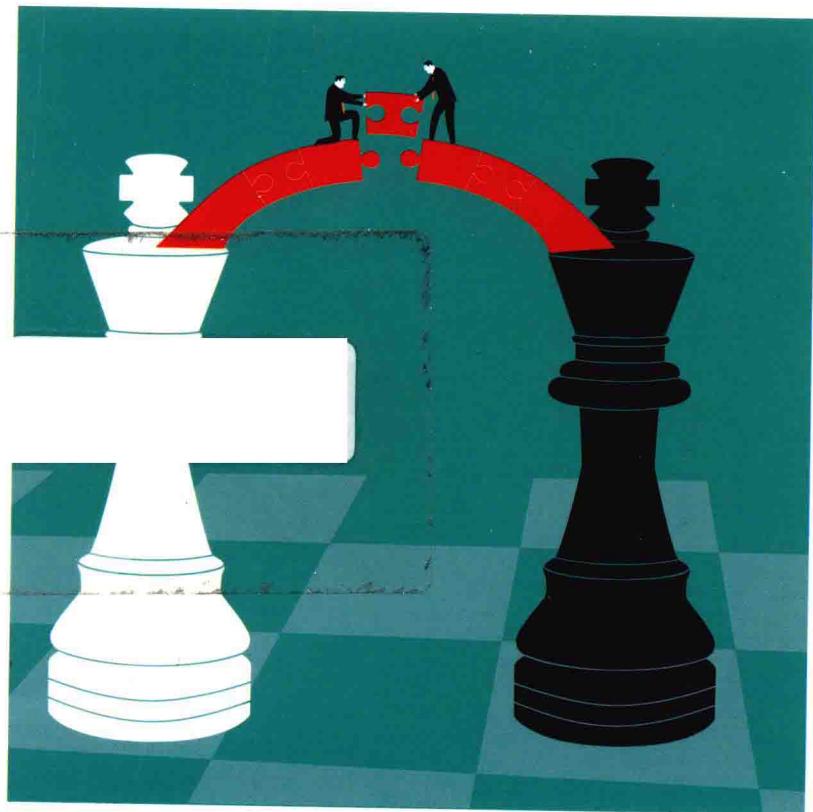


# C Primer Plus

## (第6版) 中文版

*C Primer Plus  
Sixth Edition*

- 经久不衰的C语言畅销经典教程
- 针对C11标准进行全面更新



[美] | Stephen Prata | 著

姜佑 | 译



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS

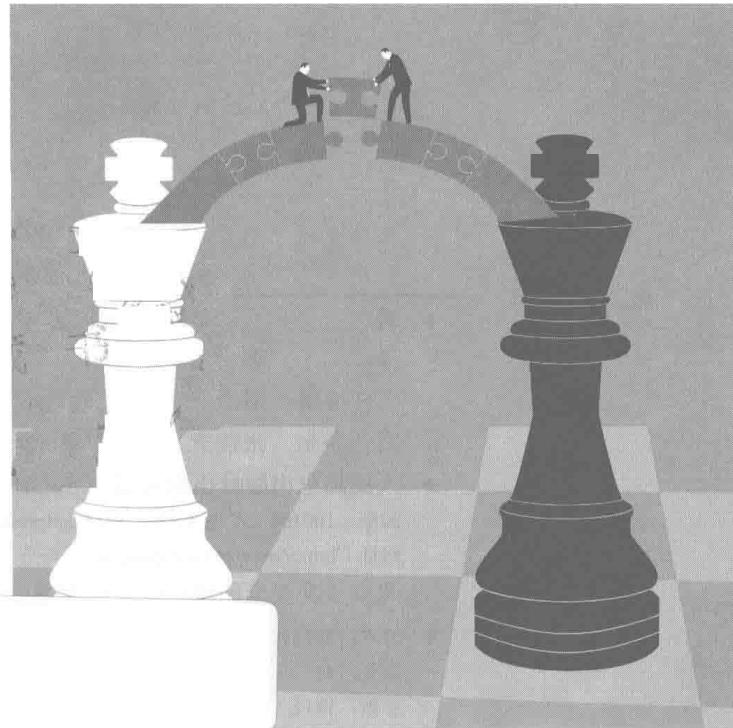
PEARSON

# C Primer Plus

## (第6版) 中文版

[美] | Stephen Prata | 著  
姜佑 | 译

*C Primer Plus  
Sixth Edition*



人民邮电出版社  
北京

## 图书在版编目（C I P）数据

C Primer Plus（第6版）中文版 /（美）普拉达  
(Prata, S.) 著；姜佑译。—北京：人民邮电出版社，  
2016.4（2016.6重印）  
ISBN 978-7-115-39059-2

I. ①C… II. ①普… ②姜… III. ①C语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2015)第084602号

## 版权声明

Authorized translation from the English language edition, entitled C Primer Plus (sixth edition), 9780321928429 by Stephen Prata, published by Pearson Education, Inc., publishing as Addison-Wesley, 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 POSTS & TELECOMMUNICATIONS PRESS Copyright © 2015.

本书封面贴有 Pearson Education (培生教育出版集团) 激光防伪标签。无标签者不得销售。

---

◆ 著 [美] Stephen Prata  
译 姜 佑  
责任编辑 傅道坤  
责任印制 张佳莹 焦志炜  
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号  
邮编 100164 电子邮件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
北京圣夫亚美印刷有限公司印刷  
◆ 开本：787×1092 1/16  
印张：47  
字数：1412千字 2016年4月第1版  
印数：8 001—13 000册 2016年6月北京第2次印刷  
著作权合同登记号 图字：01-2014-5617号

---

定价：89.00 元

读者服务热线：(010) 81055410 印装质量热线：(010) 81055316  
反盗版热线：(010) 81055315

# 目录

第1章 初识C语言 .....	1
1.1 C语言的起源 .....	1
1.2 选择C语言的理由 .....	1
1.2.1 设计特性 .....	1
1.2.2 高效性 .....	1
1.2.3 可移植性 .....	2
1.2.4 强大而灵活 .....	3
1.2.5 面向程序员 .....	3
1.2.6 缺点 .....	3
1.3 C语言的应用范围 .....	3
1.4 计算机能做什么 .....	4
1.5 高级计算机语言和编译器 .....	5
1.6 语言标准 .....	6
1.6.1 第1个ANSI/ISO C标准 .....	6
1.6.2 C99标准 .....	6
1.6.3 C11标准 .....	7
1.7 使用C语言的7个步骤 .....	7
1.7.1 第1步：定义程序的目标 .....	8
1.7.2 第2步：设计程序 .....	8
1.7.3 第3步：编写代码 .....	8
1.7.4 第4步：编译 .....	8
1.7.5 第5步：运行程序 .....	9
1.7.6 第6步：测试和调试程序 .....	9
1.7.7 第7步：维护和修改代码 .....	9
1.7.8 说明 .....	9
1.8 编程机制 .....	10
1.8.1 目标代码文件、可执行文件和库 .....	10
1.8.2 UNIX系统 .....	11
1.8.3 GNU编译器集合和LLVM项目 .....	13

1.8.4 Linux 系统 .....	13
1.8.5 PC 的命令行编译器 .....	14
1.8.6 集成开发环境（Windows） .....	14
1.8.7 Windows/Linux .....	15
1.8.8 Macintosh 中的 C .....	15
1.9 本书的组织结构 .....	15
1.10 本书的约定 .....	16
1.10.1 字体 .....	16
1.10.2 程序输出 .....	16
1.10.3 特殊元素 .....	17
1.11 本章小结 .....	17
1.12 复习题 .....	18
1.13 编程练习 .....	18
<b>第 2 章 C 语言概述 .....</b>	<b>19</b>
2.1 简单的 C 程序示例 .....	19
2.2 示例解释 .....	20
2.2.1 第 1 遍：快速概要 .....	21
2.2.2 第 2 遍：程序细节 .....	21
2.3 简单程序的结构 .....	28
2.4 提高程序可读性的技巧 .....	28
2.5 进一步使用 C .....	29
2.5.1 程序说明 .....	30
2.5.2 多条声明 .....	30
2.5.3 乘法 .....	30
2.5.4 打印多个值 .....	30
2.6 多个函数 .....	30
2.7 调试程序 .....	32
2.7.1 语法错误 .....	32
2.7.2 语义错误 .....	33
2.7.3 程序状态 .....	34
2.8 关键字和保留标识符 .....	34
2.9 关键概念 .....	35
2.10 本章小结 .....	35
2.11 复习题 .....	36

---

2.12 编程练习 .....	37
<b>第3章 数据和C .....</b>	<b>39</b>
3.1 示例程序 .....	39
3.2 变量与常量数据 .....	42
3.3 数据：数据类型关键字 .....	42
3.3.1 整数和浮点数 .....	43
3.3.2 整数 .....	43
3.3.3 浮点数 .....	43
3.4 C语言基本数据类型 .....	44
3.4.1 int类型 .....	44
3.4.2 其他整数类型 .....	47
3.4.3 使用字符：char类型 .....	50
3.4.4 _Bool类型 .....	54
3.4.5 可移植类型：stdint.h和inttypes.h .....	55
3.4.6 float、double和long double .....	56
3.4.7 复数和虚数类型 .....	60
3.4.8 其他类型 .....	60
3.4.9 类型大小 .....	62
3.5 使用数据类型 .....	63
3.6 参数和陷阱 .....	63
3.7 转义序列示例 .....	64
3.7.1 程序运行情况 .....	65
3.7.2 刷新输出 .....	65
3.8 关键概念 .....	66
3.9 本章小结 .....	66
3.10 复习题 .....	67
3.11 编程练习 .....	68
<b>第4章 字符串和格式化输入/输出 .....</b>	<b>71</b>
4.1 前导程序 .....	71
4.2 字符串简介 .....	72
4.2.1 char类型数组和null字符 .....	72
4.2.2 使用字符串 .....	73
4.2.3 strlen()函数 .....	74

4.3 常量和 C 预处理器.....	76
4.3.1 const 限定符 .....	78
4.3.2 明示常量 .....	78
4.4 printf() 和 scanf() .....	80
4.4.1 printf() 函数 .....	80
4.4.2 使用 printf() .....	81
4.4.3 printf() 的转换说明修饰符 .....	83
4.4.4 转换说明的意义 .....	87
4.4.5 使用 scanf() .....	92
4.4.6 printf() 和 scanf() 的*修饰符.....	95
4.4.7 printf() 的用法提示 .....	97
4.5 关键概念 .....	98
4.6 本章小结 .....	98
4.7 复习题 .....	99
4.8 编程练习 .....	100
<b>第 5 章 运算符、表达式和语句 .....</b>	<b>103</b>
5.1 循环简介 .....	103
5.2 基本运算符 .....	105
5.2.1 赋值运算符: = .....	105
5.2.2 加法运算符: + .....	107
5.2.3 减法运算符: - .....	107
5.2.4 符号运算符: -和+ .....	107
5.2.5 乘法运算符: * .....	108
5.2.6 除法运算符: / .....	110
5.2.7 运算符优先级 .....	110
5.2.8 优先级和求值顺序 .....	112
5.3 其他运算符 .....	113
5.3.1 sizeof 运算符和 size_t 类型 .....	113
5.3.2 求模运算符: % .....	114
5.3.3 递增运算符: ++ .....	115
5.3.4 递减运算符: -- .....	118
5.3.5 优先级 .....	118
5.3.6 不要自作聪明 .....	119
5.4 表达式和语句 .....	120

---

5.4.1 表达式 .....	120
5.4.2 语句 .....	120
5.4.3 复合语句(块) .....	123
5.5 类型转换 .....	124
5.6 带参数的函数 .....	127
5.7 示例程序 .....	129
5.8 关键概念 .....	130
5.9 本章小结 .....	130
5.10 复习题 .....	131
5.11 编程练习 .....	134

## 第6章 C 控制语句：循环.....137

6.1 再探 while 循环 .....	137
6.1.1 程序注释 .....	138
6.1.2 C 风格读取循环 .....	139
6.2 while 语句 .....	140
6.2.1 终止 while 循环 .....	140
6.2.2 何时终止循环 .....	141
6.2.3 while：入口条件循环 .....	141
6.2.4 语法要点 .....	141
6.3 用关系运算符和表达式比较大小 .....	143
6.3.1 什么是真 .....	144
6.3.2 其他真值 .....	145
6.3.3 真值的问题 .....	146
6.3.4 新的 _Bool 类型 .....	147
6.3.5 优先级和关系运算符 .....	148
6.4 不确定循环和计数循环 .....	150
6.5 for 循环 .....	151
6.6 其他赋值运算符：+=、-=、*=、/=、% .....	155
6.7 逗号运算符 .....	156
6.8 出口条件循环：do while .....	159
6.9 如何选择循环 .....	161
6.10 嵌套循环 .....	162
6.10.1 程序分析 .....	163
6.10.2 嵌套变式 .....	163

6.11 数组简介 .....	164
6.12 使用函数返回值的循环示例.....	166
6.12.1 程序分析 .....	168
6.12.2 使用带返回值的函数.....	169
6.13 关键概念 .....	169
6.14 本章小结 .....	170
6.15 复习题 .....	170
6.16 编程练习 .....	174
<b>第7章 C 控制语句：分支和跳转.....</b>	<b>177</b>
7.1 if 语句 .....	177
7.2 if else 语句.....	179
7.2.1 另一个示例：介绍 getchar() 和 putchar() .....	180
7.2.2 ctype.h 系列的字符函数 .....	182
7.2.3 多重选择 else if .....	184
7.2.4 else 与 if 配对 .....	186
7.2.5 多层嵌套的 if 语句 .....	187
7.3 逻辑运算符 .....	190
7.3.1 备选拼写：iso646.h 头文件 .....	191
7.3.2 优先级 .....	192
7.3.3 求值顺序 .....	192
7.3.4 范围 .....	193
7.4 一个统计单词的程序 .....	194
7.5 条件运算符：?: .....	196
7.6 循环辅助：continue 和 break .....	198
7.6.1 continue 语句 .....	198
7.6.2 break 语句 .....	200
7.7 多重选择：switch 和 break .....	202
7.7.1 switch 语句 .....	204
7.7.2 只读每行的首字符 .....	205
7.7.3 多重标签 .....	206
7.7.4 switch 和 if else .....	208
7.8 goto 语句.....	208
7.9 关键概念 .....	211
7.10 本章小结 .....	211

7.11 复习题 .....	212
7.12 编程练习 .....	214
<b>第8章 字符输入/输出和输入验证 .....</b>	<b>217</b>
8.1 单字符 I/O: <code>getchar()</code> 和 <code>putchar()</code> .....	217
8.2 缓冲区 .....	218
8.3 结束键盘输入 .....	219
8.3.1 文件、流和键盘输入 .....	219
8.3.2 文件结尾 .....	220
8.4 重定向和文件 .....	222
8.5 创建更友好的用户界面 .....	226
8.5.1 使用缓冲输入 .....	226
8.5.2 混合数值和字符输入 .....	228
8.6 输入验证 .....	230
8.6.1 分析程序 .....	234
8.6.2 输入流和数字 .....	234
8.7 菜单浏览 .....	235
8.7.1 任务 .....	235
8.7.2 使执行更顺利 .....	235
8.7.3 混合字符和数值输入 .....	237
8.8 关键概念 .....	240
8.9 本章小结 .....	240
8.10 复习题 .....	241
8.11 编程练习 .....	241
<b>第9章 函数 .....</b>	<b>243</b>
9.1 复习函数 .....	243
9.1.1 创建并使用简单函数 .....	244
9.1.2 分析程序 .....	245
9.1.3 函数参数 .....	247
9.1.4 定义带形式参数的函数 .....	248
9.1.5 声明带形式参数函数的原型 .....	249
9.1.6 调用带实际参数的函数 .....	249
9.1.7 黑盒视角 .....	250
9.1.8 使用 <code>return</code> 从函数中返回值 .....	250

9.1.9 函数类型 .....	252
9.2 ANSI C 函数原型 .....	253
9.2.1 问题所在 .....	253
9.2.2 ANSI 的解决方案 .....	254
9.2.3 无参数和未指定参数 .....	255
9.2.4 函数原型的优点 .....	256
9.3 递归 .....	256
9.3.1 演示递归 .....	256
9.3.2 递归的基本原理 .....	258
9.3.3 尾递归 .....	258
9.3.4 递归和倒序计算 .....	260
9.3.5 递归的优缺点 .....	262
9.4 编译多源代码文件的程序 .....	262
9.4.1 UNIX .....	263
9.4.2 Linux .....	263
9.4.3 DOS 命令行编译器 .....	263
9.4.4 Windows 和苹果的 IDE 编译器 .....	263
9.4.5 使用头文件 .....	263
9.5 查找地址: &运算符 .....	267
9.6 更改主调函数中的变量 .....	268
9.7 指针简介 .....	269
9.7.1 间接运算符: * .....	270
9.7.2 声明指针 .....	270
9.7.3 使用指针在函数间通信 .....	271
9.8 关键概念 .....	274
9.9 本章小结 .....	275
9.10 复习题 .....	275
9.11 编程练习 .....	276
<b>第 10 章 数组和指针 .....</b>	<b>277</b>
10.1 数组 .....	277
10.1.1 初始化数组 .....	277
10.1.2 指定初始化器 (C99) .....	281
10.1.3 给数组元素赋值 .....	282
10.1.4 数组边界 .....	282

10.1.5 指定数组的大小 .....	284
10.2 多维数组 .....	284
10.2.1 初始化二维数组 .....	287
10.2.2 其他多维数组 .....	288
10.3 指针和数组 .....	288
10.4 函数、数组和指针 .....	290
10.4.1 使用指针形参 .....	293
10.4.2 指针表示法和数组表示法 .....	294
10.5 指针操作 .....	295
10.6 保护数组中的数据 .....	298
10.6.1 对形式参数使用 const .....	299
10.6.2 const 的其他内容 .....	300
10.7 指针和多维数组 .....	302
10.7.1 指向多维数组的指针 .....	304
10.7.2 指针的兼容性 .....	305
10.7.3 函数和多维数组 .....	306
10.8 变长数组 (VLA) .....	309
10.9 复合字面量 .....	312
10.10 关键概念 .....	314
10.11 本章小结 .....	315
10.12 复习题 .....	316
10.13 编程练习 .....	317
<b>第 11 章 字符串和字符串函数 .....</b>	<b>321</b>
11.1 表示字符串和字符串 I/O .....	321
11.1.1 在程序中定义字符串 .....	322
11.1.2 指针和字符串 .....	328
11.2 字符串输入 .....	329
11.2.1 分配空间 .....	329
11.2.2 不幸的 gets() 函数 .....	330
11.2.3 gets() 的替代品 .....	331
11.2.4 scanf() 函数 .....	336
11.3 字符串输出 .....	337
11.3.1 puts() 函数 .....	338
11.3.2 fputs() 函数 .....	339

11.3.3 printf() 函数 .....	339
11.4 自定义输入/输出函数 .....	340
11.5 字符串函数 .....	342
11.5.1 strlen() 函数 .....	342
11.5.2 strcat() 函数 .....	343
11.5.3 strncat() 函数 .....	345
11.5.4 strcmp() 函数 .....	346
11.5.5 strcpy() 和 strncpy() 函数 .....	351
11.5.6 sprintf() 函数 .....	356
11.5.7 其他字符串函数 .....	357
11.6 字符串示例：字符串排序 .....	359
11.6.1 排序指针而非字符串 .....	360
11.6.2 选择排序算法 .....	361
11.7 ctype.h 字符函数和字符串 .....	362
11.8 命令行参数 .....	363
11.8.1 集成环境中的命令行参数 .....	365
11.8.2 Macintosh 中的命令行参数 .....	365
11.9 把字符串转换为数字 .....	365
11.10 关键概念 .....	368
11.11 本章小结 .....	368
11.12 复习题 .....	369
11.13 编程练习 .....	371
<b>第 12 章 存储类别、链接和内存管理 .....</b>	<b>373</b>
12.1 存储类别 .....	373
12.1.1 作用域 .....	374
12.1.2 链接 .....	376
12.1.3 存储期 .....	376
12.1.4 自动变量 .....	377
12.1.5 寄存器变量 .....	380
12.1.6 块作用域的静态变量 .....	381
12.1.7 外部链接的静态变量 .....	382
12.1.8 内部链接的静态变量 .....	386
12.1.9 多文件 .....	386
12.1.10 存储类别说明符 .....	387

12.1.11 存储类别和函数.....	389
12.1.12 存储类别的选择.....	389
12.2 随机数函数和静态变量 .....	390
12.3 掷骰子 .....	393
12.4 分配内存: <code>malloc()</code> 和 <code>free()</code> .....	396
12.4.1 <code>free()</code> 的重要性 .....	399
12.4.2 <code>calloc()</code> 函数 .....	400
12.4.3 动态内存分配和变长数组 .....	400
12.4.4 存储类别和动态内存分配 .....	401
12.5 ANSI C 类型限定符 .....	402
12.5.1 <code>const</code> 类型限定符 .....	403
12.5.2 <code>volatile</code> 类型限定符 .....	404
12.5.3 <code>restrict</code> 类型限定符 .....	405
12.5.4 <code>_Atomic</code> 类型限定符 (C11) .....	406
12.5.5 旧关键字的新位置 .....	406
12.6 关键概念 .....	407
12.7 本章小结 .....	407
12.8 复习题 .....	408
12.9 编程练习 .....	409
<b>第 13 章 文件输入/输出 .....</b>	<b>413</b>
13.1 与文件进行通信 .....	413
13.1.1 文件是什么 .....	413
13.1.2 文本模式和二进制模式 .....	413
13.1.3 I/O 的级别 .....	415
13.1.4 标准文件 .....	415
13.2 标准 I/O .....	415
13.2.1 检查命令行参数 .....	416
13.2.2 <code>fopen()</code> 函数 .....	416
13.2.3 <code>getc()</code> 和 <code>putc()</code> 函数 .....	417
13.2.4 文件结尾 .....	418
13.2.5 <code>fclose()</code> 函数 .....	419
13.2.6 指向标准文件的指针 .....	419
13.3 一个简单的文件压缩程序 .....	419
13.4 文件 I/O: <code>fprintf()</code> 、 <code>fscanf()</code> 、 <code>fgets()</code> 和 <code>fputs()</code> .....	421

13.4.1 <code>fprintf()</code> 和 <code>scanf()</code> 函数	421
13.4.2 <code>fgets()</code> 和 <code>fputs()</code> 函数	422
13.5 随机访问: <code>fseek()</code> 和 <code>fpos()</code>	423
13.5.1 <code>fseek()</code> 和 <code>fpos()</code> 的工作原理	424
13.5.2 二进制模式和文本模式	425
13.5.3 可移植性	425
13.5.4 <code>fgetpos()</code> 和 <code>fsetpos()</code> 函数	426
13.6 标准 I/O 的机理	426
13.7 其他标准 I/O 函数	427
13.7.1 <code>int ungetc(int c, FILE *fp)</code> 函数	427
13.7.2 <code>int fflush()</code> 函数	428
13.7.3 <code>int setvbuf()</code> 函数	428
13.7.4 二进制 I/O: <code>fread()</code> 和 <code>fwrite()</code>	428
13.7.5 <code>size_t fwrite()</code> 函数	429
13.7.6 <code>size_t fread()</code> 函数	430
13.7.7 <code>int feof(FILE *fp)</code> 和 <code>int ferror(FILE *fp)</code> 函数	430
13.7.8 一个程序示例	430
13.7.9 用二进制 I/O 进行随机访问	433
13.8 关键概念	435
13.9 本章小结	435
13.10 复习题	435
13.11 编程练习	437
<b>第 14 章 结构和其他数据形式</b>	<b>439</b>
14.1 示例问题: 创建图书目录	439
14.2 建立结构声明	441
14.3 定义结构变量	441
14.3.1 初始化结构	442
14.3.2 访问结构成员	443
14.3.3 结构的初始化器	443
14.4 结构数组	444
14.4.1 声明结构数组	446
14.4.2 标识结构数组的成员	447
14.4.3 程序讨论	447
14.5 嵌套结构	448

---

14.6 指向结构的指针 .....	449
14.6.1 声明和初始化结构指针 .....	450
14.6.2 用指针访问成员 .....	451
14.7 向函数传递结构的信息 .....	451
14.7.1 传递结构成员 .....	451
14.7.2 传递结构的地址 .....	452
14.7.3 传递结构 .....	453
14.7.4 其他结构特性 .....	454
14.7.5 结构和结构指针的选择 .....	458
14.7.6 结构中的字符数组和字符指针 .....	458
14.7.7 结构、指针和 malloc() .....	459
14.7.8 复合字面量和结构 (C99) .....	462
14.7.9 伸缩型数组成员 (C99) .....	463
14.7.10 匿名结构 (C11) .....	465
14.7.11 使用结构数组的函数 .....	466
14.8 把结构内容保存到文件中 .....	467
14.8.1 保存结构的程序示例 .....	468
14.8.2 程序要点 .....	470
14.9 链式结构 .....	471
14.10 联合简介 .....	472
14.10.1 使用联合 .....	472
14.10.2 匿名联合 (C11) .....	473
14.11 枚举类型 .....	474
14.11.1 enum 常量 .....	475
14.11.2 默认值 .....	475
14.11.3 赋值 .....	475
14.11.4 enum 的用法 .....	476
14.11.5 共享名称空间 .....	477
14.12 typedef 简介 .....	478
14.13 其他复杂的声明 .....	479
14.14 函数和指针 .....	481
14.15 关键概念 .....	487
14.16 本章小结 .....	487
14.17 复习题 .....	488
14.18 编程练习 .....	490

第 15 章 位操作 .....	493
15.1 二进制数、位和字节 .....	493
15.1.1 二进制整数 .....	494
15.1.2 有符号整数 .....	494
15.1.3 二进制浮点数 .....	495
15.2 其他进制数 .....	495
15.2.1 八进制 .....	495
15.2.2 十六进制 .....	496
15.3 C 按位运算符 .....	496
15.3.1 按位逻辑运算符 .....	497
15.3.2 用法：掩码 .....	498
15.3.3 用法：打开位（设置位） .....	498
15.3.4 用法：关闭位（清空位） .....	499
15.3.5 用法：切换位 .....	499
15.3.6 用法：检查位的值 .....	500
15.3.7 移位运算符 .....	500
15.3.8 编程示例 .....	501
15.3.9 另一个例子 .....	503
15.4 位字段 .....	505
15.4.1 位字段示例 .....	506
15.4.2 位字段和按位运算符 .....	509
15.5 对齐特性 (C11) .....	515
15.6 关键概念 .....	516
15.7 本章小结 .....	516
15.8 复习题 .....	517
15.9 编程练习 .....	518
第 16 章 C 预处理器和 C 库 .....	521
16.1 翻译程序的第一步 .....	521
16.2 明示常量：#define .....	522
16.2.1 记号 .....	525
16.2.2 重定义常量 .....	525
16.3 在#define 中使用参数 .....	525
16.3.1 用宏参数创建字符串：#运算符 .....	527

---

16.3.2 预处理器黏合剂: ##运算符	528
16.3.3 变参宏: ... 和 __VA_ARGS__	529
16.4 宏和函数的选择	530
16.5 文件包含: #include	531
16.5.1 头文件示例	531
16.5.2 使用头文件	533
16.6 其他指令	534
16.6.1 #undef 指令	534
16.6.2 从 C 预处理器角度看已定义	534
16.6.3 条件编译	535
16.6.4 预定义宏	539
16.6.5 #line 和#error	540
16.6.6 #pragma	540
16.6.7 泛型选择 (C11)	541
16.7 内联函数 (C99)	542
16.8 __Noreturn 函数 (C11)	544
16.9 C 库	544
16.9.1 访问 C 库	544
16.9.2 使用库描述	545
16.10 数学库	546
16.10.1 三角问题	547
16.10.2 类型变体	548
16.10.3 tgmath.h 库 (C99)	550
16.11 通用工具库	550
16.11.1 exit() 和 atexit() 函数	550
16.11.2 qsort() 函数	552
16.12 断言库	556
16.12.1 assert 的用法	556
16.12.2 _Static_assert (C11)	557
16.13 string.h 库中的 memcpy() 和 memmove()	558
16.14 可变参数: stdarg.h	560
16.15 关键概念	562
16.16 本章小结	562
16.17 复习题	562
16.18 编程练习	563

第 17 章 高级数据表示 .....	567
17.1 研究数据表示 .....	567
17.2 从数组到链表 .....	570
17.2.1 使用链表 .....	572
17.2.2 反思 .....	576
17.3 抽象数据类型 (ADT) .....	576
17.3.1 建立抽象 .....	577
17.3.2 建立接口 .....	578
17.3.3 使用接口 .....	581
17.3.4 实现接口 .....	583
17.4 队列 ADT .....	589
17.4.1 定义队列抽象数据类型 .....	590
17.4.2 定义一个接口 .....	590
17.4.3 实现接口数据表示 .....	591
17.4.4 测试队列 .....	598
17.5 用队列进行模拟 .....	600
17.6 链表和数组 .....	605
17.7 二叉查找树 .....	608
17.7.1 二叉树 ADT .....	608
17.7.2 二叉查找树接口 .....	609
17.7.3 二叉树的实现 .....	611
17.7.4 使用二叉树 .....	624
17.7.5 树的思想 .....	628
17.8 其他说明 .....	629
17.9 关键概念 .....	630
17.10 本章小结 .....	630
17.11 复习题 .....	630
17.12 编程练习 .....	631
附录 A 复习题答案 .....	633
附录 B 参考资料 .....	665
B.1 参考资料 I: 补充阅读 .....	665
B.2 参考资料 II: C 运算符 .....	667

B.3 参考资料 III: 基本类型和存储类别.....	671
B.4 参考资料 IV: 表达式、语句和程序流.....	675
B.5 参考资料 V: 新增 C99 和 C11 的 ANSI C 库.....	679
B.6 参考资料 VI: 扩展的整数类型.....	714
B.7 参考资料 VII: 扩展字符支持.....	716
B.8 参考资料 VIII: C99/C11 数值计算增强.....	720
B.9 参考资料 IX: C 和 C++的区别.....	726

# 第1章

## 初识 C 语言

本章介绍以下内容：

- C 的历史和特性
- 编写程序的步骤
- 编译器和链接器的一些知识
- C 标准

欢迎来到 C 语言的世界。C 是一门功能强大的专业化编程语言，深受业余编程爱好者和专业程序员的喜爱。本章为读者学习这一强大而流行的语言打好基础，并介绍几种开发 C 程序最可能使用的环境。

我们先来了解 C 语言的起源和一些特性，包括它的优缺点。然后，介绍编程的起源并探讨一些编程的基本原则。最后，讨论如何在一些常见系统中运行 C 程序。

### 1.1 C 语言的起源

1972 年，贝尔实验室的丹尼斯·里奇（Dennis Ritchie）和肯·汤普逊（Ken Thompson）在开发 UNIX 操作系统时设计了 C 语言。然而，C 语言不完全是里奇突发奇想而来，他是在 B 语言（汤普逊发明）的基础上进行设计。至于 B 语言的起源，那是另一个故事。C 语言设计的初衷是将其作为程序员使用的一种编程工具，因此，其主要目标是成为有用的语言。

虽然绝大多数语言都以实用为目标，但是通常也会考虑其他方面。例如，Pascal 的主要目标是为更好地学习编程原理提供扎实的基础；而 BASIC 的主要目标是开发出类似英文的语言，让不熟悉计算机的学生轻松学习编程。这些目标固然很重要，但是随着计算机的迅猛发展，它们已经不是主流语言。然而，最初为程序员设计开发的 C 语言，现在已成为首选的编程语言之一。

### 1.2 选择 C 语言的理由

在过去 40 多年里，C 语言已成为最重要、最流行的编程语言之一。它的成长归功于使用过的人都对它很满意。过去 20 多年里，虽然许多人都从 C 语言转而使用其他编程语言（如，C++、Objective C、Java 等），但是 C 语言仍凭借自身实力在众多语言中脱颖而出。在学习 C 语言的过程中，会发现它的许多优点（见图 1.1）。下面，我们来看看其中较为突出的几点。

#### 1.2.1 设计特性

C 是一门流行的语言，融合了计算机科学理论和实践的控制特性。C 语言的设计理念让用户能轻松地完成自顶向下的规划、结构化编程和模块化设计。因此，用 C 语言编写的程序更易懂、更可靠。

#### 1.2.2 高效性

C 是高效的语言。在设计上，它充分利用了当前计算机的优势，因此 C 程序相对更紧凑，而且运行速

度很快。实际上，C语言具有通常是汇编语言才具有的微调控制能力（汇编语言是为特殊的中央处理单元设计的一系列内部指令，使用助记符来表示；不同的CPU系列使用不同的汇编语言），可以根据具体情况微调程序以获得最大运行速度或最有效地使用内存。



图1.1 C语言的优点

### 1.2.3 可移植性

C是可移植的语言。这意味着，在一种系统中编写的C程序稍作修改或不修改就能在其他系统运行。如需修改，也只需简单更改主程序头文件中的少许项即可。大部分语言都希望成为可移植语言，但是，如果经历过把IBM PC BASIC程序转换成苹果BASIC（两者是近亲），或者在UNIX系统中运行IBM大型机的FORTRAN程序的人都知道，移植是最麻烦的事。C语言是可移植方面的佼佼者。从8位微处理器到克雷超级计算机，许多计算机体系结构都可以使用C编译器（C编译器是把C代码转换成计算机内部指令的程序）。但是要注意，程序中针对特殊硬件设备（如，显示监视器）或操作系统特殊功能（如，Windows 8或OS X）编写的部分，通常是不可移植的。

由于C语言与UNIX关系密切，UNIX系统通常会将C编译器作为软件包的一部分。安装Linux时，通常也会安装C编译器。供个人计算机使用的C编译器很多，运行各种版本的Windows和Macintosh（即，Mac）的PC都能找到合适的C编译器。因此，无论是使用家庭计算机、专业工作站，还是大型机，都能找到针对特定系统的C编译器。

## 1.2.4 强大而灵活

C 语言功能强大且灵活（计算机领域经常使用这两个词）。例如，功能强大且灵活的 UNIX 操作系统，大部分是用 C 语言写的；其他语言（如，FORTRAN、Perl、Python、Pascal、LISP、Logo、BASIC）的许多编译器和解释器都是用 C 语言编写的。因此，在 UNIX 机上使用 FORTRAN 时，最终是由 C 程序生成最后的可执行程序。C 程序可以用于解决物理学和工程学的问题，甚至可用于制作电影的动画特效。

## 1.2.5 面向程序员

C 语言是为了满足程序员的需求而设计的，程序员利用 C 可以访问硬件、操控内存中的位。C 语言有丰富的运算符，能让程序员简洁地表达自己的意图。C 没有 Pascal 严谨，但是却比 C++ 的限制多。这样的灵活性既是优点也是缺点。优点是，许多任务用 C 来处理都非常简洁（如，转换数据的格式）；缺点是，你可能会犯一些莫名其妙的错误，这些错误不可能在其他语言中出现。C 语言在提供更多自由的同时，也让使用者承担了更大的责任。

另外，大多数 C 实现都有一个大型的库，包含众多有用的 C 函数。这些函数用于处理程序员经常需要解决的问题。

## 1.2.6 缺点

人无完人，金无足赤。C 语言也有一些缺点。例如，前面提到的，要享受用 C 语言自由编程的乐趣，就必须承担更多的责任。特别是，C 语言使用指针，而涉及指针的编程错误往往难以察觉。有句话说的好：想拥有自由就必须时刻保持警惕。

C 语言紧凑简洁，结合了大量的运算符。正因如此，我们也可以编写出让人极其费解的代码。虽然没必要强迫自己编写晦涩的代码，但是有兴趣写写也无妨。试问，除 C 语言外还为哪种语言举办过年度混乱代码大赛<sup>1</sup>？

瑕不掩瑜，C 语言的优点比缺点多很多。我们不想在这里多费笔墨，还是来聊聊 C 语言的其他话题。

## 1.3 C 语言的应用范围

早在 20 世纪 80 年代，C 语言就已经成为小型计算机（UNIX 系统）使用的主流语言。从那以后，C 语言的应用范围扩展到微型机（个人计算机）和大型机（庞然大物）。如图 1.2 所示，许多软件公司都用 C 语言来开发文字处理程序、电子表格、编译器和其他产品，因为用 C 语言编写的程序紧凑而高效。更重要的是，C 程序很方便修改，而且移植到新型号的计算机中也没什么问题。

无论是软件公司、经验丰富的 C 程序员，还是其他用户，都能从 C 语言中受益。越来越多的计算机用户已转而求助 C 语言解决一些安全问题。不一定非得是计算机专家也能使用 C 语言。

20 世纪 90 年代，许多软件公司开始改用 C++ 来开发大型的编程项目。C++ 在 C 语言的基础上嫁接了面向对象编程工具（面向对象编程是一门哲学，它通过对语言建模来适应问题，而不是对问题建模以适应语言）。C++几乎是 C 的超集，这意味着任何 C 程序差不多就是一个 C++ 程序。学习 C 语言，也相当于学习了许多 C++ 的知识。

<sup>1</sup> 国际 C 语言混乱代码大赛（IOCCC，The International Obfuscated C Code Contest）。这是一项国际编程赛事，从 1984 年开始，每年举办一次（1997、1999、2002、2003 和 2006 年除外），目的是写出最有创意且最让人难以理解的 C 语言代码。——译者注

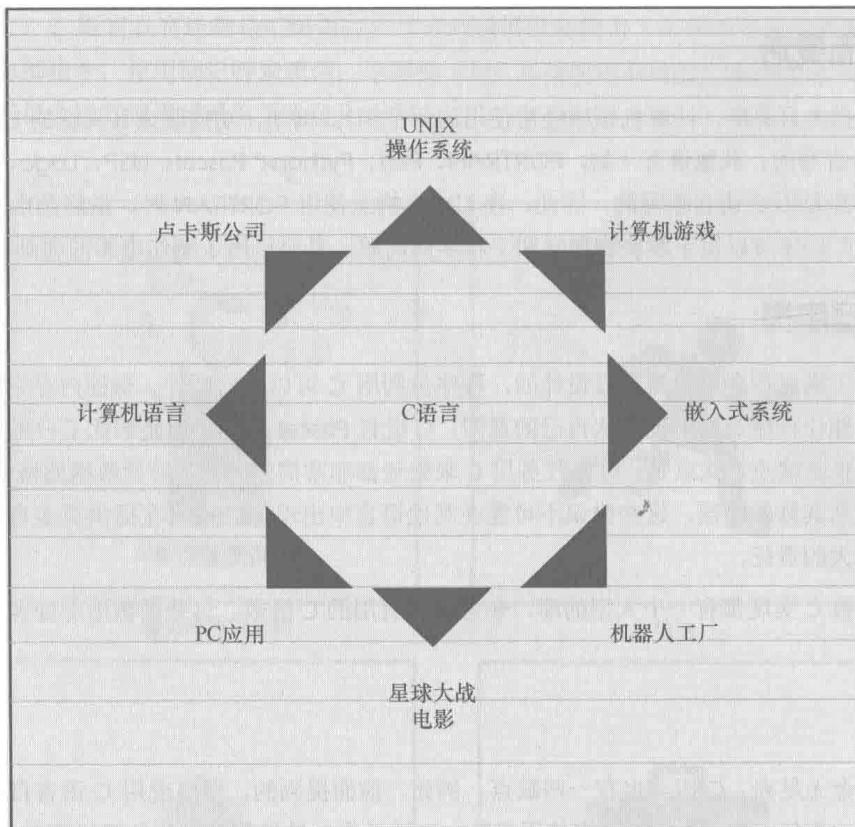


图 1.2 C 语言的应用范围

虽然这些年来 C++ 和 JAVA 非常流行，但是 C 语言仍是软件业中的核心技能。在最想具备的技能中，C 语言通常位居前十。特别是，C 语言已成为嵌入式系统编程的流行语言。也就是说，越来越多的汽车、照相机、DVD 播放机和其他现代化设备的微处理器都用 C 语言进行编程。除此之外，C 语言还从长期被 FORTRAN 独占的科学编程领域分得一杯羹。最终，作为开发操作系统的卓越语言，C 在 Linux 开发中扮演着极其重要的角色。因此，在进入 21 世纪的第 2 个 10 年中，C 语言仍然保持着强劲的势头。

简而言之，C 语言是最重要的编程语言之一，将来也是如此。如果你想拿下一份编程的工作，被问到是否会 C 语言时，最好回答“是”。

## 1.4 计算机能做什么

在学习如何用 C 语言编程之前，最好先了解一下计算机的工作原理。这些知识有助于你理解用 C 语言编写程序和运行 C 程序时所发生的事情之间有什么联系。

现代的计算机由多种部件构成。中央处理单元 (CPU) 承担绝大部分的运算工作。随机存取内存 (RAM) 是存储程序和文件的工作区；而永久内存存储设备（过去一般指机械硬盘，现在还包括固态硬盘）即使在关闭计算机后，也不会丢失之前储存的程序和文件。另外，还有各种外围设备（如，键盘、鼠标、触摸屏、监视器）提供人与计算机之间的交互。CPU 负责处理程序，接下来我们重点讨论它的工作原理。

CPU 的工作非常简单，至少从以下简短的描述中看是这样。它从内存中获取并执行一条指令，然后再从内存中获取并执行下一条指令，诸如此类（一个吉赫兹的 CPU 一秒钟能重复这样的操作大约十亿次，因此，CPU 能以惊人的速度从事枯燥的工作）。CPU 有自己的小工作区——由若干个寄存器组成，每个寄存器都可以储存一个数字。一个寄存器储存下一条指令的内存地址，CPU 使用该地址来获取和更新下一条指令。在获取指令后，CPU 在另一个寄存器中储存该指令，并更新第 1 个寄存器储存下一条指令的地址。CPU

能理解的指令有限（这些指令的集合叫作指令集）。而且，这些指令相当具体，其中的许多指令都是用于请求计算机把一个数字从一个位置移动到另一个位置。例如，从内存移动到寄存器。

下面介绍两个有趣的知识。其一，储存在计算机中的所有内容都是数字。计算机以数字形式储存数字和字符（如，在文本文档中使用的字母）。每个字符都有一个数码。计算机载入寄存器的指令也以数字形式储存，指令集中的每条指令都有一个数码。其二，计算机程序最终必须以数字指令码（即，机器语言）来表示。

简而言之，计算机的工作原理是：如果希望计算机做某些事，就必须为其提供特殊的指令列表（程序），确切地告诉计算机要做的事以及如何做。你必须用计算机能直接明白的语言（机器语言）创建程序。这是一项繁琐、乏味、费力的任务。计算机要完成诸如两数相加这样简单的事，就得分成类似以下几个步骤。

1. 从内存位置 2000 上把一个数字拷贝到寄存器 1。
2. 从内存位置 2004 上把另一个数字拷贝到寄存器 2。
3. 把寄存器 2 中的内容与寄存器 1 中的内容相加，把结果储存在寄存器 1 中。
4. 把寄存器 1 中的内容拷贝到内存位置 2008。

而你要做的是，必须用数码来表示以上的每个步骤！

如果以这种方式编写程序很合你的意，那不得不说抱歉，因为用机器语言编程的黄金时代已一去不复返。但是，如果你对有趣的事情比较感兴趣，不妨试试高级编程语言。

## 1.5 高级计算机语言和编译器

高级编程语言（如，C）以多种方式简化了编程工作。首先，不必用数码表示指令；其次，使用的指令更贴近你如何想这个问题，而不是类似计算机那样繁琐的步骤。使用高级编程语言，可以在更抽象的层面表达你的想法，不用考虑 CPU 在完成任务时具体需要哪些步骤。例如，对于两数相加，可以这样写：

```
total = mine + yours;
```

对我们而言，光看这行代码就知道要计算机做什么；而看用机器语言写成的等价指令（多条以数码形式表现的指令）则费劲得多。但是，对计算机而言却恰恰相反。在计算机看来，高级指令就是一堆无法理解的无用数据。编译器在这里派上了用场。编译器是把高级语言程序翻译成计算机能理解的机器语言指令集的程序。程序员进行高级思维活动，而编译器则负责处理冗长乏味的细节工作。

编译器还有一个优势。一般而言，不同 CPU 制造商使用的指令系统和编码格式不同。例如，用 Intel Core i7（英特尔酷睿 i7）CPU 编写的机器语言程序对于 ARM Cortex-A57 CPU 而言什么都不是。但是，可以找到与特定类型 CPU 匹配的编译器。因此，使用合适的编译器或编译器集，便可把一种高级语言程序转换成供各种不同类型 CPU 使用的机器语言程序。一旦解决了一个编程问题，便可让编译器集翻译成不同 CPU 使用的机器语言。

简而言之，高级语言（如 C、Java、Pascal）以更抽象的方式描述行为，不受限于特定 CPU 或指令集。而且，高级语言简单易学，用高级语言编程比用机器语言编程容易得多。

1964 年，控制数据公司（Control Data Corporation）研制出了 CDC 6600 计算机。这台庞然大物是世界上首台超级计算机，当时的售价是 600 万美元。它是高能核物理研究的首选。然而，现在的普通智能手机在计算能力和内存方面都超过它数百倍，而且能看视频，放音乐。

1964 年，在工程和科学领域的主流编程语言是 FORTRAN。虽然编程语言不如硬件发展那么突飞猛进，但是也发生了很大变化。为了应对越来越大型的编程项目，语言先后为结构化编程和面向对象编程提供了更多的支持。随着时间的推移，不仅新语言层出不穷，而且现有语言也会发生变化。

## 1.6 语言标准

目前，有许多 C 实现可用。在理想情况下，编写 C 程序时，假设该程序中未使用机器特定的编程技术，那么它的运行情况在任何实现中都应该相同。要在实践中做到这一点，不同的实现要遵循同一个标准。

C 语言发展之初，并没有所谓的 C 标准。1987 年，布莱恩·柯林汉（Brian Kernighan）和丹尼斯·里奇（Dennis Ritchie）合著的 *The C Programming Language*（《C 语言程序设计》）第 1 版是公认的 C 标准，通常称之为 K&R C 或经典 C。特别是，该书中的附录中的“C 语言参考手册”已成为实现 C 的指导标准。例如，编译器都声称提供完整的 K&R 实现。虽然这本书中的附录定义了 C 语言，但却没有定义 C 库。与大多数语言不同的是，C 语言比其他语言更依赖库，因此需要一个标准库。实际上，由于缺乏官方标准，UNIX 实现提供的库已成为了标准库。

### 1.6.1 第1个ANSI/ISO C 标准

随着 C 的不断发展，越来越广泛地应用于更多系统中，C 社区意识到需要一个更全面、更新颖、更严格的标准。鉴于此，美国国家标准协会（ANSI）于 1983 年组建了一个委员会（X3J11），开发了一套新标准，并于 1989 年正式公布。该标准（ANSI C）定义了 C 语言和 C 标准库。国际标准化组织于 1990 年采用了这套 C 标准（ISO C）。ISO C 和 ANSI C 是完全相同的标准。ANSI/ISO 标准的最终版本通常叫作 C89（因为 ANSI 于 1989 年批准该标准）或 C90（因为 ISO 于 1990 年批准该标准）。另外，由于 ANSI 先公布 C 标准，因此业界人士通常使用 ANSI C。

在该委员会制定的指导原则中，最有趣的可能是：保持 C 的精神。委员会在表述这一精神时列出了以下几点：

- 信任程序员；
- 不要妨碍程序员做需要做的事；
- 保持语言精练简单；
- 只提供一种方法执行一项操作；
- 让程序运行更快，即使不能保证其可移植性。

在最后一点上，标准委员会的用意是：作为实现，应该针对目标计算机来定义最合适的某特定操作，而不是强加一个抽象、统一的定义。在学习 C 语言过程中，许多方面都反映了这一哲学思想。

### 1.6.2 C99 标准

1994 年，ANSI/ISO 联合委员会（C9X 委员会）开始修订 C 标准，最终发布了 C99 标准。该委员会遵循了最初 C90 标准的原则，包括保持语言的精练简单。委员会的用意不是在 C 语言中添加新特性，而是为了达到新的目标。第 1 个目标是，支持国际化编程。例如，提供多种方法处理国际字符集。第 2 个目标是，“调整现有实践致力于解决明显的缺陷”。因此，在遇到需要将 C 移至 64 位处理器时，委员会根据现实生活中处理问题的经验来添加标准。第 3 个目标是，为适应科学和工程项目中的关键数值计算，提高 C 的适应性，让 C 比 FORTRAN 更有竞争力。

这 3 点（国际化、弥补缺陷和提高计算的实用性）是主要的修订目标。在其他方面的改变则更为保守，例如，尽量与 C90、C++兼容，让语言在概念上保持简单。用委员会的话说：“……委员会很满意让 C++ 成为大型、功能强大的语言”。

