Practical C Programming

第三版

实用C语言编程

O'REILLY®

中国电力出版社

Steve Oualline 著

郭大海 译

TP312C
100

00011845

实用C语言编程

JS95/12

Steve Oualline 著

郭大海 译

C0487913

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

中国电力出版社

图书在版编目（CIP）数据

实用C语言编程/（美）奥莱恩著；郭大海译 . - 北京：中国电力出版社，2000. 5
书名原文：Practical C Programming
ISBN 7-5083-0308-3

Ⅰ. 实… Ⅱ. ①奥… ②郭… Ⅲ. C语言程序设计 Ⅳ. TP312
中国版本图书馆CIP数据核字（2000）第08994号

北京市版权局著作权合同登记
图字：01-1999-3746号

© 1997 by O'Reilly & Associates, Inc.

Simplified Chinese Edition, jointly published by O'Reilly & Associates, Inc. and China Electric Power Press, 2000. Authorized translation of the English edition,1997 O'Reilly & Associates, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly & Associates, Inc. 出版 1997。

简体中文版由中国电力出版社出版 2000，英文原版的翻译得到O'Reilly & Associates, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly & Associates, Inc.的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

书　　名/ 实用C语言编程
书　　号/ ISBN 7-5083-0308-3
责任编辑/ 刘君，关敏，蒙虎
封面设计/ Ellie Volckhausen，Hanna Dyer，张健
出版发行/ 中国电力出版社
地　　址/ 北京三里河路6号（邮政编码100044）
经　　销/ 全国新华书店
印　　刷/ 北京市地矿印刷厂
开　　本/ 787毫米×1092毫米　16开本　31印张　310千字
版　　次/ 2000年5月第一版　2000年5月第一次印刷
印　　数/ 0001-5000册
定　　价/ 49.00元（册）

目录

前言 1

第一部分 基础 11

第一章 什么是C? 13

编程原理 14

C语言简史 17

C如何工作 17

如何学习C 19

第二章 编程基础 21

程序从概念到运行 21

编写一个真正的程序 22

使用命令行编译器编程 23

使用集成开发环境（IDE）编程 27

获取UNIX帮助 45

获取集成开发环境帮助 45

集成开发环境菜单 45
编程练习 48

第三章 风格 49

基础编码练习 54
编码盲从 56
缩进与编码格式 56
清晰 57
简明 58
小结 59

第四章 基本定义与表达式 60

程序要素 60
程序的基本结构 61
简单表达式 62
变量和存储 64
变量定义 65
整型 66
赋值语句 66
printf 函数 68
浮点型 70
浮点数与整数的除法运算 70
字符 73
答案 74
编程练习 75

第五章 数组、修饰符与读取数字 76

数组 76
串 78
读取串 81

多维数组 84
读取数字 86
变量初始化 88
整型 90
浮点型 92
常量说明 93
十六进制与八进制常量 93
快捷运算符 94
副作用 95
++x 或 x++ 96
更多的副作用问题 97
答案 98
编程练习 99

第六章 条件和控制语句 101

if 语句 101
else 语句 102
怎样避免误用 strcmp 函数 104
循环语句 104
while 语句 105
break 语句 107
continue 语句 108
随处赋值的副作用 109
答案 111
编程练习 111

第七章 程序设计过程 113

设置 115
程序规范 116
代码设计 116

原型 118
Makefile 119
测试 123
调试 124
维护 126
修改 126
代码分析 127
注释程序 128
使用调试器 128
用文本编辑器浏览 128
增加注释 128
编程练习 131

第二部分 简单程序设计 133

第八章 更多的控制语句 135

for 语句 135
switch 语句 139
switch, break 和 continue 145
答案 145
编程练习 147

第九章 变量作用域和函数 149

作用域和类 149
函数 153
无参数的函数 157
结构化程序设计 158
递归 160
答案 161
编程练习 162

第十章 C 预处理器 163

#define 语句 163
条件编译 170
包含文件 173
带参数的宏 174
高级特征 176
小结 176
答案 177
编程练习 180

第十一章 位运算 181

位运算符 183
与运算符（&） 183
按位或（\） 186
按位异或（^） 187
非运算符（~） 187
左移与右移运算符（<<, >>） 188
设置、清除和检测位 190
位图图形 194
答案 200
编程练习 201

第十二章 高级类型 202

结构 202
联合 205
typedef 207
枚举类型 209
强制类型转换 210
位字段或紧缩结构 210
结构数组 212

小结 213
编程练习 213

第十三章 简单指针 215

函数自变量指针 220
常量指针 222
指针和数组 224
如何不使用指针 229
用指针分隔字符串 231
指针和结构 235
命令行参数 236
编程练习 242
答案 242

第十四章 文件输入/输出 245

转换程序 249
二进制和ASCII码文件 252
行尾难题 253
二进制I/O 255
缓冲问题 257
非缓冲I/O 258
设计文件格式 264
答案 266
编程练习 267

第十五章 调试和优化 268

调试 268
交互调试器 280
调试一个二分查找程序 285
实时运行错误 297

公开声明调试方法 299
优化 300
答案 309
编程练习 309

第十六章 浮点数 310

浮点数格式 310
浮点数加法/减法 312
乘法 313
除法 313
上溢和下溢 314
舍入误差 314
精度 315
舍入误差最小化 316
判定精度 317
精度和速度 318
幂级数 319
编程练习 321

第三部分 高级编程观念 323

第十七章 高级指针 325

指针和结构 325
free 函数 329
链表 330
结构指针运算符 333
顺序链表 334
双向链表 337
树 340

树的打印 344
程序的剩余部分 345
象棋程序中用到的数据结构 349
答案 351
编程练习 353

第十八章 模块化编程 354

模块 354
公用和专用 355
extern 修饰符 356
头文件 358
模块体 361
使用无限数组的程序 361
用于多文件的 Makefile 364
使用无限数组 368
把一项任务分成模块 376
模块划分实例：文本编辑器 376
编译器 378
电子表格 380
模块设计准则 380
编程练习 380

第十九章 旧式编译器 382

K&R 风格的函数 382
库的发展 386
遗漏的特性 386
Free/Malloc 的发展 387
lint 388
答案 388

第二十章 移植问题 391

模块化 391
字大小 392
字节顺序问题 392
对齐问题 393
NULL 指针问题 395
文件名问题 396
文件类型 397
小结 397
答案 398

第二十一章 C 内的“角落” 399

do/while 399
goto 400
?: 指令 401
, 运算符 402
不稳定限定词 402
答案 402

第二十二章 组合到一起 403

需求 403
规范说明 404
代码设计 406
编码 412
功能描述 412
扩展 414
测试 415
修改 416
最后的警告 416

程序文件 …… 416
编程练习 …… 443

第二十三章 程序设计格言 …… 444
概述 …… 444
设计 …… 445
定义 …… 445
switch 语句 …… 445
预处理器 …… 446
风格 …… 446
编译 …… 446
最后的注解 …… 447
答案 …… 447

第四部分 其他语言特性 …… 449

附录一 ASCII 表 …… 451

附录二 范围和参数传递转换 …… 453

附录三 运算符优先规则 …… 455

附录四 使用幂级数计算正弦函数的程序 …… 457

词汇表 …… 463

前言

本书讲述如何使用C语言进行真正的编程。C是目前软件开发者们最主要的编程语言，这也是它受到广泛传播并且成为标准的原因。现在也有新的编程语言出现，如C++，但这些语言仍然在演化中，C仍然是进行健壮的、可移植编程的首选语言。

本书侧重介绍在实际编程过程中需要知道的技巧问题。它不仅告诉你有关C语言的机制，也告诉你用C进行编程的整个过程（包括程序的含义、设计、编码、方法、调试、发布、文档、维护和版本更新等）。

本书也介绍了用C编程的风格和艺术。要写出一个好的程序，需要做许多工作，而不仅仅是敲一些代码。这是一门有关写作和编程的技巧，并将其合二为一的艺术，从而使你写出的程序成为一件杰作。这样创作出来的作品才是真正的艺术。一个非常好的程序不仅仅要功能正确，而且应该简单易读。在程序中允许加进一些评论性的描述性文字。当清晰的评注被加进程序中时，这样的程序就会得到其他人的高度评价。

程序应该尽可能简单。程序员应该避免玩一些聪明的技巧。在本书中我们强调简单、实用的原则。例如，在C中有15条关于操作符的规则，但它们可以被简化为下面两个：

1. 先乘除，后加减。

2. 括号优先于其他一切运算。

考虑两个程序：一个是由聪明的程序员使用了所有技巧编写的程序，程序没有包含任何注解，但是可以运行；另外一个程序有很好的注解和结构，但是它不能运行。两个程序哪一个更有用呢？从长远的角度看，应该是那个不能运行的程序，因为它存在的问题可以被解决掉。尽管那个聪明的程序员的程序现在可以运行，但迟早该程序会被修改的。那么到那时，最坏的事情就会出现 —— 你不得不去修改一个充满技巧、但没有任何注解的程序。

这本书是针对那些没有编程经验，或者已经对C语言有所了解但是还想进一步提高他们的编程风格和程序可靠性的人所作的。在使用这本书之前，你应该知道如何使用计算机并且知道如何使用一些基本的工具，诸如文本编辑器和文件系统。

对于那些想在UNIX操作系统上用generic cc编译器或者自由软件基金会的gcc编译器编写和运行程序的读者，我们给出了一些特别的指导。对于MS-DOS/Windows 的用户，我们也给出了包括Borland C++, Turbo C++和 Microsoft Visual C++的使用说明(这些编译器可以对C 和 C++的代码进行编译)。在本书中也给出了一些使用编程工具 make 进行自动程序编译的例子。

这本书是如何组织的

在你学会跑步之前，你必须先学会走路。

在第一部分 —— 基础中，你将会学习如何走路。本章只讲述如何编写一个非常简单的程序。读者从编程技巧和编程风格入手。下一步，你将学习如何使用变量和非常简单的判断和控制语句。在第七章程序设计过程中，我们才将编写真正程序的完整过程展现给读者。

在第二部分 —— 简单程序设计中，我们介绍编程时所有需要的简单语句和运算符。读者还将学习到如何把这些语句组织成为简单函数的方法。

在第三部分 —— 高级编程概念中，我们将介绍一些基本的说明和语句来构造高

级类型的方法，如构造结构（struct）、联合（union）和类（class）。这一部分还将向读者介绍指针的概念。

最后，在第四部分我们还会介绍一些其他的语言特性。

每章详细介绍

第一章“什么是C？”我们给出了一个简单的描述以及它的用法。在本章中还介绍了该语言的一些背景知识。

第二章“编程基础”。在本章中介绍了基本的编程过程，并教会你如何写出一个非常简单的程序

第三章“风格”。讨论一些编程的风格，介绍如何给程序加注解，以及如何编写简洁明了的程序代码。

第四章“基本定义和表达式”，介绍简单的C语句。同时也详细介绍了基本的变量和赋值语句，以及一些算术运算符 +、-、*、/ 和 %。

第五章“数组、修饰符与读取数字”，介绍了数组和更复杂的变量。还介绍了诸如 ++ 和 %= 这样的快捷运算符。

第六章“条件和控制语句”，解释了一些简单的判断语句，包括 **if**, **else**, 和 **for**。有关 == 和 = 的问题也有相关的介绍。

第七章“程序设计过程”，本章将引导读者通过创建一个简单程序的所有必要步骤，说明从文档到程序发布的整个过程。另外还讨论了结构化程序设计、快速建模和调试。

第八章“更多的控制语句”，描述控制语句的其余内容，包括 **while**、**break** 和 **continue**，也包括了有关 **switch** 语句的讨论

第九章“变量作用域和函数”，介绍了局部变量、函数和参数。

第十章“C预处理器”，描述了C预处理器给程序员写代码提供了很大的自由度。本章也给程序员讲解了容易制造混乱的各种情况，并对在预处理器中避免出现这些情况的简单规则进行了描述。

第十一章“位运算”，讨论以位为基础的逻辑C运算符。

第十二章“高级类型”，解释结构和其他高级类型，还包括sizeof运算符和枚举类型。

第十三章“简单指针”，介绍C指针变量并列出了它们的用法。

第十四章“文件输入/输出”，描述缓冲和非缓冲的输入/输出。讨论相对于二进制文件的ASCII代码，以及怎样生成一个简单文件。

第十五章“调试和优化”，描述怎样调试一个程序，以及怎样使用一个交互调试器。本章不仅列出了怎样调试程序，也介绍了怎样写一个易于调试的程序，同时也描述了许多优化技巧，这些技巧能使你的程序运行得更快和更有效。

第十六章“浮点数”，使用简单十进制浮点数的格式来介绍浮点数问题，如越界错误、精度损失、上溢和下溢。

第十七章“高级指针”，描述指针的高级应用，即构造动态结构，如链表和树。

第十八章“模块化编程”，描述应用模块化程序设计技术，如何把程序分成几个模块，并对make程序做了更详尽的解释。

第十九章“旧式编译器”，描述标准C以前的老式C语言。虽然这种编译器今天已很少，但在它们的基础上曾写出了很多代码而且还有大量的程序仍在使用旧语法。

第二十章“移植问题”，描述移植程序时会发生的问题（从一台机器移到另一台机器上运行）。

第二十一章“C内的‘角落’”，描述do/while语句，逗号运算符，以及?和:操作符。

第二十二章“组合到一起”，描述一个复杂的程序从概念到完成所需的详细步骤，重点强调了信息隐藏和模块化编程技术。

第二十三章“程序设计格言”，列出了一些编程格言以帮助读者构建优良的C程序。

附录一“ASCII表”，列出了几乎已在全世界通用的ASCII字符集的八进制、十六进制和十进制表示法。

附录二“范围和参数传递转换”，列出了用不同大小的内存分配来处理数据时可能会出现的问题。

附录三“运算符优先规则”，列出了一些很难记忆的规则。当你遇到粗心人写的未使用足够括号的代码时，它会给你提供帮助。

附录四“使用幂级数计算正弦函数的程序”，列出了浮点（实）数的操作，这是在本书其他章节中未给予足够重视的一个问题。

“词汇表”定义了本书中使用的一些技术性的专业词语。

学习计算机语言最好的方法就是通过编写程序和调试程序。你可能在凌晨两点还在挥汗如雨地调试程序，而最终发现错误仅仅是将“==”敲成了“=”，这样的经验会对你将来大有益处。在本书中有大量的编程例子可供参考。有些例子可能不能运行，希望读者能够动手来解决它们。我们鼓励你亲自动手来实验每个实例并运行和调试它们。这些练习将有助于你知道如何在小程序中发现一般性的错误，从而在大的程序中能够较为容易地发现这样的错误。在每章的最后，都附有一些问题的答案。另外，在很多章的后面，你都会找到“编程练习”一节。这些章节中的练习可能会被用在一些编程课程中来测试你的C编程知识。

第三版说明

自从《Practical C Programming》（译注）第一版出版后，C语言本身已经有了很大的变化。回想当初，ANSI编译器还很少见，而且当时K&R的语法结构很常见。但是现在情况完全相反。

第三版反映了整个业界已经将标准转向ANSI标准的编译器。本书中的所有程序和范例都已经更新为ANSI标准。实际上，较老的K&R语法仅在第十九章中有一些讨论。

本书中的一些其他变化和新添内容有：

- 添加了一些其他的针对generic UNIX 编译器、自由软件基金会的gcc编译器、Borland C++、Turbo C++ 和 Microsoft Visual C++ 的编译器介绍。
- 第二十二章完全进行了重写。在本章中使用了一个统计程序，这对大多数的读者来讲是极有益的。

最后我要声明的是，我是一个很实际的人。我希望你能明白我的意思，语言是为目的服务的。在整本书中，我使用“他”来指某个程序员。少数人以“政治性纠正”的理由来表明这种用法带有大男子主义色彩。他们同时也声明在书中的某些段落中带有种族歧视，我认为这是不正确的。

请注意，当我使用“他”时，我是指某个程序员，并不代表性别。其次，当我建议一些功底很差的程序员应该被枪毙时，请读者勿从字面理解其意。

我的风格是以清楚明了、简单并带有一点幽默感的方式进行交流。如果这对你有所冒犯的话，在此表示歉意。

印刷约定

本书排版时采用了以下约定：

译注：　即本书英文版的第一版（1991年7月出版）。

Italic（斜体）

用于目录和文件名，以及强调新的术语和概念时使用。同时也用于标注范例中的注解。

Bold（粗体）

用于C的关键字。

`Constant Width`（固定宽度）

在正文和范例中表示运算符、变量和从命令（或程序）的输出。

`Constant Bold`（加粗固定宽度）

用于范例中由用户输入的命令或其他一些文字。（例如，**`rm foo`** 表示要让你键入“rm foo”）

`Constant Italic`（固定宽度斜体）

用于在范例中表示需要根据上下文而替换的变量。例如*`filename`*变量，就需要被一些实际的文件名所替换。

“”

用于标识系统信息或代码段中的解释性文字。

%

UNIX 外壳提示符。

[ ]

方括号内的值用于表示程序语法中的可选项。

. . .

用于代表需省略的文字，通常指程序或命令输出。

符号 CTRL-X 或 ^X 表示使用控制符。此符号表示在你按住“control”键的同时按住“x”键。一些其他的符号也有类似的用法。RETURN 表示回车键。

取得源代码

本书中的所有练习均有电子版，你可以通过使用FTP和FTPMAIL获得。如果你可以直接连接Internet，那么可以使用FTP。如果你不能连接Internet，但可以发送和接收Internet上的电子邮件，那么你可以使用FTPMAIL来获得这些练习题的电子版本。

FTP

如果你能够连接到Internet上，最容易获取练习题的电子版本的方法是在web浏览器上使用FTP。只需在你的浏览器的URL栏中输入：

ftp://ftp.oreilly.com/published/oreilly/nutshell/practical_c3/examples.tar.gz

如果你没有web浏览器，你可以使用命令行的FTP客户端软件来获取，在Windows NT（或Windows 9x）平台上均可。如果你是在Windows上使用，你应下载*examples.zip*而不是*examples.tar.gz*文件。

```
% ftp ftp.oreilly.com
Connected to ftp.oreilly.com.
220 ftp.oreilly.com FTP server (Version 6.34 Thu Oct 22 14:32:01 EDT 1992)
ready.
Name (ftp.oreilly.com:username): anonymous
331 Guest login ok, send e-mail address as password.
Password:username@hostname      Use your username and host here
230 Guest login ok, access restrictions apply.
ftp> cd /published/oreilly/nutshell/practical_c3
250 CWD command successful.
ftp> binary
200 Type set to I.
ftp> get examples.tar.gz
200 PORT command successful.
150 Opening BINARY mode data connection for examples.tar.gz (xxxx bytes).
226 Transfer complete. local: exercises remote: exercises
xxxx bytes received in xxx seconds (xxx Kbytes/s)
ftp> quit
221 Goodbye.
%
```

FTPMAIL

任何人只要能通过Internet网站收发电子邮件，就能使用FTPMAIL。以下就是实际的使用方法。

发一份电子邮件到*ftpmail@online.oreilly.com*，在信件的信体正文中写上要执行的FTP命令。服务器会运行匿名FTP，并且会自动回复邮件给你。想取得完整的说明文件，只要发一封空白标题的电子邮件，信体正文写上"help"即可。下面是一个获取示例文件的例子。该命令会发送一个指定目录中的文件列表。该列表会对将来文件的更新有所帮助。如果你是在Windows平台上使用，那么你应该下载*examples.zip*而不是*examples.tar.gz*。

```
Subject:
reply-to username@hostname   (Message Body) Where you want files mailed
open
cd /published/oreilly/nutshell/practical_c3
dir
mode binary
uuencode
get examples.tar.gz
quit
.
```

信息后面可以有签名文件——只要它出现在"quit"后面。

建议和评论

奥莱理软件（北京）有限公司建立了专门的站点，并回答用户提出的关于本书的问题。如果你发现了本书的错误，请将你的发现通过以下电子邮件地址告诉我们，以便我们在本书的下一个版本中更正或者直接采用你的建议：

info@mail.oreilly.com.cn

你可以在网上找到我们：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

我要感谢我的父亲在本书的编写过程中给予我的帮助，同时也要感谢Arthur Marquez在本书排版上的帮助。

我还要感谢在Writers' Haven和书店的工作人员：Pearl、Alex和Clyde，是他们为我提供了坚持不懈的支持。在本书的编辑过程中也得到了Peg Kovar的大力帮助。也要特别感谢Dale Dougherty，正是他教会我如何将本书分开，并正确地归并到一起。我还要感谢那些在O'Reilly & Associates工作的产品部门的工作人员——特别是Rosanne Wagger和Mike Sierra，正是他们制作完成了本书。最后，我要感谢Jean Graham，她多年来一直支持我的写作工作。

第三版致谢

特别要感谢本书的技术编辑Andy Oram，同时也要感谢O'Reilly & Associates的工作人员，Nicole Gipson Arigo是本书的产品经理，Clairemarie Fisher O'Leary和Sheryl Avruch负责本书的质量控制。Mike Sierra在工作中应用了一些制作工具。Chris Reilley和Robert Romano为本书制作了精美的插图。Nancy Priest为本书进行了内文设计，Edie Freedman设计了本书的封面。Bench Productions, Inc.负责本书的产品制作、排版和提供索引。

第一部分

基础

这部分内容将讲解基础性的C语言语句，学完后你可以编写设计良好而且构思极佳的C程序。较早地强调风格问题，目的是可以让你立即应用良好的编程风格着手写程序。尽管整个部分读者都被局限在短小的程序中，但这些都是写得很好的程序。

- 第一章“什么是C?”，对C语言及其用法的简单描述。本章包括C语言发展历史的一些背景材料。
- 第二章“编程基础”，解释基本编程过程，提供编写简单程序所需的足够信息。
- 第三章“风格”，讨论编程风格，包括程序的注释以及简单明了的代码的编写。
- 第四章“基本定义和表达式”，介绍简单C语句，详细讲述了基本变量和赋值语句，以及算术运算符 +、-、*、/ 和 %。
- 第五章“数组、修饰符与读取数字”，包括数组和更复杂的变量。速记运算符如 ++ 和 %= 也有涉及。
- 第六章“条件和控制语句”，解释简单判断语句，包括 **if, else** 和 **for**，还包括 = = 和 = 的讨论。

- 第七章“程序设计过程”，帮助读者完成从规范说明到交付使用的所有必要步骤来创建一个简单程序。另外还讨论了结构化程序设计、快速建模和调试。

本章内容

- 编程原理
- C语言简史
- C如何工作
- 如何学习C

第一章 什么是C?

咒骂是所有程序员都能理解的一门语言。

无名氏

在现代社会中，组织及处理信息的能力是成功的关键。设计计算机的目的就是为了快速而有效地处理大量信息，但是除非有人告诉计算机该干什么，否则它什么事情也不能做。

这正是C语言诞生的原因。它是一种能够让软件工程师与计算机进行有效对话的介于汇编语言与高级语言之间的编程语言。

C语言非常灵活而且适应性强。自1970年诞生之日起，它一直被用来开发各种各样的程序，包括用于微控制器的固化软件、操作系统、应用程序和图形程序。

C语言较为稳定，是目前世界上使用最为广泛的语言之一。C++是C语言改进后的产物，已经被用于各种软件的开发，同时仍在不断地完善。这种语言最初被称作带类的C语言，它增加了一些新的特性，其中最重要的特性就是引入了类。类是根据面向对象的程序设计（OOD）思想来创建的，它能使代码便于重复使用。

谈到C和C++哪一个更好？答案取决于所面向的用户。C++可以自动做大量的事情，如为变量自动调用构造函数和析构函数。这一功能使得一些类型的程序易于开发，但也使程序的静态检查变得困难。如果正运行嵌入式的控制应用程序，你还必须能准确地分辨出程序所要完成的功能。所以有些人认为C++更好，其原因是C++能自动化而C不能。另外一些人则因为几乎同样的原因而推崇C语言。

但是，C++毕竟是一种相对较新且仍在不断完善的语言。C语言的代码比C++多得多，而且C语言的代码也将会继续使用并升级，因此C语言的使用周期会很长

编程原理

与计算机进行通信并非易事，它需要精确而详细的指令。如果我们能用英语编写程序会不会简单一些呢？我们可以告诉计算机，"把我的帐单和存款汇总，然后告诉我合计数"，计算机就会自动把我们的支票簿结算清楚。

但是当书写精确指令的时候，你会发现英语是一种很糟糕的语言。它充满模棱两可，而且不精确。计算机之母Grace Hopper女士，曾经评论过她在一个香波瓶子上发现的指令：

洗发

冲洗

重复

她按照指令去做，结果把洗发水用光了。（洗-冲-重复，洗-冲-重复，洗-冲-重复……）

当然，我们可以试着用精确的英语写程序。但是我们必须谨慎细致，确保词汇拼写正确，并保证能处理所有的情况。如果我们工作得足够努力，是否就可以写出精确的英语指令呢？

其实，有许多人在从事规范英语的写作工作，这些人被称做公务员，他们撰写的文稿被当做政府法规。不幸的是，公务员越努力想使法规精确，文稿的可读性就越差。如果你曾经读过有关税收条款的规定，就知道规范英语是怎么一回事了。

但是，即使法律条文加入了所有必要的字眼，也还是会出问题。几年前，美国加州通过了一项法律，要求摩托车驾驶员都要戴头盔。这项法律刚颁布不久，一位警察拦住了一位没戴头盔驾驶摩托车的小伙子，小伙子建议警官仔细再看看这条法规条文。

该条法律有两条要求：1）摩托车驾驶员应有一个好的防撞头盔，2）把头盔用带子绑牢。警察最终也没能给这位驾驶员开罚单，因为他确实绑了一个头盔——在膝盖上。

所以说，因其自身存在的问题，英语不适于做计算机语言。那么，我们怎样才能与计算机交流呢？

世界上第一台计算机的价值达数百万美元，当时，一个高级程序员的年薪约15,000美元。程序员不得不使用一种能把所有的指令转化为一串数字的语言来编写程序，这种语言叫做"机器语言"（machine language），可以直接输入到计算机中，典型的机器语言程序如下：

```
1010 1111
0011 0111
0111 0110
.. 等等几百个指令
```

机器以数字来"思考"，而人却不行。为了给这些老式的机器编程，软件工程师使用了一种简单的语言来书写程序。在这种语言中，每个字代表一个单独的指令。因为程序员必须自行把每行程序翻译或汇编为机器语言，所以这种语言被称为"汇编语言"（assembly language）。

一个典型的程序如下所示：

```
Program          Translation
MOV A,47         1010 1111
ADD A,B          0011 0111
HALT             0111 0110
.. 等等几百个指令
```

这个过程如图1-1所示。

翻译是一项非常困难、单调而且要求精确的工作。有位软件工程师认为这项工作完全适合于计算机，所以他写了一个可自动完成编译的程序，叫做汇编程序（assembler）。

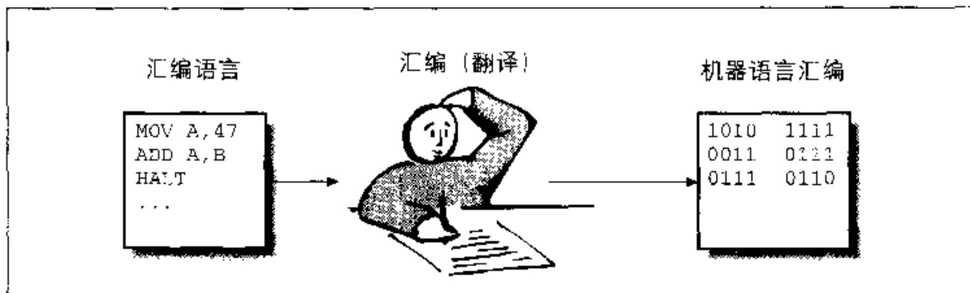


图 1-1 汇编一个程序

当他把这项新创造告诉他的老板时，却立即遭到了责骂："你竟敢设想让这样一台昂贵的机器仅仅去干'文书'的工作？"。当然如果拿计算机运转一小时的花费与程序员工作一小时的费用相比，这种说法并非毫无道理。

值得庆幸地是，随着时代的前进，程序员的价值越来越高，而计算机的费用却在不断下降。因此让程序员用汇编语言编写程序，并将这些程序翻译为机器语言，这样的过程是最节省费用的。

汇编语言用程序员很容易理解的方式来组织程序，但是，机器使用这些程序则较为困难，同时在机器运行它之前必须先进行翻译。这种方式是一个趋势的开始：编程语言变得越来越方便于程序员使用，而计算机用于翻译的时间却越来越长

这些年来，研究人员已设计了一系列的高级语言（high-level language）。这些语言都力图让程序员用易于理解的方法来书写程序，同时又必须保证足够精确而且简单，以便计算机能够接受。

早期的高级语言只能处理具体的应用程序。如FORTRAN用于数值计算；COBOL用于编写商业报告；而PASCAL用于教学。（在这些语言中，有许多远远超出了它们最初的使用范围。Nicklaus Wirth 曾经说过："如果我知道 PASCAL 会变得这样成功，在当初设计时，我会更加仔细。"）

C语言简史

在1970年、一位名叫Dennis Ritchie的程序员首创了一种取名为C的新语言。(之所以取这个名字、是因为C取代了他们原先使用的B语言。)设计C语言的主要目的是编写操作系统。它极其简单、灵活、很快被用来编写各种类型的程序、逐渐成为世界上最流行的语言之一。

C语言的流行归功于两个主要因素:其一是这门语言不为程序员设置障碍。通过使用正确的C语言指令几乎可以完成任何任务;(我们也会看到,这种灵活性也有缺点、它会使程序做那些程序员无意去做的事情。)其二是可移植的C编译器的广泛采用,使人们可以轻而易举地给机器安装C编译器却所耗无几。

1980年,Bjarne Stroustrup开始用一种新的语言工作、这种语言被称做"带类的C语言"。它增加了大量的新特性、全面改进了C语言、其中最重要的特性就是类。这种语言经过改进、扩充,最终成为了C++。

Java作为最新的语言之一,是基于C++发展来的。Java被设计成"带有固定BUGS的C++"。在本书编写期间,尽管Sun公司和其他公司正极力将Java推向市场,但使用仍然有限。

C如何工作

C语言是为建立程序员与裸机之间的桥梁而设计的。最初的设想是让程序员用一种易于理解的语言来设计程序、然后、编译程序再把这种语言翻译成机器可以识别的语言。

计算机程序主要由两部分组成:数据和指令。计算机并没有刻意去组织这两部分,但计算机总是被设计得尽可能通用,程序员则应把自己的组织方式用于计算机上。

计算机中的数据被存储为一系列的字节,C语言将这些字节组织成有用的数据。程序员使用数据说明来描述他要操作的信息。例如:

```
int total;        /* Total number accounts */
```

这条语句告诉C我们想用计算机的一段内存来存放一个取名为total整数。至于具体使用内存中的哪些字节来保存这个整数，可以让编译器去做，我们没有必要去理会。

total 变量是一个简单变量（simple variable），仅能保存一个整数，并且只能描述一个变量。如果有一系列的整数，可用数组来保存，如：

```
int balance[100];   /* Balance (in cents) for all 100 accounts */
```

在内存中存放数组的细节问题由C编译器来处理。对于更复杂的数据类型，例如，一个矩形可能由一个宽度、一个高度、一种颜色和一种填充方式组成。C允许将这四个属性定义成一个结构（structure）：

```
struct rectangle {
        int width;          /* Width of rectangle in pixels */
        int height;         /* Height of rectangle in pixels    */
        color_type color; /* Color of the rectangle */
        fill_type fill;     /* Fill pattern    */
};
```

结构能使程序员将数据排列得满足他的需求，而不管这些数据多么简单或者复杂。将数据说明翻译成计算机能识别的信息是编译器的工作，不用程序员来做。但是，数据只是程序的一部分，我们还需要指令。计算机丝毫也不了解指令是如何组织的，它只知道执行当前指令，并知道到哪里取出下一条指令。

C是一种高级语言。它允许程序员书写下面这样的语句：

```
area = (base * height) / 2.0;  /* Compute area of triangle */
```

编译器会把这条语句翻译成一系列的低级机器指令。这种语句称为赋值语句（assignment statement），它用来计算并存储算术表达式的值。

我们还可以使用控制语句（control statement）来控制程序的流程，如**if**和**switch**语句能让计算机进行简单的判断，**while**和**for**语句可以重复地执行一些语句。

语句集可以组成函数(function)。如果我们想画一个矩形，只需写一个通用函数，而且只写一次即可。当想画新的矩形时，可以再次调用这个函数。C为程序设计提供了一组丰富的标准函数（standard functions)，用这些函数可以完成常见的功能，如查找、排序、输入和输出。

一组相关函数可以组织成一个单独的源文件，众多的源文件经过编译就形成了程序（program)。

C语言的一个重要用途是把指令组织成可重复使用的组件。这样，如果你可以借用别人已有的代码，那么写程序就会快得多。多组可重用函数可组合成一个库(library)。例如，如果你需要一个排序程序，可以调用库中的标准函数Qsort，把它与你的程序连接在一起。

计算机数据的定义、结构和控制语句，以及其他C语言要素，不会减轻计算机的负担，计算机并不能分辨一百万随机字节和一个真正的程序之间的差异。C语言设计多个要素的目的就是让程序员能按照一种适合自己而不是适合计算机的方式来清晰地表达和组织他的思想。

组织是编好程序的关键。比如本书，你知道目录在前面，而索引在后面，采用这种结构是因为这是书要求的组织方法。这种组织方法使得本书的使用更加方便。

C语言遵循特定的语法（syntax）规则来组织程序。本书除了讲解C语言的语法，还介绍一些编程风格，使用户可以编制出可读性强的程序。强大的语法加上好的编程风格，就可以创建出复杂而奇妙的大型程序，并且可以依照一定的方法组织起来，这种方法会在需要变动的时候便于你的理解。

如何学习C

学习编程的唯一方法就是编写程序。通过编写程序并调试程序，可以从中学到许多东西，比只读这本书要多得多。本书包含许多编程练习，读者应该尽量都做一遍。做练习时，应该保持良好的编程风格。即使你只是自己练习，最好也写上注释。注释可以帮助你理顺思路；当编写真正的程序时，给程序加注释会是一种好的习惯。

当然不要总想着，“我写这些程序是为自己使用的，所以不需要为它们写注释。”首先，编写时看上去非常简单的代码，过一阵再读时往往弄不明白。给程序写注释有助于你在编写真正的代码前组织好思路。(如果你能用英语编写算法，那么写注释就会事半功倍。)

最后，程序可能会用在意想不到的地方。我曾经在Caltech的一台计算机上写过一个程序，因为我是唯一使用它的人，所以我把程序设计为：当我在命令行输入了错误的命令后，程序会打印出下列信息：

```
?LSTUIT User is a twit
```

几年后，我在Syracuse大学读书。计算机科学学院的一位秘书需要一个程序，它与我在Caltech编写的程序很相似，所以我把那个程序修改了一下，交给了那位秘书去用，不幸的是，我忘记了输入错误命令时出现的那个玩笑信息。

可以想像，当计算机科学学院秘书长把我叫到办公室谈话时，我是多么的震惊。这位女士很有权势，连系主任都怕她三分。她看着我，说道：“用户是个大笨蛋，嗯？”，好在她很幽默，否则，我也不会有今天。

本书中会有一些“失败”的程序，请花些时间找出错误之所在。有些问题非常微妙，比如，分号放错了位置，或者该写“==”的地方误写为“=”等，本书教给你在小程序中找出这些错误的方法。当在大程序中出现类似错误时，你将能迅速地找出错误。

第二章

编程基础

本章内容

- 程序从概念到运行
- 编写一个真正的程序
- 使用命令行编译器编程
- 使用集成开发环境编程
- 获得UNIX帮助
- 获取集成开发环境帮助
- 集成开发环境菜单
- 编程练习

至少对今天的作家来说至关重要的是，
语言简洁流畅，入木三分。
—— 海明威（美国作家）

程序开始于人类编写的一套指令。在应用于计算机之前，这些指令必须几经转化。本章我们将学习怎样集成程序，并把它转化成计算机能使用的东西，然后运行它。详细的步骤可以参见UNIX和DOS/Windows编译器。

程序从概念到运行

C程序是用一种高级语言编写的，其中使用了字符、数字以及能在计算机键盘上找到的其他符号。实际上，计算机运行的是一种非常低级的语言（low-level language），这种语言称为机器代码（一串数字）。所以，在使用程序之前，必须先做转化。

程序最初只是程序员头脑中的一个想法。他使用文本编辑器把思路写到一个文件中，这个含有源代码的文件叫源文件，编译器把这个文件转换为目标文件，然后，连接程序把目标文件与标准库中的预定义程序结合在一起，产生一个可执行的文件（一组机器语言指令）。在本章随后的介绍中，你会了解到这几种不同形式的程序是如何相互配合工作并生成最终可执行文件的。

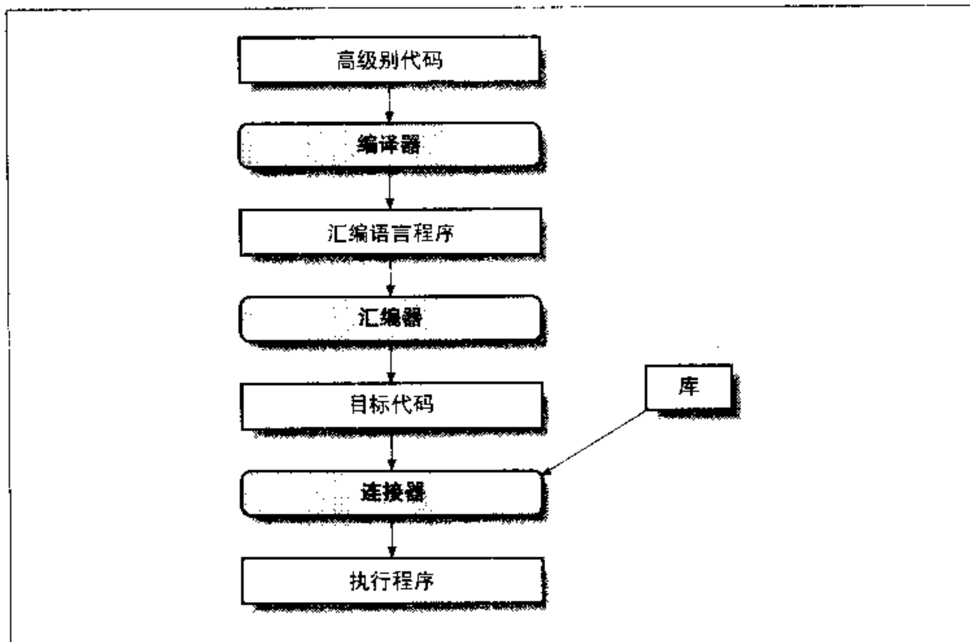


图 2-1　用高级语言编写的一个程序转为可执行文件时所需的步骤。

包装

幸运的是，你不必单独运行编译器、汇编程序和连接程序。大多数C编译器使用"包装"程序，由它来决定需要运行哪些程序，并使之运行。

有些编程系统更进了一步，它们为开发人员提供一个集成的开发环境(IDE)。IDE包含了编辑器、编译器、连接程序、项目管理程序、调试器和其他一些工具，这些都集成在一个方便的包内。Borland和Microsoft公司都为他们的编译程序提供了IDE。

编写一个真正的程序

在你能真正动手编制自己的程序之前，需要了解如何使用基本的编程工具。这一部分将向读者详细介绍录入、编译并运行一个简单程序的各个步骤。

这一部分将介绍两类不同编译器的使用方法。第一种类型为独立的或是命令行编译器。这类编译器以批处理的方式从命令行运行。换句话说，你要键入一个命令，然后编译器把源代码转化为可执行程序。另一类编译器包含在IDE中。

大部分UNIX系统使用命令行编译器，只有很少的几个IDE类型编译器运行在UNIX平台下。另一方面，用于MS-DOS和Windows的编译器差不多都包含集成开发环境，因为命令行方式根深蒂固，所以这些编译器也包含命令行编译器。

使用命令行编译器编程

这一部分将学习使用命令行编译器编写程序的详细步骤。我们将列出使用普通UNIX编译器、自由软件基金会(Free Software Foundation)的gcc编译器、Turbo C++、Borland C++和Microsoft Visual C++（注1）的命令。

当然，如果你正使用Borland或是Microsoft编译器，不妨转到介绍IDE使用方法的部分。

第1步：为你的程序找个地方

如果为每个程序建立一个单独的目录，那么管理工作将非常简单。本例中，将建立一个名为*hello*的目录，该目录下存放*hello*程序。

在UNIX系统下，键入如下命令：

```
% mkdir hello
% cd hello
```

在MS-DOS系统下，键入如下命令：

```
C:> MKDIR HELLO
C:> CD HELLO
```

注1： Turbo C++、Borland C++和Microsoft Visual C++都是C++编译器，它们也能编译C代码。

第2步：创建一个程序

程序最初是一个文本文件。示例2-1列出了 *hello* 程序的源代码。

例2-1: hello/hello.c

```
[File: hello/hello.c]
#include <stdio.h>
int main()
{
     printf("Hello World\n");
     return (0);
}
```

使用你喜欢的文本编辑器，然后输入这个程序。在UNIX系统下，相应的文件名为 *hello.cc*，在MS-DOS/Windows下，文件名为 *hello.c*。

警告：MS-DOS/Windows用户不要用字处理软件写程序，如不要使用Microsoft Word或是WordPerfect来写。字处理软件在文件中加入了排版格式符号，而编译器不能正确区分这些符号。你必须使用能编辑ASCII文件的文本编辑器，如MS-DOS中的EDIT程序。

第3步：运行编译器

编译器把你刚录入的源文件转变为可执行的程序。每个编译器都有不同的命令行，下面是一些最流行的编译器。

UNIX系统下的cc编译器（通用UNIX）

大多数基于UNIX的编译器都遵从于同一个标准。C编译器名为CC，用下列命令就可以编译我们的 `hello` 程序：

```
% cc -g -o hello hello.c
```

-g选项表示可以进行调试。（编译器在程序中增加了额外的信息，以使调试更容易一些。）开关 -o 告诉编译器最后生成的程序名为 `hello`，最后一项 `hello.c` 是源文件的名字，其他选项的用法请参见你使用的编译器手册。在UNIX系统下，有几个不同的C编译器，所以你使用的命令行可能略有不同。

自由软件基金会的gcc编译器

自由软件基金会研制了多个高质量的程序（如想得到他们的软件，请查阅书和词汇表中“Free Software Foundation”词条的内容），同时提供了一个名为*gcc*的C编译器。

用gcc编译器来编译程序时，要键入下列命令：

```
% gcc -g -Wall -o hello hello.c
```

附加开关 -Wall 可以显示所有警告信息。

GNU编译器在C语言基础上做了几处扩展。如果你想关掉这些特性，使用下列的命令行：

```
%  gcc -g -Wall -ansi -pedantic -o hello hello.c
```

-ansi 开关关掉了和ANSI C不相匹配的GNU C特性。-pedantic开关可以使编译器对遇到的任何非ANSI特征发出警告。

Borland公司用于MS-DOS的Turbo C++

Borland公司开发了一个便捷的MS-DOS平台下的C++编译器，名为Turbo C++，这个编译器既能编译C代码也能编译C++代码。这里仅描述如何编译C代码。对学习编程的人来说，Turbo C++ 是理想的编译器。Turbo C++ 命令行为：

```
C:> tcc -ml -v -N -w -e hello hello.c
```

-ml 选项告诉Turbo C++使用大内存模式。（该PC机可以使用多种内存模式，目前仅使用这种大模式就行了，不同模式之间的差异，只有专业的计算机程序员才需要了解。）

-v 开关告诉Turbo C++编译器在程序中加入调试信息；-w选项将打开警告；-N选项将检查堆栈。最后一个选项-e hello告诉Turbo C++建立一个名为hello的程序，而hello.c为源文件名。全部选项列表请参看 Turbo C++参考手册。

Windows 编程

在目前广泛采用 Windows 编程的时候，你或许想知道为什么还要提及 MS-DOS 编程。原因很简单，因为在 Windows 环境下编程要比在 MS-DOS 下编程复杂得多。

例如，要打印“Hello World”在 MS-DOS 下直接打印出这条消息就行了。在 Windows 状态下，必须创建一个窗口，设置一个用于处理该窗口消息的函数，选择一种字体，选择设成该字体的位置，然后才能输出该消息。

跑之前你须先学会走。因此，本书将范围限定在 MS-DOS 或 Easy-Win（即简化的 Windows）程序中。

MS-DOS 和 Windows 下的 Borland C++

除了 Turbo C++ 编译器，Borland 公司还有另外一个用于 MS-DOS/Windows 的全功能、专业级的编译器，名为 Borland C++。它的命令行为：

```
C:> bcc -ml -v -N -P -w -e hello hello.c
```

Turbo C++ 与 Borland C++ 的编译器使用的命令行选项是一样的。

Microsoft Visual C++

Microsoft Visual C++ 是用于 MS-DOS/Windows 的另一个 C/C++ 编译器。编译时使用下列的命令行：

```
C:> cl /AL /Zi /W1 hello.c
```

`/AL` 选项告诉编译器使用大内存模式。打开调试开关用 `/Zi` 选项，警告信息用 `/W1` 选项。

第 4 步：运行程序

运行程序（在 UNIX 或 MS-DOS/Windows 状态下）键入：

```
% hello
```

屏幕上将显示下列信息:

```
Hello World
```

使用集成开发环境（IDE）编程

集成开发环境提供编程时所必需的工具，这些工具有编辑器、编译器和调试器，它们集成在一个软件包内供程序员使用。

第1步：为你的程序找一个地方

如果为每个程序建立一个单独的目录，那么管理工作将非常简单。本例中，将建立一个名为HELLO的目录来存放*hello*程序。

在MS-DOS系统，键入如下命令:

```
C:> MKDIR HELLO
C:> CD HELLO
```

第2步：使用IDE录入、编译并运行程序

每个IDE都略有不同，针对每个编译器，我们列出各自的指令。

Turbo C++

1. 使用下列命令启动Turbo C++的集成开发环境（IDE）:

   ```
   c:> TC
   ```

2. 选择*Window|Close All*菜单项以清除桌面上原来的窗口。从清屏开始，屏幕应显示如图2-2。

3. 使用*Options|Compiler|Code Generation*命令打开Code Generation对话框，对话框如图2-3所示。把内存模式改为大模式。

图 2-2　清除桌面

图 2-3　代码生成对话框

4. 选中 *Options|Compiler|Entry/Exit* 菜单项打开如图 2-4 所示的 "Test Stack overflow"。

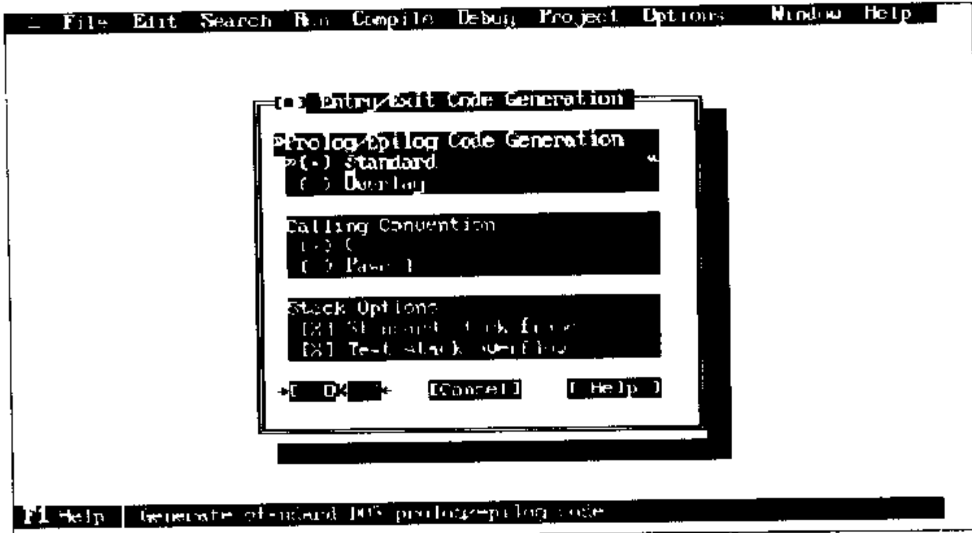


图 2-4 Entry/Exit 代码生成对话框

5. 使用*Options|Compiler|Messages|Display*命令打开如图2-5所示的Compiler Messages 对话框。选择 All 以显示所有的警告信息。

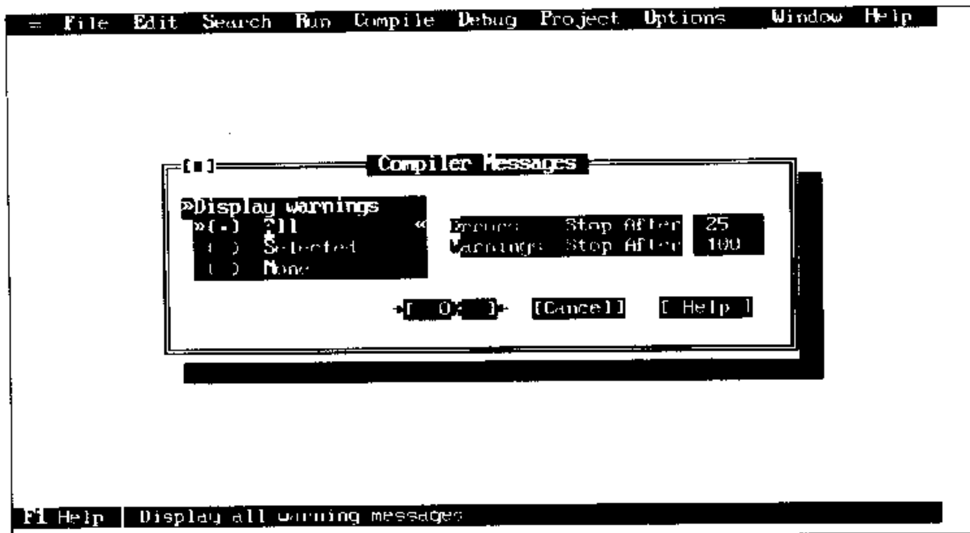


图 2-5 编译器信息对话框

6. 使用 *Options|Save* 命令，把到目前为止使用过的选项保存起来。

7. 使用Project\Open命令来选择一个项目文件。本例的项目文件叫做HELLO.PRJ。屏幕显示如图2-6所示。

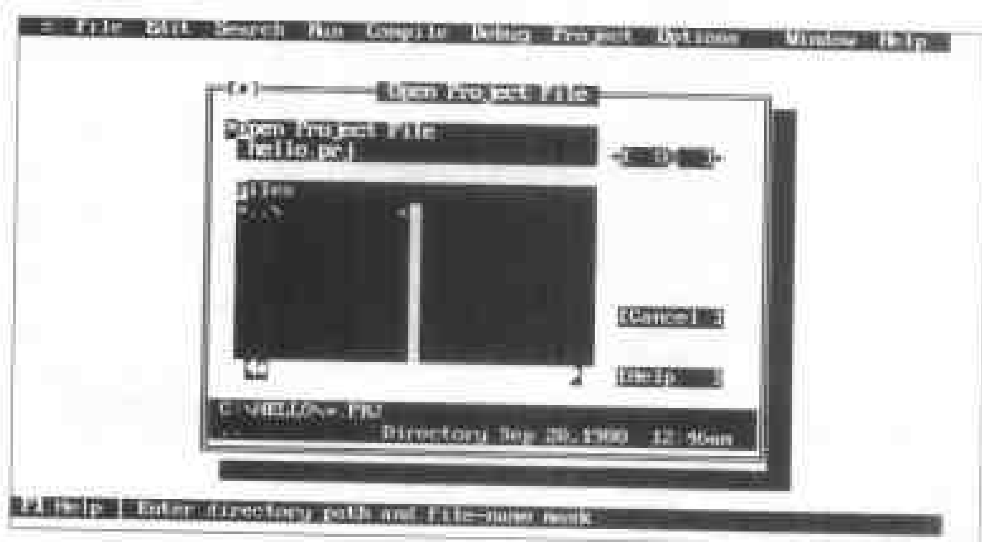


图2-6　打开项目文件对话框

8. 按Insert键，把文件加入到项目中。我们要加入的文件是HELLO.C如图2-7所示。

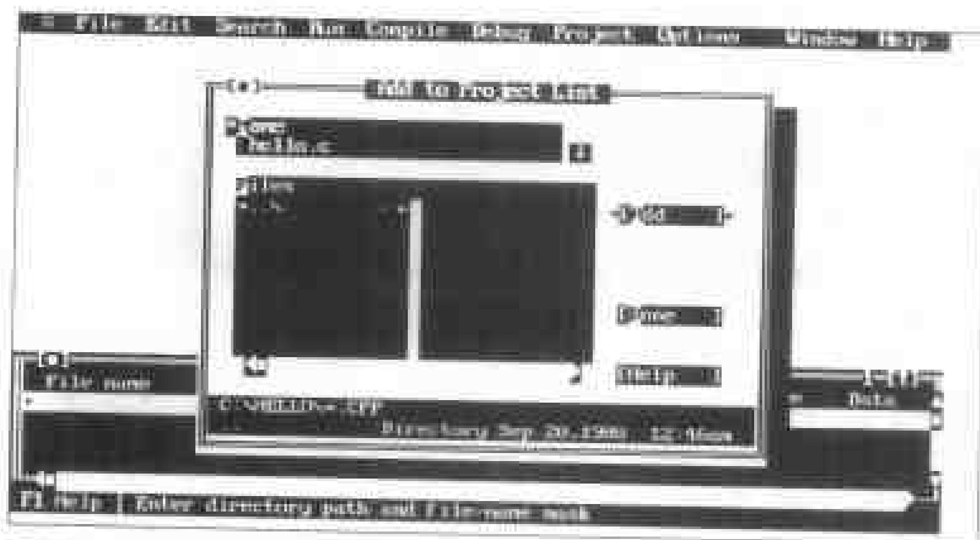


图2-7　项目列表添加对话框

9. 按ESC，退出添加文件（add-file）循环。

10. 按向上箭头键，上移一行。*HELLO.C* 所在行将呈高亮度显示，如图2-8。

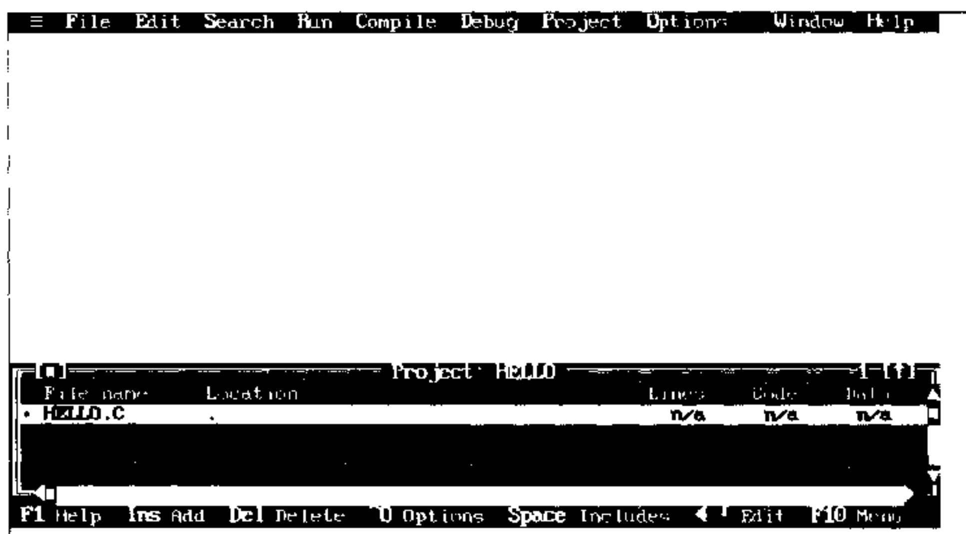


图2-8 Hello 项目

11. 按回车键，编辑文件。

12. 输入例2-2。

例2-2：hello/hello.c

```
[File: hello/hello.c]
#include <stdio.h>
int main()
{
     printf("Hello World\n");
     return (0);
}
```

输入完毕如图2-9所示。

13. 选择 *Run|Run* 菜单项执行程序。

14. 程序运行后，由集成开发环境进行控制，这种控制上的改变意味着看不到程序的输出。如果想看到程序结果必须使用 *Window|User* 菜单项目切换到用户窗口。按任意键返回IDE，程序输出如图2-10所示。

图 2-9　完成项目

图 2-10　用户运行程序屏幕输出

15．完成后可选择 *File|Save* 菜单项目保存程序。

16．要退出 IDE，选择 *File|Quit* 菜单项目。

BorLand C++

1. 创建名为*HELLO*的子目录来存放Hello World程序。你可以使用Windows的文件管理器（File Manager）创建子目录，或者在MS-DOS状态下键入下列命令：

```
C:> mkdir \HELLO
```

2. 在Windows中双击“Borland C++图标以启动IDE。选择*Window|Close all*菜单项清除原有窗口。程序开始运行，并显示图2-11所示空白工作区。

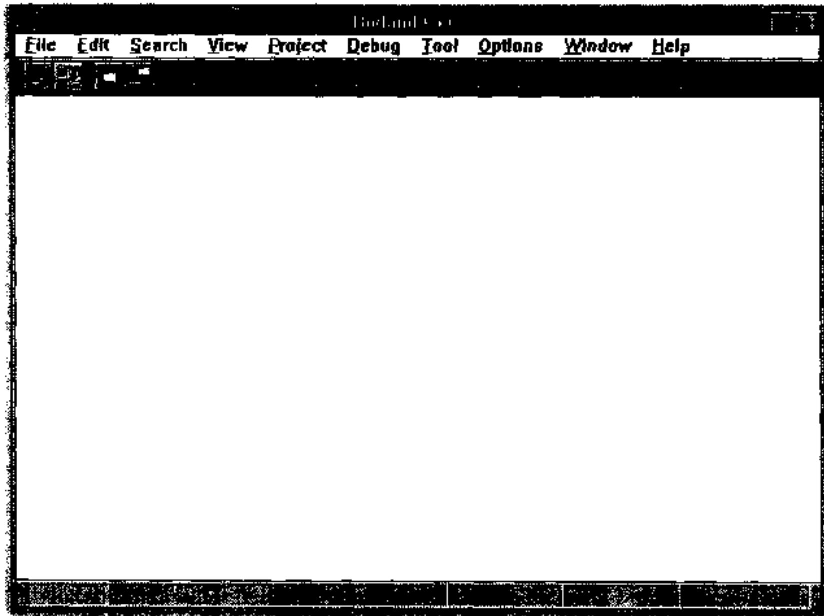


图2-11 Boland C++初始化屏幕

3. 选取*Project|New Project*菜单，为我们的示例程序建立一个项目。在“Project Path and Name栏中填入C:\hello\hello.ide。在Target Type选择框中选择*Easy Win[.exe]*。在*Target Model*下拉列表中选择*Large*。相应的窗口如图2-12。

4. 单击*Advanced*按钮打开Advanced Options对话框。清除*.rc*和*.def*项目并设置*.c Node*项如图2-13。

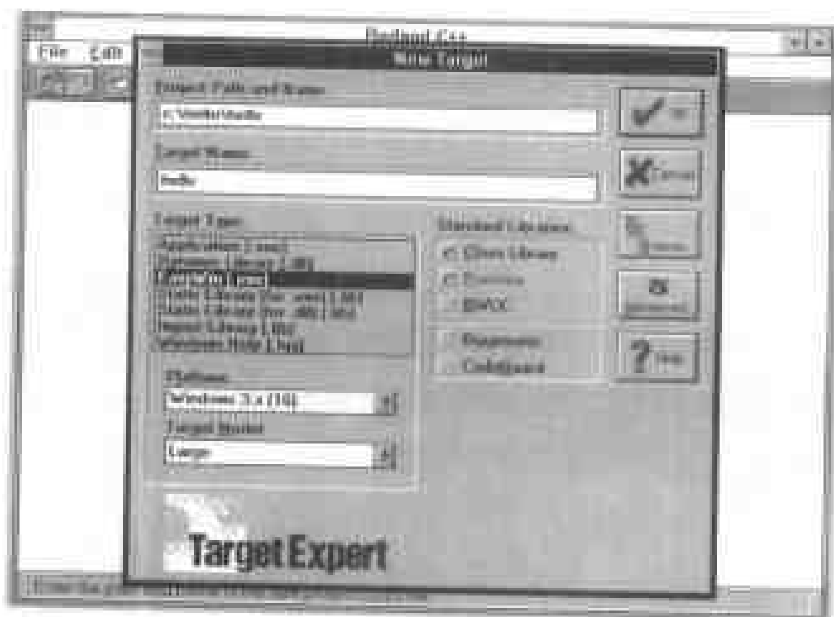


图 2-12　新的目标对话框

图 2-13　高级选项对话框

5. 单击 *OK* 返回 New Target 对话框。再次单击 *OK* 返回主窗口。

6. 按 `ALT-F10` 激活如图 2-14 所示节点子菜单。

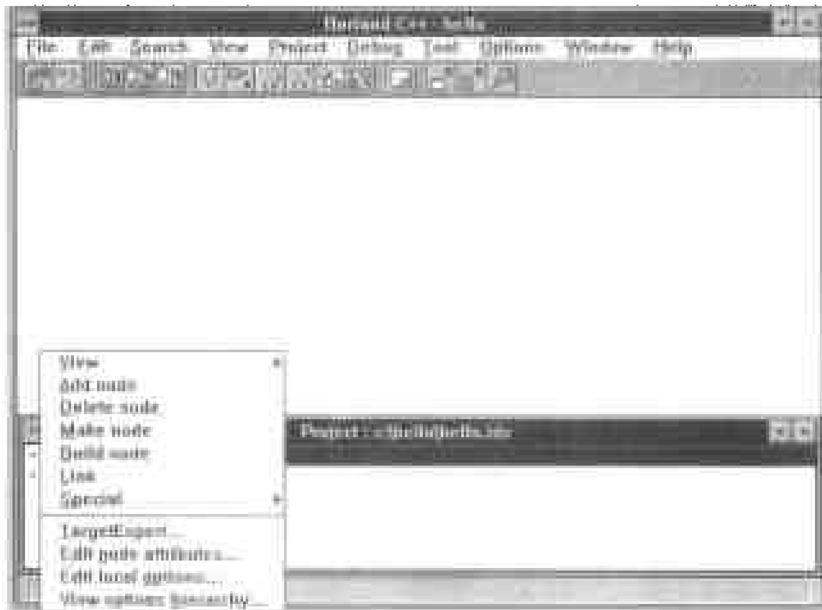


图2-14 目标选项对话框

7. 选择 *Edit Node Attributes*，出现的对话框如图 2-15 所示。在 *Style Sheet* 空格处选择“*Debug Info and Diagnostics*”单击 *OK* 后返回主窗口。

8. 选择 *Options|Project Options* 项，进入 Project Options 对话框。选项条向下移动到 Compiler 项，点击“+”打开这个项目。

