, 130  
 signed int, 129  
 signed long int, 129  
 signed long, 129  
 signed short int, 129  
 signed short int, 129  
 signed, 29, 128, 375  
 sin(), 134, 145, 332  
 single program, 121  
 sinh(), 332  
 size\_t, 200  
 sizeof program, 134, 268  
 sizeof, 29, 134 ~ 135, 256  
 sl struct, 268  
 sleep(), 353  
 slow\_sort(), 302  
 slow\_srt程序, 302  
 smalltalk, 315  
 spawn..., 353  
 spawn1(), 352  
 sprintf(), 277, 337  
 sqrt(), 98, 134, 332  
 sqrt\_pow程序, 134  
 sqrt\_pow解析, 135  
 srand(), 107, 154 ~ 155, 343  
 sscanf(), 277, 286 ~ 287, 340  
 statexttr程序, 188



stats(), 203  
stats程序, 202 ~ 203  
stdarg.h, 329, 334, 340  
\_\_STDC\_\_, 360, 375  
stddef.h, 329, 335  
stderr, 278, 337  
stdin, 278, 337  
stdio.h, 9, 38, 45, 88, 200 ~ 201, 329, 342  
stdlib.h, 278, 337  
stdout, 78  
strcat(), 219, 347  
strchr(), 347  
strcoll(), 347, 348  
strcpy(), 219, 347  
strcspn(), 347  
strerror(), 330, 347  
strftime(), 351 ~ 352  
string.h, 9, 218 ~ 219, 348  
string\_to\_list()程序, 256  
string\_to\_list, 347  
string程序, 256  
strlen(), 348  
strmp(), 219, 346, 349  
strncat(), 348  
strncmp(), 348  
strncpy(), 348  
strpbrk(), 348  
strrchr(), 348  
strspn(), 348  
strstr(), 348  
strtod(), 344 ~ 345  
strtok(), 348  
strtol(), 344  
strtoul(), 345  
struct, 129  
struct, 29  
strxfrm(), 349  
stty程序, 24

student structs, 246 ~ 247, 251  
student.h, 252  
student\_data struct, 251  
student程序, 251  
sum(), 101, 195, 202, 239, 240  
sum\_arg程序, 101  
sum\_ar程序, 200  
sum\_dbl程序, 70  
sum\_xy程序, 7  
sun机器, 143, 160, 156, 358  
swap(), 170, 181, 228, 329  
swap()解析, 170  
swap程序, 170, 181  
switch, 65, 375  
sys\_sort程序, 312  
system(), 343  
-s编译器选项, 299

## T

tan(), 134, 332  
tanh(), 332  
test\_err程序, 21  
time(), 154 ~ 155, 303, 349  
time.h, 303, 329, 349  
time\_keeper程序, 304  
time\_mlt程序, 305  
tm struct, 349  
tmpfile(), 281, 336, 346  
tmpnam(), 296, 338  
toascii(), 117  
tolower(), 117, 330 ~ 331, 378  
toss(), 94  
touch命令, 309  
toupper(), 114 ~ 117, 283, 330 ~ 331, 378  
towers\_of\_hanoi, 232  
traditional C, 375 ~ 379  
triangle程序, 125  
try\_me程序, 41, 291  
Turbo C关键字, 29

typedef

typedef, 次序, 260

## U

u\_lib.h, 304

Union, 29

universe程序, 176, 376

UNIX, 99

4个字节的机器字, 143, 155

control-d, 15, 18, 114

cp命令, 122

EOF, 114

make实用程序, 311

vi编辑器, 295

编译器选项, 297

标准头文件, 37, 354

调试器, 305

二进制文件, 299

工具, 310

库, 45

文档库存储器, 143

文件打开, 284

文件机制, 286

文件尾标记, 15, 18, 114

终止程序, 70

unpack(), 368

unpack()解析, 368

unsigned char, 128, 130

unsigned char, 29, 128, 131 ~ 132, 134

unsigned int, 128, 130

unsigned int, 129

unsigned long int, 128, 132

unsigned long int, 129

unsigned long, 128, 132

unsigned long, 129

unsigned program, 145

unsigned short int, 129

unsigned short int, 129

unsigned short, 129

unsigned short, 129

unsigned, 29

unsigned, 29, 128, 131 ~ 132, 134

UNIX 中的vi编辑器, 295

## V

va\_arg(), 334

va\_end(), 334

va\_start(), 334

va\_sum(), 335

vegetable struct, 246

veggies程序, 162

vfork(), 353

vfprint(), 340

vi命令, 17

void\*, 169, 375

void, 29, 78, 80, 82, 98, 102, 169, 376

volatile, 29, 179, 376

## W

warnsdorf规则, 243

wcstombs(), 345

wctomb(), 345

wc命令, 93, 310

wd-back程序, 227

while, 20 ~ 14, 29, 56

wife struct, 267

word\_cnt(), 217

word\_cnt程序, 119, 218

word\_cnt解析, 119

wr\_bkwd程序, 226

wr\_rand程序, 291

write(), 352

wrt\_bkwd解析, 227

-w编译器选项, 41

## Y

yacc命令, 310

中文索引项 (按拼音排序)