C99 的修订保留了 C 语言的精髓，C 仍是一门简洁高效的语言。本书指出了许多 C99 修改的地方。虽然该标准已发布了很长时间，但并非所有的编译器都完全实现 C99 的所有改动。因此，你可能发现 C99 的一些改动在自己的系统中不可用，或者只有改变编译器的设置才可用。

### 1.6.3 C11 标准

维护标准任重道远。标准委员会在 2007 年承诺 C 标准的下一个版本是 C1X，2011 年终于发布了 C11 标准。此次，委员会提出了一些新的指导原则。出于对当前编程安全的担忧，不那么强调“信任程序员”目标了。而且，供应商并未像对 C90 那样很好地接受和支持 C99。这使得 C99 的一些特性成为 C11 的可选项。因为委员会认为，不应要求服务小型机市场的供应商支持其目标环境中用不到的特性。另外需要强调的是，修订标准的原因不是因为原标准不能用，而是需要跟进新的技术。例如，新标准添加了可选项支持当前使用多处理器的计算机。对于 C11 标准，我们浅尝辄止，深入分析这部分内容已超出本书讨论的范围。

#### 注意

本书使用术语 ANSI C、ISO C 或 ANSI/ISO C 讲解 C89/90 和较新标准共有的特性，用 C99 或 C11 介绍新的特性。有时也使用 C90（例如，讨论一个特性被首次加入 C 语言时）。

## 1.7 使用 C 语言的 7 个步骤

C 是编译型语言。如果之前使用过编译型语言（如，Pascal 或 FORTRAN），就会很熟悉组建 C 程序的几个基本步骤。但是，如果以前使用的是解释型语言（如，BASIC）或面向图形界面语言（如，Visual Basic），或者甚至没接触过任何编程语言，就有必要学习如何编译。别担心，这并不复杂。首先，为了让读者对编程有大概的了解，我们把编写 C 程序的过程分解成 7 个步骤（见图 1.3）。注意，这是理想状态。在实际的使用过程中，尤其是在较大型的项目中，可能要做一些重复的工作，根据下一个步骤的情况来调整或改进上一个步骤。

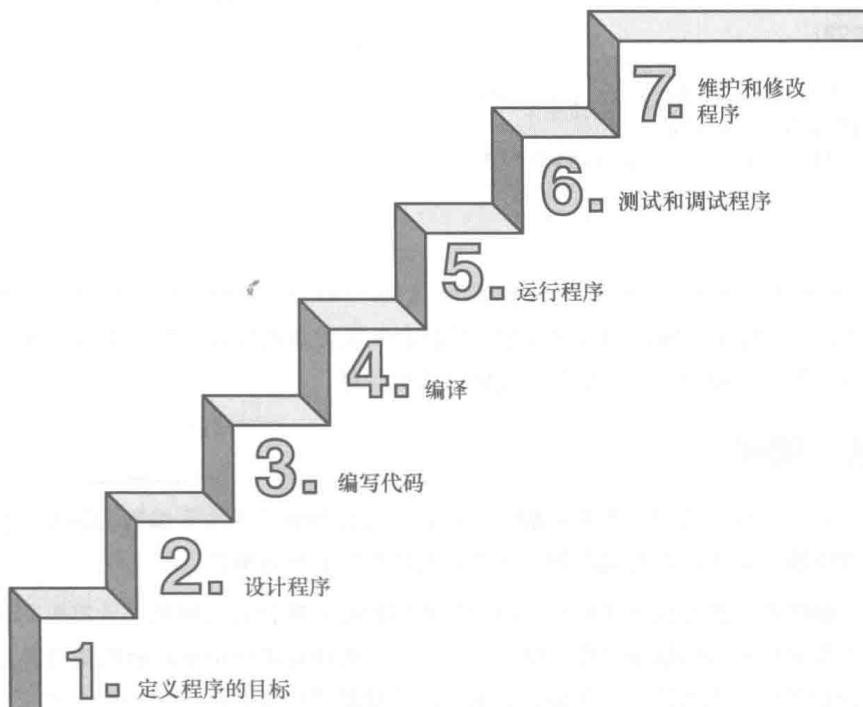


图 1.3 编程的 7 个步骤

## 1.7.1 第1步：定义程序的目标

在动手写程序之前，要在脑中有清晰的思路。想要程序去做什么首先自己要明确自己想做什么，思考你的程序需要哪些信息，要进行哪些计算和控制，以及程序应该要报告什么信息。在这一步骤中，不涉及具体的计算机语言，应该用一般术语来描述问题。

## 1.7.2 第2步：设计程序

对程序应该完成什么任务有概念性的认识后，就应该考虑如何用程序来完成它。例如，用户界面应该是怎样的？如何组织程序？目标用户是谁？准备花多长时间来完成这个程序？

除此之外，还要决定在程序（还可能是辅助文件）中如何表示数据，以及用什么方法处理数据。学习C语言之初，遇到的问题都很简单，没什么可选的。但是，随着要处理的情况越来越复杂，需要决策和考虑的方面也越来越多。通常，选择一个合适的方式表示信息可以更容易地设计程序和处理数据。

再次强调，应该用一般术语来描述问题，而不是用具体的代码。但是，你的某些决策可能取决于语言的特性。例如，在数据表示方面，C的程序员就比Pascal的程序员有更多选择。

## 1.7.3 第3步：编写代码

设计好程序后，就可以编写代码来实现它。也就是说，把你设计的程序翻译成C语言。这里是真正需要使用C语言的地方。可以把思路写在纸上，但是最终还是要把代码输入计算机。这个过程的机制取决于编程环境，我们稍后会详细介绍一些常见的环境。一般而言，使用文本编辑器创建源代码文件。该文件中内容就是你翻译的C语言代码。程序清单1.1是一个C源代码的示例。

程序清单1.1 C源代码示例

---

```
#include <stdio.h>
int main(void)
{
    int dogs;

    printf("How many dogs do you have?\n");
    scanf("%d", &dogs);
    printf("So you have %d dog(s)!\n", dogs);

    return 0;
}
```

---

在这一步骤中，应该给自己编写的程序添加文字注释。最简单的方式是使用C的注释工具在源代码中加入对代码的解释。第2章将详细介绍如何在代码中添加注释。

## 1.7.4 第4步：编译

接下来的这一步是编译源代码。再次提醒读者注意，编译的细节取决于编程的环境，我们稍后马上介绍一些常见的编程环境。现在，先从概念的角度讲解编译发生了什么事情。

前面介绍过，编译器是把源代码转换成可执行代码的程序。可执行代码是用计算机的机器语言表示的代码。这种语言由数字码表示的指令组成。如前所述，不同的计算机使用不同的机器语言方案。C编译器负责把C代码翻译成特定的机器语言。此外，C编译器还将源代码与C库（库中包含大量的标准函数供用户使用，如printf()和scanf()）的代码合并成最终的程序（更精确地说，应该是由一个被称为链接器

的程序来链接库函数，但是在大多数系统中，编译器运行链接器）。其结果是，生成一个用户可以运行的可执行文件，其中包含着计算机能理解的代码。

编译器还会检查 C 语言程序是否有效。如果 C 编译器发现错误，就不生成可执行文件并报错。理解特定编译器报告的错误或警告信息是程序员要掌握的另一项技能。

## 1.7.5 第 5 步：运行程序

传统上，可执行文件是可运行的程序。在常见环境（包括 Windows 命令提示符模式、UNIX 终端模式和 Linux 终端模式）中运行程序要输入可执行文件的文件名，而其他环境可能要运行命令（如，在 VAX 中的 VMS<sup>1</sup>）或一些其他机制。例如，在 Windows 和 Macintosh 提供的集成开发环境（IDE）中，用户可以在 IDE 中通过选择菜单中的选项或按下特殊键来编辑和执行 C 程序。最终生成的程序可通过单击或双击文件名或图标直接在操作系统中运行。

## 1.7.6 第 6 步：测试和调试程序

程序能运行是个好迹象，但有时也可能会出现运行错误。接下来，应该检查程序是否按照你所设计的思路运行。你会发现你的程序中有一些错误，计算机行话叫作 bug。查找并修复程序错误的过程叫调试。学习的过程中不可避免会犯错，学习编程也是如此。因此，当你把所学的知识应用于编程时，最好为自己会犯错做好心理准备。随着你越来越老练，你所写的程序中的错误也会越来越不易察觉。

将来犯错的机会很多。你可能会犯基本的设计错误，可能错误地实现了一个好想法，可能忽视了输入检查导致程序瘫痪，可能会把圆括号放错地方，可能误用 C 语言或打错字，等等。把你将来犯错的地方列出来，这份错误列表应该会很长。

看到这里你可能会有些绝望，但是情况没那么糟。现在的编译器会捕获许多错误，而且自己也可以找到编译器未发现的错误。在学习本书的过程中，我们会给读者提供一些调试的建议。

## 1.7.7 第 7 步：维护和修改代码

创建完程序后，你发现程序有错，或者想扩展程序的用途，这时就要修改程序。例如，用户输入以 Zz 开头的姓名时程序出现错误、你想到了一个更好的解决方案、想添加一个更好的新特性，或者要修改程序使其能在不同的计算机系统中运行，等等。如果在编写程序时清楚地做了注释并采用了合理的设计方案，这些事情都很简单。

## 1.7.8 说明

编程并非像描述那样是一个线性的过程。有时，要在不同的步骤之间往复。例如，在写代码时发现之前的设计不切实际，或者想到了一个更好的解决方案，或者等程序运行后，想改变原来的设计思路。对程序做文字注释为今后的修改提供了方便。

许多初学者经常忽略第 1 步和第 2 步（定义程序目标和设计程序），直接跳到第 3 步（编写代码）。刚开始学习时，编写的程序非常简单，完全可以在脑中构思好整个过程。即使写错了，也很容易发现。但是，随着编写的程序越来越庞大、越来越复杂，动脑不动手可不行，而且程序中隐藏的错误也越来越难找。最终，那些跳过前两个步骤的人往往浪费了更多的时间，因为他们写出的程序难看、缺乏条理、让人难以理解。要编写的程序越大越复杂，事先定义和设计程序环节的工作量就越大。

<sup>1</sup> VAX( Virtual Address eXtension )是一种可支持机器语言和虚拟地址的 32 位小型计算机。VMS( Virtual Memory System )是旧名，现在叫 OpenVMS，是一种用于服务器的操作系统，可在 VAX、Alpha 或 Itanium 处理器系列平台上运行。  
——译者注

磨刀不误砍柴工，应该养成先规划再动手编写代码的好习惯，用纸和笔记录下程序的目标和设计框架。这样在编写代码的过程中会更加得心应手、条理清晰。

## 1.8 编程机制

生成程序的具体过程因计算机环境而异。C是可移植性语言，因此可以在许多环境中使用，包括UNIX、Linux、MS-DOS（一些人仍在使用）、Windows和Macintosh OS。有些产品会随着时间的推移发生演变或被取代，本书无法涵盖所有环境。

首先，来看看许多C环境（包括上面提到的5种环境）共有的一些方面。虽然不必详细了解计算机内部如何运行C程序，但是，了解一下编程机制不仅能丰富编程相关的背景知识，还有助于理解为何要经过一些特殊的步骤才能得到C程序。

用C语言编写程序时，编写的内容被储存在文本文件中，该文件被称为源代码文件（*source code file*）。大部分C系统，包括之前提到的，都要求文件名以.c结尾（如，`wordcount.c`和`budget.c`）。在文件名中，点号(.)前面的部分称为基本名（*basename*），点号后面的部分称为扩展名（*extension*）。因此，`budget`是基本名，`c`是扩展名。基本名与扩展名的组合（`budget.c`）就是文件名。文件名应该满足特定计算机操作系统的特殊要求。例如，MS-DOS是IBM PC及其兼容机的操作系统，比较老旧，它要求基本名不能超过8个字符。因此，刚才提到的文件名`wordcount.c`就是无效的DOS文件名。有些UNIX系统限制整个文件名（包括扩展名）不超过14个字符，而有些UNIX系统则允许使用更长的文件名，最多255个字符。Linux、Windows和Macintosh OS都允许使用长文件名。

接下来，我们来看一下具体的应用，假设有一个名为`concrete.c`的源文件，其中的C源代码如程序清单1.2所示。

程序清单1.2 C程序

---

```
#include <stdio.h>
int main(void)
{
    printf("Concrete contains gravel and cement.\n");
    return 0;
}
```

---

如果看不懂程序清单1.2中的代码，不用担心，我们将在第2章学习相关知识。

### 1.8.1 目标代码文件、可执行文件和库

C编程的基本策略是，用程序把源代码文件转换为可执行文件（其中包含可直接运行的机器语言代码）。典型的C实现通过编译和链接两个步骤来完成这一过程。编译器把源代码转换成中间代码，链接器把中间代码和其他代码合并，生成可执行文件。C使用这种分而治之的方法方便对程序进行模块化，可以独立编译单独的模块，稍后再用链接器合并已编译的模块。通过这种方式，如果只更改某个模块，不必因此重新编译其他模块。另外，链接器还将你编写的程序和预编译的库代码合并。

中间文件有多种形式。我们在这里描述的是最普遍的一种形式，即把源代码转换为机器语言代码，并把结果放在目标代码文件（或简称目标文件）中（这里假设源代码只有一个文件）。虽然目标文件中包含机器语言代码，但是并不能直接运行该文件。因为目标文件中储存的是编译器翻译的源代码，这不是一个完整的程序。

目标代码文件缺失启动代码（*startup code*）。启动代码充当着程序和操作系统之间的接口。例如，可以在MS Windows或Linux系统下运行IBM PC兼容机。这两种情况所使用的硬件相同，所以目标代码相同，

但是 Windows 和 Linux 所需的启动代码不同，因为这些系统处理程序的方式不同。

目标代码还缺少库函数。几乎所有的 C 程序都要使用 C 标准库中的函数。例如，concrete.c 中就使用了 printf() 函数。目标代码文件并不包含该函数的代码，它只包含了使用 printf() 函数的指令。printf() 函数真正的代码储存在另一个被称为库的文件中。库文件中有许多函数的目标代码。

链接器的作用是，把你编写的目标代码、系统的标准启动代码和库代码这 3 部分合并成一个文件，即可执行文件。对于库代码，链接器只会把程序中要用到的库函数代码提取出来（见图 1.4）。

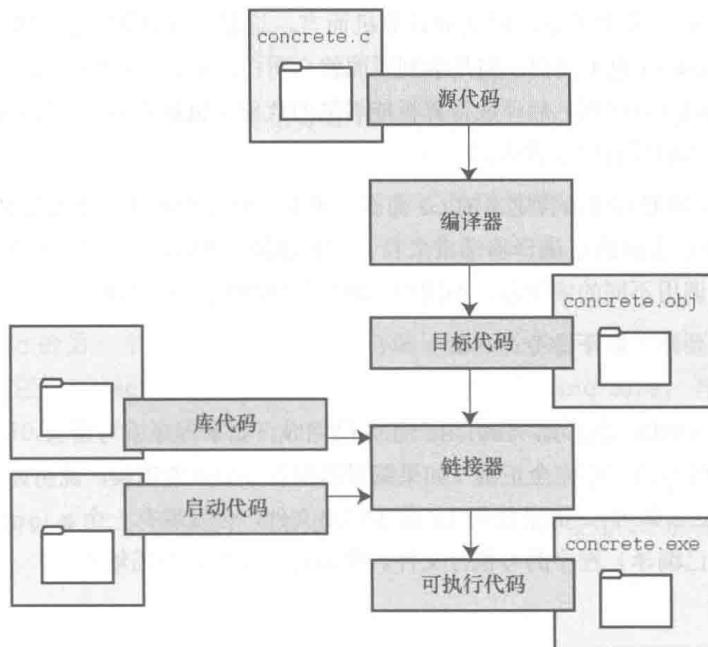


图 1.4 编译器和链接器

简而言之，目标文件和可执行文件都由机器语言指令组成的。然而，目标文件中只包含编译器为你编写的代码翻译的机器语言代码，可执行文件中还包含你编写的程序中使用的库函数和启动代码的机器代码。

在有些系统中，必须分别运行编译程序和链接程序，而在另一些系统中，编译器会自动启动链接器，用户只需给出编译命令即可。

接下来，了解一些具体的系统。

## 1.8.2 UNIX 系统

由于 C 语言因 UNIX 系统而生，也因此而流行，所以我们从 UNIX 系统开始（注意：我们提到的 UNIX 还包含其他系统，如 FreeBSD，它是 UNIX 的一个分支，但是由于法律原因不使用该名称）。

### 1. 在 UNIX 系统上编辑

UNIX C 没有自己的编辑器，但是可以使用通用的 UNIX 编辑器，如 emacs、jove、vi 或 X Window System 文本编辑器。

作为程序员，要负责输入正确的程序和为储存该程序的文件起一个合适的文件名。如前所述，文件名应该以 .c 结尾。注意，UNIX 区分大小写。因此，budget.c、BUDGET.c 和 Budget.c 是 3 个不同但都有效的 C 源文件名。但是 BUDGET.C 是无效文件名，因为该名称的扩展名使用了大写 C 而不是小写 c。

假设我们在 vi 编译器中编写了下面的程序，并将其储存在 inform.c 文件中：

```
#include <stdio.h>
int main(void)
```

```

{
    printf("A .c is used to end a C program filename.\n");

    return 0;
}

```

以上文本就是源代码，inform.c是源文件。注意，源文件是整个编译过程的开始，不是结束。

## 2. 在UNIX系统上编译

虽然在我们看来，程序完美无缺，但是对计算机而言，这是一堆乱码。计算机不明白#include和printf是什么（也许你现在也不明白，但是学到后面就会明白，而计算机却不会）。如前所述，我们需要编译器将我们编写的代码（源代码）翻译成计算机能看懂的代码（机器代码）。最后生成的可执行文件中包含计算机要完成任务所需的所有机器代码。

以前，UNIX C编译器要调用语言定义的cc命令。但是，它没有跟上标准发展的脚步，已经退出了历史舞台。但是，UNIX系统提供的C编译器通常来自一些其他源，然后以cc命令作为编译器的别名。因此，虽然在不同的系统中会调用不同的编译器，但用户仍可以继续使用相同的命令。

编译inform.c，要输入以下命令：

```
cc inform.c
```

几秒钟后，会返回UNIX的提示，告诉用户任务已完成。如果程序编写错误，你可能会看到警告或错误消息，但我们先假设编写的程序完全正确（如果编译器报告void的错误，说明你的系统未更新成ANSI C编译器，只需删除void即可）。如果使用ls命令列出文件，会发现有一个a.out文件（见图1.5）。该文件是包含已翻译（或已编译）程序的可执行文件。要运行该文件，只需输入：

```
a.out
```

输出内容如下：

```
A .c is used to end a C program filename.
```

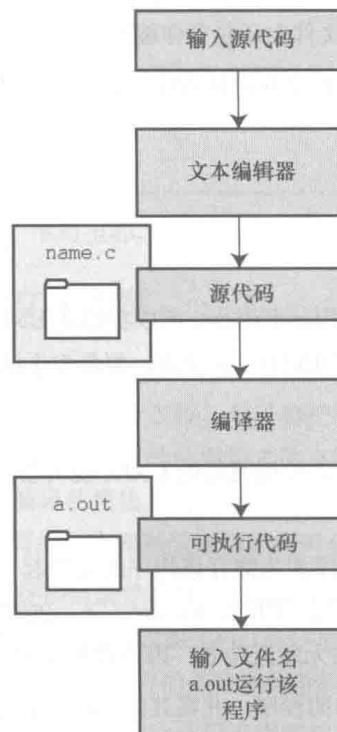


图1.5 用UNIX准备C程序

如果要储存可执行文件（`a.out`），应该把它重命名。否则，该文件会被下一次编译程序时生成的新 `a.out` 文件替换。

如何处理目标代码？C 编译器会创建一个与源代码基本名相同的目标代码文件，但是其扩展名是`.o`。在该例中，目标代码文件是 `inform.o`。然而，却找不到这个文件，因为一旦链接器生成了完整的可执行程序，就会将其删除。如果原始程序有多个源代码文件，则保留目标代码文件。学到后面多文件程序时，你会明白到这样做的好处。

### 1.8.3 GNU 编译器集合和 LLVM 项目

GNU 项目始于 1987 年，是一个开发大量免费 UNIX 软件的集合（GNU 的意思是“GNU's Not UNIX”，即 GNU 不是 UNIX）。GNU 编译器集合（也被称为 GCC，其中包含 GCC C 编译器）是该项目的产品之一。GCC 在一个指导委员会的带领下，持续不断地开发，它的 C 编译器紧跟 C 标准的改动。GCC 有各种版本以适应不同的硬件平台和操作系统，包括 UNIX、Linux 和 Windows。用 `gcc` 命令便可调用 GCC C 编译器。许多使用 `gcc` 的系统都用 `cc` 作为 `gcc` 的别名。

LLVM 项目成为 `cc` 的另一个替代品。该项目是与编译器相关的开源软件集合，始于伊利诺伊大学的 2000 份研究项目。它的 Clang 编译器处理 C 代码，可以通过 `clang` 调用。有多种版本供不同的平台使用，包括 Linux。2012 年，Clang 成为 FreeBSD 的默认 C 编译器。Clang 也对最新的 C 标准支持得很好。

GNU 和 LLVM 都可以使用`-v` 选项来显示版本信息，因此各系统都使用 `cc` 别名来代替 `gcc` 或 `clang` 命令。以下组合：

```
cc -v
```

显示你所使用的编译器及其版本。

`gcc` 和 `clang` 命令都可以根据不同的版本选择运行时选项来调用不同 C 标准。

```
gcc -std=c99 inform.c1
gcc -std=clx inform.c
gcc -std=c11 inform.c
```

第 1 行调用 C99 标准，第 2 行调用 GCC 接受 C11 之前的草案标准，第 3 行调用 GCC 接受的 C11 标准版本。Clang 编译器在这一点上用法与 GCC 相同。

### 1.8.4 Linux 系统

Linux 是一个开源、流行、类似于 UNIX 的操作系统，可在不同平台（包括 PC 和 Mac）上运行。在 Linux 中准备 C 程序与在 UNIX 系统中几乎一样，不同的是要使用 GNU 提供的 GCC 公共域 C 编译器。编译命令类似于：

```
gcc inform.c
```

注意，在安装 Linux 时，可选择是否安装 GCC。如果之前没有安装 GCC，则必须安装。通常，安装过程会将 `cc` 作为 `gcc` 的别名，因此可以在命令行中使用 `cc` 来代替 `gcc`。

欲详细了解 GCC 和最新发布的版本，请访问 <http://www.gnu.org/software/gcc/index.html>。

---

<sup>1</sup> GCC 最基本的用法是：`gcc [options] [filenames]`，其中 `options` 是所需的参数，`filenames` 是文件名。——译者注

## 1.8.5 PC的命令行编译器

C编译器不是标准Windows软件包的一部分，因此需要从别处获取并安装C编译器。可以从互联网免费下载Cygwin和MinGW，这样便可在PC上通过命令行使用GCC编译器。Cygwin在自己的视窗运行，模仿Linux命令行环境，有一行命令提示。MinGW在Windows的命令提示模式中运行。这和GCC的最新版本一样，支持C99和C11最新的一些功能。Borland的C++编译器5.5也可以免费下载，支持C90。

源代码文件应该是文本文件，不是字处理器文件（字处理器文件包含许多额外的信息，如字体和格式等）。因此，要使用文本编辑器（如，Windows Notepad）来编辑源代码。如果使用字处理器，要以文本模式另存文件。源代码文件的扩展名应该是.c。一些字处理器会为文本文件自动添加.txt扩展名。如果出现这种情况，要更改文件名，把txt替换成c。

通常，C编译器生成的中间目标代码文件的扩展名是.obj（也可能是其他扩展名）。与UNIX编译器不同，这些编译器在完成编译后通常不会删除这些中间文件。有些编译器生成带.asm扩展名的汇编语言文件，而有些编译器则使用自己特有的格式。

一些编译器在编译后会自动运行链接器，另一些要求用户手动运行链接器。在可执行文件中链接的结果是，在原始的源代码基本名后面加上.exe扩展名。例如，编译和链接concrete.c源代码文件，生成的是concrete.exe文件。可以在命令行输入基本名来运行该程序：

```
C>concrete
```

## 1.8.6 集成开发环境(Windows)

许多供应商（包括微软、Embarcadero、Digital Mars）都提供Windows下的集成开发环境，或称为IDE（目前，大多数IDE都是C和C++结合的编译器）。可以免费下载的IDE有Microsoft Visual Studio Express和Pelles C。利用集成开发环境可以快速开发C程序。关键是，这些IDE都内置了用于编写C程序的编辑器。这类集成开发环境都提供了各种菜单（如，命名、保存源代码文件、编译程序、运行程序等），用户不用离开IDE就能顺利编写、编译和运行程序。如果编译器发现错误，会返回编辑器中，标出有错误的行号，并简单描述情况。

初次接触Windows IDE可能会望而生畏，因为它提供了多种目标(target)，即运行程序的多种环境。例如，IDE提供了32位Windows程序、64位Windows程序、动态链接库文件(DLL)等。许多目标都涉及Windows图形界面。要管理这些（及其他）选择，通常要先创建一个项目(project)，以便稍后在其中添加待使用的源代码文件名。不同的产品具体步骤不同。一般而言，首先使用【文件】菜单或【项目】菜单创建一个项目。选择正确的项目形式非常重要。本书中的例子都是一般示例，针对在简单的命令行环境中运行而设计。Windows IDE提供多种选择以满足用户的不同需求。例如，Microsoft Visual Studio提供【Win32控制台应用程序】选项。对于其他系统，查找一个诸如【DOS EXE】、【Console】或【Character Mode】的可执行选项。选择这些模式后，将在一个类控制台窗口中运行可执行程序。选择好正确的项目类型后，使用IDE的菜单打开一个新的源代码文件。对于大多数产品而言，使用【文件】菜单就能完成。你可能需要其他步骤将源文件添加到项目中。

通常，Windows IDE既可处理C也可处理C++，因此要指定待处理的程序是C还是C++。有些产品用项目类型来区分两者，有些产品（如，Microsoft Visual C++）用.c文件扩展名来指明使用C而不是C++。当然，大多数C程序也可以作为C++程序运行。欲了解C和C++的区别，请参阅参考资料IX。

你可能会遇到一个问题：在程序执行完毕后，执行程序的窗口立即消失。如果不希望出现这种情况，可以让程序暂停，直到按下Enter键，窗口才消失。要实现这种效果，可以在程序的最后(return这行代码之前)添加下面一行代码：

```
getchar();
```

该行读取一次键的按下，所以程序在用户按下 **Enter** 键之前会暂停。有时根据程序的需要，可能还需要一个击键等待。这种情况下，必须用两次 `getchar()`：

```
getchar();
getchar();
```

例如，程序在最后提示用户输入体重。用户键入体重后，按下 **Enter** 键以输入数据。程序将读取体重，第 1 个 `getchar()` 读取 **Enter** 键，第 2 个 `getchar()` 会导致程序暂停，直至用户再次按下 **Enter** 键。如果你现在不知所云，没关系，在学完 C 输出后就会明白。到时，我们会提醒读者使用这种方法。

虽然许多 IDE 在使用上大体一致，但是细节上有所不同。就一个产品的系列而言，不同版本也是如此。要经过一段时间的实践，才会熟悉编译器的工作方式。必要时，还需阅读使用手册或网上教程。

### Microsoft Visual Studio 和 C 标准

在 Windows 软件开发中，Microsoft Visual Studio 及其免费版本 Microsoft Visual Studio Express 都久负盛名，它们与 C 标准的关系也很重要。然而，微软鼓励程序员从 C 转向 C++ 和 C#。虽然 Visual Studio 支持 C89/90，但是到目前为止，它只选择性地支持那些在 C++ 新特性中能找到的 C 标准（如，`long long` 类型）。而且，自 2012 版本起，Visual Studio 不再把 C 作为项目类型的选项。尽管如此，本书中的绝大多数程序仍可用 Visual Studio 来编译。在新建项目时，选择 C++ 选项，然后选择【Win32 控制台应用程序】，在应用设置中选择【空项目】。几乎所有的 C 程序都能与 C++ 程序兼容。所以，本书中的绝大多数 C 程序都可作为 C++ 程序运行。或者，在选择 C++ 选项后，将默认的源文件扩展名 `.cpp` 替换成 `.c`，编译器便会使用 C 语言的规则代替 C++。

### 1.8.7 Windows/Linux

许多 Linux 发行版都可以安装在 Windows 系统中，以创建双系统。一些存储器会为 Linux 系统预留空间，以便可以启动 Windows 或 Linux。可以在 Windows 系统中运行 Linux 程序，或在 Linux 系统中运行 Windows 程序。不能通过 Windows 系统访问 Linux 文件，但是可以通过 Linux 系统访问 Windows 文档。

### 1.8.8 Macintosh 中的 C

目前，苹果免费提供 Xcode 开发系统下载（过去，它有时免费，有时付费）。它允许用户选择不同的编程语言，包括 C 语言。

Xcode 凭借可处理多种编程语言的能力，可用于多平台，开发超大型的项目。但是，首先要学会如何编写简单的 C 程序。在 Xcode 4.6 中，通过【File】菜单选择【New Project】，然后选择【OS X Application Command Line Tool】，接着输入产品名并选择 C 类型。Xcode 使用 Clang 或 GCC C 编译器来编译 C 代码，它以前默认使用 GCC，但是现在默认使用 Clang。可以设置选择使用哪一个编译器和哪一套 C 标准（因为许可方面的事宜，Xcode 中 Clang 的版本比 GCC 的版本要新）。

UNIX 系统内置 Mac OS X，终端工具打开的窗口是让用户在 UNIX 命令行环境中运行程序。苹果在标准软件包中不提供命令行编译器，但是，如果下载了 Xcode，还可以下载可选的命令行工具，这样就可以使用 `clang` 和 `gcc` 命令在命令行模式中编译。

## 1.9 本书的组织结构

本书采用多种方式编排内容，其中最直接的方法是介绍 A 主题的所有内容、介绍 B 主题的所有内容，

等等。这对参考类书籍来说尤为重要，读者可以在同一处找到与主题相关的所有内容。但是，这通常不是学习的最佳顺序。例如，如果在开始学习英语时，先学完所有的名词，那你的表达能力一定很有限。虽然可以指着物品说出名称，但是，如果稍微学习一些名词、动词、形容词等，再学习一些造句规则，那么你的表达能力一定会大幅提高。

为了让读者更好地吸收知识，本书采用螺旋式方法，先在前几个章节中介绍一些主题，在后面章节再详细讨论相关内容。例如，对学习C语言而言，理解函数至关重要。因此，我们在前几个章节中安排一些与函数相关的内容，等读者学到第9章时，已对函数有所了解，学习使用函数会更加容易。与此类似，前几章还概述了一些字符串和循环的内容。这样，读者在完全弄懂这些内容之前，就可以在自己的程序中使用这些有用的工具。

## 1.10 本书的约定

在学习C语言之前，先介绍一下本书的格式。

### 1.10.1 字体

本书用类似在屏幕上或打印输出时的字体（一种等宽字体），表示文本程序和计算机输入、输出。前面已经出现了多次，如果读者没有注意到，字体如下所示：

```
#include <stdio.h>
int main(void)
{
    printf("Concrete contains gravel and cement.\n");

    return 0;
}
```

在涉及与代码相关的术语时，也使用相同的等宽字体，如 `stdio.h`。本书用等宽斜体表示占位符，可以用具体的项替换这些占位符。例如，下面是一个声明的模型：

`type_name variable_name;`

这里，可用 `int` 替换 `type_name`，用 `zebra_count` 替换 `variable_name`。

### 1.10.2 程序输出

本书用相同的字体表示计算机的输出，粗体表示用户输入。例如，下面是第14章中一个程序的输出：

```
Please enter the book title.
Press [enter] at the start of a line to stop.
```

```
My Life as a Budgie
Now enter the author.
```

```
Mack Zackles
```

如上所示，以标准计算机字体显示的行表示程序的输出，粗体行表示用户的输入。

可以通过多种方式与计算机交互。在这里，我们假设读者使用键盘键入内容，在屏幕上阅读计算机的响应。

#### 1. 特殊的击键

通常，通过按下标有 `Enter`、`c/r`、`Return` 或一些其他文字的键来发送指令。本书将这些按键统一称为 `Enter` 键。一般情况下，我们默认你在每行输入的末尾都会按下 `Enter` 键。尽管如此，为了标示一些特

定的位置，本书使用 [enter] 显式标出 Enter 键。方括号表示按下次 Enter 键，而不是输入 enter。

除此之外，书中还会提到控制字符(如, Ctrl+D)。这种写法的意思是，在按下 Ctrl 键(也可能是 Control 键)的同时按下 D 键。

## 2. 本书使用的系统

C 语言的某些方面（如，储存数字的空间大小）因系统而异。本书在示例中提到“我们的系统”时，通常是指在 iMac 上运行 OS X 10.8.4，使用 Xcode 4.6.2 开发系统的 Clang 3.2 编译器。本书的大部分程序都能使用 Windows7 系统的 Microsoft Visual Studio Express 2012 和 Pelles C 7.0，以及 Ubuntu13.04 Linux 系统的 GCC 4.7.3 进行编译。

## 3. 读者的系统

你需要一个 C 编译器或访问一个 C 编译器。C 程序可以在多种计算机系统中运行，因此你的选择面很广。确保你使用的 C 编译器与当前使用的计算机系统匹配。本书中，除了某些示例要求编译器支持 C99 或 C11 标准，其余大部分示例都可在 C90 编译器中运行。如果你使用的编译器是早于 ANSI/ISO 的老式编译器，在编译时肯定要经常调整，很不方便。与其如此，不如换个新的编译器。

大部分编译器供应商都为学生和教学人员提供特惠版本，详情请查看供应商的网站。

### 1.10.3 特殊元素

本书包含一些强调特定知识点的特殊元素，提示、注意、警告，将以如下形式出现在本书中：

#### 边栏

边栏提供更深入的讨论或额外的背景，有助于解释当前的主题。

#### 提示

提示一般都短小精悍，帮助读者理解一些特殊的编程情况。

#### 警告

用于警告读者注意一些潜在的陷阱。

#### 注意

提供一些评论，提醒读者不要误入歧途。

## 1.11 本章小结

C 是强大而简洁的编程语言。它之所以流行，在于自身提供大量的实用编程工具，能很好地控制硬件。而且，与大多数其他程序相比，C 程序更容易从一个系统移植到另一个系统。

C 是编译型语言。C 编译器和链接器是把 C 语言源代码转换成可执行代码的程序。

用 C 语言编程可能费力、困难，让你感到沮丧，但是它也可以激发你的兴趣，让你兴奋、满意。我们希望你在愉快的学习过程中爱上 C。

## 1.12 复习题

复习题的参考答案在附录 A 中。

1. 对编程而言，可移植性意味着什么？
2. 解释源代码文件、目标代码文件和可执行文件有什么区别？
3. 编程的 7 个主要步骤是什么？
4. 编译器的任务是什么？
5. 链接器的任务是什么？

## 1.13 编程练习

我们尚未要求你编写 C 代码，该练习侧重于编程过程的早期步骤。

1. 你刚被 MacroMuscle 有限公司聘用。该公司准备进入欧洲市场，需要一个把英寸单位转换为厘米单位（1 英寸=2.54 厘米）的程序。该程序要提示用户输入英寸值。你的任务是定义程序目标和设计程序（编程过程的第一步和第二步）。

# C 语言概述

本章介绍以下内容：

- 运算符：=
- 函数：main()、printf()
- 编写一个简单的 C 程序
- 创建整型变量，为其赋值并在屏幕上显示其值
- 换行字符
- 如何在程序中写注释，创建包含多个函数的程序，发现程序的错误
- 什么是关键字

C 程序是什么样子的？浏览本书，能看到许多示例。初见 C 程序会觉得有些古怪，程序中有许多 {、}、cp->tort 和 \*ptr++ 这样的符号。然而，在学习 C 的过程中，对这些符号和 C 语言特有的其他符号会越来越熟悉，甚至会喜欢上它们。如果熟悉与 C 相关的其他语言，会对 C 语言有似曾相识的感觉。本章，我们从演示一个简单的程序示例开始，解释该程序的功能。同时，强调一些 C 语言的基本特性。

## 2.1 简单的 C 程序示例

我们来看一个简单的 C 程序，如程序清单 2.1 所示。该程序演示了用 C 语言编程的一些基本特性。请先通读程序清单 2.1，看看自己是否能明白该程序的用途，再认真阅读后面的解释。

程序清单 2.1 first.c 程序

```
#include <stdio.h>
int main(void)          /* 一个简单的 C 程序 */
{
    int num;           /* 定义一个名为 num 的变量 */
    num = 1;            /* 为 num 赋一个值 */

    printf("I am a simple "); /* 使用 printf() 函数 */
    printf("computer.\n");
    printf("My favorite number is %d because it is first.\n", num);

    return 0;
}
```

如果你认为该程序会在屏幕上打印一些内容，那就对了！光看程序也许并不知道打印的具体内容，所以，运行该程序，并查看结果。首先，用你熟悉的编辑器（或者编译器提供的编辑器）创建一个包含程序清单 2.1 中所有内容的文件。给该文件命名，并以 .c 作为扩展名，以满足当前系统对文件名的要求。例如，可以使用 first.c。现在，编译并运行该程序（查看第 1 章，复习该步骤的具体内容）。如果一切运行正常，该程序的输出应该是：

```
I am a simple computer.  
My favorite number is 1 because it is first.
```

总而言之，结果在意料之中，但是程序中的\n 和%d 是什么？程序中有几行代码看起来有点奇怪。接下来，我们逐行解释这个程序。

### 程序调整

程序的输出是否在屏幕上一闪而过？某些窗口环境会在单独的窗口运行程序，然后在程序运行结束后自动关闭窗口。如果遇到这种情况，可以在程序中添加额外的代码，让窗口等待用户按下一个键后才关闭。一种方法是，在程序的 return 语句前添加一行代码：

```
getchar();
```

这行代码会让程序等待击键，窗口会在用户按下一个键后才关闭。在第 8 章中会详细介绍 getchar() 的内容。

## 2.2 示例解释

我们会把程序清单 2.1 的程序分析两遍。第 1 遍（快速概要）概述程序中每行代码的作用，帮助读者初步了解程序。第 2 遍（程序细节）详细分析代码的具体含义，帮助读者深入理解程序。

图 2.1 总结了组成 C 程序的几个部分<sup>1</sup>，图中包含的元素比第 1 个程序多。

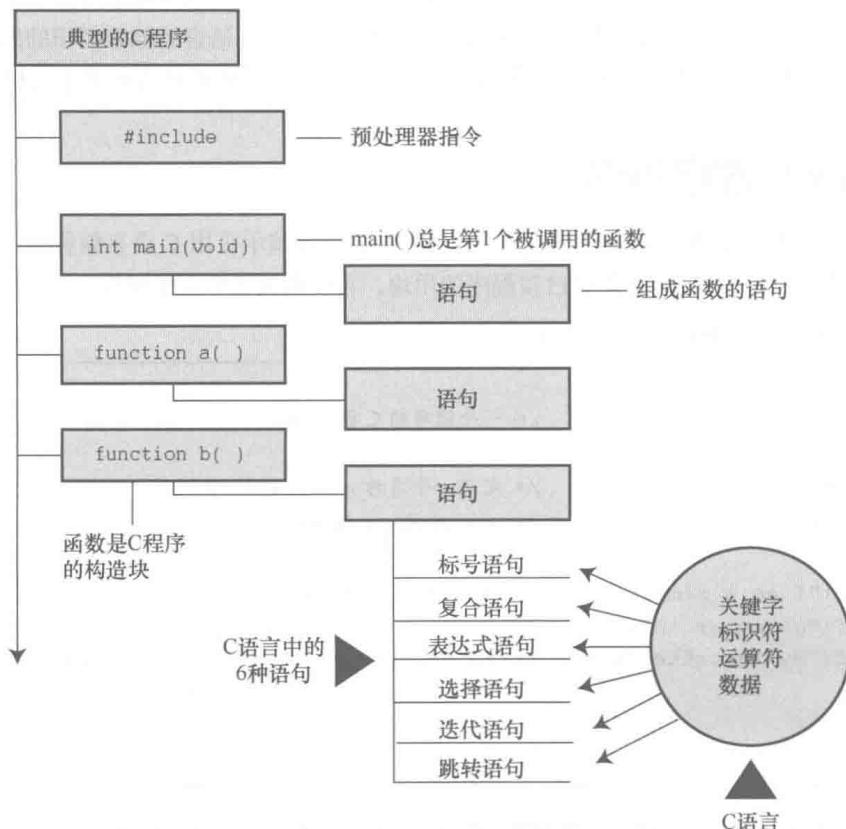


图 2.1 C 程序解剖

<sup>1</sup> 原书图中叙述有误。根据 C11 标准，C 语言有 6 种语句，已在图中更正。——译者注

## 2.2.1 第1遍：快速概要

本节简述程序中的每行代码的作用。下一节详细讨论代码的含义。

```
#include<stdio.h>      ←包含另一个文件
```

该行告诉编译器把 stdio.h 中的内容包含在当前程序中。stdio.h 是 C 编译器软件包的标准部分，它提供键盘输入和屏幕输出的支持。

```
int main(void)          ←函数名
```

C 程序包含一个或多个函数，它们是 C 程序的基本模块。程序清单 2.1 的程序中有一个名为 main() 的函数。圆括号表明 main() 是一个函数名。int 表明 main() 函数返回一个整数，void 表明 main() 不带任何参数。这些内容我们稍后详述。现在，只需记住 int 和 void 是标准 ANSI C 定义 main() 的一部分（如果使用 ANSI C 之前的编译器，请省略 void；考虑到兼容的问题，请尽量使用较新的 C 编译器）。

```
/* 一个简单的 C 程序 */      ←注释
```

注释在/\*和\*/两个符号之间，这些注释能提高程序的可读性。注意，注释只是为了帮助读者理解程序，编译器会忽略它们。

```
{      ←函数体开始
```

左花括号表示函数定义开始，右花括号（）表示函数定义结束。

```
int num;      ←声明
```

该声明表明，将使用一个名为 num 的变量，而且 num 是 int（整数）类型。

```
num = 1;      ←赋值表达式语句
```

语句 num = 1；把值 1 赋给名为 num 的变量。

```
printf("I am a simple ");  ←调用一个函数
```

该语句使用 printf() 函数，在屏幕上显示 I am a simple，光标停在同一行。printf() 是标准的 C 库函数。在程序中使用函数叫作调用函数。

```
printf("computer.\n");    ←调用另一个函数
```

接下来调用的这个 printf() 函数在上条语句打印出来的内容后面加上“computer”。代码\n 告诉计算机另起一行，即把光标移至下一行。

```
printf("My favorite number is %d because it is first.\n", num);
```

最后调用的 printf() 把 num 的值（1）内嵌在用双引号括起来的内容中一并打印。%d 告诉计算机以何种形式输出 num 的值，打印在何处。

```
return 0;      ←return 语句
```

C 函数可以给调用方提供（或返回）一个数。目前，可暂时把该行看作是结束 main() 函数的要求。

```
}      ←结束
```

必须以右花括号表示程序结束。

## 2.2.2 第2遍：程序细节

浏览完程序清单 2.1 后，我们来仔细分析这个程序。再次强调，本节将逐行分析程序中的代码，以每行代码为出发点，深入分析代码背后的细节，为更全面地学习 C 语言编程的特性夯实基础。

### 1. #include 指令和头文件

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

这是程序的第1行。`#include <stdio.h>`的作用相当于把 `stdio.h` 文件中的所有内容都输入该行所在的位置。实际上，这是一种“拷贝-粘贴”的操作。`include` 文件提供了一种方便的途径共享许多程序共有的信息。

`#include` 这行代码是一条 C 预处理器指令 (*preprocessor directive*)。通常，C 编译器在编译前会对源代码做一些准备工作，即预处理 (*preprocessing*)。

所有的 C 编译器软件包都提供 `stdio.h` 文件。该文件中包含了供编译器使用的输入和输出函数（如，`printf()`）信息。该文件名的含义是标准输入/输出头文件。通常，在 C 程序顶部的信息集合被称为头文件 (*header*)。

在大多数情况下，头文件包含了编译器创建最终可执行程序要用到的信息。例如，头文件中可以定义一些常量，或者指明函数名以及如何使用它们。但是，函数的实际代码在一个预编译代码的库文件中。简而言之，头文件帮助编译器把你的程序正确地组合在一起。

ANSI/ISO C 规定了 C 编译器必须提供哪些头文件。有些程序要包含 `stdio.h`，而有些不用。特定 C 实现的文档中应该包含对 C 库函数的说明。这些说明确定了使用哪些函数需要包含哪些头文件。例如，要使用 `printf()` 函数，必须包含 `stdio.h` 头文件。省略必要的头文件可能不会影响某一特定程序，但是最好不要这样做。本书每次用到库函数，都会用`#include` 指令包含 ANSI/ISO 标准指定的头文件。

## 注意 为何不内置输入和输出

读者一定很好奇，为何不把输入和输出这些基本功能内置在语言中。原因之一是，并非所有的程序都会用到 I/O（输入/输出）包。轻装上阵表现了 C 语言的哲学。正是这种经济使用资源的原则，使得 C 语言成为流行的嵌入式编程语言（例如，编写控制汽车自动燃油系统或蓝光播放机芯片的代码）。`#include` 中的`#` 符号表明，C 预处理器在编译器接手之前处理这条指令。本书后面章节中会介绍更多预处理器指令的示例，第 16 章将更详细地讨论相关内容。

## 2. main() 函数

```
int main(void);
```

程序清单 2.1 中的第 2 行表明该函数名为 `main`。的确，`main` 是一个极其普通的名称，但是这是唯一的选择。C 程序一定从 `main()` 函数开始执行（目前不必考虑例外的情况）。除了 `main()` 函数，你可以任意命名其他函数，而且 `main()` 函数必须是开始的函数。圆括号有什么功能？用于识别 `main()` 是一个函数。很快你将学到更多的函数。就目前而言，只需记住函数是 C 程序的基本模块。

`int` 是 `main()` 函数的返回类型。这表明 `main()` 函数返回的值是整数。返回到哪里？返回给操作系统。我们将在第 6 章中再来探讨这个问题。

通常，函数名后面的圆括号中包含一些传入函数的信息。该例中没有传递任何信息。因此，圆括号内是单词 `void`（第 11 章将介绍把信息从 `main()` 函数传回操作系统的另一种形式）。

如果浏览旧式的 C 代码，会发现程序以如下形式开始：

```
main()
```

C90 标准勉强接受这种形式，但是 C99 和 C11 标准不允许这样写。因此，即使你使用的编译器允许，也不要这样写。

你还会看到下面这种形式：

```
void main()
```

一些编译器允许这样写，但是所有的标准都未认可这种写法。因此，编译器不必接受这种形式，而且

许多编译器都不能这样写。需要强调的是，只要坚持使用标准形式，把程序从一个编译器移至另一个编译器时就不会出什么问题。

### 3. 注释

```
/*一个简单的程序*/
```

在程序中，被`/* */`两个符号括起来的部分是程序的注释。写注释能让他人（包括自己）更容易明白你所写的程序。C 语言注释的好处之一是，可将注释放在任意的地方，甚至是与要解释的内容在同一行。较长的注释可单独放一行或多行。在`/*`和`*/`之间的内容都会被编译器忽略。下面列出了一些有效和无效的注释形式：

```
/* 这是一条 C 注释。 */
```

```
/* 这也是一条注释，
```

```
    被分成两行。 */
```

```
/*
```

```
    也可以这样写注释。
```

```
*/
```

```
/* 这条注释无效，因为缺少了结束标记。
```

C99 新增了另一种风格的注释，普遍用于 C++ 和 Java。这种新风格使用`//`符号创建注释，仅限于单行。

```
// 这种注释只能写成一行。
```

```
int rigue; // 这种注释也可置于此。
```

因为一行末尾就标志着注释的结束，所以这种风格的注释只需在注释开始处标明`//`符号即可。

这种新形式的注释是为了解决旧形式注释存在的潜在问题。假设有下面的代码：

```
/*
```

```
    希望能运行。
```

```
*/
```

```
x = 100;
```

```
y = 200;
```

```
/* 其他内容已省略。 */
```

接下来，假设你决定删除第 4 行，但不小心删掉了第 3 行`(*)`。代码如下所示：

```
/*
```

```
    希望能运行。
```

```
y = 200;
```

```
/*其他内容已省略。 */
```