   选中 *Test stack overflow* 选项，见图 2-16，单击 *OK*，保存这些选项。

9. 单击 *OK* 返回主窗口。按下箭头，在 Project 窗口中选择 *hello[.c]* 项，如图 2-17。

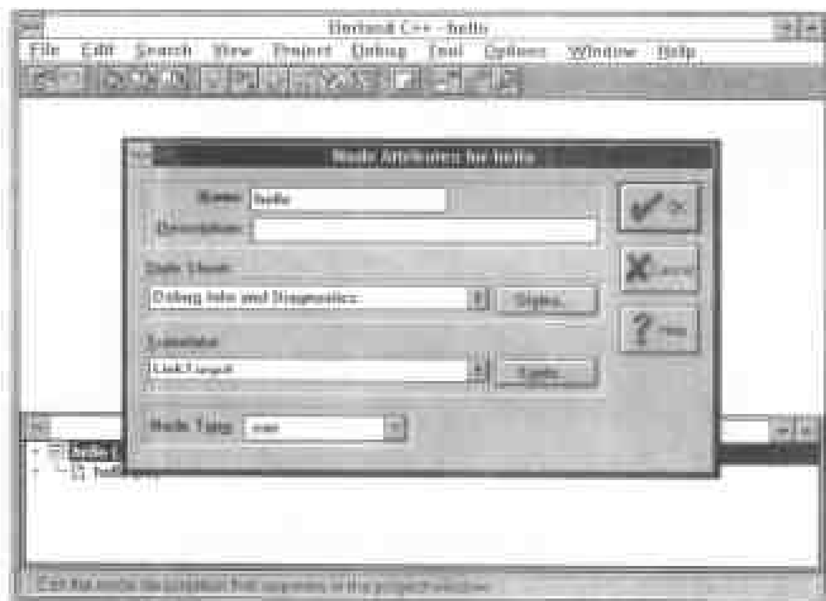


图 2-15 节点属性对话框

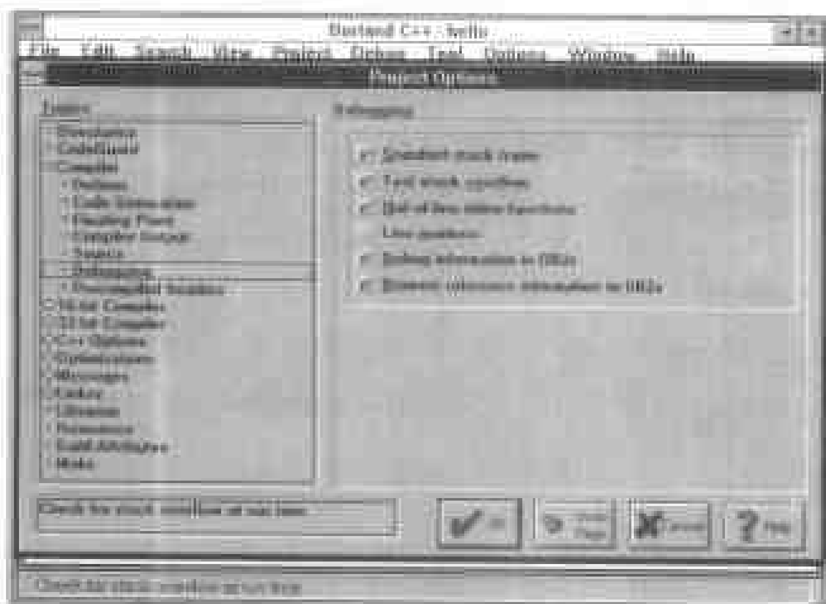


图 2-16 项目选项对话框

图2-17 Hello 项目

10. 按 Return，开始编辑文件 *hello.c*。键入下列代码：

例2-3：hello/hello.c

```
#include <stdio.h>
int main()
{
     printf("Hello World\n");
     return (0);
}
```

完成后屏幕显示如图 2-18。

11. 选取 *Debug|Run* 菜单项，运行这个程序将显示“Hello World”如图 2-19 所示。

图2-18　Hello World 程序

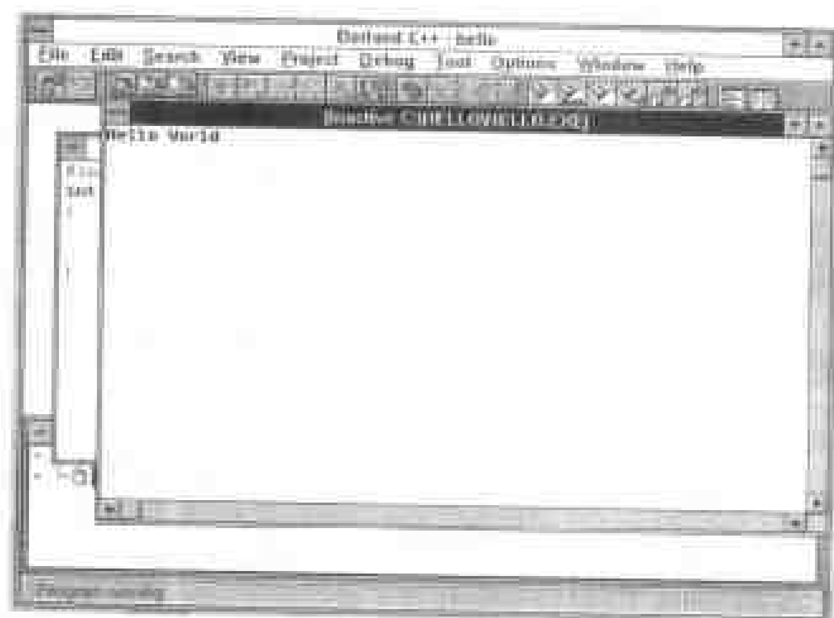


图2-19　执行后的 Hello World 程序

Microsoft Visual C++

1. 创建名为*\HELLO*的子目录来存放Hello World程序。你可以使用Windows的文件管理器(File Manager)创建子目录，或者在MS-DOS状态下键入下列命令：

```
C:> mkdir \HELLO
```

2. 在Windows中双击Microsoft Visual C++，启动IDE。选择*Window|Close all*菜单项，清除原有窗口，程序开始运行，并显示图2-20所示空白工作区。

图2-20 Microsoft Visual C++ 初始化屏幕

3. 单击*Project|New*菜单项目打开New Project对话框，如图2-21。

   在项目名称空白栏内填写"\hello\hello.mak，将Project Type改为*QuickWin application[.exe]*。

4. 使用Visual C++的Edit对话框给这个项目的源文件取名称(见图2-22)。本例中，我们只有一个文件*hello.c*。单击*Add*，把这个名字加入项目中，然后单击*Close*，告诉Visual C++已没有别的文件。

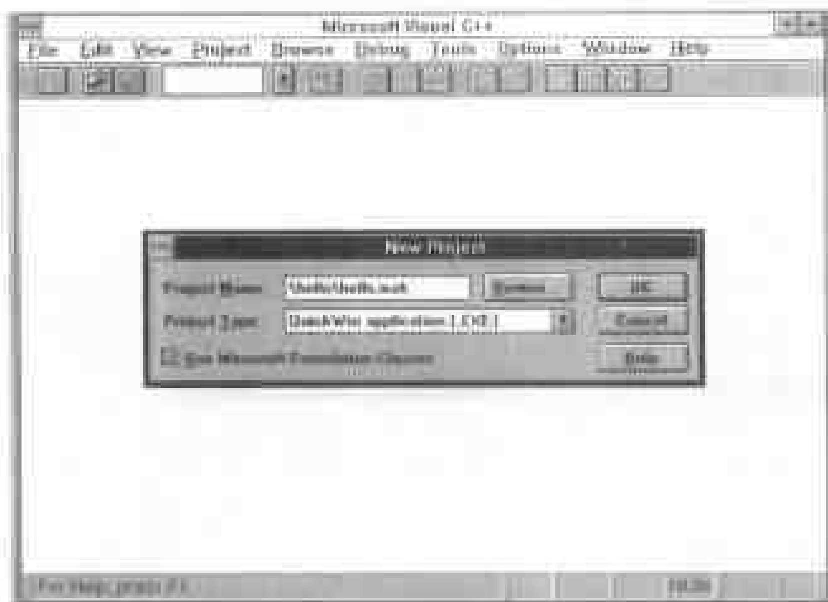


图 2-21　新项目对话框

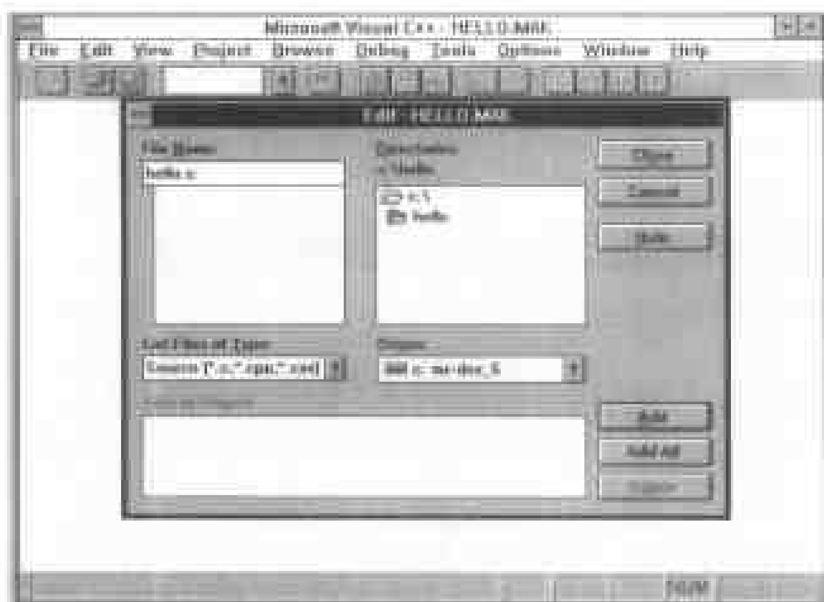


图 2-22　编辑项目对话框

5. 选择 *Options|Project Options* 菜单项，出现的 Project Options 对话框如图 2-23。

   单击 *Compiler* 按钮，修改编译选项。

图 2-23 项目选项对话框

6. 在 *Category* 中，向下移动亮条到 *Custom Options* 项，把警告级别改为 4，如图 2-24。
7. 选择 *Memory Model* 类别，并把它改为大模式。
8. 单击 *OK* 关闭对话框，返回 Project Options 对话框。同样单击 *OK* 退出这个对话框。
9. 选取 *File|New* 菜单项目启动一个新程序文件，键入例 2-4。

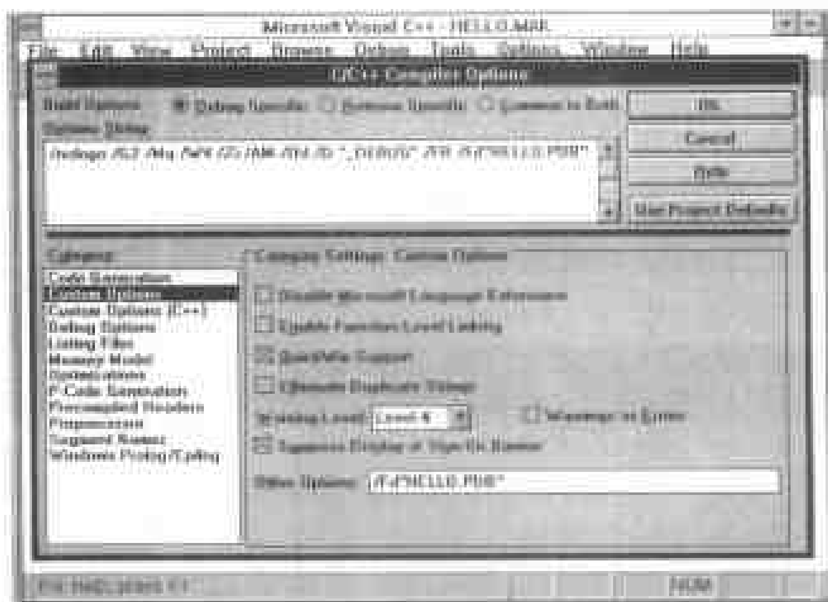


图 2-24　C/C++编译器选项对话框

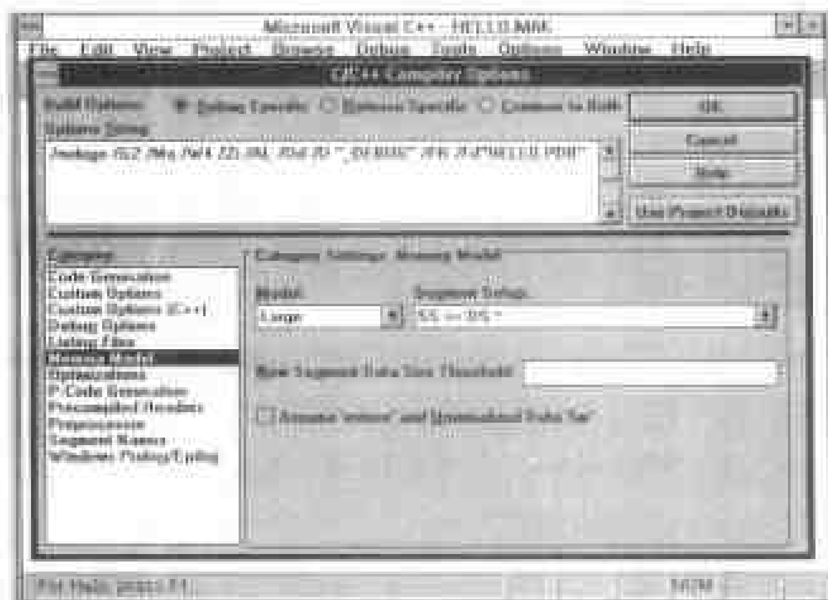


图 2-25　内存模式选项

例 2-4: hello/hello.c

```
[File: hello/hello.c]
#include <stdio.h>
int main()
{
     printf("Hello World\n");
     return (0);
}
```

结果应如图 2-26 所示。

图 2-26　敲入 Hello World 程序后的 Microsoft Visual C++ 界面

10. 在 *hello.c* 文件名下使用 *File|Save As* 菜单项目保存文件。

11. 使用 *Project|Build* 菜单项编译程序，程序建立的同时编译器将输出信息。编译结束时屏幕显示如图 2-27。

12. 现在可以使用 *Debug|Go* 命令来运行程序，程序执行的结果见图 2-28。

图2-27　Microsoft Visual C++ 项目 build 屏幕

图2-28　Hello World 程序运行结果

获取 UNIX 帮助

多数 UNIX 系统都有名为“man pages”的联机文档系统，使用 man 命令就可以访问该文档（UNIX 用 man 作为手册“manual”的缩写）。如果要得到有关一个特定主题的信息，可使用下列命令：

```
man subject
```

例如，为找出 printf 函数中所定义的类，你应键入：

```
man printf
```

该命令也可按关键字来进行查找，如：

```
man -k keyword
```

如想得到本标题下含有“output”关键字的手册页的名字，可以使用如下命令：

```
man -k output
```

获取集成开发环境帮助

像 Turbo C++、Borland C++ 和 Microsoft C++ 这样的集成开发环境，都有一个 *Help* 菜单项。使用这个菜单项可以激活基于超文本的 Help 系统。

集成开发环境菜单

这一部分包含了本章所讲的三种集成开发环境下集成、编译并执行一个简单程序所使用的命令的简要总结。

Turbo C++

1.	Window|Close All	清除所有无用程序
2.	Options|Compiler|Code Generation Memory Model = Large	对简单程序，使用大内存模式。
3.	Options|Compiler|Entry/Exit Test stack overflow = On	普通编程错误测试功能启动
4.	Options|Compiler|Messages|Display Display warnings = All	启动所有诊断程序
5.	Options|Save	保存选项
6.	Project|Open Project file = *program*.PRJ	创建一个新项目
7.	Insert Add file *program*.c	在项目中添加程序文件
8.	ESC	跳出"添加"循环
9.	UP-ARROW	移到 *program*.c 行中
10.	RETURN	编辑程序文件
11.	Type in the program	输入程序文本
12.	Run|Run	执行程序
13.	Window|User	显示程序结果
14.	File|Save	保存程序
15.	File|Quit	退出

Borland C++

1.	Window|Close All	清除所有无用程序
2.	Project|New Project Project Path and Name = c:\\ *program**program*.ide Target Type = EasyWin(.exe) Target Model = Large	创建一个新项目
3.	单击 Advanced 按钮 Set .c Node Clear .rc and .def	启动 C 程序

Borland C++（续）

4.	单击 OK	回到 New Target 窗口
5.	单击 OK	回到主窗口
6.	ALT-F10	选择子菜单
7.	Edit|Node Attributes Style Sheet = Debug Info and Diagnostics	启动 debugging
8.	单击 OK 按钮	回到主菜单
9.	Options|Project Options 单击 Compiler 下的 + 按钮 Test stack overflow = On	启动有效的运行过程测试
10.	单击 OK 按钮	保存选项
11.	单击 OK 按钮	回到主窗口
12.	DOWN-ARROW	进入 *program*[.c]行中
13.	RETURN	编辑程序文件
14.	Type in the program	在程序中输入文本
15.	Debug|Run	运行程序

Microsoft Visual C++

1.	Window|Close All	清除所有无用程序
2.	Project|New Project Name = *program**program*.mak Project Type = QuickWin application (.EXE)	启动项目 创建项目 单击 OK 按钮 进入 Edit 对话框
3.	File name = *program*.c	输入程序名
4.	单击 Add 按钮	在项目中添加程序
5.	单击 Close 按钮	关闭项目
6.	Options|Project Options	进入 Project Options 对话框
7.	单击 Compiler 按钮	进入 C|C++ Compiler Options
8.	选择 Custom Options category Warning Level = 4	开启所有警告

Microsoft Visual C++（续）

9.	选择 the Memory Model category Memory Model = Large	对简单程序使用大内存模式
10.	单击 OK 按钮	回到 Project Options 对话框
11.	单击 OK 按钮	回到主窗口
12.	File|New	打开程序文件
13.	Type in the program	编辑程序文件
14.	File|Save As --File name = *program*.c	保存文件
15.	Project|Build	编译程序
16.	Debug|Go	执行程序

注：这些指令适用于 Microsoft Visual C++ 4.0 版本。Microsoft 通常根据版本的变化而改变其界面。所以这些指令时常会有一些修改。

编程练习

练习 2-1：在你的计算机上，键入 *hello* 程序，并运行它。

练习 2-2：找几个程序例子，录入并运行之。

本章内容

- 基础编码练习
- 编码盲从
- 缩进与编码格式
- 清晰
- 简明
- 小结

第三章 风格

不管怎样结构化，都没有编程语言，因为那会阻止程序员编写坏程序

——L.Flon

正是风格上的贵族气使得近40年的人们无法读懂19世纪40年代作家们的作品。

——斯汤达（法国作家）

本章将讨论如何利用良好的编程风格来建立一个简单易读的程序。当然在你还不知道如何编程的时候去讨论风格问题，好像有点本末倒置，但实际上编程风格是编程中最重要的部分。风格是去除糟粕的法宝，也是辨别真假编程艺术家的试金石。在键入第一行代码前，应先学习好的编程风格，这样写的程序才能保质保量。

与通常的看法相反，程序员在花费一定的时间写程序后，更多的时间则花在维护、升级和调试已有的代码上，而不是用来编写新的代码。资料显示，程序维护所耗的时间正在飞速增长。从1980年到1990年，典型应用程序中代码的平均行数从2.3万行增涨到120万行，而完善系统所需的平均时间则从4.75年延长到9.4年。

更为糟糕的事情是，据软件维护协会1990年的年度报告报道，在被调查的管理人员中，有74%的人认为：在他们的部门里正使用的系统必须由特定的人来维护，因为其他的人根本不懂这些系统。

大多数软件都是建立在已有软件的基础上。最近我完成了12个新程序，其中只有一个是从零做起；另外的11个程序都是在修改已有的程序基础上完成的。

一些程序员认为编程的目的只有一个，即给计算机提供一组紧凑的指令。其实这种观点并不正确，仅为机器而写的程序有两个问题：

- 难于判定正确性，因为有时作者也不理解它们；
- 程序的修改和升级很难，因为维护人员必须花费大量的时间来阅读源代码。

理想的程序服务于两个目的：一是为计算机提供一套指令，二是给程序员提供一个有关程序功能的描述。

例 2-1 包含一个明显的错误。这个错误是许多程序员的通病，它带来的麻烦比其他任何问题都大，错误就是程序中没有加注释

一个运行正常但没有注释的程序如同一个等待爆炸的定时炸弹，因为早晚会有人修改或升级这个程序，而注释的缺乏会使工作难上十倍。可以说，一个具有良好注释的程序是一件艺术品。因此，学习如何写注释与学习如何正确地编写代码同样重要。

C 的注释以斜线星号（/*）开头，以星号斜线（*/）结束。例 3-1 是 2-1 的改进版。

例 3-1: hello2/hello2.c

```
[File: hello2/hello2.c]
/********************************************************
 * hello -- program to print out "Hello World".         *
 *      Not an especially earth-shattering program.     *
 *                                                      *
 * Author:  Steve Oualline.                             *
 *                                                      *
 * Purpose:  Demonstration of a simple program.         *
 *                                                      *
 * Usage:                                               *
 *      Runs the program and the message appears.       *
 ********************************************************/
#include <stdio.h>

int main()
{
```

```
        /* Tell the world hello */
        printf ("Hello World\n")
         return  (0);
    }
```

本例中，开始部分的注释放在一个由星号（*）组成的框内，这个框通常被称为注释框（comment box）。这样做的目的是为了突出较重要的注释，这很像本书中标题所用的粗体字。不太重要的注释不加框。例如：

```
/* Tell the world hello */
printf ("Hello World\n");
```

在写程序前，必须对要做什么有个明确的思路。最好的办法是用一种清楚易懂的语言把它们写下来，一旦操作过程清楚了，就可以把它们转化为计算机程序。

明白要做什么，是编程中最重要的部分。我曾经写过满满两页纸的注释，来描述一个复杂的图形算法。在编写程序代码前，我两次修改了这段注释，实际上最后的程序只占半页。因为我把思路整理得很清楚，所以这个程序编译一次就通过了。

程序读起来应该像一篇文章，它应该尽可能地清楚易懂。良好的程序设计风格来自于经验和实践，后面将提到的风格是作者多年来的编程经验之谈，读者可以借鉴并形成自己的风格。但是这些并不是规则，仅仅只是建议。规则只有一条：尽可能地让你的程序清楚、简明、易读。

程序的开头是一个注释块，包含了有关程序的信息。通过给这段注释加框，以使它更加醒目。下面我们列出在程序的开始部分应该说明的内容，当然并不是所有的程序都要包含所有的部分，应根据具体情况选择使用。

- **开头** 第一段注释中除包含程序名外，还应该有该程序的简短描述。你可能已拥有最令人惊异的程序，它能解决世界上所有的问题，但是如果没有人知道它是做什么的，那它就一无用处了。
- **作者** 为了设计这个程序你解决了很多的麻烦。如果以后其他人要修改这个程序，他或她也可以向你索取相关的信息，并得到你的帮助。
- **目的** 为什么编制这个程序？它是干什么的？

- **用法** 下面将简单叙述程序是如何运行的。在理想情况下，每个程序都应该有叙述使用方法的若干文档，但事实并非如此。Oualline 关于文档的法则说：有90%的程序没有文档，在剩余的10%中，又有9%的程序，其文档的修改速度赶不上程序的修改速度，所以这些文档是完全无用的。最后，实际上只有1%的程序有文档，而且文档的修改与程序的修改是同步的，这些文档是用中文书写的（注1）。为了避免成为Oualline文档法则的牺牲品，请把文档与程序放到一起。
- **出处** 拷贝程序是有效的程序设计方法（如果你没有违反版权法的话）。实际上，只要你拿到了程序，采用什么方式倒无关紧要，但是，贪功是不可取的。在这一部分中，你应该提及所拷贝的程序的原作者。
- **文件格式** 列出程序中要读写的文件，并简要描述它们的格式。
- **限制** 列出程序中的所有局限或限制。例如，数据文件的格式必须要正确；程序不检查输入错误等。
- **修改历史** 这一部分含有一张表，指出是什么人、在何时修改了这个程序的哪些地方。许多计算机有源控制系统（UNIX是RCS和SCCS；MS-DOS/Windows是MKS-RCS、PCVS），这些系统可以为你记录这些信息。
- **错误处理** 如果程序发现错误，它该怎么处理？
- **注释** 包括一些特殊的注释，或是注释中没有的一些信息。

程序开始部分的注释格式依赖于编程环境的需要。例如，如果你是学生，导师会要求你在程序开头写清作业号、姓名、学号和其他信息。在工业界，应该包含项目号或是部门号。

程序员需要了解的程序的有关内容也应该在注释中说明。程序的注释应避免过多（这很少发生，但确实出现过）。当你确定开头注释的格式时，一定要保证所包含的每个字都有必要。

真正的程序代码包含两部分：变量和可执行的指令。变量用来存放程序中使用的数据，可执行的指令告诉计算机对数据进行何种操作。

注1： 我太太是香港人，她有一个电子语音翻译器，这个东西非常昂贵并且非常复杂，随机附有一份150页的手册，整本手册是用中文写的。

排版符号的不足

排版时可以使用不同的字型和字的大小、用粗体和斜体来代表文本中不同的部分。但在设计程序时，只能使用占一个位置的字符，程序员可以用各种巧妙的方法来克服这方面的不足。

下面是几种不同的注释方法：

```
/********************************************************
 ********************************************************
 ******** WARNING:  This is an example of a       *******
 ********   warning message that grabs the        *******
 ********   attention of the programmer.          *******
 ********************************************************
 ********************************************************/

/*------------> Another, less important warning <-----------*/

/*>>>>>>>>>>>>  Major section header  <<<<<<<<<<<<<<<< */

/********************************************************
 * We use boxed comments in this book to denote the     *
 * beginning of a section or program.                   *
 ********************************************************/

/*------------------------------------------------------*\
 * This is another way of drawing boxes.                *
\*------------------------------------------------------*/

/*
 * This is the beginning of a section.
 * ^^^^ ^^ ^^^ ^^^^^^^^^ ^^ ^ ^^^^^^^^
 *
 * In the paragraph that follows, we explain what
 * the section does and how it works.
 */

/*
 * A medium-level comment explaining the next
 * dozen (or so) lines of code.  Even though we don't have
 * the bold typeface, we can **emphasize** words.
 */

/* A simple comment explaining the next line */
```

基础编码练习

变量是计算机内存里存放数值的地方，C语言用变量名来标识这个地方。名字的长度可以任意定，所以最好挑选一个意义明确的名字（实际上，名字的长度还是有限制的，但这个数很大，你可能永远也不会遇到这个麻烦）。C语言中的每个变量都必须被说明，变量说明将在第九章讨论。下面的说明告诉C你准备使用三个整型（**int**）变量，名字分别为p、q和r：

```
int p, q, r;
```

但读者并不知道这三个变量是做什么的，它们可以表示插头的角度，或表示Space Invaders游戏中plasma bolt的位置和加速度。应避免缩写词，过分的缩写很难读，也不易理解（例如用Exs. abb. are diff. to rd. and hd. to ustnd. 代表Excess abbreviations are difficult to read and hand to understand）。

现在再看看另一个定义：

```
int account_number;
int balance_owed;
```

我们知道，我们正处理一个记账程序，但我们还需要使用更多的信息。例如，balance_owed是以元还是以分为单位？如果在每个说明之后加上注释，解释我们的目的就好多了。

```
int account_number;      /* Index for account table */
int balance_owed;        /* Total owed us  (in pennies) */
```

在每个说明的后面加一个注释，实际上就建立了一个小字典。在字典中，定义了每个变量名的意义。由于每个变量的说明都在一个已知的位置，所以很容易查到一个名字的具体含义（程序设计工具，如编辑器、交叉引用器和grep等查找工具，也可以帮助你快速地找到一个变量的定义）。

单位很重要。有一次别人要求我修改一个程序，这个程序把土地数据文件从一种格式转换为另一种格式。在程序中到处使用了许多不同的长度单位，而且在变量说明中没有注释。我努力想找出程序中使用的是什么单位，但实际上这是不可能的。最后，我不得不放弃，并在程序中加上了下面的注释：

```
/*****************************************************
 * Note:  I have no idea what the input units are, nor  *
 *      do I have any idea what the output units are,   *
 *      but I have discovered that if I divide by 3     *
 *      the plot sizes look about right.                *
 *****************************************************/
```

你应该尽可能地确保你的程序清楚易懂，不要耍小聪明，那样只会产生不可读且不可维护的程序。从程序本身来看，的确非常复杂。你所做的任何降低这种复杂性的努力，都会使程序变得更好。下面是由一位聪明的程序员编写的代码（注2）：

```
while ('\n' != (*p++ = *q++));
```

读者看一遍之后，几乎不可能说出这么多乱七八糟的符号是什么意思。再看看另一种写法：

```
while (1) {
    *destination_ptr = *source_ptr;
    if  (*destination_ptr == '\n'
        break;    /* Exit the loop if at end of line */
    destination_ptr++;
    source_ptr++;
}
```

虽然后一种写法较长，但它更清楚易懂（注3）。即使不太熟悉C语言的程序员，也能说出这段程序是把数据从源地址移到目标地址中。计算机并不在意程序员使用哪一种写法，好的编译器会为两种写法产生相同的机器代码，只有程序员才是其中的受益者。

注2：　注意：这句代码在本书第一版中是这么写的：

```
while ('/n' != *p++ = *q++);
```

它带有一个句法错误，因而不能被编译。在本书第一版出版后的头五年里没有人发现这个错误。

注3：　专业C编程人员能看出两种写法的细微差别，但两者均可实现同一功能。

编码盲从

计算机学家已设计出了多种编程方法，包括结构化程序设计、自顶向下的程序设计和少用goto语句的程序设计。每种方法都有自己的追随者或崇拜者。这里之所以使用“盲从”一词，是因为人们总是被告之要遵从一些规则，却不知道其中的缘由。例如，盲目追求少用goto语句的人从来不使用一个**goto**语句，即使在该使用的时候也不用。

本书中介绍的规则是多年编程实践之心得，按照这些规则，可以写出更好的程序。当然读者也不必盲目地遵从这些规则，如果你能找到更好的一套规则，不妨使用它（如果它真的能起作用，请告诉我，我很乐意使用）。

缩进与编码格式

为了增强程序的可读性，大多数程序员都在他们的程序中采用缩进的方式。C程序的一般规则是每一个新块或是条件语句都要缩进一层，上例中有三层逻辑，各自都有缩进层。**While**语句在最外层，**While**所包含的语句是下一层，**If（break）**内的语句在最内层。

有两种形式的缩进写法，第一种是简短形式：

```
while (! done) {
    printf("Processing\n");
    next_entry();
}
if (total <= 0) {
    printf("You owe nothing\n");
    total = 0;
} else {
    printf("ou owe %d dollars\n", total);
    all_totals = all_totals + total;
}
```

在这种写法中，大多数括号（{}）和语句放在同一行中。另一种风格是括号单独占一行：

```
while (! done)
{
     printf("Processing\n");
     next_entry();
}
if (total <= 0)
{
   printf("You owe nothing\n");
   total = 0;
}
else
{
   printf("You owe %d dollars\n", total);
   all_totals = all_totals + total;
}
```

这两种格式都很常用，用户可使用自己感觉舒服的格式。本书采用的是简短格式，因为它更直接并且能节省空间。

缩进量由程序员自己决定，常见的有2个、4个和8个空格。研究表明，4个空格的缩进格式可使程序更易读。但保持缩进格式的一贯性要比所使用的缩进尺寸重要得多。

有些编辑器，像UNIX Emacs、Turbo C++、Borland C++以及Microsoft Visual C++的内部编辑器，本身就包含了自动缩进的功能。虽然这些编辑器的缩进功能并不完美，但为帮助程序员创建适宜的代码格式也确实做了不少工作。

清晰

程序读起来应该像一篇技术论文，分为节和段。程序自然地形成节（第九章我们将学到函数），程序员把代码组织成段落。每段开头有一个主题注释句，段落之间用空行隔开。例如：

```
/* poor programming practice */
temp = box_x1;
box_x1 = box_x2;
box_x2 = temp;
```

```
temp = box_y1;
box_y1 = box_y2;
box_y2 = temp;
```

下面是改进后的写法:

```
/*
 * Swap the two corners
 */

/* Swap X coordinate */
temp = box_x1;
box_x1 = box_x2;
box_x2 = temp;

/* Swap Y coordinate */
temp = box_y1;
box_y1 = box_y2;
box_y2 = temp;
```

简明

程序应尽量简单。下面是一些经验之谈:

- 每个函数不应长于两页或三页(见第九章)。如果函数很长,可以把它分成两个更简单一些的函数。毕竟人脑的短期记忆容量是有限的,通常三页纸是人脑一次记住的最大量。对于那些超过三页的函数,其定义要么不是一个单独的操作,要么就是包含了过多的细节。
- 避免使用像多个嵌套的if语句这样的复杂逻辑。代码越复杂,需要的缩进层也就越多。当代码接近右边界时,就该考虑把代码划分为多个过程,从而减小代码的复杂程度。
- 你是否读过这样的句子,作者写了又写,一个句子接着一个句子,中间用"and"连接起来,或公式有可能要占一两行,就要把它分成两个短些的式子。
- 最后,最重要的一条规则是:让程序尽量简单易懂,即使违背了某些规则也无关紧要。我们的目的是使代码清楚,本章中所列出的这些规则都是为了帮

程序员来达到这一目的。如果某些规则有悖于此，可以不去管它。我曾见过一个程序，它只有一条语句，该语句长达20多页；但由于程序在设计上采用了适宜的规则，所以很简单且易于理解。

小结

程序应该简明易读。它既是一组计算机指令，也是描述算法和数据的参考书。应该尽量使用注释，注释有两个目的：其一，描述程序；其二，帮助程序员记录做过的事情。

课堂讨论：画出一张风格表作为课堂作业，讨论程序中应该包含哪些注释及其原因。

第四章

基本定义与表达式

本章内容

- 程序要素
- 程序的基本结构
- 简单表达式
- 变量和存储
- 变量定义
- 整型
- 赋值语句
- Printf 函数
- 浮点型
- 浮点数与整数的除法运算
- 字符
- 答案
- 编程练习

千里之行，始于足下。

——老子

如果木匠盖楼像程序员编程一样，则第一个啄木鸟的出现就将毁灭整个文明。

——Weinberg 第二定律

程序要素

盖楼需要准备两件事：砖和蓝图，其中蓝图说明如何把砖垒在一起。在计算机上设计程序也需要两件事：数据（变量）和指令（代码），其中变量是构筑程序的基本材料，而指令则告诉计算机该对变量进行何种操作。

注释用来描述变量和指令，是作者为程序而作的注解，应该清楚易读。计算机在编译时忽略掉所有的注释。

在建筑行业，开始一项工程之前，必须预订所需的原材料："需要500块大砖、80块半截砖和4块铺路石"。类似地，在C中，使用变量之前必须先定义它们，如同为每块"砖"命名，并告诉C要使用的砖是何种类型。

变量定义后才可以使用。建筑上的基本结构是房间，把许多房间组合在一起就形成了一座建筑物。在C中，基本的结构是函数，函数组合在一起便成为程序。

初出茅庐的建筑师不会先去盖帝国大厦，而须先从一居室的房子着手。本章中，你应先集中精力学习编写简单的单个函数。

程序的基本结构

程序的基本要素是数据定义、函数和注释。下面让我们看看这些要素是怎样组织成一个简单的C语言程序。

单个函数的基本结构如下：

```
/********************************************************
 * ...Heading comments...                               *
 ********************************************************/
...Data declarations...
int main()
{
        ...Executable statements...
        return (0);
}
```

首段注释记载有关该函数的信息，数据定义描述了函数中将使用的数据。

函数main()是唯一的。这个函数名比较特殊，因为它是调用的第一个函数，其它的函数都直接或间接地在main()中调用。main()函数的开头是：

```
int main()
{
```

它的结尾是：

```
        return (0);
}
```

return(0)语句用来告诉操作系统（UNIX或MS-DOS/Windows）程序已正常退出（Status=0）。非零的status表明有错误——返回值越大，错误越严重。一般而言，return(1)表示因各种简单的错误而退出，如缺少文件或命令行语法错误。

下面让我们来看看 Hello World 程序（例 3-1）。程序的开头是一个由 /* 和 */ 围成的注释框，注释框的下面是这样一行：

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

这一语句告诉C将使用标准I/O函数库，这是一种类型的数据定义（注1）。在此定义之后，才可以使用这个函数库中的 printf 函数。

main（）函数包含下列指令：

```
printf("Hello World\n");
```

这是一条可执行语句，它将指示C在屏幕上显示“Hello World”信息。C使用分号（;）来结束一条语句。就像我们用句号结束一段句子一样。与BASIC这种面向行的语言不同，C语言中行的结束并不意味着语句的结束。本书中的句子可以写在若干行内 —— 行尾只看作是分隔词的一个空格。C采用同样的方法，一条语句可以跨好几行。同样也可以把几个C语句放在同一行，正如同可以在同一行写几个句子一样。在大多数情况下，如果每条语句均开始于单独一行，则程序可读性将更强。

标准函数 printf 用来输出信息。常用函数库（library routine）是一些写好的C程序或函数的集合，这些常用函数可执行排序、输入、输出、数学运算以及文件处理。函数库的完整列表见C参考手册。

Hello World 是最简单的C程序之一。它不包含计算，仅仅是把一条信息显示在屏幕上。但这只是个开端，掌握了这一简单程序之后，才能正确地处理一些东西。

简单表达式

计算机不仅能打印字符串，还能执行计算，通常用表达式来说明简单计算。C语言的五种简单运算符如表 4-1 所示。

注1：　技术上讲，该指令调用了一个 include 文件中的系列数据声明，在第十章“C预处理器”中将讨论 include 文件。

表 4-1 简单操作

运算符	含义
*	乘法
/	除法
+	加法
-	减法
%	取模（返回整除后的余数）

乘（*）、除（/）和模（%）的优先级比加（+），减（-）的优先级高。括号用来进行几项综合运算，如：

```
(1 + 2) * 4
```

结果为12，又如：

```
1 + 2 * 4
```

结果为9。

例4-1计算表达式（1+2）*4的值。

例4-1：simple/simple.c

```
int main()
{
    (1 + 2) * 4;
    return (0);
}
```

虽然程序计算出了结果，但并没有用它（该程序将产生“null effect”的警告信息，指出程序中会有编写正确、但不起作用的语句）。

设想一下，盖房子时，如果我们对工人说：“推上你的手推车，在卡车与房基之间来回走，”

“你是不是想让我们运砖？”

“不，只是来回走”。那位工人肯定大惑不解。

因此程序应将计算的结果储存起来。

变量和存储

C允许把值存放到变量中，每个变量都由一个变量名来标识。另外，每个变量都有一个变量类型。变量类型告诉C该变量随后的用法及保存的类型（实型或整型等）。名字由字母或下划线（_）开头，后跟任意个字母、数字或下划线。大、小写不等同，所以名字sam、Sam和SAM是指三个不同的变量。不过为了避免混淆，应该使用不同的变量名，而不要以大小写来区分。

创建以下划线开头的名字也可以，但是，这样的名字通常被保留为内含的和系统名，容易混淆。

大多数C程序员习惯使用全小写变量名。在C语言中，**int**、**while**、**for**和**float**等称为保留字，具有特定的含义，不能用作变量名。

下面是一些变量名的例子：

```
average            /* average of all grades */
pi                 /* pi to 6 decimal places */
number_of_students /* number students in this class */
```

下列字符串不能作为变量名：

```
3rd_entry   /* Begins with a number */
all$done    /* Contains a "$" */
the end     /* Contains a space */
int         /* Reserved word */
```

应避免相似的变量名。例如，下面的变量名就取得不好：

```
total       /* total number of items in current entry */
totals      /* total of all entries */
```

一组较好的变量名如下:

```
entry_total /* total number of items in current entry */
all_total   /* total of all entries */
```

变量定义

C语言中在使用一个变量之前，必须先进行变量定义。

变量定义具有三个目的:

1. 定义变量名。
2. 定义变量类型（整型、实型、字符型等）。
3. 向程序员描述该变量，如对某变量answer的说明如下:

```
int answer;      /* the result of our expression */
```

关键字**int**告诉C这个变量包含一个整型值（整型在后面定义），变量名是answer。

分号表示一条语句结束，注释用来解释这个变量，它是给程序员的信息（对每个C语言的变量定义进行注释的要求只是一种风格规则，C允许不写注释，但任何有经验的程序员都不会漏掉）。

变量定义的一般格式是:

```
type  name;    /* comment */
```

type 指的是变量类型；*name* 是任意的有效变量名。这一变量定义说明了变量叫什么以及用来做什么的（在第九章中，我们将看到局部变量怎样定义）。

变量定义位于程序的开头，在main()一行的前面。

整型

整型是变量类型之一、没有小数部分或是小数点，如1、87和-222都是整数。8.3就不是整数，因为它包含一个小数点。整型说明的一般形式如下:

```
int  name;    /* comment */
```

8位的计算器只能处理99999999到-99999999之间的数，如果想让99999999加1则会出现溢出错误。计算机也有类似的限制，整数的限制依实现的不同而不同，就是说，计算机系统不同，整数的限制也不同。

计算器使用十进制数字（0-9），而计算机使用二进制数字（0-1），每个二进制数字称之为位（bit）。8个位组成一个字节（byte），存储整数的位数依机器系统的不同而不同，使用时数字将由二进制转化为十进制。

在大部分UNIX机器上，整数是32位(4个字节)的，它们表示的范围是2147483647（2^{31}-1）到-2147483648（-2^{31}）。在PC机上的Turbo C系统下，用16位（2个字节）来表示整数，所以表示的范围为32767（2^{15}-1）到-32768（-2^{15}），这些范围比较典型。标准的头文件limits.h定义了不同数字范围的常量（更多关于头文件的信息见第十八章“模块化程序设计”）。

标准C不限定数字的确切范围，其程序依赖于特定的整数（如32位），当转移到其他机器时常常不能运行。

问题4-1：下列代码可以在UNIX机器上运行，但在PC机上运行失败。

```
int zip;        /* zip code for current address */
..........
zip = 92126;
```

为什么会失败，在PC机上运行结果将会如何?

赋值语句

通过赋值语句可以给变量赋一个值，例如:

```
answer = (1 + 2) * 4;
```

等号（=）左边的变量answer被赋以等号右侧表达式（1+2）*4的值，分号结束该语句行。

变量定义为变量创建空间，图4-1A说明了变量answer的定义，这里还没有给它赋值，所以称其为未初始化的变量。问号表明这一变量的值尚属未知。

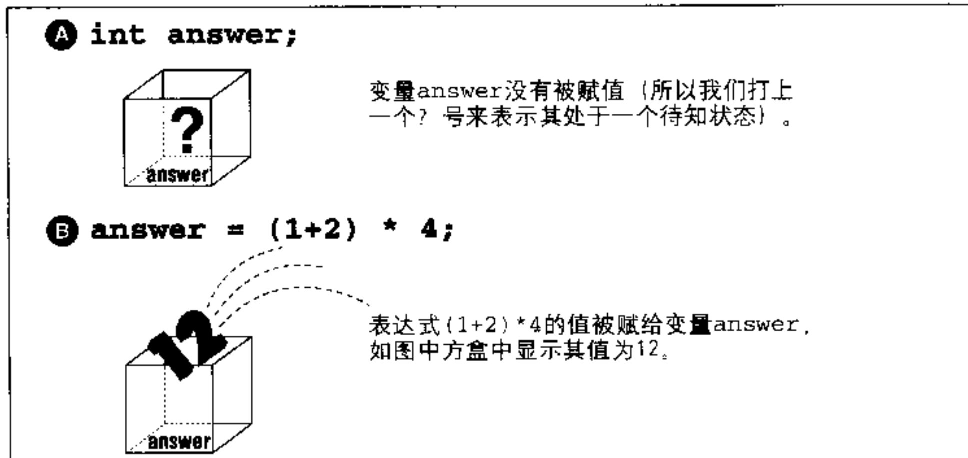


图4-1　定义answer变量并给其赋值

赋值语句用来给变量一个值，例如：

```
answer = (1 + 2) * 4;
```

等号（=）左边的变量answer被赋以等号右侧表达式（1+2）*4的值。所以变量answer获得值12，如图4-1B。

赋值语句的一般形式如下：

```
变量 = 表达式;
```

等号用来赋值。确切的意思是：计算表达式并把表达式的值分配给变量（在其它一些语言里，如PASCAL，等号用来测试是否相等，而在C中则用来赋值）。

例4-2中，用变量term存储一个整数值，这个值在随后的两个表达式中要用到。

例 4-2: term/term.c

```
[File: term/term.c]
int term;       /* term used in two expressions */
int term_2;     /* twice term */
int term_3;     /* three times term */
int main()
{
    term = 3 * 5;
    term_2 = 2 * term;
    term_3 = 3 * term;
    return (0);
}
```

这个程序存在一个问题：怎么才能知道它是否运行？我们需要用某种方式来显示出结果。

printf 函数

标准函数 printf 能用来显示结果，如果在程序中加入下列语句：

```
printf("Twice %d is %d\n", term, 2 * term);
```

程序运行后，就会显示：

```
Twice 15 is 30
```

%d 为整型格式符。函数中出现 %d 时，就会把下一个表达式的值显示在格式串的后面，称为参数列表（parameter list）。

printf 语句的一般形式是：

```
printf (format, expression-1, expression-2, ...);
```

format 是描述显示内容的字符串，串内的所有字符将逐字显示，但 %d 除外。表达示 *-1* 的值显示在第一个 %d 的位置上，表达示 -2 的值显示在第二个 %d 的位置上，依此类推。

图 4-2 显示了 printf 语句的各个组成部分如何共同运行而产生的最终结果。

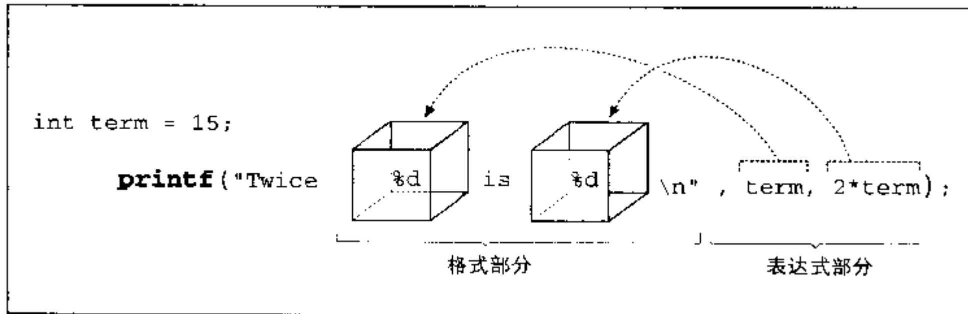


图 4-2 printf 结构

格式中“Twice %d is %d\n”告诉 printf 显示 Twice，空格，第一个表达式的值，空格，is，空格，第二个表达式的值，换行符（以 \n 表示）。

例 4-3 列出的程序计算了 term 的值并通过两个 printf 函数显示结果。

例 4-3: twice/twice.c

```
[File: twice/twice.c]
#include <stdio.h>

int term;       /* term used in two expressions */
int main()
{
    term = 3 * 5;
    printf("Twice %d is %d\n", term, 2*term);
    printf("Three times %d is %d\n", term, 3*term);
    return (0);
}
```

printf 函数格式中 %d 的数目应该与表达式的数目一一对应，C 不会对此进行核对（实际上，如果打开适当的警告程序，GNU *gcc* 编译器会对 printf 函数进行

检查)。如果表达式数目过多，多余的就会被忽略；如果表达式不足，C就会产生一些错误的数字。

浮点型

实数也被称作浮点数，5.5、8.3和-12.6都是浮点数。C采用小数点区分浮点数和整数，如5.0是浮点数而5是整数。浮点数必须包含一个小数点，如3.14159、0.5、1.0和8.88。

虽然可以忽略掉小数点以前的数字，如把0.5写成.5，但额外的0会使浮点数更清楚。用12.来表示12.0时也有类似的问题，浮点数的零应该写作0.0。

另外，浮点数可以包含指数形式 e+exp。

例如，1.2×10^{34} 可以缩写成1.2e34。

浮点型定义的一般形式如下：

```
float 变量;    /*注释*/
```

计算机处理浮点数也是有范围限制的。计算机系统不同，处理的范围也有很大的不同。有关浮点数的精度将在第十六章“浮点数”中讨论。

当用printf显示一个浮点数时，会用到%f来转换。要显示表达式1.0/3.0，我们使用如下的语句：

```
printf("The answer is %f\n", 1.0/3.0);
```

浮点数与整数的除法运算

除法运算符很特别，整数除法和浮点数除法有很大的不同。在整数除法中，结果要取整（即舍去小数部分），如19/10的结果为1。

如果除数或被除数中有一个是浮点数，那么将执行浮点数除法，如19.0/10.0的结果为1.9（19/10.0与19.0/10也是浮点数除法，但最好用19.0/10.0这种形式）。表4-2中还有几个例子。

表4-2　表达式示例

表达式	结果	结果类型
1 + 2	3	整型
1.0 + 2.0	3.0	浮点数
19 / 10	1	整型
19.0 / 10.0	1.9	浮点数

C允许把一个整数表达式的值赋给一个浮点变量，此时将自动进行整数到浮点数的转换。同样，把一个浮点数赋给整型变量时也要做类似的转换，例如：

```
int   integer;  /* an integer */
float floating; /* a floating-point number */

int main()
{
    floating = 1.0 / 2.0;         /* assign "floating" 0.5 */
    integer = 1 / 3;              /* assign integer 0 */
    floating = (1 / 2) + (1 / 2); /* assign floating 0.0 */
    floating = 3.0 / 2.0;         /* assign floating 1.5 */
    integer = floating;           /* assign integer 1 */
    return (0);
}
```

注意表达式 1 / 2 是一个整型表达式，所以要执行整数除法，结果是整数 0。

问题4-2：例4-4的结果为什么是“0”对这个程序要进行哪些必要的修改？

例 4-4: q_zero/q_zero.c

```
#include <stdio.h>
float answer;   /* The result of our calculation */
int main()
{
    answer = 1/3;
```

```
    printf("The value of 1/3 is %f\n", answer);
    return (0);
}
```

问题4-3：为什么2+2=5928？（你的结果可能不同。见例4-5。）

例 4-5: two/two.c

```
[File: two/two.c]
#include <stdio.h>

/* Variable for computation results */
int answer;

int main()
{
    answer = 2 + 2;

    printf("the answer is %d\n")
    return (0);
}
```

问题4-4：为什么会显示错误的结果？（见例4-6）

例 4-6: div/div.c

```
[File: div/div.c]
#include <stdio.h>

float result;    /* Result of the divide */

int main()
{
    result = 7.0 / 22.0;

    printf("The result is %d\n", result);
    return (0);
}
```

字符

char 类型表示单个字符，字符定义的形式如下：

```
char 变量;  /*注释*/
```

字符要放到单引号（'）内，如'A'、'a'、'!'是字符变量。反斜杠（\）称作 escape 符，表明后面的字符是一个特殊字符。例如，字符串里，\" 表示双引号，\' 表示单引号。\n是换行符，使输出设备定位到下一行的行首（与打字机上的回车键类似）。字符\\表示其本身。最后，字符可由\nnn 来指定，nnn的位置是字符的八进制码。表 4-3 归纳了这些字符。附表 A，ASCII 表中包含了 ASCII 字符码列表。

表 4-3： 特殊字符

字符	名称	含义
`\b`	Backspace	将光标左移一个字符
`\f`	Form Feed	走纸换页
`\n`	Newline	转到下一新行
`\r`	Return	回车
`\t`	Tab	下一个 tab 停止位
`\'`	Apostrophe	字符 '
`\"`	Double quote	字符 "
`\\`	Backslash	字符
`\nnn`		字符数 *nnn*（八进制）

注意： 字符用单引号（'），字符串用双引号（"）。记住这两种类型差别的一个方法是：单字符用单引号，字符串包含任何数目的字符（包括一个字符），要用双引号。

字符使用 `printf` 中的 `%c`，例 4-7 颠倒三个字符。

例 4-7：rev/rev.c

```
[File: rev/rev.c]
#include <stdio.h>
```

```
char char1;     /* first character */
char char2;     /* second character */
char char3;     /* third character */

int main()
{
    char1 = 'A';
    char2 = 'B';
    char3 = 'C';
    printf("%c%c%c reversed is %c%c%c\n",
        char1, char2, char3,
        char3, char2, char1);
    return (0);
}
```

执行此程序后，将显示：

```
ABC reversed is CBA
```

答案

解答4-1：在多数UNIX机器上，一个int中可存储的最大数是2147483647。使用Turbo C++时，最大数是32767。zip码中92126比32767大。所以，程序不能运行，输出结果为26590。

用**long int**取代**int**就可以解决这个问题。我们将在第五章"数组、修饰符与读取数据"中讨论整数的不同类型。

解答4-2：这个问题涉及除法：1/3。数1和数3都是整数，所以这是一个整数除法。在整数除法中，小数要被截断。表达式应写成：

```
answer = 1.0 / 3.0
```

解答4-3：下列printf语句：

```
printf("The answer is %d\n");
```

告诉程序显示一个十进制数，但是没有指定变量。C不会检查并确认`printf`函数被赋了正确的常量。由于没有值，C生成了一个值。正确的`printf`语句应是：

```
printf("The answer is %d\n", answer);
```

解答4-4：问题在`printf`语句中，我们使用`%d`来指定要显示的整数，但要转化的量是个浮点数。`printf`函数无法检查参量类型，所以如果把浮点数赋给函数，而格式指定的是整数，函数就会把这个数当作整数来看待，并显示出意料之外的结果。

编程练习

练习4-1：编程打印你的名字、身份证号和出生日期。

练习4-2：编程打印一个用星号（*）组成的字母E，E有7行高，5个字符宽。

练习4-3：编写一个程序，计算宽三英寸、长五英寸的矩形的面积和周长。如果要计算的矩形是长6.8英寸、宽2.3英寸的话，需要做哪些修改？

练习4-4：编程，用字母组成"HELLO"，显示的每个字母7行高，5个字符宽。

练习4-5：编程，令该程序中故意包含下列错误：

- 用`%d`显示一个浮点数。
- 用`%f`显示一个整数。
- 用`%d`显示一个字符。

第五章

数组、修饰符与读取数字

本章内容

- 数组
- 串
- 读取串
- 多维数组
- 读取数字
- 变量初始化
- 整型
- 浮点型
- 常量说明
- 十六进制与八进制常量
- 快捷运算符
- 副作用
- ++x 或 x++
- 更多的副作用问题
- 答案
- 编程练习

政治计算神秘而独立的变量是：公众观点。

——赫胥黎（英国科学家）

数组

在盖房子过程中，我们给每块砖都取了名字。砖数不大时还好说，当我们想盖更大的建筑时该怎么办呢？我们会指着一块砖说：“这块砖用来建左墙，砖块1、砖块2、砖块3……”。

数组允许对变量进行类似的处理。一个数组是一组连续的内存空间，用来保存数据。数组中的每一项称为一个元素。典型的数组定义为：

```
/* List of data to be sorted and averaged */
int     data_list[3];
```

上例定义data_list为一个含3个元素的一维数组，data_list[0]、data_list[1]和data_list[2]是相互独立的变量。要使用数组元素，就要利用一个称为下标的数——方括号中的数。C是一个有趣的语言，它喜欢从0开始计数，所以这三个元素从0到2编号。

注意：常识会使人觉得把data_list定义为三元素数组时，data_list[3]是有效的。但data_list[3]是非法元素。

例5-1计算五个数字的合计数和平均数。

例5-1: array/array.c

```
[File: array/array.c]
#include <stdio.h>

float data[5];  /* data to average and total */
float total;    /* the total of the data items */
float average;  /* average of the items */

int main ()
{
    data[0] = 34.0;
    data[1] = 27.0;
    data[2] = 45.0;
    data[3] = 82.0;
    data[4] = 22.0;

    total = data[0] + data[1] + data[2] + data[3] + data[4];
    average =  total / 5.0;
    printf ("Total %f Average %f\n", total, average);
    return  (0);
}
```

该程序的输出结果为：

```
Total 210.000000 Average 42.000000
```

串

串是一组字符。C没有内部串类型，相反，串由字符型数组创建。实际上，串就是带有若干限制的字符型数组，其中的一个限制条件是用'\0'(NULL)来表示串的结束。

例如:

```
char    name[4];

int main()
{
    name[0] = 's';
    name[1] = 'a';
    name[2] = 'm';
    name[3] = '\0';
     return  (0);
}
```

此程序段建立了一个四元素的字符型数组。注意：必须为串结束符分配一个空间

串常量由双引号之间的正文组成。你或许已经注意到函数printf的第一个参量是一个串常量。C不允许把一个数组赋给另一个数组，所以不能写成如下形式的赋值语句:

```
name = "Sam",     /* Illegal */
```

使用标准库函数strcpy能将串常量拷贝到变量中（strcpy将拷贝完整的串，包括串结束符)。要把变量name初始化为"Sam"可以这样写:

```
#include <string.h>
char    name[4];
int main()
{
     strcpy(name, "Sam");/* Legal */
     return  (0);
}
```

C可使用不同长度的串。例如，下面的定义:

```
#include <string.h>
char string[50];
int main ()
{
    strcpy (string,"Sam";
}
```

将建立一个可以包含多达50个字符的数组。数组的大小是50，但串的长度是3。49个字符以内的任何一个串都可以存储在string内（有一个字符的位置留给NULL表示串结束）。

注意： 字符串和字符有很大不同，字符串由双引号表示而字符由单引号表示。"X"是一个单字符的串，而'Y'则只是一个单字符（串"X"占两个字符，一个是X，一个是串结束符(\0)，而字符'Y'占一个字节）。

有几个标准函数可以对串变量进行操作，如表5-1。

表5-1　部分串函数列表

功能	描述
strcpy (*string1*, *string2*)	把指向*string1*的字符拷贝到*string2*中去
strcat (*string1*, *string2*)	把字符串*string2*接到*string1*后面，*string1*最后面的'\0'被取消。
length = strlen (*string*)	统计字符串*string*中字符的个数（不包括终止符'\0'）
strcmp (*string1*, *string2*)	比较2个字符串*string1*，*string2*，当两个字符串相等时返回0，否则返回非0整数。

printf函数使用%s显示串变量的值，如例5-2所示。

例5-2: str/str.c

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

char name[30];     /* First name of someone */
int main ()
```

```
{
    strcpy (name, "Sam");     /* Initialize the name */
    printf ("The name is %s\n", name);
    return  (0);
}
```

例 5-3 为给名和姓赋值并把两个串联结在一起。

程序初始化变量first为名字（Steve），将姓氏（Oualline）放进变量last。为得到全名，把名字拷进full_name，然后用strcat加一个空格，再次调用strcat加入姓。

串变量的大小定为100，因为人的名字不会长过99个字符（如果真遇到了一个超过99个字符的名字，程序就会出现混乱。真正会发生的事情是：把名字写进内存，却无法访问它。访问将使程序崩溃，运行正常而给出错误的结果，或者出现其他不可预料的结果）。

例 5-3: full/full.c

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

char first[100];        /* first name */
char last[100];         /* last name */
char full_name[200];    /* full version of first and last name */

int main()
{
    strcpy(first, "Steve");        /* Initialize first name */
    strcpy(last, "Oualline");      /* Initialize last name */

    strcpy(full_name, first);      /* full = "Steve" */
    /* Note: strcat not strcpy */
    strcat(full_name, " ");        /* full = "Steve " */
    strcat(full_name, last);       /* full = "Steve Oualline" */

    printf("The full name is %s\n", full_name);
    return (0);
}
```

这个程序的输出结果是：

```
The full name is Steve Oualline
```

读取串

标准函数 fgets 能从键盘读取串，一般形式的 fgets 调用如下:

```
fgets(name, sizeof(name), stdin);
```

*name*定义为一个串变量（fgets 将在第十四章“文件输入/输出”作详细解释）.