## A

按位二进制逻辑运算符, 364

按位取反, 363

按值调用, 95, 165, 251

按值调用, 97

## B

八进制数字转义序列, 112

八进制整数, 30, 143

八进制整型常量, 131

把参数传递给main(), 218

把数组传递给函数, 195

半数值算法, Knuth, 175

保留字, 6

贝尔实验室风格, 16, 67

比较程序, 301

边界, 192, 203, 220

边界检查, 192, 203

编辑器, 2

编译器, 1~2, 17, 26, 36, 87, 297

编译时错误, 17

编译选项

-C, 299

-D, 299

-E, 299

-g, 299

-I, 299

-lm, 134, 143

-M, 299

-MM, 299

-O, 299

-o, 299

-p, 299

-S, 299

-w, 41

变量, 5, 80, 128

标识符, 379

参数, 334

环境, 296

字符, 111

标点符号, 31

标号标识符, 379

标记, 26, 27

标记名, 151~152, 158, 379

结构, 260

标识符, 6, 29

变量, 379

辨别, 29, 42

标号, 379

结构, 246

内部的, 42

外部的, 42

系统, 30

标志符, 271, 298

标准库, 37, 218, 329~353

表

&& 和 //的真值表, 52

333.7777e-22的浮点常量部分, 132

C++文件模式参数, 287

C头文件, 329

C中的标准文件, 287

float和unsigned的组合, 132

I/O控制符, 124

int的二进制表示, 369

long 和 unsigned的组合, 131

printf()转换字符, 10, 269

scanf()例子, 276

scanf()转换, 12

scanf()转换字符, 274

stdio.h中的标准C文件, 278

非表达式的值, 51

非打印和难以打印字符, 112

赋值运算符, 35

更多的工具, 310

关键字, 29

关系、等式和逻辑运算符, 48

- 关系表达式的值, 49
- 基本类型, 146
- 基本数据类型, 129
- 基本数据类型long型, 129
- 可以调用sum()的各种方法, 196
- 每次循环后的数组a[]的元素, 197
- 美国信息交换标准码, 380
- 能用于程序的字符, 27
- 使用strftime(), 351
- 输出数字转换格式, 272
- 输出转换格式, 270
- 数组的行和列, 197
- 数组维, 197
- 特殊的scanf()转换字符, 274
- 位表达式的值, 51
- 位运算符, 363
- 一些缩写的含义, 380
- 一些有用的编译器选项, 299
- 一些字符常量及其ASCII整数值, 111
- 移位运算符表达式, 366
- 有用的工具, 310
- 预定义宏, 360
- 运算符优先级和结合性, 33, 49, 248, 381
- 字符宏, 117
- 字符宏和函数, 117
- 表达式, 5, 128, 376
  - 位, 363
  - 相等, 50
- 别名, 305
- 冰雹 (hailstones), 104
- 并发进程, 352
- 不等于运算符 !=, 48, 50
- 布尔函数, 61
- C
- 参数, 79, 83, 95
  - Argc, 218
  - Argv, 218
  - 变量, 334
  - 到main(), 218
  - 宏, 356
  - 命令行, 218
- 参数, 80
- 参数, 数组, 195
- 参数类型表, 79, 83 ~ 84
- 操作系统, 2, 17
- 操作系统命令, 295
- 查找链表, 258
- 常见的定义, 335
- 常量, 30, 375
  - 八进制整数, 131
  - 串, 31, 211 ~ 212, 220, 375
  - 浮点, 8, 132, 142 ~ 143
  - 符号, 9, 191, 354 ~ 355
  - 负整数, 30
  - 枚举, 30
  - 十进制整数, 30, 131
  - 十六进制整数, 131
  - 数字, 375
  - 整数, 131, 142
  - 字符, 31, 111, 211
- 超出数组边界, 192
- 超前, 181
- 成员, 245
  - 访问, 246
  - 结构, 245
  - 名称, 379
  - 运算符, 246
- 乘等于运算符 \*=, 35
- 乘法运算符 \*, 6, 31
- 程序
  - add\_ten, 13
  - add\_to\_n, 60
  - array, 190
  - backward, 284
  - bases, 124
  - birthday, 40

bool\_tb, 161  
bubble, 196  
call\_va, 195  
calls, 335  
caps, 115, 116  
card\_tst, 250  
cast\_day, 153  
change, 215  
char\_bk, 228  
chk\_arr, 302  
circle, 12  
class, 246  
cnt\_abc, 193  
cnt\_char, 177  
coins, 22  
cointoss, 107  
colors160  
comma, 76  
compare, 301  
compute, 138  
count, 257  
count\_dn, 225  
day, 162  
dbl\_out, 29, 113, 114  
dbl\_sp, 279  
dbl\_with\_caps, 282  
def\_arg, 101  
depth, 5  
doubles, 148  
env\_var, 296  
EOF, 113  
exmple\_, 40  
exmple\_, 41  
extern, 173  
fail, 247  
find-max, 57  
find-min, 55  
find-sum, 14  
fps\_arr, 190 ~ 203  
fps\_marr, 198  
free, 259  
headtai, 193  
hello\_r, 107  
hello1, 3 ~ 4, 19, 294  
insert, 258  
is\_good, 159  
large, 90  
large\_i, 21  
lcas\_dir, 295  
limits, 148  
list, 255  
long\_orl, 146  
lookup, 258  
lucky, 187  
max\_ovld, 101  
message, 79 ~ 80  
min\_sec, 23  
minimum, 81  
moon\_sun, 358  
my\_copy, 122  
my\_echo, 218  
new\_del, 205  
next\_day, 152  
nice\_day, 213  
offset, 335  
old\_func, 100  
p\_print, 314  
p\_r\_s, 154  
pack\_unp, 368  
pattern, 229  
pointer, 168, 190  
pow\_of, 36  
prn\_age, 12  
prn\_arf, 302  
prn\_char, 7  
prn\_rabd, 38  
profiler, 301  
r\_strcmp, 241

- rand\_num, 88
- rec\_deep, 239
- redirect, 19
- redirection, 84
- repeat, 117
- run\_sums, 84
- single, 121
- single, 125
- sizeof, 268
- slow-srt, 301
- sqrtpow, 134
- statextr, 188
- stats, 202 ~ 203
- string, 256
- student, 251
- sum, 27
- sum\_ar, 200
- sum\_arg, 101
- sum\_db, 170
- sum\_xy, 7
- swap, 170, 181
- sys\_sort, 312
- test\_err, 21
- time\_keeper, 304
- time\_mlt, 305
- try\_me, 41, 291
- universe, 99
- unsigned, 145
- veggies, 162
- wd\_back, 227
- word\_cnt, 119, 218
- wr\_bkwd, 226
- wr\_rand, 291
- 程序, 27
- 程序, 退出, 345
- 程序格式, 6
- 程序挂起函数, 353
- 程序失败, 247
- 程序中断control-c, 19, 70, 98
- 程序中断rubout键, 18
- 程序状态, 98
- 抽象数据类型, 245 ~ 260, 315
- 出界, 203, 220
- 初始化, 181
- 初始化, 8, 152, 376
  - 串, 176, 212
  - 寄存器变量, 176
  - 结构, 176, 253, 376
  - 静态变量, 176
  - 联合, 176, 376
  - 缺省, 176
  - 数组, 176, 192, 376
  - 外部变量, 176
  - 指针, 176
  - 自动变量, 176
  - 自动聚集, 376
  - 自动数组, 193
- 除等于运算符 /=, 35
- 除运算符 /, 31
- 串, 211
  - 比较, 230
  - 长度, 211
  - 常量, 31, 211 ~ 212, 375
  - 尺寸, 211
  - 初始化, 176, 212
  - 处理函数, 219, 347
  - 从键盘读, 118
  - 递归, 230
  - 多字节函数, 345
  - 继续 \, 45
  - 空, 230
  - 控制, 6
  - 文字, 31
  - 指针, 211, 214 ~ 221
  - 终止, 220
  - 转换函数, 344
- 串化, 347
- 串结束标志, 211

垂直指标 \v, 112

垂直制表, 112

词汇元素, 26 ~ 27

存储类, 165, 172 ~ 176

存储类型

auto29, 88, 172 ~ 173, 178

extern29, 88, 172 ~ 173, 175

register29, 88 ~ 89, 172, 174, 334

static29, 88, 99, 172, 173, 174 ~ 176, 178, 192

存储区定位, 260

存储映射函数, 198

错误

&&vs.&, 370

=vs.=, 68

参数, 143

赋值, 121

精度, 70

数组, 220

## D

打开并寻找文件尾模式, 287

打开但不创建模式, 287

打开文件, 285

打开文件, 336

打开文件模式, 279

打印int的位, 366

打印int位, 366

大写字母, 27

大写字母, 9, 297, 308

大于等于运算符 >=, 48 ~ 49

大于运算符 >, 48 ~ 49

代码

定时, 303

目标, 1, 37

源, 1

带有标号的语句, 63

得库, 299

等式相等, 70

等于运算符 ==, 14, 39, 48, 50, 68 ~ 69, 260

低, 48

地址

结构成员, 247

日期函数, 349

时间函数, 349 ~ 350

时钟, 303, 349

顺序文件, 282

随机文件, 283

文件, 278

文件位置指示器, 338

地址传递, 170

地址运算符 &, 11, 166, 167, 182

递归

递归, 99

调节文件, 336

调试, 182

调试器, 305

调用, 78, 95

调用程序, 334

调用函数, 78

迭代程序, 14, 237

定界符, 220

丢弃内容并打开模式, 287

丢失终止符, 220

动态内存分配, 203

动态内存分配错误, 203

逗号标点, 31

逗号运算符, 61, 378

短路求值, 52

对错误处理, 339

对结构赋值, 249

对象编码, 2

多维数组, 199 ~ 200, 377

多余分号, 69

多字节串函数, 345

## E

二进制按位记数, 129  
 二进制补码补码, 363  
 二进制模式, 284, 286  
 二进制数字, 129 ~ 130  
 二进制文件, 280, 283, 286 ~ 287, 294  
 二进制运算符, 6, 49  
 二维数组, 197

## F

反斜杠 \, 30 ~ 31, 46  
 返回, 29, 66, 80, 176, 181  
 返回值, 276  
 范围, 133  
 范围错误, 330, 332 ~ 333  
 非打印字符, 111  
 非运算符!, 48 ~ 49, 52  
 分而治之算法, 230, 239  
 分号, 31, 53, 69  
 风格, 97