现在，编译器把第 1 行的`/*`和第 4 行的`*/`配对，导致 4 行代码全都成了注释（包括应作为代码的那一行）。而`//`形式的注释只对单行有效，不会导致这种“消失代码”的问题。

一些编译器可能不支持这一特性。还有一些编译器需要更改设置，才能支持 C99 或 C11 的特性。

考虑到只用一种注释风格过于死板乏味，本书在示例中采用两种风格的注释。

### 4. 花括号、函数体和块

```
{  
    ...  
}
```

程序清单 2.1 中，花括号把`main()`函数括起来。一般而言，所有的 C 函数都使用花括号标记函数体的开始和结束。这是规定，不能省略。只有花括号`({})`能起这种作用，圆括号`(( ))`和方括号`[[[]]]`都不行。

花括号还可用于把函数中的多条语句合并为一个单元或块。如果读者熟悉 Pascal、ADA、Modula-2 或者 Algol，就会明白花括号在 C 语言中的作用类似于这些语言中的 begin 和 end。

## 5. 声明

```
int num;
```

程序清单 2.1 中，这行代码叫作声明（*declaration*）。声明是 C 语言最重要的特性之一。在该例中，声明完成了两件事。其一，在函数中有一个名为 num 的变量（*variable*）。其二，int 表明 num 是一个整数（即，没有小数点或小数部分的数）。int 是一种数据类型。编译器使用这些信息为 num 变量在内存中分配存储空间。分号在 C 语言中是大部分语句和声明的一部分，不像在 Pascal 中只是语句间的分隔符。

int 是 C 语言的一个关键字（*keyword*），表示一种基本的 C 语言数据类型。关键字是语言定义的单词，不能做其他用途。例如，不能用 int 作为函数名和变量名。但是，这些关键字在该语言以外不起作用，所以把一只猫或一个可爱的小孩叫 int 是可以的（尽管某些地方的当地习俗或法律可能不允许）。

示例中的 num 是一个标识符（*identifier*），也就一个变量、函数或其他实体的名称。因此，声明把特定标识符与计算机内存中的特定位置联系起来，同时也确定了储存在某位置的信息类型或数据类型。

在 C 语言中，所有变量都必须先声明才能使用。这意味着必须列出程序中用到的所有变量名及其类型。

以前的 C 语言，还要求把变量声明在块的顶部，其他语句不能在任何声明的前面。也就是说，main() 函数体如下所示：

```
int main() //旧规则
{
    int doors;
    int dogs;
    doors = 5;
    dogs = 3;
    // 其他语句
}
```

C99 和 C11 遵循 C++ 的惯例，可以把声明放在块中的任何位置。尽管如此，首次使用变量之前一定要先声明它。因此，如果编译器支持这一新特性，可以这样编写上面的代码：

```
int main()          // 目前的 C 规则
{
    // 一些语句
    int doors;
    doors = 5; // 第 1 次使用 doors
    // 其他语句
    int dogs;
    dogs = 3; // 第 1 次使用 dogs
    // 其他语句
}
```

为了与旧系统更好地兼容，本书沿用最初的规则（即，把变量声明都写在块的顶部）。

现在，读者可能有 3 个问题：什么是数据类型？如何命名？为何要声明变量？请往下看。

### 数据类型

C 语言可以处理多种类型的数据，如整数、字符和浮点数。把变量声明为整型或字符类型，计算机才能正确地储存、读取和解释数据。下一章将详细介绍 C 语言中的各种数据类型。

## 命名

给变量命名时要使用有意义的变量名或标识符（如，程序中需要一个变量数羊，该变量名应该是 sheep\_count 而不是 x3）。如果变量名无法清楚地表达自身的用途，可在注释中进一步说明。这是一种良好的编程习惯和编程技巧。

C99 和 C11 允许使用更长的标识符名，但是编译器只识别前 63 个字符。对于外部标识符（参阅第 12 章），只允许使用 31 个字符。（以前 C90 只允许 6 个字符，这是一个很大的进步。旧式编译器通常最多只允许使用 8 个字符。）实际上，你可以使用更长的字符，但是编译器会忽略超出的字符。也就是说，如果有两个标识符名都有 63 个字符，只有一个字符不同，那么编译器会识别这是两个不同的名称。如果两个标识符都是 64 个字符，只有最后一个字符不同，那么编译器可能将其视为同一个名称，也可能不会。标准并未定义在这种情况下会发生什么。

可以用小写字母、大写字母、数字和下划线（\_）来命名。而且，名称的第一个字符必须是字符或下划线，不能是数字。表 2.1 给出了一些示例。

表 2.1 有效和无效的名称

有效的名称	无效的名称
wiggles	\$Z]**
cat2	2cat
Hot_Tub	Hot-Tub
taxRate	tax rate
_kcab	don't

操作系统和 C 库经常使用以一个或两个下划线字符开始的标识符（如，\_kcab），因此最好避免在自己的程序中使用这种名称。标准标签都以一个或两个下划线字符开始，如库标识符。这样的标识符都是保留的。这意味着，虽然使用它们没有语法错误，但是会导致名称冲突。

C 语言的名称区分大小写，即把一个字母的大写和小写视为两个不同的字符。因此，stars 和 Stars、STARS 都不同。

为了让 C 语言更加国际化，C99 和 C11 根据通用字符名（即 UCN）机制添加了扩展字符集。其中包含了除英文字母以外的部分字符。欲了解详细内容，请参阅附录 B 的“参考资料 VII：扩展字符支持”。

## 声明变量的 4 个理由

一些更老的语言（如，FORTRAN 和 BASIC 的最初形式）都允许直接使用变量，不必先声明。为何 C 语言不采用这种简单易行的方法？原因如下。

- 把所有的变量放在一处，方便读者查找和理解程序的用途。如果变量名都是有意义的（如，taxtate 而不是 r），这样做效果很好。如果变量名无法表述清楚，在注释中解释变量的含义。这种方法让程序的可读性更高。
- 声明变量会促使你在编写程序之前做一些计划。程序在开始时要获得哪些信息？希望程序如何输出？表示数据最好的方式是什么？
- 声明变量有助于发现隐藏在程序中的小错误，如变量名拼写错误。例如，假设在某些不需要声明就可以直接使用变量的语言中，编写如下语句：

```
RADIUS1 = 20.4;
```

在后面的程序中，误写成：

```
CIRCUM = 6.28 * RADIUS1;
```

你不小心把数字 1 打成小写字母 l。这些语言会创建一个新的变量 RADIUS1，并使用该变量中的值（也许是 0，也许是垃圾值），导致赋给 CIRCUM 的值是错误值。你可能要花很久时间才能查出原因。这样的错误在 C 语言中不会发生（除非你很不明智地声明了两个极其相似的变量），因为编译器在发现未声明的 RADIUS1 时会报错。

- 如果事先未声明变量，C 程序将无法通过编译。如果前几个理由还不足以说服你，这个理由总可以让你认真考虑一下了。

如果要声明变量，应该声明在何处？前面提到过，C99 之前的标准要求把声明都置于块的顶部，这样规定的好处是：把声明放在一起更容易理解程序的用途。C99 允许在需要时才声明变量，这样做的好处是：在给变量赋值之前声明变量，就不会忘记给变量赋值。但是实际上，许多编译器都还不支持 C99。

## 6. 赋值

```
num = 1;
```

程序清单中的这行代码是赋值表达式语句<sup>1</sup>。赋值是 C 语言的基本操作之一。该行代码的意思是“把值 1 赋给变量 num”。在执行 int num; 声明时，编译器在计算机内存中为变量 num 预留了空间，然后在执行这行赋值表达式语句时，把值储存在之前预留的位置。可以给 num 赋不同的值，这就是 num 之所以被称为变量（variable）的原因。注意，该赋值表达式语句从右侧把值赋到左侧。另外，该语句以分号结尾，如图 2.2 所示。



图 2.2 赋值是 C 语言中的基本操作之一

## 7. printf() 函数

```
printf("I am a simple ");
printf("computer.\n");
printf("My favorite number is %d because it is first.\n", num);
```

这 3 行都使用了 C 语言的一个标准函数：printf()。圆括号表明 printf 是一个函数名。圆括号中的内容是从 main() 函数传递给 printf() 函数的信息。例如，上面的第 1 行把 I am a simple 传递给 printf() 函数。该信息被称为参数，或者更确切地说，是函数的实际参数（actual argument），如图 2.3 所示。（在 C 语言中，实际参数（简称实参）是传递给函数的特定值，形式参数（简称形参）是函数中用于储存值的变量。第 5 章中将详述相关内容。）printf() 函数用参数来做什么？该函数会查看双引号中的内容，并将其打印在屏幕上。

<sup>1</sup> C 语言是通过赋值运算符而不是赋值语句完成赋值操作。根据 C 标准，C 语言并没有所谓的“赋值语句”，本书及一些其他书籍中提到的“赋值语句”实际上是表达式语句（C 语言的 6 种基本语句之一）。本书把“赋值语句”均译为“赋值表达式语句”，以提醒初学者注意。——译者注

```
printf("That's mere contrariness");
```



实际参数

图 2.3 带实参的 printf() 函数

第 1 行 `printf()` 演示了在 C 语言中如何调用函数。只需输入函数名，把所需的参数填入圆括号即可。当程序运行到这一行时，控制权被转给已命名的函数（该例中是 `printf()`）。函数执行结束后，控制权被返回至主调函数（*calling function*），该例中是 `main()`。

第 2 行 `printf()` 函数的双引号中的 `\n` 字符并未输出。这是为什么？`\n` 的意思是换行。`\n` 组合（依次输入这两个字符）代表一个换行符（*newline character*）。对于 `printf()` 而言，它的意思是“在下一行的最左边开始新的一行”。也就是说，打印换行符的效果与在键盘按下 **Enter** 键相同。既然如此，为何不在键入 `printf()` 参数时直接使用 **Enter** 键？因为编辑器可能认为这是直接的命令，而不是储存在源代码中的指令。换句话说，如果直接按下 **Enter** 键，编辑器会退出当前行并开始新的一行。但是，换行符仅会影响程序输出的显示格式。

换行符是一个转义序列（*escape sequence*）。转义序列用于代表难以表示或无法输入的字符。如，`\t` 代表 **Tab** 键，`\b` 代表 **Backspace** 键（退格键）。每个转义序列都以反斜杠字符（`\`）开始。我们在第 3 章中再来探讨相关内容。

这样，就解释了为什么 3 行 `printf()` 语句只打印出两行：第 1 个 `printf()` 打印的内容中不含换行符，但是第 2 和第 3 个 `printf()` 中都有换行符。

第 3 个 `printf()` 还有一些不明之处：参数中的 `%d` 在打印时有什么作用？先来看该函数的输出：

```
My favorite number is 1 because it is first.
```

对比发现，参数中的 `%d` 被数字 1 代替了，而 1 就是变量 `num` 的值。`%d` 相当于是一个占位符，其作用是指明输出 `num` 值的位置。该行和下面的 BASIC 语句很像：

```
PRINT "My favorite number is "; num; " because it is first."
```

实际上，C 语言的 `printf()` 比 BASIC 的这条语句做的事情多一些。`%` 提醒程序，要在该处打印一个变量，`d` 表明把变量作为十进制整数打印。`printf()` 函数名中的 `f` 提醒用户，这是一种格式化打印函数。`printf()` 函数有多种打印变量的格式，包括小数和十六进制整数。后面章节在介绍数据类型时，会详细介绍相关内容。

## 8. return 语句

```
return 0;
```

`return` 语句<sup>1</sup> 是程序清单 2.1 的最后一条语句。`int main(void)` 中的 `int` 表明 `main()` 函数应返回一个整数。C 标准要求 `main()` 这样做。有返回值的 C 函数要有 `return` 语句。该语句以 `return` 关键字开始，后面是待返回的值，并以分号结尾。如果遗漏 `main()` 函数中的 `return` 语句，程序在运行至最外面的右花括号（`}`）时会返回 0。因此，可以省略 `main()` 函数末尾的 `return` 语句。但是，不要在其他有返回值的函数中漏掉它。因此，强烈建议读者养成在 `main()` 函数中保留 `return` 语句的好习惯。在这种情况下，可将其看作是统一代码风格。但对于某些操作系统（包括 Linux 和 UNIX），`return` 语句有实际的用途。第 11 章再详述这个主题。

<sup>1</sup> 在 C 语言中，`return` 语句是一种跳转语句。——译者注

## 2.3 简单程序的结构

在看过一个具体的程序示例后，我们来了解一下C程序的基本结构。程序由一个或多个函数组成，必须有 `main()` 函数。函数由函数头和函数体组成。函数头包括函数名、传入该函数的信息类型和函数的返回类型。通过函数名后的圆括号可识别出函数，圆括号里可能为空，可能有参数。函数体被花括号括起来，由一系列语句、声明组成，如图 2.4 所示。本章的程序示例中有一条声明，声明了程序使用的变量名和类型。然后是一条赋值表达式语句，变量被赋予一个值。接下来是 3 条 `printf()` 语句<sup>1</sup>，调用 `printf()` 函数 3 次。最后，`main()` 以 `return` 语句结束。

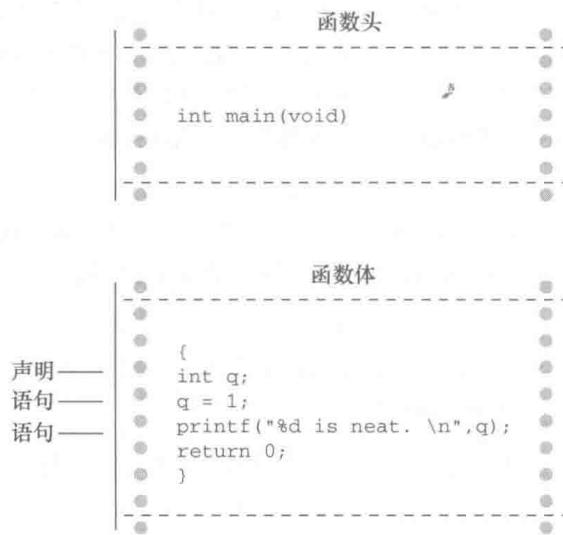


图 2.4 函数包含函数头和函数体

简而言之，一个简单的C程序的格式如下：

```
#include <stdio.h>
int main(void)
{
    语句
    return 0;
}
```

(大部分语句都以分号结尾。)

## 2.4 提高程序可读性的技巧

编写可读性高的程序是良好的编程习惯。可读性高的程序更容易理解，以后也更容易修改和更正。提高程序的可读性还有助于你理清编程思路。

前面介绍过两种提高程序可读性的技巧：选择有意义的函数名和写注释。注意，使用这两种技巧时应相得益彰，避免重复啰嗦。如果变量名是 `width`，就不必写注释说明该变量表示宽度，但是如果变量名是

<sup>1</sup> 市面上许多书籍（包括本书）都把这种语句叫作“函数调用语句”，但是历年的C标准中从来没有函数调用语句！值得一提的是，函数调用本身是一个表达式，圆括号是运算符，圆括号左边的函数名是运算对象。在C11标准中，这样的表达式是一种后缀表达式。在表达式末尾加上分号，就成了表达式语句。请初学者注意，这样的“函数调用语句”实质是表达式语句。本书的错误之处已在翻译过程中更正。——译者注

`video_routine_4`, 就要解释一下该变量名的含义。

提高程序可读性的第 3 个技巧是：在函数中用空行分隔概念上的多个部分。例如，程序清单 2.1 中用空行把声明部分和程序的其他部分区分开来。C 语言并未规定一定要使用空行，但是多使用空行能提高程序的可读性。

提高程序可读性的第 4 个技巧是：每条语句各占一行。同样，这也不是 C 语言的要求。C 语言的格式比较自由，可以把多条语句放在一行，也可以每条语句独占一行。下面的语句都没问题，但是不好看：

```
int main( void ) { int four; four
=
4
;
printf(
    "%d\n",
four); return 0; }
```

分号告诉编译器一条语句在哪里结束、下一条语句在哪里开始。如果按照本章示例的约定来编写代码（见图 2.5），程序的逻辑会更清晰。



图 2.5 提高程序的可读性

## 2.5 进一步使用 C

本章的第一个程序相当简单，下面的程序清单 2.2 也不太难。

### 程序清单 2.2 fathom\_ft.c 程序

```
// fathom_ft.c -- 把2音寻转换成英寸
#include <stdio.h>
int main(void)
{
    int feet, fathoms;

    fathoms = 2;
    feet = 6 * fathoms;
    printf("There are %d feet in %d fathoms!\n", feet, fathoms);
    printf("Yes, I said %d feet!\n", 6 * fathoms);

    return 0;
}
```

与程序清单 2.1 相比，以上代码有什么新内容？这段代码提供了程序描述，声明了多个变量，进行了乘法运算，并打印了两个变量的值。下面我们更详细地分析这些内容。

### 2.5.1 程序说明

程序在开始处有一条注释（使用新的注释风格），给出了文件名和程序的目的。写这种程序说明很简单、不费时，而且在以后浏览或打印程序时很有帮助。

### 2.5.2 多条声明

接下来，程序在一条声明中声明了两个变量，而不是一个变量。为此，要在声明中用逗号隔开两个变量（feet 和 fathoms）。也就是说，

```
int feet, fathoms;
```

和

```
int feet;  
int fathoms;
```

等价。

### 2.5.3 乘法

然后，程序进行了乘法运算。利用计算机强大的计算能力来计算 6 乘以 2。C 语言和许多其他语言一样，用 \* 表示乘法。因此，语句

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

的意思是“查找变量 fathoms 的值，用 6 乘以该值，并把计算结果赋给变量 feet”。

### 2.5.4 打印多个值

最后，程序以新的方式使用 printf() 函数。如果编译并运行该程序，输出应该是这样：

```
There are 12 feet in 2 fathoms!  
Yes, I said 12 feet!
```

程序的第一个 printf() 中进行了两次替换。双引号后面的第 1 个变量（feet）替换了双引号中的第 1 个%d；双引号后面的第 2 个变量（fathoms）替换了双引号中的第 2 个%d。注意，待输出的变量列于双引号的后面。还要注意，变量之间要用逗号隔开。

第 2 个 printf() 函数说明待打印的值不一定是变量，只要可求值得出合适类型值的项即可，如 6 \* fathoms。

该程序涉及的范围有限，但它是把音寻<sup>1</sup>转换成英寸程序的核心部分。我们还需要把其他值通过交互的方式赋给 feet，其方法将在后面章节中介绍。

## 2.6 多个函数

到目前为止，介绍的几个程序都只使用了 printf() 函数。程序清单 2.3 演示了除 main() 以外，如何把自己的函数加入程序中。

<sup>1</sup> 音寻，也称为寻。航海用的深度单位，1 英寻=6 英尺=1.8 米，通常用在海图上测量水深。——译者注

**程序清单 2.3 two\_func.c 程序**

```
/* two_func.c -- 一个文件中包含两个函数 */
#include <stdio.h>
void butler(void); /* ANSI/ISO C 函数原型 */
int main(void)
{
    printf("I will summon the butler function.\n");
    butler();
    printf("Yes. Bring me some tea and writeable DVDs.\n");

    return 0;
}
void butler(void) /* 函数定义开始 */
{
    printf("You rang, sir?\n");
}
```

该程序的输出如下：

```
I will summon the butler function.
You rang, sir?
Yes. Bring me some tea and writeable DVDs.
```

butler() 函数在程序中出现了 3 次。第 1 次是函数原型 (*prototype*)，告知编译器在程序中要使用该函数；第 2 次以函数调用 (*function call*) 的形式出现在 main() 中；最后一次出现在函数定义 (*function definition*) 中，函数定义即是函数本身的源代码。下面逐一分析。

C90 标准新增了函数原型，旧式的编译器可能无法识别（稍后我们将介绍，如果使用这种编译器应该怎么做）。函数原型是一种声明形式，告知编译器正在使用某函数，因此函数原型也被称为函数声明 (*function declaration*)。函数原型还指明了函数的属性。例如，butler() 函数原型中的第 1 个 void 表明，butler() 函数没有返回值（通常，被调函数会向主调函数返回一个值，但是 butler() 函数没有）。第 2 个 void (butler(void) 中的 void) 的意思是 butler() 函数不带参数。因此，当编译器运行至此，会检查 butler() 是否使用得当。注意，void 在这里的意思是“空的”，而不是“无效”。

早期的 C 语言支持一种更简单的函数声明，只需指定返回类型，不用描述参数：

```
void butler();
```

早期的 C 代码中的函数声明就类似上面这样，不是现在的函数原型。C90、C99 和 C11 标准都承认旧版本的形式，但是也表明了会逐渐淘汰这种过时的写法。如果要使用以前写的 C 代码，就需要把旧式声明转换成函数原型。本书在后面的章节会继续介绍函数原型的相关内容。

接下来我们继续分析程序。在 main() 中调用 butler() 很简单，写出函数名和圆括号即可。当 butler() 执行完毕后，程序会继续执行 main() 中的下一条语句。

程序的最后部分是 butler() 函数的定义，其形式和 main() 相同，都包含函数头和用花括号括起来的函数体。函数头重述了函数原型的信息：butler() 不带任何参数，且没有返回值。如果使用老式编译器，请去掉圆括号中的 void。

这里要注意，何时执行 butler() 函数取决于它在 main() 中被调用的位置，而不是 butler() 的定义在文件中的位置。例如，把 butler() 函数的定义放在 main() 定义之前，不会改变程序的执行顺序，butler() 函数仍然在两次 printf() 调用之间被调用。记住，无论 main() 在程序文件处于什么位置，所有的 C 程序都从 main() 开始执行。但是，C 的惯例是把 main() 放在开头，因为它提供了程序的基本框架。

C标准建议，要为程序中用到的所有函数提供函数原型。标准 include 文件（包含文件）为标准库函数提供可函数原型。例如，在C标准中，stdio.h文件包含了printf()的函数原型。第6章最后一个示例演示了如何使用带返回值的函数，第9章将详细全面地介绍函数。

## 2.7 调试程序

现在，你可以编写一个简单的C程序，但是可能会犯一些简单的错误。程序的错误通常叫做bug，找出并修正错误的过程叫做调试(debug)。程序清单2.4是一个有错误的程序，看看你能找出几处。

程序清单2.4 nogood.c程序

```
/* nogood.c -- 有错误的程序 */
#include <stdio.h>
int main(void)
{
    int n, int n2, int n3;

    /* 该程序有多处错误
    n = 5;
    n2 = n * n;
    n3 = n2 * n2;
    printf("n = %d, n squared = %d, n cubed = %d\n", n, n2, n3)

    return 0;
}
```

### 2.7.1 语法错误

程序清单2.4中有多处语法错误。如果不遵循C语言的规则就会犯语法错误。这类似于英文中的语法错误。例如，看看这个句子：*Bugs frustrate be can<sup>1</sup>*。该句子中的英文单词都是有效的单词（即，拼写正确），但是并未按照正确的顺序组织句子，而且用词也不妥。C语言的语法错误指的是，把有效的C符号放在错误的地方。

nogood.c程序中有哪些错误？其一，main()函数体使用圆括号来代替花括号。这就是把C符号用错了地方。其二，变量声明应该这样写：

```
int n, n2, n3;
```

或者，这样写：

```
int n;
int n2;
int n3;
```

其三，main()中的注释末尾漏掉了\*/（另一种修改方案是，用//替换/\*）。最后，printf()语句末尾漏掉了分号。

如何发现程序的语法错误？首先，在编译之前，浏览源代码看是否能发现一些明显的错误。接下来，查看编译器是否发现错误，检查程序的语法错误是它的工作之一。在编译程序时，编译器发现错误会报告错误信息，指出每一处错误的性质和具体位置。

<sup>1</sup> 要理解该句子存在语法错误，需要具备基本的英文语法知识。——译者注

尽管如此，编译器也有出错的时候。也许某处隐藏的语法错误会导致编译器误判。例如，由于 `nogood.c` 程序未正确声明 `n2` 和 `n3`，会导致编译器在使用这些变量时发现更多问题。实际上，有时不用把编译器报告的所有错误逐一修正，仅修正第 1 条或前几处错误后，错误信息就会少很多。继续这样做，直到编译器不再报错。编译器另一个常见的毛病是，报错的位置比真正的错误位置滞后一行。例如，编译器在编译下一行时才会发现上一行缺少分号。因此，如果编译器报错某行缺少分号，请检查上一行。

## 2.7.2 语义错误

语义错误是指意思上的错误。例如，考虑这个句子：*Scornful derivatives sing greenly*（轻蔑的衍生物不熟练地唱歌）。句中的形容词、名词、动词和副词都在正确的位置上，所以语法正确。但是，却让人不知所云。在 C 语言中，如果遵循了 C 规则，但是结果不正确，那就是犯了语义错误。程序示例中有这样的错误：

```
n3 = n2 * n2;
```

此处，`n3` 原意表示 `n` 的 3 次方，但是代码中的 `n3` 被设置成 `n` 的 4 次方 (`n2 = n * n`)。

编译器无法检测语义错误，因为这类错误并未违反 C 语言的规则。编译器无法了解你的真正意图，所以你只能自己找出这些错误。例如，假设你修正了程序的语法错误，程序应该如程序清单 2.5 所示：

**程序清单 2.5 stillbad.c 程序**

---

```
/* stillbad.c -- 修复了语法错误的程序 */
#include <stdio.h>
int main(void)
{
    int n, n2, n3;

    /* 该程序有一个语义错误 */
    n = 5;
    n2 = n * n;
    n3 = n2 * n2;
    printf("n = %d, n squared = %d, n cubed = %d\n", n, n2, n3);

    return 0;
}
```

---

该程序的输出如下：

```
n = 5, n squared = 25, n cubed = 625
```

如果对简单的立方比较熟悉，就会注意到 625 不对。下一步是跟踪程序的执行步骤，找出程序如何得出这个答案。对于本例，通过查看代码就会发现其中的错误，但是，还应该学习更系统的方法。方法之一是，把自己想象成计算机，跟着程序的步骤一步一步地执行。下面，我们来试试这种方法。

`main()` 函数体一开始就声明了 3 个变量：`n`、`n2`、`n3`。你可以画出 3 个盒子并把变量名写在盒子上来模拟这种情况（见图 2.6）。接下来，程序把 5 赋给变量 `n`。你可以在标签为 `n` 的盒子里写上 5。接着，程序把 `n` 和 `n` 相乘，并把乘积赋给 `n2`。因此，查看标签为 `n` 的盒子，其值是 5，5 乘以 5 得 25，于是把 25 放进标签为 `n2` 的盒子里。为了模拟下一条语句 (`n3 = n2 * n2`)，查看 `n2` 盒子，发现其值是 25。25 乘以 25 得 625，把 625 放进标签为 `n3` 的盒子。原来如此！程序中计算的是 `n2` 的平方，不是用 `n2` 乘以 `n` 得到 `n` 的 3 次方。

对于上面的程序示例，检查程序的过程可能过于繁琐。但是，用这种方法一步一步查看程序的执行情况，通常是发现程序问题所在的良方。

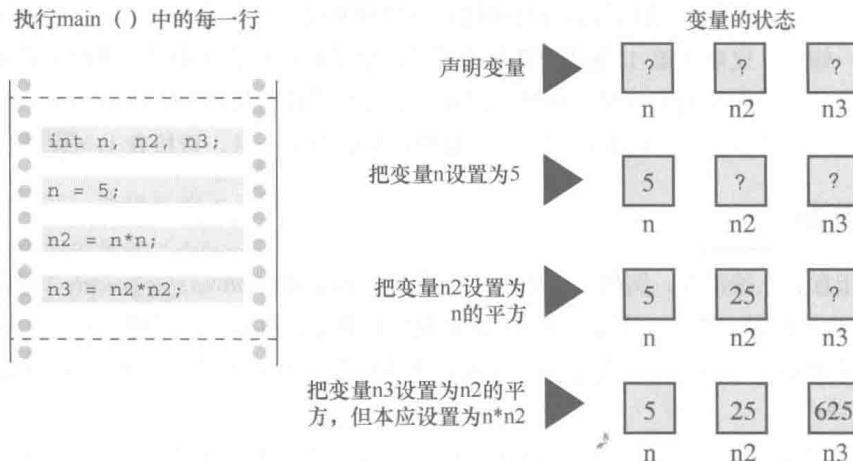


图 2.6 跟踪程序的执行步骤

### 2.7.3 程序状态

通过逐步跟踪程序的执行步骤，并记录每个变量，便可监视程序的状态。程序状态（*program state*）是在程序的执行过程中，某给定点上所有变量值的集合。它是计算机当前状态的一个快照。

我们刚刚讨论了一种跟踪程序状态的方法：自己模拟计算机逐步执行程序。但是，如果程序中有 10000 次循环，这种方法恐怕行不通。不过，你可以跟踪一小部分循环，看看程序是否按照预期的方式执行。另外，还要考虑一种情况：你很可能按照自己所想去执行程序，而不是根据实际写出来的代码去执行。因此，要尽量忠实代码来模拟。

定位语义错误的另一种方法是：在程序中的关键点插入额外的 `printf()` 语句，以监视制定变量值的变化。通过查看值的变化可以了解程序的执行情况。对程序的执行满意后，便可删除额外的 `printf()` 语句，然后重新编译。

检测程序状态的第 3 种方法是使用调试器。调试器（*debugger*）是一种程序，让你一步一步运行另一个程序，并检查该程序变量的值。调试器有不同的使用难度和复杂度。较高级的调试器会显示正在执行的源代码行号。这在检查有多条执行路径的程序时很方便，因为很容易知道正在执行哪条路径。如果你的编译器自带调试器，现在可以花点时间学会怎么使用它。例如，试着调试一下程序清单 2.4。

## 2.8 关键字和保留标识符

关键字是 C 语言的词汇。它们对 C 而言比较特殊，不能用它们作为标识符（如，变量名）。许多关键字用于指定不同的类型，如 `int`。还有一些关键字（如，`if`）用于控制程序中语句的执行顺序。在表 2.2 中所列的 C 语言关键字中，粗体表示的是 C90 标准新增的关键字，斜体表示的 C99 标准新增的关键字，粗斜体表示的是 C11 标准新增的关键字。

表 2.2 ISO C 关键字

<code>auto</code>	<code>extern</code>	<code>short</code>	<code>while</code>
<code>break</code>	<code>float</code>	<code>signed</code>	<code>_Alignas</code>
<code>case</code>	<code>for</code>	<code>sizeof</code>	<code>_Alignof</code>
<code>char</code>	<code>goto</code>	<code>static</code>	<code>_Atomic</code>

续表

<code>const</code>	<code>if</code>	<code>struct</code>	<code>_Bool</code>
<code>continue</code>	<code>inline</code>	<code>switch</code>	<code>_Complex</code>
<code>default</code>	<code>int</code>	<code>typedef</code>	<code>_Generic</code>
<code>do</code>	<code>long</code>	<code>union</code>	<code>_Imaginary</code>
<code>double</code>	<code>register</code>	<code>unsigned</code>	<code>_Noreturn</code>
<code>else</code>	<code>restrict</code>	<code>void</code>	<code>_Static_assert</code>
<code>enum</code>	<code>return</code>	<code>volatile</code>	<code>_Thread_local</code>

如果使用关键字不当（如，用关键字作为变量名），编译器会将其视为语法错误。还有一些保留标识符（*reserved identifier*），C 语言已经指定了它们的用途或保留它们的使用权，如果你使用这些标识符来表示其他意思会导致一些问题。因此，尽管它们也是有效的名称，不会引起语法错误，也不能随便使用。保留标识符包括那些以下划线字符开头的标识符和标准库函数名，如 `printf()`。

## 2.9 关键概念

编程是一件富有挑战性的事情。程序员要具备抽象和逻辑的思维，并谨慎地处理细节问题（编译器会强迫你注意细节问题）。平时和朋友交流时，可能用错几个单词，犯一两个语法错误，或者说几句不完整的句子，但是对方能明白你想说什么。而编译器不允许这样，对它而言，几乎正确仍然是错误。

编译器不会在下面讲到的概念性问题上帮助你。因此，本书在这一章中介绍一些关键概念帮助读者弥补这部分的内容。

在本章中，读者的目标应该是理解什么是 C 程序。可以把程序看作是你希望计算机如何完成任务的描述。编译器负责处理一些细节工作，例如把你希望计算机完成的任务转换成底层的机器语言（如果从量化方面来解释编译器所做的工作，它可以把 1KB 的源文件创建成 60KB 的可执行文件；即使是一个很简单的 C 程序也要用大量的机器语言来表示）。由于编译器不具有真正的智能，所以你必须用编译器能理解的术语表达你的意图，这些术语就是 C 语言标准规定的形式规则（尽管有些约束，但总比直接用机器语言方便得多）。

编译器希望接收到特定格式的指令，我们在本章已经介绍过。作为程序员的任务是，在符合 C 标准的编译器框架中，表达你希望程序应该如何完成任务的想法。

## 2.10 本章小结

C 程序由一个或多个 C 函数组成。每个 C 程序必须包含一个 `main()` 函数，这是 C 程序要调用的第一个函数。简单的函数由函数头和后面的一对花括号组成，花括号中是由声明、语句组成的函数体。

在 C 语言中，大部分语句都以分号结尾。声明为变量创建变量名和标识该变量中储存的数据类型。变量名是一种标识符。赋值表达式语句把值赋给变量，或者更一般地说，把值赋给存储空间。函数表达式语句用于调用指定的已命名函数。调用函数执行完毕后，程序会返回到函数调用后面的语句继续执行。

`printf()` 函数用于输出想要表达的内容和变量的值。

一门语言的语法是一套规则，用于管理语言中各有效语句组合在一起的方式。语句的语义是语句要表达的意思。编译器可以检测出语法错误，但是程序里的语义错误只有在编译完之后才能从程序的行为中表现出来。检查程序是否有语义错误要跟踪程序的状态，即程序每执行一步后所有变量的值。

最后，关键字是 C 语言的词汇。

## 2.11 复习题

复习题的参考答案在附录A中。

1. C语言的基本模块是什么？
2. 什么是语法错误？写出一个英语例子和C语言例子。
3. 什么是语义错误？写出一个英语例子和C语言例子。
4. Indiana Sloth 编写了下面的程序，并征求你的意见。请帮助他评定。

```
include studio.h
int main(void) /* 该程序打印一年有多少周 */
{
    int s

    s := 56;
    print(There are s weeks in a year.);
    return 0;
```

5. 假设下面的4个例子都是完整程序中的一部分，它们都输出什么结果？

```
a. printf("Baa Baa Black Sheep.");
printf("Have you any wool?\n");
b. printf("Begone!\nO creature of lard!\n");
c. printf("What?\nNo/fish?\n");
d. int num;
num = 2;
printf("%d + %d = %d", num, num, num + num);
```

6. 在main、int、function、char、=中，哪些是C语言的关键字？

7. 如何以下面的格式输出变量words和lines的值（这里，3020和350代表两个变量的值）？  
There were 3020 words and 350 lines.

8. 考虑下面的程序：

```
#include <stdio.h>
int main(void)
{
    int a, b;

    a = 5;
    b = 2; /* 第7行 */
    b = a; /* 第8行 */
    a = b; /* 第9行 */
    printf("%d %d\n", b, a);
    return 0;
}
```

请问，在执行完第7、第8、第9行后，程序的状态分别是什么？

9. 考虑下面的程序：

```
#include <stdio.h>
int main(void)
{
    int x, y;

    x = 10;
```

```

y = 5;      /* 第 7 行 */
y = x + y; /* 第 8 行 */
x = x*y;   /* 第 9 行 */
printf("%d %d\n", x, y);
return 0;
}

```

请问，在执行完第 7、第 8、第 9 行后，程序的状态分别是什么？

## 2.12 编程练习

纸上得来终觉浅，绝知此事要躬行。读者应该试着编写一两个简单的程序，体会一下编写程序是否和阅读本章介绍的这样轻松。题目中会给出一些建议，但是应该尽量自己思考这些问题。一些编程答案练习的答案可在出版商网站获取。

1. 编写一个程序，调用一次 printf() 函数，把你的姓名打印在一行。再调用一次 printf() 函数，把你的姓名分别打印在两行。然后，再调用两次 printf() 函数，把你的姓名打印在一行。输出应如下所示（当然要把示例的内容换成你的姓名）：

Gustav Mahler	← 第 1 次打印的内容
Gustav	← 第 2 次打印的内容
Mahler	← 仍是第 2 次打印的内容
Gustav Mahler	← 第 3 次和第 4 次打印的内容

2. 编写一个程序，打印你的姓名和地址。
3. 编写一个程序把你的年龄转换成天数，并显示这两个值。这里不用考虑闰年的问题。
4. 编写一个程序，生成以下输出：

```

For he's a jolly good fellow!
For he's a jolly good fellow!
For he's a jolly good fellow!
Which nobody can deny!

```

除了 main() 函数以外，该程序还要调用两个自定义函数：一个名为 jolly()，用于打印前 3 条消息，调用一次打印一条；另一个函数名为 deny()，打印最后一条消息。

5. 编写一个程序，生成以下输出：
- ```

Brazil, Russia, India, China
India, China,
Brazil, Russia

```
- 除了 main() 以外，该程序还要调用两个自定义函数：一个名为 br()，调用一次打印一次“Brazil, Russia”；另一个名为 ic()，调用一次打印一次 “India, China”。其他内容在 main() 函数中完成。
6. 编写一个程序，创建一个整型变量 toes，并将 toes 设置为 10。程序中还要计算 toes 的两倍和 toes 的平方。该程序应打印 3 个值，并分别描述以示区分。
  7. 许多研究表明，微笑益处多多。编写一个程序，生成以下格式的输出：

```

Smile!Smile!Smile!
Smile!Smile!
Smile!

```

该程序要定义一个函数，该函数被调用一次打印一次 “Smile!”，根据程序的需要使用该函数。

8. 在 C 语言中，函数可以调用另一个函数。编写一个程序，调用一个名为 one\_three() 的函数。该

函数在一行打印单词“one”，再调用第2个函数 two()，然后在另一行打印单词“three”。two()  
函数在一行显示单词“two”。main() 函数在调用 one\_three() 函数前要打印短语“starting  
now:”，并在调用完毕后显示短语“done!”。因此，该程序的输出应如下所示：

starting now:

one

two

three

done!

输出结果如下图所示：

```
cd Desktop/学习/C语言/第2章
gcc test.c
./test
starting now:
one
two
three
done!
```

本章介绍以下内容：

- 关键字：int、short、long、unsigned、char、float、double、\_Bool、\_Complex、\_Imaginary
- 运算符：sizeof()
- 函数：scanf()
- 整数类型和浮点数类型的区别
- 如何书写整型和浮点型常数，如何声明这些类型的变量
- 如何使用printf()和scanf()函数读写不同类型的值

程序离不开数据。把数字、字母和文字输入计算机，就是希望它利用这些数据完成某些任务。例如，需要计算一份利息或显示一份葡萄酒商的排序列表。本章除了介绍如何读取数据外，还将教会读者如何操控数据。

C 语言提供两大系列的多种数据类型。本章详细介绍两大数据类型：整数类型和浮点数类型，讲解这些数据类型是什么、如何声明它们、如何以及何时使用它们。除此之外，还将介绍常量和变量的区别。读者很快就能看到第 1 个交互式程序。

## 3.1 示例程序

本章仍从一个简单的程序开始。如果发现有不熟悉的内容，别担心，我们稍后会详细解释。该程序的意图比较明了，请试着编译并运行程序清单 3.1 中的源代码。为了节省时间，在输入源代码时可省略注释。

程序清单 3.1 platinum.c 程序

```
/* platinum.c -- your weight in platinum */
#include <stdio.h>
int main(void)
{
    float weight; /* 你的体重 */ 
    float value; /* 相等重量的白金价值 */

    printf("Are you worth your weight in platinum?\n");
    printf("Let's check it out.\n");
    printf("Please enter your weight in pounds: ");

    /* 获得用户的输入 */
    scanf("%f", &weight);
    /* 假设白金的价格是每盎司$1700 */
}
```

```

/* 14.5833 用于把英镑常衡盎司转换为金衡盎司1 */
value = 1700.0 * weight * 14.5833;
printf("Your weight in platinum is worth $%.2f.\n", value);
printf("You are easily worth that! If platinum prices drop,\n");
printf("eat more to maintain your value.\n");

return 0;
}

```

### 提示 错误与警告

如果输入程序时打错（如，漏了一个分号），编译器会报告语法错误消息。然而，即使输入正确无误，编译器也可能给出一些警告，如“警告：从 double 类型转换成 float 类型可能会丢失数据”。错误消息表明程序中有错，不能进行编译。而警告则表明，尽管编写的代码有效，但可能不是程序员想要的。警告并不终止编译。特殊的警告与 C 如何处理 1700.0 这样的值有关。本例不必理会这个问题，本章稍后会进一步说明。

输入该程序时，可以把 1700.0 改成贵金属白金当前的市价，但是不要改动 14.5833，该数是 1 英镑的金衡盎司数（金衡盎司用于衡量贵金属，而英镑常衡盎司用于衡量人的体重）。

注意，“enter your weight”的意思是输入你的体重，然后按下 **Enter** 或 **Return** 键（不要键入体重后就一直等着）。按下 **Enter** 键是告知计算机，你已完成输入数据。该程序需要你输入一个数字（如，155），而不是单词（如，too much）。如果输入字母而不是数字，会导致程序出问题。这个问题要用 **if** 语句来解决（详见第 7 章），因此请先输入数字。下面是程序的输出示例：

```

Are you worth your weight in platinum?
Let's check it out.
Please enter your weight in pounds: 156
Your weight in platinum is worth $3867491.25.
You are easily worth that! If platinum prices drop,
eat more to maintain your value.

