火龙果•整理
uml.org.cn

Stephen G. Kochan 著
贾洪峰 译

PEARSON
Broadview
www.broadview.com.cn

Programming in C
(4th Edition)

C语言程序设计
（第4版）

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

火龙果•整理
uml.org.cn

内 容 简 介

本书全面介绍了 C 语言的各种特性，包括 C11 中增加的内容。书中包含大量完整的示例及详细的讲解。无论是否拥有编程经验，你都可以通过本书透彻地理解 C 语言。每章最后附有习题，便于课堂学习或自学。

Authorized translation from the English language edition, entitled Programming in C, 4E, 978-0321776419 by Stephen G. Kochan, published by PEARSON EDUCATION, INC., publishing as Addison-Wesley Professional, Copyright © 2014 Pearson Education, Inc.
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 PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright © 2015.

本书简体中文版专有出版权由 Pearson Education 培生教育出版亚洲有限公司授予电子工业出版社。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。
本书简体中文版贴有 Pearson Education 培生教育出版集团激光防伪标签，无标签者不得销售。

版权贸易合同登记号 图字：01-2014-8174

图书在版编目（CIP）数据

C 语言程序设计：第 4 版 /（美）寇肯（Kochan,S.G.）著；贾洪峰译. —北京：电子工业出版社，2015.5
书名原文：Programming in C (4th Edition)
ISBN 978-7-121-25888-6
Ⅰ. ①C… Ⅱ. ①寇… ②贾… Ⅲ. ①C 语言－程序设计 Ⅳ. ①TP312
中国版本图书馆 CIP 数据核字(2015)第 074580 号

策划编辑：张春雨
责任编辑：张春雨
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
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
开　　本：720×1000　1/16　印张：34.5　字数：607.2 千字
版　　次：2015 年 5 月第 1 版
印　　次：2015 年 5 月第 1 次印刷
定　　价：108.00 元
凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
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。
服务热线：（010）88258888。

火龙果•整理
uml.org.cn

速览

前言 1
1 基础知识 5
2 编译和运行第一个程序 13
3 变量、数据类型和算术表达式 23
4 程序循环 45
5 做出决策 69
6 使用数组 101
7 使用函数 127
8 使用结构体 175
9 字符串 205
10 指针 247
11 位运算 295
12 预处理器 317
13 用枚举数据类型、类型定义和数据类型转换扩展数据类型 341
14 处理大型程序 353
15 C 语言中的输入与输出操作 369
16 其他及高级特性 397
17 调试程序 415

火龙果•整理
uml.org.cn

18　面向对象程序设计 437
附录 A　C 语言概要 453
附录 B　标准 C 语言库 497
附录 C　用 gcc 编译程序 525
附录 D　常见编程错误 527
附录 E　资源 533

火龙果•整理
uml.org.cn

目录

前言 1

1 基础知识 5

程序设计 5
高级语言 6
操作系统 7
编译程序 7
集成开发环境 10
语言解释器 11

2 编译和运行第一个程序 13

编译程序 14
运行程序 14
理解第一个程序 15
显示变量的值 17
注释 19
练习 21

3 变量、数据类型和算术表达式 23

理解数据类型和常量 23
整数类型 int 24
浮点数值类型 float 25
扩展精度类型 double 26
单字符类型 char 26
布尔数据类型_Bool 27

火龙果•整理
uml.org.cn

类型说明符：long、long long、short、unsigned 和 signed...28
使用变量 31
使用算术表达式 33
整数算术及一元减运算符 36
运算与赋值相结合：赋值运算符 42
_Complex 和_Imaginary 类型 43
练习 43
4 程序循环 45
三角数 45
for 语句 46
关系运算符 48
输出的对齐 52
程序输入 53
嵌套 for 循环 55
for 循环的变体 57
while 语句 59
do 语句 63
break 语句 65
continue 语句 65
练习 66
5 做出决策 69
if 语句 69
if-else 构造 73
复合关系测试 76
嵌套 if 语句 79
else if 构造 81
switch 语句 88
布尔变量 91

火龙果•整理
uml.org.cn

条件运算符 96
练习 97

6 使用数组 101

定义数组 102
以数组元素为计数器 106
生成斐波那契数 109
用数组生成质数 111
初始化数组 113
字符数组 115
使用数组的基数变换 116
const 限定符 118
多维数组 120
变长数组 122
练习 124

7 使用函数 127

定义函数 127
参数和局部变量 131
函数原型声明 132
自动局部变量 132
返回函数结果 134
函数调用 139
声明返回类型和参数类型 142
检查函数参数 144
自上而下的程序设计 146
函数与数组 146
赋值运算符 151
数组排序 152
多维数组 155

火龙果•整理
uml.org.cn

全局变量 161
自动变量与静态变量 165
递归函数 168
练习 171

8 使用结构体 175

结构体的基础知识 175
用于存储日期的结构体 176
在表达式中使用结构体 178
函数与结构体 181
用于存储时间的结构体 187
初始化结构体 190
复合字面量 191
结构体的数组 192
包含结构体的结构体 196
包含数组的结构体 198
结构体变体 202
练习 203

9 字符串 205

复习字符串的基础知识 205
字符数组 206
变长字符串 209
字符串的初始化与显示 211
判断两个字符串是否相等 214
输入字符串 216
单字符输入 219
空字符串 224
转义字符 228
再论常量字符串 230

火龙果•整理
uml.org.cn

字符串、结构体和数组 231
一种更好的搜索方法 235
字符操作 240
练习 243

10 指针 247

指针和间接性 247
定义指针变量 248
在表达式中使用指针 252
使用指针和结构体 253
包含指针的结构体 256
链表 258
关键字 const 和指针 266
指针和函数 267
指针和数组 273
稍许离题：程序优化 277
是数组还是指针？ 278
指向字符串的指针 280
常量字符串和指针 282
递增和递减运算符回顾 283
指针操作 287
指向函数的指针 288
指针和内存地址 289
练习 291

11 位运算 295

位的基础知识 295
位运算符 297
按位与运算符 297
按位或运算符 300

火龙果•整理
uml.org.cn

按位异或运算符 301
二进制求补运算符 301
左移运算符 304
右移运算符 304
移位函数 305
循环移位 307
位段 309
练习 314
12 预处理器 317
#define 语句 317
程序可扩展性 322
程序可移植性 323
更高级的定义类型 324
#运算符 330
##运算符 331
#include 语句 332
系统包含文件 335
条件编译 335
#ifdef、#endif、#else 和#ifndef 语句 336
#if 和#elif 预处理器语句 338
#undef 语句 339
练习 339
13 用枚举数据类型、类型定义和数据类型转换扩展数据类型 341
枚举数据类型 341
typedef 语句 345
数据类型转换 348
符号扩展 350
参数转换 350

火龙果•整理
uml.org.cn

练习 351

14 处理大型程序 353

将程序分为多个文件 353
由命令行编译多个源文件 354
模块之间的通信 356
外部变量 357
静态、动态变量与函数 359
有效使用头文件 362
其他处理大型程序的实用工具 364
make 实用工具 364
cvs 实用工具 366
Unix 实用工具：ar、grep、sed 等 366

15 C 语言中的输入与输出操作 369

字符 I/O：getchar()和 putchar() 369
格式化 I/O：printf()和 scanf() 370
printf()函数 370
scanf()函数 377
文件输入与输出操作 382
将 I/O 重定向到文件 382
文件尾 385
处理文件的特殊函数 386
fopen 函数 387
getc()和 putc()函数 389
fclose()函数 389
feof 函数 391
fprintf()和 fscanf()函数 392
fgets()和 fputs()函数 392
stdin、stdout 和 stderr 393

火龙果•整理
uml.org.cn

exit()函数 394
重命名和删除文件 395
练习 396

16 其他及高级特性 397

其他语言语句 397
goto 语句 397
空语句 398
使用联合体 399
逗号运算符 402
类型限定符 403
register 限定符 403
volatile 限定符 403
restrict 限定符 404
命令行参数 404
动态内存分配 408
calloc()和 malloc()函数 409
sizeof 运算符 410
free 函数 413
练习 414

17 调试程序 415

用预处理器调试 415
用 gdb 调试程序 422
处理变量 425
源文件显示 426
控制程序的执行 427
获得栈轨迹 432
调试函数，设置数组和结构体 432
用 gdb 命令获取帮助 433

火龙果•整理
uml.org.cn

杂项 435

18 面向对象程序设计 437

到底什么是对象？ 437
实例与方法 438
编写一个处理分数的 C 程序 440
定义一个处理分数的 Objective-C 类 441
定义一个处理分数的 C++类 446
定义一个处理分数的 C#类 448

附录 A C 语言概要 453

1.0 双字符组和标识符 453
2.0 注释 454
3.0 常量 455
4.0 数据类型和声明 458
4.6 类型限定符 `const`、`volatile` 和 `restrict` 468
5.0 表达式 469
6.0 存储类和作用域 482
7.0 函数 483
8.0 语句 486
9.0 预处理器 490

附录 B 标准 C 语言库 497

标准头文件 497
字符串函数 500
内存函数 502
字符函数 503
I/O 函数 503
内存中的格式转换函数 509
字符串转换为数字 510
动态内存分配函数 512

火龙果•整理
uml.org.cn

数学函数 513
复数运算 519
通用函数 521

附录 C　用 gcc 编译程序 525

通用命令格式 525
命令行选项 526

附录 D　常见编程错误 527

附录 E　资源 533

C 程序设计语言 533
C 编译器和集成开发环境 534
其他 535

火龙果•整理
uml.org.cn

关于作者

Stephen G. Kochan 使用 C 语言设计软件已经有超过 30 年的时间。他是几本最畅销的 C 语言书籍的作者，包括 *Programming in C*、*Programming in Objective-C* 和 *Topics in C Programming*。他在 Unix 方面也有大量著述，是 *Exploring the Unix System* 和 *Unix Shell Programming* 等书的著者或合著者。

有贡献的作者，第 4 版

Dean Miller 是一位作家和编辑，在出版和特许消费者产品领域都拥有超过 20 年的经验。他是 *Sams Teach Yourself C in One Hour a Day* 和 *Sams Teach Yourself Beginning Programming in 24 Hours* 最近几个版本的合著者。

致谢

衷心地感谢下列各位在我准备本书各个版本时提供的帮助：Douglas McCormick、Jim Scharf、Henry Tabickman、Dick Fritz、Steve Levy、Tony Ianinno 和 Ken Brown。还要感谢纽约大学的 Henry Mullish 在写作方面给我的诸多教益，感谢他带我走进出版领域。

在 Pearson 的工作人员中，我要感谢 Mark Taber 和我的责任编辑 Mandie Frank，还要感谢我的文字编辑 Charlotte Kughen 和我的技术编辑 Siddhartha Singh。最后，在 Pearson 还有许多参与本项目但没有与我直接接触的工作人员，在此向他们所有人表示感谢。

火龙果•整理
uml.org.cn

前　言

C 程序设计语言是由 AT&T 贝尔实验室的 Dennis Ritchie 在 20 世纪 70 年代设计开发的。但一直到 20 世纪 70 年代晚期，这一程序设计语言才开始得到广泛普及和支持。这是因为，在此之前的 C 编译器无法轻松应用于贝尔实验室之外的商业用途。最初，C 语言普及程度得以提高的部分原因是 Unix 操作系统以相同甚至更快的速度得到普及。这一操作系统也是在贝尔实验室开发的，它以 C 语言作为“标准”程序设计语言。事实上，这个操作系统本身就有超过 90% 的部分是用 C 语言编写的！

IBM PC 及类似计算机的巨大成功很快使 MS-DOS 成为最流行的 C 语言环境。随着 C 语言在不同操作系统上变得日益普及，越来越多的厂商加入了这一潮流，开始推销自己的 C 编译器。其中绝大多数的 C 语言版本都基于第一本 C 语言程序设计教科书中的一个附录，这本书就是 Brian Kernighan 和 Dennis Ritchie 的《C 程序设计语言》（*The C Programming Language*）。遗憾的是，这个附录中并没有给出一个完整的、无歧义的 C 语言定义，也就是说，这门语言的某些方面要留待供应商自行解读。

20 世纪 80 年代早期，出现了对 C 语言的定义进行标准化的需求。美国国家标准委员会（ANSI）作为处理此类事务的一家组织，其在 1983 年因此成立了 ANSI C 委员会（称为 X3J11），负责实现 C 语言的标准化。1989 年，该委员会的工作获得批准。1990 年，C 语言的第一个正式 ANSI 标准定义发布。

因为 C 语言在世界各地都有应用，所以国际标准组织（ISO）很快参与了进来。他们采用了上述标准，称之为 ISO/IEC 989:1990。在此之后，又对 C 语言进行了其他修改。最近的标准是 2011 年采用的，称为 ANSI C11 或 ISO/IEC 9899:2011。本书便基于 C 语言的这一版本。

火龙果•整理
uml.org.cn

C 语言是一种“高级语言”，但它提供的功能可以让用户“接近”硬件，在非常低的层次上使用计算机。这是因为，尽管 C 是一种通用的结构化程序设计语言，但它在最初设计时就考虑了系统程序设计应用，从而为用户提供了强大的功能和灵活性。

本书希望能够教会你如何用 C 语言设计程序。它假定读者之前没有接触过此语言，希望能够吸引初学者和富有经验的程序员。如果之前拥有程序设计经验，你将会发现 C 语言的工作方式非常独特，可能不同于之前用过的其他语言。

本书将讨论 C 语言的每个特性。在每次介绍一个新的特性时，通常会给出一个小型的**完整**程序范例，用以介绍该特性。这种做法反映了本书的首要编写理念：寓教于例。一幅图片胜过千言万语，一个正确选择的程序范例也是如此。如果你可以使用一台支持 C 程序设计语言的计算机，强烈建议下载并运行本书给出的每个程序，并将你在自己系统中获得的结果与书中给出的结果进行对比。这样，你不仅可以学习这门语言及其语法，还能熟悉键入、编译和运行 C 程序的过程。

你会发现本书自始至终都在强调程序的可读性。这是因为我坚信，编写程序应当使其易于理解，无论是对于编写者还是对于其他人，都应如此。借助经验、常识，你会发现这样的程序总是更容易编写、调试和修改。另外，只要严格遵守一套结构化程序设计规则，自然就能开发出易于理解的程序。

因为本书是作为教材编写的，所以每一章讨论的内容都以之前给出的内容为基础。因此，依次阅读每一章可以由此书获得最大裨益，强烈建议不要“跳跃式阅读”。另外，在继续学习下一章内容之前，应当完成每章最后给出的练习。

第 1 章“基础知识”介绍了一些有关高级程序设计语言的基础术语，还有程序的编译过程。这一章的目的是确保你能理解本书其余部分使用的描述语言。从第 2 章“编译和运行第一个程序”开始，本书会一步步地向你介绍 C 语言。等到学习了第 15 章“C 语言中的输入与输出操作”，这门语言的所有基本特性都已介绍完毕。第 15 章更深入地研究了 C 语言中的 I/O 操作。第 16 章“其他及高级特性”包含了这门语言中更为高级或更为深奥的一些特性。

第 17 章“调试程序”讲解如何利用 C 语言预处理器来帮助调试程序。这一章还介绍了交互式调试，并选择了流行的调试器 gdb 来阐明这一调试方法。

在过去的十年里，程序设计世界已经到处充斥着“面向对象程序设计”

火龙果•整理
uml.org.cn

（OOP）的概念。C 语言不是 OOP 语言；但是，其他几种以 C 语言为基础的程序设计语言是 OOP 语言。第 18 章“面向对象程序设计”简单地介绍了 OPP 和它的一些术语，并对三种基于 C 语言的 OOP 语言进行了概述，即 C++、C#和 Objective-C。

附录 A“C 语言概要”提供了该语言的完整概要，供参考使用。

附录 B“标准 C 语言库”提供了大量标准库例程的概要，在所有支持 C 语言的系统中都可以找到这些标准库例程。

附录 C“用 gcc 编译程序”总结了在用 GNU 的 C 编译器 gcc 编译程序时的许多常用选项。

在附录 D“常见编程错误”中，可以找到一份常见编程错误的清单。

最后的附录 E“资源”给出了一份资源清单，可用于获取有关 C 语言的更多信息，以做深入研究。

对于实现 C 语言的计算机系统或操作系统，本书未做任何具体假设，只是简要地提到如何使用流行的 GNU C 编译器 gcc 来编译和执行程序。

火龙果•整理
uml.org.cn

1 基础知识

本章介绍了一些基础术语，必须在理解这些术语之后才能学习如何用 C 语言设计程序。这一章对高级语言程序设计的本质进行了一般概述，还讨论了使用高级语言开发的程序的编译过程。

程序设计

计算机实际上是非常愚钝的机器，因为它们只会完成你让它们完成的工作。大多数计算机系统都在一个非常原始的层次上执行其操作。例如，大多数计算机知道如何将一个数字与另一个数字相加，或者如何判断一个数字是否等于 0。基本运算的复杂程度通常比上述运算高不了多少。计算机系统的基本运算就构成了计算机的**指令集**。

要使用一台计算机解决一个问题，就必须用这一特定计算机的指令来表述这一问题的解。计算机**程序**就是解决一个特定问题所需要的一组指令。用于解决问题的方法称为**算法**。例如，如果希望开发一个判断数字奇偶性的程序，求解该问题的语句集就变成了程序，用于判断数字奇偶性的方法就是算法。正常情况下，要开发一个解决某一特定问题的程序，首先要用一种算法来表述这个问题的解，然后再开发一个程序来实现该算法。用于解决奇偶性问题的算法可表述如下：首先，将数字除以 2。如果除法得到的余数为 0，则该数字为偶数；否则为奇数。有了这一算法，接下来就可以编写必要的指令，在一个特定的计算机系统上实现该算法。这些指令将用一种特定的计算机语言的语句来表述，比如 Java、C++、Objective-C，或是 C。

火龙果•整理
uml.org.cn

高级语言

在最初开发计算机时，对其进行程序控制的唯一方法是使用一些二进制数字，它们直接对应于特定的机器指令和计算机内存中的位置。随后，技术软件的发展进步体现在**汇编语言**的开发上，这种语言可以让程序员在稍微高级一点的层次上使用计算机。程序员在使用汇编语言时，不再需要指定二进制数字序列来执行特定任务，而可以用符号名来执行各种操作、引用内存位置。一种称为**汇编器**的特殊程序将汇编语言程序由其符号格式转换为计算机系统的特定机器指令。

因为每条汇编语言语句与特定机器指令之间仍然存在一一对应关系，所以汇编语言被看作低级语言。程序员仍然必须学习特定计算机系统的指令集，才能用汇编语言编写程序，最终得到的程序是不**可移植的**；也就是说，要想在不同类型的处理器上运行程序，必须重新编写。这是因为不同类型的处理器拥有不同的指令集，而汇编语言程序是用这些指令集编写的，它们是随计算机而变化的。

