

C 语言编程

—— 一本全面的 C 语言入门教程（第三版） ——

Programming in C——A complete introduction to the
C programming language, Third Edition

[美] Stephen G. Kochan 著
张小潘 译

电子工业出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书是极负盛名的 C 语言入门经典教材,其第一版发行至今已有 20 年的历史。本书内容详实全面,由浅入深,示例丰富,并在每个章节后面附有部分习题,非常适合读者自学使用。除此之外,《C 语言编程》一书对于 C 语言标准的最新进展、C 语言常见开发工具以及管理 C 语言大型项目等重要方面,也进行了深入浅出的说明。

Authorized Translation from the English language edition, entitled PROGRAMMING IN C, 3rd Edition, ISBN: 0672326663 by Kochan, Stephen published by Pearson Education, Inc, publishing as SAMS, Copyright © 2005 by Sams Publishing

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PUBLISHING HOUSE OF ELECTRONICS INDUSTRY, Copyright © 2006

本书中文简体版专有出版权由 Pearson Education 授予电子工业出版社,未经许可,不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号:图字:01-2004-2710

图书在版编目(CIP)数据

C 语言编程:一本全面的 C 语言入门教程:第三版 / (美)科汉(Kochan,S.G)著;张小潘译. —北京:电子工业出版社,2006.3

书名原文:Programming in C: A Complete introduction to the C programming language, Third Edition
ISBN 7-121-00735-5

I.C... II.①科...②张... III.C 语言—程序设计—教材 IV.TP312

中国版本图书馆 CIP 数据核字(2006)第 012628 号

责任编辑:周 筠 刘铁锋

印刷:北京智力达印刷有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

经 销:各地新华书店

开 本:787×980 1/16 印张:35.75 字数:700 千字

印 次:2006 年 3 月第 1 次印刷

定 价:59.00 元

凡购买电子工业出版社的图书,如有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系。联系电话:(010) 68279077。质量投诉请发邮件至 zlts@phei.com.cn,盗版侵权举报请发邮件至 dbqq@phei.com.cn。

译者序

在计算机编程语言的发展历史上，C语言无疑占据着极其重要的位置。在各种现代编程语言中，我们都能或多或少地看到C语言的影子。即使在面向对象编程语言大行其道的今天，C语言仍然在系统编程、嵌入式编程等领域发挥着巨大的作用。

随着IT技术日新月异的发展，无可否认的是，C语言已经不再占据编程语言舞台的中心位置了。然而，全世界几乎所有的计算机学科仍然把C语言当作最基础的科目之一。C语言本身既具有高级语言的强大功能，语言本身的很多概念又与实际的计算机有着紧密的对应关系，因此该语言通常被人称为中级语言。学习和掌握C语言，既可以增进对于计算机底层的工作机制的了解，又为进一步学习其他高级语言打下了坚实的基础。

笔者在IT领域内工作多年，虽然现在在实际工作中已经很少使用C语言了，然而当年深入钻研C语言而领悟的道理，直到今天仍然使我获益匪浅。通过此次翻译本书，笔者又有机会重温了C语言，虽然使用过众多的编程语言，但C语言的优雅、简洁、灵活以及内在的一致性仍然使笔者赞叹不已。笔者坚信，学习C语言、掌握C语言仍然是每一个在IT领域工作的技术人员最重要的基本功之一。

作为C语言教材，本书堪称经典。自从20年前第一版出版以来，本书就一直被公认为C语言最出色的入门书籍之一。读者现在看到的是原作者根据最新的C语言标准——ANSI C99，对于原有版本进行修订后推出的第三版。本书涵盖了标准C语言的各个方面，其最大特色是深入浅出，文字浅显易懂，非常适合于初学计算机编程的读者使用。几乎所有重要的C语言概念，书中都配有完整的示例程序而非用代码片断进行说明，大大方便了读者亲手实践。特别值得一提的是，本书每个章节之后都列出了适量的习题，对于复习和巩固该章节的知识大有裨益。英语中有一句谚语：Practice makes perfect，中文也有类似的谚语：熟能生巧，相信读者能够体会到这些练习的作用。顺便说一下，这些题目的答案，可以在本书的网站www.kochan-wood.com上找到。

在翻译本书的过程中，笔者得到了博文视点陈元玉、张菲、刘铁锋等各位编辑的大力协助，没有他们的辛勤工作，本书的完成是不可想象的。在这里我还要感谢博文视点所有参与了本书出版工作的人们，虽然我和他们中间很多人都没有直接接触过。最后，我还要感谢我的家人，她们在我日以继夜的工作中，给予了我无微不至的关爱。我希望，在本书完成之后，我能有更多的时间陪伴她们。

张小潘

2006年02月



To my mother and father



Preface

序言

很难相信，从我写这本书的第一版那时算起，20年已经过去了。那个时候，除了这本书之外，Kernighan和Ritchie先生的《C语言》是市场上另外唯一一本与C语言相关的书籍。看看时间改变了多少东西！

当上个世纪80年代初期C语言的ANSI标准开始出现时，本书曾一度被分为两个主题：原来的内容仍然被称作《C语言编程》，而与ANSI C标准相关的部分则被称为《ANSI C语言编程》。在那个时候，主要的C语言编译器厂商花费了数年功夫，才发布了符合ANSI标准的编译器并使得它们逐渐流行开来。而本人觉得在同一本入门教材中既包含ANSI标准的内容，又包括不符合标准的内容，对于初学者来说有点太混乱了，因此作出了上述分拆主题的决定。

自从ANSI C标准在1989年发布以来，标准本身又经过了数次修订。最新的标准版本被称为C99，这个标准的发布是本书写作的主要动机。书中专门强调了新标准对C语言带来的变化。

除了介绍C99标准新增的一些特性之外，本书还增加了新的两个章节。其中之一介绍了如何调试C语言程序。另外一章对于目前流行的面向对象编程进行了简单的介绍，因为目前最流行的几种面向对象编程语言C++、C#、Java和Objective-C都是从C语言发展来的。

在此，对于这些年来一直拥有本书的读者，我谨致以诚挚的感谢。来自你们的反馈让我深受感动，并鼓励我直到今天仍然坚持写作。

对于新读者，我欢迎来自您的评论，并希望本书能够满足您的期望。

Stephen Kochan
June 2004
steve@kochan-wood.com

About the Author

关于作者

Stephen Kochan先生有20多年使用C语言开发软件的经验。他是多部C语言畅销书的作者或者合著者，如《C语言编程》、《ANSI C语言编程》、《C语言话题》(Topics in C Programming)。Kochan先生还编写了多本Unix操作系统方面的技术书籍，如《探索Unix操作系统》、《Unix Shell编程》和《Unix操作系统安全》。Kochan先生的最新力作《Objective-C语言编程》是一本关于Objective-C语言的入门教程。

Acknowledgements



在此，我谨对那些帮助过本书各个版本出版工作的人们表示感谢。他们有：Douglas McCormick、Jim Scharf、Henry Tabickman、Dick Fritz、Steve Levy、Tony Ianinno和Ken Brown。我还要感谢纽约大学的Henry Mullish先生，是他教给了我写作方面的众多知识，并帮助我开始写作生涯。

我还要感谢Sam出版公司的Mark Renfrow和Dan Knott两位编辑先生；还有文字编辑Karen Annett和技术编辑Bradley Jones。最后，我还要对Sam出版公司所有参与本书出版的工作人员表示感谢，即使他们中间很多人并没有和我直接在一起工作。

Contents At a Glance

目录一览

序言	xvii
第1章 入门	1
第2章 基础知识	5
第3章 编译并运行第一个程序	11
第4章 变量、数据类型和算术表达式	21
第5章 循环	43
第6章 进行判断	65
第7章 使用数组	95
第8章 使用函数	119
第9章 使用结构	165
第10章 字符串	195
第11章 指针	235
第12章 位运算	279
第13章 预处理器	299
第14章 进一步讨论数据类型	321
第15章 处理大型程序	333
第16章 C语言的输入输出	347
第17章 杂项和高级特性	373
第18章 调试程序	389
第19章 面向对象编程	411
附录A C语言小结	425
附录B C语言标准库	467
附录C 使用gcc编译程序	493
附录D 常见编程错误	497
附录E C语言的其他资源	501
索引	505

Table of Contents

目录

第 1 章 入门.....	1
第 2 章 基础知识.....	5
程序设计.....	5
高级编程语言.....	6
操作系统.....	6
编译程序.....	7
集成开发环境.....	10
解释型语言.....	10
第 3 章 编译并运行第一个程序.....	11
编译我们的第一个程序.....	11
运行我们的第一个程序.....	12
理解我们的第一个程序.....	13
显示变量的值.....	15
注释.....	17
练习.....	19
第 4 章 变量、数据类型和算术表达式.....	21
使用变量.....	21
数据类型与常量.....	23
基本整数类型 int.....	23
存储空间与范围.....	24
浮点数类型 float.....	24
扩展精度的浮点类型 double.....	25
字符类型 char.....	25
布尔类型 _Bool.....	26
类型修饰符: long, long long, short, unsigned 和 signed.....	28
使用算数表达式.....	30

整数算术和单目减法操作符	33
余数操作符	35
整型数与浮点数之间的转换	36
结合运算和赋值：运算赋值操作符	38
类型 <code>_Complex</code> 和 <code>_Imaginary</code>	39
练习	40
第 5 章 循环	43
for 语句	44
关系操作符	46
对齐输出	50
程序输入	51
嵌套的 for 循环	53
for 循环的变体	54
while 语句	56
do 语句	60
break 语句	62
continue 语句	62
练习	63
第 6 章 进行判断	65
if 语句	65
if - else 结构	69
复合关系表达式	72
嵌套的 if 语句	75
else if 结构	76
switch 语句	84
布尔变量	87
条件操作符	91
练习	93
第 7 章 使用数组	95
定义数组	96
使用数组计数	100
产生 Fibonacci 数	103
使用数组产生质数	104
数组初始化	106
字符数组	108
使用数组完成基数转换	109
const 修饰符	111
多维数组	113

变量长度的数组	115
练习	117
第 8 章 使用函数	119
定义函数	119
参数与局部变量	122
函数原型声明	124
自动局部变量	124
函数的返回值	126
函数调用	131
声明返回值类型以及参数类型	134
检查函数的参数	135
自顶向下的程序设计	137
函数与数组	137
赋值表达式	142
数组排序	143
多维数组	146
函数与可变长度的多维数组	150
全局变量	152
自动变量和静态变量	156
递归函数	159
练习	162
第 9 章 使用结构	165
用于存储日期的结构	166
在表达式中使用结构	168
函数与结构	171
用于存储时间的结构	177
结构的初始化	180
复合字面量	181
结构数组	182
包含结构的结构	185
包含数组的结构	187
结构的变形形式	190
练习	191
第 10 章 字符串	195
字符数组	196
可变长度的字符串	198
字符串的初始化和显示	201
检验字符串相等	204

输入字符串	206
单字符输入	208
空字符串	213
转义字符	216
关于字符串常量的进一步讨论	218
字符串、结构与数组	219
一个更好的搜索算法	222
字符运算	227
练习	230
第 11 章 指针	235
定义指针变量	235
在表达式中运用指针	239
使用指针和结构	240
包含指针的结构	243
链表	244
关键字 const 和指针	253
指针和函数	254
指针和数组	259
稍微离题一下——关于程序的优化	263
数组还是指针	264
指向字符串的指针	266
字符串常量和指针	267
再次谈谈递增和递减运算符	268
指针运算	272
指向函数的指针	273
指针和内存地址	274
练习	276
第 12 章 位运算	279
位运算符	280
按位与运算符 (&)	281
按位或运算符 ()	283
按位异或运算符 (^)	284
取反运算符 (~)	285
左移运算符 (<<)	287
右移运算符 (>>)	287
移位函数	288
旋转移位	290
位域	292
练习	297

第 13 章 预处理器	299
#define 语句	299
程序的可扩展性	303
程序的可移植性	305
预定义符号的高级形式	306
#操作符	312
##操作符	313
#include 语句	313
系统头文件	316
条件编译	316
#ifdef、#endif、#else 和#ifndef 语句	316
#if 和#elif 语句	318
#undef 语句	319
练习	320
第 14 章 进一步讨论数据类型	321
枚举类型	321
typedef 语句	325
数据类型转换	327
符号扩展	329
参数转换	329
练习	330
第 15 章 处理大型程序	333
将程序分为多个文件	333
在命令行上编译多个源文件	334
模块之间的通信	336
外部变量	336
静态变量与外部变量/函数	339
有效的使用头文件	341
用于处理大型程序的其他工具	342
make	343
cvs	344
Unix 的其他工具: ar、grep、sed 等等	345
第 16 章 C 语言的输入输出	347
字符 I/O: getchar 函数和 putchar 函数	348
格式化 I/O: printf 函数和 scanf 函数	348
printf 函数	348
scanf 函数	355

文件输入输出操作	359
将 I/O 操作重定向到文件中	359
文件结束标志	361
用于读写文件的特殊函数	363
fopen 函数	363
getc 和 putc 函数	365
fclose 函数	365
函数 feof	367
fprintf 函数和 fscanf 函数	368
fgets 函数和 fputs 函数	368
标准输入 stdin、标准输出 stdout 和标准错误 stderr	369
exit 函数	370
重命名和删除文件	371
练习	371
第 17 章 杂项和高级特性	373
杂项语句	373
goto 语句	373
空语句	374
使用联合	375
逗号操作符	378
类型修饰符	378
register 修饰符	378
volatile 修饰符	379
restrict 修饰符	379
命令行参数	380
动态内存分配	383
malloc 和 calloc 函数	384
sizeof 操作符	385
free 函数	387
第 18 章 调试程序	389
使用预处理器嵌入调试语句	389
使用 gdb 调试程序	395
查看和设置变量	398
显示源文件	399
控制程序的执行	400
查看调用堆栈	405
调用函数和给数组、结构变量赋值	405
获取 gdb 的命令帮助	406
其他零碎的东西	408

第 19 章 面向对象编程	411
什么是对象	411
实例和方法	412
编写处理分数的 C 语言程序	413
使用 Objective-C 定义用于处理分数的类	414
使用 C++ 编写分数类	419
使用 C# 语言处理分数	422
附录 A C 语言小结	425
1.0 字元和标识符	425
1.1 字元	425
1.2 标识符	425
2.0 注释	426
3.0 常量	427
3.1 整数常量	427
3.2 浮点数常量	427
3.3 字符常量	428
3.4 字符串常量	429
3.5 枚举常量	430
4.0 数据类型与声明	430
4.1 声明	430
4.2 基本数据类型	430
4.3 导出数据类型	432
4.4 枚举数据类型	438
4.5 typedef 语句	438
4.6 类型修饰符 const、volatile 和 restrict	439
5.0 表达式	439
5.1 C 语言的操作符总结	440
5.2 常量表达式	442
5.3 算术操作符	443
5.4 逻辑操作符	444
5.5 关系操作符	444
5.6 位操作符	445
5.7 自增和自减操作符	445
5.8 赋值操作符	446
5.9 条件操作符	446
5.10 类型转换操作符	446
5.11 sizeof 操作符	447
5.12 逗号操作符	447
5.13 数组的基本操作	447
5.14 结构的基本操作	448

5.15 指针的基本操作	448
5.16 复合字面量	450
5.17 基本数据类型的转换规则	451
6.0 存储类型与作用域	452
6.1 函数	452
6.2 变量	452
7.0 函数	454
7.1 函数定义	454
7.2 函数调用	455
7.3 函数指针	456
8.0 语句	456
8.1 复合语句	456
8.2 break 语句	456
8.3 continue 语句	457
8.4 do 语句	457
8.5 for 语句	457
8.6 goto 语句	458
8.7 if 语句	458
8.8 空语句	458
8.9 return 语句	459
8.10 switch 语句	459
8.11 while 语句	460
9.0 预处理器	460
9.1 三元组	460
9.2 预处理器指令	461
9.3 预定义符号	466
附录 B C 语言标准库	467
标准头文件	467
<stddef.h>	467
<limits.h>	468
<stdbool.h>	469
<float.h>	469
<stdint.h>	469
字符串函数	470
内存函数	472
字符函数	473
输入输出函数	473
内存中的格式转换函数	478
字符串到数字的转换	479
动态内存分配函数	481
数学函数	482

复数算术	488
通用函数	490
附录 C 使用 gcc 编译程序	493
命令的一般格式	493
命令行选项	494
附录 D 常见编程错误	497
附录 E C 语言的其他资源	501
练习题答案和勘误表	501
C 语言	501
书籍	501
网站	502
新闻组	502
C 语言编译器和集成开发环境	502
gcc	502
MinGW	502
CygWin	502
Visual Studio	503
Code Warrior	503
Kylix	503
杂项	503
面向对象编程	503
C++ 编程语言	503
C# 编程语言	503
Objective-C 编程语言	504
开发工具	504
索引	505

Introduction

入门

上个世纪70年代初期Dennis Ritchie先生在贝尔实验室发明了C语言，但直到70年代后期，C语言才得以慢慢流传开来。因为那个年代，在贝尔实验室之外不容易获得可用于商业目的的C语言编译器。C语言早期的传播部分地归功于Unix操作系统，该操作系统在当时流传的速度丝毫不亚于C语言。同C语言一样，Unix操作系统也发源于贝尔实验室。这个操作系统90%以上的代码用C语言完成，从而也奠定了C语言作为该操作系统的“标准”语言的地位。

IBM PC及其兼容机的巨大成功，使得它们的操作系统MS-DOS很快成为C语言最流行的运行环境。随着C语言在多种操作系统上的普及，越来越多的开发商抓住这个机遇，推出了自己的C语言编译器。绝大多数编译器遵循的语言标准出自最早的C语言专著《C语言》的一个附录，该书由Brian Kernighan与Dennis Ritchie合著。遗憾的是，这个附录并没有完整、严格地定义什么是C语言，因此在编译器的开发过程中，C语言的很多实现细节需要厂商自行决定。

在上世纪80年代初期，有了对C语言定义标准化的需求。美国国家标准化协会（The American National Standards Institute, ANSI）开始着手这方面的工作，并在1983年成立了ANSI C标准化委员会（也称作X3J11）。其工作成果于1989年审批通过，并于1990年作为第一部C语言的官方标准（ANSI C）正式发布。

由于C语言在全世界的广泛应用，国际标准化组织（The International Standard Organization, ISO）也很快参与到标准化工作中来。该组织决定采用ANSI的工作成果，并将其编号为ISO/IEC 9899:1990。从那以后，C语言还陆续进行了一些补充修正。C语言的最新标准于1999年发布，该标准被称作ANSI C99或者ISO/IEC 9899:1999，本书讲的正是基于该标准的C语言。

C是一种高级语言，然而它提供了很多途径，使程序员能够在接近底层硬件的水平上

对计算机编程。这是因为，C语言虽然被设计为一种通用的结构化语言，但在最初构想的时候，设计者也打算使用它进行系统级别的编程。正因为如此，C语言具有强大的功能，又不失灵活性。

本书的目的在于教会读者如何使用C语言编程。本书不要求读者具有其他计算机语言的背景知识，初学者和有经验的程序员都可以使用本书。如果读者以前使用过其他语言，那么多半会发现，C语言在很多地方都相当独特，与你使用过的语言很可能有所不同。

本书涵盖了标准C语言的每个特性。在介绍这些特性的同时，书中还会附上诠释该特性的一小段完整的程序——用例子帮助读者学习是本书写作过程中遵循的一条原则。正像一幅图可以顶替一千句话，合适的例子也是这样。如果读者有自己的计算机，并且可以运行C语言的话，我强烈建议你下载运行书中的每个程序，并和书中的结果进行比较。通过这个过程，读者不但可以学习语言本身，也可以逐步熟悉编辑、编译和运行C语言的整个过程。

本书非常强调程序的可读性，因为笔者坚信程序应该以易于阅读的方式编写，无论是供作者还是其他人阅读。根据我的经验和常识判断，易读的程序也容易编写、调试和修改。除此之外，程序的易读性也是坚持结构化程序风格的一个自然结果。

本书定位于C语言的入门教程，每一章节中涉及到的知识都不会超出前面章节的范围，因此按照顺序阅读本书将为读者带来最大的收益。因此，我强烈建议读者不要跳过前面的章节而去阅读后面的内容。为了更好的巩固学习成果，读者还应该首先完成每章后面的习题，再开始学习下一章。

第2章“基础知识”，涵盖高级程序设计语言的基础知识和编译运行程序的过程，这些知识为读者的后继学习做好铺垫；从第3章“编译并运行第一个程序”开始到第16章，我们将逐步研讨C语言的各个基本环节。第16章“C语言的输入输出”将深入介绍C语言的I/O操作；第17章“杂项和高级特性”则介绍了该语言更深奥复杂的一些特性。

第18章“调试程序”，介绍如何使用C语言的预处理器来帮助我们调试程序。该章还讲述了交互式调试技术，使用流行的调试器gdb作为交互式调试工具。

从90年代开始，程序开发领域兴起了新的“面向对象编程”方法，或者简称为OOP。C语言本身并不是一种面向对象的编程语言，然而有几种脱胎于C语言的程序设计语言确实支持OOP编程。本书第19章“面向对象的程序设计”，简单说明了OOP及其相关术语，并且对三种起源于C语言的OOP语言——C++、C#以及Objective-C有一个简短的了解。

附录A “C语言小结”，完整列出了C语言语法，本附录可以当作参考手册使用。

附录B “C语言标准库”，列出了许多C语言的标准库函数，所有支持C语言的平台都会提供这些函数。

附录C “使用gcc编译程序”，列举了GNU的C语言编译器gcc的一些常用选项。

附录D “常见编程错误”，列出了使用C语言编程中常见的一些错误。

最后，附录E “C语言的其他资源”列出了很多C语言的相关资源，读者可以从这些资源获取C语言更多的资料，以供深入学习。

每一章后面习题的答案可以在网站www.kochan-wood.com中找到。

本书中的C语言实现不依赖于特定计算机系统或操作系统。书中简单介绍了如何使用流行的GNU C编译器gcc编译和运行C语言的程序。

在这里我要感谢那些帮助我撰写此书的人，他们是：Douglas McCormick、Jim Scharf、Henry Tabickman、Dick Fritz、Steve Levy、Tony Ianinno，还有Ken Brown。我还要感谢纽约大学的Henry Mullish，是他教会了我如何进行写作，带我入门出版业务。

本书的早期版本还要归功于Maureen Connelly，他是本书第一版的出版商Hayden Book公司的前任责任编辑。

Some Fundamentals

基础知识

本章介绍使用C语言编程必须具备的一些基础知识。我们先从总体上介绍所有高级编程语言的特点，然后讨论将高级语言编写的程序编译为可执行程序的一般步骤。

程序设计

从本质上讲，计算机实际上是一种相当死板的装置，它仅仅能够按照人们提供给它的指令运行。绝大多数计算机系统所能执行的指令都相当原始，比如给某个数字增加1，或者检验某个数字是否等于0等等。复合指令比起我们这里给出的例子，也复杂不了多少。计算机系统能够执行的基本指令的集合，通常称为该计算机的**指令集**。

为了使用计算机解决某个问题，我们必须使用计算机的基本指令描述这个问题的解决方法。所谓**计算机程序**，实际上也就是解决某个具体问题的计算机指令集合。按照计算机科学的术语，解决某个具体问题的方法被称为**算法** (algorithm)。举例来说，如果我们想要判断一个数字是奇数还是偶数，那么解决这个问题所需要的指令集合就是一个计算机程序，检验某个数是奇数还是偶数的方法就是算法。一般来讲，如果我们需要使用计算机解决某个问题，首先需要找到解决该问题的算法，再用一段计算机程序来表达这个算法。接着上面的例子，如果我们想要判断某个数字是奇数还是偶数，我们可以采用下面的算法：用这个数字除以2，如果余数是0，那么这个数字是偶数，否则这个数字是奇数。有了算法以后，我们就可以在特定的计算机上，采用该计算机所能接受的指令来实现这个算法。这些指令可以是某种特定的计算机编程语言，如Visual Basic, Java, C++或者C。

高级编程语言

最早的计算机只能理解二进制形式的指令，这些指令通常直接操作内存的某个地址。这种形式的指令被称为机器语言。第二代计算机编程语言——汇编语言（*Assembly Language*）允许程序员使用稍微高级一些的指令形式。汇编语言可以使用某些字符形式的符号来表示指令和数据。因为计算机只能够理解二进制形式的指令，所以人们使用一个特殊的程序——汇编器（*Assembler*），把汇编语言的程序翻译为具体的机器语言。

汇编语言使用的符号与计算机的二进制指令实际上是一一对应的。正因为如此，汇编语言被看作是低级语言。为了使用低级语言编写程序，程序员们必须学习某个特定的计算机系统的指令集。由于不同的计算机系统，其指令集常常是不同的，因此使用低级语言编写的程序不具备可移植性。也就是说，在一种计算机上能运行的程序，如果不进行修改的话，就不能在另外一种计算机上运行。

为了克服低级语言的缺点，人们发明了高级语言。FORTRAN（FORmula TRANslation）是第一种高级语言。使用Fortran语言，程序员不再需要了解具体计算机系统的结构，Fortran语言提供了更为高级的操作指令。一条Fortran语言的指令，或者称为表达式（*statement*），都对应着很多条机器指令。在这点上，Fortran语言与汇编语言很不相同，后者每条指令只能对应一条机器指令。

高级语言通常有一套特定的语法，这个语法与具体的计算机系统无关。因此用高级语言书写的程序，可以独立于具体的计算机系统。也就是说，使用高级语言书写的程序，几乎无需作任何修改，就可以运行在支持该语言的计算机上。

为了支持某种高级语言，人们需要针对特定计算机系统开发一个特殊的程序，该程序将高级语言编写的程序翻译为特定计算机系统能够理解的机器指令。这种计算机程序被称为编译器（*compiler*）。

操作系统

在继续介绍编译器之前，我们有必要先介绍一种特殊的计算机程序——操作系统（*Operating System*）。

操作系统就是控制计算机系统所有操作的程序。例如，计算机系统所有的输入输出操作（也称为I/O操作）都要通过操作系统来完成。操作系统必须管理计算机的所有资源，还要负责运行所有的程序。

Unix操作系统最早由贝尔实验室开发，它是当今最流行的一种操作系统。Unix是一种相当独特的操作系统，它拥有很多变种，能运行在多种计算机系统中，如Linux或者Mac OS

X。在Unix之前，一种操作系统通常只能运行在一种计算机系统中。但是Unix的开发主要使用了高级语言C，而且在开发这个操作系统时，它的设计者使它尽量独立于底层硬件环境，因此Unix就可以相对轻松地移植到多种计算机系统中。

微软的Windows XP也是一种流行的操作系统，它主要运行在Pentium（或Pentium系列）处理器上。

编译程序

编译器也是一个程序。从本质上来讲，它和本书中的其他程序是一样的，当然，编译器要比它们复杂得多。编译器可以分析使用高级语言编写的程序，然后把它翻译成特定的计算机系统能够执行的指令。

图2.1显示了在Unix操作系统中，输入、编译和执行一个C语言程序所需要的典型步骤，以及在Unix命令行下需要输入的典型命令。

要编译一个使用高级语言编写的程序，我们首先要把这个程序输入到计算机文件中。虽然在不同的计算机系统中，命名文件的习惯有所不同，但从总体上来说，文件名的选取还是由使用者决定。一般来讲，包含C语言程序的文件以“.c”两个字母作为文件名的结尾（这一点在更大程度上来说只是一个惯例，而并非强制要求）。因此，文件名prog1.c对于读者正在使用的计算机来说，可能就是一个合法的C语言程序文件名。

我们通常文本编辑器把C语言编写的程序输入到计算机系统的文件中。例如，vi就是Unix操作系统中一种非常流行的文本编辑器（译者注：本书的文字就是使用VIM——vi的一个变种输入的）。使用文本编辑器生成的文件包含了C语言程序的原始形式，因此通常称为源文件（Source file）。程序一旦输入到源文件中，我们就可以着手来编译它了。

为了开始编译输入的源文件，我们需要让计算机执行特定的命令。当我们在命令行中输入这个编译命令时，还必须在后面跟上源文件的名字。在Unix操作系统中，开始编译C语言源文件的命令是cc。如果读者使用的是GNU C编译器，那么启动编译器的命令则是gcc。在命令行上输入下面的命令：

```
gcc prog1.c
```

就可以开始编译源文件prog1.c中包含的C语言程序。

在编译的第一阶段，编译器首先检查源程序的每一条语句，看它是否符合语言的语法和词法¹。如果编译器在这个阶段发现了错误，便会将这些错误报告给用户，然后停止运行。

¹准确地说，C语言的编译器首先会对源程序进行一遍预处理，检查程序中一些特定的指令。这些技术细节将在第13章“预处理器”中介绍。

程序员必须使用文本编辑器改正这些错误，并重新开始编译。这一阶段发现的典型错误通常包括不匹配的括号（词法错误），或者使用了未定义的变量（语法错误）等等。

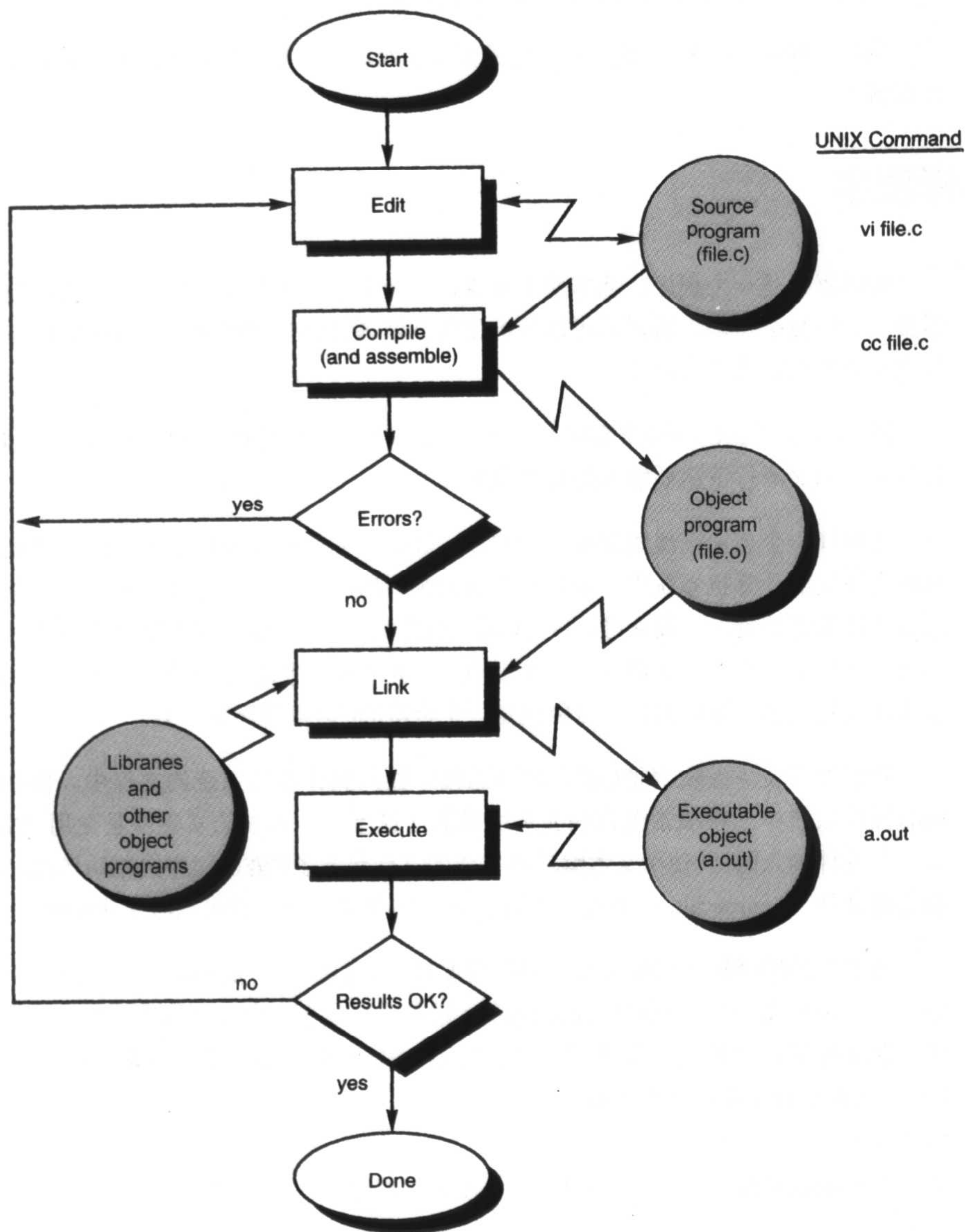


图 2.1 从命令行输入、编译和运行 C 语言程序的典型步骤

当程序中所有的语法和语义错误都被改正以后，编译器就会把高级语言编写的源程序翻译为较低级的形式。在绝大多数计算机系统上，这些高级语言程序通常首先被翻译为汇编语言程序，这些汇编语言程序完成的功能与高级语言程序相同。

源程序被翻译为对应的汇编语言程序之后，编译器还需要将这些汇编语言程序翻译成为实际的机器指令。这个步骤有时需要借助汇编器完成。在绝大多数计算机系统中，编译器通常会调用汇编器。

汇编器读入编译器生成的汇编语言程序，将其翻译为二进制格式的代码，这种代码被称为目标码。汇编器将这些生成的目标码保存在目标文件中。在Unix操作系统中，目标文件通常使用与源文件相同的文件名保存，但是文件名的结尾不是“.c”，而是“.o”。在Windows操作系统中，目标文件的结尾则是.obj。

生成目标文件以后，我们就可以进行下一个步骤——“连接”（Link）了。如果我们使用的是Unix平台下的cc命令或者gcc命令的话，这一步骤也是自动的。连接的主要作用是将目标代码转化为具体的计算机系统上实际的可执行程序。如果我们在源程序中调用了其他程序的话，那么连接程序就会把这些程序的目标代码和我们程序的目标代码连接在一起。如果我们的程序还使用了系统提供的库函数，这些库函数的代码也会被连接到最后生成的可执行程序中。

编译和连接一个程序的过程也常常被称为构建（building）。

连接步骤生成的可执行代码被连接器（Linker）保存在系统的可执行文件中。在Unix环境中，连接器生成的可执行文件，其默认文件名是a.out。在Windows操作系统中，这个可执行文件的名字通常与源文件名字相同，但是其文件名的结尾是.exe。

为了运行生成的可执行文件，我们只需要简单在系统命令行中输入下面的命令：

```
a.out
```

这个命令将可执行文件装入计算机的内存，然后开始运行其中的指令。

当程序开始运行后，计算机将会按顺序执行程序中的指令。如果程序需要从用户那里接收某些数据（也就是输入），系统将暂时挂起程序，以使用户输入。有时程序也可以开始等待某些事件的发生，如鼠标点击等。程序运行的结果通常输出到一个窗口中，这个窗口被称为终端（console）。有时程序也可以直接输出到系统文件中。

如果一切顺利（当程序第一次运行时，多半不会如此），程序将完成它的工作。如果程序的结果不正确，那么我们需要回过头来重新分析程序，排除程序逻辑错误的过程称为调试（debug）。在调试过程中，我们通常需要修改源文件，这时我们还需要重新编译、连接，以生成可执行程序。这个调试过程将不断重复，直到程序产生我们所需要的结果。

集成开发环境

我们前面大致介绍了使用C语言开发程序的每一个步骤，以及每个阶段需要输入的命令。在现代软件开发环境中，编辑、编译、运行和调试过程通常由单个应用程序控制，这个应用程序被称做集成开发环境（Integrated Development Environment, IDE）。IDE通常基于窗口环境，通过IDE我们可以很方便管理大型的软件，在窗口中编辑文件，以及编译、连接、运行、调试您的程序。

在Mac OS X操作系统中，Code Warrior和XCode是两个很流行的IDE。在Windows操作系统中，Microsoft Visual Studio是一个非常受欢迎的IDE。Kylix则是Linux下非常热门的IDE工具。IDE工具可以大大简化应用程序的开发过程，因此在学习软件开发时，掌握一种IDE是非常必要的。绝大多数IDE都支持使用好几种语言编写程序，比如除了C语言以外，还同时支持C#或者C++。

如果读者想了解IDE方面更多的知识，可以参照本书的附录E“C语言的其他资源”。

解释型语言

在结束对于编译高级语言程序的讨论之前，我们要特别提醒读者，还有另外一种方法，可以分析、执行使用高级语言编写的程序。这种方法不是编译程序，而是对程序进行解释（*interpreted*）执行，相应所需的程序被称为解释器。解释器不需要独立的编译、连接过程，它一边分析源程序，一边执行这些程序。当采用解释执行方法的时候，程序调试起来要容易一些，但是解释执行的程序通常比编译执行的程序要慢，因为编译型语言在执行程序之前，已经预先将程序中的指令翻译成为机器指令。

Basic和Java是解释型高级语言的两个例子。Unix的*shell*和python也是解释型的语言。有些软件商也提供C语言的解释器。

Compiling and Running Your First Program

编译并运行第一个程序

本章我们正式开始学习C语言，读者将明白使用C语言编程到底是怎么回事。为了达到这个目的，有什么方法比给出用C语言写的实际程序更好呢？

让我们先从一个比较简单的例子程序开始。这个程序在屏幕上打印"Programming is fun."（译者注：编程是有趣的）。程序3.1就是完成这个任务的C语言代码。

程序3.1 我们的第一个C语言程序

```
#include <stdio.h>

int main(void)
{
    printf("Programming is fun.\n");

    return 0;
}
```

在C语言中，大写字母和小写字母被看作不同的字符。另外，C语言不关心程序在文本行的开始位置，也就是说，我们可以在任意位置输入程序。利用这一点，我们可以对源程序进行排版，比如使用Tab键缩进某些行，这样写出的程序容易阅读一些。

编译我们的第一个程序

让我们回到刚才的程序。首先，我们需要把它输入到一个文件之中。文本编辑器可以帮助我们完成这个任务。Unix用户常常使用vi和emacs这两种文本编辑器。

绝大多数C语言编译器将文件名结尾为.c的文件看作C语言的源程序文件。我们假定读者将程序3.1输入并保存到一个名为prog1.c的文件中。下面我们来编译这个程序。

如果我们使用GNU C的编译器，编译过程会非常简单。我们只需要输入gcc，后面再跟上源文件的名称就可以了。编译命令如下：

```
$ gcc prog1.c
$
```

如果读者使用的是Unix附带的标准C语言编译器，那么使用的命令将是cc，而不是gcc。本书中，我们使用粗体来表示读者键入的内容。\$符号是shell的命令提示符。考虑到读者正在使用的计算机系统具体情况，读者实际所看到的命令提示符也许是其他的形式。

读者如果在输入程序的时候有某些拼写错误，在使用了gcc命令编译之后，编译器就会把它们显示出来。显示的错误信息通常还包括错误所在行的行号。如果程序顺利地通过了编译，系统将会显示另一个命令提示符，就像前边显示的那样。

编译器编译和连接完程序后，就会生成相应的可执行程序。如果读者使用的是GNU C编译器或者Unix标准C编译器，那么生成的可执行程序的文件名是a.out。在Windows平台中，生成的可执行文件名常常是a.exe（译者注：此处大概有误，Windows下各类C语言编译器生成的可执行文件的名称通常为prog1.exe）。

运行我们的第一个程序

只要在命令行输入可执行文件的名称，就可以运行我们生成的可执行程序¹：

```
$ a.out
Programming is fun.
$
```

如果我们打算给最终生成的可执行文件另外起个名字，可以使用编译器的-o选项，后面再跟上想要的可执行文件的名称。例如，如果我们使用下面的命令行运行编译器：

```
$ gcc prog1.c -o prog1
```

那么最终生成的可执行文件的名称就是prog1。在命令行上输入新生成的可执行文件的名称，程序同样可以运行。

```
$ prog1
Programming is fun.
$
```

¹ 如果读者运行的时候，系统提示下面的错误：No such file or directory，很可能是因为当前目录不在系统的环境变量PATH中。有两个方法能够解决这个问题：1.将当前目录增加到环境变量PATH中；2.运行下面的命令：./a.out。

理解我们的第一个程序

下面我们来仔细考察刚才的程序。程序的第一行如下：

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

它的作用是告诉编译器printf函数的一些基本信息，我们在后面的程序中就可以使用这个函数。几乎我们后面的程序中都会出现这一行语句。第13“预处理器”，将会更为细致地分析这行语句的作用。

下面的一行程序

```
int main(void)
```

定义了一个名字为main的函数，这个函数的返回值是整形数。整形数在C语言中被缩写为int。在C语言中，main是一个特殊的函数名字，当程序运行的时候，首先会从main函数开始执行。在main后面的一对大括号表明main是一个函数，括号中的关键字void表示main函数不需要任何参数。有关函数部分的知识将会在第8章“使用函数”中详细介绍。

到目前为止，我们已经告诉了编译器程序的main函数（也就是程序入口）所在。下面我们需要说明main函数到底要做些什么。所有被一对大括号{ }包围起来的语句都是main函数的语句。在程序3.1中，大括号中共有两条语句。第一条语句调用了名为printf的函数，并把一个字符串传递给它。这个字符串如下：

```
"Programming is fun.\n"
```

printf函数是C语言的一个标准库函数。这个函数把传递给它的参数打印在屏幕上。后面我们还将看到，printf函数也能接收多个参数。字符串的最后两个字符，也就是反斜线\和字母n，合起来被称为换行符。系统在处理输出的时候，如果遇到换行符，就会把后面的输出移到屏幕的下面一行。实际上，换行符的作用与打字机上的换行键功能类似。

C语言要求所有的语句必须以分号结尾。在main函数中，调用printf函数的那一行语句后面就有一个分号。

main函数的最后一个语句如下：

```
return 0;
```

这个语句将结束main函数的运行，并向操作系统返回一个整数0，作为程序的结束状态。一个程序返回值不必一定为0，实际上，任何整数都可以作为返回值。不过，按照惯例，如果程序返回0值，那就意味着程序运行一切正常。除了0以外的其他数字，可以用来表示各种不同的错误情况（例如文件没有找到等）。系统中的其他程序（如Unix shell）可以检查这个返回值，用以判断程序是否成功运行。

到此为止, 我们已经完成了对程序3.1的分析。下面我们可以尝试着修改一下这个程序, 让它再打印出一句"And programming in C is even more fun."。我们在程序中再加入一条调用printf函数的语句, 就可以做到这一点, 如程序3.2所示。这里我们再次提醒读者, C语言的每一条语句, 都需要以分号结尾。

程序3.2

```
#include <stdio.h>

int main(void)
{
    printf("Programming is fun.\n");
    printf("And programming in C is even more fun.\n");

    return 0;
}
```

输入程序3.2, 编译并运行它, 我们就可以在程序的输出窗口(有时也被称为终端)上看到如下的输出:

程序3.2 输出

```
Programming is fun.
And programming in C is even more fun.
```

如果我们要输出两行文字, 并不是一定要调用两次printf函数, 程序3.3向读者演示了这一点。请读者仔细看看下面的程序, 并分析一下这个程序的输出将会是怎样的(别欺骗自己哦!)。

程序3.3 显示多行输出

```
#include <stdio.h>

int main (void)
{
    printf ("Testing...\n..1\n...2\n....3\n");

    return 0;
}
```

程序3.3 输出

```
Testing...
..1
...2
....3
```

显示变量的值

printf函数是本书中使用最频繁的函数，它可以很方便地显示程序运行的结果。printf 函数不但可以显示简单的文字，也可以显示变量的值以及程序计算的结果。程序3.4使用printf 函数，显示50和25相加的结果。

程序3.4 显示变量的内容

```
#include <stdio.h>

int main (void)
{

    int sum;

    sum = 50 + 25;
    printf ("The sum of 50 and 25 is %i\n", sum);

    return 0;
}
```

程序3.4 输出

```
The sum of 50 and 25 is 75
```

在程序3.4中，main 函数的第一条语句声明了一个类型为整数的变量sum。C语言要求函数中使用的变量必须在函数开始的地方声明。一个变量的类型表明程序使用这个变量的方式，编译器可以根据变量的类型信息，决定如何存储和访问这个变量，并生成相应的指令。整型变量只能用来存放整数（也就是说，不能存放带有小数点的数值）。3、5、-20 和 0 都是整数。带有小数点的数，例如3.14、2.455、27.0 等，在C语言中被称为浮点数，或者实数。

在程序3.4中，整型变量sum 用来存储两个整数50 和25 的相加结果。在这里我们提请读者注意，程序在声明变量sum的语句后面放置了一个空行，这个空行把声明变量sum 的语句和main函数的其他语句分隔开来。这是一种程序书写的风格，它可以让程序看上去更容易阅读。空行本身并不是C语言的要求。

下面的程序语句：

```
sum = 50 + 25;
```

在绝大多数计算机语言中都代表同样的意思：把50 和25 加在一起（这个动作作用加号代表），然后把结果存储在变量sum 中（这个动作作用等号代表）。

程序3.4中的printf 函数，它后面的括号中有两个参数，这两个参数之间使用逗号隔开。printf函数的第一个参数用来表示需要显示的字符串，但是我们常常还需要显示程序中某些变量的值。比如在程序3.4中，我们就想要在字符串

The sum of 50 and 25 is

的后面显示变量sum 的值。为了达到这个目的，我们可以在传递给printf 函数的第一个字符串中，嵌入特殊的格式化输出符号。格式化输出符号以一个%号开始，后面跟上一个字母，这个字母通常代表需要输出变量的类型。printf 函数规定，在格式化输出符号中，字母 i 代表整型数²，这样，为了显示变量sum 的值，我们在传递给printf 函数的第一个字符串中放入一个整型数的格式化输出符号%i。

当printf 函数发现接收的第一个字符串中包含有格式化输出符号%i，这个函数就会找出传给自己的第二个参数的值，并把它显示在第一个字符串中格式化输出符号的位置上。在本例中，变量sum 的值就被显示在字符串"The sum of 50 and 25 is"的后面。

现在，请读者试着分析一下，下面的程序3.5将会如何显示。

程序3.5 显示多个变量的值

```
#include <stdio.h>

int main (void)
{
    int value1, value2, sum;

    value1 = 50;
    value2 = 25;
    sum = value1 + value2;
    printf ("The sum of %i and %i is %i\n", value1, value2, sum);

    return 0;
}
```

程序3.5 输出

```
The sum of 50 and 25 is 75
```

² printf 函数的格式化输出符号中，d 也代表整型数。为了一致起见，本书在后面的章节中坚持使用字母 i。

程序3.5的第一行声明了三个整型变量value1、value2和sum。这个语句与下面三条语句的作用相同：

```
int value1;  
int value2;  
int sum;
```

声明完变量之后，我们给变量value1赋值50，变量value2 赋值25，然后把这两个变量相加，并将结果存放到变量sum 中。

在程序3.4中，我们给printf 函数传递了四个参数。第一个参数仍然是一个字符串，这个参数通常被称为格式字符串，printf 函数使用它来决定如何显示传递给自己的所有参数。程序3.4中printf 函数的格式字符串中包含三个格式化输出符号，因此printf 函数依次将value1、value2和sum 值插入到格式化输出符号的相应位置，并显示整个字符串。

注释

我们通过本章的最后一个程序3.6，向读者展示如何在C语言中使用注释。注释可以用来说明程序的作用，进而增加程序的可读性。程序中的注释可以把程序书写者在编写程序时的想法、假设、意图等描述出来，并展示给阅读程序的人——无论是程序的原作者，还是程序的维护者。

程序3.6 在程序中使用注释

```
/* This program adds two integer values  
   and displays the results */  
  
#include <stdio.h>  
  
int main (void)  
{  
  
    // Declare variables  
    int value1, value2, sum;  
  
    // Assign values and calculate their sum  
    value1 = 50;  
    value2 = 25;  
    sum = value1 + value2;
```

程序3.6 续

```
// Display the result
printf ("The sum of %i and %i is %i\n", value1, value2, sum);

return 0;
}
```

程序3.6 输出

```
The sum of 50 and 25 is 75
```

有两种方法可以在C语言中加入注释。第一种方法，我们可以用 `/*` 开始一段注释。对于这种方法加入的注释，我们必须手工指定注释的结束位置，这可以通过在注释结束的地方加上符号 `*/` 来完成。需要注意的是，开始注释的两个字符 `/` 和 `*` 以及结束注释的两个字符 `*` 和 `/` 之间都不能有空格。所有在 `/*` 和 `*/` 之间的内容，C语言编译器都会把它当作注释而忽略。如果注释由多行文字组成，我们通常使用这种形式。在程序中加入注释的第二种方法是使用两个连续的斜线——`//`。在两个斜线之后的内容，直到一行结束，都被编译器当作是注释。

在程序3.6中，一共使用了四条注释。除了这些注释语句，程序3.6和3.5实际上是一样的。老实说，程序3.6中的注释，除了第一条有用以外，其他都是为了说明使用注释的方法而人为加上的（实际上，在程序中加入过多的注释，程序的可读性不但不会增加，反而会下降!）。

在程序中合理的使用注释是非常重要的。仅仅过了六个月，程序员就发现已经想不起来原来写的某个特定的函数或者一组语句的作用，这种事情实在是屡见不鲜。如果在合适地方有一个注释，将会大大缩短程序员重新理解某段程序逻辑的时间。

在输入程序的时候，同时输入注释是个很好的习惯。这样做有如下几个好处：1.当我们正在编写程序的时候，脑子里程序的逻辑还很清楚，这个时候写注释比其他时候写要容易一些。2.如果我们在程序的编写阶段就输入注释，那么当我们调试程序的时候，这些注释就可以帮助定位和解决逻辑错误。实际上，注释不但可以帮助我们阅读程序，通过注释，我们也容易发现程序中的逻辑错误。3.老实说，我还没有发现过喜欢注释的程序员。程序员输完程序并且调试通过以后，多半就不愿意回过头来重新在程序中插入注释。因此，如果我们在输入程序的时候就输入注释，就可以在不知不觉中完成这件显得有些琐碎的工作。

到了该结束本章的时候了。到这儿为止，读者对于使用C语言编程应该有了一个大概的印象，也能够独立书写简单的C语言程序了。在下一章，我们将会进一步介绍这门强大而又灵活的计算机编程语言中更复杂的一些知识。但是在那之前，请读者先试着完成后面的习题，以便巩固本章学到的知识。

练习

1. 输入并运行本章的六个程序。把每个程序的输出结果与书中列出的结果进行对比。
2. 用C语言写一个程序，在终端上输出下面的句子。

1. In C, lowercase letters are significant.
2. main is where program execution begins.
3. Opening and closing braces enclose program statements in a routine.
4. All program statements must be terminated by a semicolon.

3. 下面这段程序的输出是什么？

```
int main (void)
{
    printf ("Testing...");
    printf ("....1");
    printf ("...2");
    printf ("..3");
    printf ("\n");
    return 0;
}
```

4. 写出计算87减去15的C语言程序，并把结果和合适的提示信息打印出来。
5. 找出下面程序中的语法错误。当你确信找出了所有的错误之后，请输入改正后的程序，并编译运行。

```
#include <stdio.h>

int main (Void)
{
```

```
    INT sum;
    /* COMPUTE RESULT
    sum = 25 + 37 - 19
    /* DISPLAY RESULTS //
    printf ("The answer is %i\n" sum);
    return 0;
}
```

6. 下面程序的输出是什么？

```
#include <stdio.h>

int main (void)
{
    int answer, result;

    answer = 100;
    result = answer - 10;
    printf ("The result is %i\n", result + 5);

    return 0;
}
```


Variables, Data Types, and Arithmetic Expressions

变量、数据类型和算术表达式

在本章中，读者将会了解到C语言中变量和常量更多的知识。我们还将认真研究C语言的基本数据类型，以及使用C语言书写算术表达式的基本规则。

使用变量

对于早期的程序员来说，使用机器语言书写程序是一件繁重的工作。程序员必须把所有的计算机指令表达为二进制的数字，然后才能将程序输入计算机。另外，如果需要存取内存中的数据，程序员还必须使用特殊格式的内存地址。

现代的编程语言让程序员能够专注于需要解决的问题，而不是特定的机器指令或者内存地址。程序员可以使用符号（这些符号被称为变量）来代表计算中使用的数据以及结果。我们也可以根据所存储数据的类别和作用，为符号选择合适的名称（变量名）。

在第3章“编译和运行第一个程序”，我们已经使用了几个变量存储整数，如程序3.4中的变量sum，我们用它来存储50和25的和。

C语言提供的数据类型不仅限于整数，我们还可以声明浮点数、字符甚至是用来表示计算机中某个内存位置的指针类型变量。

变量的命名规则很简单：一个变量名开始必须是一个字母或者下画线（_），后面可以是字母（大写或者小写）、下画线或者数字（0-9）的任意组合。下面列出了一些合法的变量名：

```
sum
pieceFlag
i
J5x7
Number_of_moves
_sysflag
```

下面的变量名则是不合法的：

sum\$value	变量名中不允许使用\$符号
piece flag	变量名中不允许使用空格
3Spencer	数字不能作为变量名的开始
int	int是一个保留字

因为int是C语言的保留字，所以它不能用来作为变量的名字。所谓保留字就是对于C语言的编译器来说有着特殊意义的符号。一般说来，所有的保留字都不能用作变量名。附录A，“C语言小结”，给出了C语言关键字的完整列表。

变量名的长度是任意的，但是对于编译器来说，只有前63个字符是有意义的。在某些特殊情况下（如附录A中所述），则只有前31个字符有意义。使用过长的变量名并没有太多实际意义，并且这样的变量名输入起来也很麻烦。比如，虽然下面的语句是正确的：

```
theAmountOfMoneyWeMadeThisYear = theAmountOfMoneyLeftAttheEndOfTheYear -
    theAmountOfMoneyAtTheStartOfTheYear;
```

但是表达同样的意思，下面的语句就要简洁得多。

```
moneyMadeThisYear = moneyAtEnd - moneyAtStart;
```

给变量取名时，请读者记住下面的忠告：不要偷懒，要给变量起一个能够反映其用途的名字。和注释的作用类似，合适的变量名能够大大增加程序的可读性，从而减少程序员在调试和文档方面的工作量。实际上，如果变量名起得合适的话，文档方面的工作就可以大幅度减少，因为仅仅阅读源程序就很容易理解程序。

数据类型与常量

我们已经见过C语言的一种基本数据类型——`int`。前面我们介绍过，整型变量只能用来存储整数，也就是说，没有小数点的数字。

C语言还有四种基本的数据类型：`float`、`double`、`char`和`_Bool`。`float`类型的变量可以存储浮点数（也就是带有小数点的数字）。`double`类型与`float`类型相同，但是它的精度是`float`类型的两倍。`char`类型的变量可以用来存储单个字符，例如字母`a`，数字`6`，或者一个分号`(;)`（以后我们还将仔细的介绍`char`类型）。最后，`_Bool`类型只能用来存储数字`0`或者`1`。这个类型的变量可以用来表示开关、是否、真假等。

在C语言中，任何数字、单个字符或者字符串都被看作是常量。例如，数字`58`在C语言中代表一个整数类型的常量，字符串`"Programming in C is fun.\n"`是一个字符串常量。而全部由常量组成的表达式就是常量表达式。因此下面的表达式

```
128 + 7 - 17
```

就是一个常量表达式，因为组成该表达式的每一项都是一个常量。

如果`i`是一个整型变量，那么下面的表达式

```
128 + 7 - i
```

就不是常量表达式。

基本整数类型 `int`

在C语言中，整数是一个或者多个数字组成的序列。如果在这个序列前面有一个减号`(-)`，这个数字则是一个负数。`158`、`-10`和`0`都是合法的整数。整数之中不能嵌入空格，大于`999`的整数也不能包含逗号（因此，`12,000`不是合法的整数，应该写作`12000`）。

C语言用两种专门的格式来表示不是以`10`为基数的整数。如果一个整数的开始数字是`0`，那么这个数字就被看作一个八进制数，八进制数的每一个数字只能是`0-7`。如果要表示八进制数`50`（八进制数`50`等于十进制数`40`），就应该写成`050`。与之类似，八进制数`0177`代表十进制数`127`（ $1 \times 64 + 7 \times 8 + 7$ ）。如果要在终端上显示八进制数，可以在`printf`函数的格式化字符串中使用`%o`。用`%o`输出的八进制数前面不显示`0`。如果要显示`0`，则需要使用`%#o`。

如果一个整数常量的开始符号是`0`和`x`（不论是大写还是小写），这个数字就被当作是十

六进制数。0x后面的数字都是十六进制的数字，这些数字包括0-9和a-f（或者A-F），其中字母用来表示10到15。如上所述，如果我们要把十六进制数FFEF0D赋值给名为rgbColor的变量，可以使用下面的语句

```
rgbColor = 0xFFEF0D;
```

格式化输出符号%x可以用来输出十六进制数，这个符号输出的十六进制数前面没有0x，并且使用小写字母a-f。如果需要在输出中显示十六进制数前面的0x，我们应该使用符号%x。因此，如果要显示变量rgbColor的值，可以使用下面的语句

```
printf ("Color is %x\n", rgbColor);
```

如果在格式化输出符号中使用大写的X，如%X或者%X，那么在终端上输出的十六进制数前面的X和字母就是大写的。

存储空间与范围

无论是整数、浮点数还是字符变量，所存储的数字都有一个范围。这个存储范围与存储该类型变量所需要的空间有关。一般来说，C语言本身并没有对这方面进行具体规定，这些数据取决于具体的计算机系统，也就是说，这些数据是实现相关或者机器相关的。例如，读者使用的计算机可能使用32位来存储整数，也可能使用64位来存储整数。因此我们在编写程序的时候，千万不要对于数据类型的存储空间进行假定。但是，C语言确实规定了每个基本数据类型所需要的最小存储空间。例如，一个整数所占用的存储空间最少是32位，在许多计算机平台上，这个大小刚好就是一个内存字的大小。读者可以从附录A的表A.4中找到这方面更多的信息。

浮点数类型 float

浮点类型的变量可以存储带有小数点的数字。在表示浮点数常量时，有时可以省略小数点前面或者后面的数字，但是不能两者都省略。3.、125.8、-.0001都是合法的浮点数常量。如果需要在终端上显示浮点数，可以使用格式化输出符号%f。

我们也可以使用科学计数法表示浮点数。表达式1.7e4就是科学计数法的例子，它代表 1.7×10^{-4} 。在科学计数法中，字母e前面的数字是尾数，e后面的数字是指数。指数是以10为底的，前面可以有正号（+）或者负号（-）。因此，在表达式2.25e-3中，2.25是尾数，-3是指数。这个表达式代表的数字是 2.25×10^{-3} ，或者0.00225。顺便说明一下，用来分隔尾数和指数的字母e，大小写都是合法的。

如果需要在终端上使用科学计数法显示浮点数，可以使用格式化输出符号`%e`。另外，输出浮点数时如果使用格式化符号`%g`，那么`printf`函数就会自行决定如何显示一个浮点数：如果这个数的指数小于-4或者大于5，`printf`就会使用科学计数法，否则将使用正常的格式。

我们建议读者坚持使用`%g`输出浮点数，它的输出格式总是会美观一些。

十六进制的浮点数表达式前面是`0x`或者`0X`，后面是一个或者多个十六进制数字，在后面是字母`p`或者`P`，最后是一个可选的以2为底的指数。例如，十六进制数`0x0.3p10`代表数值 $3/16 \times 2^{10} = 192$ 。

扩展精度的浮点类型 double

`double`类型与`float`类型非常相似，但是它提供了比`float`类型更高的存储精度。一般来说，`double`使用的存储空间是`float`的两倍。绝大多数计算机使用64位空间存储`double`类型的变量。

除非特别指明，所有的浮点数常量都被C语言当作`double`类型处理。为了明确说明一个`float`类型的浮点常量，我们需要在表达式后面加上字符`f`或者`F`，如下所示：

```
12.5f
```

显示`double`类型的变量使用的格式化符号与`float`类型相同，即`%f`、`%e`或者`%g`。

字符类型 char

`char`类型的变量用于存储单个字符¹。一个字符常量由一对单引号和这对单引号括起来的字符组成。`'a'`、`';`和`'0'`都是合法的字符常量表达式。其中，第一个代表字母a，第二个代表分号，第三个代表字符0——注意，字符0和数字0是不同的。另外，在这里我们特别提醒读者，不要把字符常量和字符串搞混，前者是用单引号括起来的单个字符，后者是用双引号括起来的任意多个字符。

`'\n'`也是一个合法的字符常量，虽然看上去它和我们前面提出的规则并不符合。这是因为，反斜线`\`是C语言中的特殊字符，编译器并不把它当作一个字符。也就是说，虽然`'\n'`中包含两个字符，编译器实际上只把它当作单个字符。C语言中还有一些以反斜线开始的特殊字符，附录A给出了一个完整的列表。

¹附录A讨论了如何使用特殊转义序列、通用字符以及宽字符来存储扩展字符集中的字符。

如果要在终端上显示单个字符，可以使用格式化输出符号%c。

布尔类型 _Bool

布尔类型的变量可以用来存储0和1。C语言本身并没有明确规定布尔变量所需要的存储空间。当程序中需要保存布尔条件时，例如记录数据是否应该全部从文件中读出，就可以使用布尔变量。

按照C语言的惯例，0代表假，1代表真。当我们给布尔类型的变量赋零值时，该变量存储0，赋其他值的时候，该变量存储1。

为了方便程序员使用布尔变量，标准的C语言头文件<stdbool.h>定义了符号bool、true和false。第6章“进行判断”中的示例程序6.10A，演示了如何使用这些预定义的值。

程序4.1演示了如何使用C语言的基本类型。

程序4.1 使用基本数据类型

```
#include <stdio.h>

int main (void)
{
    int integerVar = 100;
    float floatingVar = 331.79;
    double doubleVar = 8.44e+11;
    char charVar = 'W';

    _Bool boolVar = 0;

    printf ("integerVar = %i\n", integerVar);
    printf ("floatingVar = %f\n", floatingVar);
    printf ("doubleVar = %e\n", doubleVar);
    printf ("doubleVar = %g\n", doubleVar);
    printf ("charVar = %c\n", charVar);

    printf ("boolVar = %i\n", boolVar);

    return 0;
}
```

程序4.1 输出

```
integerVar = 100
floatingVar = 331.790009
doubleVar = 8.440000e+11
```

程序4.1 输出（续）

```
doubleVar = 8.44e+  
charVar = W  
boolVar = 0;
```

程序4.1的第一行声明了整型变量integerVar，并给它赋以初值100。这条语句与下面两条语句的作用相同。

```
int integerVar;  
integerVar = 100;
```

请读者注意程序输出的第二行：我们在程序中给浮点变量floatingVar赋以初值331.79，但是计算机的输出却是331.790009。实际的输出结果取决于读者所使用的计算机。这个误差是由计算机内部表示浮点数的格式引起的。读者在使用计算器的时候很可能也遇到过这类现象。比如，如果我们用计算器计算1除以3，得出的结果是.33333333，或者再多几个3。这就是计算器给出的三分之一的近似值。从理论上来说，如果用浮点数表示三分之一的话，应该有无穷多个3，但是计算器存储精度是有限的，因此造成了这个误差。同样，有很多浮点数也无法在计算机中精确表示，因此出现了我们在上面程序中看到的误差。

printf函数提供了三种格式化输出符号，用于显示浮点数的值。%f使用标准的格式显示浮点数。如果没有特别指定，printf函数在使用%f显示浮点数时，只显示6位数字。本章后面我们还会介绍，如何显示更多的位数。

%e使用科学计数法显示浮点数，如果没有特别指定，该格式同样只显示6位数字。

当使用%g输出浮点数时，printf函数自动选择使用标准格式还是科学计数法。%g不显示浮点数后面多余的0。如果某个浮点数在小数点后没有位数，%g还将省略小数点。

上面程序中倒数第二条printf语句使用%c显示字符变量charVar的值，该变量被赋值为'W'。这里再次提醒读者注意，字符串（就是我们传给printf函数的第一个参数）总是使用双引号括起来，而字符常量则使用单引号。

对于_Bool类型的变量，我们可以使用%i显示其内容，如程序4.1的最后一条语句所示。

类型修饰符: long, long long, short, unsigned 和 signed

声明整型变量时, `int` 前面如果使用修饰符 `long`, 该变量表示的数字范围在某些计算机系统上将会增加。下面是一个例子。

```
long int factorial;
```

采用这种方式声明的变量被称为长整型变量。和 `float`、`double` 类型相似, 长整型变量能够存储的数字范围依赖于具体的计算机系统。在许多计算机系统中, 长整型和普通整型, 其表示数字的范围是相同的, 都是32位 (231 -1, 或者 -2, 147, 483, 647)。

在整型常量表达式后面加上字母 `l` (大小写均可), 就代表长整型的常量表达式。数字和字母 `l` 之间不允许有空格。下面的语句声明了一个长整型变量 `numberOfPoints`, 并给它赋以初值 131, 071, 100。

```
long int numberOfPoints = 131071100L;
```

使用 `printf` 函数显示长整型变量时, 需要在格式化输出符号 `i`、`o` 和 `x` 前面加上修饰符 `l`。也就是说, 使用 `%li` 是以十进制显示一个长整型变量, 使用 `%lo` 显示为八进制, 使用 `%lx` 则显示为十六进制。

我们还可以使用 `long long` 修饰符修饰整型变量。例如:

```
long long int maxAllowedStorage;
```

这种类型的变量, 最少占有64位的存储空间。如果要输出 `long long int` 类型的变量, 需要在普通的格式化输出符号前面加上两个 `ll`, 例如 `%lli`。

`long` 也可以用于修饰 `double` 类型的变量, 例如:

```
long double US_deficit_2004;
```

在普通的浮点数表达式后面加上 `l` 或者 `L`, 形成 `long double` 类型的常量表达式。如:

```
1.234e+7L
```

如果要显示 `long double` 类型的变量, 在浮点数格式化输出符号前面加上 `L` 就可以了。也就是说, `%Lf` 以标准格式显示 `long double` 类型, `%Le` 以科学计数法显示, `%Lg` 则自动在 `%Lf` 和 `%Le` 之间选择。

与 `long` 修饰符相反, 还有一种 `short` 修饰符。如果我们在 `int` 型变量声明前面加上 `short` 修饰符, 编译器就假定该变量只是用来存放相对较小的整数。使用 `short` 修饰符的主要原因是节约内存。特别当程序需要大量的内存, 而内存的数量又相当有限时。

在某些计算机系统中，short int类型的变量占用的存储空间是普通int类型变量的一半。无论如何，short int类型变量的存储空间不会少于16位。

C语言没有提供short int类型的常量表达式。如果需要输出short int类型的变量，可以在普通整型数的格式化输出符号前面放一个字母h，如%hi，%ho，%hx。实际上我们也可以直接使用整型数的格式化输出符号：%i，%o和%x。当一个short int类型的变量作为参数传递给printf函数时，C语言会自动将其转化为普通的整型变量。

我们这里介绍的最后一个修饰符是unsigned。当我们需要声明一个只存储正数的整型变量时，可以使用这个修饰符，如下所示：

```
unsigned int counter;
```

告诉编译器，我们只在counter变量中保存正数。当我们使用unsigned修饰符限定一个变量中只存储正数以后，这个变量所能存储数的范围就扩展了。（译者注：实际上并未增加，只不过unsigned变量能够存储更大的正数。）

在普通的整数常量表达式后面放一个字母u（或者U），就形成unsigned int类型的常量表达式。例如：

```
0x00ffU
```

字母u（或者U）和字母l（或者L）可以混合使用，例如：

```
20000UL
```

则是一个长整型无符号常量表达式。

我们已经介绍了数种整型常量表达式，下面来看看C语言对于常量表达式的处理规则：当一个整型常量表达式后面没有任何修饰符如u、U、l或者L时，C语言就假定它是一个int类型的表达式。如果这个表达式的值无法存放到普通的int中，那么编译器将其作为一个unsigned int处理。如果unsigned int也无法存放该数字，C语言编译器则将其作为一个long int处理。如果long int还是无法存储这个数字，编译器则尝试用unsigned long int存储该数。如果unsigned long int也无法存放，编译器还将依次尝试long long int和unsigned long long int。

当我们声明long long int、long int、short int或者unsigned int类型的变量时，可以省略关键字int。所以，前面例子中定义无符号变量counter也可以使用下面的语句声明：

```
unsigned counter;
```

char类型的变量声明前面，也可以使用unsigned修饰符。

signed修饰符用于明确地通知编译器，某个变量是有符号的。signed修饰符主要用于char类型的声明，在第14章“进一步讨论数据类型”中，我们还将详细研究这个问题。

如果读者觉得我们这里讨论的修饰符有些深奥难懂的话，不必担心。在本书后面的章节中，我们将会通过许多实际的代码来演示如何使用这些类型，在第14章将详细讨论这些问题。

表4.1中总结了我们迄今为止讨论的基本数据类型和修饰符。

表4.1 基本数据类型

Type	Constant Examples	printf chars
char	'a', '\n'	%c
_Bool	0,1	%i, %u
short int	-	%hi, %hx, %ho
unsigned short int	-	%hu, %hx, %ho
int	12, -97, 0xFFE0, 0177	%i, %x, %o
unsigned int	12u, 100U, 0xFFu	%u, %x, %o
long int	12L, -2001, 0xffffL	%li, %lx, %lo
unsigned long int	12UL, 100ul, 0xffeeUL	%lu, %lx, %lo
long long int	0xe5e5e5e5LL, 50011	%lli, %llx, %llo
unsigned long long int	12ull, 0xffeeULL	%llu, %llx, %llo
float	12.34f, 3.1e-5f, 0x1.5p10, 0x1P-1	%f, %e, %g, %a
double	12.34, 3.1e-5, 0x.1p3	%f, %e, %g, %a
long double	12.341, 3.1e-51	%Lf, %Le, %Lg

使用算术表达式

在C语言中——实际上几乎所有的编程语言中，加号(+)用于计算两个数的和，减号(-)用于计算两个数的差，星号(*)用于表示两数相乘，斜线(/)用于表示两数相除。因为这些操作符需要两个操作数，因此它们被称为二元操作符。

我们前面已经演示过在C语言中如何进行相加运算。程序4.2进一步给出了减法、乘法和除法的例子。程序4.2的最后两个运算中引入了操作符优先级的概念。实际上，C语言中的每一个操作符都有自己的优先级，对于包含多个操作符的表达式，C语言首先处理优先级高的操作符。如果一个表达式中包含多个相同优先级的操作符，则根据具体操作符的不同，有可能从左向右求值，也可能从右向左求值。这个特性被称为操作符的关联性。附录A给出了C语言操作符优先级以及关联性的完整列表。

程序4.2 使用算术操作符

```
// Illustrate the use of various arithmetic operators

#include <stdio.h>

int main (void)
{
```

程序4.2 续

```
int a = 100;
int b = 2;
int c = 25;
int d = 4;
int result;

result = a - b; // subtraction
printf ("a - b = %i\n", result);

result = b * c; // multiplication
printf ("b * c = %i\n", result);

result = a / c; // division
printf ("a / c = %i\n", result);

result = a + b * c; // precedence
printf ("a + b * c = %i\n", result);

printf ("a * b + c * d = %i\n", a * b + c * d);

return 0;
}
```

程序4.2 输出

```
a - b = 98
b * c = 50
a / c = 4
a + b * c = 150
a * b + c * d = 300
```

程序4.2首先声明了5个整型变量a、b、c、d和result，并给a、b、c、d赋以初值。随后程序计算a减去b的结果，并使用printf函数将其显示在终端中。

随后的语句

```
result = b * c;
```

计算b乘以c的积，并把结果存放在result变量中。接着，程序使用我们熟悉的printf函数显示result中的数值。

下面的程序使用了除法操作符/。我们计算100除以25，并将结果4使用printf函数显示出来。

在某些计算机系统中，使用0作为除数会导致程序异常退出²。即使程序没有非正常退出，使用0作为除数，得出的结果也是没有意义的。

在第6章，我们将介绍如何在进行除法操作之前，检查除数是否为0。如果除数等于0，我们可以采用合适的补救措施，避免进行除法运算。

接下来，我们在程序中看到，下面的表达式

```
a + b * c
```

其结果并不等于2550 (102 * 25)。实际上，我们从printf的输出可以看出，表达式的结果是150。和绝大多数编程语言一样，C语言在计算表达式的值时，采用了操作符优先级的规则。通常情况下，表达式从左向右求值。然而，由于乘法和除法的优先级高于加法和减法，因此，表达式

```
a + b * c
```

实际上是按照下面的顺序求值的。

```
a + ( b * c )
```

这个规则和我们在普通代数课中学到的求值顺序相同。

如果想要改变表达式的求值顺序，我们可以使用括号。实际上，上面列出的表达式是一个完全合法的C语言表达式，它的计算结果与程序4.2中的表达式完全相同。然后，如果我们用下面的表达式

```
result = ( a + b ) * c
```

那么result中的结果就是2550了。因为使用了括号以后，a (100) 和b (2) 首先被加在一起，然后再和c (25) 相乘。括号也可以嵌套，遇到这种情况，C语言按照从里向外的顺序对括号中的项求值。使用括号的时候，读者务必注意左括号与右括号的数量要匹配。

在程序4.2的最后一个语句中，我们将表达式作为一个参数直接传递给printf函数，而不是先把它赋给result变量。在C语言中这样做是完全可行的。C语言首先对表达式

```
a * b + c * d
```

² 当在 Windows 平台中使用 gcc 编译器时就会发生这种情况。在某些 Unix 系统中，程序并不会结束，而是根据被除数的情况，如果是整数，给出结果 0；如果是浮点数，会给出结果无穷大。

按照下面的顺序求值

$(a * b) + (c * d)$

即

$(100 * 2) + (25 * 4)$

然后再将结果300传递给printf函数。

整数算术和单目减法操作符

程序4.3进一步巩固了我们前面介绍的知识，并引入整数算术的概念。

程序4.3 关于算术操作符的进一步范例

```
// More arithmetic expressions

#include <stdio.h>

int main (void)
{
    int a = 25;
    int b = 2;

    float c = 25.0;
    float d = 2.0;

    printf ("6 + a / 5 * b = %i\n", 6 + a / 5 * b);
    printf ("a / b * b = %i\n", a / b * b);
    printf ("c / d * d = %f\n", c / d * d);
    printf ("-a = %i\n", -a);

    return 0;
}
```

程序4.3 输出

```
6 + a / 5 * b = 16
a / b * b = 24
c / d * d = 25.000000
-a = -25
```

程序的开始声明了四个变量a, b, c和d。为了达到对齐的效果，我们声明变量a, b时，在int和变量名之间加入了额外的空格。这样程序的可读性会好一些。读者可能已经注意到，迄今为止，在我们所有的程序中，每个操作符两边都有空格，这也是为了使得程序显

得美观，C语言本身并不要求这样。一般说来，如果程序某个地方允许输入一个空格，那么多个空格也会合法。为了最终的程序美观一些，多按几下空格键总是值得的。

程序4.3中第一个printf函数中的表达式也体现了操作符优先级的概念。这个表达式按照如下的顺序求值：

1. 因为除法的优先级高于加法，所以计算机首先计算a (25) 除以5，其结果是5，并作为中间结果保存起来。
2. 因为乘法的优先级也高于加法，所以计算机接下来会用中间结果5乘以b的值2，得出一个新的中间结果10。
3. 最后，6和新的中间结果10加在一起，得出最终结果16。

第二个printf语句又有一些新花样。读者可能会以为用a除以b然后再乘以b，最后的结果还是a的值，也就是25，实际上，printf显示的结果是24。乍看上去，计算机在运算过程中弄丢了一位数字，实际上这是因为计算机使用整数算术对该表达式求值的结果。

在程序的开始，我们将a、b都声明为整型变量。无论什么时候，如果某个运算的两个操作数都是整型数，C语言就会使用整数算术的规则执行该运算。这种情况下，所有运算的小数点部分都会被丢弃。按照这个规则，当使用a除以b，也就是25除以2时，我们得到的中间结果将是12，而不是12.5。将这个中间结果乘以2，就得到了最终结果24，这也解释了为什么我们的运算结果看上去弄丢了数字。总之，如果在两个整数之间进行除法操作，得到的结果一定是一个整数。

在程序4.3的第三个printf语句中我们可以看到，如果使用浮点数进行上述运算，那么结果就会如我们所想的一样，得出25。

在编写程序的时候到底是使用int型的变量还是使用float型的变量，是根据程序的实际需要决定的。如果我们不需要任何小数结果，那么就应该使用整型变量。使用整型变量的程序效率较高，在很多计算机上都会执行得较快。另一方面，如果我们需要精确的结果，就应该使用浮点数。还有一点，C语言本身提供了三类浮点数：float、double和long double，具体在程序中使用哪种类型，要根据程序处理的计算精度和数字大小决定。

在最后一个printf语句中，我们使用单目减法操作符，得出了a的相反数。单目操作符就是只需要一个操作数的操作符。与之相对应，双目操作符需要两个操作数。

除了单目加法操作符(+)之外，单目减法操作符的优先级比其他算术操作符的优先级都要高，而这两个单目操作符的优先级则是相同的。因此，程序4.3的最后一个printf语句

中执行的运算是 $-a$ 乘以 b 。我们再次提醒读者，如果想要完整地了解操作符的优先级，可以参阅附录A中的表格。

余数操作符

下面我们将要介绍余数操作符，它的符号是百分号（%）。读者可以用心看看程序4.4，研究一下余数操作符是如何工作的。

程序4.4 演示求余操作符

```
// The modulus operator

#include <stdio.h>

int main (void)
{
    int a = 25, b = 5, c = 10, d = 7;

    printf ("a %% b = %i\n", a % b);
    printf ("a %% c = %i\n", a % c);
    printf ("a %% d = %i\n", a % d);
    printf ("a / d * d + a %% d = %i\n",
            a / d * d + a % d);

    return 0;
}
```

程序4.4 输出

```
a % b = 0
a % c = 5
a % d = 4
a / d * d + a % d = 25
```

程序4.4的第一行声明了四个变量 a 、 b 、 c 和 d ，并给它们赋以初值。

我们已经知道，`printf`函数中，%后面紧跟的一般是格式化输出符号，用以描述如何输出传递给`printf`函数的其他参数。但是，如果%后面还跟着一个%的话，它的意思是要求`printf`真正地输出一个%。

看完程序4.4和它的输出以后，读者可能已经发现，余数操作符的作用是计算第一个数除以第二个数所得的余数。在第一个`printf`语句中，25除以5，余数是0。如果用25除以10，那么余数就是5，正如第二个`printf`语句显示的那样。在第三个`printf`语句中，我们使用25除以7，得到的余数是4。

程序4.4的最后一行输出需要解释一下。首先，读者会注意到语句写在了两行上，这在C语言中是完全合法的。实际上，在C语言的一条语句中任何一个可以使用空格的地方，我们都可以将空格后面的内容写到下一行中（但是在字符串中这样做是不可以的，具体情况见第10章，“字符串”）。在某些情况下，将一条语句某些部分另起一行书写，不仅是可以的，也是必须的，例如在一条语句过长的情况下。程序4.4中，我们将续行书写的语句缩进排版，以便突出它是上一行语句的继续。

现在我们把注意力转到如何对最后一个表达式求值。首先，请读者注意，两个整数之间的任何运算都采用整数运算的法则，两个数相除所得的余数将被舍弃，所以， a/d ，也就是 $25/7$ ，其结果为3。然后这个中间结果再和 d 相乘，即3乘以7，我们得出中间结果21。最后，我们把这个中间结果和 a 除以 d ——即 $a\%d$ 的余数相加，得出最终结果25。这个结果恰好等于 a 的值25。这并不是一种巧合，一般说来，只要 a 和 b 都是整数，下面的表达式

$$a / b * b + a \% b$$

的值总是等于 a 。实际上，余数操作符只作用于整数。

余数操作符的优先级与乘除法的优先级相同。也就是说，下面的表达式：

```
table + value % TABLE_SIZE
```

实际上等价于

```
table + (value % TABLE_SIZE)
```

整型数与浮点数之间的转换

为了更好的编写C语言程序，我们还必须了解C语言中整型数和浮点数之间隐式转换的规则。程序4.5演示了数据类型转换的几种简单情况。某些编译器在编译这个程序时可能会给出警告，用于提醒编程者程序中隐式转换的存在。

程序4.5 整型数与浮点数之间的转换

```
// Basic conversions in C

#include <stdio.h>

int main (void)
{
    float f1 = 123.125, f2;
```


程序4.5 续

```
int i1, i2 = -150;
char c = 'a';

i1 = f1;           // floating to integer conversion
printf ("%f assigned to an int produces %i\n", f1, i1);

f1 = i2;           // integer to floating conversion
printf ("%i assigned to a float produces %f\n", i2, f1);

f1 = i2 / 100;      // integer divided by integer
printf ("%i divided by 100 produces %f\n", i2, f1);

f2 = i2 / 100.0;    // integer divided by a float
printf ("%i divided by 100.0 produces %f\n", i2, f2);

f2 = (float) i2 / 100; // type cast operator
printf ("(float) %i divided by 100 produces %f\n", i2, f2);

return 0;
}
```

程序4.5 输出

```
123.125000 assigned to an int produces 123
-150 assigned to a float produces -150.000000
-150 divided by 100 produces -1.000000
-150 divided by 100.0 produces -1.500000
(float) -150 divided by 100 produces -1.500000
```

在C语言中，如果我们把一个浮点数赋值给一个整型数，浮点数的小数部分将被截断。所以，在程序4.5中，当我们把f1的值赋给i1的时候，123.15的小数部分将被截去。只有它的整数部分，也就是123，保存在变量i1中。程序的第一行输出验证了这种情况。

把一个整型变量赋值给一个浮点变量不会丢失任何精度，系统将自动对其进行转换。程序的第二行输出验证了这一点：i2的值被完整地保存到f1中。

程序的下面两行输出演示了在C语言中编写算术表达式时必须牢记的两个规则。第一个是关于我们前面讨论过的整数算术的。无论何时，只要运算式中的两个操作数都是整数（也包括short、unsigned、long、long long），其运算就遵循整数算术规则：两个数相除时，其余数将被丢弃。即使我们将运算结果保存在一个浮点数中也是这样（正如我们在

程序中所作的那样)。因此,当我们使用*i2*除以100,系统将执行整数除法,也就是用-150除以100,结果是-1。浮点变量*f1*中保存的结果正是-1。

程序4.5的下一个运算是用整型数除以浮点常数。在C语言中,如果一个运算的两个操作数中有一个是浮点数——不论是浮点常数还是浮点变量,这个运算都被看作是浮点运算。因此,当我们使用*i2*除以100.0的时候,系统执行浮点数除法,得出的结果将是-1.5,这个结果保存在变量*f1*中。

类型转换操作符

程序4.5中的最后一个除法运算语句如下:

```
f2 = (float)i2 / 100;          // Type cast operator
```

这个语句中使用了类型转换操作符。这个类型转换操作符使得C语言在对该表达式求值的时候,临时将*i2*看作一个整型变量。这个操作符并不会影响*i2*中实际存放的值。读者可以把它和其他的单目操作符对比。正如表达式-a不会影响变量*a*中存放的数值,表达式(float)*a*对于变量*a*本身也没有影响。

除了单目加法和单目减法操作符之外,类型转换操作符的优先级高于其他任何算术操作符。当然了,我们总是可以使用括号控制表达式的求值过程,使其按照我们需要的顺序进行。

下面又是一个使用类型转换操作符的例子,语句

```
(int) 29.55 + (int) 21.99
```

实际上相当于

```
29 + 21
```

因为将一个浮点数转换为整型数,将会截断其小数部分。下面的语句

```
(float) 6 / (float) 4
```

其结果是1.5,它相当于下面的语句

```
(float) 6 / 4
```

结合运算和赋值: 运算赋值操作符

在C语言中,我们可以使用格式“op=”将赋值操作和运算操作结合起来。

在这个格式中，`op`代表任何算术操作符，包括`+`、`-`、`*`、`/`和`%`。另外，`op`也可以是我们后面将要讨论的位操作符。

让我们研究一下下面的语句

```
count += 10;
```

在这里，加等操作符（`+=`）的作用是将其左面的操作数和右面的操作数加起来，然后将其结果再存放在左面的操作数中。因此，这个语句实际上与下面的语句等价

```
count = count + 10
```

下面的语句

```
count -= 5
```

实际上就是用`count`中的值减去5，然后将结果再存回到`count`中。

下面这个表达式稍微复杂一些

```
a /= b + c
```

它的意思是用`a`除以右面运算的结果——也就是`b+c`的值，然后再将结果存回变量`a`中。因为加法操作符的优先级高于`/=`操作符，所以C语言首先计算`b + c`的值。实际上，除了逗号操作符以外（该操作符与运算赋值操作符的优先级相同），运算赋值操作符的优先级比其他任何操作符的优先级都要低。

上面这个语句实际上等价于

```
a = a / (b + c)
```

使用运算赋值操作符的主要原因有三个：1.这种写法可以简化程序，因为我们不必在操作符右面再次书写左面的操作数符号；2.这样的写法可读性较好；3.使用这种操作符编写的程序某些情况下，执行将会更快一些，因为编译器生成的用于求值的指令较少。（译者注：实际上第三个理由现在已经基本不存在了，现代的编译器在代码生成优化方面已经相当成熟。）

类型 `_Complex` 和 `_Imaginary`

在结束本章之前，我们还要提一下两种用于处理复数和虚数的数据类型：`_Complex`和`_Imaginary`。

对于编译器来说，对这两个数据类型的支持不是必须的³。如果想要了解这两个类型的更多信息，请参阅附录A。

³ 在写作本书的时候，gcc 3.3 还未完全支持这些数据类型。

练习

1. 输入并运行本章的五个程序，并将运行结果与书中列出的结果对比。
2. 下面的符号哪些是合法的变量名，为什么？

Int	char	6_05
Calloc	Xx	alpha_beta_routine
floating	_1312	z
ReInitialize	_	A\$

3. 下面哪些是不合法的常量，为什么？

123.456	0x10.5	0X0G1
0001	0xFFFF	123L
0Xab05	0L	-597.25
123.5e2	.0001	+12
98.6F	98.7U	17777s
0996	-12E-12	07777
1234uL	1.2Fe-7	15,000
1.234L	197u	100U
0XABCDEFL	0xabcu	+123

4. 利用下面给出的公式，编写一个程序，将华氏 (F) 27度换算为摄氏度 (C)。

$$C = (F - 32) / 1.8$$

5. 下面程序的输出是什么？

```
#include <stdio.h>

int main (void)
{
    char c, d;

    c = 'd';
    d = c;
    printf ("d = %c\n", d);

    return 0;
}
```

6. 书写程序，计算下面多项式的值

$$3x^3 - 5x^2 + 6, \quad x = 2.55$$

7. 编写程序，计算下面表达式的值，并用指数形式显示结果。

$$(3.31 * 10^{-8} * 2.01 * 10^{-7} / (7.16 * 10^{-6} + 2.01 * 10^{-8}))$$

8. 给出整数*i*和*j*，为了计算刚好比*i*大的下一个能够整除*j*的整数，我们可以使用下面的公式

$$\text{Next_multiple} = i + j - i \% j$$

例如，将256天圆整为能够包含整数个星期的天数，我们可以使用上面的公式，其中*i*=256，*j*=7。

$$\begin{aligned}\text{Next_multiple} &= 256 + 7 - 256 \% 7 \\ &= 256 + 7 - 4 \\ &= 259\end{aligned}$$

书写一个程序，计算当*i*和*j*等于下面给出的值时，下一个能够整除*j*的数。

<i>i</i>	<i>j</i>
365	7
12,258	23
996	4

Program Looping

循环

如果我们试着用15个点摆成一个三角形，最终的结果有可能是下面的样子

```
      .  
     . .  
    . . .  
   . . . .  
  . . . . .
```

三角形的第一行包含一个点，第二行包含两个点，以此类推。一般说来，摆出有 n 行的三角形所需要的点数是从1到 n 所有整数的和。这些数被称为三角形数。如果从1开始算起，那么第四个三角形数就是从1到4这四个整数的和，即 $(1+2+3+4)=10$ 。

现在，让我们来写一个程序，计算第八个三角形数是多少，并将其在终端中显示出来。虽然通过心算，我们可以很容易给出答案，但是为了适应这里的讨论需要，我们假定要用C语言写一个程序来完成这个工作。这个程序就是5.1。

程序5.1中使用的技术，对于计算较小的三角形数工作的很好。但是假定我们需要计算第200个三角形数的值，那应该怎么办呢？如果还是按照程序5.1中那样编写，无疑是非常繁琐的。幸运的是，C语言为我们提供了简便的方法。

程序5.1 计算第八个三角形数

```
// Program to calculate the eighth triangular number  
  
#include <stdio.h>
```

程序5.1 续

```
int main (void)
{
    int triangularNumber;

    triangularNumber = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;

    printf ("The eighth triangular number is %i\n",
            triangularNumber);

    return 0;
}
```

程序5.1 输出

The eighth triangular number is 36

能够重复执行一组指令，是计算机的一个基本特点，这种特点被称为循环。如果没有循环的话，某些程序可能需要数千条甚至数万条指令来完成，而使用循环就可以相当简明地编写这类程序。C语言提供了三种语句用于循环，它们分别是for语句、while语句和do语句。本章将详细介绍这些语句。

for 语句

我们首先研究for语句。程序5.2的作用是计算第200个三角形数。读者看看能不能找出for语句的工作方式。

程序5.2 计算第200个三角形数

```
/* Program to calculate the 200th triangular number
   Introduction of the for statement */

#include <stdio.h>

int main (void)
{
    int n, triangularNumber;

    triangularNumber = 0;
```

程序5.2 续

```
for ( n = 1; n <= 200; n = n + 1 )
    triangularNumber = triangularNumber + n;

printf ("The 200th triangular number is %i\n", triangularNumber);

return 0;
}
```

程序5.2 输出

The 200th triangular number is 20100

我们下面来解释程序5.2。程序中用于计算三角形数的算法和程序5.1中计算第8个三角形数的算法是相同的——从整数1加到整数200。但是，使用for语句以后，我们就不必在程序中一个一个地写出从1到200这200个整数。从某种意义上来说，循环自动为我们生成了这些数字。

for语句的一般形式如下：

```
for (初始化表达式; 循环条件; 循环表达式)
    程序代码
```

被括号括起来的三个表达式：初始化表达式、循环条件和循环表达式建立了循环运行的条件，紧跟在for后面的语句组成了循环体，它可以是任何合法的C语言语句。这个语句执行的次数由前面建立的循环条件决定。

建立循环条件的第一个表达式是初始化表达式 用于循环开始之前对循环变量进行初始化。在程序5.2中，我们利用初始化表达式将变量n的值初始化为1。读者可以看到，在C语言中，赋值语句也是一个合法的表达式。

第二个表达式被称为循环条件，只要这个条件得到满足，循环体就会被执行。程序5.2中，循环条件是一个关系表达式

```
n <= 200
```

这个表达式的意思是判断n是不是小于或者等于200。其中的<=符号是C语言的一种关系操作符，它的意思是小于或者等于。关系操作符用于判断特定的条件是否成立。如果特定的条件成立的话，表达式的值就是真，否则，表达式的值就是假。

关系操作符

表5.1列出了C语言中所有的关系操作符。

表5.1 关系操作符

操作符	用途	示例
==	相等	count == 10
!=	不等	flag != DONE
<	小于	a < b
<=	小于或者等于	low <= high
>	大于	pointer > end_of_list
>=	大于或者等于	j >= 0

关系操作符的优先级低于算术操作符。例如，下面的表达式：

```
a < b + c
```

的求值顺序如下：

```
a < ( b + c )
```

如果a 小于 b + c 的和，这个表达式的值为TRUE，否则为FALSE。

我们要特别提醒读者注意，不要把相等关系操作符(==)和赋值操作符(=)搞混了。

下面的表达式

```
a == 2
```

用于判断a的值是否等于2。而下面的表达式

```
a = 2
```

则是把2赋给变量a。

在编写程序的时候，我们可以根据实际情况和个人偏好来选择使用哪些关系操作符。

例如下面的表达式

```
n <= 200
```

与下面的表达式

```
n < 201
```

它们的意思实际上是一样的。(译者注：这里假定n是整型变量)

在讲述了关系操作符的相关知识之后，让我们回到程序5.2。程序5.2循环体中的语句如下

```
triangularNumber = triangularNumber + n;
```

只要for语句中的条件判断语句结果为真，也就是说，n小于或者等于200，这个循环体语句就将被重复执行。该语句的作用是计算变量triangularNumber和变量n的和，并将结果存回变量triangularNumber中。

当循环条件不再满足后，程序将退出循环，转到for语句后面的语句执行。在程序5.2中，循环结束后，程序将执行后面的printf语句。

在for语句的括号中还有一个表达式循环条件，每次当循环体执行一遍后，这个表达式都被计算一次。在程序5.2中，这个表达式会将n的值增加1。因此，每次当n被加到变量triangularNumber中以后，n的值都将增加1。在循环执行中，n的值从1一直增加到201。

值得读者注意的是，n的最后一个值，201，并没有加到变量triangularNumber中。因为一旦循环条件没有得到满足，循环就结束了。

现在我们总结一下for语句的执行顺序：

1. 初始化表达式首先被求值。这个表达式常常用于初始化一个变量，该变量将被用在循环体中。这个变量通常被称为循环变量。循环变量的初始值通常是0或者1。
2. 下来，循环条件表达式将被求值。如果表达式中指定的条件不满足（表达式的值为假），循环将立即结束。程序将转到for语句后面的语句执行。
3. 如果条件满足，循环体将被执行。
4. 循环表达式被求值。这个表达式通常用于修改循环变量，比如加上1或者减去1。
5. 回到第2步。

需要提醒读者注意的是，循环条件表达式总是在循环体的执行之前被求值，即使第一次也如此。因此，如果一开始循环条件就不满足的话，循环体有可能一次都不执行。另外，千万不要在for语句的括号后面放置一个分号，那样将会立刻结束循环。

程序5.2在计算第200个三角形数的时候，实际上也计算出了所有前200个三角形数，因此我们可以把它们都打印出来。为了节省篇幅，我们假定只需要打印前10个三角形数。程序5.3完成了这个工作。

程序5.3 打印三角形数表

```
// Program to generate a table of triangular numbers

#include <stdio.h>

int main (void)
{
    int n, triangularNumber;
```

程序5.3 续

```

printf ("TABLE OF TRIANGULAR NUMBERS\n\n");
printf (" n      Sum from 1 to n\n");
printf ("---      -----\n");

triangularNumber = 0;

for ( n = 1; n <= 10; ++n ) {
    triangularNumber += n;
    printf (" %i      %i\n", n, triangularNumber);
}

return 0;
}
    
```

程序5.3 输出

TABLE OF TRIANGULAR NUMBERS

n	Sum from 1 to n
---	-----
1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

在程序中加上更多的printf语句以便在输出中表达更多的意思是个不错的主意。程序5.3中，前三个printf语句的主要作用就是为输出打印表头，说明每列输出的作用。请读者注意，在第一个printf语句中，有两个换行符。它不仅使随后的输出在新行上开始，而且还额外多插入了一个空行。

输出了表头信息之后，程序开始计算前10个三角形数。其中，变量n用来保存我们当前正在计算第几个三角形数，而变量triangularNumber用于保存计算的结果。

程序将循环变量n的值设为1，然后开始执行。我们前面讲过，紧跟在for语句后面的语句是被重复执行的循环体。但是如果我们想要重复执行一组语句，而不是一条语句，那

么该如何书写呢？我们可以把这组语句用大花括号括起来。系统就会把花括号中的所有语句当作单独的一块执行。一般说来，在C语言中，任何允许单条语句的地方，我们都可以使用以大花括号括起来的一组语句，或者称为语句块。

因此，在程序5.3中，把循环变量n加到变量triangularNumber上的那条语句和后面的printf语句，共同组成了for循环的循环体。请读者注意，在程序5.3中，循环体的所有语句都使用了缩进，这使得我们很容易看清楚那些语句是循环体。除此之外，程序员还可以使用其他编程风格，例如，一些程序员喜欢将循环语句写成下面的形式

```
for ( n = 1; n <= 10; ++n )
{
    triangularNumber += n;
    printf ( " %i      %i\n", n, triangularNumber);
}
```

这种风格将循环体的左花括号另起一行书写。读者可以根据喜好任意选择一种风格，这对于程序没有任何影响。

在第n-1个三角形数上加上n，就可以得出第n个三角形数。这里我们使用的加等操作符(+=)。我们在第4章“变量、数据类型和算术操作符”中介绍过这个操作符。实际上，下面的语句

```
triangularNumber += n;
```

相等于

```
triangularNumber = triangularNumber + n;
```

在第一次运行循环体的时候，“前一个”三角形数等于0，因此，当n等于1的时候，变量triangularNumber的值与n的值相等，也就是1。我们在输出的格式化字符串中加入了适量的空格，这样最终的输出结果就可以和前面输出的表头上下对齐。

在循环体执行完以后，程序下来接着执行循环表达式。这个循环表达最初看上去有些奇怪，读者可能以为我们这里的循环表达式应该是

```
n = n + 1
```

而看上去有点好笑的

```
++n
```

似乎一个拼写错误。

实际上，++n是一个完全合法的C语言表达式。在这里我们要向读者介绍C语言中一个新的操作符——自增操作符。这两个加号——自增操作符——是将其操作数的值增加1。因

为在程序中给某个变量增加1很常用，因此C语言专门引入了这个操作符。`++n`和 `n = n + 1` 的意思是完全一样的。虽然初看起来后面一个表达式的可读性要更好一些，实际上读者会很快适应新的操作符，并领会到其简洁之处。

当然了，任何拥有自增操作符的语言，都会定义相应的自减操作符，否则语言就会显得不完整。自减操作符由两个减号组成。下面的表达式

```
bean_counter = bean_count - 1
```

可以使用自减操作符改写为：

```
--bean_count
```

一些程序员喜欢将自增和自减操作符放到操作数的后面，例如 `n++` 或者 `bean_counter--`。这也是合法的，读者可以根据自己的喜好任意选择其中一种。

对齐输出

程序5.3中，有一小问题让我们感到不愉快，第十个三角形数的输出和前面的输出没有对齐。因为10这个数字占据了两个位置，而前面的1到9只需要一个位置，因此55被向右挤了一个位置。如果我们用下面的printf语句代替程序5.3中的printf语句，就可以解决这点小问题。

```
printf ("%2i          %i\n", n, triangularNumber);
```

为了验证这条语句确实能修正刚才的小问题，我们列出修改后的程序（不妨称作程序5.3A）的输出如下。

程序5.3A 输出

TABLE OF TRIANGULAR NUMBERS

n	Sum from 1 to n
---	-----
1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

在程序5.3A中，我们在printf的格式化输出符号前面加上了一个输出宽度限定数。字符串%2i不仅告诉printf函数我们要输出一个整形数，而且还告诉它我们希望输出的整数占的宽度为两列。如果某个整数输出的宽度不足两列，那么就在前面用空格补齐。这种输出格式称为右对齐。

因此，通过在格式化输出符号中指定输出宽度，我们可以保证n的输出最少占两列，从而使得triangularNumber一列实现了对齐。

如果某个数输出的宽度超过了输出宽度限定符中指定的数值，printf将会忽略该限定符，而按照该整数实际需要的宽度显示其数值。

输出宽度限定符也可以用于其它类型的数值，我们在本章随后的程序中将会很快看到这一点。

程序输入

程序5.2只能用来计算第200个三角形数。如果我们想要计算第50个或者第100个三角形数，我们必须重新修改程序，使得for循环执行适当的次数。同时，我们还要修改最后的printf语句以显示正确的信息。

为了能够方便的计算某个三角形数，我们可以让程序主动询问用户需要计算哪个三角形数。当用户输入了需要计算的三角形数的编号，程序就可以提供正确的答案了。用C语言中的库函数scanf就可以做到这点。scanf函数在概念上与printf函数非常相似。printf用于将变量的值显示在终端上，而scanf允许我们给程序输入数值。程序5.4首先询问用户需要计算哪一个三角形数，然后计算该数并打印结果。

程序5.4 要求用户输入

```
#include <stdio.h>

int main (void)
{
    int n, number, triangularNumber;

    printf ("What triangular number do you want? ");
    scanf ("%i", &number);

    triangularNumber = 0;

    for ( n = 1; n <= number; ++n )
        triangularNumber += n;
```

程序5.4 续

```
printf ("Triangular number %i is %i\n", number, triangularNumber);

return 0;
}
```

在程序5.4的输出中，用户输入的数字（100）用粗体显示，以和程序的输出相区别。

程序5.4 输出

```
What triangular number do you want? 100
Triangular number 100 is 5050
```

从上面的显示我们看出，用户输入的数字是100。程序接着计算出第100个三角形数的值，并将其显示在终端上。用户也可以输入10或者30，或者任何需要计算的三角形数。

程序5.4的第一个printf函数用于提示用户输入一个数值。总的说来，当我们的程序需要输入时，提醒用户总是一件好事。在显示提示信息之后，我们的程序调用了scanf函数。传递给scanf函数的第一个参数是一个格式化字符串，它和用于printf函数的格式化字符串非常相似。在这里，格式化字符串不是用来表示我们要如何输出变量，而是我们要按怎样的格式读入数值。和printf函数中的约定一样，%i用于读入一个整数。

第二个参数用于告诉scanf函数，用户输入的数值保存在什么地方。在变量名前面的&符号是必须的。读者不必为这个符号的作用感到困惑，我们将在第11章“指针”非常详细地讨论这个符号（实际上是操作符）的作用。现在我们只要记住，在传递scanf函数作为参数的变量名之前，都必须加一个&符号就可以了。如果忘记了这一点，程序运行的结果将无法预料，甚至有可能意外崩溃。

在经过前面的讨论之后，读者应该已经明白，程序5.4中的scanf函数实际上从用户输入中读取一个整数，并将其存放在变量number中。这个数值就是用户想要计算的三角形数的编号。

当用户输入了这个数字之后（用户通常还需要按下键盘上的Enter键或者Return键以通知程序输入已经完成），程序就开始计算用户请求的三角形数。计算的方法与程序5.2完全相同，只不过这一次我们不再使用200作为循环的终点，而是使用用户输入的数字number。

当计算出该三角形数之后，程序打印该数字并结束运行。

嵌套的 for 循环

程序5.4可以让用户计算任何想要的三角形数，但是，如果用户想要计算5个三角形数，那么他就需要运行这个程序5次，每次输入一个需要计算的数。

如果我们的目的在于学习和使用C语言的话，那么就有另外一种更加有意思的办法来完成上面的任务。我们可以在程序中加入一个循环，这个循环运行5次，每次都把输入和计算三角形数的过程执行一遍。读者已经知道，for语言可以辅助我们完成这类循环工作，程序5.5及其输出演示了如何做到这一点。

程序5.5 使用嵌套循环

```
#include <stdio.h>

int main (void)
{
    int n, number, triangularNumber, counter;

    for ( counter = 1; counter <= 5; ++counter ) {
        printf ("What triangular number do you want? ");
        scanf ("%i", &number);

        triangularNumber = 0;

        for ( n = 1; n <= number; ++n )
            triangularNumber += n;

        printf("Triangular    number    %i    is    %i\n\n",    number,
            triangularNumber);
    }

    return 0;
}
```

程序5.5 输出

```
What triangular number do you want? 12
Triangular number 12 is 78

What triangular number do you want? 25
Triangular number 25 is 325

What triangular number do you want? 50
Triangular number 50 is 1275
```

程序5.5 续

```

What triangular number do you want? 75
Triangular number 75 is 2850

What triangular number do you want? 83
Triangular number 83 is 3486
    
```

程序5.5中使用了两层for语句。外层for的语句是：

```
for ( counter = 1; counter <=5; ++counter )
```

因为counter的值最初被设置为1，每次循环都增加1，直到counter不再小于或者等于5，这样这个外层循环需要执行5次。

与前面给出的程序不同的是，程序5.5中的其他地方都不再使用循环变量counter，它仅仅起到循环计数的作用。但是，由于它是一个变量，因此必须在程序中声明。

正像大括号显示的那样，这个外层循环的语句几乎包含了程序剩余的所有语句。如果按照下面的方式对这个程序进行抽象总结的话，读者也许能够更容易理解一些。

循环5次

```

{
    获得用户输入的数字

    计算相应的三角形数

    显示结果
}
    
```

在上面抽象总结中，“计算相应的三角形数”对应着实际程序中将变量triangularNumber赋值为0和利用循环计算三角形数这段语句。因此，我们实际上看到一个for循环作为另外一个for循环的循环体。这在C语言中是完全合法的，实际上，C语言允许最多127层循环。

在处理更复杂的程序——例如嵌套的for语句时，正确地缩进语句显得尤为重要。根据缩进，我们很容易判断某个for循环到底包含哪些语句。（如果读者想知道没有很好排版的源程序有多么难懂，可以看看本章末尾的练习5。）

for 循环的变体

for语句在语法上有几种变体形式。比如，读者可能想要在循环开始之前初始化几个变量，或者在每次循环中要对几个表达式求值，就可以采用这些形式。

多个表达式

在for语句括号中的每一个地方，我们都可以使用用逗号隔开的多个表达式。例如，下面的for语句：

```
for ( int i = 0, j = 0; i < 10; ++i )  
...
```

在循环开始之前，将变量*i*和*j*的值都赋为0。两个表达式 *i* = 0 和 *j* = 0之间用逗号隔开，他们都被看作是for语句初始化表达式init_expression 的一部分。再来看另外一个例子：

```
for ( i = 0, j = 100; i < 10; ++i, j = j - 10 )  
...
```

在循环中使用了两个循环变量*i*和*j*，在循环开始之前，*i*被初始化为0，*j*被初始化为100。每次循环体执行完成以后，*i*的值增加1，而*j*的值减少10。

省略某些表达式

有的时候我们想要在for语句的初始化或者循环表达式中使用多个表达式，而有的时候恰恰相反，我们可能想要省略某个表达式。这时我们可以不在相应的位置写上表达式，而仅仅用分号标出其位置。在for循环中最常被省略的就是它的初始化表达式，这种时候，我们只需要保留相应的分号，而不必写出初始化表达式。下面就是一个例子：

```
for ( ; j != 100; ++j )  
...
```

如果在进入循环之前的某个地方，我们已经给循环变量*j* 赋了初值，就可以使用上面的语句。

如果在for语句中我们省略了循环条件表达式，那么该循环就是一个无限循环，理论上它将永远执行下去。如果我们在循环体中提供了其他退出循环的方式，如return、break或者goto语句（这些语句我们将在本书的其他地方讨论），那么就可以使用这种形式的循环。

声明变量

在for循环的初始化表达式中，我们也可以声明变量，声明的方式和普通变量相同。下面的for循环示例声明了一个整形循环变量counter，并将它赋以初值1。

```
for ( int counter = 1; counter <= 5; ++counter)
```

这个变量只在for循环语句的范围内有效（它也被称为局部变量），在循环之外，我们不能使用它。下面是另外一个例子：

```
for ( int n = 1, triangularNumber = 0; n <= 200; ++n )  
    triangularNumber += n;
```

这个例子中的for语句定义了两个变量并分别赋以初值。

while 语句

利用C语言提供的while语句，我们可以编写出更复杂的循环来。while语句的常用形式如下：

```
while ( 表达式 )  
    语句
```

程序在执行的时候，首先对括号中的表达式求值，如果结果为真，程序就执行下面的语句。如果有多个语句需要执行，可以用大括号把这些语句括起来。在语句执行完成之后，程序对括号中的表达式再次求值。如果结果是真，那么就接着执行后面的循环体，这个过程不断重复，直到表达式的值为假。循环执行完后，程序将接着执行循环体后面的语句。

为了演示while语句的用法，程序5.6建立了一个while循环，并把1—5的数字打印出来。

程序5.6 介绍while语句

```
// Program to introduce the while statement  
  
#include <stdio.h>  
  
int main (void)  
{  
    int count = 1;  
  
    while ( count <= 5 ) {  
        printf ("%i\n", count);  
        ++count;  
    }  
  
    return 0;  
}
```


程序5.6 输出

```
1  
2  
3  
4  
5
```

在程序的开始，我们将变量count的值初始化为1，然后程序开始执行while循环。因为count的值小于5，所以程序开始执行while语句的循环体。大括号中语句全部属于while语句的循环体，这个循环体不但包含常规的for语句循环体完成的工作，还要负责给循环变量的值加1。从程序的输出我们可以看到，while语句的循环体刚好执行了5次，循环结束后，变量count的值是6。

读者可能已经意识到，刚才的程序也很容易使用for语句来编写。实际上，一个for语句总是能够翻译成一个对应的while语句，反过来也一样。例如，for语句的一般形式

```
for (初始化表达式; 循环条件; 循环表达式)  
    程序代码
```

可以使用while语句表示为如下等价的形式

```
初始化表达式;  
while (循环条件) {  
    程序代码  
    循环表达式;  
}
```

当读者对于while语句有了更多的了解以后，就能更好的判断什么情况下应该使用for语句，而什么时候应该使用while语句。

一般说来，如果循环的次数预先已经知道，那么我们首选应该使用for语句。另外，如果初始化表达式、循环表达式和循环条件使用了同一个变量，很可能也应该使用for语句。

下面又是一个使用while语句的例子。在这个程序中，我们计算两个整数的最大公约数。所谓最大公约数(gcd)，就是两个数都能够整除的最大的数。例如5就是10和15的最大公约数，因为5是10和15都能整除的那些数中最大的。

有一个专门的算法，可以用于计算任意两个整数的最大公约数。这个算法是由阿基米德在大约公元前300年发现的。这个算法可以描述如下。

问题：找出两个非负整数 u 和 v 的最大公约数

第一步：如果 v 等于0，那么 u 就是最大公约数。

第二步：执行下面的算式： $temp = u \% v, u = v, v = temp$ ，然后回到第一步。

在这里，我们不打算花时间去研究为什么这个算法能够正确的工作，我们就相信它是正确的好了。我们将花更多的精力来研究如何编程实现这个算法，而不是分析它的工作原理。

既然我们已经找到了计算两个整数最大公约数的办法，并将其表达为算法步骤，那么编写相应的计算机程序则是一件简单的事情。对于算法稍作分析，我们就可以知道，只要 v 的值不等于0，那么算法的第二步就需要不断的执行。一旦认识到这一点，我们很容易使用C语言程序5.7的while循环语句来表达这个算法。

程序5.7 计算最大公约数

```
/* Program to find the greatest common divisor
   of two nonnegative integer values */

#include <stdio.h>

int main (void)
{
    int u, v, temp;

    printf ("Please type in two nonnegative integers.\n");
    scanf ("%i%i", &u, &v);

    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }

    printf ("Their greatest common divisor is %i\n", u);

    return 0;
}
```


程序5.7 输出

```
Please type in two nonnegative integers.  
150 35  
Their greatest common divisor is 5
```

程序5.7 输出（再次运行）

```
Please type in two nonnegative integers.  
1026 405  
Their greatest common divisor is 27
```

我们在调用scanf函数时，其格式化输入字符串中使用了两个%i，用于从键盘读入两个整数。第一个整数保存在变量u中，第二个保存在变量v中。在实际输入数据的时候，我们可以用一个或者多个空格，或者换行符来分隔这两个整数。

当程序读入了用户从键盘输入的两个整数之后，程序就进入一个循环，该循环用于计算这两个整数的最大公约数。当循环结束以后，变量u中就保存着这两个整数的最大公约数。程序随后将该结果打印出来，并附上适当的说明信息。

程序5.8给出了使用while循环语句的另外一个例子，这个程序从键盘读入一个用户输入的数字，然后将该数字的每一位数字逆序排列。例如，如果用户输入1234，程序输出的结果就是4321。

为了编写这个程序，我们首先需要找出解决上面问题的算法。很多时候，对于在手工状态下完成某项任务的方式进行一点分析，我们就可以得出解决该问题相应的计算机算法。为了逆序排列某个数的每一位数字，我们只需要从右向左连续的读出该数就可以了。同样，我们也可以编写一个程序，从最右边开始，逐个找出某个数字的每一位数。每找出一位数字，我们就可以将其作为逆序数字的一位输出。

将一个整数除以10得出的余数就是该整数最右面一位的数字。例如，1234除以10，余数是4，这个数字刚好就是1234最右面的一位数。这个数字也是1234逆序排列以后的第一位数字。在C语言中，计算两个数相除以后的余数，可以使用%操作符。只要记住整数除法的规则，我们可以继续使用上面的方法接着找出其他位上的数字。例如，1234/10的结果是123，123%10的结果是3，这个数字就是逆序排列的第二位数字。

上面的过程可以一直重复下去，直到我们找出所有位上的数字。当最后一次除法给出的结果为0时，我们就已经找出了给定整数的每一位上的数字。

程序5.8 逆序输出某个数字

```
// Program to reverse the digits of a number

#include <stdio.h>

int main (void)
{
    int number, right_digit;

    printf ("Enter your number.\n");
    scanf ("%i", &number);

    while ( number != 0 ) {
        right_digit = number % 10;
        printf ("%i", right_digit);
        number = number / 10;
    }

    printf ("\n");

    return 0;
}
```

程序5.8 输出

```
Enter your number.
13579
97531
```

在程序中，我们每计算出一位数字，就把它显示出来。这里需要提醒读者注意的是，在while循环体中的printf语句中，我们没有输出换行符，这样所有输出的数字就会显示在同一行上。我们在程序的最后使用了printf函数，该函数仅仅输出一个换行符，这样，在程序输出的所有的数字之后，光标将被移到下一行的开始。

do 语句

截止现在，本章已经介绍了两种循环语句。这些循环语句在执行循环体之前，都会检查特定的循环条件。因此如果一开始程序就不满足循环条件的话，循环体实际上一次也不执行。在编写程序的时候，有时我们需要在循环体结束的时候，而不是在循环开始的地方检验循环条件。为此处理这种情况，C语言设计了专门的语法结构，这就是do语句。do语句的一般格式如下：


```
do  
    程序语句  
while( 循环表达式 );
```

do语句的执行顺序如下：程序首先执行程序语句所代表的语句，然后程序对表达式循环表达式求值。如果求值结果为真，程序将重新执行程序语句。只要循环表达式的值是真，程序就将重复执行程序语句。当循环条件的值变成假以后，程序将退出循环，接着执行后面的语句。

我们可以简单的将do语句看作while语句的循环条件移到循环体之后所形成的一种变形形式。do语句的特点在于，该语句可以保证循环体最少将被执行一次。

在程序5.8中，我们使用while循环逆序输出整数。我们可以试着重新运行该程序，这次我们的输入不是13579，而是数字0。当输入的整数是0时，while循环体实际上一次也不执行，我们程序的输出仅仅是一个空行（也就是最后的printf语句的输出）。如果我们使用do语句而不是while语句，就可以保证循环体最少执行一次。这样，无论用户输入什么数字，我们都将最少显示一位数字。程序5.9是修订过的程序。

程序5.9 修订过的逆序输出程序

```
// Program to reverse the digits of a number  
  
#include <stdio.h>  
  
int main ()  
{  
  
    int number, right_digit;  
  
    printf ("Enter your number.\n");  
    scanf ("%i", &number);  
  
    do {  
        right_digit = number % 10;  
        printf ("%i", right_digit);  
        number = number / 10;  
    }  
    while ( number != 0 );  
  
    printf ("\n");  
  
    return 0;  
}
```

程序5.9 输出

```
Enter your number.  
13579  
97531
```

程序5.9 输出 (再次运行)

```
Enter your number.  
0  
0
```

我们可以看到，程序5.9可以正确处理输入为0的情况。

break 语句

在某些情况下，我们需要在某个条件成立的时候立刻退出循环，例如，当出现了某些错误，或者输入数据意外结束等。在这种情况下我们可以使用break语句。

使用break语句可以使程序立刻退出正在执行的循环，无论循环语句是for、while还是do。实际上，当程序执行遇到break语句时，循环将被立即中止，break后面的语句将被跳过。程序接着循环后面的语句开始执行。

如果在嵌套循环中使用break语句，那么只有最里面的循环中止执行。

break语句的格式非常简单，只需要在关键字break后面跟上分号就可以了。如下所示：

```
break;
```

continue 语句

continue语句与break语句有些类似，但是它并不中止循环的执行。实际上，正像这个语句使用的单词所示意的那样，continue语句将使循环继续执行。当程序执行遇到continue语句时，该语句后面的所有语句都将被跳过。循环的其他部分仍将照常执行。

continue语句最常见的用途是：当满足某些条件的时候，跳过循环体中的部分语句，反之则执行这些语句。continue语句的格式如下：

```
continue;
```

作为初学者，除非对于这两个语句非常熟悉，否则读者应该避免在程序中使用break和continue。这两个语句很容易被滥用，导致编写出来的程序很难阅读和调试。

到此为止，读者已经学习了C语言所有基本的循环语法，我们接下来将要学习另一类语句，这些语句让我们能够在执行程序的时候进行某些判断。第6章“进行判断”将详细介绍这些语句。在继续学习下一章之前，请读者完成下面的习题，以巩固本章学到的有关循环的知识。

练习

1. 输入并运行本章给出的9个程序。将程序的输出结果与书中给出的结果进行对比。
2. 编写程序计算从1到10这十个数的平方，并将结果打印成表格。记得要输出适当的标题行。
3. 对于任意的整数 n ，我们也可以通过下面的公式计算三角形数：

$$\text{triangularNumber} = n (n + 1) / 2$$

例如，计算第10个三角形数，我们可以用10代替上面公式中的 n ，并得出结果55。编写一个程序，使用上面的公式，计算从5到50之间间隔为5的所有数字（也就是5, 10, 15, ..., 50）的三角形数。

4. n 的阶乘（可以写作 $n!$ ），是从1到 n 所有数字相乘的积。例如，5的阶乘计算如下

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

编写一个程序，计算1到10的阶乘，并打印成表格。

5. 下面程序是完全合法的C语言程序，但是在编写的时候没有注意源文件的格式。读者可以注意到，这个程序很难阅读（实际上，我们还可以让这个程序变得更难阅读！）

参照本章中给出的程序的格式，重新编排这个程序的格式。输入该程序并运行之。

```
#include <stdio.h>
int main(void){
    int n,two_to_the_n;
    printf("TABLE OF POWERS OF TWO\n\n");
    printf(" n 2 to the n\n");
    printf("---\n");
    two_to_the_n=1;
    for(n=0;n<=10;++n){
        printf("%2i %i\n",n,two_to_the_n); two_to_the_n*=2;}
    return 0;}
```

6. 在格式化输出符号的宽度限定符前面使用一个减号 (-), 可以使得printf语句的输出左对齐。用下面的printf语句替换程序5.2中的相应语句, 运行该程序, 将两个程序的输出进行对比。

7. 格式化输出符号的宽度限定符前面的小数点 (.) 有着特殊的作用。输入下面的程序, 编译并运行它, 看看读者能不能观察出小数点的作用。实验时请试着输入不同的 n 值。

```
#include <stdio.h>

int main (void)
{
    int dollars, cents, count;

    for ( count = 1; count <= 10; ++count ) {
        printf ("Enter dollars: ");
        scanf ("%i", &dollars);
        printf ("Enter cents: ");
        scanf ("%i", &cents);
        printf ("$$%i.%2i\n\n", dollars, cents);
    }
    return 0;
}
```

8. 程序5.5仅仅允许我们计算5个三角形数。请读者对于该程序进行修改, 使得用户可以自己输入想要计算的三角形数的个数。

9. 使用while语句重写程序5.2到5.5。运行每一个改写过的程序, 确保修改是正确的。

10. 如果在程序5.8中输入一个负数将会发生什么情况。请读者试一试。

11. 编写一个程序, 计算某个整数所有的位上的数字的和。例如, 数字2155所有位上数字的和是 $2 + 1 + 5 + 5$, 即13。程序应该允许用户输入任意的整数。

Making Decisions

进行判断

在第5章“使用循环”中，我们知道计算机的一个基本功能是循环执行一组重复的指令，它的另外一个基本功能就是进行判断。我们已经在前面一章中的循环结束条件中看到了计算机如何进行判断。如果没有循环结束条件的话，我们就没有办法退出循环，而只能永远（在理论上）重复执行一组语句。实际上，这类循环常常被称为无穷循环。