```

### 程序调整

即使用第 2 章介绍的方法，在程序中添加下面一行代码：

```
getchar();
```

程序的输出是否依旧在屏幕上一闪而过？本例，需要调用两次 **getchar()** 函数：

```
getchar();
getchar();
```

**getchar()** 函数读取下一个输入字符，因此程序会等待用户输入。在这种情况下，键入 156 并按下 **Enter**（或 **Return**）键（发送一个换行符），然后 **scanf()** 读取键入的数字，第 1 个 **getchar()** 读取换行符，第 2 个 **getchar()** 让程序暂停，等待输入。

<sup>1</sup> 欧美日常使用的度量衡单位是常衡盎司 (*avoirdupois ounce*)，而欧美黄金市场上使用的黄金交易计量单位是金衡盎司 (*troy ounce*)。国际黄金市场上的报价，其单位“盎司”都指的是黄金盎司。常衡盎司属英制计量单位，做重量单位时也称为英两。相关换算参考如下：1 常衡盎司 = 28.350 克，1 金衡盎司 = 31.104 克，16 常衡盎司 = 1 磅。该程序的单位转换思路是：把磅换算成金衡盎司，即  $28.350 \div 31.104 \times 16 = 14.5833$ 。——译者注

### 3.1.1 程序中的新元素

程序清单 3.1 中包含 C 语言的一些新元素。

- 注意，代码中使用了一种新的变量声明。前面的例子中只使用了整数类型的变量（int），但是本例使用了浮点数类型（float）的变量，以便处理更大范围的数据。float 类型可以储存带小数的数字。
- 程序中演示了常量的几种新写法。现在可以使用带小数点的数了。
- 为了打印新类型的变量，在 printf() 中使用%f 来处理浮点值。%.2f 中的.2 用于精确控制输出，指定输出的浮点数只显示小数点后面两位。
- scanf() 函数用于读取键盘的输入。%f 说明 scanf() 要读取用户从键盘输入的浮点数，&weight 告诉 scanf() 把输入的值赋给名为 weight 的变量。scanf() 函数使用& 符号表明找到 weight 变量的地点。下一章将详细讨论&。就目前而言，请按照这样写。
- 也许本程序最突出的新特点是它的交互性。计算机向用户询问信息，然后用户输入数字。与非交互式程序相比，交互式程序用起来更有趣。更重要的是，交互式使得程序更加灵活。例如，示例程序可以使用任何合理的体重，而不只是 156 磅。不必重写程序，就可以根据不同体重进行计算。scanf() 和 printf() 函数用于实现这种交互。scanf() 函数读取用户从键盘输入的数据，并把数据传递给程序；printf() 函数读取程序中的数据，并把数据显示在屏幕上。把两个函数结合起来，就可以建立人机双向通信（见图 3.1），这让使用计算机更加饶有趣味。

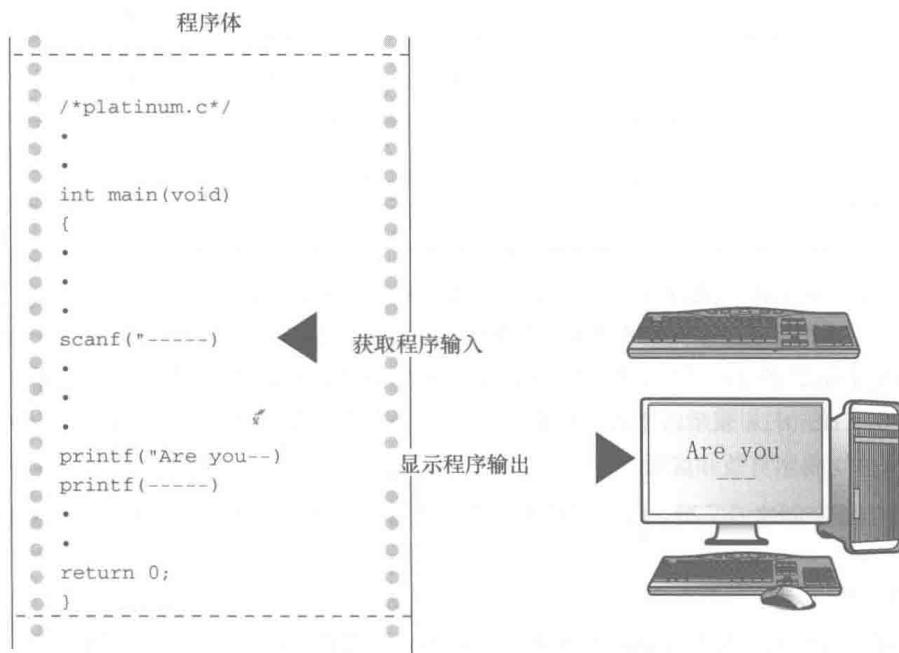


图 3.1 程序中的 scanf() 和 printf() 函数

本章着重解释上述新特性中的前两项：各种数据类型的变量和常量。第 4 章将介绍后 3 项。

## 3.2 变量与常量数据

在程序的指导下，计算机可以做许多事情，如数值计算、名字排序、执行语言或视频命令、计算彗星轨道、准备邮件列表、拨电话号码、画画、做决策或其他你能想到的事情。要完成这些任务，程序需要使用数据，即承载信息的数字和字符。有些数据类型在程序使用之前已经预先设定好了，在整个程序的运行过程中没有变化，这些称为常量（*constant*）。其他数据类型在程序运行期间可能会改变或被赋值，这些称为变量（*variable*）。在示例程序中，`weight` 是一个变量，`14.5833` 是一个常量。那么，`1700.0` 是常量还是变量？在现实生活中，白金的价格不会是常量，但是在程序中，像 `1700.0` 这样的价格被视为常量。

## 3.3 数据：数据类型关键字

不仅变量和常量不同，不同的数据类型之间也有差异。一些数据类型表示数字，一些数据类型表示字母（更普遍地说是字符）。C 通过识别一些基本的数据类型来区分和使用这些不同的数据类型。如果数据是常量，编译器一般通过用户书写的形势来识别类型（如，`42` 是整数，`42.100` 是浮点数）。但是，对变量而言，要在声明时指定其类型。稍后会详细介绍如何声明变量。现在，我们先来了解一下 C 语言的基本类型关键字。K&C 给出了 7 个与类型相关的关键字。C90 标准添加了 2 个关键字，C99 标准又添加了 3 个关键字（见表 3.1）。

表 3.1 C 语言的数据类型关键字

| 最初 K&R 给出的关键字         | C90 标准添加的关键字        | C99 标准添加的关键字            |
|-----------------------|---------------------|-------------------------|
| <code>int</code>      | <code>signed</code> | <code>_Bool</code>      |
| <code>long</code>     | <code>void</code>   | <code>_Complex</code>   |
| <code>short</code>    |                     | <code>_Imaginary</code> |
| <code>unsigned</code> |                     |                         |
| <code>char</code>     |                     |                         |
| <code>float</code>    |                     |                         |
| <code>double</code>   |                     |                         |

在 C 语言中，用 `int` 关键字来表示基本的整数类型。后 3 个关键字（`long`、`short` 和 `unsigned`）和 C90 新增的 `signed` 用于提供基本整数类型的变式，例如 `unsigned short int` 和 `long long int`。`char` 关键字用于指定字母和其他字符（如，#、\$、% 和 \*）。另外，`char` 类型也可以表示较小的整数。`float`、`double` 和 `long double` 表示带小数点的数。`_Bool` 类型表示布尔值（`true` 或 `false`），`_complex` 和 `_Imaginary` 分别表示复数和虚数。

通过这些关键字创建的类型，按计算机的储存方式可分为两大基本类型：整数类型和浮点数类型。

### 位、字节和字

位、字节和字是描述计算机数据单元或存储单元的术语。这里主要指存储单元。

最小的存储单元是位（*bit*），可以储存 0 或 1（或者说，位用于设置“开”或“关”）。虽然 1 位储存的信息有限，但是计算机中位的数量十分庞大。位是计算机内存的基本构建块。

字节（*byte*）是常用的计算机存储单位。对于几乎所有的机器，1 字节均为 8 位。这是字节的标准定义，至少在衡量存储单位时是这样（但是，C 语言对此有不同的定义，请参阅本章 3.4.3 节）。既然 1 位可以表示 0 或 1，那么 8 位字节就有 256（2 的 8 次方）种可能的 0、1 的组合。通过二进制编码（仅

用 0 和 1 便可表示数字），便可表示 0~255 的整数或一组字符（第 15 章将详细讨论二进制编码，如果感兴趣可以现在浏览一下该章的内容）。

字 (word) 是设计计算机时给定的自然存储单位。对于 8 位的微型计算机（如，最初的苹果机），1 个字长只有 8 位。从那以后，个人计算机字长增至 16 位、32 位，直到目前的 64 位。计算机的字长越大，其数据转移越快，允许的内存访问也更多。

### 3.3.1 整数和浮点数

整数类型？浮点数类型？如果觉得这些术语非常陌生，别担心，下面先简述它们的含义。如果不熟悉位、字节和字的概念，请阅读上面方框中的内容。刚开始学习时，不必了解所有的细节，就像学习开车之前不必详细了解汽车内部引擎的原理一样。但是，了解一些计算机或汽车引擎内部的原理会对你有所帮助。

对我们而言，整数和浮点数的区别是它们的书写方式不同。对计算机而言，它们的区别是储存方式不同。下面详细介绍整数和浮点数。

### 3.3.2 整数

和数学的概念一样，在 C 语言中，整数是没有小数部分的数。例如，2、-23 和 2456 都是整数。而 3.14、0.22 和 2.000 都不是整数。计算机以二进制数字储存整数，例如，整数 7 以二进制写是 111。因此，要在 8 位字节中储存该数字，需要把前 5 位都设置成 0，后 3 位设置成 1（如图 3.2 所示）。

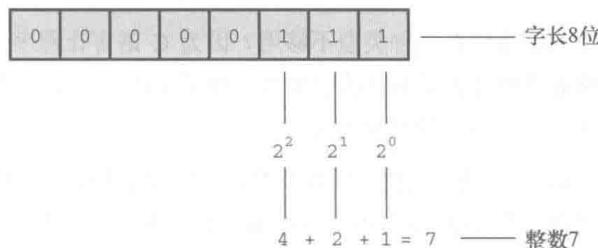


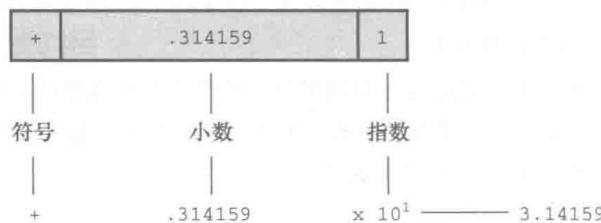
图 3.2 使用二进制编码储存整数 7

### 3.3.3 浮点数

浮点数与数学中实数的概念差不多。2.75、3.16E7、7.00 和 2e-8 都是浮点数。注意，在一个值后面加上一个小数点，该值就成为一个浮点值。所以，7 是整数，7.00 是浮点数。显然，书写浮点数有多种形式。稍后将详细介绍 e 记数法，这里先做简要介绍：3.16E7 表示  $3.16 \times 10^7$  (3.16 乘以 10 的 7 次方)。其中， $10^7=10000000$ ，7 被称为 10 的指数。

这里关键要理解浮点数和整数的储存方案不同。计算机把浮点数分成小数部分和指数部分来表示，而且分开储存这两部分。因此，虽然 7.00 和 7 在数值上相同，但是它们的储存方式不同。在十进制下，可以把 7.0 写成 0.7E1。这里，0.7 是小数部分，1 是指数部分。图 3.3 演示了一个储存浮点数的例子。当然，计算机在内部使用二进制和 2 的幂进行储存，而不是 10 的幂。第 15 章将详述相关内容。现在，我们着重讲解这两种类型的实际区别。

- 整数没有小数部分，浮点数有小数部分。
- 浮点数可以表示的范围比整数大。参见本章末的表 3.3。
- 对于一些算术运算（如，两个很大的数相减），浮点数损失的精度更多。

图 3.3 以浮点格式（十进制）储存  $\pi$  的值

- 因为在任何区间内（如，1.0 到 2.0 之间）都存在无穷多个实数，所以计算机的浮点数不能表示区间内所有的值。浮点数通常只是实际值的近似值。例如，7.0 可能被储存为浮点值 6.99999。稍后会讨论更多精度方面的内容。
- 过去，浮点运算比整数运算慢。不过，现在许多 CPU 都包含浮点处理器，缩小了速度上的差距。

## 3.4 C 语言基本数据类型

本节将详细介绍 C 语言的基本数据类型，包括如何声明变量、如何表示字面值常量（如，5 或 2.78），以及典型的用法。一些老式的 C 语言编译器无法支持这里提到的所有类型，请查阅你使用的编译器文档，了解可以使用哪些类型。

### 3.4.1 int 类型

C 语言提供了许多整数类型，为什么一种类型不够用？因为 C 语言让程序员针对不同情况选择不同的类型。特别是，C 语言中的整数类型可表示不同的取值范围和正值。一般情况使用 `int` 类型即可，但是为满足特定任务和机器的要求，还可以选择其他类型。

`int` 类型是有符号整型，即 `int` 类型的值必须是整数，可以是正整数、负整数或零。其取值范围依计算机系统而异。一般而言，储存一个 `int` 要占用一个机器字长。因此，早期的 16 位 IBM PC 兼容机使用 16 位来储存一个 `int` 值，其取值范围（即 `int` 值的取值范围）是 -32768~32767。目前的个人计算机一般是 32 位，因此用 32 位储存一个 `int` 值。现在，个人计算机产业正逐步向着 64 位处理器发展，自然能储存更大的整数。ISO C 规定 `int` 的取值范围最小为 -32768~32767。一般而言，系统用一个特殊位的值表示有符号整数的正负号。第 15 章将介绍常用的方法。

#### 1. 声明 `int` 变量

第 2 章中已经用 `int` 声明过基本整型变量。先写上 `int`，然后写变量名，最后加上一个分号。要声明多个变量，可以单独声明每个变量，也可在 `int` 后面列出多个变量名，变量名之间用逗号分隔。下面都是有效的声明：

```
int erns;
int hogs, cows, goats;
```

可以分别在 4 条声明中声明各变量，也可以在一条声明中声明 4 个变量。两种方法的效果相同，都为 4 个 `int` 大小的变量赋予名称并分配内存空间。

以上声明创建了变量，但是并没有给它们提供值。变量如何获得值？前面介绍过在程序中获取值的两种途径。第 1 种途径是赋值：

```
cows = 112;
```

第 2 种途径是，通过函数（如，`scanf()`）获得值。接下来，我们着重介绍第 3 种途径。

## 2. 初始化变量

初始化 (*initialize*) 变量就是为变量赋一个初始值。在 C 语言中，初始化可以直接在声明中完成。只需在变量名后面加上赋值运算符 (=) 和待赋给变量的值即可。如下所示：

```
int hogs = 21;
int cows = 32, goats = 14;
int dogs, cats = 94; /* 有效，但是这种格式很糟糕 */
```

以上示例的最后一行，只初始化了 `cats`，并未初始化 `dogs`。这种写法很容易让人误认为 `dogs` 也被初始化为 94，所以最好不要把初始化的变量和未初始化的变量放在同一条声明中。

简而言之，声明为变量创建和标记存储空间，并为其指定初始值（如图 3.4 所示）。

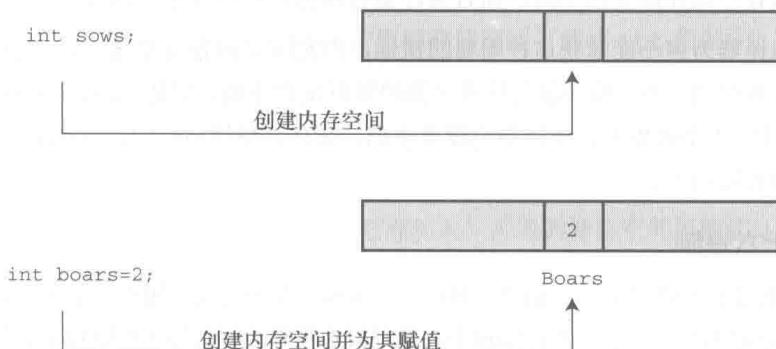


图 3.4 定义并初始化变量

## 3. int 类型常量

上面示例中出现的整数（21、32、14 和 94）都是整型常量或整型字面量。C 语言把不含小数点和指数的数作为整数。因此，22 和 -44 都是整型常量，但是 22.0 和 2.2E1 则不是。C 语言把大多数整型常量视为 `int` 类型，但是非常大的整数除外。详见后面“`long` 常量和 `long long` 常量”小节对 `long int` 类型的讨论。

## 4. 打印 `int` 值

可以使用 `printf()` 函数打印 `int` 类型的值。第 2 章中介绍过，`%d` 指明了在一行中打印整数的位置。`%d` 称为转换说明，它指定了 `printf()` 应使用什么格式来显示一个值。格式化字符串中的每个 `%d` 都与待打印变量列表中相应的 `int` 值匹配。这个值可以是 `int` 类型的变量、`int` 类型的常量或其他任何值为 `int` 类型的表达式。作为程序员，要确保转换说明的数量与待打印值的数量相同，编译器不会捕获这类型的错误。程序清单 3.2 演示了一个简单的程序，程序中初始化了一个变量，并打印该变量的值、一个常量值和一个简单表达式的值。另外，程序还演示了如果粗心犯错会导致什么结果。

### 程序清单 3.2 `print1.c` 程序

---

```
/* print1.c - 演示 printf() 的一些特性 */
#include <stdio.h>
int main(void)
{
    int ten = 10;
    int two = 2;

    printf("Doing it right: ");
    printf("%d minus %d is %d\n", ten, 2, ten - two);
    printf("Doing it wrong: ");
    printf("%d minus %d is %d\n", ten); // 遗漏 2 个参数
```

```
    return 0;
}
```

编译并运行该程序，输出如下：

```
Doing it right: 10 minus 2 is 8
Doing it wrong: 10 minus 16 is 1650287143
```

在第一行输出中，第1个%d对应int类型变量ten；第2个%d对应int类型常量2；第3个%d对应int类型表达式ten - two的值。在第二行输出中，第1个%d对应ten的值，但是由于没有给后两个%d提供任何值，所以打印出的值是内存中的任意值（读者在运行该程序时显示的这两个数值会与输出示例中的数值不同，因为内存中储存的数据不同，而且编译器管理内存的位置也不同）。

你可能会抱怨编译器为何不能捕获这种明显的错误，但实际上问题出在printf()不寻常的设计。大部分函数都需要指定数目的参数，编译器会检查参数的数目是否正确。但是，printf()函数的参数数目不定，可以有1个、2个、3个或更多，编译器也爱莫能助。记住，使用printf()函数时，要确保转换说明的数量与待打印值的数量相等。

## 5. 八进制和十六进制

通常，C语言都假定整型常量是十进制数。然而，许多程序员很喜欢使用八进制和十六进制数。因为8和16都是2的幂，而10却不是。显然，八进制和十六进制记数系统在表达与计算机相关的值时很方便。例如，十进制数65536经常出现在16位机中，用十六进制表示正好是10000。另外，十六进制数的每一位的数恰好由4位二进制数表示。例如，十六进制数3是0011，十六进制数5是0101。因此，十六进制数35的位组合(bit pattern)是00110101，十六进制数53的位组合是01010011。这种对应关系使得十六进制和二进制的转换非常方便。但是，计算机如何知道10000是十进制、十六进制还是二进制？在C语言中，用特定的前缀表示使用哪种进制。0x或0X前缀表示十六进制值，所以十进制数16表示成十六进制是0x10或0X10。与此类似，0前缀表示八进制。例如，十进制数16表示成八进制是020。第15章将更全面地介绍进制相关的内容。

要清楚，使用不同的进制数是为了方便，不会影响数被储存的方式。也就是说，无论把数字写成16、020或0x10，储存该数的方式都相同，因为计算机内部都以二进制进行编码。

## 6. 显示八进制和十六进制

在C程序中，既可以使用和显示不同进制的数。不同的进制要使用不同的转换说明。以十进制显示数字，使用%d；以八进制显示数字，使用%o；以十六进制显示数字，使用%x。另外，要显示各进制数的前缀0、0x和0X，必须分别使用%#o、%#x、%#X。程序清单3.3演示了一个小程序。回忆一下，在某些集成开发环境(IDE)下编写的代码中插入getchar();语句，程序在执行完毕后不会立即关闭执行窗口。

程序清单3.3 bases.c程序

---

```
/* bases.c--以十进制、八进制、十六进制打印十进制数100 */
#include <stdio.h>
int main(void)
{
    int x = 100;

    printf("dec = %d; octal = %o; hex = %x\n", x, x, x);
    printf("dec = %d; octal = %#o; hex = %#x\n", x, x, x);

    return 0;
}
```

---

编译并运行该程序，输出如下：

```
dec = 100; octal = 144; hex = 64
dec = 100; octal = 0144; hex = 0x64
```

该程序以 3 种不同记数系统显示同一个值。printf() 函数做了相应的转换。注意，如果要在八进制和十六进制值前显示 0 和 0x 前缀，要分别在转换说明中加入#。

### 3.4.2 其他整数类型

初学 C 语言时，int 类型应该能满足大多数程序的整数类型需求。尽管如此，还应了解一下整型的其他形式。当然，也可以略过本节跳至 3.4.3 节阅读 char 类型的相关内容，以后有需要时再阅读本节。

C 语言提供 3 个附属关键字修饰基本整数类型：short、long 和 unsigned。应记住以下几点。

- short int 类型（或者简写为 short）占用的存储空间可能比 int 类型少，常用于较小数值的场合以节省空间。与 int 类似，short 是有符号类型。
- long int 或 long 占用的存储空间可能比 int 多，适用于较大数值的场合。与 int 类似，long 是有符号类型。
- long long int 或 long long（C99 标准加入）占用的储存空间可能比 long 多，适用于更大数值的场合。该类型至少占 64 位。与 int 类似，long long 是有符号类型。
- unsigned int 或 unsigned 只用于非负值的场合。这种类型与有符号类型表示的范围不同。例如，16 位 unsigned int 允许的取值范围是 0~65535，而不是-32768~32767。用于表示正负号的位现在用于表示另一个二进制位，所以无符号整型可以表示更大的数。
- 在 C90 标准中，添加了 unsigned long int 或 unsigned long 和 unsigned int 或 unsigned short 类型。C99 标准又添加了 unsigned long long int 或 unsigned long long。
- 在任何有符号类型前面添加关键字 signed，可强调使用有符号类型的意图。例如，short、short int、signed short、signed short int 都表示同一种类型。

#### 1. 声明其他整数类型

其他整数类型的声明方式与 int 类型相同，下面列出了一些例子。不是所有的 C 编译器都能识别最后 3 条声明，最后一个例子所有的类型是 C99 标准新增的。

```
long int estine;
long johns;
short int erns;
short ribs;
unsigned int s_count;
unsigned players;
unsigned long headcount;
unsigned short yesvotes;
long long ago;
```

#### 2. 使用多种整数类型的原因

为什么说 short 类型“可能”比 int 类型占用的空间少，long 类型“可能”比 int 类型占用的空间多？因为 C 语言只规定了 short 占用的存储空间不能多于 int，long 占用的存储空间不能少于 int。这样规定是为了适应不同的机器。例如，过去的一台运行 Windows 3 的机器上，int 类型和 short 类型都占 16 位，long 类型占 32 位。后来，Windows 和苹果系统都使用 16 位储存 short 类型，32 位储存 int 类型和 long 类型（使用 32 位可以表示的整数数值超过 20 亿）。现在，计算机普遍使用 64 位处理器，为了

储存 64 位的整数，才引入了 long long 类型。

现在，个人计算机上最常见的设置是，long long 占 64 位，long 占 32 位，short 占 16 位，int 占 16 位或 32 位（依计算机的自然字长而定）。原则上，这 4 种类型代表 4 种不同的大小，但是在实际使用中，有些类型之间通常有重叠。

C 标准对基本数据类型只规定了允许的最小大小。对于 16 位机，short 和 int 的最小取值范围是 [-32767,32767]；对于 32 位机，long 的最小取值范围是 [-2147483647,2147483647]。对于 unsigned short 和 unsigned int，最小取值范围是 [0,65535]；对于 unsigned long，最小取值范围是 [0,4294967295]。long long 类型是为了支持 64 位的需求，最小取值范围是 [-9223372036854775807,9223372036854775807]；unsigned long long 的最小取值范围是 [0,18446744073709551615]。如果要开支票，这个数是一千八百亿亿（兆）六千七百四十四万亿零七百三十七亿零九百五十五万一千六百一十五。但是，谁会去数？

int 类型那么多，应该如何选择？首先，考虑 unsigned 类型。这种类型的数常用于计数，因为计数不用负数。而且，unsigned 类型可以表示更大的正数。

如果一个数超出了 int 类型的取值范围，且在 long 类型的取值范围内时，使用 long 类型。然而，对于那些 long 占用的空间比 int 大的系统，使用 long 类型会减慢运算速度。因此，如非必要，请不要使用 long 类型。另外要注意一点：如果在 long 类型和 int 类型占用空间相同的机器上编写代码，当确实需要 32 位的整数时，应使用 long 类型而不是 int 类型，以便把程序移植到 16 位机后仍然可以正常工作。类似地，如果确实需要 64 位的整数，应使用 long long 类型。

如果在 int 设置为 32 位的系统中要使用 16 位的值，应使用 short 类型以节省存储空间。通常，只有当程序使用相对于系统可用内存较大的整型数组时，才需要重点考虑节省空间的问题。使用 short 类型的另一个原因是，计算机中某些组件使用的硬件寄存器是 16 位。

### 3. long 常量和 long long 常量

通常，程序代码中使用的数字（如，2345）都被储存为 int 类型。如果使用 1000000 这样的大数字，超出了 int 类型能表示的范围，编译器会将其视为 long int 类型（假设这种类型可以表示该数字）。如果数字超出 long 可表示的最大值，编译器则将其视为 unsigned long 类型。如果还不够大，编译器则将其视为 long long 或 unsigned long long 类型（前提是编译器能识别这些类型）。

八进制和十六进制常量被视为 int 类型。如果值太大，编译器会尝试使用 unsigned int。如果还不够大，编译器会依次使用 long、unsigned long、long long 和 unsigned long long 类型。

有些情况下，需要编译器以 long 类型储存一个小数字。例如，编程时要显式使用 IBM PC 上的内存地址时。另外，一些 C 标准函数也要求使用 long 类型的值。要把一个较小的常量作为 long 类型对待，可以在值的末尾加上 l（小写的 L）或 L 后缀。使用 L 后缀更好，因为 l 看上去和数字 1 很像。因此，在 int 为 16 位、long 为 32 位的系统中，会把 7 作为 16 位储存，把 7L 作为 32 位储存。l 或 L 后缀也可用于八进制和十六进制整数，如 020L 和 0x10L。

类似地，在支持 long long 类型的系统中，也可以使用 ll 或 LL 后缀来表示 long long 类型的值，如 3LL。另外，u 或 U 后缀表示 unsigned long long，如 5ull、10LLU、6LLU 或 9ull。

### 整数溢出

如果整数超出了相应类型的取值范围会怎样？下面分别将有符号类型和无符号类型的整数设置为比最大值略大，看看会发生什么（printf() 函数使用 %u 说明显示 unsigned int 类型的值）。

```
/* toobig.c-- 超出系统允许的最大 int 值*/
```

```
#include <stdio.h>
int main(void)
{
    int i = 2147483647;
    unsigned int j = 4294967295;

    printf("%d %d %d\n", i, i+1, i+2);
    printf("%u %u %u\n", j, j+1, j+2);

    return 0;
}
```

在我们的系统下输出的结果是：

```
2147483647 -2147483648 -2147483647
4294967295 0 1
```

可以把无符号整数 *j* 看作是汽车的里程表。当达到它能表示的最大值时，会重新从起始点开始。整数 *i* 也是类似的情况。它们主要的区别是，在超过最大值时，*unsigned int* 类型的变量 *j* 从 0 开始；而 *int* 类型的变量 *i* 则从 -2147483648 开始。注意，当 *i* 超出（溢出）其相应类型所能表示的最大值时，系统并未通知用户。因此，在编程时必须自己注意这类问题。

溢出行行为是未定义的行为，C 标准并未定义有符号类型的溢出规则。以上描述的溢出行为比较有代表性，但是也可能会出现其他情况。

#### 4. 打印 short、long、long long 和 unsigned 类型

打印 *unsigned int* 类型的值，使用 *%u* 转换说明；打印 *long* 类型的值，使用 *%ld* 转换说明。如果系统中 *int* 和 *long* 的大小相同，使用 *%d* 就行。但是，这样的程序被移植到其他系统（*int* 和 *long* 类型的大小不同）中会无法正常工作。在 *x* 和 *o* 前面可以使用 *l* 前缀，*%lx* 表示以十六进制格式打印 *long* 类型整数，*%lo* 表示以八进制格式打印 *long* 类型整数。注意，虽然 C 允许使用大写或小写的常量后缀，但是在转换说明中只能用小写。

C 语言有多种 *printf()* 格式。对于 *short* 类型，可以使用 *h* 前缀。*%hd* 表示以十进制显示 *short* 类型的整数，*%ho* 表示以八进制显示 *short* 类型的整数。*h* 和 *l* 前缀都可以和 *u* 一起使用，用于表示无符号类型。例如，*%lu* 表示打印 *unsigned long* 类型的值。程序清单 3.4 演示了一些例子。对于支持 *long long* 类型的系统，*%lld* 和 *%llu* 分别表示有符号和无符号类型。第 4 章将详细介绍转换说明。

程序清单 3.4 print2.c 程序

```
/* print2.c--更多 printf() 的特性 */
#include <stdio.h>
int main(void)
{
    unsigned int un = 3000000000; /* int 为 32 位和 short 为 16 位的系统 */
    short end = 200;
    long big = 65537;
    long long verybig = 12345678908642;

    printf("un = %u and not %d\n", un, un);
    printf("end = %hd and %d\n", end, end);
    printf("big = %ld and not %hd\n", big, big);
    printf("verybig= %lld and not %ld\n", verybig, verybig);

    return 0;
}
```

在特定的系统中输出如下(输出的结果可能不同):

```
un = 3000000000 and not -1294967296
end = 200 and 200
big = 65537 and not 1
verybig= 12345678908642 and not 1942899938
```

该例表明, 使用错误的转换说明会得到意想不到的结果。第1行输出, 对于无符号变量 un, 使用%d会生成负值! 其原因是, 无符号值 3000000000 和有符号值-129496296 在系统内存中的内部表示完全相同(详见第15章)。因此, 如果告诉 printf() 该数是无符号数, 它打印一个值; 如果告诉它该数是有符号数, 它将打印另一个值。在待打印的值大于有符号值的最大值时, 会发生这种情况。对于较小的正数(如96), 有符号和无符号类型的存储、显示都相同。

第2行输出, 对于 short 类型的变量 end, 在 printf() 中无论指定以 short 类型(%hd)还是 int 类型(%d)打印, 打印出来的值都相同。这是因为在给函数传递参数时, C 编译器把 short 类型的值自动转换成 int 类型的值。你可能会提出疑问: 为什么要进行转换? h 修饰符有什么用? 第1个问题的答案是, int 类型被认为是计算机处理整数类型时最高效的类型。因此, 在 short 和 int 类型的大小不同的计算机中, 用 int 类型的参数传递速度更快。第2个问题的答案是, 使用 h 修饰符可以显示较大整数被截断成 short 类型值的情况。第3行输出就演示了这种情况。把 65537 以二进制格式写成一个 32 位数是 0000000000000000100000000000000001。使用%hd, printf() 只会查看后 16 位, 所以显示的值是 1。与此类似, 输出的最后一行先显示了 verybig 的完整值, 然后由于使用了%ld, printf() 只显示了储存在后 32 位的值。

本章前面介绍过, 程序员必须确保转换说明的数量和待打印值的数量相同。以上内容也提醒读者, 程序员还必须根据待打印值的类型使用正确的转换说明。

### 提示 匹配 printf() 说明符的类型

在使用 printf() 函数时, 切记检查每个待打印值都有对应的转换说明, 还要检查转换说明的类型是否与待打印值的类型相匹配。

### 3.4.3 使用字符: char 类型

char 类型用于储存字符(如, 字母或标点符号), 但是从技术层面看, char 是整数类型。因为 char 类型实际上储存的是整数而不是字符。计算机使用数字编码来处理字符, 即用特定的整数表示特定的字符。美国最常用的编码是 ASCII 编码, 本书也使用此编码。例如, 在 ASCII 码中, 整数 65 代表大写字母 A。因此, 储存字母 A 实际上储存的是整数 65(许多 IBM 的大型主机使用另一种编码——EBCDIC, 其原理相同。另外, 其他国家的计算机系统可能使用完全不同的编码)。

标准 ASCII 码的范围是 0~127, 只需 7 位二进制数即可表示。通常, char 类型被定义为 8 位的存储单元, 因此容纳标准 ASCII 码绰绰有余。许多其他系统(如 IMB PC 和苹果 Macs)还提供扩展 ASCII 码, 也在 8 位的表示范围之内。一般而言, C 语言会保证 char 类型足够大, 以储存系统(实现 C 语言的系统)的基本字符集。

许多字符集都超过了 127, 甚至多于 255。例如, 日本汉字(kanji)字符集。商用的统一码(Unicode)创建了一个能表示世界范围内多种字符集的系统, 目前包含的字符已超过 110000 个。国际标准化组织(ISO)和国际电工技术委员会(IEC)为字符集开发了 ISO/IEC 10646 标准。统一码标准也与 ISO/IEC 10646 标准兼容。

C 语言把 1 字节定义为 char 类型占用的位 (bit) 数, 因此无论是 16 位还是 32 位系统, 都可以使用 char 类型。

### 1. 声明 char 类型变量

char 类型变量的声明方式与其他类型变量的声明方式相同。下面是一些例子:

```
char response;
char itable, latan;
```

以上声明创建了 3 个 char 类型的变量: response、itable 和 latan。

### 2. 字符常量和初始化

如果要把一个字符常量初始化为字母 A, 不必背下 ASCII 码, 用计算机语言很容易做到。通过以下初始化把字母 A 赋给 grade 即可:

```
char grade = 'A';
```

在 C 语言中, 用单引号括起来的单个字符被称为字符常量 (*character constant*)。编译器一发现'A', 就会将其转换成相应的代码值。单引号必不可少。下面还有一些其他的例子:

```
char broiled;      /* 声明一个 char 类型的变量 */
broiled = 'T';     /* 为其赋值, 正确 */
broiled = T;       /* 错误! 此时 T 是一个变量 */
broiled = "T";     /* 错误! 此时 "T" 是一个字符串 */
```

如上所示, 如果省略单引号, 编译器认为 T 是一个变量名; 如果把 T 用双引号括起来, 编译器则认为 "T" 是一个字符串。字符串的内容将在第 4 章中介绍。

实际上, 字符是以数值形式储存的, 所以也可使用数字代码值来赋值:

```
char grade = 65; /* 对于 ASCII, 这样做没问题, 但这是一种不好的编程风格 */
```

在本例中, 虽然 65 是 int 类型, 但是它在 char 类型能表示的范围内, 所以将其赋值给 grade 没问题。由于 65 是字母 A 对应的 ASCII 码, 因此本例是把 A 赋给 grade。注意, 能这样做的前提是系统使用 ASCII 码。其实, 用'A'代替 65 才是较为妥当的做法, 这样在任何系统中都不会出问题。因此, 最好使用字符常量, 而不是数字代码值。

奇怪的是, C 语言将字符常量视为 int 类型而非 char 类型。例如, 在 int 为 32 位、char 为 8 位的 ASCII 系统中, 有下面的代码:

```
char grade = 'B';
```

本来'B'对应的数值 66 储存在 32 位的存储单元中, 现在却可以储存在 8 位的存储单元中 (grade)。利用字符常量的这种特性, 可以定义一个字符常量'FATE', 即把 4 个独立的 8 位 ASCII 码储存在一个 32 位存储单元中。如果把这样的字符常量赋给 char 类型变量 grade, 只有最后 8 位有效。因此, grade 的值是'E'。

### 3. 非打印字符

单引号只适用于字符、数字和标点符号, 浏览 ASCII 表会发现, 有些 ASCII 字符打印不出来。例如, 一些代表行为的字符 (如, 退格、换行、终端响铃或蜂鸣)。C 语言提供了 3 种方法表示这些字符。

第 1 种方法前面介绍过——使用 ASCII 码。例如, 蜂鸣字符的 ASCII 值是 7, 因此可以这样写:

```
char beep = 7;
```

第 2 种方法是, 使用特殊的符号序列表示一些特殊的字符。这些符号序列叫作转义序列 (*escape sequence*)。表 3.2 列出了转义序列及其含义。

把转义序列赋给字符变量时，必须用单引号把转义序列括起来。例如，假设有下面一行代码：

```
char nerf = '\n';
```

稍后打印变量 nerf 的效果是，在打印机或屏幕上另起一行。

表 3.2 转义序列

| 转义序列 | 含义                                         |
|------|--------------------------------------------|
| \a   | 警报 (ANSI C)                                |
| \b   | 退格                                         |
| \f   | 换页                                         |
| \n   | 换行                                         |
| \r   | 回车                                         |
| \t   | 水平制表符                                      |
| \v   | 垂直制表符                                      |
| \\\  | 反斜杠 (\)                                    |
| \'   | 单引号                                        |
| \"   | 双引号                                        |
| \?   | 问号                                         |
| \ooo | 八进制值 (oo 必须是有效的八进制数，即每个 o 可表示 0~7 中的一个数)   |
| \xhh | 十六进制值 (hh 必须是有效的十六进制数，即每个 h 可表示 0~f 中的一个数) |

现在，我们来仔细分析一下转义序列。使用 C90 新增的警报字符 (\a) 是否能产生听到或看到的警报，取决于计算机的硬件，蜂鸣是最常见的警报（在一些系统中，警报字符不起作用）。C 标准规定警报字符不得改变活跃位置。标准中的活跃位置 (*active position*) 指的是显示设备（屏幕、电传打字机、打印机等）中下一个字符将出现的位置。简而言之，平时常说的屏幕光标位置就是活跃位置。在程序中把警报字符输出在屏幕上的效果是，发出一声蜂鸣，但不会移动屏幕光标。

接下来的转义字符 \b、\f、\n、\r、\t 和 \v 是常用的输出设备控制字符。了解它们最好的方式是查看它们对活跃位置的影响。换页符 (\f) 把活跃位置移至下一页的开始处；换行符 (\n) 把活跃位置移至下一行的开始处；回车符 (\r) 把活跃位置移动到当前行的开始处；水平制表符 (\t) 将活跃位置移至下一个水平制表点（通常是第 1 个、第 9 个、第 17 个、第 25 个等字符位置）；垂直制表符 (\v) 把活跃位置移至下一个垂直制表点。

这些转义序列字符不一定在所有的显示设备上都起作用。例如，换页符和垂直制表符在 PC 屏幕上会生成奇怪的符号，光标并不会移动。只有将其输出到打印机上时才会产生前面描述的效果。

接下来的 3 个转义序列 (\\\、\'、\") 用于打印 \、'、" 字符（由于这些字符用于定义字符常量，是 printf() 函数的一部分，若直接使用它们会造成混乱）。如果打印下面一行内容：

```
Gramps sez, "a \ is a backslash."
```

应这样编写代码：

```
printf("Gramps sez, \"a \\\ is a backslash.\\"\\n");
```

表 3.2 中的最后两个转义序列 (\ooo 和 \xhh) 是 ASCII 码的特殊表示。如果要用八进制 ASCII 码表示一个字符，可以在编码值前面加一个反斜杠 (\) 并用单引号括起来。例如，如果编译器不识别警报字符 (\a)，可以使用 ASCII 码来代替：

```
beep = '\007';
```

可以省略前面的 0，'\07'甚至'\7'都可以。即使没有前缀 0，编译器在处理这种写法时，仍会解释为八进制。

从 C90 开始，不仅可以用十进制、八进制形式表示字符常量，C 语言还提供了第 3 种选择——用十六进制形式表示字符常量，即反斜杠后面跟一个 x 或 X，再加上 1~3 位十六进制数字。例如，**Ctrl+P** 字符的 ASCII 十六进制码是 10（相当于十进制的 16），可表示为'\x10'或'\x010'。图 3.5 列出了一些整数类型的不同进制形式。

| 整型常量的例子            |         |         |       |
|--------------------|---------|---------|-------|
| 类型                 | 十六进制    | 八进制     | 十进制   |
| char               | \0x41   | \0101   | N.A.  |
| int                | 0x41    | 0101    | 65    |
| unsigned int       | 0x41u   | 0101u   | 65u   |
| long               | 0x41L   | 0101L   | 65L   |
| unsigned long      | 0x41UL  | 0101UL  | 65UL  |
| long long          | 0x41LL  | 0101LL  | 65LL  |
| unsigned long long | 0x41ULL | 0101ULL | 65ULL |

图 3.5 int 系列类型的常量写法示例

使用 ASCII 码时，注意数字和数字字符的区别。例如，字符 4 对应的 ASCII 码是 52。'4'表示字符 4，而不是数值 4。

关于转义序列，读者可能有下面 3 个问题。

- 上面最后一个例子 (`printf("Gramps sez, \"a \\ is a backslash\\\"\\n")`)，为何没有用单引号把转义序列括起来？无论是普通字符还是转义序列，只要是双引号括起来的字符集合，就无需用单引号括起来。双引号中的字符集合叫作字符串（详见第 4 章）。注意，该例中的其他字符（G、r、a、m、p、s 等）都没有用单引号括起来。与此类似，`printf("Hello!\007\\n")`；将打印 Hello！并发出一声蜂鸣，而 `printf("Hello!7\\n")`；则打印 Hello!7。不是转义序列中的数字将作为普通字符被打印出来。
- 何时使用 ASCII 码？何时使用转义序列？如果要在转义序列（假设使用'\f'）和 ASCII 码（'\014'）之间选择，请选择前者（即'\f'）。这样的写法不仅更好记，而且可移植性更高。'\f'在不使用 ASCII 码的系统中，仍然有效。
- 如果要使用 ASCII 码，为何要写成'\032'而不是 032？首先，'\032'能更清晰地表达程序员使用字符编码的意图。其次，类似\032 这样的转义序列可以嵌入 C 的字符串中，如 `printf("Hello!\007\\n")`；中就嵌入了\007。

#### 4. 打印字符

`printf()` 函数用%c 指明待打印的字符。前面介绍过，一个字符变量实际上被储存为 1 字节的整数值。因此，如果用%d 转换说明打印 `char` 类型变量的值，打印的是一个整数。而%c 转换说明告诉 `printf()` 打印该整数值对应的字符。程序清单 3.5 演示了打印 `char` 类型变量的两种方式。

## 程序清单 3.5 charcode.c 程序

```
/* charcode.c 显示字符的代码编号 */
#include <stdio.h>
int main(void)
{
    char ch;

    printf("Please enter a character.\n");
    scanf("%c", &ch); /* 用户输入字符 */
    printf("The code for %c is %d.\n", ch, ch);

    return 0;
}
```

运行该程序后，输出示例如下：

```
Please enter a character.
C
The code for C is 67.
```

运行该程序时，在输入字母后不要忘记按下 **Enter** 或 **Return** 键。随后，`scanf()` 函数会读取用户输入的字符，`&` 符号表示把输入的字符赋给变量 `ch`。接着，`printf()` 函数打印 `ch` 的值两次，第 1 次打印一个字符（对应代码中的`%c`），第 2 次打印一个十进制整数值（对应代码中的`%d`）。注意，`printf()` 函数中的转换说明决定了数据的显示方式，而不是数据的储存方式（见图 3.6）。

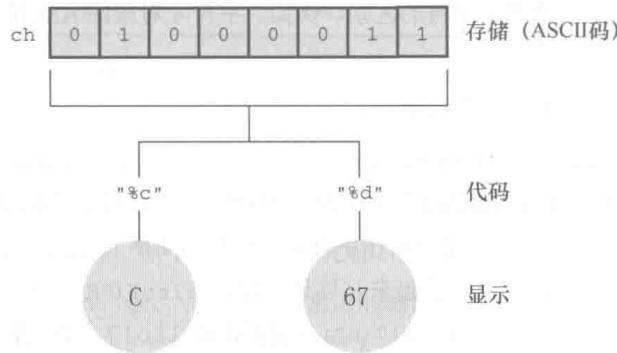


图 3.6 数据显示和数据存储

## 5. 有符号还是无符号

有些 C 编译器把 `char` 实现为有符号类型，这意味着 `char` 可表示的范围是 -128~127。而有些 C 编译器把 `char` 实现为无符号类型，那么 `char` 可表示的范围是 0~255。请查阅相应的编译器手册，确定正在使用的编译器如何实现 `char` 类型。或者，可以查阅 `limits.h` 头文件。下一章将详细介绍头文件的内容。

根据 C90 标准，C 语言允许在关键字 `char` 前面使用 `signed` 或 `unsigned`。这样，无论编译器默认 `char` 是什么类型，`signed char` 表示有符号类型，而 `unsigned char` 表示无符号类型。这在用 `char` 类型处理小整数时很有用。如果只用 `char` 处理字符，那么 `char` 前面无需使用任何修饰符。

### 3.4.4 \_Bool 类型

C99 标准添加了 `_Bool` 类型，用于表示布尔值，即逻辑值 `true` 和 `false`。因为 C 语言用值 1 表示 `true`，值 0 表示 `false`，所以 `_Bool` 类型实际上也是一种整数类型。但原则上它仅占用 1 位存储空间，因为对 0 和 1 而言，1 位的存储空间足够了。

程序通过布尔值可选择执行哪部分代码。我们将在第 6 章和第 7 章中详述相关内容。

### 3.4.5 可移植类型: stdint.h 和 inttypes.h

C 语言提供了许多有用的整数类型。但是，某些类型名在不同系统中的功能不一样。C99 新增了两个头文件 stdint.h 和 inttypes.h，以确保 C 语言的类型在各系统中的功能相同。

C 语言为现有类型创建了更多类型名。这些新的类型名定义在 stdint.h 头文件中。例如，int32\_t 表示 32 位的有符号整数类型。在使用 32 位 int 的系统中，头文件会把 int32\_t 作为 int 的别名。不同的系统也可以定义相同的类型名。例如，int 为 16 位、long 为 32 位的系统会把 int32\_t 作为 long 的别名。然后，使用 int32\_t 类型编写程序，并包含 stdint.h 头文件时，编译器会把 int 或 long 替换成与当前系统匹配的类型。

上面讨论的类型别名是精确宽度整数类型 (*exact-width integer type*) 的示例。int32\_t 表示整数类型的宽度正好是 32 位。但是，计算机的底层系统可能不支持。因此，精确宽度整数类型是可选项。

如果系统不支持精确宽度整数类型怎么办？C99 和 C11 提供了第 2 类别名集合。一些类型名保证所表示的类型一定是至少有指定宽度的最小整数类型。这组类型集合被称为最小宽度类型 (*minimum width type*)。例如，int\_least8\_t 是可容纳 8 位有符号整数值的类型中宽度最小的类型的一个别名。如果某系统的最小整数类型是 16 位，可能不会定义 int8\_t 类型。尽管如此，该系统仍可使用 int\_least8\_t 类型，但可能把该类型实现为 16 位的整数类型。

当然，一些程序员更关心速度而非空间。为此，C99 和 C11 定义了一组可使计算达到最快的类型集合。这组类型集合被称为最快最小宽度类型 (*fastest minimum width type*)。例如，int\_fast8\_t 被定义为系统中对 8 位有符号值而言运算最快的整数类型的别名。

另外，有些程序员需要系统的最大整数类型。为此，C99 定义了最大的有符号整数类型 intmax\_t，可储存任何有效的有符号整数值。类似地，uintmax\_t 表示最大的无符号整数类型。顺带一提，这些类型有可能比 long long 和 unsigned long 类型更大，因为 C 编译器除了实现标准规定的类型以外，还可利用 C 语言实现其他类型。例如，一些编译器在标准引入 long long 类型之前，已提前实现了该类型。

C99 和 C11 不仅提供可移植的类型名，还提供相应的输入和输出。例如，printf() 打印特定类型时要求与相应的转换说明匹配。如果要打印 int32\_t 类型的值，有些定义使用%d，而有些定义使用%ld，怎么办？C 标准针对这一情况，提供了一些字符串宏（第 4 章中详细介绍）来显示可移植类型。例如，inttypes.h 头文件中定义了 PRIId32 字符串宏，代表打印 32 位有符号值的合适转换说明（如 d 或 l）。程序清单 3.6 演示了一种可移植类型和相应转换说明的用法。

程序清单 3.6 altnames.c 程序

---

```
/* altnames.c -- 可移植整数类型名 */
#include <stdio.h>
#include <inttypes.h> // 支持可移植类型
int main(void)
{
    int32_t me32;      // me32 是一个 32 位有符号整型变量
    me32 = 45933945;
    printf("First, assume int32_t is int: ");
    printf("me32 = %d\n", me32);
    printf("Next, let's not make any assumptions.\n");
```

```

printf("Instead, use a \"macro\" from inttypes.h: ");
printf("me32 = %" PRId32 "\n", me32);

return 0;
}

```

该程序最后一个 `printf()` 中，参数 `PRId32` 被定义在 `inttypes.h` 中的“`d`”替换，因而这条语句等价于：

```
printf("me16 = %" "d" "\n", me16);
```

在 C 语言中，可以把多个连续的字符串组合成一个字符串，所以这条语句又等价于：

```
printf("me16 = %d\n", me16);
```

下面是该程序的输出，注意，程序中使用了“`\`转义序列来显示双引号：

```

First, assume int32_t is int: me32 = 45933945
Next, let's not make any assumptions.
Instead, use a "macro" from inttypes.h: me32 = 45933945

```

篇幅有限，无法介绍扩展的所有整数类型。本节主要是为了让读者知道，在需要时可进行这种级别的类型控制。附录 B 中的参考资料 VI “扩展的整数类型”介绍了完整的 `inttypes.h` 和 `stdint.h` 头文件。

### 注意 对 C99/C11 的支持

C 语言发展至今，虽然 ISO 已发布了 C11 标准，但是编译器供应商对 C99 的实现程度却各不相同。

在本书第 6 版的编写过程中，一些编译器仍未实现 `inttypes.h` 头文件及其相关功能。

## 3.4.6 float、double 和 long double

各种整数类型对大多数软件开发项目而言够用了。然而，面向金融和数学的程序经常使用浮点数。C 语言中的浮点类型有 `float`、`double` 和 `long double` 类型。它们与 FORTRAN 和 Pascal 中的 `real` 类型一致。前面提到过，浮点类型能表示包括小数在内的更大范围的数。浮点数的表示类似于科学记数法（即用小数乘以 10 的幂来表示数字）。该记数系统常用于表示非常大或非常小的数。表 3.3 列出了一些示例。

表 3.3 记数法示例

| 数字         | 科学记数法                | 指数记数法    |
|------------|----------------------|----------|
| 1000000000 | $1.0 \times 10^9$    | 1.0e9    |
| 123000     | $1.23 \times 10^5$   | 1.23e5   |
| 322.56     | $3.2256 \times 10^2$ | 3.2256e2 |
| 0.000056   | $5.6 \times 10^{-5}$ | 5.6e-5   |

第 1 列是一般记数法；第 2 列是科学记数法；第 3 列是指数记数法（或称为 e 记数法），这是科学记数法在计算机中的写法，e 后面的数字代表 10 的指数。图 3.7 演示了更多的浮点数写法。

C 标准规定，`float` 类型必须至少能表示 6 位有效数字，且取值范围至少是  $10^{-37} \sim 10^{+37}$ 。前一项规定指 `float` 类型必须至少精确表示小数点后的 6 位有效数字，如 33.333333。后一项规定用于方便地表示诸如太阳质量（ $2.0\text{e}30$  千克）、一个质子的电荷量（ $1.6\text{-}19$  库仑）或国家债务之类的数字。通常，系统储存一个浮点数要占用 32 位。其中 8 位用于表示指数的值和符号，剩下 24 位用于表示非指数部分（也叫作尾数或有效数）及其符号。

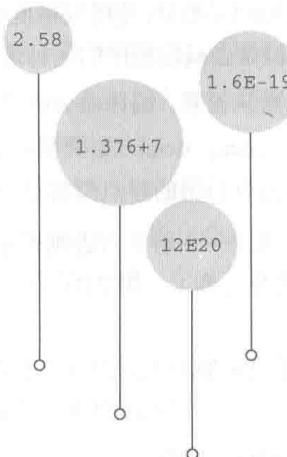


图 3.7 更多浮点数写法示例

C 语言提供的另一种浮点类型是 `double` (意为双精度)。`double` 类型和 `float` 类型的最小取值范围相同，但至少必须能表示 10 位有效数字。一般情况下，`double` 占用 64 位而不是 32 位。一些系统将多出的 32 位全部用来表示非指数部分，这不仅增加了有效数字的位数（即提高了精度），而且还减少了舍入误差。另一些系统把其中的一些位分配给指数部分，以容纳更大的指数，从而增加了可表示数的范围。无论哪种方法，`double` 类型的值至少有 13 位有效数字，超过了标准的最低位数规定。

C 语言的第 3 种浮点类型是 `long double`，以满足比 `double` 类型更高的精度要求。不过，C 只保证 `long double` 类型至少与 `double` 类型的精度相同。

## 1. 声明浮点型变量

浮点型变量的声明和初始化方式与整型变量相同，下面是一些例子：

```
float noah, jonah;
double trouble;
float planck = 6.63e-34;
long double gnp;
```

## 2. 浮点型常量

在代码中，可以用多种形式书写浮点型常量。浮点型常量的基本形式是：有符号的数字（包括小数点），后面紧跟 e 或 E，最后是一个有符号数表示 10 的指数。下面是两个有效的浮点型常量：

```
-1.56E+12
2.87e-3
```

正号可以省略。可以没有小数点（如，`2E5`）或指数部分（如，`19.28`），但是不能同时省略两者。可以省略小数部分（如，`3.E16`）或整数部分（如，`.45E-6`），但是不能同时省略两者。下面是更多的有效浮点型常量示例：

```
3.14159
.2
4e16
.8E-5
100.
```

不要在浮点型常量中间加空格：`1.56 E+12`（错误！）

默认情况下，编译器假定浮点型常量是 `double` 类型的精度。例如，假设 `some` 是 `float` 类型的变量，编写下面的语句：

```
some = 4.0 * 2.0;
```

通常,4.0 和 2.0 被储存为 64 位的 double 类型,使用双精度进行乘法运算,然后将乘积截断成 float 类型的宽度。这样做虽然计算精度更高,但是会减慢程序的运行速度。

在浮点数后面加上 f 或 F 后缀可覆盖默认设置,编译器会将浮点型常量看作 float 类型,如 2.3f 和 9.11E9F。使用 l 或 L 后缀使得数字成为 long double 类型,如 54.31 和 4.32L。注意,建议使用 L 后缀,因为字母 l 和数字 1 很容易混淆。没有后缀的浮点型常量是 double 类型。

C99 标准添加了一种新的浮点型常量格式——用十六进制表示浮点型常量,即在十六进制数前加上十六进制前缀(0x 或 0X),用 p 和 P 分别代替 e 和 E,用 2 的幂代替 10 的幂(即,p 计数法)。如下所示:

0xa.1fp10

十六进制 a 等于十进制 10,.1f 是 1/16 加上 15/256(十六进制 f 等于十进制 15),p10 是 2<sup>10</sup> 或 1024。0xa.1fp10 表示的值是(10 + 1/16 + 15/256) × 1024(即,十进制 10364.0)。