于是就出现了所谓的高级语言，FORTRAN（FORmula TRANslation）语言是最早的高级语言之一。用 FORTRAN 开发程序的程序员不再需要关心特定计算机的体系结构，于是就出现了所谓的高级语言，FORTRAN 语言是最早的高级语言之一。用 FORTRAN 开发程序的程序员不再需要关心特定计算机的体系结构，FORTRAN 的操作级别要高得多、复杂得多，与特定计算机的指令集没有直接关系。一条 FORTRAN 指令或**语句**会导致许多不同机器指令的执行，而不像汇编语言的语句那样，与机器指令之间存在一一对应关系。

高级语言语法的标准化，意味着可以用一种与机器无关的语言来编写程序。也就是说，只需稍做或不做修改，程序就能在任何支持该语言的计算机上运行。

为了支持高级语言，必须开发一种特殊的计算机程序，能将以高级语言开发的程序语句转换为计算机可以理解的形式，换句话说，转换为计算机的特定指令。这种程序称为**编译器**。

火龙果•整理
uml.org.cn

操作系统

有一种计算机程序被称为**操作系统**，在继续讨论编译器之前，值得花点时间了解一下操作系统扮演的角色。

操作系统是一种控制计算机系统所有操作的程序。在计算机系统上执行的所有输入和输出（即 I/O）操作都是通过操作系统这一渠道来完成的。操作系统还必须管理计算机系统的资源，必须操控程序的执行。

当今最流行的操作系统之一是 Unix 操作系统，它是在贝尔实验室开发出来的。Unix 是一种相当独特的操作系统，因为它可以在许多不同类型的计算机系统中找到，而且拥有不同“变体（flavor）”，比如 Linux 或 Mac OS X。在历史上，操作系统通常仅与一种类型的计算机系统相关联。但因为 Unix 主要是用 C 语言编写的，在计算机的体系结构上几乎没有进行什么假设，所以它已经被成功地移植到许多不同的计算机系统上，移植所耗费的工作量是相当少的。

Microsoft Windows 是流行操作系统的另一个实例。该系统主要运行在英特尔（或英特尔兼容的）处理器上。

最近出现的优秀操作系统是为便携设备（比如移动电话和平板电脑）开发的一些系统。苹果公司的 iOS 和谷歌公司的 Android 操作系统是其中最流行的两个实例。

编译程序

编译器是一种软件程序，从原理上来说，它与你将在本书中看到的程序没有什么区别，当然，它肯定要复杂得多。编译器分析以特定计算机语言编写的程序，然后将它转换为适合在特定计算机系统上执行的形式。

图 1.1 针对一个用 C 程序设计语言开发的计算机程序，给出了其输入、编译和执行步骤，以及在命令行中输入的典型 Unix 命令。

首先将待编译程序键入计算机系统上的一个**文件**中。计算机设备对文件的命名有着各种不同约定，但一般情况下，名字的选择由你自己来决定。C 程序通常可以取任意名字，只要最后两个字符是“.c”即可（这更多的是一种约定，而不是强制要求）。因此，`prog1.c` 在你的系统上应当是一个有效的 C 程序名字。

火龙果•整理
uml.org.cn

通常使用文本编辑器将 C 程序输入到文件中。例如，vim 是 Unix 系统上一个很流行的文本编辑器。输入到文件中的程序称为**源程序**，因为它代表着用 C 语言表述的原始程序形式。在将源程序输入到一个文件中之后，就可以开始编译它了。

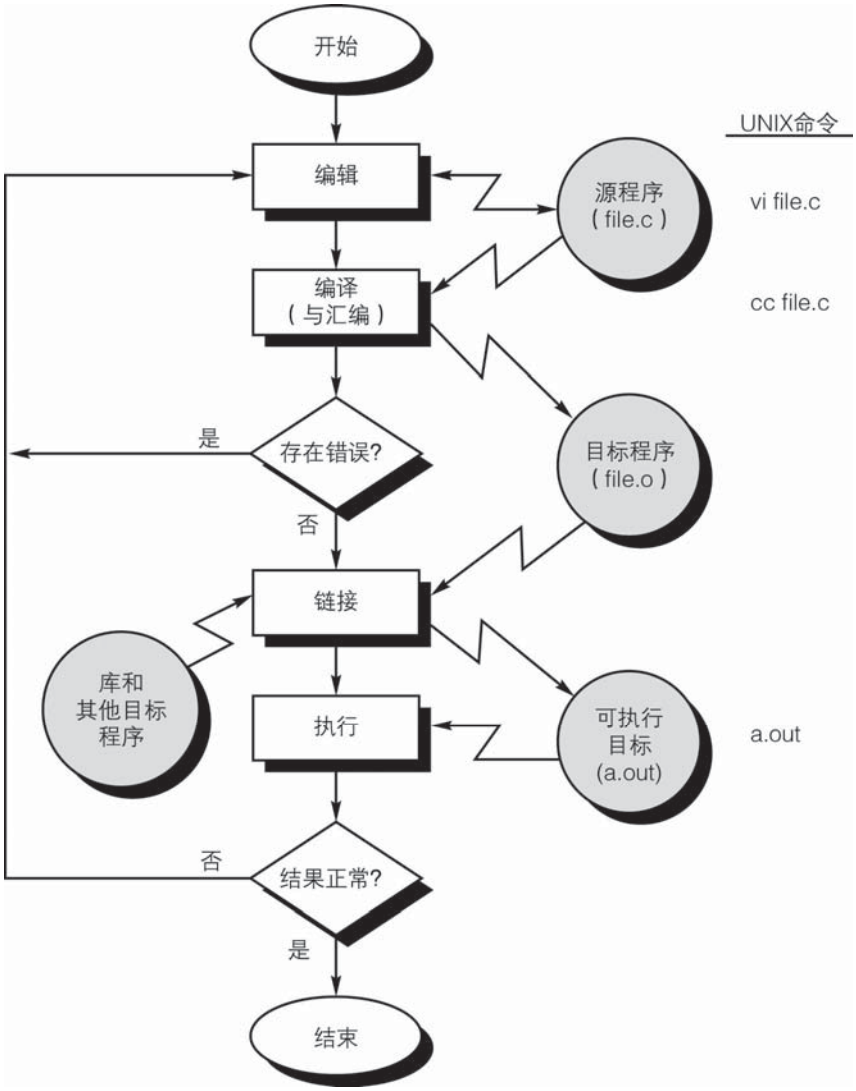


图 1.1　由命令行输入、编译和执行 C 程序的典型步骤

火龙果•整理
uml.org.cn

在系统中键入一个特殊命令即可启动编译过程。在键入这个命令时，必须指定源程序所在文件的名字。例如，在 Unix 系统中，启动程序编译过程的命令称为 cc。如果使用的是流行的 GNU C 编译器，那要使用的命令就是 gcc。键入以下命令行

```
gcc prog1.c
```

其效果就是对 prog1.c 中包含的源程序启动编译过程。

编译过程的第一个步骤就是由编译器检视源程序中包含的每条程序语句，对其进行核查，确保它符合 C 语言的语法和语义[①]。如果编译器在这一阶段发现了任何错误，会将它们报告给用户，编译过程就此结束。随后必须在源程序中纠正这些错误（使用编辑器完成），然后必须重新启动编译过程。在这一编译阶段报告的典型错误可能是表达式中的括号不平衡（句法错误），也可能是使用了一个未“定义”的变量（语义错误）。

在清除了程序中的所有句法和语义错误之后，编译器接下来会获取程序中的每条语句，将其转换为一种“较低级”的形式。在大多数系统中，这意味着每条语句会被编译器转换为执行同一任务所需要的等价汇编语句。

在将程序转换为等价的汇编语言程序之后，编译过程的下一个步骤就是将汇编语言语句转换为实际的机器指令。这一步骤可能会执行一种称为**汇编器**的独立程序，也可能不会涉及。在大多数系统中，汇编器是作为编译过程的一部分自动执行的。

汇编器获取每条汇编语言语句，将其转换为一种称为**目标代码**的二进制形式，然后将其写到系统上的另一个文件中。在 Unix 中，这一文件通常与源文件同名，最后一个字母是“o”（表示 **object**，**目标**），而不再是“c”。在 Windows 系统中，通常会用后缀字母“obj”代替文件名中最后的“c”。

在将程序转换为目标代码之后，就可以对其进行**链接**了。同样，在 Unix 中，只要发出了 cc 或 gcc 命令，就会自动执行这一过程。链接阶段的目的是将程序变为在计算机上执行的最终形式。如果程序中用到了之前由编译器处理过的程序，会在这一阶段将这些程序链接在一起。在这一阶段，还会从系统程

① 严格来说，C 编译器通常会对程序进行一次预扫描，查找特殊语句。这一预处理阶段会在第 12 章中详细介绍。

火龙果•整理
uml.org.cn

序**库**中查找会用到的程序，并将它们与目标程序链接在一起。

编译和链接一个程序的整个过程通常称为**生成**。

链接得到的最终文件具有一种**可执行目标**代码格式，存储在系统上的另一个文件中，为运行或**执行**做好了准备。在 Unix 中，这个文件的默认名为 `a.out`。在 Windows 中，可执行文件通常与源文件同名，但用 `exe` 扩展名代替了 `c` 扩展名。

随后要执行程序时，只需要键入可执行目标文件的名字即可。因此，命令

```
a.out
```

的效果就是将一个名为 `a.out` 的程序**加载**到计算机的内存中，并开始执行。

在执行程序时，程序中的每条语句都被依次顺序执行。如果程序需要来自用户的任意数据（称为**输入**），可能会暂时挂起程序的执行，以便键入这些输入内容。或者，这个程序可能就是等待着发生某一**事件**，比如单击一次鼠标。由程序显示的结果称为**输出**，将出现在一个窗口中，这个窗口有时称为**控制台**。或者，输出内容可以被直接写入系统上的一个文件中。

如果一切正常（可能不是第一次执行该程序），程序会执行其预期功能。如果程序没有给出预期结果，就需要回过头来，重新分析程序的逻辑。这一阶段称为**调试阶段**，在此阶段，将尝试清除程序中的所有已知问题或 **bug**（**错误**）。为此，很可能需要修改最初的源程序。在这种情况下，必须重复编译、链接和执行程序的整个过程，直到获得预期结果为止。

集成开发环境

前面概括了开发 C 程序时涉及的各个步骤，给出了在每一步骤输入的典型命令。这个编辑、编译、运行和调试程序的过程经常由单个集成应用程序进行管理，这种应用程序称为集成开发环境，或简称为 IDE。IDE 是一个基于窗口的程序，利用它可以轻松管理大型软件程序，在窗口中编辑文件，编译、链接、运行和调试程序。

在 Mac OS X 中，Xcode 是由苹果公司支持的一个 IDE，许多程序员都在使用它。在 Windows 中，Microsoft Visual Studio 是流行 IDE 的一个优秀例子。所有这些 IDE 应用程序都极大地简化了整个程序开发过程，因此值得花点时间

火龙果•整理
uml.org.cn

来学习使用其中一种。除了 C 语言之外，大多数 IDE 还支持用其他几种不同的程序设计语言来开发程序，比如 Objective-C、Java、C#和 C++。

有关 IDE 的更多信息，请参考附录 E。

语言解释器

在结束对编译过程的讨论之前，还要注意，有另外一种方法可用于分析和执行用高级语言开发的程序。在采用这些方法时，不是对程序进行编译，而是进行**解释**。解释器对程序语句的分析和执行是同时进行的。这种方法通常可以让程序变得更容易调试。但另一方面，解释语言的速度通常要慢于相应的编译语言，因为程序语句在执行之前没有被转换为低级形式。

BASIC 和 JavaScript 这两种程序设计语言的程序通常是进行解释而不是编译的。其他此类例子包括 Unix 系统的 **shell** 和 Python。一些厂商也提供 C 程序设计语言的解释器。

火龙果•整理
uml.org.cn

2 编译和运行第一个程序

本章将为你介绍 C 语言，让你了解用 C 语言设计程序的整体过程。要了解这一语言，还有什么办法会比查看一个真正的 C 程序更好呢？

这一章很简短，但你会惊讶地发现，在这样短短的一章里你可以了解到非常丰富的内容，包括：

- 编写第一个程序
- 修改程序，以改变其输出
- 理解 `main()` 函数
- 用 `printf()` 函数输出信息
- 用注释提高程序的可读性

首先，你要选择一个相当简单的例子——一个在窗口中显示“Programming is fun.”的程序。程序 2.1 给出了一个完成这一任务的 C 程序。

程序 2.1　编写第一个 C 程序

```
#include <stdio.h>

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

   return 0;
}
```

在 C 程序设计语言中，大写与小写字母是有区别的。此外，在 C 语言中，在一行中的什么位置开始键入语句是无关紧要的——可以在一行中的任意位置开始键入语句。在开发程序时可以利用这一点，提高程序的可读性。程序员经常使用制表符来方便地缩进代码行。

火龙果•整理
uml.org.cn

编译程序

回到你的第一个 C 程序，首先需要将它键入到一个文件中。可以使用任意文本编辑器来完成这一任务。Unix 用户经常使用诸如 vi 或 emacs 之类的编辑器。

C 编译器将以字符“.”和“c”结尾的文件名识别为 C 程序。因此，假定你将程序 2.1 键入到一个名为 `prog1.c` 的文件中。接下来需要编译该程序。

如果使用 GNU C 编译器，这一过程非常简单，就是在终端上发出后面跟有文件名的 `gcc` 命令，比如：

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

如果使用的是标准 Unix C 编译器，这一命令就是 `cc`，而不是 `gcc`。这里，要键入的文本以粗体表示。如果你正在通过命令行编译 C 程序，那这个美元符号就是命令提示符。实际命令符可能是美元符号之外的某些字符。

如果在键入程序时出错，编译器会在你输入 `gcc` 命令之后列出这些错误，通常会给出程序中包含错误的行号。如果出现了另一个命令提示符，就像上面的例子那样，那就说明没有在程序中发现错误。

在编译器编译和链接程序时，它会为你的程序生成一个可执行版本。使用 GNU 或标准 C 编译器时，这个程序的默认名为 `a.out`。在 Windows 中通常为 `a.exe`。

运行程序

现在只需要在命令行中键入可执行文件的名字就可以执行它了：[①]

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

在编译程序时，也可以为可执行文件指定一个不同的名字。为此，可以在 `-o`（字符 O）选项后面给出可执行文件的名字。例如，下面的命令行

① 如果看到类似于下面这样的错误：`a.out: No such file or directory`，那可能意味着当前目录不在你的 `PATH` 中。可以将它添加到 `PATH` 中，也可以在命令提示符后面键入：`./a.out`。

火龙果•整理
uml.org.cn


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

将编译程序 prog1.c，并把可执行文件放在文件 prog1 中，以后就能通过指定其名字来运行它：

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

理解第一个程序

更仔细地观察一下你的第一个程序。程序中的第一行是

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

在你编写的几乎所有程序的开头，都应当包含这一行。它向编译器通告了 printf()输出例程的相关信息，在程序的后面将会用到这一例程。第 12 章会详细讨论这一代码行做了些什么。

程序中的代码行

```
int main(void)
```

告诉系统，这个程序的名字是 main()，它**返回**一个整数值，缩写为“int”。main()是一个特殊的名字，它准确地指明了应当从什么**位置**开始执行程序。紧跟在 main 后面的左右括号表明 main()是一个**函数**的名字。放在括号里面的关键字 void 表明函数 main()没有参数，也就是说，参数为 **void**（**空**）。第 7 章非常详细地解释了这些概念。

> **注意**
>
> 如果使用的是 IDE，你可能发现它为你生成了一个模板 main()。在这种情况下，你可能会发现 main()函数的第一行更可能如下所示：
>
> ```
> int main(int argc, char *argv[])
> ```
>
> 它不会影响程序的操作，所以现在只要忽略这些区别即可。

向系统确认了 main()之后，就可以清晰地指明这个例程要做些什么了。为此，可将该例程的所有程序语句都放在一对大括号中。包含在这对大括号之间的所有程序语句都被系统看作 main()例程的一部分。在程序 2.1 中，这种语句只有两个。第一个语句指明，要**调用**一个名为 printf()的例程。要传递

火龙果•整理
uml.org.cn

给 printf() 例程的参数或**实参**是一个字符串：

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

printf() 例程是 C 语言库中的一个函数，它的功能就是在屏幕上显示它的参数（稍后将会看到多个参数的情况）。字符串的最后两个字符，也就是反斜线（\）和字母 n，合在一起被称为**换行(newline)**符。换行符要求系统完成的操作正如其名字所示——移至一个新行。任何将在换行符之后打印的字符都会出现在显示器的下一行中。事实上，换行符在概念上类似于打字机上的回车键。（还记得这些吗？）

C 中的所有程序语句都**必须**以分号（;）结尾。这就是为什么在 printf() 调用的右括号之后马上跟有一个分号的原因。

main() 中的最后一个语句是

```
return 0;
```

它要求完成 main() 的执行，并向系统返回一个状态值 0。这里可以使用任意整数。根据约定，0 表示程序成功完成，也就是说，没有遇到任何错误。可以用不同的数字表示发生了不同类型错误的情景（比如未找到某个文件）。这一退出状态可供其他程序（比如 Unix shell）测试，以了解该程序的运行是否成功。

现在已经完成了对第一个程序的分析，可以对其进行修改，也可使其显示句子“And programming in C is even more fun.”。只需再调用一次 printf() 例程即可，如程序 2.2 中所示。别忘了，每个 C 程序语句都必须以分号结尾。

程序 2.2

```
#include <stdio.h>

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

   return 0;
}
```

如果键入程序 2.2，然后编译并执行，应当可以在程序的输出窗口（有时称为“控制台”）中看到如下输出。

火龙果•整理
uml.org.cn

程序 2.2　输出

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

在下一个程序示例中将会看到，并不需要为每行输出内容分别调用 printf() 例程。研究程序 2.3，并在查看输出之前预测其结果。（现在可别作弊！）

程序 2.3　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 2.3　输出

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

显示变量的值

printf() 例程是本书中最常用的例程。这是因为它提供了一种显示程序结果的简便方式。它不仅可以显示简单的句子，还能显示**变量**的值和计算结果。事实上，程序 2.4 使用 printf() 例程显示了将两个数字 50 和 25 相加的结果。

程序 2.4　显示变量

```
#include <stdio.h>

int main (void)
{
   int sum;

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

   return 0;
}
```