除了循环结束条件以外，C语言还有其他几类用于条件判断的语句。我们将在本章介绍这些语句，它们分别是：

- if语句
- switch语句
- 条件操作符

if 语句

if语句是C语言中通用的if判断语句。该语句的一般形式如下：

```
if ( 表达式 )  
    程序代码
```

读者可以试着将我们日常用语中的句子“如果天不下雨，那么我就去游泳”翻译为C语言中的if语句。翻译的结果类似于下面的形式：

```
if ( 天不下雨 )  
    我去游泳
```

如果某个条件成立，我们需要执行一条语句（或者用大花括号括起来的一组语句）时，就可以使用if语句。例如“如果天不下雨，那么我就去游泳”。类似的，在下面的语句中

```
if ( count > COUNT_LIMIT )  
    printf ("Count limit exceeded\n");
```

如果变量count中保存的值大于COUNT_LIMIT，printf语句就被执行，否则，该语句将被跳过。

下面我们举个实际的例子，以便读者能够更好的理解if语句的使用。假定我们要编写一个程序，从终端接受用户输入的整数，然后计算这个数的绝对值。计算绝对值的方法非常简单：如果一个数字小于0，那么就在它前面加个负号，否则不做处理。前面的叙述中，“如果一个数字小于0”这句话意味着我们的程序需要进行某些判断。我们可以使用if语句完成这项工作，如程序6.1所示。

程序6.1 计算某个整数的绝对值

```
// Program to calculate the absolute value of an integer  
int main (void)  
{  
    int number;  
  
    printf ("Type in your number: ");  
    scanf ("%i", &number);  
  
    if ( number < 0 )  
        number = -number;  
  
    printf ("The absolute value is %i\n", number);  
  
    return 0;  
}
```

程序6.1 输出

```
Type in your number: -100  
The absolute value is 100
```

程序6.1 输出（再次运行）

```
Type in your number: 2000  
The absolute value is 2000
```


我们将程序6.1运行了两遍，以验证该程序能够正确地处理正数和负数两种情况。当然，为了确保该程序的正确性，我们还可以再运行几遍程序。但是通过上面的运行，我们起码能够确信，对于条件判断的两种情况，程序都能正确处理。

程序6.1首先输出适当的信息，提示用户输入一个整数。接下来程序读入用户输入的整数，并将其保存在变量number中。接下来程序检验number是否小于0，如果number确实小于0，那么下面的语句（也就是计算number相反数的这条语句）将被执行。如果number中的数不满足小于0的条件（也就是大于或者等于0），这条语句将被自动跳过（如果这个数已经是非负数，那么就不需要在前面加上负号了）。程序最后打印出number的绝对值并结束运行。

下面我们来看看程序6.2如何使用if语句。假定我们有一组考试成绩，需要计算其平均值。除此之外，我们还要统计出其中不及格的成绩的个数。对于这个问题，我们假定小于65分的成绩是不及格的。

为了计算不及格成绩的个数，我们必须判断每个成绩是及格的还是不及格的，我们同样使用if语句来完成这项工作。

程序6.2 计算一组成绩的平均值，并统计不及格分数的个数

```
/* Program to calculate the average of a set of grades and count
   the number of failing test grades */

#include <stdio.h>

int main (void)
{
    int numberOfGrades, i, grade;
    int gradeTotal = 0;
    int failureCount = 0;
    float average;

    printf ("How many grades will you be entering? ");
    scanf ("%i", &numberOfGrades);

    for ( i = 1; i <= numberOfGrades; ++i ) {
        printf ("Enter grade #%i: ", i);
        scanf ("%i", &grade);

        gradeTotal = gradeTotal + grade;
```

程序6.2 续

```
        if ( grade < 65 )  
            ++failureCount;  
    }  
  
    average = (float) gradeTotal / numberOfGrades;  
  
    printf ("\nGrade average = %.2f\n", average);  
    printf ("Number of failures = %i\n", failureCount);  
  
    return 0;  
}
```

程序6.2 输出

```
How many grades will you be entering? 7  
Enter grade #1: 93  
Enter grade #2: 63  
Enter grade #3: 87  
Enter grade #4: 65  
Enter grade #5: 62  
Enter grade #6: 88  
Enter grade #7: 76  
  
Grade average = 76.29  
Number of failures = 2
```

我们用变量`gradeTotal`保存所有成绩的和, 这个变量在程序开始的时候被初始化为0。变量`failureCount`统计不及格的成绩的个数, 它也被初始化为0。保存平均成绩的变量`average`被声明为浮点变量, 因为成绩的平均数不一定是整数。

接下来, 程序要求用户输入成绩的总个数, 这个数保存在变量`numberOfGrades`中。程序随后使用循环读入并处理每个成绩。我们用变量`grade`保存单个成绩, 这个变量名很合适。

程序接着把每一个`grade`的值累计到变量`gradeTotal`中。随后, 我们检查这个数值是否及格, 如果没有及格, 就给变量`failureCount`的值增加1。整个循环体就执行完一遍, 接下来开始读入并处理下一个成绩。

当我们读入并合计了所有的成绩之后, 需要计算平均成绩。如果不加考虑的话, 读者可能会使用下面的语句

```
average = gradeTotal / numberOfGrades;
```


实际上，这个语句的计算结果将会丢弃平均成绩的小数部分。因为在C语言中，如果一个除法表达式的除数和被除数都是整数的话，计算机实际上按照整数除法的规则进行。

有两种方法可以解决这个问题。第一种是将变量`numberOfGrades`或者`gradeTotal`声明为浮点数，这样在上面的表达式中，最少有一个操作数是浮点数，C语言将使用浮点数的除法规则。这个解决办法有一点小问题，那就是在程序中变量`numberOfGrades`和`gradeTotal`只用来保存整数，将它们声明为浮点数会让阅读程序的人感到有些迷惑，一般来说，这个办法不是太好。

另外一种方法就是使用类型转换操作符将两个操作数中的任何一个强制转型为一个浮点数。我们在程序中采用了这种方法，将`gradeTotal`强制转型为`float`类型。在计算机实际执行这个操作的时候，会首先执行类型转换，然后再执行除法。这样实际上两个操作数中已经有一个是浮点数，因此除法将按照浮点数的除法规则进行。当然，我们就可以保留平均成绩的小数位了。

程序6.2在屏幕上输出平均成绩，这个浮点数显示时输出了两位小数。要做到这一点，我们可以在格式化输出字符串中，在代表浮点数输出的字符`f`（或者`e`）的前面，加上一个小数点，后面跟上需要输出的小数位数，这个小数点和后面的数字被称为输出精度修饰符。程序6.2中使用`.2`作为精度修饰符，因此程序的输出中就有两位小数。

最后，我们打印出不及格成绩的个数，然后结束程序的运行。

if – else 结构

如果有人问我们某个整数是一个奇数还是一个偶数，我们多半会观察一下这个数字的最后一位。如果最后一位数字是0、2、4、6或者8，那么这个数就是一个偶数，否则就是一个奇数。

如果使用计算机来回答这个问题，那么更方便的方法是用这个数除以2，如果能够整除的话，这个数就是一个偶数，否则是一个奇数。

我们知道求余操作符可以用来计算两个整数相除的余数。采用上面的算法，我们也可以很方便地使用这个操作符来判断某个数是奇数还是偶数。

程序6.3使用了这个方法判断整数奇偶性。它从终端读入用户输入的整数，根据计算的结果打印出适当的信息。

程序6.3 判断数字是奇数还是偶数

```
// Program to determine if a number is even or odd

#include <stdio.h>

int main (void)
{
    int number_to_test, remainder;

    printf ("Enter your number to be tested.: ");
    scanf ("%i", &number_to_test);

    remainder = number_to_test % 2;

    if ( remainder == 0 )
        printf ("The number is even.\n");

    if ( remainder != 0 )
        printf ("The number is odd.\n");

    return 0;
}
```

程序6.3 输出

```
Enter your number to be tested: 2455
The number is odd.
```

程序6.3 输出 (再次运行)

```
Enter your number to be tested: 1210
The number is even.
```

当用户输入数字以后，程序就计算出它除以2的余数。第一个if语句判断这个余数是否等于0，如果是的话，那么就输出文字“This number is even”（译者注：这个数是个偶数）。

第二个if语句判断余数是否不等于0，如果是这种情况，程序就打印消息说这个数是奇数。

实际上，如果第一个if语句的条件为真的话，第二个if语句的条件肯定是假，反过来也一样。因为一个数要么能够被2整除，要么不能，两者必居其一。

在书写程序的时候，这两种情况必居其一的情况经常出现，因此，几乎所有的编程语言都提供了专门的语法结构来处理这种情况。在C语言中，用if-else语法结构处理这种情况。该结构的一般形式如下：

```
if ( 表达式 )
    语句1
else
    语句2
```

if-else结构实际上是对if语句的一种扩展。如果表达式(expression)的值为真，那么就执行语句1，否则的话，就执行语句2。无论如何，语句1和语句2总要执行一个，但是不会都执行。

我们可以用if-else结构替换程序6.3中的两个if语句，这样写出来的程序要简单一些，更容易阅读和理解。程序6.4就是替换后的结果。

程序6.4 使用if-else结构修订后的判断数字奇偶性的程序

```
// Program to determine if a number is even or odd (Ver. 2)

#include <stdio.h>

int main ()
{
    int number_to_test, remainder;

    printf ("Enter your number to be tested: ");
    scanf ("%i", &number_to_test);

    remainder = number_to_test % 2;

    if ( remainder == 0 )
        printf ("The number is even.\n");
    else
        printf ("The number is odd.\n");

    return 0;
}
```

程序6.4 输出

```
Enter your number to be tested: 1234
The number is even.
```

程序6.4 输出（再次运行）

```
Enter your number to be tested: 6551
The number is odd.
```

这里再次提醒读者注意，两个等号(==)用来判断两个操作数是否相等，一个等号(=)则是赋值操作符。如果读者在编写程序的时候疏忽了这一点，比如在if的条件判断表达式中用混了，那麻烦可就大了。

复合关系表达式

直到现在为止，我们在if语句中使用的条件判断都只是简单地比较两个数。在程序6.1中，把数字number的值和0比较；在程序6.2中，把变量grade和65比较。有的时候，我们需要使用更复杂的条件，比如在程序6.2中，不是要统计不及格的成绩的个数，而是要统计位于70和79这个范围之间的成绩。这时，我们不是把grade的值和一个边界值比较，而是和两个边界值比较，这样才能知道这个数值是否在70和79之间。

C语言也可以处理这种类型的条件判断，就是用复合关系表达式。所谓复合条件表达式，就是使用逻辑与（AND）或者逻辑或（OR）操作符连接起来的多个简单条件判断表达式。逻辑与的符号是&&，逻辑或的符号是||（两条竖线）。下面是一个使用复合关系表达式的例子：

```
if ( grade >= 70 && grade <= 79 )
    ++grades_70_to_79;
```

这个语句判断grade的值是否在70和79之间，如果是的话，将变量grades_70_to_79的值增加1。与之类似，下面的语句

```
if ( index < 0 || index > 99 )
    printf ("Error - index out of range\n");
```

如果变量index的值小于0或者大于99，printf语句就会打印出警告信息。

我们可以使用复合关系操作符（&&和||）组成很复杂的复合关系表达式。C语言在表达式语法方面是非常灵活的。在实践中，人们常常滥用这种灵活性。实际上，简单的表达式总是更容易阅读，也更容易测试。

在书写复合关系表达式的时候，我们鼓励读者在适当的地方使用括号，这样可以增加程序的可读性，也能够避免因为弄错操作符的优先级而带来的麻烦。读者也可以使用空格使程序显得清楚一些，比如在复合关系操作符（&&和||）的周围放置额外的空格，这样就可以把操作符本身和它们连接的表达式清楚区分开来。

为了进一步说明复合关系表达式的使用，我们下面再给出一个实际的程序，该程序用来计算某个年份是否是闰年。一般来说，能够被4整除的年份是闰年。读者也许还不知道，如果一个年份能够被100整除的话，那么它还必须被400整除，才是闰年。

读者可以试着想一下，如何为上面的判断建立一个复合关系表达式。首先，我们可以先计算出年份除以4、100和400的余数，分别保存在变量rem_4、rem_100和rem_400中。下面我们就可以针对这几个数字，建立复合关系表达式，判断某个年份是否是闰年了。

如果我们把前面判断某个年份是否是闰年的条件重新组织一下，可以像下面这样叙述：如果一个年份能被4整除但是不能被100整除，或者能够被400整除，那么这个年份就是闰年。读者可以再考虑一下，这个叙述和前面的叙述是否等价。这样重新组织过以后，我们就可以比较容易地把它翻译为程序了。

```
if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )  
    printf ("It's a leap year.\n");
```

表达式

```
rem_4 == 0 && rem_100 != 0
```

周围的括号是不必要的，因为按照操作符的优先级，这个表达式的执行顺序和不加括号的时候是一样的（译者注：但是加上括号更清楚一些）。

我们在这个语句前后再加上一些必要的变量声明和读取用户输入的语句，就可以形成一个完整的判断闰年的程序，如程序6.5所示。

程序6.5 判断某个年份是否是闰年

```
// Program to determines if a year is a leap year  
  
#include <stdio.h>  
  
int main (void)  
{  
    int year, rem_4, rem_100, rem_400;  
  
    printf ("Enter the year to be tested: ");  
    scanf ("%i", &year);  
  
    rem_4 = year % 4;  
    rem_100 = year % 100;  
    rem_400 = year % 400;  
  
    if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )  
        printf ("It's a leap year.\n");  
}
```

程序6.5 续

```
else
    printf ("Nope, it's not a leap year.\n");

return 0;
}
```

程序6.5 输出

```
Enter the year to be tested: 1955
Nope, it's not a leap year.
```

程序6.5 输出 (再次运行)

```
Enter the year to be tested: 2000
It's a leap year.
```

程序6.5 输出 (第三次运行)

```
Enter the year to be tested: 1800
Nope, it's not a leap year.
```

前面一共给出了三个例子，一个不能被4整除的年份（1955），它不是闰年；一个能够被400整除的年份（2000），是闰年，能够被100整除但是不能被400整除的年份（1800），不是闰年。为了完整起见，我们还应该测试能够被4整除但是不能被100整除的年份。这个作为练习留给读者。

我们前面提到过，C语言在表达式语法方面是非常灵活的。比如，在前面的程序中，实际上可以不使用rem_4、rem_100和rem_400这三个变量，而是直接在if语句的条件表达式中进行计算，如下所示：

```
if ( ( year % 4 == 0 && year % 100 != 0 ) || year % 400 == 0 )
```

同样，我们在表达式中加入适量的空格，使得整个语句看上去更容易阅读。如果我们不使用空格，同时去掉那个不必要的括号，那么表达式看上去就是下面这个样子：

```
if(year%4==0&&year%100!=0)||year%400==0)
```

这个表达式也是完全合法的（难以置信，是吗），而且和前面的表达式的功能完全一样。从这个例子我们可以看出，空格对于帮助我们理解程序有多么重要。

嵌套的 if 语句

读者可能还记得，在if语句的一般形式中，如果条件表达式的值是真，那么计算机就接着执行if后面的语句。这个语句也有可能是另外一个if语句，就像下面的例子所显示的那样。

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Your Move\n");
```

如果变量gameIsOver的值是0，计算机接着执行后面的程序，这个语句刚好又是一个if语句。这个if语句的条件表达式将变量playerToMove的值与变量YOU比较，如果这两个变量的值相等，那么程序就在屏幕上输出字符串“You move”。我们可以看到，只有gameIsOver的值等于0，而且playerToMove的值等于YOU，printf语句才会被执行。实际上，前面的语句可以用复合条件表达式重新改写为下面的等价形式：

```
if ( gameIsOver == 0 && playerToMove == YOU )
    printf ("Your Move\n");
```

下面是一个更实际一点的嵌套if语句的例子，它比前面的例子多出了else部分。

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Your Move\n");
    else
        printf ("My Move\n");
```

这段程序的执行方式和前面相同，但是如果变量gameIsOver的值等于0，而变量playerToMove的值不等于YOU，那么计算机实际上要执行else子句部分，这部分程序将在终端上打印消息“My move”。如果变量gameIsOver的值不等于0，那么后面的整个语句都将被忽略，包括与内层if语句相关联的else子句。

请读者仔细观察一下，上面这个例子中的else子句，实际上是和测试playerToMove的值是否等于YOU的那个if语句相关联，而不是和测试gameIsOver的值是否等于0的那个if语句相关联。一般说来，else子句总是和离它最近，而且没有关联的else语句的那个if语句相关联。

我们也可以给外层的if语句加上相应的else语句，这个else子句当gameIsOver的值不等于0的时候执行。程序如下：

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Your Move\n");
    else
        printf ("My Move\n");
else
    printf ("The game is over\n");
```

读者可以看到，合理的使用缩进，可以大大增加程序的可读性。

当然了，缩进只是用来表示我们希望C语言如何解释程序，实际上编译器并不关心缩进。因此，有些时候，我们缩进程序的方式和计算机执行程序的方式可能并不一致。例如，如果我们将前面例子中的第一个else子句，将会得到如下缩进的程序

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Your Move\n");
else
    printf ("The game is over\n");
```

这个程序中的第二个else语句虽然看上去和外层的if语句相关联，实际上，C语言的编译器按照下面的格式解释这些语句

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Your Move\n");
    else
        printf ("The game is over\n");
```

因为编译器总是把else语句和离它最近并且没有配对的if语句联系在一起。如果在书写程序的时候，真的遇到内层if语句没有配对的else子句，而外层的if语句有的话，我们可以使用括号来告诉编译器这一点。如下面的例子代码所示：

```
if ( gameIsOver == 0 ) {
    if ( playerToMove == YOU )
        printf ("Your Move\n");
}
else
    printf ("The game is over\n");
```

这段程序正确地反映了我们的意图，当gameIsOver的变量值不等于0的时候，程序在屏幕上输出消息“The game is over”。

else if 结构

我们前面已经演示了当测试条件的结果有两种可能性的时候，如何使用else语句。比如：一个整数不是奇数就是偶数、一个年份或者是闰年，或者不是闰年。但是，我们在实际编写程序的时候需要进行的判断，并不一定总是如此非此即彼。考虑我们需要编写下面一个程序：读入用户输入的数字，如果该数字小于0，程序显示-1；当该数字等于0的时候显示0；当该数字大于0的时候显示1（这实际上就是符号函数完成的工作）。很明显，在这个程序中我们需要进行三种判断——输入的数字是大于0、等于0还是小于0。这个时候，简单的if-else结构就派不上用场了。当然，我们也可以使用三个独立的if语句，但是这并不是一个通用的解决方法。如果需要测试的条件并不是互斥的话，我们就无法使用这种方法。

通过在else子句中使用if语句，我们就可以处理这种情况。在C语言中，else子句后面可以是任何合法的C语句，所以当然也可以是if语句。因此在这种情况下，我们一般使用如下的语法结构来处理具有三种可能结果的测试条件。

```
if ( 表达式 1 )  
    语句 1  
else  
    if ( 表达式 2 )  
        语句 2  
    else  
        语句 3
```

我们还可以在最后的else子句后面再增加新的if语句，这样就可以描述4、5直到有任意个可能结果的测试条件。

在编写程序的时候，人们经常使用上面的语法结构，为了简化和清楚起见，人们更倾向于使用下面的格式，

```
if ( 表达式 1 )  
    语句 1  
else if ( 表达式 2 )  
    语句 2  
else  
    语句 3
```

通过这个方式排列语句，能够更容易地看出这是一个有三种可能结果的条件判断结构。

程序6.6使用else if结构实现了我们刚刚讨论过的符号函数。

程序6.6 符号函数的实现

```
// Program to implement the sign function  
  
#include <stdio.h>  
  
int main (void)  
{  
    int number, sign;  
  
    printf ("Please type in a number: ");  
    scanf ("%i", &number);
```

程序6.6 续

```
if ( number < 0 )
    sign = -1;
else if ( number == 0 )
    sign = 0;
else // Must be positive
    sign = 1;

printf ("Sign = %i\n", sign);

return 0;
}
```

程序6.6 输出

```
type in a number: 1121
Sign = 1
```

程序6.6 输出（再次运行）

```
Please type in a number: -158
Sign = -1
```

程序6.6 输出（第三次运行）

```
Please type in a number: 0
Sign = 0
```

在程序6.6中，如果用户输入的数值小于0，我们就将变量sign的值赋为-1；如果输入的值等于0，我们就给它赋值0；如果上述两个判断都不成立的话，这个数肯定大于0，于是我们就将变量sign的值赋为1。

程序6.7从终端读入一个字符，然后对该字符进行分类，判断它是一个字母（a-z或者A-Z），数字（0-9）还是其他特殊字符。如果要从终端读入字符，可以在scanf函数的格式化输入字符串中使用%c。

程序6.7 对终端输入的字符进行分类

```
// Program to categorize a single character that is entered at the terminal

#include <stdio.h>

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

程序6.7 续

```
printf ("Enter a single character:\n");
scanf ("%c", &c);

if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
    printf ("It's an alphabetic character.\n");
else if ( c >= '0' && c <= '9' )
    printf ("It's a digit.\n");
else
    printf ("It's a special character.\n");

return 0;
}
```

程序6.7 输出

```
Enter a single character:
&
It's a special character.
```

程序6.7 输出（再次运行）

```
Enter a single character:
8
It's a digit.
```

程序6.7 输出（第三次运行）

```
Enter a single character:
B
It's an alphabetic character.
```

在程序读取用户输入的字符并将其保存在字符类型的变量c中以后,我们首先判断该字符是否是一个字母。一个字母可以是写大的,也可以是小写的。我们可以使用下面的条件表达式判断某个字符是否是小写字母:

```
( c >= 'a' && c <= 'z' )
```

在这个表达式中,如果字符c的值在a-z之间,表达式的值就是真,也就是说,该字符就是一个小写字母。同样,下面的表达式判断字符c是否是一个大写字母:

```
( c >= 'A' && c <= 'Z' )
```

上面的判断方法，在所有使用ASCII格式保存字符的计算机上都是有效的¹。

如果变量c中保存的字符是一个字母，那么第一个if语句的条件表达式的值就会为真，于是，程序打印出消息 “It's an alphabetic character.”（译者注：它是一个字母字符。）如果字符c不满足该条件，计算机将接着执行else if子句，该子句判断字符c是不是一个数字。这里需要提醒读者注意，我们将字符c与字符常量'0'和'9'进行比较，而不是数字常量0和9。因为，从终端读入的是一个字符，而字符'0'、'9'和数字0、9在C语言中代表的是不同的东西。实际上，在使用ASCII码的计算机中，字符'0'在内部被表示为数字48，而字符'1'表示为数字49，依此类推。

如果字符c的值满足第二个条件表达式，我们的程序将打印出 “It's a digit.”（译者注：它是一个数字。）否则，字符c既不是一个字母，也不是一个数字，计算机将执行最后的else语句，最终打印出 “It's a special character.”（译者注：它是一个特殊字符。）然后程序结束运行。

读者也许会注意到，虽然在程序6.7中我们仅仅从终端读入一个字符，但是用户还是必须按下回车（Enter）键或者Return键。一般说来，如果我们的程序从终端读取输入的话，那么只有用户按下回车（Enter）键，计算机才会把用户输入的内容传递给程序。

在下面的例子程序中，我们将允许用户输入类似下面的简单表达式：

数字 操作符 数字

程序将求出该表达式的值，并以两位小数的精度将结果输出到屏幕上。我们只允许用户使用普通的算术操作符，也就是加、减、乘和除。程序6.8实现了上述功能，其中我们使用了一个大型的if语句组合来处理各种可能的操作符。

程序6.8 对简单的表达式求值

```
/* Program to evaluate simple expressions of the form
   number operator number */
#include <stdio.h>

int main (void)
{
    float value1, value2;
    char operator;
```

¹ 在实践中，读者应该使用标准库函数如 islower、isupper 来进行上述判断，这样写出的程序就不会依赖于计算机存储字符所使用的具体编码方式。我们这里给出的例子主要起说明作用。

程序6.8 续

```
printf ("Type in your expression.\n");
scanf ("%f %c %f", &value1, &operator, &value2);

if ( operator == '+' )
    printf ("%f\n", value1 + value2);
else if ( operator == '-' )
    printf ("%f\n", value1 - value2);
else if ( operator == '*' )
    printf ("%f\n", value1 * value2);
else if ( operator == '/' )
    printf ("%f\n", value1 / value2);

return 0;
}
```

程序6.8 输出

```
Type in your expression.
123.5 + 59.3
182.80
```

程序6.8 输出（再次运行）

```
Type in your expression.
198.7 / 26
7.64
```

程序6.8 输出（第三次运行）

```
Type in your expression.
89.3 * 2.5
223.25
```

在程序6.8中，我们首先使用scanf函数将三个值读入到变量value1、operator和value2中。浮点数的格式化输入字符和格式化输出字符一样，都是%f，因此，我们在scanf中使用%f读入第一个操作数value1。

下来我们需要读入操作符。因为操作符是一个字符（'+'、'-'、'*'或'/'）而不是一个数字，所以我们使用字符型变量operator来保存操作符。在这里我们使用格式化字符%c来告诉系统，我们需要读入一个字符型的变量。在%f和%c之间有一个空格，这个空格表示我们将略去输入中该位置的任意多个空格。因此，用户在输入的时候可以使用空格将操作数和操作符分开。如果我们在程序中使用格式化字符串%f%c%f，那么用户在输入表达式的时候，在操作符和第一个操作数之间就不能使用空格。因为我们使用格式化输入字符%c表

明我们需要读入一个字符，而空格也是一个合法的字符。这样，我们就将读到一个空格而不是操作符。但是，值得注意的是，当scanf函数读入一个整型数或者浮点数的时候，数字前面的空格总是被忽略掉。因此，使用格式化字符串%f %c%f和程序中格式化字符串效果相同。

当程序读入第二个操作符的值，并将其存储在变量value2中后，程序就将变量operator中存放的字符与四种可能的操作符进行比较。如果该字符与某个操作符匹配的话，程序就将执行相应的printf语句，并显示计算结果。程序的执行就此结束。

现在，我们该向读者介绍程序完整性方面的一些知识了。虽然前面的程序的确能够完成所需的工作，但是该程序并不完整，因为它没有考虑用户输入错误这样的情况。比如，如果用户错误地输入一个?作为操作符时，这个程序将不执行任何一个printf语句，也不输出任何提示信息以帮助用户发现其错误。

这个程序还忽略了另外一种可能性，那就是用户输入0作为除数。我们知道，在C语言中，我们永远不应该用某个数除以0。因此，程序还应该检查这种情况。

事先觉察到那些可能导致程序失败的情况，并采取预防性的措施，对于编写正确、可靠的程序是非常重要的。如果我们针对某个程序进行足够多的测试，就会常常发现那些对特定情况没有进行充分考虑的程序片断。仅仅测试还是不够的，在编写程序的时候，我们还应该经常多次反问自己“如果这样将会如何，如果那样将会如何”，并加入相应的语句处理这些情况。

程序6.8A是程序6.8的修订版，它考虑了除数为0和未知操作符这两种情况。

程序6.8A 简单表达式求值的修订版

```
/* Program to evaluate simple expressions of the form
   value operator value */

#include <stdio.h>

int main (void)
{
    float value1, value2;
    char operator;

    printf ("Type in your expression.\n");
    scanf ("%f %c %f", &value1, &operator, &value2);
```

程序6.8A 续

```
if ( operator == '+' )
    printf ("%2f\n", value1 + value2);
else if ( operator == '-' )
    printf ("%2f\n", value1 - value2);
else if ( operator == '*' )
    printf ("%2f\n", value1 * value2);
else if ( operator == '/' )
    if ( value2 == 0 )
        printf ("Division by zero.\n");
    else
        printf ("%2f\n", value1 / value2);
else
    printf ("Unknown operator.\n");

return 0;
}
```

程序6.8A 输出

```
Type in your expression.
123.5 + 59.3
182.80
```

程序6.8A 输出 (再次运行)

```
Type in your expression.
198.7 / 0
Division by zero.
```

程序6.8A 输出 (第三次运行)

```
Type in your expression.
125 $ 28
Unknown operator.
```

如果用户输入的操作符是除法（斜线），我们就在程序中新增一个if语句，判断变量value2是否为0。如果是0的话，就在屏幕上输出适当的提示信息，否则就进行除法运算并输出结果。对于本程序中的嵌套if语句与else子句的配对情况，读者需要特别留意。

程序中的最后一个else语句用于处理所有那些不满足前面条件的情况。任何与四种基本的操作符不匹配的操作符都将导致程序的执行流程转到这个else语句，并在终端上显示“Unknown operator”（译者注：未知的操作符）。

switch 语句

在编程实践中，将一个变量与多个值进行比较，并决定下一步如何执行是非常常见的。前面我们用else-if结构来处理这种编程任务，实际上，C语言提供了一种专门的结构用于完成这个任务，这就是switch语句。switch语句的一般格式如下：

```
switch ( 表达式 )
{
    case value1:
        程序代码
        程序代码
        ...
        break;
    case value2:
        程序代码
        程序代码
        ...
        break;
    ...
    case valuen:
        程序代码
        程序代码
        ...
        break;
    default:
        程序代码
        程序代码
        ...
        break;
}
```

switch后面括号中的表达式将被逐个与value1、value2、...、valuen的值进行比较，如果表达式与某个value表达式的值相等的话，该value表达式后面的语句将被执行。这里的value1，value2，...valuen都必须是常量或者常量表达式。需要注意的是，即使case语句后面跟有多条语句，也不需要把它们用大花括号括起来。

break语句表示某个case语句的结束，同时switch语句的执行也将中止。读者一定要记得在每一个case的最后加上break语句，否则程序的流程就会转到后面的case中去。

如果表达式(expression)的值与任何一个value表达式的值都不相等，那么程序将执行特殊的case——default语句。default语句相当于我们前面例子中最后的那个else语句的作用。实际上，switch语句可以用下面等价的if语句结构来代替：


```
if ( 表达式 == value1 )
{
    程序代码
    程序代码
    ...
}
else if ( 表达式 == value2 )
{
    程序代码
    程序代码    ...
}
...
else if ( 表达式 == valuen )
{
    程序代码
    程序代码
    ...
}
else
{
    程序代码
    程序代码
    ...
}
```

只要记住这一点，我们就能够把大块if语句转换为对应的switch语句。程序6.9就是用switch语句代替程序6.8中的if语句形成的结果。

程序6.9 简单表达式求值的修订版 第二版

```
/* Program to evaluate simple expressions of the form
   value operator value */

#include <stdio.h>

int main (void)
{
    float value1, value2;
    char operator;

    printf ("Type in your expression.\n");
    scanf ("%f %c %f", &value1, &operator, &value2);

    switch (operator)
    {
        case '+':
            printf ("%f\n", value1 + value2);
```

程序6.9 续

```

        break;
    case '-':
        printf ("%f\n", value1 - value2);
        break;
    case '*':
        printf ("%f\n", value1 * value2);
        break;
    case '/':
        if ( value2 == 0 )
            printf ("Division by zero.\n");
        else
            printf ("%f\n", value1 / value2);
        break;
    default:
        printf ("Unknown operator.\n");
        break;
}

return 0;
}

```

程序6.9 输出

```

Type in your expression.
178.99 - 326.8
-147.81

```

当程序读入表达式之后，就把变量operator的值和每一个case语句中的常量表达式进行比较。如果找到了一个匹配的表达式，程序就接着执行这个case语句后面对应的语句，直到遇到一个break语句为止。如果变量operator的值和所有的case语句都不匹配，那么程序就转而执行default后面的语句，这些语句将打印消息“Unknown operator”（译者注：未知的操作符），并结束switch语句的执行。

在上面的例子中，default后面的break语句实际上是不必要的，因为default后面再没有别的case语句了。但是，在switch语句的每一个case中加上break语句是一个很好的编程习惯。

在switch语句中，case后面的常量表达式不能相同。但是，不同的case表达式的值可以执行同一组语句。为了做到这一点，我们仅仅需要在需要执行的语句前面列出所有这些值（就是在这些值前面加上关键字case，然后后面跟上冒号）。下面是多个值执行同一段语句的示例代码。只要变量operator的值等于字符星号或者小写的字符x，程序就将执行对应的printf语句，该语句计算两个操作符的乘积，并将结果显示在终端上。


```
switch (operator)
{
    ...
    case '*':
    case 'x':
        printf ("%f\n", value1 * value2);
        break;
    ...
}
```

布尔变量

许多新手程序员都会遇到生成质数表的编程任务。回忆一下，所谓质数就是除1之外所有只能被1和自身整除的整数。第一个质数是2，下一个是3，因为它只能被1和3整除。4不是质数，因为它可以被2整除。

有好几种方法可以用来生成质数表。如果我们需要生成50以内的质数表，那么最简单的方法就是将1到50所有的质数 p 都执行如下测试：用从2到 $p-1$ 的每一个整数去除 p ，如果能够整除的话，那么 p 就不是质数，如果每一个整数都不能整除的话，那么 p 就是质数。程序6.10实现了这个算法。

程序6.10 生成质数表

```
// Program to generate a table of prime numbers

#include <stdio.h>

int main (void)
{
    int p, d;
    _Bool isPrime;

    for ( p = 2; p <= 50; ++p ) {
        isPrime = 1;

        for ( d = 2; d < p; ++d )
            if ( p % d == 0 )
                isPrime = 0;

        if ( isPrime != 0 )
            printf ("%i ", p);
    }
}
```

程序6.10 续

```

    printf ("\n");
    return 0;
}

```

程序6.10 输出

```

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

```

对于程序6.10，有如下几点需要说明。程序首先建立一个外层循环，循环变量将遍历整数2到50。我们将测试每个循环变量的值是否为质数。在循环体中的第一条语句首先将变量isPrime的值设置为1。我们随后将会说明这个变量的用途。

第二个循环将利用2到p-1的整数作为除数，对外层循环变量p进行除法测试，检查余数是否为0。如果某个除法的余数是0，我们就知道p不是质数。因为除了其自身和1之外，还有别的整数可以整除它。为了记录p不再可能是一个质数，我们把变量isPrime的值设置为0。

当内层循环执行完毕之后，我们检查变量isPrime的值。如果该变量的值不等于0，那么从2到p-1之间，没有整数能够整除p，因此p肯定是一个质数，于是我们就将p的值显示在屏幕上。

读者也许会注意到，变量isPrime的值不是0就是1，不会再是其他别的数字。这也是我们将其声明为_Bool类型的原因。如果p是一个质数，那么在整个内层循环的执行过程中，变量isPrime的值总是1。只要有一个整数能够整除p，我们就将它的值设置为0，以表示p不再符合质数的标准。这种用途的变量通常被称为标志变量。标志变量常常用于那些只能在两个不同的值之中取一个值的场合。除此之外，程序中经常（至少一次）对标志变量进行测试，检查它是打开的（TRUE）还是关闭的（FALSE）。根据测试结果的不同，程序采取不同的步骤。

采用数字1代表条件为“真”，用数字0代表条件为“假”，这是一种很自然的做法。因此，在程序6.10中，我们在循环中将变量isPrime的值设置为1，实际上就是表示变量p是一个质数。如果在执行内层循环的时候发现了一个整数能够整除p，我们就把变量isPrime的值设置为“假”，表示p不是一个质数。

在程序6.10中，我们使用数字1代表条件为“真”和0代表条件为“假”并不是一种巧合。这与计算机内部“位”的概念有关。当某个位“打开”（on）的时候其值为1，当该位

“关闭”(off)的时候,其值为0。但是在C语言中,还有更加充分的理由使得我们用1对应“真”,0对应“假”。这就是C语言中逻辑值“真”和“假”的处理方式。

在本章开始的时候,我们说过,如果if语句中指定的条件被满足的话,if中的语句就被执行。但是,这里“满足”的确切意思是什么呢?在C语言中,满足条件就是指表达式的结果不为0,再没有其他标准了。请看下面的语句

```
if ( 100 )  
    printf ( "This will always be printed.\n");
```

这个if语句中,条件表达式的值不等于0(它等于100),因此这个条件总是满足的,因而printf语句总是被执行。

在本章的每一个程序中,都隐含了“非0表示满足条件”和“0表示不满足条件”这个概念。对于C语言的条件表达式来说,如果条件满足的话,整个表达式的值就是1;如果条件不满足,那么表达式的值就是0。下面的表达式:

```
if ( number < 0 )  
    number = -number;
```

实际上在C语言中按照下面的方式执行:

1. 计算机首先判断条件`number < 0`是否满足,如果条件满足的话(也就是说`number`的值小于0),整个表达式的值就是1,否则的话,表达式的值就是0。
2. if语句接下来检查条件表达式的值。如果表达式的值不等于0,那么就接着执行后面的语句;如果表达式的值等于0,后面的语句就被跳过。

前面的讨论对于for语句、while语句和do语句中的条件表达式也是适用的。具体来说,对于下面的复合条件表达式

```
while ( char != 'e' && count != 80 )
```

其求值的方式与前面的讨论也类似。如果两个条件都满足,整个表达式的值就是1;如果有任何一个不满足,表达式的值就是0。接下来,while语句将对符合条件表达式的结果进行检查,如果其值为0,那么就终止循环执行,否则的话继续执行循环。

让我们回到程序6.10。对于标志变量isPrime来说,我们同样可以用下面的if语句来检验它是否代表条件为“真”:

```
if ( isPrime)
```

而不是下面的语句:

```
if( isPrime != 0 )
```

为了便于测试某个条件是否为“假”，我们也可以使用逻辑非操作符!。下面的语句

```
if ( ! isPrime)
```

使用逻辑非操作符检查变量isPrime是否代表条件“假”（读者可以把这个表达式念做“如果非isPrime”）。一般说来，类似下面的表达式

! 表达式

会将表达式 (expression) 的逻辑值反过来。也就是说，如果表达式 (expression) 的值是0，那么!表达式 (expression) 的值就是1；如果表达式 (expression) 的值是非0，那么!表达式 (expression) 的值就是0。

使用逻辑非操作符，我们可以很方便地反转一个标志变量的值。如下面的语句所示：

```
myMove = ! myMove
```

正与读者推测的相同，逻辑非操作符的优先级与单目操作符的优先级相同。也就是说，它的优先级高于所有的二元算数操作符和所有的逻辑操作符。因此，如果我们用下面的语句来测试条件x不小于y

```
! ( x < y )
```

那么就必须使用括号。当然我们也可以使用下面的等价语句

```
x >= y
```

在第4章“变量、数据类型和算数表达式”中，我们已经知道，C语言中有一些特殊的定义可以方便布尔变量的使用。当我们和布尔变量打交道的时候，可以使用这些定义。它们包括类型bool，值true和false。为了使用这些特殊的定义，我们需要在程序中包含头文件<stdbool.h>。程序6.10A使用这些特殊的定义重写了程序6.10。

程序6.10A 生成质数的程序（修订版）

```
// Program to generate a table of prime numbers

#include <stdio.h>
#include <stdbool.h>

int main (void)
{
    int p, d;
    bool isPrime;

    for ( p = 2; p <= 50; ++p ) {
        isPrime = true;
```


程序6.10A 续

```
for ( d = 2; d < p; ++d )
    if ( p % d == 0 )
        isPrime = false;

    if ( isPrime != false )
        printf ("%i ", p);
}

printf ("\n");
return 0;
}
```

程序6.10A 输出

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

正像我们在程序6.10A中看到的那样，一旦包含了标准头文件<stdbool.h>，我们就可以在程序中使用类型bool而不是_Bool来声明变量。这种变化的主要好处，仅仅在于前者比起后者较为美观，也容易输入。并且和C语言的其它基本变量类型，如int、float和char等看上去更为一致。

条件操作符

条件操作符也许是C语言中最不寻常的操作符。我们前面介绍的操作符不是单目的就是双目的，而条件操作符则是一个三目的操作符，也就是说，它需要三个操作数。条件操作符使用两个符号：问号（?）和冒号（:），第一个操作数放在问号之前，第二个操作数放在问号和冒号之间，第三个操作数放在冒号后面。

条件操作符的一般格式如下：

(condition) 条件表达式 ? 表达式 (expression) 1 : 表达式 (expression) 2

其中，表达式condition通常是一个条件表达式，在计算条件操作符组成的表达式时，首先对它求值。如果求值的结果为真（也就是说不为0），程序接下来对于表达式expression1求值，求值的结果就是整个条件操作符表达式的值；如果条件表达式condition的求值结果为假（就是说为0），程序就对表达式expression2求值，并将其结果作为整个条件操作符表达式的结果。

在需要根据某些条件判断给变量赋两个值中的一个时，我们经常运用条件表达式。例如，假定我们声明了两个变量x和s，如果x小于0，我们将s的值设置为-1，反之则将s的值设置为x的平方。利用条件操作符，我们可以编写如下的语句完成这项工作：

```
s = ( x < 0 ) ? -1 : x * x;
```

程序首先计算表达式 $x < 0$ 的结果。为了清楚起见，我们条件表达式加上了括号。这个括号并不是必要的，因为条件操作符的优先级非常的低，除了赋值操作符(=)和逗号操作符(,)之外，它的优先级比其他的操作符都低。

如果x的值小于0，程序就计算?后面的表达式的值。在上面的例子中，?后面是一个常量表达式-1，程序就将这个值赋给x。

如果x的值不小于0，程序就计算:后面的表达式的值，并将其结果赋值给s。在上面的例子中，如果x的值大于等于0，那么s就被设置为x的平方。

下面是使用条件操作符表达式的另外一个例子。这个例子比较变量a和b的值，并将其中较大的一个赋给变量maxValue。

```
maxValue = ( a > b ) ? a : b;
```

在条件操作符表达式的后面附上另外一个条件操作符表达式，就可以达到else if语句的效果。例如，我们在前面的程序6.6中实现的sign函数，可以用一条包含两个条件操作符的语句来完成，如下所示：

```
sign = ( number < 0 ) ? -1 : (( number == 0 ) ? 0 : 1);
```

在上面的语句中，如果number小于0，变量sign的值就被设置为-1；如果number的值等于0，sign的值就被设置为0；如果这两种情况都不满足的话，sign的值就被设置为1。上面语句中，第二个条件表达式周围的括号是不必要的，因为C语言中规定，条件操作符的关联性是从右向左，也就是说，如果在一个语句中使用了多个条件操作符，如：

```
e1 ? e2 : e3 ? e4 : e5
```

那么编译器将对其从右向左分组，结果如下：

```
e1 ? e2 : ( e3 ? e4 : e5 )
```

条件操作符表达式并不一定要出现在赋值操作符的右面，我们可以在任何需要表达式的地方使用它。下面的printf语句打印出变量number的符号，该语句并未使用变量存储中间结果，如下所示：

```
printf("Sign = %i\n", ( number < 0 ) ? -1 : ( number == 0 ) ? 0 : 1);
```

在定义宏的时候，使用条件表达式会带来很多方便。我们将在第13章“预处理器”中详细介绍其使用。

第6章到这里就结束了。在第7章“使用数组”中，我们将会第一次接触到复杂的数据类型。数组是非常有用的概念，在很多C语言程序中，我们都会用到它。和以往一样，在学习下一章之前，请读者先完成本章的习题，以巩固学到的知识。

练习

1. 输入并运行本章的10个例子程序，并将结果与书中列出的结果进行对比。试验输入其他的值，观察程序的运行结果。
2. 编写程序，从终端读入用户输入的两个整数，判断第一个数能否整除第二个数，并在屏幕上输出适当的信息。
3. 编写程序，从终端读入用户输入的两个整数，显示第一个数除以第二个数的结果，结果要求显示3位小数。切记检查除数是否为0。
4. 编写简单的计算器。该程序允许用户输入如下格式的算式：

数字 操作符

其中操作符可以是下面的任意一个：

+ - * / S E

操作符S将计算器中的“记忆数字”设置为输入的数值。操作符E告诉程序结束执行。如果输入的是算术操作符，那么就把计算器中的“记忆数字”作为第一个操作数，用户输入的数字作为第二个操作数，计算结果作为“记忆数字”重新保存起来。下面是我们的计算器运行的一个示意过程：

Begin Calculations	
10 S	设置记忆数字为10
= 10.000000	显示记忆数字
2 /	除以2
= 5.000000	显示记忆数字
55 -	减去55
-50.000000	
100.25 S	设置记忆数字为 100.25
= 100.250000	
4 *	Multiply (乘以4)
= 401.000000	
0 E	结束运行
= 401.000000	
End of Calculations.	

在程序中要求检查除数为0和未知的操作符这两种意外情况。

5. 在程序5.9中，我们逆序显示一个数所有位上的数字。但是，如果我们输入一个负数，这个程序的结果就不正确了。请读者检查错误发生的原因，并对程序进行相应的修正。负数逆序输出举例：输入-8645，输出应该是5468-。

6. 编写程序，从终端读取一个数字，然后使用英语逐位显示该数。例如，如果用户输入932，程序的输出如下：

```
nine three two
```

这里特别要求，如果用户输入0，那么程序应该显示zero（注意：这个练习有相当难度）

7. 程序6.10有几个缺陷。没有针对偶数特别处理就是其中之一，因为任何大于2的偶数都不可能是质数，所以我们可以处理跳过所有大于2的偶数，既不将其作为质数的候选者，也不将其作为除数。另外，程序的内层for循环也有缺陷，因为无论中间结果如何，我们总是要用p逐个除以2到p-1之间的数。我们可以在内层for循环的循环条件的测试表达式中增加一个对isPrime的测试来避免这些不必要的计算。按照我们这里叙述的方法，对程序6.10进行修改，并运行修改后的程序以验证修改是否正确。（注意：在第7章中，我们将讲述更加高效的产生质数的方法。）

Working with Arrays

使用数组

C语言中的数组可以用来存储一组有序的数据。本章将向读者介绍如何定义和使用数组。在后续章节，读者将会了解到数组的更多知识，例如数组如何与函数、结构、字符串和指针等共同工作。

假定现在手头有一组学生成绩，我们想要把它们读入计算机并进行某些操作。如按照升序排序、计算平均值或者找出它们的中间数等。在程序6.2中，我们可以逐个读入成绩，计算它们的总和并在最后求出平均值。但是，如果我们试着按照升序排列这组成绩，就需要更多的工作。认真想一想，我们就会发现，只有读入了全部数据，我们才能开始排序操作。因此，如果按照程序6.2中的算法，我们就必须为每个成绩定义一个单独的变量，用于存放其数值，最终的代码可能如下所示：

```
printf ("Enter grade 1\n");  
scanf ("%i", &grade1);  
printf ("Enter grade 2\n");  
scanf ("%i", &grade2);  
. . .
```

在输入了所有成绩之后，我们可以使用一系列if语句在这些成绩之间进行比较，找出其中最小的数、次小的数，以此类推，直到我们找出这组成绩中最大的数，就完成了升序排序的任务。如果认真着手去写这个程序，读者很快就会意识到，对于任何有意义的成绩数量（比如说10个成绩），最终写出的程序也会相当巨大和复杂。不要灰心，这正是C语言的数组施展威力的地方。

定义数组

使用数组，我们可以定义一个变量`grades`，用于代表一组成绩，而不是单个成绩。我们可以使用下标或者索引的方式访问这组成绩中的每一个。在数学中，一个带下标的符号，如 x_i 表示一组元素 x 中的第 i 个。在C语言中，我们用下面的记号来表示下标：

`x[i]`

按照这种记法，下面的表达式

`grades[5]`

就代表数组`grades`中的5号元素。因为C语言中数组的下标从0开始，所以下面的表达式：

`grades[0]`

实际上代表的是数组中的第一个元素（正因为如此，读者很快就会习惯将该表达式看作数组中的0号元素，而不是第一个元素）。

在任何可以使用普通变量的地方，我们都可以使用单个数组元素。例如，下面赋值语句就使用了数组。

`g = grades[50];`

这个语句将数组元素`grades[50]`中保存的数值赋给变量`g`。更一般的，如果`i`是一个整型变量，那么下面的语句

`g = grades[i];`

将`grades`数组中的`i`号元素赋值给变量`g`。例如，如果`i`等于7，上面的语句就把`grades`数组中的7号元素的值赋给变量`g`。

只要将数组元素放置在赋值表达式左面，我们就可以把数字保存在该数组元素中。如下面的语句所示：

`grades[100] = 95;`

这个语句将数字95保存在数组`grades`的100号元素中。下面的语句

`grades[i] = g;`

则将变量`g`保存在数组`grades`的`i`号元素中。

将一组相关联的数据保存在单个数组中之后，我们就可以简明又高效地开发程序了。例如，我们可以使用一个下标变量，很轻松地遍历所有的数据。例如，下面的循环语句

```
for ( i = 0; i < 100; ++i )  
    sum += grades[i];
```


遍历访问数组grades的前100个数据(0号到99号),并将它们加到变量sum中。如果sum的初值为0,那么这段程序运行完毕之后,变量sum中就保存着grades数组中前100个成绩的和。

在使用数组的时候,请读者务必记住,数组的第一个元素用下标0访问,数组的最后一个元素的下标是数组中元素的个数减去1。

除了整数常量之外,任何结果是整型数的表达式都可以放在方括号内,用作数组的下标。例如,如果low和high都是整型变量的话,那么下面的语句

```
next_value = sorted_data[(low + high) / 2];
```

用方括号中表达式的结果作为下标,将相应的数组元素赋值给变量next_value。如果low等于1,high等于9,那么数组sorted_data的5号元素就被赋值给变量next_value。如果low等于1而high等于10,由于整数除法规则的缘故($11 / 2 = 5$),next_value的值还是等于数组sorted_data的5号元素。

和普通变量一样,在使用数组之前,我们必须先声明它。数组的声明包括两个要素,一个是数组中保存的元素类型,如int、float或者char;另外一个为数组中需要保存的元素最大个数。C语言的编译器使用这些信息计算数组需要的存储空间。

下面是数组声明的一个例子

```
int grades[100]
```

这个语句声明了一个能够容纳100个整数的整型数组grades。这个数组的有效下标是从0到99。读者在C语言中使用数组的时候,一定要留意下标的有效性。C语言本身对于下标并不进行范围检查,因此如果我们使用该数组的150号元素,程序运行并不一定会报错,但是很可能产生不可预料或者非预期的运行结果。

下面的语句声明了一个能够保存200个浮点数的数组。

```
float averages[200]
```

编译器看到这条语句,就会为该变量分配一块能够保存200个浮点数的空间。与之类似,下面的语句

```
int values[10]
```

声明了一个能够容纳10个整数的整型数组。读者可以参照图7.1来理解数组的概念。

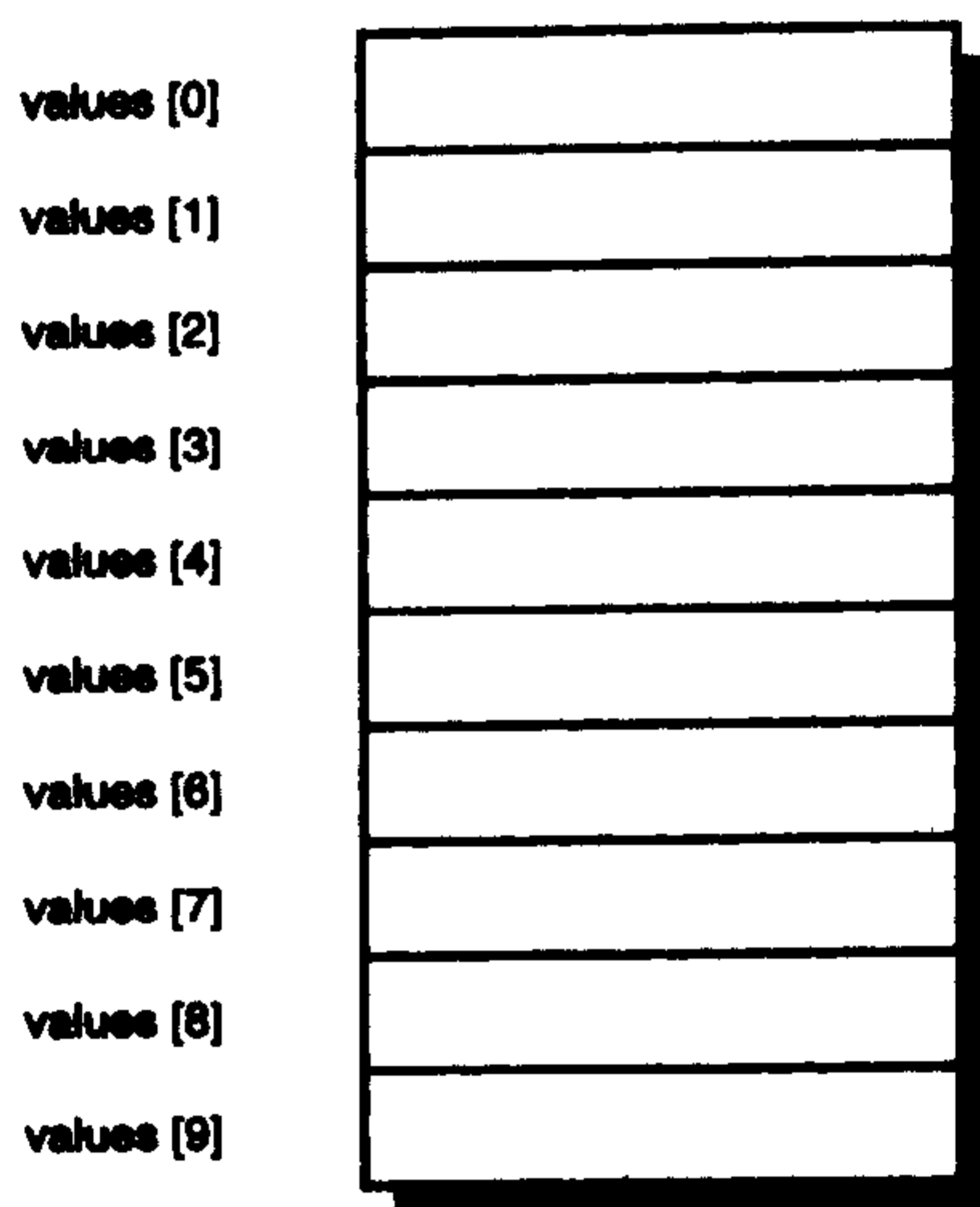


图 7.1 数组的内存结构

无论数组的类型是int、float还是char，我们都可以像使用普通变量那样使用这些数组的元素，比如给它们赋值、在终端上输出、执行加减法操作等。例如，如果执行下面的语句，那么数组values中保存的数值就如图7.2所示。

```
int values[10];

values[0] = 197;
values[2] = -100;
values[5] = 350;
values[3] = values[0] + values[5];
values[9] = values[5] / 10;
--values[2];
```

第一个赋值语句将数值197保存到数组元素values[0]中。与之类似，第二个和第三个赋值语句分别把数值-100和350保存到数组元素values[2]和values[5]中。接下来的语句将数组元素values[0]（也就是197）中的内容和数组元素values[5]（也就是350）中的内容加起来，将其结果547保存在数组元素values[3]中。然后，values[5]中的内容（350）又被除以10，然后其结果被保存在数组元素values[9]中。最后一个语句将数组元素values[2]的值减去1，使得values[2]中的值从-100变为-101。

前面的语句组成了程序7.1的主干。这个程序中使用一个for循环遍历数组values中的每一个元素，并将其显示在终端上。

values [0]	197
values [1]	
values [2]	-101
values [3]	547
values [4]	
values [5]	350
values [6]	
values [7]	
values [8]	
values [9]	35

图 7.2 初始化后的数组

程序7.1 使用数组

```
#include <stdio.h>

int main (void)
{
    int values[10];
    int index;

    values[0] = 197;
    values[2] = -100;
    values[5] = 350;
    values[3] = values[0] + values[5];
    values[9] =
    values[5] / 10;
    --values[2];

    for ( index = 0; index < 10; ++index )
        printf ("values[%i] = %i\n", index, values[index]);

    return 0;
}
```

程序7.1 输出

```
values[0] = 197
values[1] = 0
values[2] = -101
values[3] = 547
values[4] = 0
values[5] = 350
values[6] = 0
values[7] = 0
values[8] = 0
values[9] = 35
```

由于0号元素的存在，数组的有效下标要比数组中所能容纳的最大元素个数小1。因此在程序7.1中，下标变量index从0到9循环。因为我们前面并未对数组中另外五个元素（1号、4号和6到8号）赋值，所以终端上对这些元素所显示的值是没有意义的。虽然在我们的输出中，这些数组元素的值等于0，但是C语言中对未初始化的数组元素的初值并没有定义。正因为如此，在编写程序的时候，我们不要对数组中未初始化的值进行任何假定。

使用数组计数

现在我们给出使用数组更实际一些的例子。假定我们要进行一次电话调查，找出某个电视节目的受欢迎程度，评分的标准是从1分到10分。在调查了5000位观众之后，我们得到了一份长度为5000的数字列表。现在我们要对这个列表进行分析。首先，我们想要找出观众评价的分布，也就是说，多少个观众打1分，多少个观众打2分等等，并将计算结果输出为相应表格。

虽然我们可以手工查看每一份调查结果，并统计出调查结果的分布情况，但这将是非常繁琐的一项工作。如果观众有超过10个以上可能的答案（比如考虑到观众的年龄），手工完成这项工作就更难以想象。正因为如此，我们决定开发一个程序来完成这项工作。读者最初的想法可能是使用10个计数变量，分别命名为rating_1、rating_2直到rating_10，然后读入每一个调查数据，并给相应的计数器增加1。但是如果可能的答案有10个以上，这个方法就有点繁琐。如果使用数组，我们就可以针对这个问题设计出简洁得多的解决方案。

我们可以建立一个计数器的数组ratingCounters，每读入一个调查数据，我们就给相应的计数器加1。为了节约篇幅，程序7.2中仅仅处理了20个输入。在这里我们提醒读者，在

调试程序的时候，首先试着让它在较小的测试数据集上工作，然后再使用完整的数据，这样我们就更容易发现和定位程序中的错误。

程序7.2 使用数组计数

```
#include <stdio.h>

int main (void)
{
    int ratingCounters[11], i, response;

    for ( i = 1; i <= 10; ++i )
        ratingCounters[i] = 0;

    printf ("Enter your responses\n");

    for ( i = 1; i <= 20; ++i ) {
        scanf ("%i", &response);

        if ( response < 1 || response > 10 )
            printf ("Bad response: %i\n", response);
        else
            ++ratingCounters[response];
    }

    printf ("\n\nRating Number of Responses\n");
    printf ("-----\n");

    for ( i = 1; i <= 10; ++i )
        printf ("%4i%14i\n", i, ratingCounters[i]);

    return 0;
}
```

程序7.2 输出

```
Enter your responses
6
5
8
3
9
6
5
7
15
```

程序7.2 输出 (续)

```

Bad response: 15
5
5
1
7
4
10
5
5
6
8
9

Rating Number of Responses
-----
1 1
2 0
3 1
4 1
5 6
6 3
7 2
8 2
9 2
10 1
    
```

我们定义数组`ratingCounters`的大小能够容纳11个元素。读者可能要问，“既然我们只有10个可能的调查结果，为什么要使用11个元素的数组”，这样做的原因是因为我们使用计数器数组的方式。因为每个观众评分都是在从1到10之间，我们在读入评分之后，就根据分数，将相应的计数器数组元素的数值增加1（当然，我们需要认真检查用户的输入是否确实在1到10之间）。例如，如果用户输入5，我们就将数组元素`ratingCounters[5]`的值增加1。这样，`ratingCounters[5]`中最后就保存评分为5的观众的数量。

现在读者应该明白我们为什么要定义11个元素的数组，而不是10个。因为最高的可能评分是10，所以我们要定义数组的大小是11，这样我们才能够使用10作为下标访问该数组的元素。这里再次提醒读者，由于0号元素的存在，数组能容纳的元素个数总是比能使用的最大下标大1。因为用户的评分不可能是0分，所以我们从来不使用数组元素`ratingCounters[0]`。实际上，在程序初始化和显示数组内容的时候，循环变量的值都是从1开始，这样就跳过了数组元素`ratingCounters[0]`。

读者也可以使用容纳10个元素的数组来完成上面这个程序。这样，当读取用户输入之后，我们将给数组元素ratingCounters[response - 1]的值增加1。最后，0号数组元素中存放着评分为1的观众数量，1号数组元素中存放着评分为2的观众数量，以此类推。这种方法也是完全可行的。我们没有使用这种方法的唯一原因是：把评分为n的观众数量保存在n号数组元素中，看上去要直观一些。

产生 Fibonacci 数

程序7.3产生前15个Fibonacci（斐波纳契数列）数，请读者研究一下这个程序，并试着猜测它的输出是什么样子的。表中的每一个数之间有什么关系？

程序7.3 产生Fibonacci数

```
// Program to generate the first 15 Fibonacci numbers
#include <stdio.h>

int main (void)
{
    int Fibonacci[15], i;

    Fibonacci[0] = 0; // by definition
    Fibonacci[1] = 1; // ditto

    for ( i = 2; i < 15; ++i )
        Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];

    for ( i = 0; i < 15; ++i )
        printf ("%i\n", Fibonacci[i]);

    return 0;
}
```

程序7.3 输出

```
0
1
1
2
3
5
8
13
21
```

程序7.3 输出（续）

```

34
55
89
144
233
377
    
```

前两个Fibonacci数， F_0 和 F_1 ，分别被定义为0和1。随后，每一个Fibonacci数 F_i 被定义为前面两个Fibonacci数 F_{i-1} 和 F_{i-2} 之和。也就是说， F_2 等于 F_1 与 F_0 之和。在前面的程序中，这个步骤对应于数组元素Fibonacci[0]与Fibonacci[1]相加，并将结果赋予数组元素Fibonacci[2]。按照类似的步骤，我们在程序中利用循环计算出 F_2 到 F_{14} 的值（也就是Fibonacci[2]到Fibonacci[14]）。

Fibonacci数在数学和程序设计算法领域有很多实际应用。这个数列起源于著名的“兔子问题”：假定我们最初有一对兔子，并且每对兔子每个月生出两只兔子，每对新出生的兔子在它们出生后的第二个月，就可以按照上述规律繁殖新的兔子，如果兔子从来不死亡的话，那么一年以后，我们一共会有多少对兔子？这个问题的答案就是Fibonacci数列：在第 n 个月共有 F_{n+2} 对兔子。因此，根据程序7.3的输出，我们知道在第12个月，我们将会拥有377对兔子。

使用数组产生质数

现在让我们回到在第6章“进行判断”中遇到的产生质数表的问题，我们将研究如何使用数组来增加程序的效率。程序6.10A中，我们判断质数的标准是：如果一个数不能被从2到该数减一之间的任何一个数整除，那么这个数就是质数。在第6章的练习题7中，我们还提到了两个可以改进算法效率并易于实施的措施。尽管如此，这个算法的效率还是不够高。虽然在产生较小的质数表时，效率的问题并不是很重要，但是在产生大型的质数表（比如1,000,000以内的质数），算法的效率就是一个至关重要的问题了。

下面是关于某个数是否是质数的一个改进了的判断标准：如果一个数不能被任何质数整除，那么这个数就是一个质数。这个标准可以由如下的基本定理推断出来：任何一个非质整数都可以分解为多个质数的积（例如，20的质因数是2、2和5）。这个标准可以大大减少我们在判断某个数是否为质数的时候所进行的除法操作的数量。我们只需要判断当前的数字能否被我们前面找出的质数整除就可以了。读者可能已经意识到了，短语“前面找出的质数”意味着我们要在程序中使用数组，例如使用它保存每一个产生的质数。

为了进一步优化程序的效率，我们还可以使用另外一个定理：任何一个非质数的整数，肯定会有一个小于其平方根的质因数。也就是说，在判断某个数是否为质数时，我们只需要尝试那些小于其平方根的质数就可以了。

程序7.4使用了我们前面讨论的改进标准产生50以内的质数表。

程序7.4 产生质数表（修订版2）

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

// Modified program to generate prime numbers

int main (void)
{
    int p, i, primes[50], primeIndex = 2;
    bool isPrime;

    primes[0] = 2;
    primes[1] = 3;

    for ( p = 5; p <= 50; p = p + 2 ) {
        isPrime = true;

        for ( i = 1; isPrime && p / primes[i] >= primes[i]; ++i )
            if ( p % primes[i] == 0 )
                isPrime = false;

        if ( isPrime == true ) {
            primes[primeIndex] = p;
            ++primeIndex;
        }
    }

    for ( i = 0; i < primeIndex; ++i )
        printf ("%i ", primes[i]);

    printf ("\n");

    return 0;
}
```

程序7.4 输出

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

最内层循环中的语句

```
p / primes[i] >= primes[i]
```

用于判断`primes[i]`不超过`p`的平方根。关于这一点我们前面已经讨论过。（如果想要理解这个判断标准，读者可能需要一些数学知识）

程序7.4开始将数字2和3作为最初的两个质数保存在数组`primes`中。数组`primes`可以容纳50个元素，虽然我们并不会用到那么大的存储空间。变量`primeIndex`用来表示数组`primes`中下一个空闲的位置，该变量被初始化为2。接下来，程序使用循环遍历从5到50之间的所有奇数（译者注：注意循环变量每次递增2）。在循环中我们首先将标志变量`isPrime`设置为`true`，然后程序再进入内层循环。内层循环使用数组`primes`中保存的每一个质数去除当前的数字`p`。内层循环变量的起始值是1，因为我们没有将任何偶数作为质数的候选者，因此没有必要用第一个质数`primes[0]`（也就是2）去尝试作为除数。在内层循环中，如果`p`能够整除`primes[i]`，那么标志变量`isPrime`就被设置为`false`。内层循环继续的条件是：标志变量的值为`true`，而且`primes[i]`的值没有超过`p`的平方根。

在内层循环运行完成之后，我们检验标志变量`isPrime`。如果`p`是一个质数，我们就把它保存在`primes`数组的下一个空闲位置中。

在所有的候选者都被测试之后，我们使用循环打印出`primes`数组的内容。用于显示循环变量的变化范围是从0到`primeIndex-1`，因为`primeIndex`变量总是指向下一个空闲的位置。

数组初始化

正像我们在声明变量的时候可以给其赋以初值一样，我们也可以在声明数组的时候初始化其内容。要做到这一点，需要在声明数组的时候，从第一个元素开始逐个列出其初始值，并将这些值用逗号分隔开，最后用一对大括号将其包围起来即可。

下面的语句声明了一个有5个元素的整型数组，并将其中的每一个元素初始化为0。

```
int counters[5] = { 0, 0, 0, 0, 0 };
```

与之类似，下面的语句同样声明了5个元素的数组，并将0号元素初始化为0，1号元素初始化为1，依此类推。

```
int integers[5] = { 0, 1, 2, 3, 4 };
```


我们也可以使用类似的方式初始化字符数组，如下所示

```
char letters[5] = { 'a', 'b', 'c', 'd', 'e' };
```

这条语句定义了一个拥有5个元素的字符型数组letters，并将其中的5个元素分别初始化为'a'、'b'、'c'、'd'和'e'。

在声明数组的时候，我们并不需要给出和数组所能容纳元素数量同样多的初始值。如果给出的初始值小于数组所能容纳元素的个数，C语言则只初始化有限个数组元素。数组中的其他元素将被初始化为0。所以，下面的语句将数组sample_data的前三个元素分别初始化为100.0、300.0和500.5，并将剩余的元素初始化为0。

```
float sample_data[500] = { 100.0, 300.0, 500.5 };
```

我们可以在初始化值列表中使用方括号和下标指定我们要初始化哪些数组元素，这些下标可以按照任意的顺序排列，如下所示：

```
float sample_data[500] = { [2] = 500.5, [1] = 300.0, [0] = 100.0 };
```

该语句的效果和前面的语句相同。

下面的语句定义了一个拥有10个元素的整型数组，并将最后一个元素初始化为x+1（或者说1234），前三个元素分别初始化为1、2和3。

```
int x = 1233;
int a[10] = { [9] = x + 1, [2] = 3, [1] = 2, [0] = 1 };
```

不幸的是，C语言对于数组初始化没有提供任何快捷的形式。比如，我们不能声明一个有500个元素的数组，然后使用一个重复量将所有的数组元素都初始化为1。实际上，我们只能逐个写出这些初始值。这种时候，我们最好在程序中使用某种形式的循环来完成初始化的工作。

程序7.5演示了数组初始化的两种技术。

程序7.5 数组初始化

```
#include <stdio.h>

int main (void)
{
    int array_values[10] = { 0, 1, 4, 9, 16 };
    int i;

    for ( i = 5; i < 10; ++i )
        array_values[i] = i * i;

    for ( i = 0; i < 10; ++i )
        printf ("array_values[%i] = %i\n", i, array_values[i]);

    return 0;
}
```

程序7.5 输出

```
array_values[0] = 0
array_values[1] = 1
array_values[2] = 4
array_values[3] = 9
array_values[4] = 16
array_values[5] = 25
array_values[6] = 36
array_values[7] = 49
array_values[8] = 64
array_values[9] = 81
```

在声明数组array_values的时候，我们将它的前5个元素初始化为其对应下标的平方（例如，第3号元素被初始化为9）。下面，程序演示了如何使用for循环完成同样的初始化工作。这个循环设置了数组array_values第5号元素到第9号元素的值。程序的第二个循环遍历所有的数组元素，将它们值打印出来。

字符数组

程序7.6演示如何使用字符数组。但是在这个程序中，有一个技术点值得我们讨论，读者注意到了吗？

程序7.6 字符数组的使用

```
#include <stdio.h>

int main (void)
{
    char word[] = { 'H', 'e', 'l', 'l', 'o', '!' };
    int i;

    for ( i = 0; i < 6; ++i )
        printf ("%c", word[i]);

    printf ("\n");

    return 0;
}
```

程序7.6 输出

```
Hello!
```

在程序7.6中，最值得关注的是字符数组word的声明方式——我们在声明的时候并没有指定数组的长度。C语言允许我们在声明数组的时候不指定数组的长度。这个时候，数组的长度由初始化列表中值的个数决定。因为在程序6.7中，字符数组word的初始化列表中共有6个字符，所以C语言自动将该数组的长度设置为6。

如果在声明数组的时候可以决定数组中每一个元素的值，上面就是很好的声明数组的方式。如果不是这样的话，我们就必须显式指定数组的长度。

如果使用了带有下标的初始化列表形式，那么C语言使用初始化列表中最大下标决定数组的长度。例如下面的语句：

```
float sample_data[] = { [0] = 1.0, [49] = 100.0, [99] = 200.0 };
```

由于初始化列表中的最大下标是99，所以数组sample_data的长度是100。

使用数组完成基数转换

下面的程序演示了整数数组和字符数组的使用。这个程序将一个以10为基数的正整数转换为其他基数的数字，基数的范围可以从2到16。在程序开始，我们读入需要转换的整数和需要转换的相应基数，随后程序完成转换工作并打印转换结果。

编写这个程序的第一步是设计一个合适的算法，用于将以10为基数的数字转换为其他基数。这个算法可以描述如下：用该数字与基数进行求余操作，得出的结果就是转换结果的一位数字；随后，用该数字除以基数，丢弃结果的小数部分，并对所得的结果重复以上的步骤，直到结果为0。

上面的算法从最右边开始，逐个产生转换后的数字的每一位数。下面的例子演示了这个算法的工作过程。假定我们要将10转换为以2为基数的数字，表7.1列出了转换过程的每一个步骤。

表7.1 将以10为基数的整数转换为以2为基数

Number	Number Modulo 2	Number / 2
10	0	5
5	1	2
2	0	1
1	1	0

根据表7.1，我们将Number Modulo 2一列从下向上看，可以看出转换的结果是1010。

根据上面的算法，编写一个程序完成转换工作的话，我们还有几件事情需要考虑。首先，这个算法产生数字的顺序是反过来的，这一点让我们觉得有些不满意。我们当然不能期望读者从右向左或者从下向上阅读程序的输出。我们应该想办法修正这个问题。在计算转换结果的过程中，我们不是计算出一位数字，就将其显示出来，而是将其保存到一个数组中。这样，在计算完成之后，我们可以按照逆序显示该数组的内容，得到正确的结果。

其次，我们允许用户转换的最大基数为16。这样，我们就不能使用10—15这些数字显示转换的结果，而应该使用字母A-F。这是我们用得到字符数组的地方。

请读者认真研究一下程序7.7，看看它是怎样解决这两个问题的。这个程序还演示了C语言的一个新的类型修饰符const，这个修饰符用来表示某个变量的值不会改变。

程序7.7 将一个正整数转换为其他基数

```
// Program to convert a positive integer to another base

#include <stdio.h>

int main (void)
{
    const char baseDigits[16] = {
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
    int convertedNumber[64];
    long int numberToConvert;
    int nextDigit, base, index = 0;

    // get the number and the base

    printf ("Number to be converted? ");
    scanf ("%ld", &numberToConvert);
    printf ("Base? ");
    scanf ("%i", &base);

    // convert to the indicated base

    do {
        convertedNumber[index] = numberToConvert % base;
        ++index;
        numberToConvert = numberToConvert / base;
    }
```

程序7.7 续

```
    }  
    while ( numberToConvert != 0 );  
  
    // display the results in reverse order  
  
    printf ("Converted number = ");  
  
    for (--index; index >= 0; --index ) {  
        nextDigit = convertedNumber[index];  
        printf ("%c", baseDigits[nextDigit]);  
    }  
  
    printf ("\n");  
    return 0;  
}
```

程序7.7 输出

```
Number to be converted? 10  
Base? 2  
Converted number = 1010
```

程序7.7 输出（再次运行）

```
Number to be converted? 128362  
Base? 16  
Converted number = 1F56A
```

const 修饰符

C语言编译器允许我们使用const修饰符，标明那些在程序运行期间值不会变化的变量。也就是说，我们用这个修饰符告诉编译器，该变量的值在程序运行期间为常量。如果在程序中，我们在const变量初始化完成之后，试图给其设置新的值，增加或者减少其值，编译器就可能给出错误信息（虽然ANSI规范并不要求一定如此）。C语言中const修饰符的主要动机在于，编译器可以决定，将const变量放置到只读内存中。（一般说来，我们的程序指令也被放置到只读内存中。）

下面我们给出使用const修饰符的一个例子。

```
const double pi = 3.141592654;
```

上面的语句声明了一个const变量pi，这个语句告诉编译器，程序在运行过程中不会修改变量pi的值。因此，如果在后面的程序中我们又使用了如下的语句

```
pi = pi / 2;
```

编译器（以gcc为例）就会给出类似下面的警告信息：

```
foo.c:16: warning: assignment of read-only variable 'pi'
```

我们现在回头看看程序7.7，在该程序中，字符数组baseDigits用于保存转换后数值每一位上所有可能的16个数字。该数组被声明为const，因为一旦初始化完成，我们就不再需要修改数组元素的值。另外，使用const修饰符也可以提高程序的可读性。

我们定义数组convertedNumber的大小为64，在所有的计算机上，这个大小可以容纳最大的long整数转化为最小的基数(2)后的结果。变量numberToConvert被声明为long int型的整数，这样程序可以转换相当大的整数。最后，变量base（保存需要转换的基数）和index（convertedNumber数组的下标）都被声明为int类型的整数。

当用户输入需要进行转换的数字和基数之后，程序使用一个do循环完成转换工作。这里提醒读者注意：使用scanf读入长整型数的格式化输入符号是%ld。因为即使用户的输入是0，我们也要最少输出一位数字，所以这里使用了do循环。

在do循环的循环体中，我们计算numberToConvert和base相除的余数，将该余数保存到数组convertedNumber中，并相应的增加该数组的下标index。随后我们使用整数除法规则计算numberToConvert除以base的结果，如果该结果等于0，我们就结束循环，否则，循环将继续执行。

当循环结束之后，变量index中就保存着数组convertedNumber中转换后数字的个数。因为该变量在前面的do循环中多增加了1，所以在下面for循环的开始，它的值首先被减去1。这个for循环逆序遍历数组convertedNumber的每一个元素，将转换后的数字的每一位按照正确的顺序显示出来。

在for循环的循环体中，数组convertedNumber中的当前元素被赋给变量nextDigit。为了将数字10-15正确显示为A-F，我们使用nextDigit作为下标访问数组baseDigits中的元素。数字0-9在baseDigits中的对应元素是普通的字符'0'-'9'（读者在这里可以回忆一下数字0和字符'0'之间的区别），数字10-15对应的数组元素则是字符'A'-'F'。也就是说，如果nextDigit的值为10，则baseDigits[10]中存放的字符'A'被显示在终端上，如果nextDigit的值为8，那么baseDigits[8]中对应的字符'8'被显示出来。

当变量index的值小于0时，for循环就结束了。最后程序输出一个换行符并结束运行。

顺便提一下，在打印输出的printf语句中，我们也可以省略中间变量nextDigit，而直接在baseDigits数组的下标中使用表达式convertedNumber[index]。也就是说，我们在printf语句中可以直接使用下面的表达式：

```
baseDigits[ convertedNumber[index] ]
```

使用这个表达式，程序最后的输出结果是完全相同的。当然，这种方法写出的程序，比使用中间变量的写法看上去要复杂一些。

我们还要指出，程序7.7还有一些小缺陷——它对于用户输入的基数没有进行检查，用户输入的基数应该在2和16之间。如果用户输入0作为基数，那么在do循环中将会出现除以0的错误。我们当然不能允许这种情况。另外，如果用户输入的基数是1，那么程序就会进入一个无限循环，因为numberToConvert无论运算多少次，都不会变为0。还有，如果用户输入的基数大于16，我们在程序中使用baseDigits数组的时候下标就有可能超过允许的范围。这种问题我们同样努力避免，因为C语言本身并不会去检测这类错误。

在第8章“使用函数”中，我们将会重写这个程序，消除上面所说的缺陷。不过现在我们还是接着介绍关于数组其他一些有趣的扩展吧。

多维数组

迄今为止我们介绍的所有数组都是线性的——也就是说，它们只有一个维度。C语言允许定义任意维度的数组，本小节将介绍二维数组。

二维数组最常见的用途是处理矩阵。表7.2显示了一个4×5的矩阵。

表7.2 4×5的矩阵

10	5	-3	17	82
9	0	0	8	-7
32	20	1	0	14
0	0	8	7	6

在数学中，我们通常使用双下标来引用矩阵中的元素。所以，如果我们将前面的矩阵称为M的话，那么 M_{ij} 就代表第i行第j列上的那个元素，其中i的范围从1到4，j的范围从1到5。按照这个规则，我们很容易推断出 $M_{3,2}$ 的值等于20，也就是第3行、第2列上的那个数字，而 $M_{4,5}$ 代表第4行、第5列上的数字6。

我们可以用类似的方式引用C语言二维数组中的元素。不过，由于C语言通常从0开始计数，因此矩阵的第一行实际上在C语言中为第0行，矩阵第一列在C语言中为第0列。表7.3显示了C语言中二维数组的行列编号。

表7.3 C语言中的4×5矩阵

列 (j)	0	1	2	3	4
行 (i)					
0	10	5	-3	17	82
1	9	0	0	8	-7
2	32	20	1	0	14
3	0	0	8	7	6

与数学中的矩阵记法 M_{ij} 相类似，C语言中使用下面形式的表达式引用二维数组中的元素。

`M[i][j]`

上面的表达式中，第一个下标*i*代表行，第二个下标*j*代表列。因此，下面的表达式：

```
sum = M[0][2] + M[2][4];
```

实际上将第一行第三列的数字——-3和第三行第五列的数字14相加，并将结果11保存在变量sum中。

二维数组的声明方法与一维数组的方式类似，如下所示：

```
int M[4][5];
```

这个语句声明了一个拥有4行、5列，总共20个元素的二维数组，数组的每一个元素都是一个整型数。

我们也可以像一位数组那样初始化二维数组。在书写二维数组的初始列表时，所有的值按行写出，每一行的初始化数据用大括号括起来。例如，如果需要初始化向上面列出的二维数组，可以使用下面的初始化语句：

```
int M[4][5] = {
    { 10, 5, -3, 17, 82 },
    { 9, 0, 0, 8, -7 },
    { 32, 20, 1, 0, 14 },
    { 0, 0, 8, 7, 6 }
};
```


这个语句中有一点需要提醒读者注意：在界定每一行初始化数据的大括号后面，除了最后一行，都需要加上一个逗号。另外，初始化列表中的大括号实际上并不是必须的，如果没有大括号的话，C语言也将按照行来读入这些初始化值。因此，上面的语句也可以写成下面的形式：

```
int M[4][5] = { 10, 5, -3, 17, 82, 9, 0, 0, 8, -7, 32,
               20, 1, 0, 14, 0, 0, 8, 7, 6 };
```

与一维数组相类似，C语言允许我们在声明数组的时候只初始化部分元素。下面的这个语句只初始化二维数组每一行的前三个元素，其他剩余的元素将被初始化为0。

```
int M[4][5] = {
    { 10, 5, -3 },
    { 9, 0, 0 },
    { 32, 20, 1 },
    { 0, 0, 8 }
};
```

这里提醒读者注意，如果我们只初始化数组的一部分，那么初始化列表中的大括号就是必须的了。如果没有它们，C语言编译器就会将这些数值用于初始化前两行和第三行的前两个元素。（读者可以自己写程序来验证一下。）

在二维数组的初始化列表中，我们也可以使用下标。如下所示：

```
int matrix[4][3] = { [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

上面的语句只初始化列表中指定的元素，而其他的元素都将被初始化为0。

变量长度的数组¹

本小节将介绍C语言的一个特性，这个特性允许我们在声明数组的时候，不必限定它们的大小为常数。

在本章前面的例子中，我们已经看到如何声明具有某个特定长度的数组。C语言也允许我们使用变量指定数组的长度。在程序7.3中，我们仅仅计算前15个Fibonacci数。但是如果我们需要计算100个或者500个Fibonacci数，或者，我们需要用户输入需要计算的Fibonacci数的编号，那应该怎么办呢？读者可以仔细研究一下程序7.8，看看解决这个问题的方式。

¹在本书写作的过程中，并不是所有的编译器都完全支持变量长度数组这个新特性。在使用这个特性之前，请读者先检查一下自己使用的编译器的文档。

程序7.8 使用变量长度数组计算Fibonacci数

```
// Generate Fibonacci numbers using variable length arrays

#include <stdio.h>

int main (void)
{
    int i, numFibs;

    printf ("How many Fibonacci numbers do you want (between 1 and 75)? ");
    scanf ("%i", &numFibs);

    if (numFibs < 1 || numFibs > 75) {
        printf ("Bad number, sorry!\n");
        return 1;
    }

    unsigned long long int Fibonacci[numFibs];

    Fibonacci[0] = 0; // by definition
    Fibonacci[1] = 1; // ditto

    for ( i = 2; i < numFibs; ++i )
        Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];

    for ( i = 0; i < numFibs; ++i )
        printf ("%llu ", Fibonacci[i]);

    printf ("\n");

    return 0;
}
```

程序7.8 输出

```
How many Fibonacci numbers do you want (between 1 and 75)? 50
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229
832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817
39088169 63245986 102334155 165580141 267914296 433494437 701408733
1134903170 1836311903 2971215073 4807526976 7778742049
```

在程序7.8中有几点需要读者注意一下。在程序开始，我们声明变量*i*和*numFibs*。后者存储用户需要计算的Fibonacci数的编号。在开始计算之前，程序还要对用户输入的数字进行范围检查，这是一个很好的编程习惯。如果输入的数字超出范围（也就是小于1或者大于75），我们的程序仅仅打印出一条消息，然后执行return 1这个语句。C语言在程序执行的

时候遇到return语句，将会跳过所有剩余的语句，立刻结束程序的运行。正如在第3章“编译和运行第一个程序”中所叙述的那样，按照惯例，非0的返回值表示程序在运行中发生了错误，如果需要的话，其他的程序可以读取这个值。

在用户输入了需要计算的Fibonacci数的编号之后，我们使用下面的语句声明一个数组，这个数组用于存储计算出的Fibonacci数。

```
unsigned long long int Fibonacci[numFibs];
```

我们声明数组Fibonacci包含numFibs个数字，因为数组的长度由一个变量表示，而不是一个常量，所以我们这里实际上使用的是声明变量长度数组的语法。前面我们提到过，C语言允许在程序的任何位置声明变量，只要变量声明位于变量使用之前即可。所以这里的数组声明初看上去位置不正确，实际上在C语言中这是完全合法的。但是，按照编程惯例，我们通常在程序开始声明所有要用到的变量，这样阅读程序的人可以在同一个地方看到所有变量的类型。

因为Fibonacci数增长得非常快，所以我们使用了计算机所能允许的最大的整数类型，也就是unsigned long long int。作为一个练习，读者可以试一下在自己的计算机上，unsigned long long int类型的变量最大能够容纳的Fibonacci数是多少。

程序剩下的部分不需要再详细解释了。这些代码计算出所需的Fibonacci数，然后将其显示在终端上，程序随后终止执行。

C语言中有一种名为“动态内存分配”的技术，该技术允许程序在运行时为数组分配内存空间。这种技术涉及到使用标准库函数malloc和calloc。在第17章“杂项和高级特性”中，我们将会详细介绍这项技术。

到此为止，我们已经看到如何使用数组这一强有力的工具编写程序。实际上，几乎所有的编程语言都提供数组。在第8章“使用函数”中，我们还将介绍有关多维数组的另外一个例子。第8章详细讲述了C语言中最重要的概念之一——“函数”。但是，在继续学习下一章之前，请先完成下面的这些习题。

练习

1. 输入并运行本章的8个程序，并把自己的输出结果与书中给出的结果相比较。
2. 修改程序7.1，使用for循环将数组values中的元素初值设为0。
3. 程序7.2只允许用户输入20个调查结果。修改这个程序，以使用户能够输入任意多个调查分数，这样用户就不必计算列表中总共有多少个调查结果。另外，用户可以输入999表示

输入结束（提示：当读者需要退出循环的时候，可以使用break语句）。

4. 编写一个程序，计算拥有10个元素的浮点数数组中所有元素的平均值。

5. 下面程序的输出是什么？

```
#include <stdio.h>

int main (void)
{
    int numbers[10] = { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    int i, j;

    for ( j = 0; j < 10; ++j )
        for ( i = 0; i < j; ++i )
            numbers[j] += numbers[i];

    for ( j = 0; j < 10; ++j )
        printf ("%i ", numbers[j]);

    printf ("\n");

    return 0;
}
```

6. 在计算Fibonacci数的时候，我们不必一定使用数组——使用3个变量就可以完成计算过程，两个用于存储前两个Fibonacci数，还有一个用于存储当前的Fibonacci数。使用上述方式重写程序7.3。这里提醒读者注意，因为我们不再使用数组存储产生的Fibonacci数，因此每产生一个数字，我们需要立刻把它显示出来。

7. 还有一种有名的埃拉托色尼筛网法 (*Sieve of Erastosthenes*)，可以用来产生质数。这个算法的步骤列在下面。请读者编写一个程序，实现这个算法，并使用程序找出150以内所有的质数。将这个方法与我们在范例程序中使用的算法进行比较，读者能够得出哪些结论？

埃拉托色尼筛网法的步骤

- (1) 定义整数数组 P ，将所有的数组元素设置为0。
- (2) 设置变量 i 等于2。
- (3) 如果 $i > n$ ，算法结束。
- (4) 如果 P_i 等于0，那么 i 是一个质数。
- (5) 对于所有的正整数 j ，如果 $i * j \leq n$ ，将数组元素 P_{i*j} 设置为1。
- (6) 将 i 的值增加1，回到第3步。

Working with Functions

使用函数

所有良好的C语言程序都依赖于同一种语言要素——函数。迄今为止，我们遇到的每一个程序都使用了函数。printf和scanf都是函数的例子。实际上，每个程序都要包含一个名为main的函数。读者可能要问，函数到底是个什么样的东西，值得我们如此重视？实际上，使用函数可以让程序容易书写、阅读、被其他人理解、调试、修改乃至维护。很明显，如果一个东西能够同时完成上述所有这些事情，那么它确实值得我们大大地关注。

定义函数

首先，我们必须知道什么是函数，然后我们才能理解如何在编写程序的时候有效地使用它。让我们回头看看在本书中遇到的第一个程序（程序3.1），就是在屏幕上显示“Programming is fun”的那个。下面列出了它的代码。

```
#include <stdio.h>

int main (void)
{
    printf ("Programming is fun.\n");

    return 0;
}
```

下面是一个名为printMessage的函数定义，它完成的工作与程序3.1相同。

```
void printMessage (void)
{
    printf ("Programming is fun.\n");
}
```


printMessage函数与main函数相比, 仅仅第一行和最后一行有所不同。一个函数定义的第一行(按照从左到右的顺序)告诉编译器四件事情:

1. 谁能够调用这个函数(在第15章“处理大型程序”将会更多地讨论这个问题)
2. 函数返回值的类型
3. 函数的名字
4. 函数的参数

按照上面的规则, printMessage函数的定义告诉编译器该函数不返回任何值(第一个void关键字); 函数的名字是printMessage; 这个函数不需要任何参数(第二个void关键字)。我们随后将会更详细地介绍关键字void。

很明显, 与变量名的选择类似, 给函数选择一个有意义的名字是非常重要的。名字的选择往往大大地影响程序的可读性。

请读者回忆一下, 我们在第3章中讲过, main在C语言中是一个特殊的函数名字, 程序的执行总是从main函数开始。每一个程序都必须定义main函数。下面, 我们给前面的printMessage函数定义再加上适当的main函数, 以便形成一个完整的程序, 即程序8.1。

程序8.1 在C语言中定义函数

```

#include <stdio.h>

void printMessage (void)
{
    printf ("Programming is fun.\n");
}

int main (void)
{
    printMessage ();

    return 0;
}
    
```

程序8.1 输出

```

Programming is fun.
    
```

程序8.1中定义了两个函数: printMessage和main。C语言程序的执行总是从main开始。在main函数中, 有下面一条语句:

```

printMessage();
    
```


计算机如果遇到这条语句，那么就应该执行printMessage函数。printMessage后面的一对小括号告诉编译器，这个名字代表一个函数，在调用函数的时候，我们没有给函数传递任何参数（这和函数的定义是一致的）。当计算机执行函数调用的时候，程序的流程将跳转到该函数。在printMessage函数中，包含有一条printf语句，该语句在终端上打印出“Programming is fun.”。在打印出这条消息之后，printMessage函数的执行就结束了（正像右大括号所指示的那样），程序的执行流程重新回到main函数中调用printMessage的地方。这里我们提醒读者注意，我们也可以在printMessage函数的最后增加如下一条语句：

```
return;
```

因为printMessage被定义为不返回任何值，所以这个return语句后面没有数值。在不返回值的函数中，这个return语句是可有可无的，因为C语言在执行函数的时候，如果没有遇到return语句就到达了函数的结尾，其实际效果与执行没有返回值的return语句的作用是相同的——它们都将结束函数的执行。

如同我们前面提到的那样，在程序中调用某个函数并不是什么新概念。printf和scanf都是函数。它们与printMessage函数的主要区别在于，printf和scanf是标准函数库的一部分，我们不需要动手去编写它。当我们在程序中调用printf函数显示某些信息的时候，程序的执行就跳转到printf函数中，在该函数完成了相应的工作之后，计算机的执行流程重新回到我们的程序中。无论是调用标准库函数，还是调用自己编写的函数，在函数调用返回之后，计算机都将接着执行调用函数那条语句后面的语句。

程序8.2 调用函数

```
#include <stdio.h>

void printMessage (void)
{
    printf ("Programming is fun.\n");
}

int main (void)
{
    printMessage ();
    printMessage ();

    return 0;
}
```

程序8.2 输出

```
Programming is fun.  
Programming is fun.
```

程序8.2同样从main函数开始执行，该函数中调用了printMessage函数两次。当计算机执行第一个printMessage函数调用时，执行流程将跳转到printMessage函数中，该函数在终端上打印“Programming is fun.”，然后执行流程又回到了main函数。接下来在main函数中又是一个对printMessage函数的调用语句，结果计算机将该函数重新执行一次。当这次执行流程从printMessage函数中返回之后，程序的执行就结束了。

作为printMessage函数的最后一个例子，请读者研究一下程序8.3，并推断其输出是怎样的。

程序8.3 调用更多的函数

```
#include <stdio.h>  
  
void printMessage (void)  
{  
    printf ("Programming is fun.\n");  
}  
  
int main (void)  
{  
    int i;  
  
    for ( i = 1; i <= 5; ++i )  
        printMessage ();  
  
    return 0;  
}
```

程序8.3 输出

```
Programming is fun.  
Programming is fun.  
Programming is fun.  
Programming is fun.  
Programming is fun.
```

参数与局部变量

在前面的程序中，每次调用printf函数的时候，我们总是向函数传递了若干个值，第一个值通常是格式化字符串，后面的值则是需要显示的程序计算结果。这些值在C语言

中被称为函数参数。使用函数参数可以大大增加函数的灵活性。我们编写的printMessage函数每次总是显示相同的消息,而可以接收参数的printf函数则显示其需要显示的内容。

我们也可以定义接收参数的函数。在第5章“循环”中,我们开发了很多程序用于计算三角形参数。在学习了函数之后,就可以定义一个名为calculateTriangularNumber的函数,专门用于计算三角形参数。该函数接收一个参数,用于指定需要计算的三角形参数的编号。程序所完成的工作是计算这个三角形参数,并将其在屏幕上打印出来。程序8.4即是完成上述工作的。

程序8.4 计算第n个三角形数

```
// 计算第n个三角形数的函数

#include <stdio.h>

void calculateTriangularNumber (int n)
{
    int i, triangularNumber = 0;

    for ( i = 1; i <= n; ++i )
        triangularNumber += i;

    printf ("Triangular number %i is %i\n", n, triangularNumber);
}

int main (void)
{
    calculateTriangularNumber (10);
    calculateTriangularNumber (20);
    calculateTriangularNumber (50);

    return 0;
}
```

程序8.4 输出

```
Triangular number 10 is 55
Triangular number 20 is 210
Triangular number 50 is 1275
```

函数原型声明

下面详细解释函数`calculateTriangularNumber`。该函数的第一行如下：

```
void calculateTriangularNumber (int n)
```

这行语句被称为函数原型声明。它的作用是告诉编译器`calculateTriangularNumber`是一个函数，该函数不返回任何值（用关键字`void`标明），函数接受一个参数`n`，该参数是一个整型数。在函数的原型声明中，函数参数的正式名字是形式参数。形式参数的名字可以是任何C语言合法的变量名。有关C语言变量的取名规则，我们在第4章“变量、数据类型和算术表达式”中有过详细的介绍。显而易见的是，在编写程序的名字时，应该给形式参数尽可能起一个有意义的名字。

一旦定义了形式参数，就可以在函数体的任何地方引用这个形式参数。

函数原型声明下面的左大括号表示函数体正式开始，因为我们需要在函数中计算第`n`个三角形数，所以声明一个变量存放该数值。同样，我们还需要定义一个变量用作循环下标。为此，在函数体的开始声明了整型变量`triangularNumber`和`i`，用作上述目的。这些变量的初始化方法和使用方法与以前在`main`函数中的使用方法相同。

自动局部变量

在函数体内部的变量通常被称为自动局部变量。因为每次当函数被调用的时候，编译器自动创建这些变量，而且除了函数之外，程序的其他部分无法访问这些变量。如果在函数中声明变量的时候给予其赋予初值，那么每次调用该函数的时候，该变量都会被重新赋予该初值。

从更规范一些的角度来讲，每次在函数体内定义变量的时候，都应该使用`auto`这个关键字。下面是使用该关键字的一个例子。

```
auto int i, triangularNumber = 0;
```

因为C语言编译器默认任何在函数体内部定义的变量都是自动局部变量，所以在编程实践中，我们通常省略`auto`这个关键字，在本书中也是如此。

让我们回到前面`calculateTriangularNumber`这个函数，在定义完局部变量之后，程序计算出指定的三角形参数，并将其打印出来。随后的右大括号表示了函数的结束。

在`main`函数中，当第一次调用`calculateTriangularNumber`函数的时候向其传递参

数10。随后，程序的执行流程转到了该函数中，这时，函数体中的形式参数n的值就等于10。于是，函数将完成计算和显示第10个三角形参数的工作。

当我们第二次调用calculateTriangularNumber函数时，我们传递的参数是20。这次，calculateTriangularNumber函数体中的形式参数n的值就等于20。函数于是计算并显示了第20个三角形参数的值。

为了给出可以接受多个参数的函数的例子，我们下面使用函数的形式重写了前面计算最大公约数的那个例子（程序5.7）。函数的两个参数则是我们需要找出最大公约数的那两个整数。请看程序8.5。

程序8.5 计算最大公约数（修订版）

```
/* 计算两个整数的最大公约数的函数 */

#include <stdio.h>

void gcd (int u, int v)
{
    int temp;

    printf ("The gcd of %i and %i is ", u, v);

    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }

    printf ("%i\n", u);
}

int main (void)
{
    gcd (150, 35);
    gcd (1026, 405);
    gcd (83, 240);

    return 0;
}
```

程序8.5 输出

```
The gcd of 150 and 35 is 5  
The gcd of 1026 and 405 is 27  
The gcd of 83 and 240 is 1
```

函数gcd需要接受两个整型形式参数，这两个参数的名称分别为u和v。在函数体中，我们首先声明一个整型变量temp，随后我们打印出参数u和v的值以及适当的提示信息。函数随后开始计算并显示u和v的最大公约数。

读者可能会觉得比较奇怪：为什么在gcd函数中有两条printf语句？因为我们在循环中将要改变变量u和v的值，所以必须在进入循环之前就将其值打印出来。等到计算出了最大公约数之后，这些形式参数的值已经和原来传递给它们的值不一样了。我们也可以采用另外一种方式来解决这个问题，那就是使用两个变量存储u和v的初值，然后在程序的最后，使用一条printf语句将所有信息都打印出来。

函数的返回值

在程序8.4和程序8.5中，我们定义的函数直接进行必要的计算，并在终端上显示计算的结果。然而，很多时候我们并不想显示计算的结果，为此，C语言提供了一种方便的机制——函数返回值。利用这种机制，一个函数可以将某个值返回给它的调用者。这种机制我们并不陌生，因为在前面的程序中，所有的main函数都使用了返回值。在函数中返回一个值通常采用如下的语法形式：

```
return 表达式 (expression);
```

这个语句将表达式expression的值返回给函数的调用者。有些程序员习惯在表达式expression的周围加上小括号，这属于个人的编程风格，这些小括号本身并不是必须的。

如果要从函数中返回一个值，仅仅使用return语句还不够。还需要在声明函数的时候声明函数返回值的类型。这个声明应该放在函数名之前。本书前面所有的程序中定义的main函数，它的返回值都是int类型，因此，所有的main函数名前面都有一个关键字int。像下面这样的函数声明语句：

```
float kmh_to_mph(float km_speed)
```

定义了一个名为kmh_to_mph的函数，这个函数接收一个float类型的参数，其返回值也是float类型。以此类推，下面的函数声明：

```
int gcd(int u, int v)
```


定义了一个函数gcd，该函数接收两个整型的参数u和v，并返回一个整型的值。我们可以利用这种方式修改程序8.5，不是在函数gcd中显示出最大公约数，而是将其作为函数返回值传递给主程序，如程序8.6所示。

程序8.6 找出最大公约数并将其返回

```
/* 找出两个非负整数的最大公约数，并返回结果的函数 */
#include <stdio.h>

int gcd (int u, int v)
{
    int temp;

    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }

    return u;
}

int main (void)
{
    int result;

    result = gcd (150, 35);
    printf ("The gcd of 150 and 35 is %i\n", result);

    result = gcd (1026, 405);
    printf ("The gcd of 1026 and 405 is %i\n", result);

    printf ("The gcd of 83 and 240 is %i\n", gcd (83, 240));
    return 0;
}
```

程序8.6 输出

```
The gcd of 150 and 35 is 5
The gcd of 1026 and 405 is 27
The gcd of 83 and 240 is 1
```

当我们在函数gcd中计算出最大公约数之后，计算机将执行下面的语句：

```
return u;
```

这条语句将计算出的最大公约数返回给主程序。

读者可能要问，我们应该如何处理函数的返回值呢？在程序8.6的主程序中我们可以看到，在前面两个算例中，函数的返回值被保存在变量result中，更确切地说，下面的语句：

```
result = gcd(150, 35);
```

调用了函数gcd，并将150和35作为参数传递给它，在函数执行完成之后，其返回值被保存在变量result之中。

我们不一定必须把函数的返回值保存在某个变量中，在main函数的第三个算例中，我们把函数gcd的返回值直接传递给printf函数，printf函数随后将其结果显示出来。

C语言的函数只能够返回一个值。不像其他的编程语言，C语言对于函数和过程并不区分，C语言中只有函数，每个函数可以返回一个值，或者不返回值。如果我们在声明函数的时候不指定函数的返回值类型，那么编译器就假定该函数返回一个整型数——如果这个函数最终返回了一个值的话。某些程序员利用C语言的函数默认返回值是整型这一点，有时会省略对于函数返回值类型的声明，这是一种不好的编程习惯，应该尽力避免。如果一个函数返回某个值给它的调用者，我们一定要在函数的声明中明确写出，这样可以增加程序的可读性。如果我们坚持这样做的话，通过函数声明，我们不仅可以得知函数的名称、参数的个数及其类型，还可以知道函数是否返回一个值以及返回值的类型。

前面已经提到过，如果在函数声明前面加上void关键字，就是通知编译器，这个函数不返回值。如果我们随后将这个函数用于表达式中（这要求函数返回一个值），那么编译器就会给出错误信息。例如，因为程序8.4中的函数calculateTriangularNumber函数不返回值，在该函数的声明语句中，我们在函数名前面加上了关键字void。随后如果我们像下面这条语句那样使用该函数，编译器就会报告错误信息。

```
number = calculateTriangularNumber(20);
```

从某种意义上来说，void类型就是没有数据类型。因此如果一个函数声明其返回值是void类型，就是说该函数并不返回值，因此不能用在表达式中。

在第6章“进行判断”中，我们编写了一个程序，计算某个数字的绝对值并打印出来。下面，我们使用函数计算数字的绝对值，并将结果作为函数的返回值。在程序6.1中计算的是整型数的绝对值，这里我们改进一下，计算浮点数的绝对值。与之类似，函数的返回值也是浮点数，示例如程序8.7。

程序8.7 计算绝对值

```
/*计算绝对值的函数*/

#include <stdio.h>

float absoluteValue (float x)
{
    if ( x < 0 )
        x = -x;

    return x;
}

int main (void)
{
    float f1 = -15.5, f2 = 20.0, f3 = -5.0;
    int i1 = -716;
    float result;

    result = absoluteValue (f1);
    printf ("result = %.2f\n", result);
    printf ("f1 = %.2f\n", f1);

    result = absoluteValue (f2) + absoluteValue (f3);
    printf ("result = %.2f\n", result);

    result = absoluteValue ( (float) i1 );
    printf ("result = %.2f\n", result);

    result = absoluteValue (i1);
    printf ("result = %.2f\n", result);

    printf ("%f\n", absoluteValue (-6.0) / 4 );
    return 0;
}
```

程序8.7 输出

```
result = 15.50
f1 = -15.50
result = 25.00
result = 716.00
result = 716.00
1.50
```

函数absoluteValue相当简单，函数首先对于形式参数x与0进行比较，如果x小于0，我们就对x取反，随后函数使用return语句将计算出的绝对值返回给调用者。

在这个程序中，有几点需要读者注意：在第一次调用定义的函数时，我们将变量f1的值初始化为-15.5，然后将其传递给函数absoluteValue。在函数体中，该值被赋给变量x，由于x的值小于0，所以函数中的语句对x的值求反，并将结果保存在x中。也就是说，这个时候变量x的值等于15.5。随后，函数将x的值返回给main函数，在main函数中，返回结果被保存在变量result中，随后我们再使用printf语句将result的值打印出来。

当我们在函数absoluteValue中改变形式参数x的值的时候，绝对不会影响到变量f1的值。实际上，当我们把变量f1作为参数传递给函数absoluteValue的时候，系统会自动将该变量的值拷贝到形式参数x中。随后，在函数之中对于形式参数x的任何修改，都不会影响到变量f1。main函数中的第二个printf语句证明了这一点，读者可以看到它输出的f1的值还是原来的-15.5。在这里读者务必要领会下面的原理：一个函数永远无法修改它的参数的值，它只能修改这些参数的拷贝。

主程序后面的函数调用显示了我们可以将函数的返回值用于算术表达式之中。我们将变量f2的绝对值和变量f3的绝对值加在一起，并将结果赋给变量result。

主程序对于函数absoulteValue的第四个调用说明了下面的事实：我们传递给函数的参数应该和函数声明的参数类型一致。由于函数absoluteValue期望的参数是一个浮点数，因此我们在调用函数之前，首先使用类型转换操作符将整型变量i1转换为浮点数。如果我们在程序中没有这样做的话，那么编译器将会自动进行类型转换（编译器根据函数的声明已经得知函数需要的参数类型）。第五个函数调用演示了这种情况。当然，在程序中我们也可以手动进行类似的转换，而不是依赖系统完成这些工作，可以使程序显得更清楚一些。

主程序对于absoluteValue函数的最后一个调用说明，编译器在对函数的返回值进行处理的时候，依然使用我们前面讨论过的算术表达式求值规则。由于该函数返回的值是一个浮点数，因此编译器将除法表达式看作是一个浮点数除以一个整型数。由于除法操作的操作数之一是一个浮点数，因此整个除法操作都是用浮点算术规则。按照这个规则，-6.0

的绝对值除以4的结果是1.5。

到这里我们已经定义了计算绝对值的函数，在将来的程序中我们可以按照需要使用这个函数。实际上，下一个程序（程序8.8）就使用了这个函数。

函数调用……

现在，计算器已经成为一种像钟表一样常见的物品。如果我们需要计算某个数字的平方根，使用计算器完成这项工作将是举手之劳。但是就在数年前，学生在学校中都要学习如何手工计算某个数字的平方根的近似值。在这些手工计算方法中，最容易的一种近似方法是Newton-Raphson（译者注：牛顿—拉普森）迭代法。在程序8.8中，我们使用这种方法计算某个数的平方根。

Newton-Raphson方法的主要步骤如下：首先，对于给定的数字，我们猜测一个可能的平方根。我们猜测的数字越接近于实际的平方根，那么随后需要进行的计算就越少。为了讨论方便，我们假定读者并不“善于”猜测，这样我们总是采用1作为最初的猜测值。

随后，我们使用需要计算平方根的那个数除以最初猜测的值，然后将所得的商与我们最初猜测的值相加，再把所得的和除以2。我们将除法所得的结果作为我们新的猜测的平方根，重新执行上面的步骤。也就是说，把这个数再和需要计算平方根的数相加，除以2，然后再开始下一轮。

我们并不想将上述步骤永远执行下去，因此这里需要预先设定停止计算的条件。在不断重复上述计算步骤之后，我们猜测得到的数值将会越来越接近真正的平方根，我们可以设定一个极限值，比如 ϵ ，如果我们猜测的数字的平方与需要计算平方根的那个数字之间的差小于这个极限值 ϵ ，就停止计算。

我们对上述过程总结一下，将其列成下面的算法步骤：

第1步：设置guess最初的猜测值为1。

第2步：如果 $|guess^2 - x| < \epsilon$ ，执行第4步。

第3步：设置变量guess的值等于 $(x / guess + guess) / 2$ ，重新执行第2步。

第4步：变量guess的值就是所需的平方根。

在第二步的时候，我们需要将 guess^2 与 x 的差的绝对值与 ϵ 相比较，因为 guess 可能从任意的方向接近 x 的平方根。

到此为止，我们已经设计出了计算某个数字平方根的算法，编写一个程序来实现这个算法则是一件相对简单的事。 ϵ 的值的选取是任意的，在程序8.8中，我们将其定为0.0001。

程序8.8 计算某数的平方根

```
// 计算某数平方根的函数

#include <stdio.h>

float absoluteValue (float x)
{
    if ( x < 0 )
        x = -x;
    return (x);
}

//计算某数平方根的函数
float squareRoot (float x)
{
    const float epsilon = .00001;
    float guess = 1.0;

    while ( absoluteValue (guess * guess - x) >= epsilon )
        guess = ( x / guess + guess ) / 2.0;

    return guess;
}

int main (void)
{
    printf ("squareRoot (2.0) = %f\n", squareRoot (2.0));
    printf ("squareRoot (144.0) = %f\n", squareRoot (144.0));
    printf ("squareRoot (17.5) = %f\n", squareRoot (17.5));

    return 0;
}
```


程序8.8 输出

```
squareRoot (2.0) = 1.414216  
squareRoot (144.0) = 12.000000  
squareRoot (17.5) = 4.183300
```

在读者自己的计算机上，使用这个程序计算出的平方根最后几位可能和这里列出的有所不同。

下面对前面的程序进行详细的分析。首先，我们定义了函数absoluteValue，这个函数与程序8.7中的函数相同。

接下来，我们定义了函数squareRoot，这个函数接收一个浮点型参数x，其返回值也是浮点数。在函数体中，我们首先定义两个变量epsilon和guess，其中，变量epsilon用于保存判断循环结束的极限条件值，其初值被设置0.00001。我们将初始猜测的值定为1，将其保存在变量guess中。每次我们调用函数squareRoot时，这两个变量的值都将被重新初始化。

在声明了局部变量之后，我们使用while循环开始计算平方根。只要 guess^2 与x的差的绝对值大于或者等于epsilon，循环就一直执行下去。在循环体中，我们计算了表达式

```
guess * guess - x
```

的值，并将其传递给函数absoluteValue，然后再将该函数返回的值与epsilon进行比较。如果返回值的绝对值大于或者等于epsilon，就意味着我们计算出的平方根的精度还没有达到要求，为此，我们将继续执行循环。

最终，变量guess中包含的值达到了我们所需要的精度，于是我们退出循环，将该变量的值作为返回值返回给调用者。在主程序中，我们将返回值传递给printf函数，将其显示出来。

读者可能已经注意到，函数absoluteValue和squareRoot的形式参数的名字都是x。C语言编译器会自动区分它们，而不会产生混乱。实际上，每个函数都有自己的形式参数，因此函数absoluteValue中的形式参数x与函数squareRoot中的形式参数x是完全不同的。

这个规则对于函数内部的局部变量也是适用的。在多个函数内，我们可以任意声明多个同名的变量。由于一个函数内部定义的变量，也就是局部变量，只能在那个函数内部被访问，因此C语言编译器不会把它们搞混。也可以说，函数内部定义的变量，其作用域范围就是该函数。（在第11章“指针”中，我们将会看到，C语言提供了一种方法，可以在函数的外部存取该函数内部的局部变量）。

根据以上讨论可以知道，当程序在函数`squareRoot`中，将表达式`guess2 - x`的值传递给函数`absoluteValue`时，函数`absoluteValue`其形式参数`x`值的变化对于函数`squareRoot`中形式参数`x`的值没有任何影响。

声明返回值类型以及参数类型

前面我们已经提到过，C语言编译器默认情况下假定函数的返回值是`int`类型。更特殊一些，除非出现了如下两个条件之一，否则C语言的编译器在遇到函数调用的时候，将假定函数的返回值是整型数。

1. 在函数调用语句之前，编译器已经“看到了”该函数的具体定义。
2. 在函数调用语句之前，编译器已经“看到了”该函数的声明。

在程序8.8中，在我们调用函数`absoluteValue`之前，已经定义了该函数。因此，编译器在遇到调用语句的时候已经知道该函数的返回值是`float`类型。如果我们把函数`absoluteValue`的定义放在函数`squareRoot`的后面，那么编译器在遇到调用`absoluteValue`的语句时，就会先假定该函数的返回值是整型。随后，当编译器真正看到函数定义的时候，绝大部分现代的C语言编译器都会给出错误或者警告信息。

如果我们想要将函数`absoluteValue`定义在函数`squareRoot`的后面（甚至是另外一个文件中——参见第15章），必须在调用该函数之前，预先声明该函数的返回值类型。`absoluteValue`函数的声明可以放在`squareRoot`中，也可以放在所有的函数之外。在后一种情况下，通常将函数声明放在程序的开始。

函数的原型声明不但可以告诉编译器函数的返回值类型，还包括函数需要的参数个数以及它们的数据类型。

为了声明符号`absoluteValue`是一个函数，其返回值是浮点数，该函数需要一个浮点类型的参数，我们可以使用如下的声明语句：

```
float absoluteValue(float);
```

正像读者看到的那样，在声明函数原型的时候，我们可以只给出函数参数的类型而不必指定其名字。我们也可以在函数声明的时候给出参数的“哑”变量名，这个名字不必和函数定义时使用的参数名称相同，编译器将忽略这里给出的名字。

为了防止拼写错误，我们可以简单地将函数定义的第一行拷贝作为函数的声明。另外，读者需要记住在函数声明的后面有一个分号。

如果函数不需要任何参数，我们可以在参数括号之间使用关键字void。我们也需要在声明的时候指出这一点，以防止编译器将函数用于表达式中。下面是函数声明的例子：

```
void calculateTriangularNumber( int n);
```

如果函数接收不定数目的参数（比如printf函数和scanf函数），我们可以用下面的方式通知编译器：

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

上面的声明告诉编译器，printf函数的第一个参数是字符型指针（关于指针我们随后还会更多的加以说明），在这个参数后还可以有任意多个参数（用...说明）。文件stdio.h中包含了printf函数和scanf函数的原型声明，这也是我们为什么要在我们源文件开始放上如下代码

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

的原因。

如果没有这些声明，编译器将会假定printf函数和scanf函数接收固定多个参数，并生成不正确的可执行代码。

编译器只有在已经看到了函数的定义或者函数原型的情况下，才能在函数调用的时候进行适当的类型转换，因此，函数声明是非常重要的。

下面是关于在C语言中使用函数的一些要点，供读者参考：

1. C语言的编译器默认假定函数的返回值是int类型。
2. 如果我们的函数返回值是int类型，请明确定义它。
3. 如果函数不返回值，请用void关键字明确说明。
4. 只有预先定义或者声明了函数，编译器才能自动对函数调用的参数进行类型转换。
5. 为了安全起见，请预先声明程序中的所有函数，即使它们的定义在其调用之前（因为读者有可能以后又将这些函数移动到文件的其它位置甚至是其他文件中）。

检查函数的参数

如果需要计算负数的平方根，我们不得不扩展数的领域，引入虚数的概念。与之类似，如果我们给函数squareRoot传递一个负数作为参数，将会发生些什么呢？在这种情况下，Newton-Raphson方法将会永不收敛，也就是说，每次循环时候，变量guess的值不会变得更接近我们需要的平方根。因此，循环结束的条件永远都不会满足，程序将进入一个无限循环。为了结束程序，我们必须在终端上按下某些特殊的键（比如Ctrl+C）。

很明显，我们需要改进原来的程序，增加对于这种情况的处理。我们可以把这些任务交给调用者，要求它们必须给函数squareRoot传递一个非负的参数。虽然这种解决方案初看上去有一定道理，但是它有重大的缺陷。总有一天，我们会在调用squareRoot函数的时候，忘记检查调用参数是否为负值，于是，我们的程序就将陷入无限循环之中，从而不得不使用前面所说的办法来强行终止它。

把检查参数是否为负数的代码放置到squareRoot函数内部是一个更安全、更明智的办法。采用这个办法，函数为所有调用设置了一层保护。一种可行的检查办法是在函数内部将x与0进行比较，如果x为负数，就可以打印一条（可选的）出错信息，然后立刻返回到调用者。为了表明函数squareRoot没有正常地完成工作，我们可以向调用者返回一个正常情况下不可能返回的值。¹

下面是改写过的函数squareRoot，这个版本中我们增加了对于参数的检查，还在适当的位置包括了函数absoluteValue的原型声明。

```

/* 计算某数平方根的函数，如果传入负数，将显示一条消息并返回-1 */

float squareRoot (float x)
{
    const float epsilon = .00001;
    float guess = 1.0;
    float absoluteValue (float x);

    if ( x < 0 )
    {
        printf ("Negative argument to squareRoot.\n");
        return -1.0;
    }

    while ( absoluteValue (guess * guess - x) >= epsilon )
        guess = ( x / guess + guess ) / 2.0;

    return guess;
}
    
```

如果我们给这个函数传递一个负数，函数将打印出报错信息，并将-1.0作为返回值返回给调用者。如果参数非负，函数就执行正常计算的过程。

¹ C语言的标准库函数sqrt在接收一个负数参数时，将返回错误值domain error。这个符号的具体值是“和”实现相关的。在某些系统中，如果我们试图显示这个数字，将得到结果nan，意思是不是一个数字（not a number）。

如同读者在上面的squareRoot的修改版中看到的那样（在第7章“使用数组”也有这一现象），在单个函数中可以有多条return语句。当程序遇到return语句的时候，控制流将会立刻回到调用者，return语句后面的所有语句都不再被执行了。对于那些不返回任何值的函数来说，return语句的这种用法非常方便。我们使用如下简单形式的语句就可以达到效果。

```
return;
```

很明显，对于需要返回某个值的函数来说，就不能使用这种形式的return语句。

自顶向下的程序设计

一个函数可以调用另外一个函数，被调用的函数还可以调用其它的函数，依次下去，最终我们就得到一个结构良好的程序骨架。程序8.8的main函数调用了squareRoot函数若干次，在这个程序中，所有计算平方根的步骤都由函数squareRoot负责处理，main函数不需要理会这些算法细节。因此，只要我们正确声明了函数squareRoot的原型，就可以在具体编写函数squareRoot之前就调用它。

随后，当我们开始编写squareRoot函数的时候，还可以应用这种自顶向下的设计方法：我们可以声明absoluteValue函数并调用它，不需要操心这个函数的实现细节。在编写squareRoot函数的时候，我们只需要确信将来我们能够书写函数absoluteValue，用它来计算某个数的绝对值。

这种程序设计的方法，不仅能帮助我们更容易地开发某个程序，也能够增加程序的可读性。比如，如果一个程序员阅读程序8.8的代码，他并不需要仔细地阅读计算平方根的算法实现细节，就很容易知道这个程序计算了三个数字的平方根并将其显示出来。如果该程序员想要了解该程序的更多细节，他可以更仔细地阅读squareRoot函数中的代码。上面的讨论同样适用于squareRoot函数中的absoluteValue函数调用：在阅读函数squareRoot的代码时，程序员也不需要了解函数absoluteValue的更多细节。这些细节都被absoluteValue函数封装起来，只有在必要的时候，才会被进一步研究。

函数与数组

如同普通变量一样，在C语言中我们也能够将数组元素甚至是整个数组作为参数传递给函数。如果要将单个数组元素作为参数传递给某个函数（正如在第7章中我们将数组元素作为参数传递给printf函数，将其显示出来一样），我们只要使用普通的数组元素表达式就可以了。因此，如果我们要计算数组元素averages[i]的平方根，并将结果保存在变量

sq_root_result中, 可以使用如下的语句:

```
sq_root_result = squareRoot( averages[i] );
```

在函数squareRoot的内部, 我们不需要针对数组元素采用任何特殊的措施。如同普通的变量一样, 该数组元素的值被拷贝到形式参数之中, 然后在函数中正常使用。

将整个数组作为参数传递给函数与传递单个变量或者数组元素完全不同。如果我们要把一个数组传递给某个函数, 需要在参数调用的位置写上数组的名字, 不需要添加任何下标表达式。例如, 假定gradeScores是一个长度为100的数组, 如果我们要把这个数组作为参数传递给minimum函数, 可以使用下面的表达式:

```
minimum (gradeScores);
```

另一方面, 为了让minimum函数能够接收数组形式的参数, 我们必须使用恰当的函数声明。如果我们要声明参数为数组的函数, 可以采用下面的形式:

```
int minimum (int values[100])
{
    ...
    return minValue;
}
```

上面的语句定义了minmum函数, 该函数接收一个长度为100的整型数组作为参数, 并返回一个整型数。在函数内部, 我们将形式参数values当作一个整型数组, 对于该数组的任何元素的引用, 都将对应到实际的数组 (也就是作为参数的那个数组) 的元素。例如, 如果我们在函数minmum中使用表达式values[4], 实际上对应于数组元素gradeScores[4]。