注意,并非所有的编译器都支持 C99 的这一特性。

### 3. 打印浮点值

printf() 函数使用 %f 转换说明打印十进制记数法的 float 和 double 类型浮点数,用 %e 打印指数记数法的浮点数。如果系统支持十六进制格式的浮点数,可用 a 和 A 分别代替 e 和 E。打印 long double 类型要使用 %Lf、%Le 或 %La 转换说明。给那些未在函数原型中显式说明参数类型的函数(如,printf())传递参数时,C 编译器会把 float 类型的值自动转换成 double 类型。程序清单 3.7 演示了这些特性。

程序清单 3.7 showf\_pt.c 程序

```
/* showf_pt.c -- 以两种方式显示 float 类型的值 */
#include <stdio.h>
int main(void)
{
    float aboat = 32000.0;
    double abet = 2.14e9;
    long double dip = 5.32e-5;

    printf("%f can be written %e\n", aboat, aboat);
    // 下一行要求编译器支持 C99 或其中的相关特性
    printf("And it's %a in hexadecimal, powers of 2 notation\n", aboat);
    printf("%f can be written %e\n", abet, abet);
    printf("%Lf can be written %Le\n", dip, dip);

    return 0;
}
```

该程序的输出如下,前提是编译器支持 C99/C11:

```
32000.000000 can be written 3.200000e+04
And it's 0x1.f4p+14 in hexadecimal, powers of 2 notation
2140000000.000000 can be written 2.140000e+09
0.000053 can be written 5.320000e-05
```

该程序示例演示了默认的输出效果。下一章将介绍如何通过设置字段宽度和小数位数来控制输出格式。

### 4. 浮点值的上溢和下溢

假设系统的最大 float 类型值是 3.4E38,编写如下代码:

```
float toobig = 3.4E38 * 100.0f;
printf("%e\n", toobig);
```

会发生什么？这是一个上溢 (*overflow*) 的示例。当计算导致数字过大，超过当前类型能表达的范围时，就会发生上溢。这种行为在过去是未定义的，不过现在 C 语言规定，在这种情况下会给 `toobig` 赋一个表示无穷大的特定值，而且 `printf()` 显示该值为 `inf` 或 `infinity`（或者具有无穷含义的其他内容）。

当除以一个很小的数时，情况更为复杂。回忆一下，`float` 类型的数以指数和尾数部分来储存。存在这样一个数，它的指数部分是最小值，即由全部可用位表示的最小尾数值。该数字是 `float` 类型能用全部精度表示的最小数字。现在把它除以 2。通常，这个操作会减小指数部分，但是假设的情况下，指数已经是最小值了。所以计算机只好把尾数部分的位向右移，空出第 1 个二进制位，并丢弃最后一个二进制数。以十进制为例，把一个有 4 位有效数字的数（如，`0.1234E-10`）除以 10，得到的结果是 `0.0123E-10`。虽然得到了结果，但是在计算过程中却损失了原末尾有效位上的数字。这种情况叫作下溢 (*underflow*)。C 语言把损失了类型全精度的浮点值称为低于正常的 (*subnormal*) 浮点值。因此，把最小的正浮点数除以 2 将得到一个低于正常的值。如果除以一个非常大的值，会导致所有的位都为 0。现在，C 库已提供了用于检查计算是否会产生低于正常值的函数。

还有另一个特殊的浮点值 `Nan`（*not a number* 的缩写）。例如，给 `asin()` 函数传递一个值，该函数将返回一个角度，该角度的正弦就是传入函数的值。但是正弦值不能大于 1，因此，如果传入的参数大于 1，该函数的行为是未定义的。在这种情况下，该函数将返回 `Nan` 值，`printf()` 函数可将其显示为 `nan`、`NaN` 或其他类似的内容。

## 浮点数舍入错误

给定一个数，加上 1，再减去原来给定的数，结果是多少？你一定认为是 1。但是，下面的浮点运算给出了不同的答案：

```
/* floaterr.c--演示舍入错误 */
#include <stdio.h>
int main(void)
{
    float a,b;

    b = 2.0e20 + 1.0;
    a = b - 2.0e20;
    printf("%f \n", a);

    return 0;
}
```

该程序的输出如下：

```
0.000000 ←Linux 系统下的老式 gcc
-13584010575872.000000 ←Turbo C 1.5
4008175468544.000000 ←XCode 4.5、Visual Studio 2012、当前版本的 gcc
```

得出这些奇怪答案的原因是，计算机缺少足够的小数位来完成正确的运算。`2.0e20` 是 2 后面有 20 个 0。如果把该数加 1，那么发生变化的是第 21 位。要正确运算，程序至少要储存 21 位数字。而 `float` 类型的数字通常只能储存按指数比例缩小或放大的 6 或 7 位有效数字。在这种情况下，计算结果一定是错误的。另一方面，如果把 `2.0e20` 改成 `2.0e4`，计算结果就没问题。因为 `2.0e4` 加 1 只需改变第 5 位上的数字，`float` 类型的精度足够进行这样的计算。

## 浮点数表示法

上一个方框中列出了由于计算机使用的系统不同，一个程序有不同的输出。原因是，根据前面介

绍的知识，实现浮点数表示法的方法有多种。为了尽可能地统一实现，电子和电气工程师协会（IEEE）为浮点数计算和表示法开发了一套标准。现在，许多硬件浮点单元都采用该标准。2011年，该标准被ISO/IEC/IEEE 60559:2011标准收录。该标准作为C99和C11的可选项，符合硬件要求的平台可开启。`floaterr.c`程序的第3个输出示例即是支持该浮点标准的系统显示的结果。支持C标准的编译器还包含捕获异常问题的工具。详见附录B.5，参考资料V。

### 3.4.7 复数和虚数类型

许多科学和工程计算都要用到复数和虚数。C99标准支持复数类型和虚数类型，但是有所保留。一些独立实现，如嵌入式处理器的实现，就不需要使用复数和虚数（VCR芯片就不需要复数）。一般而言，虚数类型都是可选项。C11标准把整个复数软件包都作为可选项。

简而言之，C语言有3种复数类型：`float_Complex`、`double_Complex`和`long double_Complex`。例如，`float_Complex`类型的变量应包含两个`float`类型的值，分别表示复数的实部和虚部。类似地，C语言的3种虚数类型是`float_Imaginary`、`double_Imaginary`和`long double_Imaginary`。

如果包含`complex.h`头文件，便可用`complex`代替`_Complex`，用`imaginary`代替`_Imaginary`，还可以用`I`代替`-1`的平方根。

为何C标准不直接用`complex`作为关键字来代替`_Complex`，而要添加一个头文件（该头文件中把`complex`定义为`_Complex`）？因为标准委员会考虑到，如果使用新的关键字，会导致以该关键字作为标识符的现有代码全部失效。例如，之前的C99，许多程序员已经使用`struct complex`定义一个结构来表示复数或者心理学程序中的心理状况（关键字`struct`用于定义能储存多个值的结构，详见第14章）。让`complex`成为关键字会导致之前的这些代码出现语法错误。但是，使用`struct_Complex`的人很少，特别是标准使用首字母是下划线的标识符作为预留字以后。因此，标准委员会选定`_Complex`作为关键字，在不用考虑名称冲突的情况下可选择使用`complex`。

### 3.4.8 其他类型

现在已经介绍完C语言的所有基本数据类型。有些人认为这些类型实在太多了，但有些人觉得还不够用。注意，虽然C语言没有字符串类型，但也能很好地处理字符串。第4章将详细介绍相关内容。

C语言还有一些从基本类型衍生的其他类型，包括数组、指针、结构和联合。尽管后面章节中会详细介绍这些类型，但是本章的程序示例中已经用到了指针（指针（pointer）指向变量或其他数据对象位置）。例如，在`scanf()`函数中用到的前缀`&`，便创建了一个指针，告诉`scanf()`把数据放在何处。

#### 小结：基本数据类型

##### 关键字：

基本数据类型由11个关键字组成：`int`、`long`、`short`、`unsigned`、`char`、`float`、`double`、`signed`、`_Bool`、`_Complex`和`_Imaginary`。

##### 有符号整型：

有符号整型可用于表示正整数和负整数。

- `int`——系统给定的基本整数类型。C语言规定`int`类型不小于16位。

- short 或 short int ——最大的 short 类型整数小于或等于最大的 int 类型整数。C 语言规定 short 类型至少占 16 位。
- long 或 long int ——该类型可表示的整数大于或等于最大的 int 类型整数。C 语言规定 long 类型至少占 32 位。
- long long 或 long long int ——该类型可表示的整数大于或等于最大的 long 类型整数。Long long 类型至少占 64 位。

一般而言，long 类型占用的内存比 short 类型大，int 类型的宽度要么和 long 类型相同，要么和 short 类型相同。例如，旧 DOS 系统的 PC 提供 16 位的 short 和 int，以及 32 位的 long；Windows 95 系统提供 16 位的 short 以及 32 位的 int 和 long。

#### 无符号整型：

无符号整型只能用于表示零和正整数，因此无符号整型可表示的正整数比有符号整型的大。在整型类型前加上关键字 unsigned 表明该类型是无符号整型：unsignedint、unsigned long、unsigned short。单独的 unsigned 相当于 unsignedint。

#### 字符类型：

可打印出来的符号（如 A、& 和 +）都是字符。根据定义，char 类型表示一个字符要占用 1 字节内存。出于历史原因，1 字节通常是 8 位，但是如果要表示基本字符集，也可以是 16 位或更大。

- char ——字符类型的关键字。有些编译器使用有符号的 char，而有些则使用无符号的 char。  
在需要时，可在 char 前面加上关键字 signed 或 unsigned 来指明具体使用哪一种类型。

#### 布尔类型：

布尔值表示 true 和 false。C 语言用 1 表示 true，0 表示 false。

- \_Bool ——布尔类型的关键字。布尔类型是无符号 int 类型，所占用的空间只要能储存 0 或 1 即可。

#### 实浮点类型：

实浮点类型可表示正浮点数和负浮点数。

- float ——系统的基本浮点类型，可精确表示至少 6 位有效数字。
- double ——储存浮点数的范围（可能）更大，能表示比 float 类型更多的有效数字（至少 10 位，通常会更多）和更大的指数。
- long double ——储存浮点数的范围（可能）比 double 更大，能表示比 double 更多的有效数字和更大的指数。

#### 复数和虚数浮点数：

虚数类型是可选的类型。复数的实部和虚部类型都基于实浮点类型来构成：

- float \_Complex
- double \_Complex
- long double \_Complex
- float \_Imaginary
- double \_Imaginary
- long long \_Imaginary

### 小结：如何声明简单变量

1. 选择需要的类型。
2. 使用有效的字符给变量起一个变量名。
3. 按以下格式进行声明：

类型说明符 变量名；

类型说明符由一个或多个关键字组成。下面是一些示例：

```
int erest;
unsigned short cash;
```

4. 可以同时声明相同类型的多个变量，用逗号分隔各变量名，如下所示：

```
char ch, init, ans;
```

5. 在声明的同时还可以初始化变量：

```
float mass = 6.0E24;
```

### 3.4.9 类型大小

如何知道当前系统的指定类型的大小是多少？运行程序清单 3.8，会列出当前系统的各类型的大小。

程序清单 3.8 typesize.c 程序

```
/* typesize.c -- 打印类型大小 */
#include <stdio.h>
int main(void)
{
    /* C99 为类型大小提供%zd 转换说明 */
    printf("Type int has a size of %zd bytes.\n", sizeof(int));
    printf("Type char has a size of %zd bytes.\n", sizeof(char));
    printf("Type long has a size of %zd bytes.\n", sizeof(long));
    printf("Type long long has a size of %zd bytes.\n",
           sizeof(long long));
    printf("Type double has a size of %zd bytes.\n",
           sizeof(double));
    printf("Type long double has a size of %zd bytes.\n",
           sizeof(long double));
    return 0;
}
```

`sizeof` 是 C 语言的内置运算符，以字节为单位给出指定类型的大小。C99 和 C11 提供`%zd` 转换说明匹配`sizeof` 的返回类型<sup>1</sup>。一些不支持 C99 和 C11 的编译器可用`%u` 或`%lu` 代替`%zd`。

该程序的输出如下：

```
Type int has a size of 4 bytes.
Type char has a size of 1 bytes.
Type long has a size of 8 bytes.
Type long long has a size of 8 bytes.
Type double has a size of 8 bytes.
Type long double has a size of 16 bytes.
```

该程序列出了 6 种类型的大小，你也可以把程序中的类型更换成感兴趣的其他类型。注意，因为 C 语言定义了`char` 类型是 1 字节，所以`char` 类型的大小一定是 1 字节。而在`char` 类型为 16 位、`double`

<sup>1</sup> 即，`size_t` 类型。——译者注

类型为 64 位的系统中，`sizeof` 给出的 `double` 是 4 字节。在 `limits.h` 和 `float.h` 头文件中有类型限制的相关信息（下一章将详细介绍这两个头文件）。

顺带一提，注意该程序最后几行 `printf()` 语句都被分为两行，只要不在引号内部或一个单词中间断行，就可以这样写。

## 3.5 使用数据类型

编写程序时，应注意合理选择所需的变量及其类型。通常，用 `int` 或 `float` 类型表示数字，`char` 类型表示字符。在使用变量之前必须先声明，并选择有意义的变量名。初始化变量应使用与变量类型匹配的常数类型。例如：

```
int apples = 3;      /* 正确 */
int oranges = 3.0;   /* 不好的形式 */
```

与 Pascal 相比，C 在检查类型匹配方面不太严格。C 编译器甚至允许二次初始化，但在激活了较高级别警告时，会给出警告。最好不要养成这样的习惯。

把一个类型的数值初始化给不同类型的变量时，编译器会把值转换成与变量匹配的类型，这将导致部分数据丢失。例如，下面的初始化：

```
int cost = 12.99;    /* 用 double 类型的值初始化 int 类型的变量 */
float pi = 3.1415926536; /* 用 double 类型的值初始化 float 类型的变量 */
```

第 1 个声明，`cost` 的值是 12。C 编译器把浮点数转换成整数时，会直接丢弃（截断）小数部分，而不进行四舍五入。第 2 个声明会损失一些精度，因为 C 只保证了 `float` 类型前 6 位的精度。编译器对这样的初始化可能给出警告。读者在编译程序清单 3.1 时可能就遇到了这种警告。

许多程序员和公司内部都有系统化的命名约定，在变量名中体现其类型。例如，用 `i_` 前缀表示 `int` 类型，`us_` 前缀表示 `unsigned short` 类型。这样，一眼就能看出来 `i_smart` 是 `int` 类型的变量，`us_versmart` 是 `unsigned short` 类型的变量。

## 3.6 参数和陷阱

有必要再次提醒读者注意 `printf()` 函数的用法。读者应该还记得，传递给函数的信息被称为参数。例如，`printf("Hello, pal.")` 函数调用有一个参数：`"Hello, pal."`。双引号中的字符序列（如，`"Hello, pal."`）被称为字符串（string），第 4 章将详细讲解相关内容。现在，关键是要理解无论双引号中包含多少个字符和标点符号，一个字符串就是一个参数。

与此类似，`scanf("%d", &weight)` 函数调用有两个参数：`%d` 和 `&weight`。C 语言用逗号分隔函数中的参数。`printf()` 和 `scanf()` 函数与一般函数不同，它们的参数个数是可变的。例如，前面的程序示例中调用过带一个、两个，甚至三个参数的 `printf()` 函数。程序要知道函数的参数个数才能正常工作。`printf()` 和 `scanf()` 函数用第 1 个参数表明后续有多少个参数，即第 1 个字符串中的转换说明与后面的参数一一对应。例如，下面的语句有两个 `%d` 转换说明，说明后面还有两个参数：

```
printf("%d cats ate %d cans of tuna\n", cats, cans);
```

后面的确还有两个参数：`cats` 和 `cans`。

程序员要负责确保转换说明的数量、类型与后面参数的数量、类型相匹配。现在，C 语言通过函数原型机制检查函数调用时参数的个数和类型是否正确。但是，该机制对 `printf()` 和 `scanf()` 不起作用，因为这两个函数的参数个数可变。如果参数在匹配上有问题，会出现什么情况？假设你编写了程序清单 3.9

中的程序。

#### 程序清单 3.9 badcount.c 程序

---

```
/* badcount.c -- 参数错误的情况 */
#include <stdio.h>
int main(void)
{
    int n = 4;
    int m = 5;
    float f = 7.0f;
    float g = 8.0f;

    printf("%d\n", n, m);      /* 参数太多 */
    printf("%d %d %d\n", n);   /* 参数太少 */
    printf("%d %d\n", f, g);   /* 值的类型不匹配 */

    return 0;
}
```

---

XCode 4.6 (OS 10.8) 的输出如下：

```
4
4 1 -706337836
1606414344 1
```

Microsoft Visual Studio Express 2012 (Windows 7) 的输出如下：

```
4
4 0 0
0 1075576832
```

注意，用%d 显示 float 类型的值，其值不会被转换成 int 类型。在不同的平台下，缺少参数或参数类型不匹配导致的结果不同。

所有编译器都能顺利编译并运行该程序，但其中大部分会给出警告。的确，有些编译器会捕获到这类问题，然而 C 标准对此未作要求。因此，计算机在运行时可能不会捕获这类错误。如果程序正常运行，很难觉察出来。如果程序没有打印出期望值或打印出意想不到的值，你才会检查 printf() 函数中的参数个数和类型是否得当。

## 3.7 转义序列示例

再来看一个程序示例，该程序使用了一些特殊的转义序列。程序清单 3.10 演示了退格 (\b)、水平制表符 (\t) 和回车 (\r) 的工作方式。这些概念在计算机使用电传打字机作为输出设备时就有了，但是它们不一定能与现代的图形接口兼容。例如，程序清单 3.10 在某些 Macintosh 的实现中就无法正常运行。

#### 程序清单 3.10 escape.c 程序

---

```
/* escape.c -- 使用转移序列 */
#include <stdio.h>
int main(void)
{
    float salary;

    printf("\aEnter your desired monthly salary:"); /* 1 */
    printf(" $ _____\b\b\b\b\b\b\b");                /* 2 */
```

```

scanf("%f", &salary);
printf("\n\t$.2f a month is $.2f a year.", salary,
      salary * 12.0); /* 3 */
printf("\rGee!\n"); /* 4 */

return 0;
}

```

### 3.7.1 程序运行情况

假设在系统中运行的转义序列行为与本章描述的行为一致（实际行为可能不同。例如，XCode 4.6 把\ a、\ b 和\ r 显示为颠倒的问号），下面我们来分析这个程序。

第1条 printf()语句（注释中标为1）发出一声警报（因为使用了\ a），然后打印下面的内容：

Enter your desired monthly salary:

因为 printf() 中的字符串末尾没有\n，所以光标停留在冒号后面。

第2条 printf()语句在光标处接着打印，屏幕上显示的内容是：

Enter your desired monthly salary: \$ \_\_\_\_\_

冒号和美元符号之间有一个空格，这是因为第2条 printf() 语句中的字符串以一个空格开始。7个退格字符使得光标左移7个位置，即把光标移至7个下划线字符的前面，紧跟在美元符号后面。通常，退格不会擦除退回所经过的字符，但有些实现是擦除的，这和本例不同。

假设键入的数据是 4000.00（并按下 Enter 键），屏幕显示的内容应该是：

Enter your desired monthly salary: \$4000.00

键入的字符替换了下划线字符。按下 Enter 键后，光标移至下一行的起始处。

第3条 printf()语句中的字符串以\n\t 开始。换行字符使光标移至下一行起始处。水平制表符使光标移至该行的下一个制表点，一般是第9列（但不一定）。然后打印字符串中的其他内容。执行完该语句后，此时屏幕显示的内容应该是：

Enter your desired monthly salary: \$4000.00  
\$4000.00 a month is \$48000.00 a year.

因为这条 printf() 语句中没有使用换行字符，所以光标停留在最后的点号后面。

第4条 printf()语句以\r 开始。这使得光标回到当前行的起始处。然后打印 Gee!，接着\n 使光标移至下一行的起始处。屏幕最后显示的内容应该是：

Enter your desired monthly salary: \$4000.00  
Gee! \$4000.00 a month is \$48000.00 a year.

### 3.7.2 刷新输出

printf() 何时把输出发送到屏幕上？最初，printf() 语句把输出发送到一个叫作缓冲区（buffer）的中间存储区域，然后缓冲区中的内容再不断被发送到屏幕上。C 标准明确规定了何时把缓冲区中的内容发送到屏幕：当缓冲区满、遇到换行字符或需要输入的时候（从缓冲区把数据发送到屏幕或文件被称为刷新缓冲区）。例如，前两个 printf() 语句既没有填满缓冲区，也没有换行符，但是下一条 scanf() 语句要求用户输入，这迫使 printf() 的输出被发送到屏幕上。

旧式编译器遇到 scanf() 也不会强行刷新缓冲区，程序会停在那里不显示任何提示内容，等待用户输入数据。在这种情况下，可以使用换行字符刷新缓冲区。代码应改为：

```
printf("Enter your desired monthly salary:\n");
```

```
scanf("%f", &salary);
```

无论接下来的输入是否能刷新缓冲区，代码都会正常运行。这将导致光标移至下一行起始处，用户无法在提示内容同一行输入数据。还有一种刷新缓冲区的方法是使用 `fflush()` 函数，详见第 13 章。

## 3.8 关键概念

C 语言提供了大量的数值类型，目的是为程序员提供方便。那以整数类型为例，C 认为一种整型不够，提供了有符号、无符号，以及大小不同的整型，以满足不同程序的需求。

计算机中的浮点数和整数在本质上不同，其存储方式和运算过程有很大区别。即使两个 32 位存储单元储存的位组合完全相同，但是一个解释为 `float` 类型，另一个解释为 `long` 类型，这两个相同的位组合表示的值也完全不同。例如，在 PC 中，假设一个位组合表示 `float` 类型的数 256.0，如果将其解释为 `long` 类型，得到的值是 113246208。C 语言允许编写混合数据类型的表达式，但是会进行自动类型转换，以便在实际运算时统一使用一种类型。

计算机在内存中用数值编码来表示字符。美国最常用的是 ASCII 码，除此之外 C 也支持其他编码。字符常量是计算机系统使用的数值编码的符号表示，它表示为单引号括起来的字符，如 '`A`'。

## 3.9 本章小结

C 有多种的数据类型。基本数据类型分为两大类：整数类型和浮点数类型。通过为类型分配的储存量以及是有符号还是无符号，区分不同的整数类型。最小的整数类型是 `char`，因实现不同，可以是有符号的 `char` 或无符号的 `char`，即 `unsigned char` 或 `signed char`。但是，通常用 `char` 类型表示小整数时才这样显示说明。其他整数类型有 `short`、`int`、`long` 和 `long long` 类型。C 规定，后面的类型不能小于前面的类型。上述都是有符号类型，但也可以使用 `unsigned` 关键字创建相应的无符号类型：`unsigned short`、`unsigned int`、`unsigned long` 和 `unsigned long long`。或者，在类型名前加上 `signed` 修饰符显式表明该类型是有符号类型。最后，`_Bool` 类型是一种无符号类型，可储存 0 或 1，分别代表 `false` 和 `true`。

浮点类型有 3 种：`float`、`double` 和 C90 新增的 `long double`。后面的类型应大于或等于前面的类型。有些实现可选择支持复数类型和虚数类型，通过关键字 `_Complex` 和 `_Imaginary` 与浮点类型的关键字组合（如，`double _Complex` 类型和 `float _Imaginary` 类型）来表示这些类型。

整数可以表示为十进制、八进制或十六进制。0 前缀表示八进制数，`0x` 或 `0X` 前缀表示十六进制数。例如，`32`、`040`、`0x20` 分别以十进制、八进制、十六进制表示同一个值。`l` 或 `L` 前缀表明该值是 `long` 类型，`ll` 或 `LL` 前缀表明该值是 `long long` 类型。

在 C 语言中，直接表示一个字符常量的方法是：把该字符用单引号括起来，如 '`Q`'、'`8`' 和 '`$`'。C 语言的转义序列（如，'`\n`'）表示某些非打印字符。另外，还可以在八进制或十六进制数前加上一个反斜杠（如，'`\007`'），表示 ASCII 码中的一个字符。

浮点数可写成固定小数点的形式（如，`9393.912`）或指数形式（如，`7.38E10`）。C99 和 C11 提供了第 3 种指数表示法，即用十六进制数和 2 的幂来表示（如，`0xa.1fp10`）。

`printf()` 函数根据转换说明打印各种类型的值。转换说明最简单的形式由一个百分号（%）和一个转换字符组成，如 `%d` 或 `%f`。

## 3.10 复习题

复习题的参考答案在附录 A 中。

1. 指出下面各种数据使用的合适数据类型（有些可使用多种数据类型）：
  - a. East Simpleton 的人口
  - b. DVD 影碟的价格
  - c. 本章出现次数最多的字母
  - d. 本章出现次数最多的字母次数
2. 在什么情况下要用 long 类型的变量代替 int 类型的变量？
3. 使用哪些可移植的数据类型可以获得 32 位有符号整数？选择的理由是什么？
4. 指出下列常量的类型和含义（如果有的话）：
  - a. '\b'
  - b. 1066
  - c. 99.44
  - d. 0XAA
  - e. 2.0e30
5. Dottie Cawm 编写了一个程序，请找出程序中的错误。

```
include <stdio.h>
main
(
    float g; h;
    float tax, rate;

    g = e21;
    tax = rate*g;
)
```

6. 写出下列常量在声明中使用的数据类型和在 printf() 中对应的转换说明：

| 常量        | 类型     | 转换说明 (% 转换字符) |
|-----------|--------|---------------|
| 12        | int    |               |
| 0X3       | int    |               |
| 'C'       | char   |               |
| 2.34E07   | float  |               |
| '\040'    | char   |               |
| 7.0       | double |               |
| 6L        | long   |               |
| 6.0f      | float  |               |
| 0x5.b6p12 | double |               |

7. 写出下列常量在声明中使用的数据类型和在 printf() 中对应的转换说明（假设 int 为 16 位）：

| 常量      | 类型 | 转换说明 (%转换字符) |
|---------|----|--------------|
| 012     |    |              |
| 2.9e05L |    |              |
| 's'     |    |              |
| 100000  |    |              |
| '\n'    |    |              |
| 20.0f   |    |              |
| 0x44    |    |              |
| -40     |    |              |

8. 假设程序的开头有下列声明:

```
int imate = 2;
long shot = 53456;
char grade = 'A';
float log = 2.71828;
```

把下面 printf() 语句中的转换字符补充完整:

```
printf("The odds against the %__ were %__ to 1.\n", imate, shot);
printf("A score of %__ is not an %__ grade.\n", log, grade);
```

9. 假设 ch 是 char 类型的变量。分别使用转义序列、十进制值、八进制字符常量和十六进制字符常量把回车字符赋给 ch (假设使用 ASCII 编码值)。

10. 修正下面的程序 (在 C 中, / 表示除以)。

```
void main(int) / this program is perfect /
{
    cows, legs integer;
    printf("How many cow legs did you count?\n");
    scanf("%c", legs);
    cows = legs / 4;
    printf("That implies there are %f cows.\n", cows)
}
```

11. 指出下列转义序列的含义:

- a. \n
- b. \\
- c. \"
- d. \t

## 3.11 编程练习

- 通过试验 (即编写带有此类问题的程序) 观察系统如何处理整数上溢、浮点数上溢和浮点数下溢的情况。
- 编写一个程序, 要求提示输入一个 ASCII 码值 (如, 66), 然后打印输入的字符。
- 编写一个程序, 发出一声警报, 然后打印下面的文本:  

```
Startled by the sudden sound, Sally shouted,
"By the Great Pumpkin, what was that!"
```
- 编写一个程序, 读取一个浮点数, 先打印成小数点形式, 再打印成指数形式。然后, 如果系统支持,

再打印成 p 记数法（即十六进制记数法）。按以下格式输出（实际显示的指数位数因系统而异）：

```
Enter a floating-point value: 64.25
fixed-point notation: 64.250000
exponential notation: 6.425000e+01
p notation: 0x1.01p+6
```

5. 一年大约有  $3.156 \times 10^7$  秒。编写一个程序，提示用户输入年龄，然后显示该年龄对应的秒数。
6. 1 个水分子的质量约为  $3.0 \times 10^{-23}$  克。1 夸脱水大约是 950 克。编写一个程序，提示用户输入水的夸脱数，并显示水分子的数量。
7. 1 英寸相当于 2.54 厘米。编写一个程序，提示用户输入身高（/英寸），然后以厘米为单位显示身高。
8. 在美国的体积测量系统中，1 品脱等于 2 杯，1 杯等于 8 盎司，1 盎司等于 2 大汤勺，1 大汤勺等于 3 茶勺。编写一个程序，提示用户输入杯数，并以品脱、盎司、汤勺、茶勺为单位显示等价容量。  
思考对于该程序，为何使用浮点类型比整数类型更合适？



# 字符串和格式化输入/输出

本章介绍以下内容：

- 函数：strlen()
- 关键字：const
- 字符串
- 如何创建、存储字符串
- 如何使用 strlen() 函数获取字符串的长度
- 用 C 预处理器指令#define 和 ANSI C 的 const 修饰符创建符号常量

本章重点介绍输入和输出。与程序交互和使用字符串可以编写个性化的程序，本章将详细介绍 C 语言的两个输入/输出函数：printf() 和 scanf()。学会使用这两个函数，不仅能与用户交互，还可根据个人喜好和任务要求格式化输出。最后，简要介绍一个重要的工具——C 预处理器指令，并学习如何定义、使用符号常量。

## 4.1 前导程序

与前两章一样，本章以一个简单的程序开始。程序清单 4.1 与用户进行简单的交互。为了使程序的形式灵活多样，代码中使用了新的注释风格。

程序清单 4.1 talkback.c 程序

```
// talkback.c -- 演示与用户交互
#include <stdio.h>
#include <string.h>      // 提供 strlen() 函数的原型
#define DENSITY 62.4     // 人体密度（单位：磅/立方英尺）
int main()
{
    float weight, volume;
    int size, letters;
    char name[40];        // name 是一个可容纳 40 个字符的数组

    printf("Hi! What's your first name?\n");
    scanf("%s", name);
    printf("%s, what's your weight in pounds?\n", name);
    scanf("%f", &weight);
    size = sizeof name;
    letters = strlen(name);
    volume = weight / DENSITY;
    printf("Well, %s, your volume is %.2f cubic feet.\n",
           name, volume);
    printf("Also, your first name has %d letters,\n",
           letters);
```

```

    printf("and we have %d bytes to store it.\n", size);

    return 0;
}

```

运行 talkback.c 程序，输入结果如下：

```

Hi! What's your first name?
Christine
Christine, what's your weight in pounds?
154
Well, Christine, your volume is 2.47 cubic feet.
Also, your first name has 9 letters,
and we have 40 bytes to store it.

```

该程序包含以下新特性。

- 用数组 (*array*) 储存字符串 (*character string*)。在该程序中，用户输入的名被储存在数组中，该数组占用内存中 40 个连续的字节，每个字节储存一个字符值。
- 使用 %s 转换说明来处理字符串的输入和输出。注意，在 `scanf()` 中，`name` 没有&前缀，而 `weight` 有（稍后解释，`&weight` 和 `name` 都是地址）。
- 用 C 预处理器把字符常量 DENSITY 定义为 62.4。
- 用 C 函数 `strlen()` 获取字符串的长度。

对于 BASIC 的输入/输出而言，C 的输入/输出看上去有些复杂。不过，复杂换来的是程序的高效和方便控制输入/输出。而且，一旦熟悉用法后，会发现它很简单。

## 4.2 字符串简介

字符串 (*character string*) 是一个或多个字符的序列，如下所示：

```
"Zing went the strings of my heart!"
```

双引号不是字符串的一部分。双引号仅告知编译器它括起来的是字符串，正如单引号用于标识单个字符一样。

### 4.2.1 char 类型数组和 null 字符

C 语言没有专门用于储存字符串的变量类型，字符串都被储存在 `char` 类型的数组中。数组由连续的存储单元组成，字符串中的字符被储存在相邻的存储单元中，每个单元储存一个字符（见图 4.1）。

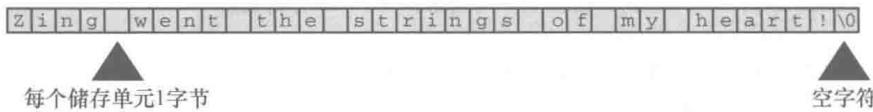


图 4.1 数组中的字符串

注意图 4.1 中数组末尾位置的字符 `\0`。这是空字符 (*null character*)，C 语言用它标记字符串的结束。空字符不是数字 0，它是非打印字符，其 ASCII 码值是（或等价于）0。C 中的字符串一定以空字符结束，这意味着数组的容量必须至少比待存储字符串中的字符数多 1。因此，程序清单 4.1 中有 40 个存储单元的字符串，只能储存 39 个字符，剩下一个字节留给空字符。

那么，什么是数组？可以把数组看作是一行连续的多个存储单元。用更正式的说法是，数组是同类型

数据元素的有序序列。程序清单 4.1 通过以下声明创建了一个包含 40 个存储单元（或元素）的数组，每个单元储存一个 char 类型的值：

```
char name[40];
```

name 后面的方括号表明这是一个数组，方括号中的 40 表明该数组中的元素数量。char 表明每个元素的类型（见图 4.2）。

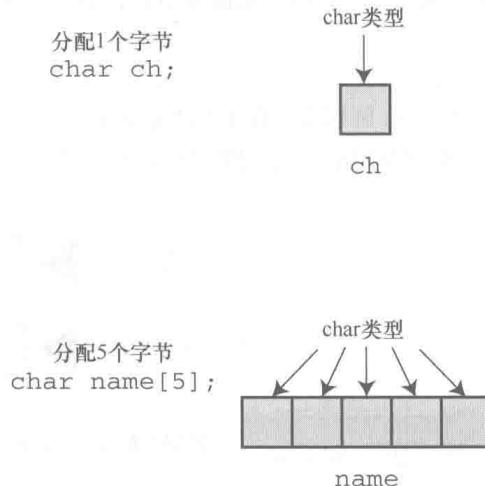


图 4.2 声明一个变量和声明一个数组

字符串看上去比较复杂！必须先创建一个数组，把字符串中的字符逐个放入数组，还要记得在末尾加上一个\0。还好，计算机可以自己处理这些细节。

## 4.2.2 使用字符串

试着运行程序清单 4.2，使用字符串其实很简单。

### 程序清单 4.2 praise1.c 程序

---

```
/* praise1.c -- 使用不同类型的字符串 */
#include <stdio.h>
#define PRAISE "You are an extraordinary being."
int main(void)
{
    char name[40];

    printf("What's your name? ");
    scanf("%s", name);
    printf("Hello, %s. %s\n", name, PRAISE);

    return 0;
}
```

---

%s 告诉 printf() 打印一个字符串。%s 出现了两次，因为程序要打印两个字符串：一个储存在 name 数组中；一个由 PRAISE 来表示。运行 praise1.c，其输出如下所示：

```
What's your name? Angela Plains
Hello, Angela. You are an extraordinary being.
```

你不用亲自把空字符放入字符串末尾，scanf() 在读取输入时就已完成这项工作。也不用在字符串常

量 PRAISE 末尾添加空字符。稍后我们会解释#define 指令，现在先理解 PRAISE 后面用双引号括起来的文本是一个字符串。编译器会在末尾加上空字符。

注意（这很重要），scanf() 只读取了 Angela Plains 中的 Angela，它在遇到第 1 个空白（空格、制表符或换行符）时就不再读取输入。因此，scanf() 在读到 Angela 和 Plains 之间的空格时就停止了。一般而言，根据%s 转换说明，scanf() 只会读取字符串中的一个单词，而不是一整句。C 语言还有其他的输入函数（如，fgets()），用于读取一般字符串。后面章节将详细介绍这些函数。

### 字符串和字符

字符串常量 "x" 和字符常量 'x' 不同。区别之一在于 'x' 是基本类型 (char)，而 "x" 是派生类型 (char 数组)；区别之二是 "x" 实际上由两个字符组成：'x' 和空字符 \0（见图 4.3）。

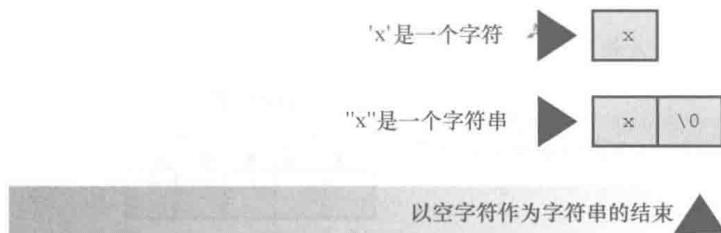


图 4.3 字符 'x' 和字符串 "x"

### 4.2.3 strlen() 函数

上一章提到了 sizeof 运算符，它以字节为单位给出对象的大小。strlen() 函数给出字符串中的字符长度。因为 1 字节储存一个字符，读者可能认为把两种方法应用于字符串得到的结果相同，但事实并非如此。请根据程序清单 4.3，在程序清单 4.2 中添加几行代码，看看为什么会这样。

程序清单 4.3 praise2.c 程序

```
/* praise2.c */
// 如果编译器不识别%zd, 尝试换成%u 或%lu。
#include <stdio.h>
#include <string.h>      /* 提供 strlen() 函数的原型 */
#define PRAISE "You are an extraordinary being."
int main(void)
{
    char name[40];

    printf("What's your name? ");
    scanf("%s", name);
    printf("Hello, %s. %s\n", name, PRAISE);
    printf("Your name of %zd letters occupies %zd memory cells.\n",
           strlen(name), sizeof name);
    printf("The phrase of praise has %zd letters ",
           strlen(PRAISE));
    printf("and occupies %zd memory cells.\n", sizeof PRAISE);

    return 0;
}
```

如果使用 ANSI C 之前的编译器，必须移除这一行：

```
#include <string.h>
```

string.h 头文件包含多个与字符串相关的函数原型，包括 `strlen()`。第 11 章将详细介绍该头文件（顺带一提，一些 ANSI 之前的 UNIX 系统用 `strings.h` 代替 `string.h`，其中也包含了一些字符串函数的声明）。

一般而言，C 把函数库中相关的函数归为一类，并为每类函数提供一个头文件。例如，`printf()` 和 `scanf()` 都隶属标准输入和输出函数，使用 `stdio.h` 头文件。`string.h` 头文件中包含了 `strlen()` 函数和其他一些与字符串相关的函数（如拷贝字符串的函数和字符串查找函数）。

注意，程序清单 4.3 使用了两种方法处理很长的 `printf()` 语句。第 1 种方法是将 `printf()` 语句分为两行（可以在参数之间断为两行，但是不要在双引号中的字符串中间断开）；第 2 种方法是使用两个 `printf()` 语句打印一行内容，只在第 2 条 `printf()` 语句中使用换行符 (`\n`)。运行该程序，其交互输出如下：

```
What's your name? Serendipity Chance
Hello, Serendipity. You are an extraordinary being.
Your name of 11 letters occupies 40 memory cells.
The phrase of praise has 31 letters and occupies 32 memory cells.
```

`sizeof` 运算符报告，`name` 数组有 40 个存储单元。但是，只有前 11 个单元用来储存 `Serendipity`，所以 `strlen()` 得出的结果是 11。`name` 数组的第 12 个单元储存空字符，`strlen()` 并未将其计入。图 4.4 演示了这个概念。

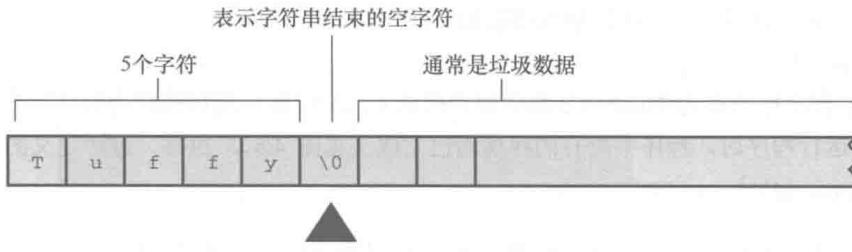


图 4.4 `strlen()` 函数知道在何处停止

对于 `PRAISE`，用 `strlen()` 得出的也是字符串中的字符数（包括空格和标点符号）。然而，`sizeof` 运算符给出的数更大，因为它把字符串末尾不可见的空字符也计算在内。该程序并未明确告诉计算机要给字符串预留多少空间，所以它必须计算双引号内的字符数。

第 3 章提到过，C99 和 C11 标准专门为 `sizeof` 运算符的返回类型添加了 `%zd` 转换说明，这对于 `strlen()` 同样适用。对于早期的 C，还要知道 `sizeof` 和 `strlen()` 返回的实际类型（通常是 `unsigned` 或 `unsigned long`）。

另外，还要注意一点：上一章的 `sizeof` 使用了圆括号，但本例没有。圆括号的使用时机否取决于运算对象是类型还是特定量？运算对象是类型时，圆括号必不可少，但是对于特定量，可有可无。也就是说，对于类型，应写成 `sizeof(char)` 或 `sizeof(float)`；对于特定量，可写成 `sizeof name` 或 `sizeof 6.28`。尽管如此，还是建议所有情况下都使用圆括号，如 `sizeof(6.28)`。

程序清单 4.3 中使用 `strlen()` 和 `sizeof`，完全是为了满足读者的好奇心。在实际应用中，`strlen()` 和 `sizeof` 是非常重要的编程工具。例如，在各种要处理字符串的程序中，`strlen()` 很有用。详见第 11 章。

下面我们来学习 `#define` 指令。

## 4.3 常量和 C 预处理器

有时，在程序中要使用常量。例如，可以这样计算圆的周长：

```
circumference = 3.14159 * diameter;
```

这里，常量 3.14159 代表著名的常量 pi (π)。在该例中，输入实际值便可使用这个常量。然而，这种情况使用符号常量 (*symbolic constant*) 会更好。也就是说，使用下面的语句，计算机稍后会用实际值完成替换：

```
circumference = pi * diameter;
```

为什么使用符号常量更好？首先，常量名比数字表达的信息更多。请比较以下两条语句：

```
owed = 0.015 * housevalue;
owed = taxrate * housevalue;
```

如果阅读一个很长的程序，第 2 条语句所表达的含义更清楚。

另外，假设程序中的多处使用一个常量，有时需要改变它的值。毕竟，税率通常是浮动的。如果程序使用符号常量，则只需更改符号常量的定义，不用在程序中查找使用常量的地方，然后逐一修改。

那么，如何创建符号常量？方法之一是声明一个变量，然后将该变量设置为所需的常量。可以这样写：

```
float taxrate;
taxrate = 0.015;
```

这样做提供了一个符号名，但是 taxrate 是一个变量，程序可能会无意间改变它的值。C 语言还提供了一个更好的方案——C 预处理器。第 2 章中介绍了预处理器如何使用 #include 包含其他文件的信息。预处理器也可用来定义常量。只需在程序顶部添加下面一行：

```
#define TAXRATE 0.015
```

编译程序时，程序中所有的 TAXRATE 都会被替换成 0.015。这一过程被称为编译时替换 (*compile-time substitution*)。在运行程序时，程序中所有的替换均已完成（见图 4.5）。通常，这样定义的常量也称为明示常量 (*manifest constant*)<sup>1</sup>。

请注意格式，首先是#define，接着是符号常量名 (TAXRATE)，然后是符号常量的值 (0.015)（注意，其中并没有=符号）。所以，其通用格式如下：

```
#define NAME value
```

实际应用时，用选定的符号常量名和合适的值来替换 NAME 和 value。注意，末尾不用加分号，因为这是一种由预处理器处理的替换机制。为什么 TAXRATE 要用大写？用大写表示符号常量是 C 语言一贯的传统。这样，在程序中看到全大写的名称就立刻明白这是一个符号常量，而非变量。大写常量只是为了提高程序的可读性，即使全用小写来表示符号常量，程序也能照常运行。尽管如此，初学者还是应该养成大写常量的好习惯。

另外，还有一个不常用的命名约定，即在名称前带 c\_ 或 k\_ 前缀来表示常量(如, c\_level 或 k\_line)。

符号常量的命名规则与变量相同。可以使用大小写字母、数字和下划线字符，首字符不能为数字。程序清单 4.4 演示了一个简单的示例。

<sup>1</sup> 其实，符号常量的概念在 K&R 合著的《C 语言程序设计》中介绍过。但是，在历年的 C 标准中（包括最新的 C11），并没有符号常量的概念，只提到过#define 最简单的用法是定义一个“明示常量”。市面上各编程书籍对此概念的理解不同，有些作者把#define 宏定义实现的“常量”归为“明示常量”；有些作者（如，本书的作者）则认为“明示常量”相当于“符号常量”。——译者注

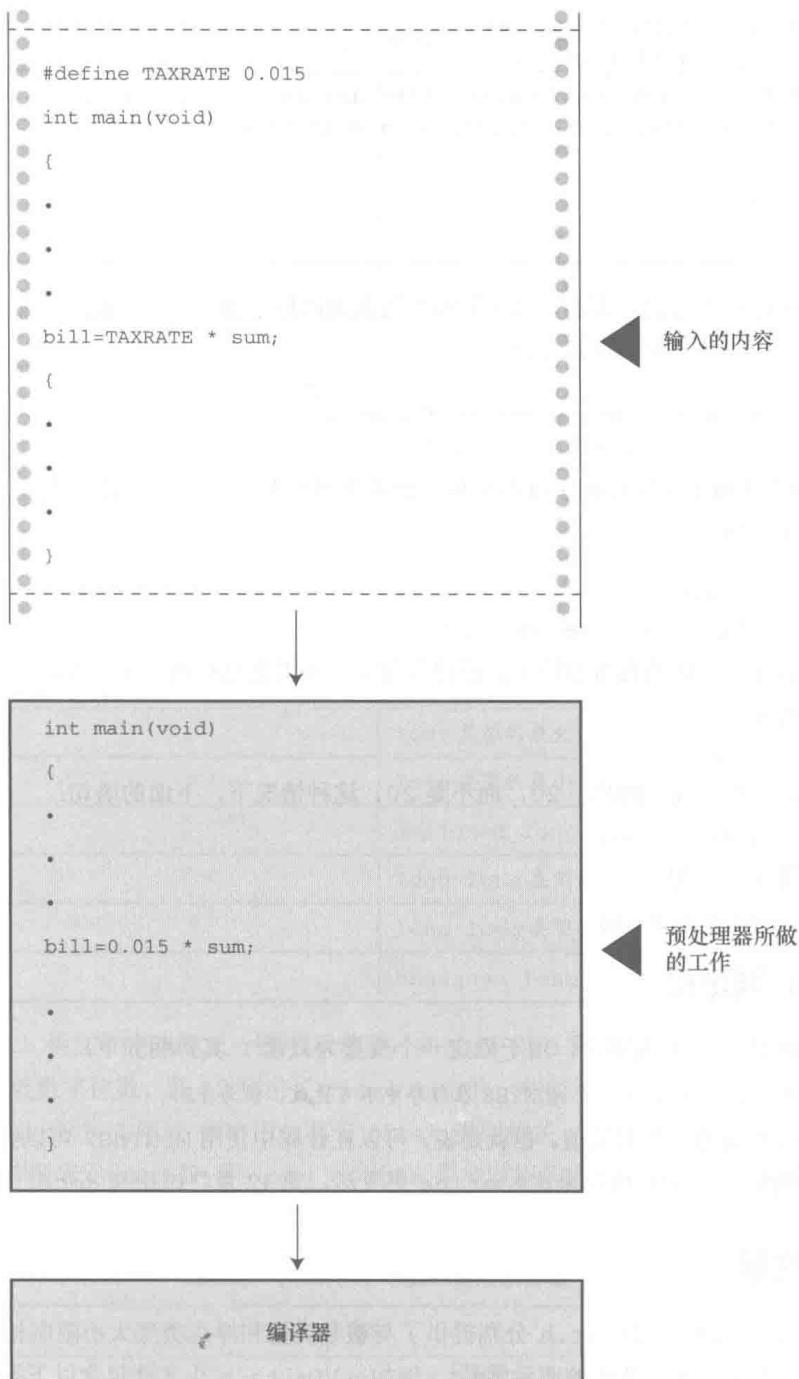


图 4.5 输入的内容和编译后的内容

**程序清单 4.4 pizza.c 程序**

```
/* pizza.c -- 在比萨饼程序中使用已定义的常量 */
#include <stdio.h>
#define PI 3.14159
int main(void)
{
    float area, circum, radius;

    printf("What is the radius of your pizza?\n");
    scanf("%f", &radius);
```

```

area = PI * radius * radius;
circum = 2.0 * PI *radius;
printf("Your basic pizza parameters are as follows:\n");
printf("circumference = %1.2f, area = %1.2f\n", circum,area);

return 0;
}

```

printf()语句中的%1.2f表明，结果被四舍五入为两位小数输出。下面是一个输出示例：

```

What is the radius of your pizza?
6.0
Your basic pizza parameters are as follows:
circumference = 37.70, area = 113.10

```

#define指令还可定义字符和字符串常量。前者使用单引号，后者使用双引号。如下所示：

```

#define BEEP '\a'
#define TEE 'T'
#define ESC '\033'
#define OOPS "Now you have done it!"