++, 38

C++, 325

return, 181

贝尔实验室, 16, 67

标识符, 120, 141

常量, 141

处理字符, 120

递归, 237

结构, 260

枚举, 158

数组, 201

文件打开, 284

无限, 141

系统命令, 310

学生, 67

指针, 180, 220

注释, 40

浮点, 29, 128, 132

浮点表达式, 142

浮点常量, 8, 132, 142 ~ 143

浮点类型, 129

浮点数, 7

浮点限制, 330

符号

符号常量, 9, 191, 354 ~ 355

负常整数, 30

复合语句, 14, 53

副作用, 45, 189

fputc(), 340

getc(), 339

函数, 175, 181

宏求值, 339

全局变量使用, 174

增量运算符, 34, 39

赋值兼容, 169

赋值运算符, 6, 8, 34 ~ 35, 68 ~ 69, 260

覆盖进程, 353

## G

改变程序, 215

格式, 6, 9, 269, 272

%%, 269, 271, 274

%, 31

%c, 111, 163, 269, 274

%d, 269, 274

%e, 269, 274

%E, 274

%f, 269, 274

%G, 140, 269, 274

%g, 142, 252

%hd, 269, 274

%I, 269, 274

%n, 269, 274

%o, 269, 274

%p, 168, 183, 269, 274

%s, 212, 269, 274



- %u, 169, 269, 274
- %x, 269, 274
- %X, 274
- 格式化输入/输出, 340
- 挂起程序函数, 353
- 关闭文件, 336
- 关键字
  - asm, 29
  - cdecl, 29
  - huge, 29
  - interrupt, 29
  - near, 29
  - pascal, 29
- 关键字, 29
  - asm(Borland), 29
  - auto, 29, 151 ~ 152, 178
  - break, 29, 64, 65
  - case, 29, 65, 153, 375
  - cdecl (Borland), 29
  - char, 7, 29, 108, 128 ~ 129, 143, 211, 218
  - const, 29, 179, 327, 376
  - continue, 29, 64, 65
  - default, 29, 65
  - do, 29, 62
  - double, 7 ~ 8, 29, 128, 132
  - else, 29, 54
  - enum, 29, 151, 375
  - extern, 29, 88, 172 ~ 173, 175
  - far (Borland), 29
  - float, 7 ~ 8, 29, 128, 132
  - for29, 38
  - goto, 29, 63, 65, 333
  - huge (Borland)
  - if, 29, 53
  - if-else, 53 ~ 54, 66
  - int, 29, 128, 129
  - interrupt (Borland), 29
  - long double7, 42, 128, 132, 136, 375
  - long float, 12, 375
  - long29, 128, 131 ~ 132, 142
  - near (Borland), 29
  - pascal (Borland), 29
  - register, 29, 88 ~ 89, 172, 174, 334
  - return, 29, 66, 80, 97, 176, 181
  - short, 29, 128, 375
  - signed char, 128, 130
  - signed int, 129
  - signed long int, 129
  - signed short int, 129
  - signed, 29, 128, 375
  - sizeof, 29, 134 ~ 135, 256
  - static, 29, 88, 99, 172, 173, 174 ~ 176, 178, 192
  - struct, 29
  - switch, 29, 65 ~ 66, 375
  - typedef
  - union, 29
  - unsigned char, 128, 130
  - unsigned int, 129
  - unsigned long int, 129
  - unsigned long, 128, 132
  - unsigned short int, 129
  - unsigned short, 129
  - unsigned, 29, 128, 131 ~ 132, 134
  - void\*, 169, 375
  - void, 29, 78, 80, 82, 98, 102, 169, 376
  - volatile, 29, 179, 376
  - while, 20 ~ 14, 29, 56
- 关系, 48 ~ 49
- 关系表达式, 48 ~ 49
- 关系运算符, 48 ~ 49, 69
- 关系运算符, 69
- 惯用的数学转换, 49, 136
- 归档器, 45, 299

## H

函数, 330

函数参数, 143

函数定义, 78, 89, 179, 376

函数和宏

abort(), 345

abs(), 143, 333, 343

acos(), 332

add(), 254

asctime(), 349

asin(), 332

assert(), 87, 329, 361

assign-values(), 249

atan(), 332

atan2(), 332

atexit(), 345 ~ 346

atof(), 344

atoi(), 344

atol(), 344

average(), 201, 237

bit-print(), 366

bsearch(), 342

bubble(), 196

calloc(), 169, 200 ~ 201, 203, 286

ceil(), 333

cfopen(), 291

change(), 215

chk-arrays(), 302

clearerr(), 339

clock(), 303

close(), 352

compare(), 157

compute(), 139

compute-sum(), 95

cos(), 134

cosh(), 332

count(), 257

count-down(), 225

create-employee-data(), 370

ctime(), 349

dbl-out(), 114

delete-list(), 259

difftime(), 350

display(), 229

div(), 344

dot-product(), 254

double-space(), 280

draw(), 229

exec..., 352

exec1(), 351 ~ 352

execle(), 352

execlp(), 352

execlpe(), 352

execv(), 353

execve(), 353

execvp(), 353

execvpe(), 353

exit(), 98, 346

exp(), 134, 332

extract(), 252

extract-values(), 250

f(), 87, 100, 108, 173, 361

fabs(), 143, 332

factorial(), 238

factorial-decrement(), 238

factorial-positive(), 238

fail(), 247

fclose(), 278, 281, 337

fct2(), 91

fctl(), 91

feof()339

ferror(), 339

fflush(), 280, 336 ~ 337

fgetc(), 339

fgetpos(), 287, 338 ~ 339

fgets(), 275, 338

find-next-day(), 153 ~ 154

floor(), 333

- fmod(), 333
- fopen(), 278 ~ 280, 284, 336
- fork(), 353
- found-next-word(), 119
- fprintf(), 277, 285, 340
- fputc(), 339 ~ 340
- fputs(), 339
- fread(), 341
- free(), 201, 220
- freopen(), 337
- frexp(), 333
- fscanf(), 277
- fseek(), 280, 281, 287, 336, 339
- fsetpos(), 279, 287, 338
- ftell(), 283, 285, 340
- func(), 334
- fwrite(), 341
- g(), 107
- gcalloc(), 204, 299
- gcd(), 241
- get(), 280, 339
- get-call-from-user(), 93
- getchar(), 112, 339
- getenv(), 297, 343
- get-info(), 291
- gets(), 339
- get-word(), 227
- gfopen(), 282, 300
- gmtime(), 350 ~ 351
- insert(), 258
- isalnum(), 117, 329
- isalpha(), 117, 213, 329, 362
- isascii(), 117
- iscentrl(), 117
- isdigit(), 117
- isgraph(), 117
- islower(), 114 ~ 117, 330
- is-prime(), -105
- isprint(), 117, 329
- ispunct(), 117, 329
- isspace(), 117, 329
- isupper(), 117, 193, 329
- isxdigit(), 117, 329
- labs(), 343
- ldiv(), 344
- localtime(), 349
- log(), 134, 332
- log10(), 332
- longjmp(), 333
- lookup(), 258
- lsexp(), 333
- main(), 78
- majority(), 372
- malloc(), 169, 200 ~ 201, 255 ~ 256, 286, 342 ~ 343
- max(), 82
- maximum(), 201
- mblen(), 345
- mbstowcs(), 345
- mbtowc(), 345
- memchr(), 346
- memcmp(), 346
- memcpy(), 346
- memmove(), 346
- memset(), 346
- min(), 81, 82
- minmax(), 230
- mktime(), 350
- modf(), 333
- open(), 352
- pack(), 368
- pen-help(), 155
- perror(), 330, 338
- pipe(), 353
- play(), 93
- pow(), 134, 333
- printf(), 4, 9, 13, 16, 38, 339
- prn-array(), 302