火龙果•整理
uml.org.cn

程序 2.4　输出

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

在程序 2.4 中，第一个 C 程序语句将**变量** sum **声明**为**整数**类型。C 语言要求，所有程序变量都必须在声明之后才能在程序中使用。变量的声明告诉 C 编译器，程序将会如何使用一个特定变量。编译器需要这一信息来生成正确指令，以在变量中存入值和从中提取值。被声明为 int 类型的变量只能用于保存整数，也就是没有小数位的数。整数值的例子包括 3、5、–20 和 0。含有小数位的数，比如 3.14、2.455 和 27.0 被称为**浮点**数。

整型变量 sum 用于存储将两个数字 50 和 25 相加的结果。在这个变量的声明语句之后有意留出了一个空行，以便在视觉上将例程的变量声明与程序语句隔离开来；这完全是风格问题。有时，在程序中增加一个空行有助于提高程序的可读性。

程序语句

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

与其在大多数其他程序设计语言中时的含义相同：将数字 50 与数字 25 相加（由加号可知），并将结果存储在变量 sum 中（由**赋值运算符**，即等号可知）。

在程序 2.4 的 printf() 例程调用中，其括号中现在有两项，或者说两个**参数**。这些参数由逗号分隔。printf() 例程的第一个参数总是要显示的字符串。但是，在显示字符串时，经常还会希望随之一同显示特定程序变量的值。在本例中，希望将变量 sum 的值显示在如下字符的最后：

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

第一个参数中的百分号字符是一种可由 printf() 函数识别的特殊字符。紧跟在百分号之后的字符指明了将要在该点显示什么**类型**的值。在前面的程序中，字符 i 被 printf() 例程理解为要显示一个整数值。[①]

当 printf() 例程在一个字符串中发现 %i 字符时，它会自动显示 printf() 例程下一个参数的值。因为 sum 是 printf() 的下一个参数，所以它的值会在显示了字符“The sum of 50 and 25 is ”之后自动显示。

① 注意，printf 还允许指定 %d 格式字符来显示整数。本书在后续各章将一直使用 %i。

火龙果•整理
uml.org.cn

现在尝试预测程序 2.5 的输出。

程序 2.5　显示多个值

```
#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 2.5　输出

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

第一个程序语句将名为 value1、value2、sum 的三个变量均声明为 int 类型。这个语句也可以表示为三个独立的声明语句，其效果是等价的，如下所示：

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

在声明了这三个变量之后，程序为变量 value1 赋值 50，为 value2 赋值 25。然后计算这两个变量的和，并将结果赋值给变量 sum。

对 printf() 例程的调用现在包含了四个参数。第一个参数通常被称为**格式字符串**，用于向系统描述如何显示其余参数。value1 的值将在显示了“The sum of ”字符之后立即显示。同样，value2 和 sum 的值也将输出在适当位置，这些位置由格式字符串中随后出现的两个%i 字符指明。

注释

本章的最后一个程序（程序 2.6）介绍了**注释**的概念。程序中用注释语句来注解程序，提高其可读性。在下面的例子中将会看到，注释就是要告诉程序的

火龙果•整理
uml.org.cn

读者（程序员或其他负责维护此程序的某个人），程序员在编写某一程序或者某一语句序列时是怎么想的。

程序 2.6　在程序中使用注释

```
/*此程序将两个整数值相加
  并显示结果            */

#include <stdio.h>

int main(void)
{
   //声明变量
   int value1, value2, sum;

   //赋值并计算它们的和
   value1 = 50;
   value2 = 25;
   sum = value1 + value2;

   //显示结果
   printf("The sum of %i and %i is %i\n", value1, value2, sum);

   return 0;
}
```

程序 2.6　输出

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

向 C 程序中插入注释的方式有两种。一种注释可以由两个字符/和*开始。它们标志着注释的**开头**。这种类型的注释必须明确**终结**。要结束该注释，要使用没有任何嵌入空格的*和/。左/*和右*/之间的所有字符都被看作注释语句的组成部分，被 C 编译器忽略。当注释内容需分跨在程序中的多行时，经常使用这种注释形式。第二种向程序中添加注释的方式是使用两个连续的斜线字符//。从这两个斜线直到该行结尾的所有字符都会被编译器忽略。

在程序 2.6 中，使用了四个独立的注释语句。这个程序的其他内容与程序 2.5 相同。应当承认，这个例子有点弄巧成拙了，因为只有程序开头的第一个注释是有用的。（确实，如果在一个程序中插入如此之多的注释，实际效果将会降低而不是提高程序的可读性！）

在程序中恰当地使用注释语句，是如何强调都不为过的。在许多时候，当

火龙果•整理
uml.org.cn

程序员再次查看自己之前编写的程序时，虽然只是过去了六个月的时间，他却惊慌地发现自己完全记不起一个特定例程或一组特定语句的目的了。在程序中的特定位置精心插入一条简单的注释语句，也许就能省下大量时间，而不必再在反思例程或语句集的逻辑上浪费时间。

养成在编写或键入程序时插入注释语句的习惯，是一个很不错的主意。它的理由有很多。第一，在你还非常清楚程序逻辑时对程序进行注解，显然要比在程序完成之后再回头重新思考逻辑时进行注解容易得多。第二，提前在这一阶段向程序中插入注释之后，在调试阶段隔离调试程序逻辑错误时，也能从这些注释中受益。一条注释不仅有助于读通程序，还可以帮助指出逻辑错误的来源。最后一个理由：我还没有发现真心喜欢对程序进行解释的程序员。事实上，在完成程序的调试之后，你可能不会再愿意回过头来向程序中添加注释了。在开发程序时就插入注释，可以让这个时常显得乏味的任务稍微容易忍受一些。

本章对于如何用 C 语言开发程序做了初步介绍，其内容到此结束。现在，你应该已经很好地了解了在用 C 语言编写程序时会涉及哪些内容，而且应该可以自行开发小型程序了。下一章，你将开始学习这一极为强大和灵活的程序设计语言的一些更出色的复杂内容。但首先，请着手完成以下练习，以确认自己理解了本章给出的概念。

练习

1. 键入并运行本章给出的 6 个程序。将每个程序产生的输出与书中每个程序之后给出的输出进行对比。
2. 编写一个程序，输出以下文本。
   （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. 你认为以下程序会给出什么样的输出？

```
#include <stdio.h>

int main(void)
{
```

火龙果•整理
uml.org.cn


```
   printf ("Testing...");
   printf("....1");
   printf("...2");
   printf("..3");
   printf("\n");

   return 0;
}
```

4. 编写一个程序，从 87 中减去 15，并在终端上显示计算结果及一条适当的信息。
5. 找出以下程序中的句法错误，然后键入并运行纠正过的程序，确保你正确地找出了所有错误。

```
#include <stdio.h>

int main(Void)
(
   INT sum;
   /* 计算结果
   sum = 25 + 37 - 19
   /*  显示结果 //
   printf ("The answer is %i\n" sum);
   return 0;
}
```

6. 你认为以下程序会给出什么样的输出？

```
#include <stdio.h>

int main(void)
{
   int answer, result;

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

   return 0;
}
```

火龙果•整理
uml.org.cn

3

变量、数据类型和算术表达式

程序的真正本领在于它们对数据的操作。为充分发挥程序的这一本领，需要很好地理解所能使用的不同数据类型，以及如何创建变量并为其命名。C 语言拥有大量可用于处理数据的数学运算符。本章将会学习：

- int、float、double、char 和_Bool 数据类型
- 用 short、long 和 long long 修改数据类型
- 变量的命名规则
- 基本数学运算符和算术表达式
- 类型转换

理解数据类型和常量

我们已经见过 C 语言的基本数据类型 int。你一定想起来了，声明为 int 类型的变量只能包含整数值，也就是不包含小数位的值。

C 程序设计语言提供了其他四种基本数据类型：float、double、char 和_Bool。声明为 float 类型的变量可用于存储浮点数（包含小数位的值）。double 类型与 float 类型相同，只是拥有大约两倍的精度。char 数据类型可用于存储单个字符，比如字符'a'、数字'6'或分号（';'）（后面还会再次讨论这一字符）。最后是_Bool 数据类型，它只能用于存储数值 0 或 1。这一类型的变量用于表示开/关、是/否、真/假等情景。这种非此即彼的选择也称为二元选择。

在 C 语言中，任意数字、单个字符或字符串都称为**常量**。例如，数字 58 表示一个常量整数值。字符串"Programming in C is fun. \n"是常量字符串的一个例子。完全由常量值组成的表达式称为**常量表达式**。因此，表达式

火龙果•整理
uml.org.cn


```
128 + 7 - 17
```

是一个常量表达式，因为表达式中的每一项都是一个常数值。但如果 i 被声明为整型变量，那表达式

```
128 + 7 - i
```

就不再表示常量表达式，因为它的值会根据 i 的取值而变化。如果 i 为 10，该表达式等于 125，但如果 i 为 200，该表达式就等于–65。

整数类型 int

在 C 语言中，一个整数常量包括由一个或多个数位组成的序列。序列前面的负号表示这个值是负数。数值 158、-10 和 0 都是有效的整数常量实例。在数位之间不允许嵌入空格，大于 999 的整数值不能使用逗号。（因此，数值 12,000 不是有效的整数常量，必须写为 12000。）

C 语言中的两种特殊格式允许将整数常量表示为非十进制（十进制基数为 10）。如果整数值的第一个数位为 0，就认为该整数是用**八进制**表示的，也就是说基数为 8。在这种情况下，该整数值的其余数位必须是有效的八进制数位，也就是说，必须为 0～7。因此，要在 C 语言中表示八进制的数值 50（等价于十进制的数值 40），应当使用符号 050。同理，八进制常量 0177 表示十进制数 127（1 × 64 + 7 × 8 + 7）。在 printf()语句的格式字符串中使用格式字符%o，就可以在终端上以八进制显示整数值。在这种情况下，该数值显示为不带前导零的八进制。使用格式字符%#o 则可以在八进制值之前显示一个前导零。

如果一个整数常量前面带有一个 0 和一个字母 x（无论是大写还是小写），该数值会被看作是用十六进制（基数为 16）符号表示的。紧跟在字符 x 之后的是十六进制数值的数位，包括数字 0～9 和字母 a～f（或 A～F）。这些字母分别表示数值 10～15。因此，要将十六进制值 FFEF0D 赋给一个名为 rgbColor 的整型变量，可以使用语句

```
rgbColor = 0xFFEF0D;
```

使用格式字符%x 可以十六进制格式显示一个值，而且不带前导的 0x，十六进制数位用小写字母 a～f 表示。要显示带有前导 0x 的值，可以使用格式字符%#x，如下所示：

火龙果•整理
uml.org.cn


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

要用大写字母显示前导 x 和随后的十六进制数位，可使用大写 x，如%X 或%#X。

存储大小与范围

每种数值，无论是字符、整数还是浮点数，都有一个相关联的取值**范围**。为存储某一特定数据类型，需要为其分配一定数量的存储，取值范围与这一数量有关。一般情况下，这一数量并没有在语言中做出规定。它通常取决于你正在使用的计算机，因此，称为**实现**相关的，或者说是**机器**相关的。例如，一个整数可能在你的计算机上占用 32 位，也可能存储为 64 位。在编写程序时，绝对不应当对数据类型的大小做出任何假设。但可以向你保证的是，会为每种基本数据类型留出最少量的存储空间。例如，可以保证一个整数值的最小存储空间为 32 位，这是许多计算机上一个“字”的大小。

浮点数值类型 float

被声明为 float 类型的变量可用于存储包含小数位的值。浮点常量的标识是其中存在一个小数点。可以省略小数点之前的数位，也可以删除之后的数位，但显然不能同时删除两者。数值 3.、125.8 和-.0001 都是浮点常量的有效示例。为了在终端上显示一个浮点数，可以使用 printf 转换字符%f。

浮点常量还可以用**科学记数法**表示。数值 1.7e4 是用此种方式表示的一个浮点值，代表数值 1.7×10^4。字母 e 之前的数值称为**尾数**，之后的数值称为**指数**。指数表示 10 的幂（前面带有一个可选的正号或负号），尾数将与这个幂值相乘。因此，在常量 2.25e-3 中，2.25 是尾数值，-3 是指数值。这个常量表示的数值为 2.25×10^{-3} 或 0.00225。顺便说一下，将尾数与指数分隔开来的字母 e 既可以是小写，也可以是大写。

要以科学记数法显示一个值，应当在 printf()格式字符串中指定格式字符%e。利用 printf()格式字符%g，可以让 printf()来判断是用普通的浮点形式来显示浮点值，还是用科学记数法来显示。其判断依据是指数的取值：如果小于–4 或大于 5，则使用%e（科学记数法）格式；否则，使用%f 格式。

应当用%g 格式字符来显示浮点数，它生成的输出格式最为美观。

十六进制浮点数包括一个前导 0x 或 0X，后面是一个或多个十进制或十六

火龙果•整理
uml.org.cn

进制数位，然后是一个 p 或 P，接下来是一个可视需要带有正负号的二进制指数。例如，0x0.3p10 表示数值 $3/16\times2^{10}=0.5$。

扩展精度类型 double

double 类型与 float 类型非常相似，当 float 变量提供的范围不足时，就会使用 double 类型。声明为 double 类型的变量所能存储的有效位数大约是 float 类型变量的两倍。大多数计算机都使用 64 位表示 double。

除非另行告知，否则 C 编译器会将所有浮点常量看作 double 值。为明确地表示一个 float 常量，可在数字的最后追加一个 f 或 F，如下所示：

```
12.5f
```

要显示一个 double 值，可以使用格式字符%f、%e 或%g，与用于显示 float 值的格式字符相同。

单字符类型 char

char 变量可用于存储单个字符。[1]字符常量由包围在一对单引号中的字符构成。因此，'a'、';'和'0'都是字符常量的有效例子。第一个常量表示字母 a；第二个常量是一个分号；第三个是字符 0，它与数字 0 不同。不要将字符常量与字符串相混淆，前者是放在单引号中的单个字符，后者是放在双引号中的任意个字符。

字符常量'\n'（换行符）尽管看起来似乎与前面提到的规则相抵触，但它确实是一个有效的字符常量。这是因为反斜线字符在 C 系统中是一个特殊字符，不能真正算作一个字符。换句话说，尽管'\n'实际上由两个字符组成，但 C 编译器将它看作单个字符。还有其他一些以反斜线字符开头的特殊字符。其完整列表请参阅附录 A。

在 printf()调用中使用格式字符%c，可以在终端上显示一个 char 变量的值。

---

① 附录 A 讨论了利用特殊转义序列、通用字符和宽字符来表示扩展字符集中各个字符的方法。

火龙果•整理
uml.org.cn

布尔数据类型_Bool

根据 C 语言中的定义，_Bool 变量的大小仅足以存储数值 0 和 1。但对它所使用的准确内存数量未做规定。_Bool 变量用在需要表示布尔条件的程序中。例如，这种类型的变量可用于表示是否已经从文件中读取了所有数据。

根据惯例，0 用于表示一个假值，1 用于表示一个真值。在为一个_Bool 变量赋值时，数值 0 被存储为变量中的 0，而任何非零值都被存储为 1。

为了简化_Bool 变量在程序中的使用，标准头文件<stdbool.h>中定义了值 bool、true 和 false。第 5 章的程序 5.10A 中给出了它的一个例子。

程序 3.1 中使用了 C 语言的基本数据类型。

程序 3.1　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 3.1　输出

```
integerVar = 100
floatingVar = 331.790009
doubleVar = 8.440000e+11
doubleVar = 8.44e+11
charVar = W
boolVar = 0
```

火龙果•整理
uml.org.cn

程序 3.1 的第一条语句将变量 integerVar 声明为一个整型变量，还为它赋了初始值 100，就像是使用了下面两条语句：

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

在程序输出的第二行中，请注意，被赋给 floatingVar 的数值 331.79 实际上被显示为 331.790009。事实上，实际显示的数值取决于正在使用的具体计算机系统。这种不精确现象的出现，是由计算机内部对数字的特定表示方式造成的。当我们在袖珍计算器上处理数字时，可能已经遇到过相同类型的不精确现象。如果在计算器上用 1 除以 3，得到的结果为 0.33333333，3 的个数也许会更多一些。这个由 3 组成的字符串是计算器对三分之一的近似表示。理论上，应当有无穷多个 3。但计算器只能容纳这么多位，从而造成计算器固有的不精确性。相同类型的不精确现象也出现在这里：某些浮点值无法在计算机内存中准确表示。

在显示 float 或 double 变量的数值时，可以选择三种不同格式。%f 字符用于以标准方式显示数值。除非另行告知，否则 printf()总是将 float 或 double 值显示为 6 位小数，之后的部分将采用四舍五入法处理。在本章后面将会看到如何选择要显示的小数位数。

%e 字符用于以科学计数法显示 float 或 double 的值。同样，系统会自动显示 6 位小数。

在使用%g 字符时，printf()在%f 和%e 之间进行选择，还会自动清除显示内容最后的所有 0。如果小数点之后没有数位，小数点也不会显示。

在倒数第二个 printf()语句中，%c 字符用于显示单个字符'W'，这个字符在声明变量 charVar 时被赋给了该变量。请记住，字符串（比如 printf()的第一个参数）放在一对双引号内部，字符常量必须总是放在一对单引号内部。

最后的 printf()表明，_Bool 变量的取值可以用整数格式字符%i 显示。

类型说明符：long、long long、short、unsigned 和 signed

如果直接将说明符 long 放在 int 声明之前，所声明的整型变量将拥有某些计算机系统上的扩展范围。long int 声明的一个示例如下：

```
long int factorial;
```

火龙果•整理
uml.org.cn

这个语句将变量 factorial 声明为一个 long 整型变量。与 float 和 double 一样，long 变量的具体精度也取决于具体的计算机系统。在许多系统上，int 和 long int 的取值范围相同，都可用于存储最高 32 位宽（$2^{31}-1$ 或 2147483647）的整数值。

根据需要在一个整数常量的最后追加一个字母 L（大小写均可），可以构成一个 long int 类型的值。数字与 L 之间不允许存在空格。因此，声明语句

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

将变量 numberOfPoints 声明为 long int 类型，其初始值为 131071100。

要使用 printf()显示一个 long int 值，可以在整数格式字符 i、o 和 x 之前以字母 l 作为修饰符。这就是说，可以使用格式字符%li 以十进制格式显示一个 long int 值，字符%lo 可以八进制格式显示该数值，而字符%lx 可以十六进制格式显示该数值。

还有一种 long long 整数数据类型，因此，

```
long long int maxAllowedStorage;
```

将所示变量声明为指定的扩展精度，可以确保这一精度至少宽 64 位。为了显示 long long 整数，printf 字符串中使用的不再是单个字母 l，而是两个 l，比如"%lli"。

long 说明符还可以出现在 double 声明的前面，如下所示：