```

记住，符号常量名后面的内容被用来替换符号常量。不要犯这样的常见错误：

```

/* 错误的格式 */
#define TOES = 20

```

如果这样做，替换TOES的是= 20，而不是20。这种情况下，下面的语句：

```

digits = fingers + TOES;

```

将被转换成错误的语句：

```

digits = fingers + = 20;

```

### 4.3.1 const 限定符

C90标准新增了const关键字，用于限定一个变量为只读<sup>1</sup>。其声明如下：

```

const int MONTHS = 12; // MONTHS在程序中不可更改，值为12

```

这使得MONTHS成为一个只读值。也就是说，可以在计算中使用MONTHS，可以打印MONTHS，但是不能更改MONTHS的值。const用起来比#define更灵活，第12章将讨论与const相关的内容。

### 4.3.2 明示常量

C头文件limits.h和float.h分别提供了与整数类型和浮点类型大小限制相关的详细信息。每个头文件都定义了一系列供实现使用的明示常量<sup>2</sup>。例如，limits.h头文件包含以下类似的代码：

```

#define INT_MAX +32767
#define INT_MIN -32768

```

这些明示常量代表int类型可表示的最大值和最小值。如果系统使用32位的int，该头文件会为这些明示常量提供不同的值。如果在程序中包含limits.h头文件，就可编写下面的代码：

```

printf("Maximum int value on this system = %d\n", INT_MAX);

```

如果系统使用4字节的int，limits.h头文件会提供符合4字节int的INT\_MAX和INT\_MIN。表4.1列出了limits.h中能找到的一些明示常量。

<sup>1</sup> 注意，在C语言中，用const类型限定符声明的是变量，不是常量。——译者注

<sup>2</sup> 再次提醒读者注意，本书作者认为“明示常量”相当于“符号常量”，经常在书中混用这两个术语。——译者注

表 4.1 limits.h 中的一些明示常量

| 明示常量       | 含义                        |
|------------|---------------------------|
| CHAR_BIT   | char 类型的位数                |
| CHAR_MAX   | char 类型的最大值               |
| CHAR_MIN   | char 类型的最小值               |
| SCHAR_MAX  | signed char 类型的最大值        |
| SCHAR_MIN  | signed char 类型的最小值        |
| UCHAR_MAX  | unsigned char 类型的最大值      |
| SHRT_MAX   | short 类型的最大值              |
| SHRT_MIN   | short 类型的最小值              |
| USHRT_MAX  | unsigned short 类型的最大值     |
| INT_MAX    | int 类型的最大值                |
| INT_MIN    | int 类型的最小值                |
| UINT_MAX   | unsigned int 的最大值         |
| LONG_MAX   | long 类型的最大值               |
| LONG_MIN   | long 类型的最小值               |
| ULONG_MAX  | unsigned long 类型的最大值      |
| LLONG_MAX  | long long 类型的最大值          |
| LLONG_MIN  | long long 类型的最小值          |
| ULLONG_MAX | unsigned long long 类型的最大值 |

类似地，float.h 头文件中也定义一些明示常量，如 FLT\_DIG 和 DBL\_DIG，分别表示 float 类型和 double 类型的有效数字位数。表 4.2 列出了 float.h 中的一些明示常量（可以使用文本编辑器打开并查看系统使用的 float.h 头文件）。表中所列都与 float 类型相关。把明示常量名中的 FLT 分别替换成 DBL 和 LDBL，即可分别表示 double 和 long double 类型对应的明示常量（表中假设系统使用 2 的幂来表示浮点数）。

表 4.2 float.h 中的一些明示常量

| 明示常量           | 含义                               |
|----------------|----------------------------------|
| FLT_MANT_DIG   | float 类型的尾数位数                    |
| FLT_DIG        | float 类型的最少有效数字位数（十进制）           |
| FLT_MIN_10_EXP | 带全部有效数字的 float 类型的最小负指数（以 10 为底） |
| FLT_MAX_10_EXP | float 类型的最大正指数（以 10 为底）          |
| FLT_MIN        | 保留全部精度的 float 类型最小正数             |
| FLT_MAX        | float 类型的最大正数                    |
| FLT_EPSILON    | 1.00 和比 1.00 大的最小 float 类型值之间的差值 |

程序清单 4.5 演示了如何使用 float.h 和 limits.h 中的数据（注意，编译器要完全支持 C99 标准才能识别 LLONG\_MIN 标识符）。

**程序清单 4.5 defines.c 程序**

```
// defines.c -- 使用 limit.h 和 float 头文件中定义的明示常量
#include <stdio.h>
#include <limits.h>      // 整型限制
#include <float.h>        // 浮点型限制
int main(void)
{
    printf("Some number limits for this system:\n");
    printf("Biggest int: %d\n", INT_MAX);
    printf("Smallest long long: %lld\n", LLONG_MIN);
    printf("One byte = %d bits on this system.\n", CHAR_BIT);
    printf("Largest double: %e\n", DBL_MAX);
    printf("Smallest normal float: %e\n", FLT_MIN);
    printf("float precision = %d digits\n", FLT_DIG);
    printf("float epsilon = %e\n", FLT_EPSILON);

    return 0;
}
```

该程序的输出示例如下：

```
Some number limits for this system:
Biggest int: 2147483647
Smallest long long: -9223372036854775808
One byte = 8 bits on this system.
Largest double: 1.797693e+308
Smallest normal float: 1.175494e-38
float precision = 6 digits
float epsilon = 1.192093e-07
```

C 预处理器是非常有用的工具，要好好利用它。本书的后面章节中会介绍更多相关应用。

## 4.4 printf() 和 scanf()

printf() 函数和 scanf() 函数能让用户可以与程序交流，它们是输入/输出函数，或简称为 I/O 函数。它们不仅是 C 语言中的 I/O 函数，而且是最才多艺的函数。过去，这些函数和 C 库的一些其他函数一样，并不是 C 语言定义的一部分。最初，C 把输入/输出的实现留给了编译器的作者，这样可以针对特殊的机器更好地匹配输入/输出。后来，考虑到兼容性的问题，各编译器都提供不同版本的 printf() 和 scanf()。尽管如此，各版本之间偶尔有一些差异。C90 和 C99 标准规定了这些函数的标准版本，本书亦遵循这一标准。

虽然 printf() 是输出函数，scanf() 是输入函数，但是它们的工作原理几乎相同。两个函数都使用格式字符串和参数列表。我们先介绍 printf()，再介绍 scanf()。

### 4.4.1 printf() 函数

请求 printf() 函数打印数据的指令要与待打印数据的类型相匹配。例如，打印整数时使用%d，打印字符串时使用%c。这些符号被称为转换说明（conversion specification），它们指定了如何把数据转换成可显示的形式。我们先列出 ANSI C 标准为 printf() 提供的转换说明，然后再示范如何使用一些较常见的转换说明。表 4.3 列出了一些转换说明和各自对应的输出类型。

表 4.3 转换说明及其打印的输出结果

| 转换说明 | 输出                                         |
|------|--------------------------------------------|
| %a   | 浮点数、十六进制数和 p 记数法 (C99/C11)                 |
| %A   | 浮点数、十六进制数和 p 记数法 (C99/C11)                 |
| %c   | 单个字符                                       |
| %d   | 有符号十进制整数                                   |
| %e   | 浮点数，e 记数法                                  |
| %E   | 浮点数，e 记数法                                  |
| %f   | 浮点数，十进制记数法                                 |
| %g   | 根据值的不同，自动选择%f 或%e。%e 格式用于指数小于-4 或者大于或等于精度时 |
| %G   | 根据值的不同，自动选择%f 或%E。%E 格式用于指数小于-4 或者大于或等于精度时 |
| %i   | 有符号十进制整数（与%d 相同）                           |
| %o   | 无符号八进制整数                                   |
| %p   | 指针                                         |
| %s   | 字符串                                        |
| %u   | 无符号十进制整数                                   |
| %x   | 无符号十六进制整数，使用十六进制数 0F                       |
| %X   | 无符号十六进制整数，使用十六进制数 0F                       |
| %%   | 打印一个百分号                                    |

## 4.4.2 使用 printf()

程序清单 4.6 的程序中使用了一些转换说明。

程序清单 4.6 printout.c 程序

```
/* printout.c -- 使用转换说明 */
#include <stdio.h>
#define PI 3.141593
int main(void)
{
    int number = 7;
    float pies = 12.75;
    int cost = 7800;

    printf("The %d contestants ate %f berry pies.\n", number,
           pies);
    printf("The value of pi is %f.\n", PI);
    printf("Farewell! thou art too dear for my possessing,\n");
    printf("%c%d\n", '$', 2 * cost);

    return 0;
}
```

该程序的输出如下：

```
The 7 contestants ate 12.750000 berry pies.  
The value of pi is 3.141593.  
Farewell! thou art too dear for my possessing,  
$15600
```

这是 printf() 函数的格式：

```
printf( 格式字符串, 待打印项 1, 待打印项 2, ... );
```

待打印项 1、待打印项 2 等都是要打印的项。它们可以是变量、常量，甚至是在打印之前先要计算的表达式。第 3 章提到过，格式字符串应包含每个待打印项对应的转换说明。例如，考虑下面的语句：

```
printf("The %d contestants ate %f berry pies.\n", number,pies);
```

格式字符串是双引号括起来的内容。上面语句的格式字符串包含了两个待打印项 number 和 poes 对应的两个转换说明。图 4.6 演示了 printf() 语句的另一个例子。

下面是程序清单 4.6 中的另一行：

```
printf("The value of pi is %f.\n", PI);
```

该语句中，待打印项列表只有一个项——符号常量 PI。

如图 4.7 所示，格式字符串包含两种形式不同的信息：

- 实际要打印的字符；
- 转换说明。



图 4.6 printf() 的参数

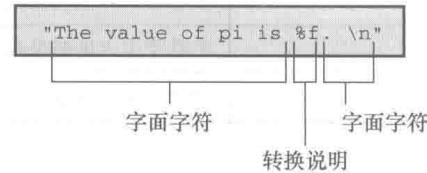


图 4.7 剖析格式字符串

## 警告

格式字符串中的转换说明一定要与后面的每个项相匹配，若忘记这个基本要求会导致严重的后果。  
千万别写成下面这样：

```
printf("The score was Squids %d, Slugs %d.\n", score1);
```

这里，第 2 个%d 没有对应任何项。系统不同，导致的结果也不同。不过，出现这种问题最好的状况是得到无意义的值。

如果只打印短语或句子，就不需要使用任何转换说明。如果只打印数据，也不用加入说明文字。程序清单 4.6 中的最后两个 printf() 语句都没问题：

```
printf("Farewell! thou art too dear for my possessing,\n");  
printf("%c%d\n", '$', 2 * cost);
```

注意第 2 条语句，待打印列表的第一个项是一个字符常量，不是变量；第 2 个项是一个乘法表达式。这说明 printf() 使用的是值，无论是变量、常量还是表达式的值。

由于 printf() 函数使用%符号来标识转换说明，因此打印%符号就成了个问题。如果单独使用一个% 符号，编译器会认为漏掉了一个转换字符。解决方法很简单，使用两个% 符号就行了：

```
pc = 2*6;  
printf("Only %d%% of Sally's gribbles were edible.\n", pc);
```

下面是输出结果：

```
Only 12% of Sally's gribbles were edible.
```

### 4.4.3 printf() 的转换说明修饰符

在%和转换字符之间插入修饰符可修饰基本的转换说明。表 4.4 和表 4.5 列出可作为修饰符的合法字符。如果要插入多个字符，其书写顺序应该与表 4.4 中列出的顺序相同。不是所有的组合都可行。表中有些字符是 C99 新增的，如果编译器不支持 C99，则可能不支持表中的所有项。

表 4.4 printf() 的修饰符

| 修饰符  | 含义                                                                                                                                                                                                               |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 标记   | 表 4.5 描述了 5 种标记 (-、+、空格、# 和 0)，可以不使用标记或使用多个标记<br>示例："%-10d"                                                                                                                                                      |
| 数字   | 最小字段宽度<br>如果该字段不能容纳待打印的数字或字符串，系统会使用更宽的字段<br>示例："%4d"                                                                                                                                                             |
| . 数字 | 精度<br>对于%e、%E 和%f 转换，表示小数点右边数字的位数<br>对于%g 和%G 转换，表示有效数字最大位数<br>对于%s 转换，表示待打印字符的最大数量<br>对于整型转换，表示待打印数字的最小位数<br>如有必要，使用前导 0 来达到这个位数<br>只使用. 表示其后跟随一个 0，所以%.f 和%.0f 相同<br>示例："%5.2f" 打印一个浮点数，字段宽度为 5 字符，其中小数点后有两位数字 |
| h    | 和整型转换说明一起使用，表示 short int 或 unsigned short int 类型的值<br>示例："%hu"、"%hx"、"%6.4hd"                                                                                                                                    |
| hh   | 和整型转换说明一起使用，表示 signed char 或 unsigned char 类型的值<br>示例："%hhu"、"%hhx"、"%6.4hhd"                                                                                                                                    |
| j    | 和整型转换说明一起使用，表示 intmax_t 或 uintmax_t 类型的值。这些类型定义在 stdint.h 中<br>示例："%jd"、"%8jx"                                                                                                                                   |
| l    | 和整型转换说明一起使用，表示 long int 或 unsigned long int 类型的值<br>示例："%ld"、"%8lu"                                                                                                                                              |
| ll   | 和整型转换说明一起使用，表示 long long int 或 unsigned long long int 类型的值 (C99)<br>示例："%lld"、"%8llu"                                                                                                                            |
| L    | 和浮点转换说明一起使用，表示 long double 类型的值<br>示例："%Ld"、"%10.4Le"                                                                                                                                                            |
| t    | 和整型转换说明一起使用，表示 ptrdiff_t 类型的值。ptrdiff_t 是两个指针差值的类型 (C99)<br>示例："%td"、"%12ti"                                                                                                                                     |
| z    | 和整型转换说明一起使用，表示 size_t 类型的值。size_t 是 sizeof 返回的类型 (C99)<br>示例："%zd"、"%12zd"                                                                                                                                       |

### 注意 类型可移植性

`sizeof` 运算符以字节为单位返回类型或值的大小。这应该是某种形式的整数，但是标准只规定了该值是无符号整数。在不同的实现中，它可以是 `unsigned int`、`unsigned long` 甚至是 `unsigned long long`。因此，如果要用 `printf()` 函数显示 `sizeof` 表达式，根据不同系统，可能使用 `%u`、`%lu` 或 `%llu`。这意味着要查找你当前系统的用法，如果把程序移植到不同的系统还要进行修改。鉴于此，C 提供了可移植性更好的类型。首先，`stddef.h` 头文件（在包含 `stdio.h` 头文件时已包含其中）把 `size_t` 定义成系统使用 `sizeof` 返回的类型，这被称为底层类型（*underlying type*）。其次，`printf()` 使用 `z` 修饰符表示打印相应的类型。同样，C 还定义了 `ptrdiff_t` 类型和 `t` 修饰符来表示系统使用的两个地址差值的底层有符号整数类型。

### 注意 float 参数的转换

对于浮点类型，有用于 `double` 和 `long double` 类型的转换说明，却没有 `float` 类型的。这是因为在 K&R C 中，表达式或参数中的 `float` 类型值会被自动转换成 `double` 类型。一般而言，ANSI C 不会把 `float` 自动转换成 `double`。然而，为保护大量假设 `float` 类型的参数被自动转换成 `double` 的现有程序，`printf()` 函数中所有 `float` 类型的参数（对未使用显式原型的所有 C 函数都有效）仍自动转换成 `double` 类型。因此，无论是 K&R C 还是 ANSI C，都没有显示 `float` 类型值专用的转换说明。

表 4.5 `printf()` 中的标记

| 标记 | 含义                                                                                                                                                                                                                                                                                                                           |
|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -  | 待打印项左对齐。即，从字段的左侧开始打印该项<br>示例：" <code>%-20s</code> "                                                                                                                                                                                                                                                                          |
| +  | 有符号值若为正，则在值前面显示加号；若为负，则在值前面显示减号<br>示例： <code>"%+6.2f"</code>                                                                                                                                                                                                                                                                 |
| 空格 | 有符号值若为正，则在值前面显示前导空格（不显示任何符号）；若为负，则在值前面显示减号<br>+标记覆盖一个空格<br>示例： <code>"% 6.2f"</code>                                                                                                                                                                                                                                         |
| #  | 把结果转换为另一种形式。如果是 <code>%o</code> 格式，则以 <code>0</code> 开始；如果是 <code>%x</code> 或 <code>%X</code> 格式，则以 <code>0x</code> 或 <code>0X</code> 开始；对于所有的浮点格式，#保证了即使后面没有任何数字，也打印一个小数点字符。对于 <code>%g</code> 和 <code>%G</code> 格式，#防止结果后面的 <code>0</code> 被删除<br>示例： <code>"%#o"</code> 、 <code>"%#8.0f"</code> 、 <code>"%+#10.3e"</code> |
| 0  | 对于数值格式，用前导 <code>0</code> 代替空格填充字段宽度。对于整数格式，如果出现 - 标记或指定精度，则忽略该标记                                                                                                                                                                                                                                                            |

### 1. 使用修饰符和标记的示例

接下来，用程序示例演示如何使用这些修饰符和标记。先来看看字段宽度在打印整数时的效果。考虑程序清单 4.7 中的程序。

#### 程序清单 4.7 width.c 程序

```
/* width.c -- 字段宽度 */
#include <stdio.h>
```

```
#define PAGES 959
int main(void)
{
    printf("*%d\n", PAGES);
    printf("*%2d\n", PAGES);
    printf("*%10d\n", PAGES);
    printf("*%-10d\n", PAGES);

    return 0;
}
```

程序清单 4.7 通过 4 种不同的转换说明把相同的值打印了 4 次。程序中使用星号 (\*) 标出每个字段的开始和结束。其输出结果如下所示：

```
*959*
*959*
*      959*
*959      *
```

第 1 个转换说明 %d 不带任何修饰符，其对应的输出结果与带整数段宽度的转换说明的输出结果相同。在默认情况下，没有任何修饰符的转换说明，就是这样的打印结果。第 2 个转换说明是 %2d，其对应的输出结果应该是 2 字段宽度。因为待打印的整数有 3 位数字，所以字段宽度自动扩大以符合整数的长度。第 3 个转换说明是 %10d，其对应的输出结果有 10 个空格宽度，实际上在两个星号之间有 7 个空格和 3 位数字，并且数字位于字段的右侧。最后一个转换说明是 %-10d，其对应的输出结果同样是 10 个空格宽度，- 标记说明打印的数字位于字段的左侧。熟悉它们的用法后，能很好地控制输出格式。试着改变 PAGES 的值，看看编译器如何打印不同位数的数字。

接下来看看浮点型格式。请输入、编译并运行程序清单 4.8 中的程序。

#### 程序清单 4.8 floats.c 程序

```
// floats.c -- 一些浮点型修饰符的组合
#include <stdio.h>

int main(void)
{
    const double RENT = 3852.99; // const 变量

    printf("*%f\n", RENT);
    printf("*%e\n", RENT);
    printf("*%4.2f\n", RENT);
    printf("*%3.1f\n", RENT);
    printf("*%10.3f\n", RENT);
    printf("*%10.3E\n", RENT);
    printf("*%+4.2f\n", RENT);
    printf("*%010.2f\n", RENT);

    return 0;
}
```

该程序中使用了 const 关键字，限定变量为只读。该程序的输出如下：

```
*3852.990000*
*3.852990e+03*
```

```
*3852.99*
*3853.0*
* 3852.990*
* 3.853E+03*
*+3852.99*
*0003852.99*
```

本例的第一个转换说明是%f。在这种情况下，字段宽度和小数点后面的位数均为系统默认设置，即字段宽度是容纳带打印数字所需的位数和小数点后打印6位数字。

第二个转换说明是%e。默认情况下，编译器在小数点的左侧打印1个数字，在小数点的右侧打印6个数字。这样打印的数字太多！解决方案是指定小数点右侧显示的位数，程序中接下来的4个例子就是这样做的。请注意，第4个和第6个例子对输出结果进行了四舍五入。另外，第6个例子用E代替了e。

第七个转换说明中包含了+标记，这使得打印的值前面多了一个代数符号(+)。0标记使得打印的值前面以0填充以满足字段要求。注意，转换说明%010.2f的第一个0是标记，句点(.)之前、标记之后的数字(本例为10)是指定的字段宽度。

尝试修改RENT的值，看看编译器如何打印不同大小的值。程序清单4.9演示了其他组合。

#### 程序清单4.9 flags.c程序

---

```
/* flags.c -- 演示一些格式标记 */
#include <stdio.h>
int main(void)
{
    printf("%x %X %#x\n", 31, 31, 31);
    printf("**%d*** d**% d**\n", 42, 42, -42);
    printf("***%5d**%5.3d**%05d***%05.3d**\n", 6, 6, 6, 6);

    return 0;
}
```

---

该程序的输出如下：

```
1f 1F 0x1f
**42** 42***42**
**      6** 006***00006** 006**
```

第1行输出中，1f是十六进制数，等于十进制数31。第1行printf()语句中，根据%x打印出1f，%F打印出1F，%#x打印出0x1f。

第2行输出演示了如何在转换说明中用空格在输出的正值前面生成前导空格，负值前面不产生前导空格。这样的输出结果比较美观，因为打印出来的正值和负值在相同字段宽度下的有效数字位数相同。

第3行输出演示了如何在整型格式中使用精度(%5.3d)生成足够的前导0以满足最小位数的要求(本例是3)。然而，使用0标记会使得编译器用前导0填充满整个字段宽度。最后，如果0标记和精度一起出现，0标记会被忽略。

下面来看看字符串格式的示例。考虑程序清单4.10中的程序。

#### 程序清单4.10 stringf.c程序

---

```
/* stringf.c -- 字符串格式 */
#include <stdio.h>
#define BLURB "Authentic imitation!"
int main(void)
```

```

{
    printf("[%2s]\n", BLURB);
    printf("[%24s]\n", BLURB);
    printf("[%24.5s]\n", BLURB);
    printf("[%-.24.5s]\n", BLURB);

    return 0;
}

```

该程序的输出如下：

```

[Authentic imitation!]
[      Authentic imitation!]
[          Authe]
[Authe           ]

```

注意，虽然第 1 个转换说明是 %2s，但是字段被扩大为可容纳字符串中的所有字符。还需注意，精度限制了待打印字符的个数。.5 告诉 printf() 只打印 5 个字符。另外，- 标记使得文本左对齐输出。

## 2. 学以致用

学习完以上几个示例，试试如何用一个语句打印以下格式的内容：

```
The NAME family just may be $XXX.XX dollars richer!
```

这里，NAME 和 XXX.XX 代表程序中变量（如 name[40] 和 cash）的值。可参考以下代码：

```
printf("The %s family just may be $%.2f richer!\n", name, cash);
```

### 4.4.4 转换说明的意义

下面深入探讨一下转换说明的意义。转换说明把以二进制格式储存在计算机中的值转换成一系列字符（字符串）以便于显示。例如，数字 76 在计算机内部的存储格式是二进制数 01001100。%d 转换说明将其转换成字符 7 和 6，并显示为 76；%x 转换说明把相同的值（01001100）转换成十六进制记数法 4c；%c 转换说明把 01001100 转换成字符 L。

转换(*conversion*)可能会误导读者认为原始值被转替换成转换后的值。实际上，转换说明是翻译说明，%d 的意思是“把给定的值翻译成十进制整数文本并打印出来”。

#### 1. 转换不匹配

前面强调过，转换说明应该与待打印值的类型相匹配。通常都有多种选择。例如，如果要打印一个 int 类型的值，可以使用%d、%x 或%o。这些转换说明都可用于打印 int 类型的值，其区别在于它们分别表示一个值的形式不同。类似地，打印 double 类型的值时，可使用%f、%e 或%g。

转换说明与待打印值的类型不匹配会怎样？上一章中介绍过不匹配导致的一些问题。匹配非常重要，一定要牢记于心。程序清单 4.11 演示了一些不匹配的整型转换示例。

**程序清单 4.11 intconv.c 程序**

---

```

/* intconv.c -- 一些不匹配的整型转换 */
#include <stdio.h>
#define PAGES 336
#define WORDS 65618
int main(void)
{
    short num = PAGES;
    short mnum = -PAGES;

```

```

printf("num as short and unsigned short: %hd %hu\n", num, num);
printf("-num as short and unsigned short: %hd %hu\n", mnum, mnum);
printf("num as int and char: %d %c\n", num, num);
printf("WORDS as int, short, and char: %d %hd %c\n", WORDS, WORDS, WORDS);

return 0;
}

```

在我们的系统中，该程序的输出如下：

```

num as short and unsigned short: 336 336
-num as short and unsigned short: -336 65200
num as int and char: 336 P
WORDS as int, short, and char: 65618 82 R

```

请看输出的第1行，`num`变量对应的转换说明`%hd`和`%hu`输出的结果都是336。这没有任何问题。然而，第2行`mnum`变量对应的转换说明`%u`（无符号）输出的结果却为65200，并非期望的336。这是由于有符号`short int`类型的值在我们的参考系统中的表示方式所致。首先，`short int`的大小是2字节；其次，系统使用二进制补码来表示有符号整数。这种方法，数字0~32767代表它们本身，而数字32768~65535则表示负数。其中，65535表示-1，65534表示-2，以此类推。因此，-336表示为65200（即，65536-336）。所以被解释成有符号`int`时，65200代表-336；而被解释成无符号`int`时，65200则代表65200。一定要谨慎！一个数字可以被解释成两个不同的值。尽管并非所有的系统都使用这种方法来表示负整数，但要注意一点：别期望用`%u`转换说明能把数字和符号分开。

第3行演示了如果把一个大于255的值转换成字符会发生什么情况。在我们的系统中，`short int`是2字节，`char`是1字节。当`printf()`使用`%c`打印336时，它只会查看储存336的2字节中的后1字节。这种截断（见图4.8）相当于用一个整数除以256，只保留其余数。在这种情况下，余数是80，对应的ASCII值是字符P。用专业术语来说，该数字被解释成“以256为模”（modulo 256），即该数字除以256后取其余数。

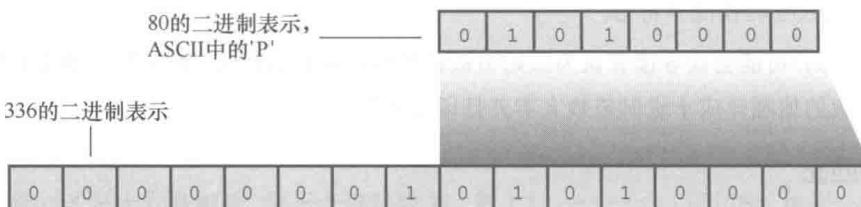


图4.8 把336转换成字符

最后，我们在该系统中打印比`short int`类型最大整数（32767）更大的整数（65618）。这次，计算机也进行了求模运算。在本系统中，应把数字65618储存为4字节的`int`类型值。用`%hd`转换说明打印时，`printf()`只使用最后2个字节。这相当于65618除以65536的余数。这里，余数是82。鉴于负数的储存方法，如果余数在32768~65536范围内会被打印成负数。对于整数大小不同的系统，相应的处理行为类似，但是产生的值可能不同。

混淆整型和浮点型，结果更奇怪。考虑程序清单4.12。

#### 程序清单4.12 floatcnv.c程序

```

/* floatcnv.c -- 不匹配的浮点型转换 */
#include <stdio.h>
int main(void)
{

```

```

float n1 = 3.0;
double n2 = 3.0;
long n3 = 2000000000;
long n4 = 1234567890;

printf("%.1e %.1e %.1e %.1e\n", n1, n2, n3, n4);
printf("%ld %ld\n", n3, n4);
printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);

return 0;
}

```

在我们的系统中，该程序的输出如下：

```

3.0e+00 3.0e+00 3.1e+46 1.7e+266
2000000000 1234567890
0 1074266112 0 1074266112

```

第1行输出显示，%e转换说明没有把整数转换成浮点数。考虑一下，如果使用%e转换说明打印n3(long类型)会发生什么情况。首先，%e转换说明让printf()函数认为待打印的值是double类型(本系统中double为8字节)。当printf()查看n3(本系统中是4字节的值)时，除了查看n3的4字节外，还会查看n3相邻的4字节，共8字节单元。接着，它将8字节单元中的位组合解释成浮点数(如，把一部分位组合解释成指数)。因此，即使n3的位数正确，根据%e转换说明和%ld转换说明解释出来的值也不同。最终得到的结果是无意义的值。

第1行也说明了前面提到的内容：float类型的值作为printf()参数时会被转换成double类型。在本系统中，float是4字节，但是为了printf()能正确地显示该值，n1被扩成8字节。

第2行输出显示，只要使用正确的转换说明，printf()就可以打印n3和n4。

第3行输出显示，如果printf()语句有其他不匹配的地方，即使用对了转换说明也会生成虚假的结果。用%ld转换说明打印浮点数会失败，但是在这里，用%ld打印long类型的数竟然也失败了！问题出在C如何把信息传递给函数。具体情况因编译器实现而异。“参数传递”框中针对一个有代表性的系统进行了讨论。

## 参数传递

参数传递机制因实现而异。下面以我们的系统为例，分析参数传递的原理。函数调用如下：

```
printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);
```

该调用告诉计算机把变量n1、n2、n3和n4的值传递给程序。这是一种常见的参数传递方式。程序把传入的值放入被称为栈(stack)的内存区域。计算机根据变量类型(不是根据转换说明)把这些值放入栈中。因此，n1被储存在栈中，占8字节(float类型被转换成double类型)。同样，n2也在栈中占8字节，而n3和n4在栈中分别占4字节。然后，控制转到printf()函数。该函数根据转换说明(不是根据变量类型)从栈中读取值。%ld转换说明表明printf()应该读取4字节，所以printf()读取栈中的前4字节作为第1个值。这是n1的前半部分，将被解释成一个long类型的整数。根据下一个%ld转换说明，printf()再读取4字节，这是n1的后半部分，将被解释成第2个long类型的整数(见图4.9)。类似地，根据第3个和第4个%ld，printf()读取n2的前半部分和后半部分，并解释成两个long类型的整数。因此，对于n3和n4，虽然用对了转换说明，但printf()还是读错了字节。

```

float n1; /* 作为 double 类型传递 */
double n2;
long n3, n4;

```

```
...
printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);
```

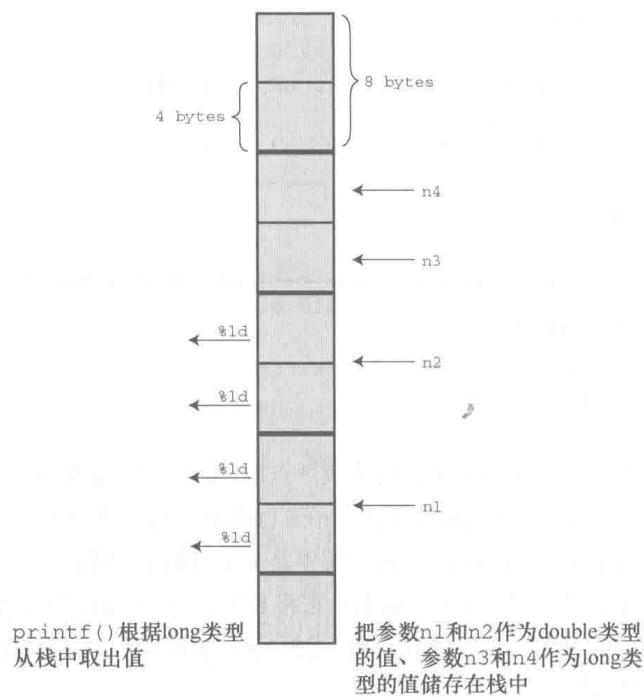


图 4.9 传递参数

## 2. printf()的返回值

第2章提到过，大部分C函数都有一个返回值，这是函数计算并返回给主调程序（*calling program*）的值。例如，C库包含一个sqrt()函数，接受一个数作为参数，并返回该数的平方根。可以把返回值赋给变量，也可以用于计算，还可以作为参数传递。总之，可以把返回值像其他值一样使用。printf()函数也有一个返回值，它返回打印字符的个数。如果有输出错误，printf()则返回一个负值（printf()的旧版本会返回不同的值）。

printf()的返回值是其打印输出功能的附带用途，通常很少用到，但在检查输出错误时可能会用到（如，在写入文件时很常用）。如果一张已满的CD或DVD拒绝写入时，程序应该采取相应的行动，例如终端蜂鸣30秒。不过，要实现这种情况必须先了解if语句。程序清单4.13演示了如何确定函数的返回值。

程序清单4.13 prntval.c程序

```
/* prntval.c -- printf() 的返回值 */
#include <stdio.h>
int main(void)
{
    int bph2o = 212;
    int rv;

    rv = printf("%d F is water's boiling point.\n", bph2o);
    printf("The printf() function printed %d characters.\n",
           rv);
    return 0;
}
```

该程序的输出如下：

```
212 F is water's boiling point.  
The printf() function printed 32 characters.
```

首先，程序用 `rv = printf(...);` 的形式把 `printf()` 的返回值赋给 `rv`。因此，该语句执行了两项任务：打印信息和给变量赋值。其次，注意计算针对所有字符数，包括空格和不可见的换行符 (`\n`)。

### 3. 打印较长的字符串

有时，`printf()` 语句太长，在屏幕上不方便阅读。如果空白（空格、制表符、换行符）仅用于分隔不同的部分，C 编译器会忽略它们。因此，一条语句可以写成多行，只需在不同部分之间输入空白即可。例如，程序清单 4.13 中的一条 `printf()` 语句：

```
printf("The printf() function printed %d characters.\n",
       rv);
```

该语句在逗号和 `rv` 之间断行。为了让读者知道该行未完，示例缩进了 `rv`。C 编译器会忽略多余的空白。

但是，不能在双引号括起来的字符串中间断行。如果这样写：

```
printf("The printf() function printed %d
       characters.\n", rv);
```

C 编译器会报错：字符串常量中有非法字符。在字符串中，可以使用 `\n` 来表示换行字符，但是不能通过按下 **Enter**（或 **Return**）键产生实际的换行符。

给字符串断行有 3 种方法，如程序清单 4.14 所示。

程序清单 4.14 longstrg.c 程序

---

```
/* longstrg.c --打印较长的字符串 */
#include <stdio.h>
int main(void)
{
    printf("Here's one way to print a ");
    printf("long string.\n");
    printf("Here's another way to print a \
long string.\n");
    printf("Here's the newest way to print a "
           "long string.\n"); /* ANSI C */

    return 0;
}
```

---

该程序的输出如下：

```
Here's one way to print a long string.  
Here's another way to print a long string.  
Here's the newest way to print a long string.
```

方法 1：使用多个 `printf()` 语句。因为第 1 个字符串没有以 `\n` 字符结束，所以第 2 个字符串紧跟第 1 个字符串末尾输出。

方法 2：用反斜杠 (`\`) 和 **Enter**（或 **Return**）键组合来断行。这使得光标移至下一行，而且字符串中不会包含换行符。其效果是在下一行继续输出。但是，下一行代码必须和程序清单中的代码一样从最左边开始。如果缩进该行，比如缩进 5 个空格，那么这 5 个空格就会成为字符串的一部分。

方法 3：ANSI C 引入的字符串连接。在两个用双引号括起来的字符串之间用空白隔开，C 编译器会把

多个字符串看作是一个字符串。因此，以下3种形式是等效的：

```
printf("Hello, young lovers, wherever you are.");
printf("Hello, young "      "lovers" " ", wherever you are.");
printf("Hello, young lovers"
      ", wherever you are.");
```

上述方法中，要记得在字符串中包含所需的空格。如，“young”“lovers”会成为“younglovers”，而“young ” “lovers”才是“young lovers”。

#### 4.4.5 使用 scanf()

刚学完输出，接下来我们转至输入——学习 `scanf()` 函数。C 库包含了多个输入函数，`scanf()` 是最通用的一个，因为它可以读取不同格式的数据。当然，从键盘输入的都是文本，因为键盘只能生成文本字符：字母、数字和标点符号。如果要输入整数 2014，就要键入字符 2、0、1、4。如果要将其储存为数值而不是字符串，程序就必须把字符依次转换成数值，这就是 `scanf()` 要做的。`scanf()` 把输入的字符串转换成整数、浮点数、字符或字符串，而 `printf()` 正好与它相反，把整数、浮点数、字符和字符串转换成显示在屏幕上的文本。

`scanf()` 和 `printf()` 类似，也使用格式字符串和参数列表。`scanf()` 中的格式字符串表明字符输入流的目标数据类型。两个函数主要的区别在参数列表中。`printf()` 函数使用变量、常量和表达式，而 `scanf()` 函数使用指向变量的指针。这里，读者不必了解如何使用指针，只需记住以下两条简单的规则：

- 如果用 `scanf()` 读取基本变量类型的值，在变量名前加上一个`&`；
- 如果用 `scanf()` 把字符串读入字符数组中，不要使用`&`。

程序清单 4.15 中的小程序演示了这两条规则。

程序清单 4.15 input.c 程序

---

```
// input.c -- 何时使用&
#include <stdio.h>
int main(void)
{
    int age;           // 变量
    float assets;     // 变量
    char pet[30];     // 字符数组，用于储存字符串

    printf("Enter your age, assets, and favorite pet.\n");
    scanf("%d %f", &age, &assets); // 这里要使用&
    scanf("%s", pet);           // 字符数组不使用&
    printf("%d %.2f %s\n", age, assets, pet);

    return 0;
}
```

---

下面是该程序与用户交互的示例：

```
Enter your age, assets, and favorite pet.
```

```
38
```

```
92360.88 llama
```

```
38 $92360.88 llama
```

`scanf()` 函数使用空白（换行符、制表符和空格）把输入分成多个字段。在依次把转换说明和字段匹

配时跳过空白。注意，上面示例的输入项（粗体部分是用户的输入）分成了两行。只要在每个输入项之间输入至少一个换行符、空格或制表符即可，可以在一行或多行输入：

```
Enter your age, assets, and favorite pet.
```

42

2121.45

**guppy**

42 \$2121.45 guppy

唯一例外的是%c 转换说明。根据%c, scanf()会读取每个字符，包括空白。我们稍后详述这部分。

scanf() 函数所用的转换说明与 printf() 函数几乎相同。主要的区别是，对于 float 类型和 double 类型，printf() 都使用%f、%e、%E、%g 和%G 转换说明。而 scanf() 只把它们用于 float 类型，对于 double 类型时要使用l修饰符。表 4.6 列出了 C99 标准中常用的转换说明。

表 4.6 ANSI C 中 scanf() 的转换说明

| 转换说明        | 含义                                          |
|-------------|---------------------------------------------|
| %c          | 把输入解释成字符                                    |
| %d          | 把输入解释成有符号十进制整数                              |
| %e、%f、%g、%a | 把输入解释成浮点数 (C99 标准新增了%a)                     |
| %E、%F、%G、%A | 把输入解释成浮点数 (C99 标准新增了%A)                     |
| %i          | 把输入解释成有符号十进制整数                              |
| %o          | 把输入解释成有符号八进制整数                              |
| %p          | 把输入解释成指针 (地址)                               |
| %s          | 把输入解释成字符串。从第 1 个非空白字符开始，到下一个空白字符之前的所有字符都是输入 |
| %u          | 把输入解释成无符号十进制整数                              |
| %x、%X       | 把输入解释成有符号十六进制整数                             |

可以在表 4.6 所列的转换说明中（百分号和转换字符之间）使用修饰符。如果要使用多个修饰符，必须按表 4.7 所列的顺序书写。

表 4.7 scanf() 转换说明中的修饰符

| 转换说明 | 含义                                                                |
|------|-------------------------------------------------------------------|
| *    | 抑制赋值（详见后面解释）<br>示例："%*d"                                          |
| 数字   | 最大字段宽度。输入达到最大字段宽度处，或第 1 次遇到空白字符时停止<br>示例："%10s"                   |
| hh   | 把整数作为 signed char 或 unsigned char 类型读取<br>示例："%hd"、"%hu"          |
| ll   | 把整数作为 long long 或 unsigned long long 类型读取 (C99)<br>示例："lld"、"llu" |

续表

| 转换说明    | 含义                                                                                                                                                                                                                                                                                                                                                      |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| h、l 或 L | "%hd" 和 "%hi" 表明把对应的值储存为 short int 类型<br>"%ho"、"%hx" 和 "%hu" 表明把对应的值储存为 unsigned short int 类型<br>"%ld" 和 "%li" 表明把对应的值储存为 long 类型<br>"%lo"、"%lx" 和 "%lu" 表明把对应的值储存为 unsigned long 类型<br>"%le"、"%lf" 和 "%lg" 表明把对应的值储存为 double 类型<br>在 e、f 和 g 前面使用 L 而不是 l，表明把对应的值被储存为 long double 类型。<br>如果没有修饰符，d、i、o 和 x 表明对应的值被储存为 int 类型，f 和 g 表明把对应的值储存为 float 类型 |
| j       | 在整型转换说明后面时，表明使用 intmax_t 或 uintmax_t 类型 (C99)<br>示例："%zd"、"%zo"                                                                                                                                                                                                                                                                                         |
| z       | 在整型转换说明后面时，表明使用 sizeof 的返回类型 (C99)                                                                                                                                                                                                                                                                                                                      |
| t       | 在整型转换说明后面时，表明使用表示两个指针差值的类型 (C99)<br>示例："%td"、"%tx"                                                                                                                                                                                                                                                                                                      |

如你所见，使用转换说明比较复杂，而且这些表中还省略了一些特性。省略的主要特性是，从高度格式化源中读取选定数据，如穿孔卡或其他数据记录。因为在本书中，`scanf()` 主要作为与程序交互的便利工具，所以我们不在书中讨论更复杂的特性。

## 1. 从 `scanf()` 角度看输入

接下来，我们更详细地研究 `scanf()` 怎样读取输入。假设 `scanf()` 根据一个%a 转换说明读取一个整数。`scanf()` 函数每次读取一个字符，跳过所有的空白字符，直至遇到第 1 个非空白字符才开始读取。因为要读取整数，所以 `scanf()` 希望发现一个数字字符或者一个符号 (+或-)。如果找到一个数字或符号，它便保存该字符，并读取下一个字符。如果下一个字符是数字，它便保存该数字并读取下一个字符。`scanf()` 不断地读取和保存字符，直至遇到非数字字符。如果遇到一个非数字字符，它便认为读到了整数的末尾。然后，`scanf()` 把非数字字符放回输入。这意味着程序在下一次读取输入时，首先读到的是上一次读取丢弃的非数字字符。最后，`scanf()` 计算已读取数字（可能还有符号）相应的数值，并将计算后的值放入指定的变量中。

如果使用字段宽度，`scanf()` 会在字段结尾或第 1 个空白字符处停止读取（满足两个条件之一便停止）。

如果第 1 个非空白字符是 A 而不是数字，会发生什么情况？`scanf()` 将停在那里，并把 A 放回输入中，不会把值赋给指定变量。程序在下一次读取输入时，首先读到的字符是 A。如果程序只使用%d 转换说明，`scanf()` 就一直无法越过 A 读下一个字符。另外，如果使用带多个转换说明的 `scanf()`，C 规定在第 1 个出错处停止读取输入。

用其他数值匹配的转换说明读取输入和用%d 的情况相同。区别在于 `scanf()` 会把更多字符识别成数字的一部分。例如，%x 转换说明要求 `scanf()` 识别十六进制数 a~f 和 A~F。浮点转换说明要求 `scanf()` 识别小数点、e 记数法（指数记数法）和新增的 p 记数法（十六进制指数记数法）。

如果使用%s 转换说明，`scanf()` 会读取除空白以外的所有字符。`scanf()` 跳过空白开始读取第 1 个非空白字符，并保存非空白字符直到再次遇到空白。这意味着 `scanf()` 根据%s 转换说明读取一个单词，即不包含空白字符的字符串。如果使用字段宽度，`scanf()` 在字段末尾或第 1 个空白字符处停止读取。无法利用字段宽度让只有一个%s 的 `scanf()` 读取多个单词。最后要注意一点：当 `scanf()` 把字符串放进指定数组中时，它会在字符串的末尾加上'\0'，让数组中的内容成为一个 C 字符串。

实际上，在 C 语言中 `scanf()` 并不是最常用的输入函数。这里重点介绍它是因为它能读取不同类型的

数据。C 语言还有其他的输入函数，如 `getchar()` 和 `fgets()`。这两个函数更适合处理一些特殊情况，如读取单个字符或包含空格的字符串。我们将在第 7 章、第 11 章、第 13 章中讨论这些函数。目前，无论程序中需要读取整数、小数、字符还是字符串，都可以使用 `scanf()` 函数。

## 2. 格式字符串中的普通字符

`scanf()` 函数允许把普通字符放在格式字符串中。除空格字符外的普通字符必须与输入字符串严格匹配。例如，假设在两个转换说明中添加一个逗号：

```
scanf("%d,%d", &n, &m);
```

`scanf()` 函数将其解释成：用户将输入一个数字、一个逗号，然后再输入一个数字。也就是说，用户必须像下面这样进行输入两个整数：

```
88,121
```

由于格式字符串中，`%d` 后面紧跟逗号，所以必须在输入 88 后再输入一个逗号。但是，由于 `scanf()` 会跳过整数前面的空白，所以下面两种输入方式都可以：

```
88, 121
```

和

```
88,  
121
```

格式字符串中的空白意味着跳过下一个输入项前面的所有空白。例如，对于下面的语句：

```
scanf("%d , %d", &n, &m);
```

以下的输入格式都没问题：

```
88,121
```

```
88 ,121
```

```
88 , 121
```

请注意，“所有空白”的概念包括没有空格的特殊情况。

除了`%c`，其他转换说明都会自动跳过待输入值前面所有的空白。因此，`scanf("%d%d", &n, &m)` 与 `scanf("%d %d", &n, &m)` 的行为相同。对于`%c`，在格式字符串中添加一个空格字符会有所不同。例如，如果把`%c` 放在格式字符串中的空格前面，`scanf()` 便会跳过空格，从第 1 个非空白字符开始读取。也就是说，`scanf("%c", &ch)` 从输入中的第 1 个字符开始读取，而 `scanf(" %c", &ch)` 则从第 1 个非空白字符开始读取。

## 3. `scanf()` 的返回值

`scanf()` 函数返回成功读取的项数。如果没有读取任何项，且需要读取一个数字而用户却输入一个非数值字符串，`scanf()` 便返回 0。当 `scanf()` 检测到“文件结尾”时，会返回 EOF（EOF 是 `stdio.h` 中定义的特殊值，通常用`#define` 指令把 EOF 定义为 -1）。我们将在第 6 章中讨论文件结尾的相关内容以及如何利用 `scanf()` 的返回值。在读者学会 `if` 语句和 `while` 语句后，便可使用 `scanf()` 的返回值来检测和处理不匹配的输入。

## 4.4.6 printf() 和 scanf() 的\*修饰符

`printf()` 和 `scanf()` 都可以使用\*修饰符来修改转换说明的含义。但是，它们的用法不太一样。首先，我们来看 `printf()` 的\*修饰符。

如果你不想预先指定字段宽度，希望通过程序来指定，那么可以用\*修饰符代替字段宽度。但还是要用一个参数告诉函数，字段宽度应该是多少。也就是说，如果转换说明是`%*d`，那么参数列表中应包含\*和 d 对应的值。这个技巧也可用于浮点值指定精度和字段宽度。程序清单 4.16 演示了相关用法。

**程序清单 4.16 varwid.c 程序**


---

```
/* varwid.c -- 使用变宽输出字段 */
#include <stdio.h>
int main(void)
{
    unsigned width, precision;
    int number = 256;
    double weight = 242.5;

    printf("Enter a field width:\n");
    scanf("%d", &width);
    printf("The number is :%*d:\n", width, number);
    printf("Now enter a width and a precision:\n");
    scanf("%d %d", &width, &precision);
    printf("Weight = %.*f\n", width, precision, weight);
    printf("Done!\n");

    return 0;
}
```