作为使用数组作为函数参数的第一个例子, 我们编写一个名为minmum的函数, 用于找出一个长度为10的整型数组中的最小值。程序8.9还加上必要的main函数, 以构成完整的程序。

程序8.9 找出数组中的最小元素

```
//找出数组中最小元素的函数

#include <stdio.h>

int minimum (int values[10])
{
    int minValue, i;

    minValue = values[0];

    for ( i = 1; i < 10; ++i )
        if ( values[i] < minValue )
            minValue = values[i];
}
```

程序8.9 续

```
    return minValue;
}

int main (void)
{
    int scores[10], i, minScore;
    int minimum (int values[10]);

    printf ("Enter 10 scores\n");

    for ( i = 0; i < 10; ++i )
        scanf ("%i", &scores[i]);

    minScore = minimum (scores);
    printf ("\nMinimum score is %i\n", minScore);

    return 0;
}
```

程序8.9 输出

```
Enter 10 scores
69
97
65
87
69
86
78
67
92
90
Minimum score is 65
```

在main函数中，我们首先应该注意minimum函数的原型声明。这个原型声明语句告诉编译器minimum函数接收一个长度为10的整型数组作为参数，其返回值为整型数。读者也

许还记得我们前面讲过的语法规则：因为在程序的开始处已经定义了函数`minimum`，所以这里的函数声明语句并不是必须的。然后，为了安全起见，在本书所有的后续程序中，我们都将坚持使用声明函数原型的方式。

在主程序开始处，我们定义了长度为10的整型数组，随后，提示用户输入10个整型数。这10个整型数通过一个循环读入，循环变量的变化范围是从0到9，每次循环使用`scanf`函数读取下标与循环变量对应的数组元素。当我们读取了所有的值之后，程序调用`minimum`函数，并将数组作为参数传递给它。

在函数`minimum`中，我们使用形式参数`values`代表传递进来的数组。`values`被声明为一个长度为10的整型数组。局部变量`minValue`用于存储数组中的最小值，它的初值被设置为等于数组的0号元素`values[0]`。接下来，我们使用循环将数组中的其他元素依次与变量`minValue`进行比较，如果数组元素`values[i]`的值小于`minValue`，那么我们就找到了一个新的最小值，我们将`minValue`的重新设置为`values[i]`，然后继续进行循环。

当循环执行完毕之后，我们将变量`minValue`的值返回给主程序，该值随后被显示出来。程序运行结束。

在编写了通用的`minimum`函数之后，我们可以用它来找出任何长度为10的整型数组中的最小值。如果我们手头有5个长度为10的整型数组，我们可以简单地调用函数`minimum`五次，分别找出这些数组中的最小值。另外，我们还可以定义其他的便利函数完成类似的任务。例如，找出最大值的函数、找出平均值的函数、找出中间值的函数等等。

通过定义一组小巧、独立、并能够完成事先良好界定的单个任务的函数，我们可以在这组函数上建立更加复杂的程序，并将这组函数应用于其他相似的任务之中。比如，我们可以定义一个名为`statistics`的函数，该函数接收一个数组作为参数。`statistics`函数在其内部可以依次调用平均值函数`mean`，标准方差函数`standardDeviation`等等，用于找出某个数组的统计数据。上面介绍的程序设计方法是开发易于编写、阅读、修改和维护的应用程序的一个关键因素。

当然，我们前面编写的`minimum`函数并不是太通用，因为它只能处理长度为10的数组。这个缺陷很容易改正。我们可以给该函数增加一个参数，用于表示传递进来的数组参数的长度，另外，我们还要从函数的原型声明中去掉形式参数数组的长度。实际上，所有的C语言编译器都会忽略关于形式参数数组长度的声明，编译器只关心该函数的形式参数的类型是一个数组，而不需要了解数组的具体长度。

程序8.10对于程序8.9进行了改进，这个程序中的`minimum`函数可以找出任意长度数组中的最小值。

程序8.10 找出数组中的最小值（修订版）

//找出数组中的最小值的函数

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

```
int minimum (int values[], int numberOfElements)
```

```
{
```

```
    int minValue, i;
```

```
    minValue = values[0];
```

```
    for ( i = 1; i < numberOfElements; ++i )
```

```
        if ( values[i] < minValue )
```

```
            minValue = values[i];
```

```
    return minValue;
```

```
}
```

```
int main (void)
```

```
{
```

```
    int array1[5] = { 157, -28, -37, 26, 10 };
```

```
    int array2[7] = { 12, 45, 1, 10, 5, 3, 22 };
```

```
    int minimum (int values[], int numberOfElements);
```

```
    printf ("array1 minimum: %i\n", minimum (array1, 5));
```

```
    printf ("array2 minimum: %i\n", minimum (array2, 7));
```

```
    return 0;
```

```
}
```

程序8.10 输出

```
array1 minimum: -37
```

```
array2 minimum: 1
```

在程序8.10中，minimum函数接收两个参数：第一个参数是我们需要找出最小值的数组，第二个参数是数组的长度。在函数的头部，形式参数values后面的一对方括号通知编译器，该参数是一个整型数组。如前所示，编译器并不需要知道该数组的大小。

在函数的for循环中，形式参数numberOfElements代替了常数10作为循环的上限。因此for循环的循环变量变化范围是从1到numberOfElements-1，刚好遍历了所有的数组元素。

在main函数中定义了两个数组array1和array2，它们分别包含5个和7个数组元素。

在第一个printf语句中嵌入了对于minimum函数的调用，调用的参数分别是数组array1和常数5。常数5在这里用于表示数组的长度。minimum函数找出该数组的最小值-37，随后printf函数将其输出到终端上。当我们第二次调用minimum函数的时候，用数组array2及其长度7作为参数。这次minimum函数的返回值是1，随后，printf函数将结果打印出来。

赋值表达式

下面请读者研究一下程序8.11。在查看程序的实际输出之前，请读者先预测一下程序的输出结果。

程序8.11 在函数中改变数组元素

```
#include <stdio.h>

void multiplyBy2 (float array[], int n)
{
    int i;
    for ( i = 0; i < n; ++i )
        array[i] *= 2;
}

int main (void)
{
    float floatVals[4] = { 1.2f, -3.7f, 6.2f, 8.55f };
    int i;
    void multiplyBy2 (float array[], int n);

    multiplyBy2 (floatVals, 4);

    for ( i = 0; i < 4; ++i )
        printf ("%0.2f ", floatVals[i]);

    printf ("\n");
    return 0;
}
```

程序8.11 输出

2.40 -7.40 12.40 17.10

当读者查看程序8.11的时候，注意力肯定被下面的表达式所吸引：

```
array[1] *= 2;
```

这个“乘等”操作符的作用，是将操作符左面的表达式的值与操作符右面的表达式的值相乘，并将结果重新保存在左面的表达式中。所以，上面的表达式与下面的表达式等价：

```
array[i] = array[i] * 2;
```

下来，让我们重新回到这段程序想要说明的问题上来。读者可能已经意识到了，经过调用函数multipleBy2之后，数组floatVals的值已经改变了。这和我们前面介绍过的函数不会改变实际参数的值岂不是矛盾了吗？实际上这并不矛盾。

这段程序诠释了数组作为函数参数的一个重要法则，这个法则读者应该牢记在心。如果我们把数组作为参数传递给某个函数，并在函数中改变数组元素的值，那么这些改变在函数调用结束之后依然保留。

为什么在C语言中将普通变量作为参数时，函数不能改变其值，而将数组作为参数的时候，函数就能改变其值呢？下面我们来解释一下。如同前面所说的，当我们调用某个函数的时候，调用参数的值是被拷贝到形式参数之中，无论对于数组还是普通变量，这个原则都是正确的。但是对于数组来说，我们并不是把实际的整个数组拷贝到形式参数的数组中，而是把实际的数组所在的内存位置拷贝到形式参数之中。因此，在函数中对形式参数所代表数组的任何元素的操作，实际上是对实际参数所代表数组进行的操作，而不是对于实际数组的副本进行操作。因此，当函数返回的时候，对实际数组的改变就保留下来了。

读者千万记住，我们这里的讨论只适用于将数组作为参数传递给函数。如果只是传递一个数组元素的话，那么这个数组元素的值还是会被拷贝到对应的形式参数中，因此在函数中就无法修改实际的数组元素了。在第11章中，我们还将详细讨论这些问题。

数组排序

为了进一步说明函数可以改变传递给它的数组参数的值这个原理，我们下面编写一个对整型数组进行排序的程序。因为排序是计算科学中极其常见的一类问题，所以科学家们花费了很多精力研究与排序相关的算法。对于给定的一组信息，如何使用最少的时间和计算机内存对其进行排序，已经有了很多成熟的算法。不过，本书的目的在于教授C语言而不是关注计算机算法，因此我们将使用比较简单的排序算法将数组元素按照“升序”重新排列。所谓升序的意思就是将数组中的元素按照其大小递增排列。在升序排序之后，数组中最小的元素排列在数组的第一个位置，最大的元素在最后一个位置，其他的元素也按照从小到大的顺序依次排列。

为了对一个拥有 n 个元素的数组进行升序排列，我们可以把数组中的每一个元素两两进行比较。首先，我们把数组的第一个元素与第二个元素进行比较，如果第一个元素大于第二个元素，我们就交换这两个元素——也就是交换这两个元素中保存的数值。

接下来，我们把数组中的第一个元素（我们已经知道这个元素是数组的头两个元素中较小的那个）和数组的第三个元素进行比较。如果第一个元素大于第三个元素，我们同样交换这两个数组元素。否则的话，我们就保持它们原来的位置不动。经过这个步骤之后，数组前三个元素中最小的一个就保存在数组中的第一个位置。

如果我们重复上述的过程——也就是将数组的第一个元素依次和数组的其他元素进行比较，直到最后一个元素，那么在上述过程结束之后，数组的第一个位置中就保存着数组中的最小元素。

现在，我们可以对数组中的第二个元素重复同样的过程。也就是说，把数组的第二个元素和第三个、第四个直到最后一个进行比较，如果该元素大于与它进行比较的元素，那么就交换它们的值。在这个过程结束之后，数组中的第二个位置上就保存着第二小的元素。

现在读者应该已经清楚如何进行剩下的比较工作了。如果我们已经比较了数组的最后一个和倒数第二个元素的值，并根据需要对它们进行了交换，整个排序的过程也就结束了。这时，整个数组的元素就已经按照升序重新排列了。

假定我们需要对长度为 n 的数组进行排序，下面给出按照上述算法的详细步骤描述：

简单交换法排序

第1步：将 i 设为0。

第2步：将 j 设为 $i+1$ 。

第3步：如果 $a[i] > a[j]$ ，交换它们的值。

第4步：执行 $j = j + 1$ ，如果 $j < n$ ，回到第3步。

第5步：执行 $i = i + 1$ ，如果 $i < n - 1$ ，回到第2步。

第6步： a 已经按照升序排列。

我们在程序8.12的sort函数中实现了上述算法，该函数接收两个参数，第一个是需要排序的数组，第二个是数组的长度。

程序8.12 将数组按照升序排列

//将数组按照升序排列的程序

```
#include <stdio.h>

void sort (int a[], int n)
{
    int i, j, temp;

    for ( i = 0; i < n - 1; ++i )
        for ( j = i + 1; j < n; ++j )
            if ( a[i] > a[j] ) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
}

int main (void)
{
    int i;
    int array[16] = { 34, -5, 6, 0, 12, 100, 56, 22,
        44, -3, -9, 12, 17, 22, 6, 11 };
    void sort (int a[], int n);

    printf ("The array before the sort:\n");

    for ( i = 0; i < 16; ++i )
        printf ("%i ", array[i]);

    sort (array, 16);

    printf ("\n\nThe array after the sort:\n");

    for ( i = 0; i < 16; ++i )
        printf ("%i ", array[i]);

    printf ("\n");

    return 0;
}
```

程序8.12 输出

```
The array before the sort:
34 -5 6 0 12 100 56 22 44 -3 -9 12 17 22 6 11
The array after the sort:
-9 -5 -3 0 6 6 11 12 12 17 22 22 34 44 56 100
```

sort函数使用了一组嵌套循环来实现我们的排序算法。外层循环从数组的第一个元素到倒数第二个元素($a[n-2]$)执行遍历,对于每一个元素,内层循环从外层循环当前元素的下一个元素开始,遍历直到数组的最后一个元素。

如果数组元素的顺序不正确(也就是说 $a[i] > a[j]$),我们就交换这两个元素。在交换过程中,我们使用变量temp临时存储交换的数值。

当两个循环都执行完成之后,数组中的值就已经按照升序排列了,函数的执行随之结束。

在主程序中,我们首先初始化数组array,该数组包含16个整数值。随后,程序显示了该数组的内容,然后调用函数sort,并将该数组以及数组的长度16作为参数传递进去。在函数返回之后,我们重新输出数组的内容。正像读者在输出中看到的那样,数组的内容已经按照升序排列好了。

程序8.12中的sort函数相当简单。为了这种简单,我们付出的代价是程序的执行时间。如果我们必须对很大的数组进行排序(例如包含有上千个元素),那么这里给出的程序可能就需要运行相当长的一段时间。如果我们想要更快的完成排序工作的话,必须使用更为精巧复杂的算法。对于此类问题,《计算机编程艺术——第三卷,排序和搜索》(Donald E. Knuth著,Addison-Wesley出版)是一本经典的书籍。²

多维数组

如同普通变量和一维数组元素一样,我们也可以将多维数组的元素作为参数传递给函数。下面的语句调用了squareRoot函数,并将二维数组元素matrix[i][j]作为参数传递给它。

```
squareRoot(matrix[i][j]);
```

²在C语言的标准库函数中有一个qsort函数,该函数可以对包含任何数据类型的数组进行排序。但是,在使用该函数之前,我们必须理解函数指针的概念,该概念将在第11章中介绍。

我们也可以像一维数组那样，将整个多维数组作为参数传递给函数：只要在调用的时候我们给出数组的名字就可以了。例如，如果我们声明符号`measured_value`为一个整型二维数组，那么下面的语句

```
scalarMultiply(measured_values, constant);
```

将整个数组作为参数传递给函数`scalarMultiply`，该函数将每个数组元素与给定的常数相乘。读者可以看出，这意味着在函数中也可以改变数组元素的值。我们前面讨论过把一维数组作为参数传递给函数的时候，函数可以改变数组元素值的原理，这个原理同样适用于多维数组。如果在函数中修改了多维数组的内容，那么实际的多维数组的元素也将发生改变。

我们已经知道，如果函数的形式参数是一维数组的时候，可以在函数声明和定义中忽略掉数组的长度，仅仅需要给出一对方括号，通知编译器这个形式参数是一个数组即可。对于多维数组来说，情况则不完全是这样。对于二维数组，我们可以省略数组的行数，但是函数的声明中必须说明每一行中包含多少个元素。因此，下面的声明

```
int array_values[100][50]
```

和下面的声明

```
int array_values[][50]
```

都是合法的。它们都可以用来接收一个包含100行50列的二维数组。但是下面的声明：

```
int array_values[100][]
```

以及

```
int array_values[][]
```

都是不合法的。因为我们必须说明二维数组的列的个数。

在程序8.13中，我们定义了一个函数`scalarMultiply`，这个函数将一个二维数组的每一个元素乘以一个固定的常数。作为演示，我们假定数组的大小是3*5。在主程序中，我们调用函数`scalarMultiply`两次，每次调用之后我们还将调用函数`displayMatrix`显示数组的内容。请读者特别注意在函数`scalarMatrix`和`displayMatrix`中用于遍历数组元素的嵌套循环。

程序8.13 使用多维数组和函数

```
#include <stdio.h>
int main (void)
{
```

程序8.13 续

```
void scalarMultiply (int matrix[3][5], int scalar);
void displayMatrix (int matrix[3][5]);
int sampleMatrix[3][5] =
{
    { 7, 16, 55, 13, 12 },
    { 12, 10, 52, 0, 7 },
    { -2, 1, 2, 4, 9 }
};

printf ("Original matrix:\n");
displayMatrix (sampleMatrix);

scalarMultiply (sampleMatrix, 2);

printf ("\nMultiplied by 2:\n");
displayMatrix (sampleMatrix);

scalarMultiply (sampleMatrix, -1);

printf ("\nThen multiplied by -1:\n");
displayMatrix (sampleMatrix);
return 0;
}
```

//根据标号计算3*5的数组的函数

```
void scalarMultiply (int matrix[3][5], int scalar)
{
    int row, column;

    for ( row = 0; row < 3; ++row )
        for ( column = 0; column < 5; ++column )
            matrix[row][column] *= scalar;
}

void displayMatrix (int matrix[3][5])
{
    int row, column;

    for ( row = 0; row < 3; ++row ) {
        for ( column = 0; column < 5; ++column )
            printf ("%5i", matrix[row][column]);
    }
}
```


程序8.13 续

```

        printf ("\n");
    }
}

```

程序8.13 输出

Original matrix:

```

    7 16 55 13 12
    12 10 52 0 7
    -2 1 2 4 9

```

Multiplied by 2:

```

    14 32 110 26 24
    24 20 104 0 14
    -4 2 4 8 18

```

Then multiplied by -1:

```

    -14 -32 -110 -26 -24
    -24 -20 -104 0 -14
    4 -2 -4 -8 -18

```

在主程序中，我们定义了矩阵sampleValues，并使用函数displayMatrix将其初值显示在终端中。请读者注意该函数中的嵌套for循环：外层循环用于遍历二维数组的每一行，因此循环变量row的值从0变化到2，对于row的每一行，内层循环用于遍历其每一列上的元素，因此内层循环变量column的变化范围是从0到4。

在显示某个特定的矩阵元素的时候，我们在printf函数调用中使用格式化字符串%5i，这样可以确保所有输出的矩阵元素都是对齐的。在内层循环执行完之后——也就是说完成输出某一行元素之后，我们在后面附加输出一个换行符号，这样二维数组的下一行输出就会另起一行。

在第一次调用函数scalarMutiply的时候，我们让整个矩阵乘以2。在该函数中，使用一个简单的嵌套循环来遍历数组的每一个元素。接下来，我们使用当前的数组元素matrix[row][column]乘以2，并将结果重新保存在元素matrix[row][column]中——请注意*=操作符的用法。在该函数执行完成之后，程序的流程返回到main函数中，随后重新调用displayMatrix函数，显示数组sampleMatrix的内容。可以看到，数组的每一个元素的值都被乘以了2。

接下来,第二次调用函数scalarMultiply,这次将整个矩阵乘以-1。随后,再次显示二维数组的内容,并结束程序的运行。

函数与可变长度的多维数组

C语言支持变量长度数组,我们可以利用这个特性编写接收任意大小数组参数的函数。按照上述原理,我们可以重新编写程序8.13,使得函数displayMatrix和scalarMatrix可以接收任意大小的二维数组。请看程序8.13A。

程序8.13A 多维变量数组

```
#include <stdio.h>

int main (void)
{
    void scalarMultiply (int nRows, int nCols,
                        int matrix[nRows][nCols], int scalar);
    void displayMatrix (int nRows, int nCols, int matrix[nRows][nCols]);
    int sampleMatrix[3][5] =
    {
        { 7, 16, 55, 13, 12 },
        { 12, 10, 52, 0, 7 },
        { -2, 1, 2, 4, 9 }
    };

    printf ("Original matrix:\n");
    displayMatrix (3, 5, sampleMatrix);

    scalarMultiply (3, 5, sampleMatrix, 2);
    printf ("\nMultiplied by 2:\n");
    displayMatrix (3, 5, sampleMatrix);

    scalarMultiply (3, 5, sampleMatrix, -1);
    printf ("\nThen multiplied by -1:\n");
    displayMatrix (3, 5, sampleMatrix);

    return 0;
}

// 根据标号相乘一个矩阵的函数

void scalarMultiply (int nRows, int nCols,
                    int matrix[nRows][nCols], int scalar)
```


程序8.13A 续

```

{
    int row, column;

    for ( row = 0; row < nRows; ++row )
        for ( column = 0; column < nCols; ++column )
            matrix[row][column] *= scalar;
}

void displayMatrix (int nRows, int nCols, int matrix[nRows][nCols])
{
    int row, column;

    for ( row = 0; row < nRows; ++row ) {
        for ( column = 0; column < nCols; ++column )
            printf ("%5i", matrix[row][column]);

        printf ("\n");
    }
}

```

程序8.13A 输出

Original matrix:

```

7 16 55 13 12
12 10 52 0 7
-2 1 2 4 9

```

Multiplied by 2:

```

14 32 110 26 24
24 20 104 0 14
-4 2 4 8 18

```

Then multiplied by -1:

```

-14 -32 -110 -26 -24
-24 -20 -104 0 -14
4 -2 -4 -8 -18

```

函数scalarMultiply的声明形式如下:

```

void scalarMultiply (int nRows, int nCols,
                    int matrix[nRows][nCols], int scalar)

```

我们将矩阵作为数组的形式参数，其大小也必须作为参数传递给函数，并且这两个形式参数的定义必须在数组形式参数的前面。只有这样，编译器才能知道如何处理函数调用。如果我们使用下面形式的函数定义，编译器将会报告一个错误：因为它并不知道nRows和nCols代表的是什么东西。

```

void scalarMultiply(int matrix[nRows][nCols], int nRows, int nCols, int scalar)

```

读者可以看到，程序8.13A的输出与程序8.13是相同的。这样，我们就有了两个能够处理任意大小二维数组的函数（`scalarMultiply`和`displayMatrix`）。这也就是使用变量长度数组的好处。³

全局变量

现在，我们要介绍一个新的知识点，并把它和本章前面学到的知识结合起来，给出一个范例。程序7.7的功能是将一个正整数转换为其他基数，现在我们使用函数的形式来重写它。为了完成这项任务，首先需要在逻辑上将该程序划分为若干段。只需要看看程序7.7中`main`函数的三个注释，就会发现该程序的功能可以分为三个函数：从用户输入读取需要转换的数字和基数，将该数字改为需要的基数，显示计算结果。

我们可以定义三个函数分别完成这些工作。第一个函数名为`getNumberAndBase`，这个函数提示用户输入需要转换的数字以及基数，并从终端读取这些值。在这里对于程序7.7进行一点小改进——如果用户输入的基数小于2或者大于16，那么程序就输出报错信息，然后将基数设置为10。如此一来，程序实际上将原来的数字又重新显示给用户（也可以提示用户重新输入基数，这一点作为练习留给读者）。

第二个函数是`convertNumber`，这个函数接收用户输入的数值，将其转换为需要的基数，并将转换的结果保存在数组`convertedNumber`中。

第三个函数是`displayConvertedNumber`，这个函数读取数组中的值，然后按照正确顺序将其显示在终端上。对于每一个要显示的数字，在数组`baseDigits`中查找需要显示的字符。

我们这里定义的三个函数通过全局变量来互相通信。前面已经说过，局部变量只能在定义它的函数内被访问，对于全局变量则没有这个限制，也就是说，程序中的任何函数都可以访问全局变量的值。

与局部变量不同，全局变量的声明应该放置在程序的任何函数之外。这种声明形式告诉编译器该变量是全局性质的——它不从属于任何函数。程序中的任何函数都可以按照需要读取和修改该变量的值。

³ 正如前面声明的那样，请读者首先确认，自己使用的编译器是否对变量长度数组提供了完全地支持。

在程序8.14中，我们定义了4个全局变量，每一个变量都至少被两个函数访问。因为数组baseDigits和变量nextDigit只在函数displayConvertedNumber中使用，因此不把它们定义为全局变量，与之相反，我们将其定义为函数displayConvertedNumber中的局部变量。

全局变量的定义出现在程序的开始。因为它们的定义不属于任何函数，因此这些变量是全局的，可以在任何函数中引用这些变量。

程序8.14 将一个正整数转换为其他基数

```
//将一个正整数转为其他基数的函数
#include <stdio.h>

int convertedNumber[64];
long int numberToConvert;
int base;
int digit = 0;

void getNumberAndBase (void)
{
    printf ("Number to be converted? ");
    scanf ("%li", &numberToConvert);

    printf ("Base? ");
    scanf ("%i", &base);

    if ( base < 2 || base > 16 ) {
        printf ("Bad base - must be between 2 and 16\n");
        base = 10;
    }
}

void convertNumber (void)
{
    do {
        convertedNumber[digit] = numberToConvert % base;
        ++digit;
        numberToConvert /= base;
    }
    while ( numberToConvert != 0 );
}

void displayConvertedNumber (void)
{

```

程序8.14 续

```
const char baseDigits[16] =
    { '0', '1', '2', '3', '4', '5', '6', '7',
      '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
int nextDigit;

printf ("Converted number = ");

for (--digit; digit >= 0; --digit ) {
    nextDigit = convertedNumber[digit];
    printf ("%c", baseDigits[nextDigit]);
}

printf ("\n");
}

int main (void)
{
    void getNumberAndBase (void), convertNumber (void),
        displayConvertedNumber (void);

    getNumberAndBase ();
    convertNumber ();
    displayConvertedNumber ();

    return 0;
}
```

程序8.14 输出

```
Number to be converted? 100
Base? 8
Converted number = 144
```

程序8.14 输出（再次运行）

```
Number to be converted? 1983
Base? 0
Bad base - must be between 2 and 16
Converted number = 1983
```

由于我们给函数选取的名字比较合适，现在程序8.14看上去逻辑相当清楚。只要读一下主程序：获取一个数组和基数、转换这个数组、显示结果，我们就能明白程序的意图。这种程序可读性上的改进，是我们将程序划分为三个小的、独立的编程任务所带来的直接

结果。我们现在甚至不需要额外的注释语句，因为函数的名字已经明确地表明了它们完成的工作。

全局变量最常见的用途是多个函数需要存取同一个变量的场合。通过声明全局变量，我们不需要将某个数值在函数之间传来传去，而是直接在函数中引用该数值。使用全局变量有一个缺点：因为函数中使用了全局变量，它的通用性就有所下降。每次当我们重用该函数的时候，我们都必须保证对应的全局变量存在，而且名字也要和函数中使用的名字相一致。

例如，要使用程序8.14中的函数convertNumber，那么要转换的数字必须保存在全局变量numberToConvert中，而要转换的基数必须保存在全局变量base中。另外，我们还必须定义全局变量digit和数组convertedNumber，如果这些变量能够采用形式参数的形式传递给函数的话，函数就会灵活得多。

虽然使用全局变量可以减少传递给函数的参数个数，但是我们付出的代价是函数的通用性，有时候还包括程序的可读性。全局变量影响程序的可读性主要体现在两个方面：1. 无法通过阅读函数的开头部分就得知函数使用的所有变量。2. 仅仅通过阅读函数的调用语句，无法得知函数所需的所有输入和输出。

有些编程者采用了这样一个约定：给所有的全局变量名前面加上一个字母g。例如，程序8.14中声明的全局变量如果采用这种约定，将会是如下的形式：

```
int gConvertedNumber[64];
long int gNumberToConvert;
int gBase;
int gDigit = 0;
```

采用这种命名约定以后，我们就能够很方便地把全局变量和局部变量区分开来。例如，下面的语句：

```
nextMove = gCurrentMove + 1;
```

nextMove是一个局部变量，而gCurrentMove是一个全局变量。通过这些变量名，阅读者就清楚该到哪儿去寻找这些变量的具体定义。

关于全局变量，还有最后一点需要说明：所有的全局变量的初始值都为0。因此，如果我们使用下面的语句声明全局变量gData：

```
int gData[100]
```

那么，在程序开始执行的时候，数组gData的每一个元素都被初始化为0。

再次提醒读者：全局变量的初始化值为0，局部变量的初始值是不确定的，需要在程序中显式地进行初始化。

自动变量和静态变量

如果我们在函数中声明某个局部变量，如下面的squareRoot函数中的guess和epsilon：

```
float squareRoot (float x)
{
    const float epsilon = .00001;
    float guess = 1.0;
    . . .
}
```

我们实际上声明的是自动局部变量。读者可能还能回忆起来，声明自动变量的时候，可以在变量声明前面加上关键字auto。但是实际上通常省略这个关键字，因为它是默认的。在函数每次被调用的时候，所有的自动变量实际上被重新创建出来——也就是分配内存空间。在前面的例子中，每次当我们调用函数squareRoot的时候，编译器都重新为自动变量guess和epsilon分配空间。一旦函数squareRoot结束运行，这些变量就不能够再被访问，它们所占用的内存也被自动回收。实际上，这也是称它们为自动变量的原因。

在声明自动变量的时候，我们可以给它赋予初值。实际上，初值可以是任何表达式。每次当我们调用函数的时候，这个表达式的值将被重新计算并赋给该自动变量。⁴由于函数执行结束之后，我们就再也不能够访问自动变量，因而其中保存的值也随之一并消失。也就是说，编译器保证保存在自动变量中的值，在函数调用结束之后就会消失，而决不会保存到下次调用的时候。

如果我们在变量声明前面放置关键字static，那么情况就完全不同了。在C语言中，static并不代表静电，而是意味着某些东西是静态的。这是静态变量的核心思想——它不会随着函数的调用和退出发生变化。也就是说，在上次调用函数的时候如果我们给静态变量赋了某个值的话，下次调用该函数的时候，这个值将保持不变。

静态变量的初始化也与自动变量不同。静态局部变量的初始化只在程序开始的时候执行一次，而不是在每次调用函数的时候都进行。另外，静态局部变量的初始化表达式必须是一个常量或者常量表达式。静态变量的初始值为0，这一点也与自动变量不同。

⁴ 对于常量形式的自动变量来说，由于这些变量的值保存在只读内存中，因此每次调用函数的时候不一定需要初始化（这些细节由编译器决定）。

下面的语句定义了函数auto_static。

```
void auto_static (void)
{
    static int staticVar = 100;
    .
    .
    .
}
```

在这个函数定义中，变量staticVar只有在程序开始执行的时候，其值才被初始化为100。如果我们需要在每次调用该函数的时候都要将其值设置为100，则需要使用显式的初始化语句，如下所示：

```
void auto_static (void)
{
    static int staticVar;
    staticVar = 100;
    .
    .
    .
}
```

当然了，如果我们这样给静态变量赋值的话，实际上违背了我们使用静态局部变量的本意。

程序8.15进一步阐明了自动局部变量和静态局部变量的区别。

程序8.15 演示自动变量与局部变量的使用

```
//演示自动变量与局部变量的使用的程序
#include <stdio.h>

void auto_static (void)
{
    int autoVar = 1;
    static int staticVar = 1;

    printf ("automatic = %i, static = %i\n", autoVar, staticVar);

    ++autoVar;
    ++staticVar;
}

int main (void)
{
```

程序8.15 续

```
int i;  
void auto_static (void);  
  
for ( i = 0; i < 5; ++i )  
    auto_static ();  
  
return 0;  
}
```

程序8.15 输出

```
automatic = 1, static = 1  
automatic = 1, static = 2  
automatic = 1, static = 3  
automatic = 1, static = 4  
automatic = 1, static = 5
```

在函数auto_static中，我们声明了两个局部变量：第一个变量autoVar是一个自动变量，类型为int，初始值为1；第二个变量staticVar是静态变量，类型还是int，初始值也是1。在函数中我们首先显示这两个变量的值，然后将其值分别加1，随后结束函数的运行。

在main函数中，我们使用一个for循环调用auto_static函数5次。程序8.15的输出很清楚地显示了这两类变量的区别。在每一行输出中，变量automatic的值都是1，因为在函数的开始处，每次我们都将其初值设置为1。与之相比，局部静态变量的值却从1逐步增加到5。这是因为该变量的值，仅仅在程序开始执行的时候被初始化一次，而每次函数调用完成之后，该变量的值都保留下来。

声明局部变量是静态的还是自动的，要根据程序的需要决定。如果我们需要某个变量的值在多次调用之间保留下来（比如该变量用于保存函数被调用的次数），那么我们可以使用静态变量。还有，如果函数的值只需初始化一次，以后不会改变，那么我们也可以将该变量声明为静态变量，这样可以避免每次都初始化该变量，从而提高程序的效率。特别是当我们需要初始化数组的时候，这种效率方面的考虑就更重要一些。

从另一方面来说，如果一个局部变量的值每次都需要初始化，那么使用自动变量看上去就更符合逻辑一些。

递归函数

C语言支持对于函数的递归调用。递归函数可以非常简洁和有效地解决某些计算问题。这类问题通常的求解步骤能够归结为：将同样的步骤应用于越来越小的问题子集，最终得出答案。这类问题的一个典型例子是对于含有嵌套括号的表达式求值。对于树和列表进行搜索或者排序，也是常见的递归问题。

最常用于展现递归函数能力的例子是计算正整数的阶乘。读者可以回忆一下： n 的阶乘就是从1到 n 这 n 个数的乘积，用符号表示就是 $n!$ 。0的阶乘是一种特殊情况，其值规定为1。按照上述规则，5的阶乘可以用下式计算：

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

而

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$$

读者观察一下 $6!$ 和 $5!$ 就会发现，前者的值是后者的6倍，也就是说： $6! = 6 \times 5!$ 。一般来说，任何正整数 n 的阶乘等于 n 乘以 $n-1$ 的阶乘：

$$n! = n \times (n-1)!$$

前面的公式是阶乘的一种递归定义形式，因为一个数的阶乘用另外一个数的阶乘来定义。实际上，我们可以按照上述的递归定义编写一个函数用于计算正整数 n 的阶乘。这个程序如程序8.16所示。

程序8.16 递归计算阶乘

```
#include <stdio.h>

int main (void)
{
    unsigned int j;
    unsigned long int factorial (unsigned int n);

    for ( j = 0; j < 11; ++j )
        printf ("%2u! = %lu\n", j, factorial (j));

    return 0;
}

//计算正整数阶乘的递归函数
```

程序8.16 续

```
unsigned long int factorial (unsigned int n)
{
    unsigned long int result;

    if ( n == 0 )
        result = 1;
    else
        result = n * factorial (n - 1);

    return result;
}
```

程序8.16 输出

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

因为我们在函数factorial内部又调用了这个函数，因此factorial函数是一个递归函数。当我们使用该函数计算3的阶乘时，函数的形式参数n首先被设置为3。因为这个值不等于0，因此程序接下来执行下面的语句：

```
result = n * factorial (n - 1);
```

这个语句又重新调用了函数factorial，这次传递给该函数的参数是2。因此，我们首先暂停对于3的阶乘的计算，转而计算2的阶乘。

虽然我们在这里又调用了函数factorial，但是在概念上，我们应该把它当作另外一个函数。在C语言中，每次当我们调用一个函数的时候——无论是递归调用或者非递归调用，该函数都会得到自己独有的一组局部变量和形式参数。因此，当我们使用参数3调用factorial函数时的形式参数n以及局部变量result，与我们使用参数2调用factorial函数时的形式参数n以及局部变量result是完全不同的。

当我们使用参数2调用factorial函数的时候，计算机将执行factorial函数的下面这条语句：

```
result = n * factorial ( n -1 );
```

这条语句实际上等价于：

```
result = 2 * factorial(1);
```

我们再次重复了上面的函数调用过程，首先把对于2的阶乘计算放在一边，转而去计算1的阶乘。

当n = 1的时候，计算机还是执行factorial函数的下面这条语句：

```
result = n * factorial ( n -1 );
```

这个语句等价于：

```
result = 1 * factorial(0);
```

当我们以参数0调用factorial函数的时候，函数将变量result的值设置为1，然后将该值返回给调用者。这个返回语句又重新启动了所有前面被暂停的阶乘计算。当factorial(0)的计算结果(也就是1)，被返回给其调用者的时候(这里恰好也是factorial函数)，该结果与1相乘，然后将计算结果赋给变量result。这个计算结果——也就是1，被当作factorial(1)的计算结果，返回给其调用者(这里恰好还是factorial函数)。在调用者函数中，这个返回值(1)和2相乘，其结果保存在变量result中，并被作为函数调用factorial(2)的结果返回给其调用者。最后，这个返回值(2)和3相乘，完成了对于3的阶乘的计算。这个计算结果——6，作为factorial函数的最终结果，返回给main函数。main函数随后将这个结果显示在终端上。

总结一下，表达式factorial(3)的计算过程可以用下面的算式来表示：

```
factorial (3) = 3 * factorial (2)
              = 3 * 2 * factorial (1)
              = 3 * 2 * 1 * factorial (0)
              = 3 * 2 * 1 * 1
              = 6
```

如果读者对于理解上面的过程有些困难，那么可以试着用笔和纸列出上面具体的过程。假定我们用参数4调用函数factorial，请列出每一次调用函数factorial时的，n和result的值。

到此为止，我们已经完成了对于函数的讨论。函数是C语言中一种非常强有力的工具。将整个程序分解为独立的小程序的好处很多，这里就不一一尽述了。在本书以后的部分，我们将大量地使用函数。读者应该再次浏览本章，反复研究那些尚未完全理解的知识。试着完成下面的练习，它们也可以帮助读者巩固本章学到的知识。

练习

1. 输入并运行本章给出的16个程序。将程序的输出结果与书中给出的结果进行对比。
2. 修改程序8.4中的函数`calculateTriangularNumber`，使其返回计算结果而不是打印它。然后再回到程序5.5，使其调用新版本的函数。
3. 修改程序8.8，将`epsilon`的值也作为参数传递给函数。试验不同的`epsilon`值，观察`epsilon`值的变化对计算出的平方根的影响。
4. 修改程序8.8，使得每次循环的时候都打印出变量`guess`的值。观察`guess`的收敛速度、迭代运行的次数、需要计算平方根的数字以及最初的猜测值之间有什么关系？
5. 在程序8.8中，如果我们计算非常大的或者非常小的数的平方根，那么原来结束循环的条件就有些不合适了。这里给出一种新的标准：不是比较`x`和`guess2`之间的差，而是比较这两个数的比值与1的差，比值越接近1，所计算出的平方根值就越准确。

使用这个新标准修改程序8.8。

6. 修改程序8.8，将函数`squareRoot`的形式参数改为`double`类型，同样将其返回值也修改为`double`类型。读者还要记得修改变量`epsilon`的类型，这样才能够反映出采用双精度数以后所计算出平方根的精度变化。
7. 编写一个程序，计算整数的正整数次方。假定这个函数的名字为`x_to_the_n`，该函数接收两个参数`x`和`n`，函数的返回值类型为`long int`，其值为`xn`。
8. 下面的算式代表一个二次方程。

$$ax^2 + bx + c = 0$$

上式中的 a 、 b 、 c 都是常数。下面的算式就是一个二次方程

$$4x^2 - 17x - 15 = 0$$

其中 $a = 4, b = -17, c = -15$ 。能够使上面的算式成立的 x 的值，也称作方程的根，可以通过将 a, b, c 的值代入下面的公式中得到：

如果表达式 $b^2 - 4ac$ 的值大于等于0，那么方程的两个根 x_1 和 x_2 可以用下面的公式计算：

$$x_1 = (-b + \sqrt{b^2 - 4ac}) / (2a)$$

$$x_2 = (-b - \sqrt{b^2 - 4ac}) / (2a)$$

如果表达式 $b^2 - 4ac$ ，也就是判别式的值小于0，那么方程的两个根 x_1 和 x_2 都是复数。

编写一个程序，用于求解二次方程。方程的系数 a, b 和 c 应该由用户输入。如果判别式小于0，程序应该输出提示信息，告诉用户方程的根是复数，否则，程序应当计算出 x_1 和 x_2 的值。（读者还应该利用我们本章编写的squareRoot函数）。

9. 整数 u, v 的最小公倍数就是可以同时整除 u 和 v 的最小正整数。例如，15和10的最小公倍数是30，因为30是能够同时整除10和15的最小正整数。编写一个函数 lcm ，用于计算两个整数的最小公倍数，该函数接收两个整型参数，返回它们的最小公倍数。该函数可以使用下面的公式计算两个整数的最小公倍数：

$$lcm(u, v) = uv / gcd(u, v) \quad u, v \geq 0$$

上面的算式中， $gcd(u, v)$ 代表 u, v 的最大公约数。这里请读者使用我们在程序8.6中编写的最大公约数函数。

10. 编写一个名为prime函数，该函数接收一个整型参数。如果该参数为质数，该函数返回1，否则的话返回0。
11. 编写一个名为arraySum的函数，该函数接收一个数组及其长度作为参数，返回值是参数数组中所有元素的和。
12. 一个有 i 行， j 列的矩阵 M 可以被转置为 j 行 i 列的矩阵 N 。转置的公式是 $M_{ij} = N_{ji}$

a 编写一个名为transposeMatrix的函数，该函数接收两个参数，一个是4x5的矩阵，另一个是5x4的矩阵。该函数将4x5的矩阵转置，然后将结果存放在5x4的矩阵之中。编写一个main函数来测试自己编写的函数。

b 使用变量长度数组作为参数，重写上面的程序。将需要转置的矩阵、转置的结果矩阵以及原始矩阵的行列数作为参数传递给transposeMatrix函数。

13. 修改程序8.12中的sort函数，给函数增加一个参数，用来表示函数应该按照升序还是降序对数组进行排序。随后，根据该参数修改sort函数中使用的算法。
14. 使用全局变量重新编写上面四个练习中编写的函数。例如，前面的例子现在应该对一个全局数组进行排序。

15. 修改程序8.14, 当用户输入一个无效的基数时, 提示用户输入错误并要求用户重新输入, 直到用户输入正确的基数为止。
16. 修改程序8.14, 使得用户可以转换任意多个整数。当用户输入0时, 程序停止运行。

Working with Structures

使用结构

在第7章“使用数组”中，我们向读者介绍了数组的使用。数组允许将一组相同类型的数值保存在单个的逻辑实体中。为了访问数组中的某个元素，我们可以简单地使用数组名和一个下标。

C语言还提供了另外一种将若干个数据组合为单个逻辑实体的方法，这就是结构。结构是本章的主要讨论话题。读者随后将会看到，在很多C语言程序中，结构都是一个非常有用的工具。

假定我们要在程序中存储一个日期，比如说2004年9月25日，用于程序输出，或者某些日期计算。我们可以采用如下的简单方法存储该日期：用一个名为year的变量保存年，用一个名为month的变量保存月，用一个名为day的变量保存日。如下面的语句所示：

```
int year = 2004, month = 9, day = 25;
```

虽然这种方法可以达到我们的要求，但是如果需要存储另外的日期——比如说购买某个物品的时间，就需要再定义三个变量：purchaseYear、purchaseMonth和purchaseDay。每次当我们需要在程序中使用日期的时候，都访问这三个变量。

读者可以看到，使用这种方法表示日期，需要在程序中同时关注三个不同的变量，虽然这三个变量所保存的数值在逻辑上是关联的。如果我们能够把这三个变量编为一组，那么就要好得多。C语言的结构正是用于这个目的。

用于存储日期的结构

我们可以定义一个结构date，该结构包含三个成员，分别用于保存年、月、日。定义该结构的语句如下：

```
struct date
{
    int year;
    int month;
    int day;
};
```

上面定义的结构包含三个成员变量：year、month和day。从某种程度上来说，上面的语句实际上定义了一种新的数据类型，随后我们可以声明类型为struct date的变量，如下面的语句所示：

```
struct date today;
```

我们也可以将两个声明放在同一行上，如下所示：

```
struct date today, purchaseDate;
```

与普通的变量类型——float、int、char等不同，存取结构需要特殊的语法。如果我们要使用结构变量的某个成员，应该首先写出该结构变量的名字，后面跟上一个点，然后再写出成员变量的名字。例如，如果要把结构变量today的成员变量day设置为25，我们可以使用下面的语句：

```
today.day = 25;
```

这里提醒读者注意，在结构变量名和点号之间以及点号与成员变量名之间是不允许有空格的。同样类似，如果我们要设置结构变量today的成员year为2004，可以使用下面的语句：

```
today.year = 2004;
```

还有，比如我们要检查月份的值是否等于12，以便决定下一个月的编号，则可以使用如下的语句：

```
if( today.month == 12 )
    nextMonth = 1;
```

请读者试着推测一下，下面的语句完成怎样的任务：

```
if ( today.month == 1 && today.day == 1 )
    printf ("Happy New Year!!!\n");
```

程序9.1综合展示了我们前面讨论的关于结构的一些知识点。

程序9.1 结构的使用

```
// 显示结构使用的程序

#include <stdio.h>

int main (void)
{
    struct date
    {
        int month;
        int day;
        int year;
    };

    struct date today;

    today.month = 9;
    today.day = 25;
    today.year = 2004;

    printf ("Today's date is %i/%i/%.2i.\n", today.month, today.day,
        today.year % 100);

    return 0;
}
```

程序9.1 输出

Today's date is 9/25/04.

main函数的第一条语句定义了一个名为date的结构，该结构包含三个成员变量：year、month和day。在第二条语句中，我们定义了一个类型为struct date的结构变量today。第一条语句的作用在于通知C语言编译器date结构类型的组成，该语句并不会导致计算机内存的分配。第二条语句定义了一个类型为struct date的变量today，因此编译器将会在计算机中给该变量分配内存空间，用于存储date结构的三个整型数。读者应该清楚地认识定义结构和声明一个结构变量之间的区别。

在我们声明了变量today之后，程序接着给该结构变量的三个成员分别赋值，赋值的结果如图9.1所示。

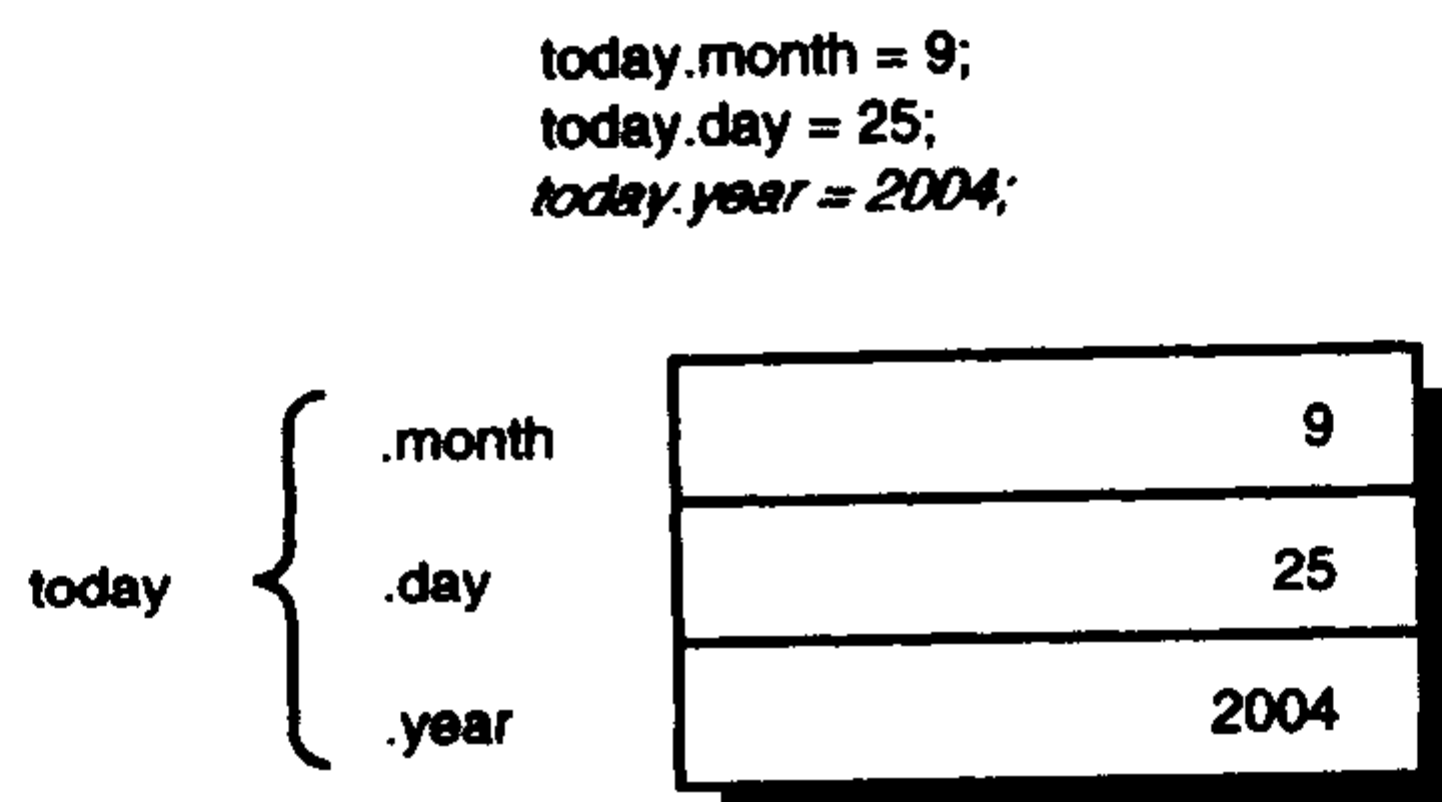


图 9.1 给结构的成员变量赋值

当我们给结构变量`today`的成员赋值之后，使用`printf`语句将该结构的值打印出来。在调用`printf`语句之前，首先计算年份与100相除的余数，这样在显示年份的时候，只显示04。读者可能还记得，格式化输出字符串`%02i`的作用是将某个整数按两位数显示，不足的在左面补上0，这个格式化输出字符串保证了年份的正确显示。

在表达式中使用结构

结构的成员变量在表达式中的计算规则与普通的变量是完全相同的，因此，当一个整型的结构成员变量与一个整型数相除的时候，C语言同样使用整数除法规则。例如下面的语句就按照整数除法规则进行。

```
century = today.year / 100 + 1;
```

假定我们要编写一个程序，该程序接收用户输入的日期，计算出下一天的日期，并将其显示给用户。初看上去，这个任务很容易完成，我们可以读入用户的输入，然后使用下面的语句计算第二天：

```

tomorrow.month = today.month;
tomorrow.day = today.day + 1;
tomorrow.year = today.year;
    
```

对于绝大多数日期来说，上面的例子工作得很好。但是在下面两个特殊情况发生时，这个算法就会失败：

1. 如果用户输入的日期是某月的最后一天；
2. 如果用户输入的日期是某年的最后一天。

为了检查某一天是否是某个月的最后一天，我们可以使用一个整型数组存储每个月的天数，然后以月份为下标查找该数组，得出每月的天数，如下面的语句所示：

```
int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```


该语句定义了一个名为daysPerMonth的整型数组，该数组包含了12个整型数。对于编号为i的月份，数组元素daysPerMonth[i-1]为对应月份的天数。例如4月份（也就是每年的第四个月）包含的天数等于数组元素daysPerMonth[3]的值，即30。（我们也可以定义包含13个元素的数组，然后让编号为i的月份的天数等于下标为i的数组元素的值，这样我们就可以直接使用月份的编号作为下标存取数组的值，而不再需要减去1了。在程序中使用12个元素的数组还是使用13个元素的数组纯粹取决于编程者的个人喜好。）

如果我们发现用户输入的日期落在某个月的最后一天，我们可以给结构成员变量month的值增加1，然后将day的值设为1即可。

为了处理上面所说的第二种特殊情况，我们还必须检查用户输入的日期是否是某个月的最后一天，并且月份等于12。如果出现这种情况，可以把日期和月份的值都设置为1，然后给年份增加1即可。

程序9.2请求用户输入一个日期，然后计算出第二天的日期，并将其显示出来。

程序9.2 计算第二天

// 计算第二天日期的程序

```

#include <stdio.h>

int main (void)
{
    struct date
    {
        int month;
        int day;
        int year;
    };

    struct date today, tomorrow;

    const int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30,
                                    31, 31, 30, 31, 30, 31 };

    printf ("Enter today's date (mm dd yyyy): ");
    scanf ("%i%i%i", &today.month, &today.day, &today.year);

    if ( today.day != daysPerMonth[today.month - 1] ) {
        tomorrow.day = today.day + 1;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    }
}
    
```

程序9.2 续

```
else if ( today.month == 12 ) { // 年尾
    tomorrow.day = 1;
    tomorrow.month = 1;
    tomorrow.year = today.year + 1;
}
else { // 月末
    tomorrow.day = 1;
    tomorrow.month = today.month + 1;
    tomorrow.year = today.year;
}

printf ("Tomorrow's date is %i/%i/%.2i.\n", tomorrow.month,
        tomorrow.day, tomorrow.year % 100);

return 0;
}
```

程序9.2 输出

```
Enter today's date (mm dd yyyy): 12 17 2004
Tomorrow's date is 12/18/04.
```

程序9.2 输出（再次运行）

```
Enter today's date (mm dd yyyy): 12 31 2005
Tomorrow's date is 1/1/06.
```

程序9.2 输出（第三次运行）

```
Enter today's date (mm dd yyyy): 2 28 2004
Tomorrow's date is 3/1/04.
```

如果读者仔细查看程序的输出，很快就会发现一个错误：2004年2月28日的下一天应该是2004年2月29日，而不是程序计算的结果：2004年3月1日。我们的程序忘记处理闰年的情况了。在下一节中，我们将修正这个错误。现在我们先关注于分析上面程序的逻辑。

程序中首先定义了结构date，然后再定义了两个结构变量today和tomorrow。随后程序要求用户输入日期，并将读入的三个整数分别保存在today变量的三个成员变量today.year、today.month和today.day中。接下来，我们比较today.day和daysPerMonth[today.month-1]的值，以判断用户输入的日期是否是某个月的最后一天，如果不是的话，我们就设置tomorrow的day成员等于today的day成员加1，而year和month成员则与其相等。

如果用户输入日期等于某个月的最后一天，我们再判断该日期是否是一年的最后一天。如果月份等于12，那么我们就知道该日期确实是某年的最后一天，我们将tomorrow设置为第二年的第一天，如果月份不等于12，那么我们就将tomorrow设置为下一个月的第一天。

当程序计算出第二天之后，我们使用printf语句将结果显示给用户，随后程序退出运行。

函数与结构

现在让我们回过头来解决程序9.2中的遗留问题。我们在程序中假定2月总是只有28天，因此如果用该程序计算2月28日的下一天时，程序总是回答3月1日。为了改正这个错误，我们需要在程序中对于闰年进行特别的测试。如果某个年份是闰年，那么这一年的2月一共有29天，反之，我们则通过查找数组daysPerMonth来获取某个月份的天数。

为了在程序中处理这些特殊情况，我们最好编写一个名为numberOfDays的函数。这个函数内部对于闰年进行检查，并查找daysPerMonth数组。使用该函数，我们只需要修改main函数的if语句那一行就可以了，也就是将today.day与daysPerMonth[today.month-1]比较改为today.day与函数numberOfDays的返回值进行比较就可以了。

请读者仔细研究一下程序9.3，特别注意传递给函数numberOfDays的参数。

程序9.3 对于计算下一天程序的修正

```
// 计算下一天日期的程序
#include <stdio.h>
#include <stdbool.h>

struct date
{
    int month;
    int day;
    int year;
};

int main (void)
{
```

程序9.3 续

```
struct date today, tomorrow;
int numberOfDays (struct date d);

printf ("Enter today's date (mm dd yyyy): ");
scanf ("%i%i%i", &today.month, &today.day, &today.year);
if ( today.day != numberOfDays (today) ) {
    tomorrow.day = today.day + 1;
    tomorrow.month = today.month;
    tomorrow.year = today.year;
}
else if ( today.month == 12 ) { // 年尾
    tomorrow.day = 1;
    tomorrow.month = 1;
    tomorrow.year = today.year + 1;
}
else { // 月末
    tomorrow.day = 1;
    tomorrow.month = today.month + 1;
    tomorrow.year = today.year;
}
printf ("Tomorrow's date is %i/%i/%.2i.\n", tomorrow.month,
    tomorrow.day, tomorrow.year % 100);
return 0;
}

// 查找一个月中日期数的函数
int numberOfDays (struct date d)
{
    int days;
    bool isLeapYear (struct date d);
    const int daysPerMonth[12] =
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    if ( isLeapYear (d) == true && d.month == 2 )
        days = 29;
    else
        days = daysPerMonth[d.month - 1];
    return days;
}
```

程序9.3 续

```
// 判断是否为闰年的函数
bool isLeapYear (struct date d)
{
    bool leapYearFlag;

    if ( (d.year % 4 == 0 && d.year % 100 != 0) ||
        d.year % 400 == 0 )
        leapYearFlag = true; // 闰年
    else
        leapYearFlag = false; // 非闰年

    return leapYearFlag;
}
```

程序9.3 输出

```
Enter today's date (mm dd yyyy): 2 28 2004
Tomorrow's date is 2/29/04.
```

程序9.3 输出（再次运行）

```
Enter today's date (mm dd yyyy): 2 28 2005
Tomorrow's date is 3/1/05.
```

在上面的程序中，读者一眼就会发现：date结构的定义出现在所有函数的外面，这种方式的声明使得在整个文件内我们都可以使用date结构。在这一点上，结构和变量非常相似。如果一个结构在一个函数中被定义，那么只有在这个函数内部可以使用该结构，也就是说，该结构定义的作用域是局部的，只限于定义它的函数。如果结构的定义被放在所有函数的外面，那么该结构定义的作用域就是全局的，在整个源文件中都可以使用该结构的定义。

在main函数中，下面的结构声明告诉编译器numberOfDays函数接收一个struct date类型的参数，其返回值为整型。

```
int numberOfDays (struct date d);
```

在main函数中，我们不是像程序9.2那样，将today.day的值与daysPerMonth[today.month-1]进行比较，而是使用如下的语句：

```
if( today.day != numberOfDays(today))
```

在该语句中，我们将结构变量`today`作为参数传递给函数`numberOfDays`。同时，在`main`函数中，我们必须采用适当的形式声明该函数，以便通知编译器该函数的参数类型。该函数的声明语句如下所示：

```
int numberOfDay( struct date d);
```

虽然结构也是一种将一组值组合起来的手段，但是结构变量作为函数参数的时候，它的行为却与普通变量相类似，而与数组不同。具体来说，在被调用的函数中处理的只是实际参数的拷贝，对于形式参数的任何修改，都不会影响到原来的实际参数。

在函数`numberOfDays`的开始，我们首先判断年份是否是闰年以及月份是否是二月。第一个判断通过调用另外一个函数`isLeapYear`来完成，稍后我们将介绍这个函数。从下面的语句读者可以看出，函数`isLeapYear`的返回值类型是`bool`，当指定的年份是闰年，返回值为`true`，否则为`false`。在第6章我们已经讨论过布尔类型的变量，并且知道，如果要使用该类型的变量，应该包含头文件`<stdbool.h>`，这也是我们在程序9.3的开始包含该头文件的原因。

```
if ( isLeapYear( d ) == true && d.month == 2 )
```

关于这个`if`语句中判断是否是闰年的函数的命名，我们还有一点要特别说明一下：这个函数的名字取得很恰当，使得理解这条`if`语句变得非常容易：我们很容易看出来该函数的返回值是某种布尔值。

让我们接着讨论下来的程序。如果函数`numberOfDays`发现给定参数代表闰年的二月份，它就将变量`days`的值设置为29，否则的话，就以月份为下标查找数组`dayPerMonth`，并将其值赋给变量`days`，随后，程序将`days`变量的值返回给调用者。`main`函数随后的执行流程与程序9.2相同。

函数`isLeapYear`也很容易理解。该函数对于参数进行判断，如果给定的年份是闰年，它就返回`true`，否则的话，返回`false`。

作为一个将程序结构调整得更合理的练习，我们将计算某个日期的下一天的任务全部放在一个单独的函数中完成。该函数名为`updateDate`，并接收当天的日期作为参数。函数`updateDate`计算出第二天的值，并将其作为一个完整的结构返回给调用者。

程序9.4 对于计算下一天程序的修正（第二版）

```
// 计算下一天日期的程序

#include <stdio.h>
#include <stdbool.h>

struct date
{
    int month;
    int day;
```

程序9.4 续

```
    int year;
};
// 计算下一天日期的函数

struct date dateUpdate (struct date today)
{
    struct date tomorrow;
    int numberOfDays (struct date d);

    if ( today.day != numberOfDays (today) ) {
        tomorrow.day = today.day + 1;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    }
    else if ( today.month == 12 ) { // 年尾
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year + 1;
    }
    else { // 月末
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }
    return tomorrow;
}

// 查找一月中日期数的函数
int numberOfDays (struct date d)
{
    int days;
    bool isLeapYear (struct date d);
    const int daysPerMonth[12] =
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    if ( isLeapYear && d.month == 2 )
        days = 29;
    else
        days = daysPerMonth[d.month - 1];

    return days;
}
```

程序9.4 续

```
// 判断是否为闰年的函数
bool isLeapYear (struct date d)
{
    bool leapYearFlag;

    if ( (d.year % 4 == 0 && d.year % 100 != 0) ||
        d.year % 400 == 0 )
        leapYearFlag = true; // It's a leap year
    else
        leapYearFlag = false; // Not a leap year
    return leapYearFlag;
}

int main (void)
{
    struct date dateUpdate (struct date today);
    struct date thisDay, nextDay;

    printf ("Enter today's date (mm dd yyyy): ");
    scanf ("%i%i%i", &thisDay.month, &thisDay.day,
        &thisDay.year);

    nextDay = dateUpdate (thisDay);

    printf ("Tomorrow's date is %i/%i/%.2i.\n", nextDay.month,
        nextDay.day, nextDay.year % 100);

    return 0;
}
```

程序9.4 输出

```
Enter today's date (mm dd yyyy): 2 28 2008
Tomorrow's date is 2/29/08.
```

程序9.4 输出（再次运行）

```
Enter today's date (mm dd yyyy): 2 22 2005
Tomorrow's date is 2/23/05.
```

```
next_date = dateUpdate(thisDay);
```


main函数中下面一条语句演示了既可以将结构作为参数传递给函数，也可以将其作为函数的返回值。我们通过给出函数updateDate适当的原型声明来告诉编译器这一点。updateDate函数中的代码与程序9.3的main函数相同，numberOfDays函数和isLeapYear函数没有什么变化。

由于程序9.4的函数调用关系稍微复杂一些，这里再说明一下：main函数调用updateDate函数，updateDate函数随后再调用numberOfDays函数，而numberOfDays函数接着调用了isLeapYear函数。

用于存储时间的结构

假定我们在程序中需要存储时间，存储的内容包括小时、分钟和秒。我们前面已经看到了，使用结构来处理日期是多么方便，因此这里我们很自然地也会想到使用结构来处理时间。表示时间的结构定义如下：

```
struct time
{
    int hour;
    int minutes;
    int seconds;
};
```

在绝大多数计算机系统中，都采用24小时制来表示时间。这种表达法可以不用给时间再加上a.m.或者p.m.的修饰符。在24小时制中，午夜12时用0代表，每小时递增1，直到到达23，也就是午夜11:00。因此，24小时制的4:30实际上是4:30 a.m.，而16:30表示的是4:30 p.m. 12:00代表中午，而00:01表示午夜过后的一分钟。

所有的计算机系统都有一个内置的时钟。该时钟的用途非常广泛：它可以告诉用户当前的时间；它可以在指定的时间激活一个事件或者运行某个程序；或者记录某个事件发生的具体时间。在计算机系统中通常有若干个程序与时钟相关，其中之一的作用就是更新系统的时间，该程序通常每秒钟执行一次。

假定我们要编写一个与前面的时钟类似功能的程序，该程序以秒为单位更新时间。读者也许很快就会发现，这个程序和我们前面计算下一天的程序非常的类似。正像计算下一天的时候需要考虑一些特殊情况一样，我们的时钟程序也要考虑某些特殊情况，这些情况分别如下：

1. 如果秒数达到了60，我们就将秒数重置为0，并将分钟数加一；
2. 如果分钟数达到了60，我们就将其重置为0，然后将小时数加一；
3. 如果小时数达到了24，我们就将小时数、分钟数和秒数全部重置为0。

程序9.5编写了函数timeUpdate，该函数接收当前的时间作为参数，返回一秒后的时间。

程序9.5 以秒为单位更新时间

```
// 以秒为单位更新时间的程序

#include <stdio.h>

struct time
{
    int hour;
    int minutes;
    int seconds;
};

int main (void)
{
    struct time timeUpdate (struct time now);
    struct time currentTime, nextTime;

    printf ("Enter the time (hh:mm:ss): ");
    scanf ("%i:%i:%i", &currentTime.hour,
            &currentTime.minutes, &currentTime.seconds);

    nextTime = timeUpdate (currentTime);
    printf ("Updated time is %i.%i.%i\n", nextTime.hour,
            nextTime.minutes, nextTime.seconds );

    return 0;
}

// 以秒为单位更新时间的函数
struct time timeUpdate (struct time now)
{
    ++now.seconds;
```

程序9.5 续

```
if ( now.seconds == 60 ) { // 下一分钟
    now.seconds = 0;
    ++now.minutes;

    if ( now.minutes == 60 ) { // 下一小时
        now.minutes = 0;
        ++now.hour;

        if ( now.hour == 24 ) // 午夜
            now.hour = 0;
    }
}
return now;
}
```

程序9.5 输出

```
Enter the time (hh:mm:ss): 12:23:55
Updated time is 12:23:56
```

程序9.5 输出（再次运行）

```
Enter the time (hh:mm:ss): 16:12:59
Updated time is 16:13:00
```

程序9.5 输出（第三次运行）

```
Enter the time (hh:mm:ss): 23:59:59
Updated time is 00:00:00
```

在main函数中，我们要求用户输入时间。程序中使用了如下的格式化输入字符串：

```
"%i:%i:%i"
```

当使用scanf函数的时候，如果我们在格式化输入字符串中指定了非格式化字符，这意味着读者的输入中也应该包含着这些字符。因此，上面的格式化输入字符串要求用户输入三个整数，第一个和第二个用冒号隔开，第二个和第三个也用冒号隔开。在第16章“C语言的输入输出”时，我们将会学到如何利用scanf函数的返回值来判断用户输入的格式是否正确。

当程序读入了用户输入的时间之后，我们接下来调用timeUpdate函数，并将保存输入时间的结构变量currentTime传递给它。该函数的返回值被保存在结构变量nextTime中，随后我们再用printf语句将其显示给用户。

timeUpdate函数首先将秒数增加1，然后判断秒数是否到达了60，如果是的话，将秒数设置为0，然后将分钟数加1；随后再判断分钟数是否到达了60，如果是的话，将分钟数设置为0，然后将小时数加1；最后，在前两个条件都满足的情况下，我们判断小时数是否等于24，如果是的话，那么当前的时间正是午夜，我们将小时数、分钟数和秒数全部置0，然后将结构变量now返回给调用者，该结构其中包含着计算出来的下一秒的时间。

结构的初始化

结构变量的初始化和数组的初始化很类似，我们可以用逗号将结构成员的值分割开来，并用一对大括号将其包围起来。例如，如果我们要将struct date类型的结构变量today的值设置为2005年7月2日，可以使用如下的语句：

```
struct date today = {7, 2, 2005};
```

而下面的语句则定义了时间结构变量this_time，并将其设置为3:29:55。

```
struct time this_time = {3, 29, 55};
```

结构变量在初始化语句的执行方式上与其他普通变量也是类似的，如果this_time是一个局部变量，那么每次进入函数体的时候，该变量的初始化语句都要重新执行。如果this_time是一个静态变量（通过在声明语句前面加上static关键字可以做到这一点），那么该初始化语句只在程序开始执行的时候执行一次。另外，C语言编译器要求结构初始化表达式必须是常量表达式。

和数组的初始化类似，在初始化表达式中给出的常量表达式个数也可以少于结构变量的成员变量个数。比如，下面的表达式

```
struct time time1 = {12, 10};
```

将结构变量time1的hour成员初值设置为12，minute成员初值设置为10，而没有给出second成员的初值，在这种情况下，该成员变量的初值是未定义的（也就是说，由编译器的实现自行决定，可能为0，也可能为任何值）。

我们也可以在初始化列表中给出需要初始化的成员变量的名字，这种时候，我们应该采用

```
.member = value
```

的形式。通过给出成员变量的名字，我们可以以任意的顺序初始化结构变量的成员，或者只初始化部分成员，如下面的语句所示：

```
struct time time1 = {.hour = 12, .minute = 10};
```


这个语句的初始化作用与前面的语句相同。而下面的语句，
`struct date today = {.year = 2004};`

则只将结构变量today的成员变量year的值设置为2004。

复合字面量

我们可以使用复合字面量的方式，在一个语句中给结构变量的多个成员变量赋值。比如假定我们已经声明了struct date类型的变量today，在程序9.1中我们使用了三条语句给该变量赋初值，这些语句也可以用如下的一条语句替代：

```
today = (struct date){9, 25, 2004};
```

请读者注意，这是一条赋值语句而不是变量声明语句，它可以出现在程序的任何位置。语句中的类型转换操作符用于告诉编译器后面的常量表达式的类型，而常量表达式本身的组成规则与我们前面讲过的结构变量初始化表达式的规则相同。

在复合字面量表达式中，我们同样可以指定需要初始化的成员变量的名字，如下面的语句所示：

```
today = (struct date) {.month = 9, .day = 25, .year = 2004};
```

同样，使用成员变量可以让我们按照任意的顺序初始化结构成员变量。如果不使用成员变量名的话，我们就必须按照该结构类型声明时成员变量的顺序来给出相应的初始化值。

下面的一段程序使用了复合字面量来重写了程序9.4的dateUpdate函数。

// 利用复合字面量计算下一天的日期的函数

```
struct date dateUpdate (struct date today)
{
    struct date tomorrow;
    int numberOfDays (struct date d);

    if ( today.day != numberOfDays (today) )
        tomorrow = (struct date) { today.month,
                                     today.day + 1, today.year };
    else if ( today.month == 12 ) // 年尾
        tomorrow = (struct date) { 1, 1, today.year + 1 };
    else // 月末
        tomorrow = (struct date) { today.month + 1, 1, today.year };

    return tomorrow;
}
```

是否在程序中使用复合字面量，这一点取决于读者自己。在前面的例子中，使用复合字面量在某些程度上提高了程序的可读性。

在程序中任何需要结构变量的地方，我们实际上都可以使用复合字面量。下面是一个虽然不太实用，但是却完全合乎语法的例子：

```
nextDay = dateUpdate( (struct date){5, 11, 2004});
```

函数dateUpdate期望的参数类型是一个struct date，而我们的复合字面量也正是这个类型，因此对于编译器来说，这个语句是合法的。

结构数组

读者已经看到，结构可以让我们将逻辑上相关的一组值组织起来，作为一个单独的变量处理。例如，使用time结构，我们不需要跟踪三个独立的变量，而只需要考虑一个变量就可以了。因此，如果我们的程序需要处理10个时间，我们只需要处理10个变量，而不是30个变量。

有一个更好的方式可以帮助我们处理10个时间变量，那就是将C语言提供的两个工具——数组和结构结合起来。C语言并不限制我们只能在数组中保存简单的数据类型，我们同样可以在数组中保存结构，比如下面的语句：

```
struct time experiments[10];
```

定义了一个结构数组，该数组包含10个元素，而每个元素都是一个struct time类型的变量。同样，下面的语句：

```
struct date birthdays[15];
```

定义了一个包含15个struct date类型元素的结构数组。我们可以按照习惯的方式引用结构数组中的元素，比如，要将birthdays数组中的第二个结构变量的值赋为1986年8月8日，我们可以使用下面一组语句：

```
birthdays[1].month = 8;  
birthdays[1].day = 8;  
birthdays[1].year = 1986;
```

如果要将experiments数组中的第4号元素传递给函数checkTime，我们可以使用如下的语句：

```
checkTime(experiments[4]);
```

当然，函数checkTime必须接受struct time类型的参数，其定义可能如下所示：

```
void checkTime (struct time t0)  
{  
    .  
    .  
    .  
}
```


对于结构数组进行初始化和多维数组初始化是类似的，例如下面的语句：

```
struct time runTime [5] =
    { {12, 0, 0}, {12, 30, 0}, {13, 15, 0} };
```

声明了一个拥有5个元素的结构数组runTime，并将前三个元素初始化为12:00:00、12:30:00和13:15:00。内层的大括号实际上是可有可无的，也就是说，下面的语句实际上和上面的初始化语句是等价的。

```
struct time runTime[5] =
    { 12, 0, 0, 12, 30, 0, 13, 15, 0 };
```

下面的语句仅仅初始化了结构数组的第2号元素：

```
struct time runTime[5] =
    { [2] = {12, 0, 0} };
```

而下面的语句，则仅仅初始化了结构数组第1号元素的成员变量hour和minute。

```
static struct time runTime[5] = { [1].hour = 12, [1].minutes = 30 };
```

在程序9.6中，我们建立了一个名为testTimes的结构数组，然后该程序再针对结构数组的每个元素调用了我们在程序9.5中定义timeUpdate函数。为了节省篇幅，我们并没有在源程序清单中列出该函数，相反，我们使用注释语句表明该函数应该存在的位置。

程序9.6中的testTimes结构数组包含5个元素，这些元素分别被初始化为11:59:59、12:00:00、1:29:59、23:59:59和19:12:27这5个不同的值。图9.2可以帮助读者理解在初始化完成之后，该结构数组在计算机内存中是如何保存的。我们可以使用下标0~4来访问结构数组testTimes的某个特定结构元素。为了访问结构元素的成员变量，还需要加上圆点操作符。

程序9.6 演示结构数组

```
// 演示结构数组的程序

#include <stdio.h>

struct time
{
```

程序 9.6 续

```
int hour;
int minutes;
int seconds;
};
int main (void)
{
    struct time timeUpdate (struct time now);
    struct time testTimes[5] =
        { { 11, 59, 59 }, { 12, 0, 0 }, { 1, 29, 59 },
          { 23, 59, 59 }, { 19, 12, 27 } };
    int i;

    for ( i = 0; i < 5; ++i ) {
        printf ("Time is %.2i:%.2i:%.2i", testTimes[i].hour,
               testTimes[i].minutes, testTimes[i].seconds);

        testTimes[i] = timeUpdate (testTimes[i]);

        printf (" ...one second later it's %.2i:%.2i:%.2i\n",
               testTimes[i].hour,
               testTimes[i].minutes,
               testTimes[i].seconds);
    }

    return 0;
}

// ***** Include the timeUpdate function here *****
```

程序 9.6 输出

```
Time is 11:59:59 ...one second later it's 12:00:00
Time is 12:00:00 ...one second later it's 12:00:01
Time is 01:29:59 ...one second later it's 01:30:00
Time is 23:59:59 ...one second later it's 00:00:00
Time is 19:12:27 ...one second later it's 19:12:28
```

结构数组是C语言中非常有用的一个工具，请读者务必理解这部分的内容之后，再继续后续章节的学习。

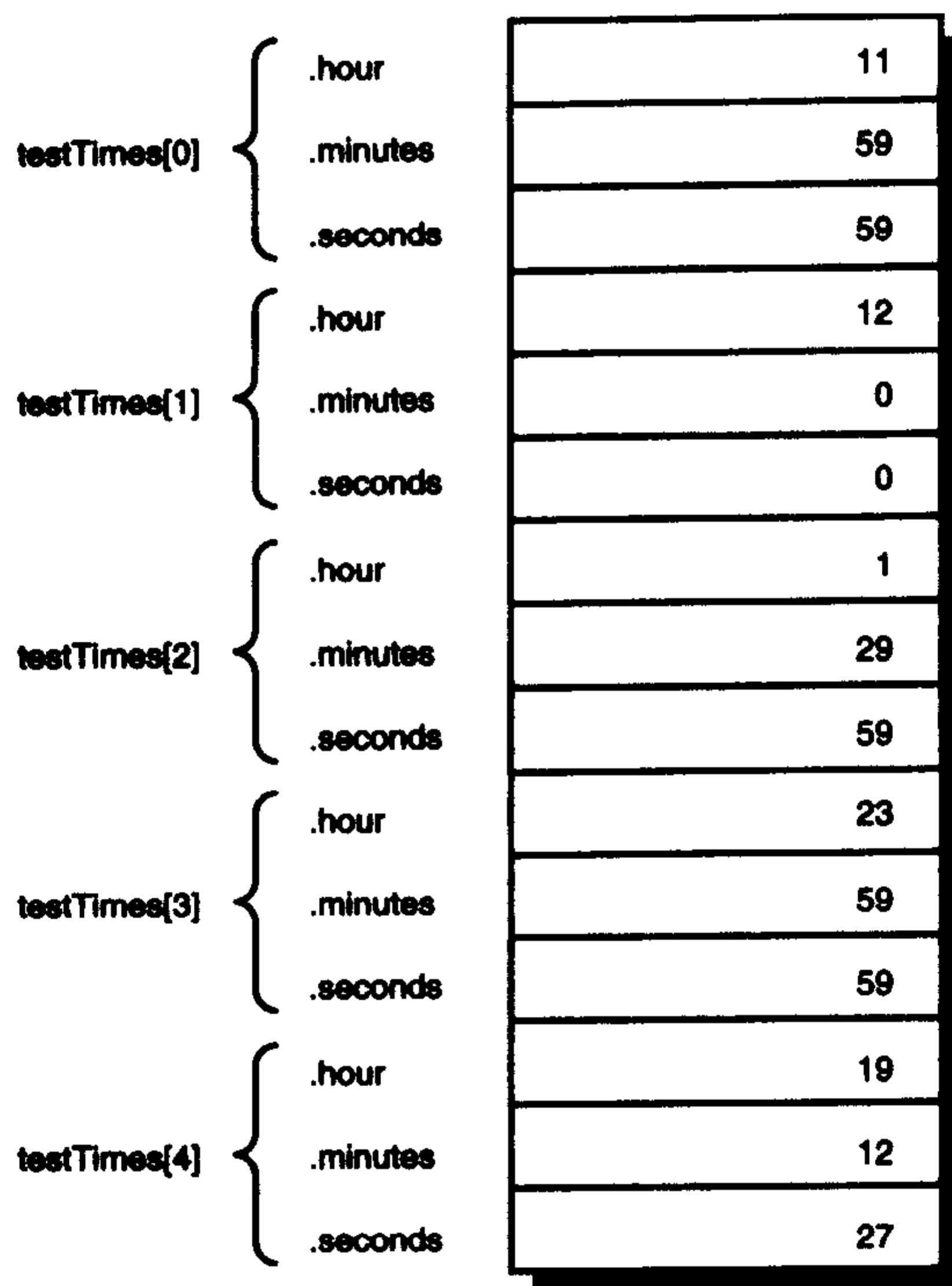


图 9.2 结构数组在内存的表示方法

包含结构的结构

在定义结构方面C语言为我们提供了极大的灵活性。比如，我们可以定义一个结构，其成员变量是另外一个结构，或者，可以定义成员变量为数组类型的结构。

前面已经看到如何把年月日组合成为date结构，而把时分秒组合为time结构。在某些应用场合，我们可能还需要把日期和时间组合起来，例如当我们需要精确记录某些事件发生的时候。

从前面的叙述中，读者可能已经猜到，我们可以很方便地完成上面的任务，那就是定义一个新的结构类型，比如名为dateAndTime，该结构拥有两个成员变量，一个是struct date类型，而另外一个为struct time类型，该结构的定义如下：

```
struct dateAndTime
{
    struct date sdate;
    struct time stime;
};
```

dateAndTime结构的第一个成员变量名为sdate，第二个成员变量名为stime。当然，在定义dateAndTime结构之前，我们必须首先分别定义date结构和time结构。

有了上述类型定义之后，我们就可以声明struct dateAndTime类型的变量了，比如下面的语句：

```
struct dateAndTime event;
```

为了引用结构中的sdate成员，我们可以使用如下的表达式：

```
event.sdate
```

同样，我们也可以把sdate成员传递给前面定义的函数dateUpdate，并将函数的返回值重新赋值给该成员，如下面的语句所示：

```
event.sdate = dateUpdate (event.sdate);
```

你也可以使用dateUpdate结构中的time结构来完成同样的事情：

```
event.stime = timeUpdate (event.stime);
```

为了引用成员变量中的成员，我们还需要在表达式后面再加上圆点操作符，如下所示：

```
event.sdate.month = 10;
```

这条语句将event的发生时间定义为10月份。而下面的语句则将event的发生秒数增加1：

```
++event.stime.seconds;
```

关于包含结构的结构变量的初始化方式，读者想来也不难推测出来，下面的初始化语句将event设置为2004年2月1日，3:30:00。

```
struct dateAndTime event =
    { { 2, 1, 2004 }, { 3, 30, 0 } };
```

我们当然也可以使用成员变量名的初始化方式，如下所示：

```
struct dateAndTime event =
    { { .month = 2, .day = 1, .year = 2004 },
      { .hour = 3, .minutes = 30, .seconds = 0 }
    };
```


我们也可以针对包含结构的结构类型声明数组，如下所示：

```
struct dateAndTime events[100];
```

这个语句声明了一个包含有100个struct dateAndTime类型元素的数组，如果我们要访问该数组中的第四个元素，可以使用表达式events[3]，如果要针对第i号元素调用dateUpdate函数，则可以使用如下的语句：

```
events[i].sdate = dateUpdate (events[i].sdate);
```

如果要将第一个结构元素的时间设置为正午，则可以使用下面一组语句：

```
events[0].stime.hour = 12;  
events[0].stime.minutes = 0;  
events[0].stime.seconds = 0;
```

包含数组的结构

就像本小节的标题所暗示的那样，我们同样可以定义包含数组成员的结构。在包含数组成员的结构类型中，最常见的是字符数组。例如，假定我们需要定义一个名为month的结构，该结构包含两个成员，某个月份的天数和该月份名字的三个字符缩写。该结构的定义如下所示：

```
struct month  
{  
    int numberOfDays;  
    char name[3];  
};
```

上面的语句定义了一个名为month的结构，该结构包含一个名为numberOfDays的整型成员变量和名为name的字符数组成员变量。我们可以按照下面的方式定义类型为struct month的变量：

```
struct month aMonth;
```

使用下面的语句，我们可以给一个类型为struct month的结构变量赋予适当的初值：

```
aMonth.numberOfDays = 31;  
aMonth.name[0] = 'J';  
aMonth.name[1] = 'a';  
aMonth.name[2] = 'n';
```

或者，我们可以用下面的赋值语句达到同样的效果：

```
struct month aMonth = { 31, { 'J', 'a', 'n' } };
```

再进一步，我们可以使用一个包含有12个month元素的数组来表示一年中的所有月份，如下所示：

```
struct month months[12];
```

程序9.7演示了结构数组的作用，该程序的作用只是简单设置数组元素的初值，然后将其打印出来。

通过图9.3，读者也许能够更好地理解结构数组。

程序9.7 演示结构数组

```
// 演示结构数组的程序

#include <stdio.h>

int main (void)
{
    int i;
    struct month
    {
        int numberOfDays;
        char name[3];
    };

    const struct month months[12] =
        { { 31, { 'J', 'a', 'n' } }, { 28, { 'F', 'e', 'b' } },
          { 31, { 'M', 'a', 'r' } }, { 30, { 'A', 'p', 'r' } },
          { 31, { 'M', 'a', 'y' } }, { 30, { 'J', 'u', 'n' } },
          { 31, { 'J', 'u', 'l' } }, { 31, { 'A', 'u', 'g' } },
          { 30, { 'S', 'e', 'p' } }, { 31, { 'O', 'c', 't' } },
          { 30, { 'N', 'o', 'v' } }, { 31, { 'D', 'e', 'c' } } };

    printf ("Month Number of Days\n");
    printf ("-----\n");

    for ( i = 0; i < 12; ++i )
        printf ("%c%c%c %i\n",
            months[i].name[0], months[i].name[1],
            months[i].name[2], months[i].numberOfDays);

    return 0;
}
```


程序9.7 输出

```
Month Number of Days
-----
Jan 31
Feb 28
Mar 31
Apr 30
May 31
Jun 30
Jul 31
Aug 31
Sep 30
Oct 31
Nov 30
Dec 31
```

正像我们在图9.3中看到的那样，表达式

```
months[0]
```

代表的是months数组中的第一个元素，也就是整个month结构。因此，如果我们想要把months[0]作为参数传递给某个函数，该函数必须被声明为接受struct month类型的参数。

再进一步研究程序，我们知道表达式

```
months[0].numberOfDays
```

代表的是数组months中的第一个结构元素的numberOfDay成员变量，因此该表达式的值是一个整型数。而下面的表达式

```
months[0].name
```

代表的是数组months中的第一个结构元素的name成员变量，如果要将该表达式作为参数传递给某个函数，则该函数必须被声明为接受一个字符数组类型的参数。

最后，表达式

```
months[0].name[0]
```

表示的是结构变量months[0]的成员数组name的第一个元素，也就是字符'J'。

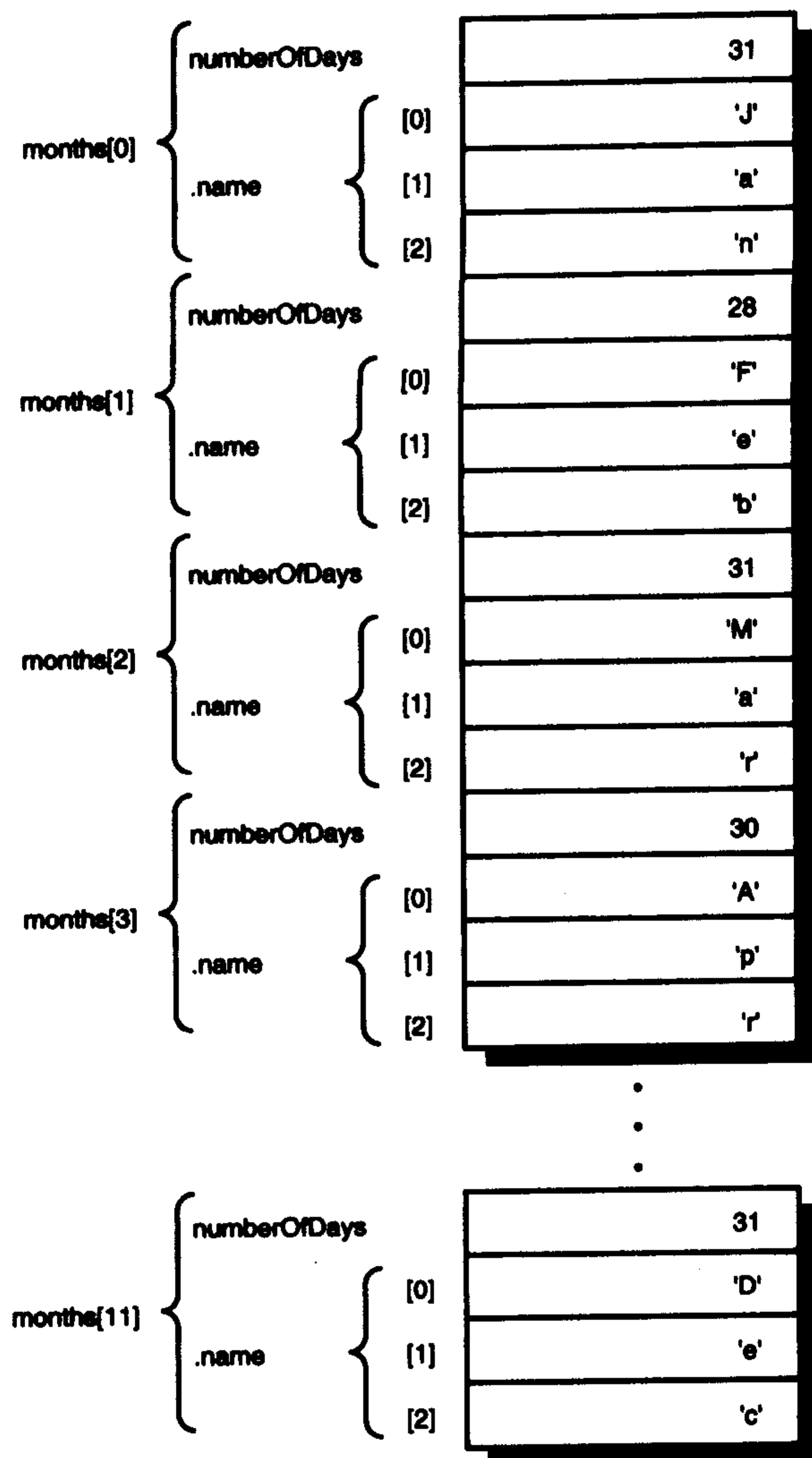


图 9.3 month 数组

结构的变形形式

在定义结构类型和结构变量的时候，我们还可以采用其他一些更灵活的语法形式。比如，C语言允许我们在定义结构类型的时候同时声明结构变量，为了达到这一点，我们只要在结构定义语句的结束分号前面加上变量名就可以了，如下面的语句所示：


```
struct date
{
    int month;
    int day;
    int year;
} todaysDate, purchaseDate;
```

该语句定义了结构类型`struct date`，并且定义了两个该类型的变量`todaysDate`和`purchaseDate`。采用这种方式我们也可以赋予结构变量初值，例如下面的语句：

```
struct date
{
    int month;
    int day;
    int year;
} todaysDate = { 1, 11, 2005 };
```

定义了结构类型`date`，定义了结构变量`todaysDate`，并给其赋予初值。

如果在定义结构类型的时候，同时定义了该类型的所有结构变量，那么实际上可以省去结构类型的名称，比如下面的语句：

```
struct
{
    int month;
    int day;
    int year;
} dates[100];
```

该语句定义了一个包含100个元素的数组`dates`，该数组的每一个元素都是一个结构变量，该结构包含三个整型数：`month`、`day`和`year`。因为我们在声明结构元素数组的时候并没有给出结构的名称，因此如果想要再声明其他同样类型的变量，只能重新定义该结构。

到此为止，我们已经学习了如何使用结构将逻辑上相关的数值用一个变量表达出来，也看到了如何将结构与数组结合起来使用，并将其作为函数的参数。下一章中，我们将学到有关字符数组——也就是字符串的知识。在继续学习之前，还是请读者先完成本章的习题。

练习

1. 输入并运行本章给出的7个程序。将程序的输出结果与书中给出的结果进行对比。
2. 在某些金融领域的应用中，我们常常需要计算两个任意日期之间的天数。比如，2005年7月2日和2005年7月16日之间的天数是14，但是2004年8月8日和2005年2月22日之间有多少天呢？找出这个答案可是需要思考一小会儿的。

幸运的是，我们有一个现成的公式来计算两个日期之间的天数，该公式首先对于任意的日期计算出一个对应的整数N，然后再计算这两个整数之间的差就可以了。N的计算方法如下：

$$N = 1461 \times f(\text{年, 月}) / 4 + 153 \times g(\text{月}) / 5 + \text{日}$$

其中：

$$f(\text{年, 月}) = \begin{cases} \text{year} - 1 & \text{如果 month} \leq 2 \\ \text{year} & \text{其他情况} \end{cases}$$

$$g(\text{month}) = \begin{cases} \text{month} + 13 & \text{如果 month} \leq 2 \\ \text{month} + 1 & \text{其他情况} \end{cases}$$

作为示例，我们使用上述公式计算2004年8月8日和2005年2月22日之间的天数。将各年的年、月、日数代入上述公式可以得到：

$$\begin{aligned} N1 &= 1461 \times f(2004, 8) / 4 + 153 \times g(8) / 5 + 3 \\ &= (1461 \times 2004) / 4 + (153 \times 9) / 5 + 3 \\ &= 2,927,844 / 4 + 1,377 / 5 + 3 \\ &= 731,961 + 275 + 3 \\ &= 732,239 \end{aligned}$$

$$\begin{aligned} N2 &= 1461 \times f(2005, 2) / 4 + 153 \times g(2) / 5 + 21 \\ &= (1461 \times 2004) / 4 + (153 \times 15) / 5 + 21 \\ &= 2,927,844 / 4 + 2295 / 5 + 21 \\ &= 731,961 + 459 + 21 \\ &= 732,441 \end{aligned}$$

$$\begin{aligned} \text{期间的天数} &= N2 - N1 \\ &= 732,441 - 732,239 \\ &= 202 \end{aligned}$$

因此，这两个日期之间的天数是202。上面的公式对于任何1900年3月1日以后的日子都是有效的。（对于1800年3月1日到1900年2月28日之间的日期来说，计算出来的N的值应该再加上1，对于1700年3月1日到1800年2月28日之间的日期来说，计算出来的N值应该再加上2）。

请读者编写一个程序，读取用户输入的两个日期，然后计算这两个日期之间的天数。为了逻辑清楚起见，请试着将程序划分为多个函数。例如，读者应该有一个专门的函数，

该函数接受一个struct date类型的参数，并返回计算出来的N值。这个函数在程序中应该被调用2次，针对用户输入的日期分别计算出N值，然后再计算两个日期之间的天数。

3. 编写一个名为elapsed_time的函数，该函数接受两个类型为struct time的参数，返回一个struct time类型的参数，用于表示两个参数之间流逝的时间（使用时、分、秒的形式）。比如下面语句

```
elapsed_time(time1, time2)
```

如果time1代表3:45:15，而time2代表9:44:03，那么返回的值应该是5小时58分48秒。编写的时候，请注意时间穿越午夜的特殊情况。

4. 如果我们将习题2中计算出来的N值减去621049，然后将结果对于7求余数，我们将会得出一个0~6的数字，该数字代表的日期对应的是星期几（从星期天到星期六）。比如，2004年8月8日对应的N值是732239,732239-621049的结果是111190，而111190%7结果是2，也就是说2004年8月8日是星期2。

编写一个函数，使用上面的公式计算某个特定的日期是星期几。记住最后的程序应该显示出星期的名字，而不是一个数字。

5. 编写一个名为clockKeeper的函数，该函数接受一个类型为struct dateAndTime类型的参数。该函数内部调用timeUpdate函数，如果日期到达了午夜，那么还应该调用dateUpdate函数，最后，将更新过的日期作为一个dateAndTime结构返回给调用者。
6. 将程序9.4中的dateUpdate函数替换为使用复合字面量的版本。重新运行该程序以检验修改的效果。

Character Strings

字符串

本章我们将详细考察C语言中的字符串。我们已经在第3章“编译和运行第一个程序”中初步接触过字符串，在下面的语句中：

```
printf("Programming in C is fun.\n");
```

传递给printf函数的参数就是字符串。

```
"Programming in C is fun.\n"
```

字符串一般使用双引号包围起来，在双引号中我们可以任意使用字母、数字或者特殊字符，只要不包含双引号即可。随后我们还将看到，在字符串中也可以包含双引号。

我们在前面介绍过C语言的char类型，这种类型的变量仅仅能够保存单个字符。如果想要把某个字符赋值给字符变量，字符必须使用单引号括起来。下面的语句将字符 '+' 赋给变量plusSign。

```
plusSign = '+';
```

这里我们假定变量plusSign已经被适当的定义过了。除此之外，我们也介绍过，单引号表达式与双引号表达式有着很大区别，如果plusSign是一个字符类型的变量，那么下面的语句

```
plusSign = "+";
```

就是不正确的。我们在这里再次提醒读者，单引号表达式和双引号表达式代表了C语言中两类不同的常量。

字符数组

如果我们想要保存多个字符，就应该使用字符数组¹。

在程序7.6中，我们用下面的语句定义了字符数组word：

```
char word[] = {'H', 'e', 'l', 'l', 'o', '!' };
```

请读者回忆一下，当我们定义一个数组的时候，如果不指定它的长度，那么C语言编译器会自动根据表达式中元素的个数来计算数组的长度，并分配相应的内存。上面的语句分配了6个字符大小的内存，如图10.1所示。

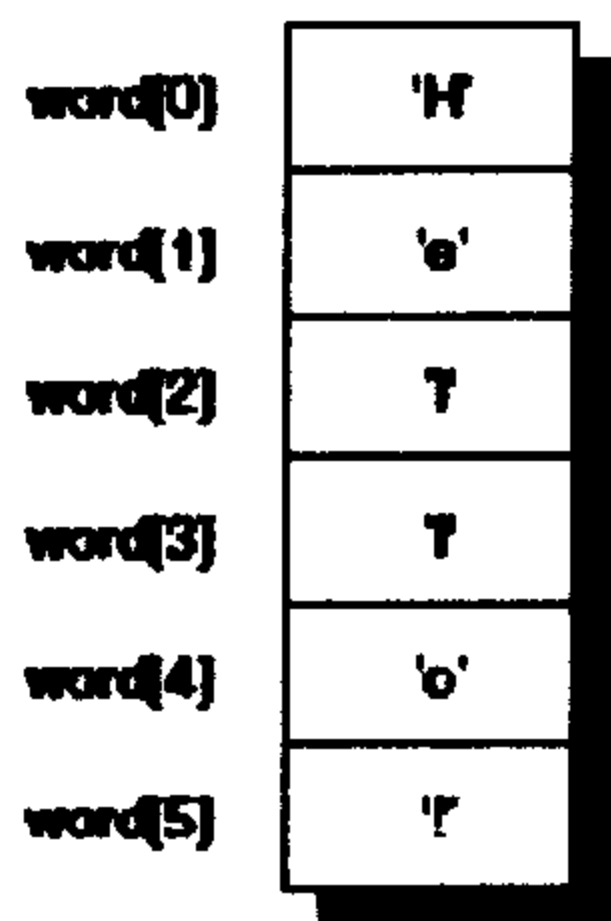


图 10.1 内存中的 word 数组

为了打印字符数组word的内容，我们可以循环遍历数组的每一个元素，使用printf函数和格式描述符%c，将它们打印出来。

通过使用字符数组保存字符串，我们可以定义一些便利函数，专门用于处理字符串。这些常用的功能包括：将两个字符串合并为一个字符串，将一个字符串拷贝到另外一个字符串中，或者提取一个字符串中的某个特定部分，判断某两个字符串是否相等，等等。以上面提到的第一个操作为例，我们下面定义一个用于字符串合并的便利函数。该函数的调用形式如下所示：

```
concat(result, str1, n1, str2, n2);
```

其中，str1和str2代表将要合并的两个字符数组，n1和n2分别代表两个字符数组的长度。这样的函数接口非常灵活，我们可以使用它合并任意长度的两个字符数组。参数result是一个字符数组，其中存放合并后的字符串，合并的时候，str1在前，str2在后。请看程序10.1。

¹ C语言中还可以使用wchar_t来保存字符，该类型的长度虽然大于普通的char，但是它主要用于保存国际字符集中的单个字符，我们本章主要讨论保存多个字符的问题。

程序10.1 合并字符数组

```
// 合并字符数组的函数

#include <stdio.h>

void concat (char result[], const char str1[], int n1,
             const char str2[], int n2)
{
    int i, j;

    // 复制str1到result
    for ( i = 0; i < n1; ++i )
        result[i] = str1[i];

    // 复制 str2到result
    for ( j = 0; j < n2; ++j )
        result[n1 + j] = str2[j];
}

int main (void)
{
    void concat (char result[], const char str1[], int n1,
                 const char str2[], int n2);
    const char s1[5] = { 'T', 'e', 's', 't', ' ' };
    const char s2[6] = { 'w', 'o', 'r', 'k', 's', '.' };
    char s3[11];
    int i;

    concat (s3, s1, 5, s2, 6);

    for ( i = 0; i < 11; ++i )
        printf ("%c", s3[i]);
    printf ("\n");

    return 0;
}
```

程序10.1 输出

Test works.

concat函数中的第一个循环将str1中的字符拷贝到result数组中,这个循环一共执行n1次,n1恰好是str1数组中字符的个数。

concat函数中的第二个循环将str2中的字符拷贝到result数组中,因为字符数组str1中有n1个字符,所以result数组中存放str2数组中的字符位置从n1开始,这个位置刚好紧跟在str1的最后一个字符后面。当这个循环执行完成以后,result数组中存放着字符串str1和str2的连接结果,总共n1+n2个字符。

在主程序中定义了两个常量字符数组s1和s2。s1中存放着5个字符'T','e','s','t'和' '。这里需要提醒读者注意,s1的最后一个字符是一个空格,这在C语言中是完全合法的。第二个数组中存放着6个字符'w','o','r','k','s'和'.'。第三个字符数组s3,定义的长度是11个字符,这个长度刚好能够容纳字符数组s1和字符数组s2的连接结果。因为我们马上将要修改s3的内容,所以它没有定义为常量。

下面的语句

```
concat(s3, s1, 5, s2, 6);
```

调用了我们定义的concat函数,将字符数组s1和s2的内容连接起来,并将结果存放到数组s3中。函数调用中的5,6两个参数分别代表了数组s1和数组s2的长度。

当concat函数执行完成之后,计算机接着执行main函数中concat函数调用后面的循环。s3数组的11个字符被逐个显示在终端上。从程序显示的结果来看,我们的concat函数工作得很好。在前面的例子中,我们假定传递给concat函数用于存放字符串连接结果的字符数组拥有足够的空间,如果不是这样的话,程序运行的时候将会产生不可预料的结果。

可变长度的字符串

按照前面的方式,我们还可以定义很多类似的函数,用于处理字符数组。这些函数的调用参数是一个或者多个字符数组,还有一个或者多个整数,表示字符数组中字符的个数。不幸的是,如果我们长时间使用这些函数,就会发现要记住你在程序中使用的每一个字符数组中包含的字符个数,实在是一件烦人的事情,特别是这些字符数组中存放的字符个数随着程序运行不断变化的时候。我们需要一种方法,能够自动的处理字符数组,而不需要编程人员逐个记住每个字符数组中到底存放着多少字符。

这样的方法是存在的,它的核心思想是在每一个字符串的结尾存放一个特殊的字符。利用这个办法,函数在扫描字符串的时候,如果遇到这个特殊字符,就知道自己已经到达了字符串的结尾。如果我们开发的字符串函数处理的全部是这种类型的字符串,那么在调

用函数的时候我们就不必指定字符串的长度了。

在C语言中，用于指定字符串结束的特殊字符是`null`，或者称为空字符，它的表达式是`'\0'`。下面的语句

```
const char word[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
```

定义了一个包含7个字符的字符数组，这个字符数组的最后一个字符就是空字符。空字符的表达式看上去有些奇怪，因为单引号之间有`\`和`0`两个字符。我们可以回忆一下前面介绍的知识，实际上`\`在C语言中是一个特殊的字符，当它单独出现在表达式中的时候，并不被当作一个字符，因此`'\0'`实际上代表了一个字符。字符数组`word`的存储格式如图10.2所示。

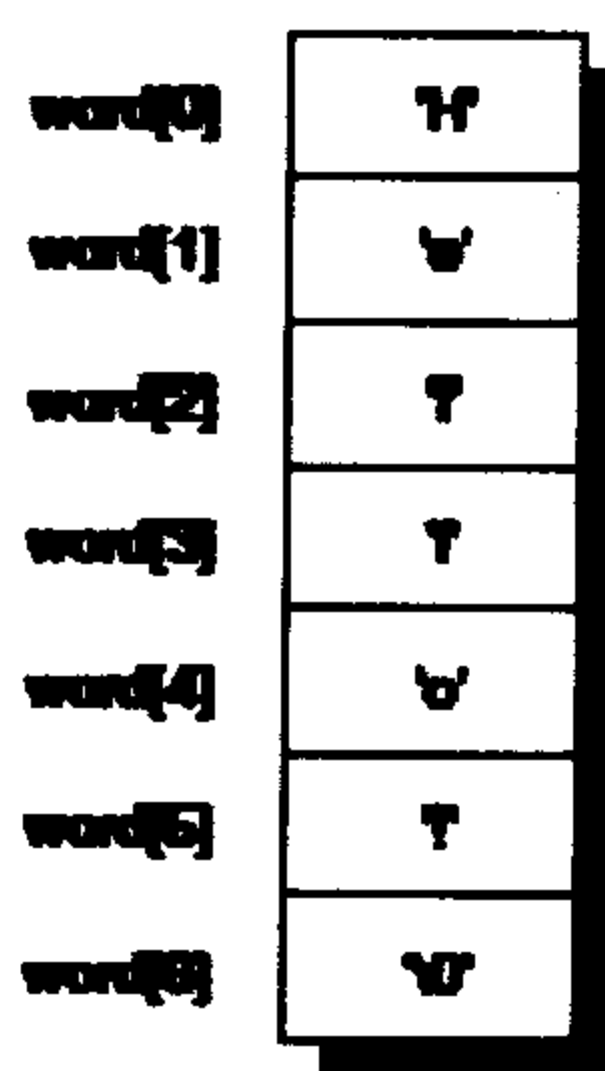


图 10.2 包含一个结束空字符的字符数组 `word`

为了演示如何在程序中使用这种风格的可变长度的字符串，我们首先给出一个计算字符串长度的函数例子，如程序10.2所示。这个函数的名字是`stringLength`，它接受一个以空字符结尾的字符数组作为参数。这个函数计算参数字符数组中的字符个数，并将计数的结果返回给调用它的程序。对于以空字符结尾的字符数组，我们定义它的长度为除了结尾空字符以外所有的字符个数，因此，如果我们定义如下的字符数组：

```
char characterString[] = {'c', 'a', 't', '\0' };
```

那么下面的函数调用

```
stringLength (characterString)
```

的返回值就应该是3。

程序10.2 计算字符串中的字符个数

```
// 计算字符串中的字符个数的函数
```

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

程序10.2 续

```
int stringLength (const char string[])
{
    int count = 0;

    while ( string[count] != '\0' )
        ++count;

    return count;
}

int main (void)
{
    int stringLength (const char string[]);
    const char word1[] = { 'a', 's', 't', 'e', 'r', '\0' };
    const char word2[] = { 'a', 't', '\0' };
    const char word3[] = { 'a', 'w', 'e', '\0' };

    printf ("%i %i %i\n", stringLength (word1),
            stringLength (word2), stringLength (word3));

    return 0;
}
```

程序10.2 输出

5 2 3

stringLength函数的参数被声明为const，因为在函数体中我们仅仅计算字符数组中字符的个数，并不改变数组的内容。

下面我们来仔细考察stringLength函数。在函数的开始，我们定义了一个count变量，并将其初始化为0。接下来程序使用一个while循环来扫描string数组的内容，直到遇到一个空字符。当我们遇到这个表示字符数组结尾的空字符时，就结束while循环并返回count的值。count变量中恰好存放了字符数组string中总的字符个数（不包括结尾的空字符）。我们可以使用一个小一点的字符数组来跟踪测试stringLength函数，以便确认当循环退出的时候，count的值恰好等于字符数组除结尾空字符之外的所有字符的个数。

主程序中定义了三个以null字符结尾的字符数组word1, word2和word3。我们针对这三个字符数组分别调用stringLength函数，并调用printf函数一次显示这三个调用返回的结果。

字符串的初始化和显示

现在，让我们重新看看10.1节编写的连接字符串的函数，并对它进行修改，使之能够接受以空字符结尾的字符串，这样我们就不必传递额外用于表示字符串长度的参数。改写过的函数现在只需要三个参数，两个将要被连接的字符数组和用于存放连接结果的字符数组。

在详细考察新的字符串连接函数之前，让我们先来看看C语言中专门用于处理字符串的两个很好的特性。

第一个特性与字符数组的初始化有关。在初始化字符数组的时候，C语言允许使用单个的常量字符串表达式，而不是一组单个的字符。所以下面的语句

```
char word[] = { "Hello!" };
```

对字符数组word进行了初始化，该语句执行后，word数组中包含'H', 'e', 'l', 'l', 'o', '!', 和 '\0' 共七个字符。另外，C语言还允许在初始化字符串数组的时候省略大括号，所以，下面的表达式

```
char word[] = "Hello!";
```

也是完全正确的。上面的两个初始化语句都和下面的语句等价。

```
char word[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
```

但是书写上要简化很多。

我们也可以在初始化的时候显式指定字符数组的长度，这种时候要注意声明的长度一定包含结尾的空字符。所以对于下面的语句

```
char word[7] = { "Hello!" };
```

编译器在处理的时候可以在字符数组中存放整个字符串（包括结尾的空字符）。但是，对于下面的语句

```
char word[6] = { "Hello!" };
```

编译器就无法在word字符数组中存放结尾的空字符了。（糟糕的是，编译器在这种情况下并不发出警告）

一般来说，C语言中的所有常数字符串都是以空字符结尾的。这样很多常用的函数才能够确定字符串的结尾在哪里。比如下面的语句

```
printf ("Programming in C is fun.\n");
```

编译器会自动在字符串的结尾（也就是换行符后面）放置一个空字符，printf函数利用这一点确定它在扫描显示格式字符串的时候，是否到达了结尾。

第二个特点与字符串的显示有关。在printf函数中，我们可以使用格式描述字符%s来显示空字符结尾的字符串。因此，如果word是一个空字符结尾的字符数组，下面的printf语句

```
printf ("%s\n", word);
```

可以在终端上显示整个word字符数组的内容。printf函数假定在格式字符串中遇到的%s描述符对应一个空字符结尾的字符数组。

程序10.3展示了我们修订之后的concat函数，其中的main函数使用了我们上面说明的这两个C语言的特点。因为我们不再把字符串的长度传递给concat函数，因此函数必须在运行中检查当前字符是否是空字符，用于决定是否到达了字符串的结尾。同时，在把str1的字符拷贝到result数组中时，千万不要拷贝str1的结尾空字符，因为那样会使得其他处理result数组的函数以为result代表的字符串的结尾在那里。然而，当我们把字符数组str2的内容拷贝到字符数组result中之后，要记得在最后放置一个空字符，用于标识字符数组result中存放的字符串的结尾。

程序10.3 连接字符串

```

#include <stdio.h>

int main (void)
{
    void concat (char result[], const char str1[], const char str2[]);
    const char s1[] = { "Test " };
    const char s2[] = { "works." };
    char s3[20];

    concat (s3, s1, s2);

    printf ("%s\n", s3);
    return 0;
}

// 连接两个字符串的函数
void concat (char result[], const char str1[], const char str2[])
{
    int i, j;

    // 复制str1到result
    for ( i = 0; str1[i] != '\0'; ++i )
        result[i] = str1[i];

    // 复制str2到sresult

```

程序10.3 续

```
for ( j = 0; str2[j] != '\0'; ++j )
    result[i + j] = str2[j];

// 使用null字符作为连接字符串的结尾
result [i + j] = '\0';
}
```

程序10.3 输出

Test works.

在concat函数的第一个循环中, 字符数组str1中的字符被逐个拷贝到字符数组result中, 直到遇到空字符为止。因为一旦遇到空字符, 循环就退出, 所以str1的结尾空字符不会被拷贝到result中。

在第二个循环中, 字符数组str2中的字符被拷贝到result中紧跟着str1的位置后面。因为当第一个循环结束的时候, 循环变量i的值刚好等于str1中的字符个数(不包括结尾空字符), 因此下面的赋值语句

```
result[i+j] = str2[j];
```

正好将字符数组str2中的字符拷贝到result中的适当位置。

当第二个循环完成之后, concat函数后面的语句在result的结尾位置放置一个空字符。读者务必认真研究上面这段程序中循环变量i和j的用法。很多人在使用下标存取字符数组中的字符时, 都容易犯下标位置错一位的错误。

在处理有关空字符结尾字符数组编程问题时, 必须牢记以下几点: 1. 字符数组的第一个字符下标是0; 2. 如果字符数组string不算结尾的空字符共包含n个字符时, 那么表达式string[n-1]指的是字符串的最后一个非空字符, 而表达式string[n]指的是结尾的空字符。另外, 在定义字符数组string的时候, 大小必须是n+1, 以便存放结尾的空字符。

让我们回到程序10.3。main函数定义了两个字符数组s1和s2, 并使用我们前面刚刚介绍的方法对它们进行了初始化。字符数组s3的长度被定义为20, 以便编译器能够分配足够大的空间, 存放s1和s2的连接结果, 并为我们免去精确计算连接后字符串大小的麻烦。

然后我们调用concat函数, 并将字符数组s1, s2和s3作为参数。字符串连接的结果保存在字符数组s3中, 我们使用%s格式描述符调用printf函数将它显示出来。虽然s3定义为拥有20个字符大小的字符数组, 但是printf函数仅仅显示到空字符为止。

检验字符串相等

C语言不允许我们使用==操作符来检验两个字符串是否相等。下面的语句

```
if ( string1 == string2 )
```

是不合法的，因为==操作符只能用于简单的变量类型，例如float，int或者char，而不能用于复杂的数据类型，例如结构或者数组。

为了检验两个字符串是否相等，我们必须逐个比较这两个字符串的所有字符。如果在比较的过程中，我们同时到达两个字符串的结尾（空字符），而且在此之前比较的所有字符都相同，那么这两个字符串才是相等的，否则它们不相等。

为了简化字符串的比较工作，我们编写一个用于比较字符串是否相等的函数，如程序10.4所示。我们把这个函数命名为equalStrings，函数的参数是需要比较的两个字符串。因为我们只关心两个字符串是否相等，因此我们可以让equalStrings函数的返回值是bool类型，当两个字符串相等时，返回true（或者非零值），当它们不相等的时候，返回false（或者零值）。这样的话，我们就可以在表达式中直接使用该函数，如下面的语句所示：

```
if ( equalStrings( string1, string2) )
    ...
```

程序10.4 检验字符串是否相等

```
// 检验字符串是否相等的函数

#include <stdio.h>
#include <stdbool.h>

bool equalStrings (const char s1[], const char s2[])
{
    int i = 0;
    bool areEqual;

    while ( s1[i] == s2[i] && s1[i] != '\0' && s2[i] != '\0' )
        ++i;

    if ( s1[i] == '\0' && s2[i] == '\0' )
        areEqual = true;
    else
        areEqual = false;
}
```


程序10.4 续

```
    return areEqual;
}

int main (void)
{
    bool equalStrings (const char s1[], const char s2[]);
    const char stra[] = "string compare test";
    const char strb[] = "string";

    printf ("%i\n", equalStrings (stra, strb));
    printf ("%i\n", equalStrings (stra, stra));
    printf ("%i\n", equalStrings (strb, "string"));

    return 0;
}
```

程序10.4 输出

```
0
1
0
```

equalStrings函数使用一个while循环遍历字符串s1和s2。循环继续的条件是两个字符串的当前字符相等(s1[i] == s2[i])，而且其中的任何一个都没有到达结尾(s1[i] != '\0' && s1[i] != '\0')。循环变量i用来作为两个字符数组的下标，每次循环都增加1。

循环语句后面的if条件语句用于判断我们在扫描字符串s1和s2的时候，是否同时到达了两个字符串的结尾。语句

```
if( s1[i] == s2[i] )
    ...
```

可以达到同样的效果。如果我们同时到达了字符串s1和s2的结尾，那么这两个字符串肯定是相等的，这时我们设置变量areEqual为true，并将它返回给调用的函数。否则的话，这两个字符串是不相等的，我们设置变量areEqual为false，然后返回。

在main函数中，我们声明了两个字符数组stra和strb，并将其分别初始化。随后，在程序中我们第一次调用equalString函数，并将这两个字符数组作为参数传递进去。因为这两个字符串的值并不相等，因此函数的返回值是false，或者说是0。

在程序中第二次调用equalString的时候，我们将stra传递给它两次。这次该函数判断出这两个参数是相等的，因此返回值为true。

程序中对于equalString函数的第三次调用要更有趣一些。从代码中我们可以看到，虽然该函数声明接受的参数类型是字符数组，我们也可以将常量字符串传递给它。在第11章“指针”中，我们将会理解这种写法的工作原理。这次函数将strb与常量字符串“string”进行比较，最后的结论是这两个字符串是相等的。

输入字符串

截至目前为止，读者对于使用格式化输出符号%s输出字符串的做法已经熟悉了。但是我们如何从终端上读入一个字符串呢？实际上，在C语言中，有若干个库函数可以帮助我们完成这项工作。如果使用scanf函数，并使用格式化输入符号%s，那么该函数将从终端上读入一个字符串，直到遇到空格、制表符或者换行符中的任何一个为止。因此，下面的语句：

```
char string[81];  
scanf ("%s", string);
```

将会读取用户在终端上输入的字符串，并将其存储在字符数组string中。这里提醒读者注意，当把字符数组作为参数传递给scanf函数的时候，我们不需要在变量名前面加上&符号（关于这一点的原因，我们将在第11章说明）。

对于前面的scanf语句，如果用户在终端上输入下面的字符串：

```
Shawshank
```

那么scanf函数就将读取该字符串，并将其保存到string字符数组中。如果用户输入的是下面的内容：

```
iTunes playlist
```

那么只有iTunes被保存在string字符数组中，因为scanf函数在读入的时候如果遇到空白、Tab或者换行符，就会停止读取。此时，如果我们在程序中再次调用scanf函数，那么剩下的字符串playlist就将被读入，因为scanf总是接着上次读入的字符串继续执行读取操作。

当使用scanf函数读取字符串的时候，它会自动给字符串后面增加一个空字符null。因此，如果我们在程序中调用scanf函数，而用户在终端输入如下的字符串：

```
abcdefghijklmnopqrstuvwxy
```

那么字符数组string中将包含小写字母表，而数组元素string[26]则等于空字符。

如果我们定义了三个具有合适大小的字符数组s1、s2和s3，并使用如下的scanf语句：

```
scanf ("%s%s%s", s1, s2, s3);
```

那么，当用户在终端上输入如下的字符串时：

```
micro computer system
```

最终s1中将包含字符串micro，s2中包含computer，而s3中包含system。如果我们输入如下的字符串：

```
system expansion
```

那么scanf函数将字符串system保存在字符数组s1中，expansion保存到字符数组s2中，因为用户没有输入其它的字符，因此scanf函数将进入等待状态，直到用户输入更多的字符。

在程序10.5中，我们使用scanf函数从终端读取字符串。

程序10.5 使用scanf函数读取字符串

```
// 使用scanf函数读取字符串的函数

#include <stdio.h>

int main (void)
{
    char s1[81], s2[81], s3[81];

    printf ("Enter text:\n");

    scanf ("%s%s%s", s1, s2, s3);

    printf ("\ns1 = %s\ns2 = %s\ns3 = %s\n", s1, s2, s3);
    return 0;
}
```

程序10.5 输出

```
Enter text:
system expansion
bus
```

```
s1 = system
s2 = expansion
s3 = bus
```

在程序10.5中，我们使用scanf函数读取三个字符串s1、s2和s3。因为用户输入的第一行只包含两个字符串（要记住scanf函数将空白、Tab和换行作为字符串的结束），因此程序将等待用户输入更多的字符。当我们在第二行上输入更多的字符之后，程序使用printf语句将保存在三个字符数组的字符串分别打印出来，打印结果显示scanf函数正确的完成了它的工作。

如果用户在终端上连续输入80个以上的字符，并且在其中不加入空格、Tab或者换行符的话，scanf读取的内容将超过字符数组的容量。这有可能导致程序意外终止，或者发生不可意料的行为。不幸的是，scanf函数没有办法能够知道我们传递给它的字符数组有多大。当使用格式化输入符号%s读取字符串的时候，scanf将一直读入字符，直到遇到终止符号（空格、Tab或者换行符）为止。

如果我们在格式化输入符号%s的百分号后面放置一个数字的话，就可以避免这个问题。这个数字用于告诉scanf函数，我们能够读取的字符的最大数量。因此，如果我们使用下面的语句替代程序10.5中的scanf语句，那么scanf函数就不会向s1、s2或者s3中存储超过80个字符（我们还必须为字符串的结尾空字符留出必要的空间，因此我们使用%80s而不是%81s）。

```
scanf ("%80s%80s%80s", s1, s2, s3);
```

单字符输入

在C语言的标准库中还有好几个可以用于终端输入输出的函数，从读写单个字符到整个字符串。例如，getchar函数可以用于从终端读取一个字符，多次调用该函数，我们就可以逐个的读取用户在终端上的输入，如果读入到达了一行的末尾，该函数则返回换行符'\n'。因此，如果用户在终端上输入字符串'abc'，然后再按下回车键，则第一次调用getchar函数返回字符'a'，第二次调用getchar函数返回字符'b'，第三次调用返回字符'c'，第四次调用则返回换行符'\n'。如果我们第五次调用该函数，那么程序的执行将被暂停，直到用户输入更多的字符。

读者也许要问，我们完全可以使用格式化输入符号%c，通过scanf函数来读取单个字符，为什么还要使用getchar函数。使用getchar函数的好处在于设计该函数的唯一目的就是读取单个字符，因此该函数使用比较方便，我们不需要传递任何参数给它。该函数将读取的字符作为返回值直接传递给调用者，我们可以将这个值赋于某个变量，也可以在表达式中直接使用它。

在很多文字处理方面的应用中，我们常常需要一次读入一行文字，然后将其保存在某个缓冲区中，以便随后对其进行处理。使用scanf函数和格式化输入符号%s并不能完成这件工作，因为scanf函数在遇到空白的时候就会停止读取（而一行文本中通常都会有空白字符）。

在C语言的标准库函数中有一个名为gets的函数，该函数的主要作用就是读入一行文本。作为一个编程练习，我们在程序10.6中以getchar函数为基础，编写了一个功能与gets类似的函数，该函数名为readLine。readLine函数需要一个字符数组类型的参数，读取的文本将被存放到该字符数组之中。该函数返回从终端上读取的一行文本，但是不包括结尾的换行符。

程序10.6 读取一行数据

```
#include <stdio.h>

int main (void)
{
    int i;
    char line[81];
    void readLine (char buffer[]);

    for ( i = 0; i < 3; ++i )
    {
        readLine (line);
        printf ("%s\n\n", line);
    }

    return 0;
}

// 从终端读入一行文字的函数
void readLine (char buffer[])
{
    char character;
    int i = 0;

    do
    {
        character = getchar ();
        buffer[i] = character;
        ++i;
    } while ( character != '\n' );

    buffer[i - 1] = '\0';
}
```

程序10.6 输出

```

This is a sample line of text.
This is a sample line of text.
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
runtime library routines
runtime library routines
    
```

readLine函数中使用一个do循环将用户在终端上输入的字符保存到缓冲区中，在循环中我们使用getchar函数读取字符，并依次存放到字符数组的下一个位置。如果我们遇到了换行符，这表明一行已经结束，我们就退出do循环。随后，我们在字符数组的结尾用空字符null覆盖掉最后一次读入的换行符。因为在循环最后一次执行的时候，我们的循环变量已经增加过了，因此字符数组的最后一个位置的下标应该是i-1。

在主程序中，我们定义了一个能够容纳81个字符的字符数组。这样保证了该数组可以用于存放一整行的文字再加上结尾的空字符（由于历史原因，标准终端的一行的长度被定义为80个字符）。但实际上，即使终端的一行长度不超过80个字符，但如果用户输入到一行行末的时候不按回车键，而是继续输入的话（大多数操作系统这时都会允许用户输入、并自动回显到下一行），我们的程序还是有缓冲区溢出的危险。一个更好的做法是把缓冲区的长度作为第二个参数传递给readLine函数，这样函数就可以保证不将多于缓冲区大小的字符串保存在缓冲区中了。

主程序随后进入一个for循环，在该循环中我们读取三行字符，每读取一行之后，就将其回显给用户。读完三行文字之后，我们的程序就结束执行。

下一个程序（程序10.7），我们将编写一个实用的文字处理程序——统计一个字符串中有多少个单词。程序中包含一个名为countWords的函数，该函数接受一个字符数组参数，并将其中的单词数作为返回值。为了简单起见，我们不妨定义单词就是英文字母的序列，countWords函数将对字符串进行扫描，然后将碰到的第一个字母作为某个单词的开始，继续向后扫描，直到遇到某个非字母的字符，我们就认为已经找到了一个单词，并对剩余的字符串继续执行上述的扫描过程。

程序10.7 计算单词的个数

```
// 判断一个字符是否为字母的函数

#include <stdio.h>
#include <stdbool.h>

bool alphabetic (const char c)
{
    if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
        return true;
    else
        return false;
}

/* 计算一个字符串中单词数的函数 */

int countWords (const char string[])
{
    int i, wordCount = 0;
    bool lookingForWord = true, alphabetic (const char c);

    for ( i = 0; string[i] != '\0'; ++i )
        if ( alphabetic(string[i]) )
        {
            if ( lookingForWord )
            {
                ++wordCount;
                lookingForWord = false;
            }
        }
        else
            lookingForWord = true;

    return wordCount;
}

int main (void)
{
    const char text1[] = "Well, here goes.";
    const char text2[] = "And here we go... again.";
    int countWords (const char string[]);

    printf ("%s - words = %i\n", text1, countWords (text1));
    printf ("%s - words = %i\n", text2, countWords (text2));

    return 0;
}
```

程序10.7 输出

```
Well, here goes. - words = 3  
And here we go... again. - words = 5
```

程序中的`alphabetic`函数很简单，它接受一个字符类型的参数，并检验其是否在大写字母或者小写字母的范围之内，如果是的话，该函数返回布尔值`true`，否则的话返回`false`。

但是`countWords`函数就不是那么简单了。首先我们介绍一下几个局部变量的用途：变量`i`作为下标，用于遍历整个字符串；整型变量`lookingForWord`实际上起一个标志的作用，用于表示当前程序是否正在查找一个新词的开始字母。在程序刚开始执行的时候，很明显我们在查找第一个词，所以该变量在初始化的时候被指定为`true`，局部变量`wordCount`则用于统计字符串中单词的个数。

对于字符串中的每一个字符，我们调用`alphabetic`函数判断其是否是一个字母，如果该字符是一个字母，我们再检查`lookingForWord`标志，如果该标志为`true`，我们就给变量`wordCount`加1，然后将标志`lookingForWord`设置为`false`，表示我们现在正在一个词的内部进行扫描。

如果当前的字符是一个字母，而`lookingForWord`标志的值等于`false`，这说明我们正在一个单词的内部扫描，在这种情况下，我们不进行任何处理，简单的开始下一次循环即可。

如果当前字符不是字母，这说明我们遇到了某个单词的结尾，或者还没有找到下一个单词的开始。无论是哪一种情况，我们都将标志`lookingForWord`设置为`true`（即使它的值可能已经是`true`）。

当字符串的所有字符都被扫描完了之后，我们将变量`wordCount`的值作为找到的单词的个数，返回给调用者。

如果我们将`countWords`函数执行过程的每一步，各个变量的值制作成为一个表格，对于读者理解这个程序将会有很大的帮助。这个表格如表10.1所示，该表格显示了主程序中第一次调用`countWords`函数的执行过程。该表的第一行显示了循环开始之前各个变量的值，以后各行显示相应的字符被处理之后，变量的状态。因此，第二行代表了字符串中的第一个字符（也就是字符‘w’）被读入并处理之后，各个变量的状态。表格的最后一行显示了程序终止执行之后各个变量的值。读者应该花点时间仔细研究一下这个表格，研究完后，读者就应该对于函数`countWords`中的算法有更深一步的了解。

表10.1 函数countWords的执行过程

i	string[i]	wordCount	lookingForWord
0	true		
0	'W'	1	false
1	'e'	1	false
2	'l'	1	false
3	'l'	1	false
4	','	1	true
5	' '	1	true
6	'h'	2	false
7	'e'	2	false
8	'r'	2	false
9	'e'	2	false
10	' '	2	true
11	'g'	3	false
12	'o'	3	false
13	'e'	3	false
14	's'	3	false
15	'.'	3	true
16	'\0'	3	true

空字符串

现在让我们考虑使用前面完成的countWords函数来完成一些更有实际意义的工作。我们将使用readLine函数从终端上读取用户的输入，然后统计用户输入中单词总的个数，并将结果显示在终端上。

为了使得程序更加灵活，我们将不限制用户输入的行数。因此，我们需要找到一种方式，用于通知程序用户已经完成了输入。有一个简单的方法可以完成这个任务，那就是当用户输入最后一行文字以后，再额外多输入一个空行。当我们使用readLine函数读取这个额外的空行时，程序将在用作缓冲区的数组的第一个位置上存储一个空字符。我们程序可以专门检查这种情况，并由此得知用户已经完成了输入。

一个只包含空字符的字符串在C语言中有一个专门的名称：空字符串(*null*字符串)。回想一下，我们前面编写的程序实际上也可以很好的处理空字符串。比如，stringLength对于空字符串将返回0值，而concat函数可以将空字符串和其他任何字符串连接起来，另外，equalString也可以将空字符串和其他任何字符串进行比较，当传递给它两个空字符串时，它也能正确的判断出这两个字符串是相等的。

读者需要牢记下面的事实：空字符串中也是有一个字符的，不过它是一个空字符罢了。

有些时候，我们需要将程序中的某些字符串设置为空字符串，在C语言中，空字符串用两个相连的双引号表示。比如，下面的语句定义了一个名为buffer的字符数组，并将其值设置为空字符串。

```
char buffer[100] = "";
```

这里请读者务必注意，空字符串""和字符串" "是不同的，后者并不是空字符串，其中包含一个空格。如果读者还是感到疑惑的话，可以用我们前面编写的equalString函数对这两个字符串进行比较，看看函数返回的结果。

程序10.8使用了我们前面编写的readLine、alphabetic和countWords函数，为了节省空间起见，在程序清单中我们没有列出这些函数。

程序10.8 计算一段文本的单词数目

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

/***** 在这里插入alphabetic函数 *****/
/***** 在这里插入readLine函数 *****/
/***** 在这里插入countWords函数 *****/

int main (void)
{
    char text[81];
    int totalWords = 0;
    int countWords (const char string[]);
    void readLine (char buffer[]);
    bool endOfText = false;

    printf ("Type in your text.\n");
    printf ("When you are done, press 'RETURN'.\n\n");

    while ( ! endOfText )
    {
        readLine (text);
        if ( text[0] == '\0' )
            endOfText = true;
    }
}
```

程序10.8 续

```
        else
            totalWords += countWords (text);
    }
    printf ("\nThere are %i words in the above text.\n", totalWords);

    return 0;
}
```

程序10.8 输出

Type in your text.
When you are done, press 'RETURN'.