```
long double US_deficit_2004;
```

long double 常量被写为一个后面紧跟着字母 l 或 L 的浮点常量，比如

```
1.234e+7L
```

要显示一个 long double 值，可使用 L 修饰符。因此，%Lf 以浮点表示法显示一个 long double 值，%Le 以科学记数法显示同一数值，%Lg 告诉 printf()在%Lf 和%Le 之间进行选择。

当 int 声明的前面放有说明符 short 时，它告诉 C 编译器，被声明的变量用于存储相当小的整数值。使用 short 变量的动机主要是为了节省内存空间，当程序需要大量内存，而可用内存数量又有限时，内存空间可能会成为一个问题。

火龙果•整理
uml.org.cn

在某些计算机上，short int 占用的存储数量是普通 int 变量的一半。在任意情况下，都可以保证为 short int 分配的空间数量不小于 16 位。

在 C 语言中，没有什么办法可以明确写出一个 short int 类型的常量。要显示一个 short int 变量，可以将字母 h 放在任何普通整数转换字符的前面：%hi、%ho 或%hx。或者，也可以使用任意整数转换字符来显示 short int，这是因为在将 short int 作为参数传送给 printf()例程时，可以将它们转换为整数。

当一个整型变量仅用于存储正数时，可以用 unsigned 说明符放在 int 变量的前面。声明

```
unsigned int counter;
```

向编译器声明，变量 counter 仅用于存储正数值。将一个整型变量的应用范围限制在正整数的独占存储中，可以扩展此整型变量的精度。

在常量之后放置字母 u（或 U），即可构成 unsigned int，如下所示：

```
0x00ffU
```

在写出一个整数常量时，可以结合使用字母 u（或 U）和 l（或 L），因此，

```
20000UL
```

是告诉编译器，将常量 20000 看作一个 unsigned long。

如果一个整数常量的后面没有跟着字母 u、U、l 或 L 中的任何一个，而且又大得无法装入普通大小的 int 中，编译器就会将它看作 unsigned int。如果它因为太大（译者注：原书此处为“小”，疑有误）而无法装入 unsigned int 中，编译器就将它看作 long int。如果它还不能装在 long int 中，编译器就将它当作 unsigned long int。如果不能装入 unsigned long int 中，但能装入 long long int 中，编译器就将它看作 long long int，否则就看作 unsigned long long int。

在将变量声明为 long long int、long int、short int 或 unsigned int 类型时，可以省略关键字 int。因此，unsigned 变量 counter 同样可以声明如下：

```
unsigned counter;
```

火龙果•整理
uml.org.cn

char 变量也可以声明为 unsigned 的。

可以使用 signed 限定符明确地告诉编译器，一个特定变量是有符号量。它主要用在 char 声明的前面。深入讨论将推迟到第 13 章进行。

如果关于这些说明符的讨论现在对你来说显得有些深奥，也不用担心。在本书的后续章节中，许多不同类型都会在实际程序范例中进行演示。第 13 章更详细地讨论了数据类型及其转换。

表 3.1 汇总了基本数据类型和限定符。

表 3.1　基本数据类型

类　　型	常量举例	printf 字符
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 , 500ll	%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	2.34 , 3.1e-5, 0x.1p3	%f , %e , %g, %a
long double	12.341 , 3.1e-5l	%Lf , %Le , %Lg

使用变量

早期计算机程序员的任务非常繁重，要对哪种计算机实现程序控制，就得使用这种计算机的二进制语言编写程序。这就意味着必须由程序员将计算机指令以人工方式转换为二进制数字，然后再将其输入计算机中。另外，程序员还必须通过具体数字或者内存地址，对计算机内存中的存储位置进行显式赋值和引用。

现在的程序设计语言允许程序员将更多精力放在解决手边的具体问题上，而不用再为特定机器的代码或内存位置费心。这些语言允许为符号名称（称为

火龙果•整理
uml.org.cn

变量名）赋值，以存储程序计算结果。可以选择一个有意义的变量名，反映出要在该变量中存储的值的类型。

第 2 章曾经使用过几个变量来存储整数值。例如，在程序 2.4 中使用变量 sum 来存储将两个整数 50 和 25 相加的结果。

C 语言还允许将整数之外的其他数据类型存储在变量中，但在程序中使用变量**之前**，需要先给出正确声明。变量可用于存储浮点数、字符，甚至可以存储指向计算机内存位置的**指针**。

变量名的构成规则非常简单：必须以字符或下划线（_）开头，之后可以跟有字母（大写或小写）、下划线或数字 0～9 的任意组合。下面是有效变量名的一份列表：

```
sum
pieceFlag
i
J5x7
Number_of_moves
sysflag
```

而下面的变量名则是无效的，原因如右：

```
sum$value       $不是有效字符。
piece flag      不允许存在嵌入空格。
3Spencer        变量名不能以数字开头。
int             int 是保留字。
```

int 不能用作变量名，因为它的应用对于 C 编译器有着特殊意义。这一用法称为保留名字或保留字。一般情况下，任何对 C 编译器具有特殊意义的名字都不能用作变量名。附录 A 给出了这些保留名字的一个完整列表。

一定要记住，C 语言中的大小写字母是有区别的。因此，变量名 sum、Sum 和 SUM 各是一个不同变量。

变量名的长度不限，但只有前 63 个字符是有效的，在某些情况下（如附录 A 所述），只有前 31 个字符是有效的。太长的变量名是不实用的，原因很简单：必须额外完成大量键入工作。例如，下面的代码行是有效的：

```
theAmountOfMoneyWeMadeThisYear = TheAmountOfMoneyLeftAttheEndOfTheYear -
          theAmountOfMoneyAtTheStartOfTheYear;
```

而代码行

火龙果•整理
uml.org.cn


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

传达的信息与上面几乎一样多，但使用的空间却要少得多。

在选择变量名时，请记住一条建议——不要偷懒。一定要选取能反映变量用途的名字。原因很明显：和注释一样，有意义的变量名可以大幅提高程序的可读性，对调试和文档编写阶段有利。事实上，由于程序变得清晰易懂、不言自明，所以有可能大幅减少文档编写任务。

使用算术表达式

在 C 语言中，和几乎所有程序设计语言中一样，加号（+）用于将两个数值相加，减号（-）用于将两个数值相减，星号（*）用于将两个数值相乘，斜线（/）用于将两个数值相除。这些运算符称为**二元**算术运算符，这是因为它们是对两个值或两项进行运算的。

我们已经看到如何在 C 语言中执行诸如加法之类的简单运算。程序 3.2 进一步演示了减、乘、除运算。这个程序中执行的最后两个运算介绍了一种概念：一个运算符的**优先级**可能优于另一个运算符。事实上，C 语言中的每个运算符都有一个相关联的优先级。这个优先级用于判断如何对拥有多个运算符的表达式进行求值：首先处理具有较高优先级的运算符。如果表达式中的运算符具有相同的优先级，将根据具体运算符，自左向右或自右向左进行求值。这一性质称为运算符的**结合**性。附录 A 中给出了运算符优先级及其结合规则的完整列表。

程序 3.2　使用算术运算符

```
// 演示各种算术运算符的使用

#include <stdio.h>

int main (void)
{
    int a = 100;
    int b = 2;
    int c = 25;
    int d = 4;
    int result;

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

火龙果•整理
uml.org.cn


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

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

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

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

    return 0;
}
```

程序 3.2　输出

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

在声明了整型变量 a、b、c、d 和 result 之后，程序将 a 减 b 的结果赋值给 result，然后用一个适当的 printf() 调用显示 result 的值。

下一条语句

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

的效果是将 b 的值乘以 c 的值，并将乘积存储在 result 中。然后用一个 printf() 调用显示乘法的结果，现在对于这个调用你应当已经很熟悉了。

下一条程序语句介绍除法运算符——斜线。在 a 除以 c 的运算之后紧跟着 printf() 语句，将显示 100 除以 25 所得的结果 4。

在某些计算机系统中，试图将一个数字除以 0 时，会造成程序异常终止。[①] 即使程序没有异常终止，这一除法得到的结果也是没有意义的。

在第 5 章将会看到，可以在执行除法运算之前先来检查“被零除”。如果确定除数为 0，可以采取适当操作，避开除法运算。

① 使用 Windows 中的 gcc 编译器时就会出现这一情况。在 Unix 系统中，程序不会异常终止，在以整数除以 0 时，会给出一个 0 值作为结果，而在以浮点数除以 0 时，会给出“无穷大”作为结果。

火龙果•整理
uml.org.cn

表达式

```
a + b * c
```

不会给出结果 2550（102 × 25）；而相应 printf() 语句显示的结果是 150。这是因为 C 语言和其他大多数程序设计语言一样，对于表达式中多个运算或多个项目的求值顺序制定了相关规则。表达式的求值顺序通常是自左向右进行。但是，乘法和除法运算的优先级要高于加法和减法运算。因此，表达式

```
a + b * c
```

会被 C 程序设计语言当作

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

进行计算。（这种表达式求值顺序与应用基本代数规则时的顺序相同。）

如果希望改变表达式中各个项目的求值顺序，可以使用括号。事实上，前面给出的表达式是一个完全有效的 C 表达式。因此，在程序 3.2 中代入语句

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

完全可以得到相同的结果。但是，如果使用表达式

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

那赋给 result 的值将是 2250，这是因为首先会将 a 的值（100）与 b 的值（2）相加，然后才会乘以 c 的值（25）。括号还可以嵌套，在这种情况下，表达式的求值顺序是从最内层括号开始向外进行。但一定要确认，左右括号的个数要相等。

由程序 3.2 的最后一条 printf() 语句可以注意到，将一个表达式作为 printf() 的参数是完全有效的，并不需要先将表达式的求值结果赋值给一个变量。根据前述规则，表达式

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

的求值顺序就是

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

也就是

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

计算结果 300 被传送给 printf() 例程。

火龙果•整理
uml.org.cn

整数算术及一元减运算符

程序 3.3 巩固了我们刚刚学到的知识，并介绍了整数算术的概念。

程序 3.3　关于算术运算符的更多例子

```
// 更多算术表达式

#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 3.3　输出

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

在前四条语句中，int 与 a、b、c、d 的声明之间插入了多余的空格，目的是让每个变量的声明对齐。这样有助于提高程序的可读性。你可能还注意到，在目前给出的所有程序中，每个运算符的前后都有一个空格。这种做法也不是必需的，只是出于美观考虑。一般情况下，在允许使用单个空格的地方都可以加入更多空格。如果最终得到的程序更容易读懂，那多按几次空格键也是值得的。

程序 3.3 第一个 printf() 调用中的表达式巩固了运算符优先级的概念。这一表达式的求值过程如下：

1. 因为除法的优先级高于加法，所以首先将 a 的值（25）除以 5，得到中间结果 5。

火龙果•整理
uml.org.cn

2．因为乘法的优先级也高于加法，所以接下来将中间结果 5 乘以 b 的值（2），得出一个新的中间结果 10。

3．最后，将 6 与 10 相加，得到最终结果 16。

第二个 printf()语句介绍了另外一种常被曲解的现象。你可能以为，将 a 除以 b 后再乘以 b，会返回 a 的值，它被设定为 25。但事实似乎并非如此，显示结果竟然是 24。似乎计算机在计算过程中的某个位置犯错了。事实是，这个表达式是使用整数算术进行求值的。

如果回头看一下变量 a 和 b 的声明，会发现它们都被声明为 int 类型。只要表达式中某个等待求值的项目是由两个整数组成的，C 系统就会使用整数算术进行运算。在这种情况下，数字的所有小数部分都会丢失。因此，当 a 的值除以 b 的值，也就是 25 除以 2 时，将会得到中间结果 12，而**不是**预期的 12.5。将这个中间结果乘以 2，会得到最终结果 24，这就解释了“丢失”的数位。别忘了，如果对两个整数相除，总会得到整数结果。另外，也请记住，不会发生四舍五入，小数部分将直接被删除，因此，准确结果为 12.01、12.5 或 12.99 的所有整数除法最终都会得到同一个值——12。

从程序 3.3 中的倒数第二个 printf()语句可以看出，如果使用浮点数代替整数进行相同运算，将会得到预期结果。

到底应当使用 float 变量还是使用 int 变量，应当根据变量的预期用途进行选择。如果不需要任何小数位，那就使用整型变量，这样可以得到更高效的程序，也就是说，它在许多计算机上的执行速度会更快一些。但另一方面，如果需要准确的小数位，那选择也就很明确了。于是只有一个必须回答的问题了：是应当使用 float、double，还是 long double？这个问题的回答取决于所处理数字的预期精度及其大小。

在最后一条 printf()语句中，使用了一元减运算符对变量 a 的值求相反数。**一元**运算符是对单个数值执行运算的运算符，它不同于对两个数值执行运算的二元运算符。这个减号实际上具有双重角色：作为二元运算符，用于对两个数值相减；作为一元运算符，用于求一个数值的相反数。

一元减运算符的优先级高于所有其他算术运算符，但一元加运算符（+）除外，它与一元减运算符具有相同的优先级。因此，表达式

```
c = -a * b;
```

火龙果•整理
uml.org.cn

的结果将得到-a 与 b 的乘积。再次提醒，可以在附录 A 中找到一个总结各种运算符及其优先级的表格。

求模运算符

有一个极有价值的运算符，你可能还没有遇到过，它就是求模运算符，用百分号（%）表示。请通过分析程序 3.4 来判断这一运算符是如何工作的。

程序 3.4　演示求模运算符

```
// 求模运算符

#include <stdio.h>

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

    printf ("a = %i, b = %i, c = %i, and d = %i\n", a, b, c, d);
    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 3.4　输出

```
a = 25, b = 5, c = 10, and d = 7
a % b = 0
a % c = 5
a % d = 4
a / d * d + a % d = 25
```

main()中的第一条语句定义并初始化了变量 a、b、c 和 d，所有这些工作都在一条语句中完成。

为进行提示，在输出一系列使用求模运算符的语句之前，第一个 printf()语句首先输出了程序中使用的四个变量的值。这一步骤并非至关重要的，但它是一个很好的提示，有助于别人理解你的程序。对于其余的 printf()代码，如你所知，printf()利用紧跟在百分号之后的字符来判断如何打印下一个参数。但是，如果它后面跟有另一个百分号，那 printf()就将其看作一种指示，

火龙果•整理
uml.org.cn

表明你真的希望显示一个百分号，并在程序输出中的适当位置插入一个。

如果你得出结论，求模运算符%的功能就是给出第一个值除以第二个值时的余数，那就没错。在第一个例子中，25 除以 5 所得的余数显示为 0。如果将 25 除以 10，将得到余数 5，第三行输出验证了这一点。将 25 除以 7 得到余数 4，如第四个输出行所示。

程序 3.4 中的最后一个输出行需要做一点解释。首先，你会注意到这个程序语句被写在两行。这在 C 语言中是完全有效的。事实上，一条程序语句可以持续到下一行中任何一个允许使用空白区域的位置（在处理字符串时例外，字符串将在第 9 章进行讨论）。有些时候，将一条程序语句持续到下一行不只是合乎我们的心意，可能还是必不可少的。程序 3.4 中对 `printf()` 调用的持续部分进行了缩进，以便在视觉上表明它是上一条程序语句的持续。

现在来关注在最后一条语句中进行求值的表达式。你会想起，在 C 语言中，任何两个整数值之间的运算都是以整数算术执行的。因此，两个整数值相除时得到的余数将直接被丢弃。用表达式 `a / d` 表示的 25 除以 7，将会给出中间结果 3。将这个值乘以 `d` 的取值 7，会得出中间结果 21。最后，加上 `a` 除以 `d` 的余数（用表达式 `a % d` 表示），得出最终结果 25。这个值与变量 `a` 的值相同，这并不是巧合。一般情况下，表达式

```
a / b * b + a % b
```

的输出总是等于 `a` 的值，当然，得假定 `a` 和 `b` 都是整数值。事实上，求模运算符%就被定义为仅处理整数值。

说到优先级，求模运算符与乘除法运算符具有相同的优先级。这当然意味着，（比如，）表达式

```
table + value % TABLE_SIZE
```

的求值顺序为

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

整数与浮点转换

为了高效开发 C 程序，必须理解 C 语言中关于浮点值与整数值之间进行隐式转换的规则。程序 3.5 演示了数值类型之间的一些简单转换。应当注意，一些编译器可能会给出警告信息，提醒正在执行转换。

火龙果•整理
uml.org.cn

程序 3.5　整数与浮点数之间的转换

```
// C语言中的基本转换

#include <stdio.h>

int main (void)
{
     float  f1 = 123.125, f2;
     int   i1, i2 = -150;
     char     c = 'a';

     i1 = f1;              // 浮点数转换为整数
     printf ("%f assigned to an int produces %i\n", f1, i1);

     f1 = i2;              // 整数转换为浮点数
     printf ("%i assigned to a float produces %f\n", i2, f1);

     f1 = i2 / 100;        // 整数除以整数
     printf ("%i divided by 100 produces %f\n", i2, f1);

     f2 = i2 / 100.0;            // 整数除以浮点数
     printf ("%i divided by 100.0 produces %f\n", i2, f2);

     f2 = (float) i2 / 100;    // 类型转换运算符
     printf ("(float) %i divided by 100 produces %f\n", i2, f2);

     return 0;
}
```

程序 3.5　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，只要将一个浮点数赋值给一个整型变量，浮点数的小数部分就会被截断。因此，在上面的例子中，当 `f1` 的值被赋给 `i1` 时，数值 123.125 被**截断**，也就是说，只有它的整数部分 123 被存储在 `i1` 中。程序输出的第一行证实了确实如此。

将一个整型变量赋值给浮点变量不会导致数值的任何变化；该数值将被系统直接转换，并存储在浮点变量中。程序输出的第二行证实，`i2` 的值(–150)被

火龙果•整理
uml.org.cn

正确转换，并存储在 float 变量 f1 中。

程序的随后两行输出说明了在组成算术表达式时必须记住的两点。第一点与本章前面讨论的整数算术有关。只要一个表达式的两个操作数是整数（同样适用于 short、unsigned、long 和 long long），该运算将根据整数算术法则执行。因此，除法运算得到的所有小数部分都会被丢弃，即使像程序 3.5 中所做的那样，将结果赋值给一个浮点变量时也是如此。因此，当整型变量 i2 除以整数常量 100 时，系统将该除法作为整数除法执行。因此，在 float 变量 f1 中存储的数值就是将–150 除以 100 的结果，也就是–1。

下一个除法涉及一个整型变量和一个浮点常量。在 C 语言中，如果两个数值中的任意一个为浮点变量或常数，那么这两个数值之间的所有运算都将作为浮点运算执行。因此，当 i2 的值除以 100.0 时，系统将此除法当作浮点除法，得出结果–1.5，然后将它赋给 float 变量 f1。

类型转换运算符

程序 3.5 中的最后一个除法运算是