讨论如下:

name

是一个字符型数组的名称，包括结束行字符在内的整行被读进这个数组。

sizeof (*name*)

表示要读取字符的最大数目（算上结束串的字符要加1）。sizeof 函数提供了一种便捷的方式来限制要读取的字符的最大数。这个函数将在第十四章做更详细的讨论。

stdin

为要读取的文件。本例中，是标准的输入或键盘文件。其他类型的文件见第十四章。

例 5-4 从键盘读取了一行字符并输出了它的长度。

例 5-4: length/length.c

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

char line[100]; /* Line we are looking at */

int main()
{
    printf("Enter a line: ");
    fgets(line, sizeof(line), stdin);

    printf("The length of the line is: %d\n", strlen(line));
```

```
    return (0);
}
```

运行这个程序，会得到：

```
Enter a line: test
The length of the line is: 5
```

但test串只有四个字符，多出的字符从哪来的呢？fgets包括结束行的串，所以第五个字符是换行符（\n）。

假设我们想把名字程序改成用以询问用户的姓名，例5-5显示了改进后程序的写法。

例5-5: full1/full1.c

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

char first[100];        /* First name of person we are working with */
char last[100];         /* His last name */

/* First and last name of the person (computed) */
char full[200];

int main() {
    printf("Enter first name: ");
    fgets(first, sizeof(first), stdin);

    printf("Enter last name: ");
    fgets(last, sizeof(last), stdin);

    strcpy(full, first);
    strcat(full, " ");
    strcat(full, last);

    printf("The name is %s\n", full);
    return (0);
}
```

然而，运行这个程序后却得到如下结果：

```
% name2
```

```
Enter first name: John
Enter last name: Doe
The name is John
 Doe
%
```

"John"与"Doe"本来应该在同一行显示，是什么原因使得它们没有在同一行呢？fgets函数显示整行，包括结束行，因此在显示前必须去掉这个字符。

例如，名字"John"存储如下：

```
first[0] = 'J'
first[1] = 'o'
first[2] = 'h'
first[3] = 'n'
first[4] = '\n'
first[5] = '\0'    /* end of string */
```

通过把`first[4]`设为NULL（`'\0'`），可以把串缩短为一个字符并去掉不想要的新行，这一改变可由下列语句来完成：

```
first[4] = '\0';
```

问题是这种方法只适合于四个字符的名字。如何用一个一般的算法来解决这个问题呢？串的长度是串尾字符零的标记，而结束字符的前面就是要去掉的字符。为了改进这个字符串，可使用如下的语句：

```
first[strlen(first)-1] = '\0';
```

新程序见例5-6。

例5-6: full2/full2.c

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

char first[100];        /* First name of person we are working with */
char last[100];         /* His last name */

/* First and last name of the person (computed) */
```

```
char full[200];

int main() {
    printf("Enter first name: ");
    fgets(first, sizeof(first), stdin);
    /* trim off last character */
    first[strlen(first)-1] = '\0';

    printf("Enter last name: ");
    fgets(last, sizeof(last), stdin);
    /* trim off last character */
    last[strlen(last)-1] = '\0';

    strcpy(full, first);
    strcat(full, "");
    strcat(full, last);

    printf("The name is %s\n", full);
    return (0);
}
```

运行这个程序显示如下结果:

```
Enter first name:  John
Enter last name:  Smith
The name is John Smith
```

多维数组

数组可以多维。两维数组的定义如下:

```
type variable[size1][size2];  /* Comment */
```

例如:

```
int matrix[2][4]; /* a typical matrix */
```

注意: C 并没有使用其他语言中的惯用法 matrix[10,12]。

要访问 matrix 中的元素，可以用下列语句:

```
matrix[1][2] = 10;
```

C 对数组的维数并没有限制（只受机器的内存所限），可以加另外的维，如：

```
four_dimensions[10][12][9][5];
```

问题 5-1：为何例 5-7 显示错误？

例 5-7: p_array/p_array.c

```
#include <stdio.h>

int array[3][2];          /* Array of numbers */

int main()
{
    int x,y;     /* Loop indicies */

    array[0][0] = 0 * 10 + 0;
    array[0][1] = 0 * 10 + 1;
    array[1][0] = 1 * 10 + 0;
    array[1][1] = 1 * 10 + 1;
    array[2][0] = 2 * 10 + 0;
    array[2][1] = 2 * 10 + 1;

    printf("array[%d] ", 0);
    printf("%d ", array[0,0]);
    printf("%d ", array[0,1]);
    printf("\n");

    printf("array[%d] ", 1);
    printf("%d ", array[1,0]);
    printf("%d ", array[1,1]);
    printf("\n");

    printf("array[%d] ", 2);
    printf("%d ", array[2,0]);
    printf("%d ", array[2,1]);
    printf("\n");

    return (0);
}
```

读取数字

到目前为止，我们只讲了读取简单的字符串。同样地还需要让程序能读取数字。函数scanf工作原理和printf相似，但scanf只读数字，不写数字。它提供了一种简单容易的读取方式，但因为它对结束行处理得不好，对专业人员来说几乎没有用处。

于是，人们很快找到了一种弥补scanf缺陷的简单方法——不单独使用它。取而代之的是，用fgets读取输入的一行并用sscanf把文本转换成数字型（名称sscanf代表“string scanf”，sscanf 类似于scanf，但它作用于串而不是标准输入）。

下面语句中的变量line用于从键盘读取各行：

```
char line[100];     /* Line of keyboard input */
```

处理输入的数据可使用下列语句：

```
fgets(line, sizeof(line), stdin);
sscanf(line, format, &variable1, &variable2 . . .;
```

这里fgets读取一行，sscanf处理它。*format*和函数printf格式串相似。注意：变量名前的&符号。这个符号用来表明sscanf会改变相关变量的值（至于为什么需要&，见第十三章“简单指针”）。

注意：如果你忘了给sscanf的每个变量前放&符号，结果将出现“Segmentation violation core dumped”或者“Illegal memory access”错误。有时会改变随机变量或指令，UNIX系统下，破坏只局限于当前程序；但是，在MS-DOS/Windows下，由于缺乏内存保护措施，这个错误容易导致更大的破坏；在MS-DOS/Windows下，漏掉&将导致程序甚至系统崩溃。

例5-8中，使用sscanf得到用户输入的一个数，然后倍乘这个数：

例5-8：double/double.c

```
[File: double/double.c]
```

```
#include <stdio.h>
char  line[100];    /* input line from console */
int   value;        /* a value to double */

int main()
{
    printf("Enter a value: ");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &value);

    printf("Twice %d is %d\n", value, value * 2);
    return (0);

}
```

该程序读取一个数，然后倍乘它。注意，在“`Enter a value:`”的后面没有`\n`。这种遗漏是故意的，因为我们不想在提示后换行。该程序运行时的情形如下：

```
Enter a value: 12
Twice 12 is 24
```

如果把“`Enter a value:`”换为“`Enter a value:\n`”，则结果会是：

```
Enter a value:
12
Twice 12 is 24
```

问题 5-2：例 5-9 计算给定了底和高的三角形的面积。由于某种奇怪的原因，编译器误认为没有说明变量`width`，但说明就在第三行，即 height 变量说明之后。编译器为什么看不到它呢？

例 5-9: tri/tri.c

```
#include <stdio.h>
char line[100];/* line of input data */
int  height;    /* the height of the triangle
int  width;     /* the width of the triangle */
int  area;      /* area of the triangle (computed) */

int main()
{
```

```
    printf("Enter width height? ");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d %d", &width, &height);

    area = (width * height) / 2;
    printf("The area is %d\n", area);
    return (0);
}
```

变量初始化

C 允许在定义语句中对变量进行初始化。例如，下列语句说明了一个整型量 counter，并赋给它初值 0。

```
int counter = 0;    /* number cases counted so far */
```

用这种方法也可以给数组赋值。元素列表必须用括号（{}）括起来，例如：

```
/* Product numbers for the parts we are making */
int product_codes[3] = {10, 972, 45};
```

前面的初始化等价于：

```
product_codes[0] = 10;
product_codes[1] = 972;
product_codes[2] = 45;
```

{}中的元素个数不一定与数组大小一致。如果数据过多，会出现一条警告信息；如果数据不够，C 将把多余的元素初始化为 0。

如果没有数组维数，C 将从初始化元素的个数来决定维数。例如，可以用下列语句给变量 product_codes 初始化：

```
/* Product numbers for the parts we are making */
int product_codes[] = {10, 972, 45};
```

给多维数组赋初值与给一维数组赋初值类似。每维元素都放到一对括号（{}）中。下列定义：

```
int matrix[2][4]; /* a typical matrix */
```

可以看作是一个2维数组定义，这个数组的每个元素都是一个4维数组。对这个数组的初始化如下:

```
/* a typical matrix */
int matrix[2][4] =
    {
        {1, 2, 3, 4},
        {10, 20, 30, 40}
    };
```

串可以用同样方式初始化。例如，初始化变量name为串“sam”，可使用下列语句:

```
char    name[] = {'S', 'a', 'm', '\0'};
```

C初始化串有一种特殊的简化方式：把串放在双引号内来简化初始化。前例也可写成:

```
char name[] = "Sam";
```

name的维数为4，因为C在串尾分配了一个空间放“\0”字符。

下列定义:

```
char string[50] = "Sam";
```

等同于:

```
char string[50];
.
.
.
strcpy(string,"Sam");
```

该数组含50个字符，串的长度为3。

整型

C被看作是一种中级语言，因为它允许用户接近机器的实际硬件。有些语言，像BASIC（注1），不让用户了解处理器工作的细节，这种做法使语言降低了效率。C能让用户详细了解有关硬件使用的信息。

例如，大多数机器允许使用不同长度的数。简单的BASIC仅允许程序员使用一种类型的数，虽然这一限制条件简化了编程，却使BASIC程序的效率极低。C允许程序员定义许多整数，从而最大限度地利用硬件资源。

类型说明符**int**告诉C使用最有效长度（对使用的机器而言）的整数，根据机器的不同可以是2到4个字节（一些不太普通的机器使用奇怪的整数长度如9或40位）。

有时需要额外的位来存放比一般**int**所允许的还要大一些的数，下列定义：

```
long int answer;      /* the result of our calculations */
```

用来分配一个长整数。限定词**long**通知C用户希望为这个整数分配额外的存储空间，如果想用小数字并希望减少内存，可以用**short**作修饰词，例如：

```
short int year;            /* Year including the 19xx part */
```

C会确保**short**的存储空间小于**int**，而**int**小于或等于**long**。在实际中，**short**几乎总是分配2个字节；**long**分配4个字节，而**int**分配2个或4个字节（有关数的范围见附录二"范围和参数传递转化"）。

short int类型通常使用2字节，或16位。其中15位一般用来存放数字，1位放符号。这一类型的数的范围是-32768（-2^{15}）到32767（$2^{15}-1$）。**unsigned short int**数字全部使用16位，给定的范围是0到65535（2^{16}）。所有的**int**定义都默认为**signed**，如下定义：

```
signed long int answer;    /* final result */
```

注1：　有些高版本的BASIC有数字类型，但是，此处我们只讨论基本的BASIC。

等同于:

```
long int answer;          /* final result */
```

还有一种极短的整型: **char**类型。字符变量占1字节，可用来表示-128到127 (**signed char**) 或是0到255 (**unsigned char**) 之间的数。和整数不同，它不默认为signed; 默认值由编译器决定。极短整型数据的显示可采用整型转化 (%d)。

不能直接读取一个很短的整数，必须先将数据读入一个整型变量中，然后使用一个赋值语句。例如:

```
#include <stdio.h>
signed char ver_short;   /* A very short integer */
char line[100];          /* Input buffer */
int temp;                /* A temporary number */

int main()
{
    /* Read a very short integer */
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &temp);
    very_short = temp;
}
```

表5-2包含了整型的printf和sscanf转化。

表5-2　整型printf/sscanf转化

%转化	用法
%hd	(有符号) 短整型
%d	(有符号) int
%ld	(有符号) 长整型
%hu	无符号短整型
%u	无符号 int
%lu	无符号长整型

不同形式整数型的范围列表见附录二。

long int 数据允许程序明确指定必要的额外精度（在消耗内存时），**short int** 数据节省空间但范围有限。最简洁的整型是 **char** 类型，但也有一定的范围。

unsigned数据提供了一种以除去负数为代价来加倍正数范围的方式，这种方式对那些决不会是负数的东西如计数器和下标是有用的。

一般而言，使用何种数据类型取决于程序和存储的要求。

浮点型

浮点型也有很多种。**float** 表示正常的精度（通常为4字节），**Double** 代表双精度（通常8字节），双精度变量比单精度（float）变量的范围和精度都要大很多。

修饰词 **long double** 代表扩大精度。在一些系统中，它和 **double** 相同，而在其它系统中，它能提供附加精度。所有类型的浮点数都有符号。

表 5-3 包含浮点数的 `printf` 和 `sscanf` 转换。

表 5-3　浮点数的 printf/sscanf 转换

% 转换	用法	备注
`%f`	`float`	仅用于 `printf`
`%lf`	`double`	仅用于 `scanf`
`%Lf`	`long double`	在所有编译器中均不能使用

在一些机器上，单精度、浮点数指令比双精度指令执行得快一些（但精确性较低），双精度是以时间和内存为代价来获得较高的精确性。在多数情况下使用 **float** 就足够了，不过如果对精确性要求比较高的话，可以转换成 **double**（见第十六章“浮点数”）。

常量说明

在写程序的过程中，有时会用到一些不会改变的值，如π等。用关键字**const**可以定义这些值不改变的变量，例如可以用下列语句说明PI这个值：

```
const float PI = 3.1415927;     /* The classic circle constant */
```

注意：按照约定，变量只能使用小写字母，常量名只能用大写字母。但语言中并没有要求这种结构，有些系统使用完全不同的约定。

常量必须在定义时赋初值，而且这个值永远不能改变。例如，如果把PI的值重置为3.0，将会产生一条错误信息：

```
PI = 3.0;       /* Illegal */
```

当说明数组大小时，可以采用整型变量：

```
/* Max. number of elements in the total list.*/
const int TOTAL_MAX = 50;
float total_list[TOTAL_MAX]; /* Total values for each catigory */
```

注意：在C语言中以这种方式来定义常量是一种相对较新的方法，但不是所有的编译器都完全支持这种方法。

十六进制与八进制常量

整数是一串数字，如1234，88，-123等等，这些串是十进制数（基为10），如174或174_{10}。计算机处理的是二进制数（基为2）：10101110。八进制数（基为8）与二进制之间易于转化，三位二进制数（2^3=8）可以转换为一位八进制数，因此101011102可以写成10 101 110并变成8进制的256。十六进制数（基为16）有类似的规则；只是4位为一组。

表示八进制和十六进制数时，C语言有约定：以零开头表示一个八进制常量，例如0123是123（八进制）或83（十进制）。数字以“0x”开头表示一个十六进制常量（基为16），因此0x15就是21（十进制）。表5-4三种进制下的几个数。

表 5-4　三种进制下的整数

基为 10	基为 8	基为 16
6	06	0x6
9	011	0x9
15	017	0xF

快捷运算符

C不仅提供了一组丰富的数据类型，还提供了大量具有特殊目的的运算符。

程序员常常想把一个变量值增量（变量值加1）。如果使用通常的赋值语句，这个操作应书写如下：

```
total_entries = total_entries + 1;
```

C提供了一种快捷方式来执行这种任务，运算符 ++ 可用于增量操作：

```
++total_entries;
```

类似的运算符 —— 可用于减量（变量值减1）如下列运算：

```
--number_left;
/* is the same as */
number_left = number_left - 1;
```

但假设我们想加2而不是1，可用如下运算：

```
total_entries += 2;
```

这个运算等同于：

```
total_entries = total_entries + 2;
```

表 5-5 中所示的简单运算符都能用于这种运算。

表5-5 等值语句运算符

运算符	简写	等值语句
+=	x += 2;	x = x + 2;
-=	x -= 2;	x = x - 2;
*=	x *= 2;	x = x * 2;
/=	x /= 2;	x = x / 2;
%=	x %= 2;	x = x % 2;

副作用

不幸的是，C允许程序员使用副作用。副作用是指语句执行主操作时附带执行的操作，例如，下列合法的C代码：

```
size = 5;
result = ++size;
```

第一条语句给size赋值为5，第二条语句给result赋值为size（主操作）及size增1（副作用）。

但这一执行过程顺序如何呢？有四种可能的答案。

1. 把size的值（5）赋给result，然后size加1。最后，result是5，size是6。
2. size增1，然后把size的值（6）赋给result。最后，result是6，size是6。
3. 由编译器决定，而且视计算机的不同，最后的结果也不同。
4. 如果不写这样的代码，就不必担心这类问题。

正确答案是2：加1操作在赋值操作之前进行，但4是更好的回答。C的主要操作已足够我们学习的了，没有必要再去为副作用操心。

注意：有些程序员喜欢很紧凑的代码，这是从计算机发展的早期遗留下来的，因为那时内存的价格非常昂贵。依我看来编程艺术发展到今天，简洁的代码比紧凑的代码更有价值（许多人喜欢读的长篇小说并不是用速记法写出来的）。

实际上，C提供了两种形式的++运算符：一种是变量++；另一种是++变量。第一种：

```
number = 5;
result = number++;
```

先进行表达式的赋值，然后number加1，result值为5。第二种：

```
number = 5;
result = ++number;
```

先进行增量计算，然后进行表达式的赋值，result的值为6。但这样使用++或--可能会编出令人意想不到的代码：

```
o = --o - o--;
```

这一行的问题看上去像是有人在写莫尔斯电报码。程序员不会去读这行语句，而是先破译它。如果从不把++或--作为其它语句的一部分，而是总让它们单独一行，那么两种形式的差别就不会显得那么突出了。

++X 或 X++

这两种形式的增量操作符被称为前缀形式（++x）和后缀形式（x++）。实际上在C语言中究竟选择何种形式是无关紧要的，不过如果使用的是有大量运算符的C++，前缀形式（++x）会更好些（注2），因此为了学习C++，采用前缀形式不失为一种好的习惯（注3）。

注2：详情请参阅《Practical C++ Programming》（O'Reilly & Associates公司出版）

注3：有种具有讽刺意义的说法，即使用后缀名为语言名称的C++，实际上在实际运用中，以前缀形式作为增减运算符效率会更高，所以如果称为++C也许会更贴切。

更多的副作用问题

更为复杂的副作用甚至会使C编译器混乱，考虑下列程序片段：

```
value = 1;
result = (value++ * 5) + (value++ * 3);
```

这个表达式告诉C执行下列步骤：

1. value乘以5，并加1给value。
2. value乘以3，并加1给value。
3. 把两乘法的结果相加。

步骤1和2具有同等的优先权（和前例不同），所以编译器可以选择执行顺序。假设编译器先执行步骤1，如图5-1所示。

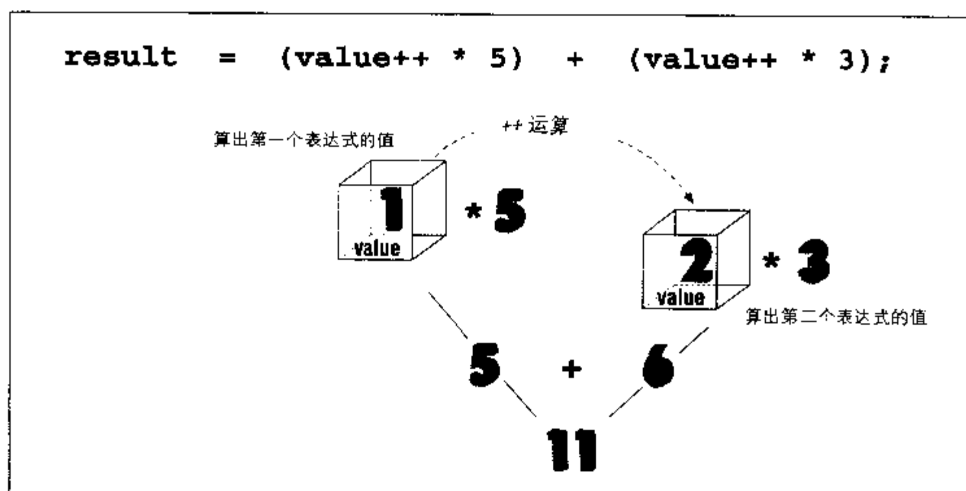


图5-1 表达式结算方法1

或假设编译器先执行步骤2，如图5-2。

第一种方法结果为11，第二种方法结果为13。这个表达式的结果模棱两可，取决于编译器如何执行，会有11或13两种结果。更糟的是，如果有优化功能，有些

编译器还会改变举动。那么“运行着”的代码在进行最优化处理后会产生什么破坏呢？

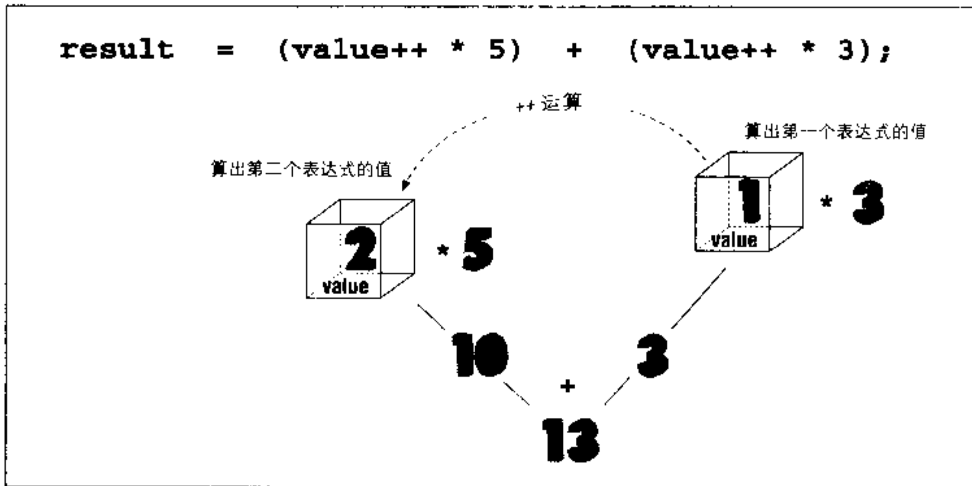


图 5-2　表达式结算方法 2

在一个较大的表达式中间使用++运算符也会产生问题（这个问题不是++和--导致的唯一问题，在第十章中我们会陷入更大的麻烦）。

为了避免出问题并保持简单，应总是把++和--单独放在一行。

答案

解答 5-1：问题是在`printf`语句中使用了表达式`array[x,y]`：

```
printf("%d ", array[x,y]);
```

多维数组的每个下标都必须放在自己的那组方括号中，语句应写成：

```
printf("%d ", array[x][y]);
```

如果想提前一点读取，逗号运算符可以用来把多个表达式连到一起。这个运算符的值是最后一个表达式的值，因此`[x,y]`和`y`相等，而且`array[y]`实际上是

数组y行的一个指针。由于指针有特殊值，printf会输出奇怪的结果。(见第十七章“高级指针”及第二十章“C内的角落”)。

解答5-2：程序员在注释完成之后不经意忘掉了结束注释(*/)，注释会延续到下一行，并把说明也包含在内，如例5-10。

例5-10:

```
#include <stdio.h>
char line[100];/* line of input data */
int  height;   /* the height of the triangle
int  width;    /* the width of the triangle */
int  area;     /* area of the triangle (computed) */

int main()
{
    printf("Enter width height? ");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d %d", &width, &height);

    area = (width * height) / 2;
    printf("The area is %d\n", area);
    return (0);
}
```

考虑该程序另外的一个次要问题。如果width和height都是多余的变量，将得到一个有轻微错误的答案。(怎么来改正这个错误呢？)

编程练习

练习5-1：编程把摄氏温度转为华氏温度。 ($F=\frac{9}{5}C+32$)

练习5-2：编程计算球的体积。 ($\frac{4}{3}\pi r^3$)

练习5-3：编程计算给定了长和宽的矩形的周长。(周长=2 × (宽+长))

练习5-4：编写程序把每小时英里数转换为每小时公里数。(英里=公里 × 0.6213712)

练习 5-5: 编程输入小时和分钟，然后计算总的分钟数。(1小时30分钟=90分钟)

练习 5-6: 编程输入分钟数，然后输出小时数和分钟数。(90分钟=1小时30分钟)

本章内容

- if语句
- else语句
- 怎样避免误用strcmp函数
- 循环语句
- while语句
- break语句
- continue语句
- 避免赋值的副作用
- 答案
- 编程练习

第六章

条件和控制语句

一旦做了决定，我就不担心后面的事情了。

——杜鲁门（美国总统）

计算机程序语句除了计算和表达式以外，还包括条件和控制语句，这些语句指定程序的执行顺序。

到目前为止，我们已经编写了线性程序（linear program），这些程序呈直线型，语句按顺序执行。在本章中，你将看到如何使用分支语句（branching statement）和循环语句（looping statement）来改变一个程序的控制流（control flow）。分支语句可以根据条件句（conditional clause）来决定一段代码是否执行，循环语句用来重复执行同一段代码若干次或执行代码直到某种条件出现。

if语句

if语句允许在程序中加入某些判断，if语句的一般形式如下：

```
if (条件)
    语句;
```

如果表达式为真（非零），则执行该语句；如果表达式为假（零），则不执行该语句。例如，假设我们要编写一个帐单程序，如果顾客不欠钱或者出现贷方余额（欠

我们一个负数金额)，就输出一条信息。在C中，这个程序书写如下：

```
if (total_owed <= 0)
    printf("You owe nothing.\n");
```

运算符<=是关系运算符，表示小于或等于。该语句读作“如果total_owed小于或等于零，则打印信息”。关系运算符的完整列表可在表6-1中找到。

表6-1 关系运算符

运算符	意义
<=	小于或等于
<	小于
>	大于
>=	大于或等于
==	等于(注)
!=	不等于

注：相等测试(==)和赋值运算符不同。C程序员易混淆它们是最常见的问题之一。

多条语句可成组地放进括号{}中。例如：

```
if (total_owed <= 0) {
    ++zero_count;
    printf("You owe nothing.\n");
}
```

从可读性考虑，括号{}中的语句要缩进一些，这样程序员可以迅速分辨出哪些语句的执行是有条件的。读者在后面将会看到，错误的缩进会对程序的含意产生误导，而且程序的可读性很差。

else语句

if语句的另一种形式是：

```
if (条件)
    语句;
else
    语句;
```

如果条件为真，将执行第一条语句；如果条件为假，则执行第二条语句。在前面的示例中，当顾客没有结余时才输出信息。实际上，如果有结余金额，还可以告诉顾客结余有多少：

```
if (total_owed <= 0)
    printf("You owe nothing.\n");
else
    printf("You owe %d dollars\n", total_owed);
```

现在考虑下列的程序段（存在不正确的缩进）：

```
if (count < 10)      /* if #1 */
    if ((count % 4) == 2)    /* if #2 */
        printf("Condition:White\n");
  else
      printf("Condition:Tan\n");
```

注意：PASCAL程序员注意：和PASCAL语言不同，C要求在else之前的语句的末尾加一个分号。

这里有两个**if**语句和一个**else**，这个**else**属于哪个**if**呢？

a. 它属于第一个**if**。

b. 它属于第二个**if**。

c. 如果读者从来没有这样写代码，就不必考虑这个问题。

正确的回答是b。根据C语法规则，**else**属于最近的一个**if**，所以b是对的。但写这样的代码是违背KISS（即Keep It Simple, Stupid）原则的，最好编写尽可能简单的程序。这段代码最好这样写：

```
if (count < 10) {        /* if #1 */
    if ((count % 4) == 2)    /* if #2 */
```

```
        printf("Condition:White\n");
    else
        printf("Condition:Tan\n");
}
```

在第一个例子中，很难弄清楚**else**子句属于哪个**if**语句，但是，增加一对括号{}后，大大增强了程序的可读性、可理解性及清晰度。

怎样避免使用 strcmp 函数

strcmp函数比较两个串，如果两个串相等，返回零；如果不等，返回非零。要检查两个串是否相等，使用下列代码：

```
/* Check to see if string1 == string2 */
if (strcmp(string1, string2) == 0)
    printf("Strings equal\n");
else
    printf("Strings not equal\n");
```

有些程序员漏掉了注释和==0子句，这样会引出下列令人费解的语句：

```
if (strcmp(string1, string2))
    printf("......");
```

乍看上去，程序显然是在比较两个串，当它们相等时，执行printf语句。不幸的是，这明显是错误的。如果串相等，strcmp返回零，不执行printf。由于strcmp函数有返回值，所以在使用strcmp时务必小心，并注释它的用途。同时用一个注释语句来解释程序的功能，也是有必要的。

循环语句

循环语句允许程序重复一部分代码若干次，或重复到某种条件出现。例如，循环语句可用来统计文档中的字数或者统计帐目的收支数。

while 语句

当程序重复执行某些操作时，可以使用 **while** 语句。**while** 语句的一般形式是:

```
while (条件)
    语句;
```

程序重复执行 **while** 中的语句直到条件为假（0）结束（如果开始时条件就为假，则语句一次也不执行）。

例如，本章例6-1计算100以内的斐波那契数（Fibonacci number）。斐波那契数列为:

```
1 1 2 3 5 8
```

这些项是通过下面等式算出的:

```
1
1
2 = 1 + 1
3 = 1 + 2
5 = 2 + 3
etc.
```

写成公式是:

$$f_n = f_{n-1} + f_{n-2}$$

这是一个使用数学变量名（f_n）的数学等式。数学家用精炼的方法给变量命名，但在设计程序时，精炼是危险的，所以我们还是把这些变量名变成适合C语言的长名。表6-2列出了这种改变。

表6-2　数学名称翻译成C名称

数学风格名称	C风格名称
f_n	next_number
f_{n-1}	current_number
f_{n-2}	old_number

在C中，这个等式表示为：

```
next_number = current_number + old_number;
```

要循环到当前项大于或等于100为止。**while** 循环：

```
while (current_number < 100)
```

将重复计算并输出结果，直到满足条件时结束。

执行程序时变量的情况如图6-1所示。开始时，current_number和old_number为1，显示当前项的值。然后计算变量next_number的值（值为2）。接着前进一项把next_number 的值赋给current_number，再把current_number的值赋给old_number。重复这个过程，直到计算出最后一项，退出**while**循环。

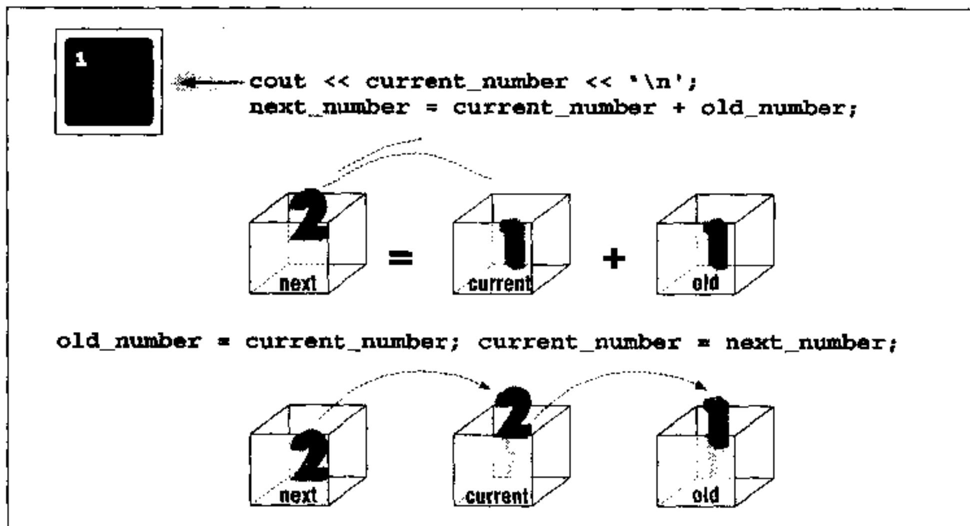


图6-1 斐波那契执行过程

这只是完成了循环体的一次操作。斐波那契数的前两项是1和1，再用这些值初始化前两项，放到一起将得到例6-1的程序

例6-1：fib/fib.c

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

```
int   old_number;      /* previous Fibonacci number */
int   current_number; /* current Fibonacci number */
int   next_number;     /* next number in the series */

int main()
{
    /* start things out */
    old_number = 1;
    current_number = 1;

    printf("1\n");      /* Print first number */

    while (current_number < 100) {

        printf("%d\n", current_number);
        next_number = current_number + old_number;

        old_number = current_number;
        current_number = next_number;
    }
    return (0);
}
```

break 语句

我们已经使用**while**语句计算了100以内的斐波那契数。当开始条件变为假时，退出循环，当然也可以在任何时刻使用 **break** 语句退出循环。

假设我们想累加一串数，但并不知道要加多少项，因此必须设法告诉程序所有的数是否已经加完。在例 6-2 中，我们使用数字零（0）作为结束的标记。

注意，**while** 语句是这样开始的：

```
while (1) {
```

该程序将永远循环下去，因为仅当表达式 1 为 0 时，才退出 **while** 循环，此时退出循环的唯一办法，就是通过 **break** 语句。

当遇到结束标志（0）时，使用语句：

```
if (item == 0)
      break;
```

退出循环。

例 6-2: total/total.c

```
#include <stdio.h>
char  line[100];/* line of data for input */
int   total;    /* Running total of all numbers so far */
int   item;     /* next item to add to the list */

int main()
{
    total = 0;
    while (1) {
        printf("Enter # to add \n");
        printf(" or 0 to stop:");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &item);

        if (item == 0)
            break;

        total += item;
        printf("Total: %d\n", total);
    }
    printf("Final total %d\n", total);
    return (0);
}
```

continue 语句

continue 语句很像 **break** 语句，不同的是，它不结束循环，而是跳到循环体的开头，再次执行循环体。例如，修改前面的程序，只累加大于0的数，程序如例 6-3 所示。

例 6-3: totalb/totalb.c

```
[File: totalb/totalb.c]
```

```
#include <stdio.h>
char  line[100];    /* line from input */
int   total;        /* Running total of all numbers so far */
int   item;         /* next item to add to the list */
int   minus_items;  /* number of negative items */

int main()
{
    total = 0;
    minus_items = 0;
    while (1) {
        printf("Enter # to add\n");
        printf(" or 0 to stop:");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &item);

        if (item == 0)
            break;

        if (item < 0) {
            ++minus_items;
            continue;
        }
        total += item;
        printf("Total: %d\n", total);
    }
    printf("Final total %d\n", total);
    printf("with %d negative items omitted\n",
               minus_items);
    return (0);
}
```

随处赋值的副作用

C几乎允许在所有的地方使用赋值语句。例如，可以把一个赋值语句放在另一个赋值语句中：

```
/* don't program like this */
average = total_value / (number_of_entries = last - first);
```

它等价于：

```
/* program like this */
number_of_entries = last - first;
average = total_value / number_of_entries;
```

第一种写法把number_of_entries的赋值藏在表达式中，虽然程序也能运行，但编程最重要的规则就是保持程序简单，应尽量避免隐藏任何成份。

C 也允许程序员把赋值语句放在 **while** 条件中。例如：

```
/* do not program like this */
while ((current_number = last_number + old_number) < 100)
    printf("Term %d\n", current_number);
```

读者应尽量避免编写这类程序。看看下面程序的逻辑是多么清晰：

```
/* program like this */
while (1) {
    current_number = last_number + old_number;
    if (current_number >= 100)
        break;
    printf("Term %d\n", current_number);
}
```

问题6-1：由于某种奇怪的原因，例6-4程序把所有人的结余都计作0，为什么？

例 6-4: owe0/owe0.c

```
#include <stdio.h>
char  line[80];          /* input line */
int   balance_owed;      /* amount owed */

int main()
{
    printf("Enter number of dollars owed:");
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &balance_owed);

    if (balance_owed = 0)
        printf("You owe nothing.\n");
    else
```

```
        printf("You owe %d dollars.\n", balance_owed);

    return (0);
}
```

执行程序时输出:

```
Enter number of dollars owed: 12
You owe 0 dollars.
```

答案

解答 6-1: 这个程序列出了最平常也最令人灰心丧气的 C 错误，问题就出在 C 允许把赋值语句放在 **if** 条件中。语句:

```
if (balance_owed = 0)
```

等价于:

```
balance_owed = 0;
if (balanced_owed != 0)
```

语句应该写成:

```
if (balance_owed == 0)
```

这个错误是初学程序的人最常犯的。

编程练习

练习 6-1: 编程计算平面内两点间距离。(更进一步的问题是，计算空间距离，这个问题涉及到使用标准函数 sqrt。使用帮助系统找出使用此函数的更多信息。)

练习 6-2: 一位教授使用表 6-3 建立评分等级。

表6-3 等级值

得分	等级
0-60	F
61-70	D
71-80	C
81-90	B
91-100	A

给出一个百分成绩，输出对应的评分等级。

注意：程序员常常要修改其他人写的程序。好的练习应该是把别人的程序拿来，比如例6-4的程序，并修改它。

练习6-3：修改前面的程序，根据百分成绩中最后一位数的大小，在评分等级后输出+或-。评分等级修改内容参照表6-4。

表6-4 等级修改值

最后一位	修改符
1-3	-
4-7	<空>
8-0	+

例如，81=B-, 94=A, 68=D+。注意：F只有F。没有F+或F-。

练习6-5：闰年是指年份能被4整除，不包含能被100整除但不能被400整除的年份。编写一个程序，判定某一年是否是闰年。

练习6-6：给定一名雇员的工作时间（小时）及每小时的工资数，编程计算周工资额。40小时以上的工资数，按超过原工资标准半倍的加班工资来统计。

第七章

程序设计过程

本章内容

- 设置
- 程序规范
- 代码设计
- 原型
- Makefile
- 测试
- 调试
- 维护
- 修改
- 代码分析
- 标记程序
- 使用调试器
- 用文本编辑器作浏览器
- 增加注释
- 编程练习

编程只是一个简单的问题。

——某从未写过程序的老板

编软件不仅仅是写代码，它有一个生命周期，从问世、成长、成熟到最后被新的产品所代替而消亡。图7-1解释了软件的生命周期。理解这个周期很重要，因为对于一个程序员而言，在写新代码上花的时间只是一小部分，而大部分编程时间则花在对已有代码的修改和调试上。软件并不存在于真空中，它一定要有文档、要维护、要增强、并且被销售。本章我们只着眼于小程序项目，而大的项目将在十八章“模块化程序设计”中讨论。虽然最终的代码还不到100行，但其中的原理可以适用于含有几千行代码的程序。

设计程序的主要步骤是：

- **需求**。当某人有个想法，并要求你来实现它的时候，程序就有了其诞生的根基。需求文档通常用很一般的术语来描述用户的要求。

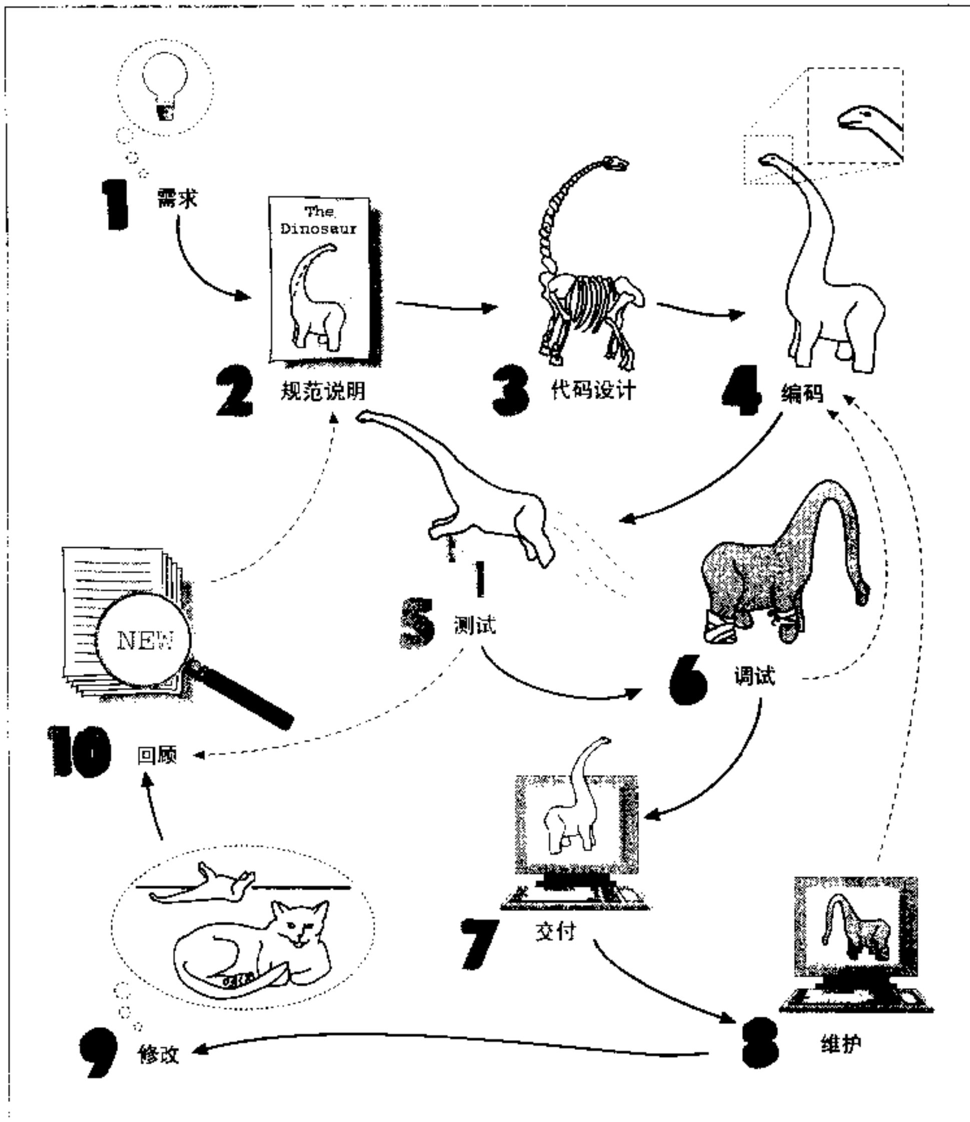


图7-1 软件生命周期

- **程序规范**。描述程序要做什么。开始，初步的程序规范叙述程序要做什么，以后，随着程序越来越精炼，规范说明也越来越详细。最后，当程序完成时，程序规范也就成为对程序的完整说明。
- **代码设计**。程序员负责程序的总体设计。设计工作应该包括主要的算法、模

决定义、文件格式以及数据结构。

- **编码**。接下来的步骤是写程序。先写一个原型框架，然后再填进代码，使之成为一个完整的程序。
- **测试**。程序应该拟定一个测试计划来测试程序。如果有可能，应该让其他人来测试。
- **调试**。不幸的是，很少有程序第一次就能运行，必须经过一再修改和测试
- **交付**。包装程序、编辑文档，问世使用。
- **维护**。程序从来不会十全十美。总会发生错误，并需要修改，这个步骤就是编程的维护阶段。
- **修改和升级**。程序工作一段时间后，用户可能想改变它，比如想要更多的功能和更智能的算法。此时将产生新的规范说明，程序设计过程就又开始了。

设置

操作系统允许把大量的文件放在同一目录中，与文件夹把纸夹在一起放在文件柜中一样，目录把文件放在一起（Windows 9x 更进一步，把目录叫做"文件夹"）。本章我们将编制一个简单的计算器程序，这个程序的所有文件都存储在名为*calc*的目录下。在 UNIX 中根目录下建立一个新目录，然后进入该目录，如下所示：

```
% cd ~
% mkdir calc
% cd ~/calc
```

在 MS-DOS 下，键入：

```
C:\> cd \
C:\> mkdir calc
C:\> cd \calc
C:\CALC>
```

建立目录非常简单。当你编制的程序越来越多时，你或许想有一个更为精致的目录结构，组织目录或文件夹的详细信息可以参见操作系统手册。

程序规范

本章我们假设接受的任务是"写一个模拟四功能计算器的程序"，显然给的任务不明确也不完整，程序员应把它提炼出来，以确切限制要编的程序。所以第一步是写用户初步说明文档，主要描述程序做什么以及如何使用。这个文档不包括程序的内部结构和用户打算使用的算法。四功能计算器程序的说明样本在下列标题为"Calc：一个四功能计算器"的方框中。

初步的规范说明有两个用途：第一、把它交给老板（或是顾客）以确认他们同意你在说明上所说的一切；第二、在同事中传阅，看他们是否有其他建议或改正方案。

这份初步规范说明传阅后得到下面的注释：

- 你准备如何退出程序？
- 当0作除数时该怎么办？

为此，加入一个新的运算符q，用来退出程序，并加入下列叙述语句：

"当0作除数时显示一条错误信息，同时结果寄存器中的数不变。

代码设计

改进了初步规范说明后，就可以着手设计代码了。代码的设计阶段应有计划。如果有很多人参与某大型编程项目，那么应将代码分成多个模块，分别交给各个程序员去编写。在这一阶段，要制定文件格式、设计数据结构、确定主要算法。

我们要编制的简单的计算器程序由于不使用文件，也不需要特殊的数据结构，因此这一阶段就只剩下设计主要算法这项工作。算法用伪代码来描述，所谓伪代码是介于自然语言和真正代码之间的一种简写形式，主要算法如下：

```
loop
    读一个运算符和一个数
```

```
    做计算
    显示结果
end-loop
```

Calc: 一个四功能计算器
初步规范说明

1989年12月10日

Steve Oualline

警告：这是一个初步规范说明，任何软件之间的任何相似之处纯属巧合。

Calc是一个允许用户把2000美元的计算机变成一个1.98美元计算器的程序。这个程序可以进行简单整数的四则运算。

运行程序时，它把结果寄存器清0，并显示它的内容，然后用户可以键入一个运算符和一个数，结果会被更新并显示出来。下面的运算符是有效的：

运算符	含义
+	加
-	减
*	乘
/	除

例如(用户输入的内容用黑体显示)：

```
calc
Result:  0
Enter operator and number:  + 123
Result:  123
Enter operator and number:  - 23
Result:  100
Enter operator and number:  / 25
Result:  4
Enter operator and number:  * 4
Result:  16
0
```

原型

代码设计完成后，就可以开始写程序了。但是，与其一次写一个完整的程序，然后再调试它，还不如使用一种称为快速原型（fast prototyping）的方法。该方法是先完成规范说明中最小的部分，然后在此基础上再作增补。本例中，可以先把四功能计算器简化为单功能计算器。小程序运行过了，再把其余的功能加上去。另外，原型还可以让老板先看到程序的样子。当你演示给他看时，他还可以给出进一步的指示。良好的交流是编制良好程序的关键，看你演示的人越多越好。四功能计算器的第一版代码见示例 7-1。

IV+IX=XIII?

有一位大学教师，有一次给他的学生布置了一道作业：“实现一个四功能的计算器”。一位学生注意到这个规范说明很不确切，就决定和教授开个玩笑。教授没有说必须使用什么样的数，所以这位学生写了一个程序，它只能计算罗马数字(IV+III=VII)。程序还有一份完整的用户手册——用拉丁文写的。

例 7-1: calc1/calc1.c

```
#include <stdio.h>
char  line[100];/* line of data from the input */
int   result;   /* the result of the calculations */
char  operator; /* operator the user specified */
int   value;    /* value specified after the operator */

int main()
{
    result = 0; /* initialize the result */

    /* Loop forever (or till we hit the break statement) */
    while (1) {
        printf("Result: %d\n", result);

        printf("Enter operator and number: ");
        fgets(line, sizeof(line), stdin);
        sscanf(line, "%c %d", &operator, &value);

        if (operator = '+') {
            result += value;
```

```
            } else {
                printf("Unknown operator %c\n", operator);
            }
        }
    }
```

开始时，程序把变量result初始化为0。程序的主体是一个循环，它是从下面这条语句开始的：

```
    while (1) {
```

遇到**break**语句时循环结束。代码如下：

```
    printf("Enter operator and number: ");
        fgets(line, sizeof(line), stdin);
        sscanf(line,"%c %d", &operator, &value);
```

要求用户输入一个运算符和一个数，输入的内容经过分析后存入变量operator和value中，接着开始检查运算符。如果运算符是加号，就执行加法操作：

```
    if (operator = '+' {
        result -= value;
```

现在，这个程序只认识加法运算符。一旦它运行通过，就可以再增加更多的if语句，从而为程序增加更多的运算符。

最后，如果输入的是非法的运算符，则代码：

```
    } else {
         printf("Unknown operator %c\n", operator);
    }
```

将输出错误信息，告诉用户输入错误。

Makefile

输入源代码后，就要编译并连接它。到目前为止，我们已经手工运行了编译器，这种编译方法很乏味，而且容易出错，同时，更大的程序，包括许多模块，手工编

译它们非常困难。所幸的是，UNIX和MS-DOS/Windows都有一个称为*make*（注1）的实用工具，它能处理所有的编译工作。现在，拿这个例子作示范，用calc代替程序名。第十八章将详细讨论*make*。程序查看名为*Makefile*的文件，了解编译用户程序的方式，并据此运行编译器。

因为Makefile包含编译规则，所以它将为编译器定制规则。下面是适合本书提到的所有编译器的一套*Makefile*。

Generic UNIX

```
File: calc1/makefile.unx
#-----------------------------------------#
#       Makefile for Unix systems         #
#    using a GNU C compiler               #
#-----------------------------------------#
CC=gcc
CFLAGS=-g
#
# Compiler flags:
#       -g      --Enable debugging

calc1: calc1.c
        $(CC) $(CFLAGS) -o calc1 calc1.c

clean:
        rm -f calc1
```

警告：实用工具make中，命令行

```
$(CC) $(CFLAGS) -o calc1 calc1.c
```

必须以制表符(tab)开头。八个空格无法运行，一个空格和一个制表符也不能运行。这一行必须以一个制表符开头。查看编辑器并确定你能说出一个制表符和一串空格之间的区别。

注1：　Microsoft公司的Visual C++称这种实用工具为nmake。

UNIX 下自由软件基金会开发的 gcc 编译器

```
File: calc1/makefile.gcc
#-----------------------------------------------#
#        Makefile for UNIX systems              #
#     using a GNU C compiler                    #
#-----------------------------------------------#
CC=gcc
CFLAGS=-g -D__USE_FIXED_PROTOTYPES__ -ansi
#
# Compiler flags:
#       -g      --Enable debugging
#       -Wall   --Turn on all warnings (not used since it gives away
#                       the bug in this program)
#       -D__USE_FIXED_PROTOTYPES__
#               --Force the compiler to use the correct headers
#       -ansi   --Don't use GNU extensions.  Stick to ANSI C.

calc1: calc1.c
        $(CC) $(CFLAGS) -o calc1 calc1.c

clean:
        rm -f calc1
```

Borland C++

```
[File: calc1/makefile.bcc]
#
# Makefile for Borland's Borland C++ compiler
#
CC=bcc
#
# Flags
#       -N  --Check for stack overflow
#       -v  --Enable debugging
#       -w  --Turn on all warnings
#       -ml --Large model
#
CFLAGS=-N -v -w -ml

calc1.exe: calc1.c
        $(CC) $(CFLAGS) -ecalc1 calc1.c
```

```
clean:
        erase calc1.exe
```

Turbo C++

```
File: calc1/makefile.tcc
#-----------------------------------------------#
#       Makefile for DOS systems                #
#    using a Turbo C compiler.                  #
#-----------------------------------------------#
CC=tcc
CFLAGS=-v -w -ml

calc1.exe: calc1.c
        $(CC) $(CFLAGS) -ecalc1.exe calc1.c

clean:
        del calc1.exe
```

Visual C++

```
[File: calc1/makefile.msc]
#-----------------------------------------------------#
#       Makefile for DOS systems                      #
#    using a Microsoft Visual C++ compiler.           #
#-----------------------------------------------------#
CC=cl
#
# Flags
#       AL --Compile for large model
#       Zi --Enable debugging
#       W1 --Turn on warnings
#
CFLAGS=/AL /Zi /W1

calc1.exe: calc1.c
        $(CC) $(CFLAGS) calc1.c

clean:
        erase calc1.exe
```

要编译这个程序，只需执行make命令，它会决定需用哪一个编译命令来执行它们。make命令根据文件的最后修改日期来判断是否需要编译它，编译将产生一个目标文件，目标文件的修改日期应晚于其源文件的修改日期。如果源文件被编辑过，那么它的修改日期将更新，目标文件就过时了。make检查这些日期，如果源文件在目标文件生成之后被修改过，那么make就重新编译目标文件。

测试

程序编译通过后，就可以进入测试阶段，现在要着手写测试计划文档，这个文档仅仅是列出所要执行的步骤以确认程序能够运行。写这个计划文档有两个原因：

- 如果发现错误，可以重新生成它。
- 如果改动过程序，则要重新测试它，以保证新的代码不会打乱原来可以运行的程序。

测试计划为：

进行下列操作：

```
+ 123   结果应为 123
+ 52    结果应为 175
x 37    应该显示错误信息
```

运行这个程序，得到：

```
Result: 0
Enter operator and number: + 123
Result: 123
Enter operator and number: + 52
Result: 175
Enter operator and number: x 37
Result: 212
```

有个地方显然错了。输入x 37后应该得到一条出错信息，但是没有。程序中存在一处错误，所以我们可以开始进入调试阶段。先做一个小的原型程序的优点之一就是可以尽早地找出错误。

调试

首先检查程序，看看是否能发现错误。在这样一个小程序中，找错很容易。但假设不是一个21行的程序，而是一个大得多的5000行的程序，这样一个程序会使查错异常困难，所以需要进入下一步骤

多数系统都有C调试程序，而各个系统又有所不同，另外有些系统没有调试器。本例中要使用一条诊断显示语句，这个作法很简单：在确定数据是正确的地方（要确保它真是对的）放一条printf语句，然后在数据不好的地方再放一条printf语句。运行程序，反复放printf语句，直到找出错误所在。程序中加入诊断printf语句后如下：

```
printf("Enter operator and number: ");
fgets(line, sizeof(line), stdin);
sscanf("%d %c", &value, &operator);
printf("## after scanf %c\n", operator);
if (operator = '+') {
    printf("## after if %c\n", operator);
    result += value;
```

注意：每个printf语句开头的##表示该行是暂时的调试行。调试完成后，##会使相关语句易于识别并去掉。

再次运行程序，结果如下：

```
Result: 0
Enter operator and number: [_F_]+ 123[_F_]
Result: 123
Enter operator and number: [_F_]+ 52[_F_]
## after scanf +
## after if +
Result: 175
Enter operator and number: [_F_]x 37[_F_]
## after scanf x
## after if +
Result: 212
```

从这里可以看出if语句出了错，但是不知出错的原因，运算符变量输入的是x，输出的却是+，仔细检查后发现犯的是老毛病，用=代替了==。改正这个错误后，程序就运行正确了。在此基础上，增加处理其他运算符-、*、/的代码，结果如例7-2所示:

例 7-2: calc2/calc2.c

```
#include <stdio.h>
char  line[100];/* line of text from input */

int   result;   /* the result of the calculations */
char  operator; /* operator the user specified */
int   value;    /* value specified after the operator */

int main()
{
    result = 0; /* initialize the result */

    /* loop forever (or until break reached) */
    while (1) {
        printf("Result: %d\n", result);
        printf("Enter operator and number: ");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%c %d", &operator, &value);

        if ((operator == 'q') || (operator == 'Q'))
            break;

        if (operator == '+') {
            result += value;
        } else if (operator == '-') {
            result -= value;
        } else if (operator == '*') {
            result *= value;
        } else if (operator == '/') {
            if (value == 0) {
                printf("Error:Divide by zero\n");
                printf("   operation ignored\n");
            } else
                result /= value;
        } else {
```

```
                printf("Unknown operator %c\n", operator);
            }
        }
        return (0);
    }
```

扩展包括新运算符的测试计划，并再次测试：

```
+ 123   结果应该是 123
+ 52    结果应该是 175
x 37    应该输出错误信息
- 175   结果应该是 零
+ 10    结果应该是 10
/ 5     结果应该是 2
* 8     结果应该是 16
q       程序退出
```

令人吃惊的是，测试这个程序会发现它居然能运行。从规范说明中删去“初步”一词，程序、测试计划和规范说明就可以交付了。

维护

好的程序员在让程序面世之前要经过长期严格的测试过程。但当第一个用户几乎立刻就发现一个错误时，维护阶段就开始了。修正错误、测试程序（确定修正没有造成任何破坏），然后再次交付程序。

修改

虽然形式上程序写完了，实际上工作还没有结束。程序使用几个月后，会有人找到你问：“你能不能加一个取模运算符？”，此时就得修正规范说明、修改程序、更新测试计划、测试程序直到再次交付程序。

又过了一段时间，更多的人找到你，要求修改程序。很快，程序具有了三角计算功能、线性回归功能、统计功能、二进制算术功能以及财务计算功能。我们的设计是基于单字符运算符的，不久，你会发现字符不够用了。此时，程序所能处理

的事情已远远多于你最初的设计。迟早你会觉得必须废弃这个程序，并写一个新程序取代它。这时，又开始写新的初步规范说明，再次开始编程过程。

代码分析

代码分析是一门艺术，它研究已有代码，从中找出令人惊讶的东西（比如这段代码如何以及为什么会执行）。

不幸的是，多数程序员并不是从设计这一步开始。他们急不可耐地进入维护和修正阶段并必须面对可能是最糟糕的工作：理解并修改别人的代码。

计算机能有效地帮你研究别人的代码的真正含义。检测和格式化代码时有许多工具可用，包括：

- **交叉引用**。这些程序都有如xref、cxref 和cross之类的名称，Unix的V系统有一个实用工具cscope。交叉引用能打印变量表，并指出变量在什么地方使用。
- **程序缩排器**。像cb和indent这样的程序可以正确地缩排程序（正确缩排由工具使用者限定）。
- **漂亮的打印程序**。像vgind和cprint这样的打印程序可以对源代码进行排版，以便在激光打印机上输出。
- **调用图**。在Unix的V系统上程序cflow可用来分析程序结构。其他系统有一个公共领域实用工具calls，可产生调用图，并能显示调用者及被调用者。

究竟使用哪种工具？当然是使用适合你的那一种。不同的程序运行方式不同。下面是一些检查代码的方法，你可以从中选择一种来使用

注释程序

打印一份程序，并作注释。用红笔或蓝笔写，以区分打印内容和注释内容。用重点符号突出重要段落，这些注释非常有用；把它们放入程序中，重新打印一份，重复本步骤。

使用调试器

调试器是一种十分强大的工具，它可以使程序员了解程序是如何运行的。多数调试器允许单步执行程序，即一次执行一行，并检查变量，从而明白程序真正的运行方式。一旦你找出代码的真正用意，就记下来，把它们作为注释加到程序中。

用文本编辑器浏览

从头至尾研究别人的程序时，文本编辑器是最好的工具之一。假定你想找出变量sc的用途，使用查找命令找到sc使用的第一个地方，再次查找找到第二次使用的地方，一直找下去直到明白变量的含义为止。

假设找出sc是作为计数器使用的。因为已经处在编辑器中，所以很容易进行全局查找和替换，把sc 换成sequence_counter。（严正警告：在进行替换之前要确定sequence_counter还未被使用，此外，当心不需要的替换发生，比如把“escape”里的sc也作了改动。）注释所作的定义并按照自己的方式编写具有较强可理解性的程序。

增加注释

不要害怕把你得到的信息加到注释中，哪怕这些信息微不足道。我曾经使用过的信息包括：

```
int state;  /* Controls some sort of state machine */
int rmxy;   /* Something to do with color correction ? */
```

还有一种引人注意的注释:

```
int idn;    /* ??? */
```

意思是"我还没想好这个变量做什么用"。虽然还不清楚变量的用途，但现在标上可以表明此处需要做更多的工作。

在向别人的代码中加入注释并改进编程风格的过程中，程序结构变得越来越清晰；通过插入注释，也使得以后的程序员更容易理解这个程序。

例如，假设面对这样一个由推崇"越简洁越好"设计风格的人编写的程序，任务就是找出这个程序的用途。首先，用铅笔加注释，如下图 7-2:

```
#include <iostream.h>
#include <stdlib.h>
int   g, l, h, c, n;
char  line[80];
main()
{
    while (1) {
        /*Not Really*/
        g = rand() % 100 + 1;
        l = 0;
        h = 100;
        c = 0;
        while (1) {
            cout << "Bounds " << l << " - " << h << '\n';
            cout << "Value[" << c << "]? ";
            ++c;
            cin >> n;
            if (n == g)
                break;
            if (n < g)
                l = n;
            else
                h = n;
        }
        cout << "Bingo\n";
    }
    return (0);
}
```


图 7-2　一个简短程序

这个密码程序需要一些修改，通读之后并应用这一节所阐述的原则，就得到了一个注释充分、易于理解的程序，见例 7-3。

例 7-3: good/good.c

```
/********************************************************
 * guess   A simple guessing game.                      *
 *                                                      *
 * Usage:                                               *
 *      guess                                           *
 *                                                      *
 *      A random number is chosen between 1 and 100.    *
 *      The player is given a set of bounds and         *
 *      must choose a number between them.              *
 *      If the player chooses the correct number, he wins. *
 *      Otherwise, the bounds are adjusted to reflect   *
 *      the player's guess and the game continues.      *
 *                                                      *
 *                                                      *
 * Restrictions:                                        *
 *      The random number is generated by the statement *
 *      rand() % 100.  Because rand() returns a number  *
 *      0 <= rand() <= maxint  this slightly favors     *
 *      the lower numbers.                              *
 ********************************************************/
#include <stdio.h>
#include <stdlib.h>
int   number_to_guess;  /* random number to be guessed */
int   low_limit;        /* current lower limit of player's range */
int   high_limit;       /* current upper limit of player's range */
int   guess_count;      /* number of times player guessed */
int   player_number;    /* number gotten from the player */
char  line[80];         /* input buffer for a single line */
int main()
{
    while (1) {
        /*
         * Not a pure random number, see restrictions
         */
        number_to_guess = rand() % 100 + 1;
        /* Initialize variables for loop */
        low_limit = 0;
        high_limit = 100;
        guess_count = 0;
        while (1) {
            /* tell user what the bounds are and get his guess */
            printf("Bounds %d - %d\n", low_limit, high_limit);
```

```
            printf("Value[%d]? ", guess_count);
            ++guess_count;
            fgets(line, sizeof(line), stdin);
            sscanf(line, "%d", &player_number);
            /* did he guess right? */
            if (player_number == number_to_guess)
                break;
            /* adjust bounds for next guess */
            if (player_number < number_to_guess)
                low_limit = player_number;
            else
                high_limit = player_number;
        }
        printf("Bingo\n");
    }
}
```

编程练习

对于每一道练习，请按照软件生命周期，完成从规范说明到交付使用的各个步骤。

练习 7-1: 编程把英制单位转换成公制单位（如，英里变成公里，加仑变成升等等）。要包括规范说明和代码设计。

练习 7-2: 编程计算天数，如1990年6月6日到1992年4月3日有多少天。包括规范说明和代码设计。

练习 7-3: 连续行传输每秒钟可以传送960个字符。编写一个程序，计算传送一个给定大小的文件需要多长时间。用一个400MB（419,430,440字节）的文件来测试这个程序。使用合适的单位。（一个400MB的文件所花费的时间要按天来计算。）

练习 7-4: 编程，对一个给定金额，加上8%的销售税，并对分位进行四舍五入。

练习 7-5: 编程，判定一个数是否是素数。

练习 7-6: 编程，统计一串数中正数和负数的个数。

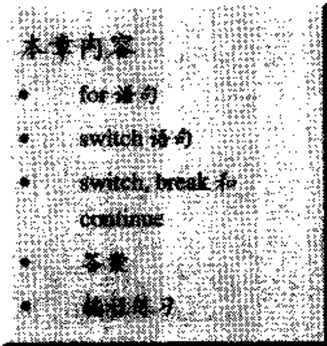
第二部分

简单程序设计

本部分将进一步描述简单的C编程。我们将学习其余的控制语句和一些更高级的操作，如位运算。最后，介绍一些更复杂的编程任务如I/O文件和调试。

- 第八章“更多的控制语句”，描述控制语句的其余内容，包括 **for**,**break** 和 **continue**，**switch** 语句的详细讨论。
- 第九章“变量作用域和函数”，介绍局部变量，函数和参数。
- 第十章“C预处理器”，描述C预处理器给程序员写代码提供了很大的自由度。本章也给程序员讲解了容易造成混乱的各种情况，并对在预处理器中避免这些情况的简单规则进行了描述。
- 第十一章“位运算”，讨论以位为基础的逻辑C运算符。
- 第十二章“高级类型”，解释结构和其他高级类型，还包括 **sizeof** 运算符和枚举类型。
- 第十三章“简单指针”，介绍C指针变量并列出了它们的用法。
- 第十四章“文件输入/输出”，描述缓冲和非缓冲的输入/输出。讨论和对于二进制文件的ASCII码，以及怎样生成一个简单文件。

- 第十五章“调试和优化”，描述怎样调试一个程序以及怎样使用一个交互调试器。本章不仅列出了怎样调试程序，也介绍了怎样写一个易于调试的程序，同时也描述了许多优化技巧，这些技巧能使你的程序运行更快和更有效。
- 第十六章“浮点数”，使用简单十进制浮点数的格式来介绍浮点数问题，如超界错误、精度损失、上溢和下溢。



第八章

更多的控制语句

语法，懂得怎样控制
甚至国王…
莫里埃（法国剧作家）

for语句

for语句允许程序员指定一段代码的重复执行次数。**for**语句的一般形式是：

```
for (初始语句; 条件; 重复语句)
    语句体
```

它等价于：

```
初始语句;
while (条件) {
    语句体;
    重复语句;
}
```

例如，例8-1使用**while**循环累加5个数。

例8-1：total5w/totalw.c

```
#include <stdio.h>

int total;       /* total of all the numbers */
int current;     /* current value from the user */
int counter;     /* while loop counter */
```

```
char line[80];  /* Line from keyboard */

int main() {
    total = 0;
    counter = 0;
    while (counter < 5) {
        printf("Number? ");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &current);
        total += current;

        ++counter;
    }
    printf("The grand total is %d\n", total);
    return (0);
}
```

该语句还可以用 **for** 语句重写，见例 8-2：

例 8-2：total5f/total5f.c