Wendy glanced up at the ceiling where the mound of lasagna loomed like a mottled mountain range. Within seconds, she was crowned with ricotta ringlets and a tomato sauce tiara. Bits of beef formed meaty moles on her forehead. After the second thud, her culinary coronation was complete.

Enter

There are 48 words in the above text.

请读者注意，写着*Enter*的那一行输入只是表示按下Enter键或者Return键。

在程序中，我们使用标志变量endOfText来表示用户是否已经完成了输入。只要这个标志为false，程序中的while循环就一直执行。在循环体中，我们调用readLine函数从终端读取一行输入，随后的if语句用于判断用户是否只按下了Enter键（也就是说读入的字符串是否是空字符串）。如果是这样，我们设置endOfText变量为true，用于指示用户的输入已经完成了。

如果缓冲区中确实包含某些文本，我们就调用countWords函数，计算这些文本中包含的单词个数，并把该函数的返回值加到变量totalWords上。totalWords用于统计所有用户输入中单词的总数。

当while循环结束之后，我们的程序打印出单词的总数，随后结束运行。

读者可能觉得程序10.8并不是很有用处，因为我们还需要在终端上手动的输入所有的文本。在学完第16章“C语言的输入输出”之后，我们就可以看到，这个程序也可以用来统计某个磁盘文件中单词的个数。如果某位作者使用计算机来输入其作品，并将作品保存为磁盘文件，那么这个程序就能够很方便的统计出手稿中的单词数量（当然，我们需要假定这个手稿保存为普通的文本文件，而不是专有的格式，比如Microsoft Word）。

转义字符

我们前面已经提到过，反斜线\在C语言的字符串中有特殊的意思，比如用来构成换行符和空字符等。除了反斜线和字母n搭配组成换行符之外，很多其他字符和反斜线搭配时，也有特定的含义。这些字符连同反斜线一起，被称为转义字符。C语言中的转义字符如表10.2所示。

表10.2 转义字符

转义字符	字符名 (Character Name)
\a	警铃
\b	退格
\f	表单输入
\n	换行
\r	回车
\t	水平制表
\v	垂直制表
\\	反斜线
\"	双引号
\'	单引号
\?	问号
\nnn	八进制数nnn代表的字符
\unnnn	通用字符名
\Unnnn	通用字符名
\xnn	十六进制数nn代表的字符

在绝大多数输出设备上，前7个转义字符都能够完成其相应的功能。比如警铃字符\a，用于使计算机发出警铃声。因此下面的printf语句将使计算机发出警铃声，然后打印出相应的信息。

```
printf ("\aSYSTEM SHUT DOWN IN 5 MINUTES!!\n");
```

如果一个字符串包含回退字符'\b'，那么当我们使用printf语句输出该字符串的时候，计算机在遇到\b时，将会删除上一个输出的字符。还有一个转义字符——制表符\t，用于将随后的输出移到下一个制表符的位置。（一个制表符的宽度通常是8个字符）制表符对于对齐按列输出的数据时很有用处。例如，下面的语句：

```
printf ("%i\t%i\t%i\n", a, b, c);
```

在终端上显示a的值，然后移到下一个制表符的位置，显示b的值，接着再移到下一个制表符位置显示c的值。

如果我们需要在输出中包含一个反斜线，我们应该使用两个反斜线字符。比如，下面的printf语句：

```
printf ("\\t is the horizontal tab character.\\n");
```

其实际输出将会如下所示：

```
\\t is the horizontal tab character.
```

读者需要注意，因为在字符串中先是两个反斜线，然后再是字符t，所以C语言编译器将其当作是（想要输出的）一个反斜线字符和字母t，而不是一个反斜线和一个制表符。

为了在字符串中包含双引号，我们必须在前面加上反斜线。因此下面的printf语句：

```
printf ("\"Hello,\" he said.\\n");
```

其输出如下：

```
"Hello," he said.
```

为了将单引号赋值给一个字符变量，我们也需要在该单引号之前加上反斜线。因此，如果变量c是一个字符类型的变量，那么下面的语句将单引号赋值给该字符。

```
c = '\\';
```

如果在问号?前面加上一个反斜线，C语言还是将其当作一个问号。在处理非ASCII字符集中的三元字符的时候，这种表达法很有用处。关于这方面的更多细节，请参阅附录A“C语言小结”。

最后四种转义字符允许我们在C语言的字符串中包含任意的字符。如果使用'\nnn'的转义形式，那么nnn就是一个八进制数，如果使用'\xnn'的转义形式，那么nn就是一个十六进制的数字。这些数字实际上是我们想要在字符串中包含的字符的编码。例如，如果我们想要在字符串中包含一个ASCII的ESC字符，因为这个字符的编码是33，因此我们可以使用转义字符\033或\x1b。

对于前面所说的转义字符编码规则来说，空字符'\0'是一个例外。该字符表示值为0的字符。实际上，因为在C语言中空字符的值为0，程序员们常常使用这一点来简化处理字符串的程序。比如，程序10.2中strlen函数用于计算字符串长度的循环可以采用下面的简化形式：

```
while ( string[count] )  
    ++count;
```

如果当前的字符不是空字符，那么表达式的值就不为0，而一旦遇到了空字符，循环就将退出运行。

这里我们再次提醒读者，转义字符整体上应当被当作是单个字符。因此，字符串"\033\"Hello\"\\n"实际上只包含9个字符（如果不计算结尾的空字符的话）：字符'\033'、

字符`\"`、Hello和最后双引号字符和换行符。如果我们将这个字符串传递给我们编写的`strlen`函数，该函数的返回结果将是9（不包括结尾的空字符）。

通用字符名由反斜线、小写字母`u`后面跟上4个十六进制数或者大写字母`U`后面跟上8个十六进制数字构成。这些转义字符用于表示扩展字符集中的字符，这些字符的编码长度超过了标准的8位二进制数所能表示的范围。通用字符名可以用于表示16位或者32位的字符，关于这方面更多的信息，请参阅附录A。

关于字符串常量的进一步讨论

如果我们在一行代码的行尾放置一个反斜线，C语言编译器将会忽略行尾的换行符，而把下一行的内容也算作是本行的内容。这里的反斜线起到了续行的作用。构建较长的字符串是续行的一个常见用途，在第13章“预处理器”中，我们还将使用这个办法定义跨行的宏。

如果不使用反斜线，当我们试图初始化一个跨多行的字符串时，C语言编译器就会发出警告，如下面的语句所示：

```
char letters[] =
{ "abcdefghijklmnopqrstuvwxy
ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

但是如果我们在行尾使用反斜线，那么就可以把字符串常量跨行书写，如下所示：

```
char letters[] =
{ "abcdefghijklmnopqrstuvwxy \
ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

我们从续行的开始继续输入字符串，这样可以避免在整个字符串中加入多余的空格。综上所述，上面的语句定义了一个字符数组`letter`，并将其初始化为如下的初值：

```
"abcdefghijklmnopqrstuvwxyABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

C语言中还有一种拆分长字符串的方法，那就是将其写成多个相邻的字符串。这些字符串之间用0个、多个空白、制表符或者换行符隔开。编译器在编译源代码的时候，会自动

将这些字符串连接起来。因此，下面的表达式

```
"one" "two" "three"
```

实际上相当于下面的表达式

```
"onetwothree"
```

因此，前面跨行的初始化语句也可以使用下面的形式完成：

```
char letters[] =  
{ "abcdefghijklmnopqrstuvwxy"z"  
  "ABCDEFGHIJKLMNopQRSTUVWXYZ" };
```

我们最后再给出一个例子，下面的三条printf语句实际上都只接受了一个参数，因为C语言编译器会自动的将第二个和第三个调用中的参数合并起来。

```
printf ("Programming in C is fun\n");  
printf ("Programming" " in C is fun\n");  
printf ("Programming" " in C" " is fun\n");
```

字符串、结构与数组

前面我们已经看到，我们可以将C语言提供的基本元素按照多种方式组合起来形成非常有用的工具。比如在第9章“使用结构”中，我们看到了如何使用结构数组。下面的程序10.9进一步展示了如何结合使用结构、数组以及变长字符串。

假定我们要编写一个字典程序。我们可以使用该程序来查询那些我们不太明白的单词。比如，我们启动字典程序，然后输入该单词，随后该程序就会自动地在字典中查找该单词的解释。

编写这个程序需要解决的第一个问题就是如何在计算机中表示某个单词以及其对应的解释。很明显，因为一个单词和它的解释在逻辑概念上是关联的，因此我们很容易想到使用结构来处理它们。我们可以定义如下的结构entry，用于保存单词以及其解释：

```
struct entry  
{  
    char word[15];  
    char definition[50];  
};
```

在结构entry中，我们定义了单词长度最大为14个字母（因为我们还要为结尾的空字符留出空间），其解释的最大长度为49个字母。下面的语句定义了一个struct entry类型的结构变量，并将其初始化为单词blob以及对应的解释：

```
struct entry word1 = { "blob", "an amorphous mass" };
```

因为我们的字典中要容纳很多单词，所以我们应该使用结构数组，如下所示：

```
struct entry dictionary[100];
```

上面的语句定义了一个能够容纳100个单词及其解释的结构数组。很明显，这个大小对于任何有价值的英文字典（至少需要10万个条目以上）来说，都是远远不足的。对于实际的字典来说，我们通常会把内容保存在计算机的磁盘上，而不是内存中。

在定义完字典词条之后，我们还需要考虑一下所有词条的组织问题。绝大多数字典的词条都是按照字母顺序排序的。这种做法对于我们的字典也是合适的。我们先暂定这种排序方式使得字典看上去比较容易阅读，随后我们还将看到按照字母顺序排序带来的实际好处。

接下来，我们可以开始编写实际的程序了。为了方便起见，我们定义一个名为lookup的函数，用于在字典中查找单词。如果找到了单词的解释，该函数返回这个单词在字典对应词条的下标号码，如果没有的话，该函数就返回-1。综上所述，在程序中对于lookup函数的调用形式一般如下：

```
entry = lookup (dictionary, word, entries);
```

在上面这个调用语句中，dictionary代表实际的字典，word代表需要查找的单词，entries代表字典中条目的个数。如果找到单词的话，变量entry中就包含该单词对应词条的下标，否则包含-1。

程序10.9使用了我们在程序10.4中编写的字符串函数equalString，用于判断单词是否与字典中的词条相匹配。

程序10.9 字典查找

```
// 字典查找的函数
#include <stdio.h>
#include <stdbool.h>

struct entry
{
    char word[15];
```


程序10.9 续

```
char definition[50];
};

/***** 在这里插入equalStrings函数 *****/

// 在字典中查找单词的函数
int lookup (const struct entry dictionary[], const char search[],
            const int entries)
{
    int i;
    bool equalStrings (const char s1[], const char s2[]);

    for ( i = 0; i < entries; ++i )
        if ( equalStrings (search, dictionary[i].word) )
            return i;

    return -1;
}

int main (void)
{
    const struct entry dictionary[100] = {
        { "aardvark", "a burrowing African mammal" },
        { "abyss", "a bottomless pit" },
        { "acumen", "mentally sharp; keen" },
        { "addle", "to become confused" },
        { "aerie", "a high nest" },
        { "affix", "to append; attach" },
        { "agar", "a jelly made from seaweed" },
        { "ahoy", "a nautical call of greeting" },
        { "aigrette", "an ornamental cluster of feathers" },
        { "ajar", "partially opened" } };

    char word[10];
    int entries = 10;
    int entry;
    int lookup (const struct entry dictionary[], const char search[],
                const int entries);

    printf ("Enter word: ");
    scanf ("%14s", word);
    entry = lookup (dictionary, word, entries);

    if ( entry != -1 )
        printf ("%s\n", dictionary[entry].definition);
}
```

程序10.9 续

```
else
    printf ("Sorry, the word %s is not in my dictionary.\n", word);

return 0;
}
```

程序10.9 输出

```
Enter word: agar
a jelly made from seaweed
```

程序10.9 输出（再次运行）

```
Enter word: accede
Sorry, the word accede is not in my dictionary.
```

程序中的lookup函数遍历字典的所有词条，对于每一个词条，我们调用equalString函数，判断用户给出的单词是否与该词条的单词相同。如果相同的话，我们将变量i的值作为返回值，这个值正好是该词条在字典中的下标。当找到单词之后，我们将立即执行return语句，虽然我们还在执行一个循环语句（这在C语言中是完全合法的）。

如果lookup函数遍历完了所有的词条，还是没有找到单词的话，那么该函数就返回-1，用于通知调用者，没有找到单词对应的解释。

一个更好的搜索算法

前面的lookup函数中使用的搜索算法非常简单，我们仅仅按照顺序遍历字典中的所有词条，直到找到一个匹配的词条，或者到达字典的结尾。对于一个较小的字典，这个搜索方法是完全合适的，但是对于那些有着成百上千个词条的字典，因为每查找一个单词，我们都要遍历所有的词条，因此就显得效率不高了。遍历这么大的结构数组需要相当多的时间（虽然在现代的计算机上，相当多可能只意味着几分之一秒）。对于任何信息查找类型的应用来讲，检索速度都是一个至关重要的指标，因为在应用程序中信息查找的使用频率非常高，因此人们花费了大量的时间和精力来寻找更加有效的搜索算法。（排序算法与搜索算法类似，也是人们关注的重点之一。）

因为我们的字典词条是按照字母顺序排列的，利用这一事实我们可以编写出更加高效的搜索算法。对于查找字典中不存在的单词，就有一个显而易见的优化方法。我们可以改写lookup函数，使之能够意识到，在剩下的尚未比较的条目中，肯定不存在需要查找的单词了。比如，如果我们正在程序10.9的字典中查找单词active，如果我们已经遍历到了单词acumen，我们就可以得出结论，字典中没有active这个单词，因为如果有这个单词的话，那么它对应的词条肯定出现在单词acumen之前。

虽然前面这个优化方法确实可以帮助我们的程序提高查找速度，但是这种方法只适用于查找不存在的单词这一类特殊情况。我们需要的是一种能够在绝大多数条件下都能够增加查找效率的算法，而不是只对某些特殊情况有用的算法。这样的算法确实存在，那就是二分查找法。

二分查找法的原理相当的简单。为了帮助读者理解该算法的原理，我们给出一个简单的猜数游戏作为例子。假定我（作者）从1到99之间随即的选取一个数，然后要求读者使用最少的次数猜中这个数。对于读者作出的每个猜测，我可以告诉你这个数是大于、小于还是等于实际的数。在试验过几次之后，读者可能会很快意识到，使用折半的方式可以更快的猜中实际的数。比如，你可以第一次选取50作为猜测的数，如果答案是这个数太大或者太小，你就可以将剩余的候选数字的数量从100减少到49个。如果答案是太大，我们就知道该数在51—99之间，否则的话，该数就在1—49之间。

我们可以针对剩下的49个数重复使用上面的方法。假定50太小，我们下来可以选取51到99中间的那个数字，也就是75作为下一次的猜测。这个过程可以一直重复下去，直到找到实际的数字为止。

上面的叙述实际上就是二分查找法的具体工作过程。下面我们使用正式的方式叙述该算法的步骤。使用该算法，我们可以在按照升序排列的有 n 个元素的数组 M 中，查找某个元素 x 。

二分搜索法

1. 设置变量 $low = 0$, $high = n - 1$ 。
2. 如果 $low > high$ ，那么 x 不在数组 M 中，算法结束。
3. 设置变量 $mid = (low + high) / 2$ 。
4. 如果 $M[mid] < x$ ，设置 $low = mid + 1$ ，回到步骤2。
5. 如果 $M[mid] > x$ ，设置 $high = mid - 1$ ，回到步骤2。
6. 如果 $M[mid]$ 等于 x ，那么我们已经找到该元素，算法结束。

注意，第三步中的除法是一个整数除法，也就是说，如果 low 等于0，而 $high$ 等于49，那么 mid 就等于24。

现在我们已经给出了二分查找法的详细算法步骤，我们可以使用这个算法重写程序10.9中的lookup函数。因为二分查找法要求知道当前元素是大于、等于还是小于需要查找的元素，因此我们还需要替换掉程序中的equalStrings函数，而使用一个能够进行上述判断的函数。我们将该函数名为compareStrings，该函数接受两个字符串参数，如果第一个从字母顺序上小于第二个，该函数返回-1，如果相等的话则返回0，如果大于的话返回1。因此，下面的函数调用语句：

```
compareStrings ("alpha", "altered")
```

的返回值是-1，也就是说从字母顺序上来说，单词"alpha"小于"altered"（读者可以想像，在字典中，单词alpha出现在单词altered之前）。同样，下面的调用语句：

```
compareStrings("zioty", "yucca");
```

的返回值是1，因为“zioty”的字母顺序大于“yucca”。

程序10.10中给出了函数compareStrings的实现。我们的lookup函数使用新的二分查找法。主函数与程序10.9相同。

程序10.10 使用二分查找法修正字典查找程序

```
// 字典查找程序
#include <stdio.h>

struct entry
{
    char word[15];
    char definition[50];
};

// 比较两个字符串的函数
int compareStrings (const char s1[], const char s2[])
{
    int i = 0, answer;

    while ( s1[i] == s2[i] && s1[i] != '\0' && s2[i] != '\0' )
        ++i;

    if ( s1[i] < s2[i] )
        answer = -1; /* s1 < s2 */
    else if ( s1[i] == s2[i] )
        answer = 0; /* s1 == s2 */
}
```


程序10.10 续

```
    else
        answer = 1; /* s1 > s2 */

    return answer;
}

// 在字典内查找单词的函数
int lookup (const struct entry dictionary[], const char search[],
            const int entries)
{
    int low = 0;
    int high = entries - 1;
    int mid, result;
    int compareStrings (const char s1[], const char s2[]);

    while ( low <= high )
    {
        mid = (low + high) / 2;
        result = compareStrings (dictionary[mid].word, search);

        if ( result == -1 )
            low = mid + 1;
        else if ( result == 1 )
            high = mid - 1;
        else
            return mid; /* found it */
    }
    return -1; /* not found */
}

int main (void)
{
    const struct entry dictionary[100] = {
        { "aardvark", "a burrowing African mammal" },
        { "abyss", "a bottomless pit" },
        { "acumen", "mentally sharp; keen" },
        { "addle", "to become confused" },
        { "aerie", "a high nest" },
        { "affix", "to append; attach" },
        { "agar", "a jelly made from seaweed" },
        { "ahoy", "a nautical call of greeting" },
        { "aigrette", "an ornamental cluster of feathers" },
        { "ajar", "partially opened" } };
}
```

程序10.10 续

```

int entries = 10;
char word[15];
int entry;
int lookup (const struct entry dictionary[], const char search[],
            const int entries);

printf ("Enter word: ");
scanf ("%14s", word);

entry = lookup (dictionary, word, entries);

if ( entry != -1 )
    printf ("%s\n", dictionary[entry].definition);
else
    printf ("Sorry, the word %s is not in my dictionary.\n", word);
return 0;
}
    
```

程序10.10 输出

```

Enter word: aigrette
an ornamental cluster of feathers
    
```

程序10.10 输出（再次运行）

```

Enter word: acerb
Sorry, that word is not in my dictionary.
    
```

直到while循环结束之前，compareStrings函数与equalStrings函数都是完全相同的。当while循环结束之后，compareStrings函数对于导致循环退出的结束字符进行分析。如果s1[i]小于s2[i]，那么字符串s1必然小于s2。在这种情况下，函数返回-1。如果s1[i]等于s2[i]，那么这两个字符串是相等的，函数返回0。如果上面这两种情况都不是的话，s1必定大于s2，这时我们返回1。

在lookup函数中，我们定义了两个变量low和high，并按照二分查找法的要求给它们赋予初值。只要变量low的值不超过high，那么循环就将一直进行下去。在循环中，我们计算low和high的和，然后用整除2的结果作为变量mid的值。下来，我们调用compareStrings函数，将字典词条dictionary[mid]中的单词与需要查找的单词进行比较，并将结果赋予变量result。

如果函数compareStrings的返回值为-1,也就是说dictionary[mid].word的值小于需要查找的单词,我们就将变量low的值设置为mid+1,如果函数的返回值是1,就说明dictionary[mid].word的值大于需要查找的单词,我们就将变量high的值设置为mid-1。如果返回值既不是1也不是-1,那么我们就已经找到了所需的词条,这时我们就将mid的值作为返回值传递给调用者。

在循环执行的过程中,如果变量low的值最终超过了high,这就说明需要查找的单词不在字典中,我们给调用者返回-1,用于指示这种情况。

字符运算

C语言经常在关系表达式和算术表达式中用到字符类型的变量或者常量。为了能够正确地运用字符类型,读者需要了解一下C语言编译器处理字符类型变量或者常量的方式。

无论什么时候,如果在表达式中用到了字符类型的常量或者变量,C语言编译器都会将其当作一个整型数值来处理。

在第6章“进行判断”中,我们曾经遇到过如下的表达式:

```
c >= 'a' && c <= 'z'
```

该表达式可以判断字符变量c的值是否是一个小写字母。当时我们作过说明,这个表达式在以ASCII码表示字符的计算机系统是正确的,因为在ASCII编码中,小写字母的值是从小到大连续排列的。在上面的表达式的第一部分中,实际上是将变量c中保存的值与小写字母a的ASCII编码的值进行比较。在ASCII编码中,'a'的值是97,'b'的值是98(其他小写字母的值依次类推)。因为ASCII编码的值大于97的字符不全是小写字母(比如大小括号),因此为了准确判断字符变量c是不是小写字母,还需要对于上限进行判断,所以上面的表达式的后半段将字符变量c的值(其值为122)与最后一个小写字母'z'进行比较。

因为在上面的表达式中,C语言编译器实际上是把字符变量c的值与小写字母'a'、'z'的ASCII值进行比较,所以下面的表达式实际上也可以用来判断字符变量c中包含的是否是小写字母。

```
c >= 97 && c <= 122
```

在实践中,我们更愿意使用第一个表达式,因为它不要求我们记住字母的具体ASCII值,而且意义看上去也更清楚一些。

下面的printf语句可以打印出计算机内部用于表示实际字符的编码值。

```
printf ("%i\n", c);
```

例如，如果读者的计算机系统使用ASCII编码，那么下面的语句的输出值将是97。

```
printf ("%i\n", 'a');
```

请读者试着预测一下，下面的语句的输出是什么。

```
c = 'a' + 1;  
printf ("%c\n", c);
```

因为'a'的ASCII编码值是97，所以上面的第一个语句实际上是把98赋值给变量c，随后，我们将该变量的内容作为一个字符类型输出，因为98正好是字母'b'的ASCII编码，所以实际上的输出将是字符b。

虽然给某个字符型变量的值加1看上去不是很实用，实际上我们可以使用这套技术在字符'0'到'9'和数值0到9之间方便地进行转换。读者可能还记得，字符'0'和数值0并不是同一个东西，字符'1'与数值1的情况也与之类似，实际上，按照ASCII编码，字符'0'的编码值是48，我们可以使用下面的输出语句来验证这一点：

```
printf ("%i\n", 0);
```

假定变量c中包含着'0'-'9'中的某个字符，我们需要将其转换为对应的实际数值0-9。因为所有的数值字符的ASCII编码是从小到大连续的，所以我们可以使用字符c的值减去字符'0'的值，从而获得对应的实际数值。因此，如果i是一个整型变量，那么下面的语句：

```
i = c - '0';
```

就会将字符c对应的数值保存在变量i之中。比如，假定变量c包含字符'5'，这个字符的ASCII编码值是53，所以上面的语句实际上计算53-48，并将结果5保存在变量i之中。即使对于一个非ASCII编码的计算机系统，上面的表达式结果也很可能是正确的，即使在该计算机的编码体系中字符'0'、'5'的具体值可能和ASCII编码有所不同。

上述技术也可以用来将一个代表某个数字的字符串用来转换为实际的数值。程序10.11中的函数strToInt即完成这项工作。该函数接受一个字符串类型的参数，然后逐个扫描字符串中的字符，直到遇到一个非数字的字符为止。该函数假定int类型的变量可以容纳最终转换所得的数值。

程序10.11 将字符串转换为对应整数

```
// 将字符串转换为对应整数的函数
#include <stdio.h>

int strToInt (const char string[])
{
    int i, intValue, result = 0;

    for ( i = 0; string[i] >= '0' && string[i] <= '9'; ++i )
    {
        intValue = string[i] - '0';
        result = result * 10 + intValue;
    }

    return result;
}

int main (void)
{
    int strToInt (const char string[]);

    printf ("%i\n", strToInt("245"));
    printf ("%i\n", strToInt("100") + 25);
    printf ("%i\n", strToInt("13x5"));

    return 0;
}
```

程序10.11 输出

```
245
125
13
```

只要string[i]代表一个数字字符，程序中的循环就将一直执行下去。每次执行循环体的时候，string[i]中包含的字符都被转换为实际对应的数字，然后该值再和变量result乘以10的结果相加。为了理解这个循环的工作原理，我们来考察一下当使用字符串"245"作为参数的时候，循环的执行过程。第一次执行循环的时候，变量intValue的值等于表达式string[0] - '0'，因为string[0]中的字符是'2'，因此intValue的值也就是2。在第一次执行循环的时候，变量result的值是0，其乘以10的结果还是0，我们将这个结果与intValue的值相加并重新存回变量result中，因此第一次循环执行完成之后，result变量中保存的数值就是2。

在第二次执行循环的时候，intValue的值是4，result乘以10的结果是20，最终循环执行完成之后，result变量中保存的值将是24。

第三次执行循环的时候，intValue的值等于'5' - '0'，也就是5，result中的数值乘以10的结果是240，最后相加后，result中保存的值将是245。

当遇到字符串结尾的空字符时，循环停止执行，并将变量result中的数值，也就是245返回给调用者。

函数strToInt还有两个改进的余地：1.这个函数不能处理负数；2.根据函数返回的结果，我们无法得知字符串参数是否包含任何有效的字符，比如，如果函数的返回值是0，我们不知道具体的调用是类似于strToInt("xxx")，还是strToInt("0")。这些问题都将作为练习留给读者。

我们对于字符串的讨论到此就将结束了。读者可以看到，C语言提供了很多功能，帮助我们高效而方便的对字符串进行处理。另外，C语言的标准库中还有很多用于处理字符串的函数，比如函数strlen用于返回字符串的长度，strcmp用于对两个字符串进行比较，函数strcat用于合并两个字符串，函数strcpy用于拷贝字符串，函数atoi用于将一个字符串转换为一个整数，而函数isupper、islower、isalpha与isdigit分别用于判断某个字符是否是一个大写字母、小写字母、字母或者数字。使用这些库函数重写本章中的某些程序对于读者来说是一个很好的练习。请读者参阅本书的附录B“C语言标准库”，以获得C语言标准库中用于处理字符串的函数的完整列表。

练习

1. 输入并运行本章的11个例子程序，并将结果与书中给出的结果进行比较。
2. 为什么我们可以使用如下的语句替换程序10.4中equalString函数的while语句，并得到相同的结果？

```
while ( s1[i] == s2[i] && s1[i] != '\0' )
```

3. 程序10.7和10.8中的countWords函数有一个错误，该函数将包含撇号（'）的单词作为两个单词处理。请修正这个错误。另外，请对这个程序进行改进，使之能够处理以正负号开头的、并包含逗号的数字，并将其作为一个单词处理。
4. 编写一个名为substring的函数，用于从某个字符串中裁减出特定的部分。该函数的调用形式如下：

```
substring (source, start, count, result);
```


在上面的调用中，source代表需要裁减一部分的源字符串，start代表需要裁减出的部分的起始位置下标，count代表要裁减出的字符串的个数，从源字符串中裁减出来的字符串存放在字符数组result中。例如，下面的语句：

```
substring ("character", 4, 3, result);
```

将字符串"character"的子串"act"（也就是从下标4开始的三个字符）存放在字符数组result中。

读者需要注意，裁减完成之后，需要在result数组结尾放置必要的结束空字符。另外，substring函数还要检查参数的有效性，也就是说，被裁减的字符串是否包含需要的字符个数，如果没有的话，裁减工作到达源字符串的结尾就结束。因此，如果执行下面的调用：

```
substring ("two words", 4, 20, result);
```

result数组中将只包含字符串"words"，虽然调用语句中指定需要裁减20个字符。

5. 编写一个名为findString的函数，用于检查一个字符串是否包含另外一个字符串。函数的第一个参数是需要被搜索的字符串，第二个参数是需要搜索的字符串。如果在第一个字符串参数中找到了第二个字符串参数，函数将第二个字符串在第一个字符串中的起始位置作为返回值返回给调用者，如果没有的话，则返回-1。因此，下面的语句

```
index = findString ("a chatterbox", "hat");
```

在字符串"a chatterbox"中查找字符串"hat"。因为第一个字符串中确实包含第二个字符串，因此该函数返回数值3，3就是字符串"hat"在第一个字符串中的起始位置。

6. 编写一个名为removeString的函数，该函数用于从一个字符串中删除一定数量的字符。该函数接受三个参数：第一个参数代表源字符串，第二个参数代表需要删除字符的起始位置，第三个参数代表需要删除的字符个数。因此如果字符数组text包含的字符串"the wrong son"，下面的调用语句：

```
removeString (text, 4, 6);
```

则会删除该字符数组中的字符串wrong以及后面的空格。字符数组text中的遗留内容则是字符串"the son"。

7. 编写一个名为insertString的函数，用于将某个字符串插入到另外一个字符串中。该函数应该接受如下几个参数：需要插入字符串的源字符串，将要被插入的字符串，源字符串的插入位置。因此，对于下面的调用语句：

```
insertString (text, "per", 10);
```

如果text数组的内容如我们前面一个练习题的内容所示的那样，那么最终text数组中保存的字符串内容将是"the wrong person"。

8. 请读者将我们前面编写的findString、removeString和insertString这三个函数组合起来，编写一个名为replaceString的函数，该函数接受三个参数，其调用形式如下：

```
replaceString (source, s1, s2);
```

函数的执行结果是将source字符串中的s1用s2替换。replaceString应该首先调用findString函数找到需要替换的部分，然后调用removeString将这部分字符串删除，最后再调用insertString在合适的位置插入需要替换的部分。

因此，对于下面的调用语句：

```
replaceString (text, "1", "one");
```

如果字符串text中包含字符串"1"的话，函数将第一个出现的"1"用字符串"one"替换掉。与之类似，下面的调用语句：

```
replaceString (text, "*", "");
```

则是将字符串text中第一个出现的"*"删除掉（因为替换的字符串是一个空字符串）。

9. 我们还可以进一步扩展上面的replaceString函数，使之变得更为有用。也可以让该函数返回一个布尔值，用于确定我们是否在源字符串中找到了需要替换的字符串。如果找到，返回值为true，否则返回值为false。因此，我们可以使用下面的循环语句：

```
do
    stillFound = replaceString (text, " ", "");
while ( stillFound = true );
```

将字符串text中所有的空白字符全部删除。

请按照上述要求改写第8题中编写的replaceString函数，并使用不同的参数字符串验证读者编写的函数是否能够正常地工作。

10. 编写一个名为dictionarySort的函数，用于对程序10.9和10.10中定义的字典进行排序。
11. 扩展程序10.11中的strToInt函数，如果字符串的第一个字符是一个减号，则将最终转换的结果变为一个负数。

12. 编写一个名为strToFloat的函数，用于将某个字符串转化为对应的浮点数。函数应该能够处理字符串的第一个字符是负号的情况。也就是说，对于下面的函数调用：

```
strToFloat ("-867.6921");
```

最终的转换结果应该是-867.6921。

13. 在使用ASCII编码的计算机系统上，如果字符变量c包含一个小写字母，那么下面的语句将其转换为对应的大写字母。

```
c = 'a' + 'A'
```

编写一个名为uppercase的函数，用于将某个字符串中所有的小写字母转换为大写字母。

14. 编写一个名为intToStr的函数，用于将一个整型数转换为对应的字符串。函数要求能够处理负数。

CHAPTER 11 Pointers

指针

本本章我们来学习指针。指针是C语言最复杂的一个特性。实际上，正是因为强大而灵活的指针功能，使得C语言与其它许多语言区分开来。运用指针，我们可以有效地描述复杂的数据结构；可以改变传递给函数的参数的值；可以“动态”分配内存（见17章，“杂项及高级特性”）；可以更加简洁、有效地处理数组。

为了理解指针操作的方式，我们首先有必要理解一下“间接”的概念。日常生活中，我们对此概念已很熟悉了。例如，假定我们需要给打印机需要购买一个新的喷墨打印头。在公司，所有的购买事宜都是由采购部办理。因此，我们吩咐采购部的jim，让他给我们订购一个新打印头。我们获得新打印头的这种方法，就是间接的，因为，打印头并不是我们自己直接从供应商那里获得的。

在C语言里，间接的概念同样地适用于指针的工作方式，它提供了一个间接的方法来存取在一个特定数据项中的数值。正如我们需要到采购部来订购新的打印头（我们并不需要知道打印头是在那个具体的商店里订购的）那样，在C语言里，我们也同样需要运用指针来解决一些问题。

定义指针变量

说了这么多，现在我们来看一下指针实际上是如何工作的。假定我们定义了如下的一个叫做count的变量：

```
int count = 10;
```


我们可以按照下面的方式定义另外一个变量 `int_pointer`，可以使用它以间接的方式来存取 `count` 的值：

```
int *int_pointer;
```

在C语言中，以上语句中的 `*` 代表 `int_pointer` 是一个整型指针变量。这样，程序就可以用 `int_pointer` 来间接地存取一个或多个整型数值。

我们在以前程序中已经见过函数调用 `scanf` 中的 `&` 操作符，这个一元操作符称为地址运算符，在C语言中用于生成指向某个目标的指针。因此，若 `x` 是一个特定类型的变量，则表达式 `&x` 是这个变量的指针。表达式 `&x` 可以被赋予任何一个指针变量，只要它被声明为和 `x` 的类型相同的指针。

因此，仿照已给出的 `count` 和 `int_pointer`，我们可以写出如下语句：

```
int_pointer = &count;
```

建立 `int_pointer` 和 `count` 之间的间接引用关系。上面的表达式将一个指向变量 `count` 的指针，而不是变量 `count` 的值赋予变量 `int_pointer`。图 11.1 说明了 `int_pointer` 和 `count` 之间的关系。图中的直连线说明 `int_pointer` 不是直接包含了 `count` 的值，而是一个指向变量 `count` 的指针。

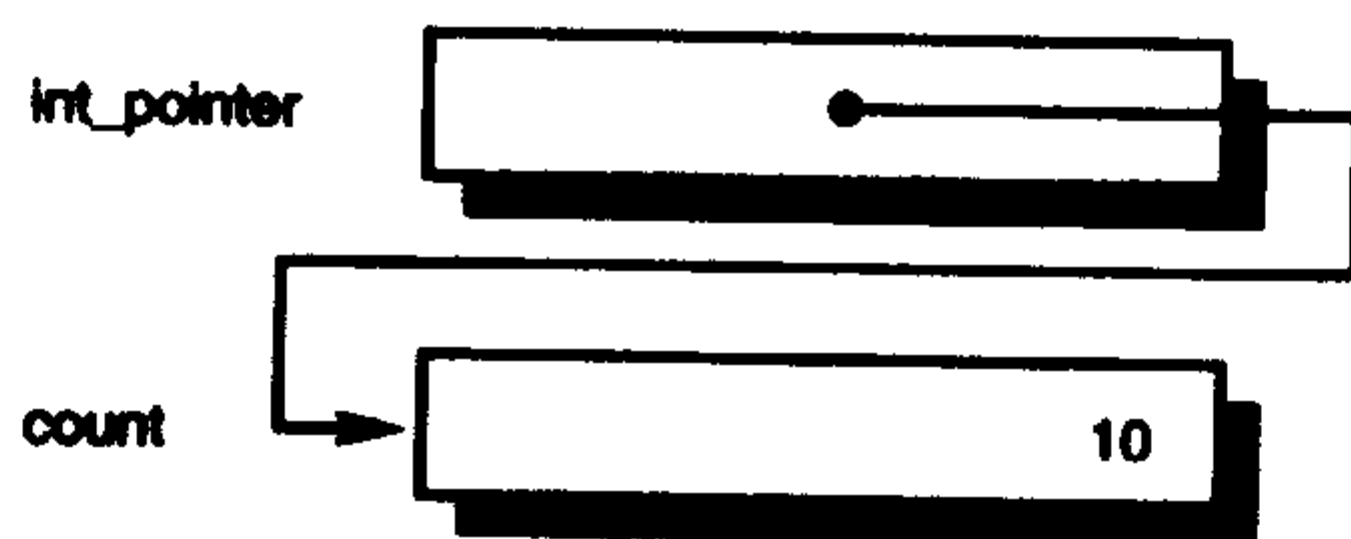


图 11.1 指向整型的指针

为了获取指针变量 `int_pointer` 指向的变量 `count` 所包含的内容，我们可以用指针运算符，即星号 `*`。因此，如果我们定义 `x` 为整型变量，则下面语句：

```
x = *int_pointer;
```

将 `int_pointer` 通过间接关系所指向的值赋予变量 `x`。因为 `int_pointer` 先前被设为指向 `count`，所以这个语句将把包含在变量 `count` 的值 10，赋予变量 `x`。

程序 11.1 包含了前面的语句，并说明了两个基本的指针操作符：地址运算符，`&` 和指针运算符，`*`。

程序11.1 举例说明指针

```
// 说明指针的程序

#include <stdio.h>

int main (void)
{
    int count = 10, x;
    int *int_pointer;

    int_pointer = &count;
    x = *int_pointer;

    printf ("count = %i, x = %i\n", count, x);

    return 0;
}
```

程序11.1 输出

```
count = 10, x = 10
```

在程序中，我们用正常方式定义了整型变量`count`和`x`。接下来一行，我们定义`int_pointer`为指向整型的指针类型变量。注意，这两行声明可以合并为一行：

```
int count = 10, x, *int_pointer;
```

接下来，我们将地址运算符应用于变量`count`，这样就生成了一个指向了这个变量的指针，然后我们将其赋予变量`int_pointer`。

程序接着执行下面语句：

```
x = *int_pointer;
```

指针运算符告诉C语言系统，将变量`int_pointer`看作包含一个指向其它数据项的指针，其类型在指针声明中被指定。因为我们在声明时告诉编译器`int_pointer`指向整型数，所以编译器知道表达式`*int_pointer`的值是一个整数。并且，因为我们在程序前面的语句中设置`int_pointer`指向整型变量`count`，所以，通过这个表达式可以间接地存取变量`count`的值。

读者应该意识到，程序11.1只是用于说明指针应用的例子程序，并不是一个实际的应用程序，当读者对程序中指针的定义和操作的基本方法逐渐熟悉以后，我们随后将会给出更为实际一些的例子。

程序11.2举例说明了一些关于指针变量的有趣应用。该程序用到了一个指向字符的指针。

程序11.2 进一步的指针基础应用

```
// 指针进一步应用的程序

#include <stdio.h>

int main (void)
{
    char c = 'Q';
    char *char_pointer = &c;

    printf ("%c %c\n", c, *char_pointer);

    c = '/';
    printf ("%c %c\n", c, *char_pointer);

    *char_pointer = '(';
    printf ("%c %c\n", c, *char_pointer);

    return 0;
}
```

程序11.2 输出

```
Q Q
/ /
( (
```

程序定义了字符变量c，并初始化其值为'Q'。在第二行，我们定义变量char_pointer为字符指针类型，这就意味着，不管存储在这个变量里的是何值，都应该间接指向一个字符。注意，我们可以采取通常方法给这个变量赋予一个初值。程序中我们使用地址操作符获取指向字符变量c的一个指针，然后将其赋予变量char_pointer。（注意：如果我们没有定义变量c的话，那么这个初始化表达式将产生一个编译错误，因为一个变量必须事先声明，然后才能在一个表达式中引用。）

变量char_pointer的声明和它的初始化可以等价地用两个独立的语句来完成：

```
char *char_pointer;
char_pointer = &c;
```


(注意, 下面的语句是不正确的, 虽然它可能看上去和单行形式的声明看上去比较像:

```
char *char_pointer;  
*char_pointer = &c;
```

)。

我们要时刻牢记, 除非用它来指向某个东西, 否则C语言指针的值是没有意义的。

第一个printf调用显示了变量c的内容以及用 char_pointer指向的变量的内容。因为我们将char_pointer设置为指向变量c的指针, 因此, 显示的值就是c的内容, 这一点在程序的第一行的输出里得到了验证。

在程序的第二行, 我们将字符'/'赋予变量c。因为char_pointer仍指向变量c, 所以随后在我们调用printf函数显示*char_pointer时, 终端上正确地显示了c的新值。这一概念很重要。除非我们改变char_pointer的值, 否则表达式*char_pointer将总是存取变量c。因此, 如果c值改变了, 则*char_pointer的值也就会改变。

前面的讨论可以帮助读者理解程序中接下来的语句是怎样执行的。除非是改变了char_pointer, 否则*char_pointer将总是引用变量c。因此, 表达式:

```
*char_pointer = '(';
```

实际上是将左圆括号字符赋予c。用更正式的说法来讲, 字符'('被赋予用char_pointer指向的变量。我们知道这个变量就是c, 因为在程序的开头, 指针变量char_pointer就是如此声明的。

前面的概念对于我们理解指针的操作很关键。若读者还不是太清楚, 请再复习一遍。

在表达式中运用指针

在程序11.3中, 我们定义了两个整型指针p1和p2。读者请注意指针引用的值是如何运用于算术表达式的。如果定义p1为整型指针, 那么在表达式中运用*p1时会得到什么结果呢?

程序11.3 在表达式中运用指针

```
// 更多指针应用  
#include <stdio.h>  
int main (void)  
{
```

程序11.3 续

```
int i1, i2;
int *p1, *p2;

i1 = 5;
p1 = &i1;
i2 = *p1 / 2 + 10;
p2 = p1;

printf ("i1 = %i, i2 = %i, *p1 = %i, *p2 = %i\n", i1, i2, *p1, *p2);

return 0;
}
```

程序 11.3 输出

```
i1 = 5, i2 = 12, *p1 = 5, *p2 = 5
```

在定义了整型变量*i1*、*i2*和整型指针变量*p1*、*p2*后，程序将*i1*赋值为5，并将*p1*指向*i1*。接下来，程序用下面的表达式计算*i2*的值：

```
i2 = *p1 / 2 + 10;
```

像在程序11.2中讨论的那样，如果一个指针*px*指向一个变量*x*，且*px*被定义为和*x*相同类型的指针，那么在表达式中，**px*的用法等同于*x*。

程序11.3中，因为我们定义变量*p1*为一个整型指针，则前面的表达式的求值使用整数运算法则。因为**p1*的值为5（*p1*指向*i1*），所以表达式的最终运算结果是12，这个值被赋值给*i2*（指针运算符 *** 比算术运算符的除法具有更高的优先级，事实上，和地址运算符一样，在C语言中，这两者的优先级比其它所有二元运算符都高）。

接下来的语句将指针*p1*赋值给*p2*，这个赋值在C语言中完全有效的，它的作用是将*p2*指向*p1*所指的数据项。因为*p1*指向*i1*，因此上面赋值语句执行后，则*p2*也指向*i1*（C语言中，如果需要，我们可以使许多指针指向同一数据项）。

`printf`函数调用验证了*i1*，**p1*和**p2*具有相同的值（5），而*i2*则被程序设置为12。

使用指针和结构

我们已经知道了如何定义指向一个基本数据类型（如整型或字符）的指针，但是，指针同样也可以指向一个结构。在第9章“使用结构”中，我们定义了如下的数据结构*date*：


```
struct date
{
    int month;
    int day;
    int year;
};
```

正如在下面语句中我们定义一个结构变量一样

```
Struct date todaysDate;
```

我们也可以如下的方式定义一个指向结构的指针：

```
Struct date *datePtr;
```

我们可以将上面刚定义的变量datePtr当作正常的指针使用。例如，我们可以用赋值语句让它指向todaysDate：

```
datePtr = &todaysDate;
```

如此赋值之后，我们便可间接地存取由datePtr所指向的date结构中的任何一个成员，如下所示：

```
(*datePtr).day = 21;
```

上述语句将datePtr所指向的date结构的day成员赋值为21。因为结构成员操作符.与指针运算符*相比有更高的优先级，所以我们需要用到小括号。

类似的，如果需要检验由datePtr所指向的date结构的month成员的值，我们可以用下述语句：

```
if ((*datePtr).month == 12)
    . . .
```

由于我们经常要用到指向结构的指针，因此在C语言中专门有一个特殊的操作符——结构指针运算符 ->，它由一个破折号后紧跟一个大于号组成，使用该运算符，我们可以将表达式(*x).y更清晰地表示为 x->y。

因此，前面的if语句可更方便地表示如下：

```
if ( datePtr->month == 12)
    . . .
```

我们下面使用结构指针的概念重写程序9.1，如程序11.4所示：

程序11.4 指向结构的指针

```
// 结构指针应用程序

#include <stdio.h>

int main (void)
{
    struct date
    {
        int month;
        int day;
        int year;
    };

    struct date today, *datePtr;

    datePtr = &today;

    datePtr->month = 9;
    datePtr->day = 25;
    datePtr->year = 2004;

    printf ("Today's date is %i/%i/%.2i.\n",
           datePtr->month, datePtr->day, datePtr->year % 100);

    return 0;
}
```

程序 11.4 输出

Today's date is 9/25/04.

图11.2显示了程序中所有的赋值语句执行完毕之后，变量today和指针datePtr中保存的内容。

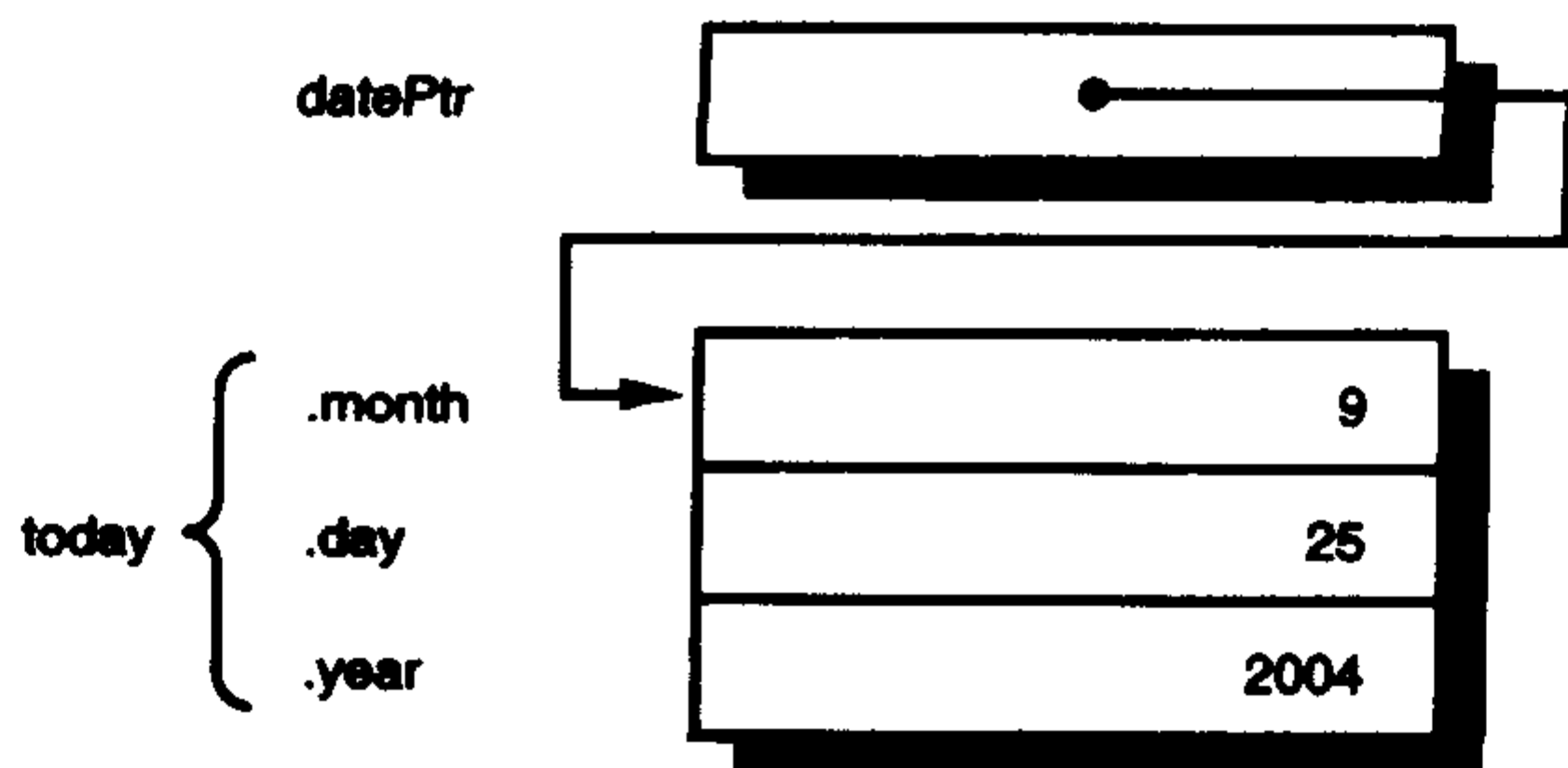


图 11.2 指向结构的指针

应该再一次指出的是，看起来我们似乎不用结构指针就可以处理得很好（像我们在程序9.1中所做的那样），为什么我们还应该甚至是不厌其烦地要运用结构指针呢？这里没有提到真正的原因，但我们不久就会发现为什么要这样做了。

包含指针的结构

自然而然，指针也可以成为结构的成员。在下面结构定义中

```
struct intPtrs
{
    int *p1;
    int *p2;
};
```

定义了一个结构intPtrs，它包含两个整型指针，第一个叫做p1，第二个叫做p2。我们可以像通常那样定义一个结构类型的变量intPtrs：

```
struct intPtrs pointers;
```

这样，变量pointers就可以以正常的方式使用了。读者需要记住变量pointers本身不是一个指针，但是却有2个作为其成员的指针。

程序11.5说明了C语言中是如何处理结构intPtrs的。

程序11.5 使用包含指针的结构

```
// 使用包含指针的结构的程序

#include <stdio.h>

int main (void)
{
    struct intPtrs
    {
        int *p1;
        int *p2;
    };

    struct intPtrs pointers;
    int i1 = 100, i2;

    pointers.p1 = &i1;
    pointers.p2 = &i2;
    *pointers.p2 = -97;
```

程序11.5

```
printf ("i1 = %i, *pointers.p1 = %i\n", i1, *pointers.p1);
printf ("i2 = %i, *pointers.p2 = %i\n", i2, *pointers.p2);
return 0;
}
```

程序11.5 输出:

```
i1 = 100, *pointers.p1 = 100
i2 = -97, *pointers.p2 = -97
```

在定义了变量之后, 下面的赋值语句:

```
pointers.p1 = &i1;
```

将pointers的成员p1指向整型变量i1, 而下一条语句

```
pointers.p2 = &i2;
```

则将pointers的成员p2指向整型变量i2。接下来, 程序将pointers.p2所指向的变量赋值为-97。因为我们刚设置了pointers的成员p2指向i2, 所以, -97被存储到变量i2中。像我们前面提到的那样, 因为结构运算符比指针运算符的优先级高, 所以这里并不需要括号。在指针运算符起作用之前, C语言会首先从结构中找出指针。当然了, 运用小括号可以更加安全一些, 因为有时要记得两个运算符的优先级顺序是不容易的。

接下来, 程序使用两个printf调用来验证赋值语句是否正确执行。

图11.3可以帮助我们理解程序11.5中赋值语句执行后, 变量i1、i2和pointers之间的关系。像我们在图11.3中所看到的那样, 成员p1指向变量i1, 它所含的值是100, 而成员p2指向变量i2, 它所含的值是-97。

链表

C语言中, 指向结构的指针和包含指针的结构这两个概念是非常重要的, 我们可以使用它们来建立复杂的数据结构, 像链表、双向链表和树等。

假定我们定义了一个如下的结构:

```
struct entry
{
    int value;
    struct entry *next;
};
```

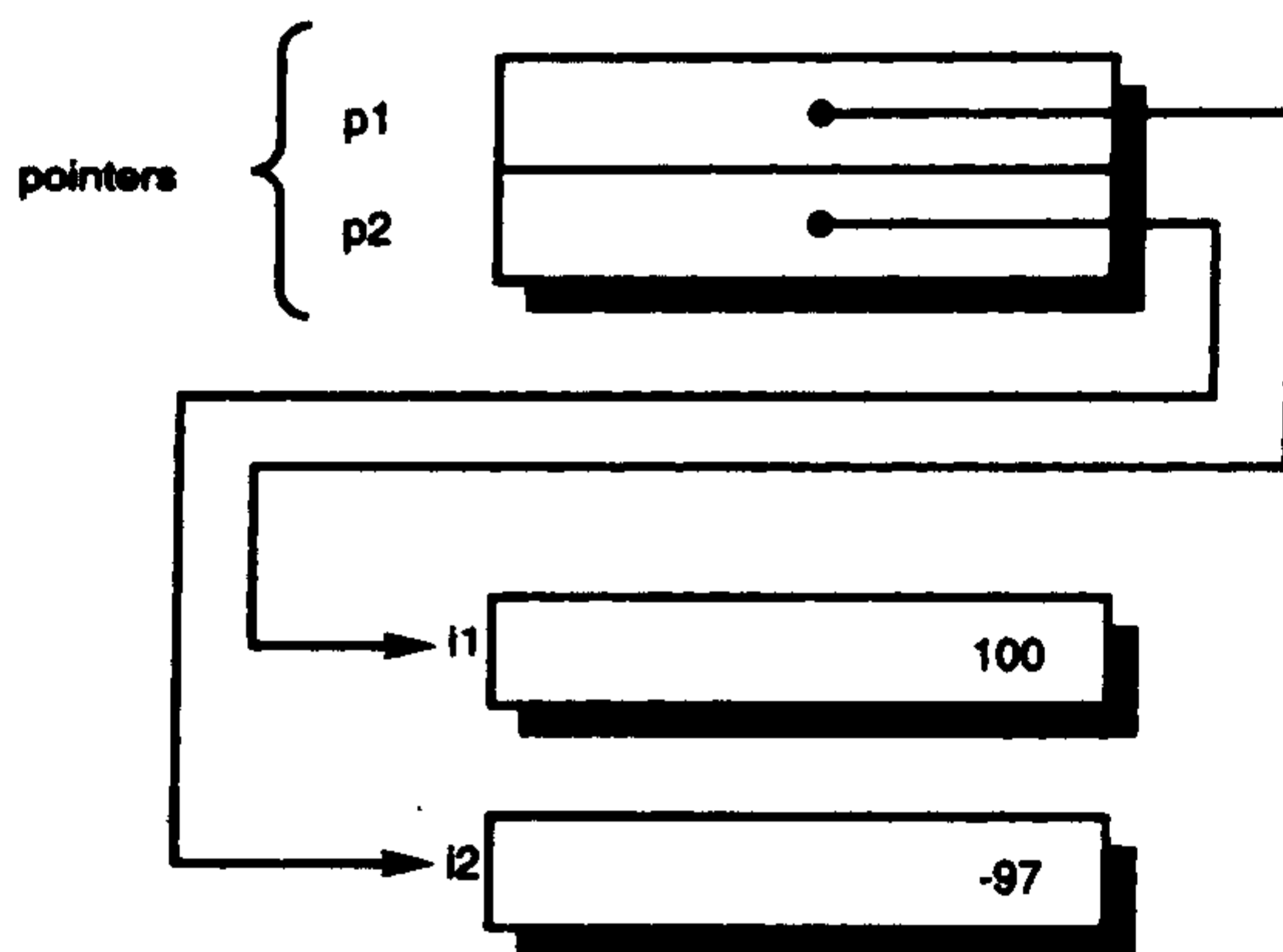



图 11.3 包含指针的结构

定义了一个叫做entry的结构，它包含两个成员：第一个成员叫做value，是一个简单的整数，第二个成员叫做next，是一个指向结构entry的指针。这里请读者仔细思考一下这个结构：一个entry结构的内部包含一个指向另一个entry结构的指针，在C语言中这是完全合法的。现在，假定我们定义两个如下的entry结构变量：

```
struct entry n1, n2;
```

我们可以用下面的语句将结构n1的next指针指向结构n2：

```
n1.next = &n2;
```

像图11.4所描述的那样，上述语句有效地建立了一个n1和n2之间的链接。

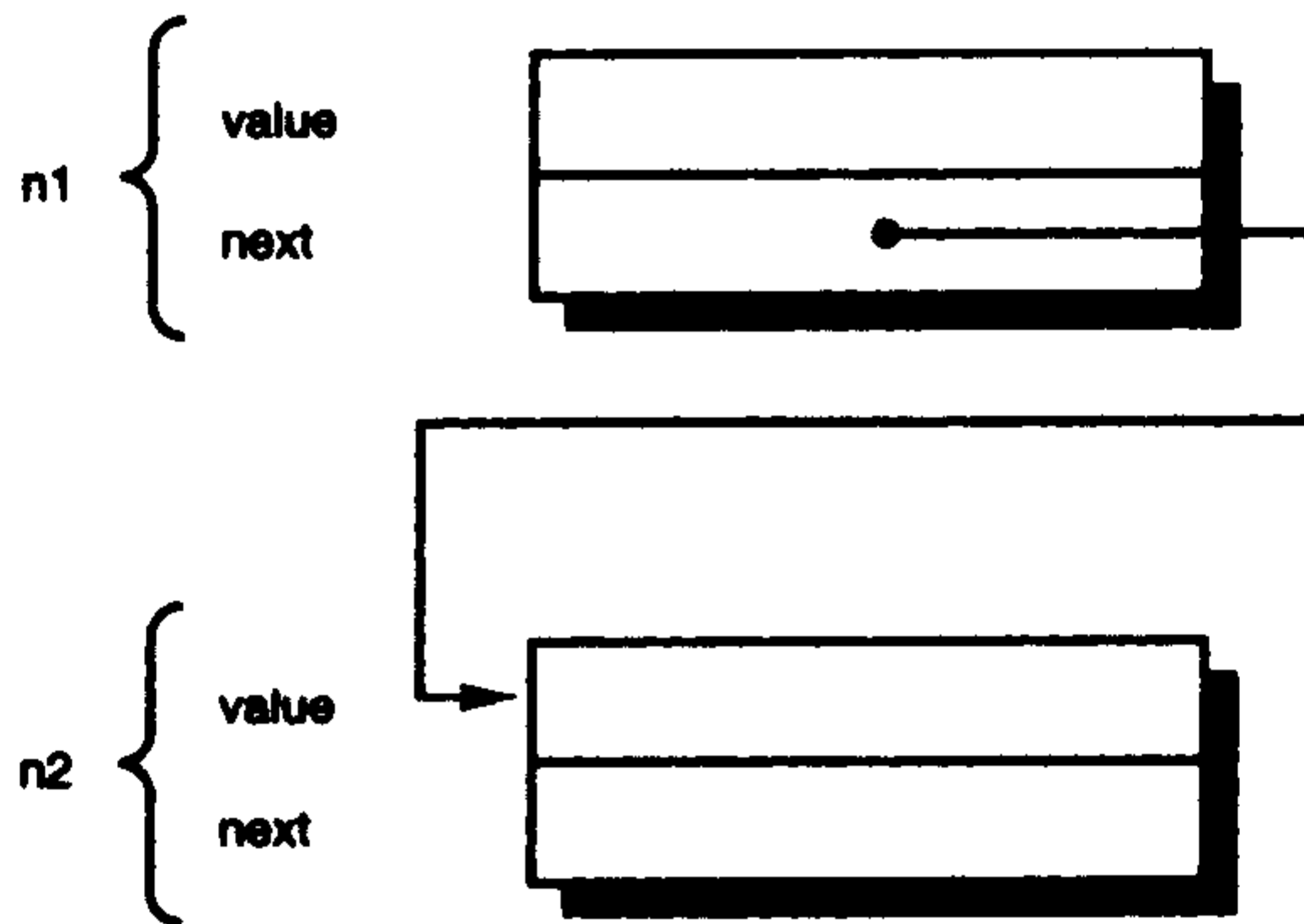


图 11.4 链接的结构

假定变量n3也是一个entry结构类型变量，我们可以用下面语句再增加一个链接：

```
n2.next = &n3;
```

这样的结果形成一个entry链，更正式地说是一个链表，我们在图11.5和程序11.6中画出了它的结构。

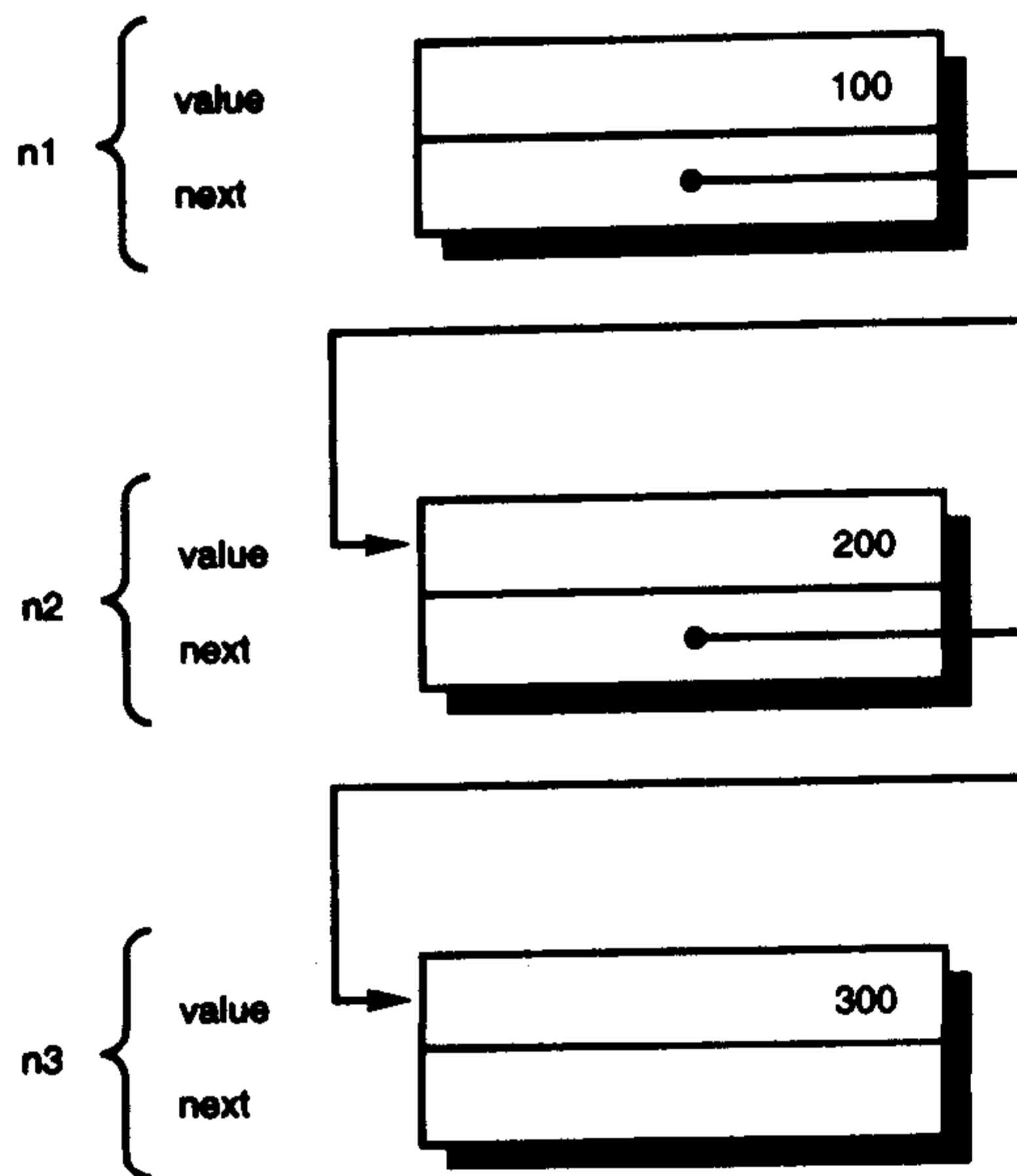


图 11.5 链表

程序11.6 应用链表

```

// 应用链表的函数

#include <stdio.h>

int main (void)
{
    struct entry
    {
        int value;
        struct entry *next;
    };

    struct entry n1, n2, n3;
    int i;

```

程序11.6 续

```

    n1.value = 100;
    n2.value = 200;
    n3.value = 300;
    n1.next = &n2;
    n2.next = &n3;
    i = n1.next->value;
    printf ("%i ", i);
    printf ("%i\n", n2.next->value);
    return 0;
}

```

程序11.6 Output

200 300

变量n1、n2和n3被定义为entry类型的结构，该结构包含一个整型成员value和一个指向另一个entry结构的指针成员next。接下来，程序分别将n1、n2和n3的value成员赋值为100、200和300。

程序的下面两行语句：

```
n1.next = &n2;  
n2.next = &n3;
```

用n1.next指向n2，用n2.next指向n3，因而建立起了一个链表。

下面的语句将n1.next指向的entry结构的value成员赋值给整型变量i。

```
i = n1.next->value;
```

因为我们将n1.next指向n2，所以，用这个语句可以存取n2的成员value。实际上，这个语句将200赋值给i，程序里用printf调用对此进行了验证。因为n1.next域包含的是一个指向结构的指针而并非一个结构本身，所以表达式n1.next->value是正确的而n1.next.value是错误的。读者必须仔细体会这个差别，如果不完全理解它，那么就很容易导致程序出错。

C语言中，结构成员运算符.和结构指针运算符->具有相同的优先级。在前面的表达式中，我们用到了这两个运算符，那么编译器就按由左向右的顺序求值。因此，这个表达式

实际上是按照下面的顺序求值的：

```
i = (n1.next)->value;
```

这正是我们所需要的。

程序11.6的第二个printf调用显示了由n2.next所指向的数据项的value成员。因为我们将n2.next指向n3，所以程序显示了n3.value的内容。

链表是程序设计中一个很重要的概念。通过使用链表，我们可以大大简化对于一个存储了很多项的排序链表中进行插入和删除操作。

例如，如果n1、n2和n3如上定义，则我们可以简单地通过将n1的next域指向n2的next域所指向的数据项，从而将n2从表中删除。

```
n1.next = n2.next;
```

上面语句将n2.next所包含的指针复制到n1.next。因为n2.next指向n3，所以，现在n1.next也指向了n3。

进而，因为n1不再指向n2，我们也就将它从表中删除了。图11.6描述了前面语句执行后的情况。当然，我们也可用如下语句直接将n1指向n3：

```
n1.next = &n3;
```

但这一语句不通用，因为我们必须事先要知道n2指向的是n3。

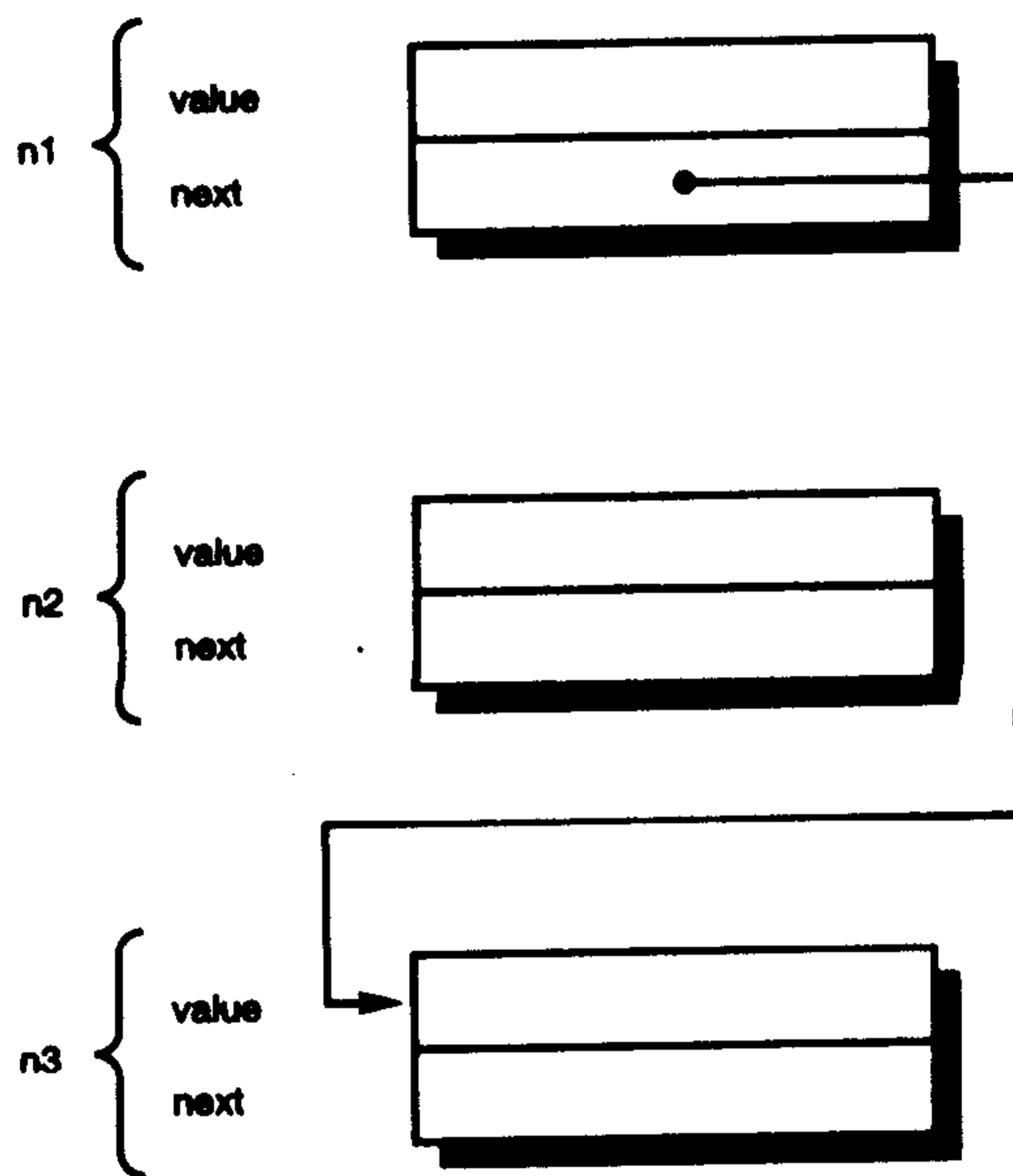


图 11.6 从链表中删除一个单元

在链表中插入一个元素也是很简单的。如果我们想要在链表中 n_2 的后面插入一个叫做 n_{2_3} 的entry结构，则我们只要简单地将 $n_{2_3}.next$ 指向 $n_2.next$ 所指向的数据项，再将 $n_2.next$ 指向 n_{2_3} 即可。因此，下面的语句将 n_{2_3} 插入 n_2 后面：

```
n2_3.next = n2.next;
n2.next = &n2_3;
```

注意，前面语句的顺序是很重要的。若是先执行了第二条语句，将会覆盖存储在 $n_2.next$ 中的指针，这样就没有机会将 $n_2.next$ 赋值给 $n_{2_3}.next$ 。图11.7显示了插入的数据项 n_{2_3} 。注意， n_{2_3} 并没有显示在 n_2 与 n_3 之间，我们通过这个图向读者强调： n_{2_3} 可以在内存中的任何地方，而不必在物理上存储在 n_1 之后和 n_3 之前。这是我们用链表的方式来存储信息的一个主要目的：表中的元素不必在内存中顺序存放，但数组中的元素就不能够这样。

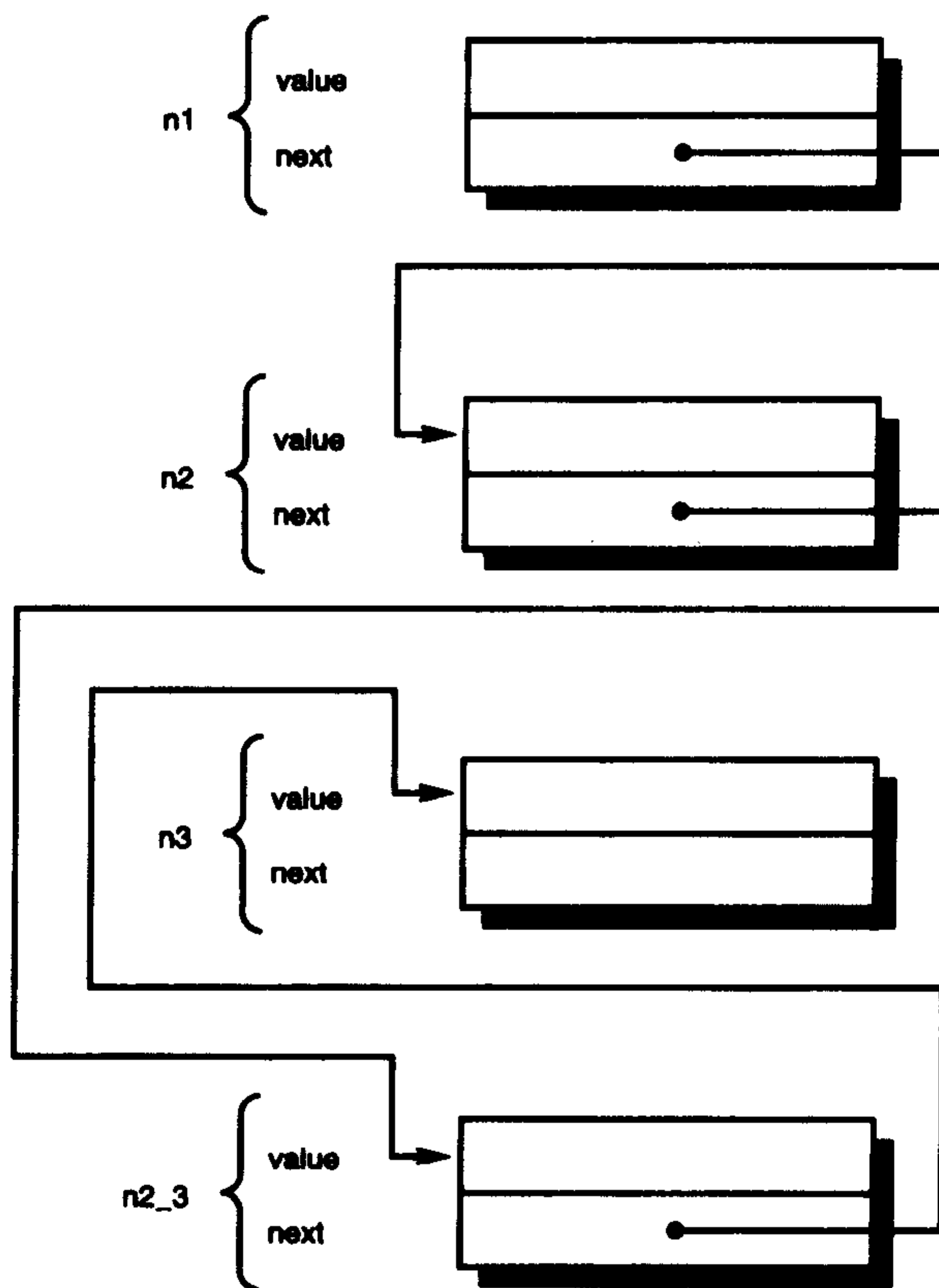


图 11.7 在链表中插入一个元素

在编制一些处理链表的函数之前，我们还必须讨论两个问题。通常，至少要有一个指针指向链表的开头。因此，在我们最初的三个元素的表中，用什么来组合n1、n2和n3呢，我们可以定义一个叫做list_pointer的变量，用以下语句将它指向链表的开头，如下面的语句所示：

```
struct entry *list_pointer = &n1;
```

像我们不久将要看到的那样，一个指向表头的指针对于按顺序遍历表中的元素是非常有用的。

第二个需要讨论的问题是我们如何来识别表尾。识别表尾是很有必要的，只有这样，一个查找表的过程才能知道何时抵达表的最后一个元素。按照惯例，我们通常用一个值为0的常量，即常说的“NULL”指针来完成这个任务。我们可以存储一个空指针到表的最后一个元素的指针域来标识表尾。¹

在这个拥有三个元素的链表中，我们可以给n3.next存储一个空指针以标识它是链表的末尾：

```
n3.next = (struct entry *) 0;
```

在第13章“预处理”中，我们将会介绍如何使这个赋值语句可以变得更易读一些。

类型转换运算符用来将常量0转换为合适的类型（比如指向结构的指针），这不是必需的，但这样做可以使得语句更加可读。

图11.8描述了从程序11.6所得到的链表，图中用一个叫做list_pointer的entry结构指针指向表的开头，并将空指针设置在n3.next域中。

程序11.7综合了刚才所讲述的概念。程序用一个while循环来遍历链表并显示表中每一个元素的成员value。

程序11.7 遍历一个链表

```
// 遍历链表的程序

#include <stdio.h>

int main (void)
{
    struct entry
    {
        int value;
        struct entry *next;
    };
}
```

¹ 编译器并不一定要将空指针在内部表示为数值0，然而，在将0值赋给某个指针变量的时候，编译器必须能够识别出这实际上是将指针设置为空指针。同样的道理也适用于比较一个指针和数值0：编译器应该将它作为检查某个指针是否为空。

程序11.7 续

```
struct entry n1, n2, n3;  
struct entry *list_pointer = &n1;  
  
n1.value = 100;  
n1.next = &n2;  
  
n2.value = 200;  
n2.next = &n3;  
  
n3.value = 300;  
n3.next = (struct entry *) 0; // Mark list end with null pointer  
  
while ( list_pointer != (struct entry *) 0 ) {  
    printf ("%i\n", list_pointer->value);  
    list_pointer = list_pointer->next;  
}  
  
return 0;  
}
```

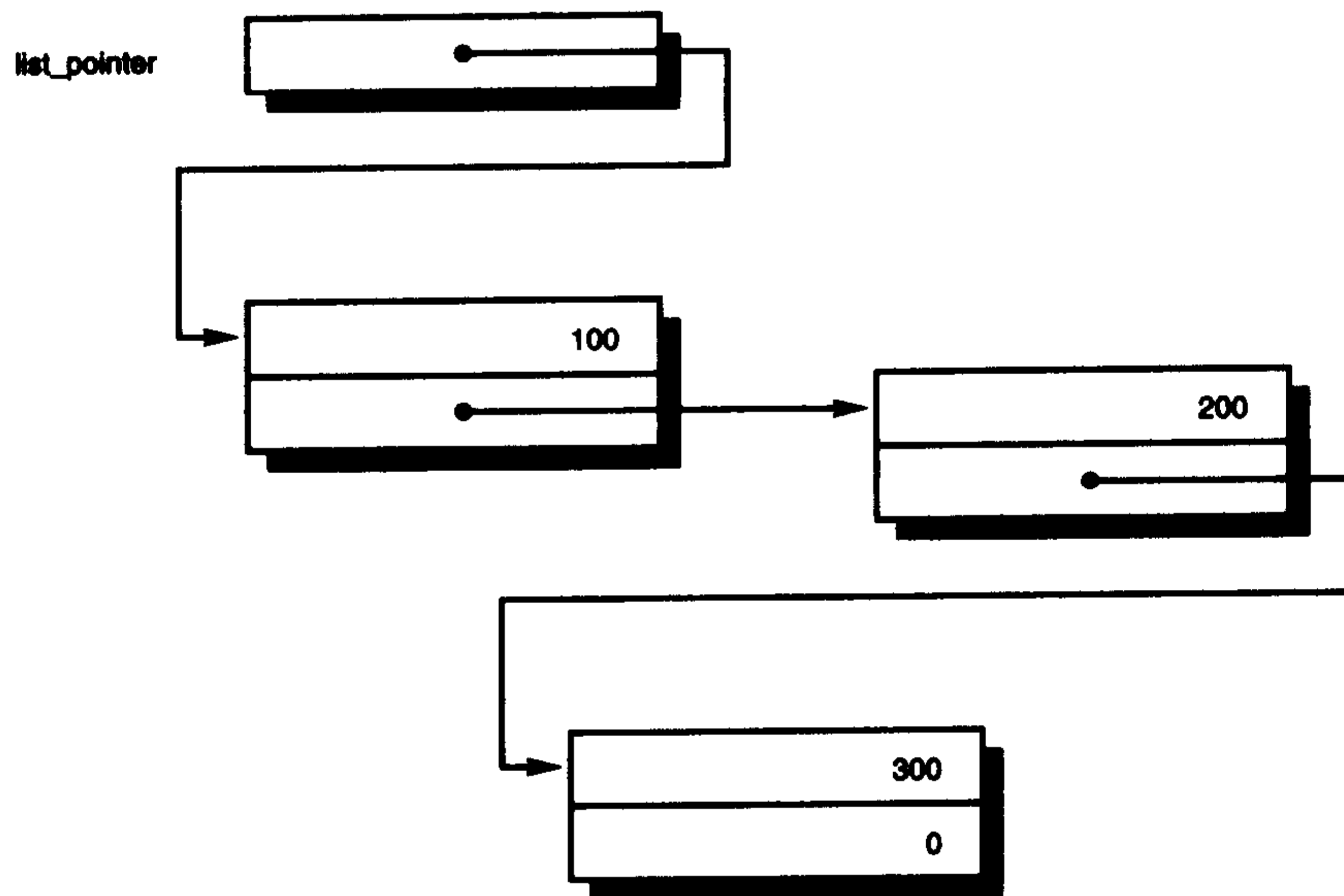


图 11.8 显示链表的头指针和作为结束的空指针

程序11.7 输出

```
100
200
300
```

程序定义了变量n1、n2和n3以及指针变量list_pointer，其中list_pointer初始化为指向n1，即表的第一个元素。接下来，程序将这三个元素链接成一个表，并将n3的成员next设置为标识表尾的空指针。

我们接下来用一个while循环来按序遍历表的每个元素。只要list_pointer的值不是空指针，这个while循环将一直执行。在while循环中的我们使用printf调用来显示由list_pointer指向的当前元素的成员value。

在printf语句后面的语句如下所示：

```
list_pointer = list_pointer->next;
```

该语句将list_pointer指向的结构的成员next的指针赋值给list_pointer。因此，第一次循环，将n1.next包含的指针（我们记得list_pointer初始化为指向n1）赋予list_pointer。因为这个值是一个指向结构n2的指针，其值不为空，所以，while循环将继续执行。

第二次while循环将显示n2.value的值，也就是200。接下来，n2的成员变量next被复制到list_pointer。因为我们设置这个值指向n3，所以第二次循环的结尾，list_pointer指向n3。

在第三次执行while循环时，printf调用显示包含在n3.value的值300。此时，list_pointer->next（实际上就是n3.next）被复制到list_pointer。因为n3的next成员为空指针，所以，while循环在执行了3次后结束。

读者可以用铅笔和纸来跟踪我们刚才讨论的while循环的操作，并注意各个变量的值的变化。弄清楚这个循环的操作是我们理解C语言中指针操作的关键。附带提一下，同样的while循环可用于顺序访问任意长度的链表，只要该链表以空指针结尾。

在程序中实际使用链表时，我们不会像本节所举例子那样，将明确定义的元素用通常的方法链接起来。我们这里所做的，只是用来举例说明使用链表的方法。在实际应用中，随着程序的执行，我们将要求系统给要链接到链表中的每个新元素分配内存。这一过程是由一种叫做“动态内存分配”的方法来完成的，它将在第17章里介绍。

关键字 const 和指针

我们已经知道如何用const修饰变量或数组的声明，以告诉编译器和读者，程序将不会修改这个变量或数组的值。对于指针来说，（使用const修饰符时）要考虑两件事情：一是指针是否会被修改，二是指针所指向的值是否会被修改。假定有如下声明：

```
char c = 'X';  
char *charPtr = &c;
```

指针变量charPtr被设为指向变量c，如果它总是指向c，则可被声明为一个指针常量如下：

```
char * const charPtr = &c;
```

（读法为“charPtr是一个指向字符的指针常量”。）因此，类似下面的一个语句：

```
charPtr = &d; // 不是有效的
```

将会使GNU C编译器产生如下的错误信息：²

```
foo.c:10: warning: assignment of read-only variable 'charPtr' (警告：给只读变量'charPtr'进行赋值)
```

相反的，如果charPtr指向的位置的值通过不会通过使用指针变量charPtr被改变，则可使用如下声明：

```
const char *charPtr = &c;
```

（读法为“charPtr指向一个字符常量”。）当然了，这并不意味着这个值不能够通过变量c所修改。这个声明只是意味着，它将不会因类似下面的语句而被修改：

```
*charPtr = 'Y'; // 不是有效的
```

上述语句会导致GNU C编译器产生一个如下的信息：

```
foo.c:11: warning: assignment of read-only location (警告：给一个只读单元赋值)
```

在指针变量和它所指向的单元都不改变的情况下，可应用下面的声明：

```
const char * const *charPtr = &c;
```

第一个const表明指针指向的单元的内容不会被改变，第二个const表明指针本身不会被改变。诚然，这个语句看起来有点儿让人糊涂，但是在讲解指针的时候还是应该提一下它。³

²读者的编译器也许会给出一个不同的警告信息，或者不给出一点儿信息。

³我们不会在每一个可以被用到的例子程序中使用 const 关键字，而只在挑选的例子中使用它。直到读者对类似于前面显示的表达式读起来熟练的时候，我们才能容易地理解更复杂一些的例子。

指针和函数

我们可以很方便的把指针和函数结合起来运用。也就是说，我们不仅可以按照正常的方式将指针作为一个参数传递给函数，而且也可以让函数返回一个指针作为其结果。

第一种情况，将指针作为参数传递给函数是很简单的。我们可以像平常那样将指针简单地包含在函数的参数表中就可以了。比如，我们要将前面的程序中的指针list_pointer传递给一个函数print_list，可以这样来写：

```
print_list (list_pointer);
```

在程序print_list里，形式参数必须声明为一个具有合适类型的指针：

```
void print_list (struct entry *pointer)
{
    ...
}
```

接下来，我们就可以将形式参数pointer当作一个正常的指针变量使用了。在将指针作为参数传递给函数时，我们需要记住：在调用函数时，指针的值将被复制到形式参数中。因此，在函数中对于形式参数的任何改变并不会影响传递给函数的实际指针。但是，这里需要提醒读者注意：尽管函数不能改变指针参数，但却可以改变指针指向的数据元素。程序11.8澄清了这一点。

程序11.8 应用指针和函数

```
// 应用指针和函数的程序

#include <stdio.h>

void test (int *int_pointer)
{
    *int_pointer = 100;
}

int main (void)
{
    void test (int *int_pointer);
    int i = 50, *p = &i;

    printf ("Before the call to test i = %i\n", i);
```

程序11.8 续

```
test (p);  
printf ("After the call to test i = %i\n", i);  
  
return 0;  
}
```

程序11.8 输出

```
Before the call to test i = 50  
After the call to test i = 100
```

函数test使用一个整型指针作为参数，在其内部，程序执行一个简单的语句，即将数值100赋值给整型指针int_pointer指向的位置。

主程序定义了一个整型变量i，并初始化其值为50。主程序还定义了一个整型指针p，将其指向变量i。接下来，程序显示i的值，调用函数test，并将指针作为参数传递给它。像我们从程序输出的第二行所看到的那样，实际上，函数test确实改变i的值，新值为100。

现在，我们来思考一下程序11.9。

程序11.9 运用指针修改数值

// 应用更多指针和函数的程序

```
#include <stdio.h>  
  
void exchange (int * const pint1, int * const pint2)  
{  
    int temp;  
  
    temp = *pint1;  
    *pint1 = *pint2;  
    *pint2 = temp;  
}  
  
int main (void)  
{  
    void exchange (int * const pint1, int * const pint2);  
    int i1 = -5, i2 = 66, *p1 = &i1, *p2 = &i2;  
  
    printf ("i1 = %i, i2 = %i\n", i1, i2);  
}
```

程序11.9 续

```

exchange (p1, p2);
printf ("i1 = %i, i2 = %i\n", i1, i2);

exchange (&i1, &i2);
printf ("i1 = %i, i2 = %i\n", i1, i2);

return 0;
}
    
```

程序11.9 输出

```

i1 = -5, i2 = 66
i1 = 66, i2 = -5
i1 = -5, i2 = 66
    
```

函数exchange的功能是交换由它的两个指针参数所指向的整型变量的值。函数的头部声明：

```
void exchange (int * const pint1, int * const pint2)
```

表明函数exchange接收两个整型指针参数，且两个指针都不会被函数所改变（用关键词const表明）。

程序使用局部整型变量temp在交换时保存一个整型的值。程序首先将pint1所指向的整数保存在temp中，接下来，程序将pint2指向的位置的值复制到pint1指向的位置，再将temp中的值保存在pint2指向的位置。这样就完成了交换。

主程序定义了两个整型变量i1和i2，并分别初始化其值为-5和66。接下来，程序定义了两个整型指针p1和p2，并分别令其指向i1和i2。随后，程序显示i1和i2的值，再调用exchange函数，并将两个指针p1和p2作为参数传递给它。Exchange函数交换由p1和p2所指向的整数的值。因为p1指向i1，p2指向i2，所以，在函数结束时，i1和i2的值就被交换了。从第二个printf调用的输出结果来看，交换操作进行正常。

第二个exchange调用更有意思一点。这一次，传递给函数的参数是指向i1和i2的指针，这两个指针是使用地址运算符对i1和i2进行操作所得到的。因为表达式&i1产生一个指向整型变量i1的指针，这也就满足了函数要求的参数类型（一个指向整型的指针）。对第二个参数也是同样的道理。从程序的输出结果我们可以看出，exchange函数仍然正常执行，将i1和i2的值又交换回它们的初始值。

读者应该认识到,若是不用指针,我们就无法写出exchange函数来交换两个整型的值,因为函数被限制为只能返回一个单值,另外,函数不能改变它的参数。请读者再认真细致地研究程序一下11.9,这个小程序说明了在C语言中处理指针时应该理解的关键概念。

程序11.10说明了一个函数是如何返回一个指针的。程序定义了一个函数findEntry,它的目的是查找一个链表以找出给定的值。如果找到此值,程序返回指向表中该元素的指针。若是打不到,则返回一个空指针。

程序11.10 从函数返回一个指针

```
#include <stdio.h>

struct entry
{
    int value;
    struct entry *next;
};

struct entry *findEntry (struct entry *listPtr, int match)
{
    while ( listPtr != (struct entry *) 0 )
        if ( listPtr->value == match )
            return (listPtr);
        else
            listPtr = listPtr->next;

    return (struct entry *) 0;
}

int main (void)
{
    struct entry *findEntry (struct entry *listPtr, int match);
    struct entry n1, n2, n3;
    struct entry *listPtr, *listStart = &n1;
    int search;

    n1.value = 100;
    n1.next = &n2;

    n2.value = 200;
    n2.next = &n3;

    n3.value = 300;
    n3.next = 0;
```

程序11.10 续

```
printf ("Enter value to locate: ");
scanf ("%i", &search);

listPtr = findEntry (listStart, search);

if ( listPtr != (struct entry *) 0 )
    printf ("Found %i.\n", listPtr->value);
else
    printf ("Not found.\n");

return 0;
}
```

程序11.10 输出

```
Enter value to locate: 200
Found 200.
```

程序11.10 输出 (回车)

```
Enter value to locate: 400
Not found.
```

程序11.10 输出 (第二次回车)

```
Enter value to locate: 300
Found 300.
```

函数findEntry的头部如下:

```
struct entry *findEntry (struct entry *listPtr, int match)
```

这条语句说明函数findEntry返回一个指向结构元素的指针, 并将这样的指针作为它的第一个参数, 将一个整型作为第二个参数。这个函数以一个while循环遍历链表的元素。直到在表中找到了匹配的值(此时立即返回listPtr的值)或是访问到了空指针(此时退出while循环, 并返回一个空指针)。

在主程序中, 我们首先像前面的程序那样建立一个链表, 然后要求用户提供一个要在表中查找的值, 接下来, 主程序调用findEntry函数, 并将指向表头的listStrart指针和用户输入的将要查找的值(search)作为参数传递给它。随后, 我们将findEntry函数返回的指针赋值给结构元素指针变量listPtr。如果listPtr不为空, 我们就将它指向的成员value显示出来。这个值应该和用户输入的值相同。如果listPtr为空, 我们则显示信息“Not found.”(译者注: “没有找到”)。

程序的输出验证了数值200和300都在表中，而数值400不在表中，事实上，它也确实不在表中。

在程序11.10中，由findEntry函数返回的指针初看上去没有任何用处（它指向的值是我们已经知道的）。然而，在很多更实际的情况下，我们可以用这个指针来存取表中数据项所包含的其它成员。例如，我们有一个链表，它由第10章“字符串”中的字典条目组成。于是，我们可以调用findEntry函数（或者将其重命名为lookup，像在第10章那样）来在字典条目链表中查找给定的单词。接下来，我们就可以用lookup函数返回的指针来存取条目中的definition成员。

将字典组织成链表有如下的优点：给字典插入一个新单词是很容易的，在决定了所要插入的位置后，只要像本章前面所举的例子那样，简单地调整一些指针就可以了；从字典中删除一个单词也是很简单的。最后，像读者将要在第17章所看到的那样，采取链表结构可以动态地扩展字典的大小。

然而，采取链表方法来组织字典也存在一个重大的缺陷：我们不能将快速的二分查找算法应用于这样的表。这个算法只能使用于可以直接索引的数组。不幸的是，因为表中的每个单元只能由前面的那个进行存取，所以对链表来说，没有一种查找方法可以比直接、按序查找更快。

还有一种方法可以对元素进行方便的插入和删除，其查找速度也很快，这就是使用一个叫做树的数据结构。其它方法，例如利用哈希表，也是可行的。读者可以查阅有关刚才所讨论的数据结构的其它相关资料，如《计算机编程艺术》第三册的排序和查找（Donald E. Knuth, Addison-Wesley），这些数据结构类型可以用前面描述过的技巧在C语言中很方便地实现。

指针和数组

C语言中，指针最常见的一个用法是将其指向一个数组。运用指向数组的指针的主要原因有两个：书写方便和程序高效。通过使用指向数组的指针，我们常常可以写出占用较少内存并且执行快速的代码。通过本节的学习，读者就会明白这个原因。

假定有一个包含100个整数的数组values，我们可以定义一个叫做valuesPtr的指针，可以用它通过下面语句来存取这个数组中的整数：

```
int *valuesPtr;
```

当定义指向数组元素的指针时，我们不需要指明这个指针是“指向数组的指针”类型，我们只需要声明这个指针是指向该数组所包含的元素的类型就可以了。

如果我们有一个叫做text的字符数组，我们可以通过下面语句简单地定义一个指向text中元素的指针：

```
char *textPtr;
```

为了设置valuesPtr指向values数组的第一个元素，我们只要简单地这样写就可以了：

```
valuesPtr = values;
```

在这种情况下，我们并不需要使用地址运算符，因为C语言编译器将一个没有带下标的数组名看作是一个指向数组的指针。实际上，不带下标的数组名values本身就是指向values的第一个元素的指针（见图11.9）。

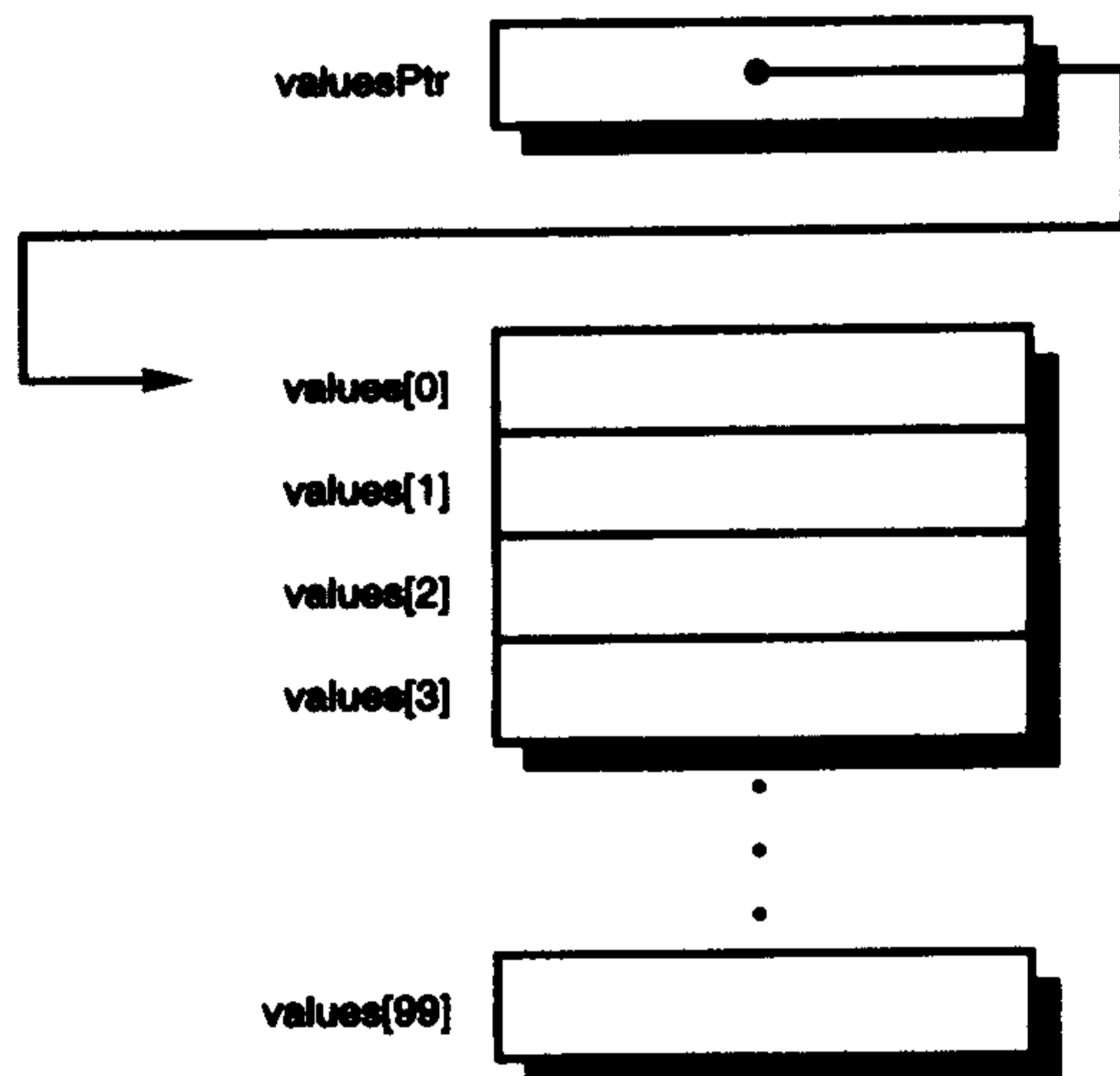


图 11.9 指向一个数组元素的指针

如果我们将地址运算符应用到数组的第一个元素中，同样可以产生一个指向values数组第一个元素的指针。因此，下面语句：

```
valuesPtr = &values[0];
```

同样可以将指向values的第一个元素的指针赋值给指针变量valuesPtr。

为了设置textPtr指向text数组的第一个字符，我们可以使用下面的语句：

```
textPtr = text;
```

或者下面的语句：

```
textPtr = &text[0];
```

选择哪个语句完全取决于编程者的个人喜好。

当我们想要按序访问数组的元素的时候，指向数组的指针就可以真正发挥其作用了。假定我们已经定义了指针变量valuesPtr，并设置其指向values的第一个元素，则下面的表达式：

```
*valuesPtr
```

可用来存取数组values的第一个元素，即values[0]。为了通过变量valuesPtr来引用values[3]，我们可以给指针变量加上3，然后再使用指针运算符，如下所示：

```
*(valuesPtr + 3)
```

通常，表达式

```
*(valuesPtr + i)
```

可以用来存取包含在values[i]里的值。

因此，为了将values[10]设置为27，我们可以使用下面的表达式来实现：

```
values[10] = 27;
```

或者，如果使用valuesPtr，则可以写成：

```
*(valuesPtr + 10) = 27;
```

为了将valuesPtr指向数组values的第二个元素，我们可以运用地址运算符到values[1]，并将其结果赋予valuesPtr：

```
valuesptr = &values[1];
```

（但是这里我们还有一种更简单的方式来完成这项工作），如果valuesPtr指向values[0]，我们只要简单地对valuesPtr加一就可以使其指向values[1]：

```
valuesPtr += 1;
```

在C语言中，这是一个很有用的表达式，我们可以对任意数据类型的指针使用它。

综上所述，如果a是一个元素为x类型的数组，px是x的指针类型，而i和n是整型常量或变量，则语句：

```
px = a;
```

用来将指针px指向a的第一个元素，表达式：

```
*(px + i)
```

用来引用包含在a[i]中的值。进而，语句：

```
px += n;
```


用来将`px`指向数组中的第`n`号元素，不管数组中的元素到底是那种类型。

在处理指针的时候，使用递增和递减运算符 `++` 和 `--` 是很方便的。应用一个递增运算符到一个指针和给指针加1具有相同的效果，同样，用一个递减运算符到一个指针和给指针减1具有相同的效果。因此，如果`textPtr`定义为一个字符指针并被设置为指向字符数组`text`的开头，则语句：

```
++textPtr;
```

将使`textPtr`指向`text`数组的下一个字符，即`text[1]`。同样的，语句：

```
--textPtr;
```

将使`textPtr`指向`text`数组的前一个字符，当然，这里我们假定在执行这个语句时`textPtr`没有指向`text`数组的开头。

在C语言中，比较两个指针变量是完全合法的，当两个指针指向同一个数组的元素时，指针的比较特别有用。例如，假定某个数组包含100个元素，通过比较`valuesPtr`指针和指向数组最后一个元素的指针，我们可以判断指针`valuesPtr`指向的位置是否超过了数组的结尾。因此，通过下面的表达式：

```
valuesPtr > &values[99]
```

如果其值为`TRUE`（非零），则表明`valuesPtr`指向的位置超过了数组`values`的最后一个元素；如果其值为`FALSE`（零），则没有超过。根据前面的讨论，因为，没有带下标的`values`就是一个指向数组`values`开头的指针（记住，它和 `&values[0]` 是相同的。），所以我们可以用下面表达式来代替前面的判断表达式：

```
valuesPtr > values + 99
```

程序11.11举例说明了指向数组的指针。函数`arraySum`用来计算一个整数数组所包含的所有元素的和。

程序11.11 使用指向数组的指针

```
// 计算整数数组所有元素之和的函数

#include <stdio.h>

int arraySum (int array[], const int n)
{
    int sum = 0, *ptr;
    int * const arrayEnd = array + n;

    for ( ptr = array; ptr < arrayEnd; ++ptr )
        sum += *ptr;
```

程序11.11 续

```
    return sum;
}

int main (void)
{
    int arraySum (int array[], const int n);
    int values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

    printf ("The sum is %i\n", arraySum (values, 10));
    return 0;
}
```

程序 11.11 输出

The sum is 21

在函数arraySum内部，我们定义了整型指针常量arrayEnd，并将其指向数组最后一个元素的后面。接下来，我们用一个for循环来遍历数组元素。在循环的初始化表达式中，我们将ptr的值指向数组的开头。每一次循环，我们将ptr指向的数组元素被加到sum之中。随后，for循环的循环表达式将ptr的值加1，从而使其指向数组的下一个元素。当指针ptr访问完数组的最后一个元素时，程序退出for循环，并将sum的值返回到调用程序。

稍微离题一下——关于程序的优化

需要指出的是，因为我们能够在for循环里明确地比较ptr的值和数组的末尾，所以局部变量arrayEnd实际上不是必需的，如下所示：

```
for ( ...; pointer <= array + n; ... )
```

使用arrayEnd的唯一目的是优化。每一次for循环，程序都要计算循环条件表达式。因为我们在循环里从没有改变表达式array+n，所以它的值在整个for循环里是一个常量。如果我们在进入循环前将它一次计算出来，我们就可以省去在每一次循环里重新计算其值的工作。虽然在这个程序中，数组中仅有十个元素，特别是arraySum函数仅被程序调用了一次，节省的计算时间几乎可以忽略不计，但是如果程序频繁地调用这个函数对一个很大的数组求和，那么节省的时间就会相当可观。

另一个值得讨论的优化问题与程序中指针的应用有关。在前面讨论的arraySum函数里，在for循环中我们使用表达式*ptr用来存取数组元素。以前，我们使用带有下标变量（如i）的for循环来编写arraySum函数，在循环中将array[i]的值加入到sum中。通常情况下，使用下标变量访问数组元素的过程会比使用指针访问花费更多的时间。实际上，这就是使用指针来存取数组元素的主要原因——这样产生的代码通常效率更高。当然了，如果我们不是按照顺序对数组进行存取，那么使用指针就没有什么优化效果了。具体到这个程序，因为表达式*(pointer + j)的执行时间是和表达式array[j]一样的，所以使用指针不再更有效率。

数组还是指针

回想一下，如果要把一个数组作为参数传递给函数，我们简单地指定数组的名字即可，就像我们前面使用的函数arraySum一样。我们还应该记得，如果需要生成指向数组的指针，只要使用数组的名字即可。这就暗示着，在调用函数arraySum时，传递给函数的实际上是一个指向数组values的指针。实际的情况正是这样，这也正好解释了为什么我们可以在一个函数内对数组的元素进行修改。

但是，如果传递给函数的参数是一个指向数组的指针，为什么在函数的声明中，我们没有将对应的形式参数声明为一个指针呢？换句话说，在函数arraySum中，形势参数数组的声明为什么不用下面的形式呢？

```
int *array;
```

另外，在函数内部所有对于数组的引用不应该都使用指针来实现吗？

为了回答这个问题，请读者回想一下前面有关指针和数组的一些讨论。如果指针变量ValuesPtr指向一个叫做values的数组的开头，并且该指针的数据类型与数组的数据类型相同，那么表达式*(valuesPtr + i)不管在任何情况下都应该等价于表达式values[i]。从这可以看出，我们也可以用表达式*(values + i)来引用数组values的第i个元素，并且，在通常情况下，对于任意类型的数组x，表达式 x[i] 在C语言中总是等价于 *(x+i)。

正如我们所看到的那样，在C语言中，指针和数组有着非常密切的关系。这就是为什么我们在函数arraySum内即将array声明为整型数组类型，也可以声明为指向整型的指针类型。在前面的程序中，这两种声明的使用都是正确的，读者可以自行试一下，看结果到底怎样。

如果我们想用下标来引用传递给一个函数的数组元素，需要将相应的形式参数声明为数组。这样就更准确地表明了函数中使用的是数组。相似的，如果我们将该形式参数作为一个指向数组的指针使用，则要将它声明为一个指针类型。

现在读者可以知道，在前面的例子程序中，我们可以声明符号array为一个整型指针，随后就可以将其作为指针使用了，从而可以省略函数的指针变量ptr，如程序11.12所示。

程序11.12 求一个数组的元素的和

```
// 计算整数数组元素之和的函数

#include <stdio.h>

int arraySum (int *array, const int n)
{
    int sum = 0;
    int * const arrayEnd = array + n;

    for ( ; array < arrayEnd; ++array )
        sum += *array;

    return sum;
}

int main (void)
{
    int arraySum (int *array, const int n);
    int values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

    printf ("The sum is %i\n", arraySum (values, 10));

    return 0;
}
```

程序11.12 输出

The sum is 21

这个程序相当简明，不需要多加解释。因为在循环开始之前没有数值必须被初始化，所以我们省去了for循环中的第一个表达式。需要重复强调的一点是，当我们调用函数arraySum时，传递了一个指向数组values的指针，在函数中该指针对应形式参数array。与由array引用的数值相反，对于array本身值的修改不会影响数组values的内容。因此，我们对于指针array应用递增运算只是递增了一个指向数组values的指针，并没有影响数组的内容。（当然我们知道，如果愿意的话，我们可以改变数组的值，只要简单地给由指针引用的元素进行赋值即可。）

指向字符串的指针

指向数组的指针最常见的应用是将指针指向字符串。这样使用的原因是为了书写的方便和程序高效。为了说明使用指向字符串的指针是如何的容易，我们编写了一个叫做 `copyString` 的函数，用来将一个字符串复制到另一个中。如果我们用普通的下标方法，则编写的函数可能如下所示：

```
void copyString (char to[], char from[])
{
    int i;

    for ( i = 0; from[i] != '\0'; ++i )
        to[i] = from[i];

    to[i] = '\0';
}
```

在空字符被复制到数组 `to` 之前，`for` 循环就退出了。因此，函数中的最后一个语句是必要的。

如果用指针来编写 `copyString`，我们就不再需要下标变量 `i` 了。程序 11.13 显示了指针版本的 `copyString` 函数。

程序 11.13 指针版本的 `copyString`

```
#include <stdio.h>

void copyString (char *to, char *from)
{
    for ( ; *from != '\0'; ++from, ++to )
        *to = *from;

    *to = '\0';
}

int main (void)
{
    void copyString (char *to, char *from);
    char string1[] = "A string to be copied.";
    char string2[50];

    copyString (string2, string1);
    printf ("%s\n", string2);

    copyString (string2, "So is this.");
    printf ("%s\n", string2);

    return 0;
}
```

程序11.13 输出

```
A string to be copied.  
So is this.
```

函数copyString定义了两个形式参数to和from,它们的类型是字符指针,而不像前面版本的copyString中那样是字符数组。这反映了在函数中使用这两个变量的方式。

接下来,程序进入一个for循环(没有初始条件),将from指向的字符串复制到to指向的字符串。每一次循环中,指针from和to都进行加1操作。这样使得from指针指向下一个将要被从源串拷贝的字符,并且,也使得to指针指向目标串中将要存储下一个字符的位置。

当from指针指向空字符时,程序退出for循环。随后,函数将空字符放置到目标串的末尾。

在主程序中, copyString函数被调用了两次。第一次将串string1的内容拷贝到string2中,第二次是将字符串常量“So is this.”拷贝到string2中。

字符串常量和指针

在前面程序中,我们使用了下面的函数调用:

```
copyString (string2, "So is this.");
```

这暗示了当一个字符串常量作为参数传递给函数时,实际上传递的是指向这个字符串的指针。实际上,不仅在这种情况下如此,在C语言中任何时候使用字符串常量时也是如此,C语言自动产生一个指向那个字符串的指针。因此,如果我们像下面那样声明textPtr是一个字符指针:

```
char *textPtr;
```

那么,语句:

```
textPtr = "A character string.";
```

将一个指向字符串常量“A character string.”的指针赋值给变量textPtr。在这里读者一定要注意字符指针和字符数组的差别。因此上面的赋值语句对于字符数组来说是无效的。例如,如果用下面的语句定义text为一个字符数组:

```
char text[80];
```

则下面这样的语句就是错误的:

```
text = "This is not valid.";
```

在C语言中,唯一可以对一个字符数组使用这样形式的赋值语句的时候,就是初始化它的时候,如:

```
char text[80] = "This is okay.";
```


用这种方式来初始化数组text，并不是将一个指向字符串"This is okay."的指针保存到数组text中，实际上，C语言将这些字符本身保存在数组array的对应位置。

如果text是一个字符指针，用下面语句初始化text：