```
f2 = (float) i2 / 100;    // 类型转换运算符
```

它介绍了类型转换运算符。类型转换运算符的效果就是将变量 i2 的值转换为 float 类型，以进行表达式求值。这个运算符绝对不会永久影响变量 i2 的值；它是一个一元运算符，其表现特性与其他一元运算符类似。因为表达式-a 对于 a 的值没有永久影响，所以表达式(float) a 也是如此。

类型转换运算符的优先级高于除一元减和一元加之外的所有其他算术运算符。当然，在必要时，总是可以在表达式中使用括号，强制根据所需顺序进行各项求值。

作为类型转换运算符的另一个使用范例，表达式

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

在 C 语言中被当作

```
29 + 21
```

进行计算，这是因为将一个浮点数转换为一个整数的效果就是截断该浮点值。表达式

火龙果•整理
uml.org.cn


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

生成结果 1.5，与下面的表达式相同：

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

运算与赋值相结合：赋值运算符

C 语言允许将算术运算符与赋值运算符结合起来，其一般格式为：*op*=

在这一格式中，*op* 是任意算术运算符，包括+、-、×、/和%。此外，*op* 还可以是任何用于移动和掩码的位运算符，这些内容将在后面讨论。

考虑下面的语句：

```
count += 10;
```

所谓的“加等”运算符+=的效果是将其右侧的表达式与左侧的表达式相加，并将结果存在左侧的变量中。因此，上面的语句等价于如下语句：

```
count = count + 10;
```

表达式

```
counter -= 5
```

使用“减等”赋值运算符来从 counter 的值中减去 5，等价于下面的表达式：

```
counter = counter - 5
```

一个稍微复杂一点的表达式是

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

它将 a 除以等号右边出现的任意内容（也就是 b 与 c 的和），并将结果存储在 a 中。这里的加法会首先执行，因为加法运算符的优先级高于赋值运算符。事实上，除逗号运算符之外，所有运算符的优先级都高于赋值运算符，所有的赋值运算符都具有相同的优先级。

在本例中，这个表达式等同于如下表达式：

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

使用赋值运算符有三重动机。第一，程序语句更容易书写，因为出现在运算符左侧的内容不必再在其右侧重复。第二，所得到的表达式通常也更容易阅

火龙果•整理
uml.org.cn

读。第三，使用这些运算符可以提高程序的运行速度，因为编译器有时只须生成较少的代码即可完成表达式的求值。

_Complex 和_Imaginary 类型

在结束本章之前，还值得注意下 C 语言中的另外两种类型，它们的名字为_Complex 和_Imaginary，用于处理复数和虚数。

从 C99 开始，对_Complex 和_Imaginary 类型的支持就已经成为 ANSI C 标准的组成部分，不过 C11 的确将这一支持作为了可选项。要知道你的编译器是否支持这些类型，最好的方法就是查看附录 A 中的数据类型汇总。

练习

1. 键入并运行本章给出的 5 个程序。将每个程序产生的输出与书中每个程序之后给出的输出进行对比。
2. 以下哪些是无效变量名？为什么？

```
int             char      6_05
Calloc          Xx        alpha_beta_routine
floating        _1312     z
ReInitialize    _         A$
```

3. 以下哪些是无效常量？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. 编写一个程序，使用如下公式将 27° 由华氏度（*F*）转换为摄氏度（*C*）：

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

5. 你认为以下程序会给出什么样的输出？

```
#include <stdio.h>

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

火龙果•整理
uml.org.cn


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

    return 0;
}
```

6. 编写一个程序，对如下所示的多项式求值（$x = 2.55$）：

$3x^3 - 5x^2 + 6$

7. 编写一个程序，对以下表达式求值，并显示结果（记住要用指数格式显示结果）：

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

8. 要将整数 i 舍入为另一整数 j 的下一个整倍数，可以使用以下公式：

```
Next_multiple = i + j - i % j
```

例如，要将 256 天舍入为一周 7 天的下一个整倍数，可以将 i = 256 和 j = 7 两个值代入上面的公式，如下所示：

```
Next_multiple = 256 + 7 - 256 % 7
              = 256 + 7 - 4
              = 259
```

编写一个程序，针对以下所列的 i、j 值，找出 j 的下一个整倍数：

```
i          j
365        7
12,258     23
996        4
```

火龙果•整理
uml.org.cn

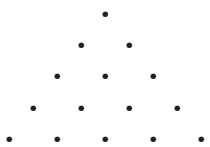
4
程序循环

计算机最强大的功能之一就是能够执行重复计算。当需要重复使用相同代码时，C 语言提供了几种构造，专门设计用来处理这些情景。这一章将帮助读者理解这些工具，包括：

- `for` 语句
- `while` 语句
- `do` 语句
- `break` 语句
- `continue` 语句

三角数

如果将 15 个点排成一个三角形，最终可能会得到一个类似于如下所示的排列方式：



这个三角形的第一行包含一个点，第二行包含两个点，以此类推。一般地，要构成一个包含 *n* 行的三角形，所需要的点数等于整数 1 至 *n* 之和。这个和值称为**三角数**。如果从 1 开始，则第 4 个三角数就是 1 至 4 之间的连续整数之和（1+2+3+4），即 10。

假定你要编写一个程序，计算第 8 个三角数的值，并在终端上显示它。显然，你可以很轻松地在大脑中计算这个数字，但为进行讨论，假定你要用 C 语

火龙果•整理
uml.org.cn

言编写一个程序来完成这一任务。程序 4.1 就是这样一个程序。

程序 4.1　计算第 8 个三角数

```
// 此程序计算第 8 个三角数

#include <stdio.h>

int main ()
{
      int  triangularNumber;

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

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

      return 0;
}
```

程序 4.1　输出

```
The eighth triangular number is 36
```

在计算相对较小的三角数时，程序 4.1 中的方法完全可以胜任。但如果需要找出第 200 个三角数的取值时，又会怎么样呢？要修改程序 4.1，使其明确地对 1 到 200 的所有整数进行求和，那当然是件很乏味的事情。幸运的是，还有一种更简单的方法。

一台计算机的基本性质之一就是它能够重复执行一组语句。利用这些**循环**功能，可以开发出包含重复过程的简洁程序，否则，可能需要执行数千条甚至数百万条的程序语句。C 程序设计语言中包含三种不同的程序语句用于实现程序循环。它们是 `for` 语句、`while` 语句和 `do` 语句。本章将详细介绍所有这些语句。

for 语句

现在就让我们开始研究一个使用 `for` 语句的程序。程序 4.2 的目的是计算第 200 个三角数。看看你是否能够确定 `for` 语句是如何工作的。

火龙果•整理
uml.org.cn

程序 4.2　计算第 200 个三角数

```
/* 此程序计算第 200 个三角数
   介绍 for 语句              */

#include <stdio.h>

int main (void)
{
    int  n, triangularNumber;

    triangularNumber = 0;

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

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

    return 0;
}
```

程序 4.2　输出

```
The 200th triangular number is 20100
```

这里需要对程序 4.2 做一点解释。用于计算第 200 个三角数的方法实际上与程序 4.1 中计算第 8 个三角数的方法相同，就是对 1 到 200 的整数求和。for 语句提供了一种机制，可以避免明确写出 1 到 200 之间的每个整数。在某种意义上，这个语句就是用来为你“生成”这些数值的。

for 语句的一般格式如下：

```
for (初始表达式; 循环条件; 循环表达式)
   程序语句
```

放在括号里面的三个表达式（“*初始表达式*”、“*循环条件*”和“*循环表达式*”）为程序循环搭建了环境。紧跟其后的程序语句（当然要以分号结尾）可以是任何有效 C 程序语句，它们构成了循环体。这个语句的执行次数由 for 语句设置的参数规定。

for 语句的第一个部分为“*初始表达式*”，用于在循环开始**之前**设定初始值。在程序 4.2 中，for 语句的这一部分用于将 n 的初始值设定为 1。可以看出，赋值是一种有效的表达式形式。

火龙果•整理
uml.org.cn

for 语句的第二部分是**使**循环得以继续执行所需要满足的条件。换句话说，**只要**满足这一条件，循环将一直持续下去。再次参考程序 4.2，注意到 for 语句的“循环条件”由下面的**关系表达式**规定：

```
n <= 200
```

这个表达式可以读作“n 小于或等于 200”。“小于或等于”运算符（它是小于号<后面紧跟着等号=）只是 C 程序设计语言中提供的几种关系运算符之一。这些运算符用于测试指定条件。如果满足条件，则测试的答案为“是”，用更常见的说法就是“真”（TRUE）；如果不满足条件，就是“否”或“假”（FALSE）。

关系运算符

表 4.1 中列出了 C 语言中的所有可用关系运算符。

表 4.1　关系运算符

运 算 符	含　义	举　例
==	等于	count == 10
!=	不等于	flag != DONE
<	小于	a < b
<=	小于或等于	low <= high
>	大于	point > endOfList
>=	大于或等于	j >= 0

关系运算符的优先级低于所有算术运算符。例如，这意味着下面的表达式

```
a < b + c
```

将会按

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

求值，与预期中一样。如果 a 的值小于 b + c 的值，则结果为 TRUE，否则为 FALSE。

特别要注意“等于”运算符==，不要将它与赋值运算符=混淆。表达式

```
a == 2
```

检查 a 的值是否等于 2，而表达式

```
a = 2
```

火龙果•整理
uml.org.cn

是将数值 2 赋给变量 a。

选择使用哪个关系运算符，显然取决于要进行的具体测试，在某些情况下，取决于程序员本人的偏好。例如在程序 4.2 中，关系表达式

```
n <= 200
```

可以等价表示为

```
n < 201
```

只要这个关系测试的结果为 TRUE，在本例中，就是只要 n 的值小于或等于 200，就会重复执行构成 for 循环体的程序语句

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

这个程序语句的效果是将 triangularNumber 的值与 n 的值相加，再将结果存入 triangularNumber。

当“*循环条件*”不再满足时，程序将继续执行紧跟在 for 循环之后的程序语句。在程序 4.2 中，在终止循环后，将继续执行 printf 语句。

for 语句的最后部分包含一个表达式，在每次执行循环体**之后**都会对其求值。在程序 4.2 中，“*循环表达式*”将 1 与 n 的值相加。因此，在每次将 n 的值与 triangularNumber 的值相加之后，都会将 n 的值递增 1，其取值范围为 1 到 201。

值得注意的是，n 达到的最后一个值，也就是 201，**不会**被与 triangularNumber 的值相加，因为**只要**循环条件不再满足，也就是说只要 n 等于 201，循环将立即终止。

总结一下，for 语句的执行过程如下：

1. 首先对初始表达式求值。这个表达式通常会为一个在循环内部使用的变量设定初始值，比如 0 或 1 等，这个变量一般称为**索引**变量。
2. 评估循环条件。如果不满足该条件（表达式为 FALSE），则循环立即终止。继续执行紧跟在循环之后的程序语句。
3. （如果满足循环条件，）执行构成循环体的程序语句。
4. 对循环表达式求值。这一表达式一般用于改变索引变量的值，通常是使它增加一个递增变量或者减去一个递减变量。
5. 返回步骤 2。

火龙果•整理
uml.org.cn

回想一下，循环条件是在进入循环时立即评估的，这时还从来没有执行过循环体。还要记住，不要在循环末尾的右括号之后放置分号（如果放置的话，会立即终止循环）。

因为程序 4.2 实际上生成了直到其最终目标的所有前 200 个三角数，所以生成这些数字的一个表格应当是不错的。但为了节省空间，假定我们只希望打印一个包含前 10 个三角数的表格。程序 4.3 准确地执行了这一任务！

程序 4.3　生成一个三角数表

```
// 此程序生成一个三角数表

#include <stdio.h>

int main (void)
{
    int  n, triangularNumber;

    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 4.3　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
```

火龙果•整理
uml.org.cn


```
9              45
10              55
```

向程序中额外增加一些 printf()语句，使输出更有意义，这个主意总是很不错的。在程序 4.3 中，前三个 printf()语句的目的就是提供一个通用标题，并标记输出的各列。注意，第一个 printf()语句中包含两个换行符。可以想到，其效果不仅是到达了下一行，还会在显示中额外插入一个空行。

在显示了适当的标题之后，程序接下来计算前 10 个三角数。在计算“从 1 到 n 的总和”时，变量 n 用于计算当前的个数，而变量 triangularNumber 用于存储三角数 n 的值。

将变量 n 的值设定为 1，开始执行 for 语句。回想一下，紧跟在 for 语句之后的程序语句构成了程序的循环体。但如果希望重复执行的不止一条程序语句，而是一组程序语句，又该怎么办呢？将所有这些程序语句放在一对大括号中就能做到这一点。这样，系统就会将这一组语句（或说语句**块**）看作一个实体。一般情况下，在 C 程序中任何可以放置单条语句的地方，都可以使用语句块，只要记住把语句块放在一对大括号里面即可。

因此，在程序 4.3 中，程序的循环体包括：将 n 加至 triangularNumber 数值的表达式，以及紧随其后的 printf()语句。请特别留意这些程序语句的缩进。很容易就能判断哪些语句是 for 循环的组成部分。还应当注意到，程序员会使用不同的编码风格。有些人喜欢像下面这样键入循环：

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

这里的左大括号放在 for 之后的下一行。这绝对是个人喜好问题，对程序没有任何影响。

只需要将 n 的值与前一个三角数相加就可以算出下一个三角数。这一次使用了“加等”运算符，在第 3 章已经介绍了这一运算符。回想一下，表达式

```
triangularNumber += n;
```

等价于表达式

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

火龙果•整理
uml.org.cn

在第一次执行 for 循环时，“前一个”三角数为 0，因此，n 等于 1 时，triangularNumber 的新值就是 n 的值，也就是 1。随后显示 n 和 triangularNumber 的值，在格式字符串中插入了适当数量的空格，以确保两个变量的值在适当的列标题之下对齐。

因为现在已经执行了循环体，所以接下来将执行循环表达式。不过，这个 for 语句中的表达式有点奇怪。似乎是出现了印刷错误，本来是要插入表达式

```
n = n + 1
```

而不是插入一个看起来有点可笑的表达式

```
++n
```

表达式++n 实际上是一个完全有效的 C 表达式。它介绍了 C 程序设计语言中一个新的（而且是相当独特的）运算符——**递增运算符**。这个“双加号”符号（或者说递增运算符）的功能是向其操作数加 1。因为“加 1”操作在程序中太常见了，所以专门为它创建了一种特殊运算符。因此，表达式++n 等价于表达式 n = n + 1。尽管 n = n + 1 看起来似乎更容易理解，但你很快就会熟悉这一运算符的功能，甚至会试着欣赏它的简洁。

当然，如果有任何一门语言仅提供了加 1 的递增运算符，却没有提供减 1 的相应运算符，那这种语言肯定是不完整的。这种减 1 的运算符叫作**递减运算符**，用双减号表示。因此，C 语言中的表达式

```
bean_counter = bean_counter - 1
```

可以用递减运算符表示为

```
--bean_counter
```

其效果是等价的。有些程序员喜欢将++或--放在变量名的后面，比如 n++或 bean_counter--。对于 for 语句的这个例子而言，这完全是个人喜好。但在第 10 章你将会了解到，在更复杂的表达式中，这个运算符放在变量的前面和后面的确是有区别的。

输出的对齐

你可能已经注意到，在程序 4.3 的输出中有一点稍微不妥的现象：第 10 个三角数没有与前面的三角数对齐。这是因为数字 10 占用了两个打印位置，而前

火龙果•整理
uml.org.cn

面的 n 值（1～9）仅占用一个打印位置。因此，数值 55 就被推后一个位置。这一点小烦恼是可以解决的，只要用下面的 printf()语句代替程序 4.3 中的相应语句即可：

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

为了验证这一修改的确是有效的，下面给出已修改程序的输出（将此程序称为程序 4.3A）。

程序 4.3A　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
```

对 printf()语句做出的主要修改就是包含了一个**字段宽度规格**。字符%2i 告诉 printf()例程，你不仅希望在特定位置显示一个整数的值，还希望待显示整数的大小占用显示位置的两列。通常占用位置少于两列的数字（也就是整数 0 至 9）在显示时会带有一个**前导**空格。这种方式称为**右对齐**。

因此，使用字段宽度规格%2i，可以保证至少使用两列来显示 n 的数值，这样就能确保 triangularNumber 的值是对齐的。

如果待显示数值需要的列数超出了指定的字段宽度，printf()在显示该数值时直接忽略字段宽度规格，需要多少列，就使用多少列。

字段宽度规格也可用于显示整数之外的数值。在稍后给出的程序中将看到这样一些例子。

程序输入

程序 4.2 计算第 200 个三角数，仅此而已。如果希望计算第 50 个或第 100

火龙果•整理
uml.org.cn

个三角数，那就必须返回去修改程序，使 for 循环能够执行正确的次数。还必须修改 printf()语句，以显示正确的信息。

一种更简单的解决方案是让程序询问你希望计算哪个三角数。在你给出回答之后，程序就能计算你需要的三角数了。在 C 语言中，这一解决方案可以用一个名为 scanf()的例程实现。scanf()例程在概念上与 printf()例程非常类似。printf()例程用于在终端显示值，而利用 scanf()例程可以将值键入程序中。程序 4.4 询问用户应当计算哪个三角数，然后计算该数字，并显示结果。

程序 4.4　向用户询问输入

```
#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;

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

    return 0;
}
```

在程序 4.4 的输出中，用户键入的数字（100）被设为粗体，以与程序显示的输出相区别。

程序 4.4　输出

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

根据输出，数字 100 是由用户键入的。程序随后计算第 100 个三角数，并在终端上显示结果 5050。如果用户希望计算第 10 个或第 30 个三角数，那完全可以键入数字 10 或 30。

程序 4.4 中的第一个 printf()语句用于提示用户键入一个数字。当然，如果能向用户提供你希望输入的内容，那总是不错的。在输出信息后，调用

火龙果•整理
uml.org.cn

scanf()例程。scanf()的第一个参数是格式字符串，与 printf()使用的格式字符串非常类似。在这种情况下，格式字符串不是告诉系统要显示什么类型的数值，而是要由终端读取什么类型的数值。与 printf()类似，%i 字符用于指定一个整数值。

scanf()例程的第二个参数指明将用户键入的值存储在什么**位置**。变量 number 之前的&字符在这种情况下是必不可少的。不过现在还不用操心它的功能。第 10 章将更详细地讨论这一字符，它实际上也是一种运算符。始终要记着在 scanf()函数调用中的变量名之前放一个前导&。如果忘了它，可能导致不可预测的结果，也可能会导致程序异常终止。