---

变量 width 提供字段宽度, number 是待打印的数字。因为转换说明中\*在 d 的前面, 所以在 printf() 的参数列表中, width 在 number 的前面。同样, width 和 precision 提供打印 weight 的格式化信息。下面是一个运行示例:

```
Enter a field width:
6
The number is : 256:
Now enter a width and a precision:
8 3
Weight = 242.500
Done!
```

这里, 用户首先输入 6, 因此 6 是程序使用的字段宽度。类似地, 接下来用户输入 8 和 3, 说明字段宽度是 8, 小数点后面显示 3 位数字。一般而言, 程序应根据 weight 的值来决定这些变量的值。

scanf() 中\*的用法与此不同。把\*放在%和转换字符之间时, 会使得 scanf() 跳过相应的输出项。程序清单 4.17 就是一个例子。

**程序清单 4.17 skip2.c 程序**


---

```
/* skiptwo.c -- 跳过输入中的前两个整数 */
#include <stdio.h>
int main(void)
{
    int n;

    printf("Please enter three integers:\n");
    scanf("%*d %*d %d", &n);
    printf("The last integer was %d\n", n);

    return 0;
}
```

---

程序清单 4.17 中的 scanf() 指示: 跳过两个整数, 把第 3 个整数拷贝给 n。下面是一个运行示例:

```
Please enter three integers:
```

```
2013 2014 2015
```

```
The last integer was 2015
```

在程序需要读取文件中特定列的内容时，这项跳过功能很有用。

#### 4.4.7 printf() 的用法提示

想把数据打印成列，指定固定字段宽度很有用。因为默认的字段宽度是待打印数字的宽度，如果同一列中打印的数字位数不同，那么下面的语句：

```
printf("%d %d %d\n", val1, val2, val3);
```

打印出来的数字可能参差不齐。例如，假设执行 3 次 printf() 语句，用户输入不同的变量，其输出可能是这样：

```
12 234 1222
4 5 23
22334 2322 10001
```

使用足够大的固定字段宽度可以让输出整齐美观。例如，若使用下面的语句：

```
printf("%9d %9d %9d\n", val1, val2, val3);
```

上面的输出将变成：

```
12      234      1222
        4          5          23
22334    2322    10001
```

在两个转换说明中间插入一个空白字符，可以确保即使一个数字溢出了自己的字段，下一个数字也不会紧跟该数字一起输出（这样两个数字看起来像是一个数字）。这是因为格式字符串中的普通字符（包括空格）会被打印出来。

另一方面，如果要在文字中嵌入一个数字，通常指定一个小于或等于该数字宽度的字段会比较方便。这样，输出数字的宽度正合适，没有不必要的空白。例如，下面的语句：

```
printf("Count Beppo ran %.2f miles in 3 hours.\n", distance);
```

其输出如下：

```
Count Beppo ran 10.22 miles in 3 hours.
```

如果把转换说明改为%10.2f，则输出如下：

```
Count Beppo ran      10.22 miles in 3 hours.
```

#### 本地化设置

美国和世界上的许多地区都使用一个点来分隔十进制值的整数部分和小数部分，如 3.14159。然而，许多其他地区用逗号来分隔，如 3,14159。读者可能注意到了，printf() 和 scanf() 都没有提供逗号的转换说明。C 语言考虑了这种情况。本书附录 B 的参考资料 V 中介绍了 C 支持的本地化概念，因此 C 程序可以选择特定的本地化设置。例如，如果指定了荷兰语言环境，printf() 和 scanf() 在显示和读取浮点值时会使用本地惯例（在这种情况下，用逗号代替点分隔浮点值的整数部分和小数部分）。另外，一旦指定了环境，便可在代码的数字中使用逗号：

```
double pi = 3,14159; // 荷兰本地化设置
```

C 标准有两个本地化设置："C" 和 ""（空字符串）。默认情况下，程序使用"C"本地化设置，基本上符合美国的用法习惯。而""本地化设置可以替换当前系统中使用的本地语言环境。原则上，这与"C"本地化设置相同。事实上，大部分操作系统（如 UNIX、Linux 和 Windows）都提供本地化设置选项列表，只不过它们提供的列表可能不同。

## 4.5 关键概念

C语言用char类型表示单个字符，用字符串表示字符序列。字符常量是一种字符串形式，即用双引号把字符括起来：“Good luck, my friend”。可以把字符串储存在字符数组（由内存中相邻的字节组成）中。字符串，无论是表示成字符常量还是储存在字符数组中，都以一个叫做空字符的隐藏字符结尾。

在程序中，最好用#define定义数值常量，用const关键字声明的变量为只读变量。在程序中使用符号常量（明示常量），提高了程序的可读性和可维护性。

C语言的标准输入函数（scanf()）和标准输出函数（printf()）都使用一种系统。在该系统中，第1个参数中的转换说明必须与后续参数中的值相匹配。例如，int转换说明%d与一个浮点值匹配会产生奇怪的结果。必须格外小心，确保转换说明的数量和类型与函数的其余参数相匹配。对于scanf()，一定要记得在变量名前加上地址运算符（&）。

空白字符（制表符、空格和换行符）在scanf()处理输入时起着至关重要的作用。除了%c模式（读取下一个字符），scanf()在读取输入时会跳过非空白字符前的所有空白字符，然后一直读取字符，直至遇到空白字符或与正在读取字符不匹配的字符。考虑一下，如果scanf()根据不同的转换说明读取相同的输入行，会发生什么情况。假设有如下输入行：

```
-13.45e12# 0
```

如果其对应的转换说明是%d，scanf()会读取3个字符（-13）并停在小数点处，小数点将被留在输入中作为下一次输入的首字符。如果其对应的转换说明是%f，scanf()会读取-13.45e12，并停在#符号处，而#将被留在输入中作为下一次输入的首字符；然后，scanf()把读取的字符序列-13.45e12转换成相应的浮点值，并储存在float类型的目标变量中。如果其对应的转换说明是%s，scanf()会读取-13.45e12#，并停在空格处，空格将被留在输入中作为下一次输入的首字符；然后，scanf()把这10个字符的字符码储存在目标字符数组中，并在末尾加上一个空字符。如果其对应的转换说明是%c，scanf()只会读取并储存第1个字符，该例中是一个空格<sup>1</sup>。

## 4.6 本章小结

字符串是一系列被视为一个处理单元的字符。在C语言中，字符串是以空字符（ASCII码是0）结尾的一系列字符。可以把字符串储存在字符数组中。数组是一系列同类型的项或元素。下面声明了一个名为name、有30个char类型元素的数组：

```
char name[30];
```

要确保有足够的元素来储存整个字符串（包括空字符）。

字符串常量是用双引号括起来的字符序列，如：“This is an example of a string”。

scanf()函数（声明在string.h头文件中）可用于获得字符串的长度（末尾的空字符不计算在内）。scanf()函数中的转换说明是%s时，可读取一个单词。

C预处理器为预处理器指令（以#符号开始）查找源代码程序，并在开始编译程序之前处理它们。处理器根据#include指令把另一个文件中的内容添加到该指令所在的位置。#define指令可以创建明示常量（符号常量），即代表常量的符号。limits.h和float.h头文件用#define定义了一组表示整型和浮点型不同属性的符号常量。另外，还可以使用const限定符创建定义后就不能修改的变量。

<sup>1</sup> 注意，“-13.45e12# 0”的负号前面有一个空格。——译者注

`printf()` 和 `scanf()` 函数对输入和输出提供多种支持。两个函数都使用格式字符串，其中包含的转换说明表明待读取或待打印数据项的数量和类型。另外，可以使用转换说明控制输出的外观：字段宽度、小数位和字段内的布局。

## 4.7 复习题

复习题的参考答案在附录 A 中。

1. 再次运行程序清单 4.1，但是在要求输入名时，请输入名和姓（根据英文书写习惯，名和姓中间有一个空格），看看会发生什么情况？为什么？

2. 假设下列示例都是完整程序中的一部分，它们打印的结果分别是什么？

- a. `printf("He sold the painting for $%2.2f.\n", 2.345e2);`
- b. `printf("%c%c%c\n", 'H', 105, '\u0411');`
- c. `#define Q "His Hamlet was funny without being vulgar."`  
`printf("%s\nhas %d characters.\n", Q, strlen(Q));`
- d. `printf("Is %2.2e the same as %2.2f?\n", 1201.0, 1201.0);`

3. 在第 2 题的 c 中，要输出包含双引号的字符串 Q，应如何修改？

4. 找出下面程序中的错误。

```
define B booboo
define X 10
main(int)
{
    int age;
    char name;
    printf("Please enter your first name.");
    scanf("%s", name);
    printf("All right, %c, what's your age?\n", name);
    scanf("%f", age);
    xp = age + X;
    printf("That's a %s! You must be at least %d.\n", B, xp);
    rerun 0;
}
```

5. 假设一个程序的开头是这样：

```
#define BOOK "War and Peace"
int main(void)
{
    float cost = 12.99;
    float percent = 80.0;
```

请构造一个使用 `BOOK`、`cost` 和 `percent` 的 `printf()` 语句，打印以下内容：

```
This copy of "War and Peace" sells for $12.99.
That is 80% of list.
```

6. 打印下列各项内容要分别使用什么转换说明？

- a. 一个字段宽度与位数相同的十进制整数
- b. 一个形如 8A、字段宽度为 4 的十六进制整数
- c. 一个形如 232.346、字段宽度为 10 的浮点数

- d. 一个形如 2.33e+002、字段宽度为 12 的浮点数
- e. 一个字段宽度为 30、左对齐的字符串
7. 打印下面各项内容要分别使用什么转换说明?
- 字段宽度为 15 的 unsigned long 类型的整数
  - 一个形如 0x8a、字段宽度为 4 的十六进制整数
  - 一个形如 2.33E+02、字段宽度为 12、左对齐的浮点数
  - 一个形如+232.346、字段宽度为 10 的浮点数
  - 一个字段宽度为 8 的字符串的前 8 个字符
8. 打印下面各项内容要分别使用什么转换说明?
- 一个字段宽度为 6、最少有 4 位数字的十进制整数
  - 一个在参数列表中给定字段宽度的八进制整数
  - 一个字段宽度为 2 的字符
  - 一个形如+3.13、字段宽度等于数字中字符数的浮点数
  - 一个字段宽度为 7、左对齐字符串中的前 5 个字符
9. 分别写出读取下列各输入行的 scanf()语句，并声明语句中用到变量和数组。
- 101
  - 22.32 8.34E-09
  - linguini
  - catch 22
  - catch 22 (但是跳过 catch)
10. 什么是空白?
11. 下面的语句有什么问题? 如何修正?
- ```
printf("The double type is %z bytes..\n", sizeof(double));
```
12. 假设要在程序中用圆括号代替花括号，以下方法是否可行?
- ```
#define ( {  
#define ) }
```

## 4.8 编程练习

- 编写一个程序，提示用户输入名和姓，然后以“名,姓”的格式打印出来。
- 编写一个程序，提示用户输入名和姓，并执行一下操作:
  - 打印名和姓，包括双引号；
  - 在宽度为 20 的字段右端打印名和姓，包括双引号；
  - 在宽度为 20 的字段左端打印名和姓，包括双引号；
  - 在比姓名宽度宽 3 的字段中打印名和姓。
- 编写一个程序，读取一个浮点数，首先以小数点记数法打印，然后以指数记数法打印。用下面的格式进行输出（系统不同，指数记数法显示的位数可能不同）:
  - 输入 21.3 或 2.1e+001；

- b. 输入`+21.290`或`2.129E+001`;
4. 编写一个程序，提示用户输入身高（单位：英寸）和姓名，然后以下面的格式显示用户刚输入的信息：

```
Dabney, you are 6.208 feet tall
```

使用`float`类型，并用`/`作为除号。如果你愿意，可以要求用户以厘米为单位输入身高，并以米为单位显示出来。

5. 编写一个程序，提示用户输入以兆位每秒（Mb/s）为单位的下载速度和以兆字节（MB）为单位的文件大小。程序中应计算文件的下载时间。注意，这里1字节等于8位。使用`float`类型，并用`/`作为除号。该程序要以下面的格式打印3个变量的值（下载速度、文件大小和下载时间），显示小数点后面两位数字：

```
At 18.12 megabits per second, a file of 2.20 megabytes
downloads in 0.97 seconds.
```

6. 编写一个程序，先提示用户输入名，然后提示用户输入姓。在一行打印用户输入的名和姓，下一行分别打印名和姓的字母数。字母数要与相应名和姓的结尾对齐，如下所示：

```
Melissa Honeybee
    7      8
```

接下来，再打印相同的信息，但是字母个数与相应名和姓的开头对齐，如下所示：

```
Melissa Honeybee
    7      8
```

7. 编写一个程序，将一个`double`类型的变量设置为`1.0/3.0`，一个`float`类型的变量设置为`1.0/3.0`。分别显示两次计算的结果各3次：一次显示小数点后面6位数字；一次显示小数点后面12位数字；一次显示小数点后面16位数字。程序中要包含`float.h`头文件，并显示`FLT_DIG`和`DBL_DIG`的值。`1.0/3.0`的值与这些值一致吗？

8. 编写一个程序，提示用户输入旅行的里程和消耗的汽油量。然后计算并显示消耗每加仑汽油行驶的英里数，显示小数点后面一位数字。接下来，使用1加仑大约3.785升，1英里大约为1.609千米，把单位是英里/加仑的值转换为升/100公里（欧洲通用的燃料消耗表示法），并显示结果，显示小数点后面1位数字。注意，美国采用的方案测量消耗单位燃料的行程（值越大越好），而欧洲则采用单位距离消耗的燃料测量方案（值越低越好）。使用`#define`创建符号常量或使用`const`限定符创建变量来表示两个转换系数。



# 运算符、表达式和语句

本章介绍以下内容：

- 关键字：while、typedef
- 运算符：=、-、\*、/、%、++、--、(类型名)
- C语言的各种运算符，包括用于普通数学运算的运算符
- 运算符优先级以及语句、表达式的含义
- while 循环
- 复合语句、自动类型转换和强制类型转换
- 如何编写带有参数的函数

现在，读者已经熟悉了如何表示数据，接下来我们学习如何处理数据。C语言为处理数据提供了大量的操作，可以在程序中进行算术运算、比较值的大小、修改变量、逻辑地组合关系等。我们先从基本的算术运算（加、减、乘、除）开始。

组织程序是处理数据的另一个方面，让程序按正确的顺序执行各个步骤。C有许多语言特性，帮助你完成组织程序的任务。循环就是其中一个特性，本章中你将窥其大概。循环能重复执行行为，让程序更有趣、更强大。

## 5.1 循环简介

程序清单 5.1 是一个简单的程序示例，该程序进行了简单的运算，计算穿 9 码男鞋的脚长（单位：英寸）。为了让读者体会循环的好处，程序的第 1 个版本演示了不使用循环编程的局限性。

程序清单 5.1 shoes1.c 程序

```
/* shoes1.c -- 把鞋码转换成英寸 */
#include <stdio.h>
#define ADJUST 7.31           // 字符常量
int main(void)
{
    const double SCALE = 0.333; // const 变量
    double shoe, foot;

    shoe = 9.0;
    foot = SCALE * shoe + ADJUST;
    printf("Shoe size (men's)    foot length\n");
    printf("%10.1f %15.2f inches\n", shoe, foot);

    return 0;
}
```

该程序的输出如下：

```
Shoe size (men's) foot length
 9.0      10.31 inches
```

该程序演示了用#define 指令创建符号常量和用 const 限定符创建在程序运行过程中不可更改的变量。程序使用了乘法和加法，假定用户穿 9 码的鞋，以英寸为单位打印用户的脚长。你可能会说：“这太简单了，我用笔算比敲程序还要快。”说得没错。写出来的程序只使用一次（本例即只根据一只鞋的尺码计算一次脚长），实在是浪费时间和精力。如果写成交互式程序会更有用，但是仍无法利用计算机的优势。

应该让计算机做一些重复计算的工作。毕竟，需要重复计算是使用计算机的主要原因。C 提供多种方法做重复计算，我们在这里简单介绍一种——while 循环。它能让你对运算符做更有趣地探索。程序清单 5.2 演示了用循环改进后的程序。

### 程序清单 5.2 shoes2.c 程序

---

```
/* shoes2.c -- 计算多个不同鞋码对应的脚长 */
#include <stdio.h>
#define ADJUST 7.31           // 字符常量
int main(void)
{
    const double SCALE = 0.333; // const 变量
    double shoe, foot;

    printf("Shoe size (men's)      foot length\n");
    shoe = 3.0;
    while (shoe < 18.5)          /* while 循环开始 */
    {
        /* 块开始 */
        foot = SCALE * shoe + ADJUST;
        printf("%10.1f %15.2f inches\n", shoe, foot);
        shoe = shoe + 1.0;
    }                           /* 块结束 */
    printf("If the shoe fits, wear it.\n");

    return 0;
}
```

---

下面是 shoes2.c 程序的输出（... 表示并未显示完整，有删节）：

```
Shoe size (men's) foot length
 3.0      8.31 inches
 4.0      8.64 inches
 5.0      8.97 inches
 6.0      9.31 inches
...
 16.0     12.64 inches
 17.0     12.97 inches
 18.0     13.30 inches
```

If the shoe fits, wear it.

（如果读者对此颇有研究，应该知道该程序不符合实际情况。程序中假定了一个统一的鞋码系统。）

下面解释一下 while 循环的原理。当程序第 1 次到达 while 循环时，会检查圆括号中的条件是否为真。该程序中，条件表达式如下：

```
shoe < 18.5
```

符号<的意思是小于。变量 shoe 被初始化为 3.0，显然小于 18.5。因此，该条件为真，程序进入块

中继续执行，把尺码转换成英寸。然后打印计算的结果。下一条语句把 `shoe` 增加 1.0，使 `shoe` 的值为 4.0：

```
shoe = shoe + 1.0;
```

此时，程序返回 `while` 入口部分检查条件。为何要返回 `while` 的入口部分？因为上面这条语句的下面是右花括号 ()，代码使用一对花括号 {} 来标出 `while` 循环的范围。花括号之间的内容就是要被重复执行的内容。花括号以及被花括号括起来的部分被称为块 (block)。现在，回到程序中。因为 4 小于 18.5，所以要重复执行被花括号括起来的所有内容（用计算机术语来说就是，程序循环这些语句）。该循环过程一直持续到 `shoe` 的值为 19.0。此时，由于 19.0 小于 18.5，所以该条件为假：

```
shoe < 18.5
```

出现这种情况后，控制转到紧跟 `while` 循环后面的第 1 条语句。该例中，是最后的 `printf()` 语句。

可以很方便地修改该程序用于其他转换。例如，把 `SCALE` 设置成 1.8、`ADJUST` 设置成 32.0，该程序便可把摄氏温度转换成华氏温度；把 `SCALE` 设置成 0.6214、`ADJUST` 设置成 0，该程序便可把公里转换成英里。注意，修改了设置后，还要更改打印的消息，以免前后表述不一。

通过 `while` 循环能便捷灵活地控制程序。现在，我们来学习程序中会用到的基本运算符。

## 5.2 基本运算符

C 用运算符 (operator) 表示算术运算。例如，+ 运算符使在它两侧的值加在一起。如果你觉得术语“运算符”很奇怪，那么请记住东西总得有个名称。与其叫“那些东西”或“运算处理符”，还不如叫“运算符”。现在，我们介绍一下用于基本算术运算的运算符：=、+、-、\* 和 / (C 没有指数运算符。不过，C 的标准数学库提供了一个 `pow()` 函数用于指数运算。例如，`pow(3.5, 2.2)` 返回 3.5 的 2.2 次幂)。

### 5.2.1 赋值运算符：=

在 C 语言中，= 并不意味着“相等”，而是一个赋值运算符。下面的赋值表达式语句：

```
bmw = 2002;
```

把值 2002 赋给变量 `bmw`。也就是说，= 号左侧是一个变量名，右侧是赋给该变量的值。符号= 被称为赋值运算符。另外，上面的语句不读作“`bmw` 等于 2002”，而读作“把值 2002 赋给变量 `bmw`”。赋值行为从右往左进行。

也许变量名和变量值的区别看上去微乎其微，但是，考虑下面这条常用的语句：

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

对数学而言，这完全行不通。如果给一个有限的数加上 1，它不可能“等于”原来的数。但是，在计算机赋值表达式语句中，这很合理。该语句的意思是：找出变量 `i` 的值，把该值加 1，然后把新值赋值变量 `i`（见图 5.1）。

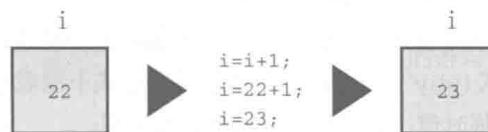


图 5.1 语句 `i = i + 1;`

在 C 语言中，类似这样的语句没有意义（实际上是无效的）：

```
2002 = bmw;
```

因为在这种情况下，2002 被称为右值 (rvale)，只能是字面常量。不能给常量赋值，常量本身就是它

的值。因此，在编写代码时要记住，=号左侧的项必须是一个变量名。实际上，赋值运算符左侧必须引用一个存储位置。最简单的方法就是使用变量名。不过，后面章节还会介绍“指针”，可用于指向一个存储位置。概括地说，C 使用可修改的左值（*modifiable lvalue*）标记那些可赋值的实体。也许“可修改的左值”不太好懂，我们再来看一些定义。

## 几个术语：数据对象、左值、右值和运算符

赋值表达式语句的目的是把值储存到内存位置上。用于储存值的数据存储区域统称为数据对象（*data object*）。C 标准只有在提到这个概念时才会用到对象这个术语。使用变量名是标识对象的一种方法。除此之外，还有其他方法，但是要在后面的章节中才学到。例如，可以指定数组的元素、结构的成员，或者使用指针表达式（指针中储存的是它所指向对象的地址）。左值（*lvalue*）是 C 语言的术语，用于标识特定数据对象的名称或表达式。因此，对象指的是实际的数据存储，而左值是用于标识或定位存储位置的标签。

对于早期的 C 语言，提到左值意味着：

1. 它指定一个对象，所以引用内存中的地址；
2. 它可用在赋值运算符的左侧，左值（*lvalue*）中的 l 源自 left。

但是后来，标准中新增了 `const` 限定符。用 `const` 创建的变量不可修改。因此，`const` 标识符满足上面的第 1 项，但是不满足第 2 项。一方面 C 继续把标识对象的表达式定义为左值，一方面某些左值却不能放在赋值运算符的左侧。有些左值不能用于赋值运算符的左侧。此时，标准对左值的定义已经不能满足当前的状况。

为此，C 标准新增了一个术语：可修改的左值（*modifiable lvalue*），用于标识可修改的对象。所以，赋值运算符的左侧应该是可修改的左值。当前标准建议，使用术语对象定位值（*object locator value*）更好。

右值（*rvalue*）指的是能赋值给可修改左值的量，且本身不是左值。例如，考虑下面的语句：

```
bmw = 2002;
```

这里，`bmw` 是可修改的左值，`2002` 是右值。读者也许猜到了，右值中的 r 源自 right。右值可以是常量、变量或其他可求值的表达式（如，函数调用）。实际上，当前标准在描述这一概念时使用的是表达式的值（*value of an expression*），而不是右值。

我们看几个简单的示例：

```
int ex;
int why;
int zee;
const int TWO = 2;
why = 42;
zee = why;
ex = TWO * (why + zee);
```

这里，`ex`、`why` 和 `zee` 都是可修改的左值（或对象定位值），它们可用于赋值运算符的左侧和右侧。`TWO` 是不可改变的左值，它只能用于赋值运算符的右侧（在该例中，`TWO` 被初始化为 2，这里的=运算符表示初始化而不是赋值，因此并未违反规则）。同时，`42` 是右值，它不能引用某指定内存位置。另外，`why` 和 `zee` 是可修改的左值，表达式（`why + zee`）是右值，该表达式不能表示特定内存位置，而且也不能给它赋值。它只是程序计算的一个临时值，在计算完毕后便会被丢弃。

在学习名称时，被称为“项”（如，赋值运算符左侧的项）的就是运算对象（*operand*）。运算对象是运算符操作的对象。例如，可以把吃汉堡描述为：“吃”运算符操作“汉堡”运算对象。类似地可以说，=运算符的左侧运算对象应该是可修改的左值。

C 的基本赋值运算符有些与众不同，请看程序清单 5.3。

**程序清单 5.3 golf.c 程序**


---

```
/* golf.c -- 高尔夫锦标赛记分卡 */
#include <stdio.h>
int main(void)
{
    int jane, tarzan, cheeta;

    cheeta = tarzan = jane = 68;
    printf("          cheeta  tarzan  jane\n");
    printf("First round score %4d %8d %8d\n", cheeta, tarzan, jane);

    return 0;
}
```

---

许多其他语言都会回避该程序中的三重赋值，但是 C 完全没问题。赋值的顺序是从右往左：首先把 86 赋给 `jane`，然后再赋给 `tarzan`，最后赋给 `cheeta`。因此，程序的输出如下：

|                   |        |      |
|-------------------|--------|------|
| cheetah           | tarzan | jane |
| First round score | 68     | 68   |

**5.2.2 加法运算符：+**

加法运算符 (*addition operator*) 用于加法运算，使其两侧的值相加。例如，语句：

```
printf("%d", 4 + 20);
```

打印的是 24，而不是表达式

```
4 + 20
```

相加的值（运算对象）可以是变量，也可以是常量。因此，执行下面的语句：

```
income = salary + bribes;
```

计算机会查看加法运算符右侧的两个变量，把它们相加，然后把和赋给变量 `income`。

在此提醒读者注意，`income`、`salary` 和 `bribes` 都是可修改的左值。因为每个变量都标识了一个可被赋值的数据对象。但是，表达式 `salary + bribe` 是一个右值。

**5.2.3 减法运算符：-**

减法运算符 (*subtraction operator*) 用于减法运算，使其左侧的数减去右侧的数。例如，下面的语句把 200.0 赋给 `takehome`：

```
takehome = 224.00 - 24.00;
```

+ 和 - 运算符都被称为二元运算符 (*binary operator*)，即这些运算符需要两个运算对象才能完成操作。

**5.2.4 符号运算符：- 和 +**

减号还可用于标明或改变一个值的代数符号。例如，执行下面的语句后，`smokey` 的值为 12：

```
rocky = -12;
smokey = -rocky;
```

以这种方式使用的负号被称为一元运算符 (*unary operator*)。一元运算符只需要一个运算对象（见图 5.2）。

C90 标准新增了一元+运算符，它不会改变运算对象的值或符号，只能这样使用：

```
dozen = +12;
```

编译器不会报错。但是在以前，这样做是不允许的。

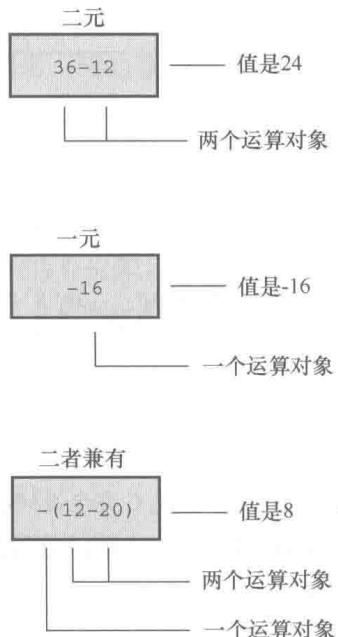


图 5.2 一元和二元运算符

### 5.2.5 乘法运算符: \*

符号`*`表示乘法。下面的语句用 2.54 乘以 `inch`, 并将结果赋给 `cm`:

```
cm = 2.54 * inch;
```

C 没有平方函数, 如果要打印一个平方表, 怎么办? 如程序清单 5.4 所示, 可以使用乘法来计算平方。

程序清单 5.4 squares.c 程序

```
/* squares.c -- 计算 1~20 的平方 */
#include <stdio.h>
int main(void)
{
    int num = 1;

    while (num < 21)
    {
        printf("%4d %6d\n", num, num * num);
        num = num + 1;
    }

    return 0;
}
```

该程序打印数字 1~20 及其平方。接下来, 我们再看一个更有趣的例子。

#### 1. 指数增长

读者可能听过这样一个故事, 一位强大的统治者想奖励做出突出贡献的学者。他问这位学者想要什么, 学者指着棋盘说, 在第 1 个方格里放 1 粒小麦、第 2 个方格里放 2 粒小麦、第 3 个方格里放 4 粒小麦, 第 4 个方格里放 8 粒小麦, 以此类推。这位统治者不熟悉数学, 很惊讶学者竟然提出如此谦虚的要求。因为他原本准备奖励给学者一大笔财产。如果程序清单 5.5 运行的结果正确, 这显然是跟统治者开了一个玩笑。程

序计算出每个方格应放多少小麦，并计算了总数。可能大多数人对小麦的产量不熟悉，该程序以谷粒数为单位，把计算的小麦总数与粗略估计的世界小麦年产量进行了比较。

程序清单 5.5 wheat.c 程序

---

```
/* wheat.c -- 指数增长 */
#include <stdio.h>
#define SQUARES 64           // 棋盘中的方格数
int main(void)
{
    const double CROP = 2E16; // 世界小麦年产谷粒数
    double current, total;
    int count = 1;

    printf("square      grains      total      ");
    printf("fraction of \n");
    printf("          added      grains      ");
    printf("world total\n");
    total = current = 1.0; /* 从 1 颗谷粒开始 */
    printf("%4d %13.2e %12.2e %12.2e\n", count, current,
           total, total / CROP);
    while (count < SQUARES)
    {
        count = count + 1;
        current = 2.0 * current; /* 下一个方格谷粒翻倍 */
        total = total + current; /* 更新总数 */
        printf("%4d %13.2e %12.2e %12.2e\n", count, current,
               total, total / CROP);
    }
    printf("That's all.\n");

    return 0;
}
```

---

程序的输出结果如下：

| square | grains   | total    | fraction of<br>world total |
|--------|----------|----------|----------------------------|
|        | added    | grains   |                            |
| 1      | 1.00e+00 | 1.00e+00 | 5.00e-17                   |
| 2      | 2.00e+00 | 3.00e+00 | 1.50e-16                   |
| 3      | 4.00e+00 | 7.00e+00 | 3.50e-16                   |
| 4      | 8.00e+00 | 1.50e+01 | 7.50e-16                   |
| 5      | 1.60e+01 | 3.10e+01 | 1.55e-15                   |
| 6      | 3.20e+01 | 6.30e+01 | 3.15e-15                   |
| 7      | 6.40e+01 | 1.27e+02 | 6.35e-15                   |
| 8      | 1.28e+02 | 2.55e+02 | 1.27e-14                   |
| 9      | 2.56e+02 | 5.11e+02 | 2.55e-14                   |
| 10     | 5.12e+02 | 1.02e+03 | 5.12e-14                   |

10 个方格以后，该学者得到的小麦仅超过了 1000 粒。但是，看看 55 个方格的小麦数是多少：

55        1.80e+16     3.60e+16     1.80e+00

总量已超过了世界年产量！不妨自己动手运行该程序，看看第 64 个方格有多少小麦。

这个程序示例演示了指数增长的现象。世界人口增长和我们使用的能源都遵循相同的模式。

## 5.2.6 除法运算符：/

C 使用符号 / 来表示除法。/ 左侧的值是被除数，右侧的值是除数。例如，下面 four 的值是 4.0：

```
four = 12.0/3.0;
```

整数除法和浮点数除法不同。浮点数除法的结果是浮点数，而整数除法的结果是整数。整数是没有小数部分的数。这使得 5 除以 3 很让人头痛，因为实际结果有小数部分。在 C 语言中，整数除法结果的小数部分被丢弃，这一过程被称为截断（truncation）。

运行程序清单 5.6 中的程序，看看截断的情况，体会整数除法和浮点数除法的区别。

**程序清单 5.6 divide.c 程序**

---

```
/* divide.c -- 演示除法 */
#include <stdio.h>
int main(void)
{
    printf("integer division: 5/4 is %d \n", 5 / 4);
    printf("integer division: 6/3 is %d \n", 6 / 3);
    printf("integer division: 7/4 is %d \n", 7 / 4);
    printf("floating division: 7./4. is %1.2f \n", 7. / 4.);
    printf("mixed division: 7./4 is %1.2f \n", 7. / 4);

    return 0;
}
```

---

程序清单 5.6 中包含一个“混合类型”的示例，即浮点值除以整型值。C 相对其他一些语言而言，在类型管理上比较宽容。尽管如此，一般情况下还是要避免使用混合类型。该程序的输出如下：

```
integer division: 5/4 is 1
integer division: 6/3 is 2
integer division: 7/4 is 1
floating division: 7./4. is 1.75
mixed division: 7./4 is 1.75
```

注意，整数除法会截断计算结果的小数部分（丢弃整个小数部分），不会四舍五入结果。混合整数和浮点数计算的结果是浮点数。实际上，计算机不能真正用浮点数除以整数，编译器会把两个运算对象转换成相同的类型。本例中，在进行除法运算前，整数会被转换成浮点数。

C99 标准以前，C 语言给语言的实现者留有一些空间，让他们来决定如何进行负数的整数除法。一种方法是，舍入过程采用小于或等于浮点数的最大整数。当然，对于 3.8 而言，处理后的 3 符合这一描述。但是 -3.8 会怎样？该方法建议四舍五入为 -4，因为 -4 小于 -3.8。但是，另一种舍入方法是直接丢弃小数部分。这种方法被称为“趋零截断”，即把 -3.8 转换成 -3。在 C99 以前，不同的实现采用不同的方法。但是 C99 规定使用趋零截断。所以，应把 -3.8 转换成 -3。

## 5.2.7 运算符优先级

考虑下面的代码：

```
butter = 25.0 + 60.0 * n / SCALE;
```

这条语句中有加法、乘法和除法运算。先算哪一个？是 25.0 加上 60.0，然后把计算的和 85.0 乘以 n，再把结果除以 SCALE？还是 60.0 乘以 n，然后把计算的结果加上 25.0，最后再把结果除以 SCALE？还是其他运算顺序？假设 n 是 6.0，SCALE 是 2.0，带入语句中计算会发现，第 1 种顺序得到的结果是 255，

第2种顺序得到的结果是 192.5。C 程序一定是采用了其他的运算顺序，因为程序运行该语句后，butter 的值是 205.0。

显然，执行各种操作的顺序很重要。C 语言对此有明确的规定，通过运算符优先级来解决操作顺序的问题。每个运算符都有自己的优先级。正如普通的算术运算那样，乘法和除法的优先级比加法和减法高，所以先执行乘法和除法。如果两个运算符的优先级相同怎么办？如果它们处理同一个运算对象，则根据它们在语句中出现的顺序来执行。对大多数运算符而言，这种情况都是按从左到右的顺序进行（= 运算符除外）。因此，语句：

```
butter = 25.0 + 60.0 * n / SCALE;
```

的运算顺序是：

|               |                                              |
|---------------|----------------------------------------------|
| 60.0 * n      | 首先计算表达式中的*或/（假设 n 的值是 6，所以 60.0*n 得 360.0）   |
| 360.0 / SCALE | 然后计算表达式中第 2 个*或/                             |
| 25.0 + 180    | 最后计算表达式里第 1 个+或-，结果为 205.0（假设 SCALE 的值是 2.0） |

许多人喜欢用表达式树（expression tree）来表示求值的顺序，如图 5.3 所示。该图演示了如何从最初的表达式逐步简化为一个值。

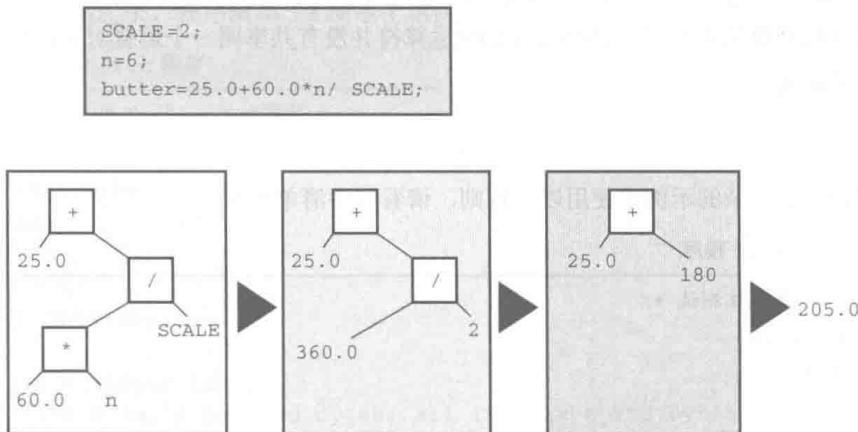


图 5.3 用表达式树演示运算符、运算对象和求值顺序

如何让加法运算在乘法运算之前执行？可以这样做：

```
flour = (25.0 + 60.0 * n) / SCALE;
```

最先执行圆括号中的部分。圆括号内部按正常的规则执行。该例中，先执行乘法运算，再执行加法运算。执行完圆括号内的表达式后，用运算结果除以 SCALE。

表 5.1 总结了到目前为止学过的运算符优先级。

表 5.1 运算符优先级（从低至高）

| 运算符      | 结合律  |
|----------|------|
| ()       | 从左往右 |
| + - (一元) | 从右往左 |
| * /      | 从左往右 |
| + - (二元) | 从左往右 |
| =        | 从右往左 |

注意正号（加号）和负号（减号）的两种不同用法。结合律栏列出了运算符如何与运算对象结合。例

如，一元负号与它右侧的量相结合，在除法中用除号左侧的运算对象除以右侧的运算对象。

## 5.2.8 优先级和求值顺序

运算符优先级为表达式中的求值顺序提供重要的依据，但是并没有规定所有的顺序。C 给语言的实现者留出选择的余地。考虑下面的语句：

```
y = 6 * 12 + 5 * 20;
```

当运算符共享一个运算对象时，优先级决定了求值顺序。例如上面的语句中，`*`是`+`和`*`运算符的运算对象。根据运算符的优先级，乘法的优先级比加法高，所以先进行乘法运算。类似地，先对 5 进行乘法运算而不是加法运算。简而言之，先进行两个乘法运算  $6 * 12$  和  $5 * 20$ ，再进行加法运算。但是，优先级并未规定到底先进行哪一个乘法。`C` 语言把主动权留给语言的实现者，根据不同的硬件来决定先计算前者还是后者。可能在一种硬件上采用某种方案效率更高，而在另一种硬件上采用另一种方案效率更高。无论采用哪种方案，表达式都会简化为  $72 + 100$ ，所以这并不影响最终的结果。但是，读者可能会根据乘法从左往右的结合律，认为应该先执行`+`运算符左边的乘法。结合律只适用于共享同一运算对象运算符。例如，在表达式  $12 / 3 * 2$  中，`/`和`*`运算符的优先级相同，共享运算对象 3。因此，从左往右的结合律在这种情况下起作用。表达式简化为  $4 * 2$ ，即 8（如果从右往左计算，会得到  $12 / 6$ ，即 2，这种情况下计算的先后顺序会影响最终的计算结果）。在该例中，两个`*`运算符并没有共享同一个运算对象，因此从左往右的结合律不适用于这种情况。

### 学以致用

接下来，我们在更复杂的示例中使用以上规则，请看程序清单 5.7。

**程序清单 5.7 rules.c 程序**

---

```
/* rules.c -- 优先级测试 */
#include <stdio.h>
int main(void)
{
    int top, score;

    top = score = -(2 + 5) * 6 + (4 + 3 * (2 + 3));
    printf("top = %d, score = %d\n", top, score);

    return 0;
}
```

---

该程序会打印什么值？先根据代码推测一下，再运行程序或阅读下面的分析来检查你的答案。

首先，圆括号的优先级最高。先计算  $-(2 + 5) * 6$  中的圆括号部分，还是先计算  $(4 + 3 * (2 + 3))$  中的圆括号部分取决于具体的实现。圆括号的最高优先级意味着，在子表达式  $-(2 + 5) * 6$  中，先计算  $(2 + 5)$  的值，得 7。然后，把一元负号应用在 7 上，得 -7。现在，表达式是：

```
top = score = -7 * 6 + (4 + 3 * (2 + 3))
```

下一步，计算  $2 + 3$  的值。表达式变成：

```
top = score = -7 * 6 + (4 + 3 * 5)
```

接下来，因为圆括号中的`*`比`+`优先级高，所以表达式变成：

```
top = score = -7 * 6 + (4 + 15)
```

然后，表达式为：

```
top = score = -7 * 6 + 19
```

$-7 \times 6$  后，得到下面的表达式：

```
top = score = -42 + 19
```

然后进行加法运算，得到：

```
top = score = -23
```

现在， $-23$  被赋值给 `score`，最终 `top` 的值也是 $-23$ 。记住，`=`运算符的结合律是从右往左。

## 5.3 其他运算符

C 语言有大约 40 个运算符，有些运算符比其他运算符常用得多。前面讨论的是最常用的，本节再介绍 4 个比较有用的运算符。

### 5.3.1 `sizeof` 运算符和 `size_t` 类型

读者在第 3 章就见过 `sizeof` 运算符。回顾一下，`sizeof` 运算符以字节为单位返回运算对象的大小（在 C 中，1 字节定义为 `char` 类型占用的空间大小。过去，1 字节通常是 8 位，但是一些字符集可能使用更大的字节）。运算对象可以是具体的数据对象（如，变量名）或类型。如果运算对象是类型（如，`float`），则必须用圆括号将其括起来。程序清单 5.8 演示了这两种用法。

程序清单 5.8 `sizeof.c` 程序

---

```
// sizeof.c -- 使用 sizeof 运算符
// 使用 C99 新增的%zd 转换说明 -- 如果编译器不支持%zd, 请将其改成%u 或%lu
#include <stdio.h>
int main(void)
{
    int n = 0;
    size_t intsize;

    intsize = sizeof (int);
    printf("n = %d, n has %zd bytes; all ints have %zd bytes.\n",
           n, sizeof n, intsize);

    return 0;
}
```

---

C 语言规定，`sizeof` 返回 `size_t` 类型的值。这是一个无符号整数类型，但它不是新类型。前面介绍过，`size_t` 是语言定义的标准类型。C 有一个 `typedef` 机制（第 14 章再详细介绍），允许程序员为现有类型创建别名。例如，

```
typedef double real;
```

这样，`real` 就是 `double` 的别名。现在，可以声明一个 `real` 类型的变量：

```
real deal; // 使用 typedef
```

编译器查看 `real` 时会发现，在 `typedef` 声明中 `real` 已成为 `double` 的别名，于是把 `deal` 创建为 `double` 类型的变量。类似地，C 头文件系统可以使用 `typedef` 把 `size_t` 作为 `unsigned int` 或 `unsigned long` 的别名。这样，在使用 `size_t` 类型时，编译器会根据不同的系统替换标准类型。

C99 做了进一步调整，新增了`%zd` 转换说明用于 `printf()` 显示 `size_t` 类型的值。如果系统不支持`%zd`，可使用`%u` 或`%lu` 代替`%zd`。

### 5.3.2 求模运算符：%

求模运算符 (*modulus operator*) 用于整数运算。求模运算符给出其左侧整数除以右侧整数的余数 (*remainder*)。例如， $13 \% 5$  (读作“13 求模 5”) 得 3，因为 13 比 5 的两倍多 3，即 13 除以 5 的余数是 3。求模运算符只能用于整数，不能用于浮点数。

乍一看会认为求模运算符像是数学家使用的深奥符号，但是实际上它非常有用。求模运算符常用于控制程序流。例如，假设你正在设计一个账单预算程序，每 3 个月要加进一笔额外的费用。这种情况可以在程序中对月份求模 3 (即，`month % 3`)，并检查结果是否为 0。如果为 0，便加进额外的费用。等到第 7 章的 `if` 语句后，读者会更明白。

程序清单 5.9 演示了%运算符的另一种用途。同时，该程序也演示了 `while` 循环的另一种用法。

程序清单 5.9 min\_sec.c 程序

---

```
// min_sec.c -- 把秒数转换成分和秒
#include <stdio.h>
#define SEC_PER_MIN 60           // 1 分钟 60 秒
int main(void)
{
    int sec, min, left;

    printf("Convert seconds to minutes and seconds!\n");
    printf("Enter the number of seconds (<=0 to quit):\n");
    scanf("%d", &sec);          // 读取秒数
    while (sec > 0)
    {
        min = sec / SEC_PER_MIN; // 截断分钟数
        left = sec % SEC_PER_MIN; // 剩下的秒数
        printf("%d seconds is %d minutes, %d seconds.\n", sec,
               min, left);
        printf("Enter next value (<=0 to quit):\n");
        scanf("%d", &sec);
    }
    printf("Done!\n");

    return 0;
}
```

---

该程序的输出如下：

```
Convert seconds to minutes and seconds!
Enter the number of seconds (<=0 to quit):
154
154 seconds is 2 minutes, 34 seconds.
Enter next value (<=0 to quit):
567
567 seconds is 9 minutes, 27 seconds.
Enter next value (<=0 to quit):
0
Done!
```

程序清单 5.2 使用一个计数器来控制 `while` 循环。当计数器超出给定的大小时，循环终止。而程序清单 5.9 则通过 `scanf()` 为变量 `sec` 获取一个新值。只要该值为正，循环就继续。当用户输入一个 0 或负值时，循环退出。这两种情况设计的要点是，每次循环都会修改被测试的变量值。

负数求模如何进行? C99 规定“趋零截断”之前, 该问题的处理方法很多。但自从有了这条规则之后, 如果第 1 个运算对象是负数, 那么求模的结果为负数; 如果第 1 个运算对象是正数, 那么求模的结果也是正数:

```
11 / 5 得 2, 11 % 5 得 1
11 / -5 得 -2, 11 % -5 得 1
-11 / -5 得 2, -11 % -5 得 -1
-11 / 5 得 -2, -11 % 5 得 -1
```

如果当前系统不支持 C99 标准, 会显示不同的结果。实际上, 标准规定: 无论何种情况, 只要  $a$  和  $b$  都是整数值, 便可通过  $a - (a/b) * b$  来计算  $a \% b$ 。例如, 可以这样计算  $-11 \% 5$ :

```
-11 - (-11/5) * 5 = -11 - (-2)*5 = -11 - (-10) = -1
```

### 5.3.3 递增运算符: ++

递增运算符 (*increment operator*) 执行简单的任务, 将其运算对象递增 1。该运算符以两种方式出现。第 1 种方式, `++` 出现在其作用的变量前面, 这是前缀模式; 第 2 种方式, `++` 出现在其作用的变量后面, 这是后缀模式。两种模式的区别在于递增行为发生的时间不同。我们先解释它们的相似之处, 再分析它们不同之处。程序清单 5.10 中的程序示例演示了递增运算符是如何工作的。

程序清单 5.10 add\_one.c 程序

---

```
/* add_one.c -- 递增: 前缀和后缀 */
#include <stdio.h>
int main(void)
{
    int ultra = 0, super = 0;

    while (super < 5)
    {
        super++;
        ++ultra;
        printf("super = %d, ultra = %d \n", super, ultra);
    }

    return 0;
}
```

---

运行该程序后, 其输出如下:

```
super = 1, ultra = 1
super = 2, ultra = 2
super = 3, ultra = 3
super = 4, ultra = 4
super = 5, ultra = 5
```

该程序两次同时计数到 5。用下面两条语句分别代替程序中的两条递增语句, 程序的输出相同:

```
super = super + 1;
ultra = ultra + 1;
```

这些都是很简单的语句, 为何还要创建两个缩写形式? 原因之一是, 紧凑结构的代码让程序更为简洁, 可读性更高。这些运算符让程序看起来很美观。例如, 可重写程序清单 5.2 (shoes2.c) 中的一部分代码:

```
shoe = 3.0;
while (shoe < 18.5)
{
    foot = SCALE * size + ADJUST;
```

```

    printf("%10.1f %20.2f inches\n", shoe, foot);
    ++shoe;
}

```

但是，这样做也没有充分利用递增运算符的优势。还可以这样缩短这段程序：

```

shoe = 2.0;
while (++shoe < 18.5)
{
    foot = SCALE*shoe + ADJUST;
    printf("%10.1f %20.2f inches\n", shoe, foot);
}

```

如上代码所示，把变量的递增过程放入 while 循环的条件中。这种结构在 C 语言中很普遍，我们来仔细分析一下。

首先，这样的 while 循环是如何工作的？很简单。shoe 的值递增 1，然后和 18.5 作比较。如果递增后的值小于 18.5，则执行花括号内的语句一次。然后，shoe 的值再递增 1，重复刚才的步骤，直到 shoe 的值不小于 18.5 为止。注意，我们把 shoe 的初始值从 3.0 改为 2.0，因为在对 foot 第 1 次求值之前，shoe 已经递增了 1（见图 5.4）。