```
char *text = "This is okay.";
```

则将一个指向字符串"This is okay."的指针赋值到text。

作为字符串与字符串指针之间差别的另外一个例子，我们用如下方法建立一个叫做days的数组，它包含指向一个星期中每一天的名字的字符指针：

```
char *days[] =  
{ "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" };
```

数组days被定义为包含7个元素，每一个元素都是一个指向字符串的指针。于是，days[0]包含指向字符串"Sunday"的指针，days[1]包含指向字符串"Monday"的指针，等等（见图11.10）。接下来，我们可以用下面的语句显示一个星期第三天的名字：

```
printf ("%s\n", days[3]);
```

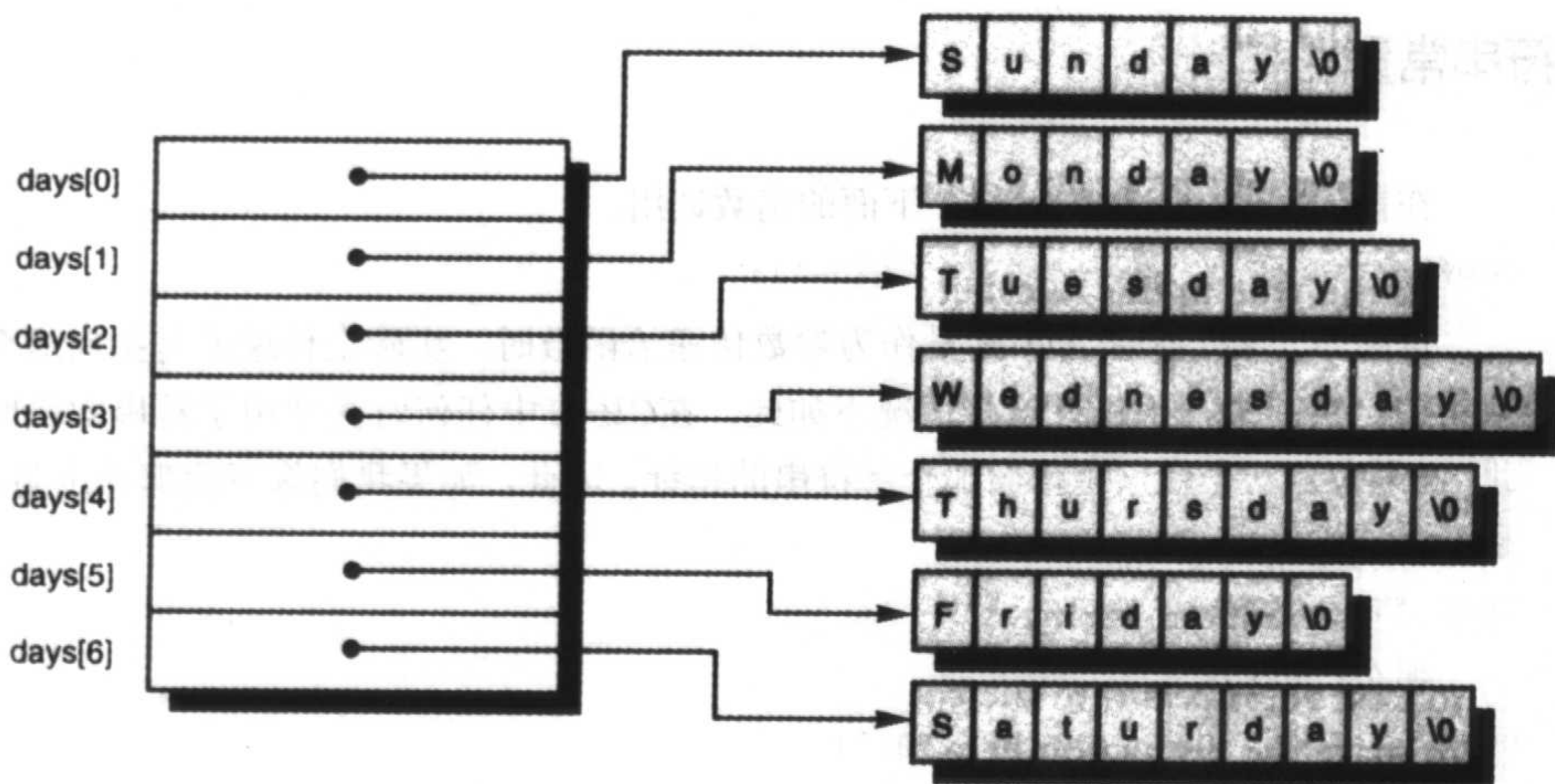


图 11.10 指针数组

再次谈谈递增和递减运算符

在前面的程序中，不论任何时候使用递增或递减运算符，它都是表达式中唯一的运算符。当写下表达式 ++x 时，我们知道这将给变量x 加一。如果x是一个指向数组的指针，这个操作就将x指向数组的下一个元素。

在使用了递增和递减运算符的表达式中，也可以使用其它运算符。这种情况下，我们

必须了解这些运算符是如何工作的。

到目前为止，我们使用递增和递减运算符时，总是将它们放置在将要进行递增或递减的变量的前面，比如，要递增一个变量*i*，可简单地写成：

```
++i;
```

事实上，像下面语句那样，将递增运算符放置在变量的后面也是有效的：

```
i++;
```

两个表达式都是合法的，也都很好完成了工作，也就是说：递增*i*的值。在第一种情况下，`++` 放置在操作数的前面，递增运算符可以看成是“前加”；在第二种情况下，`++` 放置在操作数的后面，递增运算符可以看成是“后加”。

递减运算符和递增运算符类似。因此，语句：

```
--i;
```

对*i*进行了“前减”操作，而语句：

```
i--;
```

则对*i*进行了“后减”操作。两者具有相同的效果，即对*i*的值进行了减1操作。

如果我们将递增和递减运算符运用到更复杂的表达式中时，前置和后置运算符的差别就表现出来了。

假定有两个整型变量*i*和*j*，如果我们将*i*设为0，则下面语句：

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

正如我们预料到的那样，将1赋值给*j*，而不是0。在前置运算符的情况下，变量的值首先被加1，然后再参加到表达式的求值中。因此，在前面的表达式中，变量*i*的值首先由0递增为1，然后，其值被赋给*j*。这与下面两个语句的作用相同：

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

如果我们在语句中用了“后加”运算符：

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

则i的值是在赋值给j之后才递增的。因此，前面语句执行时，先将0赋值给j，然后对i进行递增操作，即加1，这个语句与下面两个语句完成的工作相同：

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

下面再举一个例子。如果i为1，则下面语句：

```
x = a[--i];
```

将a[0]的值赋给x。因为，变量i在其值作为下标使用之前，已经进行了递减操作。而语句：

```
x = a[i--];
```

将a[1]的值赋给x。因为，变量i的值在作为下标使用过之后，才进行递减操作。

下面再给出前置和后置递增和递减运算符差别的第三个例子，函数调用：

```
printf ("%i\n", ++i);
```

在i递增后才将其它传递给printf函数，然而函数调用：

```
printf ("%i\n", i++);
```

在将i的值传递给printf函数之后，才对i进行递增操作。因此，如果i为100，则第一个printf调用显示101，而第二个printf调用显示100。但不管那种情况，在语句执行完毕后，i的值都是101。

在介绍程序11.14之前，我们给出这一主题的最后例子：如果textPtr是一个字符指针，则表达式：

```
*(++textPtr)
```

首先递增textPtr，然后取出它后指向的字符。然而，表达式：

```
*(textPtr++)
```

在它的值递增之前就取回了textPtr所指向的字符。不管是那种情况，圆括号都不是必需的，因为*和++运算符具有相同的优先级，而且它们是“自右向左”结合的。

现在再回到程序11.13的copyString函数，这次我们通过直接在赋值语句里使用递增运算符来重写它。

因为在for循环中，每次赋值语句每次执行后，我们才对指针to和from进行递增操作，所以递增操作应该作为后置运算符加到赋值语句。于是，程序11.13的for循环修订后的代码如下：

```
for ( ; *from != '\0'; )
    *to++ = *from++;
```


循环体中赋值语句的执行过程如下：首先程序得到由from指向的字符，然后对from进行递增操作，以指向源串中的下一个字符，接下来，我们讲该字符存储到由to指向的位置，然后对to进行递增操作，以指向目标串中的下一个位置。

请读者认真研究前面的赋值语句，直到完全弄明白为止。在C语言程序中，这样类型的语句是非常普遍的。正因为如此，所以在继续下面内容之前，我们有必要完全弄明白。

前面的for语句几乎没有什么用处，因为它既没有初始化表达式，也没有循环表达式。实际上，while循环能够更好的表达程序的逻辑，正如程序11.14所作的那样。这个程序包含了copyString函数的一个新版本。在while语句中，我们使用了一个常识：空字符的值等价于值0，这在有经验的C语言程序员看来是很普通的。

程序11.14 copyString函数的修订版本

```
// 复制字符串到另一指针的函数 Ver. 2

#include <stdio.h>

void copyString (char *to, char *from)
{
    while ( *from )
        *to++ = *from++;

    *to = '\0';
}

int main (void)
{
    void copyString (char *to, char *from);
    char string1[] = "A string to be copied.";
    char string2[50];

    copyString (string2, string1);
    printf ("%s\n", string2);

    copyString (string2, "So is this.");
    printf ("%s\n", string2);

    return 0;
}
```

程序11.14 输出

```
A string to be copied.  
So is this.
```

指针运算

像在本章前面所看到的那样，我们可以对指针和整数进行加、减操作。进而，我们可以对两个指针进行比较，看它们是否相同，或者是看它们的大小关系。关于指针还有一个操作：那就是对相同类型的两个指针进行减法操作。C语言中两个指针减法操作的结果是它们之间所包含元素的个数。因此，如果a是一个指向任意元素类型的数组，而b是另一个同类型的指针，该指针指向同一数组更远的一个位置，则表达式b-a代表这两个指针之间的元素个数。例如，如果p是一个指向数组x的某个元素的指针，则语句：

```
n = p - x;
```

将p所指的元素的下标数赋值给变量n（假定n是一个整型变量）⁴。因此，如果p由以下语句设置为指向第100个元素：

```
p = &x[99];
```

那么，变量n在前面的减操作完成后值为99。

作为对所学的指针减操作知识的一个实际应用，我们来编写一个第10章的stringLength函数的新版本。

在程序11.15中，我们使用字符指针cptr来遍历由string指向的字符，直到抵达空字符。在cptr到达空字符时，我们从cptr减去string，就得到了包含在string里的元素（字符）的个数。程序的输出验证了该函数工作是正常的。

程序11.15 用指针来计算字符串的长度

```
// 计算字符串长度的函数——指针版  
  
#include <stdio.h>  
  
int stringLength (const char *string)  
{  
    const char *cptr = string;
```

⁴ 由两个指针的减操作产生的结果的实际类型是ptrdiff_t，它在标准头函数文件<stddef.h>里定义。

程序11.15 续

```
while ( *cptr )
    ++cptr;
return cptr - string;
}

int main (void)
{
    int stringLength (const char *string);

    printf ("%i ", stringLength ("stringLength test"));
    printf ("%i ", stringLength (""));
    printf ("%i\n", stringLength ("complete"));

    return 0;
}
```

程序11.15 输出

17 0 8

指向函数的指针

为了完整起见，这里我们再介绍一个关于指针稍微高级一点儿的概念，即指向函数的指针。为了使用指向函数的指针，C语言编译器不仅需要知道指向函数的指针变量，还要知道函数返回值的类型，还包括它的参数的数量和类型。为了声明符号 `fnPtr` 为一个“指向一个返回整数且不带任何参数的函数”的指针，我们使用如下的声明：

```
int (*fnPtr) (void);
```

`*fnPtr`两边的小括号是必需的，否则，因为函数调用运算符`()`比指针运算符`*`的优先级高，C语言编译器将会把前面的语句作为一个函数的声明，这个函数叫做`fnPtr`，它返回一个指向整型的指针。

为了将函数指针指向一个指定的函数，我们可以简单地将函数的名字赋值给它。因此，如果`lookup`是一个函数，它返回一个整型，且没有参数，则语句：

```
fnPtr = lookup;
```

将一个指向函数的指针存储到函数指针变量`fnPtr`之中。在这里，不带后面一对小括号的函数名和不带下标的数组名的作用是类似的。C语言编译器会自动地产生一个指向该

函数的指针。函数名前面允许有一个&符号，但却是不必要的。

如果我们在程序前面没有定义lookup函数，则在赋值之前必须对函数进行声明。因此，我们需要在把指向函数的指针赋值给变量fnPtr之前，加上下面这样的语句：

```
int lookup(void);
```

如果需要通过函数指针变量间接调用函数，我们可以在函数指针变量后面加上函数调用操作符——也就是一对小括号，然后在小括号里列出要传递给函数的参数。例如：

```
entry = fnPtr ();
```

调用了由fnPtr指向的函数，并存储返回值到变量entry之中。

函数指针的一个最常用的用途是将函数作为参数传递给其它函数。例如，标准C语言库函数qsort就用到这个特性。此函数对一个数组的数据元素进行快速排序，该函数接收一个指向函数的指针作为它的参数，每当qsort需要比较数组中的要被排序的两个元素时，这个函数就会被调用。用这样方式，qsort可以对任何类型的数组进行排序，而实际上对数组中任意两个元素进行比较的函数由用户提供，而不是由qsort函数自己完成的。在附录B，“C语言标准库”中，我们对qsort函数进行了更加细致的描述，并提供了一个实际应用的例子。

函数指针的另一个常见应用是用来产生一个派遣表。我们不能将函数存储到一个数组元素中，但是我们可以将函数指针存储到数组中。通过函数指针，我们可以生成一个包含函数指针的表。例如，我们可以生成一个表，用来处理用户输入的不同的命令。表的每一个单元都包含了一个命令名和一个指向函数的指针，调用这个函数可以处理特定的命令。现在，不管用户何时输入了一个命令，我们都可以到表中查找该命令，并调用相应的函数来加以处理。

指针和内存地址

在结束有关C语言指针的讨论之前，我们还要讨论一下它们如何实现的细节。计算机的内存可以看作是一个有序的存储单元的集合。每一个内存单元都有一个编号，叫做内存地址。典型情况下，内存的第一个地址是0。在大多数计算机系统里，一个单元叫做一个字节。

计算机用内存来存储我们的程序指令，也存储和程序中变量的值。因此，如果我们声明了一个叫做count的整型变量，则系统将在内存中指派一个存储位置，在程序执行时保存count的值。例如，在计算机内存里，这个地址的编号也许是500。

幸运的是，高级编程语言，如C，有一个优点，就是我们自己不需要关心分配给变量的是那些内存地址，这些由系统自动处理。然而，知道每个变量都有一个唯一的一个关联地址，对于我们理解指针的操作会有所帮助。

在C语言里，当我们对一个变量运用地址运算符时，得到的结果是这个变量在计算机内存里的实际地址。（显然，这也就是地址运算符的名字由来。）所以，下面的语句：

```
intPtr = &count;
```

将计算机分配给变量count的内存地址赋值给intPtr。因此，如果count存储在地址500，包含了一个值为10，则这个语句将值500赋予intPtr，像图11.11所示那样：

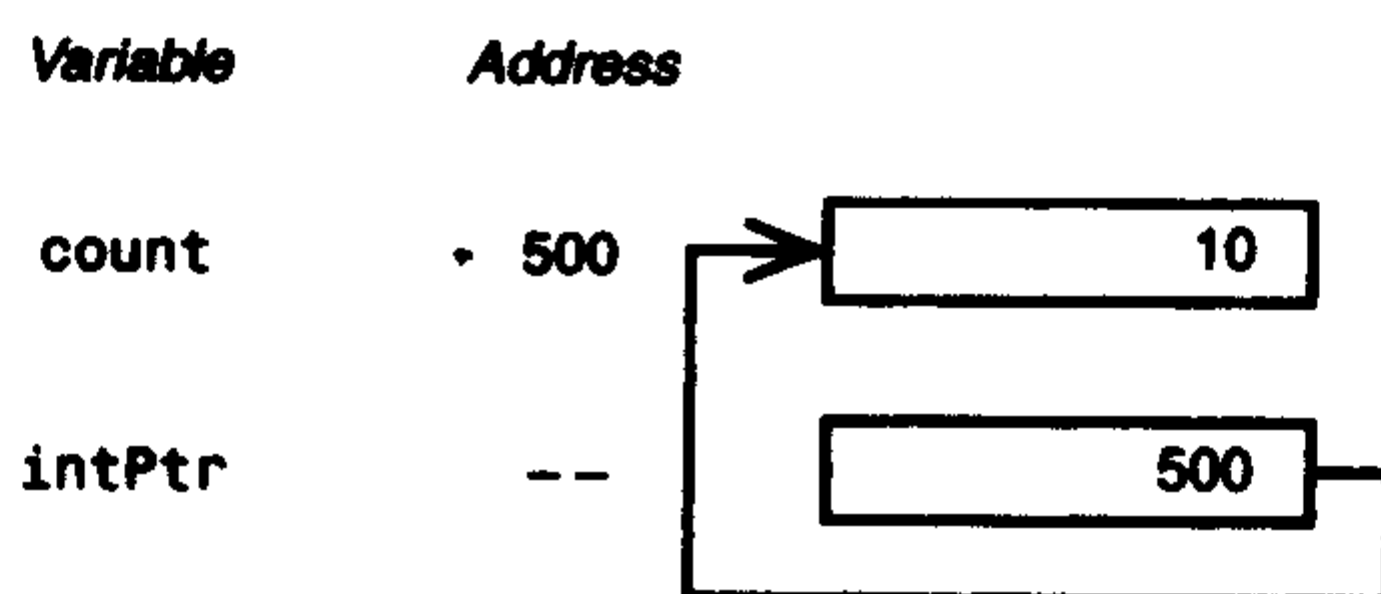


图 11.11 指针和内存地址

如图11.11显示，我们将intPtr的地址显示为--，因为它的实际值对这个例子来说是没有关系。

用对一个指针变量运用指针运算符，如下表达式所示：

```
*intPtr
```

则告诉计算机，将包含在指针变量的值看作是一个内存地址。于是，计算机取出内存地址里存储的值，并根据指针变量的类型对该值作出解释。因此，如果intPtr是一个整型指针，则通过intPtr所指向的内存地址里的值被计算机解释为一个整型数。在我们的例子里，存储在地址500里的数值被计算机取出，并解释为一个整型。最后，这个表达式的结果是10，其类型为整数。

像下面语句那样，通过指针保存某个值类似的方式进行操作。

```
*intPtr = 20;
```

intPtr的内容被取出，并被计算机看作是一个内存地址，指定的整数值被存储到那个地址里。因此，在上面的语句里，整数20被存储在内存地址为500的地方。

有些时候，系统程序员在编写程序的时候，必须存取计算机内存中的特定区域。这种情况下，知道指针变量的实际原理是很有用的。

正像我们从本章看到的那样，指针是C语言里一个功能强大的组成部分。指针变量定义方式的灵活性可以超过本章中所举的例子。例如，我们可以定义一个指向指针的指针，甚至是指向指针的指针的指针。这些用法超出了本书的范围，尽管它们只是对本章中所学的指针知识逻辑上的简单扩展。

对于初学者来说，指针很可能是最难掌握的内容。在继续向下学习之前，读者需要将本章的所有章节再读一遍，不要留有不明白的地方。完成以下的练习对于读者理解本章内容会有所帮助。

练习

1. 输入并运行本章15个程序，并将每个程序的输出与书本里程序的输出结果作以比较。
2. 编写函数insertEntry，该函数插入一个新结点到链表之中。insertEntry接收两个strut entry类型的指针作为参数（entry类型的定义如本章正文中所示），一个参数指向要被插入的单元，另一个指向将要在其后插入新单元的元素。
3. 练习2中所编写的函数只能在表中一个存在的元素后面插入一个元素，我们不能使用该函数在表头之前插入一个新元素。我们怎样才能克服这个问题呢？（提示：考虑建立一个指向表头的特殊的结构。）
4. 编写一个函数removeEntry以从链表中删除一个元素。它的唯一的参数是一个指向链表元素的指针。我们让该函数删除参数指针指向的元素后面的那个元素。（考虑一下：为什么不能删除由参数指向的那个结点呢？）我们需要用到在练习3中所建立的结构来处理一个特殊情况，即删除表中的第一个元素。
5. 双向链表指的是链表每一个结点都包含两个指针，一个指向表中前一个结点，另一个指向表中下一个结点。请给双向链表的结点定义一个适当的结构，并编写一个小程序来实现一个双向链表，并打印出表中的元素。
6. 为双向链表（在函数中它和以前练习中使用的那些单链表类似）编写insertEntry和removeEntry函数。为什么removeEntry函数现在可以将直接指向将要从链表中被删除的元素的指针作为它的参数？
7. 将第8章“使用函数”中的sort函数改编成指针版本。确保函数只使用指针，而不使用下标变量。

8. 编写一个叫做sort3的函数，它可以对三个整数按升序进行排列（不要用数组来实现）。
9. 重写第10章的readLine函数，使用字符指针而不是数组来实现其功能。
10. 重写第10章的compareStrings函数，用字符指针代替数组来实现。
11. 假定date结构的定义如本章中所示，编写一个叫做dateUpdate的函数，它用一个指向date结构的指针作为参数，并将其更新为后一天（具体算法见程序9.4）。
12. 给出以下声明：

```
char *message = "Programming in C is fun\n";  
char message2[] = "You said it\n";  
char *format = "x = %i\n";  
int x = 100;
```

判断下面printf语句集合中每个printf调用是否有效，并与集合里的其它调用产生的输出结果相同。

```
/* set 1 */  
printf ("Programming in C is fun\n");  
printf ("%s", "Programming in C is fun\n");  
printf ("%s", message);  
printf (message);
```

```
/* set 2 */  
printf ("You said it\n");  
printf ("%s", message2);  
printf (message2);  
printf ("%s", &message2[0]);
```

```
/* set 3 */  
printf ("said it\n");  
printf (message2 + 4);  
printf ("%s", message2 + 4);  
printf ("%s", &message2[4]);
```

```
/* set 4 */  
printf ("x = %i\n", x);  
printf (format, x);
```


Operations on Bits

位运算

就像前面所提到的，C语言最初设计时主要是用来编写系统程序。指针就是这方面很好的例子，因为它给程序员提供了强大的控制和访问计算机内存的能力。同样，系统程序员会经常遇到需要进行位处理的问题。因此，C语言提供了许多专门用于进行位处理的位运算符。

我们回想一下前面章节中讨论过的一个概念，字节。在大多数计算机系统里，一个字节包含8个叫做位的小单元。一个位只能有两个取值，或者为0，或者为1。因此，保存在计算机内存地址为1000的一个字节，也许会是下面所示的8个二进制数的一个位串：

01100100

一个字节最右面的位是最低位，也被称为最不重要位，而最左边的位是最高位，或者是最重要位。如果我们把位串看作一个整数，则最右边的位就表示 2^0 或1，紧跟在它左面的一位表示 2^1 或2，再朝左的一位表示 2^2 或4，如此类推。因此，前面二进制位串的十进制值就是 $2^2+2^5+2^6=4+32+64=100$ 。

用二进制串表示负数的方法有点儿不同。大多数计算机用所谓的二类补数来表示负数（译者注）。此时，最左边的一位为符号位。如果此位为1，则表示负数；为0，则表示正数。余下的位用来表示具体的数值。使用二类补数的编码方案表示时，-1的所有位都是1，即

11111111

将一个十进制负数转化为二进制符号数时，首先给这个数加1，然后用其绝对值的二进制表示，最后对这个二进制数求补就可以了。所谓求补，即将所有的1变为0，而将所有的0变为1。所以，若要将-5用二进制数表示，先加1，得到-4，然后求其绝对值，再将该值用二进制数表示，即00000100，最后求补，就得到了-5的二进制数：11111011。

译者注：原书中使用ones complement和twos complement代表用二进制串表示负数的两种方案，本书中将其翻译为一类补数和二类补数。一类补数通过将某个负数对应的正数的二进制串的每一位直接求反得到，这种表示方法的问题在于0在该编码方案中有两种表示方法。二类补数编码方案弥补了这个缺陷。

将一个负数从二进制转换为十进制的方法是：首先对其所有位求补，然后将结果转换为十进制，再改变其符号，最后再减1就可以了。

使用二类补数表示数字时，用 n 位可存储的最大正数是 $2^{n-1}-1$ 。因此，若是8位数，则可存储的最大值是 2^7-1 ，即127。类似的，用 n 位可存储的最小负数是 -2^{n-1} 。因此，一个8位字节可表示的最小值为-128。（读者知道为什么用相同的方法表示，但最大值和最小值的绝对值却不同吗？）

在大多数现代处理器上，整数一般占据计算机内存中4个连续的字节，也就是32位。此时，可存储的最大正数是 $2^{31}-1$ ，即2, 147, 483, 647，可存储的最小负数是-2, 147, 483, 648。

在第4章，“变量，数据类型，算术表达式”中，读者学习了修饰符unsigned。我们知道，使用它可以有效地增加变量的表达范围，这是因为此时我们处理的仅仅是正数，不再需要将最左边的一位用作符号位。而使用这个扩展位，可将变量中存储的值增加1倍（译者注，即为原来的2倍）。此时，可用 n 位存储的最大数为 2^n-1 。在用32位表示整数的机器里，这意味着无符号整数的范围是0到4, 294, 967, 296。

位运算符

我们已学习了一些预备知识，接下来，我们进一步学习各种的位操作运算符。C语言提供的用以处理位操作的运算符如表12.1所示：

表12.1 位运算符

符号	操作
&	按位与
	按位或
^	按位异或
~	取反
<<	左移
>>	右移

表12.1列出的所有运算符，除过“取反”运算符 `~` 外，都是二元运算符，需要两个操作数。在C语言里，对于任意类型的整数，不管是short, long, long long，也不管是符号数或无符号数，都可以应用位操作，甚至对于字符，也可以应用位操作，但是对于浮点数，就不可以了。

按位与运算符 (&)

C语言里，当两个数据进行按位与运算时，我们就对它们的二进制位进行一位一位的比较。如果两个相应的二进制位都为1，则该位的结果为1；否则，结果为0。如果**b1**和**b2**表示两个操作数的相应的二进制位，则下面的真值表显示对它们进行与运算后的所有可能的结果：

b1	b2	b1 & b2
0	0	0
0	1	0
1	0	0
1	1	1

举个例子，如果定义**w1**和**w2**为短整型数，且其值分别为25和77，则下面语句将**w3**赋值为9：

```
w3 = w1 & w2;
```

如果把**w1**、**w2**和**w3**转化为对应的二进制数，我们就会很容易理解上面的运算结果。假定我们处理的是16位的短整型，则上面的运算可用下式说明：

```

w1  0000000000011001    25
w2  0000000001001101    & 77
-----
w3  000000000001001     9
    
```

如果回想一下逻辑与运算 **&&** 工作的方式（两个操作数都为true时结果为true），我们就会很容易记住按位与运算是如何工作的。但是，读者一定不要将这两个运算混淆了！逻辑与运算符 **&&** 用于逻辑表达式，会产生一个true或false的结果，但它的两个操作数并不进行按位与运算。

按位与运算经常应用于屏蔽一个数中的某些指定位。也就是说，我们可以用它方便地将一个数据的指定位保留，而将其它位清零。例如下面的语句：

```
w3 = w1 & 3;
```

将**w1**和3按位与的结果赋值给**w3**。这就相当于将**w3**里的除过最右边2位的所有位清零，而只保留**w1**最右边的两位并将其赋予**w3**。

像C语言所有的二元算术运算符一样，二进制位运算符也可以和赋值运算结合起来，只要在其后面加上等号即可。因此，语句：

```
word &= 15;
```

和下面语句具有相同的功能：

```
word = word & 15;
```

即将**word**中除过右边四位的其它位清零。

当在位运算表达式中使用到常量时，用八进制或十六进制表示该常量通常带来很多方便。到底采取何种方式表示，通常取决于被处理数据的长度。例如，在32位计算机上进行位运算时，因为32是4的倍数（一位十六进制所含的二进制位数刚好是4），所以一般采用十六进制表示。

程序12.1举例说明了二进制与运算。因为程序中我们处理的只是正数，所以将所有整数声明为unsigned int类型。

程序12.1 二进制与运算

```
// 演示二进制与运算的程序
#include <stdio.h>

int main (void)
{
    unsigned int word1 = 077u, word2 = 0150u, word3 = 0210u;

    printf ("%o ", word1 & word2);
    printf ("%o ", word1 & word1);
    printf ("%o ", word1 & word2 & word3);
    printf ("%o\n", word1 & 1);

    return 0;
}
```

程序12.1 输出

50 77 10 1

我们记得，C语言里，一个以0开头的整数常量表示的是八进制数（以8为基数）。因此，我们将三个无符号整数变量word1、word2和word3分别初始化为八进制数077、0150和0210。并且，我们在第4章已经讲过，将一个u或U置于一个整数常量后面，表示该数是一个无符号数。

第一个printf函数调用显示了word1和word2进行按位与运算的结果，即八进制数50。下面的算式描述了这个值是如何得到的：

```
Word1   ... 000 111 111    077
Word2   ... 001 101 000    &0150
-----
        ... 000 101 000    050
```

因为最左边的位为0，所以算式中只显示了数据最右边的9位。二进制数被3个一组地列在一起，这可以使我们可以更方便地进行二进制与八进制之间的互相转换。

第二个printf函数调用显示了word1和它自己进行按位与运算的结果，即八进制数77。通过定义我们知道，不管x的值为多少，它与自己进行按位与运算的结果还是它自己。

第三个printf函数调用显示word1、word2和word3一起进行按位与运算的结果。按位与运算的操作是这样的，不管表达式是 $a \& b \& c$ ， $(a \& b) \& c$ 还是 $a \& (b \& c)$ ，其结果都是相同的。但是为了记录方便，我们从左向右进行运算。留一个练习给读者：请读者自己验证一下显示的结果，即八进制数10，是word1、word2和word3一起进行按位与运算的正确结果。

最后一个printf函数调用具有提取 word1 最右边一位的效果。这实际上是判断一个整数是奇数还是偶数的另外一种方法，因为其最右边一位是1为奇数，是0则为偶数。因此，当执行下面语句时：

```
if ( word1 & 1 )
    ...
```

当word1为奇数时（因为与运算的结果为1），表达式为true（真）；当word1为偶数时（因为与运算的结果为0），表达式为false（假）。（注意，在使用一类补数表示数字的机器里，这种方法无法应用于负数）。

按位或运算符（|）

C语言里，对两个数进行按位或运算时，我们仍对它们的二进制位逐位进行比较。两个相应的二进制位只要有一个为1，则该位的结果为1。按位或运算的真值表如下：

b1	b2	b1 b2
0	0	0
0	1	1
1	0	1
1	1	1

因此，按照上面的规则，如果w1和w2为无符号整数，其值分别为0431和0152，我们对w1和w2进行按位或运算，结果将得到一个八进制数0573：

```

w1  ... 100 011 001      0431
w2  ... 001 101 010      0152
-----
      ... 101 111 011      0573
    
```

和前面指出的关于按位与运算类似，我们一定不要将按位或运算符（|）和逻辑或运算符（||）混淆了，后者用来决定两个逻辑值是否有一个为真。

我们常用按位或运算，将一个数据的某些位设定为1。例如，语句：

```
w1 = w1 | 07;
```

将w1的最右边的3位置为1，而不管操作前它们是何状态。当然了，我们也可以在语句中使用如下特殊的赋值运算符，达成和上面语句相同的效果：

```
word |= 07;
```

后面我们将会举例说明按位或运算符在实际程序中的运用。

按位异或运算符 (^)

按位异或操作符也被称作是XOR操作符。C语言里，对两个数进行按位异或运算时，如果两个相应的二进位有一个为1，但又不是全部为1，则该位的结果为1。按位异或运算的真值表如下：

b1	b2	b1 ^ b2
0	0	0
0	1	1
1	0	1
1	1	0

如果w1和w2为八进制整数，值分别为0536和0266，则我们对w1和w2进行按位异或运算，将得到一个八进制数0750，如下面算式所示：

```

w1  ... 101 011 110    0536
w2  ... 010 110 110    ^ 0266
-----
      ... 111 101 000    0750
    
```

异或运算符有一个有趣的特性，即任何一个数与自己异或后值为0。历史上，汇编程序常用这一技巧快速地将一个值设置为0或比较两个值看它们是否相等。在C语言里，我们不推荐使用这种方法，因为这样做既不节省时间，并且还很可能导致程序晦涩难读。

异或运算的另外一个有趣的应用是，它可以在不用额外存储单元的情况下高效地交换两个数值。我们知道，正常情况下要交换两个整数i1和i2的值，一般可用如下顺序的语句：

```

temp = i1;
i1 = i2;
i2 = temp;
    
```

而使用异或运算符，我们可以在不用临时变量的情况下交换两个值：

```
i1 ^= i2;
i2 ^= i1;
i1 ^= i2;
```

给大家留一个练习：验证前面的语句确实可以将两个值*i1*和*i2*交换。

取反运算符 (~)

取反运算符 \sim 是一个一元运算符，它的作用是简单地将它的操作数按位取反，即将1变0，将0变1。取反操作的真值表如下所示：

b1	$\sim b1$
0	1
1	0

如果*w1*是一个16位的短整数，值为八进制0122457，那么我们对它进行取反运算后将得到一个八进制值0055320：

```
w1  1  010  010  100  101  111      0122457
~w1  0  101  101  011  010  000      0055320
```

我们一定不要将“取反”运算符 (\sim)、算术运算中的减法运算符 (-) 以及逻辑非运算符 (!) 弄混淆了。如果我们定义*w1*为整数，其值为0，则 $\sim w1$ 的结果还是0。如果我们对*w1*进行取反运算，则所得结果的每一位将都是1，若作为符号数来看，其值为-1。最后，如果我们对*w1*进行逻辑非运算，因为*w1*是false (0)，所以结果为true (1)。

当我们不知道正在处理的操作数的准确长度时，取反运算符就很有用了。使用它可以使程序的可移植性更好，换句话说，就是程序运行时对具体计算机的依赖性很小，因此，可以更方便地在不同的机器上运行。例如：为了将一个整数*w1*的低位设置为0，我们只需用*w1*和一个仅最右边一位是0的数进行按位与运算即可。因此，像下面的一条C语言语句：

```
w1 &= 0xFFFFF0;
```

会在一个用32位表示整数的机器上工作得很好。

如果我们用下面语句代替前面的语句：

```
w1 &= ~1;
```


则在任一机器上都可以得到相同的结果。因为我们对1取反所得结果是，除过最右边一位为0外，其余位都为1，而这些1正是我们用来填充整数的长度所需要的（一个32的整数的最左边的31位）。

程序12.2总结了迄今为止所介绍的各种位运算符。在继续下面的内容之前，我们需要提到一个重要的内容，即各种运算符的优先级。二进制位运算符与、或和异或比任何一个算术运算符和关系运算符的优先级都低，但却高于逻辑运算符与（AND）和或（OR），其中，运算符“按位与”比“异或”优先级要高，而“异或”的优先级高于“或”。一元运算符“取反”比任何二元运算符的优先级都高。在附录A“C语言小结”中，我们将对这些运算符的优先级作一个总结。

程序12.2 举例说明位运算符

```

/* 说明位运算符的程序 */

#include <stdio.h>

int main (void)
{
    unsigned int w1 = 0525u, w2 = 0707u, w3 = 0122u;

    printf ("%o %o %o\n", w1 & w2, w1 | w2, w1 ^ w2);
    printf ("%o %o %o\n", ~w1, ~w2, ~w3);
    printf ("%o %o %o\n", w1 ^ w1, w1 & ~w2, w1 | w2 | w3);
    printf ("%o %o\n", w1 | w2 & w3, w1 | w2 & ~w3);
    printf ("%o %o\n", ~(~w1 & ~w2), ~(~w1 | ~w2));

    w1 ^= w2;
    w2 ^= w1;
    w1 ^= w2;
    printf ("w1 = %o, w2 = %o\n", w1, w2);

    return 0;
}
    
```

程序12.2 输出

```

505 727 222
37777777252 37777777070 37777777655
0 20 727
527 725
727 505
w1 = 707, w2 = 525
    
```

读者应该手工验证程序12.2中的每一个运算结果，以确保真正理解这些结果是如何得到的（注：程序运行在一个用32位表示整型的计算机上）。

在第四个printf函数调用中，我们必须记住一点，即运算符“按位与”比“按位或”的优先级要高，因为这一点会影响到表达式值的结果。

第五个printf函数调用显示了德摩根定律，即： $\sim(\sim a \& \sim b)$ 等价于 $a | b$ ， $\sim(\sim a | \sim b)$ 等价于 $a \& b$ 。程序接下来的语句验证了我们在“异或运算符”一节里所讨论的交换操作。

左移运算符 (<<)

我们对一个数据进行左移时，包含在其中的位逐一向左移动。与这一操作有关的是需要移动的空间（位数）。左移时高位的数据将会丢失，而相应地在低位补0。因此，如果w1为3，则表达式：

```
w1 = w1 << 1;
```

也可表示为：

```
w1 <<= 1;
```

将3左移一位，结果是将6赋值给w1：

```
w1      ... 000 011 03
w1 << 1 ... 000 110 06
```

运算符 << 左边的操作数是将要被移位的数，而右边的操作数是将要移动的位数。如果我们对w1多进行一次左移，而w1的值就会是八进制数014：

```
w1      ... 000 110 06
w1 << 1 ... 001 100 014
```

左移1位相当于该数乘以2。实际上，一些C语言编译器自动地用左移合适的位数以产生和乘以2的某次幂相同的效果，因为在大多数计算机上，移位操作比乘法操作要快得多。

在介绍完右移操作符之后，我们将一并举例给出左移操作的例子程序。

右移运算符 (>>)

像名字暗示的那样，右移运算符 >> 对一个数据进行向右移位。右移时低位的数据将会丢失。如果进行移位的是无符号数，则相应地将0移进数据的左边（译者注，即在左边补0）。换句话说，就是将0移进高位。

若进行移位的是一个符号数，那么左边移进的又是什么呢？这要依赖于要移位的数据的符号，还要依赖于计算机系统是如何执行这种操作的。如果符号位是0（意味着数据是正数），则不管你运行的是什么机器，都将0移进左边。然而，如果符号位是1，则在一些机器上将1移进左边，而在其它一些机器上将0移进左边。我们称前面类型的操作为算术右移，而后者为逻辑右移。

对于一个系统执行的到底是算术右移，还是逻辑右移，我们不应该进行任何假定。正因为这个原因，一个对符号数进行右移的程序在一个系统上也许工作正常，而在另一个系统则有可能失败。

如果w1是一个用32位表示的无符号整数，值为十六进制数F777EE22，则我们可以使用下面语句将w1右移一位：

```
w1 >>= 1;
```

结果是将w1的值设置为7BBBF711。

运算的算式如下：

```

w1      1111 0111 0111 0111 1110 1110 0010 0010 F777EE22
w1 >> 1 0111 1011 1011 1011 1111 0111 0001 0001 7BBBF711
    
```

如果我们声明w1为一个有符号整数，则在一些计算机上会出现相同的结果，而在其它一些执行算术右移的计算机上，结果将是FBBBF711。

需要指出的是，如果我们试图对一个数进行左移或右移，而要移动的位数大于或等于用于表示这个数据的位数，此时，C语言规范并未对结果进行定义。因此，在一个用32位表示整数的机器上，如果要将一个整数左移或右移32位或更多的位数，程序就无法保证会生产一个确定的结果。我们还应注意，如果移动的位数是负数，则结果也是不确定的。

移位函数

现在我们将一个实际的例子程序12.3中使用左移和右移运算符。一些计算机有一条单独的机器指令，如果移动的位数为正，则这条指令将数据左移；如果移动的位数为负，则将数据右移。现在，我们用C语言编写一个函数来模拟这种类型的操作。我们可以让函数有两个参数：一个是要进行移位的数据，一个是要移动的位数。如果要移动的位数为正，将数据向左移动指定的位数；否则，将数据向右移动指定的位数（即要移动的位数的绝对值）。

程序12.3 实现一个移位函数

```

// 一个移动无符号int的函数，如果为正数左移，为负数右移

#include <stdio.h>
    
```

程序12.3 续

```
unsigned int shift (unsigned int value, int n)
{
    if ( n > 0 ) // 左移
        value <<= n;
    else        // 右移
        value >>= -n;

    return value;
}

int main (void)
{
    unsigned int w1 = 0177777u, w2 = 0444u;
    unsigned int shift (unsigned int value, int n);

    printf ("%o\t%o\n", shift (w1, 5), w1 << 5);
    printf ("%o\t%o\n", shift (w1, -6), w1 >> 6);
    printf ("%o\t%o\n", shift (w2, 0), w2 >> 0);
    printf ("%o\n", shift (shift (w1, -3), 3));

    return 0;
}
```

程序12.3 输出

```
7777740 7777740
1777 1777
444 444
177770
```

程序12.3所示的移位函数将参数value声明为无符号数，这样将保证右移value时将会填充0；换句话说，执行一个逻辑右移。

如果要移动的位数n比0大，函数将value左移n位。如果n为负数（或0），函数对value进行右移，而移动的位数是n的绝对值。

主程序第一次调用shift函数将w1的值左移5位。随后我们调用printf函数显示调用shift 函数后的结果，也显示直接将w1左移5位后的结果，这样，我们就可以对它们进行比较了。

第二次调用 shift 函数将 w1 右移6位。从程序的输出的结果可以看出，函数返回的结果等同于直接将w1右移6位所得的结果。

第三次调用 `shift` 函数，我们指定要移动的位数是0。此时，`shift`函数执行一个右移0位的操作。像我们从程序的输出结果看到的那样，这个操作对数据没有影响。

最后一个`printf`函数调用使用了嵌套的`shift` 函数调用。里面的`shift`函数调用首先执行，并将`w1`右移3位，结果是0017777。这个值被传递给`shift`函数，函数又将该结果左移3位。像我们从程序输出结果看到的那样，这样做实际上就是将`w1`的低3位设置为0。（当然了，我们现已知道，通过简单地将`w1`和`~7`进行按位与运算也可以实现这个目的。）

旋转移位

接下来的例子程序中，我们综合运用本章所提到的一些位操作，编写了将数据进行左或右旋转的函数。除过当一个数据左旋转时将移出的高位再移回到低位外，旋转的过程和移位相似。当一个数据向右旋转时，则将移出的低位再移回到高位。因此，如果我们处理的是用32位表示的无符号整数，其值为十六进制80000000，则将它左旋转1位后得到十六进制数00000001。因为在左移1位操作中通常丢失的符号位上的1，又被带回到最低位上。

函数用到了两个参数：第一个是要被旋转的数据，第二个是目标要被旋转的位数。如果第二个参数为正，这个数据将向左旋转；否则，向右旋转。

我们可以采用一个相当简单的方法来实现旋转函数。例如，若`x`是`int`类型的整数，而`n`的范围是0 - （整数的位数-1），则为了计算将`x`左旋转`n`位的结果，我们可以先提取`x`最左边的`n`位，再将`x`左移`n`位，最后将提取的位放回到`x`的右边。右旋转函数也可用相似的算法实现。

程序12.4用前面描述的算法实现了`rotate`（旋转）函数。这个函数假定整数在计算机上是用32位表示的。但本章结尾的练习显示了一个方法，使得我们可以在不做假设的前提下，编写出这个函数。

程序12.4 实现一个旋转函数

```
// 旋转移位的函数

#include <stdio.h>

int main (void)
{
    unsigned int w1 = 0xabcdef00u, w2 = 0xfffff1122u;
    unsigned int rotate (unsigned int value, int n);
}
```

程序12.4 续

```
printf ("%x\n", rotate (w1, 8));
printf ("%x\n", rotate (w1, -16));
printf ("%x\n", rotate (w2, 4));
printf ("%x\n", rotate (w2, -2));
printf ("%x\n", rotate (w1, 0));
printf ("%x\n", rotate (w1, 44));

return 0;
}

// 将无符号整数旋转左移或右移的函数

unsigned int rotate (unsigned int value, int n)
{
    unsigned int result, bits;

    // 缩减移动的位数到指定范围
    if ( n > 0 )
        n = n % 32;
    else
        n = -(-n % 32);

    if ( n == 0 )
        result = value;
    else if ( n > 0 ) { // 左旋
        bits = value >> (32 - n);
        result = value << n | bits;
    }
    else { // 右旋
        n = -n;
        bits = value << (32 - n);
        result = value >> n | bits;
    }

    return result;
}
```

程序12.4 输出

```
cdef00ab
ef00abcd
fff1122f
```


程序12.4 输出 (续)

```

bffc448
abcdef00
def00abc
    
```

函数首先确保要移动的位数 n 是有效的。下面的代码：

```

if ( n > 0 )
    n = n % 32;
else
    n = -(-n % 32);
    
```

先检测 n 是否为正数，如果是，对 n 和一个整数的长度（本例中假定为32）进行求余，并将结果存储回 n 。运算后的 n 中存放要移动的位数，其范围是从0至31。如果 n 是负数，在求余操作之前先对 n 求绝对值，之所以要这样做是因为C语言里没有定义操作数是负数时求余运算如何进行，这种情况下，我们的机器可能会产生正数结果，也可能产生负数结果。通过先对数据求相反数，我们可以确保结果为正。接下来，我们再将一元减法运算符作用于结果，将它变为负数。最后结果的取值范围为-31至0。

如果调整后的要移动的位数为0，函数简单地将 $value$ 赋值给 $result$ ；否则，进行旋转操作。

函数将左旋转 n 位的操作分为三步执行：首先，提取 $value$ 最左边的 n 位，这是通过将 $value$ 向右移动（整数的长度- n ）位而完成的（本例中认为整数的长度为32）。然后，将 $value$ 左移 n 位。最后，将提取的位与 $value$ 最右边的相应位进行或运算并将结果存回数据。如果我们想要对 $value$ 进行右旋转，也可以用一个相似的过程来实现。

在主程序，我们要注意一下十六进制符号的用法。第一次`rotate`函数调用将 $w1$ 左旋转8位。像我们从程序输出结果所看到的那样，`rotate`函数返回一个十六进制数`cdef00ab`，实际上，它就是将`abcdef00`左旋转8位的结果。

第二次`rotate`函数调用将 $w1$ 右旋转16位。

随后的两个`rotate`函数调用对 $w2$ 做了同样的事情。接下来的`rotate`函数调用将 $w1$ 旋转0位。程序的输出结果验证了，这种情况下函数简单地将值原封不动地返回。

最后的`rotate`函数调用将 $w1$ 左旋转44位。这实际上是将数据左旋转12位（ $44 \% 32$ 是12）。

位域

运用前面讨论的位运算符，我们可以进行各种复杂的位操作。位操作经常也用于包含了几个信息的数据项。正像在一些计算机上，我们可以使用数据类型`short int`以节省内

存空间，我们也可以将信息打包存放到一个字节或一个字的其中几位。例如：用来表示布尔值真假条件的标记在计算机上可用一个单独的位表示。在大多数计算机上，如果我们声明一个字符变量用作标记，用到的存储空间是8位（一个字节）。一个_bool变量也有可能用到8位。这样，如果我们需要将许多标记存储到一个大表中，就会浪费大量的内存空间。

C语言里，我们可以用两种方法将信息打包到一起以更好地利用内存。一种方法是简单地使用一个int表示存储的数据，并用前面章节中描述的位运算符存取int中想要的位。另一种方法是用C语言中称为位域的结构定义一个打包信息的结构体。

为了说明第一种方法是如何应用的，我们假定要将5个数据打包存放到一个字里面，因为我们要在内存中维护一个很大的关于这些变量的表。假定这些数据中有3个是标记，分别是f1、f2和f3；第四个是一个叫做type的整数，取值范围是1到255；最后一个叫做index的整数，取值范围是0到100,000。

因为每个标记的true/false值需要用一位表示，所以存储标记f1、f2和f3仅需3位即可。整数type的取值范围是1至255，所以需要8位。最后，因整数index的取值范围是0至100,000，所以需要18位。从而，我们要存储5个数据，即f1、f2、f3、type和index，总共需要存储空间29位。所以，我们可以用下面的语句定义一个整型变量，用它包含这5个数值：

```
unsigned int packed_data;
```

我们就可以随意地将packed_data里面的位分配给这5个需要存储的数据。图12.1描述了一种分配的方法，这里假定packed_data的长度是32位。

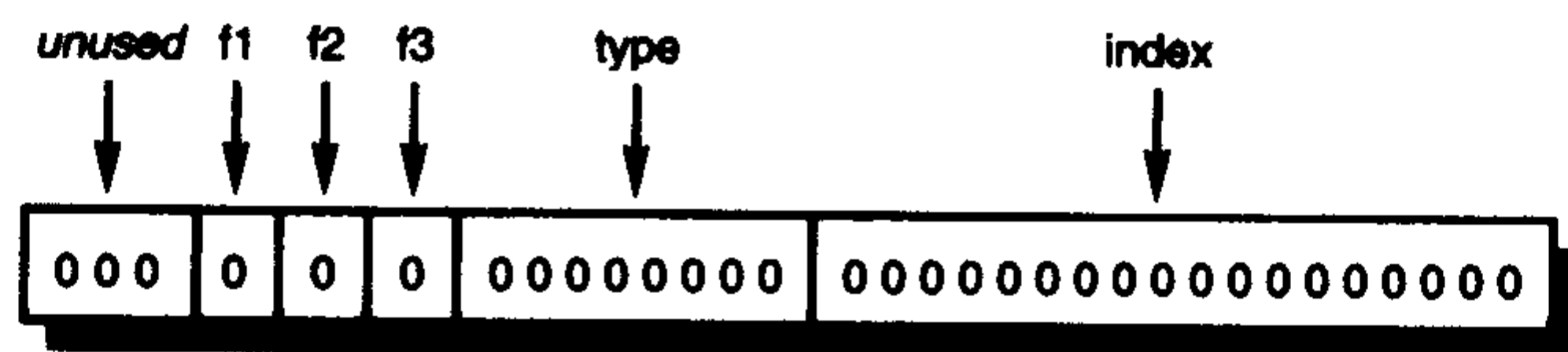


图 12.1 packed_data 里位域的分配

我们需要注意的一点是packed_data中有3位没有用到。现在通过对packed_data应用正确顺序的位操作，我们可以对整数的各个位域进行设置，也可取回其中的值。例如，我们可以通过将数值7向左移动合适的位数并进行或运算，来设置packed_data的type位域为7：

```
packed_data:
packed_data |= 7 << 18;
```


一般的，我们可以通过下面语句将type域设置为数值n，其中n的取值范围是0至255。

```
packed_data |= n << 18;
```

为了保证n是一个0至255之间的数，在它被移位之前，我们可以让它与0xff进行按位与运算。

当然，在我们知道type域是0的时候才可以使用前面的语句；否则，我们首先必须将它清0，这可以通过让它与一个在type域的位全0而在其它位全1的值（常称为掩码）进行位与操作来完成：

```
packed_data &= 0xfc03ffff;
```

为了避免必须计算出掩码的烦恼，也为了使得操作不依赖于整数的长度，我们可以用下面的语句将type域设置为0：

```
packed_data &= ~(0xff << 18);
```

通过对前面语句的总结，我们发现使用下面的语句，可以将packed_data的type域设置为n的低8位所包含的值，而不管之前这个域里面保存的是什么：

```
packed_data = (packed_data & ~(0xff << 18)) | ((n & 0xff) << 18);
```

在前面的代码中，有一些小括号是多余的，但使用它可以增加代码的可读性。

我们可以看出，前面用于完成将type域设置为指定值这一简单任务的表达式是很复杂的。但是从这些域中提取一个域的值还是比较容易的：先将指定域移到字的低位，然后再和一个合适的掩码进行位与运算即可。因此，我们要提取packed_data中的type域并将其赋值到变量n中，可以通过下面语句巧妙地完成：

```
n = (packed_data >> 18) & 0xff;
```

C语言给我们提供了一个处理位域的更方便的方法，这个方法在定义结构体时使用了一个特殊的语法，从而允许我们定义一个位域，并给它赋予一个名字。在C语言中，不管何时使用术语“位域”，指的都是这种方法。

为了定义与前面提到的packed_data作用相同的位域，我们首先定义一个如下所示的结构体packed_struct：

```
struct packed_struct
{
    unsigned int :3;
    unsigned int f1:1;
    unsigned int f2:1;
    unsigned int f3:1;
    unsigned int type:8;
    unsigned int index:18;
};
```

结构体packed_struct包含6个成员。第一个成员没有命名，:3指出这个成员占用3位。第二个成员为f1，是一个unsigned int，紧跟在成员名字之后的:1指出这个成员占用1位。成员f2和f3的定义类似于f1，各自占用1位。而成员type和index根据定义则分别占用8位和18位。

C语言编译器会自动地将前面定义的位域打包在一起存放。使用这种方法的好处是定义的packed_struct类型的变量的域现在可以像一个正常的结构体成员那样被引用。因此，如果我们如下声明一个变量packed_data:

```
struct packed_struct packed_data;
```

我们就可以用下面简单的语句方便地将packed_data的type域设置为7:

```
packed_data.type = 7;
```

或者，我们可以用相似的语句将这个域设置为值n:

```
packed_data.type = n;
```

对于后面一种情况，我们不需要担心赋予type域的n的值是否太大，因为只有n的低8位会被赋值给packed_data.type。

从一个位域中提取一个值也是自动处理的，因此，下面语句从packed_data中提取type域（系统自动将type域移到它所需要的低位）并将n赋值给它:

```
n = packed_data.type;
```

位域可以应用到正常的表达式中，且会被自动转换成整数。因此，下面语句是完全有效的:

```
i = packed_data.index / 5 + 1;
```

而像下面的语句，用来测试标记f2是真是假:

```
if ( packed_data.f2 )  
    ...
```

值得我们注意的一个事情是：在存储单元中位域的空间分配方向是无法保证的，有可能由左到右分配，也有可能由右向左分配。这一般不会带来任何问题，除非要处理的数据是由不同的程序产生的或者是使用不同机器的，也只有这时，我们才必须知道位域是如何分配的并进行合适的声明。我们可以定义结构体packed_struct如下:

```
struct packed_struct  
{  
    unsigned int index:18;  
    unsigned int type:8;  
    unsigned int f3:1;
```



```

    unsigned int f2:1;
    unsigned int f1:1;
    unsigned int :3;
};

```

上面的定义，可以在一个从右向左分配位域的机器上产生像图12.1描述的那样结果。这里我们提醒读者，不管是否包含位域成员，我们都不要对结构体成员的存储布局做出任何假设，

我们也可以在包含位域的结构体里包括正常的数据类型。因此，如果想要定义一个结构体，它包含一个int，一个char和两个1位的标记，则我们可以使用下面的定义：

```

struct table_entry
{
    int count;
    char c;
    unsigned int f1:1;
    unsigned int f2:1;
};

```

对于位域还有几点值得说明。我们只能将位域声明为整型数或者_bool类型。如果声明中只是使用了int，则它的实现将依赖于具体的编译器，该数有可能被解释为一个符号数，也可能是一个无符号数。为了安全起见，我们在声明位域的时候最好明确指定signed int 或者unsigned int。另外，我们无法使用一个位域数组，如flag:1[5]。最后，我们不能使用位域的地址，因此，显而易见，C语言里不存在一个“指向位域的指针”类型。

在结构体中定义的位域被打包到存储单元里。存储单元的长度取决于实现，一般情况下很可能是一个字（译者注：16位）。

C语言编译器不会为了试图优化存储空间而对位域的存储空间进行重新安排。

关于位域的最后一点说明涉及到一个特殊的情况，即长度为0的未命名域。它的作用是使下一个位域从下一个存储单元开始存放。

对于位运算的讨论到此就结束了。我们可以看到，C语言提供了有效、灵活的位处理功能，它提供了方便的位运算符用来进行按位与、按位或、异或、求反、左移和右移等位操作。我们还可以使用位域，将一个数据项存储到指定长度的位串中，并在不使用掩码和移位的情况下方便地设置或取回其值。

在第14章“进一步讨论数据类型”里，我们讨论了对不同整数类型的数据，例如对于一个unsigned long int和一个short int进行位运算的情况。

在进行下一章之前，请读者完成后面的练习，测试一下自己对C语言的位运算的理解情况。

练习

1. 输入并运行本章的四个程序，并将它们的输出与书中给出的输出进行比较。
2. 编写一个程序来确定读者使用的计算机执行的是算术右移还是逻辑右移。
3. 假设表达式 `~0` 产生一个所有位全1的整数，请读者编写一个函数 `int_size`，使其返回计算机上一个 `int` 所包含的位数。
4. 利用练习3的结果，修改12.4程序中的 `rotate` 函数，使得对于 `int` 的长度，我们不再需要做任何假设。
5. 写一个函数 `bit_test`，它有两个参数：一个无符号整数和一个位数 `n`。当该无符号数的第 `n` 位为1时，令函数返回1；当第 `n` 位为0时返回0（假定位0指的是整数最左边的一位）。同样编写一个叫做 `bit_set` 的函数，它也用到两个参数：一个无符号整数和一个位数 `n`，该函数将无符号整数的第 `n` 位打开（设置为1），并返回设置后的结果。
6. 写一个函数 `bitpat_search`，在一个 `unsigned int` 里查找指定的位串。该函数有三个参数且调用方式如下：

```
bitpat_search(source, pattern, n)
```

函数从最左边一位开始搜索整数 `source`，看 `pattern` 的最右边的 `n` 位是否在 `source` 中。如果找到了 `pattern`，函数返回 `pattern` 在 `source` 中开始的位数编号（假定最左边位的编号应该是0）。如果 `pattern` 没有找到，则函数返回-1。因此，下面的 `bitpat_search` 函数调用：

```
index = bitpat_search(0xelf4, 0x5, 3);
```

将在数字 `0xelf4`（=1110 0001 1111 0100，二进制）里查找，看3位的位串 `0x5`（=101二进制）是否存在。函数返回11，表示在 `source` 里找到了 `pattern`，且 `pattern` 在 `source` 里是从第12位开始的。

我们要确定函数对于 `int` 的长度不做任何的假设（参见本章的练习3）。

7. 写一个函数 `bitpat_get`，提取一个无符号整数中指定的位串。该函数有三个参数：第一个是 `unsigned int`，第二个参数是一个整数，指开始的位数，第三个整数参数指要提取的位数。按照惯例，最左边的位数是0。

函数的目的是从第一个参数中提取指定的位，并将结果返回。因此，调用：

```
bitpat_get(x, 0, 3)
```

从x中提取最左边的3位。调用：

```
bitpat_get(x, 3, 5)
```

从x中提取从第4位开始的5位。

8. 写一个函数bitpat_set，将一个特定值赋值给一个无符号数中的指定位串。函数有四个参数：第一个参数是一个指针，它指向一个unsigned int，我们要对这个无符号整数中的指定位串进行设置；第二个参数是一个unsigned int，这个无符号数就是将要设置的值；第三个参数为int，指定开始位（最左边的位数是0）；第四个参数为int，指定域的长度。因此，下面调用：

```
bitpat_set (&x, 0, 2, 5);
```

作用就是将x包含的从第3位开始的5位的值设置为0。类似的，调用：

```
bitpat_set (&x, 0x55u, 0, 8);
```

作用是：将x最左边8位的值设置为十六进制数55。

对于一个int的具体长度，我们不要做任何假设（参见本章的练习3）。

The Preprocessor

预处理器

本章我们将介绍C语言独有的一个特色——预处理器，很多其他高级语言都不具备这项功能。通过使用预处理器，我们可以更加轻松的编写、阅读、修改程序，并可以很容易的将程序移植到其他计算机系统中。我们也可以使用预处理器对于C语言进行字面上的修改，使其更加适合某种特定类型应用程序的开发，或者更加符合自己的程序编写风格。

预处理器实际上是编译器工作过程的一部分，预处理器能够识别出源程序中某些特定的语句，并对其进行处理。正如预处理器的名字所暗示的那样，它的工作实际上在真正的C语言语法分析之前进行。所有的预处理器语句都以一个#开始，该符号必须出现在一行的开始，前面不能再有任何非空白的字符。我们随后就会看到，预处理语句与一般的C语言语句有一些细小的区别。让我们先来看看#define语句。

#define 语句

#define语句（也称为宏定义语句）的一个主要用途是定义符号常量，也就是说，将一个常量赋给某个符号。下面的#define语句：

```
#define YES 1
```

定义了一个符号YES，并让其值等于1。在随后的程序中，任何用到常量1的地方，我们都可以使用符号YES来代替。当预处理器在随后的程序中遇到了该符号时，将会自动地使用常量1替换该符号。比如，我们可以像下面的语句一样使用符号YES：

```
gameOver = YES;
```

在书写这条语句的时候，我们使用了符号YES而不是1，因为我们知道该符号被定义为

常量1。上面这条语句的实际作用是将1赋值给变量gameOver。同样，下面的预处理语句：

```
#define NO 0
```

定义了符号NO，并且指定它代表常量0。因此在随后的程序中当我们使用该符号时，实际上是使用常量0。比如下面的语句：

```
gameOver = NO;
```

实际上把符号NO代表的值0赋值给变量gameOver。而下面的语句：

```
if( gameOver == NO )
    ...
```

实际上是将变量gameOver的值与符号NO所代表的值0进行比较。唯一一个不能使用预处理器所定义符号的地方是字符串，也就是说，下面的语句：

```
char *charPtr = "YES";
```

实际上将字符指针charPtr指向字符串"YES"，而不是"1"。

预处理器定义的符号并不是一个变量，我们不能给它赋值，除非该符号代表的是一个变量。无论何时，当预处理器遇到预定义的符号时，它都会使用定义该符号的语句中符号右面的字符串来替换该符号。这个替换过程和普通的文本编辑器的替换操作有些类似。

虽然定义符号的语法与C语言普通变量的赋值的语法比较类似，但是它们也有一些不同之处——我们不需要使用等号将符号的值和符号连接起来，另外在预定义符号语句的行尾也不需要分号。下面的程序13.1使用了我们前面预定义的符号YES和NO，程序13.1中的isEven函数接受一个整型参数，如果该参数为偶数则返回YES，否则的话返回NO。

程序13.1 演示#define语句

```
#include <stdio.h>

#define YES 1
#define NO 0

// 判断整数是否为偶数的函数

int isEven (int number)
{
    int answer;

    if ( number % 2 == 0 )
        answer = YES;
```

程序13.1 续

```
    else
        answer = NO;

    return answer;
}

int main (void)
{
    int isEven (int number);

    if ( isEven (17) == YES )
        printf ("yes ");
    else
        printf ("no ");

    if ( isEven (20) == YES )
        printf ("yes\n");
    else
        printf ("no\n");

    return 0;
}
```

程序13.1 输出

no yes

程序中的#define语句出现在程序开始，这并不是必须的，实际上，#define语句可以出现在程序的任何位置，只要在引用该语句定义的符号的那些语句前面即可。#define语句定义的符号与变量不同，它们没有局部作用域的概念，只要使用该语句定义了某个符号，无论该定义是在函数中还是函数外面，我们都可以后继的程序中使用这个符号。大多数程序员把这些语句放置在程序的开始处（或者放在一个包含文件¹中），这样我们就可以很容易在多个源文件中引用或者共享它们。

C语言程序员经常在程序中使用一个预定义符号NULL，该符号表示空指针²。

¹接下来我们将介绍如何编写包含文件。

²符号 NULL 通常在 C 语言的标准头文件<stddef.h>中定义。 我们下面将详细介绍包含文件。

如果我们按照下面的方式定义了符号NULL:

```
#define NULL 0
```

那么我们就可以使用该符号写出类似下面这样可读性更好的语句来:

```
while ( listPtr != NULL )
    ...
```

该语句中包含一个while循环, 循环继续执行的条件是指针listPtr不为空。

让我们再来看看使用预定义符号的另外一个例子。假定我们要编写三个函数, 分别用于计算给定半径的圆面积、圆周长和球体体积。这三个函数都要用到常数 π , 但是 π 的值不是很容易记住, 因此我们可以在程序的开始将其定义为一个符号, 然后就可以在程序中通过符号来引用其值了³。

程序13.2演示了如何在程序的开始定义常量符号, 并在程序中使用它。

程序13.2 预定义符号的更多例子

```

/* 计算给定半径的圆面积、圆周长和球体体积的函数 */

#include <stdio.h>

#define PI 3.141592654

double area (double r)
{
    return PI * r * r;
}

double circumference (double r)
{
    return 2.0 * PI * r;
}

double volume (double r)
{
    return 4.0 / 3.0 * PI * r * r * r;
}
    
```

³在C语言的标准头文件<math.h>中已经定义了符号M_PI, 其值等于 π 。我们可以在程序中直接包含该头文件来引用 π 的值。

程序13.2 续

```
int main (void)
{
    double area (double r), circumference (double r),
           volume (double r);

    printf ("radius = 1: %.4f %.4f %.4f\n",
           area(1.0), circumference(1.0), volume(1.0));

    printf ("radius = 4.98: %.4f %.4f %.4f\n",
           area(4.98), circumference(4.98), volume(4.98));

    return 0;
}
```

程序13.2 输出

```
radius = 1: 3.1416 6.2832 4.1888
radius = 4.98: 77.9128 31.2903 517.3403
```

在程序的开始，我们定义符号PI的值等于3.141592654。随后我们在计算面积、周长和体积的函数中使用该符号来代表 π 。C语言编译器将会自动将符号PI替换为合适的数值。

通过在程序中使用预定义符号代表特定的数值，我们可以避免在每次使用到该数值的时候都要回忆起其准确的值。另外，如果我们需要改变该数值的话（比如我们记错了圆周率），我们只需要修改程序中使用#define语句的地方就可以了。如果不使用#define语句的话，我们就必须在程序中查找每一个使用到该常数的地方，并对其进行修改。

读者可能已经注意到了，迄今为止，我们定义的所有符号都是由大写字母组成的。这样我们可以很容易将预定义符号和普通变量区分开来。部分程序员在程序中坚持使用这种命名规则，另外一些程序员则在所有的预定义符号名字前面加上字母k，而不是使用全大写的符号名称，比如kMaximuValues和kSignificantDigits。

程序的可扩展性

通过使用预定义符号，我们也可以增加程序的可扩展性。比如，如果我们在程序中用到了数组，那么在定义的时候我们需要给出数组的长度（无论是显式指定还是通过初始化

列表隐式指定)。在随后的程序中,我们还会经常使用该数组的长度值。例如,假定我们在程序中以如下方式定义了数组dataValues:

```
float dataValues[1000];
```

那么很有可能随后的程序中将会出现如下的语句:

```
for ( int i=0; i<1000; ++i)
    ...
```

在上面的语句中,我们使用数组的长度1000作为循环的上限。另外,下面的语句使用数组的长度来判断某个下标是否超出了数组的范围:

```
if( index > 999 )
    ...
```

现在假定我们要将数组dataValues的长度从1000改为2000,这样一来,在程序中所有用到数组长度的地方都需要进行修改。这将是一件很麻烦的事情。

一个更好的处理数组长度的办法是使用预定义符号,这也使得程序具有更好的可扩展性。对于上面的例子,我们使用如下的#define语句,将数组的长度用符号MAXIMUM_DATAVALUES代表:

```
#define MAXIMUM_DATAVALUES 1000
```

这样一来,我们就可以使用如下的语句定义数组dataValues了:

```
float dataValues[MAXIMUM_DATAVALUES];
```

在随后的程序中,所有要用到数组长度的地方,我们都可以使用符号MAXIMUM_DATAVALUES来代替。比如,如果要使用循环遍历数组的元素,我们可以使用下面的语句:

```
for ( i = 0; i < MAXIMUM_DATAVALUES; ++i )
    ...
```

如果要检查某个下标的值是否超出了范围,则可以使用如下的语句:

```
if ( index > MAXIMUM_DATAVALUES - 1 )
    ...
```

使用预定义符号的好处是我们可以很方便的将数组的长度从1000修改为2000——只需要将#define语句修改为下面的形式即可。

```
#define MAXIMUM_DATAVALUES 2000
```

只要我们在程序中任何使用数组长度的地方都坚持使用了符号MAXIMUM_DATAVALUES的话,那么这将是唯一需要修改的地方。

程序的可移植性

使用预定义符号可以帮助我们增加程序的可移植性,也就是将程序从一种计算机系统移植到另外一种计算机系统上。在编写程序的时候,有时我们会遇到一些与特定的计算机系统相关的常量,比如某个特定的计算机地址、文件名,或者是一个内存字包含的位数。读者还记得我们前面的程序12.4中编写的rotate函数,该函数就假定一个无符号int类型的变量,其大小是32位。

如果我们在一个无符号int类型的变量包含64位的计算机上运行程序12.4,rotate函数就会运行失败⁴。因此,如果我们在程序中必须使用与特定计算机系统相关的值时,我们最好把这些值与程序的其他部分隔离开来。#define语句可以很好的完成这项工作。下面是rotate函数的一个新版本,这个版本的代码可以很容易移植到具有不同int类型变量大小的计算机系统上去。请看下面的代码:

```
#include <stdio.h>

#define kIntSize 32 // *** 机器相关 !!! ***

// 将整数旋转左移或右移的函数

unsigned int rotate (unsigned int value, int n)
{
    unsigned int result, bits;

    /* 缩减移动的位数到指定范围 */
    if ( n > 0 )
        n = n % kIntSize;
    else
        n = -(-n % kIntSize);

    if ( n == 0 )
        result = value;
```

⁴我们当然也可以让rotate函数自己判断int类型的变量包含的位数,这样我们的程序就完全和具体的计算机系统无关了。请参考第12章“位操作”的练习3和练习4。


```
else if ( n > 0 ) /* 旋转左移 */
{
    bits = value >> (kIntSize - n);
    result = value << n | bits;
}
else /* 旋转右移 */
{
    n = -n;
    bits = value << (kIntSize - n) ;
    result = value >> n | bits;
}

return result;
}
```

预定义符号的高级形式

一个#define语句中不仅仅可以包含常量，也可以包含表达式，随后我们还会看到，实际上预定义符号语句几乎可以包含任何东西！

下面的#define定义了符号TWO_PI，该符号等价于2.0与圆周率的乘积。

```
#define TWO_PI 2.0 * 3.141592654
```

随后我们就可以在程序中任何用到2.0与圆周率乘积的地方使用该符号。比如前面计算周长的函数中，我们就可以该符号。如下面的语句所示：

```
return TWO_PI * r;
```

实际上，当C语言预处理器在源程序中遇到某个预定义符号的时候，预处理器将使用#define语句该符号后面的文本完全替换该符号，因此，C语言预处理器在处理上面的语句时，将使用2.0 * 3.141592654替代符号TWO_PI。

在了解了C语言预处理器对预定义符号的处理实际上是进行文本替换这一事实之后，我们就能明白为什么我们在#define语句的结尾通常不使用分号。如果使用分号的话，则分号也会被C语言编译器当作预定义符号的一部分替换到实际的语句中，比如，如果有如下的宏定义语句：

```
#define PI 3.141592654;
```

并在下面的语句中使用它：

```
return 2.0 * PI * r;
```

那么C语言编译器的预处理器将会使用3.141592654;替换符号PI, 最终编译器将会得到如下的语句:

```
return 2.0 * 3.141592654; * r;
```

最终我们将得到含有语法错误的语句。

预定义符号的内容并不一定要求是合法的C语言表达式, 只要C语言预处理器完成替换之后得到的语句是合法的就可以了。比如下面的宏定义语句就是一个合法的语句:

```
#define LEFT_SHIFT_8 << 8
```

虽然在符号LEFT_SHIFT_8的右面并不是一个完整的C语言表达式。我们可以像下面的语句这样使用该符号:

```
x = y LEFT_SHIFT_8;
```

该语句的效果是将y左移8位, 并将结果赋与变量x。下面的宏定义语句可能更有用一些:

```
#define AND &&  
#define OR ||
```

使用这些预定义符号, 我们可以写出如下的语句:

```
if ( x > 0 AND x < 10 )  
...
```

或者:

```
if ( y == 0 OR y == value )  
...
```

我们甚至可以定义如下的逻辑相等判断操作符符号EQUALS:

```
#define EQUALS ==
```

这样我们就可以写出如下的语句:

```
if ( y EQUALS 0 OR y EQUALS value )  
...
```

使用该符号, 我们可以避免在应该使用逻辑相等判断操作符(两个等号)的地方, 却错误地使用赋值操作符(一个等号), 并增加程序的可读性。

虽然上面的宏定义语句展示了预定义符号的多种用法, 但是读者要注意, 采用宏定义语句对于语言本身进行这类的修改通常是不好的, 因为对于不了解具体符号定义形式的程序员来说, 这些语句要比原始形式难以理解。

对于宏定义语句来说, 更有趣的是我们可以在一个语句中引用另外一个语句中已经定义的符号。比如, 下面的两条宏定义语句是完全合法的。

```
#define PI 3.141592654  
#define TWO_PI 2.0 * PI
```


在上面的语句中，TWO_PI的定义引用了符号PI，而符号PI本身被定义为3.141592654。

上面两个宏定义语句的顺序可以颠倒过来，如下所示：

```
#define TWO_PI 2.0 * PI
#define PI 3.141592654
```

只要我们在程序中使用某个预定义符号的时候，该符号以及该符号引用的所有其他符号都已经被定义就可以了。

在程序中恰当的运用预定义符号常常可以减少书写程序注释的必要。比如下面的语句：

```
if ( year % 4 == 0 && year % 100 != 0 || year % 400 == 0 )
...
```

我们知道该语句用于判断某个年份是否是闰年。下面请读者看看下面的宏定义语句和if语句：

```
#define IS_LEAP_YEAR year % 4 == 0 && year % 100 != 0 \
    || year % 400 == 0
...
if ( IS_LEAP_YEAR )
...
```

一般来说，一条宏定义语句占据源程序中的一行。如果宏定义语句需要跨两行，那么我们必须在第一行的结尾处放置一个反斜线\。预处理器把这个反斜线理解为续行标志，同时将下一行的内容也作为宏定义语句的一部分。反斜线本身将被预处理器忽略。对于需要占据多行的宏定义语句，上面的规则同样适用：我们需要在除了最后一行以外的每一行结尾放置反斜线。

现在我们的if语句看上去要容易理解多了。其他人不需要注释语句就可以明白程序的意思。在上面的例子中，符号IS_LEAP_YEAR的作用和一个函数类似，我们同样可以定义一个名为is_leap_year的函数来增加程序的可读性。在这里使用宏定义语句还是函数完全由程序作者自行决定。当然，如果使用函数的话，程序看上去可以更灵活一些，因为函数本身可以接受一个参数，使用函数可以检验任何变量代表的年份是否是闰年，而在前面的宏定义语句中，我们只能检验变量year的值是否是闰年。实际上，宏定义语句定义的符号也可以带有参数，请看下面的小节。

带有参数的宏定义语句

我们也可以定义接受参数的符号IS_LEAP_YEAR，如下所示：

```
#define IS_LEAP_YEAR(y) y % 4 == 0 && y % 100 != 0 \
    || y % 400 == 0
```

带有参数的宏定义语句与函数不同的一点是：我们不需要声明参数`y`的类型，因为预处理器实际上只是进行文本替换，它并不关心参数的类型。

定义带有参数的符号时，符号名称和参数列表的左括号之间不允许有空格。

根据前面的宏定义语句，我们可以使用下面的语句判断`year`是否代表一个闰年：

```
if ( IS_LEAP_YEAR (year) )  
...
```

或者使用下面的语句判断变量`next_year`是否是一个闰年：

```
if ( IS_LEAP_YEAR (next_year) )  
...
```

在处理上面的语句时，C语言预处理器将会使用`IS_LEAP_YEAR`的定义直接替换`if`语句中的符号，同时将宏定义语句中任何出现变量`y`的地方都使用`next_year`替代。因此，最终C语言编译器看到的源代码将是如下的样子：

```
if ( next_year % 4 == 0 && next_year % 100 != 0 \  
    || next_year % 400 == 0 )  
...
```

在C语言中，使用宏定义语句定义的符号常常被称作是宏，特别是那些带有一个或者多个参数的符号。在C语言中使用宏而不是函数有一个好处，那就是宏不需要关心参数类型。比如我们定义一个名为`SQUARE`的宏，该宏的作用是计算其参数的平方。其定义语句如下所示：

```
#define SQUARE(x) x * x
```

有了该宏定义之后，我们可以书写如下的语句：

```
y = SQUARE (v);
```

将变量`v`的平方赋值给变量`y`。使用宏的意义在于，无论变量`v`是什么类型——整型、浮点或者双精度，我们都可以使用该宏完成计算任务。但是如果我们将`SQUARE`定义为一个接受`int`型变量的函数，那么我们就无法使用该函数计算`double`类型变量的平方。如果读者希望在程序中使用宏，那么关于宏的下面一点也需要考虑：因为预处理器简单地对宏进行文本替换，因此最终的程序将会比使用函数的版本占用更多的内存空间。同时，因为函数涉及到调用和返回等额外附加操作，因此使用宏定义要比使用函数的程序快一些。

虽然前面的宏`SQUARE`很简单，但是其中却埋伏着使用宏最常见的一个错误。我们已经知道，下面的语句将变量`v`的平方赋值给变量`y`。

```
y = SQUARE (v);
```


但是下面的语句将会如何呢？

```
y = SQUARE (v + 1);
```

实际上，这条语句并不会像我们想象的那样，将 $v+1$ 的平方赋值给变量 y 。因为C语言预处理器仅仅进行文本替换，因此上面这条语句实际上进行的计算如下所示：

```
y = v + 1 * v + 1;
```

很明显，该语句的运算结果与我们的期望是不同的。为了正确的处理这种情况，我们需要在宏参数的周围加上小括号，最终的宏定义语句如下：

```
#define SQUARE(x) ( (x) * (x) )
```

虽然上面的宏定义语句看上去有些奇怪，但是它能够正确地处理前面的情况。读者一定要记住，C语言预处理器只是简单的对宏参数进行文本替换，因此要在参数周围使用括号。按照新的宏定义，我们的语句：

```
y = SQUARE (v + 1);
```

将被正确地展开为如下的形式：

```
y = ( (v + 1) * (v + 1) );
```

条件操作符在定义宏的时候非常有用，下面的语句定义了一个名为MAX的宏，该宏将返回它的两个参数中较大的那一个。

```
#define MAX(a,b) ( ((a) > (b)) ? (a) : (b) )
```

使用MAX宏，我们可以书写出如下的语句，该语句将 $x+y$ 和minValue中较大的值赋值给变量limit。

```
limit = MAX (x + y, minValue);
```

请读者注意，我们在整个宏的定义最外面又放置了一层小括号，这样即使对于下面的语句，我们也能保证最终的展开形式是正确的。

```
MAX (x, y) * 100
```

同样，我们也在参数外面放置了小括号，用以保证对于类似下面的语句，我们的宏能够正确展开。

```
MAX (x & y, z)
```

因为&操作符的优先级低于>操作符，因此如果没有括号的话，>操作就会在&操作之前执行，最终导致不正确的计算结果。使用括号避免了这一问题。

下面的宏检验某个字符是否是小写字母。

```
#define IS_LOWER_CASE(x) ( ((x) >= 'a') && ((x) <= 'z') )
```

使用该宏，我们可以编写如下的语句。

```
if ( IS_LOWER_CASE (c) )  
    ...
```

我们甚至可以利用这个宏定义一个新的宏，这个新的宏将任何小写字母转为大写字母，而保持其他字符不变。该宏的定义语句如下：

```
#define TO_UPPER(x) ( IS_LOWER_CASE (x) ? (x) - 'a' + 'A' : (x) )
```

下面的程序片段遍历一个字符串，将其中所有的小写字母都转换为大写字母⁵。

```
while ( *string != '\0' )
{
    *string = TO_UPPER (*string);
    ++string;
}
```

接受可变参数个数的宏

宏定义同样可以接受可变参数个数的参数列表。为了向预处理器表明我们的宏接受可变参数个数，我们可以在参数列表后面跟上三个点号 (...)。在随后的宏定义表达式中，我们可以使用特殊的符号 `__VA_ARGS__` 来代表具体的参数（译者注：请注意前后是双下划线）。让我们举一个例子。下面的语句定义了一个接受可变参数个数的宏：

```
#define debugPrintf(...) printf ("DEBUG: " __VA_ARGS__);
```

我们可以按照下面的方式使用该宏：

```
debugPrintf ("Hello world!\n");
```

也可以按照下面的方式使用：

```
debugPrintf ("i = %i, j = %i\n", i, j);
```

对于前一个语句，程序的输出如下：

```
DEBUG: Hello world!
```

对于第二个语句，如果 `i` 等于 100，`j` 等于 200，那么程序的输出如下：

```
DEBUG: i = 100, j = 200
```

下面我们来解释一下。对于第一个语句，预处理器的展开形式如下：

```
printf ("DEBUG: " "Hello world\n");
```

对于 C 语言编译器来说，任何两个相连的字符串常量将被自动合并，因此最终生成的 `printf` 语句如下所示：

```
printf ("DEBUG: Hello world\n");
```

⁵在 C 语言标准库中有很多库函数可以完成上述字符测试和转换的工作，比如 `islower` 和 `toupper` 等。更多的细节请读者参考附录 B “C 语言标准库”

#操作符

如果我们在宏定义的参数前面放置一个#，那么C语言的预处理器将使用该参数生成一个常数字符串。例如，如果我们定义宏str如下：

```
#define str(x) # x
```

如果在程序中有如下的语句：

```
str (testing)
```

那么最终的展开形式将会是：

```
"testing"
```

如果我们有如下的语句：

```
printf (str (Programming in C is fun.\n));
```

那么该语句的实际形式将会是：

```
printf ("Programming in C is fun.\n");
```

实际上，预处理器将在该参数左右加上双引号。如果参数内部本身还包含双引号或者反斜线的话，这些特殊符号在宏展开的过程中都将保留。因此，如下的语句：

```
str ("hello")
```

其展开形式将会是：

```
"\"hello\""
```

下面的宏定义语句展示了#操作符一个实际一些的用法：

```
#define printint(var) printf (#var " = %i\n", var)
```

该宏可以用来显示一个整型变量的名字及其值。例如，如果变量count的值是100，那么下面的语句：

```
printint (count);
```

将被展开为如下的形式：

```
printf ("count " = %i\n", count);
```

在C语言编译器合并的相邻的字符串之后，上述语句与下面的语句等价：

```
printf ("count = %i\n", count);
```

由此可见，#操作符可以让我们使用宏参数创建一个字符串。顺便提一下，#和宏参数之间的空格是可选的。

##操作符

在宏定义中使用这个操作符可以把两个符号连接起来。该操作符的前面或者后面可以是宏的参数。预处理器在展开宏的时候，将把该参数和##操作符前面（或者后面）的符号连接起来，以便创建一个新的符号。

下面我们举例来说明##操作符的使用方法。假定在程序中有名为x1、x2直到x100的100个变量，我们可以编写一个宏，该宏接受1到100之间的一个数字作为参数，并将对应变量的值打印出来。该宏的定义如下：

```
#define printx(n) printf("%i\n", x ## n)
```

语句中的字符串“x ## n”的将##操作符前面的记号和后面的记号连接起来（在这里就是x和参数n），合成一个新的记号。因此，下面的宏调用语句：

```
printx(20);
```

的实际展开形式如下：

```
printf("%i\n", x20);
```

我们也可以在printx宏中使用我们前面定义宏printint，以便将变量的名字和它的值一并打印出来。新的printx宏定义语句如下所示：

```
#define printx(n) printint(x ## n)
```

根据新的宏定义，下面的语句

```
printx(10);
```

首先被展开为如下的形式：

```
printint(x10);
```

随后再被展开为如下形式：

```
printf("x10" " = %i\n", x10);
```

最终C语言编译器处理的形式将是：

```
printf ("x10 = %i\n", x10);
```

#include 语句

当我们使用C语言编写程序一段时间以后，通常会积累下来一些我们在很多程序中都要使用的宏定义。通过使用C语言预处理器的#include语句，我们不需要在每一个源文件中都定义这些宏，而是把它们定义语句集中到一个单独的文件中，然后再在每一个需要这些宏定义的源程序中包含该文件。这类特殊文件的文件名通常以.h结尾，在C语言中被称为头文件或者包含文件。

假定我们要编写一组用于在各种度量值之间转换的程序。因为在每个程序中都要用到一组同样的常量，因此我们使用下面的宏对其进行定义：

```
#define INCHES_PER_CENTIMETER 0.394
#define CENTIMETERS_PER_INCH 1 / INCHES_PER_CENTIMETER

#define QUARTS_PER_LITER 1.057
#define LITERS_PER_QUART 1 / QUARTS_PER_LITER

#define OUNCES_PER_GRAM 0.035
#define GRAMS_PER_OUNCE 1 / OUNCES_PER_GRAM
```

假定我们将上述宏定义语句输入到一个名为metric.h的头文件中。随后，如果某个程序需要使用上述定义的符号中的任何一个，只需要在程序中加入下面一条语句即可：

```
#include "metric.h"
```

上面这条语句可以出现在程序的任何地方，只要在引用该文件中所定义符号的那些语句之前即可。但是通常人们把这条语句放置在源文件的开始。C语言的预处理器看到这条语句后，就在计算机系统中寻找相应的头文件，并将该头文件的内容全部拷贝到正在处理的源文件中#include语句的位置上。最终对于编译器来说看到的源程序就像程序员在文件中输入了包含文件的内容一样。

前面的#include语句中，文件名是用双引号包含起来的。这里的双引号告诉预处理器到某些特定的目录（通常是源文件所在的目录，但是实际搜索的目录和预处理器的设置有关）去寻找指定的头文件，如果在这些目录中没有找到指定的头文件，那么预处理器将在某些特定的系统目录中寻找该头文件。关于系统目录的含义，我们接下来就会说明。

如果我们使用一对尖括号<>将头文件的名字包围起来，比如下面的语句：

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

那么C语言预处理器就将到某些特定的系统目录寻找该文件。对于不同的计算机系统来说，系统目录的位置是不同的，比如在Unix操作系统中（还有Mac OS操作系统），系统目录是/usr/include，因此，预处理器将到该目录下寻找stdio.h文件，并最终使用头文件/usr/include/stdio.h。（译者注：即使源文件所在的当前目录下存在名为stdio.h的文件，预处理器也不会使用它。）

下面我们来实际演示头文件的使用方法。首先我们把前面的宏定义输入到文件metric.h中，然后输入并运行程序13.3。

程序13.3 使用#include语句

```
/* 使用#include的程序
   注意：本程序假设宏定义在metric.h的文件
*/
```

程序13.3 续

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

int main (void)
{
    float liters, gallons;

    printf ("*** Liters to Gallons ***\n\n");
    printf ("Enter the number of liters: ");
    scanf ("%f", &liters);

    gallons = liters * QUARTS_PER_LITER / 4.0;
    printf ("%g liters = %g gallons\n", liters, gallons);

    return 0;
}
```

程序13.3 输出

```
*** Liters to Gallons ***

Enter the number of liters: 55.75
55.75 liters = 14.73 gallons.
```

前面给出的例子虽然很简单——程序只是引用了头文件metric.h中定义的一个符号(QUARTS_PER_LITER)，但是它很好的说明了头文件的使用方法：所有我们保存在头文件metric.h中的符号定义，都可以在其他文件中通过适当的#include语句来使用。

使用头文件最大的好处是我们可以把符号定义集中在一个文件中，从而保证所有的源程序中使用该符号时都代表同样的值。而且，如果我们发现了头文件中的错误，我们只需要修改头文件一个地方，然后再编译所有的源文件就可以了，而不需要对于源文件进行逐个修改。

实际上我们可以在头文件中放置任何语句，而不仅仅是#define语句。在编程实践中，我们通常把与定义的宏、结构类型的定义、函数原型的声明以及全局变量的声明都放置到头文件中。

关于头文件最后要说明一点，那就是头文件的包含关系可以是嵌套的。一个头文件中也可以包含其他头文件。

系统头文件

C语言系统本身提供了很多标准的头文件，我们前面已经接触到了其中一些。比如<stddef.h>文件中包含宏定义NULL，我们经常将指针的值与该符号比较，以判断某个指针是否是空指针。还有头文件<math.h>中定义的符号M_PI，这个符号代表圆周率的近似值。

头文件<stdio.h>中包含了C语言标准库中关于输入/输出函数的信息。我们将在第16章“C语言的输入输出”中详细的介绍这些函数。如果我们在程序中用到了C语言的输入输出函数，就应该包含这个头文件。

<limits.h>和<float.h>也是两个很有用的头文件。第一个文件中通常包含与特定计算机系统相关的一些值，如各类字符类型与整数类型变量占用的存储空间大小。例如，该文件中定义了符号INT_MAX，用于代表int类型变量的最大值。还有符号ULONG_MAX代表unsigned long int类型变量的最大值等等。

头文件<float.h>中包含了浮点类型的信息，例如FLT_MAX代表float类型所能代表的最大数目，而FLT_DIG代表了float类型的变量最大能包含的数字位数。

C语言中还有很多其他系统标准头文件，它们包含其他一些系统标准库函数的信息。比如头文件<string.h>中包含着用于执行字符串操作（如拷贝、比较、合并等）的函数原型声明。

关于这些头文件的更多细节，请读者参阅附录B。

条件编译

C语言的预处理器提供了一种名为“条件编译”的特性。条件编译常常用于编写能够在多种计算机系统中编译并运行的代码。这个特性还常常用于关闭或者打开程序中某些特定的语句，例如那些跟踪程序流程或者输出变量内容的调试语句。

#ifdef、#endif、#else 和#ifndef 语句

在前面我们已经看到，通过使用下面的宏定义语句，我们可以使得函数rotate具有多种平台上的可移植性。

```
#define kIntSize 32
```

宏kIntSize帮助我们吧程序和具体平台上unsigned int类型的存储大小隔离开来。另外，我们也多次提醒读者，函数rotate实际上可以完全不依赖于具体平台，因为我们可以使用程序判断出unsigned int类型的大小。

不幸的是，在某些时刻，我们的程序必须依赖于某个特定的系统参数——例如文件名，因为在不同的操作系统中，文件的命名方式通常有所不同。

假定我们正在开发一个大程序，其中有很多依赖于具体软件、硬件系统的参数（在实践中这种依赖应该努力降至最低），那么，当我们把程序从一个平台移植到另外一个平台时，我们可能需要修改程序的很多地方。

通过使用条件编译技术，我们可以一次在程序中指定各种平台所需的参数，然后在不同的平台之间移植时，只需要简单的修改就可以了。例如下面的宏定义语句：

```
#ifdef UNIX
# define DATADIR "/uxnl/data"
#else
# define DATADIR "\usr\data"
#endif
```

如果我们定义了符号UNIX，那么DATADIR就被定义为"/uxnl/data"，否则的话，DATADIR就被定义为"\usr\data"。另外，通过上面的语句读者还可以看到，宏定义语句的#和define之间可以有一个或者多个空格。

#ifdef、#else和#endif语句的工作方式正如它们的名字所暗示的那样：如果#ifdef中指定的符号已经定义的话（无论是通过#define语句还是编译的时候在命令行上指定），那么C语言预处理器将对下面直到#else或者#endif的语句进行处理，否则的话，这些语句将被忽略（译者注：编译器将无法看到被忽略的语句）。

为了定义符号UNIX，我们可以使用下面的宏定义语句：

```
#define UNIX 1
```

或者下面的语句也可以：

```
#define UNIX
```

绝大多数编译器允许我们通过命令行参数定义宏，在gcc中，我们使用下面的命令行定义符号UNIX，以使得预处理器对于所有的#ifdef UNIX的求值结果为TRUE（命令行参数-D UNIX必须出现在文件名的前面）。这个办法允许我们在不修改源文件的情况下定义宏。

```
gcc -D UNIX program.c
```

gcc同样允许我们在命令行上定义宏的时候给出其具体的值，如下所示：

```
gcc -D GNUDIR=/c/gnustep program.c
```

上面的命令启动了gcc编译器，并将符号GNUDIR定义为/c/gnustep。

避免多次包含一个头文件

#ifndef语句的使用方法与#ifdef语句基本类似，不过它们的效果正好相反：如果#ifndef语句中的宏被定义了的话，后面的语句将被忽略而不是处理。我们经常使用该语句来避免在一个源文件中多次包含某个头文件。例如，如果我们需要某个头文件只被包含

一次的话，我们可以在该头文件中定义一个特殊的符号，并在随后的语句中检查这个符号，请看下面的例子：

```
#ifndef _MYSTDIO_H
#define _MYSTDIO_H
...
#endif /* _MYSTDIO_H */
```

假定我们在头文件mystdio.h中输入上面的宏定义语句。如果我们在某个源程序中使用如下的语句包含该头文件：

```
#include "mystdio.h"
```

那么mystdio.h文件中的#ifndef语句首先测试符号_MYSTDIO_H是否已经被定义，因为我们没有在源程序中定义这个符号，所以#ifndef和#endif之间的内容就被预处理器包含进来（我们假定该头文件中所有需要被包含的内容都放置在这两个语句之间）。这里请读者注意，被包含部分的第一条语句定义了符号_MYSTDIO_H，因此，如果我们在同一个源程序中试图再次包含mystdio.h文件时，因为第一行#ifndef语句的测试结果将为FALSE，最终mystdio.h文件中的内容将不会被包含。

C语言中的标准头文件就使用了上面的技术来避免多次包含。请读者试着打开其中一些头文件，自己查看并验证其内容。

#if 和#elif 语句

#if和#elif语句使得我们可以更灵活的控制条件编译。如果#if语句中的表达式求值结果不为0，那么从该语句一直到随后的#elif、#else或者#endif语句之间的语句都将被预处理器处理，否则的话，这些语句都被忽略。下面我们举一个条件编译的例子：假定符号OS的值为1时代表Macintosh OS，其值为2时代表Windows，其值为3时代表Linux，利用下面的宏定义语句结构，我们可以写出一组根据OS的不同值进行条件编译的语句，如下所示：

```
#if OS == 1 /* Mac OS */
...
#elif OS == 2 /* Windows */
...
#elif OS == 3 /* Linux */
...
#else
...
#endif
```

对于绝大多数编译器，我们可以使用前面讨论过的命令行选项-D给符号OS赋值。下面的语句：

```
gcc -D OS=2 program.c
```

将按照os的值为2对于程序进行编译，最终得到的结果是在Windows下可运行的程序。

在宏语句中我们可以使用一个特殊的操作符——defined来判断某个符号是否已经被定义了。因此，下面的语句：

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

和下面的语句：

```
#ifdef DEBUG
...
#endif
```

实际上是等价的。

而对于下面的语句：

```
#if defined (WINDOWS) || defined (WINDOWSNT)
# define BOOT_DRIVE "C:/"
#else
# define BOOT_DRIVE "D:/"
#endif
```

如果我们已经定义了符号WINDOWS或者WINDOWSNT中的任何一个，那么符号BOOT_DRIVE的值为"C:/", 否则的话其值为"D:/".

#undef 语句

在某些特殊情况下，我们需要取消一个预先定义的宏，#undef语句可以做到这一点。如果我们需要取消某个名字的宏，可以如下使用#undef语句：

```
#undef name
```

因此，下面的语句：

```
#undef WINDOWS_NT
```

将取消对于符号WINDOWS_NT的定义，随后的语句如#ifdef WINDOWS_NT或者#if defined(WINDOWS_NT)，其求值结果将为FALSE。

本章对于预处理器的介绍到这里就结束了。我们已经看到，如何使用预处理器使得程序更容易阅读、书写和修改。我们可以将一组宏定义放置到头文件中，然后在多个不同的源文件中共享这组宏定义。还有一些预处理语句本章并没有说明，读者如果有兴趣的话，可以参考附录A“C语言小结”。

在下一章中，我们将学习有关数据类型及其之间转换的更多知识，在继续前进之前，请首先完成本章后面的习题。

练习

1. 输入并运行本章的3个例子程序, 并将结果与书中给出的结果进行比较。记得输入程序13.3中的头文件。
2. 请读者查看自己机器上的C语言的系统头文件<stdio.h>, <limits.h>和<float.h> (在Unix系统上, 请在目录/usr/include中寻找上述文件)。
3. 定义宏MIN用于返回两个值之间较小的一个。书写一个程序测试该宏。
4. 定义宏MAX3用于返回三个值中最大的一个。书写一个程序测试该宏。
5. 定义宏SHIFT, 使其完成与程序12.3中函数shift相同的功能。
6. 定义宏IS_UPPER_CASE, 该宏检查某个字符是否是大写字母, 如果是的话返回非0值, 否则返回0。
7. 定义宏IS_ALPHABETIC, 该宏检查某个字符是否是字母, 如果是的话返回非0值, 否则返回0。请在该宏中使用本章正文中定义的宏IS_LOWER_CASE和练习6中定义的宏IS_UPPER_CASE。
8. 定义宏IS_DIGIT, 如果参数是字符'0'到'9'的话返回非0值。使用这个宏定义另外一个宏IS_SPECIAL, 用于判断某个字符是否是特殊字符。记得要在IS_SPECIAL中使用我们前面定义的宏IS_ALPHABETIC。
9. 定义ABSOLUTE_VALUE, 用于计算表达式的绝对值。注意最终的定义应该对如下的表达式:
ABSOLUTE_VALUE(x + delta)
也能够正确地给出结果。

10. 请看下面的宏定义:

```
#define printint(n) printf ("%i\n", x ## n)
```

请问, 下面的语句能够打印出变量x1到x100的值吗? 为什么?

```
for (i = 1; i < 100; ++i)  
    printx (i);
```

11. 验证如下三个C语言标准库函数isupper、isalpha和isdigit与我们前面练习6, 7, 8中定义的宏作用是相同的。为了使用这些库函数, 我们应该在程序中包含标准头文件<ctype.h>。

More on Data Types

进一步讨论数据类型

本章我们将介绍一种新的数据类型——枚举类型。另外我们还将学到typedef语句，使用typedef我们可以给C语言的基本数据类型（整数、字符）或者扩展数据类型（结构）起一个新名字。在本章的最后，我们将详细介绍C语言编译器对表达式求值时各种数据类型相互转换的规则。

枚举类型

在有些时候，我们需要定义某些变量，而且还需要指定该变量所能保存的值的范围。例如，假定我们定义了一个变量myColor，并且打算在其中存储三原色红、黄、蓝中的任意一种，而且除过这三种颜色之外，该变量中不能再保存其他值。枚举类型为我们提供了这种能力。

在C语言中，我们使用enum关键字来定义枚举类型。enum关键字处于定义语句的开始，后面再跟上该枚举类型的名字。在枚举类型名字的后面是一组该类型的变量所能取值的列表，这些值用大括号括起来。例如，下面的语句定义了枚举类型primaryColor：

```
enum primaryColor { red, yellow, blue };
```

primaryColor类型的变量中所保存的值必须是red、yellow和blue中的一个。如果试图将其他值赋给该变量，某些编译器将会发出警告，还有一些编译器则根本不检查赋给枚举变量的值。（译者注：至少Microsoft C++ compiler 13.10(VC7)和GCC 3.4.2对于这些值不进行检查，这在很大程度上抵消了枚举类型带来的好处）。

为了声明我们前面定义的枚举类型的变量；我们可以使用enum关键字加上枚举类型的名字，然后再写出变量名列表，如下所示：

```
enum primaryColor myColor, gregsColor;
```


上面这条语句定义了两个primaryColor类型的枚举变量myColor和gregsColor,对于这两个变量,我们赋给它们的值只能是red、yellow和blue中的一个。因此下面的语句:

```
myColor = red;
```

和

```
if ( gregsColor == yellow )  
    ...
```

都是合法的C语言语句。下面我们再给出一个枚举类型的例子。我们定义一个枚举类型enum month,用来表示一年中所有可能的月份。该枚举类型定义如下:

```
enum month { january, february, march, april, may, june,  
             july, august, september, october, november, december };
```

实际上,C语言编译器把枚举类型的变量当作整型常量处理。对于枚举类型的定义语句,编译器将0值赋给枚举值列表中的第一个值,将1赋给其中的第2个,依次类推。因此,按照前面的enum month定义,对于程序中的如下语句:

```
enum month thisMonth;  
...  
thisMonth = february;
```

C语言编译器实际上是将整型值1而不是名字february赋给变量thisMonth,因为february刚好是值列表中的第二个。

如果我们想赋给某个枚举值一个不同的整型数,可以在定义该枚举类型时指定这一点。在这个值之后的那些枚举值将从我们指定的数值之后递增编号。例如下面的语句:

```
enum direction { up, down, left = 10, right };
```

该语句定义了一个枚举类型direction。该类型的枚举变量有四个可能的值:up、down、left和right。因为up是所有值列表中的第一个,所以C语言编译器将0值赋给up;随后down被赋予值1;因为我们显式指定了left的值是10,因此right的值将在left之后递增,即11。

程序14.1给出了一个使用枚举类型的简单例子。我们将枚举类型month的第一个值january设置为1,这样数字1、2、3...就直接对应于月份january、february、March...。该程序首先读入一个代表月份的数字,然后通过switch语句判断用户输入的月份中究竟有多少天。因为C语言实际上把枚举类型看作整数,因此我们可以直接在case语句中使用枚举值。在程序中我们用变量days保存某个特定月份的天数,最后将其值打印出来。如果某个月份是2月,我们还需要进行一点特别的处理。

程序14.1 使用枚举类型

```
// 打印一个月中的每一天的程序

#include <stdio.h>

int main (void)
{
    enum month { january = 1, february, march, april, may, june,
        july, august, september, october, november, december };
    enum month aMonth;
    int days;
    printf ("Enter month number: ");
    scanf ("%i", &aMonth);
    switch (aMonth ) {
        case january:
        case march:
        case may:
        case july:
        case august:
        case october:
        case december:
            days = 31;
            break;
        case april:
        case june:
        case september:
        case november:
            days = 30;
            break;
        case february:
            days = 28;
            break;
        default:
            printf ("bad month number\n");
            days = 0;
            break;
    }
    if ( days != 0 )
        printf ("Number of days is %i\n", days);
    if ( aMonth == february )
        printf ("...or 29 if it's a leap year\n");
    return 0;
}
```


程序14.1 输出

```
Enter month number: 5  
Number of days is 31
```

程序14.1 输出（再次运行）

```
Enter month number: 2  
Number of days is 28  
...or 29 if it's a leap year
```

不同的枚举值可以使用相同的整数值，请看下面的例子：

```
enum switch { no=0, off=0, yes=1, on=1 };
```

在上面的语句中，枚举值no、off的值都是0，而yes和on的值都是1。

如果我们需要将其他整数值赋给某个枚举变量，可以使用类型转换操作符。例如，假定thisMonth是一个枚举变量，monthValue是一个整型变量，其值为6，则下面的语句将5赋给thisMonth。

```
thisMonth = (enum month) (monthValue - 1);
```

在程序中使用枚举变量的时候，我们应当把枚举类型看作一种新的数据类型，而不要将其看作是整数类型。枚举类型可以帮助我们将某个符号和整数值联系起来，如果我们坚持使用符号，那么当我们需要改变某个符号的值时，只需要修改定义该枚举类型的语句即可。如果我们在程序中混合使用整型数和枚举值，那么就无法得到上面的好处了。

与定义结构类似，我们也可以通过变形形式来定义枚举变量——即省略枚举类型的名字，而直接定义枚举变量。请看下面的语句：

```
enum { east, west, south, north } direction;
```

上面的语句定义了一个名为direction的枚举变量，该变量的值只能是east、west、south和north中的一个。

在作用域方面，枚举类型与结构和普通变量是类似的。如果我们在一个语句块中定义枚举类型，那么我们就只能在该语句块中使用它。与之相反，如果我们在所有的函数外面定义枚举类型，那么整个源文件中都可以使用该枚举类型。

最后还要提醒一点：枚举类型的值和同一作用域内的其他变量名或者枚举值的符号不能冲突。

typedef 语句

在C语言中，我们可以使用typedef语句给某个数据类型起一个不同的名字，请看下面的例子：

```
typedef int Counter;
```

上面的语句定义了一个新的符号Counter，该符号与C语言的数据类型int等价。我们随后可以声明Counter类型的变量，如下所示：

```
Counter j, n;
```

C语言编译器将上面的语句看作是普通的整型变量声明。使用typedef定义类型的好处主要是在声明变量的时候可以增加程序的可读性。使用了Counter类型之后，我们从声明语句中很容易看出，变量j和n的主要用作计数器。如果我们使用传统的int类型声明这两个变量，那么它们的作用就不是那么明显了。当然，我们还可以使用比Counter意义更清楚的类型名，这样声明语句就更容易被人理解了。

在很多情况下，我们都可以使用宏定义语句#define来代替typedef语句。比如前面的typedef语句就可以使用如下的#define语句来代替：

```
#define Counter int
```

但是，从本质上来讲，#define是由预处理器来处理的，而typedef语句是由编译器本身来处理的，因此typedef语句可以用来处理很多更加灵活的情况，比如给复合数据类型定义新的名字。请看下面的例子：

```
typedef char Linebuf [81];
```

上面的语句定义了一个新的数据类型LineBuf，它与包含81个字符的字符数组类型等价。我们随后可以像下面这样声明LineBuf类型的变量：

```
LineBuf text, inputLine;
```

上面的语句定义了两个包含81个字符的字符数组text和inputLine。该语句与下面的语句等价：

```
char text[81], inputLine[81];
```

这里请读者注意，对于LineBuf的定义，不存在对应的#define语句。

下面的typedef语句定义了类型StringPtr，该类型等价于C语言的字符指针：

```
typedef char *StringPtr;
```

随后我们就可以使用StringPtr定义新的变量，如下所示：

```
StringPtr buffer.
```

C语言编译器随后将变量buffer当作字符指针对待。

使用typedef语句为某个数据类型起一个新的名字的确切步骤如下：

1. 写出声明该数据类型变量的语句；
2. 将该语句中的变量名用新的类型名代替；
3. 在最前面加上typedef语句。这样就为原来的类型定义了新的类型名。

下面我们来举例说明上面的步骤。假定我们想要定义一个新的类型Date，该类型等价于包含三个整型成员变量month、day和year的结构。我们首先写出使用该结构声明变量的语句（采用结构声明的变形形式），然后将变量名（出现在分号之前）用Date代替，最后再在前面加上关键字typedef，最终得到了下面的语句：

```
typedef struct
{
    int month;
    int day;
    int year;
} Date;
```

在定义了新的数据类型Date之后，我们就可以使用该类型了，请看下面的语句。

```
Date birthdays[100];
```

该语句定义了Date类型的数组birthdays，该数组包含100个Date结构。

如果我们正在编写一个大的程序，该程序的代码分布在多个源文件中（请参见第15章“处理大型程序”），那么将所有的公共typedef语句放到一个单独的头文件中，然后在每个源文件中使用#include语句将该头文件包含进来将是一个不错的主意。

我们再举一个例子。假定我们正在开发一个图形包，其中的程序需要处理画线、画圆等各种操作。在开发的时候，我们很可能要大量涉及到坐标系方面的工作。为了简便起见，我们可以定义名为Point的数据类型，该类型等价于包含两个浮点成员x、y的结构。定义语句如下所示：

```
typedef struct
{
    float x;
    float y;
} Point;
```

我们随后就可以在图形包的开发中使用Point类型了。比如下面的语句定义两个Point类型的变量origin和currentPoint，并将origin的成员变量x和y初始化为0。

```
Point origin = { 0.0, 0.0 }, currentPoint;
```

下面是一个计算两点之间距离的函数：

```
#include <math.h>

double distance (Point p1, Point p2)
{
    double diffx, diffy;

    diffx = p1.x - p2.x;
    diffy = p1.y - p2.y;

    return sqrt (diffx * diffx + diffy * diffy);
}
```

因为我们的程序中使用了标准库函数`sqrt`，而该函数在系统头文件`math`中声明，因此我们在程序中包含了该系统头文件。

读者需要记住，`typedef`语句并没有定义新的数据类型，它只是给已经存在的数据类型起了一个新的名字。对于本小节前面定义的`Counter`类型的变量`j`和`n`，C语言编译器对它们的处理和普通的`int`类型变量是完全一样的。

数据类型转换

在第4章“变量、数据类型和算术表达式”中，我们已经知道，C语言在对表达式求值的时候，有时会进行隐式的数据类型转换。当时我们遇到的是`float`和`int`的例子，当表达式的两个操作数是浮点数和整型数的时候，整型数会被自动转化为浮点数，而整个表达式的结果也将是浮点数。

我们还知道，通过使用类型转换操作符，我们可以明确要求C语言进行类型转换操作。例如下面的语句：

```
average = (float) total / n;
```

在执行除法操作之前，变量`total`首先被转化为一个浮点数，这样可以保证程序执行的是浮点除法，从而避免丢失精度。

实际上，C语言编译器在对包含不同数据类型操作数的表达式进行求值的时候，有一套非常严格的类型转换规则。我们下面针对两个操作数的表达式，将类型转换的规则按步骤列出：

1. 如果两个操作数中有一个`long double`型，那么另外一个将被转换为`long double`型，表达式的结果也将是`long double`型；
2. 如果两个操作数中有一个`double`型，那么另外一个将被转换为`double`型，表达式的结果也将是`double`型；
3. 如果两个操作数中有一个`float`型，那么另外一个将被转换为`float`型，表达式的结果也将是`float`型；

4. 如果操作数中的任意一个是 `_Bool`、`char`、`short int`、`bit field` 或者枚举类型，那么这个操作数将被转换为 `int` 类型；
5. 如果两个操作数中有一个 `long long int` 型，那么另外一个将被转换为 `long long int` 型，表达式的结果也将是 `long long int` 型；
6. 如果两个操作数中有一个 `long int` 型，那么另外一个将被转换为 `long int` 型，表达式的结果也将是 `long int` 型；
7. 如果到达了转换规则的这一步，那么两个操作数都是 `int` 类型，最终表达式的结果也将是 `int` 类型。

上面的规则实际上只是完整规则的一个简化版本，实际的规则还要考虑无符号操作数的因素。如果读者想要了解完整的规则，可以参考附录A“C语言小结”。

C语言在表达式求值时，将按顺序执行上述规则，直到其中的一条规则符合实际的情况为止。

为了演示在表达式求值的时候上述规则如何起作用，我们下面给出一个具体的例子。假定 `f` 是一个 `float` 类型的变量，`i` 代表一个 `int` 类型的变量，`l` 是 `long int` 类型，而 `s` 是 `short int` 类型，我们将对下面的表达式求值：

`f * i + l / s`

第一步我们首先需要计算 `f*i` 的结果，也就是计算 `float` 类型与 `int` 类型变量的乘积。根据类型转换规则的第3步，因为 `f` 是一个 `float` 类型，因此另外一个操作数 `i` 也被转换为 `float` 类型，其运算结果也是一个 `float` 类型。

下面我们需要计算 `l / s`。因为 `s` 是一个 `short int` 类型，根据类型转换规则的第4步，它首先被转换为一个 `int` 类型。然后，根据规则的第6步，因为有一个操作符 (`/`) 是 `long int` 类型，因此另外一个操作数再被转换为 `long int` 型，除法运算的结果也将是 `long int` 类型，而除法结果中的小数部分将被忽略掉。

最后，因为第三步计算中的一个操作数是 `float` (`f*i` 的结果)，所以另外一个操作数也需要被转换为 `float`。我们将 `l/s` 结果转换为 `float`，然后把它和 `f*i` 的结果相加，最终所得的结果也是 `float` 类型。

我们还记得，使用类型转换操作符可以显式的对操作数的类型进行转换，使用该操作符我们可以控制表达式求值过程中的类型转换操作。

比如，如果我们希望保留 `l/s` 结果的小数部分，就可以使用类型转换操作符将两个操作数中的任何一个转换为 `float` 类型，这样整个除法就将按照浮点数除法的规则进行。转换的表达式如下：

`f * i + (float) l / s`

在上面的表达式中，因为类型转换操作符的优先级高于除法操作符，所以在进行除法操作之前，C语言编译器首先将 `l` 转换为一个浮点数。按照前面所叙述的转换规则，因为操

作数之一为浮点数，所以另外一个操作数s也将被转换为浮点数，最终得到的结果也将是float类型。

符号扩展

当一个有符号的整型数如signed int或者signed short int被转换为一个更大尺寸的整型数时，C语言执行的是有符号扩展，也就是说，使用原来数的符号位填充左面多出来的二进制位。这一点可以保证在对负数（如-5）进行扩展时，扩展后的结果仍然是一个负数（-5）。如果对无符号的整型数进行扩展，那么C语言将只是简单的用0填充左面多出来的位。

在某些计算机系统中（如Mac G4/G5，还有Pentium系列处理器），字符类型被当作是有符号的。这意味着当把一个字符转换为更大尺寸的数据类型如int时，执行的是有符号扩展。如果这个字符属于标准的ASCII字符集（其值属于0-127），那么这一点不会给我们带来麻烦，但是一旦我们使用了非标准的字符（其值属于128-255），那么扩展后的结果可能不是我们所需要的。比如在Mac计算机上的字符'\377'，在转换成为int类型时，其值将是-1，因为将该字符看作是8位整数时，它是一个负值（最高位为1）。

为了避免这个问题，我们可以使用unsigned char类型，将字符声明为无符号的，这样在扩展的时候C语言将总是使用0而不是符号位来填充多出的位。最终扩展所得的结果将保证总是大于0。对于典型的8位字节计算机，一个有符号字符变量的取值范围是-128到127，而一个无符号字符变量的取值范围是0到255。

如果我们需要有符号的字符变量，可以使用signed char显式声明它。这样，即使对于那些字符类型默认是无符号的计算机系统来说，也将执行有符号扩展。

参数转换

在本书中，我们编写的所有函数都使用了函数原型声明。我们在第8章“使用函数”中看到，这是一个非常明智的做法，因为这样一来，我们就可以把函数的具体定义放到源文件的任何位置，甚至是另外一个源文件中。另外，由于编译器已经知道函数的参数类型，因此在处理函数调用语句的时候，编译器可以自动将我们给出的参数转换为函数所需要的类型。实际上，C语言编译器只有预先看到了函数的定义或者原型声明，它才能够知道该函数所需的参数类型以及返回值的类型。

如果编译器还没有看到函数的定义或者原型声明就遇到了函数调用语句，那么它将假定该函数的返回值为int，同时编译器还将对参数类型也进行推测，如果调用语句中给出的参数类型是_Bool、char或者short，那么编译器假定函数接收的参数类型是int，并进行相应的转换，如果调用语句中给出的参数类型是float，那么编译器将其转换为double。

例如，如果编译器遇到下面的语句：

```
float x;  
...  
y = absoluteValue (x);
```

并且在此之前还没有看到函数absoluteValue的原型声明或者定义的话，那么编译器将把x转换为double类型，然后将转换的结果传递给absoluteValue函数，同时，编译器还假定该函数的返回值为int。

如果absoluteValue函数的实际定义如下，那么我们就有麻烦了。

```
float absoluteValue (float x)  
{  
    if ( x < 0.0 )  
        x = -x;  
    return x;  
}
```

该函数实际上需要一个float类型的参数，其返回值也是float类型。在这种情况下，编译器最终产生的机器代码将是不正确的。

读者一定要记住，在调用函数之前，应该包含该函数的原型声明，这样就可以防止编译器对于该函数的返回值和参数类型做出错误的假定。

到这里为止，我们完成了本章的学习。下一章我们将介绍如何将一个程序分为多个源文件。在开始第15章的学习之前，请读者先完成下面的习题。

练习

1. 使用typedef定义一个函数指针类型，该指针类型指向一个不需要参数，并且返回值为int的函数。如果感到有困难的话，请参阅第11章“指针”中如何定义函数指针的介绍。
2. 编写一个名为monthName的函数，该函数接收一个enum month类型的参数（该类型的定义如本章正文中所示），然后返回一个指向该月份名字的字符指针。通过调用这个函数，我们可以使用如下的语句显示enum month类型的枚举变量所代表月份的名字：

```
printf ("%s\n", monthName (aMonth));
```

3. 给定下面的变量声明:

```
float f = 1.00;  
short int i = 100;  
long int l = 500L;  
double d = 15.00;
```

按照本章给出的类型转换规则的7个步骤, 判断下面表达式结果的类型以及具体的值:

```
f + i  
l / d  
i / l + f  
l * i  
f / 2  
i / (d + f)  
l / (i * 2.0)  
l + i / (double) l
```


Working with Larger Programs

处理大型程序

本书中的所有程序都是非常短小和简单的。不幸的是，在实际工作中需要编写的程序通常既不短小，也不简单。本章将向读者介绍编写这类大型程序所需的一些技术。读者将会看到，C语言提供了很多特性用于支持编写大型程序。除此之外，我们还将简短的介绍几个工具，这些工具可以帮助我们更好的管理大型程序。

将程序分为多个文件

直到目前为止，我们遇到的所有程序都只有一个源文件，我们首先输入这个源程序——无论使用vim、emacs还是Windows操作系统的编辑软件，然后再编译这个源文件，最后运行可执行程序。除过系统函数如printf、scanf之外，程序用到的所有函数都包含在这个源文件中。另外，这些系统函数的原型声明头文件也被包含在这个单一的源文件中。单一的源文件对于小型程序（也就是语句数目小于100的程序）工作的很好，但是如果程序再大一些，单一源文件就有些应付不过来了。首先，随着程序中语句数目的增长，编辑修改和重新编译该程序的时间将会相应增加；另外大型程序通常需要多个程序员合作来完成，如果所有的程序员都同时修改同一个文件——甚至是该文件的私人版本，最终的结果都将是无法管理的。

C语言支持模块化编程的概念，也就是说，我们不必把一个程序的所有语句放置到单个源文件中。实际上，我们可以把程序的某个模块放置在一个源文件中，而把另外一个模块放置在其他的源文件中。这里模块的意思是指单个函数或者一组逻辑上关联的函数，我们下面将不加区别的同时使用模块和源文件两个名词。

如果读者正在使用基于图形界面的开发工具，比如Metrowerks的CodeWarrior，或者Microsoft Visual Studio，或者Apple的Xcode，那么同时管理多个源文件将是比较容易的。你只需要告诉该工具哪些源文件属于你的工程，那么工具就会自动替你处理其余所有的事情。在下面的章节里，我们将会介绍如何在没有这类工具（也被称为集成开发环境IDE）的时候，如何同时在多个源文件上工作。我们将在命令行上直接使用cc或者gcc之类的命令管理我们的源文件。

在命令行上编译多个源文件

假定我们已经将手头的程序划分为三个模块，并将第一个模块的语句输入到源文件mod1.c中，第二个模块的内容输入到mod2.c中，而主程序的内容被输入到main.c中。为了告诉编译器这三个源文件实际上属于同一个程序，我们可以在启动编译器的时候在命令行上给出这三个源文件的名称。例如，如果我们使用的编译器是gcc，那么可以在命令行上输入下面的命令：

```
$ gcc mod1.c mod2.c main.c -o dbtest
```

上面的命令将逐个编译三个源文件，并将其中的错误分别打印出来。比如，如果编译器给出了如下的错误信息：

```
mod2.c:10: mod2.c: In function 'foo':
mod2.c:10: error: 'i' undeclared (first use in this function)
mod2.c:10: error: (Each undeclared identifier is reported only once
mod2.c:10: error: for each function it appears in.)
```

上述错误信息意味着源文件mod2.c的第10行，也就是函数foo中，有一个语法错误。因为编译器没有打印出与mod1.c和main.c相关的出错信息，那么就意味着这两个模块中没有错误。

一般说来，如果编译器报告在某个模块中发现了错误，那么我们就需要编辑那个模块，修正其中的错误¹。在我们给出的例子中，因为只有mod2.c中有错误，因此我们需要编辑该文件以修正错误。在修改完成之后，我们可以重新启动编译器，命令如下：

```
$ gcc mod1.c mod2.c main.c -o dbtest
$
```

因为这次编译器没有报告任何错误信息，所以编译成功完成，最终生成的可执行文件名为dbtest。

一般说来，在编译多个源文件时，编译器将为每一个源文件生成临时目标文件。对于模块mod1.c来说，编译器生成的临时目标文件的文件名将是mod1.o（绝大多数Windows操作系统的编译器也是如此，只不过它们生成的目标文件其后缀名通常是obj而不是o）。在编译过程完成之后，编译器将会自动删除这些临时目标文件。