```
#include <stdio.h>

int total;       /* total of all the numbers */
int current;     /* current value from the user */
int counter;     /* for loop counter */

char line[80];  /* Input from keyboard */

int main() {
    total = 0;
    for (counter = 0; counter < 5; ++counter) {
        printf("Number? ");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &current);
        total += current;
    }
    printf("The grand total is %d\n", total);
    return (0);
}
```

注意counter从0变到4。一般情况下，计数会从1到5；但在C中应改变观念，从0开始计数，共有5项：0，1，2，3，4。（从1开始累加是数组溢出错误的主要原因。见第五章“数组、修饰符与读取数据”。）

仔细研究这两种写法，会发现它们很类似，详见图8-1。

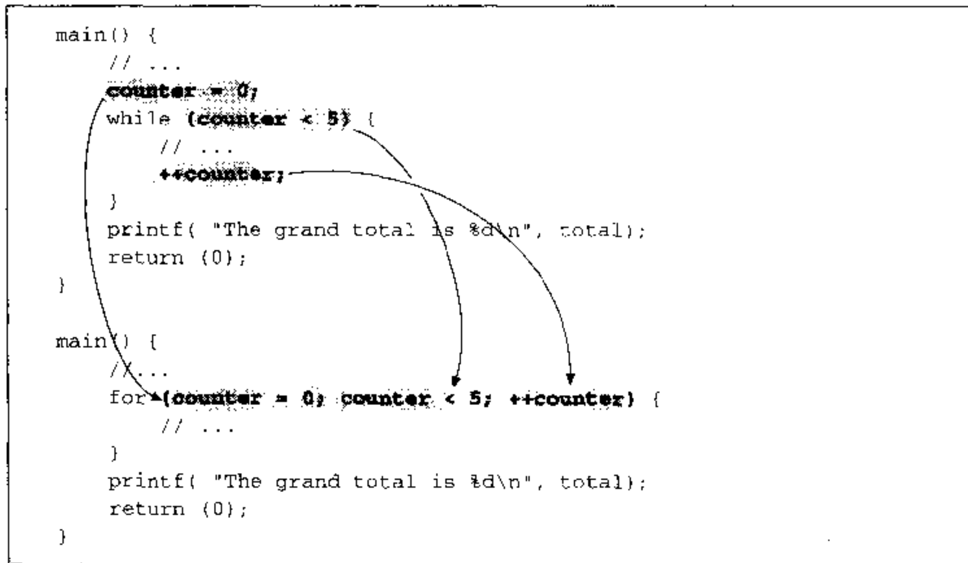


图8-1 “while”和“for”之间的相似性

许多程序设计语言不允许改变循环控制变量的值（本例中是counter）。C不严格要求这一点，你可以随时改变控制变量的值——也可以转入或跳出循环，去做那些PASCAL和FORTRAN程序员不敢做的事情（虽然C给了你做这些事情的自由，但并不意味着你非要去做这些事情）。

问题8-1：当例8-3运行后显示：

```
Celsius:101 Fahrenheit:213
```

就没有别的了。为什么？

例8-3: cent/cent.c

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

```
/*
 * This program produces a Celsius to Fahrenheit conversion
 *     chart for the numbers 0 to 100.
 */

/* The current Celsius temperature we are working with */
int celsius;
int main() {
    for (celsius = 0; celsius <= 100; ++celsius);
        printf("Celsius:%d Fahrenheit:%d\n",
            celsius, (celsius * 9) / 5 + 32);
    return (0);
}
```

问题8-2： 例8-4读取了一组5个数，并统计其中出现3和7的个数。为什么它给出了错误的答案？

例8-4: seven/seven.c

```
#include <stdio.h>
char line[100];        /* line of input */
int seven_count;       /* number of 7s in the data */
int data[5];           /* the data to count 3 and 7 in */
int three_count;       /* the number of 3s in the data */
int index;             /* index into the data */

int main() {

    seven_count = 0;
    three_count = 0;
    printf("Enter 5 numbers\n");
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d %d %d %d %d",
        &data[1], &data[2], &data[3],
        &data[4], &data[5]);

    for (index = 1; index <= 5; ++index) {

        if (data[index] == 3)
            ++three_count;

        if (data[index] == 7)
            ++seven_count;
```

```
    }
printf("Threes %d Sevens %d\n",
            three_count, seven_count);
    return (0);
}
```

当用数据 3 7 3 0 2 运行这个程序时，结果是:

```
Threes 4 Sevens 1
```

（你的答案可能多种多样。）

switch 语句

switch 语句类似于一串 **if-else** 语句，它的一般格式为:

```
switch (表达式) {
  case 常量1:
    语句
    ...
    break;

  case 常量2:
    语句
    ...
    /*Fall through*/

  default:
    语句
    ...
    break;

  case 常量3:
    语句
    ...
    break;
}
```

switch 语句计算表达式的值，然后转入 **case** 标号之一。标号是不允许重复的，所以只能选中一个 **case** 标号。表达式的结果必须是整数、字符或枚举量。

case的标号顺序不限，但必须是常量。default 标号可以放在switch语句中的任何地方，任何两个case标号都不能有相同的值。

当C遇到switch语句时，它先求表达式的值，然后查找匹配的case标号。如果没有找到，就使用default 标号。如果没有找到default，该语句就什么都不做。

注意： switch语句类似于PASCAL中的case语句，主要的区别是，PASCAL中只允许在标号后放一个语句，C允许写许多条。C会一直执行下去，直到遇到break语句为止。在PASCAL中，不能从一个case中一直执行到另一个case中。但在C中却可以。

C的switch和PASCAL的case语句另外一个主要区别是：PASCAL要求default语句（即否则语句）放在末尾，C允许default语句在switch语句的任何地方。

例8-5包含一系列if和else语句：

例 8-5: if和else语法

```
if (operator == '+') {
    result += value;
} else if (operator == '-') {
    result -= value;
} else if (operator == '*') {
    result *= value;
} else if (operator == '/') {
    if (value == 0) {
        printf("Error:Divide by zero\n");
        printf("   operation ignored\n");
    } else
        result /= value;
    } else {
        printf("Unknown operator %c\n", operator);
    }
```

这段代码很容易改写成switch语句，在switch中，对每一个操作都使用不同的case语句，default子句将处理所有的非法操作。

使用switch语句重写此程序不仅使程序简单，而且易读。修改后的calc程序见例8-6。

例 8-6: calc3/calc3.c

```
#include <stdio.h>
char  line[100];    /* line of text from input */

int   result;       /* the result of the calculations */
char  operator;     /* operator the user specified */
int   value;        /* value specified after the operator */
int main()
{
    result = 0;     /* initialize the result */

    /* loop forever (or until break reached) */
    while (1) {
        printf("Result: %d\n", result);
        printf("Enter operator and number: ");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%c %d", &operator, &value);

        if ((operator == 'q') || (operator == 'Q'))
            break;
        switch (operator) {
        case '+':
            result += value;
            break;
        case '-':
            result -= value;
            break;
        case '*':
            result *= value;
            break;
        case '/':
            if (value == 0) {
                printf("Error:Divide by zero\n");
                printf("   operation ignored\n");
            } else
                result /= value;
            break;
        default:
            printf("Unknown operator %c\n", operator);
            break;
        }
    }
```

```
    return (0);
}
```

switch 中的**break**语句告诉计算机继续执行**switch**后的语句。如果没有**break**语句，将继续执行 **switch** 内的下一条语句。

例如：

```
control = 0;
/* a not so good example of programming */
switch (control) {
        case 0:
                printf("Reset\n");
        case 1:
                printf("Initializing\n");
                break;
        case 2:
                printf("Working\n");
}
```

本例中，当control==0时，程序将显示出：

```
Reset
Initializing
```

case 0不是以一条break语句结尾。在显示Reset之后，程序转到下一条语句（case=1）并显示Initializing。

这样的语法存在一个问题：无法确定是程序原本被设计为从case0执行到case1，还是程序员忘记写 **break** 语句了。为了消除这个疑虑，**case** 部分最好还是写上 **break** 语句，或者写上注释 /*fall through*/，如下例所示：

```
/* a better example of programming */
switch (control) {
        case 0:
                printf("Reset\n");
                /* Fall through */
        case 1:
                printf("Initializing\n");
                break;
```

```
        case 2:
                printf("Working\n");
}
```

因为case 2 在最后，所以不需要**break**语句。**break** 语句总是使程序跳到**switch**的结尾，但现在已经执行到那里了。

如果我们稍微改动一下这个程序，在 **switch** 中增加另一条 **case** 语句：

```
/* We have a little problem */
switch (control) {
        case 0:
                printf("Reset\n");
                /* Fall through */
        case 1:
                printf("Initializing\n");
                break;
        case 2:
                printf("Working\n");
        case 3:
                printf("Closing down\n");
}
```

此时，如果 control == 2，程序将输出：

```
Working
Closing down
```

这个结果很令人讨厌。问题出在case 2 不再是最后一条语句了，我们继续执行了。(我们并不想这样，或者说，我们应该加上一条/*Fall through*/注释。)**break** 语句现在是必需的。如果总是写上 **break** 语句，我们就没有必要担心是不是真的需要它了。

```
/* Almost there */
switch (control) {
        case 0:
                printf("Reset\n");
                /* Fall through */
        case 1:
                printf("Initializing\n");
```

```
                break;
        case 2:
                printf("Working\n");
                break;
    }
```

最后，我们再提这样一个问题：当control==5时，会出现什么情况？本例中，因为没有匹配的**case**或是**default**子句，所以跳过整个的**switch**语句。

本例中，程序员没有包含**default**语句，因为control不会是0、1、2以外的值。但是，变量可以被赋以很奇怪的值，所以需要给程序加上一些保护措施，如下所示：

```
/* The final version */
switch (control) {
    case 0:
        printf("Reset\n");
        /* Fall through */
    case 1:
        printf("Initializing\n");
        break;
    case 2:
        printf("Working\n");
        break;
    default:
        printf(
            "Internal error, control value (%d) impossible\n",
                control);
        break;
}
```

即使**default**语句并不是必需的，也应该在每个**switch**语句中写上一条。哪怕**default**语句仅仅像下面这样：

```
default:
        /* Do nothing */
        break;
```

所以还是有这个语句才好，这种方法起码表明你想忽略掉范围以外的所有数据。

switch, break 和 continue

break 语句有两种用法。一种是用在 **switch** 语句中，它使程序跳到 **switch** 语句的最后；另一种是用在 **for** 或是 **while** 循环中，它使程序跳出循环。**continue** 语句只用在循环中，它使程序跳到循环的开头。图 8-2 显示了同一个 **switch** 语句中的 **break** 和 **continue**。

图 8-2 中的程序用来把不同进制的整数转换成其他进制。如果想知道八进制数的值输入 o（即 octal- 八进制）和数字。命令 q 用来退出程序。例如:

```
Enter conversion and number: o 45
Result is 45
Enter conversion and number: q
```

help 命令很特别，因为我们不想在 help 命令后输出结果，毕竟 help 的结果是几行文本，而不是数字。所以，在 help case 后没有放 **break** 语句，而是放了一条 **continue** 语句。**continue** 强迫程序跳到循环的开头。在 **switch** 内部，**continue** 语句作用于循环，而 **break** 语句作用于 **switch**。

switch 外面还有一个 **break** 语句用来让用户退出程序。这个程序的控制流程见图 8-2。

答案

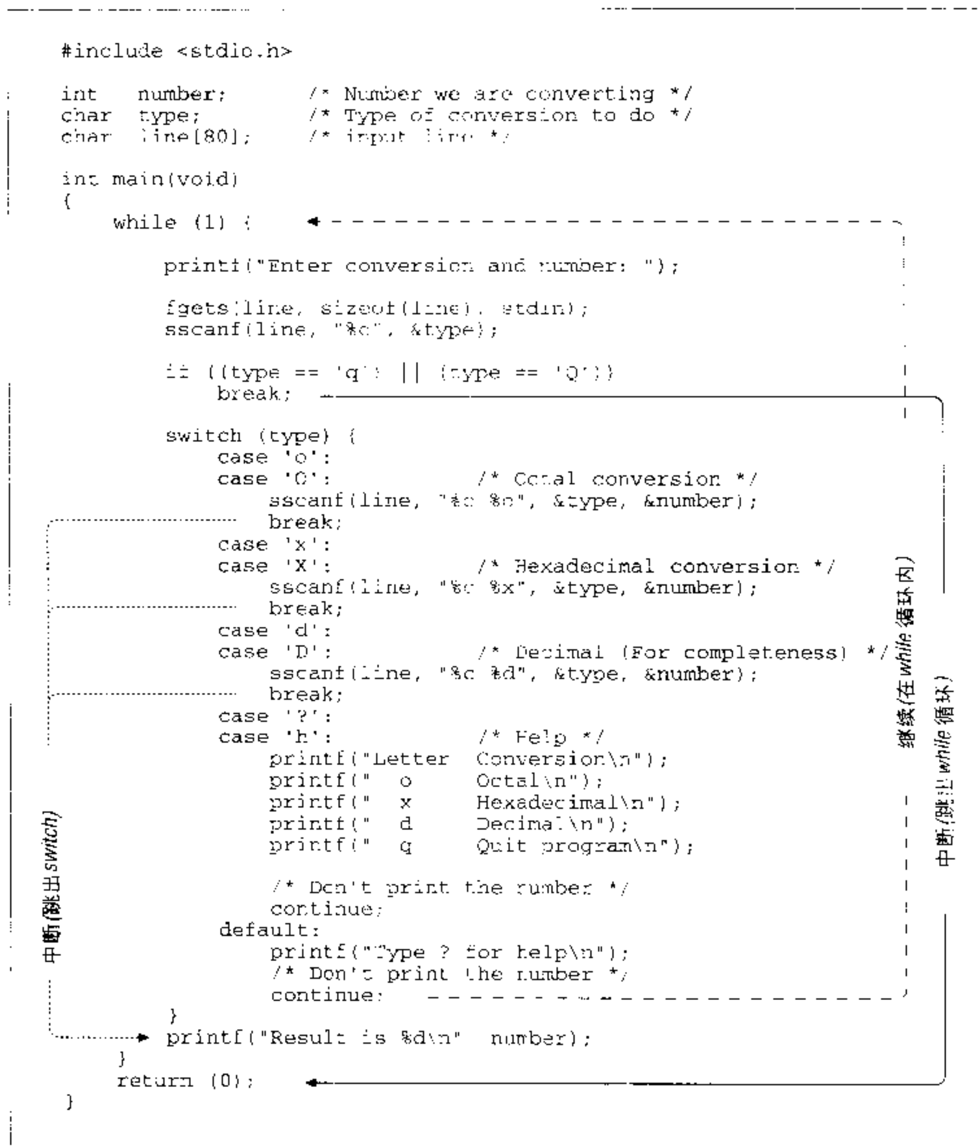
解答 8-1：问题出在 **for** 语句的结尾有一个分号。For 的语句体是在右圆括号与分号之间。本例中，语句体为空。虽然 printf 语句是缩进的，但它也不是 **for** 语句的一部分。缩进容易引起误解，C 编译器不看它是否缩进。在表达式：

```
celsius <= 100
```

等于假（celsius= =101）之前，程序什么也不能做。然后执行 printf 语句。

解答 8-2：问题出在我们把数读到 data[1] 到 data[5] 中。在 C 中，合法的数组范围是 0 到数组大小 -1，或者在本例中即从 0 到 4。data[5] 是非法的。使用

非法的数组元素时，会出现令人不可思议的怪现象：在本例中，变量 three_count 的值被改变了。解决的方法是，只使用 data[0] 到 data[4]。



```
#include <stdio.h>

int   number;        /* Number we are converting */
char  type;          /* Type of conversion to do */
char  line[80];      /* input line */

int main(void)
{
    while (1) {
        printf("Enter conversion and number: ");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%c", &type);

        if ((type == 'q') || (type == 'Q'))
            break;

        switch (type) {
            case 'o':
            case 'O':            /* Octal conversion */
                sscanf(line, "%c %o", &type, &number);
                break;
            case 'x':
            case 'X':            /* Hexadecimal conversion */
                sscanf(line, "%c %x", &type, &number);
                break;
            case 'd':
            case 'D':            /* Decimal (For completeness) */
                sscanf(line, "%c %d", &type, &number);
                break;
            case '?':
            case 'h':            /* Help */
                printf("Letter  Conversion\n");
                printf("   o    Octal\n");
                printf("   x    Hexadecimal\n");
                printf("   d    Decimal\n");
                printf("   q    Quit program\n");

                /* Don't print the number */
                continue;
            default:
                printf("Type ? for help\n");
                /* Don't print the number */
                continue;
        }
        printf("Result is %d\n", number);
    }
    return (0);
}
```


图 8-2 switch/continue

所以，需要改变读取的 sscanf 行：

```
sscanf(line, "%d %d %d %d %d",
    &data[0], &data[1], &data[2], &data[3], &data[4]);
```

同时，**for** 循环也必须从：

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

改成：

```
for (index = 0; index < 5; ++index)
```

注意： 有经验的C程序员一看到这个for循环的行，就能马上感觉到有什么地方不对劲。线索是：1)for循环从1开始，2)循环中出现了<=运算符。大多数C的for循环都从0开始，而且使用<终止。

编程练习

练习8-1：显示一张棋盘（8 × 8）。每个方块为5个字符 × 3个字符。下面是2 × 2棋盘的示例：

```
+-----+-----+
|     |     |
|     |     |
|     |     |
+-----+-----+
|     |     |
|     |     |
|     |     |
+-----+-----+
```

练习8-2：n个并联电阻的总电阻计算公式为：

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \dots + \frac{1}{R_n}$$

假定我们的电路由两个并联电阻组成，这两个电阻的阻值分别为400Ω和200Ω，计算公式为：

$$\frac{1}{R}=\frac{1}{R_1}+\frac{1}{R_2}$$

代入值后，得到：

$$\frac{1}{R}=\frac{1}{400}+\frac{1}{200}$$

$$\frac{1}{R}=\frac{3}{400}$$

$$R=\frac{400}{3}$$

所以这个电路的总电阻为133.3Ω。

编写一个程序计算任意个电阻并联后的总电阻。

练习 8-3：编程计算n个数的平均数。

练习 8-4：编程显示乘法表。

练习 8-5：编程读入一个字符，判定它是元音字母还是辅音字母，并输出结果。

练习 8-6：编程把数字转换为英文。例如：895转换为“eight nine five”。

练习 8-7：数字85读音为“eighty-five”，而不是“eight five”。修改练习8-6中程序使得在输出结果时，不仅仅输出对应的数，还要输出它的发音，例如，输入13时，输出“thirteen”；输入100时，输出“one hundred”。

本章内容

- 作用域和类
- 函数
- 无参数的函数
- 结构化程序设计
- 递归
- 答案
- 编程练习

第九章

变量作用域和函数

可是大概推测起来，

这恐怕预兆着情况将有一番非常的变故。

——莎士比亚[哈姆雷特，第1幕，第1场]

到目前为止，我们所使用的只是全局变量，在本章中，我们将了解其他种类的变量，并学习如何使用它们，同时还将介绍怎样把代码分成函数。

作用域和类

所有的变量都有两个属性：作用域和类。变量作用域是指变量在程序中的有效区域。一个全局变量在任何地方都有效（故称全局），所以它的作用域是整个程序。局部变量的作用域局限在定义它的块内，在这个块外就不能访问它。块是包含在括号（{}）内的一段代码。图 9-1 说明了局部和全局变量之间的区别。

局部变量可以定义成与全局变量同名。一般情况下，变量 count（首次定义时）的作用域将是整个程序。第二个定义，即局部变量 count，在定义它的块内要优先于全局变量。在这个块内，全局变量被隐藏起来，也可以用嵌套的局部定义来隐藏局部变量。图 9-2 列出了一个隐藏变量。

变量 count 作为局部变量和全局变量都被定义了。一般来说，全局 count 的作用域是整个程序；然而当块内发生变量定义时，在块内的变量就变得较为活跃，全局 count 在块的区域内被隐藏为局部变量 count。图中加阴影部分表示全局 count 被隐藏的区域。

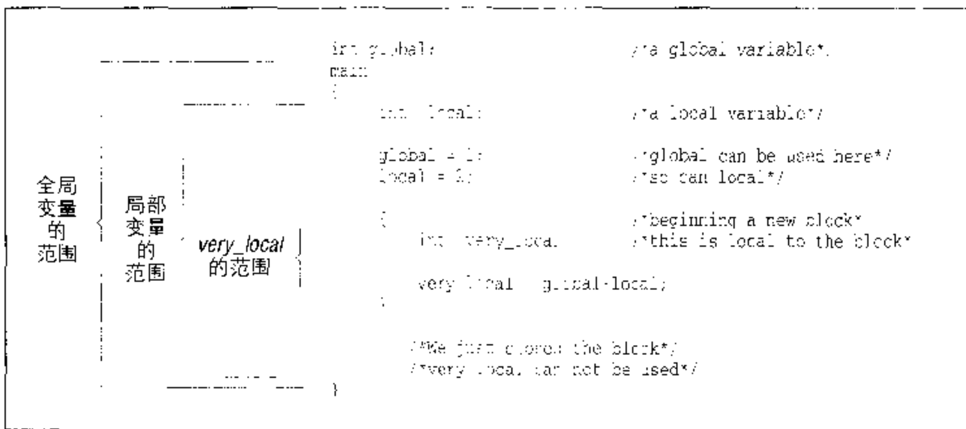


图 9-1　局部和全局变量

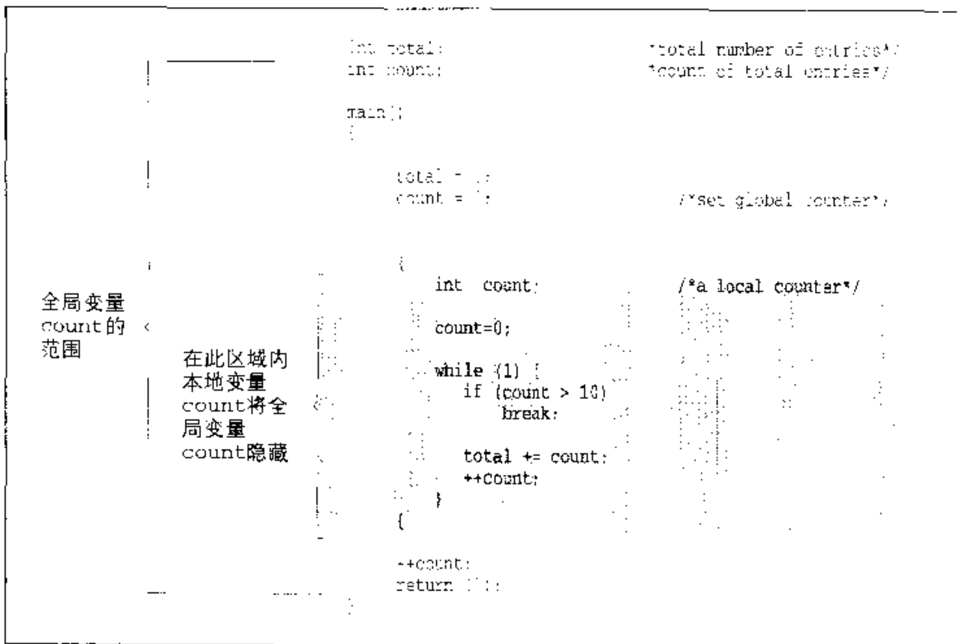


图 9-2　隐藏变量

输入下列语句就会出现问题:

```
count = 1;
```

很难判定所指的是哪个 count，是 main 开头定义的那个 count 还是 **while** 循环中的那个count？最好是给这些变量起不同的名字，如total_ count，current_ count 以及 item_ count。

变量的类既可以是永久性的，也可以是暂时性的。全局变量总是永久性的，它们在程序运行之前被建立并赋与初值，并一直保留到程序结束。在块开始时定义的临时变量从被称为堆栈的一段内存中分配空间，如果定义了太多的临时变量，会导致“堆栈上溢”错误。在块结束时，临时变量使用的空间将释放给堆栈。每次进入这个块时，临时变量都将被初始化。

堆栈的大小取决于所使用的系统和编译器。在大多数UNIX 系统上，程序会自动地尽最大可能地分配堆栈空间。其他系统中缺省的堆栈大小可以由编译器设置来改变。在 MS-DOS/Windows 系统中，堆栈空间不能大于65,536字节，这看上去是个很大的空间，但几个大数组很快就会把它用掉，所以你应该考虑把所有的大数组定义成永久性的。

局部变量如果没定义成 static 的话都将是临时性的。

注意： 当 static 用在全局变量时，它会有完全不同的意义。它表示对于当前文件是局部变量。见第十八章“模块化程序设计”。

例9-1说明了永久变量和临时变量的区别。选用的是含义鲜明的名字：temporary 是临时变量，permanent 是永久变量。在每次建立temporary 变量时（在for 语句块开始）都赋一次初值，而permanent 在程序启动时只初始化一次。

在循环中，两个变量都进行加1操作，但到循环开头时，temporary被初始化为1，如例9-1所示。

例 9-1: vars/vars.c

```
#include <stdio.h>

int main() {
    int counter;     /* loop counter */
```

```
    for (counter = 0; counter < 3; ++counter) {
        int temporary = 1;          /* A temporary variable */
        static int permanent = 1; /* A permanent variable */
        printf("Temporary %d Permanent %d\n",
            temporary, permanent);
        ++temporary;
        ++permanent;
    }
    return (0);
}
```

此程序的输出是:

```
Temporary 1 Permanent 1
Temporary 1 Permanent 2
Temporary 1 Permanent 3
```

注意：临时变量有时又称作自动变量，因为编译器自动为它们分配存储空间。修饰词auto可用来表示一个临时变量；但在实际中，它几乎从来不用。

表9-1描述了定义变量的不同方式。

表9-1 变量定义

定义	作用域	存储类	初始化
在所有的块外	全局	永久	一次性
在所有块外并用static	全局（注1）	永久	一次性
在一个块内	局部	临时	每次进入块时
在一个块内并用static	局部	永久	一次性

注1： 块外进行的static定义表明变量对于定义它的文件来说是局部的。（详见第十八章关于多文件编程的内容。）

函数

使用函数可以把常用的代码组织为一个紧凑的可重复使用的单元。我们已经见过一个函数main，它是一个在程序开始时被调用的特殊函数，其他所有的函数都直接或间接地从main中调用。

假定想写一个计算三个三角形面积的程序，可以写三次计算公式，也可以建立一个函数来做这项工作。每个函数都应该包含下列内容的注释块：

名字

函数名

描述

对函数功能的描述

参数

对函数中每个参数的描述

返回值

对函数返回值的描述信息

此外，还可以加入如文件格式、引用或注释等内容。其他内容参见第三章“风格”。

计算三角形面积的函数的开头为：

```
/********************************************
 * triangle --Computes area of a triangle.   *
 *                                           *
 * Parameters                                *
 *   width --Width of the triangle.          *
 *   height --Height of the triangle.        *
 *                                           *
 * Returns                                   *
 *   area of the triangle.                   *
 ********************************************/
```

函数本身的开始行是：

```
float triangle(float width, float height)
```

函数是 **float** 型的，两个参数是 width 和 height，它们也是 **float** 型的。

C采用的参数传递形式是"由值调用（call by value）"调用triangle时代码为：

```
triangle(1.3, 8.3);
```

C把参数值（本例是1.3和8.3）复制到函数的参量（width和height）中，然后开始执行函数代码。这种形式的参数传递，函数不能把数据传回给使用参数的调用器。（注2）

注意： C并不要求函数类型，如果不定义函数类型，系统会默认为整型。但是，如果你不写函数类型，就会弄不清是想让函数使用缺省的int型，还是仅仅忘了定义函数类型。为避免这种混乱，记住，总要定义函数类型而不要使用缺省值。

函数使用下列语句计算面积：

```
area = width * height / 2.0;
```

剩下的问题就是把结果传给调用器。使用 **return** 语句可以完成这项工作：

```
return (area);
```

完整的三角函数程序见例9-2。

例 9-2: tri-sub/tri-sub.c

```
#include <stdio.h>
/********************************************
 * triangle -- Computes area of a triangle. *
 *                                          *
 * Parameters                               *
 *   width -- Width of the triangle.        *
 *   height -- Height of the triangle.      *
 *                                          *
 * Returns                                  *
```

注2：　严格上讲，该指令不为真，我们可以通过使用指针来使C把信息回传。我们将在第十三章中详细介绍。

```
 *   area of the triangle.                    *
 ********************************************/
float triangle(float width, float height)
{
    float area;      /* Area of the triangle */

    area = width * height / 2.0;
    return (area);
}
```

下行:

```
size = triangle(1.3, 8.3);
```

调用 triangle 函数, C 把 1.3 分配给 width、把 8.3 分配给 height。

如果函数是楼里的房间，那么参量就是房间之间的门，本例中数值1.3进了门牌为width的房间。参量的门是单向的，东西能进来，但出不去，**return** 语句就是让数据从函数出来的工具。在triangle例中，函数把局部变量area赋值为5.4，然后执行 return (area); 语句。

这个函数的返回值是 5.4，所以语句:

```
size = triangle (1.3, 8.3)
```

给 size 赋值为 5.4。

例 9-3 计算了三个三角形的面积。

例 9-3: tri-prog/tri-prog.c

```
[File: tri-sub/tri-prog.c]

#include <stdio.h>

/********************************************
 * triangle --Computes area of a triangle.  *
 *                                          *
 * Parameters                               *
 *   width --Width of the triangle.         *
```

```
 *   height --Height of the triangle.        *
 *                                           *
 * Returns                                   *
 *   area of the triangle.                   *
 *********************************************/
float triangle(float width, float height)
{
    float area;      /* Area of the triangle */

    area = width * height / 2.0;
    return (area);
}

int main()
{
    printf("Triangle #1 %f\n", triangle(1.3, 8.3));
    printf("Triangle #2 %f\n", triangle(4.8, 9.8));
    printf("Triangle #3 %f\n", triangle(1.2, 2.0));
    return (0);
}
```

函数也需要像变量一样进行定义，定义可以把函数的信息告诉编译器。对于函数 `triangle` 可使用下列定义：

```
/* Compute a triangle */
float triangle (float width, float height);
```

这个定义被称作函数原型。

定义函数原型时可以不要求变量名，所以可以这样简单地写原型：

```
float triangle(float, float);
```

不过经常使用的是较长的写法，因为这样能多给程序员一些信息，且使用编辑器的剪切和粘贴功能创建原型较容易。

严格地讲，对某些函数来说原型只是可选项。如果没定义原型，C 编译器就会假定函数返回一个整数并可以携带任何数目的参量。漏用原型会去掉C编译器用来检查函数调用的有用信息。大多数编译器都有一个编译时间开关，可以警告程序员调用了哪些没有原型的函数。

无参数的函数

一个函数可以有任何数目的参数，也可以没有。但即使是使用无参变量，也应有括号：

```
value = next_index();
```

定义无参函数的原型比较棘手，不能使用下列语句：

```
int next_index();
```

原因是C编译器会根据空括号假定这是一个K&R型函数定义。该函数详见第十九章“原始编译器”。关键词**void**用来表示空的参数列表，所以next_index函数的原型是：

```
int next_index(void);
```

void也可表示不返回值的函数（Void和FORTRAN子程序或PASCAL程序类似）。如下例中函数仅显示结果，不返回值：

```
void print_answer(int answer)
{
    if (answer < 0) {
        printf("Answer corrupt\n");
        return;
    }
    printf("The answer is %d\n", answer);
}
```

问题9-1：例9-4应该计算串长度（注3），可是其算得所有串的长度却均为0，为什么？

例 9-4: len/len.c

```
/********************************************************
 * Question:                                            *
 *      Why does this program always report the length  *
 *      of any string as 0?                             *
 *                                                      *
```

注3：　该函数同库函数strlen具有相同的功能。

```
 * A sample "main" has been provided.  It will ask      *
 * for a string and then print the length.              *
 ********************************************************/
#include <stdio.h>

/********************************************************
 * length -- Computes the length of a string.           *
 *                                                      *
 * Parameters                                           *
 *      string -- The string whose length we want.      *
 *                                                      *
 * Returns                                              *
 *      the length of the string.                       *
 ********************************************************/
int  length(char string[])
{
    int            index;      /* index into the string */

    /*
     * Loop until we reach the end of string character
     */
    for (index = 0; string[index] != '\0'; ++index)
        /* do nothing */
    return (index);
}
int main()
{
     char line[100];     /* Input line from user */

     while (1) {
        printf("Enter line:");
        fgets(line, sizeof(line), stdin);

        printf("Length (including newline) is: %d\n", length(line));
    }
}
```

结构化程序设计

计算机学家花费许多年的时间和精力研究如何编程,因为他们提出了许多确实不错的程序设计方法——几乎每个月有一种新方法。这些方法包括:流程图、自顶

向下的程序设计、自底向上的程序设计、结构化的程序设计和面向对象的程序设计（OOD）。

前面已经介绍了函数，现在谈谈怎样使用结构化程序设计技术来设计程序。该技术是把一个程序分割或结构化为一些短小精炼的函数，这些函数使程序易于书写和理解。但并不是说这就是编程的最佳方法，但它恰恰却是最适合我的方法。如果其他的方法适合于你，那就采用其他方法吧。

程序设计的第一步是决定要干什么，这一部分内容已在第七章“程序设计过程”中作了介绍。接下去就是决定怎样使数据结构化。

最后是开始编码阶段。写论文时，最先会在纸上用一两个句子写出每一部分的提纲，以后再加入细节，写程序也是同样的道理。从写提纲开始，并把提纲变成main函数，细节可以放在其他函数中，见例9-5。

例 9-5: Solve the World's Problems

```
int main()
{
    init();
    solve_problems();
    finish_up();
    return (0);
}
```

当然，一些细节可随后加进来。

从写主函数开始。主函数应该少于三页，如果太长了，可以考虑把它分成两个更简短的函数。完成主函数后，可以开始编写其他函数。

这种类型的结构化程序设计叫自顶向下编程。从顶（main）开始，向下开展工作。

另一种设计方法是自底向上编程，这种方法的第一步是写最底层的函数，测试之后，把这些函数组合在一起。当我编写一个以前没有用过的新标准函数时，我试着使用了这种方法。我写了一个小函数以确认我确实知道该函数如何运行，然后在此基础上继续展开。这就是第七章在编写计算器程序时使用的方法。

可见在实际工作中这两种方法都有用。多数情况下采用自顶向下的方法，部分情况下采用自底向上的方法。计算机学家对此有一个说法：混用。编程时应遵从的一条原则是："使用最奏效的那种"。

递归

递归就是函数直接或间接地调用它自己。有些程序设计函数本身很适于递归算法，比如阶乘。

递归函数必须遵从两条原则：

- 它必须有结束点。
- 必须使问题变得简单。

阶乘的定义为：

```
fact(0) = 1

fact(n) = n * fact(n-1)
```

在C中的定义是：

```
int fact(int number)
{
    if (number == 0)
        return (1);
    /* else */
    return (number * fact(number-1));
}
```

这个定义满足两条原则，首先，它有确定的结束点（当number==0）；第二，它简化了问题，因为计算fact（number-1）比计算fact（number）简单。

递归只在number>=0时有效，但如果试着计算fact（-3）会发生什么呢?程序会因堆栈上溢或类似的原因而异常中断。fact（-3）调用fact（-4），fact（-4）又调用fact（-5），依此类推，没有结束点。这就是无穷递归错误。

许多需要重复做的事情都可以用递归实现，比如累加一个数组的所有元素，可以定义一个函数，累加数组中下标从 m 到 n 的各元素：

- 如果只有一个元素，累加很容易实现。
- 否则，把第一个元素和其余元素的累加和加起来。

在C中，这个函数是：

```
int sum(int first, int last, int array[])
{
    if (first == last)
        return (array[first]);
    /* else */
    return (array[first] + sum(first+1, last, array));
}
```

例如：

```
Sum(1 8 3 2) =
    1 + Sum(8 3 2) =
        8 + Sum(3 2) =
            3 + Sum (2) =
                2
            3 + 2 = 5
        8 + 5 = 13
    1 + 13 = 14
Answer = 14
```

答案

解答 9-1：程序员在解释这个什么也不做（除了index加1）的**for**循环时会遇到很多麻烦。因为在**for**的末尾没有分号。C一直读取直到看到一个语句（本例中是return(index)语句)），然后把它放到**for**循环中。改进后如例9-6所示。

例 9-6: len2/len2.c

```
#include <stdio.h>
int  length(char string[])
```

```
{
    int            index;       /* index into the string */
    /*
     * Loop until we reach the end-of-string character
     */
    for (index = 0; string[index] != '\0'; ++index)
        continue; /* do nothing */
    return (index);
}
int main()
{
     char line[100];     /* Input line from user */
     while (1) {
        printf("Enter line:");
        fgets(line, sizeof(line), stdin);
        printf("Length (including newline) is: %d\n", length(line));
    }
}
```

编程练习

练习 9-1：写一个程序，统计一个串中单词的个数。（你的文档中应准确描述单词的定义方式。）编写一个程序，测试这个新过程。

练习 9-2：编写一个函数 begins (string1,string2)，如果 string1 的开始是 string2 的话，返回值为真。编程测试这个函数

练习 9-3：编写一个函数 count (number,array,length)，用它来统计 number 在 array 中出现的次数。Array 是一个有 length 个元素的数组。函数应该采用递归方法。编制测试程序测试此函数。

练习 9-4：编写一个程序，它读入一个字符数组，并把数组中每个字符的值累加起来作为该数组的杂凑码（hash code），返回该值。

练习 9-5：编写一个程序，返回数组中的最大值

练习 9-6：编写一个函数，扫描一个字符数组，把其中的“-”替换为“_”。

本章内容

- #define 语句
- 条件编译
- 包含文件
- 带参数的宏
- 高级特性
- 小结
- 答案
- 编程练习

第十章
C 预处理器

人的语言就像绣花织锦，因为喜欢并且想看到图案就不得不展开它，但当织锦被卷起来的时候，图案就被隐瞒和歪曲了。

——地米斯托克利斯（古希腊政治家

最初，当C还处在开发阶段的时候，就已经有迹象表明它需要处理命名的常量、宏和引用文件。解决的办法是建立一个预处理器，在程序传递到C编译器之前，先由预处理器来识别。预处理器其实就是一个专用的文本编辑器，它的语法和C语言完全不同，对C指令也无法理解。

预处理器非常有用，没多久它就被应用于主要的C编译器中。在一些系统如UNIX中，预处理器仍然是一个独立的程序，由编译器包cc自动执行。一些新编译器如Turbo C++和Microsoft Visual C++都有内置的预处理器。

#define 语句

例10-1初始化了两个数组（data和twice），每个数组包含10个元素，假定想改变程序以使用20个元素，就必须改变数组的大小（两处）和下标范围（一处）。多处变动除了要做许多工作以外，还会导致出错。

例 10-1: init2a/init2a.c

```
int data[10];     /* some data */
int twice[10];    /* twice some data */
int main()
{
```

```
    int index;    /* index into the data */

    for (index = 0; index < 10; ++index) {
        data[index] = index;
        twice[index] = index * 2;
    }
    return (0);
}
```

写一个普通程序把数组的大小定义成常量，然后让C调整两个数组的维数。使用#define语句就能做到这一点。例10-2是例10-1的新版本。

例 10-2: init2b/init2b.c

```
#define SIZE 20      /* work on 20 elements */

int data[SIZE];      /* some data */
int twice[SIZE];     /* twice some data */

int main()
{
    int index;    /* index into the data */

    for (index = 0; index < SIZE; ++index) {
        data[index] = index;
        twice[index] = index * 2;
    }
    return (0);
}
```

`#define size 20` 一行作为发给一个特殊文本编辑器的指令，做全局改变把`size`改为20，这一行去掉了改变大小带来的苦差事和无端的臆测。

所有预处理器命令在第一列都以一个#开头，C是自由格式，但预处理器不是，它取决于第一列的#号。预处理器和C没有关系，它可以被（也正被）用来编辑C以外的其他程序。

注意： 你很容易忘掉预处理器和C编译器使用的是不同的语法，初级程序员最常见的错误之一就是想在预处理器指令中使用C指令。

预处理器指令在行末就结束了，这个格式和C语言不同，C语言用一个分号来结束一个语句。把分号放在预处理器指令末尾会导致意外的结果，而在其行尾加斜杠（\）则表示一行尚未结束。

预处理器最简单的用法是定义替代宏，例如，命令：

```
#define FOO bar
```

该命令会使预处理器在每个“FOO”出现的地方用“bar”替换。使用全部为大写的宏名称是常用的编程方法，这个方法可以很容易分辨变量（全部小写）和宏（全部大写）。

一个简单 define 语句的一般形式是：

```
#define name substitute-text
```

name 处可以是任何有效的C标识符，*substitute-text* 则可以是任何内容。下列定义：

```
#define FOR_ALL for(i = 0; i < ARRAY_SIZE; i++)
```

可以这样使用：

```
/*
 * Clear the array
 */
FOR_ALL {
    data[i] = 0;
}
```

但这种定义宏的方法并不好，这样的定义混淆了基本的程序控制流。本例中，如果程序员想知道循环做什么，他就必须到程序开头查找 FOR_ALL 的定义。

更糟的用法是定义大量替换基本C程序指令的宏，例如，作如下定义：

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

```
    if (index == 0)
    BEGIN
        printf("Starting\n");
    END
```

问题是你不再是用C语言编程，而是用一种半C/半Pascal的混合语言，你可以在Bourne命令解释程序中找到相似的情况，在该程序中使用预处理器指令定义一种酷似Algol-68的语言。

这里是一段代码样本:

```
IF (x GREATER_THAN 37) OR (Y LESS_THAN 83) THEN
    CASE value OF
        SELECT 1:
            start();
        SELECT 3:
            backspace();
        OTHERWISE:
            error();
    ESAC
FI
```

多数程序员遇到这个程序都先是没有好评价，然后就用编辑器把源文件变成合乎常理的C语言版本。

由于预处理器不检查C语法的正确性，可能会导致出人意料的问题，例10-3在11行产生了一个错误。

例 10-3: big/big.c

```
 1 #define BIG_NUMBER 10 ** 10
 2
 3 main()
 4 {
 5     /* index for our calculations */
 6     int   index;
 7
 8     index = 0;
 9
10     /* syntax error on next line */
11     while (index < BIG_NUMBER) {
```

```
12          index = index * 8;
13      }
14      return (0);
15 }
```

问题出在第1行的#define语句，但错误信息却指向第11行。原因是第1行定义使预处理器把第11行扩展成：

```
while (index < 10 ** 10)
```

因为**是非法操作符，这种扩展产生了语法错误。

问题 10-1：例10-4的结果是47而不是设想的144，为什么？（见下文提示）

例 10-4: first/first.c

```
#include <stdio.h>

#define FIRST_PART      7
#define LAST_PART       5
#define ALL_PARTS       FIRST_PART + LAST_PART

int main() {
    printf("The square of all the parts is %d\n",
        ALL_PARTS * ALL_PARTS);
    return (0);
}
```

提示：不一定会显示答案。所幸C允许通过预处理器运行程序并查看结果，UNIX命令：

```
% cc -E prog.c
```

会把预处理器的结果变成标准输出。

在MS-DOS/Windows中，下列命令有同样效果：

```
C:> cpp prog.c
```

通过预处理器运行此程序会有：

```
# 1 "first.c"
# 1 "/usr/include/stdio.h" 1


... 包含文件<stdio.h>的数据列表

# 2 "first.c" 2

main() {
    printf("The square of all the parts is %d\n",
        7 + 5 * 7 + 5);
    return (0);
}
```

问题 10-2：例 10-5 产生一条警告信息：`counter` 在设置前已被使用了，这有点令人吃惊，因为 **for** 循环应该设置了 counter 的值。你还会得到一条非常奇怪的警告："null effect（没有结果）"，指向第 11 行。

例 10-5: max/max.c

```
 1 /* warning, spacing is VERY important */
 2
 3 #include <stdio.h>
 4
 5 #define MAX =10
 6
 7 int main()
 8 {
 9     int  counter;
10
11     for (counter =MAX; counter > 0; --counter)
12         printf("Hi there\n");
13
14     return (0);
15 }
```

提示：看预处理器输出结果。

问题 10-3：例 10-6 的 `size` 值计算错误。为什么？

例 10-6: size/size.c

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

```
#define SIZE    10;
#define FUDGE   SIZE -2;
int main()
{
    int size;/* size to really use */

    size = FUDGE;
    printf("Size is %d\n", size);
    return (0);
}
```

问题10-4：例10-7预想的是在接收到错误数据时显示“Fatal Error: Abort（致命错误：异常中断）”并退出。但当输入正确数据时，它还是退出。为什么？

例10-7：die/die.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #define DIE \
 5   fprintf(stderr, "Fatal Error:Abort\n");exit(8);
 6
 7 int main() {
 8     /* a random value for testing */
 9     int value;
10
11     value = 1;
12     if (value < 0)
13         DIE;
14
15     printf("We did not die\n");
16     return (0);
17 }
```

#define 和 const

const关键字相对较新。在**const**出现之前，**#define**是定义变量的唯一关键字，所以多数老一点的代码都使用**#define**指令。然而**const**比**#define**受欢迎有几个原因，首先C会立即检查**const**语句的语法，而**#define**指令则直到使用宏时才检查；**const**使用C语法，**#define**有自己的语法；最后，**const**遵循一般的C作用域规则，而由**#define**指令定义的常量永远都可以用。

所以多数情况下，**const**语句用得比**#define**语句多。下面是用两种方法定义同一个变量：

```
#define MAX 10 /* Define a value using the preprocessor */
               /* (This definition can easily cause problems) */

const int MAX = 10; /* Define a C constant integer */
                    /* (safer) */
```

注意： 一些编译器不允许用常量定义数组大小，这些编译器应该，但还没有跟上标准化的潮流。

#define指令只能定义简单常量，**const**语句则可以定义差不多所有类型的C常量，包括结构类（structure classes）。例如：

```
struct box {
    int width, height;    /* Dimensions of the box in pixels */
};
const box pink_box ={1.0, 4.5};/* Size of a pink box to be used for
input */
```

不过对于条件编译和其他特殊用途方面，**#define**还是必要的。

条件编译

有一个问题是程序员写的程序要在不同机器上运行，理论上，C代码是可移植的；但在实际中，不同的操作系统都有需要考虑的问题。比如，本书涉及了MS-DOS/Windows编译器和UNIX C，虽然它们几乎相同，但还是存在着差异，特别是当需要接触操作系统更高级的特性时。

规范把语言的某些特性留给了使用者，这会导致另一方面的移植问题，例如整数大小就取决于实现过程。

预处理器通过使用条件编译，使程序员修改已有代码具有很大的灵活性。假定想在调试阶段加入调试代码，而在最终提交的产品中删除调试代码，可以在#ifdef/#endif部分中加入下列代码：

```
#ifdef DEBUG
    printf("In compute_hash, value %d hash %d\n", value, hash);
#endif /* DEBUG */
```

注意： /*DEBUG*/不一定要放在#endif后面，不过作为注释它是很有用的。

如果程序开头包括指令：

```
#define DEBUG       /* Turn debugging on */
```

程序中就会包含printf语句，如果程序包括指令：

```
#undef DEBUG        /* Turn debugging on */
```

printf语句就会被略过。

严格来讲，#undef DEBUG不是必需的，如果没有#define DEBUG，DEBUG自然不用定义。#undef DEBUG语句明确地表示出：DEBUG用于条件编译，并且现在处于关闭状态。

如果符号没有定义，指令 **#ifdef** 会编译随后的代码：

```
#ifndef DEBUG
    printf("Production code, no debugging enabled\n");
#endif /* DEBUG */
```

#else 指令把条件变反，例如：

```
#ifdef DEBUG
    printf("Test version. Debugging is on\n");
#else DEBUG
    printf("Production version\n");
#endif /* DEBUG */
```

程序员或许会临时删去一些代码，常用的方法是把这段代码用/**/框住。这种方法会产生问题，如下例所示：

```
1:  /***** Comment out this section
2:      section_report();
3:      /* Handle the end of section stuff */
4:      dump_table();
5:  **** end of commented out section */
```

这个程序在第五行产生了语法错误，为什么?

更好的方法是使用 **#ifdef** 指令去掉下列代码:

```
#ifdef UNDEF
    section_report();
    /* Handle the end of section stuff */
    dump_table();
#endif /* UNDEF */
```

(当然，如果有人定义了 UNDEF 也可以包含上列代码，但一般不会这么巧)

编译器开关 *-Dsymbol* 允许在命令行定义这个符号，例如命令:

```
% cc -DDEBUG -g -o prog prog.c
```

编译程序 prog.c，并引用 #ifdef DEBUG 和 #endif DEBUG 之间的所有代码，即使程序中没有 #define DEBUG。

该选项的一般形式是 *-Dsymbol* 或 *-Dsymbol=value*。例如下面的语句给MAX 赋值为 10:

```
% cc -DMAX=10 -o prog prog.c
```

注意: 程序员可以忽略程序中含指令的命令行选项。例如指令:

```
#undef DEBUG
```

会导致不论使用或没有使用 -DDEBUG 都会定义 DEBUG。

多数 C 编译器自动定义某些与系统有关的符号。例如，Turbo C++ 定义了符号 __TURBOC__ 和 __MSDOS__。ANSI 标准编译器定义了 __STDC__。多数 UNIX

编译器定义了一个用于系统的名字（如SUN、VAX、Celerity等等），但是它们很少用来编辑文档，符号 __UNIX__ 为所有的UNIX机器使用。

包含文件

#include指令可以让程序使用其他文件中的源代码。例如，在程序中可以使用如下命令：

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

这条指令告诉预处理器把文件*stdio.h*（standard I/O—标准输入/输出）插入程序中。包含在其他程序中的文件叫头文件（大部分**#include**命令都放在程序开头），括号（<>）表明该文件是一个标准的头文件。在UNIX中，这些文件放在*/usr/include*目录下；在MS-DOS/Windows系统中，这些文件放在安装编译器时指定的目录下。

标准包含文件定义了数据结构和库函数使用的宏，如printf是一个在标准输出设备上显示数据的库函数，printf使用的FILE结构和相关库函数都在*stdio.h*中定义。

有时程序员可能想写自己的包含文件，当一个程序涉及几个文件时，局部包含文件对存储常量和数据结构非常有用，尤其是当一组程序员工作于一个项目，需要传递信息时（见第十八章“模块化程序设计”）。

局部包含文件定义方法是在文件名两边用双引号（“”），例如：

```
#include "defs.h"
```

*def.h*是任何有效的文件名，这种定义可以是一个简单文件*defs.h*、一个相对路径*../../data.h*，或者是一个绝对路径*/root/include/const.h*。（在MS-DOS/Windows上，可以用斜杠（\）代替斜杠（/）作为目录分隔符。）

注意： 在#include指令中应避免使用绝对路径名，原因是它们使程序很难移植。

包含文件有可能嵌套，也可能会因这一特性产生问题。假定在*const.h*文件中定义了几个有用的常量，在文件*data.h*和*io.h*中都包括文件*const.h*，程序中加入下列指令：

```
#include "data.h"
#include "io.h"
```

这将产生错误，因为预处理器将设置两次*const.h*中的定义。一个常量定义了两次并不是致命的错误，但是，两次定义一种数据结构或是逻辑与，就是个应该避免的致命错误。

解决这个问题的一种方法是检查*const.h*，看它是否已经被包含，并且不要定义已定义过的任何符号。如果符号没定义，指令**#ifndef**符号就是真；这条指令和**#ifdef**相反。

看下列代码：

```
#ifndef _CONST_H_INCLUDED
/* define constants */
#define _CONST_H_INCLUDED_
#endif  /* _CONST_H_INCLUDED_ */
```

当包含*const.h*时，它定义了符号_CONST_H_INCLUDED_。如果这个符号已经定义（因为前面已引用过这个文件），**#ifdef**条件将把其他的定义隐藏起来，就不会发生问题了。

注意：头文件的内容包罗万象，不仅包括定义和类型，还可以包括代码、初始化数据和昨天午餐的菜谱。不过良好的编程练习应把头文件仅限制在类型和定义中。

带参数的宏

到目前为止，我们仅讨论了简单的**#define**和宏。宏可以带有参数，下面的宏可用来计算一个数的平方：

```
#define SQR(x)  ((x) * (x))        /* Square a number */
```

注意：宏名（SQR）和括号之间不能有空格。

使用时，宏使用下列自变量文本取代 x：

```
SQR(5) expands to   ((5) * (5))
```

宏的参数两侧总要放一对括号（）、否则有可能导致例 10-8 所示错误：

例 10-8: sqr/sqr.c

```
#include <stdio.h>
#define SQR(x) (x * x)
int main()
{
    int counter;    /* counter for loop */
    for (counter = 0; counter < 5; ++counter) {
        printf("X %d, x squared %d\n",
            counter+1, SQR(counter+1));
    }
    return (0);
}
```

问题 10-5：例 10-8 的输出结果是什么？在你的机器上试着运行它，为什么会输出这样的信息？检查预处理器的输出。

简化程序的编程规则让我们只有在增量（++）和减量（--）运算符单独占一行时，才可以使用它们。在宏参量中使用这两个运算符时，会导致意外的结果，如例 10-9 所示。

例 10-9: sqr-i/sqr-i.c

```
#include <stdio.h>
#define SQR(x) ((x) * (x))

int main()
{
    int counter;    /* counter for loop */
```

```
    counter = 0;
    while (counter < 5)
        printf("X %d square %d\n", counter, SQR(++counter));
    return (0);
}
```

问题 10-6: 为什么例 10-9 没有产生预期的结果?每次 counter 加的是几?

问题10-7: 例10-10显示有一个变量number没有定义,而唯一的变量就是counter。

例 10-10: rrec/rec.c

```
#include <stdio.h>
#define RECIPROCAL (number) (1.0 / (number))

int main()
{
    float   counter;    /* Counter for our table */

    for (counter = 0.0; counter < 10.0;
         counter += 1.0) {

        printf("1/%f = %f\n",
                counter, RECIPROCAL(counter));
    }
    return (0);
}
```

高级特征

本书不涉及 C 预处理器指令的完整列表。更高级的特性包括用于条件编译的 **#if** 命令的高级形式,和用于在文件中插入依赖于编译器的指令的**#pragma**命令。这些特征的详细信息请参见 C 参考手册。

小结

C预处理器是C语言中一个非常有用的部分,它所表现的形式与C完全不同,必须与C编译器区别对待。

宏定义中出现的问题常常不表现在定义宏的地方，而往往在程序内部出错，遵循下列简单规则，可以减少问题的发生：

- 在每一项的两侧都放一对括号，特别是在 **#define** 中的常量和宏参数两侧。
- 当用多条语句定义一个宏时，要把代码放在括号（{}）内。
- 预处理器不是 C 编译器，不要使用 = 或 ;。

如果做到了这些，就不会再出现大的问题。

答案

解答 10-1：程序通过预处理器运行后，printf 扩展为：

```
printf("The square of all the parts is %d\n",
      7 + 5 * 7 + 5);
```

等式7+5*7+5等于47。在宏中表达式的所有项两侧放括号。如果把ALL_PARTS的定义改为：

```
#define ALL_PARTS (FIRST_PART + LAST_PART)
```

程序将正确执行。

解答 10-2：预处理器是一个思维简单的程序，当它定义宏时，标识符后的所有内容都被作为宏的一部分。本例中，MAX 的定义是 =10，当扩展 **for** 语句时，变为：

```
for (counter==10; counter > 0; --counter)
```

C 允许计算一个结果，然后把它忽略掉（在一些编译器里会产生无结果的警告信息）。本例中，程序检查 counter 的值是不是 10 后就会忽略掉答案。改正错误的方法是去掉定义中的 = 。

解答 10-3：与前面的问题一样，预处理器并不使用C语法约定。本例中，程序员用一个分号来结束语句，但预处理器把它作为size定义的一部分，对size赋值语句扩展为：

```
size = 10; -2;;
```

结尾的两个分号不会有不良影响，但中间的一个分号却是“杀手”。这行代码告诉C做两件事：

- 把10赋给size。
- 算出值为-2后把它扔掉（这个代码导致无结果的警告信息）。删去分号就会改掉错误。

解答 10-4：预处理器的输出为：

```
void exit();
main() {
    int value;

    value = 1;
    if (value < 0)
        printf("Fatal Error:Abort\n");exit(8);
    printf("We did not die\n");
    return (0);
}
```

问题出在if行后面的两个语句。一般地，它们应该放在两行，如果正确缩进的话，应该是下面的形式：

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

main() {
    int value;  /* a random value for testing */

    value = 1;
    if (value < 0)
        printf("Fatal Error:Abort\n");
```

```
        exit(8);

        printf("We did not die\n");
        return (0);
    }
```

从这段代码中容易看出，为什么总是退出。事实上，if 后面的两个语句通过使用预处理器宏被隐藏了起来。

纠正这个错误的方法是在多语句宏的所有语句外加一对括号（{}）。

```
#define DIE {printf("Fatal Error:Abort\n");exit(8);}
```

解答 10-5：程序显示：

```
x 1 x squared 1
x 2 x squared 3
x 3 x squared 5
x 4 x squared 7
x 5 x squared 9
```

问题出在 SQR（counter+1）表达式上，扩展这个表达式得到：

```
SQR(counter+1)
(counter + 1 * counter + 1)
```

所以 SQR 宏无法运行。参数两边加括号可以解决这个问题：

```
#define SQR(x)  ((x) * (x))
```

解答 10-6：答案是每次循环后 counter 增 2，这是因为宏调用：

```
SQR(++counter)
```

扩展为：

```
((++counter) * (++counter);
```

解答 10-7: 参数宏和不带参数的宏唯一的区别是，参数宏的宏名称后紧跟有一个括号。本例中，RECIPROCAL定义后是一个空格，所以它不是一个参数宏，它只是一个简单的文本替换宏，把RECIPROCAL替换为:

```
(number) (1.0 / number)
```

删去RECIPROCAL和（number）之间的空格，问题就会解决。

编程练习

练习 10-1: 写一个宏，如果它的参数能被10整除则返回真，否则返回假。

练习 10-2: 写一个名为is_digit的宏，如果它的自变量是十进制数，则返回真。

练习 10-3: 再写一个名为is_hex的宏，如果它的自变量是十六进制数（0-9、A-F，a-f），则返回真。本题的宏要引用上题的宏。

练习 10-4: 写一个预处理器宏交换两个整数。（如果你是一个真正的高手，只通过宏来完成而不需在宏外另定义一个临时变量。）

第十一章 位运算

本章内容

- 位运算符
- 与运算符(&)
- 按位或(|)
- 按位异或(^)
- 非运算符(~)
- 左移与右移运算符(<<, >>)
- 设置、清除和检测位
- 位图图形
- 答案
- 编程练习

生存或者毁灭，这是一个问题。

——莎士比亚，用于布尔代数

[《哈姆雷特》，第三场，第一幕]

本章讨论面向位的运算。位是信息的最小单位，一般来讲它由值1或0来表示（其他的表示方法有开／关、真／假和是／否）。位运算用来在最底层控制机器，它允许程序员接触到机器的内部，许多较高级的程序从来不需要位运算，低级代码如编写设备驱动程序或像素级的图形程序才需要位运算。

8个位放在一起组成一个字节，表示C中的数据类型 **char**（注1）。

一个字节可以包含如下的位：

```
01100100
```

这种位结构也可写成十六进制数0x64（C用前缀"0x"表示十六进制数（基为16）），用十六进制数表示二进制数时很方便，因为每个十六进制数代表4个二进制位。表11-1列出了十六进制（hex）与二进制数的对应表：

注1：　从技术角度讲，标准C语言不限定字符的位数，然而我所知道的每一台机器一个C字符都是8位。

表 11-1　十六进制与二进制

十六进制	二进制	十六进制	二进制
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

所以十六进制数 0xAF 代表二进制数 10101111。

Printf 的十六进制格式是 %x，八进制格式是 %o。所以：

```
int number = 0xAF;
printf("Number is %x %d %o\n", number, number, number);
```

显示：

```
af 175 257
```

许多初学编程者对数字的表示法比较模糊，他们常常问“数字怎么知道它自己是十六进制还是十进制数？”

为列出表示法和进制的区别，图 11-1 设想了一袋子石子。

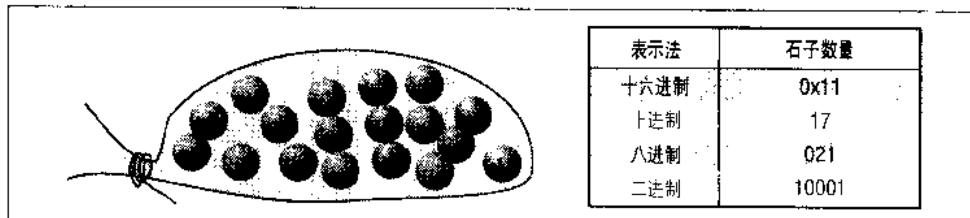


图 11-1　一袋石子

袋子里有17个石子，但那就是十六进制的0x11、十进制的17，八进制的021或二进制的10001吗?答案是数字不论是17还是别的都无关紧要，选择什么表示法取决于我们(八进制、十进制、十六进制或二进制)，石子并不在乎我们怎么数它。

位运算符

位运算符允许程序员对单个的位进行操作，例如，一个短整数可存放16位(对大多数机器而言)。位运算符把这些位看作彼此独立，相比之下，加法运算符就把这16位看成一个由16位组成的单个的数。

位运算符可以对位进行设置、清除、检测操作，还可以执行其他的运算。位运算符列表见表11-2。

表11-2　位运算符

运算符	含义
&	按位与
|	按位或
^	按位异或
~	非
<<	左移
>>	右移

这些运算符可以作用于任何的整型或是字符型数据。

与运算符 (&)

与运算符比较两个位，如果它们都为1，则结果为1。与运算符的定义如表11-3所示。

表 11-3 与运算符

位 1	位 2	位 1 & 位 2
0	0	0
0	1	0
1	0	0
1	1	1

当两个 8 位变量（char 变量）进行与运算时，与运算符独立地对每一位进行操作。下面的程序描述了这个运算：

```
int     c1, c2;

c1 = 0x45;
c2 = 0x71;
printf("Result of %x & %x = %x\n", c1, c2, (c1 & c2));
```

这个程序的输出为：

```
Result of 45 & 71 = 41
```

运算过程是：

	c1 = 0x45	二进制 01000101
&	c2 = 0x71	二进制 01110001
=	0x41	二进制 01000001

按位与（&）类似于逻辑与（&&）。在逻辑与中，如果两个操作数都为真（非0），则结果为真（1）。在按位与（&）中，如果两个操作数中对应的位都为真（1），则相应的位结果为真（1）。所以说按位与（&）是独立地对位进行操作，而逻辑与（&&）则把操作数看成一个整体。

但是，& 和 && 毕竟不是同一个运算符，示例 11-1 中说明了这个问题。

例 11-1: and/and.c

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

```
int main()
{
    int i1, i2; /* two random integers */

    i1 = 4;
    i2 = 2;

    /* Nice way of writing the conditional */
    if ((i1 != 0) && (i2 != 0))
        printf("Both are not zero\n");

    /* Shorthand way of doing the same thing */
    /* Correct C code, but rotten style */
    if (i1 && i2)
        printf("Both are not zero\n");

    /* Incorrect use of bitwise and resulting in an error */
    if (i1 & i2)
        printf("Both are not zero\n");

    return (0);
}
```

问题: 为什么进行#3检测时程序不显示“Both are not zero(两者都为零)”?

答案: 运算符 & 是按位与。按位与的结果为 0 。

i1 =4	01000100
i2 =2	00000010
&	00000000

因为按位与的结果为 0、所以条件不满足。如果程序员使用第一种形式:

```
if ((i1 != 0) && (i2 != 0))
```

同时把 && 错误地写成 &:

```
if ((i1 != 0) & (i2 != 0))
```

则程序也会正确地执行。

```
(i1 != 0)    is true (result = 1)
(i2 != 0)    is true (result = 1)
```

1 与 1 进行按位与，结果还是 1，所以表达式为真（注 2）。

你可以使用按位与运算符来判定一个数是奇数还是偶数。基数为 2 时，所有偶数的最后一位数字是 0，所有奇数的最后一位数字是 1。下列函数使用按位与运算截取最后一位数字，如果它是 0（偶数），函数返回真。