有了前面的讨论内容，现在就能理解程序 4.4 中的 scanf()调用了，它的意思是说：要从终端读取一个整数值，并存储在变量 number 中。这个整数值表明用户希望计算哪个特定的三角数。

在键入这个数字之后（并按下键盘上的 Return 或 Enter 键，表示数字键入完毕），程序将继续计算所请求的三角数。这一任务的完成与程序 4.2 中相同，唯一的区别是不再以 200 为界，而是使用 number。

> **注意**
>
> 在带有数字小键盘的键盘上，按下其中的 Enter 键可能无法将你输入的数字发送给程序。这时应当使用键盘上的 Return 键。

在计算了所需要的三角数之后，将显示结果，程序的执行过程随即结束。

嵌套 for 循环

程序 4.4 为用户提供了灵活性，可以让程序计算任何想要的三角数。但是，如果用户有一份清单，其中是五个要计算的三角数，他可以直接将程序运行五次，每次键入清单中下一个要计算的三角数。

还有另外一种方法可以达到同一目的，就学习 C 语言来说，这种方法也要有意义得多，那就是让程序来处理这一情景。最好的做法就是向程序中插入一个循环，将整个计算序列重复 5 次。你现在已经知道 for 语句可用于建立这样一个循环。程序 4.5 及其相关输出演示了这一方法。

火龙果•整理
uml.org.cn

程序 4.5　使用嵌套 for 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 4.5　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

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

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

这个程序包含两级 for 语句。最外层的 for 语句

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

指明要将程序循环恰好运行 5 次。之所以能看出这一点，是因为 counter 的值最初被设定为 1，然后被递增 1，**直到**它不再小于或等于 5（换句话说，直到它变为 6 为止）。

火龙果•整理
uml.org.cn

与前面的程序示例不同，变量 counter 未在程序中的其他任何地方使用。它的功能就是 for 语句中的循环计数器。但因为它**的确**是一个变量，所以必须在程序中声明它。

程序循环实际上由所有剩余程序语句组成，如大括号所示。如果像下面这样将程序概念化，可能更容易理解它的运行方式：

```
for 5 次
{
   从用户处获得数字。

   计算所需要的三角数。

   显示结果。
}
```

上面提到循环中有一部分是**计算所需要的三角数**，这一部分实际上包括将变量 triangularNumber 设定为 0，再**加上**计算三角数的循环。从而可以看出，有一个 for 语句实际上是包含在另一个 for 语句的**内部**。这在 C 语言中是完全有效的，这种嵌套可以一直深入到所需要的任意级别。

在处理更复杂的程序构造时，比如嵌套 for 语句，缩进的正确使用变得尤其关键。有了适当的缩进，可以很轻松地判断出每个 for 语句中包含了哪些语句。（如果想了解一下在没有对格式给予应有关注时，一个程序会变得多么难以阅读，请参见本章最后的练习 5。）

for 循环的变体

在构建 for 循环时允许进行一些句法上的变化。在编写 for 循环时可能会发现，希望在循环开始之前初始化多个变量，或者希望在每一遍循环中对多个计算求值。

多个表达式

可以在 for 循环的任意字段中包含多个表达式，只要用逗号将这些表达式隔开即可。例如，一个 for 语句的开头为

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

在循环开始之前，i 的值被设定为 0，**并且**，j 的值也被设定为 0。两个表达式

火龙果•整理
uml.org.cn

i = 0 和 j = 0 之间用逗号分隔，两个表达式都被看作该循环“*初始表达式*”字段的一部分。再举一个例子，以如下内容开头的 for 循环

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

设定了两个索引变量 i 和 j；在循环开始之前，前者初始化为 0，后者初始化为 100。每次执行完循环体之后，i 的值被递增 1，而 j 的值被递减 10。

省略字段

就像有时需要在 for 语句的一个特定字段中包含多个表达式一样，也可能需要在 for 语句中**省略**一个或多个字段。为此，只需要省略该字段，并用一个分号来标记它的位置即可。省略 for 语句中某个字段的最常见的应用情景就是没有需要求值的初始表达式。在这种情况下，可以直接将“*初始表达式*”字段“留空”，只要仍然包含分号即可：

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

如果在进入循环之前已经将 j 设定为某一初始值，那就可以使用这一语句。

一个省略了“*循环条件*”字段的 for 循环可以高效地建立一个无限循环，也就是在理论上会永远执行下去的循环。如果可以采用某一其他方式退出循环（比如执行本书其他地方讨论的 return、break 或 goto 语句），那就可以使用这种循环。

声明变量

还可以在 for 循环的初始表达式部分声明变量。其做法与前面声明变量的普通方式相同。例如，可使用下面的代码建立一个 for 循环，定义一个整型变量 counter，并将其初始化为数值 1：

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

变量 counter 只有在执行 for 循环期间是可知的，无法在循环外部访问。再举一个例子，下面的 for 循环

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

定义了两个整型变量，并相应地设定了它们的取值。

火龙果•整理
uml.org.cn

while 语句

while 语句进一步扩展了 C 语言的循环功能。这一常用构造的语法如下：

```
while (表达式)
   程序语句
```

在括号内部指明的“表达式”将被求值。如果“表达式”的求值结果为 TRUE，则立即执行随后的“程序语句”。在执行这一语句（或者放在大括号中的多条语句）之后，再次对“表达式”求值。如果求值结果为 TRUE，则再次执行“程序语句”。一直持续这一过程，直到“表达式”的求值结果最终变为 FALSE，此时将终止循环。然后程序将继续执行跟在“程序语句”之后的语句。

作为它的一个应用示例，程序 4.6 建立了一个 while 循环，它只是从 1 数到 5。

程序 4.6　介绍 while 语句

```
// 此程序介绍 while 语句

#include <stdio.h>

int main (void)
{
     int  count = 1;

     while ( count <= 5 ) {
         printf ("%i\n", count);
         ++count;
     }

    return 0;
}
```

程序 4.6　输出

```
1
2
3
4
5
```

此程序在开始时将 count 的值设定为 1。然后开始执行 while 循环。因

火龙果•整理
uml.org.cn

为 count 的值小于或等于 5，所以将执行紧跟在后面的语句。大括号用于将 printf()语句和递增 count 的语句都定义为 while 循环体。从该程序的输出容易看出，这个循环恰好执行了 5 次，或者说一直执行到 count 的值达到 6。

你可能已经由这个程序意识到，可以很轻松地用 for 语句来完成同一任务。事实上，for 语句总是可以转换为等价的 while 语句，反之亦然。例如，for 语句的一般形式

```
for (初始表达式; 循环条件; 循环表达式)
   程序语句
```

可以等价地表示为如下的 while 语句形式：

```
初始表达式;
while (循环条件){
   程序语句
   循环表达式;
}
```

在熟悉了 while 循环的使用之后，应当可以更好地感受到在什么情况下使用 while 语句更为合理，而在什么情况下应当使用 for 语句。

一般来说，将要执行预定次数的循环非常适合用 for 语句实现。另外，如果初始表达式、循环表达式和循环条件都涉及同一变量，for 语句可能是正确的选择。

下一个程序提供了 while 语句的另一个应用范例。这个程序计算两个整数值的**最大公约数**。两个整数的最大公约数（gcd）是能整除这两个整数的最大整数值。例如，10 和 15 的最大公约数是 5，这是因为 5 是能整除 10 和 15 的最大整数。

有一种过程（或说**算法**）可用来获得两个任意整数的最大公约数。这一算法的基础是最早由欧几里德在大约公元前 300 年设计的一种过程，可表示如下：

问题：找出两个非负整数 ***u*** 和 ***v*** 的最大公约数。
步骤 1：若 ***v*** 等于 0，则完成任务，**gcd** 等于 ***u***。
步骤 2：计算 ***temp = u % v, u = v, v = temp***，返回步骤 1。

不用关心以上算法的工作细节，只要相信它就行了。现在应当更多地关注如何开发一个找出最大公约数的程序，而不是对这个算法的工作方式进行分析。

对于求最大公约数的问题，在已经用算法形式给出了它的解决方案之后，

火龙果•整理
uml.org.cn

开发计算机程序的任务就容易得多了。对算法步骤进行分析后发现，只要 v 的值不等于 0，就会一直重复执行步骤 2。认识到这一点，自然就想到在 C 语言中应当用 while 语句来实现这一算法。

程序 4.7 求出由用户输入的两个非负整数值的最大公约数。

程序 4.7　求最大公约数

```
/*此程序求两个非负整数值的最大公约数*/

#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 4.7　输出

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

程序 4.7　输出（重新运行）

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

scanf()调用中的双 %i 字符表明要由键盘输入两个整数值。第一个输入值存储在整型变量 u 中，而第二个值则存储在变量 v 中。当实际由终端输入数值时，它们之间可以用一个或多个空格分隔，也可由一个回车符分隔。

火龙果•整理
uml.org.cn

当这些值已经由键盘输入并存储在变量 u 和 v 中之后，程序进入 while 循环，计算它们的最大公约数。在 while 循环退出时，u 的值就代表了 v 和原 u 值的最大公约数，它显示在终端上，同时还提供了一条适当的信息。

程序 4.8 演示了 while 语句的另一种用法，将由终端输入的整数数位颠倒。例如，如果用户键入数字 1234，则希望该程序颠倒这个数字的数位，显示结果 4321。

要编写这样一个程序，必须首先提供一个能完成所述任务的算法。在很多时候，只要对自己求解问题的方法进行分析，就能设计出一种算法。要颠倒一个数字的数位，可以将解决方法简单地表述为“自右向左连续读取数字的各个数位”。可以设计一种过程，从最右边的数位开始，持续隔离或“提取”数字的每个数位，从而让计算机程序“连续读取”这个数字的各个数位。所提取的数位可以相继显示在终端上，作为颠倒数字的下一个数位。

为提取一个整数的最右边数位，可以求该整数除以 10 后所得的余数。例如，1234 % 10 给出数值 4，它是 1234 的最右边的数位，也是颠倒后数字的第一个数位。（回忆一下求模运算符，它给出了一个整数除以另一个整数的余数。）如果先将这个数字除以 10，就可以使用同一过程获得数字的下一个数位，别忘了整数除法的工作方式。因此，1234 / 10 给出结果 123，123 % 10 给出 3，这就是已颠倒数字的下一个数位。

可以一直持续这一过程，直到提取了最后一个数位。在一般情况下，当最后一个整数除以 10 的结果为 0 时，就知道已经提取了数字的最后一位。

程序 4.8　颠倒一个数字的数位

```
// 此程序颠倒一个数字的数位

#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;
```

火龙果•整理
uml.org.cn


```
        printf ("%i", right_digit);
        number = number / 10;
    }

    printf ("\n");

    return 0;
}
```

程序 4.8　输出

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

程序在提取每个数位的同时显示它们。注意，包含在 while 循环中的 printf()语句中没有包括换行符。这就强制将每个连续数位显示在同一行。程序末尾的最后一个 printf()调用仅包含一个换行符，它使光标前进到下一行的开头。

do 语句

本章目前讨论的两个循环语句都是在执行循环**之前**进行条件测试。因此，如果条件不满足，循环体可能从来不会执行。在开发程序时，有时可能希望在循环的**末尾**而不是开头进行测试。C 语言很自然地提供了一种用于处理这种情形的特殊构造。这一循环语句称为 do 语句。这个语句的语法如下：

```
do
   程序语句
while (循环表达式);
```

do 语句的执行过程如下：首先执行“*程序语句*”，接下来对括号中的“*循环表达式*”求值。如果“*循环表达式*”的求值结果为 TRUE，则循环继续，再次执行“*程序语句*”。只要“*循环表达式*”的求值结果持续为 TRUE，就重复执行“*程序语句*”。当该表达式的求值结果变为 FALSE 时，循环终止，按照正常顺序执行程序中的下一条语句。

do 语句就是 while 语句的一个变换，将循环条件放在了循环的末尾，而不是开头。

请记住，do 语句不同于 for 和 while 循环，它会保证循环体至少执行

火龙果•整理
uml.org.cn

一次。

程序 4.8 用 while 语句颠倒了一个数字的数位。现在回到该程序，尝试判断一下，如果键入数字 0 来代替 13579，会发生什么情况。while 语句的循环不会被执行，显示器上最终只会给出一个空行（这是第二个 printf() 语句中换行符的显示结果）。如果用 do 语句代替 while 语句，可以保证程序循环至少执行一次，从而保证在所有情况下都会显示至少一个数位。程序 4.9 给出了经过修改后的程序。

程序 4.9　一个经过修改的程序，颠倒一个数字的数位

```
// 此程序颠倒一个数字的数位

#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 4.9　输出

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

程序 4.9　输出（重新运行）

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

火龙果•整理
uml.org.cn

从这个程序的输出可以看出，在将 0 键入程序时，程序会正确地显示数位 0。

break 语句

有时在执行一个循环时，可能希望在发生某一特定条件时（比如检测到一种错误条件，或过早地到达了数据的末尾），立即离开循环。break 语句可用于实现这一目的。执行 break 语句会使程序立即退出正在执行的循环，无论是 for 循环、while 循环，还是 do 循环。循环中的后续语句将被跳过，循环的执行过程终止。将继续执行跟在循环之后的语句。

如果是从一组嵌套循环的内部执行 break，那只有在其中执行 break 的最内层循环会被终止。

break 语句的格式就是在关键字 break 之后跟一个分号：

```
break;
```

continue 语句

continue 语句与 break 语句类似，只是它不会导致循环的终止。而是如其名字所示，这个语句会导致其所在循环的下一轮执行。在执行 continue 语句时，循环中任何出现在 continue 语句**之后**的语句都会被自动跳过。循环执行的其他方面将正常继续。

continue 语句最常见的应用是根据某一条件跳过循环中的一组语句，但在不满足该条件时，将继续该循环的执行。continue 语句的格式很简单，即

```
continue;
```

在非常熟悉程序循环的编写并能优雅地退出循环之前，不要使用 break 或 continue 语句。这些语句很容易被滥用，得到难以读懂的程序。

你现在已经熟悉了 C 语言提供的所有基本循环构造，可以开始学习另一类语言语句，在执行程序期间做出判断。这些判断功能将在第 5 章详细描述。但请首先尝试完成以下练习，以确认你掌握了如何在 C 语言中使用循环。

火龙果•整理
uml.org.cn

练习

1. 键入并运行本章给出的 9 个程序。将每个程序产生的输出与书中每个程序之后给出的输出进行对比。
2. 编写一个程序，对于变化范围为 1 到 10 的整数 n 值，生成并显示 n 和 n^2 的一个表格。务必输出适当的列标题。
3. 对于任意整数值 n，还可以用以下公式生成三角数：

   triangularNumber = n (n + 1) / 2

   例如，将 10 作为 n 值代入上面的公式，可以生成第 10 个三角数 55。编写一个程序，用上述公式生成一个三角数表。在 5 到 50 之间的三角数中，用本程序生成每五个中的一个（即 5,10,15,…,50）。
4. 整数 n 的阶乘记作 n!，是指从 1 到 n 的连续整数的乘积。例如，5 的阶乘可计算如下：

   5! = 5 × 4 × 3 × 2 × 1 = 120

   编写一个程序，生成并输出前 10 个阶乘的一个表格。
5. 下面是一个完全有效的 C 程序，但在编写时没有太多地注意其格式。你会看到，这个程序是非常不适合阅读的。（信不信由你，这个程序甚至还能变得更不适合阅读！）以本章给出的程序为例，重新设定此程序的格式，使其更易读。然后在计算机中键入该程序，并运行它。

```
#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=1;n<=10;++n){
printf("%2i       %i\n",n,two_to_the_n); two_to_the_n*2;}
return 0;}
```

6. 在字段宽度规格的前面放置一个负号，会使字段显示为**左对齐**。用以下 `printf()` 语句代替程序 4.2 中的相应语句，运行程序，并对比两个程序产生的输出。

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

7. `printf()` 中字段宽度规格之前的小数点具有特殊目的。键入并运行以

火龙果•整理
uml.org.cn

下程序，以确定其目的。在每次得到提示时，尝试键入不同的值。

```
#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 4.5 只允许用户键入五个不同的数。修改该程序，使用户能够键入要计算多少个三角数。
9. 重写程序 4.2 至程序 4.5，将所有用到 `for` 语句的地方改为等价的 `while` 语句。运行每个程序，验证两个版本是相同的。
10. 如果在程序 4.8 中键入一个负数，会发生什么情况？尝试输入并查看结果。
11. 编写一个程序，计算一个整数的各数位之和。例如，数字 2155 的数位之和为 2 + 1 + 5 + 5，即 13。此程序应当接受用户键入的任意整数。

火龙果•整理
uml.org.cn

5
做出决策

我们在第 4 章了解到，计算机最基本的特性之一就是能重复执行一个指令序列。但它的另一条基本特性就是能做出决策。我们已经看到，在执行各种循环语句的过程中，这些决策功能是如何被用于判断何时结束程序循环的。没有这些功能，可能永远都不会退出程序循环，最终会一遍又一遍地执行同一序列，在理论上会永远执行下去（这就是为什么这种程序循环被称为无限循环）。

C 程序设计语言还提供了其他几种决策构造，本章将对其进行介绍：

- `if` 语句
- `switch` 语句
- 条件运算符

`if` 语句

C 程序设计语言用一种称为 `if` 语句的语言构造形式，提供了通用决策功能。这一语句的通用形式如下：

```
if (表达式)
   程序语句
```

假设你要把如下语句转换为 C 语言：“如果不下雨，我就去游泳”。利用 `if` 语句的上述形式，可以将其用 C 语言写为

```
if (不下雨)
   我就去游泳
```

利用这个 `if` 语句，可以根据指定条件启动一个或多个程序语句的执行（如果是多个语句，则放在大括号中）。如果不下雨，我就去游泳。同样，在下面的程

火龙果•整理
uml.org.cn

序语句中：

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

只有当 count 的值大于 COUNT_LIMIT 的值时，printf()语句才会执行；否则它将被忽略。

一个实际程序示例有助于理解这一点。假定你想编写一个程序，接受从终端键入的整数，然后显示该整数的绝对值。计算一个整数的绝对值的直接方法是：如果这个数字小于 0，则对其求反。在前面的句子中使用了“如果这个数字小于 0”，这表明必须由程序做出决策。可以利用 if 语句来影响这一决策，如程序 5.1 所示。

程序 5.1　计算一个整数的绝对值