¹ 错误也可能位于该模块所包含的一个头文件之中。这个时候，我们就需要编辑修改该头文件而不是模块的源文件。

某些编译器（例如传统上Unix操作系统的标准C编译器），在一次编译多个文件的时候，会在编译结束后保留这些临时目标文件。利用这个特性，我们可以在只修改少数几个模块的时候，加快编译过程。比如，在前面的例子中，因为main.c和mod1.c中没有错误，因此在编译完成之后，它们对应的目标文件main.o和mod1.o都将保留下来。我们可以在调用编译器的命令行上使用这些目标文件而不是源文件，通过将目标文件传递给编译器，我们告诉编译器应该使用上次编译生成的目标文件，而不需要再从源文件开始编译这些模块。综上所述，这次我们使用的命令如下：

```
$ cc mod1.o mod2.c main.o -o dbtest
```

采用这种方式，我们既不需要编辑修改那些没有错误的源文件（mod1.c和main.c），编译器也不需要重新编译这些文件。

如果读者使用的编译器在编译完成后自动删除临时目标文件，我们也可以执行增量编译，这需要我们在编译文件的时候给编译器传递命令行选项-c。这个选项告诉编译器不要执行连接步骤（也就是说不产生最终的可执行程序），而保留其产生的临时目标文件，因此，下面的命令：

```
$ gcc -c mod2.c
```

将编译文件mod2.c，然后将生成的临时结果保存到文件mod2.o中。

总体说来，对于拥有三个模块的程序dbtest，我们可以使用如下一组命令对其进行增量编译：

```
$ gcc -c mod1.c           从mod1.c生成mod1.o
$ gcc -c mod2.c           从mod2.c生成mod2.o
$ gcc -c main.c           从main.c生成main.o
$ gcc mod1.o mod2.o main.o -o dbtest 最终生成可执行程序
```

上面这组命令对于三个源文件分别进行编译，每个文件的编译过程都没有错误。如果有错误的话，我们可以单独编辑和修正该源文件，然后重新执行编译命令。在所有的模块都编译完成之后，最后一条命令：

```
$ gcc mod1.o mod2.o main.o -o dbtest
```

将所有的目标文件连接起来，生成最终的可执行程序。请读者注意，在最后一条命令中没有使用任何源文件。

如果我们把上述技术应用到包含有更多模块的程序中，我们就会对该技术如何加快大型程序的开发看得更清楚一些。比如下面的命令：

```
$ gcc -c legal.c 编译legal.c，将输出保存到legal.o中
$ gcc legal.o makemove.o exec.o enumerator.o evaluator.o display.o -o superchess
```

该命令生成了一个包含有6个模块的程序，该程序中只有源文件legal.c需要重新被编译，而其他模块只是连接已经编译好的目标文件。

在本章的最后一节中读者可以看到，我们还可以使用一个名为make的工具将上述增量编译的过程自动化。对于本章开始提到的那些IDE工具来说，它们总是知道哪些源文件需要重新编译，从而在生成程序的过程中只编译那些需要编译的文件。

模块之间的通信

有几种方法可以帮助我们在多个模块之间传递信息。例如，如果一个源文件中的程序需要调用另外一个源文件中定义的函数，我们可以像通常那样写出调用语句并传递相应的参数。但是，在调用函数的那个模块中，我们必须包含被调用函数的原型声明，这样编译器才能知道该函数的参数以及返回值信息。在第14章“关于数据类型的进一步讨论”中，我们知道，如果没有函数原型的话，编译器将假定函数的返回值是int，而short和char类型的参数将被转换为int，float类型的参数将被转换为double，这很可能是不正确的。

在使用多个模块的时候，有一点概念非常重要，那就是虽然我们可以在命令行上同时让编译器编译多个源文件，但是对于编译器来说，它将独立的完成每个源文件的编译。也就是说，有关结构的定义形式、函数的参数以及返回值类型等信息并不会在编译多个源文件时共享。因此，我们必须保证每个模块源文件都包含必要的结构定义和函数原型信息，以便正确的对其进行编译。

外部变量

定义在多个文件之中的函数可以通过外部变量的方法来交换数据。外部变量是我们在第8章“使用函数”中介绍的全局变量概念上的扩展。

所谓外部变量，就是在某个源文件中定义，而在另外一个源文件中被访问的变量。如果我们需要在模块中访问某个外部变量，我们需要在该模块中按照普通的方式声明该变量，同时在声明语句前面加上extern关键字。这个语句告诉编译器，我们要在本模块中访问另外一个模块中定义的全局变量，该全局变量的类型如我们的声明语句所示。

假定我们需要定义一个名为moveNumber的int类型变量，而且我们需要从另外一个文件定义的函数中访问这个变量的数值。在第8章“使用函数”中，我们知道，如果我们把moveNumber变量定义在文件的最开始，所有函数的外面，那么该文件的所有函数都可以使用该变量，也就是说，moveNumber被定义为一个全局变量。

实际上，其他源文件中的函数也可以访问上面定义的全局变量moveNumber。更确切的说，上面定义的moveNumber变量，不仅是一个全局变量，而且是一个外部全局变量。为了从另外一个模块中访问该变量，我们可以在需要访问该变量的模块中加入外部全局变量的声明语句，如下所示：

```
extern int moveNumber;
```

有了这条语句之后，该模块就可以访问moveNumber变量了。如果有多个模块需要访问变量moveNumber，那么每个模块中都应该加入上述的外部全局变量声明语句。

在使用外部变量的时候，我们必须遵循一个重要的原则，那就是我们必须在某个源文件中确切的定义该变量（译者注：声明只是告诉编译器某个变量的类型，定义则明确的为该变量分配存储空间，类似的例子有函数定义和函数原型声明，我们可以在多个地方出现函数的原型声明，但是必须有一个且只有一个函数的确切定义）。有两种方法可以定义该全局变量，一种是在某个源文件中，在任何函数的外面，不使用extern关键字而声明该变量，如下所示：

```
int moveNumber;
```

使用这种方法，我们还可以同时初始化该变量。

第二种定义外部全局变量的方式是在某个源文件所有函数的外面，使用extern关键字声明该变量，同时明确的初始化该变量，如下所示：

```
extern int moveNumber = 0;
```

请读者注意，这两种方法是互相排斥的，我们不能同时使用它们声明同一个变量。

综上所述，在使用全局变量的时候，我们必须在所有的模块中有一处（且只有一处）省略extern关键字。如果在所有声明该全局变量的地方我们都使用了extern关键字，那么必须在一处（且只有一处）初始化该变量。

下面的例子演示了外部变量的使用方法。假定我们把下面的代码输入到源文件main.c中：

```
#include <stdio.h>

int i = 5;

int main (void)
{
    printf ("%i ", i);
    foo ();

    printf ("%i\n", i);
    return 0;
}
```


main.c中的程序定义了一个全局变量*i*，所有其他的模块都可以使用extern声明访问该全局变量。假定我们在另外一个源文件foo.c中有如下的语句：

```
extern int i;
void foo (void)
{
    i = 100;
}
```

随后我们使用如下的命令行将这两个源文件编译到一起：

```
$ gcc main.c foo.c
```

那么执行该程序将给出如下的输出：

```
5 100
```

上面的输出证明我们可以在函数foo中访问main.c中定义的全局变量*i*。

因为我们只是在函数foo中访问该全局变量，我们也可以将该变量的声明语句放置到函数内部，如下所示：

```
void foo (void)
{
    extern int i;

    i = 100;
}
```

如果在foo.c中有很多函数需要访问外部变量*i*，那么将该变量的声明语句放到程序开始所有函数的外面会方便一些，如果只是一个或者少数几个函数需要使用变量*i*，那么最好将该变量的声明语句放置到每个函数内部：这样能够使得整个程序组织的更为整齐一些，不使用该外部变量的函数不会感受到该变量的存在（译者注：避免名字冲突）。

当外部变量是一个数组的时候，我们在声明时不需要给出其长度。如下所示：

```
extern char text[];
```

上面的语句可以使我们在程序中使用其他源文件中定义的字符数组text。正如普通的数组声明一样，如果外部变量是一个多维数组，那么我们可以省略第一维的大小，如下所示：

```
extern int matrix[][50];
```

上面的语句声明了一个包含50列的外部二维数组matrix。

静态变量与外部变量/函数

我们已经知道，一个定义在函数外面的变量不仅是一个全局变量，而且是一个外部变量。在很多时候，我们仅仅需要定义一个全局变量，但是不希望它成为一个外部变量。也就是说，该变量在模块内部可以被所有的函数访问，但是非本模块的函数不能访问该变量。通过使用C语言的关键字`static`可以做到这一点。

如果下面的语句出现在所有的函数之外，那么它就定义了全局静态变量`moveNumber`：

```
static int moveNumber = 0;
```

对于包含该语句的源文件来说，随后定义的所有函数都可以访问变量`moveNumber`，但是定义在其他源文件中的函数则不能访问该变量。

综上所述，如果我们需要一个全局变量，但是又不希望其他模块的程序能够访问该变量，那么可以使用`static`关键字声明它。使用`static`关键字可以更好的反映该变量的用途，而且两个无关的模块即使碰巧给全局变量起了同样的名字，也不会引起问题。

在本章的前面我们提到过，与变量不同，我们不需要使用`extern`关键字，就可以直接调用其他源文件中定义的函数。实际上，在定义函数的时候，我们也可以声明它是外部的（`extern`）或者是静态的（`static`），编译器的默认值是`extern`。如果我们将某个函数声明为静态的（使用`static`关键字），那么只有在定义该函数的模块中才能够调用该函数。比如，我们有一个函数`squareRoot`，我们可以在它的函数定义前面加上`static`关键字，使得该函数只能在本模块内被调用，如下所示：

```
static double squareRoot (double x)
{
    ...
}
```

上面定义函数`squareRoot`将成为一个局部的函数，只能在本源文件中调用该函数。

使用静态函数的理由和使用静态全局变量的理由类似，在此不再赘述。

图15.1的程序综合展示了不同模块之间的通讯方式，图中画出了两个模块：`mod1.c`和`mod2.c`。其中`mod1.c`定义了两个函数`doSquare`和`main`，程序的运行从模块`mod1.c`的`main`函数开始，`main`函数首先调用`doSquare`函数，然后`doSquare`函数再调用`square`函数。`square`函数在源文件`mod2.c`中定义。

因为`doSquare`函数被定义为静态的，所以该函数只能在`mod1.c`中调用，而不能在其他模块中调用。

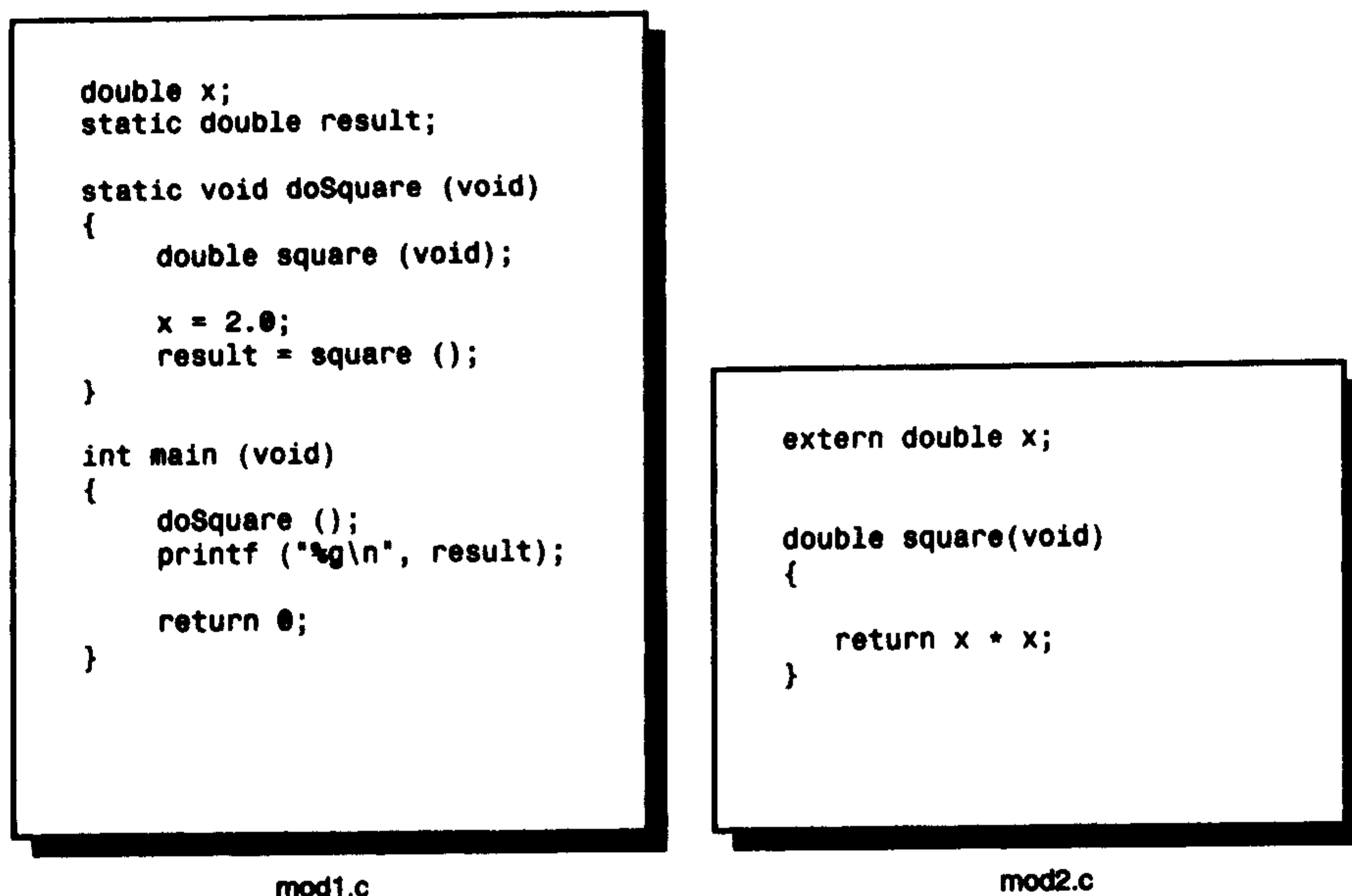


图 15.1 模块之间的通信

在模块mod1.c中定义了两个double型的全局变量x和result。x是一个外部变量，任何与mod1.c连接的模块都能够访问该变量。而result变量在声明时加上了static关键字，因此它是一个静态全局变量，只有定义在mod1.c中的函数（也就是main和doSquare）可以访问它。

在程序开始执行时，main函数首先调用函数doSquare。doSquare函数将全局变量x的值设置为2.0，然后调用函数square。因为函数square在另外一个模块中（mod2.c）定义，而且其返回值的类型并不是缺省的int，因此我们在模块mod1.c中包含了该函数的原型声明。

函数square计算全局变量x的平方，并将结果作为返回值。因为该函数需要访问全局变量x，而x被定义在另外一个模块中（mod1.c），因此我们使用extern语句在模块mod2.c中声明该外部变量（在本例中，在函数square内部声明外部变量和在该函数外部声明外部变量是一样的）。

函数square返回的值被函数doSquare保存在全局变量result中，随后执行流程转回到函数main。在main函数中，我们打印出变量result的值，最终的输出结果是4.0。（因为明显是2.0的平方）。

请读者仔细研究上面的例子，直到完全领会为止。这个例子虽然有点不太实际，而且很小，但是却展示了多模块通信中非常重要的概念。只有很好的领会和掌握了这些概念，我们才有可能更好的处理大型的程序。

有效的使用头文件

在第13章“预处理器”中，我们介绍了头文件的概念。我们可以把所有需要使用的公共定义放到一个头文件中，然后在所有需要这些定义的源文件中包含这个头文件。头文件在多模块程序的开发中显得尤为重要。

在有多个程序员参与的工程中，头文件提供了对定义进行标准化的手段：每个程序员都包含同样的头文件，从而使用了同样的定义。除此之外，每个程序员还省去了在每个源文件中手工输入这些公共定义的时间，同时还避免了可能出现的输入错误。如果我们把所有的公共结构定义、外部变量声明、typedef语句以及函数原型声明都放置在公共头文件中，那么还可以得到更大的好处。通过包含头文件中的公共定义，我们可以避免两个模块使用不同的结构定义，另外，如果我们需要修改结构的定义，那么只需要修改头文件一处即可。

请读者回忆一下第9章“使用结构”中的date结构。如果我们需要在多个模块中使用该结构，可以使用下面的头文件，该头文件中集中放置了与date结构相关的定义与原型声明等。这个例子很好的演示了如何综合运用我们迄今为止学到的各种技术：

```
// Header file for working with dates

#include <stdbool.h>

// Enumerated types
enum kMonth { January=1, February, March, April, May, June,
              July, August, September, October, November, December };

enum kDay { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday };

struct date
{
    enum kMonth month;
    enum kDay day;
    int year;
};
```



```
// Date type
typedef struct date Date;

// Functions that work with dates
Date dateUpdate (Date today);
int numberOfDays (Date d);
bool isLeapYear (Date d);

// Macro to set a date in a structure
#define setDate(s,mm,dd,yy) s = (Date) {mm, dd, yy}

// External variable reference
extern Date todaysDate;
```

该头文件中定义了两个枚举类型kMonth和kDay，该文件还定义了结构date，并在结构定义中使用了前面定义的枚举类型；随后该文件还使用typedef语句为数据类型struct Date起了一个新的名字Date，并随后声明了三个处理Date类型的函数和一个给Date结构赋值的宏，该宏使用了复合字符串。该头文件最后声明了一个名为todaysDate的外部变量，该变量将用来保存当前的日期（变量将在其他源文件中赋值）。

为了演示如何使用该头文件，我们重新编写了第9章中的dateUpdate函数，如下所示：

```
#include "date.h"

// Function to calculate tomorrow's date
Date dateUpdate (Date today)
{
    Date tomorrow;

    if ( today.day != numberOfDays (today) )
        setDate (tomorrow, today.month, today.day + 1, today.year);
    else if ( today.month == December ) // end of year
        setDate (tomorrow, January, 1, today.year + 1);
    else // end of month
        setDate (tomorrow, today.month + 1, 1, today.year);

    return tomorrow;
}
```

用于处理大型程序的其他工具

我们前面提到过，集成开发环境是管理大型程序的有效工具。但是如果我们需要从命令行来管理大型程序，那么还有一些工具可供使用。这些工具并不是C语言的一部分，但是使用它们可以帮助我们节省开发时间，这也正是这些工具的价值所在。

下面我们将介绍处理大型程序时可能用到的一些工具。如果读者是在Unix平台上工作，可能会发现类似的工具实在是太多了。我们这里能介绍的仅仅只是冰山的一角。学习一门脚本语言——比如Unix的shell，对于处理大量的文件是很有价值的。

make

make(GNU中为gnumake)允许我们在一个名为**Makefile**的文件中指定一组文件以及它们之间的依赖关系。通过检查文件的修改时间，make将自动编译那些需要重新编译的源文件。也就是说，如果我们的源文件(.c)的比目标文件(.o)新的话，那么make就将执行必要的编译命令，创建新的目标文件。我们也可以在Makefile中指定某个源文件依赖于特定的头文件，比如我们可以指定目标文件datefunc.o依赖于对应的源文件datefunc.c和它包含的头文件date.h，这样一来，如果我们修改了头文件date.h中的内容，那么make将会自动地重新编译datefunc.c文件，生成更新的目标文件datefunc.o。这是基于头文件比源文件新的简单事实。

下面是一个简单的Makefile，这个Makefile可以用来处理本章中三个模块的例子。我们假定该Makefile和所有的源程序放在同一个目录下。

```

$ cat Makefile
SRC = mod1.c mod2.c main.c
OBJ = mod1.o mod2.o main.o
PROG = dbtest

$(PROG): $(OBJ)
    gcc $(OBJ) -o $(PROG)

$(OBJ): $(SRC)
```

在这里我们就不详细介绍Makefile的编写规则了。简单的说，上面的Makefile定义了一组源文件(SRC)，一组目标文件(OBJ)，最终生成的可执行文件的名称(PROG)，还有它们之间的依赖关系。第一条依赖关系：

```
$(PROG): $(OBJ)
```

的意思是说最终的可执行文件依赖于所有的目标文件。因此如果我们修改过某个目标文件的话，就需要重新生成最终的可执行程序。从目标文件生成最终可执行程序的命令在下一行中列出，该行的开始必须是一个Tab字符。具体命令如下：

```
gcc $(OBJ) -o $(PROG)
```

Makefile的最后一行：

```
$(OBJ): $(SRC)
```


的意思是说每个目标文件都依赖于它对应的源文件。因此如果某个源文件被修改了的话，我们必须重新生成对应的目标文件。make工具知道如何从.c文件生成.o文件，因此我们不需要列出具体的命令。

下面是第一次运行make命令时候的输出结果：

```
$ make
gcc -c -o mod1.o mod1.c
gcc -c -o mod2.o mod2.c
gcc -c -o main.o main.c
gcc mod1.o mod2.o main.o -o dbtest
$
```

上面的输出告诉我们程序的编译过程一切正常，make最终将所有的目标文件连接起来，生成了可执行文件。

如果在源文件mod2.c中存在错误，那么make的输出则可能如下所示：

```
$ make
gcc -c -o mod1.o mod1.c
gcc -c -o mod2.o mod2.c
mod2.c: In function 'foo2':
mod2.c:3: error: 'i' undeclared (first use in this function)
mod2.c:3: error: (Each undeclared identifier is reported only once
mod2.c:3: error: for each function it appears in.)
make: *** [mod2.o] Error 1
$
```

上面的输出说明make在编译mod2.c时发现了错误，在这种时候，make的缺省动作是停止整个构造过程。

如果我们修正了文件mod2.c，然后再重新运行make，程序的输出如下：

```
$ make
gcc -c -o mod2.o mod2.c
gcc -c -o main.o main.c
gcc mod1.o mod2.o main.o -o dbtest
$
```

我们注意到make不再重新编译mod1.c文件，因为make可以推算出该文件不再需要重新编译。这个简短的例子说明了make工具的威力所在。

（译者注：原文中的Makefile依赖规则是有问题的，它并不能保证只重新编译修改过的文件。由于实际解释起来会占用较多篇幅，因此这里只给出正确的Makefile如下：

```
OBJ = mod1.o mod2.o main.o
PROG = dbtest

$(PROG): $(OBJ)
    gcc $(OBJ) -o $(PROG)
```

从这个简单的例子开始，读者可以尝试着使用Makefile来管理自己的程序。附录E“C语言的其他资源”列出了一些可以找到关于make工具更详细介绍的地方。

CVS

cvs是一种用于管理源代码的工具。cvs可以对于源代码的版本进行跟踪，记录每一个模块的修改历史记录，并在必要的时候重新创建出任何一个模块的历史版本（重新创建历史版本的原因通常有两个，一个是现有的版本有问题，我们需要回退到历史版本，另外一

个是客户在发布的历史版本中发现了问题，我们需要重新查看这些源代码以找出问题所在)。在使用CVS(这三个字母代表Concurrent Versions System，即并发版本系统)的时候，我们首先“检出”某个程序(使用cvs的checkout命令)，然后对其进行修改，最后再提交刚才的修改(使用cvs的commit命令)。使用上述步骤可以避免多个程序员同时修改某个源文件时可能带来的潜在冲突。通过cvs，位于多个地理位置的程序员可以利用网络共同在同一套源代码上工作。

Unix 的其他工具：ar、grep、sed 等等

Unix操作系统还提供了很多工具用于方便大型程序的开发与管理。例如，我们可以使用工具ar创建自己的函数库。当我们已经编写了大量的工具函数，并希望在多个程序中使用这些工具函数时，创建自己的函数库就显得很有价值。正如当我们在程序中使用了来自标准数学库的函数时就必须在连接时的命令行上指定-lm参数一样，我们也可以使用-l`lib`来指定程序需要和我们自己的函数库连接(其中的`lib`代表我们自己的函数库的名字)。当连接器看到这个选项之后，将会自动定位合适的函数库，并从函数库中找到被调用函数的代码，将其与我们最终的可执行程序连接起来。

grep和sed命令用于从源文件中寻找特定的字符串，或者对一组源文件进行全局性的修改。例如，通过同时使用shell脚本和sed命令，我们可以将一组源文件中的某个变量重新命名。grep命令用于在一组文件中寻找指定的字符串模式，如果我们需要在一组源文件中定位某个变量或者函数，或者需要从一组头文件中找出某个宏定义的话，这个工具就很有价值。

例如，下面的命令将会找出文件main.c中所有包含todayDate的行。

```
$ grep todayDate main.c
```

而下面的命令将会在当前目录下所有的源文件和头文件中查找字符串todayDate，并将相应的行号打印出来(-n开关用于打印行号)

```
$ grep -n todayDate *.c *.h
```

因为C语言允许我们将一个程序划分为多个源文件，并使用公共头文件在这些源文件之间共享各种定义，因此能够在多个文件中进行查找的工具将是非常有价值的。

如果读者使用集成开发环境来开发大型程序，那么管理多个模块将会非常直观。IDE会自动跟踪程序的修改情况，并在适当的时候进行必要的编译工作。如果读者使用的是gcc之类的命令行工具，那么就必须自己跟踪程序的修改情况，或者求助于类似make这样能够自动跟踪变化的工具。同时，读者可能还需要使用其他一些工具，用于对一组源文件进行全局修改，或者在一组文件中查找某个符号，或者创建和维护自己的函数库。

Input and Output Operations in C

C 语言的输入输出

直到目前为止我们编写的所有程序，其输入输出都是通过终端窗口完成的¹。如果我们想要输入某些信息，可以调用scanf函数或者getchar函数。与之类似，所有的输出操作都可以通过printf函数完成。

C语言本身并不包含任何用于完成输入输出操作（I/O）的语句，所有的I/O操作都是通过函数调用完成的。这些输入输出函数包含在标准函数库中。

读者可以回想一下，我们前面所有调用了printf函数的程序，都包含如下的语句：
`#include <stdio.h>`

这条语句在程序中包含了stdio.h文件，该头文件中的内容是C语言标准输入输出函数库中各类函数的原型声明以及宏定义。无论何时，只要我们的程序中使用了C语言标准输入输出函数库的功能，就必须包含这个头文件。

在本章中，我们将学习该函数库中的许多函数。不幸的是，由于篇幅限制，我们无法详细的讨论函数库中的每一个函数。如果读者需要的话，请参阅附录B“C语言标准库”，那里列出了绝大部分标准函数库中的函数。

¹我们在这里使用“终端”这个词代表运行程序的那个窗口，或者显示程序输出的那个窗口。在某些计算机系统中，输出窗口也被称为控制台（console）。

字符 I/O: getchar 函数和 putchar 函数

当我们需要从终端一次读入一个字符的时候，使用getchar函数将会非常方便。我们前面已经使用该函数开发了一个名为readLine的函数，readLine函数每次从终端上读入一行字符，该函数在内部重复的调用getchar函数，直到遇到一个换行符为止。

与getchar相对应，标准函数库有一个函数可以每次向终端输出一个字符，该函数就是putchar。

putchar函数的调用形式相当简单，它的唯一一个参数就是我们想要输出的字符。因此下面的调用语句：

```
putchar(c);
```

将变量c中保存的字符显示在终端上，在这里我们假定c是一个字符类型的变量。

下面的语句：

```
putchar('\n');
```

将在终端上输出一个换行符，实际的效果将使得光标移动到下一行的开始。

格式化 I/O: printf 函数和 scanf 函数

我们在前面已经多次使用了printf函数和scanf函数。在本节中我们将会学到这两个函数提供的用于格式化输入和输出的所有选项。

printf函数和scanf函数接受的第一个参数都是一个字符串指针，这个指针指向格式化字符串。在printf函数中，格式化字符串的作用是用来告诉该函数如何显示其他的参数，而在scanf函数中，格式化字符串则用来告诉该函数如何解释读入的字符。

printf 函数

在本书前面，我们已经知道，通过在格式化字符串的%和特定的格式化字符（也叫作类型限定符）之间放置特定的字符，可以更精确的控制输出的格式。比如，在程序5.3A中，我们知道可以在%和类型限定符之间放置一个整型数，用于指定输出的宽度。该程序中使用了%2i指定整数的输出宽度是2个字符，并且保持输出右对齐。我们还在第5章“循环”的练习6中看到，使用符号'-.'可以使得输出保持左对齐。

printf函数的格式化字符串的一般形式如下：

```
%[flags][width][.prec][hlL]type
```

其中可选的选项用方括号括起来。如果使用这些选项的话，它们必须按照格式中指定的顺序出现。

表16.1、表16.2和表16.3中列出了所有能在%和类型限定符之间使用的格式化字符。

表16.1 printf函数的标志

标志	意义
-	输出左对齐
+	在输出前面加上正负号
(空格)	在正数前面加上空格
0	使用0填充多余的空间
#	在八进制数前加上0; 在16进制数前加上0x (或者0X); 为浮点数显示小数点; 保持格式g或者G后面的0

表16.2 printf函数的宽度和精度限定符

限定符	意义
数字	输出的最小宽度
*	将参数列表中的下一个参数作为输出宽度
.数字	整数显示的最小位数; 在使用g格式时显示的最大位数; 对于s类型限定符允许的最大字符个数
.*	将参数列表中的下一个参数作为输出精度

表16.3 printf函数的类型修饰符

类型	意义
hh	作为字符显示
h*	作为短整数显示
l*	作为长整数显示
ll*	作为long long int类型显示
L	作为long double类型显示
j*	作为intmax_t或者uintmax_t类型显示
t*	作为ptrdiff_t类型显示
z*	作为size_t类型显示

*注意: 这些类型修饰符可以用在类型限定符n的前面, 用于说明对应指针参数的实际类型。

表16.4列出了所有可以在格式化字符串中使用的类型限定符。

表16.4 printf函数的类型限定符

字符	用于显示的类型
i或者d	整数
u	无符号整数
o	八进制数
x	十六进制数, 使用a-f
X	十六进制数, 使用A-F
f或者F	浮点数, 缺省显示6位数字
e或者E	科学计数法显示浮点数 (e在指数前显示小写的e, E在指数前显示大写的E)
g	根据需要使用f或者e显示浮点数
G	根据需要使用F或者E显示浮点数
a或者A	使用十六进制显示浮点数, 如0xd.ddddp±d
c	单个字符
s	空字符结尾的字符串
p	指针
n	该符号并不显示任何输出, 其对应的参数是一个整数指针, 用于存放直到目前为止本次调用输出的字符个数, 参看表16.3的说明
%	用于显示%

这几个表看上去也许有些太复杂了。我们看到, printf函数提供了如此多的参数用于控制输出的格式。熟悉这些内容最好的办法就是实验。在使用printf函数的时候, 我们一定要记得: 格式化字符串中%的个数应该和给出的参数个数相对应 (当然应该除过%%), 而且在整数输出的时候, 每一个宽度或者精度限定符中的*号, 也要对应一个参数。

程序16.1显示了部分printf函数格式化字符串选项的使用方法。

程序16.1 显示printf函数的格式化字符串使用

```
// 显示格式化字符串用法的程序
#include <stdio.h>

int main (void)
{
    char c = 'X';
    char s[] = "abcdefghijklmnopqrstuvwxyz";
    int i = 425;
    short int j = 17;
    unsigned int u = 0xf179U;
    long int l = 75000L;
```

程序16.1 续

```
long long int L = 0x1234567812345678LL;
float f = 12.978F;
double d = -97.4583;
char *cp = &c;
int *ip = &i;
int c1, c2;

printf ("Integers:\n");
printf ("%i %o %x %u\n", i, i, i, i);
printf ("%x %X %#x %#X\n", i, i, i, i);
printf ("%+i %i %07i %.7i\n", i, i, i, i);
printf ("%i %o %x %u\n", j, j, j, j);
printf ("%i %o %x %u\n", u, u, u, u);
printf ("%ld %lo %lx %lu\n", l, l, l, l);
printf ("%lli %llo %llx %llu\n", L, L, L, L);
printf ("\nFloats and Doubles:\n");
printf ("%f %e %g\n", f, f, f);
printf ("%2f %.2e\n", f, f);
printf ("%0f %.0e\n", f, f);
printf ("%7.2f %7.2e\n", f, f);
printf ("%f %e %g\n", d, d, d);
printf ("%.*f\n", 3, d);
printf ("%*.*f\n", 8, 2, d);
printf ("\nCharacters:\n");
printf ("%c\n", c);
printf ("%3c%3c\n", c, c);
printf ("%x\n", c);
printf ("\nStrings:\n");
printf ("%s\n", s);
printf ("%5s\n", s);
printf ("%30s\n", s);
printf ("%20.5s\n", s);
printf ("%20.5s\n", s);
printf ("%20.5s\n", s);
printf ("\nPointers:\n");
printf ("%p %p\n\n", ip, cp);
printf ("This\n is fun.%n\n", &c1, &c2);
printf ("c1 = %i, c2 = %i\n", c1, c2);
return 0;
}
```


程序16.1 输出

```

Integers:
425 651 1a9 425
1a9 1A9 0x1a9 0X1A9
+425 425 0000425 0000425
17 21 11 17
61817 170571 f179 61817
75000 222370 124f8 75000
1311768465173141112 110642547402215053170 1234567812345678 1311768465173141112

Floats and Doubles:
12.978000 1.297800e+01 12.978
12.98 1.30e+01
13 1e+01
12.98 1.30e+01
-97.458300 -9.745830e+01 -97.4583
-97.458
-97.46

Characters:
X
  X X
58

Strings:
abcdefghijklmnopqrstuvwxy
abcde
      abcdefghijklmnopqrstuvwxy
              abcde
abcde

Pointers:
0xbfffc20 0xbfffbf0

This is fun.
c1 = 4, c2 = 12
    
```

下面我们花一点时间详细分析程序的输出。程序中的第一组语句用于显示各种整型数，如短整数、长整数、无符号整数以及“普通的”int类型。程序的第一行首先使用十进制（%i）、八进制（%o）、十六进制（%x）和无符号（%u）四种格式显示变量i的值。请读者注意，输出的八进制前面没有符号0。

第二行仍然显示了变量i的值。首先我们使用十六进制（%x）显示其值，这个符号将十六进制数字显示为小写字母。接下来我们使用%x显示该变量，这次输出的十六进制字母是大写的。如果我们在输出的时候使用修饰符#的话，那么(%#x)在输出的数字前面加上小写的0x，而(%#X)在输出的数字前面加上大写的0X。

接下来的输出语句在格式化字符串中使用了一个标志+, 这个标志强迫printf在输出数字的时候, 即使该数字是一个正数, 也要打印其符号。接下来我们使用了标志空格, 这个标志强迫printf在输出正数的时候, 在前面加上一个空格。(该标志可以用来对齐输出, 对于正数前面有一个空格, 对于负数前面有一个负号) 接下来, 我们使用格式化字符串%07, 它的作用是强迫printf函数以右对齐的方式输出变量i的值, 输出宽度至少要占7个字符, 如果不足的话, 前面用0补齐。我们可以看到, 最终输出的结果是0000425。这个输出语句使用的最后一个格式化输出字符串是%.7i, 它的作用是最少使用7位宽度显示变量i的值。在本例中, 其作用与%07i相同, 输出结果也是0000425。

接下来的语句使用各种方式输出了短整数j的值, 所有能用于普通整数的格式化符号都可以用于短整数。

接下来的printf语句演示了使用%i输出无符号整数的情况。因为变量u的值大于有符号整数中所能存储的最大值, 所以printf的输出显示为一个负数。

第一组printf语句中的最后两条显示了如何使用修饰符l显示长整数和长长整数。

第二组printf语言演示了可以用于float类型和double类型的各类格式化输出符号。其中的第一条语句使用格式%f、%e和%g显示float类型。我们前面提到过, 当使用%f和%e的时候, printf语句默认的精度为6位小数。对于%g格式, printf函数根据需要输出数字的大小和指定的精度, 在%e和%f之间进行选择。如果指数小于-4或者大于指定的精度(默认为6), 那么printf函数将使用%e格式, 否则的话将使用%f格式。无论如何, printf函数都将略去小数部分最后的0。另外, 只有小数部分不为0时, 该函数才将小数点显示出来。一般来说, 使用%g显示浮点数在美观效果上是最好的。

接下来, 我们使用精度限定符.2, 该符号告诉printf语句只输出两位小数。我们可以看到, 最终输出的结果经过了四舍五入。接下来我们看到, 如果使用精度限定符.0, 那么printf语句将不输出任何小数部分, 同时还将省略小数点。最终的输出结果同样经过了四舍五入。

下一行的输出使用了精度限定符7.2, 这个符号告诉printf语句输出最少占据7个字符的位置, 其中小数部分占据两个。因为两个浮点数的输出宽度都不足7位, 因此最终的输出结果是右对齐的。

接下来的三行语句演示了如何使用各种格式输出双精度数d。所有可以用于单精度浮点数的格式化输出符号都可以用于双精度数。读者可能还记得, 根据C语言的类型转换规则, 在将参数传递给函数printf的时候, 所有的单精度数实际上被自动转换为双精度数。

下面的printf语句:

```
printf ("%.*f\n", 3, d);
```

显示双精度数d的值, 并保留3位小数。格式化输出符号中的*代表printf函数将参数列表中下一个尚未处理的数值作为输出的精度值, 在本例中, 这个数值正好是3。*的值也可以使用变量指定, 如下所示:

```
printf ("%.*f\n", accuracy, d);
```

这样就起到了动态调整输出格式的效果。

第二组输出语句中的最后一个使用了格式化字符串*.*输出双精度数d的值。在这个例子中, 总的输出宽度和小数位数都是使用变量指定的, 因此, 参数列表中的第一个数8被当作总的输出宽度, 第二个数2被当作保留的小数位数。这里需要读者注意的是, 输出的负号以及小数点都要算在总体的输出宽度中, 对于其他格式也是如此。

第三组输出语句使用各种格式显示字符变量c的值。该字符变量的值被初始化为大写字母x。第一个printf语句使用普通的%c显示该字符的内容, 第二行中, 我们使用%3c将该字符显示了两次。因为我们指定的显示宽度为3, 所以显示的结果中x前面有两个空格。

我们也可以将显示整数的任何格式化符号用于字符变量。比如, 下一行printf语句显示了变量c的十六进制值。我们可以看到, 笔者使用的计算机在内部使用数值58代表字符x。

最后一组输出语句使用各种格式显示字符串的内容。首先我们使用普通的%s显示字符串的内容, 然后我们再使用了精度限定符5, 这样printf将只显示字符串的前五个字符, 最终的结果是字母表的前5个字母。

接下来我们使用了宽度限定符30, 最终的结果是所有的字母都被打印出来并且右对齐。

接下来的语句使用宽度限定符20和精度限定符5输出字母表中的前5个字母。第一次我们采用右对齐的方式显示这5个字母, 第二次我们使用了修饰符-, 该符号使得printf语句以左对齐的方式显示这5个字母。为了达到限定的宽度20, printf语句在5个字母后面输出了15个空格, 这一点可以通过最后打印出的符号l来验证。

格式化符号%p可以用来输出指针的值。我们在程序中使用它来输出整数指针ip和字符指针cp的值。因为内存分配的关系, 当读者运行本程序的时候, %p的输出结果多半和这里给出的输出不同。

使用%p时的输出格式与C语言编译器的具体实现是相关的。在本例中, printf语句使用十六进制输出指针的值。我们可以看到, 指针变量ip的值是bffffc20, 而cp的值是bffffbf0。

最后一组语句演示了格式化输出符号`%n`的使用方法。当在`printf`语句中使用`%n`的时候，其对应的参数必须是一个整型数指针，除非我们在`n`前面使用了类型修饰符`hh`、`h`、`l`、`ll`、`j`、`z`或者`t`。当遇到`%n`的时候，`printf`语句将迄今为止输出的字符个数保存在`%n`对应的指针变量中。因此，在我们的示例程序中，整型变量`c1`中保存的值将是4，因为`printf`语句在遇到其对应的`%n`时，已经输出了4个字符。整型变量`c2`中保存的值为12，因为`printf`语句在遇到其对应的`%n`时，已经输出了12个字符。请读者注意，格式化字符串中的`%n`对于计数本身没有影响。

scanf 函数

迄今为止我们已经展示了多种可以用于`scanf`函数的格式化输入符号。和`printf`函数一样，我们也可以在`scanf`函数格式化输入字符串的`%`和类型限定符之间放置各种修饰符号，以控制`scanf`函数的行为。表16.5列出了所有的修饰符号，表16.6则列出了所有的类型限定符号。

当我们使用`scanf`函数读取输入的时候，该函数将忽略输入开始处所有空白字符。空白字符包括：空格（SPACE）、制表符（`'\t'`）、垂直制表符（`'\v'`）、回车符（`'\r'`）、换行符（`'\n'`）以及表单符号（`'\f'`）。不过`%c`是一个例外，当我们使用的时候`%c`的时候，`scanf`总是读取输入中的下一个字符——无论该字符是什么。另外，`[]`限定符也是一个另外，这个时候，`scanf`函数将根据方括号中的字符判断哪些输入是有用的，哪些输入应该被忽略。

表16.5 可用于scanf函数的修饰符

修饰符	意义
*	需要跳过的输入
数字	输入的最大宽度
hh	在有符号或者无符号字符变量中存入读取的值
h	在short int类型的变量中存入读取的值
l	在long int, double或者wchar_t中保存读取的值
j、z或者t	在size_t(%j)、ptrdiff_t(%z)、intmax_t(%t)或者uintmax_t(%t)中保存读取的值
ll	在long long int类型的变量中存入读取的值
L	在long double类型的变量中存入读取的值
类型	转换字符

表16.6 可用于scanf函数的类型限定符

字符	动作
d	读取以十进制表示的数。对应的参数是一个指向int类型变量的指针，如果使用了修饰符h、l或者ll，那么分别对应short、long或者long long int类型的整数
i	与%d的作用相当，但是它还可以读取八进制的数（以0开头）和十六进制的数（以0x或者0X开头）
u	读取整数，对应的参数是一个指向无符号int类型变量的指针
o	无论输入是否以0开始，都以八进制读取整数。对应的参数是一个int指针，如果使用了修饰符h、l或者ll，那么分别对应short、long或者long long int类型的整数指针
x	无论输入是否以0x或者0X开始，都以十六进制读取整数。对应的参数是int类型指针。可以使用h、l或者ll修饰符。
a、e、 f或者g	读取浮点数，该浮点数前面可以有符号位，也可以用指数形式表达。对应的参数是一个指向float类型变量的指针。如果使用l或者L修饰符，则是double类型或者long double类型的指针
c	从输入中读取一个字符。空白字符也将被读取。对应的参数是char指针。如果在c前面放置数字，则表示需要读取字符的数量
s	从输入中读取一组字符，将其保存在对应的字符数组或者字符指针中。开始的空白字符将被跳过，读取遇到空白字符就将停止。作为参数的字符数组或者字符指针的尺寸必须至少比读取的字符数目大1（考虑结尾的null字符）。s前面也可以放置一个数字表示需要读取字符的数量，这个时候，scanf最多只读取指定数目的字符。
[...]	同s一样，该符号也用于读取字符串。读取的字符串只能由[]中的字符组成，如果遇到了非[]中的字符，那么scanf就停止读取。可以使用^反转字符集，这种时候，[]中的字符被当作是结束字符，如果遇到了其中的任何一个，那么scanf就停止读取。（译者注：Microsoft Visual C++不完全支持上述特性）

表16.6 续

字符	动作
n	不读取任何东西。直到目前为止本次scanf调用已经读取的字符数目将被保存到对应的int指针中
p	读取一个指针的值。值的格式应该和printf的%p格式相同，对应的参数应该是void**类型
%	输入的下一个字符必须是%

在使用scanf函数读取某种数据的时候，如果已经读取了指定个数的字符（在%和类型限定符之间用数字指定），那么scanf函数就将停止继续读取；如果没有指定读取字符的个数，那么scanf函数则一直读到对于正在读取的数据来说无效的字符为止。以整数为例，有效的字符序列是一个可选的正负号和一组合法的数字（十进制是0-9，八进制是0-7，十六进制则是0-9，a-f或者A-F）。对于浮点数来说，有效的字符序列是一个可选的正负号，一组十进制数字，一个可选的小数点，后面再是一组十进制数字，然后再是可选的指数符号（e或者E），再是一组十进制数字（作为指数，可以有正负号）。对于%a（十六进制浮点数），有效的字符序列则是0x，后面跟上一组十六进制数字（中间可以有一个小数点），然后再是可选的指数符号（p或者P）和一组十六进制指数。

对于字符串（%s），任何非空白字符都是有效的。对于字符（%c），任何字符都是有效的。最后，对于方括号（[]），则只有方括号中的字符是有效的，如果[]中第一个字符是^的话（反转有效字符），那么所有不在方括号中的字符则是有效字符。

在第9章“使用结构”中，我们曾经使用scanf函数读取用户输入的时间。我们当时已经提到，对于格式化输入字符串中任何非格式的字符（也就是那些没有特殊意义的字符），用户都应该在输入的时候原封不动的给出。比如下面的scanf语句：

```
scanf ("%i:%i:%i", &hour, &minutes, &seconds);
```

该语句读入三个整型数，分别保存在变量hour、minutes和seconds中。格式化输入字符串中的两个冒号则要求用户输入的三个数字之间必须用冒号隔开，并且中间不能有其他空白字符。

如果想要用户输入一个百分号（%），那么我们必须要在格式化字符串中指定两个百分号（%%），如下面的语句所示：

```
scanf ("%i%%", &percentage);
```

格式化输入字符串中的空白可以用来匹配用户输入中任意数量的空白字符（包括0个空白字符），因此对于下面的scanf语句：

```
scanf ("%i%c", &i, &c);
```

如果我们输入下面的文字：

29 w

那么scanf语句执行完成之后，i中的值是29，而c中的值是空格。而如果我们使用下面的scanf语句：

```
scanf ("%i %c", &i, &c);
```

那么scanf语句执行完成之后，i中的值是29，而c中的值则是字符'w'。因为第二个scanf语句格式化输入字符串中的空白可以匹配用户输入中29后面的所有空白。

表16.5中的星号(*)可以用来跳过某个输入。例如，如果我们有如下的scanf调用：

```
scanf ("%i %5c %*f %s", &i1, text, string);
```

同时用户的输入如下：

```
144abcde 736.55 (wine and cheese)
```

那么scanf语句执行完成之后，整型变量i1的值是144，字符数组text中的值是abcde，接下来的736.55被%*f所匹配，但是并不赋给任何变量。随后字符串"(wine"被保存在字符串string中，后面再加上一个结束空字符null。如果我们随后再调用scanf函数，那么该函数将从上一次调用分析的终点开始分析，例如，如果我们随后调用如下的scanf语句：

```
scanf ("%s %s %i", string2, string3, &i2);
```

那么string2中的值是"and"，string3中的值是"cheese)"，然后scanf函数停下来等待用户输入更多的内容。

在使用scanf函数的时候，读者一定要记住，scanf函数需要的参数类型是变量指针，因为只有这样，scanf函数才能将读取的内容放到该变量中（请回忆一下第11章“指针”）。另外，读者还要记住，如果我们将读取的内容放到数组中，那么只要使用数组名就可以了，因为在C语言中，数组名代表指向数组第一个元素的指针。

按照上述讨论，如果text是一个有足够大小的字符数组，那么下面的语句从终端读取80个字符，并将其存储在字符数组text中。

```
scanf ("%80c", text);
```

下面的scanf语句读取除了字符'/'之外的任何字符到字符数组text中。

```
scanf ("%[^/]", text);
```

因此，如果用户在终端上输入如下的内容：

```
(wine and cheese)/
```

那么text数组中保存的内容将是字符串"(wind and cheese)"。剩下的/字符将在下一次scanf函数调用的时候被读取。

如果需要从终端上读取一整行输入，可以使用如下的scanf语句：

```
scanf ("%s", buf);
```

上面语句中格式化输入字符串最后的换行符可以匹配一行结束处的换行符，这样下一次scanf语句调用就不会读取到这个换行符了。

如果scanf函数在读取的时候遇到了对于需要读取的值类型来说不合法的字符（比如需要读取一个整数，但是却遇到了字符'x'），那么scanf函数将不再读取任何输入，而是立刻返回调用者。因为scanf函数的返回值代表了读取过程中成功读取并赋值的变量个数，因此我们可以使用这个返回值来判断在scanf函数的执行期间是否发生了错误。例如下面的语句：

```
if ( scanf ("%i %f %i", &i, &f, &l) != 3 )  
    printf ("Error on input\n");
```

该语句判断scanf函数的返回值是否是3，如果不是的话，在读取输入的过程中肯定发生了错误，我们的程序于是打印出相应的错误信息。

在这里我们需要提醒读者注意，scanf函数的返回值是成功读取并赋值的变量个数，因此对于下面的scanf语句：

```
scanf ("%i %*d %i", &i1, &i3);
```

如果该语句运行成功的话，返回值将是2而不是3，因为我们只读取并赋值了两个整数（中间跳过的一个并不计算在内）。另外，%n（用于获取迄今为止读取的字符个数）也不计算在scanf函数的返回值之中。

scanf函数的格式化输入字符串是比较复杂的，请读者自行进行一些练习。和printf函数一样，只有多做练习，才能真正学好这个函数。

文件输入输出操作

迄今为止，本书中所有的scanf函数调用都是从终端中输入数据，与之类似，所有的printf语句也都显示在终端上。本节我们将介绍如何从文件中读写数据。

将 I/O 操作重定向到文件中

在许多操作系统中（比如Unix和Windows），如果我们需要从文件而不是终端上读写数据，实际上不需要对程序进行任何修改。比如，如果我们需要将某个程序的输出全部写到一个名为data的文件中，唯一需要完成的工作就是打开一个终端，然后运行该程序，并将其输出重定向到文件data中。具体命令如下所示：

```
prog > data
```

上面这个命令告诉操作系统执行程序prog，但是将该程序的所有输出写到名为data的文件中而不是终端上。因此，我们在程序中使用printf语句输出的任何内容，都将保存到文件data中而不在终端上显示。

为了演示重定向的工作过程，我们下来做一个实验。我们输入本书的第一个程序3.1，然后像通常那样编译该文件。接下来，我们像平时运行程序那样，在终端上输入该程序的名字，命令如下所示（假定该程序的名字为prog1）：

```
prog1
```

如果一切正常的话，我们可以在终端上看到如下的输出：

```
Programming is fun.
```

接下来，我们输入下面的命令：

```
prog1 > data
```

这次我们注意到，终端上没有任何输出。这是因为操作系统将该程序的所有输出都重定向到文件data中了。如果我们来查看文件data的内容，就会发现其中包含着如下一行文本：

```
Programming is fun.
```

这说明程序的全部输出现在保存在文件data中了。如果读者还有疑问的话，可以试着运行一个输出稍微多一些的程序，看看该程序的输出是不是也能够全部被重定向到文件中。

和输出一样，我们也可以重定向程序的输入。重定向程序的输入之后，所有原来从终端读取数据的函数例如scanf或者getchar都将从指定的文件中读取数据。我们前面编写的程序5.8的作用是从终端读入一个数，然后逆序排列该数所有的数字。通过输入重定向，我们可以用这个程序来逆序排列一个文件中的数字。假定程序5.8的名字是reverse，下面的命令行可以重定向该程序的输入：

```
reverse < number
```

如果我们在文件number中输入数字2001，然后执行上面的命令，终端上将会出现如下的输出：

```
Enter your number.  
1002
```

这里提醒读者注意，虽然我们的程序reverse打印出一条提示信息，要求用户输入一个数字，但是该程序并未停下来等待用户输入。这是因为我们将该程序的输入重定向到了文件number中，因此程序中的scanf调用将直接从文件中读取需要的数据，而不必等待我们从终端输入。我们在文件中输入数字的格式必须要和在终端上输入时相同，这样scanf函数才能够正常的工作。对于scanf函数来说，它并不知道自己是从终端读取数据还是从文件中读取数据，scanf的唯一要求是所读取数据的格式必须正确。

我们也可以同时重定向一个程序的输入和输出。如下所示：

```
reverse < number > data
```

上面的命令运行程序reverse，该程序从文件number中读取需要的数据，并把程序的输出写到文件data中。因此，如果我们运行上面的命令行，那么程序reverse就从文件number中读取一个数字，然后将其逆序排列，并将输出写到文件data中。

重定向程序的输入输出是一个非常使用的技巧。例如，假定读者正在编写一个稿件，并将其中的内容保存到文件article中。我们前面编写了程序10.8，该程序可以用来统计用户在终端上输入的单词数。同样，使用重定向技术，我们也可以使用这个简易程序来统计文件article中的单词数。具体命令如下²：

```
wordcount < article
```

另外，我们要提醒读者，I/O重定向并不是ANSI C中的一部分，因此读者有可能遇到不支持重定向的操作系统。不过幸运的是，绝大多数操作系统都支持重定向。

文件结束标志

接下来我们将讨论输入数据被读完时会发生什么情况。对于文件来说，这就意味着我们到达了文件的结尾。在C语言中，如果我们读完了文件中的数据，系统将设置一个文件结束标志。如果我们的程序在系统设置了文件结束标志后继续读取数据，有可能会出现程序错误。如果我们的程序不检查这个标志的话，还有可能导致程序进入无限循环。幸运的是，C语言标准库中的绝大多数输入输出函数在遇到这种情况时，都将返回一个特殊的值，用于告诉调用者程序遇到了文件结束标志。这个特殊的值用符号EOF代表，该符号在系统头文件<stdio.h>中定义。

程序16.2演示如何使用文件结束标志与getchar函数，将文件的内容显示在终端上。程序16.2从文件中逐个读取字符，并将其显示在终端上，直到遇到文件结束标志为止。请读者注意while语句中的表达式，C语言并不要求赋值操作一定是一个单独的语句。

² Unix 系统中提供了一个命令 wc，该命令可以用来统计文件中单词的数目。另外读者还要注意，我们设计的程序只能统计文本文件的单词数，而无法统计某些字处理软件（如 Microsoft Word）文档的中的单词数。

程序16.2

```
// 显示文件所有字符的程序
#include <stdio.h>

int main (void)
{
    int c;

    while ( (c = getchar ()) != EOF )
        putchar (c);

    return 0;
}
```

如果我们编译、运行程序16.2，并使用如下的命令将其输入重定向到某个文件：

```
copyprog < infile
```

那么该程序将在终端上显示出文件infile的内容。读者可以亲自动手试一试。实际上，这个程序的功能和标准的Unix命令cat的作用类似，我们可以用它来显示任何文本文件的内容。

在程序16.2的while循环中，我们将函数getchar的返回值保存在变量c中，然后将c的值与符号EOF进行比较。如果比较结果相等的话，这就说明我们已经到达了文件结尾，这时我们就停止读取文件。对于getchar函数返回的EOF值，有一点必须说明一下：那就是getchar的返回值类型实际上是一个int，而不是char。因为为了表示文件结束标志，EOF不能等于任何一个可能的字符值，字符的值范围通常在0-255之间（无符号），因此getchar函数可以用一个值不在该范围内的int类型的数值来表示文件结束状态。由于上述原因，getchar函数的返回值被保存在int类型而不是char类型的变量中。由于C语言允许我们在int类型的变量中保存char类型的数值，因此这样做在语法上是没有问题的，但是总的来说，这种编程习惯是不好的。

如果我们将getchar函数的返回值保存在char类型的变量中，其结果是不可预料的。在那些对于字符执行符号扩展的系统上，程序也许能够正常工作。但是对于进行无符号扩展的系统来说，程序很有可能进入无限循环。

因此，读者一定要记住使用int类型的变量来保存getchar函数的返回值，这样我们就能够正确地检测出文件结束标志。

上面的程序还演示了C语言一个很灵活的特性，那就是将赋值操作和条件判断操作合并到一个语句中。因为赋值操作的优先级比不等条件判断操作的优先级低，因此我们需要使用括号将赋值操作括起来。

用于读写文件的特殊函数

在读者将要开发的大多数程序中，可能仅仅使用`getchar`、`putchar`、`scanf`和`printf`以及输入输出重定向，就可以完成所有的I/O操作。然后在某些时候，我们需要能够更加灵活地对文件进行读写操作。比如我们需要从两个或者更多个不同的文件中读取或者写入数据。为了完成这些工作，C语言设计了一些专门用于文件的函数，在接下来的小节中，我们将介绍这些函数。

fopen 函数

在我们针对一个文件执行读写操作之前，我们必须首先打开这个文件。为了打开一个文件，我们必须指定该文件的文件名。C语言系统将会检查指定的文件是否存在，在某些情况下，如果该文件不存在的话，将会自动创建这个文件。在打开文件的时候，我们还必须指定将要对该文件执行的I/O操作类型。如果我们要从该文件中读取数据，我们通常使用读模式打开该文件；如果我们要将数据保存到该文件中，则通常使用写模式。最后，如果我们需要向一个已经包含某些数据的文件中追加数据，则可以使用追加模式。在写模式和追加模式中，如果我们指定的文件不存在，系统则会自动给我们创建这个文件。而在读模式中，如果该文件不存在的话，打开文件将会引发一个错误。

因为我们有可能在程序中打开多个文件，因此必须能够区分每个打开的文件，以便对于正确的文件执行读写操作。C语言提供了一种数据类型——文件指针来完成上述操作。

C语言提供了一个名为`fopen`的函数，我们可以使用该函数来打开一个文件。该函数的返回值是一个文件指针，我们在随后的读写操作中可以使用这个文件指针，以便与其他文件区分。`fopen`函数接受两个参数，第一个参数是字符串类型，用于表示将要打开文件的文件名，第二个参数还是字符串，用于表示使用何种模式打开文件。

如果`fopen`函数不能按照要求的模式打开指定的文件，该函数将返回`NULL`。符号`NULL`一般在系统头文件`<stdio.h>`中定义³。该系统头文件中还定义了一个结构类型`FILE`，我们必须使用一个指向该类型的指针变量来存储`fopen`函数的返回值。

³按照正式标准，`NULL`应该在系统头文件`<stddef.h>`中定义，但是绝大多数C语言编译器都在`<stdio.h>`中定义该符号。

综上所述，如果我们需要用读模式来打开文件data，可以使用如下的语句：

```
#include <stdio.h>
FILE *inputFile;
inputFile = fopen ("data", "r");
```

如果需要以写模式打开，那么第二个参数应该是字符串"w"，如果需要追加模式，则使用字符串"a"。fopen将返回一个代表该文件的FILE类型指针，我们将其存储在变量inputFile之中。

接下来，我们通过测试该指针的值是否为NULL，判断文件打开操作是否成功完成，具体语句如下所示：

```
if ( inputFile == NULL )
    printf ( "**** data could not be opened.\n" );
else
    // 从文件中读取数据
```

在使用fopen函数打开文件的时候，我们一定要记得对返回的FILE指针进行检查。将NULL指针用于文件操作将产生不可预料的结果。

我们常常将文件打开操作和对返回结果的检查合并到一个语句之中，如下所示：

```
if ( (inputFile = fopen ("data", "r")) == NULL )
    printf ( "**** data could not be opened.\n" );
```

fopen函数还支持另外三种文件打开模式，它们分别是读更新("r+")、写更新("w+")和追加更新("a+")。这三种模式都允许我们同时对文件进行读写操作。读更新模式可以用于打开一个已经存在的文件，并同时进行读写操作；写更新模式也可以同时进行读写操作，但是如果指定文件存在的话，该模式首先清除原有文件的所有内容，如果文件不存在的话则首先创建该文件；追加更新模式与写更新模式非常类似，可以用于打开已经存在的文件或者创建新的文件，但是以这种模式打开的文件，我们只能在文件末尾追加数据，读操作则可以在文件的任意位置进行。

在某些区分二进制文件和文本文件的操作系统中（如Microsoft Windows），如果我们需要读写二进制文件，那么还必须在模式结尾加上字符b（译者注：字符b是英文二进制binary的头一个字母）。如果不加上这个字符的话，我们的程序在读写二进制文件的时候，有可能表现出奇怪的行为。因为在这类操作系统中，文本文件的每行结束处的回车符被存储为两个字符（回车换行），但是使用文本模式读取的时候只看到一个字符（换行符），这个转换由操作系统自动进行，但是在读写二进制文件的时候则不进行这种自动转换。

另外，如果以文本方式打开的文件中包含有字符CTRL+Z，那么程序在读到这个字符的时候就会返回EOF。如果需要以二进制方式打开某个文件，可以使用如下的语句：

```
inputFile = fopen( "data", "rb");
```

getc 和 putc 函数

我们可以使用函数getc从文件读入一个字符。这个函数的行为与我们前面介绍的函数getchar完全相同，但是我们可以给getc传递一个指向FILE类型的指针参数，用于告诉该函数我们要从哪个文件中读取数据。因此，假定我们使用前面的示例语句打开了文件data，并将返回的文件指针保存在变量inputFile之中，那么下面的语句将从该文件中读取一个字符：

```
c = getc (inputFile);
```

如果想要读取更多的字符，可以接着调用getc函数。

和getchar函数一样，当遇到文件结尾标志的时候，getc函数将返回EOF，因此我们应该使用int类型的变量来保存该函数的返回值。

读者可能已经猜测到，putc函数的行为与putchar函数也是一样的，不过putc函数要接收两个参数，第一个是需要输出的字符，第二个是FILE指针。下面的语句：

```
putc ('\n', outputFile);
```

将换行符输出到outputFile对应的文件中。当然，在此之前我们必须使用写模式或者追加模式（或者任何一个更新模式）成功的打开了该文件。

fclose 函数

关于文件我们还必须要介绍一个操作，那就是关闭文件。fclose函数可以用来完成这项工作。从某种意义上来说，fclose函数所进行的操作与fopen的操作恰好对应，该函数告诉底层的操作系统，我们不再访问某个文件。操作系统关闭文件的时候，通常需要完成一些善后事宜（例如将所有内存缓冲区中的内容都保存到磁盘上等）。当文件关闭以后，除非再次打开该文件，否则的话我们在程序中将无法再对该文件进行任何I/O操作。

当我们在程序中处理完某个文件之后就将其关闭是一个很好的编程习惯。一般来说，当程序结束运行的时候，操作系统将会自动关闭所有打开的文件。但是在适当的地方手动关闭文件要更好一些，特别是如果我们的程序需要处理大量文件的时候，因为一般来说，操作系统对于一个程序能够同时打开的文件数目有一定的限制。这个限制随着具体操作系统的不同而有所变化。如果我们的程序同时访问大量的文件，这一点有可能给我们带来麻烦。

fclose函数接收一个参数，即指向需要关闭文件的FILE指针。因此下面的语句将关闭文件指针inputFile指向的文件。

```
fclose(inputFile);
```


在介绍完 `fopen`、`putc`、`getc` 和 `fclose` 函数之后，我们可以编写一个拷贝文件的程序来练习一下。程序 16.3 将提示用户输入需要拷贝文件的文件名，然后再输入拷贝之后文件的文件名，随后程序执行拷贝操作。程序 16.3 在内容上借鉴了程序 16.2，为了理解程序 16.3，读者可能需要参考一下程序 16.2。

假定我们有一个文本文件 `copyme`，其内容如下：

```
This is a test of the file copy program
that we have just developed using the
fopen, fclose, getc, and putc functions.
```

程序 16.3

```
// 拷贝文件的程序
#include <stdio.h>
int main (void)
{
    char inName[64], outName[64];
    FILE *in, *out;
    int c;

    // 从使用者得到文件名
    printf ("Enter name of file to be copied: ");
    scanf ("%63s", inName);
    printf ("Enter name of output file: ");
    scanf ("%63s", outName);

    // 打开输入输出文件
    if ( (in = fopen (inName, "r")) == NULL ) {
        printf ("Can't open %s for reading.\n", inName);
        return 1;
    }

    if ( (out = fopen (outName, "w")) == NULL ) {
        printf ("Can't open %s for writing.\n", outName);
        return 2;
    }

    // 复制 in 到 out
```

程序16.3 续

```
while ( (c = getc (in)) != EOF )
    putc (c, out);

// 关闭打开的文件
fclose (in);
fclose (out);

printf ("File has been copied.\n");
return 0;
}
```

程序16.3 输出

```
Enter name of file to be copied: copyme
Enter name of output file: here
File has been copied.
```

现在请读者检查一下文件here的内容，该文件中应该包含与文件copyme同样的三行文字。

在程序的开始，我们调用函数scanf读取文件名的时候，指定了输入的长度不能够超过63，这样可以避免用户输入的文件名长度超过inName和outName所能容纳的范围。接下来，该程序打开相应的文件分别用于读写操作。在绝大多数系统上，如果outName指定的文件已经存在的话，那么以写模式打开该文件将会清除其中原来的所有内容。

如果在两个文件打开操作中有任何一个不成功的话，我们的程序将显示错误信息，并停止执行。这时，我们向操作系统返回一个非零值。如果两个文件操作都成功的话，我们随后在一个循环中不断的调用getc函数和putc函数，将inName指定文件中的字符逐个拷贝到outName指定的文件中。在遇到文件结束标志之后，我们的程序关闭这两个文件，并向操作系统返回0值，表示操作成功完成。

函数 feof

为了检查某个文件是否已经设置了文件结束标志，我们可以使用函数feof。该函数接收单个参数，即需要检查的文件对应的文件指针。如果我们已经执行过一次试图在文件结束后读取文件的操作，该函数返回非零值，否则的话，该函数返回零。因此，下面的语句检验文件指针inFile对应的文件是否已经设置了文件结束标志，如果已经设置的话，则显示适当的信息。

```
if ( feof (inFile) ) {
    printf ("Ran out of data.\n");
    return 1;
}
```


读者需要记住, `feof`告诉我们的, 我们是否在文件已经结束后执行过读操作, 而不是我们是否已经读取了文件最后的数据。因此, 为了让`feof`函数返回非零值, 我们必须在读取文件所有的数据之后, 再执行一次读操作。

fprintf 函数和 fscanf 函数

我们可以使用`fprintf`函数和`fscanf`函数在文件上执行与`printf`和`scanf`函数类似的功能。这两个函数比起`printf`和`scanf`来要多接收一个文件指针作为函数参数, 用于标识函数将要读写的文件。如果我们要将字符串"Programming in C is fun.\n"输出到文件指针`outFile`所对应的文件中, 应该使用如下语句:

```
fprintf (outFile, "Programming in C is fun.\n");
```

与之类似, 如果我们想要从文件指针`inFile`对应的文件中读取一个浮点数, 并将其值赋给变量`fv`, 可以使用如下的语句:

```
fscanf (inFile, "%f", &fv);
```

与`scanf`函数类似, `fscanf`函数将成功读取并赋值的参数个数作为函数的返回值, 如果在读取任何内容之前遇到了文件结束标记, 则返回符号`EOF`。

fgets 函数和 fputs 函数

如果需从文件中读写整行数据, 我们可以使用C语言提供的`fgets`和`fputs`函数。
`fgets`函数的调用形式如下:

```
fgets (buffer, n, filePtr);
```

其中, `buffer`是一个字符数组, 用于保存读入的字符, `n`代表`buffer`最大能够容纳的字符个数, `filePtr`是需要从中读取数据的文件指针。

`fgets`函数在运行时将连续地从文件中读取数据, 直到遇到换行符或者读取了`n-1`个字符为止(译者注: 或者读取到文件结束处)。在读取完成之后, 该函数将在缓冲区的结尾放置一个`NULL`字符。如果读取操作成功完成的话, 该函数返回指向缓冲区的指针, 如果读取的时候发生了错误或者在文件结束标志设置之后尝试读取, 则该函数返回`NULL`。

如果我们将`fgets`函数和`sscanf`函数结合起来使用, 那么就可以得到一种比`scanf`更为有效和更容易控制的面向行的输入方法。`sscanf`函数将在附录B中介绍。

`fputs`函数将一个字符串输出到文件中。该函数的调用形式如下:

```
fputs (buffer, filePtr);
```

fputs函数将buffer中的字符输出到filePtr对应的文件中，一直到遇到buffer中的空字符为止。最终的null字符不会被输出到文件中。

在C语言的标准库中还有两个名为gets和puts的函数，它们可以用来从终端上读写整行文本。这些函数将在附录B中介绍。

标准输入 stdin、标准输出 stdout 和标准错误 stderr

当一个C语言程序运行的时候，系统将会自动为这个程序打开三个文件。这三个文件分别用文件指针常量stdin、stdout和stderr代表，它们都在文件stdio.h中定义。文件指针stdin代表程序的标准输入，它通常和当前程序的终端窗口相联系。所有用于输入操作的I/O函数，如果没有文件指针参数的话，都默认从这个文件指针读取数据。例如我们熟悉的scanf函数实际上就是从文件指针stdin读取，因此，如果我们给fscanf函数传递文件指针stdin，那么该函数的作用就将和scanf完全相同。例如下面的语句，其作用就是从标准输入读取一个整数，实际上就是从终端读取。

```
fscanf (stdin, "%i", &i);
```

如果我们在运行程序的时候重定向了输入的话，那么上面的语句将从重定向的文件中读取所需的整数。

读者可以已经猜到，符号stdout对应于标准输出，在一般情况下，标准输出也和运行该程序的终端窗口相联系。因此，下面的语句：

```
printf ("hello there.\n");
```

实际上和下面以stdout为文件指针的fprintf语句互相等价：

```
fprintf (stdout, "hello there.\n");
```

符号stderr对应于标准错误。程序运行过程中产生的绝大多数错误应该输出到这个文件中。一般情况下，标准错误也和运行该程序的终端窗口相联系。为什么我们要使用两个用于输出的文件指针stdout和stderr，而它们实际上指向同一个输出目标呢？这是因为，当我们重定向了一个程序的输出之后，程序运行过程中输出到stdout的内容将被写到重定向的文件中，而程序的错误仍然可以在终端窗口中显示出来。有些时候，我们也想要将自己的出错信息写到标准错误中，这个任务可以通过将标准错误文件指针stderr传递给fprintf函数来完成。请看下面的例子：

```
if ( (inFile = fopen ("data", "r")) == NULL )  
{  
    fprintf (stderr, "Can't open data for reading.\n");  
    ...  
}
```


上面的程序打开名为data的文件，用于从其中读取数据。如果因为某种原因文件打开失败的话，我们就使用fprintf函数将错误信息输出到标准错误上。而且，即使我们在运行程序的时候重定向了其输出，上面的错误信息依然可以在终端窗口中显示出来。

exit 函数

在某些情况下——比如发现了程序执行中的某个错误时，我们可能希望强制结束程序的运行。我们已经知道，当main函数中的最后一条语句被执行完，或者我们从main函数中调用了return语句之后，程序就将结束其运行。为了从任何位置强制结束程序的运行，我们可以调用exit函数。该函数的调用形式如下所示：

```
exit (n);
```

exit函数将立即中止当前程序的运行。所有已经打开的文件将被系统自动关闭。exit函数的参数n被称为程序的退出状态码，其作用与从main函数中调用return语句返回的数值相当。

C语言的系统头文件<stdlib.h>中定义了两个整数类型的符号：EXIT_FAILURE和EXIT_SUCCESS。通过返回EXIT_SUCCESS，我们告诉操作系统当前程序成功的结束了运行，反之，如果返回EXIT_FAILURE的话，则表示程序运行失败。

在一个程序执行完了main中的所有语句并退出运行的情况下，其退出状态码处于一个未定义的状态（译者注：当然，最后一条语句不能是return语句）。如果其他程序需要通过该程序的退出状态码来决定该程序的运行结果时，就会得到不确定的结果。在这种情况下，我们应该通过exit函数或者在main函数的return语句中明确的指定程序的退出状态码。

下面我们举一个使用exit函数的例子。在函数openFile中，我们尝试以读模式打开函数参数中指定的文件，如果文件打开失败的话，该函数将以EXIT_FAILURE为参数调用exit函数，使得整个程序停止运行。当然了，在实际的程序中，我们应该返回一个错误，表示指定的文件无法打开，而不应该这样突然地结束程序的运行。

```
#include <stdlib.h>
#include <stdio.h>
FILE *openFile (const char *file)
{
    FILE *inFile;

    if ( (inFile = fopen (file, "r")) == NULL ) {
        fprintf (stderr, "Can't open %s for reading.\n", file);
        exit (EXIT_FAILURE);
    }

    return inFile;
}
```

调用exit函数的效果和从main函数中返回并没有任何实质上的不同。它们都将设置程序的退出状态码，并终止程序的运行。当然，如果从一个函数内执行return语句和调用exit

函数，那么就有很大的不同，前者将执行流程返回到调用者，后者则终止程序的运行。

重命名和删除文件

我们可以使用C语言标准库函数`rename`来改变某个文件的名字。该函数接受两个参数，第一个是文件原来的文件名，第二个是文件的新文件名。如果由于某种原因，该函数的运行失败了（比如，原来的文件名对应的文件不存在，或者系统不允许我们重命名某个特殊文件的话），该函数将返回一个非零值。下面的代码：

```
if ( rename ("tempfile", "database") ) {  
    fprintf (stderr, "Can't rename tempfile\n");  
    exit (EXIT_FAILURE);  
}
```

将名为`tempfile`的文件重命名为`database`，并检查程序运行的结果。

我们可以使用函数`remove`删除某个文件，只要将该文件的文件名作为参数传递给`remove`即可。如果删除失败，则该函数返回一个非零值。请看下面的示例代码：

```
if ( remove ("tempfile") )  
{  
    fprintf (stderr, "Can't remove tempfile\n");  
    exit (EXIT_FAILURE);  
}
```

上面的代码试图删除一个名为`tempfile`的文件，如果删除失败的话，这段代码在标准错误上打印出错误信息，并终止整个程序的运行。

顺便提一下，C语言标准库函数`perror`在报告错误方面很有用处，如果读者感兴趣的话，可以参考附录B。

到这里我们关于C语言中输入输出操作的介绍就结束了。我们前面已经说过，由于篇幅的缘故，这里只介绍了一部分输入输出的库函数。C语言的标准库中还包含有很多用于从字符串中输入输出，或者执行随机I/O操作，以及数学计算和动态内存分配的函数。附录B中介绍了它们中的绝大部分。

练习

1. 输入并运行本章的三个例子程序，并将结果与书中给出的结果进行比较。
2. 重新查看我们前面编写的某些程序，并试着以重定向输入输出的方式运行它们。
3. 编写一个程序，将一个文件的内容拷贝到另外一个文件中。在拷贝的时候，将原来文件中所有的小写字母都替换为对应的大写字母。

4. 编写一个程序，该程序交替的读取两个文件，并将其合并输出到标准输出中。如果一个文件中的行数小于另外一个文件，那么我们的程序应该将较长的那个文件中的剩余内容也写到标准输出中。
5. 编写一个程序，该程序将某个文件每一行上从 m 到 n 列之间的字符输出到标准输出。我们的程序应该从终端中读取 m 和 n 的值。
6. 编写一个程序，该程序每次读取并显示某个文件中的20行文本。在显示完之后，我们的程序等待用户输入一个字符，如果用户输入了字母q，则程序停止运行，否则的话，程序再显示该文件中的20行文本。

Miscellaneous and Advanced Features

杂项和高级特性

本 本章将对前面没有提到的C语言中某些特性进行介绍，并针对其中部分高级特性例如命令行参数、动态内存分配等展开详细的讨论。

杂项语句

本小节将介绍两个迄今为止读者尚未遇到过的语句：`goto`语句和空语句。

`goto` 语句

每一个学习过结构化编程语言的人可能都知道，`goto`语句有一个坏名声。但是几乎每种计算机语言都有这个语句。

执行`goto`语句可以使程序的执行流程直接调转到某个特殊点。这个跳转总是无条件立刻执行的。为了告诉计算机将要跳转到的位置，我们需要在程序中定义一个标号。标号的命名规则与普通变量相同，但是后面必须跟上一个冒号。标号应该放在我们需要跳转到的那条语句前面，而且应该和对应的`goto`语句位于同一个函数中。

例如，如果执行下面的语句：

```
goto out_of_data;
```

程序将跳转到前面有`out_of_data:`标号的那条语句去执行。该条语句可以位于函数的任意位置，既可以在`goto`语句的前面，也可以在`goto`语句的后面。比如下面的语句就是一条带有标号的合法语句：

```
out_of_data: printf ("Unexpected end of data.\n");  
...
```


懒惰的程序员经常使用goto语句跳转到程序的其他位置。由于goto语句打乱了程序的正常顺序执行流程，因此使用goto语句的程序很难跟踪。如果在程序中使用了大量的goto语句，那么我们几乎不可能理解该程序。由于这个原因，良好的编程风格规范中都要求不要使用goto语句。

空语句

C语言允许我们在任何需要一个语句的地方只放置一个单独的分号，这个单独的分号被称为空语句。空语句不进行任何操作。虽然空语句初看上去没有什么用处，但是C语言的程序员们经常在循环语句如while、for和do while中使用它。比如下面的while语句就使用了空语句，它的作用是将标准输入中读取的字符存放在字符数组text中，直到遇到换行符为止。

```
while ( (*text++ = getchar ()) != '\n' )
    ;
```

在上面的示例中，所有的读取、赋值和判断操作都在while语句的条件判断子句中完成，但是我们还是需要在其后面加上一个空语句，否则的话，C语言编译器将把while后面的其他语句作为循环体处理。

下面的语句从标准输入读取字符，并将其输出到标准输出，直到遇到文件结束标记为止。它也使用了空语句。

```
for ( ; (c = getchar ()) != EOF; putchar (c) )
    ;
```

下面的语句计算从标准输入读取的字符总个数。

```
for ( count = 0; getchar () != EOF; ++count )
    ;
```

作为空语句的最后一个例子，下面的语句将字符指针from中的字符逐个拷贝到字符指针to中。

```
while ( (*to++ = *from++) != '\0' )
    ;
```

从上面的例子读者可以看到，某些程序员倾向于将尽可能多的操作写到while循环的条件判断语句或者for循环的条件语句中。读者千万不要向这些程序员学习。一般说来，条件判断部分应该只包含条件判断语句，所有其他的工作都应该在循环体中完成。使用类似上面语句的唯一可能的原因是它可以增加程序的效率（译者注：实际上现代C语言编译器的优化功能非常之好，以至于上面的技巧实际上并不会使得最终生成的代码更为紧凑，并增进效率）。除非程序的效率是一个非常关键的因素，否则读者应该避免使用上面类型的循环语句。

上面的while循环语句如果写成下面的形式，那么就要容易阅读得多。

```
while ( *from != '\0' )  
    *to++ = *from++;  
  
*to = '\0';
```

使用联合

联合是C语言中最不寻常的特性之一。联合主要用于某些高级编程领域，在那些领域中，我们常常需要将不同类型的数据保存在同一个存储区域中。比如，假定我们定义了一个变量x，并希望其中既可以存储字符，也可以存储浮点数或者整数，那么我们就可以定义一个联合。假定我们定义的联合名为mixed，则相应的定义语句如下：

```
union mixed  
{  
    char c;  
    float f;  
    int i;  
};
```

定义联合的语句格式与定义结构的语句格式是完全相同的，只不过使用关键字union替换了关键字struct。联合与结构最主要的区别在于其存储空间的分配方式。如果像下面的语句那样声明一个union mixed类型的变量：

```
union mixed x;
```

那么最终的变量并不包含三个独立的成员变量c、f和i。实际上，该变量只包含一个成员变量，但是我们可以使用c、f或者i这三个名字引用它。通过这种方式，我们既可以在变量x中存储字符类型的数值，也可以存放浮点数值或者整数值。但是我们无法同时在其中存放三个值（甚至两个值也不可以）。我们可以使用下面的语句在联合中存放一个字符值：

```
x.c = 'K';
```

我们随后可以使用同样的记号x.c将存放在其中的字符取出。因此，假定我们要将这个字符在终端中显示出来，可以使用如下的语句：

```
printf ("Character = %c\n", x.c);
```

为了将浮点数值保存在联合变量x中，我们可以使用记号x.f，如下所示：

```
x.f = 786.3869;
```

最后，如果需要在x中存储一个整型数，比如整型变量count除以2所得的结果，我们可以使用如下的语句：

```
x.i = count / 2;
```


因为x的字符成员、浮点成员和整型成员都引用了内存中的同一个位置，因此同一时刻我们只能在x中保存一个值。另外，在使用联合的时候，程序员应该记得使用与保存时同样的方式取出其中的值。

联合的成员变量在表达式求值的时候依然遵守C语言的类型规则。因此下面的表达式：

```
x.i / 2
```

将按照整数除法的规则进行，因为表达式中的两个操作数都是整数。

我们可以在联合中定义任意多的成员。C语言的编译器将保证为联合类型变量分配能够容纳其最大成员变量的存储空间。另外，如同数组一样，联合也可以作为结构的一个成员变量。在定义联合变量的时候，C语言不要求我们一定要给出联合类型的名字，我们可以直接定义联合变量。这一点和定义结构变量相同。另外，我们也可以定义指向联合的指针。

在定义联合变量的时候，我们可以选择初始化其中的某个成员变量。如果没有指定成员变量名字的话，那么初始化值将被赋给联合的第一个成员变量，比如下面的语句将联合变量x的第一个成员变量c的初始化值设为字符#。

```
union mixed x = { '#' };
```

我们也可以在初始化联合变量的时候给任意一个成员变量赋予初值，如下面的语句所示：

```
union mixed x = { .f = 123.456; };
```

这个语句把union mixed变量x的浮点成员f设置为123.456。

我们还可以用另外一个同类型的联合变量来初始化联合变量，如下面的语句所示：

```
void foo (union mixed x)
{
    union mixed y = x;
    ...
}
```

在上面的例子中，函数foo将其联合参数x的值赋给内部自动变量y。

使用联合我们可以建立一个存储不同类型数值的数组，请看下面的语句：

```
struct
{
    char *name;
    enum symbolType type;
    union
    {
        int i;
```

```

        float f;
        char c;
    } data;
} table [kTableEntries];

```

上面的语句定义了一个名为table的数组，该数组包含kTableEntries个数组元素。数组的每一个元素都是一个结构，该结构包含一个名为name的字符指针类型成员变量，名为type的枚举类型成员变量，还有一个名为data的联合类型成员变量。data中既可以存放一个int值，也可以存放float值或者char值。我们可以使用枚举类型成员type来保存data中所存放数据的真正类型。比如，如果data中包含一个整型值，我们可以给成员type赋值INTEGER（译者注：假定INTEGER是该枚举类型的一个合法的枚举值），如果data中包含float类型，则可以给type赋值FLOATING，如果data中包含char类型，则可以给type赋值CHARACTER。根据成员变量type中的信息，我们就能够推断出如何访问data中保存的数值。

为了将字符'#'保存到数组的第5号元素中，并对对应的type域赋值，以便表示该元素中存放着一个字符类型的值，我们可以使用下面的语句：

```

table[5].data.c = '#';
table[5].type = CHARACTER;

```

在遍历table数组成员的时候，我们可以根据每个元素中type成员变量的值，推断出每个元素中保存数值的实际类型。例如，下面的语句将数组中每个成员的名字和数值在终端上显示出来：

```

enum symbolType { INTEGER, FLOATING, CHARACTER };
...

for ( j = 0; j < kTableEntries; ++j ) {
    printf ("%s ", table[j].name);

    switch ( table[j].type ) {
        case INTEGER:
            printf ("%i\n", table[j].data.i);
            break;
        case FLOATING:
            printf ("%f\n", table[j].data.f);
            break;
        case CHARACTER:
            printf ("%c\n", table[j].data.c);
            break;
        default:
            printf ("Unknown type (%i), element %i\n", table[j].type, j );
            break;
    }
}

```


上面演示的这个结构，在实际应用中可以用来构造一个符号表，其中存放每个符号的名字、类型以及对应的数值（当然还可以再附加上符号的其他信息）。

逗号操作符

初看上去，读者可能不会认为逗号也是一个操作符。实际上，逗号操作符是所有C语言操作符中优先级最低的一个。在第5章“使用循环”中我们已经知道，for语句的三个部分（初始化表达式，循环条件以及循环表达式）中都可以包含使用逗号隔开的多个表达式，如下所示：

```
for ( i = 0, j = 100; i != 10; ++i, j -= 10 )
    ...
```

在上面的循环语句中，我们使用逗号操作符在循环开始之前将变量i和j的值分别初始化为0和100，另外，每当循环体执行一遍之后，我们都将i的值增加1，将j的值减去10。

在所有可以使用普通C语言表示式的地方，我们都可以使用逗号隔开的多个表达式。这些表达式按照从左到右的顺序求值。例如，在下面的语句中：

```
while ( i < 100 )
    sum += data[i], ++i;
```

每次循环体执行的时候，都将数组元素data[i]的值加到变量sum上，然后将i的值增加1。这里需要读者注意的是，我们不需要在循环体外面加上花括号，因为while的循环体实际上只有一条语句（但该语句包含两个表达式，之间用逗号隔开）。

C语言所有的表达式都有一个值，对于逗号表达式来说，其值等于最右面的表达式的值。

读者还要注意，函数调用语句或者声明语句中用于分隔参数的逗号只是C语言语法的一部分，它并不是我们这里介绍的逗号操作符。

类型修饰符

下面我们将介绍一些可以用于修饰变量声明语句的修饰符。这些修饰符用于告诉编译器该变量的使用方式，这样编译器可以利用这些信息产生更合适的机器指令。

register 修饰符

如果在函数中频繁使用某个特殊的变量，我们常常希望对于该变量的访问能够尽可能的快。一般说来，CPU访问寄存器中的数值速度最快，因此我们希望编译器能够将该变量放置到寄存器中。C语言允许我们在变量声明语句前面放置关键字register来说明这一点。下面是使用该关键字的几个例子：

```
register int index;  
register char *textPtr;
```

函数的局部变量和形式参数都可以被声明为register类型的变量。对于不同的计算机来说,可以增加register修饰符的数据类型是不同的。一般说来,C语言的基本数据类型和所有的指针类型都可以使用register关键字修饰。

即使我们使用了register关键字来修饰某个变量,该变量在运行的时候是否真的被放置到计算机的寄存器中将取决于特定的编译器。编译器完全有可能忽略这个关键字。

还有一点需要读者注意的是:我们不能将取地址运算符(&)作用于一个register类型的变量。除此之外,register类型的变量和普通的自动变量使用方式是完全相同的。

volatile 修饰符

volatile修饰符的意思与const刚好相反。该关键字告诉编译器特定的变量值在程序运行的时候将会发生改变。我们通常使用这个关键字来告诉编译器,不要对某个变量的存取指令进行优化,例如对多余的赋值语句,或者在变量值明显没有改变的时候重复检查其值的语句。计算机的I/O端口输入输出是这方面一个很好的例子。假定我们在程序中使用一个名为outPort的变量指向某个实际I/O端口,如果我们想要向该端口输出两个字符,比如说O和N,则可以使用下面的语句:

```
*outPort = 'O';  
*outPort = 'N';
```

某个智能的C语言编译器可能会注意到这两个赋值语句实际上是连续向同一个地址赋值,而在此其间outPort的值没有被访问过,因此编译器从最终生成的代码中删除了第一条语句。为了防止这种问题,我们就可以使用volatile关键字修饰outPort变量,如下所示:

```
volatile char *outPort;
```

restrict 修饰符

restrict修饰符和register修饰符相类似,同样用于告诉编译器某些可以用于代码优化的信息。这个关键字告诉编译器,它所修饰的指针变量是唯一一个指向某个变量的指针(无论是直接还是间接),也就是说,该指针所指向的值,在整个作用域内不会再被其他的指针引用。

例如,下面的声明语句:

```
int * restrict intPtrA;  
int * restrict intPtrB;
```

告诉编译器,在intPtrA和intPtrB的作用域范围内,这两个指针始终不会指向同一个变量。如果我们使用这两个指针指向某个数组中的元素,那么它们的存取范围是互相不重叠的。

命令行参数

很多程序在运行的时候需要用户从终端窗口输入少量信息。比如需要计算的三角形数的编号，或者需要在字典中查找的单词等等。

实际上，我们可以在启动程序的时候一并提供这些信息，而不是等待程序提示我们输入。这就是所谓的命名行参数。

我们前面已经提到过，main函数与程序中其他函数唯一不同的地方就是它的函数名。main函数是一个程序的开始运行点。实际上，main函数在程序运行的时候由C语言系统（也被称为C运行时刻环境）所调用。运行时刻环境调用main函数的方式与main函数调用其它函数的方式是完全相同的。当main函数执行完成之后，程序流程实际上回到了运行时刻环境中，而运行时刻环境也由此得知我们的程序已经执行完成。

当运行时刻环境调用我们的main函数时，实际上有两个参数被传递给了main函数。按照传统，第一个参数是一个整数数，通常被称为argc（*argument count*的缩写），用于代表命令行上用户输入的参数个数。传递给main函数的第二个参数是一个字符指针数组，通常被命名为argv（*argument vector*的缩写）。在该字符指针数组中包含有argc+1个元素。通常情况下，该数组中的第一个字符指针指向当前运行程序的名字，如果操作系统没有提供这个名字的话，那么该字符指针则指向空字符NULL。字符数组中的其它指针分别指向程序运行时命令行上的其它参数。该字符数组的最后一个元素，也就是argv[argc]，其值总是NULL。

为了访问程序的命令行参数，我们必须按照新的形式声明main函数，以表示我们的程序能够接收这两个参数。该声明形式通常如下：

```
int main (int argc, char *argv[])
{
    ...
}
```

读者应该牢记，argv是一个字符指针的数组（而不是字符数组）。下面我们以前面的程序10.10为例来演示命令行参数的使用方法。该程序从字典中查找某个单词并将该单词对应的意思输出到终端窗口。我们可以使用命令行参数在启动程序的时候就给出我们需要查找的单词，如下面的命令行所示：

```
lookup aerie
```

这样程序就不需要提醒用户输入需要查找的单词了。

当我们输入如上所示的命令并回车之后,系统将启动程序lookup,并将字符串"aerie"作为参数数组中的第二个元素(也就是argv[1])传递给main函数。读者应该还记得,参数数组中的第一个元素(argv[0])代表当前运行程序的名字,也就是lookup。

修改后的主程序如下所示:

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

int main (int argc, char *argv[])
{
    const struct entry dictionary[100] =
        { { "aardvark", "a burrowing African mammal" },
          { "abyss", "a bottomless pit" },
          { "acumen", "mentally sharp; keen" },
          { "addle", "to become confused" },
          { "aerie", "a high nest" },
          { "affix", "to append; attach" },
          { "agar", "a jelly made from seaweed" },
          { "ahoy", "a nautical call of greeting" },
          { "aigrette", "an ornamental cluster of feathers" },
          { "ajar", "partially opened" } };

    int entries = 10;
    int entryNumber;
    int lookup (const struct entry dictionary [], const char search[],
                const int entries);

    if ( argc != 2 )
    {
        fprintf (stderr, "No word typed on the command line.\n");
        return EXIT_FAILURE;
    }

    entryNumber = lookup (dictionary, argv[1], entries);

    if ( entryNumber != -1 )
        printf ("%s\n", dictionary[entryNumber].definition);
    else
        printf ("Sorry, %s is not in my dictionary.\n", argv[1]);

    return EXIT_SUCCESS;
}
```

主程序首先检查用户是否在命令行上输入了需要查找的单词。如果没有输入或者输入多于一个单词的话,参数argc的值将不等于2。这时我们的程序将打印错误信息并返回值EXIT_FAILURE,用于表示程序的执行没有成功。

如果argc的值等于2，主程序将调用lookup函数，并将字符指针数组元素argv[1]的值作为参数传递给它。如果lookup函数找到了对应的单词，它就将相应的解释打印出来。

作为另外一个例子，我们将程序16.3改造为使用命令行参数，改造的结果如程序17.1所示。该程序用于拷贝文件，源文件的名称和目标文件的名称都将通过命令行参数指定，而不是提示用户输入。

程序17.1 使用命令行参数的文件拷贝程序

```
// 拷贝文件的程序 Ver 2
#include <stdio.h>

int main (int argc, char *argv[])
{
    FILE *in, *out;
    int c;

    if ( argc != 3 ) {
        fprintf (stderr, "Need two files names\n");
        return 1;
    }
    if ( (in = fopen (argv[1], "r")) == NULL ) {
        fprintf (stderr, "Can't read %s.\n", argv[1]);
        return 2;
    }
    if ( (out = fopen (argv[2], "w")) == NULL ) {
        fprintf (stderr, "Can't write %s.\n", argv[2]);
        return 3;
    }

    while ( (c = getc (in)) != EOF )
        putc (c, out);

    printf ("File has been copied.\n");
    fclose (in);
    fclose (out);

    return 0;
}
```

程序首先检查参数argc的值，确保用户输入了足够的参数。随后，程序将argv[1]作为源文件的名称，argv[2]作为目标文件的名称，并将源文件按照读模式打开，而将目标文件按照写模式打开。在确保文件打开操作成功完成之后，和以前的程序类似，该程序将源文件中的字符逐个拷贝到目标文件中。

读者需要注意，上面的程序有四条退出路径，它们分别是：不正确的命令行参数个数、源文件打开失败、目标文件打开失败和程序正常结束。如果我们想要使用程序的返回值来表示程序执行情况的话，我们一定要在每一个退出路径上明确的指定返回值。否则的话，main函数的返回值将是不确定的。

如果程序17.1生成的可执行程序名为copyf，而我们使用如下的命令行启动该程序的话，那么在main函数开始执行的时候，字符指针数组argv的内容将如图17.1所示。

```
copyf foo fool
```

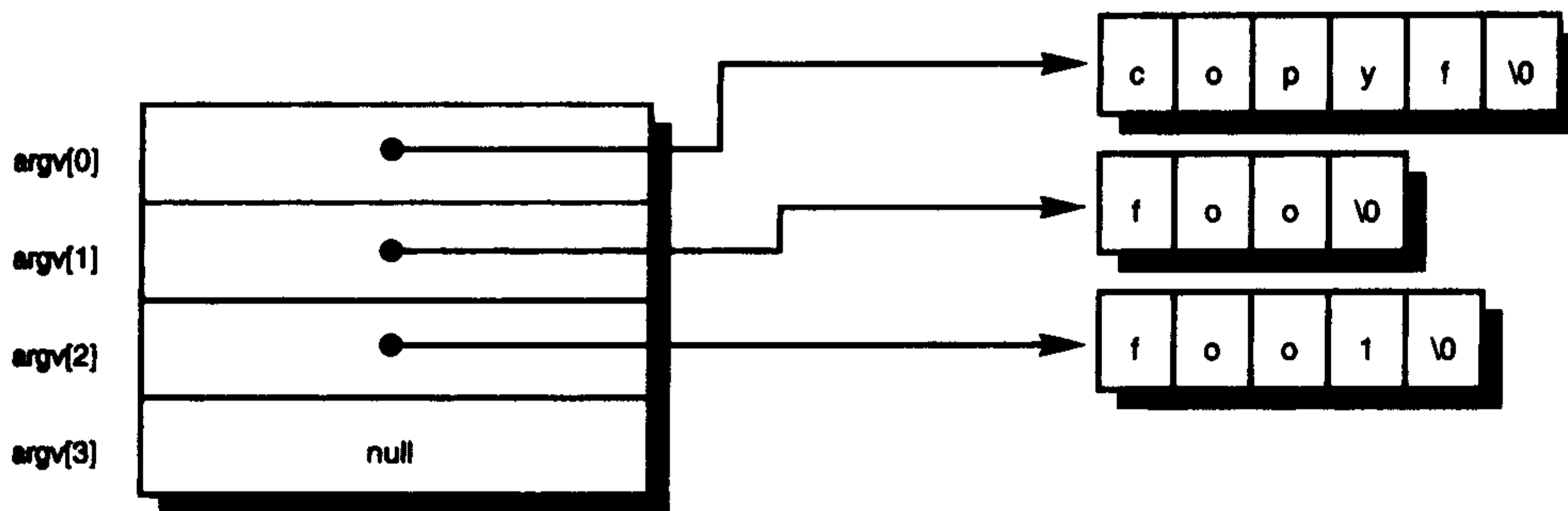


图 17.1 copyf 程序参数数组 argv 的内容

读者应该记住，命令行参数总是被作为字符串来存储的。因此，如果我们执行某个用于计算指数的程序power，并给其传递两个命令行参数2和6，如下所示：

```
power 2 6
```

那么字符指针argv[1]将指向字符串"2"，而字符数组argv[2]将指向字符串"16"。如果我们想要将这两个参数当作数字来处理（指数程序power肯定会如此），那么必须由程序自己来完成从字符串到数字的转换工作。C语言库函数sscanf、atof、atoi、strtod或者strtol都可以完成这些工作。关于这些函数的具体情况，请参见附录B“C语言标准库”。

动态内存分配

当我们在程序中定义某个变量的时候——无论该变量是简单类型、数组类型或者是一个结构，编译器都将在计算机的内存中为该变量分配一小块内存空间，用于存储该变量的值。一般来说，编译器分配的内存大小正好能够容纳该变量。

在很多情况下，我们常常需要在程序运行的时候动态的分配某些内存。比如我们要编写一个程序，将某个文件中的一组数据读入到内存中的数组中。然而，只有在程序开始执行之后，我们才能知道文件中包含的数据个数。面对这种情况，我们有三种解决办法：

- 在编写程序的时候，定义一个足够大的数组，该数组能够容纳文件中可能包含的最大数目的数据；
- 使用变量长度的数组，在运行时刻决定数组的长度；
- 使用C语言的动态内存分配函数在程序运行时为该数组动态分配内存。

使用第一种方法，我们需要定义一个能够容纳最大可能数据数量的数组，相应的程序如下所示：

```
#define kMaxElements 1000

struct dataEntry dataArray [kMaxElements];
```

采用这种办法，如果文件中包含的数据数量小于1000，那么我们的程序就可以正常的运行。然后一旦数据的数目超过了1000，我们就必须修改程序，并重新编译它。实际上无论我们选择多么大的数目，文件中的数据数目总是有可能超过该数目，因此在将来我们还是需要面对同样的问题。

如果我们在程序开始读取文件的时候就能够知道文件中数据的数量（比如通过计算文件大小），那么当采用第二种办法是很方便的。例如，我们可以按照如下的方式定义数组：

```
struct dataEntry dataArray [dataItems];
```

在上面的语句中，我们假定变量dataItems中保存着数据文件中数据的个数。

使用第三种办法——动态内存分配，我们可以在需要内存的时候再去分配它。也就是说，这种方法允许我们边执行程序边分配内存。为了使用动态内存分配，我们还需要学习三个新的C语言库函数和一个操作符。

malloc 和 calloc 函数

在C语言库函数中有两个函数malloc和calloc可以用来动态分配内存。其中，calloc函数接收两个参数，一个用于指定需要分配的元素个数，另外一个用于指定每个元素的大小。该函数的返回值是一个指向所分配的内存区域开始地址的指针。另外，所分配的内存的内容将被自动重置为0。

calloc函数返回的指针类型是void*，该类型是C语言中通用的指针类型。在将数值保存到这个指针指向的内存中之前，我们常常需要使用类型转换操作符将该指针转换为合适的指针类型。

malloc函数的工作方式与calloc函数非常类似，不过该函数只接收一个参数——以字节计算需要分配的内存数目，另外，malloc函数并不会自动将所分配内存的内容清零。

这两个动态内存分配函数的原型声明在标准头文件<stdlib.h>中,如果我们需要在程序中使用这两个函数的话,必须在源程序中包含这个头文件。

sizeof 操作符

为了确定某个具体的数据类型所需的存储空间大小,并且希望代码可以在多种计算机环境中移植的话,我们应该使用C语言的sizeof操作符。sizeof操作符返回特定的数据类型或者变量以字节为单位所占用的内存大小。sizeof操作符的操作数可以是变量、数组名、基本数据类型的名字、扩展数据类型的名字或者表达式。比如,下面的表达式:

```
sizeof(int)
```

代表保存一个整型数所需的内存字节数。在Pentium 4类型CPU的计算机上,该表达式的值等于4,因为这种计算机上的整型数占据32位。如果x是一个整型数组,其中包含100个元素,那么表达式

```
sizeof(x)
```

的值将是储存100个整型数所需的内存字节数(在Pentium 4的机器上,该数值等于400)。下面的表达式:

```
sizeof(struct dataEntry)
```

代表储存结构dataEntry需要的内存字节数。如果我们定义data是一个struct dataEntry类型的结构数组,那么表达式:

```
sizeof (data) / sizeof (struct dataEntry)
```

将代表数组data中能够容纳的元素个数(注意:这里的data数组必须在当前范围内有定义,不能是一个形式参数或者外部变量引用)。下面的表达式:

```
sizeof (data) / sizeof (data[0])
```

其结果与上面的表达式是相同的。下面的宏定义语句正是利用了这一点技巧来计算某个数组中包含的元素个数。

```
#define ELEMENTS(x) (sizeof(x) / sizeof(x[0]))
```

使用这个宏定义,我们可以编写类似下面的代码:

```
if ( i >= ELEMENTS (data) )  
...
```

或者如下的代码:

```
for ( i = 0; i < ELEMENTS (data); ++i )  
...
```

读者需要记住, sizeof是一个操作符,而不是一个函数调用,虽然它看上去很像一个函数。绝大部分时候, sizeof表达式的值在编译时就已经被确定了,编译器将使用该表达式的结果代替具体的表达式,而在程序运行的时候,该值将被当作一个常量。不过,如果我们使用变量长度数组作为该操作符的参数,那么表达式的值只有在运行时刻才能被确定。

只要有可能，我们就应该使用sizeof操作符，而避免在代码中引入硬编码的常数。

让我们回到动态内存分配的话题上来。假定我们在程序中需要分配一块能够容纳1000个整数的内存，我们可以使用如下的方式调用calloc语句：

```
#include <stdlib.h>
...
int *intPtr;
...
intPtr = (int *) calloc (sizeof (int), 1000);
```

如果使用malloc函数，则可以使用如下的形式：

```
intPtr = (int *) malloc (1000 * sizeof (int));
```

读者需要记住，malloc函数和calloc函数返回的都是通用类型的指针void*，因此我们程序应该将其转换为合适的类型。在上面的例子中，我们将这两个函数的返回值转换为int类型的指针，然后将其值赋给变量intPtr。

如果在调用malloc或者calloc函数的时候，要求分配的内存数目比当前系统所能提供的数目还要多的话，这两个函数将返回空指针null。因此，当我们使用这两个函数的时候，一定要记得对于函数的返回值进行检查，以确保内存分配操作成功完成。

下面一段代码为1000个整数分配了存储空间，并对返回的指针值进行了检查。如果内存分配操作失败的话，该程序在标准错误输出上打印出一段错误消息并退出运行。

```
#include <stdlib.h>
#include <stdio.h>
...
int *intPtr;
...
intPtr = (int *) calloc (sizeof (int), 1000);

if ( intPtr == NULL )
{
    fprintf (stderr, "calloc failed\n");
    exit (EXIT_FAILURE);
}
```

如果内存分配操作成功的话，那么我们可以将intPtr当作一个指向包含1000元素的整型数组开始位置的整型指针。因此，如果我们想要将这1000个元素的值全部设置为-1，可以使用如下的语句（假定变量p是一个整型数指针）：

```
for ( p = intPtr; p < intPtr + 1000; ++p )
    *p = -1;
```

为了分配能够容纳n个dataEntry结构的一块内存，我们首先需要定义一个指向该结构类型的指针，如下所示：

```
struct dataEntry *dataPtr;
```

随后，我们就可以调用calloc函数，分配所要求的内存了。具体代码如下所示：

```
dataPtr = (struct dataEntry *) calloc (n, sizeof (struct dataEntry));
```

上面语句的实际执行过程如下：

1. 系统首先调用calloc函数，并向其传递两个参数，一个是需要分配的元素个数，也就是n，另外一个是一个元素的大小；
2. calloc函数返回一个指向所分配内存的指针，如果无法分配所需的内存，那么calloc函数将返回一个空指针；
3. 返回的指针将被类型转换成为struct dataEntry*类型，然后再赋值给变量dataPtr。

这里我们再次强调一下，程序中应该对dataPtr的值进行检查，以确保内存分配操作成功完成。如果该指针的值不为Null，那么说明分配操作是成功的，随后我们就可以将该指针看作是一个指向包含n个dataEntry类型元素的数组的指针。比如，如果dataEntry结构包含一个名为index的整型变量成员，那么我们可以使用如下的语句将数值100赋给dataPtr指向的元素：

```
dataPtr->index = 100;
```

free 函数

当我们使用完毕calloc函数或者malloc函数分配的内存之后，我们应当调用free函数将这块内存归还给系统。free函数接收一个指向需要释放的内存的指针，也就是malloc函数或者calloc函数返回的指针。因此下面的语句：

```
free (dataPtr);
```

将前面calloc调用所分配的内存归还给系统（当然，dataPtr指针必须指向所分配的内存的开始位置）。

free函数没有返回值。

free函数所释放的内存可以供malloc函数或者calloc函数再次分配。对于那些所需内存如果一次全部分配将超过计算机所能提供内存的程序来说，这一特性是非常有用的。另外，在调用free函数的时候，传递给它的指针应该是前面调用malloc或者calloc函数所返回的指针。其他的指针值将是无效的。

动态内存分配技术在处理类似链表这样的动态数据结构时是非常有价值的。如果我们需要在链表中新增一个数据项，我们可以调用malloc函数或者calloc函数动态分配这样一块内存，然后将其附加在链表后面。例如，假定指针listEnd指向一个节点类型为struct entry类型的单向链表结尾，其中struct entry的定义如下所示：


```
struct entry
{
    int value;
    struct entry *next;
};
```

那么下面的函数addEntry将在链表的结尾处增加一个新的数据项。(原文此处似乎有误, listEnd指针并没有实际的用处) 这个函数接收一个指向链表开始的指针参数。

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

// 在链表结尾处新增一个数据项

struct entry *addEntry (struct entry *listPtr)
{
    // 寻找链表尾
    while ( listPtr->next != NULL )
        listPtr = listPtr->next;

    // 为新元素分配空间
    listPtr->next = (struct entry *) malloc (sizeof (struct entry));

    // 在新的链表尾添加null
    if ( listPtr->next != NULL )
        (listPtr->next)->next = (struct entry *) NULL;

    return listPtr->next;
}
```

如果内存分配操作成功的话, 我们将新分配的链表节点元素的next成员置为null, 以表示这是链表的结尾。

addEntry函数的返回值是一个指向新分配节点的指针。如果内存分配操作失败的话, addEntry函数将返回null。读者可以自己画一个链表图来辅助理解上面程序的工作流程。

C语言标准库中还有一个名为realloc的函数, 它可以对先前分配的内存块的大小进行调整。如果读者想要了解这个函数的更多细节, 可以参阅附录B。

到本章为止, 我们对于C语言本身特性的讨论就结束了。在下一章“调试程序”中, 我们将学到调试C语言程序所需的一些技术。其中之一就是使用预处理器。其他技术涉及到一类特殊的工具——交互式调试器的使用。

Debugging Programs

调试程序

本章将讲述两种可以用于调试C语言程序的技术。第一种是使用预处理器语句在程序中条件包含调试语句。第二种是使用交互式调试器。本章我们使用的调试器是非常流行的gdb。即使读者在开发C语言程序的时候使用其他调试器（如dbx或者IDE内建的调试器），这里讲述的内容大部分也都适用。

使用预处理器嵌入调试语句

在第13章“预处理器”中我们已经提到过，条件编译可以帮助我们调试程序。我们可以通过预处理器将调试语句加入到程序中。通过组合使用#ifdef语句，我们可以根据需要启用或者禁用程序中的调试语句。为了演示这个技术，我们专门设计了程序18.1。该程序读入3个整数，并打印出它们的和。如果我们编译程序的时候定义了符号DEBUG，那么相应的调试语句（也就是输出到标准错误的那些语句）将和程序的其他部分一起编译，否则，这些调试语句将被编译器忽略。

程序18.1 使用预处理器增加调试语句

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

int process (int i, int j, int k)
{
    return i + j + k;
}
```

程序18.1 续

```
int main (void)
{
    int i, j, k, nread;
    nread = scanf ("%d %d %d", &i, &j, &k);

    #ifdef DEBUG
        fprintf (stderr, "Number of integers read = %i\n", nread);
        fprintf (stderr, "i = %i, j = %i, k = %i\n", i, j, k);
    #endif

    printf ("%i\n", process (i, j, k));
    return 0;
}
```

程序18.1 输出

```
1 2 3
Number of integers read = 3
i = 1, j = 2, k = 3
6
```

程序18.1 输出（再次运行）

```
1 2 e
Number of integers read = 2
i = 1, j = 2, k = 0
3
```

这里提醒读者注意，第二次运行的时候，程序打印出来的变量k的值是个随机数，并不一定是0。因为我们在程序中没有初始化该变量，scanf也没有给该变量赋值。

下面的语句：

```
#ifdef DEBUG
    fprintf (stderr, "Number of integers read = %i\n", nread);
    fprintf (stderr, "i = %i, j = %i, k = %i\n", i, j, k);
#endif
```

在编译的时候将首先被预处理器分析。如果符号DEBUG在前面已经定义过了（比如使用#define语句），那么预处理器就将#ifdef和#endif之间的语句送给编译器进行编译。

如果我们还没有定义符号DEBUG的话，那么预处理器将从程序中删除这些语句（这样的话编译器将无法看到这些语句）。读者可以看到，这些调试语句将程序从终端读入的值打印出来。在第二次运行的时候，因为我们输入了一个无效字符e，因此调试语句将打印出错误信息来提醒我们。如果我们需要关掉这些调试语句的话，只需要将下面一行语句：

```
#define DEBUG
```

从程序中删除即可。这样，编译器就不再处理这些调试用的fprintf语句了。虽然程序18.1很短，读者甚至可能觉得根本没有必要这么麻烦，实际上对于大的程序（比如说上百行），仅仅修改程序中的一行代码就可以快速关闭和打开调试语句是非常方便的。

我们也可以在编译的时候通过命令行来控制调试语句的条件编译。如果读者使用的编译器是gcc，那么下面的语句：

```
gcc -D DEBUG debug.c
```

将定义符号DEBUG，然后编译文件debug.c。这相当于我们在程序debug.c的开始加上如下一行代码：

```
#define DEBUG
```

下面我们来看一个长一点的程序。程序18.2接收两个命令行参数，每个参数将被转化为一个整型数并分别赋值给变量argc1和argc2。为了将命令行上的字符串参数转化为整型数，我们使用了C语言的标准库函数atoi。这个函数接收一个字符串参数，其返回值就是该字符串对应的整数。atoi函数的原型声明包含在文件<stdlib.h>中。

在将命令行参数转化为整数之后，程序18.2调用了函数process，并将两个整型数作为参数传递给它。process函数的返回值是这两个整数的乘积。读者可以看到，当我们定义符号DEBUG之后，程序在运行的时候将会打印出很多调试信息，而当我们不定义符号DEBUG时，程序只输出最后的计算结果。

程序18.2 编译调试语句

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

int process (int i1, int i2)
{
    int val;
    #ifdef DEBUG
        fprintf (stderr, "process (%i, %i)\n", i1, i2);
    #endif

    val = i1 * i2;
    #ifdef DEBUG
        fprintf (stderr, "return %i\n", val);
    #endif
    return val;
}
```


程序18.2 续

```

int main (int argc, char *argv[])
{
    int arg1 = 0, arg2 = 0;

    if (argc > 1)
        arg1 = atoi (argv[1]);
    if (argc == 3)
        arg2 = atoi (argv[2]);

    #ifdef DEBUG
    fprintf (stderr, "processed %i arguments\n", argc - 1);
    fprintf (stderr, "arg1 = %i, arg2 = %i\n", arg1, arg2);
    #endif

    printf ("%i\n", process (arg1, arg2));
    return 0;
}

```

程序18.2 输出

```

$ gcc -D DEBUG p18-2.c 定义符号DEBUG并编译程序
$ a.out 5 10
processed 2 arguments
arg1 = 5, arg2 = 10
process (5, 10)
return 50
50

```

程序18.2 输出（再次运行）

```

$ gcc p18-2.c 不定义符号DEBUG编译程序
$ a.out 2 5
10

```

当我们开发完毕程序并准备发布它的时候，只要我们不定义DEBUG符号而编译程序，我们可以保留程序中的调试语句而不影响最终生成的可执行代码。如果程序在发布之后出现了问题，我们就可以重新打开这些调试语句，观察程序运行中到底什么地方出了问题。

前面嵌入调试语句的方法有一点不太让人满意，那就是最终的程序看上去有些臃肿，不容易阅读。我们可以通过更巧妙的使用预处理器来避免这一点。我们可以定义一个接收可变输出参数的宏，用来输出调试信息。该宏定义语句如下所示：

```
#define DEBUG(fmt, ...) fprintf (stderr, fmt, __VA_ARGS__)
```

这样我们就可以用它来替代普通的fprintf语句了，如下所示：

```
DEBUG ("process (%i, %i)\n", i1, i2);
```

这个语句将被预处理器展开为如下的形式：

```
fprintf (stderr, "process (%i, %i)\n", i1, i2);
```

我们可以在程序中使用上述形式的DEBUG宏来书写调试语句，这样最终的程序看上去就比较整洁了。请看程序18.3。

程序18.3 定义调试宏DEBUG

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

#define DEBUG(fmt, ...) fprintf (stderr, fmt, __VA_ARGS__)

int process (int i1, int i2)
{
    int val;
    DEBUG ("process (%i, %i)\n", i1, i2);
    val = i1 * i2;
    DEBUG ("return %i\n", val);
    return val;
}

int main (int argc, char *argv[])
{
    int arg1 = 0, arg2 = 0;

    if (argc > 1)
        arg1 = atoi (argv[1]);
    if (argc == 3)
        arg2 = atoi (argv[2]);

    DEBUG ("processed %i arguments\n", argc - 1);
    DEBUG ("arg1 = %i, arg2 = %i\n", arg1, arg2);
    printf ("%d\n", process (arg1, arg2));
    return 0;
}
```


程序18.3 输出

```

$ gcc pre3.c
$ a.out 8 12
processed 2 arguments
arg1 = 8, arg2 = 12
process (8, 12)
return 96
96
    
```

读者可以看到，这下子程序看上去清晰多了。如果不再需要这些调试语句，我们可以按照如下的方式重新定义宏DEBUG：

```
#define DEBUG(fmt, ...)
```

上面的语句告诉预处理器宏DEBUG对应空字符串，这样所有的调试语句都将被替换为空语句。

我们还可以再扩充一下上面的调试语句，使得我们既能够在编译的时候，也能够在运行的时候控制调试语句的运行。我们首先定义一个全局变量Debug，用来代表程序的调试级别。只有DEBUG语句的级别小于或者等于程序的调试级别，这些DEBUG语句才打印自己的输出信息。这样一来，每个调试语句都将接收两个参数，第一个代表该调试语句的级别。这些调试语句的形式如下所示：

```

DEBUG (1, "processed data\n");
DEBUG (3, "number of elements = %i\n", nelems)
    
```

如果程序的调试级别是1或者2，那么第一个DEBUG语句将打印出其信息。只有程序的调试级别大于等于3，两个调试语句才一起输出信息。我们可以在程序运行的时候通过命令行参数来指定其调试级别，如下所示：

```

a.out d1 将程序的调试级别设为1
a.out -d3 将程序的调试级别设为3
    
```

带有调试级别的DEBUG宏的定义如下所示：

```

#define DEBUG(level, fmt, ...) \
    if (Debug >= level) \
        fprintf (stderr, fmt, __VA_ARGS__)
    
```

按照上述定义，下面的调试语句：

```
DEBUG (3, "number of elements = %i\n", nelems);
```

将被展开为如下的形式：

```

if (Debug >= 3)
    fprintf (stderr, "number of elements = %i\n", nelems);
    
```

再次强调一下，如果我们将DEBUG宏定义为空字符串，那么上述语句将展开为空语句。

下面的宏定义综合了我们上面介绍的各种技术，而且还能够让我们从编译的命令行上对其进行控制：

```
#ifndef DEBON
# define DEBUG(level, fmt, ...) \
    if (Debug >= level) \
        fprintf (stderr, fmt, __VA_ARGS__)
#else
# define DEBUG(level, fmt, ...)
#endif
```

当编译器编译带有上述宏定义的程序时（比如我们将上述定义语句放置到头文件中，然后让程序包含该头文件），我们就可以在命令行上控制程序中的调试语句了。例如，如果我们按照下面的形式编译程序：

```
$ gcc prog.c
```

那么所有的调试语句都将展开为空语句，因为预处理器按照`#else`部分的`DEBUG`定义来处理调试语句。而如果我们按照如下的形式编译程序：

```
$ gcc -D DEBON prog.c
```

那么所有的调试语句都将展开为包含调试级别的`fprintf`语句。

如果我们将调试语句编译进最后的可执行程序中，那么在运行程序的时候我们就可以利用调试级别来控制这些调试语句了。比如我们可以通过如下的方式指定程序的调试级别为3：

```
$ a.out -d3
```

当然为了调试级别能够最终起作用，我们的程序还应该对命令行进行分析，并将调试级别保存到全局变量`Debug`中。按照上面的命令行，程序中只有那些调试级别小于等于3的调试语句才打印出它们的信息。

一般说来，命令行`a.out -d0`将程序的调试级别设为0，这样的话，即使我们的程序包含了调试语句，这些调试语句也不会工作。

综上所述，我们建立了一个两层的调试方案：第一层用于控制是否将调试代码编译到最终的可执行代码中；一旦最终可执行代码包含了调试语句，我们可以在第二层上控制哪些调试语句输出其信息。

使用 gdb 调试程序

`gdb`是一个功能非常强大的交互式调试器，它经常用来调试使用GNU C编译器`gcc`编译出来的可执行程序。使用`gdb`，我们可以在任意的位置暂停程序，显示和设置程序中变量的值，然后继续运行程序。我们可以使用`gdb`跟踪程序的运行过程，甚至是逐行跟踪。`gdb`也可以在程序发生了内核转储的时候用于判断发生问题的位置。所谓内核转储，就是当程序在运行中出现了某些不正常情况（例如除0、数组越界等）的时候，操作系统将会自动终止程序的执行，并将程序当时对应的内存中的数据保存到文件中的过程。这个文件通常被称为内核转储文件¹。

¹读者使用的系统可能被设置为不生成内核转储文件，因为内核转储文件一般都很大。不能生成内核转储文件的另外一个常见原因是系统对于最大可生成文件尺寸的限制。我们可以使用 `ulimit` 命令来改变这个尺寸。

为了使用gdb的全部功能，我们在使用gcc编译程序的时候应该附加上-g选项。这个选项告诉编译器将一些额外的调试信息如变量和结构的类型、源文件的名字，代码对应的C语言源程序等保存在最终生成的可执行文件中。

为了演示调试器的使用，我们使用如下的程序18.4。该程序中有一个数组元素访问越界错误。

程序18.4 使用gdb调试的一个简单程序

```
#include <stdio.h>
int main (void)
{
    const int data[5] = {1, 2, 3, 4, 5};
    int i, sum;

    for (i = 0; i >= 0; ++i)
        sum += data[i];

    printf ("sum = %i\n", sum);
    return 0;
}
```

如果我们在Mac OS X系统上运行这个程序，将会得到如下所示的结果（在其他操作系统上显示的内容可能有所不同）：

```
$ a.out
Segmentation fault
```

下面我们将使用gdb来追踪这个错误。这个例子虽然是虚构的，但是它可以很好的说明问题。

首先，我们需要确保使用了-g选项编译源程序。随后我们可以启动gdb，并将可执行程序的名字（通常是a.out）传递给它。gdb在启动的时候将打印出很多介绍性的信息，如下所示：

```
$ gcc -g p18.4.c 使用选项-g重新编译程序
```

```
$ gdb a.out 启动调试器
```

```
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec 4 15:41:30 GMT 2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin".
Reading symbols for shared libraries .. done
```

（译者注：上面打印出的主要是版权信息。）

当gdb启动就绪，准备接收命令时，它将在终端上显示提示符号(gdb)。在我们这个简单的例子中，我们可以直接输入命令run告诉gdb启动程序。gdb将运行在命令行上指定的可执行程序，直到程序结束或者运行发生异常为止。