```
int even(const int value)
{
    return ((value & 1) == 0);
}
```

注意： 这个程序使用了一种叫做“聪明把戏”的编程技术，一般来讲，应尽量避免聪明把戏；本例中使用取模（%）运算符更好一些；使用 & 的唯一原因是因为我们在这一部分中讨论它。

按位或 (\)

或运算符（也称或运算）比较两个操作数，如果其中有一个位是 1，则结果就是 1。表 11-4 列出了或运算的真值表。

表 11-4 或运算符

<table>
<tr><th>位 1</th><th>位 2</th><th>位 1 | 位 2</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

注 2： 发现程序中的这个小问题后不久，我对办公室的同事说：“现在我理解与（&）和与与（&&）之间的不同了”，他也明白我的意思，我一直不知该如何理解语言，我说了这样一句话，而且竟有人明白，这确实令我吃惊。

对一个字节执行或运算的示例为:

```
    i1 =0x47    01000111
|   i2 =0x53    01010011
=   0x57        01010111
```

按位异或(^)

当两个数中有一个为1且不同时为1时，异或运算(也称异或)的结果为1。异或运算的真值见表11-5。

表11-5　异或运算符

位1	位2	位1 ^ 位2
0	0	0
0	1	1
1	0	1
1	1	0

对一个字节执行异或运算为:

```
    i1 =0x47    01000111
^   i2 =0x53    01010011
=   0x14        00010100
```

非运算符(~)

非运算符(也称取反运算符)是一维运算符，它返回操作数的相反值，如表11-6所示。

表 11-6 非运算符

位	~位
0	1
1	0

对一个字节执行非运算为:

c=	0x45	01000101
~c=	0xBA	10111010

左移与右移运算符（<<, >>）

左移运算符把数据向左移动若干位，移出左边界的所有位都将丢失，右侧新增加的位为0。右移运算符与此类似，只是方向不同。例如:

	c=0x1C	00011100
c << 1	c=0x38	00111000
c >> 2	c=0x07	00000111

向左移动一位（x<<1）等同于被2乘（x*2），向左移动2位（x<<2）等同于被4乘（x*4，或 $x*2^2$）。由此，可以得出一个公式：向左移动n位等同于乘上 2^n。为什么用移位取代乘法？因为移位操作比乘法运算更快，所以:

```
i = j << 3;      /* Multiple j by 8 (2**3) */
```

比:

```
i  = j * 8;
```

执行得更快。或者说，如果编译器没能聪明地把乘法转换为左移的话，前一种方法还会是较快的。

有许多聪明的程序员用这种技巧加快程序执行的速度，但却失掉了简明性。不要这么做，编译器会自动地完成这样的加速，也就是说，用左移运算符不会给你带来任何好处，反而会使程序不易读。

左移运算是乘法；那么，右移运算就是除法。所以：

```
q = i >> 2;
```

就是：

```
q = i / 4;
```

同样，这种技巧也不要用在程序中。

详谈右移运算

右移运算特别复杂。当右移一个变量时，C需要填充其左边的空格。对于有符号的变量来说，C用其符号位的值来填充。对于无符号变量来说，C用0来填充。表11-7用来说明一些典型的右移操作。

表11-7　右移示例

	有符号字符	有符号字符	无符号字符
表达式	9>>2	-8>>2	248>>2
二进制值>>2	0000 1010>>2	1111 1000>>2	1111 1000>>2
结果	??00 0010	??11 1110>>2	??11 1110>>2
填充	符号位（0）	符号位（1）（注3）	0
最后结果（二进制）	0000 0010	1111 1110	0011 1110
最后结果（短整型）	2	-2	62

注3：　ANSI C标准指定正确的移位应该是算术的（带符号位）或逻辑的（带零位）。差不多所有的编译器都使用算术右移。

设置、清除和检测位

一个字符（char）包含8个位，每个位都可以看作是一个独立的标志。

位操作可以用来截取单字节中的单个位。例如，假定正在写一个底层的通信程序，准备在一个8K的缓冲区中存储字符，留待稍后再用。对每个字符，还将存储一组状态标志，标志列在表11-8中。

表11-8 通信程序状态值

名字	描述
ERROR	如果设置了任何错误，则为1
FRAMING_ERROR	本字符发生结构错误
PARITY_ERROR	字符奇偶错误
CARRIER_LOST	传送信号消失
CHANNEL_DOWN	通信设备的电源信号消失

我们可以把每个标志存储在其自身的字符变量中。这种格式意味着每个缓存字符都需要5个字节的状态存储空间。对于一个大的缓冲区来说，这个数目太大了。我们可以把每个标志位都放入一个8位的状态字符中，这样所用空间只需原来的1/5。

可以按表11-9给标志赋值。

表11-9 位赋值

位	名字
0	ERROR
1	FRAMING_ERROR
2	PARITY_ERROR
3	CARRIER_LOST
4	CHANNEL_DOWN

按惯例，位的顺序是 76543210，如表 11-9 所示。在图 11-2 中，位 4 和位 1 已经设置。

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

图 11-2 位数

表 11-10 列出了每一位常量的定义。

表 11-10 位值

位	二进制值	十六进制常量
7	10000000	0x80
6	01000000	0x40
5	00100000	0x20
4	00010000	0x10
3	00001000	0x08
2	00000100	0x04
1	00000010	0x02
0	00000001	0x01

定义过程是：

```
/* True if any error is set */
const int ERROR =       0x01;

/* A framing error occurred for this character */
const int FRAMING_ERROR =  0x02;

/* Character had the wrong parity */
const int PARITY_ERROR = 0x04;

/* The carrier signal went down */
const int CARRIER_LOST =   0x08;
```

```
/* Power was lost on the communication device */
const int CHANNEL_DOWN =    0x10;
```

位的这种定义方法不太清晰．你能不看表说出哪个位代表常量0x10吗?表11-11显示了如何使用左移运算符（<<）来定义位。

表11-11 左移运算符和位的定义

C 表达式	二进制的等效表达式	结果(二进制)	位号
1<<0	00000001 << 0	00000001	Bit 0
1<<1	00000001 << 1	00000010	Bit 1
1<<2	00000001 << 2	00000100	Bit 2
1<<3	00000001 << 3	00001000	Bit 3
1<<4	00000001 << 4	00010000	Bit 4
1<<5	00000001 << 5	00100000	Bit 5
1<<6	00000001 << 6	01000000	Bit 6
1<<7	00000001 << 7	10000000	Bit 7

虽然很难说出0x10代表哪个位，但很容易就能说出哪个位意思是1<<4。

因此，可以如下定义标志:

```
/* True if any error is set */
const int ERROR =             (1<<0);

/* A framing error occurred for this character  */
const int FRAMING_ERROR =    (1<<1);

/* Character had the wrong parity */
const int PARITY_ERROR =     (1<<2);

/* The carrier signal went down */
const int CARRIER_LOST =     (1<<3);

/* Power was lost on the communication device */
const int CHANNEL_DOWN =     (1<<4);
```

现在已经定义了位，可以操作它了。若要设置一个位的值，用 | 运算符，如：

```
char    flags = 0;  /* start all flags at 0 */

    flags |= CHANNEL_DOWN; /* Channel just died */
```

要检测一个位，使用 & 运算符把位“屏蔽起来”：

```
if ((flags & ERROR) != 0)
    printf("Error flag is set\n");
else
    printf("No error detected\n");
```

清除一个位有点难，假定想清除位 PARITY_ERROR，在二进制中，这个位是 00000100，要建立一个屏蔽，它能屏蔽掉所有的位，除了你想清除的位(11111011)外，这个步骤可以用非运算符(~)来实现，然后，屏蔽和要清除那一位的数进行与运算。

PARITY_ERROR	00000100
~PARITY_ERROR	11111011
flags	00000101
flags & ~PARITY_ERROR	00000001

在 C 中，可以用：

```
flags &= ~PARITY_ERROR; /* Who cares about parity */
```

问题 11-1：例 11-2 中，HIGH_SPEED 标志起作用，但 DIRECT_CONNECT 标志不起作用，为什么？

例 11-2: high/high.c

```
#include <stdio.h>

const int HIGH_SPEED = (1<<7);    /* modem is running fast */

/* we are using a hardwired connection */
const int DIRECT_CONNECT = (1<<8);
```

```
char flags = 0;          /* start with nothing */

int main()
{
    flags |= HIGH_SPEED;     /* we are running fast */
    flags |= DIRECT_CONNECT;/* because we are wired together */

    if ((flags & HIGH_SPEED) != 0)
        printf("High speed set\n");

    if ((flags & DIRECT_CONNECT) != 0)
        printf("Direct connect set\n");
    return (0);
}
```

位图图形

现在越来越多的计算机可以处理图形了，在PC中，已有了像EGA和VGA卡这样的图形设备；在UNIX中，有了X窗口系统。

在位图图形中，屏幕上的每个像素都由内存中的1个位来表示，例如图11-3显示的是一个14*14的位图在屏幕上的样子以及放大后的示意图。放大后，可以看清它的各个位。

假定有一个小型图形设备——一个16*16像素的单显，我们想设置（4，7）位置的点，这个设备的位图是一个位数组，如图11-4中显示的那样。

这里有一个问题，在C中没有用于数组的数据类型，最接近的是一个字节数组。所以16*16的位数组就变成了2*16的字节数组，如图11-5所示。

下面看一下怎样把x，y下标转化成一个byte_x、byte_y、bit_index和bit。

byte_y和y下标相同，转化简单：

```
byte_y = y;
```

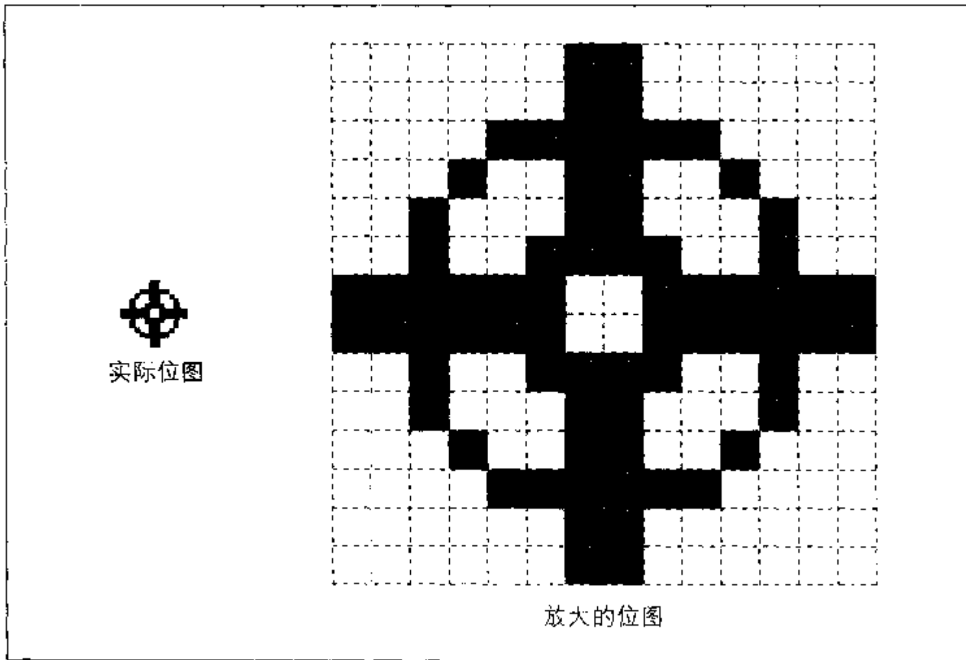


图 11-3 实际的位图和放大后的位图

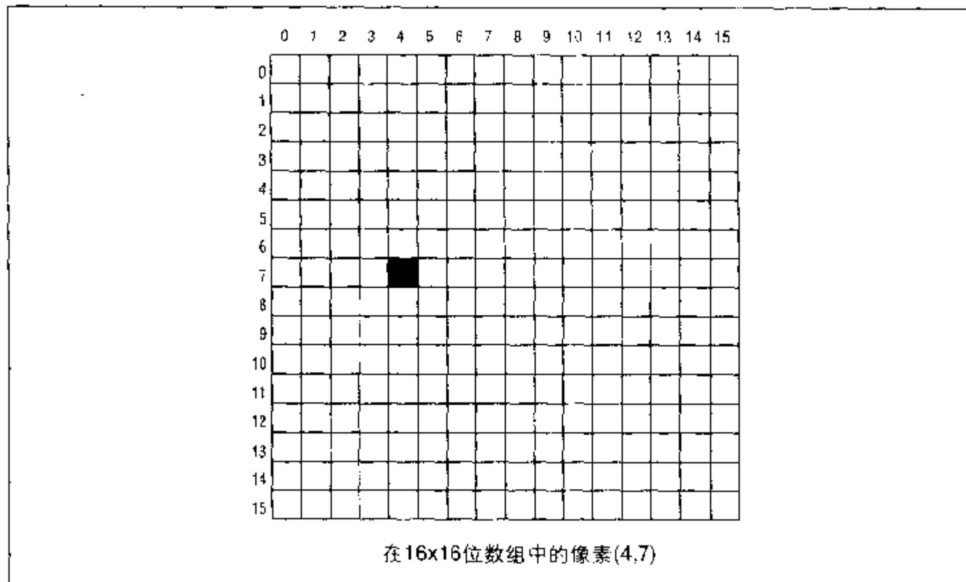


图 11-4 位数组

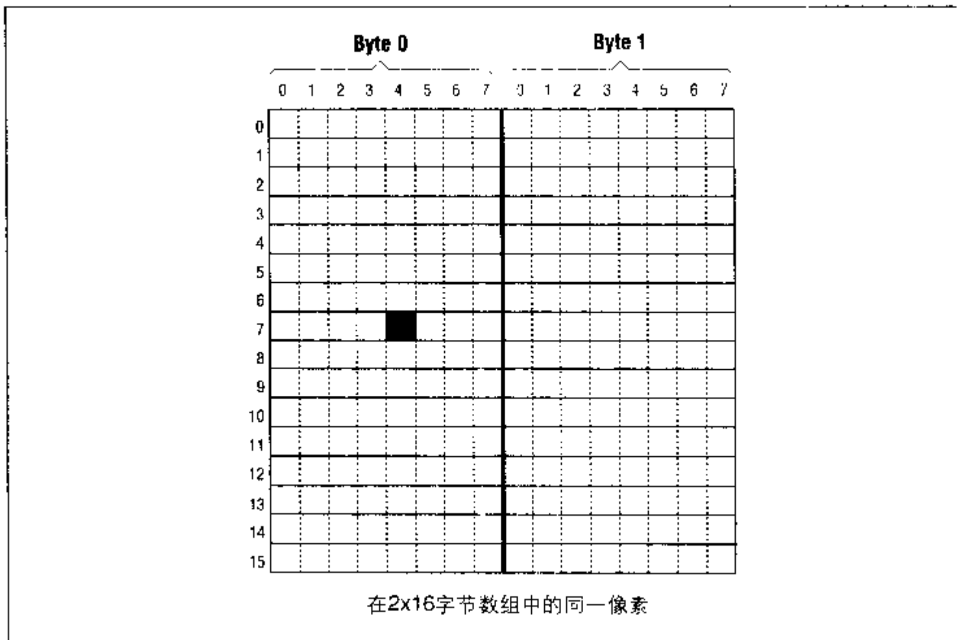


图 11-5 字节数组

1 个字节包含 8 位，所以在 X 方向中，字节下标是位下标的 8 倍。这种转化有下列代码：

```
byte_x = x / 8;
```

现在需要的是位下标。位下标从 0 到 7，然后回到 0，得到下列代码：

```
bit_index = x % 8;
```

现在说明位、位下标为零表示最左边的位或由 1000 00002 或 0x80 表示的位。位下标为 1 表示和最左边的位相临的位 0100 00002 或 0x80>>1。我们想要的位由下列表达式给出：

```
bit = 0x80 >> bit_index;
```

整个算法是：

```
byte_y = y;
byte_x = x / 8;
bit_index = x % 8;
bit = 0x80 >> bit_index;
graphics[byte_x][byte_y] |= bit;
```

该算法可以压缩成一个宏:

```
#define set_bit(x, y) graphics[(x)/8][y] |= (0x80 >> ((x)%8))
```

例如，要设置（4、7）的像素，需要设置（0、7）字节的第4个位，宏会产生下列语句:

```
bit_array[0][7] |= (0x80 >> (4));
```

例 11-3 画出图形数组的一条对角线，并在终端上输出该数组。

例 11-3: graph/graph.c

```
#include <stdio.h>

#define X_SIZE 40   /* size of array in X direction */
#define Y_SIZE 60   /* size of array in Y direction */
/*
 * We use X_SIZE/8 because we pack 8 bits per byte
 */
char graphics[X_SIZE / 8][Y_SIZE];    /* the graphics data */

#define SET_BIT(x,y) graphics[(x)/8][y] |= (0x80 >>((x)%8))

int main()
{
    int   loc;          /* current location we are setting */
    void  print_graphics(void); /* print the data */

    for (loc = 0; loc < X_SIZE; ++loc)
        SET_BIT(loc, loc);

    print_graphics();
    return (0);
}
```

```
/*****************************************************
 * print_graphics --Prints the graphics bit array    *
 *               as a set of X and .'s.              *
 *****************************************************/
void print_graphics(void)
{
    int x;              /* current x BYTE */
    int y;              /* current y location */
    unsigned int bit;   /* bit we are testing in the current byte */

    for (y = 0; y < Y_SIZE; ++y) {
        /* Loop for each byte in the array */
        for (x = 0; x < X_SIZE / 8; ++x) {
            /* Handle each bit */
            for (bit = 0x80; bit > 0; bit = (bit >> 1)) {
                if ((graphics[x][y] & bit) != 0)
                    printf("X");
                else
                    printf(".");
            }
        }
        printf("\n");
    }
}
```

程序中定义了一个位图数组:

```
char graphics[X_SIZE / 8][Y_SIZE];    /* the graphics data */
```

因为共有X_SIZE个位，转为字节就是X_SIZE/8个字节，所以要用到X_SIZE/8个常量。

主 **for** 循环:

```
for (loc = 0; loc < X_SIZE; ++loc)
        set_bit(loc, loc);
```

画出一条斜穿图形数组的对角线。

因为我们没有位图设备，所以用子程序print_graphics来仿真它。循环:

```
for (y = 0; y < Y_SIZE; ++y) {
    ....
```

可以输出每一行。循环:

```
for (x = 0; x < X_SIZE / 8; ++x) {
        ...
```

则输出行中的每个字节。每个字节中有8位，这由下面的循环来处理:

```
for (bit = 0x80; bit > 0; bit = (bit >> 1))
```

它使用了一个独特的循环计数器，该循环让变量bit从7位（最左位）开始，对循环的每一次重复，位都向右移一位，即bit = (bit >> 1)。当移完了位后，循环退出。循环计数器走了一圈:

二进制	十六进制
0000 0000 1000 0000	0x80
0000 0000 0100 0000	0x40
0000 0000 0010 0000	0x20
0000 0000 0001 0000	0x10
0000 0000 0000 1000	0x08
0000 0000 0000 0100	0x04
0000 0000 0000 0010	0x02
0000 0000 0000 0001	0x01

最后，在循环的内部有如下的代码:

```
if ((graphics[x][y] & bit) != 0)
    printf("X");
else
    printf(".");
```

这个代码用来测试一个单独的位，如果该位已被设置了，则输出“X”；如果没被设置，则输出“.”。

问题 11-2： 例 11-4 中第一循环运行、第二个循环不运行，为什么？

例 11-4: loop/loop.c

```
#include <stdio.h>
int main()
{
    short int i;            /* Loop counter */
    signed char ch;         /* Loop counter of another kind */

    /* Works */
    for (i = 0x80; i != 0; i = (i >> 1)) {
        printf("i is %x (%d)\n", i, i);
    }

    /* Fails */
    for (ch = 0x80; ch != 0; ch = (ch >> 1)) {
        printf("ch is %x (%d)\n", ch, ch);
    }
    return (0);
}
```

答案

解答 11-1： DIRECT_CONNECT 由表达式（1<<8）定义为第 8 位；但是，字符变量的 8 个位的序号是 76543210，没有第 8 位。解决办法是把 flags 说明为 16 位短整型。

解答 11-2： 问题出在 ch 是一个字符（8 位）。值 0x80 用 8 位表示就是 1000 00002。第一位是符号位，它为 1。当右移 1 位时，用符号位来填充，所以，1000 00002>>1 得到 1100 00002。

虽然变量 i 有符号，但也得出了结果，因为它有 16 位。所以，用 16 位表示时，0x80 就是 0000 0000 1000 00002。注意，设置的位不是符号位。

解决的办法是把 ch 说明为一个无符号的变量

编程练习

练习 11-1: 写一组带参数的宏: clear_bit 和 test_bit, 与例 11-3 中定义的 set_bit 函数一起运行。编写一个主程序来测试这些宏。

练习 11-2: 编写一个程序, 画一个 10*10 的位图正方形。可以从例 11-3 借用代码来显示结果。

练习 11-3: 修改示例 11-3, 在黑背景下画一条白线。

练习 11-4: 编写一个程序, 统计一个整数中设置位的个数。例如, 数 5 (十进制) 表示为 0000000000000101 (二进制), 它有两个设置位。

练习 11-5: 编写一个程序, 它把一个 32 位的整数 (长整) 划分为 8 个 4 位的数。(小心处理符号位)

练习 11-6: 编写一个程序, 它把一个数中的所有设置位移到最左边。例如, 将 01010110 (二进制) 变为 11110000 (二进制)。

第十二章

高级类型

本章内容

- 结构
- 联合
- typedef
- 枚举类型
- 强制类型转换
- 位字段或紧缩结构
- 结构数组
- 小结
- 编程练习

整个大厦的宏伟壮观，都由建筑设计师决定。

——Wallace Stevens

C提供了一组丰富的数据类型，通过使用结构、联合、枚举和类等类型，程序员可以用新的类型来扩展语言。

结构

假定你正在为仓库写一个盘货程序，仓库里堆满了箱子，箱子内的东西也分类整理好了。箱子内所有的类都作了标记，所以你不必担心箱子或其中的东西弄混了。

对每个箱子你需要了解：

- 内部各部分的名字（30个字符长的串表示）。
- 已有数量（用整数表示）。
- 价格（用整数表示，以分为单位）。

在前面的章节中，已经用数组存放过一组有相似数据类型的数据；但在本例中，你的数据是不同的类型：两个整数和一个字符串。

我们不使用数组，而使用一种新的数据类型，名为结构。在数组中，所有的元素都是同一类型的，并且有次序。而在结构中，每个元素或字段都有名字，并有自己的数据类型。

结构定义的一般形式为：

```
struct 结构名 {
    字段类型  字段名     /*注释*/
    字段类型  字段名     /*注释*/
    ....
} 变量名;
```

例如，定义一个装打印机电缆的箱子。结构定义如下：

```
struct bin {
    char    name[30];      /* name of the part */
    int     quantity;      /* how many are in the bin */
    int     cost;          /* The cost of a single part  (in cents) */
} printer_cable_bin;       /* where we put the print cables */
```

实际上，这个定义告诉C两件事，第一件事是：结构bin的定义如上所述，该语句定义了一种新的数据类型，它可以用于定义其它的变量。这条语句还定义了变量printer_cable_bin。因为bin的结构已经定义了，所以可以用它来定义其他的变量：

```
struct bin terminal_cable_box;  /* Place to put terminal cables */
```

定义中的结构名可以省略不写。

```
struct {
   char    name[30];      /* name of the part */
    int     quantity;     /* how many are in the bin */
    int     cost;         /* The cost of a single part  (in cents) */
} printer_cable_bin;      /* where we put the print cables */
```

定义了变量printer_cable_bin，但还没有建立数据类型，换句话说，printer_cable_bin是这个程序中你需要的唯一变量，这个变量的数据类型是匿名结构。

变量名部分也可以省略，下例将定义一个结构类型，但没有变量：

```
struct bin {
    char    name[30];   /* name of the part */
    int     quantity;   /* how many are in the bin */
    int     cost;       /* The cost of a single part  (in cents) */
};
```

现在可以使用新数据类型（struct bin）来定义如printer_cable_bin这样的变量。

在极少数情况下，变量名和结构名都可以省略，这种语法创造出一段正确但完全无用的代码。

已经定义了一个变量printer_cable_bin，它包含三个有名称的字段：名称、数量和成本，访问它们使用下列语法：

```
变量.字段
```

例如，如果你发现电缆的价格上涨到了12.95美元，就可以这样修改：

```
printer_cable_bin.cost = 1295;   /* $12.95 is the new price */
```

为了计算箱子中所有东西的价值，可以用数量乘上价格：

```
total_cost = printer_cable_bin.cost * printer_cable_bin.quantity;
```

结构可以在定义时进行初始化，即把每个字段的值放在括号（{}）内。

```
/*
 * Printer cables
 */
struct bin {
    char    name[30];   /* name of the part */
    int     quantity;   /* how many are in the bin */
    int     cost;       /* The cost of a single part  (in cents) */
} printer_cable_bin = {
    "Printer Cables",   /* Name of the item in the bin */
    0,                  /* Start with empty box */
    1295                /* cost   $12.95  */
```

```
};
```

联合

结构用来定义带有几个字段的数据类型，每个字段都占有一个单独的存储位置。例如，下列结构存储于内存：

```
struct rectangle {
    int width;
    int height;
};
```

联合类似于结构；只是它定义了一个可以有多个不同字段名的位置。

```
union value {
    long int i_value;    /* integer version of value */
    float f_value;       /* floating version of value */
};
```

字段 `i_value` 和 `f_value` 分享同一空间。

可以把结构认为是一个大盒子被分成不同的格子，每个格子都有自己的名字。联合是一个盒子，但一点也没分隔；只是在内部单一的格子内，标上了几个不同的标签。

图 12-1 列示了一个带有两个字段的结构，每个字段分配到结构的不同部分，联合只包含一个格子，这个格子被分配了不同的名字。

在结构内，字段互不影响，修改一个字段不会改变其他字段的内容。在联合内，所有的字段都占据同一空间，所以在同一时刻只有一个是活动的。换句话说，如果你在（i_value）中放入某个值，如再给 f_valuer 赋值则会覆盖 i_value 的值。

例 12-1 表明了联合的用法。

例 12-1：使用联合

```
/*
```

```
 * Define a variable to hold an integer or
 * a real number (but not both)
 */
```

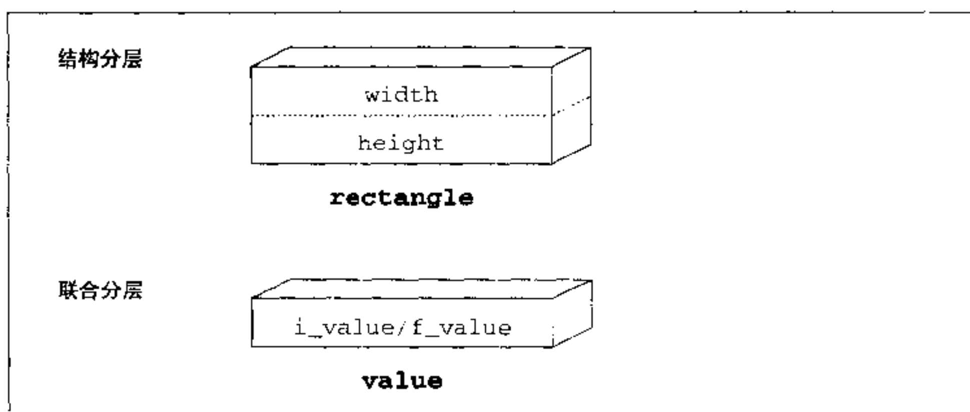


图 12-1 结构分层和联合分层

```
union value {
    long int i_value;   /* The real number */
    float f_value;      /* The floating-point number */
} data;
int i;                  /* Random integer */
float f;                /* Random floating-point number */
main()
{
    data.f_value = 5.0;
    data.i_value = 3;   /* data.f_value overwritten */
    i = data.i_value;   /* legal */
    f = data.f_value;   /* not legal, will generate unexpected results */
    data.f_value = 5.5; /* put something in f_value/clobber i_value */
    i = data.i_value;   /* not legal, will generate unexpected results */
    return(0);
}
```

联合常用于通信方面，如给遥控磁带机发送四条消息：开、关、读、写。这四条信息的数据内容因其自身差异而有很大不同，打开信息要包含磁带名；写信息包含写的内容；读信息要包含要读取字符的最大数量；关信息则不需要额外的信息。

```
#define DATA_MAX 1024 /* Maximum amount of data for a read and write */

struct open_msg {
```

```
    char name[30];       /* Name of the tape */
};

struct read_msg {
    int length;          /* Max data to tranfer in the read */
};

struct write_msg {
    int length;            /* Number of bytes to write */
    char data[DATA_MAX];  /* Data to write */
};

struct close_msg {
};

const int OPEN_CODE=0;     /* Code indicating an open message */
const int READ_CODE=1;     /* Code indicating a read message */
const int WRITE_CODE=2;    /* Code indicating a write message */
const int CLOSE_CODE=3;    /* Code indicating a close message */

struct msg {
    int msg; /* Message type */
    union {
        struct open_msg open_data;
        struct read_msg read_data;
        struct write_msg write_data;
        struct close_msg close_data
    } msg_data;
};
```

typedef

C 允许通过 **typedef** 语句定义自己的变量类型，这条语句提供了一种扩展 C 基本类型的方法。**Typedef** 的一般形式是:

```
typedef 类型定义
```

类型定义和变量定义相同，只是类型名取代了变量名，如:

```
typedef int count;
```

定义了一个和整型一样的新类型 count。

所以定义:

```
count flag;
```

等同于:

```
int flag;
```

粗看上去，这与

```
#define count int
count flag;
```

没什么区别，但是 **typedef** 可以用来定义更复杂的对象，这远远超出了简单的 **#define** 语句的范畴，例如:

```
typedef int group[10];
```

现在定义了一个新类型名 group，它表示由 10 个整数组成的一个数组。例如:

```
main()
{
    typedef int group[10];     /* Create a new type "group" */
    group totals;              /* Use the new type for a variable */
    for (i = 0; i < 10; i++)
        totals[i] = 0;
    return (0);
}
```

typedef 常用在新结构的定义中，例如:

```
struct complex_struct {
    double real;
    double imag;
};
typedef struct complex_struct complex;

complex voltag1 = {3.5, 1.2};
```

枚举类型

枚举数据类型用来定义只可取有限个值的变量，这些值都由名字（标签）引用。编译器在内部为每个标签赋予一个整数值。考虑一个应用，如一个星期中的各天，可以用 **const** 定义为一周内各天建立值，如下：

```
typedef int week_day;   /* define the type for week_days */
const int SUNDAY    = 0;
const int MONDAY    = 1;
const int TUESDAY   = 2;
const int WEDNESDAY = 3;
const int THURSDAY  = 4;
const int FRIDAY    = 5;
const int SATURDAY  = 6;
/* now to use it */
week_day today = TUESDAY;
```

这种方法不太方便，更好的方法是使用枚举类型：

```
enum week_day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY};
/* now use it */
enum week_day today = TUESDAY;
```

enum 语句的一般形式是：

```
enum 枚举名 { 标志1, 标志2, ...} 变量名
```

和结构一样，枚举名或变量名都可以省略，标记可以是任意有效的 C 标识符；不过，标记通常要大写。

C 执行枚举类型和整数具有可兼容性，所以在 C 中，虽然在有些编译器读到这行时会发出警告信息，下列语句还是完全合法的：

```
today = 5;  /* 5 is not a week_day */
```

在 C++ 中，enum 是独立的类型，和整数不兼容。

强制类型转换

有时需要把一种类型的变量转化成另一种类型，这一过程可通过强制类型转换操作来完成。强制类型转换的一般形式：

```
(类型) 表达式
```

这一操作告诉C计算表达式的值，然后把它传送给预定的类型，当处理整数和浮点数时这个操作很有用：

```
int won, lost;        /* # games won/lost so far */
float   ratio;        /* win/lose ratio */
won = 5;
lost = 3;
ratio = won / lost; /* ratio will get 1.0 (a wrong value) */
/* The following will compute the correct ratio */
ratio = ((float) won) / ((float) lost);
```

这个操作的另外一个常用的方法是把指针从一种类型转化为另一种类型。

位字段或紧缩结构

紧缩结构允许用一种占用最小空间的方法来定义结构，例如，下列结构占用6字节（在16位机上）：

```
struct item {
    unsigned int list;       /* true if item is in the list */
    unsigned int seen;       /* true if this item has been seen */
    unsigned int number;     /* item number */
};
```

这个结构的存储布局如图12-2所示，每个结构用6个字节存储(每个整数两字节)。

但是，list和seen字段只有两个值，0或1，所以表示它们只需各自用一个位。无需计划使用超过16383个项（0x3fff或14位）。可以把这个结构用位字段重新定义，使它只占用2字节，定义时，每一字段后加一个冒号，并写上这个字段要使用的位数。

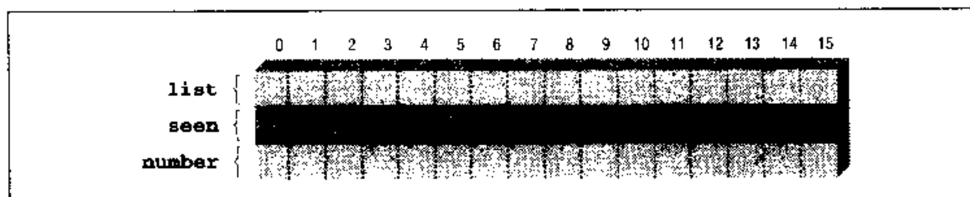


图 12-2 非紧缩结构

```
struct item {
    unsigned int list:1;    /* true if item is in the list */
    unsigned int seen:1;    /* true if this item has been seen */
    unsigned int number:14; /* item number */
};
```

本例中程序告诉编译器给 list 和 seen 分别用一位，number 用 14 位。用这种方法可以把数据压缩为两字节，如图 12-3 所示：

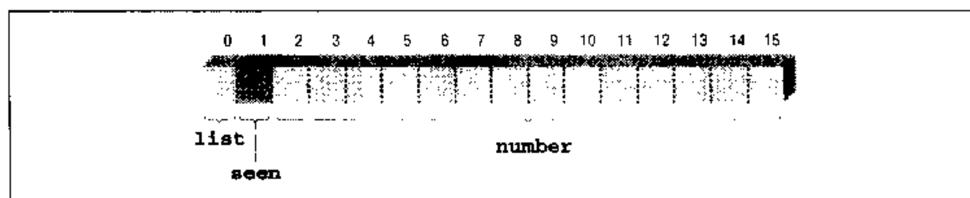


图 12-3 紧缩结构

使用紧缩结构应小心，从位字段中移出数据的代码相对较大，运行也较慢。因此除非存储非常紧张，否则不要使用紧缩结构。

在第十章"C 预处理器"中，需要存储字符数据和 8000 个字符对应的 5 个状态标志。在这个例子中，对每个标志都使用了不同的字节，所以用掉了许多存储空间（每个读取的字符占 5 个字节）。最后使用了按位运算把 5 个字节压缩到 1 个字节中。用一个紧缩结构可以完成同样的功能：

```
struct char_and_status {
    char character;       /* Character from device */
    int error:1;          /* True if any error is set */
    int framing_error:1;/* Framing error occurred */
```

```
    int parity_error:1; /* Character had the wrong parity */
    int carrier_lost:1; /* The carrier signal went down */
    int channel_down:1; /* Power was lost on the channel */
};
```

使用紧缩结构来表示标志，比使用按位运算更清楚，也不易出错。但是，按位运算有更大的灵活性，你可以使用对你来说是最清楚最简单的方法。

结构数组

结构和数组可以组合在一起。假如想记录一名运动员四圈赛跑中每一圈跑的时间，可定义一个结构来存放时间：

```
struct time {
    int hour;   /* hour (24 hour clock ) */
    int minute; /* 0-59 */
    int second; /* 0-59 */
};
 const int MAX_LAPS = 4; /* we will have only 4 laps */
/* the time of day for each lap*/
struct time lap[MAX_LAPS];
```

使用下列结构：

```
/*
 * Runner just passed the timing point
 */
lap[count].hour = hour;
lap[count].minute = minute;
lap[count].second = second;
++count;
```

这个数组可以在运行时进行初始化。结构数组的初始化类似于多维数组的初始化。

```
struct time start_stop[2] = {
    {10, 0, 0},
    {12, 0, 0}
};
```

假设想写一个处理邮件表的程序，邮件标签高5行宽60个字符，用一个结构来存放名字和地址，邮件表打印输出时将按名排序；而对于真正的邮件，则以邮编顺序来排列。邮件表的结构如下：

```
struct mailing {
    char name[60];      /* Last name, first name */
    char address1[60];/* Two lines of street address */
    char address2[60];
    char city[40];
    char state[2];      /* Two character abbreviation */
    long int zip;       /* Numeric zip code */
};
```

现在可以定义一个存放邮件表的数组了：

```
/* Our mailing list */
struct mailing list[MAX_ENTRIES];
```

小结

结构和联合是C语言中更强大的特性，你不再局限于C的内部数据类型了——你可以创立自己的数据类型。以后章节我们将看到，结构可以和数据指针结合，以创立复杂而强大的数据结构。

编程练习

练习 12-1：设计一个结构存放邮件列表数据，写一个函数来显示数据。

练习 12-2：设计一个结构来储存时间和日期，写一个函数计算两个时间相差的分钟数。

练习 12-3：设计一个飞机票预订数据结构，其中存储下列数据：

- 航班号
- 起始地代码（3个字符）

- 目的地代码（3个字符）
- 启程时间
- 到达时间

练习12-4：写一程序列出用户指定的起始和目的地两机场之间所有可能的飞行计划。

本章内容

- 函数自变量指针
- 常量指针
- 指针和数组
- 如何不使用指针
- 用指针分隔字符串
- 指针和结构
- 命令行参数
- 编程练习
- 答案

第十三章

简单指针

看法的选择是文明的第一幕。

—— Jose Ortega y Gasset

所有的变量都可以用指针表示、知道变量和指针之间的差别很重要，这种观念列示于图13-1中。

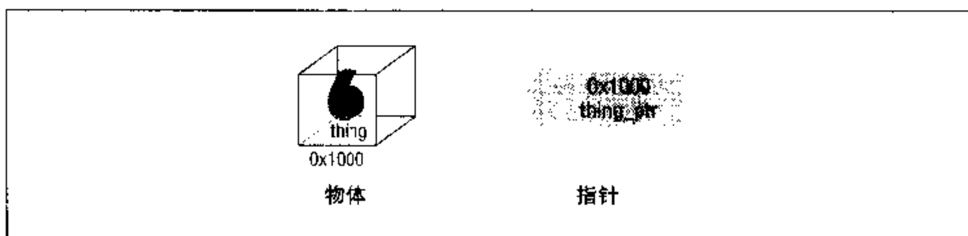


图13-1　一个物体及其指针

本书中我们使用一个盒子代表一个变量，变量名写在盒子底部。本例中的变量名是thing，变量值是6。

thing的地址是0x1000，地址是由C编译器自动为每个变量分配的。一般而言，程序员不必担心变量地址，但应知道它们的存在。

指针（thing_ptr）指向变量thing，指针也叫地址变量，因为它们包含其他变量的地址。此例中，指针所含地址为0x1000，因为这是thing的地址，所以可以说thing_ptr指向thing。

变量和指针很像街道地址和房屋，例如，地址可能是“绿山街214号”，房子有不同的形状和大小，地址大小却差不多都相同（街道、城市、州和邮编）。所以“1600 Pensylvania Ave”可能指向一个很大的白房子，“8347 Undersea Street”可能只是个一间房的小木屋，但它们的地址长度却相同。

C也一样，即物体有大有小，指针大小却一样（相对较小）。（注1）

许多初学编程者容易混淆指针和它们的内容，为避免这个问题的发生，本书中所有的指针变量都用_ptr扩展名结尾。你可能也愿意将这种用法用到你的程序中，虽然这种符号用得不普遍，但确实很有用。

许多不同的地址变量可以指向同一个物体，这种观念对街道地址同样是正确的，表13-1列示了小镇上重要的服务机构的地址。

表13-1 美国艾德镇目录

服务机构（变量名）	地址（地址值）	建筑物（物体）
消防部门	1大街	市政厅
警署	1大街	市政厅
计划办公室	1大街	市政厅
加油站	2大街	艾德加油站

本例中，政府建筑服务于许多功能，虽然只有一个地址，但有三个指针指向它。

正如本章所见，指针可以用来快速简单地访问数组。在后面的章节中，读者将发现如何用指针建立新的变量和复杂的数据结构，如链表和树。读完本书的其余章节后，读者就可以了解这些数据结构，同时也可以建立自己的结构。

在定义语句中，指针是通过在变量名之前加一个星号（*）进行定义的：

```
int thing;        /* define a thing */
```

注1：　严格来讲，这种说法对于MS-DOS/Windows编译器来说是不对的，因为8086的奇怪结构致使这些编译器既使用近（near）指针（16位）又使用远（far）指针（32位），详见C编译器使用手册。

```
int *thing_ptr;  /* define a pointer to a thing */
```

表13-2列示了与指针相关的运算符。

表13-2 指针运算符

运算符	含义
*	逆引用（给定一个指针，得到所引用的物体）
&	地址（给定一个物体，指向它）

&符号把物体改为指针，*符号把指针改为物体，这些运算符很容易混淆，表13-3列出了不同指针运算符的语法。

表13-3 指针运算符语法

C代码	描述
thing	数据本身（变量）
&thing	指向变量thing的地址
thing_ptr（译注1）	一个整数指针（可以指向整数thing，也可以不指向它）
* thing_ptr	整数

下面看一些典型的指针运算符的用法：

```
int thing; /* Declare an integer (a thing) */
thing = 4;
```

变量thing是一个数据，定义int thing不包含*，所以thing不是指针：

```
int *thing_ptr;       /* Declare a pointer to a thing */
```

变量thing_ptr是一个指针，定义中的*的符号表明这是一个指针。此外，我们把扩展名_ptr加进了名字：

```
thing_ptr = &thing;  /* Point to the thing */
```

译注1：由于本书中的指针变量都有-ptr做结尾，因而认为thing_ptr也是一个指针。

表达式&thing是指向物体的指针，变量thing是一个对象，&（运算符地址）得到一个对象（指针）的地址，所以&thing是一个指针。然后把值赋给指针型的变量thing_ptr：

```
*thing_ptr = 5;         /* Set "thing" to 5 */
                        /* We may or may not be pointing */
                        /* to the specific integer "thing" */
```

表达式*thing_ptr表示一个物体，变量thing_ptr是一个指针，*（指针运算符）告诉C看所指的数据，而不看指针本身。注意：它指向任何一个整数，它可以指或不指向具体的变量thing。这些指针运算总结于图13-2中。

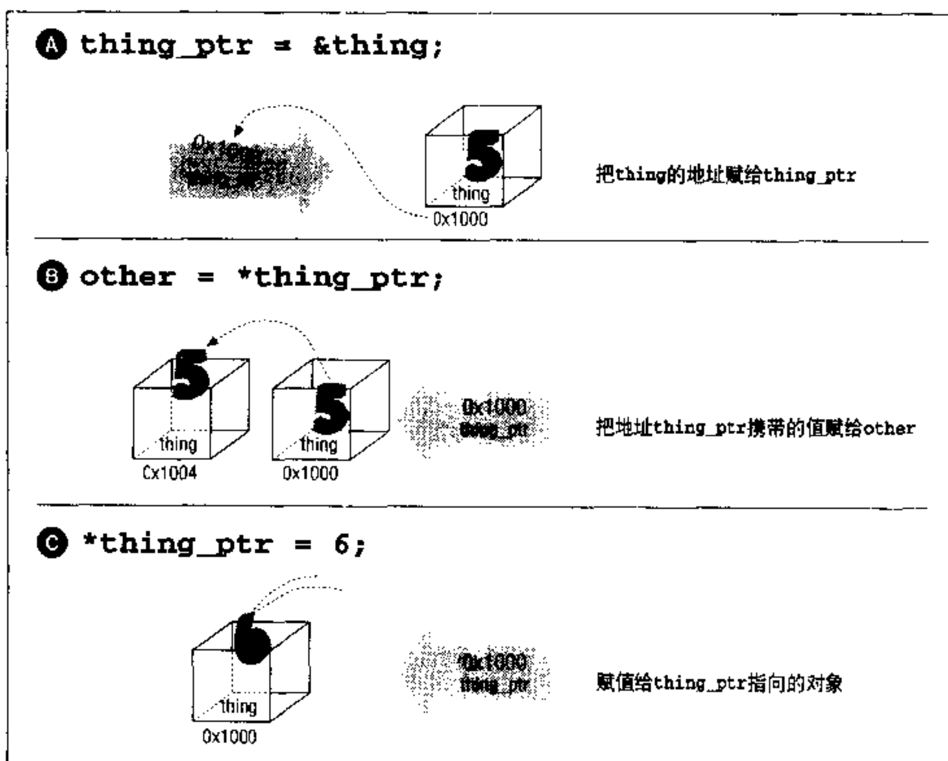


图13-2　指针运算

下列是错用指针运算符的示例：

*thing

是非法的。它告诉C要得到由变量*thing*所指向的对象。因为*thing*不是指针，此运算无效。

&thing_ptr

合法，但是很奇怪。*thing_ptr*是一个指针，&（地址运算符）得到一指向对象的指针（本例中是*thing_ptr*）。结果是一个指向指针的指针。

例13-1列出了指针的简单用法，它定义一个对象，一个thing和一个指针，thing_ptr.thing由下行明确地设置：

```
thing = 2;
```

下行：

```
thing_ptr = &thing;
```

导致C设置thing_ptr到了thing的地址。从这点来看，thing和*thing_ptr是同样的。

例13-1: thing/thing.c

```
#include <stdio.h>
int main()
{
    int   thing_var;  /* define a variable for thing */
    int  *thing_ptr;  /* define a pointer to thing */

    thing_var = 2;      /* assigning a value to thing */
    printf("Thing %d\n", thing_var);

    thing_ptr = &thing_var; /* make the pointer point to thing */
    *thing_ptr = 3;     /* thing_ptr points to thing_var so */
                        /* thing_var changes to 3 */
    printf("Thing %d\n", thing_var);

    /* another way of doing the printf */
    printf("Thing %d\n", *thing_ptr);
    return (0);
}
```

几个指针可以指向同一个物体：

```
1:      int something;
2:
3:      int      *first_ptr;      /* one pointer */
4:      int      *second_ptr;     /* another pointer */
5:
6:      something = 1;           /* give the thing a value */
7:
8:      first_ptr = &something;
9:      second_ptr = first_ptr;
```

在第 8 行中，我们使用 & 运算符把一个物体 something 变成一个指针赋给 first_ptr。由于 first_ptr 和 second_ptr 都是指针，我们可以作第 9 行的直接赋值。

执行这个程序段之后，我们处于图 13-3 的位置。

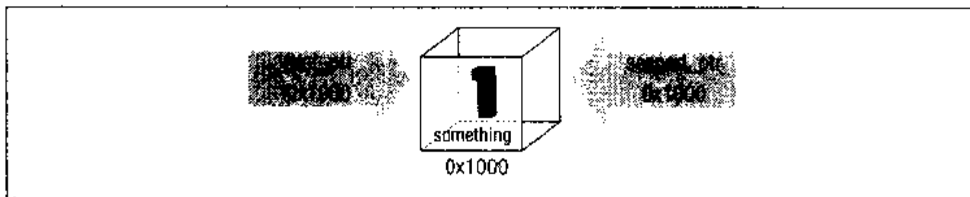


图 13-3　两个指针和一个物体

读者应该注意到已经有了三个变量，只有一个整数（something）。下列语句是等价的：

```
something = 1;
*first_ptr = 1;
*second_ptr = 1;
```

函数自变量指针

C 使用“由值调用”传递参数，也就是说，参数单向传给函数，函数的唯一结果是单个返回值，这一概念列示于图 13-4。

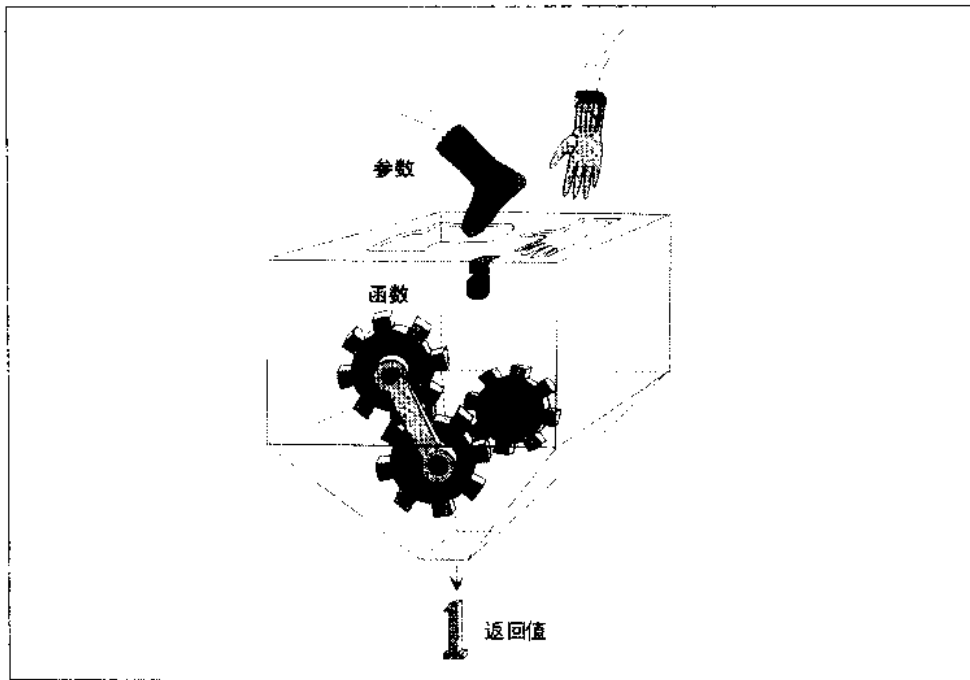


图 13-4　函数调用

不过，指针可以用来消除这种限制。

想像有两个人，Sam 和 Joe，不论何时他们见面，Sam 只能说而 Joe 只能听。Sam 怎样从 Joe 那里得到信息呢?简单：Sam 只须告诉 Joe，“我想让你把答案放进西 5 大街 335 号的邮箱里。”

C 使用同样的技巧把信息从函数传递到它的调用者。在例 13-2 中，main 想用函数 inc_count 给变量 count 增值，直接传递行不通，所以传递了一个指针（“这就是我想让你增加的变量的地址”）。注意 inc_count 的原型包括一个 int。这个格式表示给函数的单个参量是指向一个整数的指针，而不是整数本身。

例 13-2: call/call.c

```
#include <stdio.h>
void inc_count(int *count_ptr)
{
    (*count_ptr)++;
```

```
}
int main()
{
    int  count = 0;      /* number of times through */

    while (count < 10)
        inc_count(&count);

    return (0);
}
```

这个代码用图来表示见13-5，注意参量没变，它指向的变量改变了。

最后，有一个特殊的指针叫NULL，它没有指向的对象（实际的数值是0）。标准引用文件*locale.h*定义常量NULL（这个文件通常并不直接引用，但通常由引用文件*stdio.h*或*stdlib.h*产生），NULL指针的图示见13-6。

常量指针

定义常量指针是一个小技巧，例如定义：

```
const int result = 5;
```

告诉C：result是一个常量，所以

```
result = 10;       /* Illegal */
```

是非法的。但是定义：

```
const char *answer_ptr = "Forty-Two";
```

并没有告诉C变量answer_ptr是一个常量，相反，它告诉C answer_ptr所指的数据是一个常量，数据不能改变，但指针能改变，这再一次要求我们分清物体和物体指针之间的区别。

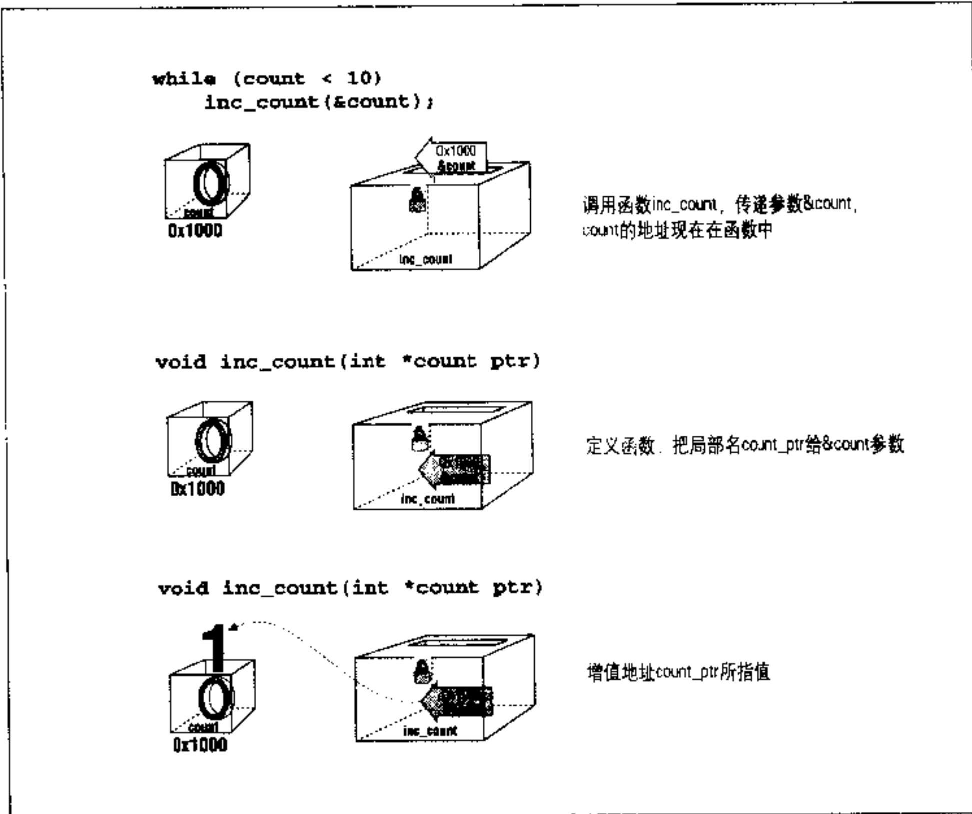


图 13-5　调用 inc_count

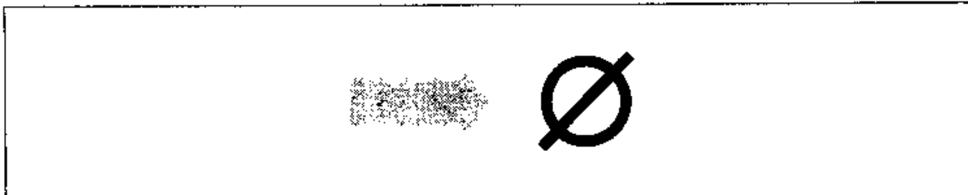


图 13-6　NULL

answer_ptr是什么?它是一个指针。它能被改变吗? 能，因为它仅是一个指针。它指向什么? 一个const char数组。由answer_ptr 指向的数据能改变吗?不能，因为它是常量。

所以在C中:

```
answer_ptr = "Fifty-One";      /* Legal (answer_ptr is a variable) */
*answer_ptr = 'X';             /* Illegal (*answer_ptr is a constant) */
```

如果把 **const** 放在 * 后面，那么告诉C的是: 指针是常量。

例如:

```
char *const name_ptr = "Test";
```

name_ptr 是什么? 一个常量指针。它能被改变吗? 不能。它指向的是什么? 一个字符。name_ptr指向的数据能改变吗? 可以。

```
name_ptr = "New";             /* Illegal (name_ptr is constant) */
*name_ptr = 'B';              /* Legal (*name_ptr is a char) */
```

最后，可以把 **const** 放到两个地方，生成一个不能被改变的指针，同时指向的数据项也不能被改变。

```
const char *const title_ptr = "Title";
```

指针和数组

C 允许指针进行运算（加或减）。假如有下面的代码:

```
char array[5];
char *array_ptr = &array[0];
```

本例中，*array_ptr 和 array[0] 相同，*(array_ptr+1) 和 array[1] 相同，*(array_ptr+2) 和 array[2] 相同等等。注意括号的使用，指针运算图示见 13-7。

然而，(*array_ptr)+1 和 array[1] 却不一样。+1 在括号外，在指针操作完成后进行加法运算，所以 (*array_ptr)+1 就是 array[0]+1。

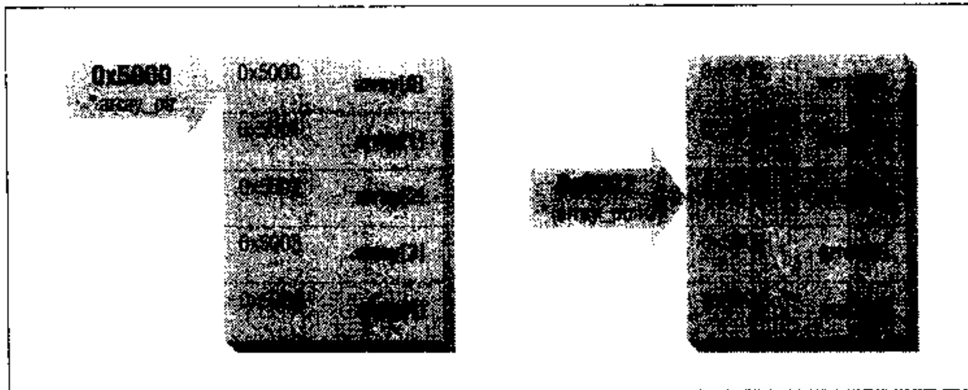


图 13-7　指针和数组

乍看上去，这好像是简单数组的复杂表示方法，从简单指针运算开始，到后面的章节，我们将使用更复杂的指针来高效地处理更困难的功能。

数组各元素都分配在连续的地址中，例如，array[0]可能放在地址0xff000024中，则array[1]应放在地址0xff000025中，依此类推，这种结构意味着指针可用来找到每个数组元素。例13-3显示了一个简单字符型数组的元素和地址。

例 13-3: array-p/array-p.c

```
#include <stdio.h>

#define ARRAY_SIZE 10   /* Number of characters in array */
/* Array to print */
char array[ARRAY_SIZE] = "0123456789";

int main()
{
    int index;  /* Index into the array */

    for (index = 0; index < ARRAY_SIZE; ++index) {
        printf("&array[index]=0x%p (array+index)=0x%p array[index]=0x%x\n",
            &array[index], (array+index), array[index]);
    }
    return (0);
}
```

注意：显示指针，应采用特殊的反转 %p。

运行时，将显示：

```
&array[index] (array+index) array[index]
0x40b0         0x40b0        0x30
0x40b1         0x40b1        0x31
0x40b2         0x40b2        0x32
0x40b3         0x40b3        0x33
0x40b4         0x40b4        0x34
0x40b5         0x40b5        0x35
0x40b6         0x40b6        0x36
0x40b7         0x40b7        0x37
0x40b8         0x40b8        0x38
0x40b9         0x40b9        0x39
```

字符通常占用一个字节，所以字符数组中的元素将分配连续的地址。一个短整型数占用两个字节，所以在短整型数组中元素的地址间隔为2。这是不是意味着array+1只能用于字符而不能用于其他类型？不。为使其运行正确，C自动进行指针的算术运算，本例中，array+1将指向下标为1的元素。

C提供了一种处理数组时的简写法。不用写：

```
array_ptr = &array[0];
```

可以写成：

```
array_ptr = array;
```

C并不认真区分指针和数组，而把它们认为是同一类事物。这里，把变量array当作指针使用，C将自动进行必要的转换。

例13-4统计元素中非零元素的个数，找到0时便停止。它不进行界限检查，所以在数组中至少有一个0。

例 13-4: ptr2/ptr2.c

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

```
int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int index;

int main()
{
    index = 0;
    while (array[index] != 0)
        ++index;

    printf("Number of elements before zero %d\n",
                  index);
    return (0);
}
```

例 13-5 是 13-4 使用指针后的改写版。

例 13-5: ptr3/ptr3.c

```
#include <stdio.h>

int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int *array_ptr;

int main()
{
    array_ptr = array;

    while ((*array_ptr) != 0)
        ++array_ptr;

    printf("Number of elements before zero %d\n",
                  array_ptr - array);
    return (0);
}
```

注意当我们想检查数组中的数据时，使用指针操作符(*)。这个操作符用于语句:

```
while ((*array_ptr) != 0)
```

想改变指针时，不使用其他操作符，例如，下行使指针增值，数据不变:

```
++array_ptr;
```

例13-4使用表达式(array[index] != 0)。这个表达式要求编译器产生一个指数运算，比简单的指针逆引用花的时间要长一些，即((*array_ptr) != 0)。该程序结尾的表达式array_ptr - array计算array_ptr在数组中的位置。

当把数组传递给一个程序时，C将自动将这个数组改为一个指针。事实上，如果在数组前放上&号，C将输出一个警告信息。例13-6表示出数组传递给子程序的不同方法。

例13-6: init-a/init-a.c

```
#define MAX 10  /* Size of the array */
/*******************************************************
 * init_array_1 --Zeroes out an array.                 *
 *                                                     *
 * Parameters                                          *
 *      data --The array to zero out.                  *
 *******************************************************/
void init_array_1(int data[])
{
    int  index;

    for (index = 0; index < MAX; ++index)
        data[index] = 0;
}

/*******************************************************
 * init_array_2 --Zeroes out an array.                 *
 *                                                     *
 * Parameters                                          *
 *      data_ptr --Pointer to array to zero.           *
 *******************************************************/
void init_array_2(int *data_ptr)
{
    int index;

    for (index = 0; index < MAX; ++index)
        *(data_ptr + index) = 0;
}
int main()
{
    int  array[MAX];
```

```
    void init_array_1();
    void init_array_2();

    /* one way of initializing the array */
    init_array_1(array);

    /* another way of initializing the array */
    init_array_1(&array[0]);

    /* works, but the compiler generates a warning */
    init_array_1(&array);

    /* Similar to the first method but  */
    /*    function is different */
    init_array_2(array);

    return (0);
}
```

如何不使用指针

本书的主要目标是教会读者创建清晰、可读、可维护的程序，不幸的是，不是所有的人都读过这本书，一些人还是认为应该使代码尽量简短。这种观念导致程序员在其他的语句里使用++或--运算符。例13-7列出了几个指针和增量运算符混用的例子。

例 13-7: 不好的指针用法

```
/* This program shows programming practices that should **NOT** be used */
/* Unfortunately, too many programmers use them */
int array[10];     /* An array for our data */
int main()
{
   int *data_ptr;    /* Pointer to the data */
   int value;        /* A data value */

   data_ptr = &array[0];/* Point to the first element */
   value = *data_ptr++;  /* Get element #0, data_ptr points to element #1 */
   value = *++data_ptr;  /* Get element #2, data_ptr points to element #2 */
   value = ++*data_ptr; /* Increment element #2, return its value */
                        /* Leave data_ptr alone */
```

为了理解每个语句的意思，必须仔细分析每一个表达式，以找出它隐含的意义。在我做程序维护时，我不想为隐晦的意思费尽心思，所以请不要这样编码，并把那些这样写程序的人枪毙掉。

图 13-8 剖析了这些语句。

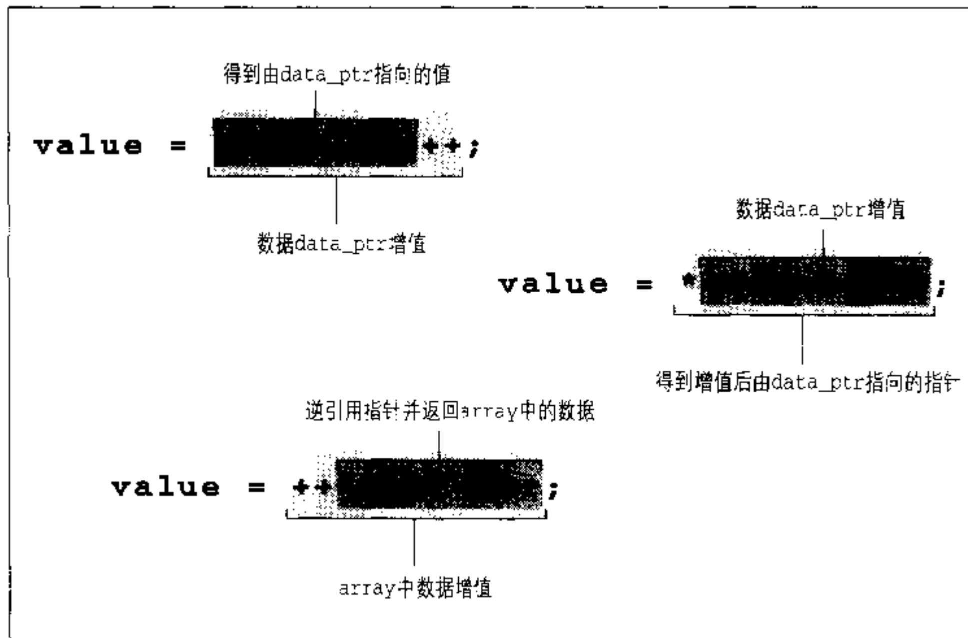


图 13-8 指针运算符剖析

本例有些极端，但它表明副作用很容易造成混淆。

例 13-8 是读者很有可能会遇到的代码，该程序从源串（p）拷贝一个串到目标串（q）。

例 13-8: 指针的隐晦用法