- prn-banner(), 84
- prn-final-report(), 93
- prn-final-status(), 155
- prn-game-status(), 155
- prn-headings(), 84
- prn-info(), 92, 280
- prn-instructions(), 95, 139, 156
- prn-it(), 226
- prn-it-by-word(), 227
- prn-message(), 79 ~ 80
- prn-random-numbers(), 89, 90
- prn-results(), 140
- prn-time(), 304
- prn-values(), 250
- probability(), 176
- process(), 177
- putc(), 280, 339
- putchar(), 112, 339
- puts(), 339 ~ 340
- qsort(), 342
- raise(), 334
- rand(), 38, 45, 88, 93, 154, 343
- random(), 175
- read(), 352
- read-and-prn-data(), 84
- read-date(), 251
- read-grades(), 252
- read-in(), 215
- realloc(), 341 ~ 342
- remove(), 296, 341
- rename(), 341
- repeat(), 117
- report(), 158
- reverse(), 228
- rewind(), 279, 283, 445, 337, 338
- r-prn-message(), 96
- r-strlen(), 229
- r-strncmp(), 229
- scanf(), 11, 12 ~ 13, 14, 16, 37, 272, 340
- selection-by-machine(), 156
- selection-by-player(), 156
- setbuf(), 337
- setjmp(), 333
- setlocale(), 331
- setvbuf(), 337
- signal(), 334
- sin(), 332
- sinh(), 337
- sleep(), 353
- slow-sort(), 301
- spawn..., 353
- spawn1(), 352
- sprintf(), 277, 285
- sqrt(), 98, 333
- srand(), 107, 154 ~ 155, 343
- sscanf(), 277, 286 ~ 287, 340
- stats(), 203
- stcreat(), 219, 347
- strchr(), 347
- strcmp(), 219, 346, 349
- strcoll(), 347, 348
- strcpy(), 219, 347
- strespn(), 347
- strerror(), 219, 347
- strftime(), 351 ~ 352
- string-to-list(), 256
- strlen(), 219 ~ 220, 347
- strncat(), 347
- strncmp(), 230, 347
- strncpy(), 347
- strpbrk(), 348
- strrchr(), 348
- strspn(), 348
- strstr(), 348
- strtod(), 344 ~ 345
- strtok(), 348
- strtol(), 344

strtoul(), 345  
 strxfrm(), 349  
 sum(), 100, 195, 239, 239  
 swap(), 170, 181, 228, 329  
 system(), 295 ~ 296, 343  
 tan(), 134, 332  
 tanh(), 332  
 time(), 154 ~ 155, 303, 349  
 tmpfile(), 281, 336, 346  
 tmpnam(), 296, 338  
 toascii(), 117  
 tolower(), 117, 283, 330 ~ 331, 378  
 toss(), 93  
 toupper(), 114 ~ 117, 283, 330 ~ 331, 378  
 ungetc(), 340  
 unlink(), 341  
     unpack(), 368  
     va-arg(), 334  
     va-end(), 334  
     va-start(), 334  
     va-sum(), 335  
     vfork(), 353  
     vfprintf(), 340  
     westombs(), 345  
     wctomb(), 345  
     word-cnt(), 217  
     write(), 352  
 函数原型, 98  
 行号, 362  
 宏  
     vs.内联, 100  
     参数, 355  
     预定义, 360  
 宏  
     --DATE--, 360  
     --DATE--, 378  
     --FILE--, 360, 375  
     --LINE--, 360

    --LINE--, 378  
     NDEBUG, 87, 329, 361  
     --STDC--, 360, 375  
     --TIME--, 360, 375  
 后置, 33 ~ 34  
 后置条件, 87  
 环境变量, 296  
     BASE, 297  
     HOME, 297  
     LOGNAME, 297  
     NAME, 297  
     SHELL, 297  
 环境函数, 343  
 换行符 \n, 111  
 回车 \r, 112  
 或 (逻辑) 运算符 //, 48, 50 ~ 52  
 或 (位) 运算符 /, 363, 364  
 或等于运算符 /=, 35  
  
 J  
  
 机器精度, 70  
 基本数据类型, 128  
 寄存器, 29, 88 ~ 89, 172, 174, 334  
 寄存器变量初始化, 176  
 加 (二元) 运算符 +, 31  
 加等于运算符 +=, 35  
 加载器, 2, 143, 298, 302  
 假, 48  
 间接, 166  
 间接引用, 166 ~ 169  
 间接引用或间接运算符 \*, 166 ~ 167, 169, 180  
 减等于运算符 -=, 35  
 减量运算符, 238  
 减运算符 -, 31  
 简档器, 300  
 键盘输入, 11, 112, 118  
 结构  
     automobile, 260

- card, 245 ~ 246, 249, 253, 267
- complex, 253
- data, 267
- date, 251
- fruit, 253
- husband, 267
- lconv, 332
- linked\_list, 255, 267
- list, 254
- personal, 251
- s1, 268
- s2, 268
- student, 246 ~ 247, 251
- student\_data, 251
- tm, 349
- vegetable, 246
- wife, 267
- 结构, 245 ~ 260, 378
  - 标记名, 246, 259
  - 标识符, 246
  - 参数, 249
  - 成员, 245
  - 初始化, 176, 253, 376
  - 赋值, 249
  - 声明, 245, 246
  - 指针运算符->, 246, 248
  - 自引用, 254
- 结构化程序, 78
- 结合性, 32, 338
- 解析
  - add\_ten, 13
  - add\_to\_n, 60
  - bit\_print(), 366
  - caps, 115
  - change(), 216
  - cnt\_abc, 193
  - compare\_sorts的makefile, 307
  - compute(), 139
  - dbl\_out, 113
  - dbl\_space, 280
  - dbl\_with\_caps, 282
  - depth, 5
  - fail(), 247
  - find\_max, 57
  - find\_min, 55
  - find\_next\_day(), 152
  - find\_sum, 14
  - hello, 3
  - message, 79
  - minimum, 81
    - minmax(), 230
    - nice\_day, 213
    - pow\_of, 36
    - prn\_rand, 38
    - prn\_random\_numbers(), 88
    - process(), 177
    - read\_and\_prn\_data(), 85
    - read\_date(), 252
    - sqrt\_pow, 134
    - string\_to\_list(), 256
    - sum, 27
    - swap(), 170
    - unpack(), 368
    - word\_cnt, 119
    - wrt\_bkwd, 226
  - 可移植的caps, 116
  - 解析, 307
  - 进程间通讯函数, 353
  - 精度, 10, 132, 270
  - 警告级别, 310
  - 警告字符, 112
  - 静态, 29, 88, 99, 172, 173, 174 ~ 176, 178, 192
  - 静态变量初始化, 176
  - 静态外部变量, 175
  - 局部化, 331
  - 绝对地址, 182

## K

科学表示法, 132  
 可移植的caps解析, 116  
 可移植的代码, 123, 183  
 可执行文件, 1, 17~18, 93  
 空白, 27, 28, 156, 212, 273  
 空串, 229  
 空格字符, 4, 12  
 空语句, 53  
 空字符 \0, 212, 220~221  
 控制串, 6  
 控制流, 13  
 控制流, 13, 48, 65~72, 96  
 库  
   assert.h, 86, 209, 329, 361  
   card.h, 249  
   ctype.h, 110, 117, 213, 329~330  
   errno.h, 329, 330, 376  
   float.h, 148, 329, 330, 379  
   g\_lib.h, 300  
   iostream.h (C++), 19~20  
   limits.h, 147, 329, 331, 367, 379  
   list.h, 255, 256  
   locale.h, 329, 331  
   math.h, 9, 98, 135, 143, 329, 332  
   setjmp.h, 329, 333  
   signal.h, 329, 333  
   stdarg.h, 329, 334, 340  
   stddef.h, 329, 325  
   stdio.h, 9, 36, 277~278, 329, 336  
   stdlib.h, 9, 38, 45, 88, 200~201, 329, 342  
   string.h, 9, 218~219, 329  
   student.h, 252  
   time.h, 349  
   u\_lib.h, 304  
 库, 36  
 标准, 37, 218, 329~353  
 创建299