```
(gdb) run
Starting program: /Users/stevekochan/MySrc/c/a.out
Reading symbols for shared libraries . done

Program received signal EXC_BAD_ACCESS, Could not access memory.
0x00001d7c in main () at p18-4.c:9
9      sum += data[i];
(gdb)
```

读者看到，程序在运行的时候发生了一个异常（正像不在调试器中运行一样），但是这次程序还在gdb的控制之下。这样，通过gdb，我们就能够了解程序发生意外的时候，程序正在运行的位置以及相应变量的值等情况。

从gdb的输出我们可以看到，当错误发生的时候，我们的程序正在运行第9行，而这段代码正在访问一个无效的内存。当程序出现问题的时候，gdb将自动显示有问题的那行代码。为了更好的观察程序运行的环境，我们可以使用list命令显示有问题的那行代码周围10行代码的情况（上面5行，下面4行），命令执行情况如下所示：

```
(gdb) list 9
4      {
5          const int data[5] = {1, 2, 3, 4, 5};
6          int i, sum;
7
8          for (i = 0; i <= 4; ++i)
9              sum += data[i];
10
11         printf ("sum = %i\n", sum);
12
13         return 0;
(gdb)
```

我们还可以使用print命令来显示变量的值。下面我们使用该命令显示出错的那一刻变量sum的值：

```
(gdb) print sum
$1 = -1089203864
```

变量sum的值看上去明显有些不对头（在读者自己的计算机上，具体的值多半会有所不同）。gdb对显示过的值进行编号，这样随后我们就可以很方便的引用这些值。

下面我们看看下标变量i的值：

```
(gdb) print i
$2 = 232
```


啊，变量*i*的值肯定有问题！我们的数组中只有5个元素，而现在程序正在尝试访问第232个元素。在不同的系统上，具体出现问题的元素编号可能会有所不同，但是或迟或早，类似的问题一定会发生。

在退出gdb之前，我们再来看看数组data的内容。gdb可以以很美观的形式将数组或者结构的内容打印出来。

```
(gdb) print data 显示data数组的内容
$3 = {1, 2, 3, 4, 5}
(gdb) print data[0] 显示data数组第一个元素的值
$4 = 1
```

随后我们还将看到显示结构内容的例子。为了完成我们这个小小的范例，读者还需要学习如何退出gdb。使用quit命令就可以做到这一点。

```
(gdb) quit
The program is running. Exit anyway? (y or n) y
$
```

虽然我们的程序在运行过程中发生了一个错误，但是从技术角度严格的讲，该程序仍然在gdb中运行。错误只是使得我们的程序暂停运行而已。因此，gdb询问我们是否真的要退出运行。

查看和设置变量

gdb提供了两个基本的命令用来帮助我们处理变量。一个是我们前面已经见到过的print命令。另外一个命令允许我们设置变量的值，这就是set命令。set命令有很多选项，如果我们想要设置变量的值，需要使用var选项。请看下面的例子：

```
(gdb) set var i=5
(gdb) print i
$1 = 5
(gdb) set var i=i*2 我们可以使用任意合法的表达式
(gdb) print i
$2 = 10
(gdb) set var i=$1+20 我们可以使用gdb提供的编号变量
(gdb) print i
$3 = 25
```

为了访问某个变量，这个变量必须在当前函数的范围可见，而且程序必须是活跃的（也就是说程序还处在运行状态，而不是已经终止）。gdb中关于程序定位有三个重要的概念：当前行（类似于编辑器），当前文件（也就是程序的源文件）和当前函数。当我们启动gdb并且不指定内核转储文件的时候，gdb将main函数作为当前函数，包含main函数的那个源文件作为当前文件，当前行则是main函数的第一行。如果使用内核转储文件的话，那么当前文件、当前函数和当前行将根据发生问题的地方来设置。

如果set或者print命令中给出的变量名不是局部变量的话，那么gdb将在程序中寻找同名的外部变量。在我们前面给出的例子中，程序发生内存访问异常的位置是在main函数中，而变量*i*则是main函数的一个局部变量。

我们也可以以 `function::variable` 的形式来指定某个特定函数中的变量，如下所示：

```
(gdb) print main::i 显示main函数中的变量i
$4 = 25
(gdb) set var main::i=0 设置main函数中的变量i
```

如果我们尝试设置“不活跃”函数中变量的值（所谓不活跃，就是说该函数既不是当前执行的函数，也不是当前函数的调用者）（译者注：用专业术语来说：就是不在调用堆栈中），那么gdb将显示如下的错误信息：

```
No symbol "var" in current context. (在当前上下文环境中找不到符号var).
```

如果要显示全局变量的名字，可以使用“文件名::变量名”的形式。使用上面的形式将强迫gdb忽略那些同名的局部变量。

我们可以使用C语言的标准语法来显示结构或者联合的内容。例如，如果 `dataPtr` 是一个 `date` 结构类型的指针，那么命令 `print dataPtr->year` 将显示成员变量 `year` 的值。

如果直接使用结构变量或者联合变量的名字，那么gdb将打印出该结构或者联合所有成员变量的值。

我们也能够让gdb使用其他进制显示变量的数值，这一点可以通过给 `print` 命令附加选项 `/`，然后再跟上代表相应进制的字母来完成。另外，很多gdb命令都有自身的缩写形式。在下面的例子中，我们使用了 `print` 命令的缩写形式 `p`。

```
(gdb) set var i=35 将变量i设置为35
(gdb) p /x i      使用十六进制显示变量i的内容
$1 = 0x23
```

显示源文件

gdb提供了几个显示源文件内容的命令。这样在需要参考源程序的时候，我们就不需要打开专门的编辑器窗口了。

前面我们已经提到过，gdb中有当前行和当前文件的概念。我们可以使用 `list` 命令来显示当前行周围的行（`list` 命令可以缩写为 `l`）。当我们连续输入 `list` 命令时，gdb将以10行为单位逐步显示当前行后面的源代码。10行是gdb的一个缺省值，这个值可以用 `listsize` 命令重新设置。

如果我们需要显示某个行号范围内的源程序，可以在使用 `list` 命令的时候将它们作为参数，如下所示：

```
(gdb) list 10,15 显示第10行到第15行的内容
```


如果我们需要显示某个函数的内容，可以将该函数的名字作为list命令的参数，如下所示：

```
(gdb) list foo 显示函数foo中的内容
```

如果指定的函数定义在另外一个源文件中，那么gdb将当前源文件自动切换到那个源文件。使用命令info source，我们可以看到该源文件的名称。

如果我们在list命令后面加上一个+号，那么gdb将显示随后的10行代码（和输入list时完全相同），如果我们在list命令后面跟上-号，那么gdb将显示前面的10行代码。+号和-号后面还可以跟上一个数字，用于告诉gdb需要显示的行相对于当前行的距离。

控制程序的执行

显示源程序中的内容并不会影响程序的执行流程。为了能够控制程序的执行，我们还应该使用其他命令。我们前面已经介绍了两个控制程序运行的命令：run用于启动程序运行，quit用于使程序退出运行。

run命令后面可以跟上命令行参数，或者是重定向输入输出符号（<或者>），gdb可以很好的处理这些参数。如果随后再次使用不带参数的run命令，那么gdb将重用前面指定的参数。我们可以使用命令show args将当前的命令行参数打印出来。

插入断点

我们可以使用break命令给程序中插入断点。当程序在调试器中执行时，如果遇到断点，那么调试器将暂停程序的执行。程序一旦暂停下来，我们就可以发出各种指令如查看变量的值等，从而判断出程序当前的执行情况。

断点可以设置在程序的任何位置，只要给出文件名、函数名或者行号就可以。如果我们只给出行号，那么调试器将在当前文件的相应行上设置断点。如果我们只给出函数的名字，那么断点将设置在该函数的第一条可执行语句上。请看下面的例子：

```
(gdb) break 12 在第12行设置断点
Breakpoint 1 at 0x1da4: file mod1.c, line 12.
(gdb) break main 在main函数的开始处设置断点
Breakpoint 2 at 0x1d6c: file mod1.c, line 3.
(gdb) break mod2.c:foo 在源文件mod2.c的foo函数处设置断点
Breakpoint 3 at 0x1dd8: file mod2.c, line 4.
```

当程序执行遇到断点之后，gdb将暂停程序，打印出断点所在的行号，并将控制权重新交还给我们。这时我们就可以开始进行各种调试命令了，如：打印变量的内容，设置或者取消断点，等等。如果需要继续运行程序，我们应该使用命令continue，这个命令的简写形式为c。

单步执行

在控制程序执行流程方面，step是一个非常有用的命令。这个命令可以简写为s。使用这个命令，我们可以单步执行程序，也就是说，计算机一次执行一行源程序，然后将控制权重新交还给我们。如果我们给step命令后面跟上一个数字，那么调试器将执行该数目表示的源程序行数，然后再暂停执行。虽然C语言允许我们在一行中书写多个语句，但是gdb总是以行为单位来执行语句，也就是说，gdb有可能一次单步执行跳过多个C语言语句。在任何可以使用continue命令的地方（比如程序收到一个信号或者遇到一个断点），我们都可以使用step命令。

如果当前行包含有函数调用，那么step命令将把我们带到该函数中（只要该函数不是系统函数，调试器一般来说并不进入系统函数）。如果我们使用另外一个单步执行命令next，那么gdb将跳过当前行而不是进入到函数中。

下面我们给出程序18.5，读者可以尝试使用各种gdb命令来调试它。

程序18.5 使用gdb调试程序

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

struct date {
    int month;
    int day;
    int year;
};

struct date foo (struct date x)
{
    ++x.day;
    return x;
}

int main (void)
{
    struct date today = {10, 11, 2004};
    int array[5] = {1, 2, 3, 4, 5};
    struct date *newdate, foo ();
    char *string = "test string";
    int i = 3;

    newdate = (struct date *) malloc (sizeof (struct date));
    newdate->month = 11;
```


程序18.5 续

```

newdate->day = 15;
newdate->year = 2004;

today = foo (today);
free (newdate);

return 0;
}

```

根据所使用的操作系统和gdb版本，读者在调试的时候看到的显示信息可能和下面调试过程列出的信息有所不同。

程序18.5 gdb调试过程

```

$ gcc -g p18-5.c
$ gdb a.out
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec 4 15:41:30 GMT 2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin".
Reading symbols for shared libraries .. done
(gdb) list main
14
15 return x;
16 )
17
18 int main (void)
19 {
20 struct date today = {10, 11, 2004};
21 int array[5] = {1, 2, 3, 4, 5};
22 struct date *newdate, foo ();
23 char *string = "test string";
(gdb) break main
Breakpoint 1 at 0x1ce8: file p18-5.c, line 20.
(gdb) run
Starting program: /Users/stevekochan/MySrc/c/a.out
Reading symbols for shared libraries . done
Breakpoint 1, main () at p18-5.c:20
20 struct date today = {10, 11, 2004};
(gdb) step

```

在main函数中设置断点

开始执行程序

执行第20行

程序18.5 续

```

21 int array[5] = {1, 2, 3, 4, 5};
(gdb) print today
$1 = {
month = 10,
day = 11,
year = 2004
}
(gdb) print array          数组还没有被初始化
$2 = {-1881069176, -1880816132, -1880815740, -1880816132, -1880846287}
(gdb) step                执行下一行
23 char *string = "test string";
(gdb) print array          再次显示数组的内容
$3 = {1, 2, 3, 4, 5}       现在已经初始化了
(gdb) list 23,28
23 char *string = "test string";
24 int i = 3;
25
26 newdate = (struct date *) malloc (sizeof (struct date));
27 newdate->month = 11;
28 newdate->day = 15;
(gdb) step 5              再向后执行5行
29 newdate->year = 2004;
(gdb) print string
$4 = 0x1fd4 "test string"
(gdb) print string[1]
$5 = 101 'e'
(gdb) print array[i]      将变量i的值设置为3
$6 = 3
(gdb) print newdate       显示指针变量的值
$7 = (struct date *) 0x100140
(gdb) print newdate->month
$8 = 11
(gdb) print newdate->day + 1 可以使用任意的C语言表达式
$9 = 18
(gdb) print $7            访问gdb的编号变量
$10 = (struct date *) 0x100140
(gdb) info locals         显示所有局部变量的值
today = {
month = 10,
day = 11,
year = 2004
}
array = {1, 2, 3, 4, 5}
newdate = (struct date *) 0x100140
string = 0x1fd4 "test string"
    
```


程序18.5 续

```

i = 3
(gdb) break foo           在foo函数的开始处放置一个断点
Breakpoint 2 at 0x1c98: file p18-5.c, line 13.
(gdb) continue           继续执行
Continuing.
Breakpoint 2, foo (x={month = 10, day = 11, year = 2004}) at p18-5.c:13
13 ++x.day; 0x8e in foo:25: {
(gdb) print today         显示变量today的值
No symbol "today" in current context
(gdb) print main::today   显示main函数中变量today的值
$11 = {
  month = 10,
  day = 11,
  year = 2004
}
(gdb) step
15 return x;
(gdb) print x.day
$12 = 12
(gdb) continue
Continuing.
Program exited normally.
(gdb)
    
```

gdb有一个特点：当调试器遇到一个断点后，它显示的是下面将要执行的语句，而不是上一条刚刚执行过的语句（译者注：也就是说，断点是插在一行代码的前面）。因此当数组初始化语句被显示出来的时候，它实际上还没有被执行。再次单步执行一行代码，程序就将初始化这个数组。另外，带有初始化值的变量声明语句也被调试器当作可以执行的源代码行，实际上，编译器也确实为这样的变量声明语句生成相应的可执行代码。

列出和删除断点

一旦我们设置了断点，这些断点就将一直保持下去，除非我们退出调试器gdb或者明确的删除它。使用命令info break，我们可以看到当前设置的所有断点，如下所示：

```

(gdb) info break
Num Type Disp Enb Address What
1 breakpoint keep y 0x00001c9c in main at p18-5.c:20
2 breakpoint keep y 0x00001c4c in foo at p18-5.c:13
    
```

我们可以使用clear命令，后面再跟上行号来删除某一行上设置的断点。另外，如果要删除在某个函数开始处设置的断点，可以使用clear命令，后面跟上该函数的名字。请看下面的示例：

```

(gdb) clear 20 删除第20行的断点
Deleted breakpoint 1
(gdb) info break
Num Type Disp Enb Address What
2 breakpoint keep y 0x00001c4c in foo at p18-5.c:13
(gdb) clear foo 删除函数foo开始处的断点
Deleted breakpoint 2
(gdb) info break
No breakpoints or watchpoints.
(gdb)
    
```

查看调用堆栈

在很多时候，当程序在调试器中暂停执行时，我们需要知道暂停点在函数调用层次中的具体位置。在检查内核转储文件的时候，这一点尤其有用。gdb调试器的backtrace命令可以显示出当前的调用堆栈。backtrace命令的简写形式为bt。下面的例子取自程序18.5。

```

(gdb) break foo
Breakpoint 1 at 0x1c4c: file p18-5.c, line 13.
(gdb) run
Starting program: /Users/stevekochoan/MySrc/c/a.out
Reading symbols for shared libraries . done
Breakpoint 1, foo (x={month = 10, day = 11, year = 2004}) at p18-5.c:13
13 ++x.day;
(gdb) bt          打印调用堆栈
#0 foo (x={month = 10, day = 11, year = 2004}) at p18-5.c:13
#1 0x00001d48 in main () at p18-5.c:31
(gdb)
    
```

当调试器在foo函数入口处的断点暂停时，我们使用backtrace命令查看了当前的调用堆栈。我们可以看到调用堆栈中有两个函数foo和main。我们还看到，函数调用的参数值也被列了出来。gdb还提供了很多命令（如up, down, frame或者info args）帮助我们查看调用堆栈的各种信息，在这里就不一一详细介绍了。

调用函数和给数组、结构变量赋值

我们可以使用如下的形式在gdb中调用某个函数。

```

(gdb) print foo(*newdate) 使用newdate指针的内容作为参数调用函数foo
    
```



```

$13 = {
  month = 11,
  day = 16,
  year = 2004
}
(gdb)
    
```

函数foo的定义如程序18.5中所示。

我们也可以使用大花括号表达式给数组或者结构变量赋值，如下所示：

```

(gdb) print array
$14 = {1, 2, 3, 4, 5}
(gdb) set var array = {100, 200}
(gdb) print array
$15 = {100, 200, 0, 0}    所有没有明确指定的元素都将被初始化为0
(gdb) print today
$16 = {
  month = 10,
  day = 11,
  year = 2004
}
(gdb) set var today={8, 8, 2004}
(gdb) print today
$17 = {
  month = 8,
  day = 8,
  year = 2004
}
(gdb)
    
```

获取 gdb 的命令帮助

我们可以使用gdb内建的help命令来查看各种命令的分类及其帮助信息。

如果使用不带任何参数的help命令，gdb将列出所有的命令分类，如下所示：

```

(gdb) help
List of classes of commands:

aliases -- Aliases of other commands    命令的缩写
breakpoints -- Making program stop at certain points 断点命令
data -- Examining data    查看数据的相关命令
files -- Specifying and examining files 查看文件的命令
internals -- Maintenance commands 内部命令
obscure -- Obscure features    一些不常用的特性
running -- Running the program 运行程序的命令
stack -- Examining the stack    检查堆栈的命令
    
```

```

status -- Status inquiries      显示状态信息的命令
support -- Support facilities    支持工具
tracepoints -- Tracing of program execution without stopping the program
跟踪工具
user-defined -- User-defined commands 用户自定义的命令
    
```

我们可以输入help命令，后面再跟上想要查看的命令分类，比如断点。gdb的输出如下所示：（译者注：为了方便，我们将略去gdb输出中的英文，直接使用中文代替）

```

(gdb) help breakpoints
给程序设置断点
List of commands:
awatch -- 针对某个表达式设置观察点
break -- 在某行上或者某个函数入口点设置断点
catch -- 设置捕捉点捕捉事件
clear -- 清除某行上或者某个函数入口点的断点
commands -- 设置当遇到断点时应执行的命令
condition -- 只有当某个条件满足时才在某个断点暂停
delete -- 删除断点或者自动显示的表达式
disable -- 禁用断点
enable -- 启用断点
future-break -- 在表达式上设置断点
hbreak -- 设置需要硬件支持的断点
ignore -- 设置某个断点的忽略次数
rbreak -- 在所有符合指定的正则表达式的函数入口设置断点
rwatch -- 针对某个表达式设置观察点
save-breakpoints -- 将现有定义的所有断点保存为一个脚本文件
set exception-catch-type-regex - 根据正则表达式匹配异常对象的类型
set exception-throw-type-regex -根据正则表达式匹配异常捕捉对象的类型
show exception-catch-type-regex - 显示匹配异常对象的正则表达式
show exception-throw-type-regex - 显示匹配异常捕捉对象的正则表达式
tbreak -- 设置临时断点
tcatch -- 设置临时捕捉点
thbreak -- 设置一个需要硬件支持的临时断点
watch -- 针对某个表达式进行观察
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
    
```


我们也可以在help后面跟上某个命令的名字来显示该命令的帮助，如下所示：

(gdb) **help break**

在某行源代码或者某个函数入口处设置断点。

该命令的参数可以是行号、函数名或者*号后面跟上一个地址。

如果指定了行号，则在该行的开始处插入一个断点。

如果指定了函数名，则在该函数入口处插入一个断点。

如果指定了地址，则在该地址处插入一个断点。

如果没有指定参数，那么将使用当前的堆栈框架和执行地址。

可以在同一处设置多个断点，对于条件断点来说这个特性很有用。

break ... if <cond> 在断点上设置条件表达式

使用“help breakpoints”可以查看其他与断点相关的命令。

(gdb)

读者已经看到，gdb中内建了大量的帮助信息，我们应该尽可能的利用这些信息，以便更有效的使用调试器。

其他零碎的东西

因为篇幅的缘故，我们这里就不详细介绍调试器gdb的其他特性了。这些特性主要包括：

- 设置临时断点，临时断点只能暂停一次。
- 临时禁用和启用断点。
- 使用特定的格式将内存中的内容转储。
- 设置一个观察表达式，当该表达式的值发生变化时，程序暂停（例如某个变量的值发生了改变）。
- 当程序暂停的时候，固定的显示一组表达式的值。
- 设置自定义的“方便变量”

表18.1列出了本章讨论过的gdb命令。命令开始的**黑体字母**表示该命令的缩写。

表18.1 普通的gdb命令

命令	含义
处理源文件的相关命令	
<code>list [n]²</code>	显示第 n 行周围的源程序, 如果没有指定 n 则显示下面10行
<code>list m,n</code>	显示第 m 行到第 n 行之间的程序
<code>list +[n]</code>	显示当前行后面第 n 行周围的源程序, 如果没有指定 n 则显示下面10行
<code>list -[n]</code>	显示当前行前面第 n 行周围的源程序, 如果没有指定 n 则显示前面10行
<code>list func</code>	显示函数 $func$ 中的源程序
<code>listsize n</code>	指定 <code>list</code> 命令显示的源程序行数
<code>info source</code>	显示当前的源程序文件名
变量与表达式的相关命令	
<code>print /fmt expr</code>	根据 fmt 代表的格式显示表达式的值, 可用的格式有: d (十进制), u (无符号数), o (八进制), x (十六进制), c (字符), f (浮点数), t (二进制), a (地址)
<code>info locals</code>	显示当前函数中所有局部变量的值
<code>set var var=expr</code>	将变量 var 的值设置为表达式 $expr$ 的值
断点的相关命令	
<code>break n</code>	在第 n 行处设置断点
<code>break func</code>	在函数 $func$ 的入口处设置断点
<code>info break</code>	显示所有断点
<code>clear [n]</code>	删除第 n 行的断点, 如果没有指定 n , 则删除下一行的断点
<code>clear func</code>	删除在函数 $func$ 入口处设置的断点
程序流程控制的相关命令	
<code>run [args]</code>	[<file] [>file] 启动程序
<code>continue</code>	继续执行程序
<code>step [n]</code>	执行下一行或者下 n 行程序, 如果遇到函数, 进入函数内部
<code>next [n]</code>	执行下一行或者下 n 行程序, 如果遇到函数, 不进入函数内部
<code>quit</code>	退出执行
帮助	
<code>help [cmd]</code>	显示某个特殊命令的帮助
<code>help [class]</code>	显示某类命令的帮助

²每一个可以跟行号或者函数名的命令, 在行号或者函数名前面都可以附加一个用冒号隔开的文件名 (例如 `list mian.c:10` 或者 `break main.c:12`)。

Object-Oriented Programming

面向对象编程

目前的程序设计领域，面向对象的编程思想非常流行，而绝大多数流行的面向对象编程语言如C++、C#、Java或者Objective-C都是从C语言发展出来的，因此我们专门安排了一章，简要的介绍一下面向对象编程。本章首先介绍面向对象编程的主要原则，随后使用前面提到的四种面向对象编程语言中的三种（也就是语言名字带有C的那三种）编写一个简单的程序。本章并不打算教给读者如何使用这些面向对象的编程语言编写程序，也不打算介绍这些语言的主要特色，只是让读者对于这些编程语言有一个基本的印象罢了。

什么是对象

对象就是一个东西。按照面向对象编程的思路，编程就是设计某个东西以及你可以使用这个东西干什么。这一点和C语言过程化的编程风格形成了明显的对比。使用C语言编程的时候，我们通常关心一步步做什么，然后编写一些函数来完成这些步骤，最后我们才关心这些动作的对象，这个过程与面向对象编程的思路恰好相反。

下面我们从日常生活中举一个例子。假定读者有一辆汽车。这个汽车很明显是一个对象——读者拥有该汽车。读者不是拥有一个汽车的概念，而是从汽车厂中制造出来的一辆实实在在的汽车，比如说底特律汽车厂、日本汽车厂或者别的什么地方的汽车厂。这辆汽车有一个独一无二的牌号以便和其他的汽车区分。

按照面向对象的编程思路，读者的汽车实际上是汽车的一个实例。如果使用面向对象的术语来表述的话，汽车是类的名字，读者的汽车是汽车类的一个实例。按照这种表述方式，每生产一辆新的汽车，就是创造了汽车类的一个新实例。每个汽车实际上就是一个对象。

读者的汽车可能是银白色的，它的内部可能是黑色的，汽车的顶棚也许是可以折叠的，也许是金属的，等等。除此之外，读者可以使用该汽车完成某些事情，比如读者可以驾驶汽车，可以给汽车加油，清洗自己的汽车，保养自己的汽车等等。表19.1列出了上面谈到的这些操作。

表19.1 对象的操作

对象	对象可以完成的事情
读者的汽车	驾驶 加油 清洗 保养

读者可以对自己的汽车执行上面的操作，也可以针对其他的汽车执行上面的操作。比如假定读者的姐姐也有一辆汽车，她也可以驾驶它，给它加油等等。

实例和方法

类的一个独立实体就是一个实例。针对该实例可以执行的操作被称为方法。某些情况下，我们可以使用某个实例的方法，也可以直接将方法应用于类本身。比如，清洗汽车就是一个实例方法（实际上表19.1中列出的所有方法都是实例方法）。如果我们需要找出某个汽车制造商制造的汽车型号数量，对应的操作应该是一个类方法。

在C++中，我们使用如下的方式调用实例方法：

```
Instance.method ();
```

在C#中，我们使用同样的语法，如下所示：

```
Instance.method ();
```

在Objective-C中，调用实例方法被称为给类发送消息，具体语法如下所示：

```
[Instance method]
```

假定符号yourCar代表一个汽车类的实例，读者可以试着对表19.1中列出的操作，使用上面的三种语言写出相应的实例方法调用语句。具体的结果如表19.2所示：

表19.2 在面向对象编程语言中调用实例方法

C++	C#	Objective-C	动作
yourCar.drive()	yourCar.drive()	[yourCar drive]	驾驶
yourCar.getGas()	yourCar.getGas()	[yourCar getGas]	加油
yourCar.wash()	yourCar.wash()	[yourCar wash]	清洗
yourCar.service()	yourCar.service()	[yourCar service]	保养

假定读者的姐姐也有一辆汽车，名为suesCar，那么我们也可以对她的汽车调用实例方法，如下所示：

```
suesCar.drive()  suesCar.drive()  [suesCar drive]
```

能够针对不同的对象调用同样的实例方法是面向对象编程中一个非常重要的概念。

面向对象编程中另外一个重要概念就是多态。多态允许我们调用不同的类的同一名字的实例方法。比如我们有一个Boat类，该类的一个实例名为myBoat，使用多态，我们可以采用如下的方式调用myBoat的方法（使用C++语言）：

```
myBoat.service()  
myBoat.wash()
```

多态的关键在于我们可以为Boat类编写一个实例方法，然后针对另外一个完全不同的类的实例调用该方法。（译者注，多态是面向对象编程中的重要概念，但是限于篇幅这里无法详细解释，原著这里的介绍也比较模糊，读者不必细究）

面向对象的编程语言和C语言之间的最大区别在于，使用前者，在编程的时候我们以对象为基本元素思考问题，而使用后者时，我们以函数为基本单元思考问题。使用C语言，在描述保养交通工具这个操作时，我们可能编写一个名为service的方法，然后在该方法内部，我们分别处理各种不同类型的交通工具如汽车、轮船等。如果增加了一种新的交通工具类型，我们需要在service函数中增加对于该类型的处理代码。如果使用面向对象编程语言，我们将为这种新的交通工具定义一个新类，然后为这个类编写类似保养的方法。在编写新类的时候，我们不需要关心其他的交通工具，也不需要修改它们的代码（实际上，我们很可能无法访问它们的代码）。

使用面向对象编程语言的时候，我们处理的对象很可能并不是汽车和轮船，而是类似窗口、矩形、剪贴板这样的概念。在C#语言中调用这些对象的实例方法的代码可能如下所示：

myWindow.erase()	清除窗口内容
myRect.getArea()	计算矩形面积
userText.spellCheck()	对文本进行拼写检查
deskCalculator.setAccumulator(0.0)	清除计算器的内容
favoritePlaylist.showSongs()	显示播放列表中的歌曲

编写处理分数的C语言程序

假定我们需要编写一个用于处理分数的C语言程序。我们可能需要对分数执行加法、减法、乘法等操作。因此我们需要定义一个结构用于保存分数，然后再定义一组函数用于处理这些结构。

如果我们使用C语言处理分数，那么最终形成的程序可能如程序19.1所示。程序19.1设置分数的分子和分母，然后显示该分数的值。

程序19.1 使用C语言处理分数

```
// 使用分数的简单程序
#include <stdio.h>

typedef struct {
    int numerator;
    int denominator;
} Fraction;

int main (void)
{
    Fraction myFract;

    myFract.numerator = 1;
    myFract.denominator = 3;
    printf ("The fraction is %i/%i\n", myFract.numerator, myFract.denominator);
    return 0;
}
```

程序19.1 输出

The fraction is 1/3

在接下来的几个小节，我们将分别介绍如何使用Objective-C、C++和C#语言来编写处理分数的程序。在19.2小节后面对于面向对象编程语言的讨论同样适用于其他语言，因此请读者按照顺序阅读下面几个小节。

使用 Objective-C 定义用于处理分数的类

Brad Cox于1980年发明了Objective-C语言。该语言来源于一种名为SmallTalk-80的语言。NeXT软件公司于1988年开始分发该语言的许可证。Apple计算机公司于1988年收购了NeXT公司，随后Apple使用NEXTSTEP作为自己的Mac OS X操作系统的基础。直到今天，绝大多数在Mac OS X操作系统上运行的应用程序都是使用Objective-C语言编写的。

程序19.2演示了如何使用Objective-C编写一个处理分数的类。

程序19.2 使用Objective-C编写一个处理分数的类

```
// 使用分数的程序- Objective-C版

#import <stdio.h>
#import <objc/Object.h>
```

程序19.2 续

```
//----- @interface section -----
@interface Fraction: Object
{
    int numerator;
    int denominator;
}
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(void) print;
@end

//----- @implementation section -----

@implementation Fraction;

// getters
-(int) numerator
{
    return numerator;
}
-(int) denominator
{
    return denominator;
}

// setters
-(void) setNumerator: (int) num
{
    numerator = num;
}
-(void) setDenominator: (int) denom
{
    denominator = denom;
}

// other
-(void) print
{
```

程序19.2 续

```
    printf ("The value of the fraction is %i/%i\n", numerator, denominator);
}
@end

//----- program section -----

int main (void)
{
    Fraction *myFract;
    myFract = [Fraction new];

    [myFract setNumerator: 1];
    [myFract setDenominator: 3];

    printf ("The numerator is %i, and the denominator is %i\n",
           [myFract numerator], [myFract denominator]);

    [myFract print]; // use the method to display the fraction
    [myFract free];
    return 0;
}
```

程序19.2 输出

```
The numerator is 1, and the denominator is 3
The value of the fraction is 1/3
```

从程序19.2的注释中我们可以看出，该程序从逻辑上来说分为三个部分：接口部分、实现部分和程序部分。这几个部分通常被放置在不同的文件中。接口部分通常放置在头文件中，以便供其他需要使用这个类的程序包含。接口部分的作用是告诉编译器这个类具有哪些变量和方法。

实现部分包含了具体实现的代码。程序部分包含了完成具体任务的代码。

程序19.2中定义类名是Fraction，它的父类是Object。一个类可以从父类继承变量和方法。

在接口部分，程序包含有如下的声明语句：

```
int numerator;  
int denominator;
```

这些语句告诉编译器Fraction类具有两个变量：numerator和denominator。

在接口部分声明的变量也被称为实例变量。每次当我们创建一个类的实例的时候，实际上就创建了一组独立的实例变量。因此，如果我们有两个不同的Fraction对象，比如fracA和fracB，那么每个对象都将有自己的实例变量。

为了完成任务，我们还需要为分数类定义具体的方法，比如我们需要能够设置分数的分子和分母。在Objective-C中，我们无法直接存取一个对象的实例变量，因此必须编写专门的方法分别用于设置分子和分母（在Objective-C中这些方法被称为设置器）。我们还需要专门的方法用于获取分子和分母的值（在Objective-C中这些方法被称为取值器）¹。

面向对象编程中还有一个重要的概念——数据封装。它的意思是说对象应该对其用户隐藏自己的实例数据。将对于实例变量的所有访问都用方法包装起来，有助于扩展和修改该类。这个时候，数据封装在类的开发者和类的使用者之间提供了一个良好的隔离层。

声明设置器的语句如下所示：

```
-(int) numerator;
```

语句前面的-号表示该方法是一个实例方法。如果在语句前面放置一个+号，则说明该方法是一个类方法。类方法是可以直接作用在类本身上的方法，比如创造该类的一个新对象。创建某个类的一个新对象与制造一辆新汽车有些类似，汽车相当于类，创建汽车的方法就是类方法。

实例方法直接作用于具体的实例对象上，比如显示某个实例的值，设置某个实例的值，获取某个实例的值等等。接着我们前面举的汽车的例子，在制造出汽车之后，我们可能需要给汽车加油，这个操作作用在某个特定的汽车上，因此给汽车加油将是一个实例方法。

¹我们也可以直接存取实例变量的值，但是在面向对象编程中，这一点通常被认为是不好的编程风格。

声明一个方法与声明一个函数有些类似, 我们需要告诉编译器这个方法是否有返回值, 如果返回的话, 返回值的类型是什么等等。返回值在开始的+-号后面的一对小括号中指定, 请看下面的例子:

```
-(int) numerator;
```

上面的语句声明了一个名为numerator的实例方法, 该方法返回一个int类型的值。与之类似, 下面的语句:

```
-(void) setNumerator: (int) num;
```

声明了一个没有返回值的方法, 该方法用于设置分数的分子。

如果某个方法需要参数, 我们将在该方法的名字后面加上一个冒号。因此语句setNumerator: 和setDenominator: 说明这两个方法分别需要一个参数。而前面的numerator和denominator方法后面没有冒号, 说明这两个方法不需要参数。

方法setNumerator接收一个整数类型的参数num, 并将其作为分子保存到实例变量numerator中。与之类似, 方法setDenominator将其参数denom的值作为分母保存到实例变量denominator中。这两个方法都直接访问了类的实例变量。

我们的程序中定义的最后一个方法名为print。该方法可以打印出对象的内容。读者可以看出, 该方法既不接收参数, 也不返回值。在print方法中, 我们使用printf语句分别打印出分子和分母的值, 并用斜线将它们分开。

在main函数中, 我们使用如下的语句定义了一个名为myFract的变量:

```
Fraction *myFract;
```

这行语句告诉编译器, 符号myFract是类Fraction的一个实例, 因此我们可以用该变量来保存分数。myFract前面的星号说明该符号是一个指针, 它实际上指向一个包含有对象实例变量的结构。

现在我们已经有了Fraction类型对象的一个指针, 下一步就可以生成该类型的对象了。这和制造一辆新汽车的概念非常类似, 具体的语句如下所示:

```
myFract = [Fraction new];
```

在生成新对象的时候, 我们需要为该对象分配内存空间, 下面的表达式:

```
[Fraction new]
```

给Fraction类发送了一个消息new。在这里我们要求Fraction执行其new方法, 但是在该类中我们并没有定义new方法, 这个方法是从哪里来的呢。实际上, Fraction类从其父类继承了这个方法。

现在我们可以设置实例的分子和分母了, 具体语句如下所示:

```
[myFract setNumerator: 1];  
[myFract setDenominator: 3];
```

第一条语句调用了myFract对象的setNumerator: 方法, 传递的参数是1。接下来, 程序的执行流程就转到Fraction类的setNumerator: 方法中。因为Objective-C的运行时刻环境知道myFract是类Fraction的一个实例, 因此它将调用正确的方法。

在setNumerator:方法内部只有一行代码, 这行代码将传递来的参数作为分子保存在实例变量numerator中。在调用该方法之后, 我们就将分数对象myFract的分子设置为1。

接下来程序调用了myFract对象的setDenominator方法, 该方法的具体工作过程与上面的叙述类似。

在分数的分子分母值分别被设置之后, 程序19.2调用了两个取值器方法用于获取我们刚才设置的值, 并使用printf语句将这些值打印出来。

接下来, 程序19.2调用了实例myFract的print方法。虽然我们可以使用取值器方法分别获取分子和分母的值, 为了说明Objective-C的实例方法, 我们还是编写了独立的print方法。

程序的最后一条语句:

```
[myFract free];
```

释放了分配给对象myFract的内存。

使用 C++ 编写分数类

程序19.3使用C++语言编写了Fraction类。C++是一种非常流行的编程语言, 它是由贝尔实验室的Bjarne Stroustrup先生发明的。至少在作者的记忆中, C++是第一种基于C语言的面向对象编程语言。

程序19.3 使用C++语言处理分数

```
#include <iostream>

class Fraction
{
private:
    int numerator;
    int denominator;

public:
    void setNumerator (int num);
    void setDenominator (int denom);
    int Numerator (void);
```

程序19.3 续

```
    int Denominator (void);
    void print (Fraction f);
};

void Fraction::setNumerator (int num)
{
    numerator = num;
}
void Fraction::setDenominator (int denom)
{
    denominator = denom;
}
int Fraction::Numerator (void)
{
    return numerator;
}
int Fraction::Denominator (void)
{
    return denominator;
}
void Fraction::print (void)
{
    std::cout << "The value of the fraction is " << numerator << '/'
        << denominator << '\n';
}
int main (void)
{
    Fraction myFract;

    myFract.setNumerator (1);
    myFract.setDenominator (3);
    myFract.print ();

    return 0;
}
```

程序19.3 输出

The value of the fraction is 1/3

我们将C++类Fraction的成员变量（类似于Objective-C中的实例变量）定义为私有的，这样可以加强数据封装，防止别的代码访问这些变量。

方法setNumerator的声明语句如下所示：

```
void Fraction::setNumerator (int num)
```

方法名前面的Fraction表示该方法属于类Fraction。

在C++中，创建类Fraction的实例的方式与C语言中声明变量的方式非常类似，具体语句如下所示：

```
Fraction myFract;
```

在main函数中，我们使用如下的语句将分数的分子和分母分别设置为1和3：

```
myFract.setNumerator (1);  
myFract.setDenominator (3);
```

随后，我们使用对象的print方法将其内容打印出来。

对于读者来说，程序19.3中看上去最古怪的语句可能是print方法中下面这条语句：

```
std::cout << "The value of the fraction is " << numerator << '/'  
<< denominator << '\n';
```

在这条语句中，cout是标准输出流的名字，它的作用和C语言中的stdout相类似。<<是流插入操作符，使用它我们可以方便的输出变量的值。读者可能还记得，<<在C语言中代表左移位操作符。这里应用了C++语言一个非常重要的特性：运算符重载。我们可以把一个运算符和一个类联系在一起，而让该运算符作用于该类时表达程序员自己定义的作用。在上面的语句中，输出流重载了左移位操作符，该操作符现在的意思是将它的右操作数在左操作数代表的输出流上输出，而不是进行左移位操作。

作为运算符重载的另外一个例子，我们也可以对Fraction类重载+操作符，用于表示两个分数相加的概念。这样，我们就可以写出如下的语句了：

```
myFract + myFract2
```

在实际运行的时候，C++运行时刻环境将自动调用Fraction类中预定义的方法。

在print方法的输出语句中，<<操作符右面的表达式首先被求值，然后再将其结果输出到标准输出中。该语句首先输出字符串“The value of the fraction is”，然后跟上分数的分子、一个斜线，最后是分数的分母和一个换行符。

C++语言的特性是非常丰富的。读者可以看看附录E“C语言的其他资源”中列出的一篇C++入门教程。

在上面C++语言的例子中，我们虽然定义了取值器方法Numerator()和Denominator()，但是并没有在程序中调用这些方法。

使用 C#语言处理分数

作为本章的最后一个例子，程序19.4演示了如何使用C#语言处理分数。该语言由微软公司发明。作为一种较新颖的面向对象编程语言，C#预计将会很快流行起来。这个语言目前已经是在.NET平台上开发应用程序的首选语言。

程序19.4 使用C#处理分数

```
using System;

class Fraction
{
    private int numerator;
    private int denominator;

    public int Numerator
    {
        get
        {
            return numerator;
        }
        set
        {
            numerator = value;
        }
    }

    public int Denominator
    {
        get
        {
            return denominator;
        }
        set
        {
            denominator = value;
        }
    }

    public void print ()
    {
        Console.WriteLine("The value of the fraction is {0}/{1}",
            numerator, denominator);
    }
}
```

程序19.4 续

```
    }  
}  
  
class example  
{  
    public static void Main()  
    {  
        Fraction myFract = new Fraction();  
        myFract.Numerator = 1;  
        myFract.Denominator = 3;  
        myFract.print ();  
    }  
}
```

程序19.4 输出

The value of the fraction is 1/3

可以看出，C#语言和其他两种面向对象语言的风格有些不同，但是还是比较明白的。我们在Fraction类的开始定义了两个私有实例变量numerator和denominator。我们还定义了两个属性Numerator和Denominator，每个属性都有自己的取值器和设置器。下面让我们仔细看看属性Numerator的定义：

```
public int Numerator  
{  
    get  
    {  
        return numerator;  
    }  
    set  
    {  
        numerator = value;  
    }  
}
```

当我们在表达式中需要获取numerator的值时，C#将会调用属性的get方法。如下所示：

```
num = myFract.Numerator;
```


当需要给numerator赋值的时候，C#将会调用属性的set方法，如下所示：

```
myFract.Numerator = 1;
```

传递给set方法的值保存在符号value中。读者需要注意set方法和get方法后面没有小括号。

我们也可以定义接收参数的方法，或者接收多个参数的设置器方法。比如，下面的语句可以一次将分数的值设置为2/5。

```
myFract.setNumAndDen (2, 5)
```

让我们接着讨论程序19.4。在程序中，如下的语句：

```
Fraction myFract = new Fraction();
```

用于生成类Fraction的一个新实例，并将结果保存在Fraction类型的变量myFract中。随后，程序使用Fraction类的设置器方法将该分数的值设为1/3。

接下来，我们在程序中调用Fraction类的print方法显示分数myFract的值。在print方法中，我们调用了Console类的WriteLine方法，向标准输出打印内容。WriteLine的调用方式与printf函数比较类似，只不过该方法用符号{}表示printf中的%符号。因此WriteLine方法将用第一个参数替换字符串中的{0}，而用第二个参数替换字符串中的{1}，依次类推。与printf不同的是，我们不需要担心输出变量的类型。

和C++的例子类似，程序中也并没有调用Fraction类的取值器方法。

到此为止，我们对于面向对象编程语言的介绍就结束了。我们希望读者在阅读完本章之后，能够明白面向对象编程大致是怎么回事，面向对象编程语言和过程编程语言如C语言有哪些不同之处。我们使用三种面向对象编程语言编写了一个处理分数的小程序。如果读者想要真正编写一个实用的分数类的话，还应该给该类增加加法、减法、乘法、除法以及取倒数等方法。这些工作对于读者来说应该不是一件困难的事情。

如果要深入学习面向对象编程语言的话，读者应该使用一本好一些的入门教材，比如附录E中列出的那本。

C Language Summary

附录 A C 语言小结

本附录将对C语言以参考手册的形式进行总结。本章并不是要完整地定义C语言，而只是对于该语言主要特征作一个非正式的描述。读者在阅读完本书的正文之后，还应该花些时间通读本附录。这样不但可以巩固前面学到的知识，还能够从更全面的角度理解C语言。

本附录的材料来源于ANSI C99（ISO/IEC 9899:1999）标准。

1.0 字元和标识符

1.1 字元

表A.1列出了所有的双字符序列（字元）以及它们对应的单个字符。

表A.1 字元

字元	含义
<:	[
:>]
<%	{
%>	}
#:	#
%%:	##

1.2 标识符

C语言中的标识符由字符序列（大写字符或者小写字符）、通用字符名（参见1.2.1小节）、

数字和下画线组成。标识符的第一个字符必须是字母、下画线或者通用字符名。对于有外部连接的标识符，其前31个字符是有意义的，对于宏或者有内部连接的标识符，其前63个字符是有意义的。

1.2.1 通用字符名

通用字符名由u后面跟上四个十六进制数字或者U后面八个十六进制数字组成。如果标识符的第一个字符是一个通用字符名，那么该通用字符名的第一个值不能是数字。当使用通用字符名作为标识符的时候，该通用字符名的值不能小于A0₁₆（不包括24₁₆、40₁₆和60₁₆），也不能位于D800₁₆和DFFF₁₆之间。

通用字符名可以用于标识符的名字、字符常量和字符串中。

1.2.2 关键字

表A.2列出了C语言的所有关键字。

表A.2 关键字列表

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

2.0 注释

在程序中有两种插入注释的方法。`//`可以用来开始一个注释，在该行上所有位于`//`后面的字符都将被编译器忽略。

注释也可以以`/*`开始，以`*/`结束。在`/*`和`*/`之间可以放置任何内容，这些内容也可以跨越多行。在程序中任何可以插入空白的地方都可以插入注释。`/*`类型的注释不允许嵌套使用。也就是说，即使前面有多个`/*`，第一个遇到的`*/`就表示注释结束了。

3.0 常量

3.1 整数常量

整数常量由一串数字组成，数字前面可以有表示正负的符号。如果第一个数字是0，那么该常量被看作八进制数，所有的数字都应该从0到7。如果第一个数字是0，而且后面紧跟着一个字母x或者X，那么该常量被看作十六进制数字，所有的数字应该从0-9和a-f（或者A-F）。

如果在十进制常量后面加上符号l或者L，那么该常量被当作long int类型。如果long int类型无法容纳该数字，那么编译器将其当作long long int类型。如果在八进制或者十六进制常量后面加上符号l或者L，那么该常量被当作long int类型。如果long int类型无法容纳该数字，那么编译器将其当作long long int类型，最后如果long long int无法容纳该数字，那么编译器就将其当作unsigned long long int类型。

如果在十进制常量后面加上符号ll或者LL，那么该常量被当作long long int类型。如果在八进制或者十六进制常量后面加上符号ll或者LL，那么该常量被当作long long int类型，如果long long int无法容纳该数字，那么编译器就将其当作unsigned long long int类型。

如果在常量后面加上符号u或者U，那么该常量被当作一个无符号数。如果unsigned int无法容纳该数字，那么该常量将被当作unsigned long int类型，如果unsigned long int类型无法容纳该数字，那么编译器就将其当作unsigned long long int类型。

整数常量后面可以同时加上无符号和长整型符号，该常量被编译器当作unsigned long int类型。如果unsigned long int类型无法容纳该数字，那么编译器就将其当作unsigned long long int类型。

整数常量后面可以同时加上无符号和长长整型符号，该常量被编译器当作unsigned long long int类型。

如果一个十进制整型常量没有任何后缀，而signed int无法容纳该常量，该常量将被当作long int类型，如果long int类型无法容纳该常量，那么编译器就将其当作long long int类型。

如果一个八进制或者十六进制整型常量没有任何后缀，而signed int无法容纳该常量，该常量将被当作unsigned int类型。如果unsigned int类型无法容纳该常量，那么该常量将被当作long int类型，如果long int类型无法容纳该常量，那么该常量将被当作unsigned long int类型，如果unsigned long int类型无法容纳该常量，那么该常量将被当作long long int类型。最后，如果long long int类型无法容纳该常量，那么编译器将该常量当作unsigned long long int类型。

3.2 浮点常量

浮点数由一组数字、一个小数点和另外一组数字组成。如果在前面加上一个负号，则表示该常量是一个负值。小数点前面的数字和小数点后面的数字都可以省略，但是不能都省略。

如果在浮点常数后面再跟上字母e或者E,然后再跟上一个有符号的整数,那么该浮点常数采用科学计数法。整个浮点常量的值是10的指数部分(整数)次方再乘以前面的小数部分。(例如,1.5e-2代表 1.5×10^{-2} 或者0.015)。

十六进制浮点常量前面以0x或者0X开始,后面跟上一个或者多个十六进制数字,然后再跟上表示指数的字符p或者P,最后是表示以2为底数的指数部分。例如,0x3p10代表 3×2^{10} 。

编译器将浮点常数当作double类型看待。如果在常量后面加上字母f或者F,那么编译器就将其当作是float类型。也可以在浮点常数后面加上l或者L表示long double类型。

3.3 字符常量

用单引号括起来的单个字符组成一个字符常量。如果在单引号中有多个字符,处理方法由编译器具体实现自行决定。在字符常量中可以使用通用字符名表示那些不在标准字符集中的字符。

3.3.1 转义序列

转义序列以一个反斜线\开始。表A.3列出了所有的转义序列。

表A.3 特殊转义序列

字符	含义
\a	声音警铃
\b	退格
\f	表单
\n	换行
\r	回车
\t	水平制表
\v	垂直制表
\\	反斜线
\"	双引号
\'	单引号
\?	问号
\nnn	八进制字符
\unnnn	通用字符名
\Unnnnnnnn	通用字符名
\xnn	十六进制数字

如果使用八进制字符，可以使用1到3位八进制数字。最后三类转义字符应该使用十六进制数字。

3.3.2 宽字符常量

宽字符常量可以写作`L'x'`，这类常量的类型是`wchar_t`，该类型在标准头文件`<stddef.h>`中定义。使用宽字符可以表示那些无法用普通的`char`类型表示的字符。

3.4 字符串常量

字符串常量由一对双引号括起来的零个或者多个字符序列组成。双引号中可以包含任何字符，包括前面列出的转义字符。编译器自动在字符串后面加上结束的空字符（`'\0'`）。

编译器将返回一个指向该字符序列第一个字符的指针。然而，如果将字符串常量用作`sizeof`操作符的参数、`&`操作符的参数或者用于初始化字符数组的话，该常量被当作字符数组类型。

程序中不能修改字符串常量。

3.4.1 字符串连接

预处理器将自动把相邻的字符串连接起来。这些字符串之间可以使用一个或者多个空白字符分隔。例如，下面的字符串：

```
"a" " character "  
"string"
```

等价于下面的单个字符串：

```
"a character string"
```

3.4.2 多字节字符串

编译器可以自行定义多字节字符串以及相应的处理方法，以便能够在字符串中包含多字节字符。

3.4.3 宽字符字符串

宽字符字符串可以使用类似`L"..."`的形式表达。这种常量的类型是`wchar_t`类型的指针，其中`wchar_t`类型在`<stddef.h>`中定义。

3.5 枚举常量

枚举类型中的标识符被当作该枚举类型的常量，编译器也可以将其当作int类型。

4.0 数据类型与声明

本节讨论C语言中的基本数据类型、导出数据类型、枚举数据类型以及typedef语句。本节还对各种数据类型的变量声明语句进行了总结。

4.1 声明

当处理结构、联合、枚举数据类型和typedef语句的时候，编译器并不分配任何存储空间。这些语句只是告诉编译器某个特定类型的结构，并分配给该结构一个名字。这些数据类型既可以在函数内部定义，也可以在函数外部定义。如果是前者，那么该类型只在函数内部起作用；如果是后者，那么该类型在整个编译单元中都起作用。

当声明了某个数据类型之后，即可声明该数据类型的变量。声明某个数据类型的变量将使得编译器为该变量分配内存空间，除非所声明的变量是一个外部变量。在声明外部变量时，编译器将根据具体情况决定是否分配内存空间。（参见小节6.0）

在定义某个结构、联合或者枚举数据类型的时候，如果在结尾的分号前面列出变量的名字，那么编译器也将为这些变量分配内存空间。

4.2 基本数据类型

表A.4列出了C语言的基本数据类型。使用如下格式的语句即可声明基本数据类型的变量：

类型 name = 初始化值；

在声明变量的时候也可同时初始化该变量。初始化变量的语法规则在6.2小节说明。使用下面的形式可以在一条语句中同时声明和初始化多个变量：

类型 name = 初始化值, name = 初始化值, ... ；

在类型声明语句的前面，还可以加上存储类型修饰符。6.2节总结了所有的存储类型修饰符。如果在声明变量的时候指定了存储类型修饰符，而该变量又是int类型，那么在声明语句中可以省略int。如下所示：

```
static counter;
```

上面的语句声明了一个static int类型的变量，名为counter。

表A.4 基本数据类型总结

类型	意义
Int	整型数，也就是没有小数点，最小保证16位；
short int	短整型数，精度小于int，在某些计算机上占用的内存是int的一半，最小保证16位。
long int	长整型数，最小保证32位
long long int	长长整型数，最小保证64位
unsigned int	无符号整数，最小保证16位
float	浮点数，可以带有小数点，最小保证6位有效数字
double	双精度浮点数，最小保证10位有效数字
long double	扩展双精度浮点数，最小保证10位有效数字
char	字符类型，在某些计算机系统上，采用有符号扩展
unsigned char	与char类型相同，但是在提升为更长的int类型时采用无符号扩展
signed char	与char类型相同，但是采用有符号扩展
_Bool	布尔类型，可以用来保存0和1
float _Complex	复数
double _Complex	扩展复数
long double _Complex	扩展高精度复数
void	无类型，用于表明某个函数没有返回值，或者用于丢弃某个表达式的结果，也可用于通用类型指针(void *)。

在数据类型short int、int、long int和long long int前面也可以使用有符号修饰符signed。这些类型默认是有符号的，因此该修饰符没有实际作用。

使用_Complex和_Imaginary类型以及一组库函数可以处理复数及其运算。为了使用复数，程序中应该包含<complex.h>文件，该文件包含有处理复数所需的各类宏定义以及

函数原型声明。例如，我们可以按照下面的形式声明 `double _Complex` 类型的变量 `c1`，并将其值设置为 `5 + 10.5i`：

```
double _Complex c1 = 5 + 10.5 * I;
```

库函数 `creal` 和 `cimag` 可以分别用来获取某个复数的实部和虚部。

C 语言的具体实现并不要求一定支持 `_Complex` 和 `_Imaginary` 类型，具体实现也可以只支持其中的一类。

头文件 `<stdbool.h>` 可以让程序中更加方便的使用布尔类型。该文件中定义了宏 `bool`、`true`、`false`。我们可以使用布尔类型写出如下的语句：

```
bool endOfData = false;
```

4.3 导出数据类型

利用基本数据类型可以建立导出数据类型。导出数据类型包括数组、结构、联合以及指针。返回某个特定数据类型的函数也被认为是一种导出数据类型。本小节将讨论除了函数之外所有的导出数据类型。函数将在小节 7.0 中讨论。

4.3.1 数组

一维数组

数组中可以包含任何基本数据类型或者导出数据类型。但是数组中不能包含函数（虽然数组中可以包含函数指针）。

数组声明采用如下的形式：

```
类型 name[n] = { 初始化表达式, 初始化表达式, ... };
```

表达式 `n` 代表了数组中所能够容纳的元素个数，如果给出了数组的初始化列表，那么可以省略 `n`。在没有指定 `n` 的情况下，编译器根据初始化列表中元素的个数或者指定的最大下标来决定数组的长度。

在定义全局数组的时候，每一个初始化表达式都必须是常量表达式。初始化列表中的元素个数可以小于数组的长度，但是不能大于该长度。如果给出的元素个数小于数组的长度，那么编译器将只初始化给定数目的数组元素，其他的数组元素将被设置为 0。

字符数组的初始化是一个特例。C 语言允许使用字符串常量来初始化字符数组，如下所示：

```
char today[] = "Monday";
```

上面的语句定义了一个字符数组 `today`，该数组的初始化值为：'M'，'o'，'n'，'d'，'a'，'y' 和 '\0'。

如果在声明字符数组的时候，给出了数组的长度，而且该长度不足以容纳字符串结尾的空字符，那么编译器将不会在字符数组的结尾处放置空字符。如下所示：

```
char today[6] = "Monday";
```

上面的语句定义了一个包含6个字符的字符数组today，它的六个元素的初值分别为：'M'，'o'，'n'，'d'，'a'和'y'。

通过在初始化列表中指定需要初始化的元素编号，C语言允许我们以任意的顺序初始化数组，如下所示：

```
int x = 1233;  
int a[] = { [9] = x + 1, [3] = 3, [2] = 2, [1] = 1 };
```

上面的语句定义了一个包含10个元素的数组a(编译器通过初始化列表中的最大下标得知数组的长度)，该数组的最后一个元素被初始化为x+1 (1234)，前三个元素分别被初始化为1、2、3。

4.3.1.1 变量长度数组

在函数内部或者语句块内部，C语言允许我们使用变量声明数组的长度。C语言运行时环境负责计算该数组的长度。请看下面的例子。

```
int makeVals (int n)  
{  
    int valArray[n];  
    ...  
}
```

上面的语句定义了一个名为valArray的数组，该数组的长度为n。C语言将在程序运行的时候计算n的值，在不同的时候调用函数makeVals，n有可能取不同的值。变量长度数组不允许在声明的时候进行初始化。

4.3.1.2 多维数组

多维数组的一般声明形式如下：

```
type name[d1][d2]...[dn] = initializationList;
```

上面的形式定义了一个名为name，包含有 $d1 \times d2 \times \dots \times dn$ 个元素，类型为type的数组。请看下面的例子：

```
int three_d [5][2][20];
```

上面的语句定义了一个名为three_d的三维数组，包含有200个元素。

如果需要访问多维数组的某个成员，应该使用一组方括号给出该元素在所有维度上的下标。下面的语句：

```
three_d [4][0][15] = 100;
```

将100保存在数组three_d指定的元素中。

多维数组初始化的方法与一维数组初始化基本相同，可以使用内嵌的花括号来控制给哪些数组元素赋值。

下面的语句声明了一个二维数组`matrix`，该数组包含4行3列。

```
int matrix[4][3] =
    { { 1, 2, 3 },
      { 4, 5, 6 },
      { 7, 8, 9 } };
```

数组`matrix`第一行上的三个元素被分别初始化为1、2、3，第二行上的三个元素被分别初始化为4、5、6，第三行上的三个元素被初始化为7、8、9。因为声明时没有给出第四行元素的初值，因此这些元素被初始化为0。下面的语句：

```
static int matrix[4][3] =
    { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

其初始化效果与前面的语句相同，因为C语言在初始化多维数组的时候按照从左向右的顺序对各维进行初始化。

下面的声明语句：

```
int matrix[4][3] =
    { { 1 },
      { 4 },
      { 7 } };
```

将`matrix`数组第一行的第一个元素设置为1，第二行的第一个元素设置为4，第三行的第一个元素设置为7，其他所有元素都被初始化为0。

最后，请看下面的声明语句。

```
int matrix[4][3] = { [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

该声明语句只对二维数组中指定的元素进行初始化。

4.3.2 结构

结构的一般声明形式如下：

```
struct name
{
    成员声明
    成员声明
    ...
} 变量列表;
```

结构`name`中包含所有声明的成员变量。每个成员变量声明由一个类型名加上一个或者多个成员变量名组成。

如果要声明结构变量，可以在结构定义的时候，在结束的分号之前加上这些变量的名字。也可以在定义结构之后使用如下形式的语句声明结构变量：

```
struct name 变量列表;
```

如果在定义结构类型的时候没有指定名字的话，就不能使用上述形式。在这种情况下，必须在定义结构类型的时候声明该类型的所有变量。

结构变量初始化的方式与数组初始化很类似，可以使用一对大花括号将结构成员变量的初始值列表包围起来。在声明全局结构变量的时候，每一个成员变量的初始化表达式都必须是常量表达式。

下面的声明语句：

```
struct point
{
    float x;
    float y;
} start = {100.0, 200.0};
```

定义了一个名为point的结构类型，同时声明了该类型的一个名为start的变量，并对该变量进行了初始化。在初始化结构变量的时候，也可以按照下面的形式指定成员变量的名字：

```
.member = value
```

请看下面的示例语句：

```
struct point end = { .y = 500, .x = 200 };
```

下面的声明语句：

```
struct entry
{
    char *word;
    char *def;
} dictionary[1000] = {
    { "a", "first letter of the alphabet" },
    { "aardvark", "a burrowing African mammal" },
    { "aback", "to startle" }
};
```

声明了一个包含有1000个元素的结构数组，并对前三个元素进行了初始化。如果指定成员变量名字的话，我们也可以按照下面的形式重写上面的语句：

```
struct entry
{
    char *word;
    char *def;
} dictionary[1000] = {
    [0].word = "a", [0].def = "first letter of the alphabet",
```



```
[1].word = "aardvark", [1].def = "a burrowing African mammal",  
[2].word = "aback", [2].def = "to startle"  
};
```

我们也可以采用下面的等价形式:

```
struct entry  
{  
    char *word;  
    char *def;  
} dictionary[1000] = {  
    { .word = "a", .def = "first letter of the alphabet" },  
    { .word = "aardvark", .def = "a burrowing African mammal" },  
    { .word = "aback", .def = "to startle" }  
};
```

C语言允许使用一个同类型的结构变量初始化另外一个结构变量, 如下所示:

```
struct date tomorrow = today;
```

上面的语句声明了结构变量`tomorrow`, 并将同类型结构变量`today`的值赋给它。

下面形式的结构成员变量声明:

```
type fieldName : n
```

说明该成员变量是一个位域, 其中`n`是一个整数值, 代表位域的宽度。在某些计算机上, 位域有可能从左向右分配, 而在另外一些机器上, 位域将从右向左分配。如果在声明位域成员变量的时候没有给出其名字, 那么编译器将为该成员保留内存空间, 但是程序不能访问这些空间。如果没有指定位域的名字, 而且`n`也等于0, 那么分配给结构下一个成员变量的内存空间将从单元边界上开始, 单元的大小由具体实现决定。位域的类型可以是`_Bool`、`int`、`signed int`或者`unsigned int`。如果指定位域类型是`int`的话, 那么由具体实现决定该位域是有符号的还是无符号的。取地址操作符不能作用于位域上, 另外C语言也不允许定义位域数组。

4.3.3 联合

联合的一般声明形式如下所示:

```
union name  
{  
    成员声明  
    成员声明  
    ...  
} 变量列表;
```

上面的形式可以用来定义名为`name`的联合, 该联合包含所有列出的成员变量。联合中的所有成员共享同一块内存空间, C语言编译器保证分配给联合的内存能够容纳其最大的成员变量。

我们可以在定义联合的时候声明该类型的变量，也可以在定义联合之后采用如下的形式声明该类型的变量：

```
union name 变量列表;
```

编程者应该保证访问联合变量时采用的成员变量和最后一次向其中存入值时采用的成员变量相同。在声明联合变量的时候，可以使用大花括号括起来的值初始化联合的第一个成员变量，如果联合变量是一个全局变量，那么初始化表达式必须是一个常量表达式。请看下面的例子：

```
union shared
{
    long long int l;
    long int w[2];
} swap = { 0xffffffff };
```

上面的语句声明了一个名为 swap 的联合，并将其成员 l 初始化为十六进制值 0xffffffff。如果要初始化其他的成员变量，可以在初始化的时候给出该成员变量的名字，请看下面的语句：

```
union shared swap2 = { .w[0] = 0x0, .w[1] = 0xffffffff; }
```

同类型的联合变量可以用于初始化另外一个联合变量，如下所示：

```
union shared swap2 = swap;
```

4.3.4 指针

声明指针变量的基本语法形式如下：

```
type *name;
```

上面的语句声明 name 为一个指向 type 类型的指针。type 既可以是基本数据类型，也可以是扩展数据类型，请看下面的示例：

```
int *ip;
```

上面的语句声明了一个指向 int 类型变量的指针 ip，而下面的语句：

```
struct entry *ep;
```

声明了一个指向 entry 结构类型的指针 ep。

指向数组的指针被声明为指向该数组所容纳元素类型的指针。例如，上面声明的指针 ip 也可以用来指向一个整型数组。

C 语言允许使用某些高级的语法形式声明指针，请看下面的语句：

```
char *tp[100];
```

上面的语句声明了一个包含有 100 个字符指针元素的数组。而下面的声明语句：

```
struct entry (*fnPtr) (int);
```


声明了一个指向返回值类型为`struct entry`的函数的指针，该函数接收一个整型参数。

C语言允许将指针变量与值为0的常量表达式进行比较，用以判断该指针是否是空指针。具体的实现内部可以用其他不等于0的值来代表空指针，然而，如果将这样的指针与0比较的话，编译器必须报告它们是相等的。

整数类型与指针类型之间的相互转化，以及能够容纳一个指针所需的整数大小是与具体机器相关的。

`void*`是通用的指针类型，C语言保证任何类型的指针都可以保存到`void*`类型中，并且随后将其从`void*`类型的指针中取出而不改变原来的值。

除了上面的特例之外，C语言不允许不同类型指针之间的转换。如果尝试转换的话，编译器有可能给出警告。

4.4 枚举数据类型

定义枚举数据类型的一般格式如下所示：

```
enum name { enum_1, enum_2, ... } variableList;
```

上面的语句形式定义了枚举类型`name`，其枚举值分别为`enum_1`、`enum_2`等等。每一个枚举值应该是一个合法的标识符，或者是一个标识符后面跟上一个等号，再加上一个常量表达式。`variableList`本身是可选的，它代表一组该类型的变量（也可以同时初始化）。

编译器将从0开始逐个给枚举值赋值。如果某个枚举值标识符后面跟有等号和常量表达式，那么编译器就将该常量表达式的值作为该枚举值的值。该枚举值后面的枚举值从这个枚举值开始逐个加1，重新编号。编译器将枚举值当作常量。

如果要声明一个枚举变量（假定该枚举类型已经在前面定义过），可以采用如下的形式：

```
enum name variableList;
```

某个枚举变量的值只能是定义时列出的枚举值之一。但是如果赋给该枚举变量其他整数值，编译器有可能并不将其作为错误看待。

4.5 typedef 语句

`typedef`语句用于给基本数据类型和导出数据类型定义一个新的名字。这个语句本身并不创造新的数据类型，而只是给已经存在的数据类型起一个新的名字。因此，编译器对使用新名字声明的变量按照使用原来名字声明的变量同样的方式对待。

书写`typedef`语句的一般规则如下：首先写出使用该类型声明变量的语句，然后将变量名使用新的数据类型名替换，最后在整个语句前面加上关键字`typedef`。

请看下面的例子：

```
typedef struct
{
    float x;
    float y;
} Point;
```

上面的语句给一个结构类型赋予名字Point，该结构类型包含两个分别名为x和y的浮点数成员变量。随后我们可以使用Point声明新的变量，如下所示：

```
Point origin = { 0.0, 0.0 };
```

4.6 类型修饰符 const、volatile 和 restrict

C语言允许在声明语句的类型前面放置关键字const，用于表示该变量的值不能被修改。因此，下面的语句：

```
const int x5 = 100;
```

定义变量x5的值总是100（也就是说，该变量的值在程序执行期间不会被修改）。C语言标准并不要求编译器在程序试图修改const变量的值时发出警告。

volatile关键字告诉编译器某个变量的值在运行的时候有可能动态的改变。当volatile变量出现在表达式中时，C语言将每次都从该变量的存储位置获取其值。

为了声明标识符port17是一个指向char类型的volatile指针，我们使用如下的语句：

```
volatile char *port17;
```

restrict关键字可以和指针变量一起使用。该关键字可以为编译器优化代码提供某些暗示（正如register关键字对于变量的作用一样）。restrict关键字告诉编译器，它所修饰的指针是指向某个特定对象的唯一一个指针，也就是说，在同一个作用域内，没有其他指针指向这个特定对象。下面的两行语句：

```
int * restrict intPtrA;
int * restrict intPtrB;
```

告诉编译器，在指针intPtrA和intPtrB的作用域期间，它们永远不会用来指向同一个整数。如果用它们指向一个整型数组的话，那么这两个指针所访问的元素范围是互斥的。

5.0 表达式

变量名、函数名、数组名、常量、函数调用、数组引用、结构以及联合引用都被C语言看作是表达式。将一个单边操作符作用在上述表达式将形成新的表达式，使用二元、三元操作符连接上述表达式最终也形成表达式。最后，用括号括起来的表达式也是新的表达式。

除了void类型之外的任何表达式都被称作一种特殊的数据对象——左值。如果能够给该数据对象赋值的话，这个对象还被称为可修改的左值。（译者注：本书中的术语左值与传统的左值定义不同。按照名著《C语言》（该书的作者为C语言的创始人）中的定义，左值代表分配了存储空间的数据对象。为了忠实原著，译者仍然采用原著中的术语，读者应该认识到，一般程序设计教材中的左值相当于这里的“可修改的左值”。）

C语言很多地方都要求有可修改的左值表达式。例如，赋值操作符的左面必须是一个可修改的左值，还有，自增和自减操作符也必须作用在可修改的左值表达式上，另外&操作符也要求可修改的左值（除非操作数是函数）。

5.1 C 语言的操作符总结

表A.5列出了C语言的各种操作符。操作符按照优先级递减的顺序列出，相同优先级的操作符放在同一组中。

表A.5 C语言的操作符总结

操作符	描述	关联性
()	函数调用	从左向右
[]	数组元素引用	
->	从指针引用成员	
.	引用结构成员	
-	单边减法	从右向左
+	单边加法	
++	自增	
--	自减	
!	逻辑非	
~	一阶补数	
*	指针引用	
&	取地址	
sizeof	取对象大小	
(type)	类型转换	
*	乘法	从左向右
/	除法	
%	求余	从左向右
+	加法	
-	减法	从左向右
<<	左位移	
>>	右位移	从左向右
<	小于	
<=	小于等于	
>	大于	
=>	大于等于	

表A.5 续

操作符	描述	关联性
==	相等	从左向右
!=	不等	
<hr/>		
&	按位与	从左向右
^	按位异或	从左向右
<hr/>		
	按位或	从左向右
<hr/>		
&&	逻辑与	从左向右
<hr/>		
	逻辑或	从左向右
<hr/>		
?:	条件	从右向左
<hr/>		
=	赋值	从右向左
*= /= %=		
+= -= &=		
^= =		
<<= >>=		
<hr/>		
,	逗号	从右向左

下面我们举一个例子来说明如何使用表A.5。考虑如下的表达式：

b | c & d * e

因为在表A.5中，乘法操作符出现在按位与和按位或两个操作符上面，所以它的优先级要高于后两者。按照同样的道理，按位与操作符的优先级高于按位或操作符，因此，上面的表达式实际上按照下面的顺序求值：

b | (c & (d * e))

再考虑下面的表达式：

b % c * d

因为求余操作符与乘法操作符在表A.5中出现在同一组里，所以它们的优先级是相同的。因为这些操作符的结合性是从左向右，因此该表达式按照下面所示的顺序求值：

(b % c) * d

再举一个例子，下面的表达式：

++a->b

的求值顺序如下所示：

++(a->b)

因为->操作符的优先级高于++操作符。

最后，因为赋值操作符的结合性是从右向左，因此下面的表达式：

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

实际上按照下面的顺序求值：

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

该表达式的最终执行结果是将a和b的值都赋为0。在下面的表达式中：

```
x[i] + ++i
```

C语言并没有明确指定应该先对加号左面求值还是先对加号右面求值，而这个求值顺序将影响我们的左操作数到底是x[i]还是x[i+1]。

在下面这个表达式中，运算符的优先级同样是未定义的：

```
x[i] = ++i
```

在上面的例子中，C语言并没有规定到底使用未增值之前的i作为x数组的下标，还是使用增值之后的i作为x数组的下标。

函数的参数求值顺序也是未定义的。因此，在下面的函数调用语句中：

```
f (i, ++i);
```

i有可能先增值。这样f函数接收的两个参数就是相同的。

C语言保证操作符||和&&是按照从左到右的顺序求值的。特别的，对于&&操作符，如果左面的表达式求值结果为0，那么将不对右边的表达式求值；而对于||操作符，如果左面的表达式求值结果不为0，那么将不对右边的表达式求值。在使用类似下面的语句时，一定要把这个原则牢记在心：

```
if ( dataFlag || checkData (myData) )  
    ...
```

在上面的例子中，只有dataFlag的值为0，函数checkData才会被调用。下面是另外一个例子：假定a是一个拥有n个元素的数组，那么下面的表达式：

```
if (index >= 0 && index < n && a[index] == 0)  
    ...
```

只有index是一个合法的下标值，我们才会用它引用数组a中的元素。

5.2 常量表达式

所谓常量表达式就是每一部分都是常量的表达式。在下面的场合必须使用常量表达式：

1. switch 语句中的每一个 case 子句后面;
2. 声明同时进行初始化的数组或者全局数组;
3. 枚举标识符的值;
4. 结构定义中位域的宽度;
5. 静态变量的初始化值;
6. 全局变量的初始化值;
7. #if 预处理语句后面的值。

对于前四种场合, 常量表达式必须由整数、字符、枚举值或者 sizeof 运算符的结果组成。表达式中只允许使用算术操作符、位操作符、关系操作符、条件表达式操作符和类型转换操作符。另外, 这里 sizeof 操作符的操作数不能是变量长度数组, 因为对于变量长度数组 sizeof 操作符的结果是在运行时刻求出的, 因此不能算是常量。

对于第五种和第六种场合, 除了上面的情况外, 还可以显式或者隐式的使用 & 操作符。但是这个操作符的操作数只能是全局、静态变量或者函数。例如, 下面的表达式:

```
&x + 10
```

是一个合法的常量表达式, 只要 x 是一个全局变量或者静态变量。下面的表达式:

```
&a[10] - 5
```

也是一个合法的常量表达式, 只要 a 是一个全局或者静态数组。最后, 因为 &a[0] 和表达式 a 等价, 因此下面的表达式:

```
a + sizeof (char) * 100
```

也是合法的常量表达式。

对于最后一种情况, 所需的常量表达式与前四种相同, 但是表达式中不能使用 sizeof 操作符、枚举常量和类型转换操作符。不过, 我们可以在其中使用特殊的 defined 操作符 (参见 9.2.3 小节)。

5.3 算术操作符

如果

a, b	void 类型的任何基本数据类型的表达式;
i, j	整型数据类型;

那么下面的表达式:

-a	数
+a	
a+b	b

a-b	b
a*b	b
a/b	b
i%j	除以j所得的余数

在每一个表达式中，运算时都要对操作数进行算术转换（参见5.17小节）。如果a是一个无符号数，那么在计算-a的时候，首先对其进行整数提升操作，然后从被转换的类型中最大的数中减去它，然后再给结果加上1。

如果除法的两个操作数都是整数，那么所得的结果将被取整。如果两个操作数中有一个是负数，那么取整的方向是未定义的（也就是说，在某些计算机上-3/2有可能得-1，而在另外一些计算机上的结果是-2）。5.15小节列出了指针的算术操作规则。

5.4 逻辑操作符

如果a、b是除了void类型外的任何基本数据类型，或者两者都是指针类型；

那么下列表达式：

a && b	a和b的值都不是0，那么结果为1，否则结果为0（只有a不等于0的时候，才对b求值）
a b	a和b中有一个不为0，那么结果为1，否则结果为0（只有a等于0的时候，才对b求值）
! a	等于0，那么结果为1，否则结果为0

在运算过程中使用5.17小节中列出的转换规则。所有的操作结果类型都是int。

5.5 关系操作符

如果

a、b id类型外的任何基本数据类型，或者两者都是指针类型；

那么下列表达式：

a < b	a小于b那么结果为1，否则结果为0
a <=b	a小于等于b那么结果为1，否则结果为0
a > b	a大于b那么结果为1，否则结果为0
a >=b	a大于等于b那么结果为1，否则结果为0
a == b	a等于b那么结果为1，否则结果为0
a != b	a不等于b那么结果为1，否则结果为0

在运算过程中使用5.17小节中列出的转换规则。如果a和b都是指针，那么只有当a和b指向同一个数组或者同一个结构或联合的成员时，这些操作才有意义。所有的操作结果类型都是int。

5.6 位操作符

如果i、j、n都是整型表达式：

那么下列表达式：

i & j	执行按位与操作
i j	和j执行按位或操作
i ^ j	和j执行按位异或操作
~i	的一类补数
i << n	左移n位
i >> n	右移n位

除了左移操作和右移操作之外，其他运算都执行5.17小节中列出的转换规则。对于左移操作和右移操作，仅仅执行整数提升操作。如果移位的位数为负或者大于等于被移位操作数的位数，那么移位操作的结果是未定义的。在某些计算机上，右移操作按照算术规则进行（用最高符号位填充空出来的位），而在另外一些计算机上则执行逻辑规则（用0填充）。移位操作的结果与被提升的左操作数类型相同。

5.7 自增和自减操作符

如果lv是一个可修改的左值，而且它没有用const修饰过；

那么下列表达式：

++lv	v的值增加1，然后将lv的值作为表达式的值
lv++	v的值作为表达式的值，然后将lv的值增加1
--lv	v的值减去1，然后将lv的值作为表达式的值
lv--	的值作为表达式的值，然后将lv的值减去1

5.15小节列出了这些操作作用于指针类型时的规则。

5.8 赋值操作符

如果：

lv 个可以修改的左值，其类型没有用const修饰
op 意一个有对应赋值操作符的操作符
a 表达式

那么下面的表达式：

lv = a 的值保存在lv中；
lv op= a p代表的操作作用于lv和a，然后将结果保存在lv中。

在上面的第一个表达式中，如果a是除了void类型之外的基本数据类型之一，编译器将会把它转换为lv的类型。如果lv是一个指针，那么a必须是一个同类型的指针，或者void*类型的指针，或者是空指针。如果lv是一个void*类型的指针，那么a可以是任何类型的指针。

上面的第二个表达式实质上相当于lv = lv op (a)，但是编译器只对lv求值一次。请读者考虑一下如下的表达式 (x[i++] += 10)。

5.9 条件操作符

如果：

a、b、c 表达式；

那么下面的表达式：

a ? b : c 不等于0的时候，表达式等于b，否则的话等于c。整个过程只对b或者c中的一个求值。

表达式b和c的类型必须是相同的。如果两者类型不同，但是都是算术数据类型，那么使用一般的算术数据类型转换规则（译者注：参见5.17小节）。如果两者中间有一个是指针类型而另外一个为0，那么后者将被当作前者类型的一个空指针。如果有一个指针是void*类型而另外一个为某种特定类型的指针，那么后者将被转换为void*类型的指针，整个表达式的结果也将是void*类型。

5.10 类型转换操作符

如果：

type 个基本数据类型、枚举数据类型（前面加上enum）、typedef语句定义的类型或者是某个导出数据类型的名字；

a 达式；

那么下面的表达式：

(type)a 将a转换为指定的类型。

5.11 sizeof 操作符

如果:

type 个如5.10小节描述的类型;
a 表达式;

那么下面的表达式:

sizeof(type) 表达式的值等于容纳指定数据类型的值所需要的内存字节数;
sizeof(a) 表达式的值等于保存表达式a的结果所需的内存字节数。

如果type是字符类型(char),那么sizeof操作符的结果为1。如果a是一个已经指定了维数的数组名(既可以显式指定,也可以通过初始化列表隐式指定),而且a不是函数的形式参数或者没有指定长度的外部数组变量,那么sizeof操作符的结果为容纳该数组所有元素所需的字节数。

sizeof表达式结果的类型为size_t,该类型在标准头文件<stddef.h>中定义。

如果a是一个变量长度数组,那么sizeof表达式将在运行时刻求值;否则的话,该表达式将在编译时刻求值,其结果被编译器当作常量表达式(参看5.2小节)

5.12 逗号操作符

如果

a, b 达式;

那么下面的表达式:

a, b 器先对a求值,然后再对b求值,整个表达式的结果和类型等于表达式b的结果与类型。

5.13 数组的基本操作

如果:

a 含有n个元素的数组;
i 为整型数的表达式;
v 达式;

那么下面的表达式:

a[0] 组a的第一个元素
a[n-1] 数组a的最后一个元素
a[i] 的第i号元素(译者注:也就是第i+1个)
a[i] = v 达式v的结果保存在a[i]中

上面所有表达式的类型等于数组a中所保存元素的类型。关于数组和指针表达式的更多介绍请参看5.15小节。

5.14 结构的基本操作¹

如果：

x 是一个可修改的左值，其类型为struct s；
y 是一个类型为struct s的表达式；
m 是类型struct s的一个成员变量的名字；
v 是一个表达式。

那么下面的表达式：

x 引用整个结构，其类型为struct s；
y.m 引用结构变量y的成员变量m，其类型为m的类型；
x.m = v 将表达式v的值保存到结构变量x的成员变量m中，其类型为m的类型；
x = y 将y的值赋给x，结果的类型为struct s；
f(y) 调用函数y并将结构变量y的内容作为参数传递给该函数。在函数f内部，形式参数的类型必须是struct s；
return y 返回结构变量y的内容。函数的返回值必须被声明为struct s类型。

5.15 指针的基本操作

如果：

x 是一个类型为t的左值表达式；
pt 是一个可修改的左值表达式，类型为指向t的指针；
v 是一个表达式；

那么下面表达式：

&x 生成一个指向x的指针，表达式的类型为指向t的指针；
pt = &x 使得指针pt指向x，表达式的类型为指向t的指针；
pt = 0 将指针pt设置为空指针；
pt == 0 判断pt是否为空指针；
*pt 取得指针pt指向的值，表达式的类型为t；
*pt = v 将表达式v的值保存在pt所指向的位置中，表达式的类型为t。

¹这里的讨论也适用于联合。

指向数组的指针

如果:

- a 是一个数组, 其元素类型为t;
- pa1 是一个可修改的左值, 其类型为指向t的指针, 指向数组a中的某个元素;
- pa2 是一个左值表达式, 类型为指向t的指针, 指向数组a中的某个元素, 或者指向数组a中最后一个元素再后面的位置;
- v 是一个表达式;
- n 是一个整型表达式;

那么下面的表达式:

- a, &a, &a[0] 产生一个指向数组第一个元素的指针;
- &a[n] 产生一个指向数组第n号元素的指针, 类型为指向t的指针;
- *pa1 引用pa1指向的元素, 类型为t;
- *pa1 = v 将表达式v的值保存在pa1指向的位置, 表达式的类型为t;
- ++pa1 将pa1指向数组a的下一个元素, 表达式类型为指向t的指针;
- pa1 将pa1指向数组a的前一个元素, 表达式类型为指向t的指针;
- ++pa1 将pa1指向数组a的下一个元素, 然后引用该元素, 表达式的类型为t;
- *pa1++ 引用pa1所指向的元素, 表达式的类型为t; 然后将pa1指向数组a的下一个元素;
- pa1 + n 生成一个指针, 该指针指向数组a中v指向元素后面第n个元素, 表达式类型为指向t的指针;
- pa1 - n 生成一个指针, 该指针指向数组a中pa1指向元素前面第n个元素, 表达式类型为指向t的指针;
- *(pa1+n) = v 将表达式v的值保存在pa1+n所指向的位置, 表达式的类型为t。
- pa1 < pa2 判断pa1指向的元素在数组中的位置是否在pa2所指的元素前面, 表达式的类型为整型数 (所有的关系操作符都可以用于指针比较)。
- pa2 - pa1 数组a中pa1所指向的元素和pa2所指向的元素之间的元素个数(假定pa2所指向的元素在pa1后面)。表达式的类型为整型数;

`a + n` 生成一个指针指向数组 `a` 的第 `n` 号元素，表达式的类型为指向 `t` 的指针，该表达式与 `&a[n]` 完全等价；

`*(a + n)` 获取数组 `a` 的第 `n` 号元素的值，表达式的类型为 `t`，该表达式与 `a[n]` 等价。

指针减法运算所得的整数值其实际类型为 `ptrdiff_t`，该类型在系统头文件 `<stddef.h>` 中定义。

指向结构的指针²

如果：

- `x` 是类型 `struct s` 的一个左值表达式；
- `ps` 是一个可修改的左值表达式，其类型为指向 `struct s` 的指针；
- `m` 是类型 `struct s` 的成员变量，其类型为 `t`；
- `v` 是一个表达式；

那么下列表达式：

- `&x` 生成一个指向 `x` 的指针，该指针的类型为“指向 `struct s` 的指针”；
- `ps = &x` 将指针 `ps` 指向 `x`，表达式的类型为“指向 `struct s` 的指针”；
- `ps->m` 引用 `ps` 所指向的结构成员变量 `m`，表达式的类型为 `t`；
- `(*ps).m` 同样引用 `ps` 所指向的结构成员变量 `m`，与 `ps->m` 完全等价；
- `ps->m = v` 将表达式 `v` 的值保存在 `ps` 所指向的结构成员变量 `m` 中，表达式的类型为 `t`。

5.16 复合字面量

复合字面量由一个用小括号括起来的类型名后面跟上初始化列表组成。该表达式产生一个指定类型的无名值。如果在语句块内定义复合字面量，那么该值的作用域范围局限在该语句块内部，如果在任何语句块外定义复合字面量，则该值的作用域是全局的。如果声明全局复合字面量的话，那么其初始化列表表达式必须是常量表达式。

请看下面的例子：

```
(struct point) {.x = 0, .y = 0}
```

上面的表达式声明了一个类型为 `struct point` 的复合字面量，同时赋予初值。复合字面量可以用来初始化另外一个同类型的变量，请看下面的例子：

```
origin = (struct point) {.x = 0, .y = 0};
```

复合字面量也可以作为参数传递给函数，如下面的例子所示：

```
moveToPoint ((struct point) {.x = 0, .y = 0});
```

²这些讨论也适用于联合。

复合字面量也可以用于定义非结构的其他类型。例如，如果 `intPtr` 的类型是 `int*`，那么下面的表达式：

```
intPtr = (int [100]) {[0] = 1, [50] = 50, [99] = 99 };
```

将 `intPtr` 指向一个长度为100的数组（上述语句可以出现在程序的任何位置），该数组的三个元素已经被初始化。

如果没有指定数组长度的话，该数组的长度由初始化列表决定。

5.17 基本数据类型的转换规则

C语言对于算术表达式中操作数的类型转换遵守一套预先定义的规则，该规则被称为普通算术转换规则。具体步骤如下：

1. 如果两个操作数中的任一个类型为 `long double`，那么另外一个操作数也将被转换为 `long double`，整个表达式的类型也是 `long double`。
2. 如果两个操作数中的任一个类型为 `double`，那么另外一个操作数也将被转换为 `double`，整个表达式的类型也是 `double`。
3. 如果两个操作数中的任一个类型为 `float`，那么另外一个操作数也将被转换为 `float`，整个表达式的类型也是 `float`。
4. 如果某个操作数的类型是 `_Bool`、`char`、`short int`、`int` 位域或者枚举数据类型，那么编译器首先将其转换为 `int` 类型。如果 `int` 不能表示该值，那么编译器将其转换为 `unsigned int`。如果转换后两个操作数类型相同，那么表达式的结果将是该类型。
5. 如果两个操作数都是有符号的，或者都是无符号的，那么编译器将较小的整数类型转换为较大的类型，该类型将是表达式的类型。
6. 如果两个操作数的符号类型不同，而无符号操作数的尺寸大于或者等于有符号操作数，那么有符号操作数将被转换为无符号操作数，表达式的类型与无符号操作数的类型相同。
7. 如果两个操作数的符号类型不同，而有符号操作数的类型能够表达所有的无符号操作数的数值，那么无符号操作数将被转换为有符号操作数，表达式的类型与有符号操作数的类型相同。
8. 如果到达了这一步，那么两个操作数都被转换为有符号类型对应的无符号类型。

其中，上面的第4步通常被称为整型提升。

上面的转换规则在绝大部分情况下都可以很好的工作，但是有如下几种情况值得读者注意：

1. 将 `char` 类型转换为 `int` 类型在某些机器上有可能执行符号扩展，除非 `char` 类型被明确声明为 `unsigned char`。

2. 将一个有符号的整数扩展为更大的整数类型时，计算机执行符号扩展；将一个无符号的整数扩展为更大的整数类型时，计算机在前面添加相应的 0。
3. 如果将某个值转换为 `_Bool` 类型时，如果该值为 0，那么转换结果为 0，否则的话转换结果为 1。
4. 将一个较大的整数转换为一个较小的整数，整数截断将从左面开始。
5. 将一个浮点数转换为整数时，浮点数的小数部分将被舍弃，如果转换的整数类型无法容纳该浮点数的整数部分，那么转换的结果是未定义的；将一个负的浮点数转换为一个无符号整数的结果也是未定义的；
6. 将一个较大的浮点数转换为一个较小的浮点数时，在截断之前有可能发生四舍五入。

6.0 存储类型与作用域

术语“存储类型”用于描述编译器为变量分配内存的方式，该术语也可以用于描述某个特定函数的使用范围。C 语言一共有四种存储类型：`auto`、`static`、`extern` 和 `register`。在声明的时候可以省略存储类型，这时编译器将使用默认的存储类型。本节后面将会讨论默认存储类型。

术语“作用域”用于描述某个特定的标识符在程序中的可见范围。定义在任何函数或者语句块（以下我们将函数和语句块统称语句块）外面的标识符可以在同一个文件中随后的任意地方被引用。定义在某个语句块内的标识符只能在该语句块内被引用，因此在该语句块外我们可以定义同名的标识符。标号和形式参数在整个语句块中都可以引用。标号名、结构名与结构成员名、联合与联合成员名、枚举类型的名字以及变量名和函数名只要求在同类中唯一即可（例如，结构类型的名字可以和枚举类型的名字相同）。但是枚举值的名字必须与同一作用域范围内的其他枚举值名字以及变量名不同。

6.1 函数

当为函数指定存储类型的时候，只能使用关键字 `static` 或者 `extern`。声明为 `static` 的函数只能在定义该函数的文件内使用。声明为 `extern`（如果不指定存储类型，则默认为 `extern`）的函数可以在其他文件中使用。

6.2 变量

表 A.6 总结了可以用于变量声明的存储类型关键字，以及各类变量的作用域及初始化方法。

表A.6 变量的存储类型、作用域以及初始化

存储类型	变量声明方式	引用方法	初始化	注释
static	语句块外部	任何位置	常量表达式	变量在程序开始运行的时候初始化; 变量的值在多次运行进入语句块时保持; 变量的初始化值为 0; 变量必须在某处声明时不使用 extern 关键字, 或者某处使用 extern 但是进行了初始化;
	语句块内部	语句块内部		
extern	语句块外部	文件的任何位置	常量表达式	变量必须在某处声明时不使用 extern 关键字, 或者某处使用 extern 但是进行了初始化;
	语句块内部	语句块内部		
auto	语句块内部	语句块内部	任何表达式	当执行流程进入该语句块的时候执行初始化操作, 没有默认值
Register	语句块内部	语句块内部	任何表达式	并不一定确保放在寄存器中; 根据实现不同, 对于能够声明的变量类型有各种限制; 不能取该类型变量的地址; 当执行流程进入该语句块的时候执行初始化操作, 没有默认值

表A.6 续

存储类型	变量声明方式	引用方法	初始化	注释
没有指定	语句块外部	本文件的任何位置，如果包含恰当的声明，也可以从其他文件中引用。	常量表达式	声明只能出现一次，程序开始执行的时候进行初始化，默认值为 0。
	语句块内部	与 auto 相同	与 auto 相同	与 auto 相同

7.0 函数

本小节总结函数相关的语法与操作。

7.1 函数定义

函数定义的一般形式如下：

```
returnType name ( type1 param1, type2 param2, ... )
{
    variableDeclarations

    programStatement
    programStatement
    ...
    return expression;
}
```

上面的定义形式中，函数的名字为`name`，其返回值类型为`returnType`，函数的形式参数是`para1`、`para2`等，其类型分别为`type1`、`type2`等。

通常在函数开始的地方声明内部使用的局部变量。但是C语言规范并不要求这一点。局部变量可以在函数的任何地方声明，只要这些声明语句出现在使用它们的那些语句之前即可。

如果函数没有返回值，那么`returnType`为`void`。

如果在参数列表的括号中指定`void`，那么函数不接收参数。如果参数列表的最后为... (...只能出现在参数列表的最后)，那么函数将接收不定数目的参数，请看下面的函数定义：

```
int printf (char *format, ...)
{
    ...
}
```

如果函数的参数类型为一维数组，那么在参数列表中不需要说明该数组的长度。如果将多维数组作为参数，那么第一个维度上的长度不需要指定。

关于return语句的讨论请看8.9小节。

在函数定义的前面可以加上inline关键字。该关键字暗示编译器将函数的实际代码插入适当的位置，而不是插入调用函数的代码，这样可以获得更快的执行速度。（译者注：编译器可以忽略这个请求）下面是一个inline的例子：

```
inline int min (int a, int b)
{
    return ( a < b ? a : b);
}
```

7.2 函数调用

函数调用的一般形式如下：

```
name ( arg1, arg2, ... )
```

上面的语句形式调用名为name的函数，并将值arg1、arg2等作为参数传递给该函数。如果函数不需要参数的话，应该在后面跟上一对空的小括号（例如initialize()）。

如果程序中的函数调用语句出现在函数定义之前，或者调用另外一个文件中的函数，那么应该写出函数原型声明。函数原型声明的一般形式如下：

```
returnType name (type1 param1, type2 param2, ... );
```

原型声明语句告诉编译器该函数的返回值类型，接收的参数个数以及每个参数的类型。下面是一个函数原型声明的例子：

```
long double power (double x, int n);
```

该语句告诉编译器power是一个函数的名字，该函数的返回值类型为long double。power函数接收两个参数，第一个参数的类型为double，第二个参数的类型为int。原型声明语句中的参数名称没有实际用途，可以省略，如下所示：

```
long double power (double, int);
```

在遇到函数调用语句后，如果前面编译器已经看到过函数的定义或者原型声明，那么调用语句中每个参数将被自动转换为函数所期望的类型（如果自动转换可以进行的话）。

如果在前面没有看到过函数的定义或者原型声明，那么编译器将假定该函数的返回值类型为int，任何float类型的参数将被自动转换为double，而所有的整型都将进行整型提升（参见5.17小节）。对于其他参数将不进行类型转换。

对于接收不定数目参数的函数，必须按照上面说的形式进行原型声明，否则编译器将根据在调用语句中看到的参数个数，推断函数所接收的参数类型。

如果函数的返回值为void的话，那么编译器将对任何使用该函数返回值的语句发出警告。

函数的参数传递按照值引用的方式进行，也就是说，在函数内部不能修改实际的参数。如果给函数传递一个指针参数的话，函数内部可以对该指针指向的位置进行修改，但是不能对实际的指针进行修改。

7.3 函数指针

如果只给出函数名而没有后面的小括号，那么该函数名被编译器当作指向函数的指针。也可以将取地址操作符（&）应用于函数名，同样产生指向该函数的指针。

如果fp是一个指向函数的指针，那么可以使用下面的形式调用该函数：

fp()

或者采用下面的形式：

(*fp)()

如果函数接收参数的话，这些参数可以在小括号中列出。

8.0 语句

C语言的语句由一个合法的表达式（通常是一个赋值表达式或者函数调用表达式）后面跟上一个分号组成；C语言的语句也可以是下面各个小节描述的特殊语句。在任何语句前面都可以加上一个标号，该标号由一个合法的标识符后面跟上冒号组成（参见8.6节）。

8.1 复合语句

包围在一对大花括号之间的一组语句被称为一条复合语句，或者是语句块。在程序中任何可以使用单条语句的地方都可以使用复合语句。在语句块内可以定义局部变量，该局部变量将覆盖在该块外定义的同名变量。这些局部变量的作用域局限在定义它们的语句块内部。

8.2 break 语句

break语句的一般形式如下：

break;

break语句只能用于for、while、do或者switch语句内部，在遇到break语句之后，这些语句将立即结束执行，计算机将接着执行这些语句后面的语句。

8.3 continue 语句

continue语句的一般形式如下：

```
continue;
```

continue语句只能用于循环语句内部。当遇到continue语句之后，循环体中continue语句后面的语句将被跳过，计算机将接着开始执行下一次循环。

8.4 do 语句

do语句的一般形式如下：

```
do  
    programStatement  
while ( expression );
```

如果表达式expression的值不为0，那么计算机将不断的执行programStatement所代表的语句。读者需要注意，因为该表达式在循环体执行完之后才被求值，所以计算机肯定至少要执行一次循环体。

8.5 for 语句

for语句的一般形式如下：

```
for ( expression_1; expression_2; expression_3 )  
    programStatement
```

在循环开始执行的时候将对表达式expression_1求值一次；接下来，计算机将对表达式expression_2求值，如果求值结果不为0，那么计算机将执行programStatement语句，然后对表达式expression_3求值。只要表达式expression_2的结果不为0，那么计算机将一直执行programStatement语句并对表达式expression_3求值。读者需要注意，因为表达式expression_2的求值在循环体programStatement的执行之前，因此如果一开始表达式expression_2的值就是0的话，计算机将一次也不执行循环体。

for语句的表达式expression_1中可以声明只在该循环内部起作用的局部变量。请看下面的例子：

```
for ( int i = 0; i < 100; ++i )  
    ...
```

上面的语句声明了整形变量i，并在开始循环的时候将其值初始化为0。循环内的任何语句都可以访问该变量，但是在循环结束之后，该变量将不可访问。

8.6 goto 语句

goto语句的一般形式如下:

```
goto identifier;
```

执行goto语句将使得程序的执行流程直接转到`identifier`对应的那条语句。该标号的语句必须和goto语句位于同一个函数中。

8.7 if 语句

if语句的一种常用形式如下:

```
if ( expression )  
    programStatement
```

如果表达式`expression`的值不为0,那么`programStatement`将被执行,否则的话该语句将被跳过。

另一种if语句的常用形式如下:

```
if ( expression )  
    programStatement_1  
else  
    programStatement_2
```

如果表达式`expression`的值不为0,那么`programStatement_1`将被执行,否则的话,计算机将执行语句`programStatement_2`。如果`programStatement_2`本身也是一个if语句,那么实际上就组成了一个if-else if语句链,如下所示:

```
if ( expression_1 )  
    programStatement_1  
else if ( expression_2 )  
    programStatement_2  
...  
else  
    programStatement_n
```

在这种情况下,else子句总是和最后一个没有else子句的if语句配对。利用大花括号可以改变这种关联性。

8.8 空语句

空语句的一般形式如下:

```
;
```

空语句的执行不产生任何效果。空语句通常被用来作为for、do或者while语句的一个组成部分用以满足C语言的语法要求。比如在下面的语句中,我们使用一个while语句将字符串`from`中的内容逐个拷贝到字符串`to`中,该语句就使用了空语句:

```
while ( *to++ = *from++ )  
    ;
```

在上面的例子中，我们使用空语句满足了在while语句后面必须有一个语句的语法要求。

8.9 return 语句

return语句的一种常见形式如下：

```
return;
```

执行return语句将使得程序的执行流程立刻回到调用者。这种形式的return语句只能用在那些不返回值的函数中。

如果计算机执行到了函数的最后一条语句，但是还没有遇到return语句，执行流程仍将返回调用者，就仿佛已经执行了return语句一样。在这种情况下，函数将不返回值。

return语句的第二种常见形式如下所示：

```
return expression;
```

这个语句将expression的值作为函数返回值返回给调用者。如果该表达式的值类型与函数声明的返回值类型不同，这个值将被自动转换为需要的类型。

8.10 switch 语句

switch语句的一般形式如下：

```
switch ( expression )
{
    case constant_1:
        programStatement
        programStatement
        ...
        break;
    case constant_2:
        programStatement
        programStatement
        ...
        break;
    ...
    case constant_n:
        programStatement
        programStatement
        ...
        break;
    default:
        programStatement
        programStatement
        ...
        break;
}
```


计算机将首先对表达式`expression`求值，然后将结果与`case`语句后面的常量表达式`constant_1`、`constant_2`、...`constant_n`进行比较。如果该表达式的值与某个常量表达式相匹配的话，计算机将执行该`case`语句后面的语句。如果一个都不匹配的话，计算机将执行`default`语句后面的语句。如果没有`default`语句的话，那么计算机将不执行`switch`语句中的任何语句。

表达式`expression`的类型必须是整型数，而且任意两个`case`语句中的常量表达式值不应该相等。如果在某个`case`语句后面没有`break`语句的话，那么计算机将接着执行下面的`case`语句中的语句。

8.11 while 语句

`while`语句的一般形式如下：

```
while ( expression )  
    programStatement
```

只要表达式`expression`的值不等于0，那么计算机就将一直执行语句`programStatement`。读者需要注意，因为表达式`expression`的求值在循环体执行前进行，因此计算机有可能一次也不执行`programStatement`语句。

9.0 预处理器

预处理器在编译器编译代码之前对其进行处理。预处理器一般完成如下几类工作：

1. 将某些三元组替换为对应的字符（参见 9.1 小节）；
2. 将所有以反斜线\结尾的行与后面一行合并为同一行；
3. 将程序分解为记号流；
4. 删除所有的注释，并用单个空格替换它们；
5. 处理预处理器指令（参见 9.2 小节）并展开所有的宏。

9.1 三元组

为了能够处理程序中的非ASCII字符，C语言定义了三元组，表A.7中列出了所有的三元组。当程序中出现三元组时，预处理器就用对应的字符替换它（在字符串中也是这样）。

表A.7 三元组序列

三元组	含义
??=	#
??([
??)]
??<	{

表A.7 续

三元组	含义
??>	}
??/	\
??'	^
??!	
??-	~

9.2 预处理器指令

所有的预处理器指令都必须以#开始，而且#必须是该行的第一个非空白字符。在#后面可以有一个或者多个空格和tab字符。

9.2.1 #define 指令

#define指令的一般形式如下：

```
#define name text
```

上面的语句定义了一个名为`name`的宏，并将该宏与其名字后的第一个空格后直到该行结束的字符串等价起来。C语言预处理器将用这个字符串替换随后程序中任何位置出现的符号`name`。

#define指令的另外一种常见形式如下：

```
#define name(param_1, param_2, ..., param_n) text
```

上面的语句定义了一个名为`name`的宏，该宏接收一组参数`param_1`、`param_2`、...、`param_n`。在随后的程序中任何出现`name`的地方，预处理器将使用后面的`text`替换该`name`，并使用实际的参数替换`text`中的形式参数。

如果宏接收不定数目的参数，那么可以在参数列表后面跟上...。在宏定义中可以使用符号`__VA_ARGS__`来引用这些参数。下面的示例语句定义了一个名为`myPrintf`的宏，该宏即可接收不定数目的参数：

```
#define myPrintf(...) printf ("DEBUG: " __VA_ARGS__);
```

我们可以按照如下的形式使用该宏定义：

```
myPrintf ("Hello world!\n");
```

也可以使用如下的形式：

```
myPrintf ("i = %i, j = %i\n", i, j);
```


如果宏定义跨越多行的话，在每一个后面有续行的行最后都应该加上反斜线\。当定义了宏之后，就可以在同一个文件后面的任意位置使用该宏了。

在接收参数的宏定义中可以使用#操作符。#后面应该跟上一个参数。预处理器将用双引号将该参数括起来，这样该参数就被编译器当作一个字符串处理。请看下面的例子：

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

按照上面的定义，下面的语句：

```
printint (count);
```

将被展开为如下的形式：

```
printf ("count" " = %i\n", count);
```

也就是下面的形式：

```
printf ("count = %i\n", count);
```

在执行#操作的时候，预处理器将给参数中的"和\前面再放置一个\符号（译者注：也就是将它们转义为真正的字符）。因此，如果有如下形式的宏定义：

```
#define str(x) # x
```

那么下面的调用语句：

```
str (The string "\t" contains a tab)
```

将被展开为如下的形式：

```
"The string \"\\t\" contains a tab"
```

在接收参数的宏定义中也可以使用##操作符。该操作符的前面或者后面是一个实际参数的名字。在展开宏的时候，编译器将把该参数的值和它前面（或者后面）的记号连接起来，形成一个新的记号。请看下面的宏定义语句：

```
#define printx(n) printf ("%i\n", x ## n );
```

按照该定义，下面的语句：

```
printx (5)
```

将被展开为如下的形式：

```
printf ("%i\n", x5);
```

同样，如果有如下的宏定义：

```
#define printx(n) printf ("x" # n " = %i\n", x ## n );
```

那么下面的语句：

```
printx(10)
```

将被展开为如下的形式:

```
printf ("x10 = %i\n", x10);
```

在#操作符和##操作符周围不允许有空格。

9.2.2 #error 指令

#error指令的一般形式如下:

```
#error text  
...
```

指令后面的text将被预处理器作为错误信息输出。

9.2.3 #if 指令

#if指令的一种常用形式如下所示:

```
#if constant_expression  
...  
#endif
```

预处理器将对常量表达式constant_expression求值, 如果求值结果为非0, 那么#if和#endif之间的语句将被处理。否则的话, 预处理器和编译器都不会处理这些语句。

另外一种#if指令的常用形式如下:

```
#if constant_expression_1  
...  
#elif constant_expression_2  
...  
#elif constant_expression_n  
...  
#else  
...  
#endif
```

如果常量表达式constant_expression_1的值不为0, 那么在该表达式后直到第一个#elif指令之间的语句将被处理, 剩下直到#endif之间的语句将被忽略。否则的话, 如果constant_expression_2的值不为0, 那么在该表达式后直到第一个#elif指令之间的语句将被处理, 剩下直到#endif之间的语句将被忽略。如果所有的常量表达式的值都是0的话, 那么#else和#endif之间的语句将被处理。

在常量表达式中可以使用一个特殊的操作符defined, 请看下面的语句:

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


在上面的语句中,如果符号DEBUG已经定义过的话,#if和#endif之间的语句将被处理。另外,符号周围的小括号也不是必须的,因此下面的语句也是可以的:

```
#if defined DEBUG
```

9.2.4 #ifdef 指令

#ifdef语句的一般使用形式如下:

```
#ifdef identifier  
...  
#endif
```

如果identifier代表的宏已经被定义过了(可能是通过#define语句,也可能是通过命令行上的-D选项),#ifdef和#endif之间的语句将被处理,否则的话,这些语句将被忽略。如同#if指令一样,#ifdef指令后面也可以有#elif指令和#else指令。

9.2.5 #ifndef 指令

#ifndef指令的一般使用形式如下:

```
#ifndef identifier  
...  
#endif
```

如果identifier代表的宏还没有被定义过,那么#ifndef和#endif之间的指令将被处理,否则的话,这些语句将被忽略。如同#if指令一样,#ifndef指令后面也可以有#elif指令和#else指令。

9.2.6 #include 指令

#include指令最常见的使用形式如下:

```
#include "fileName"
```

预处理器将在某些“实现特定”的目录中寻找指定的文件。典型的,预处理器将在当前源程序所在的目录中寻找该文件。如果在当前目录中没有找到该文件,那么编译器将在某些特定的目录中(这些目录由编译器的实现厂商决定)寻找该文件。在找到了该文件之后,文件的内容将被插入到#include指令所在的位置。随后,预处理器将对这些内容进行分析。所以一个被#include指令引入的文件中,还可以包含另外一个#include指令。

#include指令另外一个常见的形式如下:

```
#include <fileName>
```

这时,预处理器将在标准目录中寻找这些文件。找到文件之后的行为与前面所说的相同。

无论何种形式，我们都可以使用宏来代替具体被包含的文件名，请看下面的例子：

```
#define DATABASE_DEFS </usr/data/database.h>
...
#include DATABASE_DEFS
```

9.2.7 #line 指令

#line指令使用的一般形式如下：

```
#line constant "fileName"
```

当看到这个指令之后，编译器就将随后的语句当作是`fileName`中指定的文件中的语句，其行号从`constant`重新开始计算。如果没有指定`fileName`的话，那么编译器将默认是上一个#line指令中指定的文件名或者当前源文件的名称（如果从来没有使用#line指令指定过文件名的话）。

#line指令用于帮助编译器，在发出错误报告时给出正确的文件名和行号。（译者注：考虑到包含关系。）

9.2.8 #pragma 指令

#pragma指令的常见形式如下：

```
#pragma text
```

当预处理器看到这条指令之后，将执行某些与特定编译器实现相关的动作。例如下面的语句：

```
#pragma loop_opt(on)
```

有可能用来告诉编译器打开循环优化。不认识该指令的编译器将忽略该指令。

在#pragma指令后面可以使用一个特殊的关键字STDC，目前的规范中规定STDC后面可以使用的开关有：FP_CONTRACT、FENV_ACCESS和CX_LIMITED_RANGE。

9.2.9 #undef 语句

#undef语句的一般形式如下：

```
#undef identifier
```

该指令将取消前面定义的宏`identifier`。随后的#ifdef或者#ifndef指令将认为该宏没有被定义。

9.2.10 #指令

这是一个空指令，预处理器将忽略它。

9.3 预定义符号

表A.8列出了C语言编译器的预定义符号。

表A.8 预定义符号

符号	含义
__LINE__	当前编译的行号
__FILE__	当前正在编译的文件
__DATE__	当前的日期，以“月月 日日 年年年年”的形式给出
__TIME__	当前的时间，以“时时: 分分: 秒秒”的形式给出
__STDC__	如果编译器符合ANSI C标准，该宏为1，否则为0
__STDC_HOSTED__	如果实现了所有C标准库，则该宏为1，否则为0
__STDC_VERSION__	被定义为 199901L

The Standard C Library

附录 B C 语言标准库

C语言标准库中包含有大量的库函数。本附录没有全部列出这些函数，而只包括最常用的那些函数。如果读者想要获得函数的完整列表，可以参阅自己所使用的编译器附带的文档，或者浏览附录E“C语言的其他资源”中的部分资料。

本附录没有介绍的函数包括：日期与时间（比如time、ctime或者localtime）、非局部跳转（setjmp和longjmp）、诊断函数（assert）、不定参数处理（va_list、va_start、va_arg和va_end）、信号处理（signal和raise）、本地化（在标准头文件<locale.h>中定义的那些）以及宽字符处理。

标准头文件

本小节描述标准头文件<stddef.h>、<stdbool.h>、<limits.h>、<float.h>和<stdinit.h>中的内容。

<stddef.h>

该头文件包含有如下的符号定义：

定义	含义
NULL	空指针
offsetof(structure, member)	结构的成员变量member到结构开始地址的偏移量结果的类型为size_t
ptrdiff_t	两个指针相减所得结果的类型

定义	含义
size_t	sizeof操作符的运算结果的类型
wchar_t	宽字节字符类型（参见附录A：C语言小结）

<limits.h>

该头文件包含了字符类型和整数类型与具体实现的相关极限值。ANSI C标准规定了某些条目的最小值，在这些条目后的小括号中给出。

定义	含义
CHAR_BIT	char类型的位数（8）
CHAR_MAX	char类型的最大值（有符号的char是127，无符号的char是255）
CHAR_MIN	char类型的最小值（有符号的char是-127，无符号的char是0）
SCHAR_MAX	signed char类型的最大值（127）
SCHAR_MIN	signed char类型的最小值（-127）
UCHAR_MAX	unsigned char类型的最大值（255）
SHRT_MAX	short int类型的最大值（32767）
SHRT_MIN	short int类型的最小值（-32767）
USHRT_MAX	unsigned short类型的最大值（65535）
INT_MAX	int类型的最大值（32767）
INT_MIN	int类型的最小值（-32767）
UINT_MAX	unsigned int类型的最大值（65535）
LONG_MAX	long int类型的最大值（2147483647）
LONG_MIN	long int类型的最小值（-2147483647）
ULONG_MAX	unsigned long类型的最大值（4294967295）
LLONG_MAX	long long int类型的最大值（9223372036854775807）
LLONG_MIN	long long int类型的最小值（-9223372036854775807）
ULLONG_MAX	unsigned long long int类型的最大值（18446744073709551615）

<stdbool.h>

该头文件包含了用于处理布尔变量的宏。

定义	含义
bool	与_Bool类型等价
true	1
false	0

<float.h>

该头文件定义了若干与浮点数具体实现相关的极限值。ANSI C所要求的最小值在每个项目的括号中列出。读者注意，这里并没有列出所有的定义。

定义	含义
FLT_DIG	float类型的精度(6)
FLT_EPSILON	与1.0相加后所得结果不等于1.0的最小值($1e-5$)
FLT_MAX	最大的浮点数($1e+37$)
FLT_MAX_EXP	最大的浮点数($1e+37$)
FLT_MIN	最小的浮点数($1e-37$)

对于double类型和long double类型也存在类似的定义。这些宏的名字仅仅前三个字母发生变化，double类型使用DBL，long double使用LDBL。因此，DBL_DIG代表double类型的精度，而LDBL_DIG代表long double类型的精度。

在头文件<fenv.h>中还有部分函数，可以用于控制C语言的浮点数运算。例如，该头文件声明了函数fesetround，使用该函数我们可以控制C语言对于浮点数取整时的行为方式，包括FE_TONEAREST（四舍五入）、FE_UPWARD（进位）、FE_DOWNWARD（退位）和FE_TOWARDZERO（接近0）一共四种。读者也可以使用函数feclearexcept、feraiseexcept和fetestexcept函数来清除、设置或者检查浮点数异常。

<stdint.h>

该头文件中包含有很多类型定义和常量，用于帮助我们以一种与机器无关的方式使用整型数。例如，我们可以使用类型int32_t声明一个刚好32位的有符号整数，而不需要关心在程序运行的平台上到底哪个整数的长度刚好是32位。与之类似，我们也可以使用类型int_least32_t声明一个最少有32位的整型数。该文件中还包含有运算最快的整型数定义等等。如果想要了解关于这方面的更多信息，可以参考读者自己使用的编译器的相关文档。

该文件中还有几个重要的类型定义，如下所示：

定义	含义
intptr_t	能够容纳指针的整型数类型
uintptr_t	能够容纳指针的无符号整型数类型
intmax_t	最大的有符号整数类型
uintmax_t	最大的无符号整型数类型

字符串函数

本节讲述处理字符串的函数。在下面的描述中，我们假定符号 s 、 $s1$ 、 $s2$ 都代表指向以空字符结尾的字符数组的指针，而 c 代表整型数， n 代表 `size_t` 类型的整型数（该类型在标准头文件 `<stddef.h>` 中定义）。对于 `strnxxx` 系列函数， $s1$ 和 $s2$ 可以不以空字符结尾。

为了使用这些函数，读者应该在程序中包含标准头文件 `<string.h>`，如下所示：

```
#include <string.h>
```

```
char *strcat (s1, s2)
```

将 $s2$ 代表的字符串附加到 $s1$ 后面，并在最后加上空字符。函数的返回值是 $s1$ 。

```
char *strchr (s, c)
```

在字符 s 中寻找字符 c 出现的第一个位置。如果找到了该字符，那么函数返回指向该位置的指针，否则的话返回 `NULL`。

```
int strcmp(s1, s2)
```

比较字符串 $s1$ 和字符串 $s2$ 。如果 $s1 < s2$ ，返回值小于 0，如果 $s1 == s2$ ，返回值等于 0，如果 $s1 > s2$ ，返回值大于 0。

```
char *strcoll(s1, s2)
```

就像 `strcmp` 一样，只是在当前的语言环境下比较字符串 $s1$ 和字符串 $s2$ 。

```
char *strcpy(s1, s2)
```

将字符串 $s2$ 复制到 $s1$ 中，返回 $s1$ 。

char *strerror(n)

返回错误号 n 对应的错误信息。

size_t strcspn(s1, s2)

返回 $s1$ 中出现 $s2$ 中任意一个字符的第一个位置（以下标形式）。

size_t strlen(s)

返回 s 中的字符数目（不包括结尾的空字符）

char *strncat(s1, s2, n)

把 $s2$ 拷贝到 $s1$ 的结尾，直到遇到空字符或者已经拷贝了 n 个字符。返回 $s1$ 的值。

int strncmp(s1, s2, n)

与strcmp的功能相同，但是最多只比较 n 个字符。

char *strncpy(s1, s2, n)

将 $s2$ 中的内容拷贝到 $s1$ 中，直到遇到空字符或者已经拷贝了 n 个字符。返回 $s1$ 。

char *strrchr(s, c)

从后向前在 s 中查找字符 c 。如果找到的话，返回指向该位置的指针。否则的话返回NULL。

char *strpbrk(s1, s2)

在 $s1$ 中查找 $s2$ 中的任意字符。如果找到的话，返回指向该位置的指针。否则的话返回NULL。

size_t strspn(s1, s2)

返回 $s1$ 中第一个不是 $s2$ 中字符的位置（以下标形式）。

char *strstr(s1, s2)

在字符串 $s1$ 中从前向后查找 $s2$ 。如果找到的话，返回指向该字符串的指针，否则的话返回NULL。

char *strtok(s1, s2)

根据 $s2$ 中给出的分界字符将 $s1$ 分解成记号流。第一次调用的时候， $s1$ 中包含需要被分析的字符串， $s2$ 中包含分界字符。随后每次调用的时候，该函数在每个记号结束处放置一个空字符，然后返回指向该记号的指针。随后的调用中 $s1$ 应该为NULL。当 $s1$ 中再也没有记号时，该函数将返回空指针。


```
size_t strxfrm(s1, s2, n)
```

根据语言环境信息对于字符串 s_2 进行翻译，并将结果放到字符串 s_1 中。最多翻译 n 个字符。

内存函数

本小节的函数用于处理字符数组。我们可以使用这些函数高效的在内存中搜索或者拷贝数据。使用这些函数需要包含标准头文件<string.h>，如下所示：

```
#include <string.h>
```

在下面的叙述中，我们假定 m_1 和 m_2 是`void*`类型的指针， c 是`int`类型，但是在这些函数内部， c 将被转换为`unsigned char`类型。 n 是类型为`size_t`的整数。

```
void *memchr(m1, c, n)
```

在从 m_1 开始的内存中查找字符 c ，并返回指向该位置的指针。如果在查找了 n 个字符之后还没有找到，那么返回`NULL`。

```
void *memcmp(m1, m2, n)
```

比较 m_1 开始的内存和 m_2 开始的内存中的前 n 个字符。如果这两个地址对应的前 n 个字符相等，那么函数返回0；否则的话，返回导致其不等的两个字符之间的差。因此，如果 m_1 中导致不等的字符小于 m_2 中导致不等的字符，函数返回值小于0，否则的话大于0。

```
void *memcpy(m1, m2, n)
```

从 m_2 中拷贝 n 个字符到 m_1 中，返回 m_1 。

```
void *memmove(m1, m2, n)
```

与`memcpy`的作用相同，但是在 m_1 和 m_2 重叠的情况也能保证正常工作。

```
void *memset(m1, c, n)
```

将 m_1 中的前 n 个字符的值设置为 c 。返回 m_1 。

读者需要注意，本小节的函数对于空字符并不进行特殊的处理。另外，我们可以将其其他非字符数组类型的指针转型为`void*`，以使用这些函数。比如，假定`data1`和`data2`分别是包含100个整型数的数组，那么下面的代码把`data1`中的100个整数复制到`data2`中。

```
memcpy ((void *) data2, (void *) data1, sizeof (data1));
```

字符函数

本小节的函数可以用于处理单个字符。为了使用这些函数，我们需要在程序中包含标准头文件<ctype.h>，如下所示：

```
#include <ctype.h>
```

这些函数都接收一个int型的参数，如果测试条件成立的话，函数返回非零值，否则的话返回0。

名字	测试
isalnum	是否是字母或者数字
isalpha	是否是字母
isblank	是否是空白字符（空格或者Tab）
isctrl	是否是控制字符
isdigit	是否是数字
isgraph	是否是图形字符（即除了空格之外的任何字符）
islower	是否是小写字母
isprint	是否是可打印字符（包括空格）
ispunct	是否是标点符号（除了空格、字母、数字之外的任何字符）
isspace	是否是空白字符（空格、回车、换行、水平或者垂直制表、表单）
isupper	是否是大写字母
isxdigit	是否是十六进制字符

下面两个函数可以用于字符翻译：

```
int tolower(c)
```

返回与大写字母c对应的小写字母，如果c不是大写字母，那么返回值就是c本身。

```
int toupper(c)
```

返回与小写字母c对应的大写字母，如果c不是小写字母，那么返回值就是c本身。

输入输出函数

本小节描述C语言中最常用的输入输出函数。为了使用这些函数，我们应当在程序中包含标准头文件<stdio.h>，如下所示：

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


该头文件中包含着输入输出函数的原型定义, 以及下列符号定义: EOF、NULL、stdin、stdout、stderr (全部为常量), 还有类型FILE。

在下面的叙述中, *fileName*、*fileName1*、*fileName2*、*accessMode*和*format*都是指向字符串的指针; *buffer*是指向字符数组的指针; *filePtr*是文件类型的指针, *n*和*size*是size_t类型的整数, *i*和*c*是int类型的整数。

```
void clearerr (filePtr)
```

清除*filePtr*对应文件的错误标志。

```
int fclose(filePtr)
```

关闭*filePtr*对应的文件, 如果关闭成功, 返回0; 如果失败返回EOF。

```
int feof(filePtr)
```

如果*filePtr*对应的文件已经到达了结尾, 返回非零值, 否则返回0。

```
int ferror(filePtr)
```

如果*filePtr*对应的文件在I/O操作中发生了错误, 返回非零值, 否则返回0。

```
int fflush(filePtr)
```

将内部缓冲区写到文件中。如果成功返回0值, 否则的话返回EOF。

```
int fgetc(filePtr)
```

返回*filePtr*对应的文件中的下一个字符。如果遇到文件结束, 则返回EOF。读者需要注意, 该函数的返回值为int类型。

```
int fgetpos(filePtr, fpos)
```

返回*filePtr*对应文件的当前位置, 并将结果保存在*fpos*所指向的fpos_t类型的变量中 (该类型在<stdio.h>中定义)。如果成功返回0值, 如果失败返回非零值。请参看函数fsetpos的介绍。

```
char* fgets(buffer, i, filePtr)
```

从*filePtr*对应的文件中读取字符, 直到读入了*i*-1个字符或者遇到了换行符。读取的字符保存在*buffer*对应的字符数组中。读取的换行符将保存在字符数组中。如果读取的过程中发生了错误或者遇到了文件结尾, 那么函数返回NULL, 否则的话返回*buffer*。

```
FILE* fopen(fileName, accessMode)
```

使用`accessMode`指定的模式打开`fileName`对应的文件。有效的模式包括：`r`对应读，`w`对应写，`a`对应文件末尾追加，`r+`对应读写打开，但是文件指针的最初位置在文件的开始，`w+`对应读写打开，但是如果原来文件存在的话将清除其内容，`a+`对应读写打开，但是所有输出的字符都追加文件的末尾。如果需要打开的文件不存在的话，那么使用写模式（`w`，`w+`）和追加模式（`a`，`a+`）打开该文件将创建一个空白文件。如果文件以追加模式打开，那么向文件中写数据时不会覆盖原有的内容。

对于区分二进制文件和文本文件的系统来说，打开二进制文件时，必须在打开模式后面加上代表二进制的字母`b`。

如果文件打开操作成功的话，函数将返回一个对应该文件的`FILE*`类型的指针，我们随后可以使用该指针对文件进行读写操作。否则的话函数返回`NULL`。

```
int fprintf(filePtr, format, arg1, arg2, ..., argn)
```

按照参数`format`中指定的格式，将参数`arg1`、`arg2`直到`argn`输出到`filePtr`对应的文件中。该函数使用的格式化字符串与`printf`函数相同（参见第16章“C语言的输入输出”）。函数返回输出的字符个数。如果返回值小于0，说明输出过程发生了错误。

```
int fputc(c, filePtr)
```

将`c`的值（在函数内部将转换为`unsigned char`）输出到`filePtr`对应的文件中。如果输出成功的话返回`c`，失败的话返回`EOF`。

```
size_t fread(buffer, size, n, filePtr)
```

从文件中读取`n`个数据项到`buffer`对应的字符数组中。每个数据项的大小为`size`。例如，下面的调用语句：

```
numread = fread (text, sizeof (char), 80, in_file);
```

从`in_file`对应的文件中读取80个字符，并将它们保存在`text`对应的字符数组中。这个函数返回成功读取的字符数目。

```
FILE* freopen(fileName, accessMode, filePtr)
```

关闭`filePtr`对应的文件，以`accessMode`中指定的模式（参见`fopen`函数）打开`fileName`对应的文件，并将该文件与文件指针`filePtr`关联起来。

如果操作成功的话，`freopen`函数返回`filePtr`。否则的话返回`NULL`。`freopen`函数可以用于将标准输入、标准输出和标准错误重定向到其他文件。例如，下面的调用语句：

```
if ( freopen ("inputData", "r", stdin) == NULL ) {
    ...
}
```

将文件指针`stdin`重新对应到文件`inputData`上。执行上述函数与在命令行上进行重定向的作用相同。

```
int fscanf(filePtr, format, arg1, arg2, ..., argn)
```

根据格式化字符串`format`中指定的格式，从`filePtr`对应的文件中读取字符，并将转换结果赋给参数`arg1`、`arg2`等。`arg1`、`arg2`、`argn`必须是指针类型的变量。`fscanf`函数使用的格式化字符串与`scanf`函数相同（参见第16章）。`fscanf`将成功读取并赋值的参数个数（除了`%n`）作为返回值返回给调用者。如果在读取第一个参数之前遇到了文件结束标记的话，该函数返回`EOF`。

```
int fseek(filePtr, offset, mode)
```

根据`offset`中指定的偏移值，以及`mode`中指定的偏移值计算方式，重新设置文件指针的当前位置。偏移计算方式包括如下几种：`SEEK_SET`（表示从文件开始计算）、`SEEK_CUR`（表示从当前位置计算）和`SEEK_END`（表示从文件结束计算）。这三个常数在头文件`<stdio.h>`中定义。

对于区分文本文件和二进制文件的系统来说，`SEEK_END`有可能不被支持。对于文本文件来说，偏移值要么为0，要么必须是`ftell`的返回值。在后面这种情况下，`mode`必须使用`SEEK_SET`。如果`fseek`操作失败的话，函数返回非零值。

```
int fsetpos(filePtr, fpos)
```

将`filePtr`对应文件的当前位置设置为`fpos`所代表的值。`fpos`的类型为`fpos_t`，该类型在`<stdio.h>`中定义。如果函数成功则返回0，否则的话返回非零值。参看函数`fgetpos`。

```
long ftell(filePtr)
```

返回`filePtr`对应文件当前位置的偏移值。如果发生错误则返回-1L。

```
size_t fwrite(buffer, size, n, filePtr)
```

将字符数组`buffer`中包含的`n`个数据项写到`filePtr`对应的文件中。每个数据项的大小为`size`。返回成功写出的字节数。

```
int getc(filePtr)
```

读取文件中的下一个字符。如果遇到文件结束标志或者发生错误，函数将返回EOF。

```
int getchar(void)
```

读取标准输入的下一个字符。如果遇到文件结束标志或者发生错误，函数将返回EOF。

```
char *gets(buffer)
```

从标准输入读取字符，直到遇到换行符为止。读取的换行符将不保存在字符数组中，读取的字符最后将加上一个表示空字符。如果读取的过程中发生错误，或者没有读取任何数据，那么函数将返回NULL，否则的话返回`buffer`。

```
void perror(message)
```

将上一个错误的详细解释输出到标准错误，前面附加上`message`对应的字符串。例如，下面的代码：

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

if ( (in = fopen ("data", "r")) == NULL ) {
    perror ("data file read");
    exit (EXIT_FAILURE);
}
```

如果`fopen`函数操作失败，上面的代码将有可能打印出关于失败原因更详细的信息。

```
int printf(format, arg1, arg2, ..., argn)
```

按照`format`中指定的格式将参数`arg1`、`arg2`直到`argn`输出到标准输出（参见第16章）。返回输出的字符个数。

```
int putc(c, filePtr)
```

将`c`对应值（转换为一个`unsigned char`）输出到指定的文件。成功的话返回`c`，失败的话将返回EOF。

```
int putchar(c)
```

将`c`对应值（转换为一个`unsigned char`）输出到标准输出。成功的话返回`c`，失败的话将返回EOF。


```
int puts(buffer)
```

将`buffer`中的字符输出到标准输出，直到遇到空字符为止。输出的结尾将自动加上一个换行符（这一点与`fputs`函数不同）。如果发生错误的话，返回EOF。

```
int remove(fileName)
```

删除指定的文件，如果出错的话返回非零值。

```
int rename(fileName1, fileName2)
```

将文件`fileName1`重命名为`fileName2`，如果出错的话返回非零值。

```
void rewind(filePtr)
```

将文件指针重新定向到文件开始。

```
int scanf(format, arg1, arg2, ..., argn)
```

按照`format`指定的格式从标准输入读取值，并将其赋给参数`arg1`、`arg2`...`argn`。这些参数必须是指针类型。函数的返回值是成功读取并赋值的参数个数（不包括`%n`）。如果没有读取任何项目就遇到了文件结尾，函数返回值为EOF。

```
FILE* tmpfile(void)
```

创建并以写更新模式（`w+b`）打开一个二进制格式的临时文件。如果发生错误的话返回NULL。当程序结束的时候，该文件将被自动删除（C语言标准库还提供了一个名为`tmpnam`的函数用于创建临时文件）。

```
int ungetc(c, filePtr)
```

将一个字符放回到`filePtr`对应的流中。这个字符并不是被写入到文件中，而是放到`filePtr`的读取缓冲区中。下一次调用`getc`的时候将会读到这个字符。使用`ungetc`函数一次只能放回一个字符，而且在调用`ungetc`函数之前，我们必须已经在文件上执行了读操作。如果操作成功的话，该函数返回字符`c`，否则的话返回EOF。

内存中的格式转换函数

使用`sprintf`和`sscanf`函数可以在内存中对于数据进行格式转换。这两个函数与`fprintf`和`fscanf`非常类似，只不过它们使用字符串代替了`fprintf`和`fscanf`中的文件指针。为了使用这两个函数，我们应该在程序中包含标准头文件`<stdio.h>`。

```
int sprintf(buffer, format, arg1, arg2, ..., argn)
```

根据 *format* 中指定的格式对参数 *arg1*、*arg2* 直到 *argn* 进行转换（参见第16章），并将转换的结果放置到 *buffer* 对应的字符数组中。在转换结果的最后，*sprintf* 函数将会自动加上一个空字符。函数返回放置到 *buffer* 中的字符数（不包括结束空字符）。请看下面的示例代码：

```
int version = 2;
char fname[125];
...
sprintf (fname, "/usr/data%i/2005", version);
```

这段代码的执行结果是在字符数组 *fname* 中放置字符串 “/usr/data2/2005”。

```
int sscanf(buffer, format, arg1, arg2, ..., argn)
```

根据 *format* 中指定的格式，从字符数组 *buffer* 中读取字符，并对其进行转换，将结果保存在 *arg1*、*arg2* 直到 *argn* 对应的参数中。函数的返回值是成功读取并转换的参数个数。请看下面的示例代码：

```
char buffer[] = "July 16, 2004", month[10];
int day, year;
...
sscanf (buffer, "%s %d, %d", month, &day, &year);
```

上面的代码在变量 *month* 中保存字符串 “July”，在变量 *day* 中保存16，在变量 *year* 中保存2004。下面的代码：

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

if ( sscanf (argv[1], "%f", &fval) != 1 ) {
    fprintf (stderr, "Bad number: %s\n", argv[1]);
    exit (EXIT_FAILURE);
}
```

将第一个命令行参数（由 *argv[1]* 代表）转换为一个浮点数，并检查转换结果是否成功。（下一小节将介绍其他可以将字符串转化为数字的函数）。

字符串到数字的转换

本小节的函数将字符串转换为对应的数字。为了使用这些函数，读者应该在程序中包含标准头文件 *<stdlib.h>*，如下所示：

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


在下面的描述中, *s* 指向一个空字符结尾的字符串, *end* 代表一个字符指针, *base* 是一个 *int* 类型的变量。

这里描述的所有函数都将跳过字符串开始处的空白字符, 在随后的转换过程中, 如果遇到了对于正在转换的类型来说不合法的字符, 函数将结束运行。

`double atof(s)`

将 *s* 指向的字符串转换为一个 *double* 类型的数字, 返回该数字。

`int atoi(s)`

将 *s* 指向的字符串转换为一个 *int* 类型的数字, 返回该数字。

`long int atol(s)`

将 *s* 指向的字符串转换为一个 *long int* 类型的数字, 返回该数字。

`long long int atoll(s)`

将 *s* 指向的字符串转换为一个 *long long int* 类型的数字, 返回该数字。

`double strtod(s, end)`

将 *s* 指向的字符串转换为 *double* 类型并返回结果。如果 *end* 不为空指针的话, 导致转换结束的字符将被保存在该指针指向的位置。

请看下面的例子:

```
#include <stdlib.h>
...
char buffer[] = " 123.456xyz", *end;
double value;
...
value = strtod (buffer, &end);
```

上面的代码执行完成之后, *value* 的值将等于 123.456, 由于 *strtod* 函数将导致转换停止的字符保存在 *end* 中, 因此 *end* 中的值为字符 'x'。

`float strtof(s, end)`

与 *strtod* 非常类似, 但是字符串被转换为浮点数。

`long int strtol(s, end, base)`

将 *s* 转换为一个 *long int* 并返回结果。*base* 是一个在 2 到 36 之间的数字, 它表示需要转换数字的基数。如果 *base* 的值等于 0, 那么函数将根据字符串自行判断基数: 如果第一个字母是 0 而第二个字母是 x 或者 X, 那么使用十六进制, 如果第一个字母是 0 而第二个字母不是 x 或者 X, 那么使用八进制, 否则的话该数字是十进制。如果 *base* 等于 16, 那么字符串的开始可以是 0x, 也可以是 0X。

如果`end`不为空指针的话，导致转换结束的字符将被保存在该指针指向的位置。

```
long double strtold(s, end)
```

与`strtod`类似，但是字符串被转换为`long double`类型。

```
long long int strtoll(s, end, base)
```

与`strtol`类似，但是字符串被转换为`long long int`类型。

```
unsigned long int strtoul(s, end, base)
```

与`strtol`类似，但是字符串被转换为`unsigned long int`类型。

```
unsigned long long int strtoull(s, end, base)
```

与`strtol`类似，但是字符串被转换为`unsigned long long int`类型。

动态内存分配函数

本小节的函数可以用于动态内存分配与释放。对于本节讨论的函数，`n`和`size`都是类型为`size_t`的整型数，`pointer`为`void*`类型的指针。为了使用这些函数，程序中应该包含如下一行：

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

```
void *calloc(n, size)
```

为`n`个数据项分配连续的内存空间，每个数据项的大小为`size`。分配的内存将全部被初始化为0。如果成功的话，返回指向该块内存首地址的指针，否则的话返回`NULL`。

```
void free(pointer)
```

释放由`pointer`指向的，由前面的`calloc`、`malloc`或者`realloc`调用分配的内存。

```
void *malloc(size)
```

分配大小为`size`的连续内存空间。如果成功的话，返回指向该块内存首地址的指针，否则的话返回`NULL`。

```
void *realloc(pointer, size)
```

改变先前分配的一块内存的大小，重新设置为`size`字节。返回指向新内存块的首地址。如果失败的话返回`NULL`。

数学函数

本节列出了C语言中的数学函数，如果要在程序中使用这些函数，应该包含如下一行：

```
#include <math.h>
```

标准头文件<tgmath.h>中定义了部分类型无关的宏，使用这些宏，我们可以很方便的调用数学库或者复数数学库中的函数而无须担心具体的参数类型。例如，根据参数类型和返回值的类型不同，我们可以使用如下六个函数计算某个数的平方根：

```
double sqrt (double x)
float sqrtf (float x)
long double sqrtl (long double x)
double complex csqrt (double complex x)
float complex csqrtf (float complex f)
long double complex csqrtl (long double complex)
```

无需担心这六个函数，我们通过包含<tgmath.h>，而不是<math.h>和<complex.h>，我们就可以使用对应的通用参数版本的sqrt宏，该宏将根据具体的参数类型调用合适的函数。

让我们接着讨论标准头文件<math.h>中的函数。使用下面的函数，我们可以判断浮点数的某些特性。

```
int fpclassify(x)
```

检查 x 所属的类别：无效数字（FP_NAN）、无穷数字（FP_INFINITE）、正常数字（FP_NORMAL）、0值（FP_ZERO），或者其他与具体实现相关的类别等。这些常数在头文件<math.h>中定义。

```
int isfin(x)
```

x 是否代表一个有限值。

```
int isinf(x)
```

x 是否代表一个无穷值。

```
int isgreater(x, y)
```

x 是否大于 y ？

```
int isgreaterequal(x, y)
```

x 是否大于等于 y ？

```
int islessequal(x, y)
```

x 是否小于等于 y ?

```
int islessgreater(x, y)
```

x 是否大于或者小于 y ?

```
int isnan(x)
```

x 是否是一个无效数?

```
int isnormal(x)
```

x 是否是一个正常数字?

```
int isunordered(x, y)
```

x 和 y 之间是否无序（比如 x 和 y 中有一个是NaN或者都是）?

```
int signbit(x)
```

x 是否是负数?