```
void copy_string(char *p, char *q)
{
    while (*p++ = *q++);
}
```

如果有时间的话，一个好的程序员是可以破译这些代码的，然而，只需稍微罗嗦那么一点，程序理解起来就容易得多，如例13-9。

例 13-9: 指针可读性用法

```
/********************************************************
 * copy_string --Copies one string to another.          *
 *                                                      *
 * Parameters                                           *
 *      dest --Where to put the string                  *
 *      source --Where to get it                        *
 ********************************************************/
void copy_string(char *dest, char *source)
{
    while (1) {
        *dest = *source;

        /* Exit if we copied the end of string */
        if (*dest == '\0')
            return;

        ++dest;
        ++source;
    }
}
```

用指针分隔字符串

假定有一个这样的串“Last/First”，现在想把它分成两个串，一个含有名字，另一个含有姓氏。这就需要用一个函数来找到名字中的斜杠位置，标准函数 strchr 能完成这项功能。本程序中，复制这个函数并看如何运行。

函数给串定义了一个指针（string_ptr）并用 find 作为函数的自变量，函数开始 while 循环直到找到要找的字符（或者由下列代码停止）。

```
while (*string_ptr != find) {
```

下面，测试一下串是否已运行完，本例中，指针（string_ptr）指向字符串结尾，如果在找到字符以前到达串结尾，则返回 NULL：

```
    if (*string_ptr == '\0')
        return (NULL);
```

如果还没有找到要找的字符并且还没到串的结尾，则把指针移向下一字符，再循环一遍：

```
    ++string_ptr;
}
```

主程序单行读取，去掉新行的字符，函数my_strchr被调用查找斜杠（/）的位置。

这样last_ptr指向姓的开始字符，first_ptr指向斜杠，然后可以用串结束符（NULL或\0）来替换斜杠（/）。现在last_ptr仅指向了名字，而first_ptr指向了一个空串。把first_ptr移向下一个字符，则first_ptr指向名字的开头。

这个分隔串的过程如图13-9所示。

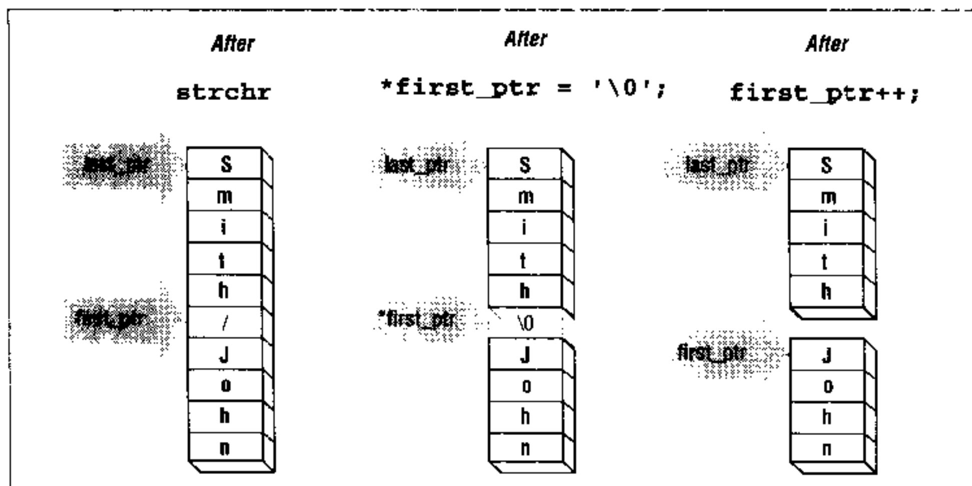


图13-9 分隔串

例13-10包含整个程序，示范了指针和字符数组怎样用于简单字串处理。

例13-10: split/split.c

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

```
#include <stdlib.h>

/********************************************************
 * my_strchr --Finds a character in a string.           *
 *      Duplicate of a standard library function,       *
 *      put here for illustrative purposes              *
 *                                                      *
 * Parameters                                           *
 *      string_ptr --String to look through.            *
 *      find --Character to find.                       *
 *                                                      *
 * Returns                                              *
 *      pointer to 1st occurrence of character          *
 *      in string or NULL for error.                    *
 ********************************************************/
char *my_strchr(char * string_ptr, char find)
{
    while (*string_ptr != find) {

       /* Check for end */

       if (*string_ptr == '\0')
           return (NULL);       /* not found */

        ++string_ptr;
    }
    return (string_ptr);        /* Found */
}

int main()
{
    char line[80];       /* The input line */
    char *first_ptr;     /* pointer to the first name */
    char *last_ptr;      /* pointer to the last name */

    fgets(line, sizeof(line), stdin);

    /* Get rid of trailing newline */
    line[strlen(line)-1] = '\0';

    last_ptr = line;    /* last name is at beginning of line */

    first_ptr = my_strchr(line, '/');       /* Find slash */
```

```
    /* Check for an error */
    if (first_ptr == NULL) {
        fprintf(stderr,
            "Error: Unable to find slash in %s\n", line);
        exit (8);
    }

    *first_ptr = '\0'    /* Zero out the slash */

    ++first_ptr;         /* Move to first character of name */

    printf("First:%s Last:%s\n", first_ptr, last_ptr);
    return (0);
}
```

问题 13-1：例 13-11 预期会显示：

```
Name: tmp1
```

但相反，得到的却是：

```
Name: !_@$#ds80
```

（你的结果可能还不同）为什么？

例 13-11: tmp-name/tmp-name.c

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

/********************************************************
 * tmp_name --Return a temporary filename.              *
 *                                                      *
 * Each time this function is called, a new name will   *
 * be returned.                                         *
 *                                                      *
 * Returns                                              *
 *      Pointer to the new filename.                    *
 ********************************************************/
char *tmp_name(void)
{
    char name[30];       /* The name we are generating */
    static int sequence = 0;    /* Sequence number for last digit */
```

```
    ++sequence; /* Move to the next filename */

    strcpy(name, "tmp");

    /* But in the sequence digit */
    name[3] = sequence + '0';

    /* End the string */
    name[4] = '\0';

    return(name);
}

int main()
{
    char *tmp_name(void);       /* Get name of temporary file */

    printf("Name: %s\n", tmp_name());
    return(0);
}
```

指针和结构

第十二章“高级类型”中，我们曾定义了一个邮件列表的结构：

```
struct mailing {
    char name[60];     /* last name, first name */
    char address1[60];/* two lines of street address */
    char address2[60];
    char city[40];
    char state[2];     /* two-character abbreviation */
    long int zip;      /* numeric zip code */
} list[MAX_ENTRIES];
```

邮件列表常常按名字和邮政编码排序，读者可以对输入内容进行排序，但是每个输入项有226个字节的内容，所以排序时要把许多数据移来移去。解决这个问题的一个办法是，定义一个指针数组，然后对指针进行排序：

```
/* Pointer to the data */
struct mailing *list_ptrs[MAX_ENTRIES];
```

```
int current;     /* current mailing list entry */

    for (current = 0; current = number_of_entries; ++current)
        list_ptrs[current] = &list[current];
    /* Sort list_ptrs by zip code */
```

现在可以不必移动226个字节的结构，只移动4个字节的指针，这样，排序的速度更快。假定读者有一个装满又重又大的箱子的仓库，读者需要快速地找到一个箱子的位置，一种办法是，把箱子按字母顺序排列。但是，这要费很大的力气移动箱子。读者也可以为每个位置指定一个号码，在索引卡上写下箱子名字和位置号码，然后按名字对卡片进行排序即可。

命令行参数

实际上main过程有两个形参，它们是argc和argv（注2）。

```
main(int argc, char *argv[])
{
```

（如果读者意识到形参是按字母顺序排列的，可很容易记住哪一个在前面。）

参数argc是命令行中参数的个数（包括程序名），数组argv包含实际的参数。例如，如果用下面的命令行运行程序args：

```
args this is a test
```

那么：

```
   argc =     5
argv[0] =    "args"
argv[1] =    "this"
argv[2] =    "is"
argv[3] =    "a"
```

注2：　实际上，它们可以以任何名字命名。但是在99.9%的程序中，它们被命名为argc和argv。当大多数的程序员碰到0.1%的情况时，他们多数会大声诅咒，然后将其改名为argc和argv。

```
argv[4] =    "test"
argv[5] =    NULL
```

注意: UNIX shell 在把命令行送给程序之前，扩充了通配符，如*、?和[]。详见 sh 或 csh 手册。

如果读者的程序连接了 WILDARG.OBJ 文件的话，Turbo C 和 Borland C++ 将扩充通配符。详见手册。

几乎所有的UNIX命令都使用标准命令行格式，这个标准也延伸到了其他的操作环境中，一个标准的UNIX命令形式为:

```
命令 选项 文件1  文件2  文件3.....
```

选项前面有一个连字符 (-)，而且通常是一个字符。例如，选项 -V 是打开一个特殊命令的详细模式，如果选项带有参数，则参数跟在字母的后面。例如，选项 -m1024 设置了最大的符号数为 1024，-ooutfile 设置输出文件名为 *outfile*。

现在让我们写一个能读取命令行自变量并运行正常的程序。这个程序使文件符合规定的格式并显示其内容，这个程序的一部分文档如下:

```
print_file [-v] [-llength] [-oname] [file1] [file2] ...
```

其中:

-v

设置 verbose 选项；打开程序执行期间的许多信息

-llength

设置每页大小为 length 行（缺省值为 66）

-oname

设置的输出文件为 name（缺省值为 print.out）

file1,file2,...

是要输出的文件列表，如果没指定文件，则输出文件 *print.in*。

while 循环将在这些命令行选项之间反复执行。真正的循环是：

```
while ((argc > 1) && (argv[1][0] == '-')) {
```

有一个自变量总是存在：即程序名，表达式（argc>1）检查额外的自变量，第一个将编为1，第一个自变量的第一个字符是 argv[1][0]。如果这个字符是个破折号，则将有一个选项。

循环结尾的代码是：

```
        --argc;
        ++argv;
    }
```

这就处理完一个自变量，自变量数减1，表示选项少了一个，指向第一个选项的指针增1，选项表向左移一个位置。（注意：第一次增值之后，argv[0]不再指向程序名。）

switch 语句用来解释选项，自变量的第0个字符是连字符（-），第1个字符是选项字符，所以可以用表达式：

```
switch (argv[1][1]) {
```

对选项进行解释。

选项 -v 没有自变量；它只设置一个标志。

选项 -o 取文件名，设置字符指针 out_file 指向串中的名字部分，而不是复制整个串。至此，有：

```
argv[1][0]    ='-'
argv[1][1]    ='o'
argv[1][2]    = first character of the filename
```

用下面的语句把 out_file 设置为指向串的指针：

```
out_file = &argv[1][2];
```

运算符&的地址用来得到输出文件名第一个字符的地址，这一过程是正确的，因为我们正把地址分配给名为out_file的字符指针。

选项-l取一个整数参数，库函数atoi用来把串转为一个整数。从前一个例子可知argv[1][2]是包含数字的串的第一个字符，这个串被传给atoi。

最后，从语法上分析所有的选项，并转到处理这些选项的循环部分，对每个文件自变量执行do_file函数。例13-12包含了print程序。

这是分析自变量列表的一种方法。使用**while**循环和**switch**语句简单易懂，但该方法有一定的局限性。自变量必须紧跟在选项后面。例如，-odata.out是可行的，但-o data.out将不可行。改进的分析方法使程序更友好，但这段代码仅可以用于简单的程序。

例13-12: print/print.c

```
[File: print/print.c]
/********************************************************
 * Program: Print                                       *
 *                                                      *
 * Purpose:                                             *
 *      Formats files for printing.                     *
 *                                                      *
 * Usage:                                               *
 *      print [options] file(s)                         *
 *                                                      *
 * Options:                                             *
 *      -v              Produces verbose messages.      *
 *      -o<file>        Sends output to a file          *
 *                      (default=print.out).            *
 *      -l<lines>       Sets the number of lines/page   *
 *                      (default=66).                   *
 ********************************************************/
#include <stdio.h>
#include <stdlib.h>

int verbose = 0;          /* verbose mode (default = false) */
char *out_file = "print.out";    /* output filename */
char *program_name;       /* name of the program (for errors) */
int line_max = 66;        /* number of lines per page */
```

```
/********************************************************
 * do_file   Dummy routine to handle a file.            *
 *                                                      *
 * Parameter                                            *
 *      name   Name of the file to print.               *
 ********************************************************/
void do_file(char *name)
{
    printf("Verbose %d Lines %d Input %s Output %s\n",
        verbose, line_max, name, out_file);
}
/********************************************************
 * usage   Tells the user how to use this program and   *
 *              exit.                                   *
 ********************************************************/
void usage(void)
{
    fprintf(stderr,"Usage is %s [options] [file-list]\n,
                                program_name);
    fprintf(stderr,"Options\n");
    fprintf(stderr,"  -v           verbose\n");
    fprintf(stderr,"  -l<number>  Number of lines\n");
    fprintf(stderr,"  -o<name>    Set output filename\n");
    exit (8);
}
int main(int argc, char *argv[])
{
    /* save the program name for future use */
    program_name = argv[0];

    /*
     * loop for each option
     *   Stop if we run out of arguments
     *   or we get an argument without a dash
     */
    while ((argc > 1) && (argv[1][0] == '-')) {
        /*
         * argv[1][1] is the actual option character
         */
        switch (argv[1][1]) {
            /*
             * -v verbose
             */
            case 'v':
```

```
            verbose = 1;
            break;
        /*
         * -o<name>  output file
         *    [0] is the dash
         *    [1] is the "o'
         *    [2] starts the name
         */
        case 'o':
            out_file = &argv[1][2];
            break;
        /*
         * -l<number> set max number of lines
         */
        case 'l':
            line_max = atoi(&argv[1][2]);
            break;
        default:
            fprintf(stderr,"Bad option %s\n", argv[1]);
            usage();
    }
    /*
     * move the argument list up one
     * move the count down one
     */
    ++argv;
    --argc;
}

/*
 * At this point, all the options have been processed.
 * Check to see if we have no files in the list.
 * If no files exist, we need to process just standard input stream.
 */
if (argc == 1) {
    do_file("print.in");
} else {
    while (argc > 1) {
      do_file(argv[1]);
      ++argv;
      --argc;
    }
}
return (0);
```

```
}
```

编程练习

练习 13-1: 编写一个程序，用指针把数组中的每个元素设置为零。

练习 13-2: 编写一个函数，它仅有一个串自变量，返回指向串中非空字符的指针。

答案

解答 13-1: 问题是变量name是个临时变量。进入函数时编译器为name分配空间，退出函数时释放空间。函数赋给name正确的值，并返回一个指向它的指针。但是，函数已经执行完了，所以name消失了。虽然有一个指针，但它的值是非法的。

解决方法是定义 name static。用这种方式定义，name将是一个永久变量，不会在函数结尾时消失。

问题 13-2: 修改函数后，试着使用两个文件名来执行函数，示例13-13应该输出：

```
Name: tmp1
Name: tmp2
```

但它没有。它输出的是什么？为什么？

例 13-13: tmp2/tmp2.c

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

/********************************************************
 * tmp_name   Returns a temporary filename.            *
 *                                                      *
 * Each time this function is called, a new name will  *
 * be returned.                                         *
 *                                                      *
```

```
 * Warning: There should be a warning here, but if we   *
 *      put it in, we would answer the question.        *
 *                                                      *
 * Returns                                              *
 *      Pointer to the new filename.                    *
 ********************************************************/
char *tmp_name(void)
{
    static char name[30];       /* The name we are generating */
    static int sequence = 0;    /* Sequence number for last digit */

    ++sequence; /* Move to the next filename */

    strcpy(name, "tmp");

    /* But in the squence digit */
    name[3] = sequence + '0';

    /* End the string */
    name[4] = '\0';

    return(name);
}

int main()
{
    char *tmp_name(void);       /* get name of temporary file */
    char *name1;                /* name of a temporary file */
    char *name2;                /* name of a temporary file */

    name1 = tmp_name();
    name2 = tmp_name();

    printf("Name1: %s\n", name1);
    printf("Name2: %s\n", name2);
    return(0);
}
```

解答 13-2：第一次调用 tmp_name 时将返回一个指向 name 的指针。此时只有一个 name。第二次调用 tmp_name时修改了name，并返回一个指向它的指针。所以你有了两个指针，它们指向同一个量：name。

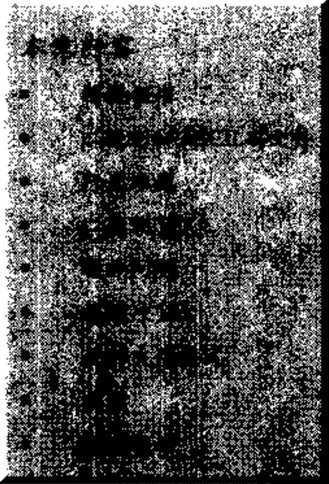
有几个库函数返回指向 **static** 串的指针。第二次调用这样的程序时将改变第一次的值，解决的办法是对值进行复制如下：

```
char name1[100];
char name2[100];
strcpy(name1, tmp_name());
strcpy(name2, tmp_name());
```

这个问题很好地表示了指针的基本意义：它不为数据创建任何新的空间，而仅指向在别处创建的数据。

这个问题也是函数设计得不好的典型例子，问题出在这个函数的使用有技巧。改进的设计会使代码使用风险较低，例如函数会使用附加参数：在串里给文件名发指令：

```
void tmp_name(char *name_to_return);
```



第十四章

文件输入/输出

我是所有旧时代的继承者，处在最古老的时光档案里

——丁尼生（英国诗人）

一个文件是相关数据的集合，C把文件看作是一系列的字节。文件大多在磁盘上，不过，像终端、打印机和磁带这样的设备也被认为是文件。

C函数库包含大量文件处理函数，文件函数使用的结构和函数定义存储在标准引用文件<stdio.h>中。在对文件做任何处理之前，必须把下行的语句放在程序开头：

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

文件变量的定义为：

```
FILE *file-variable;        /* comment */
```

例如：

```
#include <stdio.h>
FILE *in_file;  /* file containing the input data */
```

文件使用之前，必须用函数fopen打开。fopen返回一个指针到文件结构，fopen的格式是：

```
file-variable = fopen(name, mode);
```

其中:

file-variable

是一个文件变量，错误时返回 *NULL*。

name

是文件的实际名字（如 data.txt,temp.dat 等）。

mode

表示读文件或是写文件。模式（mode）是 "w" 表示写，"r" 表示读。加入标志“b”表示二进制文件。不写二进制标志表示一个 ASCII（文本）文件（见“二进制和 ASCII 文件”部分对 ASCII 和二进制文件的描述）。

标志可以合并，如“wb”是写二进制文件。

函数返回一个在随后的 I/O 操作中用到的文件标识值，如果出现 I/O 错误，返回 NULL 值:

```
FILE *in_file; /* File to read */

in_file = fopen ("input.txt", "r"); /* Open the input file */
if (in_file == NULL) { /* Test for error */
    fprintf (stderr, "Error: Unable to input file 'input.txt'\n");
    exit  (8);
}
```

函数 fclose 关闭文件，fclose 格式是:

```
status = fclose (file-variable);
```

或

```
fclose (file-variable);
```

如果 fclose 关闭成功，变量 *status* 为零，出错为非零；如果不在乎 status，也可用第二种形式关闭文件并去掉返回值。

C 提供三个预先打开的文件，见表 14-1。

表 14-1 标准文件

文件	描述
stdin	标准输入（为读打开）
stdout	标准输出（为写打开）
stderr	标准错误（为写打开）

函数fgetc从一个文件中读取单字符，如果文件里没有更多的数据存在，函数返回EOF值（EOF在*stdio.h*中定义）。注意fgetc返回一个整数，不返回字符，这种返回是必要的，因为EOF标志必须是非字符型值。

例14-1统计了*input.txt*文件中的字符数。

例14-1: copy/copy.c

```
[File: copy/copy.c]
#include <stdio.h>
const char FILE_NAME[] = "nput.txt";
#include <stdlib.h>

int main()
{
    int             count = 0;  /* number of characters seen */
    FILE           *in_file;    /* input file */

    /* character or EOF flag from input */
    int             ch;

    in_file = fopen(FILE_NAME, "r");
    if (in_file == NULL) {
        printf("Cannot open %s\n", FILE_NAME);
        exit(8);
    }

    while (1) {
        ch = fgetc(in_file);
        if (ch == EOF)
            break;
        ++count;
    }
    printf("Number of characters in %s is %d\n",
```

```
                     FILE_NAME, count);

        fclose(in_file);
        return (0);
    }
```

类似的函数 fputc，用来写单个字符，它的格式是:

```
    fputc(character, file);
```

函数 fgets 和 fputs 一次运行一行、fgets 调用的格式是:

```
    string_ptr = fgets(string, size, file);
```

其中:

string_ptr

如果读取成功就等于 *string*，文件结束或发现错误则为 *NULL*。

string

是函数放置串的字符型数组。

size

是字符型数组的大小。*fgets* 一直读取，直到得到是 *\n* 为止，或者读 size-1 个字符，然后加 *\0* 结束串。

如果size定义过大会出现问题。C通过使用 sizeof 运算符为确定适度的参数大小提供了便捷的方法。

sizeof 运算符把它的自变量字节返回给 size、例如:

```
    long int array[10];     /* (Each element contains 4 bytes) */
    char string[30];
```

则sizeof(串)是30，这个大小和长度不同，串长度可以是0-29个字符，sizeof 函数返回 string（或别的名称）的字节数。一个长整数占4字节，所以sizeof（数组）是40。

当读者使用fgets程序时，sizeof运算符特别有用，因为使用sizeof不用担心串有多大，即使有人改变了串的维数，也不用担心会发生什么。

例如：

```
char    string[100];
. . .
fgets(string, sizeof(string), in_file);
```

fputs和fgets相似，但fputs写串而不是读它。fputs函数的格式是：

```
string_ptr = fputs(string, file);
```

fputs的参数和fgets的类似，fputs不需要大小，因为它从串长度得到行的大小（一直写直到遇到'\0'。）

转换程序

到目前为止，我们仅讨论了字符和串的写入。本节将讨论一些更高级的I/O运算和转换。

为了向打印机或终端输出一个数，必须把它转换为字符，打印机只知道字符，不知道数字。例如，数567必须被转换为三个字符5, 6和7才能打印。

函数fprintf转换数据并把它写到一个文件中。函数fprintf的一般形式为：

```
count = fprintf(file, format, parameter-1, parameter-2, ...);
```

其中：

count

　　是发送的字符数或出错时的-1值。

format

　　描述如何显示自变量。

parameter-1,parameter-2,...

是要转换并发送的参量。

fprintf 有两个相关函数：printf 和 sprintf。printf () 在本书中常见到，等价于 fprintf 加一个自变量 stdout。Sprintf 和 fprintf 相似，不同的是 sprintf 是串。例如：

```
char string[40];          /* the filename */
int file_number = 0;      /* current file number for this segment */

sprintf(string, "file.%d", file_number);
++file_number;
out_file = fopen(string, "w");
```

Scanf 也有两个相关函数：fscanf 和 sscanf。fscanf 的格式是：

```
number = fscanf(file, format, &parameter-1, ...);
```

其中：

number

是成功转换的参量数，如果有输入但没转换，返回零。如果不存在数据，返回 **EOF**。

file

是打开的要读的文件。

format

描述要读的数据。

parameter-1

是要读的第一个参数。

sscanf 和 fscanf 相似，不同的是 sscanf 扫描一个串而不是文件。

scanf 对输入时的行尾字符非常敏感，用户必须键入额外的回车以避免 scanf 不能使用。

通过使用fgets从文件中读一行可以避免这个问题，然后用sscanf来分析它。fgets差不多总能准确无误地获取一个单行。

例14-2显示了一个程序，该程序从标准输入（键盘）读取两个参数，然后根据找到的实际输入的数据显示一条消息。

例14-2: 使用sscanf返回值

```
char line[100];     /* Line from the keyboard */
int count, total;   /* Number of entries & total value */
int scan_count;     /* Number of parameters scanned */

fgets(line, sizeof(line), stdin);
scan_count = sscanf(line,"%d %d", &count, &total);

switch (scan_count) {
    case EOF:
    case 0:
        printf("Didn't find any number\n");
        break;
    case 1:
        printf("Found 'count'(%d), but not 'total'/n", count);
        break;
    case 2:
        printf("Found both 'count'(%d) and 'total'(%d)\n", count, total);
        break;
    default:
        printf("This should not be possible\n");
        break;
}
```

问题14-1：不管给例14-3取什么文件名，程序都找不到它，为什么？

例14-3: fun-file/fun-file.c

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

int main()
{
    char            name[100];  /* name of the file to use  */
    FILE           *in_file;    /* file for input */
```

```
    printf("Name? ");
    fgets(name, sizeof(name), stdin);

    in_file = fopen(name, "r");
    if (in_file == NULL) {
        fprintf(stderr, "Could not open file\n");
        exit(8);
    }
    printf("File found\n");
    fclose(in_file);
    return(0);
}
```

二进制和ASCII码文件

到目前为止一直用的是ASCII码文件，ASCII代表American Standard Code For Information Interchange（美国信息交换标准代码），有一套95个可显示字符和33个控制码。ASCII文件是可读性文本，在写程序时，*prog.c*文件就是ASCII码。

终端、键盘和打印机处理字符数据。当读者想在屏幕上写一个1234这样的数时，必须先把它转换为4个字符（'1'、'2'、'3'、'4'）来写。同样地，当读者从键盘上读一个数时，数据必须从字符转换为整数，这由sscanf程序来完成。

ASCII字符'0'的值是48、'1'值为49，依此类推。当读者想把一个数字从ASCII码转换为整数时，必须减去48这个数：

```
int integer;
char ch;

ch = '5';
integer = ch - 48;
printf("Integer %d\n", integer);
```

不用记住'0'的值是48，你可以仅仅减'0'：

```
integer = ch - '0';
```

计算机使用的是二进制数据。当从ASCII码文件中读数时，程序必须通过一个转换程序如sscanf来处理字符数据，这需要额外做一些工作。二进制文件不需要转换，而且在一般情况下，它们比ASCII码文件占用更少的空间。缺点是，二进制文件不能直接输出到终端或是打印机上。（如果你看见打印机输出一段很长的内容，内容的开始很像是“!E#（@$%@^Aa^AA^^JHC%^X”，那么你应该意识到你打印了一个二进制文件。）

ASCII码文件是可以移植的（对大多数情况而言），它们可以不费什么周折地从一台机器移到另一台机器，几乎所有的二进制文件肯定不能移植。除非你是一位编程专家，否则没有办法移植一个二进制文件。

你应该使用哪种文件类型呢？大多数情况下，使用ASCII。如果读者只有少量的数据，那么转换时间将不足以影响到程序的性能。（谁会在乎是花费了0.5秒还是0.3秒呢?）使用ASCII码文件时，数据的修改也容易。

只有当你的数据量非常大，足以影响到所占空间及程序性能的时候，才使用二进制格式的文件。

行尾难题

在黑暗的BC（Before Computers——计算机前）年代，有一种名叫Teletype Model 33的魔术般的设备，这个令人吃惊的机器包含一个由马达和转子组成的移位寄存器，还有一个仅由杆杠和弹簧组成的键盘ROM。它包含一个键盘、一台打印机和一台纸带阅读/打孔机。它可以在电话线上通过调制解调器传输信息，速度是每秒钟10个字符。

但电传打字电报机有一个问题，它要用2/10秒的时间把打印头从右侧移到左侧，2/10秒的时间可以传输2个字符。如果在打印头移动的过程中，又来了一个字符，则这个字符将丢失。

电传打字电报机的研制人员通过在行尾加两个符号解决了这个问题。这两个符号是：<回车>把打印头定位在左边界，<换行>把纸向下移一行。

计算机发展早期，一些设计人员认为在每一行的结尾使用两个字符太浪费存储空间（那个时候，存储器很贵）。有的只在行尾放一个<换行>，有的选择了<回车>，有些固守陈规的人依然使用两个连续符号。

UNIX在行尾使用<换行>，新行字符\n代码是0xA（LF或是<换行>）。MS-DOS/Windows使用两个字符：<换行><回车>。Apple使用<回车>。

MS-DOS/Windows编译器设计人员有一个问题，即原有的C程序认为换行就是<换行>符。该怎么处理这个问题呢？解决的办法是给I/O库增加代码，从ASCII输入文件中删去<回车>，并在输出时把<换行>改为<换行><回车>。

在MS-DOS/Windows中，文件以ASCII码格式打开，还是以二进制代码格式打开是不同的。标志`b`用来表示二进制文件：

```
/* open ASCII file for reading */
ascii_file = fopen("name", "r");

/* open binary file for reading */
binary_file = fopen("name", "rb");
```

如果在MS-DOS/Windows下打开一个包含文本的二进制文件，须处理程序中的回车。如果你以ASCII码格式打开它，回车将在读程序时自动去掉。

问题14-2： `fputc`指令可以用来写出二进制文件的一个字节。例14-4在文件*test.out*中写出0-127个数字，在UNIX上运行良好，创建了一个128字节长的文件；然而在MS-DOS/Windows下，文件包含129个字节。为什么？

例14-4: wbin/wbin.c

```
[File: wbin/wbin.c]
#include <stdio.h>
#include <stdlib.h>
#ifndef __MSDOS__
#include <unistd.h>
#endif __MSDOS__

int main()
{
```

```
    int cur_char;    /* current character to write */
    FILE *out_file; /* output file */

    out_file = fopen("test.out", "w");
    if (out_file == NULL) {
        fprintf(stderr,"Cannot open output file\n");
        exit (8);
    }

    for (cur_char = 0; cur_char < 128; ++cur_char) {
        fputc(cur_char, out_file);
    }
    fclose(out_file);
    return (0);
}
```

提示: 下面是 MS-DOS/Windows 文件按十六进制格式的输出内容:

```
000:0001 0203 0405 0607 0809 0d0a 0b0c 0d0e
010:0f10 1112 1314 1516 1718 191a 1b1c 1d1e
020:1f20 2122 2324 2526 2728 292a 2b2c 2d2e
030:2f30 3132 3334 3536 3738 393a 3b3c 3d3e
040:3f40 4142 4344 4546 4748 494a 4b4c 4d4e
050:4f50 5152 5354 5556 5758 595a 5b5c 5d5e
060:5f60 6162 6364 6566 6768 696a 6b6c 6d6e
070:6f70 7172 7374 7576 7778 797a 7b7c 7d7e
080:7f
```

C库函数自动整理ASCII码文件，对这一点UNIX程序员不必过分担心。在UNIX中，一个文件就是一个文件，ASCII码文件与二进制文件没有区别。事实上，如果读者愿意的话，可以写一个半是ASCII半是二进制码的文件。

二进制 I/O

二进制I/O由两个指令完成：fread 和 fwrite。fread 的语法是：

```
read_size = fread(data_ptr, 1, size, file);
```

其中：

read_size

是读入数据的大小，如果这个值比 size 小，表明遇到了文件结尾或出错。

data_ptr

是读取数据的指针，如果数据是字符以外的其他类型该指针必须用强制类型转换设成字符型 (char *)。

size

是读取的字节数目。

file

是输入的文件。

例如：

```
struct {
        int     width;
        int     height;
} rectangle;
int read_size;

read_size = fread((char *)&rectangle, 1, sizeof(rectangle), in_file);
if (read_size != sizeof(rectangle)) {
        fprintf(stderr,"Unable to read rectangle\n");
        exit (8);
}
```

本例中，读入结构rectangle。& 运算符把rectangle变为一个指针。“(char *)” 转换是必需的，因为read需要一个字符数组。sizeof 运算符用来判定要读入多少个字节，并检查 read 是否成功地读完。

fwrite 和 fread 有相似的调用序列：

```
write_size = fwrite(data_ptr, 1, size, file);
```

注意：为了使编程更简单容易，我们总是使用 1 作为fread 和fwrite的第二个参数。实际上，设计 fread 和 fwrite 的目的是读目标数组。第二个变量是目标大小，第三个变量是目标数组的数目，这些函数的详细信息见 C 参考手册。

缓冲问题

缓冲I/O不是立即把数据写入文件。而是将数据保留在一个缓冲区中，直到缓冲区中的数据量足够一次写入，或是遇到一次清空缓冲操作为止。下面的程序设计将在每完成一个阶段后就输出一条信息。

```
printf("Starting");

do_step_1();
printf("Step 1 complete");

do_step_2();
printf("Step 2 complete");

do_step_3();
printf("Step 3 complete\n");
```

每步骤完成时 printf 函数不是写信息，而是把它们放进缓冲区中，只有当程序执行完毕时才清空缓冲区，同时所有的信息立即全部输出。

指令 fflush 会迫使缓冲区内容输出，上面例子的正确写法是：

```
printf("Starting");
fflush(stdout);

do_step_1();
printf("Step 1 complete");
fflush(stdout);

do_step_2();
printf("Step 2 complete");
fflush(stdout);

do_step_3();
printf("Step 3 complete\n");
fflush(stdout);
```

非缓冲 I/O

在缓冲 I/O 中，数据先缓存，然后再送到文件中；在非缓冲 I/O 中，数据立即送到文件中。

如果你把很多的纸夹掉到了地板上，你可以用缓冲或非缓冲的方式拾起它们。在缓冲方式下，你用右手拾起一个夹子，然后放到左手上。重复这样做，直到左手满了，然后把手上的纸夹子放到桌上的盒子里。在非缓冲方式下，你每拾一个夹子，就把它放到盒子里。不需要先放到左手中。

大多数情况下，应使用缓冲 I/O 而不是非缓冲 I/O。在非缓冲 I/O 中，每一次的读写操作需要一次系统调用。对操作系统的任何调用都有消耗，缓冲 I/O 使得这些调用次数最少。

仅当读写大量的二进制数据，或是需要直接控制一个设备或文件时，才应该使用非缓冲 I/O。

再回到纸夹例子中，如果你拾的是小的物体，就像纸夹这样的，你可能用左手当缓冲区。但是如果你拾的是炮弹（它很大），就不需要用缓冲区。

系统调用 open 来打开一个非缓冲文件，该调用的宏定义因系统的不同而有所不同。因为读者用的可能是 UNIX 也可能是 MS-DOS/Windows，所以要用条件编译（**#ifdef/#endif**）才能得到正确的文件。

```
#ifndef __MSDOS__        /* if we are not MS-DOS */
#define __UNIX__         /* then we are UNIX */
#endif __MSDOS__

#ifdef __UNIX__
#include <sys/types.h>  /* file defines for UNIX filesystem */
#include <sys/stat.h>
#include <fcntl.h>
#endif __UNIX__

#ifdef __MSDOS__
#include <stdlib.h>
#include <fcntl.h>      /* file defines for DOS filesystem */
```

```
#include <sys\stat.h>
#include <io.h>
#endif __MSDOS__

int      file_descriptor;
file_descriptor = open(name, flags);          /* existing file */
file_descriptor = open(name, flags, mode);   /*new file */
```

其中：

file_descriptor

用来识别读、写和关闭调用的文件。在它小于0时出错。

name

文件名字。

flags

定义在头文件*fcntl.h*中。标志在表14-2中有描述。

mode

文件的保护模式，一般对多数文件来说该值是0644。

表14-2　打开标志

标志	意义
O_RDONLY	打开只读文件
O_WRONLY	打开只写文件
O_RDWR	打开读写文件
O_APPEND	在文件尾添加新数据
O_CREAT	创建文件（指定了标志时，还要指定模式）
O_TRUNC	如果文件已存在，则把它的长度截为0
O_EXCL	如果文件已存在，则失败
O_BINARY	以二进制方式打开（仅在MS-DOS/Windows下）
O_TXT	以文本模式打开（仅在MS-DOS/Windows下）

例如，以文本方式打开一个用于输入的已存在的文件*data.txt*，语句如下：

```
data_fd = open("data.txt", O_RDONLY);
```

下面的例子显示如何创建一个只写文件 *output.dat*:

```
out_fd = open("output.dat", O_CREAT|O_WRONLY, 0644);
```

注意，你可以用 *or* (|) 运算符把标志连接起来，这是合并多个标志的快速简单的方法。当程序第一次运行时，系统已经打开了三个文件。这三个文件列在表 14-3 中。

表 14-3 标准非缓冲文件

文件号	符号名字	描述
0	STDIN_FILENO	标准输入
1	STDOUT_FILENO	标准输出
2	STDERR_FILENO	标准错误

符号名称定义在头文件 *unistd.h* 中，这些是 C 相对较新的部分且很少应用。(应该使用符号名字，但多数人不这样使用。)

read 调用的格式是:

```
read_size = read(file_descriptor, buffer, size);
```

其中:

read_size

是读取的字节数，零表示文件尾，负数表示一个错误

file_descriptor

是一个已打开的文件的描述符

buffer

指向要读入的数据的指针

size

是要读入数据的大小

write 调用的格式是:

```
write_size = write(file_descriptor, buffer, size);
```

其中:

write_size

是写入的字节数，负数表示一个错误。

file_descriptor

是一个已打开的文件的描述符。

buffer

指向数据要写入的位置的指针。

size

是要写入的数据大小。

最后，关闭文件的 close 调用为:

```
flag = close(file_descriptor)
```

其中:

flag

为 0 表示成功，负数表示错误。

file_descriptor

是一个已打开的文件的描述符。

例 14-5 复制一个文件，因为缓冲区较大，所以使用了非缓冲 I/O。没有必要用缓冲 I/O 把 1K 的数据读入一个缓冲区中，然后再把它送到 16K 缓冲区中。

例 14-5: copy2/copy2.c

```
[File: copy2/copy2.c]
/****************************************
 * copy --Copies one file to another.   *
 *                                      *
```

```
 * Usage                                  *
 *        copy <from> <to>                *
 *                                        *
 * <from> --The file to copy from.        *
 * <to>   --The file to copy into.        *
 ******************************************/
#include <stdio.h>
#ifndef __MSDOS              /* if we are not MS-DOS */
#define __UNIX               /* then we are UNIX */
#endif /* __MSDOS__ */

#include <stdlib.h>

#ifdef __UNIX__
#include <sys/types.h>  /* file defines for UNIX filesystem */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#endif /* __UNIX__ */

#ifdef __MSDOS__
#include <fcntl.h>         /* file defines for DOS filesystem */
#include <sys\stat.h>
#include <io.h>
#endif __MSDOS__

#ifndef O_BINARY
#define O_BINARY 0         /* Define the flag if not defined yet */
#endif /* O_BINARY */

#define BUFFER_SIZE (16 * 1024) /* use 16K buffers */

int main(int argc, char *argv[])
{
    char  buffer[BUFFER_SIZE];  /* buffer for data */
    int   in_file;              /* input file descriptor */
    int   out_file;             /* output file descriptor */
    int   read_size;            /* number of bytes on last read */

    if (argc != 3) {
        fprintf(stderr, "Error:Wrong number of arguments\n");
        fprintf(stderr, "Usage is: copy <from> <to>\n");
        exit(8);
    }
```

```
    in_file = open(argv[1], O_RDONLY|O_BINARY);
    if (in_file < 0) {
        fprintf("rror:Unable to open %s\n", argv[1]);
        exit(8);
    }
    out_file = open(argv[2], O_WRONLY|O_TRUNC|O_CREAT|O_BINARY, 0666);
    if (out_file < 0) {
        fprintf("Error:Unable to open %s\n", argv[2]);
        exit(8);
    }
    while (1) {
        read_size = read(in_file, buffer, sizeof(buffer));

        if (read_size == 0)
            break;                 /* end of file */

        if (read_size < 0) {
            fprintf(stderr, "Error:Read error\n");
            exit(8);
        }
        write(out_file, buffer, (unsigned int) read_size);
    }
    close(in_file);
    close(out_file);
    return (0);
}
```

问题 14-3：为什么在例 14-5 中如果不能打开输入文件，却不能输出错误信息？

在这个程序中，要注意几个问题，首先，缓冲区大小定义为一个常量，所以很容易修改。程序员不用死记 16K 就是 16384，只需使用表达式（16*1024）即可，这样更明确地表示这是 16K。

如果用户使用程序的方法不正确，会得到一条错误信息。为了帮助用户能够正确地使用它，信息中包含了使用程序的正确方法。

最后一次读数可能读不满整个缓冲区，这就是使用*read_size*的原因，为的是判定要写入的字节数。

设计文件格式

假定读者正在设计一个绘图程序，在图形配置文件中定义了高、宽、范围和缩放比例，要求读者写一个对用户十分友好的程序，该程序询问操作人员有关的问题，并据此写一个配置文件。这样，用户就不一定必须学习使用文本编辑器来写配置文件。应该怎样设计这个配置文件呢？

一种方法是：

```
height (in inches)
width (in inches)
x lower limit
x upper limit
y lower limit
y upper limit
x scale
y scale
```

典型的配置文件可能如下所示：

```
10.0
7.0
0
100
30
300
0.5
2.0
```

该文件包含了所有的数据，但在读它的时候，读者不能具体地识别出它们，例如不知道哪个值是y的下限。解决的办法是给文件加注释，也就是说，让配置程序不仅写出数据，而且还写出描述数据的字符串。

```
10.0  height (in inches)
7.0   width (in inches)
0     x lower limit
100   x upper limit
30    y lower limit
300   y upper limit
0.5   x scale
```

```
2.0    y scale
```

现在这个文件对用户来说具有可读性了。但是假设一个用户执行这个绘图程序，并键入了错误的文件名，程序得到的却是今天中午的午餐菜谱，而不是绘图配置文件。当程序试图建立一个图形时可能会弄得非常混乱，因为图形的尺寸不是“BLT on white”，而是“Meatloaf and gravy”。

结果会一塌糊涂。应该有办法识别出一个绘图配置文件，一种办法是在文件的第一行写上“Plot Configuration File”。然后，当某人给程序输入错误的文件名时，程序将输出错误信息。

这样做会解决文件错误问题，但是当要求读者增强程序的功能，绘制额外的对数图时该怎么办?简单的办法就是在配置文件中添加另外的行，但是所有的旧文件怎么办呢?没有理由让每个人都把它们扔掉。最好的办法是（从用户的角度看）认可原有的文件格式。读者可以在文件中加入版本号，这样就简单了。

典型的文件现在列示如下：

```
Plot    Configuration File V1.0
log     Logarithmic or Normal plot
10.0    height (in inches)
7.0     width (in inches)
0       x lower limit
100     x upper limit
30      y lower limit
300     y upper limit
0.5     x scale
2.0     y scale
```

在二进制文件中，在文件的前4个字节放入一个标识号是常用的办法，这个标识号称为魔术号（magic number）。魔术号应该与各类文件有所不同。

选择魔术号的一种方法是取程序的前4个字母开头（例如*list*），把它们转换为十六进制数：0x6c607374，然后再加上0x80808080作为魔术号：0xECF0F3F4

这样产生的魔术号可能是唯一的。每个字节的高位设置为1，以保证这些字节不在ASCII码范围内，从而避免在ASCII码文件和二进制文件中引起混淆。

当从包含许多不同类型结构的二进制文件中进行读写操作时，很容易弄错。例如，当你想要一个尺寸结构的时候，可能读入了一个名字结构。通常不能马上发现这个错误，要到以后程序执行才能发现。为了尽早发现这个错误，程序员可以在每个结构的开头加入魔术号。这样，如果程序读入名字结构但魔术号不对时，它就知道出了错。

结构的魔术号不需要设置每个字节的高位，四个ASCII字符做魔术号将使得从出错文件中抽取开始的结构变得更简单。

答案

解答14-1： 问题是 `fgets` 得到了包括换行字符（`\n`）在内的整行，如果你有一个文件名*sam*，程序读取的就是*sam\n*，并寻找一个叫这个名字的文件，因为没有这个文件，程序报告出错。

修改方法是从名字中去掉换行符：

```
name[strlen(name)-1] = '\0';       /* get rid of last character */
```

本例中错误信息设计得不好，实际上我们没有打开文件，但程序员可以提供给用户更多的信息。我们是为输入或输出而在试着打开文件吗?我们要打开的文件名是什么?我们甚至不知道得到的信息是否是一个错误、一个警告信息或者只是正常操作的一部分。一个改进的错误信息是：

```
fprintf(stderr,"Error:Unable to open %s for input\n", name);
```

注意这条信息也能帮助我们发现程序错误，当我们键入 `sam`，错误将是：

```
Error:Unable to open sam
for input
```

这个信息清楚地告诉我们正在打开一个含有换行符的文件名字。

解答14-2： 问题是在写ASCII文件，但想要的是二进制文件。在UNIX中，ASCII和二进制一样，所以程序运行良好。在MS-DOS/Windows中，行尾导致了问题。

当在文件中写一个新行符（0x0a）时，一个回车（0x0D）被加了进去。（记住MS-DOS/Windows 中的行尾是<回车><换行>或者0x0d，0x0a。）正是因为这种编辑，在输出文件中我们得到了一个多余的回车（0x0d）。

为了写二进制数据（不输出编辑），需要用二进制选项打开文件：

```
out_file = fopen("test.out", "wb");
```

解答 14-3：问题出在 `fprintf` 调用。`fprintf` 的第一个参数应该是一个文件；相反它是一个格式串。程序以为用的是文件实际却是格式串，导致出了错。

编程练习

练习 14-1：写一个程序，读取文件，然后统计其中的行数。

练习 14-2：写一个程序，复制文件，将其中的水平制表符转换为多个空格。

练习 14-3：写一个程序，它读入一个含有一串数的文件，把其中能被3整除的数写入一个文件中，其余的数写入另一个文件中。

练习 14-4：编写一个程序，它读入一个含有一串数的ASCII文件，并把同样的数写入一个二进制文件中，编写一个程序执行与上述相反的操作，以检验前一个程序的正确性。

练习 14-5：编写一个程序，它复制一个文件，并删除其中高位为1的所有字符（`((ch & 0x80) != 0)`）。

练习 14-6：设计一种文件模式，它存储人名、地址和其他信息。编写一个程序读入这个文件，并产生一组邮件表。

第十五章

调试和优化

本章内容

- 调试
- 交互调试器
- 调试一个二分查找程序
- 实时运行错误
- 公开声明调试方法
- 优化
- 答案
- 编程练习

我们树立下血的榜样、教会别人杀人、
结果反而自己被人所杀
——莎士比亚，用于调试
[《麦克白》，第一幕，第七场]

调试

程序最难的部分不是设计和书写，而是调试阶段。在这个阶段中，要查明程序为何能运转起来（而不是想当然的运转方式）。

为根除错误，你需做两件事：重现错误的方法和让你定位错误并修正程序的信息。

有时，发现一个错误很容易，你自己就可以发现错误，测试部门拟定一个清晰而容易的测试计划，它可以显示程序中的错误，或者总是不对程序输出。

有时，特别是在交互程序中，重复出现的错误占问题的90%。当处理由某一领域的用户送交的错误报告中的问题时，更是如此。用户打来的电话可能是这样的：

用户：

你给我的数据库程序坏了。

程序员：

出了什么错？

用户：

有时，在我排序时，它排的顺序是错误的。

程序员:

你用的是什么命令?

用户:

排序命令。

程序员(耐心地):

告诉我你具体键入的是什么,一个键一个键地说,直到出错为止。

用户:

我记不清具体的了,我做了很多次排序。

程序员:

如果我来的话,你能不能再现这样的错误?

用户:

当然。

两分钟后,程序员来到了用户的办公室,说出了关键的话:"做给我看"。用户键入命令后程序运转正常,不管用户让它做什么。

程序员没有办法,只得回到了自己的办公室,但是发现了来自用户的一条消息:"你走后五分钟程序就出错了。"

例15-1是一个简短的数据库查询程序。它要求用户输入一个名字,并在一个名字编码列表中查找该名字。虽然这个程序很简单,但它的结构与更大更复杂的交互程序毫无两样。

例 15-1: base/base.c

```
/********************************************************
 * Database --A very simple database program to         *
 *              look up names in a hardcoded list.      *
 *                                                      *
 * Usage:                                               *
 *      database                                        *
 *              Program will ask you for a name.        *
 *              Enter the name; it will tell you if     *
 *              the name is in the list.                *
```

```
 *                                                           *
 *              A blank name terminates the program.         *
 *************************************************************/
#define STRING_LENGTH 80          /* Length of typical string */
#include <stdio.h>
#include <string.h>

int main ()
{
    char name[STRING_LENGTH];    /* a name to lookup */
    int lookup (char const *const name); /* lookup a name */

    while (1) {
        printf ("Enter name: ");
        fgets (name, sizeof (name) , stdin);

        /* Check for blank name  */
        /* (remember 1 character for newline) */
        if  (strlen(name)  <= 1)
            break;

        /* Get rid of newline */
        name[strlen (name) -1] = '\0';

        if  (lookup (name))
            printf ("%s is in the list\n", name);
        else
            printf ("%s is not in the list\n", name);
    }
    return  (0);
}
/************************************************************
 * lookup --Looks up a name in a list.                       *
 *                                                           *
 * Parameters                                                *
 *      name --Name to look up.                              *
 *                                                           *
 * Returns                                                   *
 *      1 --Name in the list.                                *
 *      0 --Name not in the list.                            *
 *************************************************************/
int lookup (char const *const name)
{
    /* List of people in the database */
```

```
    /* Note: Last name is a NULL for end of list */
    static char *list[] = {
        "John",
        "Jim",
        "Jane",
        "Clyde",
        NULL
    };

    int index;          /* index into list */

    for (index = 0; list[index] != NULL; ++index) {
        if (strcmp(list[index], name) == 0)
            return (1);
    }
    return (0);
}
```

这个程序的典型执行过程是：

```
Enter name: Sam
Sam is not in the list
Enter name: John
John is in the list
Enter name:
```

当我们交付这个程序时，用户立即开始报怨那个奇怪的问题。但当程序员在现场时，它却消失了。如果有一个小精灵站在用户的肩膀上，记下用户键入的所有内容不是很好吗？不幸的是，这个小精灵是不存在的；但是，我们可以修改这个程序，让它产生一个保存文件，文件中含有用户键入的所有按键。

程序中使用下面的语句读入用户键入的数据：

```
fgets(name, sizeof(name), stdin);
```

现在写一个新的程序：extended_fgets，用它来代替fgets函数。这个新函数不仅得到一行数据，还把用户的响应保存到一个保存文件中。例15-2修订了15-1并加进了extended_fgets

例 15-2: xgets/xgets.c

```
#include <stdio.h>
/*
 * The main program opens this file if -S is on
 * the command line.
 */
FILE *save_file = NULL;
/*********************************************************
 * extended_fgets -- Gets a line from the input file     *
 *              and records it in a save file if needed. *
 *                                                       *
 * Parameters                                            *
 *      line -- The line to read.                        *
 *      size -- sizeof(line) -- maximum number of        *
 *                      characters to read.              *
 *      file -- File to read data from                   *
 *              (normally stdin).                        *
 *                                                       *
 * Returns                                               *
 *      NULL -- error or end-of-file in read             *
 *      otherwise line (just like fgets).                *
 *********************************************************/
char *extended_fgets(char *line, int size, FILE *file)
{

    char *result;               /* result of fgets */
    result = fgets(line, size, file);

    /* Did someone ask for a save file?? */
    if (save_file != NULL)
        fputs(line, save_file); /* Save line in file */

    return (result);
}
```

还应修改主程序，使它能处理新的选项："-S*file*"，这个选项用来指定一个保存文件（大写字母用于调试和其他不常用的选项）。新的主程序如例 15-3。

例 15-3: base2/base2.c

```
[File: base2/base2.c]
/*********************************************************
 * Database -- A very simple database program to         *
```

```
 *             look up names in a hardcoded list.        *
 *                                                       *
 * Usage:                                                *
 *      database [-S<file>]                              *
 *                                                       *
 *      -S<file>        Specifies save file for          *
 *                      debugging purposes               *
 *                                                       *
 *             Program will ask you for a name.          *
 *             Enter the name; program will tell you if  *
 *             it is in the list.                        *
 *                                                       *
 *             A blank name terminates the program.      *
 *********************************************************/
#include <stdio.h>
#include <stdlib.h>

FILE *save_file = NULL; /* Save file if any */
char *extended_fgets (char *, int, FILE *);

int main (int argc, char *argv[])
{
    char name[80];       /* a name to lookup */
    char *save_file_name; /* Name of the save file */

    int lookup (char const *const name); /* lookup a name */

    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            /* -S<file>  Specify a save file */
            case 'S':
                save_file_name = &argv[1][2];
                save_file = fopen (save_file_name, "w");
                if (save_file == NULL)
                    fprintf (stderr,
                        "Warning:Unable to open save file %s\n",
                        save_file_name);
                break;
            default:
                fprintf (stderr,"Bad option: %s\n", argv[1]);
                exit (8);
        }
        --argc;
        ++argv;
```

```
    }

    while (1) {
        printf("Enter name: ");
        extended_fgets(name, sizeof(name), stdin);

        /* ... Rest of program ... */
    }
}
```

现在我们有了用户键入的完整记录。看看他的输入，将发现他键入了如下的内容

```
Sam
 John
```

第二个名字的开头是一个空格。虽然"John"在列表中，但是"<空格>John"并没在列表中。这样我们发现是因为输入而导致出错。但是，更复杂的程序可能有更复杂的输入。调试时，我们可以键入所有的信息；或者，我们可以给extended_fgets添加其他的功能，给它增加一个回放文件。当使用回放文件时，就不需要在键盘上键入所有的输入，而是从文件中读出相应的内容。例15-4包含修订后的extended_fgets。

例 15-4: xgets2/xgets2.c

```
#include <stdio.h>
FILE *save_file = NULL;          /* Save input in this file */
FILE *playback_file = NULL;      /* Playback data from this file */
/*********************************************************
 * extended_fgets - Gets a line from the input file      *
 *              and records it in a save file if needed. *
 *                                                       *
 * Parameters                                            *
 *      line --The line to read.                         *
 *      size --sizeof(line) --maximum number of          *
 *                  characters to read.                  *
 *      file --File to read data from                    *
 *              (normally stdin).                        *
 *                                                       *
 * Returns                                               *
 *      NULL --error or end-of-file in read              *
 *      otherwise line (just like fgets).                *
 *********************************************************/
```

```
char *extended_fgets (char *line, int size, FILE *file)
{
    extern FILE *save_file;      /* file to save strings in */
    extern FILE *playback_file; /* file for alternate input */

    char *result;               /* result of fgets */

    if (playback_file != NULL) {
        result = fgets (line, size, playback_file);
        /* echo the input to the standard out so the user sees it */
        fputs (line, stdout);
    } else
        /* Get the line normally */
        result = fgets (line, size, file);

    /* Did someone ask for a save file? */
    if (save_file != NULL)
        fputs (line, save_file); /* Save the line in a file */

    return (result);
}
```

给命令行增加一个新回放选项：-P*file*。这个选项允许自动键入引起错误的命令。现在的主函数如示例 15-5 所示。

例 15-5: base3/base3.c

```
/*******************************************************
 * Database --A very simple database program to        *
 *              look up names in a hardcoded list.     *
 *                                                     *
 * Usage:                                              *
 *      database [-S<file>] [-P<file>]                 *
 *                                                     *
 *      -S<file>       Specifies save file for         *
 *                     debugging purposes.             *
 *                                                     *
 *      -P<file>       Specifies playback file for     *
 *                     debugging or demonstration.     *
 *                                                     *
 *                                                     *
 *             Program will ask you for a name.        *
 *             Enter the name; the program will tell   *
 *             you if it is in the list.               *
```

```
 *                                                         *
 *           A blank name terminates the program.          *
 ***********************************************************/
#include <stdio.h>
#include <stdlib.h>
FILE *save_file = NULL; /* Save file if any */
FILE *playback_file = NULL;      /* Playback file if any */
char *extended_fgets (char *, int, FILE *);
int main (int argc, char *argv[])
{
    char name[80];        /* a Name to look up */
    char *save_file_name; /* Name of the save file */
    char *playback_file_name; /* Name of the playback file */

    int lookup (char const *const name); /* lookup a name */

    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            /* -S<file>  Specify save file */
            case 'S':
                save_file_name = &argv[1][2];
                save_file = fopen (save_file_name, "W");
                if  (save_file == NULL)
                    fprintf (stderr,
                        "Warning:Unable to open save file %s\n",
                        save_file_name);
                break;
            /* -P<file>  Specify playback file */
            case 'P':
                playback_file_name = &argv[1][2];
                playback_file = fopen (playback_file_name, "r");
                if (playback_file == NULL) {
                    fprintf (stderr,
                        "Error:Unable to open playback file %s\n",
                        playback_file_name);
                    exit  (8);
                }
                break;
            default:
                fprintf (stderr,"Bad option: %s\n", argv[1]);
                exit  (8);
        }
        --argc;
        ++argv;
```

```
    }

    /* ... rest of program ... */
    return (0);
}
```

现在当用户打电话报告出错时，我们就可以告诉他："使用保存文件功能，再试一次程序，然后把文件备份给我。"用户可以运行程序，并将输入保存到文件*save.txt*中。

```
% database -Ssave.txt
Enter name: Sam
Sam is not in the list
Enter name:  John
John is in the list
Enter name:
```

在送来文件*save.txt*后，我们用回放选项运行这个程序。

```
% database -Psave.txt
Enter name: Sam
Sam is not in the list
Enter name:  John
John is in the list
Enter name:
```

现在我们有了重现程序的可靠办法。在许多情况下，这就成功了一半。一旦重现了问题，就可以进入到下一步：找到问题并解决它。

拷贝正反面

有一次，程序员要求用户送来软盘备份。第二天一份特快专递就到了，里边是软盘的照片。不过这位用户倒不是完完全全的计算机盲：他知道这是一张双面软盘，所以他把软盘的两面都照了下来。

在开始调试之前，把旧的能运行的程序保存到安全的地方（如果使用了源控制系统，例如SCCS或是RCS，那么最后的一个版本就被登记）。在许多情况下，当查找问题时，可能会发现需要尝试几种不同的解决办法，或是添加临时调试代码。

有时，会发现自己已找到了问题的根源，并需要从头做起，此时最后的一个能运行的版本就变得异常珍贵了

一旦重现了问题，就必须判定是什么原因引起的。有几种方法可以做到。

分而治之

分而治之法在第六章"条件和控制语句"中已简要讨论过。这种方法就是，在知道数据正确的地方（一定要保证数据是正确的）和数据不正确的地方之间加入几个printf语句。这种办法可以从包含错误的整段代码着手，更多的printf语句可以进一步缩小出错代码的范围，直到最终找到错误。

只用于调试的代码

分而治之法使用临时的printf语句，把它们放在需要的地方，用过再删除掉。预处理器条件编译命令用于加入和删除调试代码，例如：

```
#ifdef DEBUG
    printf ("Width %f Height %f\n", width, height);
#endif /* DEBUG */
```

该段程序在没有定义DEBUG时被编译，所以当需要调试时，可以定义它。

调试命令行开关

可以不用编译过程开关创建一个程序的特殊版本，而是在程序中永久地包含调试代码，并添加一个特殊的程序开关，这个开关打开调试信息的输出。例如：

```
if (debug)
    printf ("Width %f Height %f\n", width, height);
```

这里，debug是一个变量，当在命令行中出现 -D 选项时，该变量被设置。

这个方法有其先进性，即程序只有一个版本。同时，用户可以打开这个开关，保存输出信息，并把它送给你做进一步的分析。这种方法也有一个缺点：一个大的

可执行文件在运行时开关应该用在所有的情况下，而不应使用条件编译，除非你有某种理由不想让顾客得到调试信息。

有些程序用到了调试级别这个概念。0级仅输出最少的调试信息，1级输出多一点的信息，最高到9级，它输出所有的调试信息。

在 Aladdin 企业的 ghostscript（注 1）中增补了调试字母的算法。命令选项 -Zxxx 为每种类型的诊断输出设置了调试标志，例如，f 是用于填充算法的编码，p 是用于路径跟踪器的编码。如果想跟踪所有代码段，就需要指定 -zip。

下面的代码实现这些选项：

```
/*
 * Even though we only used one zero, C will fill in the
 * rest of the arrays with zeros.
 */
char debug[128] = {0};     /* the debugging flags */
main (int argc, char *argv[])
{
    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            /* .... normal switch .... */
            /* Debug switch */
            case 'Z':
                debug_ptr = argv[1][2];
                /* loop for each letter */
                while (*debug_ptr != '\0') {
                    debug[*debug_ptr] = 1;
                    debug_ptr++;
                }
                break;
        }
        argc--;
        argv++;
    }
    /* rest of program */
}
```

注 1：ghostscript 是一种类似 PostScript 的解释器。自由软件基金会（FSF）只收取很小的一部分复制费用。你可从以下 ftp 地址获得：*prep.ai.mit.edu:/pub/gnu*。

程序中可以这样使用:

```
if (debug['p'])
    printf ("Starting new path\n");
```

ghostscript 是一个大程序（大约 25000 行），并且很不容易调试。这种方法可让用户简单地就得到大量的信息。

通读输出

打开调试输出是得到信息的好办法。但在多数情况下，数据太多了，想要的信息很容易会被漏掉。

C 允许改变一般要输出到屏幕上的文件的输出方向，例如:

```
buggy -D9 >tmp.out
```

将运行程序 buggy，调试级别很高，并把输出结果送到文件 *tmp.out* 中。

系统中的文本编辑器还可以作为一个好的文件浏览器来使用，可以用它的查找功能找到要找的信息。

交互调试器

多数编译器生产商提供了一个交互调试器。这些调试器具有这样的能力：在程序的任何一点停止执行、检查并改变变量的值和单步执行程序。因为每种调试器都不相同，所以不可能详细讨论它们。

下面，我们将讨论一个调试器 dbx，许多运行 BSD 版 UNIX 的机器上都有这个程序。在 LINUX 和其他 UNIX 系统中，Free Software Foundation 的 gdb 调试器很流行。SYSTEM-V UNIX 使用 sdb 调试器，HP-UX 中使用 cdb 工具。每种 MS-DOS/Windows 编译器都有自己的调试器。有些编译器有多调试器，例如，Borland C++ 在 Windows 下运行的综合调试器、在 MS-DOS 下运行的独立调试器、在其他机器上运行的遥控调试器。