```
// 此程序计算一个整数的绝对值

#include <stdio.h>

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 5.1　输出

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

程序 5.1　输出（重新运行）

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

此程序被运行两次，以验证它的功能正确。当然，你也可能希望将程序再

火龙果•整理
uml.org.cn

多运行几次，以确信它真的能够正确运行，但至少我们已经对程序可能做出的两种决策结果都进行了验证。

在向用户显示一条信息，并将输入的整数值存储到 number 中之后，程序会检测 number 值，看它是否小于 0。如果小于 0，则执行接下来的程序语句，求 number 值的相反数。如果 number 的值不小于 0，则自动忽略这一程序语句。（如果它已经是正数，并不希望求它的相反数。）随后由程序显示 number 的绝对值，程序执行完毕。

下面来看程序 5.2，其中使用了 if 语句。设想有一份成绩清单，希望计算其平均值。但除了计算平均值之外，假定还需要计算清单中不及格成绩的个数。这里假定低于 65 分的成绩为不及格成绩。

要计算不及格成绩的个数，意味着必须对于一个成绩是否是不及格成绩做出决策。if 语句又一次赶来救场了。

程序 5.2　计算一组成绩的平均值，并计算不及格成绩的个数

```
/* 此程序计算一组成绩的平均值并计算不及格成绩的个数 */

#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;

       if ( grade < 65 )
          ++failureCount;
    }

   average = (float) gradeTotal / numberOfGrades;
```

火龙果•整理
uml.org.cn


```
    printf ("\nGrade average = %.2f\n", average);
    printf ("Number of failures = %i\n", failureCount);

    return 0;
}
```

程序 5.2　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
```

变量 gradeTotal 用于在键入成绩时记录它们的累计总和，它被初始设定为 0。不及格成绩的个数存储在变量 failureCount 中，它的值也被初始设定为 0。变量 average 被声明为 float 类型，这是因为一组整数的平均数本身并不一定是整数。

程序随后要求用户输入将会键入的成绩个数，并将输入的值存储在变量 numberOfGrades 中。然后建立一个将为每个成绩执行的循环。循环的第一部分提示用户输入成绩。所输入的数值存储在变量 grade 中，这个变量的名字非常恰当。

接下来将 grade 的值加到 gradeTotal 中，然后进行测试，看它是否为不及格成绩。如果是，就将 failureCount 的值递增 1。然后对清单中的下一个成绩重复整个循环。

在输入了所有成绩并求和之后，程序接下来计算成绩的平均值。乍一闪念，似乎下面这样一条语句可以完成任务：

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

但回想一下，如果使用上面的语句，那除法结果的小数部分将会丢失。这是因为除法运算的除数和被除数**都是**整数，所以将会执行整数除法。

这一问题有两种不同的解决方案。一种是将 numberOfGrades 和

火龙果•整理
uml.org.cn

gradeTotal 之一声明为 float 类型。这就保证了在执行除法时不会丢失小数位。这种方法的唯一问题是：变量 numberOfGrades 和 gradeTotal 在程序中仅用于存储整数值。将其中一个声明为 float 类型，只会模糊它们在程序中的应用，一般来说，这不是一种高明的做事方法。

另一种解决方案就是程序中使用的方法，将其中一个变量的值实际**转换**为浮点值，以进行计算。类型转换运算符（float）用于将变量 gradeTotal 的值转换为 float 类型，以进行表达式求值。因为除法是在 gradeTotal 的值被转换为浮点数**之后**才进行的，所以这一除法被看作是一个浮点数除以一个整数。因为现在有一个操作数是浮点值，所以除法运算被当作浮点运算执行。这当然就意味着，会在平均值中得到想要的小数位。

在计算平均值之后，将其显示在终端上，精度为两位小数。如果在 printf() 格式字符串中的格式字符 f（或 e）之前直接放置一个后面跟有数字的小数点（这个数字和小数点统称为**精度修饰符**），则相应的值会在进行四舍五入后，显示为指定个数的小数位。因此，程序 5.2 中使用精度修饰符.2 将 average 的值显示为两位小数。

当程序显示了不及格成绩的个数之后，程序执行完毕。

注意，如果程序的用户输入了一个 0，作为要记录的测验成绩个数，程序会生成一些很奇怪的结果，类似于 NaN（Not a Number，不是数字）或其他什么东西；具体结果会随系统变化，取决于你的计算机处理被 0 除的方式。你可能会感到奇怪，如果没有要输入的测验分数，为什么还会有人不嫌麻烦地运行程序来记录测验分数，但这就是可以向程序中添加的错误检测类型。

if-else 构造

如果有人问你一个具体的数字是偶数还是奇数，最可能的做法就是通过查看这个数字的最后一位来做出判断。如果这个数字是 0、2、4、6 或 8 中的一个，那就可以说这个数字为偶数。否则，就声明这个数字为奇数。

计算机还有一种更简单的方法来判断一个具体数字的奇偶性，它并不检查这个数字的最后一位是不是 0、2、4、6 或 8，而是直接判断这个数字是否可以被 2 整除。如果能整除，这个数字就是偶数；否则就是奇数。

前面已经看到如何用求模运算符%来计算一个整数除以另一个整数的余

火龙果•整理
uml.org.cn

数。因此，用这个运算符来判断一个整数是否能被 2 整除，是再好不过的了。如果除以 2 之后的余数为 0，这个数字就是偶数；否则为奇数。

请看程序 5.3，这个程序判断用户键入的整数值是偶数还是奇数，并在终端显示适当的信息。

程序 5.3　判断一个数字是偶数还是奇数

```
//  此程序判断一个数字是偶数还是奇数

#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 5.3　输出

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

程序 5.3　输出（重新运行）

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

在键入数字之后，计算它除以 2 所得的余数。第一个 `if` 语句检测这个余数值，看它是否等于 0。如果等于，则显示信息“The number is even.”（此数字为偶数）。

第二个 `if` 语句检测该余数，看它是否不等于 0，如果确实如此，则显示一条信息，表示这个数字为奇数。

火龙果•整理
uml.org.cn

事实上，只要第一个 if 语句成功，第二个就一定会失败，反之亦然。回想一下本节开头对奇偶数的讨论，如果一个数字能被 2 整数，则它为偶数；**否则(else)**为奇数。

在编写程序时，用到“else”这一概念的频率是如此之高，使得几乎所有现代程序设计语言都提供了一种特殊的构造来处理这一情景。在 C 语言中，这称为 if-else 构造，其一般形式如下：

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

if-else 实际上就是 if 语句一般形式的一种扩展。如果“表达式”的求值结果为 TRUE，则程序执行紧随其后的“程序语句 1”；否则执行“程序语句 2”。在任一情况下，都是执行“程序语句 1”或“程序语句 2”中的一个，但不会两者都执行。

可以将 if-else 语句整合到程序 5.3 中，用单个 if-else 语句代替两个 if 语句。这一新程序构造的应用实际上有助于降低程序的复杂度，还会提高其可读性，如程序 5.4 所示。

程序 5.4　修改程序，判断一个数字是偶数还是奇数

```
// 此程序确定一个数字是偶数还是奇数（第 2 版）

#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;
}
```

火龙果•整理
uml.org.cn

程序 5.4　输出

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

程序 5.4　输出（重新运行）

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

别忘了，双等号==是相等检测符，单等号是赋值运算符。如果忘了这一点，因为疏忽而在 if 语句中使用了赋值运算符，可能会导致非常头痛的结果。

复合关系测试

本章目前使用的 if 语句都是在两个数字之间建立简单的关系测试。程序 5.1 中将 number 的值与 0 相比较；而在程序 5.2 中，则将 grade 的值与 65 相比较。有时可能希望甚至必须建立更复杂的测试。例如，假定在程序 5.2 中，希望计算的不是不及格成绩的个数，而是介于 70 分到 79 分之间的成绩个数。在这种情况下，不是希望将 grade 与一个界限相比较，而是要与两个界限 70 和 79 相比较，以确保它会落在指定范围内。

C 语言提供了执行此类复杂关系测试所需要的机制。**复合关系测试**就是用**逻辑与**、**逻辑或**运算符联结在一起的一个或多个简单关系测试。这些运算符分别用一对&&和||（两个垂直竖线字符）表示。举个例子，C 语句

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

仅在 grade 的值不小于 70 **且**不大于 79 时递增 grades_70_to_79 的值。采用类似方式，语句

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

在 index 小于 0 或大于 99 时执行 printf()语句。

在 C 语言中，可以使用复合运算符组成极为复杂的表达式。在构建表达式方面，C 语言给予了程序员极大的灵活性。这一灵活性经常被滥用。简单表达式几乎总是更易于阅读和调试。

要构建复合关系表达式时，可以大量使用括号来帮助提高表达式的可读性，

火龙果•整理
uml.org.cn

并避免因为错误地假设表达式中各运算符的优先级而陷入麻烦。还可以使用空格来帮助提高表达式的可读性。在&&和||运算符两端增加一个空格，可以从视觉上将这些运算符与借由它们连接在一起的表达式区分开来。

为在一个实际程序例子中演示复合关系测试的应用，可编写一个程序，来判断一个年份是否为闰年。如果一个年份能被 4 整除，它就是闰年。但你可能没有意识到，一个能被 100 整除的年份如果不能被 400 整除，它就**不是**闰年。

尝试考虑一下，你会如何为这种情景建立一个测试。首先，可以计算该年份在除以 4、100 和 400 之后的余数，并将这些值分别指定给适当命名的变量，比如，rem_4、rem_100 和 rem_400。接下来可以检测这些余数，以判断其是否满足闰年的标准。

如果重新描述前面对闰年的定义，可以说，如果一个年份“能被 4 整除且不能被 100 整除”，或者“能被 400 整除”，就表示该年份为闰年。暂时停一下，理解消化一下最后这个句子，检验一下它是否与前面给出的定义等价。既然已经用这些术语重新表述了定义，那将它转换为程序语句的任务就变得相当简单，如下所示：

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

子表达式

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

两侧的括号并非必需的，因为无论如何都会这样对该表达式进行求值。

如果在这个检测之前增加一些语句，以声明你的变量，使用户能够由终端键入年份，就会得到一个程序，用来判断一个年份是否为闰年，如程序 5.5 所示。

程序 5.5　判断一个年份是否为闰年

```
// 此程序判断一个年份是否为闰年

#include <stdio.h>

int main (void)
{
   int  year, rem_4, rem_100, rem_400;
```

火龙果•整理
uml.org.cn


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

   return 0;
}
```

程序 5.5　输出

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

程序 5.5　输出（重新运行）

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

程序 5.5　输出（第二次重新运行）

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

前面的例子给出了一个因为不能被 4 整除因而不是闰年的年份（1955 年）、一个因为能被 400 整除因而是闰年的年份（2000 年），以及一个因为能被 100 整除但不能被 400 整除因而不是闰年的年份（1800）。为完成测试用例的运行，还应当尝试一个能被 4 整除但不能被 100 整除的年份。这一工作留作练习。

前面曾经提到，C 语言在构建表达式方面提供了极大的灵活性。例如，在前面的程序中，不需要计算中间结果 rem_4、rem_100 和 rem_400，可以直接在 if 语句中执行计算，如下所示：

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

用空格分隔各个运算符，仍然可以使上面的表达式易于阅读。如果决定不添加空格，并删除不必要的括号，将会得到类似于如下所示的表达式：

火龙果•整理
uml.org.cn


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

这个表达式是完全有效的，而且不管你相信与否，它的执行与前面刚刚给出的表达式完全相同。显然，那些额外的空格对于理解复杂表达式有很大的帮助。

嵌套 if 语句

在 if 语句的一般形式中，别忘了，如果对括号内部表达式的求值结果为 TRUE，将执行紧随其后的语句。如果这个程序语句是另一个 if 语句，如下面的语句一样，那也是完全有效的：

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

如果 gameIsOver 的值为 0，将执行随后的语句，它是另一个 if 语句。这个 if 语句将 playerToMove 的值与 YOU 进行比较。如果两个值相等，则在终端上显示信息“Your Move”。因此，只有当 gameIsOver 等于 0 且 playerToMove 等于 YOU 时，才会执行这个 printf 语句。事实上，这个语句也可以使用如下复合关系式做等价描述：

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

“嵌套”if 语句的一个更实用的例子是向前面的例子中增加一个 else 子句，如下所示：

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

这一语句的执行过程如前所述。但是，如果 gameIsOver 等于 0，且 playerToMove 的值不等于 YOU，将执行 else 子句。这时将在终端上显示信息“My Move”。如果 gameIsOver 不等于 0，则跳过跟在后面的整个 if 语句，包括与其关联的 else 子句。

注意，这个 else 子句是与测试 playerToMove 值的 if 语句关联在一起的，而不是与测试 gameIsOver 值的 if 子句关联在一起。else 子句的一般规则是，它总是与最后一个不包含 else 子句的 if 语句相关联。

火龙果•整理
uml.org.cn

可以再进一步，向上面例子中最外层的 if 语句添加一个 else 子句。当 gameIsOver 的值不为 0 时，执行这个 else 子句。

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

正确使用缩进对于理解复杂语句的逻辑很有帮助。

当然，即使你使用缩进方式表明了你认为一个语句在 C 语言中的解读方式，但它与编译器实际解释该语句的方式可能并非总是一致的。例如，删除上例中的第一个 else 子句：

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

这样**并不会**使该语句的解释方式与其格式指示的方式相同。而是会被解释为

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

因为这个 else 子句是与最后一个不带 else 的 if 相关联的。在这种情况下，可以使用大括号强制进行不同关联，使得最内层 if 不包含 else，但外层的 if 却包含 else。大括号的效果是“关闭”if 语句。因此，

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

就可以达到预期效果，如果 gameIsOver 的值不是 0，则显示信息“The game is over”。

火龙果•整理
uml.org.cn

else if 构造

我们已经看到在对两种可能条件（数字要么为偶数，要么为奇数；年份要么是闰年，要么不是闰年）进行判断时 else 语句所发挥的作用。但是，在程序设计时所要做出的决策并非总是这样非黑即白的。考虑这样一个任务：编写一个程序，当用户键入的数字小于 0 时显示-1，等于 0 时显示 0，大于 0 时显示 1。（这实际上就是实现了一个通常所说的**符号**函数。）显然，在这种情况下必须进行三次测试——判断键入的数字是负数、零，还是正数。简单的 if-else 构造无法完成任务。当然，在这种情况下，总是可以求助于三个独立的 if 语句，但这种解决方案通常也并非总是有效的，特别是当所做测试并非相互排斥时，更是如此。

在 else 子句中增加一个 if 语句就可以处理刚刚描述的情景。因为跟在 else 之后的语句可以是任意的有效 C 程序语句，所以它当然可以是另一个 if 语句。因此，在一般情况下，可以写为

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

这种形式非常有效地将 if 语句从两值逻辑判断扩展为一个三值逻辑判断。可以根据上述形式，继续向 else 子句中增加 if 语句，有效地将这一逻辑判断扩展为一个 *n* 值逻辑判断。

上面的构造经常会被用到，所以通常将其称为 else if 构造，采用一种不同于前面给出的形式，写为

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

后面这种形式提高了语句的可读性，可以清楚地看到正在进行一种三路判断。

程序 5.6 通过实现前面讨论的符号函数，演示了 else if 构造的使用。

火龙果•整理
uml.org.cn

程序 5.6　实现符号函数

```
// 此程序实现符号函数

#include <stdio.h>

int main (void)
{
    int  number, sign;

    printf ("Please type in a number: ");
    scanf ("%i", &number);

    if ( number < 0 )
        sign = -1;
    else if ( number == 0 )
        sign = 0;
    else        // 必然为正数
        sign = 1;

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

    return 0;
}
```

程序 5.6　输出

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

程序 5.6　输出（重新运行）

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

程序 5.6　输出（第二次重新运行）

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

如果输入的数字小于 0，则 sign 被赋值为-1；如果数字等于 0，则 sign 被赋值为 0；否则，数字必然大于 0，所以 sign 被赋值为 1。

程序 5.7 分析了由终端键入的字符，并将它归类为字母字符（a～z 或 A～Z）、数位（0～9）或特殊字符（其他任意字符）。为从终端读取单个字符，可以在 scanf() 调用中使用格式字符%c。

火龙果•整理
uml.org.cn

程序 5.7　对由终端输入的单个字符进行分类

```
// 此程序对由终端输入的单个字符进行分类

#include <stdio.h>

int main (void)
{
    char  c;

    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 5.7　输出

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

程序 5.7　输出（重新运行）

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

程序 5.7　输出（第二次重新运行）

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

在读入字符之后进行的第一次测试判断 char 变量 c 是否为字母字符。这是通过判断该字符是否为小写字母或大写字母而完成的。前一个测试由以下表达式完成：

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

火龙果•整理
uml.org.cn

如果 c 在字符'a'至'z'的范围内，也就是说，如果 c 为小写字母，则上面的表达式为 TRUE。后一个测试由以下表达式完成：

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

如果 c 在字符'A'至'Z'的范围内，也就是说，如果 c 为大写字母，则上面的表达式为 TRUE。对于所有计算机系统，只要在机器内部将字符存储为 ASCII 格式，这些测试就是有效的。[①]

如果变量 c 是一个字母字符，第一个 if 测试成功，将显示“It's an alphabetic character.”。如果该测试失败，则执行 else if 子句。这个子句判断字符是否为数字。注意，这个测试是将字母 c 与**字符**'0'和'9'进行比较，而不是与**整数** 0 和 9 相比较。这是因为字符是由终端读入的，而字符'0'到'9'与数字 0～9 并不相同。事实上，在一个使用前述 ASCII 格式的计算机系统上，字符'0'在内部实际上表示数字 48，字符'1'表示数字 49，以此类推。

如果 c 是一个数字，则显示“It's a digit.”。否则，如果 c 不是字母，也不是数字，则执行最后一个 else 子句，显示“It's a special character.”，程序随后执行完毕。

应当注意，尽管这里使用 scanf()仅读取了单个字符，但在键入字符之后，仍然必须按下 Enter（或 Return）键，以将输入发送给程序。一般情况下，在从终端上读取数据时，在按下 Enter 键之前，程序是看不到任何键入数据的。

对于下一个例子，假定希望编写一个程序，允许用户键入如下形式的简单表达式：

```
数字  运算符  数字
```

程序对表达式求值，并在终端上显示结果，准确到两位小数。希望识别的是加减乘除等常规运算符。程序 5.8 使用了一个带有许多 else if 子句的大型 if 语句，判断将要执行哪种运算。

---

① 更好的做法是使用标准库中名为 islower()和 isupper()的例程，以完全避开内部表示问题。为此，需要在程序中包含代码行#include <ctype.h>。这里给出相关内容只是为了进行说明。

火龙果•整理
uml.org.cn

程序 5.8　对简单表达式求值

```
/* 此程序对以下形式的表达式进行求值
            数字  运算符  数字      */

#include <stdio.h>

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

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

    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 == '/' )
        printf ("%.2f\n", value1 / value2);

    return 0;
}
```