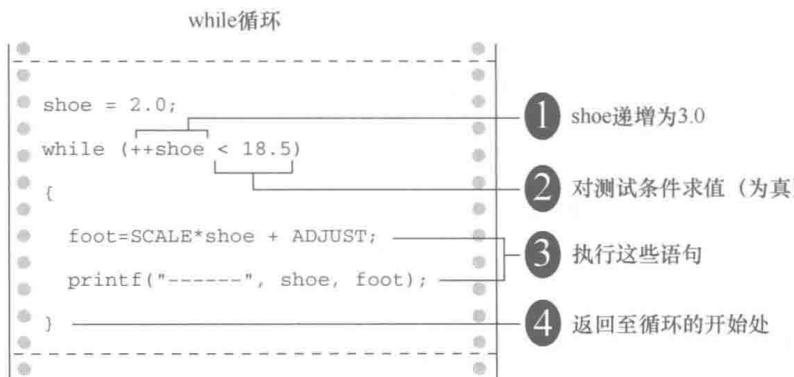


图 5.4 执行一次循环

其次，这样做有什么好处？它使得程序更加简洁。更重要的是，它把控制循环的两个过程集中在一个地方。该循环的主要过程是判断是否继续循环（本例中，要检查鞋子的尺码是否小于 18.5），次要过程是改变待测试的元素（本例中是递增鞋子的尺码）。

如果忘记改变鞋子的尺码，shoe 的值会一直小于 18.5，循环不会停止。计算机将陷入无限循环 (*infinite loop*) 中，生成无数相同的行。最后，只能强行关闭这个程序。把循环测试和更新循环放在一处，就不会忘记更新循环。

但是，把两个操作合并在一个表达式中，降低了代码的可读性，让代码难以理解。而且，还容易产生计数错误。

递增运算符的另一个优点是，通常它生成的机器语言代码效率更高，因为它和实际的机器语言指令很相似。尽管如此，随着商家推出的 C 编译器越来越智能，这一优势可能会消失。一个智能的编译器可以把 `x = x + 1` 当作 `++x` 对待。

最后，递增运算符还有一个在某些场合特别有用的特性。我们通过程序清单 5.11 来说明。

#### 程序清单 5.11 post\_pre.c 程序

```

/* post_pre.c -- 前缀和后缀 */
#include <stdio.h>

```

```

int main(void)
{
    int a = 1, b = 1;
    int a_post, pre_b;

    a_post = a++; // 后缀递增
    pre_b = ++b; // 前缀递增
    printf("a a_post b pre_b \n");
    printf("%ld %d %d %d\n", a, a_post, b, pre_b);

    return 0;
}

```

如果你的编译器没问题，那么程序的输出应该是：

|   |        |   |       |
|---|--------|---|-------|
| a | a_post | b | pre_b |
| 2 |        | 1 | 2     |

a 和 b 都递增了 1，但是，a\_post 是 a 递增之前的值，而 b\_pre 是 b 递增之后的值。这就是++的前缀形式和后缀形式的区别（见图 5.5）。



图 5.5 前缀和后缀

```

a_post = a++; // 后缀: 使用 a 的值之后, 递增 a
b_pre = ++b; // 前缀: 使用 b 的值之前, 递增 b

```

单独使用递增运算符时（如，`ego++`），使用哪种形式都没关系。但是，当运算符和运算对象是更复杂表达式的一部分时（如上面的示例），使用前缀或后缀的效果不同。例如，我们曾经建议用下面的代码：

```
while (++shoe < 18.5)
```

该测试条件相当于提供了一个鞋子尺码到 18 的表。如果使用 `shoe++` 而不是 `++shoes`，尺码表会增至 19。因为 `shoe` 会在与 18.5 进行比较之后才递增，而不是先递增再比较。

当然，使用下面这种形式也没错：

```
shoe = shoe + 1;
```

只不过，有人会怀疑你是否是真正的 C 程序员。

在学习本书的过程中，应多留意使用递增运算符的例子。自己思考是否能互换使用前缀和后缀形式，或者当前环境是否只能使用某种形式。

如果使用前缀形式和后缀形式会对代码产生不同的影响，那么最为明智的是不要那样使用它们。例如，不要使用下面的语句：

```
b = ++i; // 如果使用 i++, 会得到不同的结果
```

应该使用下列语句：

```
++i;      // 第1行
b = i;    // 如果第1行使用的是i++, 并不会影响b的值
```

尽管如此，有时小心翼翼地使用会更有意思。所以，本书会根据实际情况，采用不同的写法。

### 5.3.4 递减运算符：--

每种形式的递增运算符都有一个递减运算符（*decrement operator*）与之对应，用--代替++即可：

```
--count; // 前缀形式的递减运算符
count--; // 后缀形式的递减运算符
```

程序清单 5.12 演示了计算机可以是位出色的填词家。

程序清单 5.12 bottles.c 程序

```
#include <stdio.h>
#define MAX 100
int main(void)
{
    int count = MAX + 1;

    while (--count > 0) {
        printf("%d bottles of spring water on the wall, "
               "%d bottles of spring water!\n", count, count);
        printf("Take one down and pass it around,\n");
        printf("%d bottles of spring water!\n\n", count - 1);
    }

    return 0;
}
```

该程序的输出如下（篇幅有限，省略了中间大部分输出）：

```
100 bottles of spring water on the wall, 100 bottles of spring water!
Take one down and pass it around,
99 bottles of spring water!

99 bottles of spring water on the wall, 99 bottles of spring water!
Take one down and pass it around,
98 bottles of spring water!
...

1 bottles of spring water on the wall, 1 bottles of spring water!
Take one down and pass it around,
0 bottles of spring water!
```

显然，这位填词家在复数的表达上有点问题。在学完第 7 章中的条件运算符后，可以解决这个问题。

顺带一提，>运算符表示“大于”，<运算符表示“小于”，它们都是关系运算符（*relational operator*）。我们将在第 6 章中详细介绍关系运算符。

### 5.3.5 优先级

递增运算符和递减运算符都有很高的结合优先级，只有圆括号的优先级比它们高。因此，`x*y++`表示的是`(x)*(y++)`，而不是`(x+y)++`。不过后者无效，因为递增和递减运算符只能影响一个变量（或者，更普遍地说，只能影响一个可修改的左值），而组合`x*y`本身不是可修改的左值。

不要混淆这两个运算符的优先级和它们的求值顺序。假设有如下语句：

```
y = 2;
n = 3;
nextnum = (y + n++)*6;
```

nextnum 的值是多少？把 y 和 n 的值带入上面的第 3 条语句得：

```
nextnum = (2 + 3)*6 = 5*6 = 30
```

n 的值只有在被使用之后才会递增为 4。根据优先级的规定，++ 只作用于 n，不作用与 y + n。除此之外，根据优先级可以判断何时使用 n 的值对表达式求值，而递增运算符的性质决定了何时递增 n 的值。

如果 n++ 是表达式的一部分，可将其视为“先使用 n，再递增”；而 ++n 则表示“先递增 n，再使用”。

### 5.3.6 不要自作聪明

如果一次用太多递增运算符，自己都会糊涂。例如，利用递增运算符改进 squares.c 程序（程序清单 5.4），用下面的 while 循环替换原程序中的 while 循环：

```
while (num < 21)
{
    printf("%10d %10d\n", num, num*num++);
}
```

这个想法看上去不错。打印 num，然后计算 num\*num 得到平方值，最后把 num 递增 1。但事实上，修改后的程序只能在某些系统上能正常运行。该程序的问题是：当 printf() 获取待打印的值时，可能先对最后一个参数 (num\*num++) 求值，这样在获取其他参数的值之前就递增了 num。所以，本应打印：

5 25

却打印成：

6 25

它甚至可能从右往左执行，对最右边的 num (++ 作用的 num) 使用 5，对第 2 个 num 和最左边的 num 使用 6，结果打印出：

6 30

在 C 语言中，编译器可以自行选择先对函数中的哪个参数求值。这样做提高了编译器的效率，但是如果在函数的参数中使用了递增运算符，就会有一些问题。

类似这样的语句，也会导致一些麻烦：

```
ans = num/2 + 5*(1 + num++);
```

同样，该语句的问题是：编译器可能不会按预想的顺序来执行。你可能认为，先计算第 1 项 (num/2)，接着计算第 2 项 (5\*(1 + num++))。但是，编译器可能先计算第 2 项，递增 num，然后在 num/2 中使用 num 递增后的新值。因此，无法保证编译器到底先计算哪一项。

还有一种情况，也不确定：

```
n = 3;
y = n++ + n++;
```

可以肯定的是，执行完这两条语句后，n 的值会比旧值大 2。但是，y 的值不确定。在对 y 求值时，编译器可以使用 n 的旧值 (3) 两次，然后把 n 递增 1 两次，这使得 y 的值为 6，n 的值为 5。或者，编译器使用 n 的旧值 (3) 一次，立即递增 n，再对表达式中的第 2 个 n 使用递增后的新值，然后再递增 n，这使得 y 的值为 7，n 的值为 5。两种方案都可行。对于这种情况更精确地说，结果是未定义的，这意味着 C 标准并未定义结果应该是什么。

遵循以下规则，很容易避免类似的问题：

- 如果一个变量出现在一个函数的多个参数中，不要对该变量使用递增或递减运算符；
- 如果一个变量多次出现在一个表达式中，不要对该变量使用递增或递减运算符。

另一方面，对于何时执行递增，C 还是做了一些保证。我们在本章后面的“副作用和序列点”中学到序列点时再来讨论这部分内容。

## 5.4 表达式和语句

在前几章中，我们已经多次使用了术语表达式（*expression*）和语句（*statement*）。现在，我们来进一步学习它们。C 的基本程序步骤由语句组成，而大多数语句都由表达式构成。因此，我们先学习表达式。

### 5.4.1 表达式

表达式（*expression*）由运算符和运算对象组成（前面介绍过，运算对象是运算符操作的对象）。最简单的表达式是一个单独的运算对象，以此为基础可以建立复杂的表达式。下面是一些表达式：

```
4
-6
4+21
a*(b + c/d)/20
q = 5*2
x = ++q % 3
q > 3
```

如你所见，运算对象可以是常量、变量或二者的组合。一些表达式由子表达式（*subexpression*）组成（子表达式即较小的表达式）。例如， $c/d$  是上面例子中  $a*(b + c/d)/20$  的子表达式。

#### 每个表达式都有一个值

C 表达式的一个最重要的特性是，每个表达式都有一个值。要获得这个值，必须根据运算符优先级规定的顺序来执行操作。在上面我们列出的表达式中，前几个都很清晰明了。但是，有赋值运算符（=）的表达式的值是什么？这些表达式的值与赋值运算符左侧变量的值相同。因此，表达式  $q = 5*2$  作为一个整体的值是 10。那么，表达式  $q > 3$  的值是多少？这种关系表达式的值不是 0 就是 1，如果条件为真，表达式的值为 1；如果条件为假，表达式的值为 0。表 5.2 列出了一些表达式及其值：

表 5.2 一些表达式及其值

| 表达式             | 值  |
|-----------------|----|
| -4 + 6          | 2  |
| c = 3 + 8       | 11 |
| 5 > 3           | 1  |
| 6 + (c = 3 + 8) | 17 |

虽然最后一个表达式看上去很奇怪，但是在 C 中完全合法（但不建议使用），因为它是两个子表达式的和，每个子表达式都有一个值。

### 5.4.2 语句

语句（*statement*）是 C 程序的基本构建块。一条语句相当于一条完整的计算机指令。在 C 中，大部分语句都以分号结尾。因此，

```
legs = 4
```

只是一个表达式（它可能是一个较大表达式的一部分），而下面的代码则是一条语句：

```
legs = 4;
```

最简单的语句是空语句：

```
; //空语句
```

C 把末尾加上一个分号的表达式都看作是一条语句（即，表达式语句）。因此，像下面这样写也没问题：

```
8;  
3 + 4;
```

但是，这些语句在程序中什么也不做，不算是真正有用的语句。更确切地说，语句可以改变值或调用函数：

```
x = 25;  
++x;  
y = sqrt(x);
```

虽然一条语句（或者至少是一条有用的语句）相当于一条完整的指令，但并不是所有的指令都是语句。考虑下面的语句：

```
x = 6 + (y = 5);
```

该语句中的子表达式  $y = 5$  是一条完整的指令，但是它只是语句的一部分。因为一条完整的指令不一定是一条语句，所以分号用于识别在这种情况下的语句（即，简单语句）。

到目前为止，读者已经见过多种语句（不包括空语句）。程序清单 5.13 演示了一些常见的语句。

程序清单 5.13 addemup.c 程序

---

```
/* addemup.c -- 几种常见的语句 */  
#include <stdio.h>  
int main(void) /* 计算前 20 个整数的和 */  
{  
    int count, sum; /* 声明1 */  
  
    count = 0; /* 表达式语句 */  
    sum = 0; /* 表达式语句 */  
    while (count++ < 20) /* 迭代语句 */  
        sum = sum + count;  
    printf("sum = %d\n", sum); /* 表达式语句2 */  
  
    return 0; /* 跳转语句 */  
}
```

---

下面我们讨论程序清单 5.13。到目前为止，相信读者已经很熟悉声明了。尽管如此，我们还是要提醒读者：声明创建了名称和类型，并为其分配内存位置。注意，声明不是表达式语句。也就是说，如果删除声明后面的分号，剩下的部分不是一个表达式，也没有值：

```
int port /* 不是表达式，没有值 */
```

赋值表达式语句在程序中很常用：它为变量分配一个值。赋值表达式语句的结构是，一个变量名，后面是一个赋值运算符，再跟着一个表达式，最后以分号结尾。注意，在 while 循环中有一个赋值表达式语句。赋值表达式语句是表达式语句的一个示例。

<sup>1</sup> 根据 C 标准，声明不是语句。这与 C++有所不同。——译者注

<sup>2</sup> 在 C 语言中，赋值和函数调用都是表达式。没有所谓的“赋值语句”和“函数调用语句”，这些语句实际上都是表达式语句。本书将“assignment statement”均译为“赋值表达式语句”，以提醒读者注意。——译者注

函数表达式语句会引起函数调用。在该例中，调用 `printf()` 函数打印结果。`while` 语句有 3 个不同的部分（见图 5.6）。首先是关键字 `while`；然后，圆括号中是待测试的条件；最后如果测试条件为真，则执行 `while` 循环体中的语句。该例的 `while` 循环中只有一条语句。可以是本例那样的一条语句，不需要用花括号括起来，也可以像其他例子中那样包含多条语句。多条语句需要用花括号括起来。这种语句是复合语句，稍后马上介绍。

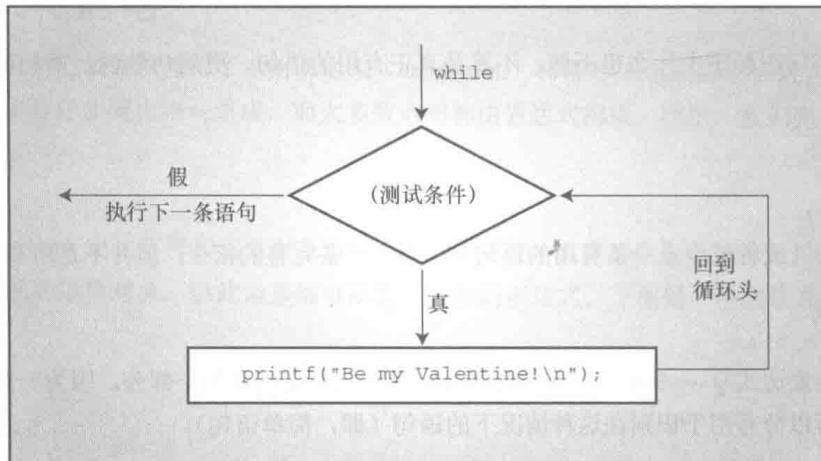


图 5.6 简单的 `while` 循环结构

`while` 语句是一种迭代语句，有时也被称为结构化语句，因为它的结构比简单的赋值表达式语句复杂。在后面的章节里，我们会遇到许多这样的语句。

### 副作用和序列点

我们再讨论一个 C 语言的术语副作用 (*side effect*)。副作用是对数据对象或文件的修改。例如，语句：

```
states = 50;
```

它的副作用是将变量的值设置为 50。副作用？这似乎更像是主要目的！但是从 C 语言的角度看，主要目的是对表达式求值。给出表达式 `4 + 6`，C 会对其求值得 10；给出表达式 `states = 50`，C 会对其求值得 50。对该表达式求值的副作用是把变量 `states` 的值改为 50。跟赋值运算符一样，递增和递减运算符也有副作用，使用它们的主要目的就是使用其副作用。

类似地，调用 `printf()` 函数时，它显示的信息其实是副作用 (`printf()` 的返回值是待显示字符的个数)。

序列点 (*sequence point*) 是程序执行的点，在该点上，所有的副作用都在进入下一步之前发生。在 C 语言中，语句中的分号标记了一个序列点。意思是，在一个语句中，赋值运算符、递增运算符和递减运算符对运算对象做的改变必须在程序执行下一条语句之前完成。后面我们要讨论的一些运算符也有序列点。另外，任何一个完整表达式的结束也是一个序列点。

什么是完整表达式？所谓完整表达式 (*full expression*)，就是指这个表达式不是另一个更大表达式的子表达式。例如，表达式语句中的表达式和 `while` 循环中的作为测试条件的表达式，都是完整表达式。

序列点有助于分析后缀递增何时发生。例如，考虑下面的代码：

```
while (guests++ < 10)
    printf("%d \n", guests);
```

对于该例，C 语言的初学者认为“先使用值，再递增它”的意思是，在 `printf()` 语句中先使用 `guests`，再递增它。但是，表达式 `guests++ < 10` 是一个完整的表达式，因为它是 `while` 循环的测试条件，所以

该表达式的结束就是一个序列点。因此，C 保证了在程序转至执行 `printf()` 之前发生副作用（即，递增 `guests`）。同时，使用后缀形式保证了 `guests` 在完成与 10 的比较后才进行递增。

现在，考虑下面这条语句：

```
y = (4 + x++) + (6 + x++);
```

表达式 `4 + x++` 不是一个完整的表达式，所以 C 无法保证 `x` 在子表达式 `4 + x++` 求值后立即递增 `x`。这里，完整表达式是整个赋值表达式语句，分号标记了序列点。所以，C 保证程序在执行下一条语句之前递增 `x` 两次。C 并未指明是在对子表达式求值以后递增 `x`，还是对所有表达式求值后再递增 `x`。因此，要尽量避免编写类似的语句。

### 5.4.3 复合语句（块）

复合语句（compound statement）是用花括号括起来的一条或多条语句，复合语句也称为块（block）。`shoes2.c` 程序使用块让 `while` 语句包含多条语句。比较下面两个程序段：

```
/* 程序段 1 */
index = 0;
while (index++ < 10)
    sam = 10 * index + 2;
printf("sam = %d\n", sam);

/* 程序段 2 */
index = 0;
while (index++ < 10)
{
    sam = 10 * index + 2;
    printf("sam = %d\n", sam);
}
```

程序段 1，`while` 循环中只有一条赋值表达式语句。没有花括号，`while` 语句从 `while` 这行运行至下一个分号。循环结束后，`printf()` 函数只会被调用一次。

程序段 2，花括号确保两条语句都是 `while` 循环的一部分，每执行一次循环就调用一次 `printf()` 函数。根据 `while` 语句的结构，整个复合语句被视为一条语句（见图 5.7）。

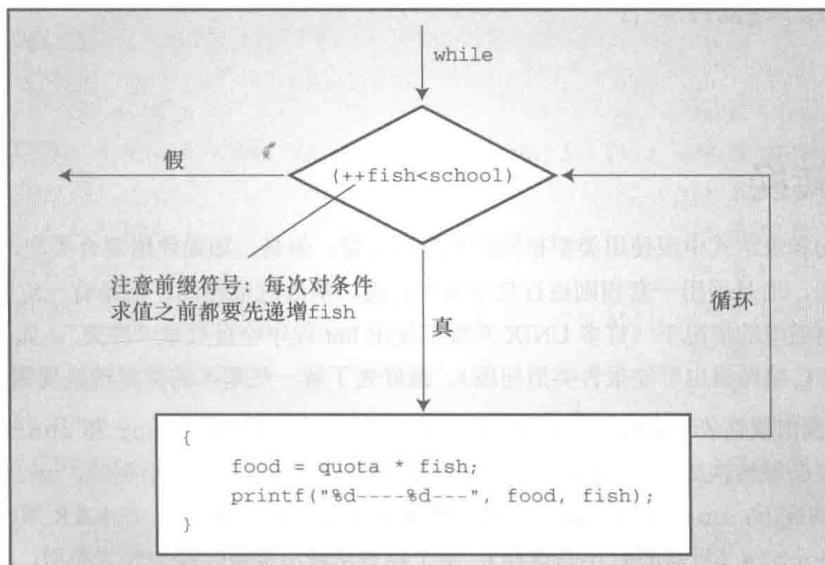


图 5.7 带复合语句的 `while` 循环

### 提示 风格提示

再看一下前面的两个 while 程序段，注意循环体中的缩进。缩进对编译器不起作用，编译器通过花括号和 while 循环的结构来识别和解释指令。这里，缩进是为了让读者一眼就可以看出程序是如何组织的。

程序段 2 中，块或复合语句放置花括号的位置是一种常见的风格。另一种常用的风格是：

```
while (index++ < 10) {
    sam = 10*index + 2;
    printf("sam = %d \n", sam);
}
```

这种风格突出了块附属于 while 循环，而前一种风格则强调语句形成一个块。对编译器而言，这两种风格完全相同。

总而言之，使用缩进可以为读者指明程序的结构。

### 总结 表达式和语句

**表达式：**

表达式由运算符和运算对象组成。最简单的表达式是不带运算符的一个常量或变量（如，22 或 beebop）。更复杂的例子是 55 + 22 和 vap = 2 \* (vip + (vup = 4))。

**语句：**

到目前为止，读者接触到的语句可分为简单语句和复合语句。简单语句以一个分号结尾。如下所示：

|          |                       |
|----------|-----------------------|
| 赋值表达式语句： | toes = 12;            |
| 函数表达式语句： | printf("%d\n", toes); |
| 空语句：     | /* 什么也不做 */           |

复合语句（或块）由花括号括起来的一条或多条语句组成。如下面的 while 语句所示：

```
while (years < 100)
{
    wisdom = wisdom * 1.05;
    printf("%d %d\n", years, wisdom);
    years = years + 1;
}
```

## 5.5 类型转换

通常，在语句和表达式中应使用类型相同的变量和常量。但是，如果使用混合类型，C 不会像 Pascal 那样停在那里死掉，而是采用一套规则进行自动类型转换。虽然这很便利，但是有一定的危险性，尤其是在无意间混合使用类型的情况下（许多 UNIX 系统都使用 lint 程序检查类型“冲突”。如果选择更高错误级别，许多非 UNIX C 编译器也可能报告类型问题）。最好先了解一些基本的类型转换规则。

- 当类型转换出现在表达式时，无论是 unsigned 还是 signed 的 char 和 short 都会被自动转换成 int，如有必要会被转换成 unsigned int（如果 short 与 int 的大小相同，unsigned short 就比 int 大。这种情况下，unsigned short 会被转换成 unsigned int）。在 K&R 那时的 C 中，float 会被自动转换成 double（目前的 C 不是这样）。由于都是从较小类型转换为较大类型，所以这些转换被称为升级（promotion）。

2. 涉及两种类型的运算，两个值会被分别转换成两种类型的更高级别。
3. 类型的级别从高至低依次是 long double、double、float、unsigned long long、long long、unsigned long、long、unsigned int、int。例外的情况是，当 long 和 int 的大小相同时，unsigned int 比 long 的级别高。之所以 short 和 char 类型没有列出，是因为它们已经被升级到 int 或 unsigned int。
4. 在赋值表达式语句中，计算的最终结果会被转换成被赋值变量的类型。这个过程可能导致类型升级或降级（demotion）。所谓降级，是指把一种类型转换成更低级别的类型。
5. 当作为函数参数传递时，char 和 short 被转换成 int，float 被转换成 double。第 9 章将介绍，函数原型会覆盖自动升级。

类型升级通常都不会有什么问题，但是类型降级会导致真正的麻烦。原因很简单：较低类型可能放不下整个数字。例如，一个 8 位的 char 类型变量储存整数 101 没问题，但是存不下 22334。

如果待转换的值与目标类型不匹配怎么办？这取决于转换涉及的类型。待赋值的值与目标类型不匹配时，规则如下。

1. 目标类型是无符号整型，且待赋的值是整数时，额外的位将被忽略。例如，如果目标类型是 8 位 unsigned char，待赋的值是原始值求模 256。
2. 如果目标类型是一个有符号整型，且待赋的值是整数，结果因实现而异。
3. 如果目标类型是一个整型，且待赋的值是浮点数，该行为是未定义的。

如果把一个浮点值转换成整数类型会怎样？当浮点类型被降级为整数类型时，原来的浮点值会被截断。例如，23.12 和 23.99 都会被截断为 23，-23.5 会被截断为 -23。

程序清单 5.14 演示了这些规则。

程序清单 5.14 convert.c 程序

---

```
/* convert.c -- 自动类型转换 */
#include <stdio.h>
int main(void)
{
    char ch;
    int i;
    float fl;

    fl = i = ch = 'C';                                /* 第 9 行 */
    printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl); /* 第 10 行 */
    ch = ch + 1;                                     /* 第 11 行 */
    i = fl + 2 * ch;                                 /* 第 12 行 */
    fl = 2.0 * ch + i;                             /* 第 13 行 */
    printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl); /* 第 14 行 */
    ch = 1107;                                       /* 第 15 行 */
    printf("Now ch = %c\n", ch);                     /* 第 16 行 */
    ch = 80.89;                                      /* 第 17 行 */
    printf("Now ch = %c\n", ch);                     /* 第 18 行 */

    return 0;
}
```

---

运行 convert.c 后输出如下：

```
ch = C, i = 67, f1 = 67.00
ch = D, i = 203, f1 = 339.00
Now ch = S
Now ch = P
```

在我们的系统中，char 是 8 位，int 是 32 位。程序的分析如下。

- 第 9 行和第 10 行：字符'C'被作为 1 字节的 ASCII 值储存在 ch 中。整数变量 i 接受由'C'转换的整数，即按 4 字节储存 67。最后，f1 接受由 67 转换的浮点数 67.00。
- 第 11 行和第 14 行：字符变量'C'被转换成整数 67，然后加 1。计算结果是 4 字节整数 68，被截断成 1 字节储存在 ch 中。根据%c 转换说明打印时，68 被解释成'D'的 ASCII 码。
- 第 12 行和第 14 行：ch 的值被转换成 4 字节的整数（68），然后 2 乘以 ch。为了和 f1 相加，乘积整数（136）被转换成浮点数。计算结果（203.00f）被转换成 int 类型，并储存在 i 中。
- 第 13 行和第 14 行：ch 的值（'D'，或 68）被转换成浮点数，然后 2 乘以 ch。为了做加法，i 的值（203）被转换为浮点类型。计算结果（339.00）被储存在 f1 中。
- 第 15 行和第 16 行：演示了类型降级的示例。把 ch 设置为一个超出其类型范围的值，忽略额外的位后，最终 ch 的值是字符 s 的 ASCII 码。或者，更确切地说，ch 的值是  $1107 \% 265$ ，即 83。
- 第 17 行和第 18 行：演示了另一个类型降级的示例。把 ch 设置为一个浮点数，发生截断后，ch 的值是字符 P 的 ASCII 码。

## 5.5.1 强制类型转换运算符

通常，应该避免自动类型转换，尤其是类型降级。但是如果能小心使用，类型转换也很方便。我们前面讨论的类型转换都是自动完成的。然而，有时需要进行精确的类型转换，或者在程序中表明类型转换的意图。这种情况下要用到强制类型转换（cast），即在某个量的前面放置用圆括号括起来的类型名，该类型名即是希望转换成的目标类型。圆括号和它括起来的类型名构成了强制类型转换运算符（cast operator），其通用形式是：

`(type)`

用实际需要的类型（如，long）替换 type 即可。

考虑下面两行代码，其中 mice 是 int 类型的变量。第 2 行包含两次 int 强制类型转换。

```
mice = 1.6 + 1.7;
mice = (int)1.6 + (int)1.7;
```

第 1 行使用自动类型转换。首先，1.6 和 1.7 相加得 3.3。然后，为了匹配 int 类型的变量，3.3 被类型转换截断为整数 3。第 2 行，1.6 和 1.7 在相加之前都被转换成整数（1），所以把 1+1 的和赋给变量 mice。本质上，两种类型转换都好不到哪里去，要考虑程序的具体情况再做取舍。

一般而言，不应该混合使用类型（因此有些语言直接不允许这样做），但是偶尔这样做也是有用的。C 语言的原则是避免给程序员设置障碍，但是程序员必须承担使用的风险和责任。

### 总结 C 的一些运算符

下面是我们学过的一些运算符。

赋值运算符：

= 将其右侧的值赋给左侧的变量

**算术运算符:**

|    |                                  |
|----|----------------------------------|
| +  | 将其左侧的值与右侧的值相加                    |
| -  | 将其左侧的值减去右侧的值                     |
| -  | 作为一元运算符，改变其右侧值的符号                |
| *  | 将其左侧的值乘以右侧的值                     |
| /  | 将其左侧的值除以右侧的值，如果两数都是整数，计算结果将被截断   |
| %  | 当其左侧的值除以右侧的值时，取其余数（只能应用于整数）      |
| ++ | 对其右侧的值加 1（前缀模式），或对其左侧的值加 1（后缀模式） |
| -- | 对其右侧的值减 1（前缀模式），或对其左侧的值减 1（后缀模式） |

**其他运算符:**

|        |                                                                                           |
|--------|-------------------------------------------------------------------------------------------|
| sizeof | 获得其右侧运算对象的大小（以字节为单位），运算对象可以是一个被圆括号括起来的类型说明符，如 sizeof(float)，或者是一个具体的变量名、数组名等，如 sizeof foo |
| (类型名)  | 强制类型转换运算符将其右侧的值转换成圆括号中指定的类型，如 (float)9 把整数 9 转换成浮点数 9.0                                   |

## 5.6 带参数的函数

现在，相信读者已经熟悉了带参数的函数。要掌握函数，还要学习如何编写自己的函数（在此之前，读者可能要复习一下程序清单 2.3 中的 butler() 函数，该函数不带任何参数）。程序清单 5.15 中有一个 pound() 函数，打印指定数量的#号（该符号也叫作编号符号或井号）。该程序还演示了类型转换的应用。

程序清单 5.15 pound.c 程序

```
/* pound.c -- 定义一个带一个参数的函数 */
#include <stdio.h>
void pound(int n); // ANSI 函数原型声明
int main(void)
{
    int times = 5;
    char ch = '!';
    float f = 6.0f;
    pound(times); // int 类型的参数
    pound(ch); // 和 pound((int)ch); 相同
    pound(f); // 和 pound((int)f); 相同

    return 0;
}

void pound(int n) // ANSI 风格函数头
{
    // 表明该函数接受一个 int 类型的参数
    while (n-- > 0)
        printf("#");
    printf("\n");
}
```

运行该程序后，输出如下：

```
#####
#####
#####
#####
```

首先，看程序的函数头：

```
void pound(int n)
```

如果函数不接受任何参数，函数头的圆括号中应该写上关键字 void。由于该函数接受一个 int 类型的参数，所以圆括号中包含一个 int 类型变量 n 的声明。参数名应遵循 C 语言的命名规则。

声明参数就创建了被称为形式参数 (*formal argument* 或 *formal parameter*，简称形参) 的变量。该例中，形式参数是 int 类型的变量 n。像 pound(10) 这样的函数调用会把 10 赋给 n。在该程序中，调用 pound(times) 就是把 times 的值 (5) 赋给 n。我们称函数调用传递的值为实际参数 (*actual argument* 或 *actual parameter*)，简称实参。所以，函数调用 pound(10) 把实际参数 10 传递给函数，然后该函数把 10 赋给形式参数 (变量 n)。也就是说，main() 中的变量 times 的值被拷贝给 pound() 中的新变量 n。

### 注意 实参和形参

在英文中，argument 和 parameter 经常可以互换使用，但是 C99 标准规定了：对于 actual argument 或 actual parameter 使用术语 argument (译为实参)；对于 formal argument 或 formal parameter 使用术语 parameter (译为形参)。为遵循这一规定，我们可以说形参是变量，实参是函数调用提供的值，实参被赋给相应的形参。因此，在程序清单 5.15 中，times 是 pound() 的实参，n 是 pound() 的形参。类似地，在函数调用 pound(times + 4) 中，表达式 times + 4 的值是该函数的实参。

变量名是函数私有的，即在函数中定义的函数名不会和别处的相同名称发生冲突。如果在 pound() 中用 times 替代 n，那么这个 times 与 main() 中的 times 不同。也就是说，程序中出现了两个同名的变量，但是程序可以区分它们。

现在，我们来学习函数调用。第 1 个函数调用是 pound(times)，times 的值 5 被赋给 n。因此，printf() 函数打印了 5 个井号和 1 个换行符。第 2 个函数调用是 pound(ch)。这里，ch 是 char 类型，被初始化为!字符，在 ASCII 中 ch 的数值是 33。但是 pound() 函数的参数类型是 int，与 char 不匹配。程序开头的函数原型在这里发挥了作用。原型 (*prototype*) 即是函数的声明，描述了函数的返回值和参数。pound() 函数的原型说明了两点：

- 该函数没有返回值 (函数名前面有 void 关键字)；
- 该函数有一个 int 类型的参数。

该例中，函数原型告诉编译器 pound() 需要一个 int 类型的参数。相应地，当编译器执行到 pound(ch) 表达式时，会把参数 ch 自动转换成 int 类型。在我们的系统中，该参数从 1 字节的 33 变成 4 字节的 33，所以现在 33 的类型满足函数的要求。与此类似，最后一次调用是 pound(f)，使得 float 类型的变量被转换成合适的类型。

在 ANSI C 之前，C 使用的是函数声明，而不是函数原型。函数声明只指明了函数名和返回类型，没有指明参数类型。为了向下兼容，C 现在仍然允许这样的形式：

```
void pound(); /* ANSI C 之前的函数声明 */
```

如果用这条函数声明代替 pound.c 程序中的函数原型会怎样？第 1 次函数调用，pound(times) 没问题，因为 times 是 int 类型。第 2 次函数调用，pound(ch) 也没问题，因为即使缺少函数原型，C 也会把 char 和 short 类型自动升级为 int 类型。第 3 次函数调用，pound(f) 会失败，因为缺少函数原型，

`float` 会被自动升级为 `double`, 这没什么用。虽然程序仍然能运行, 但是输出的内容不正确。在函数调用中显式使用强制类型转换, 可以修复这个问题:

```
pound ((int)f); // 把 f 强制类型转换为正确的类型
```

注意, 如果 `f` 的值太大, 超过了 `int` 类型表示的范围, 这样做也不行。

## 5.7 示例程序

程序清单 5.16 演示了本章介绍的几个概念, 这个程序对某些人很有用。程序看起来很长, 但是所有的计算都在程序的后面几行中。我们尽量使用大量的注释, 让程序看上去清晰明了。请通读该程序, 稍后我们会分析几处要点。

程序清单 5.16 running.c 程序

```
// running.c -- A useful program for runners
#include <stdio.h>

const int S_PER_M = 60;           // 1分钟的秒数
const int S_PER_H = 3600;          // 1小时的分钟数
const double M_PER_K = 0.62137;   // 1公里的英里数

int main(void)
{
    double distk, distm;      // 跑过的距离(分别以公里和英里为单位)
    double rate;               // 平均速度(以英里/小时为单位)
    int min, sec;              // 跑步用时(以分钟和秒为单位)
    int time;                 // 跑步用时(以秒为单位)
    double mtime;              // 跑1英里需要的时间, 以秒为单位
    int mmin, msec;            // 跑1英里需要的时间, 以分钟和秒为单位

    printf("This program converts your time for a metric race\n");
    printf("to a time for running a mile and to your average\n");
    printf("speed in miles per hour.\n");
    printf("Please enter, in kilometers, the distance run.\n");
    scanf("%lf", &distk);       // %lf 表示读取一个 double 类型的值
    printf("Next enter the time in minutes and seconds.\n");
    printf("Begin by entering the minutes.\n");
    scanf("%d", &min);
    printf("Now enter the seconds.\n");
    scanf("%d", &sec);

    time = S_PER_M * min + sec; // 把时间转换成秒
    distm = M_PER_K * distk;   // 把公里转换成英里
    rate = distm / time * S_PER_H; // 英里/秒 × 秒/小时 = 英里/小时
    mtime = (double) time / distm; // 时间/距离 = 跑1英里所用的时间
    mmin = (int) mtime / S_PER_M; // 求出分钟数
    msec = (int) mtime % S_PER_M; // 求出剩余的秒数

    printf("You ran %.2f km (%.2f miles) in %d min, %d sec.\n",
           distk, distm, min, sec);
    printf("That pace corresponds to running a mile in %d min, ",
           mmin);
    printf("%d sec.\nYour average speed was %.2f mph.\n", msec,
```

```

    rate);

    return 0;
}

```

程序清单 5.16 使用了 min\_sec 程序（程序清单 5.9）中的方法把时间转换成分钟和秒，除此之外还使用了类型转换。为什么要进行类型转换？因为程序在秒转换成分钟的部分需要整型参数，但是在公里转换成英里的部分需要浮点运算。我们使用强制类型转换运算符进行了显式转换。

实际上，我们曾经利用自动类型转换编写这个程序，即使用 int 类型的 mtime 来强制时间计算转换成整数形式。但是，在测试的 11 个系统中，这个版本的程序在 1 个系统上无法运行，这是由于编译器（版本比较老）没有遵循 C 规则。而使用强制类型转换就没有问题。对读者而言，强制类型转换强调了转换类型的意思，对编译器而言也是如此。

下面是程序清单 5.16 的输出示例：

```

This program converts your time for a metric race
to a time for running a mile and to your average
speed in miles per hour.

Please enter, in kilometers, the distance run.
10.0
Next enter the time in minutes and seconds.
Begin by entering the minutes.
36
Now enter the seconds.
23
You ran 10.00 km (6.21 miles) in 36 min, 23 sec.
That pace corresponds to running a mile in 5 min, 51 sec.
Your average speed was 10.25 mph.

```

## 5.8 关键概念

C 通过运算符提供多种操作。每个运算符的特性包括运算对象的数量、优先级和结合律。当两个运算符共享一个运算对象时，优先级和结合律决定了先进行哪项运算。每个 C 表达式都有一个值。如果不了解运算符的优先级和结合律，写出的表达式可能不合法或者表达式的值与预期不符。这会影响你成为一名优秀的程序员。

虽然 C 允许编写混合数值类型的表达式，但是算术运算要求运算对象都是相同的类型。因此，C 会进行自动类型转换。尽管如此，不要养成依赖自动类型转换的习惯，应该显式选择合适的类型或使用强制类型转换。这样，就不用担心出现不必要的自动类型转换。

## 5.9 本章小结

C 语言有许多运算符，如本章讨论的赋值运算符和算术运算符。一般而言，运算符需要一个或多个运算对象才能完成运算生成一个值。只需要一个运算对象的运算符（如负号和 sizeof）称为一元运算符，需要两个运算对象的运算符（如加法运算符和乘法运算符）称为二元运算符。

表达式由运算符和运算对象组成。在 C 语言中，每个表达式都有一个值，包括赋值表达式和比较表达式。运算符优先级规则决定了表达式中各项的求值顺序。当两个运算符共享一个运算对象时，先进行优先级高的运算。如果运算符的优先级相等，由结合律（从左往右或从右往左）决定求值顺序。

大部分语句都以分号结尾。最常用的语句是表达式语句。用花括号括起来的一条或多条语句构成了复合语句（或称为块）。while 语句是一种迭代语句，只要测试条件为真，就重复执行循环体中的语句。

在 C 语言中，许多类型转换都是自动进行的。当 char 和 short 类型出现在表达式里或作为函数的参数（函数原型除外）时，都会被升级为 int 类型；float 类型在函数参数中时，会被升级为 double 类型。在 K&R C（不是 ANSI C）下，表达式中的 float 也会被升级为 double 类型。当把一种类型的值赋给另一种类型的变量时，值将被转换成与变量的类型相同。当把较大类型转换成较小类型时（如，long 转换成 short，或 double 转换成 float），可能会丢失数据。根据本章介绍的规则，在混合类型的运算中，较小类型会被转换成较大类型。

定义带一个参数的函数时，便在函数定义中声明了一个变量，或称为形式参数。然后，在函数调用中传入的值会被赋给这个变量。这样，在函数中就可以使用该值了。

## 5.10 复习题

复习题的参考答案在附录 A 中。

1. 假设所有变量的类型都是 int，下列各项变量的值是多少：

- a.  $x = (2 + 3) * 6;$
- b.  $x = (12 + 6) / 2 * 3;$
- c.  $y = x = (2 + 3) / 4;$
- d.  $y = 3 + 2 * (x = 7 / 2);$

2. 假设所有变量的类型都是 int，下列各项变量的值是多少：

- a.  $x = (\text{int}) 3.8 + 3.3;$
- b.  $x = (2 + 3) * 10.5;$
- c.  $x = 3 / 5 * 22.0;$
- d.  $x = 22.0 * 3 / 5;$

3. 对下列各表达式求值：

- a.  $30.0 / 4.0 * 5.0;$
- b.  $30.0 / (4.0 * 5.0);$
- c.  $30 / 4 * 5;$
- d.  $30 * 5 / 4;$
- e.  $30 / 4.0 * 5;$
- f.  $30 / 4 * 5.0;$

4. 请找出下面的程序中的错误。

```
int main(void)
{
    int i = 1,
        float n;
    printf("Watch out! Here come a bunch of fractions!\n");
    while (i < 30)
        n = 1/i;
        printf(" %f", n);
    printf("That's all, folks!\n");
    return;
}
```

5. 这是程序清单 5.9 的另一个版本。从表面上看，该程序只使用了一条 scanf() 语句，比程序清单

5.9 简单。请找出不如原版之处。

```
#include <stdio.h>
#define S_TO_M 60
int main(void)
{
    int sec, min, left;

    printf("This program converts seconds to minutes and ");
    printf("seconds.\n");
    printf("Just enter the number of seconds.\n");
    printf("Enter 0 to end the program.\n");
    while (sec > 0) {
        scanf("%d", &sec);
        min = sec/S_TO_M;
        left = sec % S_TO_M;
        printf("%d sec is %d min, %d sec. \n", sec, min, left);
        printf("Next input?\n");
    }
    printf("Bye!\n");
    return 0;
}
```

6. 下面的程序将打印出什么内容？

```
#include <stdio.h>
#define FORMAT "%s! C is cool!\n"
int main(void)
{
    int num = 10;

    printf(FORMAT, FORMAT);
    printf("%d\n", ++num);
    printf("%d\n", num++);
    printf("%d\n", num--);
    printf("%d\n", num);
    return 0;
}
```

7. 下面的程序将打印出什么内容？

```
#include <stdio.h>
int main(void)
{
    char c1, c2;
    int diff;
    float num;

    c1 = 'S';
    c2 = 'O';
    diff = c1 - c2;
    num = diff;
    printf("%c%c%c:%d %3.2f\n", c1, c2, c1, diff, num);
    return 0;
}
```

8. 下面的程序将打印出什么内容？

```
#include <stdio.h>
#define TEN 10
int main(void)
```

```

{
    int n = 0;

    while (n++ < TEN)
        printf("%5d", n);
    printf("\n");
    return 0;
}

```

9. 修改上一个程序，使其可以打印字母 a~g。

10. 假设下面是完整程序中的一部分，它们分别打印什么？

a.

```

int x = 0;

while (++x < 3)
    printf("%4d", x);

```

b.

```

int x = 100;

while (x++ < 103)
    printf("%4d\n", x);
    printf("%4d\n", x);

```

c.

```

char ch = 's';

while (ch < 'w')
{
    printf("%c", ch);
    ch++;
}
printf("%c\n", ch);

```

11. 下面的程序会打印出什么？

```

#define MESG "COMPUTER BYTES DOG"
#include <stdio.h>
int main(void)
{
    int n = 0;

    while (n < 5)      ↵
        printf("%s\n", MESG);
        n++;
    printf("That's all.\n");
    return 0;
}

```

12. 分别编写一条语句，完成下列各任务（或者说，使其具有以下副作用）：

- 将变量 x 的值增加 10
- 将变量 x 的值增加 1
- 将 a 与 b 之和的两倍赋给 c
- 将 a 与 b 的两倍之和赋给 c

13. 分别编写一条语句，完成下列各任务：

- a. 将变量  $x$  的值减少 1
- b. 将  $n$  除以  $k$  的余数赋给  $m$
- c.  $q$  除以  $b$  减去  $a$ , 并将结果赋给  $p$
- d.  $a$  与  $b$  之和除以  $c$  与  $d$  的乘积, 并将结果赋给  $x$

## 5.11 编程练习

1. 编写一个程序, 把用分钟表示的时间转换成用小时和分钟表示的时间。使用#define 或 const 创建一个表示 60 的符号常量或 const 变量。通过 while 循环让用户重复输入值, 直到用户输入小于或等于 0 的值才停止循环。
2. 编写一个程序, 提示用户输入一个整数, 然后打印从该数到比该数大 10 的所有整数(例如, 用户输入 5, 则打印 5~15 的所有整数, 包括 5 和 15)。要求打印的各值之间用一个空格、制表符或换行符分开。
3. 编写一个程序, 提示用户输入天数, 然后将其转换成周数和天数。例如, 用户输入 18, 则转换成 2 周 4 天。以下面的格式显示结果:

18 days are 2 weeks, 4 days.

通过 while 循环让用户重复输入天数, 当用户输入一个非正值时(如 0 或 -20), 循环结束。

4. 编写一个程序, 提示用户输入一个身高(单位: 厘米), 并分别以厘米和英寸为单位显示该值, 允许有小数部分。程序应该能让用户重复输入身高, 直到用户输入一个非正值。其输出示例如下:

```
Enter a height in centimeters: 182
182.0 cm = 5 feet, 11.7 inches
Enter a height in centimeters (<=0 to quit): 168.7
168.0 cm = 5 feet, 6.4 inches
Enter a height in centimeters (<=0 to quit): 0
bye
```

5. 修改程序 addemup.c(程序清单 5.13), 你可以认为 addemup.c 是计算 20 天里赚多少钱的程序(假设第 1 天赚\$1、第 2 天赚\$2、第 3 天赚\$3, 以此类推)。修改程序, 使其可以与用户交互, 根据用户输入的数进行计算(即, 用读入的一个变量来代替 20)。
6. 修改编程练习 5 的程序, 使其能计算整数的平方和(可以认为第 1 天赚\$1、第 2 天赚\$4、第 3 天赚\$9, 以此类推, 这看起来很不错)。C 没有平方函数, 但是可以用  $n * n$  来表示  $n$  的平方。
7. 编写一个程序, 提示用户输入一个 double 类型的数, 并打印该数的立方值。自己设计一个函数计算并打印立方值。main() 函数要把用户输入的值传递给该函数。
8. 编写一个程序, 显示求模运算的结果。把用户输入的第一个整数作为求模运算符的第 2 个运算对象, 该数在运算过程中保持不变。用户后面输入的数是第一个运算对象。当用户输入一个非正值时, 程序结束。其输出示例如下:

```
This program computes moduli.
Enter an integer to serve as the second operand: 256
Now enter the first operand: 438
438 % 256 is 182
Enter next number for first operand (<= 0 to quit): 1234567
1234567 % 256 is 135
Enter next number for first operand (<= 0 to quit): 0
Done
```

9. 编写一个程序, 要求用户输入一个华氏温度。程序应读取 double 类型的值作为温度值, 并把该值

作为参数传递给一个用户自定义的函数 Temperatures()。该函数计算摄氏温度和开氏温度，并以小数点后面两位数字的精度显示 3 种温度。要使用不同的温标来表示这 3 个温度值。下面是华氏温度转摄氏温度的公式：

$$\text{摄氏温度} = 5.0 / 9.0 * (\text{华氏温度} - 32.0)$$

开氏温标常用于科学研究，0 表示绝对零，代表最低的温度。下面是摄氏温度转开氏温度的公式：

$$\text{开氏温度} = \text{摄氏温度} + 273.16$$

Temperatures() 函数中用 const 创建温度转换中使用的变量。在 main() 函数中使用一个循环让用户重复输入温度，当用户输入 q 或其他非数字时，循环结束。scanf() 函数返回读取数据的数量，所以如果读取数字则返回 1，如果读取 q 则不返回 1。可以使用==运算符将 scanf() 的返回值和 1 作比较，测试两值是否相等。