虽然调试器使用的具体语法各有不同，但是这里介绍的原理适用于所有的调试器。

基本的 dbx 命令是：

run
: 开始运行程序。

stop at line-number
: 在给定行插入一个断点。当程序运行到断点处时，停止运行，控制返回到调试器中。

stop in function-name
: 在给定的函数的第一行插入一个断点。命令 stop in main 用于程序开头停止运行。

cont
: 在断点后继续执行。

print expression
: 显示表达式的值。

step
: 执行程序中的一个单行。如果当前语句调用一个函数，则单步执行该函数。

next
: 执行程序中的一个单行，但把函数调用当作一个单行来对待。该命令用于跳过函数调用。

list
: 列出源程序。

where
: 显示当前活动函数的列表。

有一程序统计一串数中出现的 3 和 7 的个数。该程序在统计 7 的个数时出错，程序见例 15-6。你的结果或许不同。

例 15-6: seven/seven.c

```
 1 #include <stdio.h>
 2 char line[100];         /* line of input */
 3 int seven_count;        /* number of sevens in the data */
 4 int data[5];            /* the data to count 3 and 7 in */
 5 int three_count;        /* the number of threes in the data */
 6 int index;              /* index into the data */
 7
 8 int main() {
 9
10     seven_count = 0;
11     three_count = 0;
12     printf("Enter 5 numbers\n");
13     fgets(line, sizeof(line), stdin);
14     sscanf(line, "%d %d %d %d %d",
15         &data[1], &data[2], &data[3],
16         &data[4], &data[5]);
17
18     for (index = 1; index <= 5; ++index) {
19
20         if (data[index] == 3)
21             ++three_count;
22
23         if (data[index] == 7)
24             ++seven_count;
25     }
26     printf("Threes %d Sevens %d\n",
27             three_count, seven_count);
28     return (0);
29 }
```

用数据 7 3 7 0 2 运行程序，结果是:

```
Threes 1 Sevens 4
```

用我们要调试的程序名（seven）来调用调试器（dbx）。调试器初始化，输出提示符（dbx），并等待命令:

```
% dbx seven
Reading symbolic information...
Read 72 symbols
(dbx)
```

我们不知道变量在哪里被改变了，所以需从程序的开头着手，直到发现错误为止，每一步都显示变量seven_count以确保它的正确性。

我们需要在程序开头停下来，为的是可以单步执行它。命令stop in main告诉dbx在函数main的第一条指令处设置一个断点，命令run告诉dbx启动程序，然后运行程序，直到它遇到第一个断点：

```
(dbx) stop in main
(2) stop in main
```

dbx用数（2）来标识断点，现在需要启动程序：

```
(dbx) run
Running: seven
stopped in main at line 10 in file "/usr/sdo/seven/seven.c"
   10        seven_count = 0;
```

信息“stopped in main...”表明程序遇到了一个断点，它现在已把控制权交给了调试器。

程序已经执行到了初始化seven_count的地方。命令next将执行一个单语句，把函数调用也作为一个语句来对待（你所用的调试器中的命令名可能有所不同）。下面执行初始化部分，看看它是否起作用了：

```
(dbx) next
stopped in main at line 11 in file "/usr/sdo/seven/seven.c"
   11        three_count = 0;
(dbx) print seven_count
seven_count = 0
```

它确实执行了。继续执行下面的几行程序，每次都检查结果：

```
(dbx) next
stopped in main at line 12 in file "/usr/sdo/sevon/seven.c"
   12        get_data(data);
(dbx) print seven_count
seven_count = 0
(dbx) next
Enter 5 numbers
```

```
3 7 3 0 2
stopped in main at line 14 in file "/usr/sdo/seven/seven.c"
   14        for (index = 1; index <= 5; index++) {
(dbx) print seven_count
seven_count = 2
```

不知为何，seven_count 值变为 2。所执行的最后一个语句是get_data (data)，所以在这个函数中出了问题。我们在get_data的开始添加一个断点，移去 main 开头的那个断点，用 run 命令启动程序：

```
(dbx) stop in get_data
(4) stop in get_data
(dbx) run
Running: seven
stopped in main at line 10 in file "/usr/sdo/seven/seven.c"
   10        seven_count = 0;
```

现在处于程序开头，若想到下一个断点，加入 cont 命令继续执行：

```
(dbx) cont
Running: seven
stopped in get_data at line 31 in file "/usr/sdo/seven/seven.c"
   31        printf("Enter 5 numbers\n");
```

再次进入单步执行方式，直到找到错误：

```
(dbx) print seven_count
seven_count = 0
(dbx) next
Enter 5 numbers
stopped in get_data at line 32 in file "/usr/sdo/seven/seven.c"
   32        fgets(line, sizeof(line), stdin);
(dbx) print seven_count
seven_count = 0
(dbx) next
1 2 3 4 5
stopped in get_data at line 33 in file "/usr/sdo/seven/seven.c"
   35            &data[4], &data[5]);
(dbx) print seven_count
seven_count = 0
(dbx) next
stopped in get_data at line 36 in file "/usr/sdo/seven/seven.c"
```

```
   36    }
(dbx) print seven_count
seven_count = 5
(dbx) list 30,40
   30
   31        printf("Enter 5 numbers\n");
   32        fgets(line, sizeof(line), stdin);
   33        sscanf(line, "%d %d %d %d %d",
   34            &data[1], &data[2], &data[3],
   35            &data[4], &data[5]);
   36    }
(dbx) quit
```

32行数据正确，但当到达36行时，数据出错，所以错误的位置是程序的第33行 sscanf 命令。至此已经把问题的范围缩小到一行内，通过检查发现使用的 data[5] 是 data 数组的非法元素。

计算机偶然地把 seven_count 存储到数组 data 后面，这正是出错的地方（极易混淆）。如果不论何时超出定义数组的边界都能得到清晰的错误信息就好多了，但上下限检查较为费时，且在C中也较难实现。可以借助于一些专用调试工具如 Nu-Mega（MS-DOS/Windows）的 Bounds-Checker（上下限检查器）和 Pure Software（UNIX）的 Purify。

调试一个二分查找程序

二分查找算法非常简单。想知道所给的数是否在一个有序列表中，可以先把这个数与列表的中间元素相比较。如果它等于这个数，你很幸运 —— 可以停止查找。如果数字比列表中的中间数字大，则在列表的上半部分继续找；试一试上半部分中间位置上的数。如果数字较小，则试着在列表的下半部分去找。一直继续这个过程，每次把列表缩小一半，直到找到这个数，或是列表变为一个数为止。

例15-7使用二分查找法看一个数是否存在于文件 *numbers.dat* 中。

例 15-7: search/search1.c

```
[File: search/search1.c]
```

```
/********************************************************
 * search -- Searches a set of numbers.                 *
 *                                                      *
 * Usage:                                               *
 *      search                                          *
 *              You will be asked numbers to look up.   *
 *                                                      *
 * Files:                                               *
 *      numbers.dat -- Numbers 1 per line to search     *
 *                      (numbers must be ordered).      *
 ********************************************************/
#include <stdio.h>
#define MAX_NUMBERS      1000   /* Max numbers in file */
const char DATA_FILE[] = "numbers.dat"; /* File with numbers */

int data[MAX_NUMBERS];  /* Array of numbers to search */
int max_count;          /* Number of valid elements in data */
int main()
{
    FILE *in_file;      /* Input file */
    int middle;         /* Middle of our search range */
    int low, high;      /* Upper/lower bound */
    int search;         /* Number to search for */
    char line[80];      /* Input line */
    in_file = fopen(DATA_FILE, "r");
    if (in_file == NULL) {
        fprintf(stderr,"Error:Unable to open %s\n", DATA_FILE);
        exit (8);
    }

    /*
     * Read in data
     */

    max_count = 0;
    while (1) {
        if (fgets(line, sizeof(line),  in_file) == NULL)
            break;

        /* convert number */
        sscanf(line, "%d", data[max_count]);
        ++max_count;
    }
```

```
    while (1) {
        printf("Enter number to search for or -1 to quit:");
        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &search);

        if (search == -1)
            break;

        low = 0;
        high = max_count;

        while (1) {
            middle = (low + high) / 2;

            if (data[middle] == search) {
                printf("Found at index %d\n", middle);
            }

            if (low == high) {
                printf("Not found\n");
                break;
            }

            if (data[middle] < search)
                low = middle;
            else
                high = middle;
        }
    }
    return (0);
}
```

数据文件 *numbers.dat* 包含:

```
 4
 6
14
16
17
```

运行这个程序，得到:

```
% search
Segmentation fault (core dumped)
```

这不是一个好的结果，因为它意味着程序出错了，程序试图去读不存在的内存。错误发生时建立了一个叫*core*的文件，它包含程序执行时的快照。调试器dbx可以读取这个文件并帮助检查发生的错误：

```
% dbx search
Reading symbolic information...
Read 79 symbols
warning: core file read error: address not in data space
warning: core file read error: address not in data space
warning: core file read error: address not in data space
program terminated by signal SEGV (no mapping at the fault address)
(dbx)
```

“warning: core file,...”信息是调试器在告诉用户临时变量空间（堆栈）已无用处且包含了错误数据。where 命令告诉用户哪个函数在调用哪个函数（也称为栈轨迹）。最先打印的是当前的函数，然后是调用它的函数，依此类推，直到最外层的函数main：

```
(dbx) where
number() at 0xdd7d87a
_doscan() at 0xdd7d329
sscanf(0xdfffadc, 0x200e3, 0x0) at 0xdd7ce7f
main(0x1, 0xdfffb58, 0xdfffb60), line 41 in "search.c"
(dbx)
```

上例显示main调用sscanf，该调用来自于main的第41行。sscanf调用_doscan，因为sscanf是标准库函数，所以得不到行号信息。指令号是0xdd7ce7f，然而这个信息并不太有用。程序_doscan调用number，number就是执行非法内存访问的命令。

这个信息并不是说number中有错误，问题可能是由传送参数给number的_doscan引起的，而_doscan的参数又是从sscanf得到的，sscanf的参数从main中得到。这个链条上的任何一个函数都可能包含错误从而把错误的指针传给其他函数。

通常标准库都是经过测试的，所以应该在用户的代码中找错，查看堆栈轨迹，可查到执行程序的最后一行是 main 的第 41 行。

使用 list 命令，可以检查该行：

```
(dbx) list 41
   41          sscanf(line, "%d", data[max_count]);
(dbx)
```

就是这行出的问题。

另外一个找错的方法是单步执行程序，直到错误出现。首先列出程序段找一个适当位置插入断点，然后开始执行并单步处理：

```
(dbx) list 26,31
   22       int low, high;      /* Upper/lower bound */
   23       int search;         /* Number to search for */
   24       char line[80];      /* input line */
   25
   26       in_file = fopen(DATA_FILE, "r");
   27       if (in_file == NULL) {
   28           fprintf(stderr,"Error:Unable to open %s\n",
                                                DATA_FILE);
   29           exit (8);
   30       }
   31
(dbx) stop at 26
(1) stop at "search.c":26
(dbx) run
Running: search
stopped in main at line 26 in file "search.c"
   26       in_file = fopen(DATA_FILE, "r");
(dbx) step
stopped in main at line 27 in file "search.c"
   27       if (in_file == NULL) {
(dbx) step
stopped in main at line 35 in file "search.c"
   35       max_count = 0;
(dbx) step
stopped in main at line 37 in file "search.c"
   37           if (fgets(line, sizeof(line),  in_file) == NULL)
```

```
(dbx) step
stopped in main at line 41 in file "search.c"
   41           sscanf(line, "%d", data[max_count]);
(dbx) step
signal SEGV (no mapping at the fault address) in number at 0xdd7d87a
number+0x520:             movl    a6@(0x1c),a0
(dbx) quit
```

这种方法也指向第41行。检查发现忘记在变量sscanf前放&号了。所以把41行从

```
sscanf(line, "%d", data[max_count]);
```

改成：

```
sscanf(line, "%d", &data[max_count]);
```

再试一次。列表中的第一个数是4，用它试程序，这一次的输出是：

```
Enter number to search for or -1 to quit:4
Found at index 0
Found at index 0
Not found
Enter number to search for or -1 to quit:^C
```

程序应该找到这个数，让我们知道它的下标是0，然后要求输入另一个数。但是得到的却是两个found信息和一个not found信息。我们知道这一次程序顺利地执行到输出第一条found信息之处，但在此之后，又出现了出错。所以在程序中至少有多于一个的错误。

再回到调试器中，使用list命令定位到found信息，并插入一个断点：

```
% dbx search
Reading symbolic information...
Read 79 symbols
(dbx) list 58,60
   58
   59               if (data[middle] == search) {
   60                   printf("Found at index %d\n", middle);
   61               }
```

```
   62
   63              if (low == high) {
   64                  printf("Not found\n");
   65                  break;
   66              }
   67
(dbx) stop at 60
(3) stop at "search.c":60
(dbx) run
stopped in main at line 60 in file "search.c"
   60                  printf("Found at index %d\n", middle);
(dbx)
```

现在单步运行看会发生什么:

```
   60                  printf("Found at index %d\n", middle);
(dbx) step
Found at index 0
stopped in main at line 63 in file "search.c"
   63              if (low == high) {
(dbx) step
stopped in main at line 68 in file "search.c"
   68              if (data[middle] < search)
(dbx) step
stopped in main at line 71 in file "search.c"
   71                  high = middle;
(dbx) step
stopped in main at line 57 in file "search.c"
   57              middle = (low + high) / 2;
(dbx) quit
```

程序并没退出循环，而是用search查找。因为数已经找到，这个search就导致了奇怪的结果，原来在printf后忘了放break语句。

把:

```
if (data[middle] == search) {
    printf("Found at index %d\n", middle);
}
```

改为:

```
if (data[middle] == search) {
    printf("Found at index %d\n", middle);
    break;
}
```

做了这些修改后，再试一次程序：

```
% search
Enter number to search for or -1 to quit:4
Found at index 0
Enter number to search for or -1 to quit:6
Found at index 1
Enter number to search for or -1 to quit:3
Not found
Enter number to search for or -1 to quit:5
...program continues forever or until we abort it...
```

这一次程序失去了控制。我们并没有设置断点，只是重新运行程序。几秒钟过去了，我们相信已经陷入了死循环，只得用CTRL-C终止程序。一般情况下，这会使程序异常中断，并返回到外壳提示符下，因为我们运行的是调试程序，所以控制返回到 dbx 下。

```
% dbx search
Reading symbolic information...
Read 80 symbols
(dbx) run
Running: search
Enter number to search for or -1 to quit:5
^C
interrupt in main at line 70 in file "search.c"
   70                     low = middle;
```

现在可以使用单步执行命令一步一步地通过无限循环，看看关键值的变化情况。

```
   70                     low = middle;
(dbx) step
stopped in main at line 57 in file "search.c"
   57                 middle = (low + high) / 2;
(dbx) step
stopped in main at line 59 in file "search.c"
   59                 if (data[middle] == search) {
```

```
(dbx) print middle
middle = 0
(dbx) print data[middle]
data[middle] = 4
(dbx) print search
search = 5
(dbx) step
stopped in main at line 64 in file "search.c"
   64             if (low == high) {
(dbx) step
stopped in main at line 69 in file "search.c"
   69             if (data[middle] < search)
(dbx) step
stopped in main at line 70 in file "search.c"
   70                 low = middle;
(dbx) step
stopped in main at line 57 in file "search.c"
   57             middle = (low + high) / 2;
(dbx) step
stopped in main at line 59 in file "search.c"
   59             if (data[middle] == search) {
(dbx) step
stopped in main at line 64 in file "search.c"
   64             if (low == high) {
(dbx) step
stopped in main at line 69 in file "search.c"
   69             if (data[middle] < search)
(dbx) step
stopped in main at line 70 in file "search.c"
   70                 low = middle;
(dbx) step
stopped in main at line 57 in file "search.c"
   57             middle = (low + high) / 2;
(dbx) step
stopped in main at line 59 in file "search.c"
   59             if (data[middle] == search) {
(dbx) step
stopped in main at line 64 in file "search.c"
   64             if (low == high) {
(dbx) step
stopped in main at line 69 in file "search.c"
   69             if (data[middle] < search)
(dbx) print low,middle,high
low = 0
```

```
middle = 0
high = 1
(dbx) print search
search = 5
(dbx) print data[0], data[1]
data[0] = 4
data[1] = 6
(dbx) quit
```

问题是我们已经执行到：

```
low=     0
middle=  0
high=    1
```

我们要找的项目（值5）处在元素0（值4）和元素1（值6）之间。算法有一个错误，这种类型的错误当程序中变量的值不是它应有的值时才出现。本例中，变量是下标middle。

查找已经缩小到0和1之间，然后取这段间隔的中间数，但算法又出了问题。原因是间隔太小，middle算出来的不是元素1。所以把查找0到1“缩小”到新的0到1之间，再次查找，因为这个间隔还是开始的那个间隔，所以进入了死循环。

为解决这个问题，看看代码。如果中间元素匹配，显示“found”信息退出。那就意味着不需要再找中间元素了。所以，把下列代码：

```
if (data[middle] < search)
    low = middle;
else
    high = middle;
```

调整成：

```
if (data[middle] < search)
    low = middle +1;
else
    high = middle -1;
```

改后的整版程序见例15-8。

例 15-8: search/search4.c

```
[File: search/search4.c]
/********************************************************
 * search -- Searches a set of numbers.                 *
 *                                                      *
 * Usage:                                               *
 *      search                                          *
 *              You will be asked numbers to look up.   *
 *                                                      *
 * Files:                                               *
 *      numbers.dat -- Numbers 1 per line to search     *
 *                     (Numbers must be ordered).       *
 ********************************************************/
#include <stdio.h>
#define MAX_NUMBERS     1000          /* Max numbers in file */
const char DATA_FILE[] = "numbers.dat"; /* File with numbers */

int data[MAX_NUMBERS];  /* Array of numbers to search */
int max_count;          /* Number of valid elements in data */
int main()
{
    FILE *in_file;      /* Input file */
    int middle;         /* Middle of our search range */
    int low, high;      /* Upper/lower bound */
    int search;         /* number to search for */
    char line[80];      /* Input line */

    in_file = fopen(DATA_FILE, "r");
    if (in_file == NULL) {
        fprintf(stderr,"Error:Unable to open %s\n", DATA_FILE);
        exit (8);
    }

    /*
     * Read in data
     */

    max_count = 0;
    while (1) {
        if (fgets(line, sizeof(line),  in_file) == NULL)
            break;

        /* convert number */
```

```
        sscanf(line, "%d", &data[max_count]);
        ++max_count;
    }

    while (1) {
        printf("Enter number to search for or -1 to quit:");
        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &search);

        if (search == -1)
            break;

        low = 0;
        high = max_count;

        while (1) {
            if (low >= high) {
                printf("Not found\n");
                break;
            }

            middle = (low + high) / 2;

            if (data[middle] == search) {
                printf("Found at index %d\n", middle);
                break;
            }

            if (data[middle] < search)
                low = middle +1;
            else
                high = middle -1;
        }
    }
    return (0);
}
```

对大多数程序来说，交互式的调试器运行较好。有时它们稍稍需要一点帮助。考虑一下例15-9。调试过程中，发现当point_number为735时程序失败了。我们想在进行计算之前放一个断点，当调试器往程序中插入断点时，程序将正常执行到断点处，然后控制返回到调试器。这就允许用户检查并修改变量，也可以执行其他的调试命令。当键入continue 命令时，程序将继续执行，就好像什么

都没有发生一样。问题是在所需要调试的那一语句之前，要有734个断点，而我们并不想每次都停下来。

例 15-9: cstop/cstop.c

```
extern float lookup(int index);

float point_color(int point_number)
{
    float correction;  /* color correction factor */
    extern float red,green,blue;/* current colors */

    correction = lookup(point_number);
    return (red*correction * 100.0 +
            blue*correction * 10.0 +
            green*correction);
}
```

如何能让调试器仅在`point_number == 735`时停下来?把下列代码加入程序可以办到:

```
48    if (point_number == 735)  /* ### Temp code ### */
49        point_number = point_number;   /* ### Line to stop on ### */
```

第49行除了使调试器可以停在某一行以外什么也不做。可以用命令`stop at 49`在这一行设一个断点。程序会处理前734个点，然后执行第49行，遇到了断点。(一些调试器有一个有条件的断点。高级dbx命令`stop at 49 if point_number == 735`也会运行；不过，你的调试器可能没有这样的高级功能。)

实时运行错误

实时运行错误（Runtime error）通常是最容易修改的。一些类型的实时运行错误包括:

- 段违例。这个错误表明，程序试图去逆引用一个含不正确值的指针。
- 栈上溢。程序试图使用太多的临时变量。有时，这意味着程序太大，或是使用了很多的大临时数组，但多数情况下，它是由于无限递归问题所致。几乎

所有的UNIX系统都自动检查这个错误。Turbo C++和Borland C++只有在使用了编译过程选项-N时才检查栈上溢。

- 被0除。被0除是个明显错误。UNIX使用错误信息"Floating exception (core dumped)。"来标记这个错误并报告一个整数被0除。

这些错误都可以中止程序执行。在UNIX中，将输出运行程序的映像，名为*core file*。

实时运行错误的一个问题是当它们发生时，程序执行立即中止，用于缓冲文件的缓冲区内容也没有全部输出。这可能产生令人吃惊的结果，请考虑示例15-10。

例15-10: flush/flush.c

```
#include <stdio.h>
int main()
{
    int i,j;    /* two random integers */

    i = 1;
    j = 0;
    printf("Starting\n");
    printf("Before divide...");
    i = i / j;  /* divide by zero error */
    printf("After\n");
    return(0);
}
```

运行时，程序输出：

```
Starting
Floating exception (core dumped)
```

这可能会使你认为还没有执行到除法，而实际上，它已经执行了。那么信息"Before divide....."到哪里去了呢？printf语句已经执行，并把信息放到了缓冲区中，然后程序死了。缓冲区没有机会清空。

在代码中加一条强制清缓冲区的命令，可以反映出程序执行的真实情况，见例15-11。

例 15-11: flush2/flush2.c

```
[File: flush2/flush2.c]
#include <stdio.h>
int main()
{
    int i,j;    /* two random integers */

    i = 1;
    j = 0;

    printf("Starting\n");
    fflush(stdout);

    printf("Before divide...");
    fflush(stdout);

    i = i / j;  /* divide by zero error */

    printf("After\n");
    fflush(stdout);
    return(0);
}
```

flush 语句使得 I/O 的效率降低了，但时效性更强了。

公开声明调试方法

公开声明调试方法是这样一种方法：借助于它，程序员可以向别人解释他的程序：向感兴趣的人、向不感兴趣的人、向墙壁——不论他向谁解释多长时间都可以。

典型的声明是这样的："Hey Bill, could you take a look at this. My program has a bug in it. The output should be 8.0 and I'm getting -8.0. The output is computed using this formula and I've checked out the payment value and rate, and the date must be correct unless there is something wrong with the leap year code, which-Thank you, Bill, you've found my problem." Bill 一句话也不会回答。

这种类型的调试也叫"walkthrough（预演）"，让其他人参与进来会带来全新的视角，因为其他人常常会发现你忽略的问题。

优化

优化是检查程序并使代码更高效从而运行得更快的一门艺术。多数编译器有一个命令行开关，使它们产生优化后的代码。这种效率以编译时间为代价，优化功能打开后编译器得花更长的时间生成代码。

另一种类型的优化在程序员修改程序、以使用更有效的算法时出现。本节将讨论第二种类型的优化。

现在用一个词来说明优化：不需要。多数程序不需要进行优化，它们运行得足够快。谁关心一个交互程序用了 0.5 秒还是 0.2 秒呢？

要让程序运行得更快，最简单的办法是使用一台更快的计算机。许多时候买一台功能更强大的机器比优化一个程序要便宜，因为优化可能会把新的错误加入到代码中。不要希望优化会带来奇迹。通常多数程序只能提速 10% 到 20%。

为了让你对能完成的目标产生认识，我会优化一个样本函数。例 15-12 初始化一个矩阵（二维数组）。

例 15-12: matrix/matrix1.c

```
[File: matrix/matrix1.c]
#define X_SIZE 60
#define Y_SIZE 30

/* A random matrix */
int matrix[X_SIZE][Y_SIZE];

/********************************************************
 * init matrix --Sets every element of matrix to -1.    *
 ********************************************************/
void init_matrix(void)
{
    int x,y;    /* current element to zero */

    for (x = 0; x < X_SIZE; ++x) {
        for (y = 0; y < Y_SIZE; ++y) {
            matrix[x][y] = -1;
        }
```

```
        }
    }
```

如何对这个函数优化？首先，我们注意到使用了两个局部变量。通过用修饰符 `register` 描述这些变量，告诉编译器这些变量使用频繁，要把它们放到快速的寄存器中，而不要放到相对慢一些的主内存中。寄存器的数目因机器的不同而不同，像PC这样的慢机器有两个，多数UNIX系统有大约11个，超级计算机可以有多达128个寄存器。你可以定义比所拥有的寄存器数量多的寄存器变量，C会把多余的变量放在主内存中。

现在的程序如例15-13所示。

例15-13: matrix/matrix2.c

```
[File: matrix/matrix2.c]
#define X_SIZE 60
#define Y_SIZE 30

int matrix[X_SIZE][Y_SIZE];

/********************************************************
 * init_matrix -- Sets every element of matrix to -1.   *
 ********************************************************/
void init_matrix(void)
{
    register int x,y;     /* current element to zero */

    for (x = 0; x < X_SIZE; ++x) {
        for (y = 0; y < Y_SIZE; ++y) {
            matrix[x][y] = -1;
        }
    }
}
```

外层循环执行60次，这意味着执行内层循环所附带的开销要60次。如果我们把循环的次序颠倒一下，则只需处理30次内层循环。

一般来说，循环的次序应该是最内层的循环最复杂，最外层的循环最简单。如例15-14所示。

例 15-14: matrix/matrix3.c

```
[File: matrix/matrix3.c]
#define X_SIZE 60
#define Y_SIZE 30

int matrix[X_SIZE][Y_SIZE];

/*******************************************************
 * init_matrix --Sets every element of matrix to -1.   *
 *******************************************************/
void init_matrix(void)
{
    register int x,y;    /* current element to zero */

    for (y = 0; y < Y_SIZE; ++y) {
        for (x = 0; x < X_SIZE; ++x) {
            matrix[x][y] = -1;
        }
    }
}
```

2 的幂效能

给一个数组建立索引需执行乘法。看从前面的例子中取出的下面一行:

```
matrix[x][y] = -1;
```

为得到 -1 存放的位置，程序必须执行下列步骤:

1. 得到矩阵地址。
2. 计算 x * Y_SIZE
3. 计算 y
4. 把上述三部分加在一起，形成地址。

在 C 中相应代码是:

```
*(matrix + (x * Y_SIZE) + y) = -1;
```

然而我们不应这样写一个矩阵的访问语句，因为C会处理细节，我们应该意识到这些细节可以帮助我们产生高效的代码。

差不多所有的C编译器都把乘以2的幂（2、4、8、...）的计算转化为移位操作，所以一个昂贵的操作（乘法）就转为了不昂贵的操作（移位）。

例如：

```
i = 32 * j;
```

被编译为：

```
i = j << 5; /* 2**5 == 32 */
```

Y_SIZE是30，它不是2的幂。把Y_SIZE增加到32会浪费一些内存，但程序更快了，如例15-15所示。

例 15-15: matrix/matrix4.c

```
[File: matrix/matrix4.c]
#define X_SIZE 60
#define Y_SIZE 32

int matrix[X_SIZE][Y_SIZE];

/*******************************************************
 * init_matrix -- Sets every element of matrix to -1.   *
 *******************************************************/
void init_matrix(void)
{
    register int x,y;     /* current element to zero */

    for (y = 0; y < Y_SIZE; ++y) {
        for (x = 0; x < X_SIZE; ++x) {
            matrix[x][y] = -1;
        }
    }
}
```

因为要初始化一段连续的内存位置，所以可以从第一个位置开始，在随后的X_SIZE*Y_SIZE个元素中放入-1来初始化矩阵（见例15-16）。使用这种方法可以把循环数减少到一层。矩阵的下标从标准下标（matrix[x][y]）开始改变，需要一次移动，加入指针逆引用（*matrix_ptr）和一个增值（matrix_ptr++）。

例 15-16: matrix/matrix5.c

```
[File: matrix/matrix5.c]
#define X_SIZE 60
#define Y_SIZE 30

int matrix[X_SIZE][Y_SIZE];

/********************************************************
 * init_matrix -- set every element of matrix to -1     *
 ********************************************************/
void init_matrix(void)
{
    register int index;         /* element counter */
    register int *matrix_ptr;

    matrix_ptr = &matrix[0][0];
    for (index = 0; index < X_SIZE * Y_SIZE; ++index) {
        *matrix_ptr = -1;
        ++matrix_ptr;
    }
}
```

但是为什么既有循环计数器又有matrix_ptr？能不能将两者合并?实际上是可以的，如例15-17所示。

例 15-17: matrix/matrix6.c

```
[File: matrix/matrix6.c]
#define X_SIZE 60
#define Y_SIZE 30

int matrix[X_SIZE][Y_SIZE];
```

```
/********************************************************
 * init_matrix -- Sets every element of matrix to -1.   *
 ********************************************************/
void init_matrix(void)
{
    register int *matrix_ptr;

    for (matrix_ptr = &matrix[0][0];
            matrix_ptr <= &matrix[X_SIZE-1][Y_SIZE-1];
            ++matrix_ptr) {

        *matrix_ptr = -1;
    }
}
```

函数已经优化好了。使它更好的唯一办法是用手工编码把它变成汇编语言，这种改变可以使函数运行更快。但是汇编语言是根本不能移植的，并且极易出错。

库函数memset可以用一个单字符来填充矩阵或是数组。如例15-18所示，可以用它来初始化这个程序中的矩阵。像memset这样的常用库函数常常是用汇编语言编码，而且可以利用独特的依赖于处理器的技巧来完成一些操作，可能比在C中实现得要快一些。

例15-18: matrix/matrix7.c

```
[File: matrix/matrix7.c]
#include <memory.h>      /* Gets definition of memset */
#define X_SIZE 60
#define Y_SIZE 30

int matrix[X_SIZE][Y_SIZE];
/********************************************************
 * init_matrix -- Sets every element of matrix to -1.   *
 ********************************************************/
void init_matrix(void)
{
    memset(matrix, -1, sizeof(matrix));
}
```

现在函数中只有一条函数调用语句。为调用其他函数而不得不调用某个函数好像不应该，因为要承担两次函数调用的开销。如果从main中调用会更好。为什么不

告诉用户使用memset而不用init_matrix重写他的程序呢？因为他有几百个init_matrix调用，他可不想做那些编辑工作。

如果把函数重新定义成宏，我们有一个看起来像函数调用的init_matrix，但因为它是宏，只能做行内扩展，所以可以避免所有和函数调用相关的额外开销。见例15-19。

例 15-19: matrix/matrix8.c

```
#define X_SIZE 60
#define Y_SIZE 30

int matrix[X_SIZE][Y_SIZE];

/********************************************************
 * init_matrix -- Sets every element of matrix to -1.   *
 ********************************************************/
#define init_matrix() \
    memset(matrix, -1, sizeof(matrix));
```

问题 15-1：为什么memset能成功地把矩阵元素初始化为-1，但例15-20中试图用它把每个元素设置为1时，执行却失败了？

例 15-20: matrix/matrix9.c

```
#define X_SIZE 60
#define Y_SIZE 30

int matrix[X_SIZE][Y_SIZE];

#define init_matrix() \
    memset(matrix, 1, sizeof(matrix));
```

如何优化

我们的矩阵初始化函数说明了几种优化策略，它们是：

循环排序

嵌套循环应该排序：最简单的在最外层，最复杂的在最内层。

能量节约

应该用便宜的操作代替昂贵的操作。表15-1列出了一般操作的相对成本。

表15-1 操作的相对成本

操作	相对成本
printf 和 scanf	1000
malloc 和 free	800
三角函数（sin, cos...）	500
浮点数（任何运算）	100
整数除法	30
整数乘法	20
函数调用	10
简单数组下标	6
移位	5
加/减	5
指针逆引用	2
按位与、或、非	1
逻辑与、或、非	1

注意：使用C格式的函数printf、scanf和sscanf，其调用开销相当昂贵，原因是它们必须一个字符一个字符地读入格式串，寻找转换字符（%），然后在字符串和数之间进行代价很高的转换。在强调时间的函数中应避免使用这些函数。

2的幂

当进行整数乘法和除法时，使用2的幂。大多数编译器将用移位来代替该操作。

指针

对数组来说，使用指针比下标更快，不过使用指针需要的技巧要多一些。

宏

使用宏可去除和函数调用相关的消耗，但同时它也加大了代码且有点难于测试。

案例研究：宏与函数

有一次，我为一家大的计算机制造厂商编制一个字处理程序。其中一个函数next_char，它从当前文件中读取下一个字符。这个函数在程序的各个地方使用上万处。当第一次我们把next_char作为一个函数来测试这个程序时，程序慢得难以忍受。分析程序时我们发现，有90%的时间花在next_char上。所以我们把它改为一个宏。结果速度提高了一倍；但是，代码增加了40%，并需要一块内存扩充卡才能工作。虽然速度令人满意了，但代码的长度又不可忍受了。最后，不得不写一个自己进行优化的汇编语言程序，使程序的大小和速度都可以接受。

案例研究：优化一个涂色算法

一次我接受了一个优化程序任务，该程序为一大幅画涂色。问题是，程序用了8小时才处理完一幅画。这样，一天只能处理一幅。

我做的第一件事是在一台带有浮点加速器的机器上运行该程序。这样，时间缩减到大约6小时。接着，我获准使用另一个项目组暂时闲置的高速RISC计算机，这样，时间缩减到2小时。

仅仅使用更快的机器就节省了6小时，还没有改动代码。

在最内层循环中，有两个相对简单的函数只调用了一次。用宏重写这些函数，又节省了15分钟。

接着，我把所有能改的浮点操作改为整数操作。这又节省了1小时45分钟运行时间内的30分钟。我注意到程序花5分钟去读一个ASCII码文件，该文件中含有一长串用于转换过程的浮点数。因知道scanf是一个极其昂贵的函数，所以我把文件改为二进制后，初始化过程几乎不占什么时间。这时，总的运行时间降到了1小时10分钟。

仔细检查代码，使用我了解的每一个技巧，又节省了5分钟。离我的一小时运行一次的目标只剩下5分钟了。此时，我的项目重新进行调整，程序束之高阁，留待以后使用。

答案

解答 15-1：问题是memset是一个字符填充函数，一个整数由2或4字节（字符）组成，每字节赋值为1，所以2个字节的整数将接收下面的值：

```
integer = 0x0101;
```

-1的1个字节大小的十六进制值是0xFF，2个字节大小的十六进制值是0xFFFF。所以我们可以把两个单字节的-1放到一起，变为-1。对0也可以这样处理。换作其他的数时将产生错误答案。例如，1是0x01。两个字节的1是0x0101，或是257。

编程练习

练习 15-1：找一个以前的程序，用交互调试器运行它，并检查几个变量的中间值。

练习 15-2：编写一个矩阵乘法函数。建立一个测试程序，它不仅测试函数，还测试执行的时间。用指针优化这个程序，看看节省了多少时间。

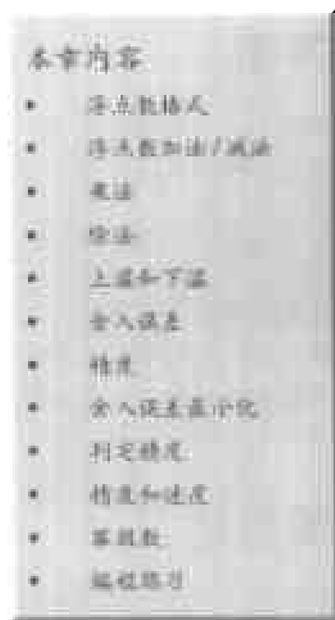
练习 15-3：编程把数组中的元素加起来，然后优化它。

练习 15-4：编程统计一个字符数组中的位数。通过寄存器整数变量来优化它。统计该程序作用于几个不同大小的数组时所花费的时间，优化后比优化前节省了多长时间。

练习 15-5：编写自己的库函数memcpy，并优化它。memcpy的实现是用汇编语言写的，这能充分利用处理器优势和奇特之处。你的memcy与之相比怎么样？

第十六章

浮点数



1的值足够大时便与2相等。

—— 无名氏

计算机非常便于处理整数，不仅算法简单、精确而且速度快。但浮点数算术却相反，计算机进行浮点数运算时有很大困难。

本章讨论浮点数的一些问题。为了理解浮点数运算涉及的原则，定义了一个简单的十进制浮点数格式。建议你不用计算机，而用笔和纸来做这些题，这样可以更直接地处理问题。

计算机使用的格式与本章中定义的非常相似，只不过计算机不使用十进制，而是使用二进制、八进制或十六进制。但是，所有这里讲述的问题计算机上都可能发生。

浮点数格式

浮点数包括三部分：符号、小数和幂。小数表示为4位十进制数，幂是1位十进制数。所以定义的格式为：

```
± f.fff × 10±e
```

其中:

±

是符号（正或负）。

f.fff

是4位小数。

± e

是1位数字的幂。

零表示为 $+0.000 \times 10^{+0}$。我们用“E”格式来表示这些数：± 0.000E ± e。

这种格式类似于许多计算机中使用的浮点数格式。IEEE 已经定义了浮点数标准（#754），但是并不是所有的机器都使用这个标准。

表16-1列出一些典型的浮点数。

表16-1 浮点数举例

表示法	数字
+1.000E+0	1.0
+3.300E+4	33000.0
-8.223E-3	-0.008223
+0.000E+0	0.0

本章定义的浮点数运算严格遵守一系列规则。为了使错误减少到最少，我们使用了一位保护位，就是在运算期间加在小数最后的一个额外位。许多计算机在它们的浮点运算部件中都使用了保护位。

浮点数加法/减法

为把两个数字如2.0和3.0加起来，必须执行下列步骤:

1. 准备数:

   +2.000E+0 这个数为2.0

   +3.000E-1 这个数为0.3

2. 给两个数加上保护位:

   +2.0000E+0 这个数为2.0

   +3.0000E-1 这个数为0.3

3. 把幂最小的数右移一位，同时幂增1。继续这个过程，直到两个数的幂匹配:

   +2.0000E+0 这个数为2.0

   +0.3000E-0 这个数为0.3

4. 把两个小数相加。结果的幂与两个数的幂相同。

   +2.0000E+0 这个数为2.0

   +0.3000E-0 这个数为0.3

   +2.3000E+0 结果为2.3

5. 标准化该数：左移或右移它，直到它的小数点左侧仅有一位非零数时为止，相应地调整它的幂。像+0.1234E+0应当标准化+1.2340E-1。因为+2.3000E+0已经是标准形式，所以我们不用再做什么了。

6. 最后，如果保护位大于或等于5，把下一位四舍五入，否则去掉。

   +2.3000E+0 去掉最后一位

   +2.300E+0 结果为2.3

对于浮点数减法，把第二个操作数的符号变反，再做加法。

乘法

当我们想相乘两个数如0.12 × 11.0时，要遵守下面的规则：

1. 给两个数加上保护位：

   +1.2000E-1　数为0.12
   +1.1000E+1　数为11.0

2. 两个小数进行乘法运算，幂数相加。（1.2 × 1.1=1.32）（-1+1=0）：

   +1.2000E-1　数为0.12
   +1.1000E+1　数为11.0
   +1.3200E+0　结果为1.32

3. 结果标准化。如果保护位大于等于5，则把下一位四舍五入，否则去掉。

   +1.3200E+0　数为1.32

注意在乘法中，不必进行移位。乘法的规则比加法要少得多。整数乘法比整数加法要慢得多，浮点数算术中的乘法速度很接近加法速度。

除法

两数相除如100.0除以30.0，必须执行下列步骤：

1. 加上保护位：

   +1.0000E+2　数为100.0
   +3.0000E+1　数为30.0

2. 小数部分进行除法，幂次相减：

   +1.0000E+2　数为100.0
   +3.0000E+1　数为30.0
   +3.0000E+1　数为3.333

3. 结果标准化:

   +3.3330E+0　数为 3.333

4. 如果保护位大于等于5，则把下一位四舍五入，否则去掉:

   +3.333E+0　结果为 3.333

上溢和下溢

计算机可以处理的数有一定的限度，下面计算的结果是什么?

```
9.000E-9 × 9.000E+9
```

相乘后，得到:

$$8.1 \times 10^{19}$$

然而受一位幂太小的限制，所以放不下19，这是上溢的一个例子（有时叫幂上溢）。当发生上溢时，有些计算机做了特殊处理，所以程序会中断，并输出一条错误信息。其他一些计算机做得没这么好，只产生一个错误结果（如8.100E+9）。遵从IEEE浮点数标准的计算机会得到一个特殊的值，名为+Infinity（正无穷大）

当数太小，计算机不能处理时，发生下溢。例如:

```
1.000E-9 × 1.000E-9
```

结果为:

$$1.0 \times 10$$

因为 -18 太小了，不能用一位数字表示，所以发生了下溢。

舍入误差

浮点数算术不精确。每个人都知道 1+1 是 2，但你知道 1/3+1/3 不等于 2/3 吗?

这个结果可以由下列浮点数计算列示出来:

2/3 表示为浮点数是 6.667E-1。

1/3 表示为浮点数是 3.333E-1。

+3.333E-1

+3.333E-1

+6.666E-1 或 0.6666

不是:

+6.667E-1

每台计算机的浮点数都存在相似的问题。例如，用二进制浮点数就不能精确地表示 0.2。

浮点数算术决不能用来表示金额，因为我们习惯于处理元和分，如你可能想把金额 1.98 美元表示为:

```
float amount=1.98
```

然而浮点数计算的次数越多，舍入误差就越大。银行、信用卡公司和 IRS 都很注意金额的正确性，拿给 IRS 一张不太精确的支票会使他们不高兴，金额应以最小单位的整数来存储。

精度

多少位小数是精确的呢？第一感觉可能会使你被误导说，所有 4 位数字。读了上一节舍入误差的读者可能会把答案改为 3 位。

答案是：精度依赖于计算。某些计算，如两个很接近的数相减，会得到不精确的结果。例如，考虑下面的计算:

```
1 - 1/3 - 1/3 - 1/3
```

或

```
 1.000E+0
-3.333E-1
-3.333E-1
-3.333E-1
```

或

```
 1.000E+0
-3.333E+0
-3.333E+0
-3.333E+0
0.0010E+0 或 1.000E-3
```

正确的答案是0.000E+0而我们得到的是1.000E-3。小数的第1位是错的，这是一个称为“舍入误差”问题的例子，在进行浮点运算时，就会产生舍入误差。

舍入误差最小化

有许多方法可以减小舍入误差，一种方法是我们已经讨论过的保护位；另一种方法是使用**double**类型取代**float**类型。这样，数的精度差不多是原来的两倍，表示的范围也为原来的两倍。舍入误差问题也少了两倍，但是舍入误差仍然存在。

减少浮点数带来的问题的高级技术，可以在数值分析的书中找到。它们超出了本书讨论的范围。本章的目的是让读者对遇到的问题有一个初步的了解。

从本质来说浮点数是不精确的。人们往往认为计算机是非常精确的机器，它们可以是精确的，但也可能得出谬之千里的结果。你应该意识到错误可能会偷偷溜到程序中的某些地方。

判定精度

有一个简单的判定浮点数精度的方法（用于简单计算中）。下面的程序累加1.0+0.1、1.0+0.01、1.0+0.001，依此类推，直到第二个数太小，不足以影响结果时为止。

C语言规定所有的浮点数都处理为双精度，这种方法意味着表达式：

```
float number1, number2;

. . .

while (number1 + number2 != number1)
```

等价于：

```
while (double(number1) + double(number2) != double(number1))
```

如果使用了1+0.001技巧，自动转化**float**和**double**会对机器准确性带来影响（本例中，对于32位格式却报告有84位精度）。例16-1计算了等式中和内存中使用的浮点数的精确性。注意用来决定内存中浮点数精度的技巧。

例16-1: float/float.c

```
#include <stdio.h>
int main()
{
    /* two numbers to work with */
    float number1, number2;
    float result;              /* result of calculation */
    int   counter;             /* loop counter and accuracy check */

    number1 = 1.0;
    number2 = 1.0;
    counter = 0;

    while (number1 + number2 != number1) {
        ++counter;
        number2 = number2 / 10.0;
    }
```

```
    printf ("%2d digits accuracy in calculations\n", counter);

    number2 = 1.0;
    counter = 0;

    while (1) {
        result = number1 + number2;
        if (result == number1)
            break;
        ++counter;
        number2 = number2 / 10.0;
    }
    printf ("%2d digits accuracy in storage\n", counter);
    return (0);
}
```

在带MC68881浮点芯片的Sun-3/50上运行该程序，得到：

```
20 digits accuracy in calculations
 8 digits accuracy in storage
```

这个程序只给出大致的浮点数计算，更准确的定义在标准引用文件*float.h*中。

精度和速度

double类型的变量的精度是普通**float**变量的两倍。大多数人设想双精度计算会比单精度计算占用更多的时间，情况并不总是这样，记住C要求所有运算必须在**double**中进行。

对于等式：

```
float answer, number1, number2;

answer = number1 + number2;
```

C必须执行下列步骤：

1. 把number1从单精度变为双精度。

2. 把 number2 从单精度变为双精度。

3. 进行双精度加法。

4. 把结果转换为单精度并存储在 *answer* 中。

如果变量是 **double** 类型的，C 将只执行下面各步：

1. 进行双精度加法。

2. 把结果存储在 *answer* 中。

正如你看到的，第二种形式简单得多，它不需要进行三次类型转换。在有些情况下，程序从单精度算法转换为双精度算法使程序运行得更快。

包含 PC 和 Sun 系列机在内的许多计算机都有一个特殊的芯片来处理所有的浮点数计算，叫作浮点数处理器。使用 Motorola 68881 浮点芯片（用在 Sun/3 中）和 PC 上的浮点芯片进行的实际测试表明，单精度和双精度的运行速度是一样的。

幂级数

许多三角函数是用幂级数来计算的。例如，正弦函数的级数为：

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

问题是，要得到四位精度时需计算多少项？表 16-2 列出了计算 sin（π/2）的各项。

表 16-2 计算 sin（π/2）时的各项

	项	值	总计
1	x	1.571E+0	
2	$\frac{x^3}{3!}$	6.462E-1	9.248E-1
3	$\frac{x^5}{5!}$	7.974E-2	1.005E+0

表16-2 计算sin（π/2）时的各项（续）

	项	值	总计
4	$\frac{x^7}{7!}$	4.686E-3	9.998E-1
5	$\frac{x^9}{9!}$	1.606E-4	1.000E+0
6	$\frac{x^{11}}{11!}$	3.604E-6	1.000E+0

从表中可以看出，我们需要计算5项。如果我们试着计算sin（π），则得到表16-3的值。

表16-3 计算sin（π）时的各项

	项	值	总计
1	x	3.142E+0	
2	$\frac{x^3}{3!}$	5.170E+0	-2.028E+0
3	$\frac{x^5}{5!}$	2.552E-0	5.241E-1
4	$\frac{x^7}{7!}$	5.998E-1	-7.570E-2
5	$\frac{x^9}{9!}$	8.224E-2	6.542E-3
6	$\frac{x^{11}}{11!}$	7.381E-3	-8.388E-4
7	$\frac{x^{13}}{13!}$	4.671E-4	-3.717E-4
8	$\frac{x^{15}}{15!}$	2.196E-5	-3.937E-4
9	$\frac{x^{17}}{17!}$	7.970E-7	-3.929E-4
10	$\frac{x^{19}}{19!}$	2.300E-8	-3.929E-4

π需要计算9项。所以，不同的角度需要计算的项数也不同。（附录四“使用级数计算正弦的程序”中列出了计算四位精度下正弦函数的源程序。）

编译器设计人员在设计正弦函数时进退两难，如果他们提前知道要使用多少项的话，他们就可以用项数优化其算法了。但是，他们对有些角度的计算是以牺牲精度为代价的，所以要在速度和精度方面进行权衡。

不要假设数字是从计算机来的就精确，库函数也可以得出错误结果——特别是当计算很大或很小的数时。多数时候这些函数不会出问题，但你应该意识到它们的局限性。

最后的一个问题是sin（1000000）是什么？浮点数格式只适合四位数，而正弦函数是个周期函数。即，sin（0）=sin（2π）=sin（4π）……所以sin（1000000）等于sin（1000000 mod 2π）。

注意这里的浮点数格式只适合四位精度，sin（1000000）实际上是sin（1000000 ± 1000），因为1000远大于2π，所以错误致使得出的正弦函数结果没有意义

它有多热？

我在Caltech参加过两位教授主持的一个物理班。有一次，一位教授给我们做关于太阳的演讲，当他说到，"…就是说，太阳内部的温度有13000000到25000000度。"此时，另外一位导师打断了他的话，问到，"是摄氏温标还是绝对温标（绝对零度或是摄氏-273）?"

教师转向黑板，过了一分钟，他说，"这有什么不同吗？"我说这个故事的寓意在于，当你的计算可能有12000000这么大的误差时，273的差距又有何妨呢？

编程练习

练习16-1：写一个程序，使用串来表示浮点数，采用本章所用的格式，典型的串如"+1.333E+2"。程序应该包含浮点数的读、写、加、减、乘和除等函数。

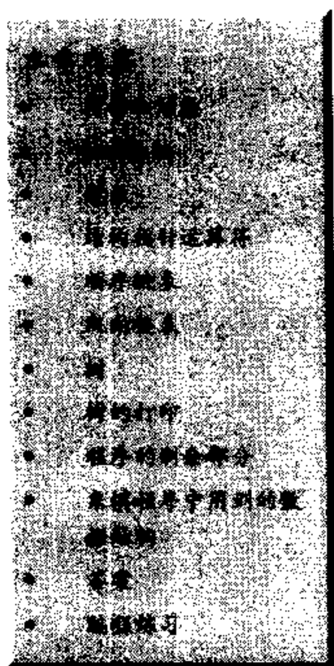
练习16-2：创建一组函数来处理定点数，定点数中小数点右边的位数是一个常量（固定的）。

第三部分

高级编程观念

本部分我们将研究C的高级特性以及高级编程任务，如模块化编程和程序移植。最后还将讨论已很少使用的特性如早期K&R C语法和C不常用的一些语法。

- 第十七章“高级指针”描述指针的高级应用，即构造动态结构如链表和树。
- 第十八章“模块化编程”描述应用模块化程序设计技术如何把程序分成几个模块，并对make实用程序做了更详尽的解释。
- 第十九章“旧式编译器”，描述老式的ANSI以前的C语言编译器。虽然这种编译器今天已很少，但在它们基础上曾写了许多代码而且还有大量的程序仍在使用旧语法。
- 第二十章“移植问题”描述移植程序时会发生的问题（从一台机器移动到另一台运行时）。
- 第二十一章“C内的角落”描述**do/while**语句，逗号运算符，及？和：运算符。
- 第二十二章“组合到一起”描述一个复杂的程序从概念到完成所需的详细步骤，重点强调了隐藏信息和模块化编程技术。
- 第二十三章“程序设计格言”列出了一些编程格言以帮助读者构建优良的C程序。



第十七章

高级指针

有这样的一个种类：

身体绑在链里，链之间相互独立，

调用每一次新的链接以向前发展。

—— Robert Buchanan

C的更有用且更复杂的特性之一就是其指针的运用。使用指针可以创建复杂的数据结构，如链表和树。图17-1列了一些数据结构。

到现在为止，接触的所有数据结构都由编译器来分配空间，不论永久变量还是临时变量。用指针可以创立和分配动态数据结构，这些数据结构可以按需要来增大或缩小。本章将学习如何使用比较常见的动态数据结构。

指针和结构

结构可以包含指针，甚至是一个指向同一结构其他实例的指针。下例中：

```
struct node {
    struct node *next_ptr;     /* Pointer to the next node */
    int value;                 /* Data for this node */
}
```

结构node见表17-2，这个结构包含两个字段、一个叫value，这里用包含数字2的部分表示，另一个字段是另一结构的指针，字段next_ptr用箭头表示。

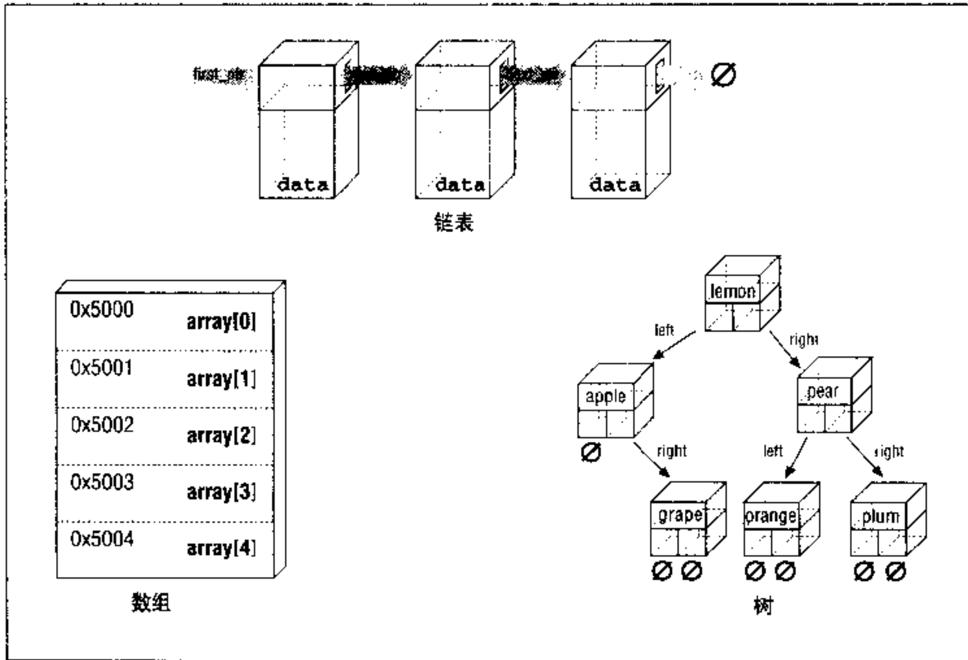


图17-1 如何使用指针

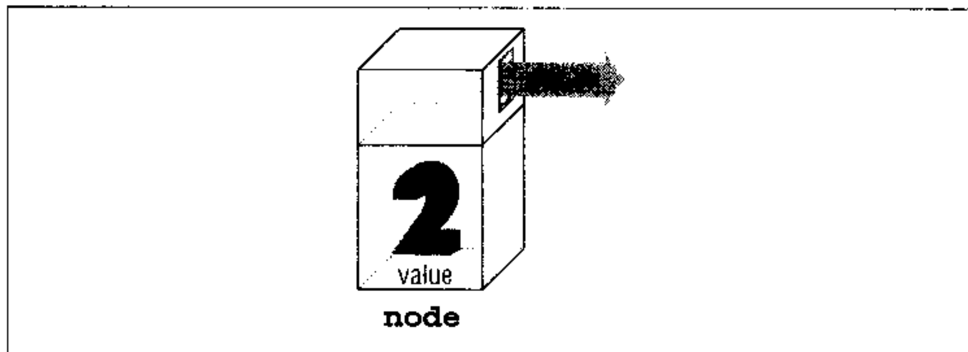


图17-2 节点

问题是：怎样创建节点（node）？可以明确地定义它们：

```
struct node *node_1;
struct node *node_2;
```

依此类推。这个结构的问题是只能定义有限的节点数，而我们需要的是一个程序，当用户说“我想要一个新节点”时，程序会创建所要的新节点。

malloc 程序能做这项工作。它为变量分配存储空间，然后返回指针。它用于在不大的地方（实际上是在一个称为堆栈的内存空间）里创建新的物体。到目前为止，我们仅用指针指向一个有名变量，所以如果使用下列语句：

```
int data;
int *number_ptr;
number_ptr = &data;
```

指针指向的就是一个有名变量（data）。函数 malloc 创建一个新的未命名的变量，返回一个指针。malloc 创建的“things”只通过指针引用，从不使用名字。

malloc 定义是：

```
void *malloc(unsigned int);
```

函数 malloc 只有一个自变量：分配的字节数。如果 malloc 用光了内存，将返回一个空指针。

在定义中，void * 用来表示 malloc 返回一个普通指针（即可以指向任何类型物体的指针）。所以 C 使用 **void** 服务于两个目的：

- 在函数定义中用其作为一种类型时，**void** 表示函数没有返回值。
- 用作指针定义时，**void** 定义一个普通指针。

通过为简单结构分配空间，便可以开始使用 malloc。随后我们将看到如何创建更大的结构，并把结构联结在一起建立很复杂的数据结构。例 17-1 为 80 个字节长的字符串（含 '\0'）分配内存。变量 string_ptr 指向这个数组。

例 17-1: 为串分配内存

```
[#include <stdlib.h>]
```

```
main()
{
    /* Pointer to a string that will be allocated from the heap */
    char *string_ptr;

    string_ptr = malloc(80);
```

假设我们正使用一个复杂的数据库，它含有邮件列表（还含其他项目）。结构person用来记录每个人的数据：

```
struct person {
    char    name[30];           /* name of the person */
    char    address[30];        /* where he lives */
    char    city_state_zip[30]; /* Part 2 of address */
    int     age;                /* his age */
    float   height;             /* his height in inches */
}
```

可以用数组来存储邮件列表，但数组是内存的一种低效率用法，每个元素都占用空间，不管它是不是被使用。需要的是一种只为那些要用的元素分配空间的方法，在需要的基础上，我们可以使用malloc。

要创建一个新person，可以使用代码：

```
/* Pointer to a person structure to be allocated from the heap */
struct person *new_item_ptr;

new_item_ptr = malloc(sizeof(struct person));
```

通过表达式sizeof (struct person) 来确定分配的字节数。如果没有sizeof运算符，我们就得完成一个难度大且易出错的操作：自己统计结构的字节数。

堆栈的大小虽然很大，但毕竟有限，malloc用完内存后将返回NULL指针。好的程序方法应告诉你检查每个malloc调用的返回值，以确保真的得到了内存。

```
new_item_ptr = malloc(sizeof(struct person));
if (new_item_ptr == NULL) {
    fprintf(stderr, "Out of memory\n");
```

```
        exit (8);
    }
```

虽然检查malloc的返回值是个好的编程方法，但这种检查经常会被省略掉，不管是不是真的获得了内存，程序员都假设得到了内存，结果当内存不足时许多程序出了问题。

这个问题如此严重，因此在设计C++的时候，就包含了一个内存不足条件下的错误处理机制。

free 函数

malloc 函数从堆栈中得到内存。要想在处理结束后释放内存，得使用 free 函数。free 函数的一般形式是：

```
free(pointer);
pointer = NULL;
```

其中*pointer*是前面malloc分配的指针（不一定非把指针设为NULL；不过设其为NULL时可以阻止使用释放后的内存）。

下例用malloc得到内存并释放它：

```
const int DATA_SIZE = (16 * 1024); /* Number of bytes in the buffer */
void copy(void)
{
    char *data_ptr;         /* Pointer to large data buffer */
    data_ptr = malloc(DATA_SIZE);        /* Get the buffer */
    /*
     * Use the data buffer to copy a file
     */
    free(data_ptr);
    data_ptr = NULL;
}
```

但如果忘记了释放指针会发生什么呢？缓冲区会死掉。也就是说，内存管理系统认为缓冲区正在被使用，但是实际上没有程序在用它。如果 free 语句从函数

copy中被删掉了，那么每次连续调用都会吃掉另外16K的内存，经常这样做，你的程序就会把内存耗光。

当使用已被释放的内存时，可能会发生其它的问题。调用free时，内存被返回给内存区并能再次使用。Free调用后使用指针，会发生类似于一个数组下标超界的错误，因为你使用的是属于别的程序的内存，这可能会得出意想不到的结果，或者使程序崩溃。

链表

假设你在写一个程序，要展示一系列闪烁卡片作为教学训练。问题是你事先不知道用户会提供多少张卡片，解决的办法之一是使用数据结构链表，用这种方法，列表可以随卡片的增加而变大。同时还将看到，链表可以和其他数据结构结合来处理极端复杂的数据。

链表是一连串的项，每一项都指向串中的下一项。想想儿时玩的找财宝的游戏。你得到第一条提示说，"在邮箱中找"。你跑到邮箱前，又发现了下一条线索，"在后院的大树下找"。这样一直下去，直到你找到财宝（或者迷路）为止。在这个游戏中，每条线索都指向下一条（注1）。

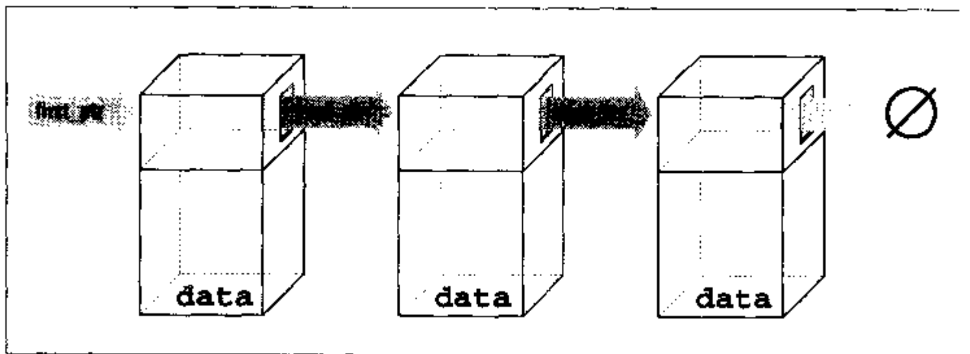


图17-3 一个链表

注1： 一位妇女在家中为女儿的童子军进行找财宝训练，这时门铃响了，是邮递员，他说"夫人，我找了大树后面，鸟窝中也找了，甚至连席子下面也看了，可还是找不到你的信。"

链表的结构定义如下:

```
struct linked_list {
    char    data[30];             /* data in this element */
    struct linked_list *next_ptr; /* pointer to next element */
};
struct linked_list *first_ptr = NULL;
```

变量first_ptr指向列表的第一个元素。起初，在向表（它是空的）中插入元素之前，该变量被初始化为NULL。

图17-4中，创建了一个新元素，然后插在一个已存在表的开头。C中要往链表中插入新元素，应执行下列步骤:

1. 为项目建立结构。

```
new_item_ptr = malloc(sizeof(struct linked_list));
```

2. 把项目存储在新元素中。

```
(*new_item_ptr).data = item;
```

3. 使表中第一元素指向新元素。

```
(*new_item_ptr).next_ptr = first_ptr;
```

4. 新元素现在成为第一元素。

```
first_ptr = new_item_ptr;
```

实际程序的代码如下:

```
void add_list(char *item)
{
    /* pointer to the next item in the list */
    struct linked_list *new_item_ptr;

    new_item_ptr = malloc(sizeof(struct linked_list));
    strcpy((*new_item_ptr).data, item);
    (*new_item_ptr).next_ptr = first_ptr;
    first ptr = new_item_ptr;
}
```