程序 5.8　输出

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

程序 5.8　输出（重新运行）

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

程序 5.8　输出

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

scanf()调用指定了三个待读入变量 value1、operator 和 value2 的值。浮点数可以用%f 格式字符读入，这一格式字符也用于输出浮点数。在为表

火龙果•整理
uml.org.cn

达式的第一个操作数 value1 变量读入数值时，使用的就是这一格式。

接下来希望读入运算符。因为运算符是一个字符（'+'、'-'、'*'或'/'）而不是数字，所以将其读入字符变量 operator 中。%c 格式字符告诉系统，由终端读入下一个字符。格式字符串中的空格表明在输入时允许存在任意数目的空格。这样，在键入这些值时，就可以用空格将操作数与运算符分隔开。但如果指定了格式字符串"%f%c%f"，那在键入的第一个数字和运算符之间不允许存在空格。这是因为 scanf()函数是用%c 格式字符读入一个字符的，输入中的下一个字符就是被读入的字符，**即使这个字符是空格**。但是，应当注意到，在一般情况下，scanf()函数在读入小数或浮点数时，**总是**会忽略前导空格。因此，格式字符串"%f %c%f"会像上面的程序中一样有效。

在键入第二个操作数并将其存储在变量 value2 中之后，程序接下来检测 operator 是否为四个允许运算符之一。当找出正常匹配时，执行相应的 printf()语句，显示计算结果。随后，程序执行完毕。

这里该对程序的完整性说两句了。尽管前面的程序的确完成了预定任务，但这个程序实际上并不完整，因为它没有考虑用户犯错的情况。例如，如果用户在键入运算符时错误地键入了一个?，会发生什么情况呢？此时程序就只是简单地“贯穿（fall through）”if 语句，不会在终端上显示信息，告诉用户他错误地键入了表达式。

另外一个被忽略的情况是用户在键入除法运算时，除数为 0。我们已经知道，在 C 语言中永远都不要尝试用 0 除某个数字。程序应当检查这一情况。

尝试预测程序可能失败或给出意外结果的方式，然后为这些情况采取预防措施，是编写优秀、可靠程序的必要步骤。针对一个程序运行足够数量的测试用例，经常会找出程序中未考虑特定情形的部分。但这还不够。在编译程序代码时，应当总是自觉地问一句“如果……，会发生什么情况”，并插入必要的程序语句进行正确处理。

程序 5.8A 是程序 5.8 的一个修改版本，考虑了被 0 除和键入未知运算符的情况。

程序 5.8A　修改程序，对简单表达式求值

```
/* 此程序对以下形式的表达式进行求值
          值  运算符  值       */
```

火龙果•整理
uml.org.cn


```
#include <stdio.h>

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

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

    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 5.8A　输出

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

程序 5.8A　输出（重新运行）

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

程序 5.8A　输出（第二次重新运行）

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

当键入的运算符是用于除法的斜线时，会进行另一个测试，判断 value2 是否为 0。如果是，则在终端显示适当的信息。如果不是，则执行除法运算，

火龙果•整理
uml.org.cn

显示结果。在这种情况下，一定要非常留意 if 语句的嵌套和相关联的 else 子句。

程序最后的 else 子句可以捕获所有“贯穿”情况。因此，任何与所有四个被检测字符均不匹配的 operator 取值都将导致执行这一 else 子句，从而显示“Unkown operator.”。

switch 语句

我们在上一个程序示例中遇到了一种 if-else 语句链，它持续将一个变量与不同取值进行比较，这种 if-else 语句在开发程序时使用得很频繁，所以 C 语言提供了一种恰好实现这一功能的特殊程序语句。这就是 switch 语句，其一般形式为

```
switch(表达式)
{
   case value1:
      程序语句
      程序语句
      …
      break;
   case value2:
      程序语句
      程序语句
      …
      break;
      …
   case valuen:
      程序语句
      程序语句
      …
      break;
   default:
      程序语句
      程序语句
      …
      break;
}
```

放在括号内的“表达式”连续与 *value1*，*value2*，…，*valuen* 各值进行比较，这些值必须是简单常量或常量表达式。如果发现某个 case 后的值与“表

火龙果•整理
uml.org.cn

达式”的值相等，将执行跟在该 case 后面的程序语句。注意，当有多个此种程序语句时，不需要将它们放在大括号中。

break 语句表示一个特定 case 语句的结束，使该 switch 语句的执行终止。别忘了在每个 case 语句的最后包含这个 break 语句。如果忘了为某一特定 case 语句包含该语句，那么在该 case 语句获得执行后，程序会继续执行到下一 case 语句。

如果“表达式”的值与任何 case 后的值都不匹配，则执行 default 语句。这在概念上等价于在前面示例中使用的“贯穿”else。事实上，switch 语句的一般形式可以等价地表示为一个 if 语句，如下所示：

```
if ( 表达式 == value1 )
{
   程序语句
   程序语句
   …
}
else if ( 表达式 == value2 )
{
   程序语句
   程序语句
   …
}
…
else if ( 表达式 == valuen )
{
   程序语句
   程序语句
   …
}
else
{
   程序语句
   程序语句
   …
}
```

请记着，可以将程序 5.8A 中的大型 if 语句转换为一个等价的 switch 语句，如程序 5.9 所示。

火龙果•整理
uml.org.cn

程序 5.9　修改程序，对简单表达式求值，第 2 版

```
/* 此程序对以下形式的表达式进行求值
            值  运算符  值       */

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

   return 0;
}
```

程序 5.9　输出

```
Type in your expression.
178.99 - 326.8
-147.81
```

在读入表达式之后，依次将 operator 的值与每个 case 语句中指定的值

火龙果•整理
uml.org.cn

进行比较。当发现两者匹配时，执行对应 case 语句中包含的程序语句。break 语句随后将执行过程带出 switch 语句，程序执行完毕。如果所有 case 后的值都与 operator 的值不匹配，则会执行 default 语句，显示“Unknown operator.”。

在上面的程序中，default 语句中的 break 语句实际上是不必要的，因为 switch 内部没有任何语句跟在 default 语句之后。不过，记着在每种情况的最后都包含 break 是一个很好的程序设计习惯。

在编写 switch 语句时，一定记着不同的 case 后不能跟着相同的值。但是，可以为一组特定程序语句关联多个值。为此，只需在要执行的共同语句之前列出多个值即可（在每个值之前带有关键字 case，在每个值之后带有一个冒号）。举个例子，在下面的 switch 语句中，如果 operator 等于星号或小写字母 x，都将执行 printf 语句，将 value1 乘以 value2。

```
switch (operator)
{
   …
   case '*':
   case 'x':
           printf("%.2f\n", value1 * value2);
           break;
           …
}
```

布尔变量

很多新程序员很快发现自己要完成一项任务：必须编写一个程序，生成一份**质数**表。复习一下，如果正整数 p 不能被任意其他整数（1 和它自身除外）整除，它就是一个质数。第一个质数被定义为 2。下一个质数为 3，因为它不能被 1、3 之外的任意整数整除。4 **不是**质数，因为它**可以**被 2 整除。

有多种方法可用于生成一份质数表。比如，你的任务是要生成不大于 50 的所有质数，那生成这样一张表格的最直接（也是最简单）的算法就是：测试每个整数 p 对于 2 到 p–1 之间所有整数的整除关系。只要 p 能被其中任何一个整数整除，它就不是质数；否则，它就是一个质数。程序 5.10 给出了生成质数表的程序。

火龙果•整理
uml.org.cn

程序 5.10　生成质数表

```
// 此程序生成质数表

#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);
    }


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

程序 5.10　输出

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

关于程序 5.10，有几点值得注意。最外层的 for 语句建立了一个循环，轮流取整数 2 至 50 中的每个整数。循环变量 p 表示当前正在测试其是否为质数的值。循环中的第一个语句为变量 isPrime 赋数值 1。稍后就会明白这一变量的用途。

第二个循环的建立是为了将 p 除以 2 至 p-1 的整数。在这个循环内部进行测试，查看 p 除以 d 的余数是否为 0。如果等于 0，就知道 p 不可能是质数，因为有一个既不是 1 也不是它本身的整数可以整除它。为了表示 p 不再可能是质数，将变量 isPrime 的值设定为 0。

在完成最内层循环的执行时，检测 isPrime 的值。如果这个值不等于 0，那就是没有找到能整除 p 的整数；因此，p 必然是一个质数，应显示它的值。

你可能已经注意到，变量 isPrime 的值或者为 0，或者为 1，不可能是其

火龙果•整理
uml.org.cn

他值。这就是我们将它声明为_Bool 变量的原因。只要还具备作为质数的资格，那这个变量的值就是 1。但只要找到一个能整除的约数，它的值就被设定为 0，表示 p 不再满足作为质数的要求。以这种方式使用的变量经常被称为**标志**。一个标志通常只取两个不同值之一。此外，标志的值通常会在程序中至少检测一次，看它是“开”（TRUE）还是“关”（FALSE），然后根据检测结果采取某一特定操作。

在 C 语言中，一个标志取 TRUE 或 FALSE 时，大多被非常自然地分别转换为数值 1 和 0。因此，在程序 5.10 中，在循环中将 isPrime 的值设定为 1 时，就是成功地将它们设定为 TRUE，表示 p“是质数”。如果在执行内层 for 循环期间找到一个能整除的约数，则将 isPrime 的值设定为 FALSE，表示 p 不再可能“是质数”。

通常用数值 1 表示 TRUE 或“开”状态，用 0 表示 FALSE 或“关”状态，这并不是一种巧合。这种表示方法对应于计算机内部的单比特概念。当一个比特为“开”时，它的值就是 1；当它为“关”时，它的值就是 0。但在 C 语言中，还有一种更具说服力的论点来支持这些逻辑值。它与 C 语言中处理 TRUE 和 FALSE 概念的方式有关。

回想本章开头，如果“满足”在 if 语句内部指定的条件，则执行紧跟其后的程序语句。但“满足”到底是什么意思呢？在 C 语言中，满足意味着不是 0，别无它意。因此，语句

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

会导致 printf()语句的执行，因为 if 语句中的条件（在本例中，就是数值 100）不是 0，因此，满足该条件。

在本章的每个程序中，都使用“非零表示满足”、“零表示不满足”的观点。这是因为在 C 语言中对一个关系表达式求值时，如果满足该表达式，就为其指定数值 1；如果不满足该表达式，就为其指定数值 0。因此，语句

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

的求值过程如下：

1. 对关系表达式 number < 0 求值。如果满足该条件，也就是说，如果

火龙果•整理
uml.org.cn

number 小于 0，则该表达式的值为 1；否则，它的值为 0。

2. if 语句检测表达式求值的结果。如果结果不为 0，则执行紧随其后的语句；否则，跳过该语句。

上面的讨论也适用于 for、while 和 do 语句中的条件求值。例如，语句

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

中复合关系表达式的求值过程也与上述讨论一致。如果两个指定条件都有效，则结果为 1；但如果任一条件无效，那求值结果就是 0。随后检查求值结果。如果结果为 0，则 while 循环终止；否则继续执行。

回到程序 5.10 和标志的概念，在 C 语言中，利用下面这样的表达式来判断一个标志的值是否为 TRUE 是完全有效的：

```
if ( isPrime )
```

而不需要使用等价表达式

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

要轻松检查一个标志的值是否为 FALSE，可以使用**逻辑求反**运算符!。在下面的表达式中

```
if ( ! isPrime )
```

逻辑求反运算符用于判断 isPrime 的值是否为 FALSE。一般情况下，表达式

```
! 表达式
```

会将“表达式”的逻辑取值取反。因此，如果“表达式”为 0，则逻辑求反运算符将产生一个 1。如果“表达式”的求值结果不是 0，则求反运算符会产生一个 0。

逻辑求反运算符可用于轻松地“翻转”一个标志的值，比如下面的表达式：

```
myMove = ! myMove;
```

不出所料，这个运算符的优先级与一元减运算符相同，这就是说，它的优先级高于所有二元算术运算符和所有关系运算符。因此，为了判断变量 x 的值是否不小于变量 y 的值，比如在

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

火龙果•整理
uml.org.cn

中，需要添加括号来确保表达式的正确求值。当然，也可以将前面的表达式等价地表示为

```
x >= y
```

在第 3 章，我们学习了 C 语言中定义的一些特殊值，可以在处理布尔值时使用。它们是类型 bool 和数值 true 和 false。为使用它们，需要在程序中包含头文件<stdbool.h>。程序 5.10A 是程序 5.10 的改写版本，其中使用了这一数据类型和这两个值。

程序 5.10A　修改程序，生成质数表

```
// 此程序生成质数表

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

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

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

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

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


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

程序 5.10A　输出

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

可以看出，在程序中包含<stdbool.h>后，就可以将变量声明为 bool 类型，而不再是_Bool 类型。这绝对是出于装饰目的，因为前者要比后者更容

火龙果•整理
uml.org.cn

易阅读和键入，而且与 C 语言其他基本数据类型的风格更为一致，比如 int、float 和 char。

条件运算符

C 语言中最不同寻常的运算符可能就是**条件**运算符了，它与 C 语言中的所有其他运算符都不一样，那些运算符或者为一元运算符，或者为二元运算符，而条件运算符却是一个**三元**运算符；也就是说，它有三个操作数。用于表示这一运算符的两个符号是：问号（?）和冒号（:）。第一个操作数放在问号之前，第二个放在问号和冒号之间，第三个则放在冒号之后。

条件运算符的一般形式为

```
条件 ?  表达式1 : 表达式2
```

其中的“条件”是一个表达式，通常是一个关系表达式，在遇到关系运算符时，首先对其进行求值。如果“条件”的求值结果为 TRUE（也就是说，不等于 0），就对“表达式 1”求值，求值结果变成整个运算的结果。如果“条件”的求值结果为 FALSE（也就是 0），就对“表达式 2”求值，它的结果变成整个运算的结果。

条件运算符最常见的应用场景是根据某一条件为一个变量指定两个值之一。例如，有一个整型变量 x 和另一个整型变量 s。如果希望在 x 小于 0 时为 s 赋数值-1，否则为 s 赋 x^2 的值，那就可以写出下面的语句：

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

在执行上面的语句时，首先检测条件 x < 0。通过在条件表达式的两侧放置括号，以帮助提高语句的可读性。这通常并非必需的，因为条件运算符的优先级非常低，事实上，低于除赋值运算符和逗号运算符之外的所有其他运算符。

如果 x 的值小于 0，则计算紧跟在问号之后的表达式。这个表达式就是常整数值-1，如果 x 小于 0，则将这个值赋给变量 s。

如果 x 的值不小于 0，则计算紧跟在冒号之后的表达式。因此，如果 x 大于或等于 0，则将 x * x，也就是 x^2 的值赋给变量 s。

作为条件运算符的另一个应用示例，下面的语句将 a 和 b 的最大值赋给变量 maxValue：

火龙果•整理
uml.org.cn


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

如果冒号之后使用的表达式（“else”部分）由另一个条件运算符组成，就可以实现“else if”子句的效果。例如，利用两个条件运算符，可以在一行之内写出程序 5.6 中实现的**符号**函数：

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

如果 number 小于 0，则为 sign 赋数值-1；否则，如果 number 等于 0，则为 sign 赋数值 0；否则，为其赋数值 1。上面表达式中“else”部分两侧的括号实际上是不必要的。这是因为，条件运算符是自右向左结合的，也就是说，如果在单个表达式中多次使用这个运算符，比如下面的表达式：

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

将条件运算符自右向左分组，可知其求值顺序如下：

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

条件运算符并非一定要用在一个赋值运算符的右侧，它可以用在任何能够使用表达式的情景中。也就是说，利用下面的 printf 语句，可以直接显示变量 number 的符号，不需要首先将它指定为一个变量：

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

在 C 语言中编写预处理**宏指令**时，使用条件运算符非常方便。详细内容可参见第 12 章。

对“做出决策”部分的讨论就此结束。第 6 章将首先带你看一些更为复杂的数据类型。**数组**是一种功能强大的概念，在用 C 语言开发的许多程序中都会找到它的影子。在继续学习后续内容之前，请先完成以下练习，测试一下你对本章内容的理解程度。

练习

1. 键入并运行本章给出的 12 个程序。将每个程序产生的输出与书中每个程序之后给出的输出进行对比。
2. 编写一个程序，要求用户由终端键入两个整数值。测试这两个数字，确定第一个数字是否能被第二个数字整除，并在终端显示一个合适的信息。

火龙果•整理
uml.org.cn

3. 编写一个程序，接受用户键入的两个整数值。显示第一个整数除以第二个整数的结果，精确到三位小数。记着让程序检查被 0 除的情况。
4. 编写一个充当简单“打印”计算器的程序。该程序应当允许用户键入如下形式的表达式：

   数字 运算符

   程序应当可以识别以下运算符：

   + - * / S E

   S 运算符告诉程序，将“累加器”设定为键入的数字。E 运算符告诉程序，运行要结束了。以键入的数字作为第二个操作数，对累加器中的内容和这个第二操作数执行算术运算。下面是一次“样本运行”，说明程序应当如何运行：

```
开始计算
10 S            设定累加器为 10
= 10.000000     累加器的内容
2 /             除以 2
= 5.000000      累加器的内容
55 -            减去 55
-50.000000
100.25 S        设定累加器为 100.25
= 100.250000
4 *             乘以 4
= 401.000000
0 E             程序结束
= 401.000000
计算结束
```

   确保程序会检测被 0 除，还会检查未知运算符。
5. 你开发了程序 4.9，将一个从终端键入的整数的各个数位颠倒顺序。但是，如果键入一个负数，这个程序就无法正常运行。查明在这种情况下发生了什么，并修改程序，使负数也能被正确处理。例如，如果键入了数字–8645，程序的输出应当是 5468–。
6. 编写一个程序，接受终端键入的一个整数，提取该整数的各个数位，并显示其英文。因此，如果用户键入了 932，则程序应当显示

   nine three two

   如果用户键入的就是一个 0，别忘了显示“zero”。（注意：这个练习比较难！）
7. 程序 5.10 有几个低效的地方。其中之一源于对偶数的检验。因为任何

火龙果•整理
uml.org.cn

大于 2 的偶数显然不可能是质数，所以该程序可以直接忽略所有偶数，不再将其作为候选质数和候选约数。内层 `for` 循环的效率也很低，因为它总是将 `p` 的值除以从 2 到 `p-1` 的所有 `d` 值。在 `for` 循环的条件中加入对 `isPrime` 值的检测，就可以避免这种低效性。这样，可以将 `for` 循环设定为，只要还没有找到约数并且 `d` 的值小于 `p`，就一直持续执行。修改程序 5.10 以加入这两处修改。然后运行程序，验证其运行情况。（注意：在第 6 章会发现一些更为高效的质数生成方法。）