得体, 299

库管理程序, 299  
 块, 171, 172

## L

垃圾, 259  
 类型, 128, 172  
   浮点, 129  
   函数, 79  
   基本, 128~143  
   枚举, 151, 160, 375  
   描述符, 151  
   算法, 129  
   限定词, 179  
   用户, 152  
   整型, 129  
   指针, 195  
 类型, 375  
 类型定义, 159  
 类型转换, 136, 137  
 连接器, 298  
 连通性, 243  
 联合, 340  
 链, 255  
 链表  
   C++面向对象设计方法, 321  
   C程序员为什么要学习C++, 316  
   dbl\_with\_caps程序应该做什么, 282  
   Hesperus的算法, 5  
   String.h, 346  
   Switch的作用, 65  
   按下列步骤完成引用调用, 171  
   贝尔实验室工业编程风格, 67  
   编程过程, 1  
   编写、运行C程序的步骤, 17  
   标准库中的一些串处理函数, 219  
   分解求和程序, 83  
   基本的链表操作, 256  
   计算累积和的算法, 14

- 结构化编程指南, 78
  - 使用dbx的步骤, 305
    - 一般算术转换, 136
  - 一些观察, 380
  - 在%和转换字符间可能有, 274
  - 在转换说明中可以出现的内容, 270
  - 怎样读该表, 380
  - 纸、石头和剪子的游戏, 154
  - 转换说明中的标志字符, 271
  - 字符处理目标, 176
  - 链表分配, 255
  - 链表计数, 257
  - 链表元素插入, 258
  - 链表元素删除, 258
  - 零做除数, 7
  - 流
    - cin(C++), 19
    - cout(C++), 19
    - stderr, 336
    - stdin, 278, 337
    - stdout, 278, 337
  - 流, 336
    - 输入, 11
    - 字符, 269
  - 逻辑, 50
  - 逻辑表达式, 50
  - 逻辑运算符, 48, 51
- M**
- 冒泡程序, 196
  - 冒泡排序, 196
  - 枚举, 151 ~ 152
  - 枚举, 159
  - 枚举类型, 151, 160, 375
  - 枚举型常量, 30
  - 面向对象编程, 265, 325
  - 命令
    - a.out, 17, 297
    - alias, 305
    - ar45, 299
    - awk, 310
    - bc, 297
    - bcc, 17, 297
    - bison, 310
    - cb, 310, 314
    - cc, 17, 297
    - clear, 313
    - cls, 313
    - continue, 305
    - copy, 122
    - cp, 122
    - date, 295
    - dbx, 305
    - diff, 310
    - dir, 18, 295
    - env, 297
    - flex, 310
    - gcc, 297
    - grep, 310
    - hc, 297
    - help, 305
    - indent, 310
    - lex, 310
    - ls, 18
    - make, 306, 311
    - man, 314
    - mv, 17
    - occ, 297
    - printenv, 297
    - prof, 300
    - qc, 297
    - ranlib, 300
    - rename, 18
    - sed, 310
    - set, 297
    - stty, 24
    - touch, 309
    - vi, 17



wc, 93, 310  
yacc, 310  
命令行参数, 218  
命名空间, 158, 379  
目标, 37, 315  
目标文件, 2

## N

内部标识符, 42  
内存  
    处理函数, 346  
    存储定位, 260  
    分配, 172, 200, 341  
    寄存器, 174  
    链表的动态分配, 255  
内存分配, 172, 200, 341  
内置的数学函数, 134  
难以打印字符, 111  
排序函数, 342

## P

屏幕输出, 112

## Q

启发式, 243  
前置, 33~34  
前置条件, 87  
强制, 83, 136  
清除命令, 313  
求反(位)运算符, 363~364  
求值, 短路, 52  
区别符, 29  
取模等于运算符 `%=`, 35  
取模运算符 `%`, 7, 31  
缺省, 29, 65  
缺省初始化, 176

## R

软件工具, 295~311

csh壳实用程序310  
Make, 3, 06  
归档器, 45, 299  
简档器, 300  
库管理程序, 299  
文本编辑器, 295

## S

三元运算符, 66  
扫描集[...], 274, 275  
扫描宽度, 274  
设计  
声明  
    函数, 87  
    结构, 245  
    外部, 379  
声明, 7, 128, 376  
    Const, 29, 326, 376  
    typedef  
        volatile29, 179, 376  
省略 ..., 83  
十进制按位记数, 129  
十进制整数, 30  
十进制整数常量, 30  
十六进制整数, 30, 143  
十六进制整型常量, 131  
十六进制转义序列, 376  
时间代码, 303  
时间函数, 349  
时钟访问函数, 349  
实数, 133  
实用程序库, 341  
输出  
    cout (C++), 19  
    Stdout, 278, 337  
    重定向, 18  
    重定向, 19  
    模式, 287  
    屏幕, 113

输出运算符 <<, 19

输入/输出

格式化, 340

函数, 335

直接, 341

字符, 339

数据函数, 349

数据结构, 267

数据类型, 317

数据命令, 295

数据隐藏, 265

数学函数, 134, 332

数字, 27

数字常量, 375

数组, 190 ~ 203

Char, 211

标引, 190, 192

表示法, 220

初始化, 176, 192, 193, 376

多维的, 199 ~ 200, 377

访问, 198

数组初始化部分, 192

形参, 195

一维的, 190

元素尺寸, 195

越界, 192

越界, 203, 220

传递给函数, 195

指针, 194, 220, 376

存储, 191

下标, 192

两维, 197

数组标引, 190, 192

数组程序, 190

数组的元素尺寸, 195

数组间的关系, 194

双引号, 16, 30, 95, 112

顺序文件访问, 282

私有, 175

搜索函数, 342

素数定理, 105

算法, 1

Seminumerical Algorithms, Knuth, 175

分而治之方法, 230, 239

金星, 5

快速排序, 231

冒泡排序, 196

排序, 342

求和, 13

同余方法, 175

折半查找, 342

算术, 129

随机数函数, 343

随机文件访问, 282

## T

特征标记, 102

提升, 136

整型, 136

添加模式, 287 ~ 288

条件编译, 358

条件编译运算符?:, 66 ~ 67

通配指针void, 169, 375

头, 379

头文件, 37 ~ 38, 379

投掷硬币程序, 107

图

ASCII, 111

C++串表示, 317

C++引用声明, 183

Make的依赖树, 306

被转义的字符, 112

编译处理, 27

表元素插入, 258

表元素删除, 258 ~ 259

串, 211

存储在内存中的c, 129  
递归, 96, 225  
风格, 68  
函数, 92  
函数没有返回值, 82  
建包含在所有.c文件中的文件a.h, 90  
交换说明, 170  
结合性, 32  
链表, 255  
面向对象程序员, 316  
数组, 192  
数组存储, 191  
线性链表, 255  
已压入了ABCD的栈, 261  
银行用C语言, 141  
引用和声明, 183  
造型, 137  
指针, 169  
指针使用, 166 ~ 167  
周末, 153  
转换, 270  
    自顶向下编码, 86  
退出程序函数, 345  
退出状态, 98  
退格 \b, 112

## W

外部变量初始化, 176  
外部表示符, 42  
外部声明, 379  
玩纸牌, 245  
为list的动态存储分配, 255  
伪随机数发生器, 343  
位表达式, 363  
位串, 129  
位运算符, 363 ~ 374  
文本编辑器, 17  
文本文件, 279, 294  
文档, 29