在下面的例子中， x 、 y 和 z 都是double类型， r 是一个弧度制表示的角度，其类型为double， n 为int类型。

关于这些函数报告错误的方法，请参考具体编译器的文档。

```
double acos(x)1
```

计算 x 的反余弦值，它是一个以弧度形式表示的角度，范围在 $[0, \pi]$ 之间。 x 的值的范围为 $[-1, 1]$ 。

```
double acosh(x)
```

计算 x 的双曲反余弦值， x 大于等于1。

```
double asin(x)
```

计算 x 的正弦值，它是一个以弧度形式表示的角度，范围在 $[-\pi/2, \pi/2]$ 之间。 x 的值的范围为 $[-1, 1]$ 。

```
double asinh(x)
```

计算 x 的双曲反正弦值。

¹数学库中包含有浮点、双精度和长双精度三种版本的函数。我们这里以 double 版本为例对其进行介绍。float 版本的函数名以 f 结尾（如 acosf），long double 版本的函数名以 l 结尾（如 acosl）。

`double atan(x)`

计算 x 的反正切值，它是一个以弧度形式表示的角度，范围在 $[-\pi/2, \pi/2]$ 之间。

`double atanh(x)`

计算 x 的双曲反正切值， $|x|$ 小于等于1。

`double atan2(x, y)`

计算 x/y 的反正切值，它是一个以弧度形式表示的角度，范围在 $[-\pi, \pi]$ 之间。

`double ceil(x)`

计算大于或者等于 x 的最小整数值。请注意函数返回值的类型为`double`。

`double copysign(x, y)`

返回一个值，该值的绝对值与 x 的绝对值相同，但是符号与 y 相同。

`double cos(x)`

计算 x 的余弦值。

`double cosh(x)`

计算 x 的双曲余弦值。

`double erf(x)`

计算 x 的错误函数。

`double erfc(x)`

计算 x 的补余错误函数。

`double exp(x)`

计算 e^x

`double expm1(x)`

计算 $e^x - 1$ 。

`double fabs(x)`

计算 x 的绝对值。

`double fdim(x, y)`

如果 $x > y$ 返回 $x - y$ ，否则的话返回0。

`double floor(x)`

返回小于等于 x 的最大整数值，注意函数返回值的类型为`double`。

double fma(x, y, z)

计算 $(x*y) + z$ 。

double fmax(x, y)

返回x和y中的最大值。

double fmin(x, y)

返回x和y中的最小值。

double fmod(x, y)

返回x除以y的余数，结果的符号与x的符号相同。

double frexp(x, exp)

将x分解为一个真分数和一个2的指数的乘积。真分数作为函数的返回值，指数保存在exp指向的位置中。如果x等于0，那么返回值和exp指向的值都为0。

int hypot(x, y)

计算 $x^2 + y^2$ 的平方根。

int ilogb(x)

提取x的指数部分。

double ldexp(x, n)

返回 $x * 2^n$ (与frexp相反)。

double lgamma(x)

计算 $\gamma(x)$ 绝对值的自然对数。

double log(x)

计算x的自然对数。

double logb(x)

计算x的符号指数。

double log1p(x)

计算 $(x+1)$ 的自然对数，x大于等于-1。

double log2(x)

计算x以2为底的对数，x大于等于0。

`double log10(x)`

计算 x 以10为底的对数， x 大于等于0。

`long int lrint(x)`

将 x 圆整到最接近的`long int`。

`long long int llrint(x)`

将 x 圆整到最接近的`long long int`。

`long long int llround(x)`

将 x 四舍五入到最接近的`long long int`。

`long int lround(x)`

将 x 四舍五入到最接近的`long int`。

`double modf(x, ipart)`

分别获得 x 的分数部分和整数部分，分数部分作为函数的返回值，整数部分保存在`double`类型指针`ipart`所指向的位置。

`double nan(s)`

返回一个无效数。如果可能的话，根据字符串 s 的内容生成该无效数。

`double nearbyint(x)`

返回最接近 x 的整型数，返回值为浮点数。

`double nextafter(x, y)`

在 y 的方向上返回 x 的下一个可表示的数。

`double nexttoward(x, y)`

在 y 的方向上返回 x 的下一个可表示的数。与`nextafter`类似，但是第二个参数的类型是`long double`。

`double pow(x, y)`

计算 x^y 。如果 x 小于0，那么 y 必须是一个整数。如果 x 等于0，那么 y 必须大于0。

`double remainder(x, y)`

计算 x/y 所得的余数。

double remquo(x, y, quo)

计算 x/y 所得的余数。将商保存在整型指针 quo 所指向的位置。

double rint(x)

以浮点数的格式返回最接近 x 的整型数。如果返回值不等于 x 的话，有可能引发浮点异常。

double round(x)

以浮点数的格式四舍五入 x 到最接近的整数。

double scalbln(x, n)

计算 $x \times \text{FLT_RADIX}^n$ ，其中 n 为long int类型。

Double scalbn(x, n)

计算 $x \times \text{FLT_RADIX}^n$ 。

double sin(x)

计算 x 的正弦值。

double sinh(x)

计算 x 的双曲正弦值。

double sqrt(x)

计算 x 的平方根， x 应该大于等于0

double tan(x)

计算 x 的正切值。

double tanh(x)

计算 x 的双曲正切值。

double tgamma(x)

计算 $\gamma(x)$ 。

double trunc(x)

提取 x 的整数部分，以浮点数的格式返回。

复数算术

头文件<complex.h>中包含了很多可以用于复数运算的函数原型声明以及符号类型定义。下面列出了该文件中部分符号定义以及可供调用的函数。

定义	含义
complex	复数类型, _Complex的新名字
_Complex_I	用于指定复数虚部的宏(例如, $4 + 6.2 * _Complex_I$ 代表 $4 + 6.2i$)
imaginary	虚数类型, _Imaginary的新名字
_Imaginary_I	用于指定虚数虚部的宏

在下面的例子中, y 和 z 都是double complex类型, x 是double类型, n 是int类型。

```
double complex cabs(z)2
```

计算 z 的绝对值。

```
double complex cacos(z)
```

计算 z 的反余弦值。

```
double complex cacosh(z)
```

计算 z 的双曲反余弦值。

```
double carg(z)
```

计算复数 z 的角度。

```
double complex casin(z)
```

计算 z 的正弦值。

```
double complex casinh(z)
```

计算 z 的双曲反正弦值。

²虚数库包含有 float complex、double complex 和 long double complex 三种版本的函数。这里以 double complex 类型为例介绍。complex 版本的函数名以 f 结尾(如 cacosf), long double complex 版本的函数名以 l 结尾(如 cacosl)

double complex catan(z)

计算 z 的反正切值。

double complex catanh(z)

计算 z 的双曲反正切值。

double complex ccos(z)

计算 z 的余弦值。

double complex ccosh(z)

计算 z 的双曲余弦值。

double complex cexp(z)

计算 z 的自然指数。

double cimag(z)

计算 z 的虚部。

double complex clog(z)

计算 z 的自然对数。

double complex conj(z)

返回 z 的共轭数（反转其虚部）。

double complex cpow(y, z)

计算 y^z

double complex cproj(z)

计算 z 在黎曼球面上的投影。

double creal(z)

计算 z 的实部。

double complex csin(z)

计算 z 的复数正弦值。

double complex csinh(z)

计算 z 的复数双曲正弦值。

double complex csqrt(z)

计算 z 的复数平方根。

double complex ctan(*z*)

计算*z*的复数正切值。

double complex ctanh(*z*)

计算*z*的复数双曲正切值。

通用函数

C语言标准库中还有一些很有用的函数，无法归类于上述类别。这些函数在标准头文件<stdlib.h>中声明。

int abs(*n*)

返回int类型参数*n*的绝对值。

void exit(*n*)

终止程序的执行，关闭所有打开的文件，并将参数*n*作为程序的返回值。<stdlib.h>中定义了两个常量：EXIT_SUCCESS和EXIT_FAILURE，分别代表程序运行成功和运行失败。库函数中还有两个函数abort和atexit，其作用与exit函数相关。

char* getenv(*s*)

返回参数*s*对应的环境变量，如果没有找到的话返回空指针。函数的运行方式与具体的平台相关。例如，在Unix平台上，下面的代码：

```
char *homedir;  
...  
homedir = getenv ("HOME");
```

可以获取用户的HOME环境变量，并将结果保存在字符串指针homedir中。

long int labs(*l*)

返回long int类型参数*l*的绝对值。

long long int llabs(*ll*)

返回long long int类型参数*ll*的绝对值。

void qsort(*arr*, *n*, *size*, *comp_fn*)

使用快速排序法对于数组*arr*进行排序。*arr*是void*类型的指针，数组中包含*n*个元素，每个元素的大小为*size*个字节。*n*和*size*都是size_t类型的整型数。第四个参数是一个指向“返回int类型，接收两个void*类型参数”的函数指针。如果qsort函数需要比较数组中的两个元素，它将调用这个函数。qsort函数要求该函数比较传递给它的两个元素，如果

第一个大于第二个，那么返回值大于0；如果等于返回值等于0，如果小于返回值小于0。

下面是使用qsort函数对包含有1000个元素的整型数组进行排序的示例代码：

```
#include <stdlib.h>

...

int main (void)
{
    int data[1000], comp_ints (void *, void *);
    ...
    qsort (data, 1000, sizeof(int), comp_ints);
    ...
}

int comp_ints (void *p1, void *p2)
{
    int i1 = * (int *) p1;
    int i2 = * (int *) p2;
    return i1 - i2;
}
```

与qsort类似还有一个名为bsearch的函数，该函数的接口形式与qsort相近，对一个排序数组执行二分查找。

```
int rand(void)
```

返回一个位于区间[0, RAND_MAX]之间的随机数，其中符号RAND_MAX在标准头文件<stdlib.h>中定义，ANSI C规定该符号的最小值为32767。参见srand函数。

```
void srand(seed)
```

为随机数发生器设置种子。seed的类型为unsigned int。

```
int system(s)
```

执行字符数组s中包含的命令，并返回具体操作系统定义的值。如果s是空指针，那么系统返回一个非零值代表当前环境拥有一个命令处理器，可以用于执行各种命令。

作为Unix操作系统下的一个例子，下面的代码：

```
system ("mkdir /usr/tmp/data");
```

将在系统中创建一个路径为/usr/tmp/data的目录（假定程序拥有相应的权限）。

Compiling Programs with gcc

附录 C 使用 gcc 编译程序

本附录列出了gcc编译器最常用的部分选项。如果需要了解gcc的全部命令行选项，在Unix操作系统下，可以输入命令 `man gcc`。读者也可以参考网址 <http://gcc.gnu.org/onlinedocs> 上的在线文档。

本附录总结了3.3版本的gcc中可用的命令行选项，但是并不包括其他供应商，例如苹果公司（Apple Computer, Inc），提供的扩展特征。

命令的一般格式

gcc命令的一般格式如下：

```
gcc [options] file [file ...]
```

其中，用方括号括起来的项目是可选的。

一般来说，命令行中给出的每一个文件都要经过预处理、编译、汇编和连接四个步骤。我们可以使用命令行选项修改这个过程。

gcc根据每一个输入文件的扩展名来决定如何对该文件进行处理。我们可以使用命令行选项 `-x` 来改变这一点（`-x` 的具体使用方法参见文档）。表C.1中列出了一些最常见的文件扩展名。

表C.1 常见的源文件扩展名

扩展名	含义
.c	C语言源文件
.cc, .cpp	C++语言源文件
.h	头文件
.m	Objective-C语言源文件
.pl	Perl语言源文件
.o	目标文件（经过编译）

命令行选项

表C.2列出了编译C语言源程序经常使用的一组选项。

表C.2 gcc的常用选项

选项	含义	举例
--help	列出命令行帮助的总结	gcc --help
-c	编译但是不连接文件，将生成的目标代码按照.o的扩展名存储到文件中。	gcc -c enumerator.c
-dumpversion	显示gcc的版本	gcc -dumpversion
-g	在生成的代码中包含调试信息。这些调试信息通常供gdb使用，如果要支持多种调试器，可以使用-ggdb选项。	gcc -g testprog.c -o testprog
-D id	在前面这种用法中，gcc将定义名为id	gcc -D DEBUG=3
-D id=value	的符号，该符号的值为1。在第二种用法中，符号id的值等于value。	test.c
-E	仅仅对源文件进行预处理，并把处理结果写到标准输出中。对于检查预处理器的处理结果很有用处。	gcc -E enumerator.c
-I dir	将dir对应的目录加入到头文件的搜索路径中，该路径将在标准目录之前搜索。	gcc -I /users/ steve/include x.c
-llibrary	指定需要搜索的库文件。该选项应该在调用了该库文件包含的函数的那个源文件之后使用。连接器将在标准库文件路径（参见-L选项）中查找名为liblibrary.a的文件。	gcc mathfuncs.c -lm
-L dir	将目录dir加入到库文件搜索路径中。该目录将在标准路径之前被搜索。	gcc -L /users/steve /lib x.c
-o execfile	指定最终生成的可执行文件名为execfile。	gcc dbtest.c -o dbtest

表C.2 续

选项	含义	举例
-Olevel	根据level中指定的级别对代码进行优化，这里的级别可以是1、2或者3。如果使用了-O而没有指定级别，那么默认为1。优化级别越高，编译所用的时间就有可能越长，并且使用调试器调试该程序的能力也就越差。	gcc -O3 m1.c m2.c -o matchfuncs
-std=standard ¹	指定编译的时候遵循的标准。使用c99的时候代表ANSI C99规范而不使用GNU的扩展。	gcc -std=c99 mod1.c mod2.c
-Wwarning	根据warning中指定的选项打开相应的警告消息。有用的选项包括：1, all, 打开所有可能的警告消息，这个选项对于绝大多数程序都是有用的。2, error, 把所有的警告消息都当作错误来看待，这样程序员就必须修正所有的警告。	gcc -Werror mod1.c mod2.c

¹ 目前的默认值是 gnu89(对应于 ANSI C90) 加上 GNU 的扩展特征，一旦实现了 C99 的全部特性，默认值将改为 gnu99 (对应于 ANSI C99) 加上 GNU 的扩展特征。

Common Programming Mistakes

附录 D 常见编程错误

本附录列出了使用C语言编程最常见的一些错误。了解这些错误可以帮助读者在编程中避免它们。

1. 错误放置的分号。

举例：

```
if ( j == 100 );  
j = 0;
```

在上面的例子中，因为if语句后面错误的放置了一个分号，所以j的值总是被设置为0。读者需要注意，这个多余的分号在语法上来说是完全正确的（编译器将把它当作一个空语句，因此不会报告任何错误）。这种类型的错误在while语句和for语句中也经常出现。

2. 混淆=操作符和==操作符。

这个错误经常出现在if语句、while语句和do语句中。

举例：

```
if ( a = 2 )  
printf ("Your turn.\n");
```

上面的语句是完全合法的，它将变量a的值设置为2，然后再执行下面的printf语句。因为if语句中条件表达式的值不为0（实际上，它的值等于2），因此计算机总是执行下面的printf语句。

3. 忽略原型声明。

举例：

```
result = squareRoot (2);
```


如果squareRoot函数的定义出现在后面的程序或者其他源文件中，而且本语句前面也没有该函数的原型声明，那么C语言编译器将假定该函数的返回值为int类型。另外编译器还将传递给该函数的float类型参数转换为double，_Bool、char和short类型参数转换为int类型。对于其他类型的参数，编译器将不作转换。

记住，无论何时，在程序中加入函数的原型声明总是安全的（可以通过包含适当的头文件或者原型声明语句），即使该函数的定义在前面已经出现过也是一样。

4. 混淆各类操作符的优先级。

举例：

```
while ( c = getchar () != EOF )  
...  
  
if ( x & 0xF == y )  
...
```

在上面的第一个例子中，函数getchar的返回值将首先和EOF进行比较，然后计算机再将比较结果赋给变量c。这是因为不等条件操作符的优先级高于赋值操作符，这样一来，变量c中的值不是true就是false：如果getchar的返回值等于EOF，c的值就是true，否则的话就是false。在第二个例子中，计算机首先判断0xF与y是否相等，因为相等条件操作符的优先级高于任何位运算操作符，然后计算机再把条件判断的结果（1或者0）和x进行按位与操作。

5. 混淆字符串常量和字符常量。

下面的语句：

```
text = 'a';
```

把单个字符a赋值给变量text，而下面的语句：

```
text = "a";
```

将指向字符串"a"的指针赋值给变量text。因此，第一个语句中的变量text应该被声明为char类型，而第二个语句中的变量text应该被声明为char*类型。

6. 使用错误的数组下标。

举例：

```
int a[100], i, sum = 0;  
...  
for ( i = 1; i <= 100; ++i )  
    sum += a[i];
```

数组合法的下标范围是从0到该数组的长度减去1。因此上面的语句是不正确的，因为它把100而不是99作为该数组的最大下标。犯这类错误的程序员也常常把下标1当作是数组的第一个元素。记住，数组的第一个元素下标为0。

7. 在声明字符数组时忘记为结尾空字符保留空间。

在声明字符数组的时候，一定要记住为结尾的空字符预留出空间。例如，如果要存储结尾空字符的话，字符串"Hello"应该占用6个字符的空间。

8. 当引用结构成员的时候混淆操作符. 和操作符->。

记住，在引用结构变量成员的时候，应该使用操作符.，如果引用结构指针的成员时，应该使用操作符->。因此，如果标识符x是一个结构变量，我们可以使用x.m引用该结构名为m的成员变量；如果x是一个指向结构变量的指针，我们应该使用x->m引用该结构的成员变量m。

9. 在调用scanf函数的时候忘记了取地址操作符&。

举例：

```
int number;
...
scanf ("%i", number);
```

记住，scanf函数格式化字符串中的每一个格式化输入符号都对应一个指针类型的变量。

10. 在初始化指针变量之前使用它。

举例：

```
char *char_pointer;
*char_pointer = 'X';
```

只有设置了指针变量指向的位置之后（无论是某个变量还是新分配的内存），我们才能给它指向的位置赋值。在上例中，因为指针变量char_pointer没有指向任何位置，因此下面的语句是无意义的（译者注：甚至可能导致程序崩溃）。

11. 在switch语句的case子句后面忘记了break语句。

记住，如果在某个case子句的结束处没有对应的break语句，那么计算机将接着执行下一个case语句中的内容。

12. 在宏定义的结尾插入额外的分号。

当读者习惯了在所有的语句后面加上分号之后，很容易犯这个错误。读者需要记住，在宏定义语句中，宏名后面的所有字符都将被预处理器当作是宏定义的一部分。因此，对于下面的宏定义语句：

```
#define END_OF_DATA 999;
```

如果我们在下面的语句中使用该宏的话，编译器将报告语法错误：

```
if ( value == END_OF_DATA )
...
```


因为编译器看到的实际语句如下所示：

```
if ( value == 999; )  
...
```

13. 在宏定义语句中忘记了用小括号将参数括起来。

举例：

```
#define reciprocal(x) 1 / x  
...  
w = reciprocal (a + b);
```

上面的语句将被展开为如下的形式，这是不正确的。

```
w = 1 / a + b;
```

14. 定义宏的时候，在宏的名字和参数列表之间加入了一个空格。

举例：

```
#define MIN (a,b) ( (a) < (b) ) ? (a) : (b) )
```

因为预处理器把宏名后面第一个空白字符后面的所有文本都当作宏定义的一部分，因此上面语句的最终结果是不正确的。按照上面的定义，下面的语句：

```
minVal = MIN (val1, val2);
```

将被展开为如下的形式：

```
minVal = (a,b) ( (a) < (b) ) ? (a) : (b) )(val1,val2);
```

很明显，这里的展开结果不是我们所需要的。

15. 在调用宏的时候使用了有副作用的表达式。

举例：

```
#define SQUARE(x) (x) * (x)  
...  
w = SQUARE (++v);
```

调用上面的宏将会使得v的值增加两遍，因为该语句被预处理器展开为如下的形式：

```
w = (++v) * (++v);
```


APPENDIX E Resources

附录 E C 语言的其他资源

读者可以在本附录中找到C语言的更多知识。某些信息位于Web站点，另外一些则可以在其他书籍中找到。如果读者无法找到所需的信息，可以给作者发送Email。本人的Email地址是：steve@kochan-wood.com。

练习题答案和勘误表

读者可以在网站www.kochan-wood.com上找到本书练习题的答案和勘误表。网站上还列出了本附录的最新版本。

C 语言

C语言已经出现了25年了，因此关于该语言的资料非常丰富。下面列出的只是冰山一角。

书籍

Kernighan, Brain W., Dennis M. Ritchie. *The C Programming Language, 2nd* (C语言第二版)。Prentice Hall出版公司，1988年。

这本书被称作C语言的圣经。它也是第一本关于C语言的书籍。书中的作者之一Dennis Ritchie是C语言的创始人。

Harbison, Samuel P. III, and Guy L. Steele Jr. *C: A Reference Manual, 5th Ed.* (C语言参考手册第5版) Prentice Hall出版公司，2002年。

C语言极出色的参考手册。

Plauger, P.J. *The Standard C Library* (C语言标准库)，Prentice Hall出版公司，1992年。

这本书涵盖了C语言标准库的各个方面。但是该书的出版日期是在ANSI C99标准发布之前，因此书中不包括复数数学库。

网站

www.kochan-wood.com

在这个网站上，读者可以找到一本C语言的新书“C语言话题”的在线版本。这本书作为本书的姊妹篇，是本人与Patrick Wook先生合著的。

www.ansi.org

这是ANSI的官方站点。在该网站上可以买到ANSI C的规范。在网站的搜索框内输入9899:1999就可以找到ANSI C99规范。

www.opengroup.org/onlinepubs/007904975/idx/index.html

这个URL是C语言库函数的参考大全。它也包括那些非ANSI标准的C语言函数。

新闻组

comp.lang.c

这是一个专注于C语言的新闻组。读者既可以在这里寻求帮助，也可以帮助他人（当然是在获得了足够的经验之后）。即使只是看看这里的文章，也是很有帮助的。通过<http://groups.google.com>可以很方便的访问该新闻组。

C 语言编译器和集成开发环境

下面列出了一组Web站点，在这些站点上读者可以下载（购买）C语言的编译器以及开发环境，并获得相应的文档。

gcc

<http://gcc.gnu.org>

gcc是自由软件基金会（Free Software Foundation, FSF）开发的C语言编译器。苹果公司在他们的Mac OS X操作系统上使用了该编译器。读者可以从网站上免费下载该编译器。

MinGW

www.mingw.org

如果读者是在Windows环境下学习编写C语言程序，可以尝试从该站点下载gcc编译器。读者还可以下载MSYS，MSYS是一个非常方便的shell环境。

CygWin

www.cygwin.com

cygwin提供了一个Windows操作系统下的Linux模拟环境。读者可以免费获得这个开发环境。

Visual Studio

<http://msdn.microsoft.com/vstudio>

Visual Studio是微软公司的集成开发环境。使用该工具，我们可以在同一个环境下使用多种语言进行开发。

Code Warrior

www.metrowerks.com/mv/products/default.htm

Code Warrior是Metrowerks公司提供的专业集成开发环境，该开发环境可以在多种操作系统如Linux、Mac OS X、Solaris和Windows下运行。

Kylix

www.borland/kylix/

kylix是由Borland公司开发的Linux操作系统中的集成开发环境。

杂项

本节列出了有关面向对象编程的一些介绍文章和开发工具。

面向对象编程

Budd, Timothy. *The Introduction to Object-Oriented Programming, 3rd Ed.* (面向对象编程入门第三版), Addison-Wesley出版公司, 2001年。

这是关于面向对象编程的经典教材。

C++编程语言

Prata, Stephen. *C++ Primer Plus, 4th Ed.* (C++入门第四版), Sams出版公司, 2001年。

Stephen的入门系列教程很受读者欢迎。这本书介绍了C++语言。

Stroustrup, Bjarne. *The C++ Programming Language, 3rd Ed.* (C++编程语言第三版), Addison-Wesley出版公司, 2000年。

这是由C++语言的发明者亲自编写的经典教材。

C#编程语言

Petzold, Charles. *Programming in the Key of C#.* (C#语言编程精髓), 微软出版社, 2003年。

这本书被认为是C#入门的好教材。

Liberty, Jesse. *Programming C#, 3rd Ed.* (C#编程语言), O'Reilly出版公司, 2003年。

这本C#语言教材更适用于有经验的编程者阅读。

Objective-C 编程语言

Kochan, Stephen. *Programming in Objective-C* (Objective-C语言编程)。Sams出版社, 2004年。

该书由本人编写。本书是Objective-C语言的入门教材, 主要针对那些在C语言和面向对象编程方面有一定经验的读者。

苹果公司, *The Objective-C Programming Language* (Objective-C编程语言) Apple公司出版, 2004年。

对于C语言程序员来说, 本书是一本关于Objective-C语言极好的参考手册。读者可以通过下面的URL免费得到该书的pdf版本。

<http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf>.

开发工具

www.gun.org/manual/manual.html

在这里读者可以找到很多非常有用的Unix命令行工具的参考手册, 如cvs, gdb, make等等。

Index

索引

Symbols

- `\'` (single quote) escape character, 217
- `\"` (double quote) escape character, 217
- `\?` (question mark) escape character, 217
- `\a` (audible alert) escape character, 216
- `\b` (backspace) escape character, 216
- `\n` (newline character), program syntax, 13
- `\nnn` (octal character value) escape character, 217
- `\t` (horizontal tab) escape character, 216
- `\xnn` (hexadecimal character value) escape character, 217
- `\\` (backslash) escape character, 217
- `^` (bitwise Exclusive-OR) operator, 284-285
- `_Bool` data type, 23, 26
- `_Complex` data type, 39
- `_Imaginary` data type, 39
- `{}` (braces), program syntax, 13
- `|` (bitwise Inclusive-OR) operator, 283-284
- `||` (logical OR) operator, compound relationship tests, 72-73
- `;` (semicolon)
 - `#define` statement, 306-307
 - program syntax, 13
- `!` (logical negation) operator, Boolean variables, 90
- `!=` (not equal to) operator, 46-50
- `'` (single quotation marks), char data type, 25-26
- `"` (double quotation marks), char data type, 25-26
- `#` operator, macros, 312
- `##` operator, macros, 313
- `#define` statement, 299-303, 461-463
 - arguments, 308-311
 - defined names, 300
 - definition types, 306-308
 - Introducing the `#define` Statement (Program 13.1), 300-302
 - macros, 308-311
 - converting character case, 311
 - defining number of arguments, 311
 - testing for lowercase characters, 310
 - More on Working with Defines (Program 13.2), 302-303
 - program extendability, 303-305
 - program portability, 305-306
 - semicolon (`;`), 306-307
- `#elif` statement, 318-319
- `#else` statement, conditional compilation, 316-318
- `#endif` statement, conditional compilation, 316-318
- `#error` statement, 463
- `#if` statement, 318-319, 463
- `#ifdef` statement, 316-318, 464
- `#ifndef` statement, 316-318, 464
- `#include` statement, 464-465
 - macro definition collections, 313-315
 - Using the `#include` Statement (Program 13.3), 314-315

#line statement, 465
#pragma statement, 465
#undef statement, 319, 465
% (modulus) arithmetic operator, 35-36
& (address) operator, 236, 260
& (bitwise AND) operator, 281-283
&& (logical AND) operator, compound relationship tests, 72-73
*** (indirection) operator**, 236
*** (multiplication sign) arithmetic operator**, 30-33
***/ (closing comments)**, 18
***= (times equal) operator**, 143
+ **(plus sign) arithmetic operator**, 30-33
++ (increment) operator, 49, 262, 268
- (minus sign)
 arithmetic operator, 30-33
 unary arithmetic operator, 33-34
-- (decrement) operator, 50, 262, 268, 445
/ (division sign) arithmetic operator, 30-33
/* (opening comments), 18
< (less than) operator, 46-50
<< (left shift) bitwise operator, 287
<= (less than or equal to) operator, 46-50
= (assignment) operator, 15
== (equal to) operator, 46-50
?\ (conditional) operator, ternary nature of, 91-92

A

A Simple Program for Use with gdb (Program 18.4), 396-398
abs() function, 490
absolute value of numbers, calculating, 129-131

absolute_value() function, 129-131
acos() function, 483
Adding Debug Statements with the Preprocessor (Program 18.1), 389-391
address (&) operator, 236, 260
adjacent strings, 218
algorithms
 binary search, 223-227
 function of, 5
 Sieve of Erastosthenes (prime numbers), 118
aligning triangular number output, 50-51
 field width specification, 51
 right justification, 51
alphabetic() function, 212-214
American National Standard Institute. See ANSI
ANSI (American National Standards Institute), 1
 C standardization efforts, 1
 C99 standard, 425
 Web site, 502
ar utility, programming functionality, 345
argc argument, 380
arguments, 16
 #define statement (macros), 308-311
 argc, 380
 argv, 380
 calling, 13
 command-line, 380
 File Copy Program Using Command-Line Arguments (Program 17.1), 382-383
 main() function, 380-381
 storing, 383
 data types, conversion of, 329-330
 format string, 17

- functions, 122-123
 - declaring*, 134-135
 - formal parameter name*, 124
 - values, checking*, 135-137
- pointer arguments, passing, 254-257
- sizeof operator, 385
- argv argument**, 380
- arithmetic operators**, 443-444
 - associative property, 30
 - binary, 30-33
 - division sign (/), 30-33
 - joining with assignment operators, 38-39
 - minus sign (-), 30-33
 - modulus (%), 35-36
 - More Examples with Arithmetic Operators (Program 4.3), 33-34
 - multiplication sign (*), 30-33
 - plus sign (+), 30-33
 - precedence, 30
 - rules example*, 34
 - type cast, precedence rules, 38
 - unary minus, 33-34
 - Using the Arithmetic Operators (Program 4.2), 30-31
- arithmetic right shift**, 288
- array of characters, Concatenating Character Arrays (Program 10.1)**, 196-198
- array operators**, 447-448
- array_sum() function**, 262-264
- arrays**
 - characters, 108-109
 - memory functions*, 472
 - const variable, 111-113
 - containment by structures, 187-189
 - declaring, 97-98
 - defining with unions, 376-377
 - dynamic memory allocation, 117
 - elements
 - as counters*, 100-103
 - initializing*, 106-108
 - sequencing through*, 96-100
 - Fibonacci numbers, generating, 103-104
 - function of, 95
 - functions, passing multidimensional arrays, 146-152
 - integer bases, conversion of, 109-111
 - multidimensional, 113-114, 433-434
 - initializing*, 114-115
 - passing to functions*, 146-152
 - multidimensional arrays, declaring, 114
 - passing to functions, 137-142
 - assignment operators*, 142-143
 - pointers to, 259-260, 449-450
 - to character string*, 266-267
 - decrement (--) operator*, 262, 268
 - increment (++) operator*, 262, 268
 - postdecrement operator*, 269-271
 - postincrement operator*, 269-271
 - predecrement operator*, 269-271
 - preincrement operator*, 269-271
 - program optimization*, 263-264
 - sequencing through pointer elements*, 261
 - prime numbers, generating, 104-106
 - programs
 - Converting a Positive Integer to Another Base (7.7)*, 110-111
 - Demonstrating an Array of Counters (7.2)*, 101-103
 - Finding the Minimum Value in an Array (8.9)*, 138-140
 - Generating Fibonacci Numbers (7.3)*, 103-104
 - Generating Fibonacci Numbers Using Variable-Length Arrays (7.8)*, 115-117
 - Illustrating Structures and Arrays (9.7)*, 188-189

Initializing Arrays (7.5), 107-108
Introducing Character Arrays (7.6),
108-109
Multidimensional Variable-Length Arrays
(8.13A), 150-152
Revising the Function to Find the
Minimum Value in an Array (8.10),
140-142
Revising the Program to Generate Prime
Numbers, Version 2 (7.4), 105-106
Sorting an Array of Integers into
Ascending Order (8.12), 144-146
Using Multidimensional Arrays and
Functions (8.13), 147-150
Working with an Array (7.1), 98-100
sequencing through elements with
pointers to arrays, 261
single-dimensional, 432-433
sorting, 143-146, 490-491
structures
 defining, 182
 initializing, 183
 Using the Dictionary Lookup Program
 (Program 10.9), 220-222
subscripts, 96
summing elements, 262-264
uses, 95
values, storing, 96
variable-length, 115-117, 433
variables, defining, 96-98
versus pointers, differentiating, 264-265
asin() function, 483
asinh() function, 483
Asking the User for Input (Program
5.4), 51-52
assemblers, 6
 programs, compiling, 9
assigning structure values via
compound literals, 181-182

assignment operators, 15, 142-143, 446
 joining with arithmetic operators, 38-39
AT&T Bell Laboratories, 1
atan() function, 484
atan2() function, 484
atanh() function, 484
atof() function, 480
atoi() function, 230, 480
atol() function, 480
audible alert (\a) escape character, 216
auto keyword, 124-126, 156
auto_static() function, 157-158
automatic local variables, 156
 functions, 124-126

B

backslash (\\) escape character, 217
backspace (\b) escape character, 216
backtrace command (gdb debugger),
405
base notations, int data types, 23-24
bases, integers, converting via arrays,
109-111
basic data types
 C language specifications, 430-432
 usual arithmetic conversion, 451-452
BASIC programming language, 10
beginning comments, character
syntax, 18
binary arithmetic operators, 30-33
binary files, opening, 475
binary search algorithm, 223-227
bit fields, 292-294
 declaring, 296
 defining, 294-295
 extracting values, 295
 units, 296

bits, 279

- bit fields, 292-294
- high-order, 279
- least significant, 279
- low-order, 279
- most significant, 279
- operators, 280
 - & (bitwise AND), 281-283
 - << (left shift), 287
 - ^ (bitwise Exclusive-OR), 284-285
 - | (bitwise Inclusive-OR), 283-284
- rotating values, 290, 292

bitwise operators, 445

- Illustrating Bitwise Operators (Program 12.2), 286-287

book resources

- The C Programming Language*, 501
- The C Reference Manual*, 501
- C# Programming in the Key of C#*, 503
- C++ Primer Plus, 4th Edition*, 503
- Introduction of Object-Oriented Programming, 3rd Edition*, 503
- Objective-C Programming Language*, 504
- Programming in Objective-C*, 503
- The Standard C Library*, 501

Boolean variables

- logical negation (!) operator, 90
- programs
 - Generating a Table of Prime Numbers (6.10)*, 87-90
 - Revising the Program to Generate a Table of Prime Numbers (6.10A)*, 90-91

braces ({}), program syntax, 13

break command (gdb debugger), 400, 409

break statement, 62, 84, 456

breakpoints in programs, debugging (gdb tool), 400

bugs in programs, 9

bytes, 279

C

C language

- ANSI standardization efforts, 1
- arithmetic operators, 443-444
- array pointers, 449-450
- arrays
 - multidimensional, 433-434
 - operators, 447-448
 - single-dimensional, 432-433
 - variable-length, 433
- assignment operators, 446
- AT&T Bell Laboratories, 1
- basic data type conversion, usual arithmetic conversion, 451-452
- as basis for Unix operating system, 1
- bitwise operators, 445
- book resources
 - The C Programming Language*, 501
 - The C Reference Manual*, 501
 - The Standard C Library*, 501
- character constants, 428
 - escape sequences, 428-429
 - wide character, 429
- character string constants, 429
 - concatenation, 429
 - multibyte, 429
- comma operators, 447
- comments, 426
- compound literals, 450-451
- conditional operators, 446
- constant expressions, 442-443
- data types
 - basic, 430-432
 - declarations, 430
 - derived, 432-438
 - enumerated, 438
 - modifiers, 439
 - typedef statement, 438-439
- decrement operators, 445

- digraph characters, 425
 - enumeration constants, 430
 - expressions, specifications, 439
 - filename extension, 7
 - floating-point constants, 427-428
 - fractions program, writing, 413-414
 - functions
 - calls, 455-456
 - definition of, 454-455
 - pointers, 456
 - identifiers, 425
 - keywords, 426
 - universal character names, 426
 - increment operators, 445
 - integer constants, 427
 - interpreters, 10
 - ISO standardization efforts, 1
 - logical operators, 444
 - operators, summary table, 440-442
 - origins, 1
 - pointers
 - declarations, 437-438
 - operators, 448
 - predefined identifiers, 466
 - preprocessor, 460
 - directives, 461-465
 - trigraph sequences, 460-461
 - relational operators, 444-445
 - scopes, 452
 - sizeof operators, 447
 - statements
 - break, 456
 - compound, 456
 - continue, 457
 - do, 457
 - for, 457
 - goto, 458
 - if, 458
 - null, 458
 - return, 459
 - switch, 459-460
 - while, 460
 - storage classes
 - functions, 452
 - variables, 452-454
 - structures
 - declarations, 434-436
 - operators, 448
 - pointers, 450
 - text editors, 7
 - type cast operators, 446
 - unions, declarations, 436-437
 - vendor marketing, 1
 - Web site resources, Kochan-Wood.com, 502
- C preprocessor**
- conditional compilation, 316
 - #else statement*, 316-318
 - #endif statement*, 316-318
 - #ifdef statement*, 316-318
 - #ifndef statement*, 316-318
 - statements, *#define*, 299-303
- The C Programming Language**, 501
- The C Reference Manual**, 501
- C# language**
- development history, 422
 - fractions program, writing, 422-424
- C# Programming in the Key of C#**, 503
- C++ language**
- development history, 419
 - fractions program, writing, 419-421
- C++ Primer Plus, 4th Edition**, 503
- cabs() function**, 488
- cacos() function**, 488
- cacosh() function**, 488

calculating

absolute value of numbers, 129-131

square roots, 131-133

triangular numbers

nested for loops (program looping), 53-54

output alignment (program looping), 50-51

program looping, 43-45

user input (program looping), 51-52

Calculating Factorials Recursively (Program 8.16), 159-161

Calculating the 200th Triangular Number (Program 5.2), 44-45

Calculating the Absolute Value (Program 8.7), 129-131

Calculating the Absolute Value of an Integer (Program 6.1), 66-67

Calculating the Average of a Set of Grades (Program 6.2), 67-69

Calculating the Eighth Triangular Number (Program 5.1), 43

Calculating the nth Triangular Number (Program 8.4), 123

Calculating the Square Root of a Number (Program 8.8), 132-133

call stacks (traces), 405

calling

functions, 121-122

C language specifications, 455-456

statements, 13

Calling Functions (Program 8.2), 121

calloc() function, 386, 481

dynamic memory allocation, 384-385

carq() function, 488

case sensitivity in programming, 11

casin() function, 488

casinh() function, 488

catan() function, 489

catanh() function, 489

Categorizing a Single Character

Entered at the Terminal (Program 6.7), 78-80

cc command (Unix), 7-9

ccos() function, 489

ccosh() function, 489

ceil() function, 484

cexp() function, 489

Changing Array Elements in Functions (Program 8.11), 142-143

char data type, 23

quote usage, 25-26

character arrays, 108-109, 196-198

character constants

C language specifications, 428

escape sequences, 428-429

wide character, 429

in expressions, 227-230

character functions, 473

character I/O operations

getchar() function, 348

putchar() function, 348

character string constants

C language specifications

concatenation, 429

multibyte, 429

pointers, 267-268

character strings, 195

adjacent, 218

combining with array of structures, 219-222

comparing, 204-206

concatenating, 196

Concatenating Character Strings (Program 10.3), 202-203

continuation of, 218-219

Converting a String to its Integer

Equivalent (Program 10.11), 228-230

converting into integers, 228

- copying, 266-267, 271
- delimiting, 195
- displaying, 201-203
- escape characters, 216-218
- initializing, 201-203
- inputting, 206-208
- length, 199, 272
- null string, 213-215
- pointers to, 266-267
- Reading Strings with scanf (Program 10.5), 207-208
- testing for equality, 204-206
- Testing Strings for Equality (Program 10.4), 204-206
- universal character name, 218
- variable length, 198-200
- characters**
 - arrays of
 - comparing, 472
 - copying, 472
 - initializing, 107
 - memory functions, 472
 - searching, 472
 - files
 - reading (getc() function), 365
 - reading (putc() function), 365
 - formation of valid variables, 22
 - pointers to, 238-239
 - sign extensions, 329
 - single-character input, 208-212
 - whitespace, scanf() function, 355
- cimaq() function, 489**
- classes (OOP)**
 - instances, 412-413
 - methods, 412-413
- clear command (gdb debugger), 409**
- clearerr() function, 474**
- clearing end of file indicators, 474**
- cloq() function, 489**
- closing files, 474**
 - fclose() function, 365-367
- Code Warrior Web site, 503**
- comma operators, 378, 447**
- command lines, multiple source files, compiling, 334-336**
- command-line arguments, 380**
 - File Copy Program Using Command-Line Arguments (Program 17.1), 382-383
 - main() function, 380-381
 - storing, 383
- comments**
 - C language specifications, 426
 - character syntax
 - beginning, 18
 - terminating, 18
 - including in programs, 17-19
 - proper usage of, 18-19
 - Using Comments in a Program ((Program 3.6), 17-19
- communication between modules**
 - external variables, 336-338
 - include files, 341-342
 - prototype declarations, 336
 - static variables, 339-340
- compare_strings() function, 224**
- comparing**
 - arrays of characters, 472
 - character strings, 204-206
 - strings, 470-471
- compilers, 6-9**
 - GNU C, 12
 - Unix C, 12
- compiling programs, 7-12**
 - assemblers, 9
 - debugging phase, 9
 - errors
 - semantic, 7-9
 - syntactic, 7-9
 - multiple source files from command lines, 334-336

- Compiling the Debug Code (Program 18.2), 391-393**
- complex arithmetic functions, 488-490**
- compound literals, 450-451**
 - structural values, assigning, 181-182
- compound relationship tests**
 - if statement, 72-74
 - Determining if a Year Is a Leap Year (Program 6.5), 73-74*
 - logical AND operator, 72-73*
 - logical OR operator, 72-73*
- compound statements, C language specifications, 456**
- concat() function, 196, 201-203**
- concatenating**
 - character string constants, C language specifications, 429
 - character strings, 196
 - strings, 470
- Concatenating Character Arrays (Program 10.1), 196-198**
- Concatenating Character Strings (Program 10.3), 202-203**
- conditional (?\) operator, ternary nature of, 91-92**
- conditional compilation, 316**
 - #else statement, 316-318**
 - #endif statement, 316-318**
 - #ifdef statement, 316-318**
 - #ifndef statement, 316-318**
- conditional expression operator, #define statement in macros, 310**
- conditional operators, 446**
- conj() function, 489**
- console windows, 9**
- const modifier, C language specifications, 439**
- const variable (arrays), 111-113**
- constant expressions, 23, 442-443**
- constant FILE pointers**
 - stderr, 369-370
 - stdin, 369-370
 - stdout, 369-370
- constant keyword (pointers), 253**
- constants (C language specifications)**
 - character constants, 428-429
 - in expressions, 227-230*
 - character strings, 267-268, 429
 - enumeration constants, 430
 - floating-point constants, 427-428
 - integer constants, 427
 - wide character constants, 429
- continue statement, 62-63**
 - C language specifications, 457
- convert_number() function, 152**
- converting**
 - arguments, data types, 329-330
 - character strings into integers, 228-229
 - data types
 - float to int, 36-38*
 - in expressions, 327-329*
 - int to float, 36-38*
 - sign extension, 329*
 - strings to numbers, 479-481
- Converting a Positive Integer to Another Base**
 - Program 7.7, 110-111
 - Program 8.14, 153-156
- Converting a String to its Integer Equivalent (Program 10.11), 228-230**
- Converting Between Integers and Floats (Program 4.5), 36-38**
- copy_string() function, 266-267, 271**
- copying**
 - arrays of characters, 472
 - character strings, 266-267, 271
 - strings, 470-471

Copying Characters from Standard Input to Standard Output (Program 16.2), 361-362

Copying Files (Program 16.3), 366-367

copysign() function, 484

cos() function, 484

cosh() function, 484

counters, array elements, 100-103

Counting the Characters in a String (Program 10.2), 199-200

Counting Words (Program 10.7), 210-212

countWords() function, 210-215

cpow() function, 489

cproj() function, 489

creal() function, 489

csin() function, 489

csinh() function, 489

csqrt() function, 489

ctan() function, 490

ctanh() function, 490

cvs utility, programming functionality, 344

CygWin Web site, 502

D

data encapsulation (OOP), 417

data types

arguments, conversion of, 329-330

arrays, declaring, 97-98

C language specifications

basic, 430-432

declarations, 430

derived, 432-438

enumerated, 438

modifiers, 439

typedef statement, 438-439

char, 23

quote usage, 25-26

conversions

order in evaluating expressions, 327-329

sign extension, 329

Converting Between Integers and Floats (Program 4.5), 36-38

converting to float, 36-38

converting to int, 36-38

double, 23-25

enumerated

defining, 321-322, 324

Using Enumerated Data Types (14.1), 322-324

float, 23

decimal notation, 24

hexadecimal notation, 25

scientific notation, 24-25

int, 23

base notations, 23-24

machine-dependent ranges, 24

ranges, 24

storage sizes, 24

valid examples of, 23

naming (typedef statement), 325-327

specifiers

long, 28-30

long long, 28-30

short, 28-30

signed, 28-30

unsigned, 28-30

storage of differing types (unions), 375-378

Using the Basic Data Types (Program 4.1), 26-27

void, 128

value storage, 26

debugging

gdb tool, 395-398

backtrace command, 405

break command, 400, 409

- breakpoint deletion*, 404-405
- clear command*, 409
- function calls*, 405-406
- help command*, 406-408
- info break command*, 404-405
- info source command*, 409
- inserting breakpoints*, 400
- list command*, 399-400, 409
- miscellaneous features*, 408
- next command*, 409
- print command*, 409
- program execution controls*, 400
- quit command*, 409
- run command*, 400, 409
- session example*, 402-404
- set var command*, 398-399, 409
- single stepping*, 401-404
- stacktrace retrieval*, 405
- step command*, 401-404, 409
- viewing source files*, 399-400
- preprocessor**, 389-395
- programs**
 - A Simple Program for Use with gdb (18.4)*, 396-398
 - Adding Debug Statements with the Preprocessor (18.1)*, 389-391
 - Compiling the Debug Code (18.2)*, 391-393
 - Defining the DEBUG Macro (18.3)*, 393-395
 - Working with gdb (18.5)*, 401-402
- decimal notation in float data types**, 24
- declaring**
 - arguments in functions, 134-135
 - arrays
 - data types, 97-98
 - multidimensional, 114
 - bit fields, 296
 - data types, C language specifications, 430
 - return types in functions, 126, 134-135
 - structures, 166
 - unions, 375
 - variables, 15
 - in for loops*, 55-56
- decrement (--) operator**, 50, 262, 268, 445
- defined names**
 - NULL, 301
 - values, 300
- defined values, referencing (#define statement)**, 307-308
- defining**
 - arrays
 - of structures*, 182
 - with unions*, 376-377
 - bit fields, 294-295
 - data types, enumerated, 321-324
 - external variables, 337
 - functions, 119-122
 - local variables in functions, 124-126
 - pointer variables, 235-239
 - structures, 166-168
 - global structure definition*, 173
 - unions, members, 376
 - variables in arrays, 96-98
- Defining the DEBUG Macro (Program 18.3)**, 393-395
- deleting files via remove() function**, 371
- delimiting character strings**, 195
- Demonstrating an Array of Counters (Program 7.2)**, 101-103
- derived data types**
 - C language specifications, 432
 - multidimensional arrays*, 433-434
 - pointers*, 437-438
 - single-dimensional arrays*, 432-433

- structures*, 434-436
 - unions*, 436-437
 - variable-length arrays*, 433
- Determining if a Number Is Even or Odd (Program 6.3), 69-71**
- Determining if a Year Is a Leap Year (Program 6.5), 73-74**
- Determining Tomorrow's Date (Program 9.2), 169-171**
- digraph characters, C language specifications, 425**
- directives, 299**
 - preprocessors, 461
 - #define*, 461-463
 - #error*, 463
 - #if*, 463
 - #ifdef*, 464
 - #ifndef*, 464
 - #include*, 464-465
 - #line*, 465
 - #pragma*, 465
 - #undef*, 465
- dispatch tables, 274**
- display_converted_number() function, 152**
- Displaying Multiple Lines of Output (Program 3.3), 14**
- Displaying Multiple Variables (Program 3.5), 16-17**
- Displaying Variables (Program 3.4), 15-16**
- division sign (/), arithmetic operator, 30-33**
- do statement, 60-62**
 - C language specifications, 457
 - Implementing a Revised Program to Reverse the Digits of a Number (Program 5.9), 61-62
 - programming looping usage, 44
- double data type, 23-25**

- double quotation marks ("")**
 - char data types, 25-26
 - character strings, declaring, 195
- double quote (\") escape character, 217**
- doubly linked lists (pointers), 244-252**
- dynamic memory allocation, 383-384**
 - arrays, 117
 - calloc() function, 384-386
 - free() function, 387-388
 - functions, 481
 - linked lists, 387-388
 - malloc() function, 384-386
 - returning memory to system, 387-388
 - sizeof operator, 385-387

E

- editing programs with modular programming, 333-334**
- elements (array)**
 - counters, 100-103
 - initializing, 106-108
 - sequencing through, 96-100
 - summing, 262-264
 - values, storing, 96
- else if construct (if statement), 76-83**
 - Categorizing a Single Character Entered at the Terminal (Program 6.7), 78-80
 - Evaluating Simple Expressions (Program 6.8), 80-82
 - Implementing the Sign Function (Program 6.6), 77-78
 - Revising the Program to Evaluate Simple Expressions (Program 6.8A), 82-83
 - sign function, 76
- emacs text editor, 11**
- end-of-file conditions**
 - clearing, 474
 - I/O operations, 361-362
- enum keyword, 321-324**

enumerated data types

- C language specifications, 438
- defining, 321-324
- Using Enumerated Data Types (14.1), 322-324

enumeration constants, C language specifications, 430

EOF values, getchar() function, 362

equal to (=) operator, 46-50

equal_strings() function, 204

errors in programming, troubleshooting, 497-500

escape characters, 216-218

escape sequences, C language specifications, 428-429

evaluating order of operators, 442

Evaluating Simple Expressions (Program 6.8), 80-82

.exe filename extension, 9

exf() function, 484

exfc() function, 484

exit() function, 490

- programs, terminating, 370-371

exiting

- loops, 62
- programs, 490

exp() function, 484

expml() function, 484

expressions

- C language specifications, 439
 - constant expressions, 442-443
 - summary table, 440-442
- character constants in, 227-230
- constant, 23
- data types, conversion order, 327-329
- pointers, 239-240
- structures, 168-171

extensions (filenames), 9

external variables

- defining, 337
- modules, communicating between, 336-338
- versus static variables, 339-340

F

fabs() function, 484

factorial() function, 159

fclose() function, 474

- files, closing, 365-367

fdim() function, 484

feof() function, 474

- testing files for EOF conditions, 367-368

ferror() function, 474

fflush() function, 474

fgetc() function, 474

fgetpos() function, 474

fgets() function, 474

- files, reading to, 368

Fibonacci numbers, generating, 103-104

- Generating Fibonacci Numbers Using Variable-Length Arrays (Program 7.8), 115-117

field width specification, triangular number output, 51

fields, omitting in for loops, 55

File Copy Program Using Command-Line Arguments (Program 17.1), 382-383

file I/O operations

- fclose() function, 365-367
- feof() function, 367-368
- fgetc() function, 368
- fopen() function, 363-364
- fprint() function, 368
- fputs() function, 368
- getc() function, 365

putc() function, 365

remove() function, 371

rename() function, 371

FILE pointers

stderr, 369-370

stdin, 369-370

stdout, 369-370

filename extensions, 9

files

a.out (Unix executable), 9

characters

reading (getc() function), 365

reading (putc() function), 365

closing (fclose() function), 365-367, 474

copying (Program 16.3), 366-367

current position, returning, 474

deleting (remove() function), 371

EOF conditions, testing (fclose()
function), 367-368

executable (Unix), 9

header, 467

I/O operations

end-of-file conditions, 361-362

redirection of, 359-361

include, 341-342

modular programming organization,
333-334

naming, 7

opening, 475

opening (fopen() function), 363-364

printing (fprintf() function), 368

programming utilities

ar, 345

cvs, 344

grep, 345

make, 343-344

sed, 345

reading to (fgets() function), 368

renaming (rename() function), 371, 478

temporary files, creating, 478

writing to (fputs() function), 368

find_entry() function, 257

**Finding the Greatest Common
Divisor and Returning the Results
(Program 8.6), 127-128**

**Finding the Minimum Value in an
Array (Program 8.9), 138-140**

float data type, 23

 converting to, 36-38

 decimal notation, 24

 hexadecimal notation, 25

 scientific notation, 24-25

float.h header file, 316, 469

floating point numbers, 15

**floating-point constants, C language
specifications, 427-428**

floor() function, 484

fma() function, 485

fmax() function, 485

fmin() function, 485

fmod() function, 485

fopen() function, 363-364, 475

for statement

 array elements, sequencing through,
 98-100

 C language specifications, 457

 init expression, 45

 loop condition, 45

 loop expression, 45

 nested, 53-54

 program looping

relational operators, 46-50

triangular number calculation, 44-45

 programming looping usage, 44

 variants, 54

field omission, 55

multiple expressions, 55

variable declaration, 55-56

**formal parameter name, function
arguments, 124**

format string, 17

formatted I/O operations

- printf() function**, 348
 - conversion characters*, 350
 - flags*, 348
 - Illustrating the printf Formats (Program 16.1)*, 350-355
 - type modifiers*, 349
 - width and precision modifiers*, 349
- scanf() function**, 355
 - conversion characters*, 356-359
 - conversion modifiers*, 355

FORTRAN (FORmula TRANslation)
language, 6

- fpclassify() function**, 482
- fprintf() function**, 368, 475
- fputc() function**, 475
- fputs() function**, 368, 475

fractions program

- Working with Fractions in C (19.1), 413-414
- Working with Fractions in C# (19.4), 422-424
- Working with Fractions in C++ (19.3), 419-421
- Working with Fractions in Objective-C (19.2), 414-419
- writing in C, 413-414
- writing in C#, 422-424
- writing in C++, 419-421
- writing in Objective-C, 414-419

- fread() function**, 475
- free() function**, 387-388, 481
- freopen() function**, 475
- frexp() function**, 485
- fscanf() function**, 476
- fseek() function**, 476
- fsetpos() function**, 476
- ftell() function**, 476
- function calls (gdb debugger)**, 405-406

functions, 119

- abs()**, 490
- absolute value of numbers, calculating**, 129-131
- absolute_value()**, 129-131
- acos()**, 483
- alphabetic()**, 212, 214
- arguments**, 16, 122-123
 - checking values*, 135-137
 - declaring*, 134-135
 - formal parameter name*, 124
 - format string*, 17
 - pointer arguments, passing*, 254-257
- array_sum()**, 262-264
- arrays**
 - passing multidimensional arrays*, 146-152
 - passing to*, 137-142
- asin()**, 483
- asinh()**, 483
- atan()**, 484
- atan2()**, 484
- atanh()**, 484
- atof()**, 480
- atoi()**, 230, 480
- atol()**, 480
- auto_static()**, 157-158
- automatic local variables**, 124-126, 156
- C language specifications**, 452
 - calls*, 455-456
 - definition of*, 454-455
 - pointers*, 456
- cabs()**, 488
- cacos()**, 488
- cacosh()**, 488
- calling**, 121-122
- calloc()**, 386, 481
- carq()**, 488
- casin()**, 488
- casinh()**, 488

- catan(), 489
- catanh(), 489
- ccos(), 489
- ccosh(), 489
- ceil(), 484
- cexp(), 489
- character functions, 473
- cimaq(), 489
- clearerr(), 474
- clock(), 489
- compare_strings(), 224
- complex arithmetic functions, 488-490
- concat(), 196, 201-203
- conj(), 489
- convert_number(), 152
- copy_string(), 266-267, 271
- copysign(), 484
- cos(), 484
- cosh(), 484
- count_words(), 210, 214-215
- cpow(), 489
- cproj(), 489
- creal(), 489
- csin(), 489
- csinh(), 489
- csqrt(), 489
- ctan(), 490
- ctanh(), 490
- declaring return value type, 126
- defining, 119-122
 - global structure definition*, 173
- display_converted_number(), 152
- dynamic memory allocation
 - calloc()*, 384-385
 - functions*, 481
 - malloc()*, 384-385
- equal_strings(), 204
- exf(), 484
- exfc(), 484
- exit(), 490
- exp(), 484
- expml(), 484
- fabs(), 484
- factorial(), 159
- fclose(), 474
- fdim(), 484
- feof(), 474
- ferror(), 474
- fflush(), 474
- fgetc(), 474
- fgetpos(), 474
- fgets(), 474
- find_entry(), 257
- floor(), 484
- fma(), 485
- fmax(), 485
- fmin(), 485
- fmod(), 485
- fopen(), 475
- fpclassify(), 482
- fprintf(), 475
- fputc(), 475
- fputs(), 475
- fread(), 475
- free(), 387-388, 481
- freopen(), 475
- frexp(), 485
- fscanf(), 476
- fseek(), 476
- fsetpos(), 476
- ftell(), 476
- fwrite(), 477
- get_number_and_base(), 152
- getc(), 477
- getchar(), 208-210, 477
- getenv(), 490
- gets(), 209-212, 477
- global variables, 152-156
- hypot(), 485

- I/O, 473-478
 - fclose()*, 365-367
 - feof()*, 367-368
 - fgets()*, 368
 - fopen()*, 363-364
 - fprint()*, 368
 - fputs()*, 368
 - getc()*, 365
 - getchar()*, 348
 - printf()*, 348-355
 - putc()*, 365
 - putchar()*, 348
 - remove()*, 371
 - rename()*, 371
 - scanf()*, 355-359
- ilogb()*, 485
- in-memory format conversion, 478-479
- is_leap_year()*, 174
- isalnum()*, 473
- isalpha()*, 230, 473
- iscntrl()*, 473
- isdigit()*, 230, 473
- isfin()*, 482
- isgraph()*, 473
- isgreater()*, 482
- isgreaterequal()*, 482
- isinf()*, 482
- islessequal()*, 483
- islessgreater()*, 483
- islower()*, 230, 473
- isnan()*, 483
- isnormal()*, 483
- isprint()*, 473
- ispunct()*, 473
- isspace()*, 473
- isunordered()*, 483
- isupper()*, 230, 473
- isxdigit()*, 473
- labs()*, 490
- ldexp()*, 485
- lgamma()*, 485
- llabs()*, 490
- llrint()*, 486
- llround()*, 486
- local variables, defining, 124-126
- log()*, 485
- log1()*, 486
- log2()*, 485
- logb()*, 485
- loglb()*, 485
- lookup()*, 219-223
- lrint()*, 486
- main()*, 120
 - program syntax*, 13
- malloc()*, 386, 481
- math functions, 482-487
- memchr()*, 472
- memcmp()*, 472
- memcpy()*, 472
- memmove()*, 472
- memory functions, 472
- minimum()*, 138
- modf()*, 486
- modules, 333
- nan()*, 486
- nearbyint()*, 486
- Newton-Raphson Iteration Technique, 131-133
- nextafter()*, 486
- nexttoward()*, 486
- number_of_days()*, 171-174
- passing arrays to assignment operators, 142-143
- perror()*, 477
- pointers, returning, 257
- pointers to, 273-274
- pow()*, 486
- print_message()*, 120
- printf()*, 16, 477
 - program syntax*, 13

programs

Calculating Factorials Recursively
 (8.16), 159-161
Calculating the Absolute Value (8.7),
 129-131
Calculating the nth Triangular Number
 (8.4), 123
Calculating the Square Root of a
Number (8.8), 132-133
Calling Functions (8.2), 121
Changing Array Elements in Functions
 (8.11), 142-143
Converting a Positive Integer to Another
Base (8.14), 153-156
Finding the Greatest Common Divisor
and Returning the Results (8.6),
 127-128
Finding the Minimum Value in an Array
 (8.9), 138, 140
Illustrating Static and Automatic Variables
 (8.15), 157-158
More on Calling Functions (8.3), 122
Multidimensional Variable-Length Arrays
 (8.13A), 150-152
Revising the Function to Find the
Minimum Value in an Array (8.10),
 140-142
Revising the Program to Find the
Greatest Common Divisor (8.5),
 125-126
Sorting an Array of Integers into
Ascending Order (8.12), 144-146
Updating the Time by One Second
 (9.5), 178-180
Using Multidimensional Arrays and
Functions (8.13), 147-150
Writing a Function in C (8.1),
 120-121

prototype declaration, 124

putc(), 477

putchar(), 477

puts(), 478

qsort(), 274, 490-491

rand(), 491

read_line(), 213-215

realloc(), 481

recursive, 159-161

remainder(), 486

remove(), 478

rename(), 478

returning results from, 126-135

rewind(), 478

rint(), 487

rotate(), 290-292

round(), 487

scalar_multiply(), 147

scalbln(), 487

scalbn(), 487

scanf(), 206, 478

 input values, 51-52

shift functions, 288-290

shift(), 289

signbit(), 483

sin(), 487

sinh(), 487

sort(), 143-144, 146

sprintf(), 478-479

sqrt(), 487

square_root(), 133

srand(), 491

sscanf(), 478-479

static functions, 339

static variables, 156

strcat(), 230, 470

strchr(), 470

strcmp(), 230, 470

strcpy, 470-471

strcpy(), 230

string functions, 470-472

string-to-number conversions, 479-481

string_length(), 199, 272

string_to_integer(), 228-230

strlen(), 230, 471

strncat(), 471
 strncmp(), 471
 strncpy(), 471
 strrchr(), 471
 strstr(), 471-472
 strtod(), 480
 strtol(), 480
 strtoul(), 481
 structures, 171-174, 177
 system(), 491
 tan(), 487
 tanh(), 487
 tgamma(), 487
 time_update(), 178-180, 183
 tmpfile(), 478
 tolower(), 473
 toupper(), 473
 trunc(), 487
 ungetc(), 478
 utility functions, 490-491
fwrite() function, 477

G

gcc compiler, command-line options, 493-495
gcc Web site, 493, 502
gdb tool, debugging with, 395-398
 backtrace command, 405
 break command, 400, 409
 breakpoint deletion, 404-405
 clear command, 409
 function calls, 405-406
 help command, 406-408
 info break command, 404-405
 info source command, 409
 inserting breakpoints, 400
 list command, 399-400, 409
 miscellaneous features, 408
 next command, 409
 print command, 409

program execution controls, 400
 quit command, 409
 run command, 400, 409
 session example, 402-404
 set var command, 398-399, 409
 single stepping, 401-404
 stacktrace retrieval, 405
 step command, 401-404, 409
 viewing source files, 399-400
Generating a Table of Prime Numbers (Program 6.10), 87-90
Generating a Table of Triangular Numbers (Program 5.3), 47-50
Generating Fibonacci Numbers (Program 7.3), 103-104
Generating Fibonacci Numbers Using Variable-Length Arrays (Program 7.8), 115-117
get_number_and_base() function, 152
getc() function, 365, 477
getchar() function, 208-210, 348, 477
getenv() function, 490
gets() function, 209-212, 477
getter methods (OOP), 417
global variables
 default initial values, 155
 functions, 152-156
GNU C compiler, 12
GNU.org Web site, command-line tools, 504
Google Groups Web site, 502
goto statement
 C language specifications, 458
 execution of, 373
 labels, 373
 programming abuse, 374
greater than or equal to (>=) operator, 46-50
grep utility, programming functionality, 345

H

header files

- #include statement, 313-315
- float.h, 316, 469
- limits.h, 316, 468
- math.h, 482
- modular programming, use of, 341-342
- stdbool.h, 469
- stddef.h, 467
- stdint.h, 469-470
- stdlib.h, 490-491

help command (gdb debugger), 406-408

hexadecimal character value (\xnn) escape character, 217

hexadecimal notation

- float data types, 25
- int data type, 23-24

high-order bit, 279

higher-level languages, 6

- assembly languages, 6
- compilers, 6
- FORTRAN, 6
- interpreters, 10
- syntax standardization, 6

horizontal tab (\t) escape character, 216

hypot() function, 485

I

I/O functions, 473-478

I/O operations, 347

character functions

- getchar()*, 348
- putchar()*, 348

Copying Characters from Standard
Input to Standard Output (Program
16.2), 361-362

file functions

- fclose()*, 365-367
- feof()*, 367-368
- fgets()*, 368
- fopen()*, 363-364
- fprint()*, 368
- fputs()*, 368
- getc()*, 365
- putc()*, 365
- remove()*, 371
- rename()*, 371

files

- end-of-file conditions, 361-362
- redirecting to, 359-361

formatted functions

- printf()*, 348-355
- scanf()*, 355-359

function calls, 347

identifiers, C language specifications, 425

- keywords, 426
- predefined, 466
- universal character names, 426

IDEs (Integrated Development Environments), 10, 334

function of, 10

Linux, 10

Mac OS X

CodeWarrior, 10

Xcode, 10

Windows OS, Visual Studio, 10

if statement, 65

- C language specifications, 458
- compound relational tests, 72-74
- else if construct, 77-83
- general format, 65
- if-else construct, 69-72
- nested, 75-76

- programs
 - Calculating the Absolute Value of an Integer* (6.1), 66-67
 - Calculating the Average of a Set of Grades* (6.2), 67-69
 - Determining if a Number Is Even or Odd* (6.3), 69-71
 - Revising the Program to Determine if a Number Is Even or Odd* (6.4), 71-72
- if-else construct (if statement), 69-72**
 - Determining if a Number Is Even or Odd (Program 6.3), 69-71
 - Revising the Program to Determine if a Number Is Even or Odd (Program 6.4), 71-72
- Illustrating a Structure (Program 9.1), 166-168**
- Illustrating Arrays of Structures (Program 9.6), 183-184**
- Illustrating Bitwise Operators (Program 12.2), 286-287**
- Illustrating Pointers (Program 11.1), 236-237**
- Illustrating Static and Automatic Variables (Program 8.15), 157-158**
- Illustrating Structures and Arrays (Program 9.7), 188-189**
- Illustrating the Modulus Operator (Program 4.4), 35-36**
- ilogb() function, 485**
- Implementing a Revised Program to Reverse the Digits of a Number (Program 5.9), 61-62**
- Implementing a Rotate Function (Program 12.4), 290-292**
- Implementing a Shift Function (Program 12.3), 288-290**
- Implementing the Sign Function (Program 6.6), 77-78**
- in-memory format conversion functions, 478-479**
- include files, modular programming, 341-342**
- include statement, program syntax, 13**
- increment (++) operator, 49, 262, 268, 445**
- index number (arrays), 96**
- indirection, 235-236**
- infinite loops, 65**
- info break command (gdb debugger), 404-405**
- info source command (gdb debugger), 409**
- init expressions (for statement), 45**
- initializing**
 - arrays
 - characters, 107
 - elements, 106-108
 - multidimensional arrays, 114-115
 - of structures, 183
 - character strings, 201-203
 - structures, 180-181
 - union variables, 376
 - variables (static), 156-158
- Initializing Arrays (Program 7.5), 107-108**
- input**
 - programs, 9
 - single-character, 208-212
- input/output operations. See I/O operations**
- inputting character strings, 206-208**
- instances, classes (OOP), 412-413**
- instruction sets, 5**
- int data type, 23**
 - base notations, 23-24
 - converting to, 36-38
 - machine-dependent ranges, 24
 - ranges, 24
 - storage sizes, 24
 - valid examples of, 23

integers

- base conversion via arrays, 109-111
- constants, C language specifications, 427
- pointers, 239-240

Integrated Development Environments. *See* IDEs

International Standard Organization (ISO), 1

interpreters, 10

Introducing Character Arrays (Program 7.6), 108-109

Introducing the #define Statement (Program 13.1), 300-302

Introduction of Object-Oriented Programming, 3rd Edition, 503

isalnum() function, 473

isalpha() function, 230, 473

isctrl() function, 473

isdigit() function, 230, 473

isfin() function, 482

isgraph() function, 473

isgreater() function, 482

isgreaterequal() function, 482

isinf() function, 482

islessequal() function, 483

islessgreater() function, 483

islower() function, 230, 473

isnan() function, 483

isnormal() function, 483

ISO (International Standard Organization), 1

isprint() function, 473

ispunct() function, 473

isspace() function, 473

isunordered() function, 483

isupper() function, 230, 473

isxdigit() function, 473

J - K

JAVA programming language, interpretive nature of, 10

joining tokens in macros (## operator), 313

keywords

auto, 124-126, 156

C language specifications, 426

enum, 321-324

static, 156

void, 128

Kochan-Wood Web site, 502

book exercises and errata resources, 501

Kylix (Linux IDE), 10

L

labels in goto statements, 373

labs() function, 490

ldexp() function, 485

least significant bit, 279

left shift (<<) bitwise operator, 287

length of strings, 471

less than (<) operator, 46-50

less than or equal to (<=) operator, 46-50

lgamma() function, 485

limits.h header file, 316, 468

linked lists

dynamic memory allocation, 387-388

pointers, 244-252

linking programs, 9

Linux, Kylix IDE, 10

list command (gdb debugger), 399-400, 409

llabs() function, 490

llrint() function, 486

llround() function, 486
loading programs, 9
local variables
 automatic (functions), 124-126, 156
 defining (functions), 124-126
log() function, 485
log1() function, 486
log2() function, 485
logb() function, 485
logical AND (&&) operator,
 compound relationship tests, 72-73
logical negation (!) operator, Boolean
 variables, 90
logical operators, 444
logical OR (||) operator, compound
 relationship tests, 72-73
logical right shift, 288
loglb() function, 485
long long specifier (data types), 28-30
long specifier (data types), 28-30
lookup() function, 219-223
loop condition (for statement), 45
loop expressions (for statement), 45
loops
 Boolean variables
 Generating a Table of Prime Numbers
 (Program 6.10), 87-90
 Revising the Program to Generate a Table
 of Prime Numbers (Program 6.10A),
 90-91
 break statement, 62, 84
 continue statement, 62-63
 do statement, 60-62
 for statement, 44-45
 field omission, 55
 multiple expressions, 55
 nested, 53-54

 sequencing through array elements,
 98-100
 variable declaration, 55-56
 variants, 54-56
 if statement, 65
 Calculating the Absolute Value of an
 Integer (Program 6.1), 66-67
 Calculating the Average of a Set of
 Grades (Program 6.2), 67-69
 if-else construct, 69-72
 infinite, 65
 null statement, 374
 relational operators, 46-50
 switch statement, 84
 Revising the Program to Evaluate Simple
 Expressions, Version 2 (Program 6.9),
 85-86
 while statement, 56-60
low-order bit, 279
lrint() function, 486

M

Mac OS X
 IDEs
 CodeWarrior, 10
 Xcode, 10
 Objective-C language usage, 414
macros
 # operator, 312
 ## operator, 313
 #define statement, 308-311
 conditional expression operator, 310
 converting character case, 311
 defining number of arguments, 311
 testing for lowercase characters, 310
 #include statement, header files,
 313-315
 tokens, joining (## operator), 313

- main() function, 120**
 - command-line arguments, 380-381
 - program syntax, 13
- make utility, programming functionality, 343-344**
- malloc() function, 386, 481**
 - dynamic memory allocation, 384-385
- math functions, 482-487**
- math.h header file, 482**
- members (unions)**
 - arithmetic rules, 376
 - defining, 376
- memchr() function, 472**
- memcmp() function, 472**
- memcpy() function, 472**
- memmove() function, 472**
- memory, dynamic memory allocation, 383-384**
 - calloc() function, 386
 - free() function, 387-388
 - functions, 481
 - linked lists, 387-388
 - malloc() function, 386
 - returning memory to system, 387-388
 - sizeof operator, 385-386
- memory addresses (pointers), 274-276**
- memory functions, 472**
- message expressions (OOP), 412-413**
- methods, classes (OOP), 412-413**
 - getters, 417
 - setters, 417
- Metrowerks Web site, 503**
- MinGW Web site, 502**
- minimum() function, 138**
- minus sign (-), arithmetic operator, 30-33**
- modf() function, 486**
- modifiers, C language specifications, 439**
- Modifying the Dictionary Lookup Using Binary Search (Program 10.10), 224-227**
- modular programming**
 - file organization, 333-334
 - header files, use of, 341-342
 - IDE (Integrated Development Environment), 334
 - multiple source files, compiling from command line, 334-336
- modules, 333**
 - communicating between
 - include files, 341-342
 - static variables, 339-340
 - compiling, 334-336
 - external variables, communicating between, 336-338
 - prototype declarations, communicating between, 336
- modulus (%) arithmetic operator, 35-36**
- More Examples with Arithmetic Operators (Program 4.3), 33-34**
- More on Calling Functions (Program 8.3), 122**
- More on Working with Defines (Program 13.2), 302-303**
- More Pointer Basics (Program 11.2), 238**
- most significant bit, 279**
- multidimensional arrays, 113-114, 433-434**
 - declaring, 114
 - initializing, 114-115
 - Multidimensional Variable-Length Arrays (Program 8.13A), 150-152
 - passing to functions, 146-152
 - variable-length, 150-152
- Multidimensional Variable-Length Arrays (Program 8.13A), 150-152**
- multiple expressions, use in for loops, 55**

**multiple source files, compiling from
command line, 334-336**
**multiplication sign (*), arithmetic
operator, 30-33**

N

naming

data types (typedef statement), 325-327
files, 7
program constants, #define statement,
299-303
variables, 21
 reserved names, 22
 rules, 22

nan() function, 486

nearbyint() function, 486

negative numbers, 279-280

nested for loops, 53-54

nested if statements, 75-76

**newline character (\n), program
syntax, 13**

**newsgroups, C programming
resources, 502**

**Newton-Raphson Iteration Technique,
131-133**

next command (gdb debugger), 409

NeXT Software, 414

nextafter() function, 486

nexttoward() function, 486

not equal to (!=) operator, 46-50

null character ('\0'), 199

null statement

C language specifications, 458
example of, 374-375
loop control, 374
programming uses, 374

null strings, 213-215

number_of_days() function, 171-174

numbers

absolute values, calculating, 129-131
Fibonacci, generation of, 103-104
negative, 279-280
prime, generating with arrays, 104-106
square roots, calculating, 131-133

O

**object-oriented programming. See
OOP**

Objective-C language

as basis for Mac OS X, 414
development history, 414
fractions program, writing, 414-419

Objective-C Programming Language, 504

**octal character value (\nnn) escape
character, 217**

octal notation, int data type, 23

omitting fields in for loops, 55

**OOP (object-oriented programming),
411**

C# language, development history, 422
C++ language, development history, 419
car analogy, 411-412
classes

instances, 412-413

methods, 412-413

data encapsulation, 417

*Introduction of Object-Oriented
Programming, 3rd Edition, 503*

languages, 411

message expressions, 412-413

methods

getters, 417

setters, 417

Objective-C language, Working with
Fractions in Objective-C (Program
19.2), 414-419

overview, 411-412

versus procedural languages, 413

openf() function modes

append, 364
read, 364
update, 364
write, 364

opening files

binary files, 475
fopen() function, 363-364

operating systems

function of, 6
Unix
 development of, 6
 spin-offs, 6
Windows XP, 7

operators

#, macro definitions, 312
##, macro definitions, 313
assignment operators, 15, 142-143
 joining with arithmetic operators, 38-39
bit operators, 280
 & (bitwise AND), 281-283
 << (left shift), 287
 ^ (bitwise Exclusive-OR), 284-285
 | (bitwise Inclusive-OR), 283-284

C language specifications

arithmetic operators, 443-444
array operators, 447-448
array pointers, 449-450
assignment operators, 446
bitwise operators, 445
comma operators, 447
conditional operators, 446
decrement operators, 445
increment operators, 445
logical operators, 444
pointer operators, 448
relational operators, 444-445
sizeof operators, 447
structure operators, 448

structure pointers, 450

type cast operators, 446

comma, 378

conditional expression in macros, 310

evaluation order, 442

pointer operators, 236

precedence rules, 441-442

relational operators, 46-50

sizeof

 arguments, 385

 dynamic memory allocation, 385-387

summary table, 440-442

type cast, 69

output operations

end-of-file conditions, 361-362
redirecting to files, 359-361

P

passing arrays to functions, 137-142

assignment operators, 142-143
multidimensional arrays, 146-152

perror() function, 477

plus sign (+) arithmetic operator, 30-33

pointer operators, 448

Pointer Version of copyString() function (Program 11.13), 266-267

pointers, 235

& (address) operator, 236, 260

* (indirection) operator, 236

arrays, 259-260

 decrement (--) operator, 262, 268

 increment (++) operator, 262

 postdecrement operator, 269-271

 postincrement operator, 269-271

 predecrement operator, 269-271

 preincrement operator, 269-271

 program optimization, 263-264

 sequencing through array elements, 261

- character string constants, 266-268
- const keyword, 253
- declarations, 437-438
- defining, 235-239
- expressions, 239-240
- functions, 273-274
 - C language specifications, 456
 - passing pointer arguments, 254-257
 - returning pointers, 257
- indirection, 235
- integers, 240
- memory addresses, 274-276
- programs
 - Illustrating Pointers (11.1)*, 236-237
 - More Pointer Basics (11.2)*, 238
 - Pointer Version of copyString() function (11.13)*, 266-267
 - Returning a Pointer from a Function (11.10)*, 257-259
 - Revised Version of copyString() function (11.14)*, 271-272
 - Summing the Elements of an Array (11.12)*, 264-265
 - Traversing a Linked List (11.7)*, 250-252
 - Using Linked Lists (11.6)*, 246-250
 - Using Pointers and Functions (11.8)*, 254-255
 - Using Pointers in Expressions (11.3)*, 239-240
 - Using Pointers to Exchange Values (11.9)*, 255-257
 - Using Pointers to Find Length of a String (11.15)*, 272-273
 - Using Pointers to Structures (11.4)*, 241-243
 - Using Structures Containing Pointers (11.5)*, 243-244
 - Working with Pointers to Arrays (11.11)*, 262-263
- structures, 240-243
 - linked lists*, 244-252
 - structures containing pointers*, 243-244
- subtracting, 272
- versus arrays, differentiating, 264-265
- postdecrement operators**, 269-271
- postincrement operators**, 269-271
- pow() function**, 486
- precedence rules**
 - arithmetic operators, 30
 - operators, 441-442
 - rules example, 34
- precision modifiers**, 69
- predecrement operators**, 269-271
- predefined identifiers (directives)**, 466
- preincrement operators**, 269-271
- preprocessor**
 - Adding Debug Statements with the Preprocessor (Program 18.1), 389-391
 - C language specifications, 460
 - directives*, 461-465
 - trigraph sequences*, 460-461
 - debugging with, 389-395
- preprocessor statements**, 299
 - #define**, 299-303
 - arguments*, 308-311
 - definition types*, 306-308
 - macros*, 308-311
 - program extendability*, 303-305
 - program portability*, 305-306
 - #elif**, 318-319
 - #if**, 318-319
 - #include**, macro definition collections, 313-315
 - #undef**, 319
- prime numbers**
 - Generating a Table of Prime Numbers (Program 6.10), 87-90
 - generating via arrays, 104-106

- Revising the Program to Generate a Table of Prime Numbers (Program 6.10A), 90-91
- Sieve of Erastosthenes algorithm, 118
- print_message() function**, 120
- print command (gdb debugger)**, 409
- printf routine**
 - output, 14
 - variables
 - displaying multiple values*, 16-17
 - displaying values*, 15-16
- printf() function**, 16, 348, 477
 - conversion characters, 350
 - flags, 348
 - Illustrating the printf Formats (Program 16.1), 350-355
 - program syntax, 13
 - type modifiers, 349
 - width and precision modifiers, 349
- printing files via fprintf() function**, 368
- procedural languages versus OOP languages**, 413
- program constants, symbolic names**, 299-303
- program looping**
 - break statement, 62
 - Calculating the Eighth Triangular Number (Program 5.1), 43
 - continue statement, 62-63
 - do statement, 44, 60-62
 - for statement, 44
 - Generating a Table of Triangular Numbers (5.3)*, 47-50
 - relational operators, 46-50
 - scanf() function, Asking the User for Input (Program 5.4), 51-52
 - triangular number calculation, 43-45
 - nested for loops*, 53-54
 - output alignment*, 50-51
 - user input*, 51-52
 - while statement, 44, 56-60
- programming**
 - algorithms, 5
 - assembly languages, 6
 - case sensitivity, 11
 - common mistakes, troubleshooting, 497-500
 - higher-level languages, 6
 - instruction sets, 5
 - modular programming, 333-334
 - overview, 5
 - top-down, 137
- Programming in Objective-C**, 503
- programming utilities**
 - ar, 345
 - cvs, 344
 - grep, 345
 - make, 343-344
 - sed, 345
- programs**
 - #define statement
 - Introducing the #define Statement (13.1)*, 300-302
 - More on Working with Defines (13.2)*, 302-303
 - A Simple Program for Use with gdb (18.4), 396-398
 - Adding Debug Statements with the Preprocessor (18.1), 389-391
 - arguments, calling, 13
 - arrays
 - Converting a Positive Integer to Another Base (7.7)*, 110-111
 - Generating Fibonacci Numbers Using Variable -Length Arrays (7.8)*, 115-117
 - Introducing Character Arrays (7.6)*, 108-109
 - Multidimensional Variable-Length Arrays (8.13A)*, 150-152
 - Asking the User for Input (5.4), 51-52
 - assemblers, 6, 9

- bitwise operators, *Illustrating Bitwise Operators* (12.2), 286-287
- bugs, 9
- Calculating the Eighth Triangular Number (5.1), 43
- Calculating the 200th Triangular Number (5.2), 44-45
- Categorizing a Single Character Entered at the Terminal (6.7), 78-80
- comment statements, including, 17-19
- compiling, 7-12
 - debugging phase*, 9
 - semantic errors*, 7-9
 - syntactic errors*, 7-9
- Compiling the Debug Code (18.2), 391-393
- compound relational tests, *Determining if a Year Is a Leap Year* (Program 6.5), 73-74
- Concatenating Character Arrays (10.1), 196-198
- Concatenating Character Strings (10.3), 202-203
- Converting a String to its Integer Equivalent (10.11), 228-230
- Converting Between Integers and Floats (4.5), 36-38
- Copying Files (16.3), 366-367
- Counting the Characters in a String (Program 10.2), 199-200
- Counting Words (Program 10.7), 210-212
- Counting Words in a Piece of Text (10.8), 214-215
- debugging, 9
- Defining the DEBUG Macro (18.3), 393-395
- Demonstrating an Array of Counters (7.2), 101-103
- Determining if a Number Is Even or Odd (6.3), 69-71
- Displaying Multiple Lines of Output (3.3), 14
- Displaying Multiple Variables (3.5), 16-17
- Displaying Variables (3.4), 15-16
- editing (modular programming), 333-334
- Evaluating Simple Expressions (6.8), 80-82
- exiting, 490
- File Copy Program Using Command-Line Arguments (17.1), 382-383
- Finding the Greatest Common Divisor (5.7), 58-59
- fractions
 - writing in C*, 413-414
 - writing in C#*, 422-424
 - writing in C++*, 419-421
 - writing in Objective-C*, 414-419
- functions
 - Calculating Factorials Recursively* (8.16), 159-161
 - Calculating the Absolute Value* (8.7), 129-131
 - Calculating the nth Triangular Number* (8.4), 123
 - Calculating the Square Root of a Number* (8.8), 132-133
 - Calling Functions* (8.2), 121
 - Changing Array Elements in Functions* (8.11), 142-143
 - Converting a Positive Integer to Another Base* (8.14), 153-156
 - defining*, 119-122
 - Finding the Greatest Common Divisor and Returning the Results* (8.6), 127-128
 - Finding the Minimum Value in an Array* (8.9), 138-140
 - Illustrating Static and Automatic Variables* (8.15), 157-158
 - More on Calling Functions* (8.3), 122
 - Revising the Function to Find the Minimum Value in an Array* (8.10), 140-142

- Revising the Program to Find the Greatest Common Divisor (8.5), 125-126*
- Sorting an Array of Integers into Ascending Order (8.12), 144-146*
- Updating the Time by One Second (9.5), 178-180*
- Using Multidimensional Arrays and Functions (8.13), 147-150*
- Writing in Function in C (8.1), 120-121*
- Generating a Table of Prime Numbers (6.10), 87-90*
- Generating a Table of Triangular Numbers (5.3), 47-50*
- Generating Fibonacci Numbers (7.3), 103-104*
- I/O operations, Copying Characters from Standard Input to Standard Output (16.2), 361-362*
- Illustrating Pointers (11.1), 236-237*
- Illustrating the Modulus Operator (4.4), 35-36*
- Illustrating the printf Formats (16.1), 350-355*
- Implementing a Revised Program to Reverse the Digits of a Number (5.9), 61-62*
- Implementing the Sign Function (6.6), 77-78*
- Initializing Arrays (7.5), 107-108*
- input, 9*
- interpreting, 10*
- Introducing the while Statement (5.6), 56-58*
- linking, 9*
- loading, 9*
- Modifying the Dictionary Lookup Using Binary Search (10.10), 224-227*
- More Examples with Arithmetic Operators (4.3), 33-34*
- More Pointer Basics (11.2), 238*
- output, 9*
- Pointer Version of copyString() function (11.13), 266-267*
- portability of, 6*
- proper termination of, 383*
- Reading Lines of Input (10.6), 209-210*
- Reading Strings with scanf (10.5), 207-208*
- Returning a Pointer from a Function (11.10), 257-259*
- Reversing the Digits of a Number (5.8), 59-60*
- Revised Version of copyString() function (11.14), 271-272*
- Revising the Program to Determine if a Number Is Even or Odd (6.4), 71-72*
- Revising the Program to Evaluate Simple Expressions (6.8A), 82-83*
- Revising the Program to Evaluate Simple Expressions, Version 2 (6.9), 85-86*
- Revising the Program to Generate a Table of Prime Numbers (6.10A), 90-91*
- Revising the Program to Generate Prime Numbers, Version 2 (7.4), 105-106*
- rotating bit values, Implementing a Rotate Function (12.4), 290-292*
- running, 12*
- shift functions, Implementing a Shift Function (12.3), 288-290*
- statements, calling, 13*
- structures*
 - Determining Tomorrow's Date (9.2), 169-171*
 - Illustrating a Structure (9.1), 166-168*
 - Illustrating Arrays of Structures (9.6), 183-184*
 - Illustrating Structures and Arrays (9.7), 188-189*
 - Revising the Program to Determine Tomorrow's Date (9.3), 171-174*
 - Revising the Program to Determine Tomorrow's Date, Version 2 (9.4), 174-177*
- Summing the Elements of an Array (11.12), 264-265*

syntax

- braces ({}), 13**
- include statement, 13**
- main() function, 13**
- newline character (\n), 13**
- printf() function, 13**
- terminating (exit() function), 370-371**
- Testing Strings for Equality (10.4), 204-206**
- Traversing a Linked List (11.7), 250-252**
- undefined exit status, 383**
- Using Comments in a Program (3.6), 17-19**
- Using Enumerated Data Types (14.1), 322-324**
- Using Linked Lists (11.6), 246-250**
- Using Nested for Loops (5.5), 53-54**
- Using Pointers and Functions (11.8), 254-255**
- Using Pointers in Expressions (11.3), 239-240**
- Using Pointers to Exchange Values (11.9), 255-257**
- Using Pointers to Find Length of a String (11.15), 272-273**
- Using Pointers to Structures (11.4), 241-243**
- Using Structures Containing Pointers (11.5), 243-244**
- Using the #include Statement (13.3), 314-315**
- Using the Arithmetic Operators (4.2), 30-31**
- Using the Basic Data Types (4.1), 26-27**
- Using the Dictionary Lookup Program (10.9), 220-222**
- Working with an Array (7.1), 98, 100**
- Working with Fractions in C (19.1), 413-414**
- Working with Fractions in C# (19.4), 422-424**
- Working with Fractions in C++ (19.3), 419-421**

Working with Fractions in Objective-C (19.2), 414-419

Working with gdb (18.5), 401-402

Working with Pointers to Arrays (11.11), 262-263

Writing Your First C Program (3.1), 11

prototype declarations

functions, 124

modules, communicating between, 336

putc() function, 365, 477

putchar() function, 348, 477

puts() function, 478

Python programming language, 10

Q - R

qsort() function, 274, 490-491

qualifiers (variables)

register, 378-379

restrict, 379

volatile, 379

question mark (\?) escape character, 217

quit command (gdb debugger), 409

quotation marks, declaring character strings, 195

rand() function, 491

read_line() function, 213-215

reading files via fgets() function, 368

Reading Lines of Data (Program 10.6), 209-210

Reading Strings with scanf (Program 10.5), 207-208

real numbers, 15

realloc() function, 481

recursive functions, 159-161

redirecting I/O operations to files, 359-361

referencing defined values (#define statement), 307-308

- register qualifier (variables), 378-379**
 - relational operators, 46-50, 444-445**
 - remainder() function, 486**
 - remove() function, 478**
 - files, deleting, 371
 - rename() function, 478**
 - files, renaming, 371
 - renaming files, 478**
 - rename() function, 371
 - reserved names (variables), 22**
 - restrict modifier, C language specifications, 439**
 - restrict qualifier (variables), 379**
 - return statement (functions), 126**
 - C language specifications, 459
 - returning**
 - function results, 126-131
 - declaring return types, 134-135
 - pointers, 257
 - Returning a Pointer from a Function (Program 11.10), 257-259**
 - Revised Version of copyString() function (Program 11.14), 271-272**
 - Revising the Function to Find the Minimum Value in an Array (Program 8.10), 140-142**
 - Revising the Program to Determine if a Number Is Even or Odd (Program 6.4), 71-72**
 - Revising the Program to Determine Tomorrow's Date (9.3), 171-174**
 - Revising the Program to Determine Tomorrow's Date, Version 2 (9.4), 174-177**
 - Revising the Program to Evaluate Simple Expressions (Program 6.8A), 82-83**
 - Revising the Program to Evaluate Simple Expressions, Version 2 (Program 6.9), 85-86**
 - Revising the Program to Find the Greatest Common Divisor (Program 8.5), 125-126**
 - Revising the Program to Generate a Table of Prime Numbers (Program 6.10A), 90-91**
 - Revising the Program to Generate Prime Numbers, Version 2 (Program 7.4), 105-106**
 - rewind() function, 478**
 - right justification, triangular number output, 51**
 - right shift () bitwise operator, 287-288**
 - rint() function, 487**
 - Ritchie, Dennis, 1**
 - rotate() function, 290-292**
 - rotating bit values, 290-292**
 - round() function, 487**
 - routines. See also functions**
 - printf, 14
 - displaying multiple variable values, 16-17
 - displaying variable values, 15-16
 - output, 14
 - run command (gdb debugger), 400, 409**
-
- S**
-
- scalar_multiply() function, 147**
 - scalbln() function, 487**
 - scalbn() function, 487**
 - scanf() function, 206, 355, 478**
 - %s format characters, 206
 - conversion characters, 356-359
 - conversion modifiers, 355
 - input values, 51-52
 - skipping fields, 358
 - scientific notation, float data types, 24-25**
 - scopes, 452**

- search methods**
 - binary search algorithm, 223-227
 - lookup() function, 222-223
- searches**
 - arrays of characters, 472
 - strings, 470-471
- sed utility, programming functionality,** 345
- semantic errors in programs,**
 - compiling, 7-9
- semicolon (;)**
 - #define statement, 306-307
 - program syntax, 13
- set var command (gdb debugger),** 398-399, 409
- setters, methods (OOP),** 417
- shell programming language,** 10
- shift functions, 288-290**
 - programs, Implementing a Shift Function (12.3), 288-290
- shift() function, 289**
- short specifier (data types), 28-30**
- Sieve of Erastosthenes algorithm,**
 - prime number generation, 118
- sign bit, 279-280**
- sign extension, data type conversions,** 329
- sign function**
 - else if construct (if statement), 76-83
 - Categorizing a Single Character Entered at the Terminal (Program 6.7), 78-80*
 - Implementing the Sign Function (Program 6.6), 77-78*
- signbit() function, 483**
- sin() function, 487**
- single quotation marks (')**
 - char data types, 25-26
 - character strings, declaring, 195
- single quote (\') escape character, 217**
- single-character input, 208-213**
 - Counting Words (Program 10.7), 210-212
 - Reading Lines of Data (Program 10.6), 209-210
- single-dimensional arrays, 432-433**
- sinh() function, 487**
- sizeof operators, 447**
 - arguments, 385
 - dynamic memory allocation, 385-387
- sort() function, 143-144**
- sorting arrays, 143-146, 490-491**
- Sorting an Array of Integers into Ascending Order (Program 8.12),** 144-146
- source programs, 7**
- specifiers (data types)**
 - long, 28-30
 - long long, 28-30
 - short, 28-30
 - unsigned, 28-30
- sprintf() function, 478-479**
- sqrt() function, 487**
- square roots, calculating, 131-133**
- square_root() function, 133**
- srand() function, 491**
- sscanf() function, 478-479**
- The Standard C Library, 501**
- statements**
 - #define, 299-303
 - arguments, 308-311
 - definition types, 306-308
 - macros, 308-311
 - program extendability, 303-305
 - program portability, 305-306
 - #elif, 318-319
 - #if, 318-319
 - #include, macro definition collections, 313-315
 - #undef, 319

- break, 62, 84
- C language specifications, 456
 - break, 456
 - compound, 456
 - continue, 457
 - do, 457
 - for, 457
 - goto, 458
 - if, 458
 - null, 458
 - return, 459
 - switch, 459-460
 - while, 460
- calling, 13
- conditional compilation
 - #else, 316-318
 - #endif, 316-318
 - #ifdef, 316-318
 - #ifndef, 316-318
- continue, 62-63
- do, 60-62
- for, 44-45
 - nested, 53-54
- FORTTRAN statements, 6
- goto
 - execution of, 373
 - programming abuse, 374
- if, 65
 - Calculating the Absolute Value of an Integer (Program 6.1)*, 66-67
 - Calculating the Average of a Set of Grades (Program 6.2)*, 67-69
 - compound relational tests, 72-74
 - else if construct, 76-83
 - general format, 65
 - if-else construct, 69-72
 - nested, 75-76
- include, program syntax, 13
- null
 - example of, 374-375
 - programming uses, 374
- return (functions), 126
- switch, 84
 - Revising the Program to Evaluate Simple Expressions, Version 2 (Program 6.9)*, 85-86
- terminating, 14
- typedef, data types, naming, 325-327
- while, 56-60
- static functions, 339
- static keyword, 156
- static variables, 156
 - initializing, 156-158
 - versus external variables, 339-340
- stdbool.h header file, 469
- stddef.h header file, 467
- stderr FILE pointer, 369-370
- stdin FILE pointer, 369-370
- stdint.h header file, 469-470
- stdlib.h header file, 490-491
- stdout FILE pointer, 369-370
- step command (gdb debugger), 401-404, 409
- storage classes
 - functions, 452
 - variables, 452-454
- storing
 - different data types (unions), 375-378
 - time in programs, 177-180
 - values in arrays, 96
 - variables via dynamic memory allocation, 383-384
- strcat() function, 230, 470
- strchr() function, 470
- strcmp() function, 230, 470
- strcpy() function, 230, 470-471
- string functions, 470-472

- string_length() function**, 199, 272
- string_to_integer() function**, 228-230
- strings**
 - character strings, 195
 - adjacent, 218
 - combining with array of structures, 219-222
 - comparing, 204-206
 - concatenating, 196
 - continuation of, 218-219
 - converting into integers, 228-229
 - copying, 266-267, 271
 - delimiting, 195
 - displaying, 201-203
 - escape characters, 216-218
 - initializing, 201-203
 - inputting, 206-208
 - length, 199, 272
 - pointers to, 266-267
 - testing for equality, 204-206
 - variable-length, 198-200
 - comparing, 470-471
 - concatenating, 470
 - converting to numbers, 479-481
 - copying, 470-471
 - length, 471
 - null, 213-215
 - searches, 470-471
 - searching, 471
- strlen() function**, 230, 471
- strncat() function**, 471
- strncmp() function**, 471
- strncpy() function**, 471
- strrchr() function**, 471
- strstr() function**, 471-472
- strtod() function**, 480
- strtol() function**, 480
- strtoul() function**, 481
- structure operators**, 448
- structure pointers**, 241, 450
- structures**
 - arrays of, 182
 - combining with character strings, 219-222
 - defining, 182
 - initializing, 183
 - compound literal values, assigning, 181-182
 - containing arrays, 187-189
 - containing other structures, 185-187
 - containing pointers, 243-244
 - declarations, 166, 434-436
 - defining, 166-168
 - expressions, 168-171
 - function of, 165
 - functions, 171-174, 177
 - initializing, 180-181
 - pointers to, 240-243
 - linked lists, 244-252
 - programs
 - Determining Tomorrow's Date (9.2)*, 169-171
 - Illustrating a Structure (9.1)*, 166-168
 - Illustrating Arrays of Structures (9.6)*, 183-184
 - Illustrating Structures and Arrays (9.7)*, 188-189
 - Revising the Program to Determine Tomorrow's Date (9.3)*, 171-174
 - Revising the Program to Determine Tomorrow's Date, Version 2 (9.4)*, 174-177
 - time, updating, 177-180, 183
 - uses, 165
 - variants, 190-191
- subscripts (arrays)**, 96
- subtracting pointers**, 272
- summing array elements**, 262-264

Summing the Elements of an Array
(Program 11.12), 264-265

switch statement, 84

C language specifications, 459-460

programs, Revising the Program to
Evaluate Simple Expressions, Version 2
(6.9), 85-86

symbolic names, program constants,
299-303

syntactic errors, programs, compiling,
7-9

system include files

float.h file, 316

limits.h file, 316

system() function, 491

T

tan() function, 487

tanh() function, 487

temporary files, creating, 478

terminating

comments, character syntax, 18

programs

exit() function, 370-371

proper methods, 383

statements, 14

ternary operator, 91-92

testing

character strings for equality, 204-206

files for EOF conditions, 367-368

Testing Strings for Equality (Program
10.4), 204-206

text editors

C programming, 7

emacs, 11

vi, 7, 11

tgamma() function, 487

time, updating, 177-180

with array of structures, 183

time_update() function, 178-180, 183

times equal (*=) operator, 143

tmpfile() function, 478

tokens, joining (## operator), 313

tolower() function, 473

top-down programming, 137

toupper() function, 473

Traversing a Linked List (Program
11.7), 250-252

trees, pointers, 244-252

triangular numbers, calculating, 43-45

nested for loops (program looping),
53-54

output alignment (program looping),
50-51

user input (program looping), 51-52

trigraph sequences, preprocessors,
460-461

troubleshooting programming errors,
common mistakes, 497-500

trunc() function, 487

truth tables

& (bitwise AND) operator, 281

^ (bitwise Exclusive-OR) operator, 284

| (bitwise Inclusive-OR) operator, 283

twos complement notation, 279-280

type cast operators, 69, 446

precedence rules, 38

typedef statement

C language specifications, 438-439

data types, naming, 325-327

U

unary minus arithmetic operator,
33-34

undefined exit statuses, 383

ungetc() function, 478

unions

- arrays, defining, 376-377
- data types, storage, 375-378
- declarations, 436-437
- declaring, 375
- members
 - arithmetic rules, 376
 - defining, 376
- variables, initializing, 376

units, bit fields, 296

universal character names, 218

- C language specifications, 426

Unix operating system

- commands, 7
- compiler, 12
- development of, 6
- naming files, 7
- programming utilities
 - ar*, 345
 - grep*, 345
 - sed*, 345
- programs, linking, 9
- roots in C programming language, 1
- spin-offs, 6

unsigned specifier (data types), 28-30

updating time in programs, 177-180

**Updating the Time by One Second
(Program 9.5), 178-180**

**Using Comments in a Program
(Program 3.6), 17-19**

**Using Enumerated Data Types
(Program 14.1), 322-324**

**Using Linked Lists (Program 11.6),
246-250**

**Using Multidimensional Arrays and
Functions (Program 8.13), 147-150**

**Using Nested for Loops (Program
5.5), 53-54**

**Using Pointers and Functions
(Program 11.8), 254-255**

**Using Pointers in Expressions
(Program 11.3), 239-240**

**Using Pointers to Exchange Values
(Program 11.9), 255-257**

**Using Pointers to Find Length of a
String (Program 11.15), 272-273**

**Using Pointers to Structures (Program
11.4), 241-243**

**Using Structures Containing Pointers
(Program 11.5), 243-244**

**Using the #include Statement
(Program 13.3), 314-315**

**Using the Arithmetic Operators
(Program 4.2), 30-31**

**Using the Basic Data Types (Program
4.1), 26-27**

**Using the Dictionary Lookup
Program (Program 10.9), 220-222**

**usual arithmetic conversion, basic data
types, 451-452**

utilities (programming)

- a*, 345
- cv*, 344
- gre*, 345
- mak*, 343-344
- se*, 345

utility functions, 490-491

V

values

- arrays, storing, 96
- defined
 - names*, 300
 - referencing (#define statement)*, 307-308

variable-length arrays, 433

- Generating Fibonacci Numbers Using
Variable-Length Arrays (Program 7.8),
115-117
- multidimensional, 150-152

**variable-length character strings,
198-200**

variables

arrays, defining, 96-98

Boolean

Generating a Table of Prime Numbers
(Program 6.10), 87-90

Revising the Program to Generate a Table
of Prime Numbers (Program 6.10A),
90-91

C language specifications, 452-454

const (arrays), 111-113

data storage types, 21

declarations, 15

in for loops, 55-56

external, 336-338

defining, 337

versus static, 339-340

global (functions), 152-156

initializing static variables, 156-158

local

automatic (functions), 124-126, 156

defining (functions), 124-126

names, 21

reserved names, 22

rules, 22

pointers, defining, 235-239

qualifiers

register, 378-379

restrict, 379

volatile, 379

static, 156

initializing, 156-158

versus external, 339-340

storing via dynamic memory allocation,
383-384

union, initializing, 376

valid characters, 22

variants, structures of, 190-191

vi text editor, 7, 11

Visual Studio

Web site, 503

Windows IDE, 10

void data type, 128

void keyword, 128

volatile modifiers, C language
specifications, 439

volatile qualifiers, 379

W - Z

Web sites

C language resources

ANSI.org, 502

Code Warrior, 503

CygWin, 502

gcc compiler, 502

Kochan-Wood.com, 502

Kylix, 503

Metrowerks, 503

MinGW, 502

newsgroups, 502

Visual Studio, 503

gcc, 493

GNU.org, 504

Google Groups, 502

Kochan-Wood, book exercises and
errata, 501

OOP book resources

C# Programming in the Key of C#,
503

C++ Primer Plus, 503

Code Warrior, 503

Programming in Objective-C,
503-504

while statement, 56-60

C language specifications, 460

Finding the Greatest Common Divisor
(Program 5.7), 58-59

- Introducing the while Statement
(Program 5.6), 56-58
- programming looping usage, 44
- Reversing the Digits of a Number
(Program 5.8), 59-60
- whitespace characters, scanf()**
function, 355
- wide character constants, C language**
specifications, 429
- Working with an Array (Program 7.1),**
98, 100
- Working with Fractions in C**
(Program 19.1), 413-414
- Working with Fractions in C#**
(Program 19.4), 422-424
- Working with Fractions in C++**
(Program 19.3), 419-421
- Working with Fractions in**
Objective-C (Program 19.2), 414-419
- Working with gdb (Program 18.5),**
401-402
- Working with Pointers to Arrays**
(Program 11.11), 262-263
- writing**
 - files with fputs() function, 368
 - programs
 - for handling fractions (C language),*
413-414
 - for handling fractions (C# language),*
422-424
 - for handling fractions (C++ language),*
419-421
 - for handling fractions (Objective-C*
language), 414-419
- Writing in Function in C (Program**
8.1), 120-121
- Writing Your First C Program**
(Program 3.1), 11
- X3J11 committee (ANSI C), 1**
- Xcode, Mac OS X IDE, 10**
 - XOR operator, 284-285