文件, 269  
    .a, 299  
    .c, 90, 93, 308  
    .h, 90, 93, 308  
    .lib, 299  
    .o, 92, 306  
    .obj, 92  
    graceful open, 282  
    makefile, 306  
    打开, 336  
    打开模式, 279  
    调节, 336  
    二进制, 279, 281, 283, 294  
    访问, 278  
    关闭, 336  
    换名, 341  
    可执行, 1, 17 ~ 18, 93  
    描述符, 352  
    目标, 2  
    删除, 341  
    顺序访问, 282  
    随机访问, 282  
    头, 37 ~ 38  
    文本, 279, 294  
    源, 1  
    指示器重定位, 286, 338  
文件, 277  
    打开并寻找文件尾, 287  
    丢弃内容并打开, 287  
    输出, 287  
    输入, 287  
    添加, 287 ~ 288  
    文件打开  
文件存在, 285  
文件打开模式  
    "a", 279  
    "ab", 279  
    "r", 279  
    "rb", 279

“w”, 279  
 “wb”, 279  
 打开但不创建模式, 287  
 文件名, 285  
 文件模式, 285  
 文件尾标记, 110, 121, 355  
 文件尾标记, 336  
 文件尾标志, 121  
 文件尾标志, 15, 18  
 文字, 串31  
 无限循环, 69, 70~71, 238  
 无限循环, 69, 70~71, 238

## X

系统标识符, 30  
 系统名称, 30  
 下标, 192  
 下标出界, 192  
 下标出界, 203, 220  
 显式转换, 137  
 线性链表, 255  
 限制  
   浮点, 331  
   函数, 87  
   函数参数, 88  
   函数存储类, 88  
   函数定义, 179  
   函数原型, 179  
   数组, 88  
   整数限制, 331  
 相等表达式, 50  
 相等测试, 69  
 相互递归, 241  
 象形文字, 126  
 小写字母, 27  
 小于等于运算符,  $\leq$ , 48~49  
 小于运算符  $<$ , 48~49  
 信号处理, 334  
 信息转储, 22

形式参数, 79  
 悬挂, else55  
 学生风格, 67  
 寻址, 166~169

## Y

延展, 137  
 掩码, 366  
 一般的实用程序, 341  
 一维数组, 190  
 一元加运算, 376  
 一元运算符, 48  
 异或(位)运算符  $\wedge$ , 363, 364  
 异或等于运算符  $\wedge=$ , 35  
 抑扬符号 ?, 275  
 引号  
   double, 16, 30, 95, 112  
   single, 31, 110, 112  
 引用调用, 165, 170  
 隐式转换, 137  
 用C++进行面向对象编程, 317  
 用串递归, 230  
 用户定义类型, 152  
 优先级, 32, 338  
 右移等于运算符, 35  
 右移运算符, 363, 365  
 与(逻辑)运算符  $\&\&$ , 48, 50~52, 69  
 与(位)运算符  $\&$ , 363, 364  
 与等于运算符  $\&=$ , 35  
 语法, 17, 39~40  
 语法, 26  
 语法错误, 17  
 语法修饰, 355  
 语句  
   break, 29, 64, 65  
   case, 29, 65, 153, 375  
   compound, 14  
   continue, 29, 64, 65  
   default, 29, 65

do29, 62  
 else29, 54  
 for29, 38  
 goto29, 63, 65, 333  
 if29, 53  
 if-else, 53 ~ 54, 66  
 return, 29, 66, 80, 97, 176, 181  
 switch, 29, 65 ~ 66, 375  
 while, 20 ~ 14, 29, 56  
 语句, 53  
 语句, 6  
   标号, 63  
   表达式, 53  
   复合, 53  
   空, 53  
 预处理宏, 360  
 预处理器, 1, 26, 297, 354 ~ 362, 378  
   operator ##, 361, 375  
   operator #, 360, 375  
 指令, 3, 6, 8, 37, 354  
 预处理器运算符defined, 378  
 预处理器指令  
   #define, 8 ~ 9, 12, 89, 159, 354  
   #elif, 378  
   #endif, 358  
   #error, 362, 375  
   #if, 358  
   #ifdef, 358  
   #ifndef, 358  
   #include, 8, 37, 89, 354  
   #line, 362  
   #program, 362, 375  
   #undef, 358 ~ 359  
 预处理器指令, 3, 6, 8  
 预处理器指令, 36  
 域, 10  
 域, 330, 332 ~ 333  
 域, 332

域错误, 330, 332 ~ 333  
 域宽, 10, 270, 271  
 原型  
   函数82, 87, 99, 143, 179, 376  
 圆括号, 31 ~ 32  
 源代码, 2  
 源文件, 2  
 运算符  
   Sizeroof, 134 ~ 135, 255  
   不等于 !=, 48, 50  
   成员, 246  
   乘 \*, 6, 31  
   乘等于 \*=, 35  
   除 /, 31  
   除等于 /=, 35  
   大于 >, 48 ~ 49  
   大于等于 >=, 48 ~ 49  
   地址 &, 11, 166, 167, 182  
   定义的预处理器, 378  
   逗号, 61, 378  
   非!, 48 ~ 49, 52  
   赋值 =, 6, 8, 34 ~ 35, 68 ~ 69, 260  
   或 (逻辑) //, 48, 50 ~ 52  
   或 (位) /, 363, 364  
   或等于 /=, 35  
   加 (二元) +, 31  
   加 (一元) +, 376  
   加等于 +=, 35  
   间接引用或间接 \*, 166 ~ 167, 169, 180  
   减 -, 31  
   减等于 -=, 35  
   结构指针, 246, 248  
   求反 (位) ~, 363 ~ 364  
   取模 %, 7, 31  
   取模等于 %=, 35  
   输出 <<, 19, 123  
   条件表达式 ?:, 66 ~ 67  
   相等 ==, 14, 39, 48, 50, 68 ~ 69

小于 <, 48 ~ 49  
 小于等于 <=, 48 ~ 49  
 异或 (位) ^, 363, 364  
 异或等于 ^=, 35  
 右移 >>, 363, 365  
 右移等于 >>=, 35  
 与 (逻辑) &&, 48, 50 ~ 52, 69  
 与 (位) &, 363, 364  
 与等于 &=, 35  
 预处理器 ##, 361, 375  
 预处理器 #, 361, 375  
 自动减量 --, 33 ~ 34  
 自动增量 ++, 33 ~ 34, 39, 45  
 左移 <<, 363, 365  
 左移等于 <<=, 35  
 运算符, 26, 31  
   二元, 6, 49  
   赋值, 34  
   关系, 48 ~ 49, 69  
   结合性, 32, 338  
   逻辑, 48, 51  
   三元, 66  
   位, 363 ~ 374  
   一元, 48  
   优先级, 32, 338

## Z

折半查找算法, 342  
 真, 48  
 真值表, 75  
 诊断, 329  
 整数, 八进制, 30, 143  
 整数, 十六进制, 30, 143  
 整数常量, 131, 142  
 整数限制, 331  
 整数溢出, 131, 142  
 整数溢出, 131, 142  
 整数运算函数, 343  
 整型类型, 129  
 整型提升, 136  
 正确性, 87  
 直接输入/输出, 341  
 值, 128  
 值域, 330  
 指数表示法, 132  
 指针, 165, 169 ~ 171, 180  
   表达式, 379  
   初始化, 176  
   串, 211, 214 ~ 221  
   赋值, 165  
   声明, 165  
   数组, 195, 377  
   数组算法, 195  
   算法, 220  
   指向void, 169  
 指针, 181  
 指针, 379  
 制表符 \t, 112  
 中断(Borland), 29  
 中断, 18  
 中央处理单元, 1  
 重定向程序, 19  
 重命名命令, 18  
 逐步求精, 165, 172 ~ 176  
 注释对 /\*\*/, 28  
 注释未关闭, 41  
 转化单元, 17  
 转换, 49, 136  
   说明, 6, 9, 269, 272  
   算法, 49, 136  
   显式, 137  
   隐式, 137  
   字符, 11 ~ 12, 13, 269 ~ 270, 272  
   自动, 136 ~ 137, 142  
 转义序列, 111  
 转义序列, 十六进制, 376  
 转义字符 \, 30, 111  
 字符, 26, 129

- ASCII码, 110, 115, 380
- EBCDIC码, 111, 116
- 八进制数字转义序列, 112
- 变量, 110
- 标记, 271, 298
- 测试, 329
- 常量, 31, 111, 211
- 处理, 110, 176
- 从键盘读, 112
- 多字节函数, 345
- 非打印, 111
- 函数, 329
- 空白, 27
- 流, 269
- 难以打印, 111
- 输入/输出, 339
- 向屏幕写, 112
- 银行, 4, 12
- 映射, 329
- 转换, 11 ~ 12, 13, 269 ~ 270, 272
- 转义 \, 30, 111
- 字符处理, 110 ~ 123
- 字母, 27
- 自编文档性代码, 84, 157
  - 自顶向下, 78, 83, 90
- 自顶向下设计, 78, 83, 90
- 自动, 29, 88, 172 ~ 173, 178
- 自动变量初始化, 176
- 自动减量运算符 --, 33 ~ 34
- 自动结构, 260
- 自动聚集初始化, 376
- 自动数组初始化, 193
- 自动增量运算符 ++, 33 ~ 34, 39, 45
- 自动转换, 136 ~ 137, 142
- 自然数, 130
- 自引用, 260
- 自引用结构, 254
- 综合, 117
- 走纸 \f, 112
- 组合学, 60
- 最终递归, 226
- 左移等于运算符 <=>, 35
- 左移运算符 <<, 365
- 作用域, 171

# 计算机科学丛书

- 
- |                        |                                |
|------------------------|--------------------------------|
| 《编译原理及实践》              | Louden 著/冯博琴 等译/39.00元         |
| 《UNIX环境高级编程》           | Stevens 著/尤晋元 等译/55.00元        |
| 《分布式操作系统：概念与实践》        | Galli 著/尤晋元 等译/                |
| 《分布式系统设计》              | Jie Wu 著/高传善 等译/30.00元         |
| 《现代操作系统》               | Tanenbaum 著/陈向群 等译/40.00元      |
| 《UNIX操作系统设计》           | Bach 著/陈葆珏 等译/33.00元           |
| 《UNIX编程环境》             | Kernighan 等著/陈向群 等译/24.00元     |
| 《C语言解析》(原书第4版)         | Pohl 著/麻志毅 等译/                 |
| 《C程序接口与实现》             | Hanson 著/                      |
| 《C程序设计教程》(原书第2版)       | Deitel 等著/薛万鹏 等译/33.00元        |
| 《C++程序设计教程》(原书第2版)     | Deitel 等著/薛万鹏 等译/22.00元        |
| 《编码的奥秘》                | Petzold 著/陆丽娜 等译/24.00元        |
| 《编码的奥秘》(原书第2版)         | Petzold 著/陆丽娜 等译/              |
| 《C++核心：软件工程方法》         | Shtern 著/李师贤 等译/               |
| 《Java程序设计导引》           | Liang 著/王镁 等译/                 |
| 《Java语言解析》             | Pohl 著/                        |
| 《Java程序设计教程》(原书第3版)    | Deitel 著/袁兆山 等译/               |
| 《大型C++软件设计》            | Lakos 著/李师贤 等译/                |
| 《C++语言的设计和演化》          | Stroustrup 著/袁宗燕 译/48.00元      |
| 《C++编程思想》              | Eckel 著/刘宗田 等译/39.00元          |
| 《C++编程思想》(第2版)         | Eckel著/刘宗田 等译/                 |
| 《Java编程思想》             | Eckel 著/京京工作室 译/39.00元         |
| 《Java编程思想》(第2版)        | Eckel 著/京京工作室 译/               |
| 《电子商务》                 | Schneider 等著/成栋 译/28.00元       |
| 《多媒体应用程序的面向对象设计》       | Guzdial 著/                     |
| 《计算机文化》(原书第3版)         | Parsons 著/朱海滨 等译/50.00元        |
| 《图论导引》(原书第2版)          | West 著/                        |
| 《专家系统原理与编程》(原书第3版)     | Giarratano 著/印鉴 等译/49.00元      |
| 《神经网络》                 | Haykin 著/史忠植 等译/               |
| 《人机接口》                 | Raskin 著/                      |
| 《设计模式：可复用面向对象软件的基础》    | Gamma 等著/吕建 等译/35.00元          |
| 《软件工程：JAVA语言实现》(原书第4版) | Schach 著/袁兆山 等译/38.00元         |
| 《软件工程：实践者的研究方法》(原书第4版) | Pressman 著/黄柏素、梅宏 译/48.00元     |
| 《软件需求》                 | Wieggers 著/陆丽娜 等译/19.00元       |
| 《数据库系统导论》(原书第7版)       | Date 著/孟小峰 等译/66.00元           |
| 《数据仓库》(原书第2版)          | Inmon 著/王志海 等译/25.00元          |
| 《数据挖掘：概念与技术》           | Jiawei Han等著/范明 等译/39.00元      |
| 《数据库系统概念》(原书第3版)       | Silberschatz 等著/杨冬青 等译/49.00元  |
| 《数据库系统实现》              | Garcia-Molina 等著/杨冬青 等译/45.00元 |



- 《数据库设计》  
 《数据库管理系统基础》  
 《事务处理：概念与技术》  
 《数字逻辑：应用与设计》  
 《嵌入式计算机系统原理》  
 《并行程序设计》  
 《最新网络技术基础》  
 《计算机网络与因特网》(原书第2版)  
 《计算机网络》(原书第2版)  
 《光纤通信技术》  
 《高性能通信网络》(原书第2版)  
 《数据通信与网络》  
 《ISDN、B-ISDN与帧中继和ATM》(原书第4版)  
 《计算机网络实用教程》  
 《计算机网络实用教程实验手册》  
 《网络》(原书第2版)  
 《TCP/IP详解 卷1：协议》  
 《TCP/IP详解 卷2：实现》  
 《TCP/IP详解 卷3：TCP事务协议、HTTP、NNTP和UNIX域协议》  
 《Internet技术基础》(原书第2版)  
 《数据通信与网络教程》(原书第2版)  
 《密码学导引》  
 《分布计算的信息安全》  
 《计算机信息处理》(原书第7版)  
 《信息系统原理》(原书第3版)
- Stephens 等著/何玉洁 等译/35.00元  
 Pratt 等著/陆洪毅 等译/20.00元  
 Gary 等著/孟小峰 等译/  
 Yarbrough 著/朱海滨 等译/49.00元  
 Wolf 著/孙玉芳 等译/  
 Wilkinson 著/陆鑫达 等译/  
 Palmer 著/严伟 译/20.00元  
 Comer 著/徐贤良 等译/40.00元  
 Peterson 等著/叶新铭 等译/49.00元  
 Mynbaev 等著/吴时霖 等译/40.00元  
 Walrand 等著/史美林 等译/  
 Forouzan 等著/潘伦 等译, 吴时霖 校/48.00元  
 Stallings 著/程时端 等译/48.00元  
 Dean 著/陶华敏 等译/65.00元  
 Dean 著/陶华敏 等译/15.00元  
 Ramteke 著/侯春萍 等译/  
 Stevens 著/范建华 等译, 谢希仁校/45.00元  
 Wright/Stevens著/陆雪莹 等译, 谢希仁校/78.00元  
 Stevens 著/胡谷雨 等译, 谢希仁校/35.00元
- Comer 著/袁兆山 等译/18.00元  
 Shay 著/高传善 等译/40.00元  
 Garrete 著/  
 Bruce 著/  
 Mandell 等著/尤晓东 等译/38.00元  
 Stair 等著/张靖 等译/42.00元

## 国外经典教材

- 《编译原理》  
 《Linux操作系统内核实习》  
 《操作系统原理与实践》(原书第2版)  
 《现代操作系统》(原书第2版)  
 《C++程序设计语言》(原书第3版, 特别版)  
 《C程序设计语言》(原书第2版)  
 《程序设计实践》  
 《程序设计语言：概念与结构》(原书第2版)  
 《计算理论导引》  
 《离散数学及其应用》(原书第4版)  
 《组合数学》(原书第3版)
- Aho 著/李建中等译/  
 Nutt 著/陆丽娜 等译/29.00元  
 Nutt著/尤晋元 等译/  
 Tanenbaum 著/陈向群 等译/  
 Stroustrup 著/裘宗燕 译/  
 Kernighan/Ritchie 著/徐宝文 等译/28.00元  
 Kernighan/Pike 著/裘宗燕 译/20.00元  
 Sethi 著/裘宗燕 等译/  
 Sipser 著/张立昂 等译/30.00元  
 Rosen 著/袁崇义 等译/  
 Brualdi 著/冯舜玺 等译/38.00元

《人工智能》  
 《神经网络设计》  
 《软件工程：实践者的研究方法》(原书第5版)  
 《数据结构、算法与应用：C++语言描述》  
 《数据结构与算法分析：C语言描述》(原书第2版)  
 《数据库系统原理》(原书第4版)  
 《数据库原理、编程与性能》(原书第2版)  
 《并行计算机体系结构》  
 《结构化计算机组成》  
 《可扩展并行计算：技术、结构与编程》  
 《计算机图形学的算法基础》(原书第2版)  
 《数据通信与网络》(第2版)

Nilsson 著/郑扣根 等译/30.00元  
 Hagan 等著/戴葵 等译/  
 Pressman 著/梅宏 等译/  
 Sahni 著/戴葵 等译/49.00元  
 Weiss 著/冯舜玺 等译/  
 Silberschatz 等著/杨冬青 等译/  
 O'Neil 等著/周傲英 等译/55.00元  
 Culler/Singh/Gupta 著/李晓明 等译/  
 Tanenbaum 著/刘卫东 等译/46.00元  
 Hwang/Xu 著/陆鑫达 等译/49.00元  
 Rogers 著/石教英 等译/49.00元  
 Forouzan 等著/吴时霖 等译/68.00元

## 经典原版书库

《UNIX环境高级编程》(英文版)  
 《现代操作系统》(英文版·第2版)  
 《程序设计语言：概念与结构》(英文版·第2版)  
 《C程序设计语言》(英文版·第2版)  
 《C++语言的设计和演化》(英文版)  
 《程序设计实践》(英文版)  
 《C++编程思想》(英文版·第2版)  
 《Java编程思想》(英文版·第2版)  
 《离散数学及其应用》(英文版·第4版)  
 《组合数学》(英文版·第3版)  
 《人工智能》(英文版)  
 《设计模式：可复用面向对象软件的基础》(英文版)  
 《软件工程：Java语言实现》(英文版·第4版)  
 《软件工程：实践者的研究方法》(英文版·第4版)  
 《系统分析与设计》(英文版)  
 《数据结构、算法与应用——C++语言描述》(英文版)  
 《数据库系统导论》(英文版·第7版)  
 《数据库系统概念》(英文版·第3版)  
 《数据库系统实现》(英文版)  
 《高级计算机体系结构》(英文版)  
 《计算机体系结构：量化研究方法》(英文版·第2版)  
 《计算机组织和设计：硬件/软件方法》(英文版·第2版)  
 《并行计算机体系结构》(英文版·第2版)  
 《可扩展并行计算：技术、结构与编程》(英文版)  
 《结构化计算机组成》(英文版·第4版)

Stevens 著/  
 Tanenbaum 著/48.00元  
 Sethi 著/  
 Kernighan/Ritchie 著/  
 Stroustrup 著/  
 Kernighan/Pike 著/  
 Eckel 著/58.00元  
 Eckel 著/69.00元  
 Rosen 著/59.00元  
 Brualdi 著/35.00元  
 Nilsson 著/45.00元  
 Gamma/Helm/Johnson/Vlissides 著/  
 Schach 著/51.00元  
 Pressman 著/68.00元  
 Satzinger/Jackson 著/60.00元  
 Sahni 著/66.00元  
 Date 著/  
 Silberschatz/Korth/Sudarshan 著/65.00元  
 Molina/Ullman/Widom 著/  
 Hwang 著/59.00元  
 Patterson/Hennessy 著/88.00元  
 Hennessy/Patterson 著/80.00元  
 Culler/Singh/Gupta 著/88.00元  
 Hwang/Xu 著/69.00元  
 Tanenbaum 著/38.00元

《计算机图形学的算法基础》(英文版·第2版)  
《通信网络基础》(英文版·第2版)  
《计算机网络》(英文版·第2版)  
《高性能通信网络》(英文版·第2版)  
《网络互连:网桥·路由器·交换机和互连协议》  
(英文版·第2版)  
《数据通信与网络》(英文版)  
《ISDN、B-ISDN与帧中继和ATM》(英文版·第4版)  
《TCP/IP详解 卷1:协议》(英文版)  
《TCP/IP详解 卷2:实现》(英文版)  
《TCP/IP详解 卷3:TCP事务协议、HTTP、  
NNTP和UNIX域协议》(英文版)  
《Internet技术基础》(英文版·第3版)

Rogers 等著/  
Walrand 著/32.00元  
Peterson/Davie 著/65.00元  
Walrand/Varaiya 著/64.00元  
Perlman 著/  
  
Forouzan 等著/59.00元  
Stallings 著/  
Stevens 著/  
Wright/Stevens 著/  
Stevens 著/  
  
Comer 著/

# 教学支持说明

本书系我社获全球最大的教育出版集团——美国Pearson Education Group独家授权之英文影印/简体中文版。

Pearson Education旗下的国际知名教育图书出版公司Addison Wesley, 以其高品质的计算机类出版物而享誉全球教育界、工商界、技术界, 成为全美及全球高校采用率最高的教材。为秉承Addison Wesley出版公司对于教材类产品的一贯教学支持, 我社特获独家授权英文影印/简体中文版的《教师指导手册》, 向采纳本书作为教材的教师免费提供。

获取相关《教师指导手册》的教师烦请填写如下情况调查表, 以确保此教学辅导材料仅为教师获得。

情况调查表如下所示:

---

## 证 明

兹证明\_\_\_\_\_大学\_\_\_\_\_系/院\_\_\_\_\_学年(学期)开设的\_\_\_\_\_课程, 采用机械工业出版社出版的英文影印/简体中文版\_\_\_\_\_ (作者/书名) 作为主要教材。任课教师为\_\_\_\_\_, 学生\_\_\_\_\_个班共\_\_\_\_\_人。

任课教师需要与本书配套的教师指导手册。

电话: \_\_\_\_\_

E-mail: \_\_\_\_\_

传真: \_\_\_\_\_

联系地址: \_\_\_\_\_

邮编: \_\_\_\_\_

系/院主任: \_\_\_\_\_ (签字)

(系 院办公室章)

年\_\_\_\_月\_\_\_\_日

---

同时本书还配有其他教学辅导资料，相关事宜敬请访问Pearson Education, Prentice Hall, Addison Wesley的相关网站：<http://www.pearsoned.com>, <http://www.prenhall.com>, <http://www.aw.com>.



China Machine Press

机械工业出版社

Tel: 8610-68995261

8610-68326677-2803

Fax: 8610-68311602

E-mail: [hzedu@hzbook.com](mailto:hzedu@hzbook.com)



Pearson Education Beijing Office

培生教育出版集团北京办事处

Tel: 8610- 6891 7488 /6891 6659

Fax: 8610-68917499

E-mail: [service@pearsoned.com.cn](mailto:service@pearsoned.com.cn)

[General Information]

书名=C语言解析教程

作者=

页数=416

SS号=0

出版日期=