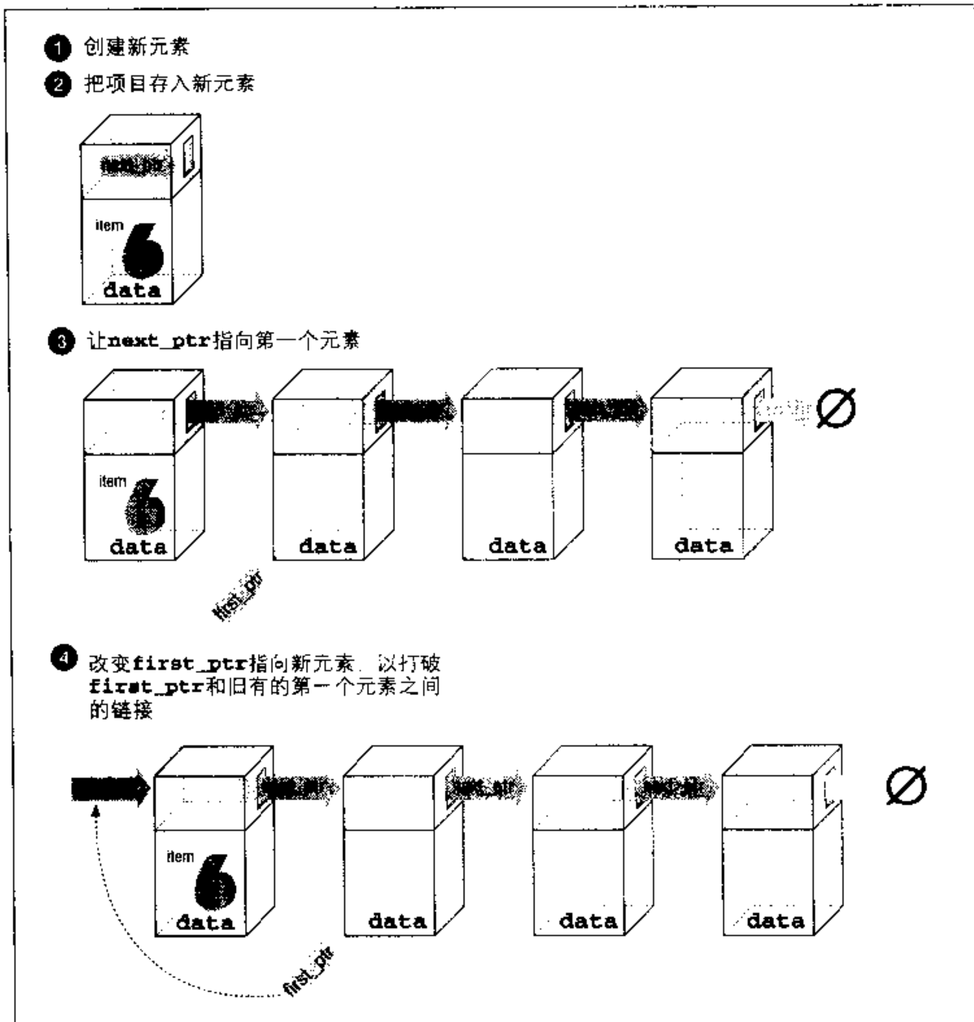


图 17-4 在表开头加入新元素

要看该名称是否在表中，必须查找表中的每一个元素直到找到该名称或者运行完所有的数据。例 17-2 包含一个 find 程序，会在表中进行项目查找。

例 17-2: find/find.c

```
[File: find/find.c]
#include <stdio.h>
#include <string.h>
```

```
struct linked_list {
    struct linked_list *next_ptr;      /* Next item in the list */
    char *data;                        /* Data for the list */
};

struct linked_list *first_ptr;
/*******************************************************
 * find   Looks for a data item in the list.           *
 *                                                     *
 * Parameters                                          *
 *      name   Name to look for in the list.           *
 *                                                     *
 * Returns                                             *
 *      1 if name is found.                            *
 *      0 if name is not found.                        *
 *******************************************************/
int find(char *name)
{
    /* current structure we are looking at */
    struct linked_list *current_ptr;

    current_ptr = first_ptr;

    while ((strcmp(current_ptr->data, name) != 0) &&
           (current_ptr != NULL))
        current_ptr = (*current_ptr)->next_ptr;

    /*
     * If current_ptr is null, we fell off the end of the list and
     * didn't find the name
     */
    return (current_ptr != NULL);
}
```

问题17-1：为什么运行这个程序有时会导致一个错误？但有时，对不存在的项目它又会返回“1”？

结构指针运算符

在find程序中，使用了一个很笨的写法（*current_ptr）.data来访问结构中的数据字段。C提供了一种简便写法，即使用->运算符，点（.）运算符表示结

构中的字段，-> 表示字段结构指针。

下面两个表达式是等价的：

```
(*current_ptr).data = value;
current_ptr->data = value;
```

顺序链表

至此，我们只在链表的开头加了新元素。假定想按顺序加元素，表17-5是顺序链表的一个例子。

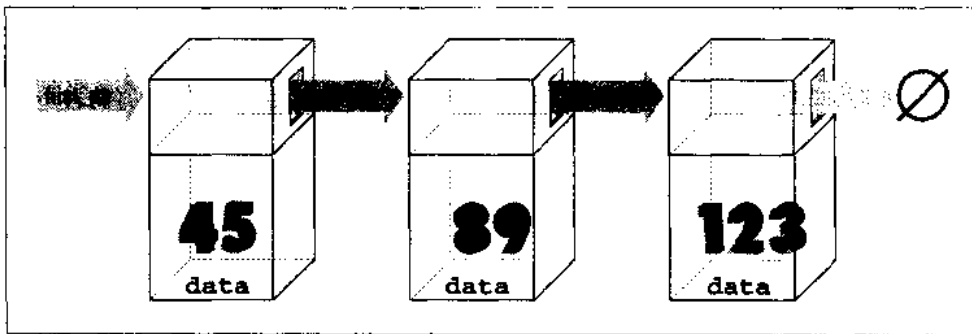


图 17-5 顺序链表

例17-3中的子程序实现了这个函数。第一步是确定要插入的位置，head_ptr指向表的第一个元素，程序在表中移动变量before_ptr直到它找到插入的正确位置。变量after_ptr指向前一个位置的指针，要插入的新元素位于这两个元素之间。

例 17-3: list/list.p1

```
void enter(struct item *first_ptr, const int value)
{
    struct item *before_ptr;          /* Item before this one */
    struct item *after_ptr;           /* Item after this one */
    struct item *new_item_ptr;        /* Item to add */
```

```
    /* Create new item to add to the list */

    before_ptr = first_ptr;             /* Start at the beginning */
    after_ptr =  before_ptr->next_ptr;

    while (1) {
        if (after_ptr == NULL)
            break;

        if (after_ptr->value >= value)
            break;

        /* Advance the pointers */
        after_ptr = after_ptr->next_ptr;
        before_ptr = before_ptr->next_ptr;
    }
```

图 17-6 中，已经定位了 before_ptr 从而使它指向插入位置的前一个元素，变量after_ptr指向插入位置之后的元素。换句话说，将把新元素放在before_ptr 和 after_ptr 之间。

现在正确的插入位置已经定位，要做的就是建立新元素并链接进来：

```
[File: list/list.p2]
    new_item_ptr = malloc(sizeof(struct item));
    new_item_ptr->value = value;        /* Set value of item */

    before_ptr->next_ptr = new_item_ptr;
    new_item_ptr->next_ptr = after_ptr;
}
```

新元素现在可以链入。所做的第一个链接是在由 before_ptr 指向的元素（45 号）和新元素 new_item_ptr（53 号）之间。用下列语句完成：

```
before_ptr->next_ptr = new_item_ptr;
```

接着必须把新元素new_item_ptr（53号）链接到after_ptr指向的元素（89号）。由下列代码完成：

```
new_item_ptr->next_ptr = after_ptr;
```

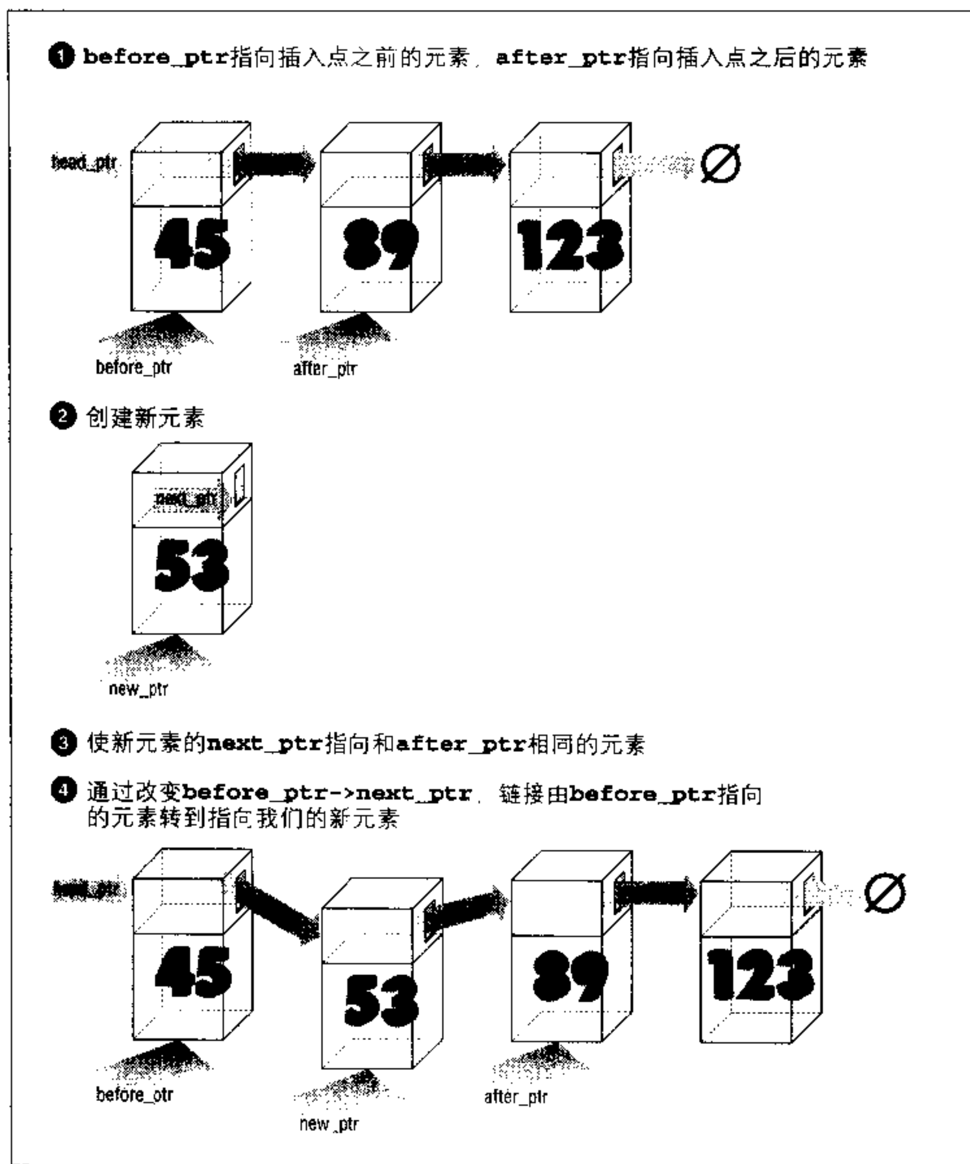


图17-6 表的顺序插入

双向链表

双向链表包含两种链接：一种链接指向后一个元素；另一种链接指向前一个元素。双向链表的结构是：

```
struct double_list {
    int data;                       /* data item */
    struct  double_list *next_ptr;     /* forward link */
    struct  double_list *previous_ptr;/* backward link */
};
```

双向链表如图17-7所示。除有两个链接一前一后外，与单向链表相似。向表中插入新元素所要求的四步见图17-8、图17-9、图17-10和图17-11。

向这个表插入新元素的程序代码是：

```
void double_enter(struct double_list *head_ptr, int item)
{
    struct list *insert_ptr; /* insert before this element */
    /*
     * Warning: This routine does not take
     *   care of the case in which the element is
     *   inserted at the head of the list
     *   or the end of the list
     */
    insert_ptr = head_ptr;
    while (1) {
        insert_ptr = insert_ptr->next;
        /* have we reached the end */
        if (insert_ptr == NULL)
            break;
        /* have we reached the right place */
        if (item >= insert_ptr->data)
            break;
    }
```

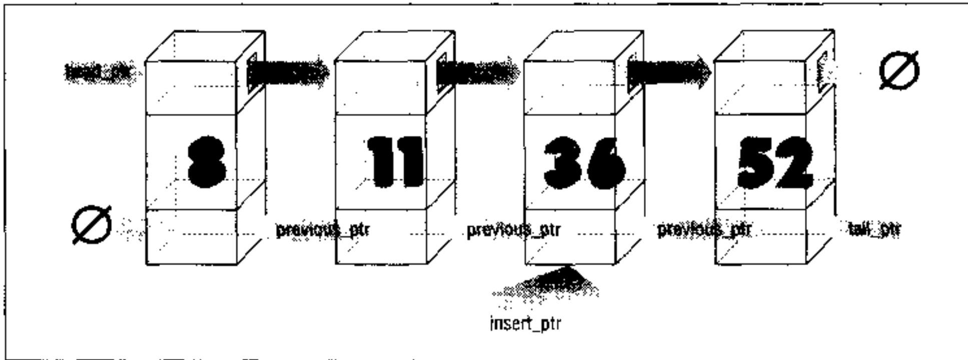


图 17-7　双向链表

下面详细检查一下，首先用下列代码设置新元素的后链接：

```
new_item_ptr->next_ptr = insert_ptr;
```

列示见图 17-8。

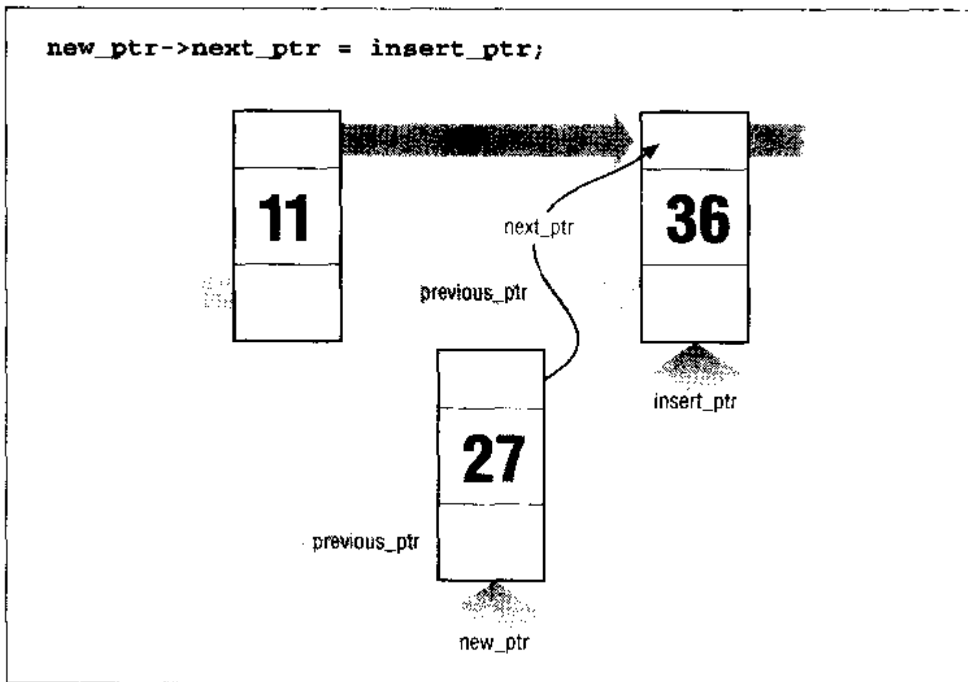


图 17-8　双向链接插入：第 1 步

现在要注意前向指针（new_item_ptr->previous_ptr）。由下列语句完成：

```
new_item_ptr->previous_ptr = insert_ptr->previous_ptr;
```

和单向链接不同，没有before_ptr指向插入点之前的元素。相反，我们使用insert_ptr->previous_ptr指向这个元素，现在链表如图17-9所示。

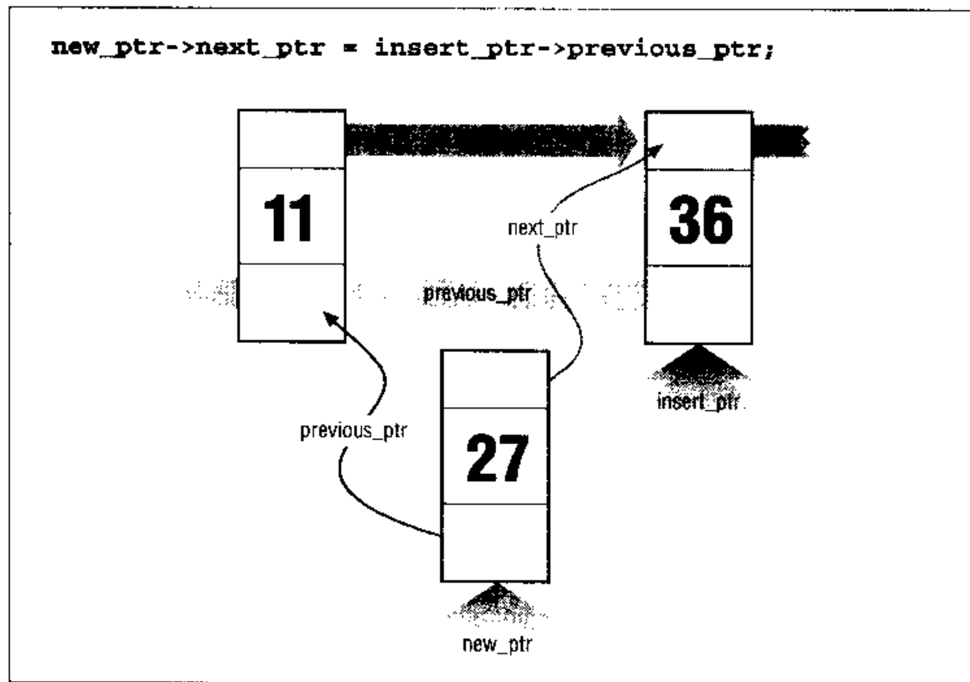


图17-9 双向链接插入：第2步

新元素已建立了正确的链接，而旧元素的链接（数字11和36）还需要调整。首先调整11号元素中的next_ptr字段，找到11元素需要一点技巧。我们从insert_ptr（36号元素）开始沿着链previous_ptr到11号元素，想要改变这个元素中的next_ptr字段，代码是：

```
insert_ptr->previous_ptr->next_ptr = new_ptr;
```

新链接见图17-10。

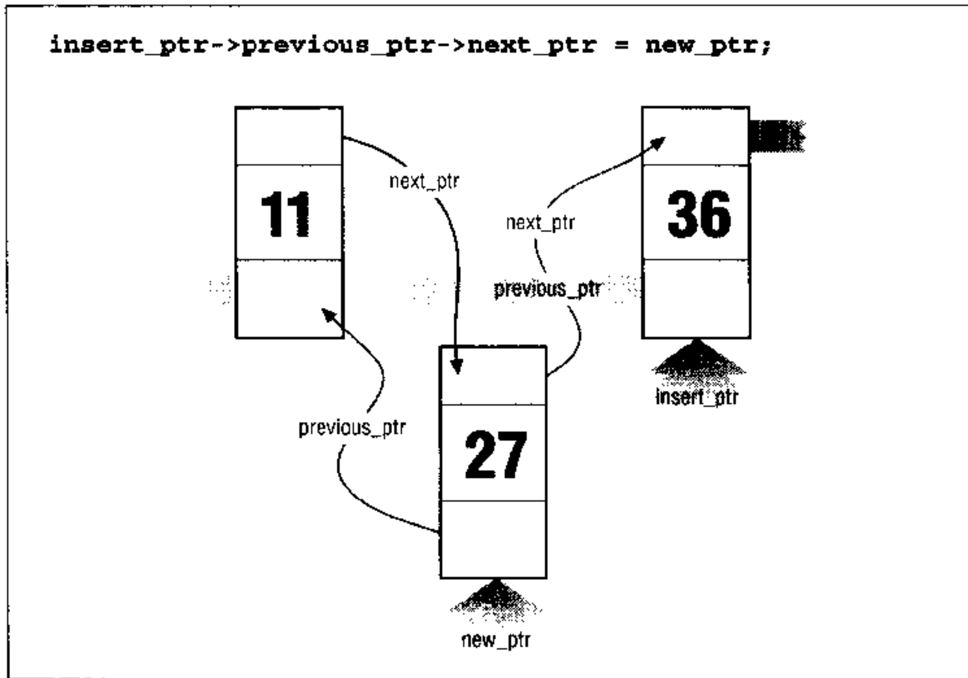


图 17-10 双向链接插入，第三步

四个链接已做了三个。最后一个是第36号元素的previous_ptr，这个链接由下面的代码设置：

```
insert_ptr->previous_ptr = new_item_ptr;
```

双向链接的最后版本如图 17-11 所示。

树

假设想创建一个出现在一篇文件中所有单词的字母表，可以用一个链表来实现。但是链表的查找很慢，因为必须检查每个元素，直到找到正确的插入位置为止。如使用一种称为树的数据类型，可以减少比较的次数。一个二叉树结构如表 17-12 所示。

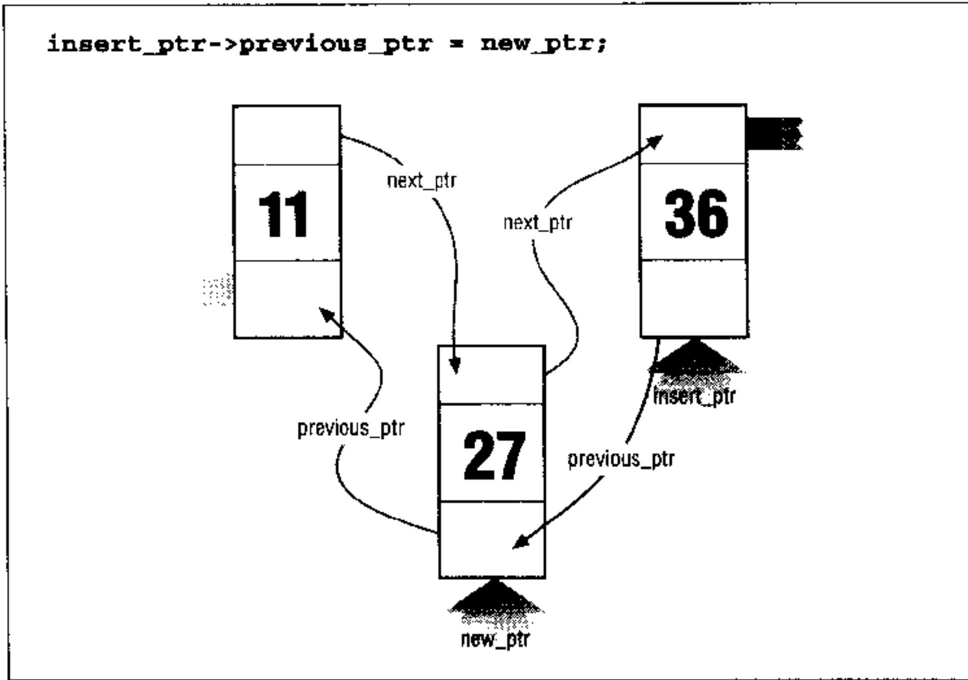


图 17-11 双向链接插入，第四步

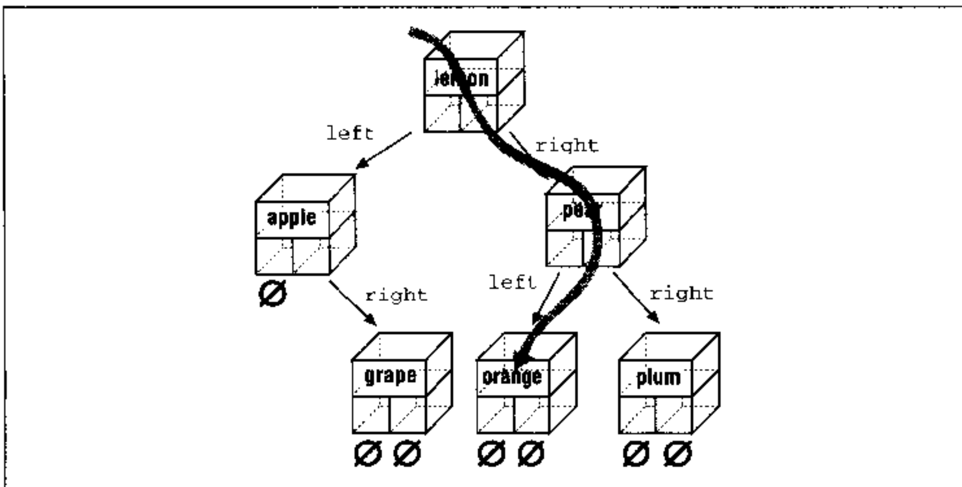


图 17-12 树

每个框叫做树的一个节点。最上边的框叫根，底层的框叫叶。每个节点有两个指针：一个左指针和一个右指针，它们分别指向左右子树。

树的结构是：

```
struct node {
    char    *data;          /* word for this tree */
    struct node *left;      /* tree to the left */
    struct node *right;     /* tree to the right */
};
```

树经常用来存储一张符号表，即程序中使用变量的列表。本章将用树来存储一个单词表，然后按字母顺序打印该表。树优于链表的特性是，它的查找时间明显地减少。

在这个例子中，每个节点存储一个单词。左子树存储所有小于当前单词的单词，而右子树存储所有比当前单词大的单词。

例如，图17-13显示了怎样降序查找“orange”。从根“lemon”开始，因为“orange” > “lemon”，向下到右链，找到“pear”。因为“orange” < “pear”，又沿左链向下，找到“orange”。

递归很适用于树，递归的规则是：

1. 函数必须使事情变得更简单。树也符合这个规则，因为当你从各个层次下降的时候要找的东西的越来越少。
2. 必须有结束点。树提供两个结束点，要么是找到了要找的数据，要么得到空节点。

在树中插入一个字的算法是：

1. 如果是空树或空子树，建立单节点的树并放入单词。
2. 如果当前节点含有这个单词，不做处理。
3. 否则对“insert word”执行递归调用，把单词依据其大小插入左或右子树。

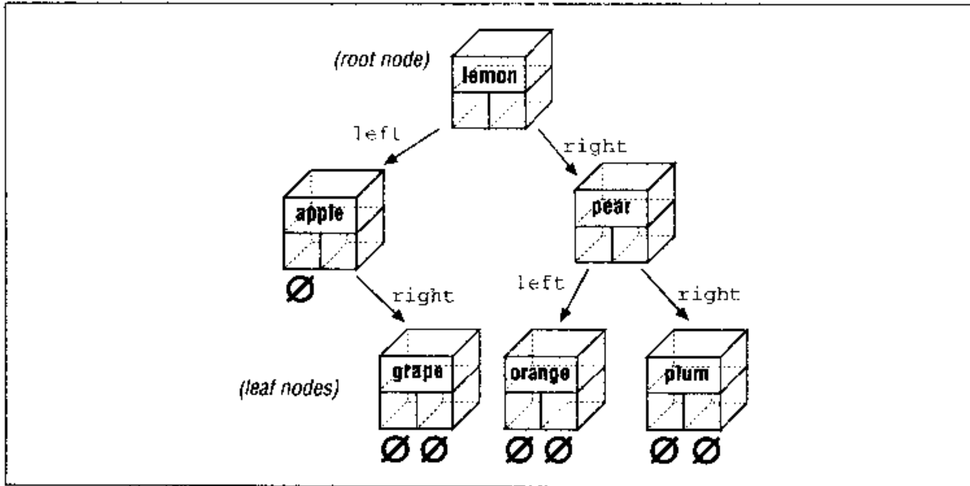


图 17-13 树查找

以图 17-13 在树中插入单词“fig”为例来看算法的运行原理。首先，比较“fig”与“lemon”，“fig”小一些，所以接下来与“apple”比较；因为“fig”比“apple”大一些，所以继续和“grape”比较，因为“fig”比“grape”小，结果试和左链比较，左链值为 NULL，所以建立一个新节点。向树中输入值的函数是：

```
void enter(struct node **node, char *word)
{
    int  result;                  /* result of strcmp */
    char *save_string();          /* save a string on the heap */
    void memory_error();          /* tell user no more room */

    /*
     * If the current node is null, then we have reached the bottom
     * of the tree and must create a new node
     */
    if ((*node) == NULL) {

        /* Allocate memory for a new node */
        (*node) = malloc(sizeof(struct node));
        if ((*node) == NULL)
            memory_error();

        /* Initialize the new node */
        (*node)->left = NULL;
```

```
        (*node)->right = NULL;
        (*node)->word = save_string(word);
        return;
    }

    /* Check to see where our word goes */
    result = strcmp((*node)->word, word);

    /* The current node
   * already contains the word,
   * no entry necessary */
    if (result == 0)
        return;

    /* The word must be entered in the left or right subtree */
    if (result < 0)
        enter(&(*node)->right, word);
    else
        enter(&(*node)->left, word);
}
```

该函数把一个指针传给树根，如果根是NULL，建立一个节点。因为正在改变指针的值，所以必须传递一个指针给该指针。(传递前一层次的指针是因为它是函数外的变量类型；传递第二个层次的指针是因为要改变它。)

树的打印

尽管树的结构复杂，打印却很容易。这里还是用递归，打印算法是：

1. 空树，什么也不打印。
2. 打印该节点之前的数据（左树）、然后打印这个节点，打印节点以后的数据（右树）。

Print_tree的代码是：

```
void print_tree(struct node *top)
{
    if (top == NULL)
```

```
        return;                 /* short tree */
    print_tree(top->left);
    printf("%s\n", top->word);
    print_tree(top->right);
}
```

程序的剩余部分

现在已经定义了数据结构，余下的还有一些函数。Main 函数检查自变量的正确数目，然后调用扫描程序（scanner）和 print_tree 程序。

scan 函数读取文件并把它分成单词，该函数使用标准宏 isalpha。这个宏定义在标准头文件 *ctype.h* 中，如果自变量是一个字母则返回非零值，否则返回 0 。找到一个单词以后，调用函数 enter 并把单词放在树里。

save_string 在堆栈中为一个串建立空间，并返回指向它的指针。

如果 malloc 失败则调用 memory_error 并退出。但发生内存不足的错误时，程序将写一条错误信息并退出。

例 17-4 列出了 *words.c*。

例 17-4: words/words.c

```
[File: words/words.c]
/********************************************************
 * words --Scan a file and print out a list of words    *
 *              in ASCII order.                         *
 *                                                      *
 * Usage:                                               *
 *      words <file>                                    *
 ********************************************************/
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

struct node {
    struct node    *left;       /* tree to the left */
```

```
    struct node     *right;        /* tree to the right */
    char            *word;         /* word for this tree */
};

/* the top of the tree */
static struct node *root = NULL;

/********************************************************
 * memory_error -- Writes error and dies.               *
 ********************************************************/
void memory_error(void)
{
    fprintf(stderr, "Error:Out of memory\n");
    exit(8);
}

/********************************************************
 * save_string -- Saves a string on the heap.           *
 *                                                      *
 * Parameters                                           *
 *      string -- String to save.                       *
 *                                                      *
 * Returns                                              *
 *      pointer to malloc-ed section of memory with     *
 *      the string copied into it.                      *
 ********************************************************/
char *save_string(char *string)
{
    char *new_string;   /* where we are going to put string */

    new_string = malloc((unsigned) (strlen(string) + 1));

    if (new_string == NULL)
        memory_error();

    strcpy(new_string, string);
    return (new_string);
}
/********************************************************
 * enter -- Enters a word into the tree.                *
 *                                                      *
 * Parameters                                           *
 *      node -- Current node we are looking at.         *
```

```
 *      word -- Word to enter.                                *
 *********************************************************/
void enter(struct node **node, char *word)
{
    int  result;        /* result of strcmp */

    char *save_string(char *);  /* save a string on the heap */

    /*
     * If the current node is null, we have reached the bottom
     * of the tree and must create a new node.
     */
    if ((*node) == NULL) {

        /* Allocate memory for a new node */
        (*node) = malloc(sizeof(struct node));
        if ((*node) == NULL)
            memory_error();

        /* Initialize the new node */
        (*node)->left = NULL;
        (*node)->right = NULL;
        (*node)->word = save_string(word);
        return;
    }
    /* Check to see where the word goes */
    result = strcmp((*node)->word, word);

    /* The current node already contains the word, no entry necessary */
    if (result == 0)
        return;

    /* The word must be entered in the left or right subtree */
    if (result < 0)
        enter(&(*node)->right, word);
    else
        enter(&(*node)->left, word);
}
/*********************************************************
 * scan -- Scans the file for words.                      *
 *                                                        *
 * Parameters                                             *
 *      name -- Name of the file to scan.                 *
 *********************************************************/
```

```
void scan(char *name)
{
    char word[100];      /* word we are working on */
    int  index;          /* index into the word */
    int  ch;             /* current character */
    FILE *in_file;       /* input file */

    in_file = fopen(name, "r");
    if (in_file == NULL) {
        fprintf(stderr, "Error:Unable to open %s\n", name);
        exit(8);
    }
    while (1) {
        /* scan past the whitespace */
        while (1) {
            ch = fgetc(in_file);

            if (isalpha(ch) || (ch == EOF))
                break;
        }

        if (ch == EOF)
            break;

        word[0] = ch;
        for (index = 1; index < sizeof(word); ++index) {
            ch = fgetc(in_file);
            if (!isalpha(ch))
                break;
            word[index] = ch;
        }
        /* put a null on the end */
        word[index] = '\0';

        enter(&root, word);
    }
    fclose(in_file);
}
/********************************************************
 * print_tree -- Prints out the words in a tree.        *
 *                                                      *
 * Parameters                                           *
 *      top -- The root of the tree to print.           *
 ********************************************************/
```

```
void print_tree(struct node *top)
{
    if (top == NULL)
        return;                 /* short tree */

    print_tree(top->left);
    printf("%s\n", top->word);
    print_tree(top->right);
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Error:Wrong number of parameters\n");
        fprintf(stderr, "      on the command line\n");
        fprintf(stderr, "Usage is:\n");
        fprintf(stderr, "    words 'file'\n");
        exit(8);
    }
    scan(argv[1]);
    print_tree(root);
    return (0);
}
```

问题17-2：有一次我编写了一个程序，它使用树结构把字典读入内存，然后在程序中用树结构查找拼错的单词。虽然认为树的操作很快，但是这个程序运行得非常慢，你会误认为我使用的是链表，为什么？

提示：画图用“able”、“baker”、“cook”、“delta”和“easy”构造一棵树，看看结果如何。

象棋程序中用到的数据结构

人工智能中经典的问题是象棋游戏。在本书付印期间，去年打败了世界上玩象棋玩得最好的计算机特级大师，今年却被计算机打败了（1997）。

我们将为象棋程序设计一个数据结构。下棋时，总有几步棋可走，你的对手可能也有许多步，对每一步，你又有许多下法。依此类推，来来回回就有了下法的不同层次。

这个数据结构开始就很像是一棵树，但它不是二叉树，因为每一个节点都有两个以上的分支，如图17-14所示。

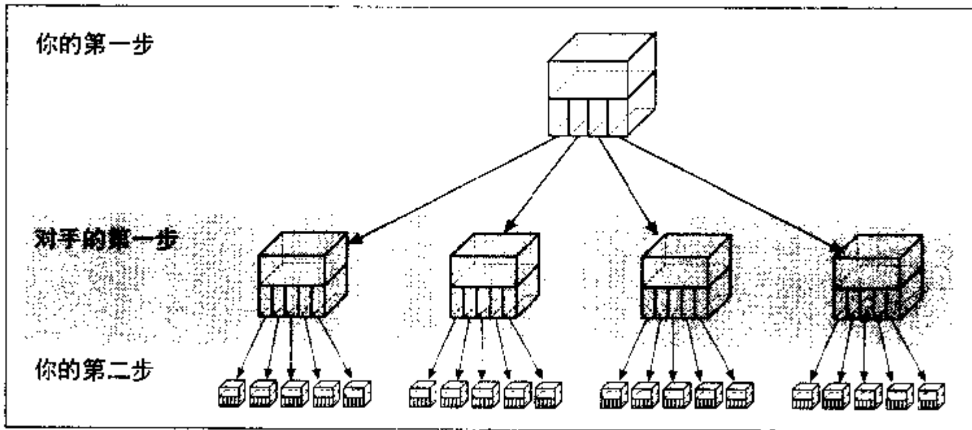


图17-14 象棋树

我们希望使用下面的数据结构：

```
struct chess {
    struct board board;     /* Current board position */
    struct next {
        struct move;        /* Our next move */
        struct *chess_ptr;  /* Pointer to the resulting position */
    } next[MAX_MOVES];
};
```

问题是，对一个给定的棋局，可以走动的数目是不定的。例如，开局时你有很多步可走（注2）。比如车、后和象可以走到一条直线上的任何格子内。当你到残局时（在一场势均力敌的比赛中），双方可能只剩下几个兵和一个主要的棋子，可走的选择就大大减少了。

我们想尽可能地提高存储效率，因为下棋程序已用到了机器的极限。把下一步可走的棋数变为链表结构，就可以减少内存的需求。最终的结构为：

注2：　琐碎的问题：下棋开盘时，你可以走的21步棋是什么？你可以让每个兵向前走一步（8种）或两步（又有8种），马可以向左向右跳起（又有4种）(8+8+4=20)。第21步棋是什么？

```
struct next {
    struct move this_mode;        /* Our next move */
    struct *chess_ptr;            /* Pointer to the resulting position */
};
struct chess {
    struct board board;           /* Current board position */
    struct next *list_ptr;        /* List of moves we can make from here */
    struct move this_move;        /* The move we are making */
};
```

我们用图 17-15 来表示这个结构。

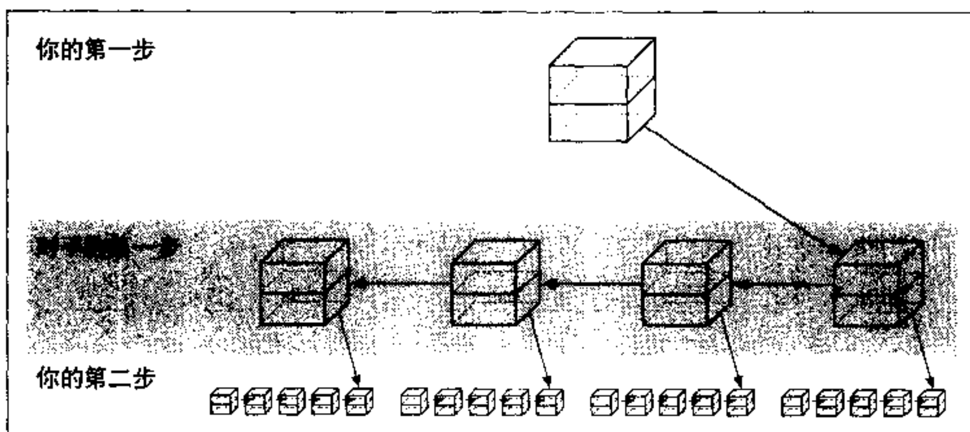


图 17-15 修订后的象棋结构

新版本增加了一些复杂度，但是节省了大量的存储空间。在前一个版本中，必须给所有可能的下法的指针分配内存。如果只有几种可能的下法，未使用的下法的指针就浪费了许多内存。而使用链表，可在需要的基础上分配内存，所以如果有 30 种可能的下法，表长就为 30；但如果只有 3 种可能的下法，表长就为 3。表只在需要时才增加，致使内存的使用更有效。

答案

解答 17-1： 问题出在下面这条语句：

```
while ((strcmp(current_ptr->data, name) != 0) &&
```

```
    (current_ptr != NULL))
```

在检查current_ptr是不是无效指针（!=NULL）之前，就先检查了current_ptr->data。如果指针是NULL，就可能检查内存中的一个随机地址，其中可能包含任何信息。解决的办法是在检查current_ptr指向目标之前，先检查它是否为空：

```
while (current_ptr != NULL) {
    if (strcmp(current_ptr->data, name) == 0)
        break;
}
```

解答17-2：问题出在：因为字典中的第一个单词总是最小，所以所有其它的单词都排在右子树上。事实上，因为整个链表是有序的，所以只用了右子树。虽然定义的是树结构，但结果还是一个链表，见图17-16。更详细讨论数据结构的书，如Niklaus Wirth写的《Algorithms +Data Structures = Programs》（《算法＋数据结构＝程序》），讨论了通过平衡二叉树避免这种情况产生的方法。

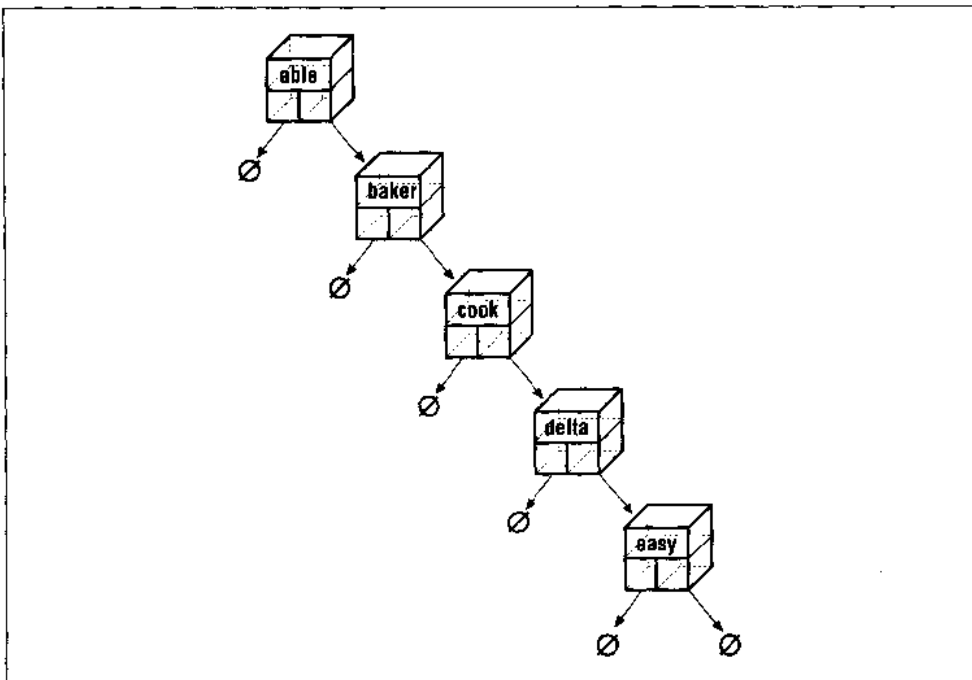


图17-16 不平衡的树

琐碎问题的回答：你可以放弃。对了，第21步棋表示认输。

编程练习

练习 17-1：编写一个交叉引用程序。

练习 17-2：编写一个删除链表中的一个元素的函数。

练习 17-3：编写一个删除双向链表中的一个元素的函数。

练习 17-4：编写一个删除树中的一个元素的函数。

第十八章

模块化编程

本章内容

- 模块
- 公共和专用
- extern 修饰符
- 头文件
- 模块体
- 使用无限数组的程序
- 用于多文件的 Makefile
- 使用无限数组
- 把一项任务分成模块
- 模块划分举例：文本编辑器
- 编译器
- 电子表格
- 模块设计准则
- 编程练习

人多好干活。

—— John Heywood

到目前为止，我们处理的都是小程序。当程序变得越来越大时，把它们分割成部分或模块就变得很必要了。C允许把程序分成多个文件，并分别对它们进行编译，然后再组合（或连接）为一个程序。

本章将仔细检查一个编程实例，讨论用C建立好模块的方法。另外还将介绍使用make把这些模块放在一起组成一个程序的方法。

模块

模块是执行相关任务的函数集合。例如，处理 `lookup`、`enter` 和 `sort` 的数据库函数可以组成一个模块。另外还有处理复杂数学运算的模块等等。而且，随着

编程问题的增多，将需要更多的程序员来完成它们。分割大项目的有效方法是给每个程序员分配不同的模块，这样，每个程序员只考虑特定模块的内部细节就行了。

本章将讨论处理无限数组的模块。这个程序包中的函数允许用户把数据存入数组而不必担心它的大小，无限数组按要求增大（只受计算机内存限制）。无限数组中可存储用于直方图的数据，但也可存储交叉引用程序中的行数或其他类型的数据。

公用和专用

模块分为两部分：公用和专用。公用部分告诉用户如何调用模块中的函数，它包含可以在模块外使用的数据结构和函数的定义。这些定义放在一个头文件中，而这个头文件必须包含在依赖该模块的任意程序中。在无限数组的例子中，我们把公共定义放在文件ia.h中，你很快就会看到。图18-1列示了无限数组包内各个部分之间的关系。

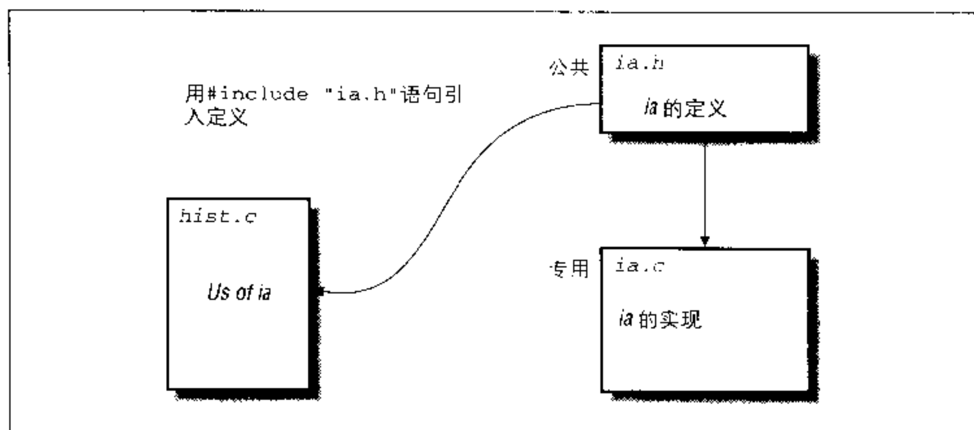


图 18-1　无限数组的定义、实施和使用

模块内部的所有东西都是专用的，外面不能直接使用的都应放在专用模块中。

C++和C相比优点之一是C++可以明确定义什么是公用的以及什么专用的，并且可以阻止对专用数据的非法修改。

extern 修饰符

extern 修饰符用来表示变量或函数定义于当前文件之外。例如，看两个文件 *main.c* 和 *count.c* 的内容。

main.c 文件

```
#include <stdio.h>
/* number of times through the loop */
extern int counter;

/* routine to increment the counter */
extern void inc_counter (void);

main ()
{
    int   index; /* loop index */

    for  (index = 0; index < 10; index++)
        inc_counter ();
    printf ("Counter is %d\n", counter);
    return  (0);
}
```

count.c 文件

```
/* number of times through the loop */
int      counter = 0;

/* trivial example */
void     inc_counter (void)
{
    ++counter;
}
```

本例中，函数main使用变量counter。*Main.c* 使用**extern**定义表示counter定义于函数之外；本例counter是定义在*counter.c* 文件中。修饰符**extern**不用于 *counter.c*，原因是它包含变量的“真正”定义。

有三个修饰符可以用来表示变量定义的位置，如表18-1所示。

表 18-1 修饰符

修饰符	意义
extern	变量/函数定义于其他文件。
<空>	变量/函数定义于本文件（公用）且可以用于其他文件。
static	变量/函数对此文件来说是局部的（专用）。

注意单词**static**有两重意思。对全局定义的数据来说，它意味着“对此文件专用”。对在一个函数中定义的数据来说，它是指“从静态内存（而不是从临时堆栈）分配的变量”。

C在**static**和**extern**修饰符的使用规则上非常自由，你可以在一个程序开头把变量定义为**extern**，随后不使用修饰符就能定义它：

```
extern sam;
int sam = 1;     /* this is legal */
```

当把所有外部变量定义于一个头文件中时，这种方法十分有用。程序引用这个头文件（并把变量定义为**extern**），然后定义真正的变量。

另一个相关的问题是在两个不同的文件里定义一个变量：

main.c 文件

```
int     flag  = 0;      /* flag is off */
main ()
{
    printf (  lag is %d\n   flag);
}
```

sub.c 文件

```
int     flag = 1;       /* flag is on */
```

此时会出现什么情况呢？

1. 由于先调入main.c，flag将被初始化为0。

2. 由于 *sub.c* 中的条目会覆盖 *main.c* 中的条目，flag 将被初始化为 1。
3. 编译器将十分仔细地分析两个程序，然后选出值可能为错的那个。

只有一个全局变量flag，它是被初始化为1还是0取决于编译器的猜测。当全局变量被两次定义时，一些更高级的编译器会输出错误信息，但多数编译器会忽略掉这个错误。即便我们把flag初始化为0且在显示之前未作变动，Main 程序也完全有可能输出：

```
flag is 1
```

为避免隐藏初始化的问题，使用关键字static把每一变量的范围限制于它所定义的文件中。

如果写了下列代码：

main.c 文件

```
static int      flag  = 0;       /* flag is off */
main ()
{
        printf ("Flag is %d\n", flag);
}
```

Sub.c 文件

```
static int      flag = 1;        /* flag is on */
```

那么 *main.c* 中的 flag 就和 *sub.c* 中的 flag 是完全不同的一个变量，但你还是应该使变量的命名不同，以避免混乱。

头文件

模块之间共享的信息应放在头文件中，按照惯例，所有头文件名应以“*.h*”结尾，我们的无限数组例子中，使用的是文件 *ia.h*。

头文件应该包含所有的公用信息，如：

- 清楚地描述模块做什么以及能为用户提供什么功能的注释块
- 一般常量
- 一般结构
- 所有公用函数原型
- 公用变量的 extern 定义

在无限数组例子中，文件 *ia.h* 中大部分是注释。这些注释并不多余，真正的代码放在程序文件 *ia.c* 中。*ia.h* 文件对外部来说既是程序文件，又是文档文件。

注意 *ia.h* 的注释中没有提及如何实现无限数组。在这一级别上，我们并不关心事情是如何完成的，而是只想知道可以用哪些函数。

例 18-1: ia.h 文件

```
/********************************************************
 * Definitions for the infinite array (ia) package.     *
 *                                                      *
 * An infinite array is an array whose size can grow    *
 * as needed.  Adding more elements to the array        *
 * will just cause it to grow.                          *
 *------------------------------------------------------*
 * struct infinite_array                                *
 *      Used to hold the information for an infinite    *
 *      array.                                          *
 *------------------------------------------------------*
 * Routines                                             *
 *                                                      *
 *      ia_init -- Initializes the array.               *
 *      ia_store -- Stores an element in the array.     *
 *      ia_get -- Gets an element from the array.       *
 ********************************************************/

/* number of elements to store in each cell of the infinite array */
#define BLOCK_SIZE      10

struct infinite_array {
    /* the data for this block */
```

```
    float   data[BLOCK_SIZE];

    /* pointer to the next array */
    struct infinite_array *next;
};

/*******************************************************
 * ia_init -- Initializes the infinite array.          *
 *                                                     *
 * Parameters                                          *
 *      array_ptr -- The array to initialize.          *
 *******************************************************/
#define ia_init (array_ptr)       {(array_ptr) ->next = NULL;}

/*******************************************************
 * ia_get -- Gets an element from an infinite array.   *
 *                                                     *
 * Parameters                                          *
 *      array_ptr -- Pointer to the array to use.      *
 *      index    -- Index into the array.              *
 *                                                     *
 * Returns                                             *
 *      The value of the element.                      *
 *                                                     *
 * Note: You can get an element that                   *
 *      has not previously been stored. The value      *
 *      of any uninitialized element is zero.          *
 *******************************************************/
extern int ia_get (struct infinite_array *array_ptr, int index);
/*******************************************************
 * ia_store -- Store an element in an infinite array.  *
 *                                                     *
 * Parameters                                          *
 *      array_ptr -- Pointer to the array to use.      *
 *      index    -- index into the array.              *
 *      store_data -- Data to store.                   *
 *******************************************************/
extern void  ia_store (struct infinite_array * array_ptr,
    int index, int store_data);
```

有关这个文件还要提及几点。在编辑的三个函数：ia_get、ia_store和ia_init中，ia_init不是真正的函数，而是一个宏。多数时候，人们使用这个模块并不需要知道一个函数是函数还是只是宏。

宏放在括号中({})，所以当使用像**if/else**等语句时不会产生语法问题。下列代码将如期运行：

```
if (flag)
  ia_init (&array);
else
  ia_store (&array, 0, 1.23);
```

文件中的全部内容包括一个常量定义、一个数据结构定义和一个外部定义。没有定义代码和给数据赋值。

模块体

模块体包含模块用到的所有函数和数据。不在模块外调用的专用函数应定义成**static**，在某函数外定义且不在模块外使用的变量也应定义为**static**。

使用无限数组的程序

一个使用简单链表储存数组元素的程序，如图18-2所示。链表可以按需要变长(直到用光内存空间)。每个表元素或存储单元可以存储10个数字。要找到38号元素，程序从开头开始，越过前3个存储单元后，从当前存储单元中抽取8号元素。

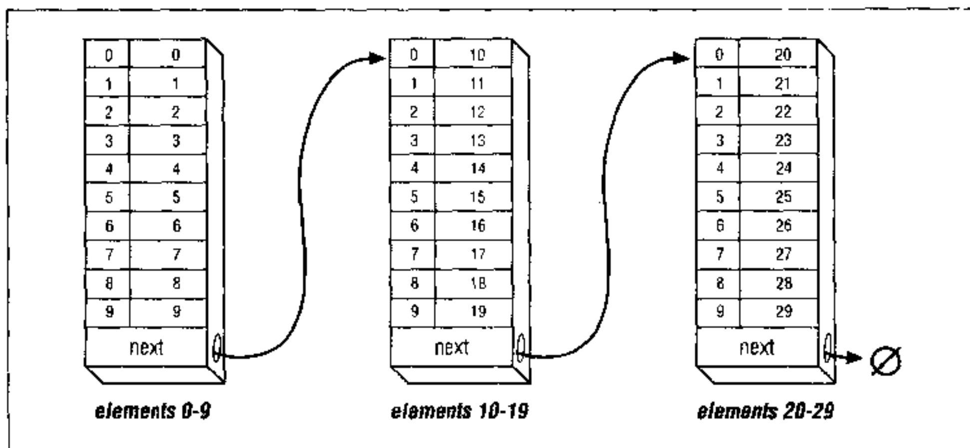


图18-2 无限数组结构

模块 *ia.c* 的代码如例 18-2 所示。

例 18-2: a/ia.c

```
/********************************************************
 * infinite-array -- routines to handle infinite arrays *
 *                                                      *
 * An infinite array is an array that grows as needed.  *
 * There is no index too large for an infinite array    *
 *  (unless we run out of memory) .                     *
 ********************************************************/
#include "ia.h"             /* get common definitions */
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>

/********************************************************
 * ia_locate -- Gets the location of an infinite array  *
 *              element.                                *
 *                                                      *
 * Parameters                                           *
 *      array_ptr -- Pointer to the array to use.       *
 *      index   -- Index into the array.                *
 *      current_index -- Pointer to the index into this *
 *              bucket  (returned) .                    *
 *                                                      *
 * Returns                                              *
 *      pointer to the current bucket                   *
 ********************************************************/
static struct infinite_array *ia_locate (
        struct infinite_array *array_ptr, int index,
        int *current_index_ptr)
{
    /* pointer to the current bucket */
    struct infinite_array *current_ptr;

    current_ptr = array_ptr;
    *current_index_ptr = index;

    while (*current_index_ptr >= BLOCK_SIZE) {
        if (current_ptr->next == NULL) {

            current_ptr->next = malloc(sizeof(struct infinite_array));
```

```
            if (current_ptr->next == NULL) {
                fprintf(stderr, "Error:Out of memory\n");
                exit(8);
            }
            memset(current_ptr->next, '\0', sizeof(struct infinite_array));
        }
        current_ptr = current_ptr->next;
        *current_index_ptr -= BLOCK_SIZE;
    }
    return (current_ptr);
}
/********************************************************
 * ia_store -- Stores an element into an infinite array.*
 *                                                      *
 * Parameters                                           *
 *      array_ptr -- Pointer to the array to use.       *
 *      index    -- Index into the array.               *
 *      store_data -- Data to store.                    *
 ********************************************************/
void ia_store(struct infinite_array * array_ptr,
    int index, int store_data)
{
    /* pointer to the current bucket */
    struct infinite_array *current_ptr;
    int   current_index;        /* index into the current bucket */
    current_ptr = ia_locate(array_ptr, index, &current_index);
    current_ptr->data[current_index] = store_data;
}
/********************************************************
 * ia_get -- Gets an element from an infinite array.    *
 *                                                      *
 * Parameters                                           *
 *      array_ptr -- Pointer to the array to use.       *
 *      index    -- Index into the array.               *
 *                                                      *
 * Returns                                              *
 *      the value of the element                        *
 *                                                      *
 * Note: You can get an element that                    *
 *      has not previously been stored. The value       *
 *      of any uninitialized element is zero.           *
 ********************************************************/
int ia_get(struct infinite_array *array_ptr, int index)
{
```

```
    /* pointer to the current bucket */
    struct infinite_array *current_ptr;
    int   current_index;         /* index into the current bucket */
    current_ptr = ia_locate(array_ptr, index, &current_index);
    return  (current_ptr->data[current_index]);
}
```

这个程序使用了一个内部程序、ia_locate。由于这个程序不能在模块外使用，所以它被定义为 **static**，该程序也放在头文件 *ia.h* 中。

用于多文件的 Makefile

make 程序的设计目的是帮助程序员编译和连接程序。在make之前，用户对程序中的每一处变动都必须明确地键入编译命令。例如：

```
% cc -g -o hello hello.c
```

随着程序的增大，建立它们需要的命令数也增多。键入一系列（10或20个）命令枯燥无味又容易发生错误、所以程序员开始写命令解释程序脚本(shell scripts)（或者MS-DOS中的*.BAT*程序）。程序员要做的就是键入脚本名字如do-it，计算机就会编译所有程序。这种方法破坏性很强，因为所有的文件不管是否需要都将被重新编译。

随着一个项目中文件数量的增多，重新编译所需的时间也随之增加。在一个小文件中做改动，开始编译，然后得等到第二天，原因是计算机得执行几百条编译命令，这未免令人沮丧，尤其是当只有一处真正需要编译的时候。

建立Make程序是用来编译的，而编译与否取决于一个文件从上次编译以来是否更新过。程序允许指明程序文件和源文件的独立性，以及从源文件中产生程序的命令。

Makefile 文件（在UNIX中对大小写有敏感性）包含make使用的规则，从而确定怎样建立程序。

Makefile 包含下列部分：

- 注释
- 宏
- 明确的规则
- 缺省规则

以 # 开头的任何行都是注释。

宏的格式是:

```
name = data
```

其中，*name* 是任何有效的修饰符，*data* 是 make 碰到 $ (*name*) 时要替换的文本。

例如:

```
#
# Very simple Makefile
#
MACRO=Doing All
all:
        echo $(MACRO)
```

明确的规则告诉 make 需要什么样的命令来建立程序，它们有几种形式，最普通的是:

```
target: source [source2] [source3]
        command
        [command]
        [command]            ...
```

其中*target*是要创建的文件名，它在源文件*source*之外“制造”或建立。如果*target*在几个文件之外建立，这几个文件都要列出来。这个列表还要包括任何由源文件引用的头文件，产生目标程序的*command*在下一行，有时需要一个以上的命令来建立目标文件，每条命令占一行，并且缩进一个制表键。

例如，规则：

```
hello: hello.c
        cc -g -o hello hello.c
```

告诉make使用下列命令从文件*hello.c*建立文件*hello*：

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

Make只在需要时创建*hello*。创立*hello*所用文件，按时间先后排列（即修改次数），见表18-2。

表18-2：建立可执行文件的顺序

UNIX	MS-DOS/Windows	
hello.c	HELLO.C	（最旧的）
hello.o	HELLO.OBJ	（旧的）
hello	HELLO.EXE	（最新的）

如果程序员改变了源文件*hello.c*，它的修改时间会使相关的其他文件过时，make会识别这一点并重新创立其他文件。

另一种形式的明确的规则是：

```
source:
        command
        [command]
```

在这种情况下，每次make运行命令都将无条件执行。如果在一个明确的规则中漏掉了命令，make将使用一套内建规则来确定执行何种命令。例如，规则：

```
hist.o: ia.h hist.c
```

告诉make从*hist.c*和*ia.h*创建*hist.o*。使用标准后缀规则从*file.c*建立*file.o*的规则是：

```
$(CC) $(CFLAGS) -c file.c
```

(make 预定义宏 $(CC) 和 $(CFLAGS)。)

我们将建立一个主程序 *hist.c* 来调用模块 *ia.c* 中的函数。两个文件都包含在头文件 *ia.h* 中，所以它们对主程序有依赖性。从 *hist.c* 和 *ia.c* 建立的 UNIX *Makefile* 是：

```
CFLAGS = -g
OBJ=ia.o hist.o

all: hist

hist: $(OBJ)
        $(CC) $(CFLAGS) -o hist $(OBJ)

hist.o:ia.h hist.c

ia.c:ia.h ia.c
```

宏 OBJ 是所有 object (*.o*) 文件的列表，下行：

```
hist: $(OBJ)
        $(CC) $(CFLAGS) -o hist $(OBJ)
```

告诉make从目标文件建立*hist*。若所有目标文件都过时，make将重新建立它们。

下行：

```
hist.o:hist.c ia.h
```

告诉 make 从 *ia.h* 和 *hist.c* 建立 *hist.o*。因为没有指定命令，将使用默认值。

MS-DOS 下，Turbo C++ 使用的 *makefile* 是：

```
#
SRCS=hist.c ia.c
OBJS=hist.obj ia.obj
CFLAGS=-ml -g -w -A
CC=tcc

ia: $(OBJS)
```

```
        $(CC) $(CFLAGS) -ehist.exe $(OBJS)

hist.obj: hist.c ia.h
        $(CC) $(CFLAGS) -c hist.c

ia.obj: ia.c ia.h
        $(CC) $(CFLAGS) -c ia.c
```

这个文件和UNIX *Makefile* 很相似，除了Turbo C++ 不提供默认规则。

make文件存在一个很大的弊端，它只检查文件是否被改动，而不看规则是否有修改。如果你在调试时使用CFLAGS=-g编译了所有程序，并需要创建产品版本（CFLAGS=-O）时，make将不重新编译。

命令touch可以改变文件的修改日期。（它不修改文件，而只是让操作系统认定它做了修改）用touch命令处理一个源文件如*hello.c*，然后运行make，就会重新建立程序。如果你改变了编译过程标志并想强制重新编译时，这个特性是很有用的。

make为建立程序提供了一套丰富的命令。这里只讨论了有限的几个。（注1）

使用无限数组

直方图程序hist使用无限数组包（直方图是表示数据出现频率的图形）。它的自变量是一个文件，该文件包含从0到99的一列数字，并可以用到输入的任何数，程序输出一个显示每个数出现频率的直方图。

程序输出的一个典型的行如下：

```
5 (    6): ***************************
```

第一个数字（5）是行下标。在样本数据中，有六个条目值为5，星号行表示6条输入。

注1：　如果要建立一个需要10或20个源文件的程序，可参考O'Reilly公司出版的《Managing Projects with make》，Andy Oram和Steve Talbott著。

一些数据超出了范围，在直方图中没有进行表示，这样的数据被计数并列示在输出结果的末尾。下面是一个输出结果的样本：

```
  1 (   9): ************************
  2 (  15): ****************************************
  3 (   9): ************************
  4 (  19): ***************************************************
  5 (  13): ***********************************
  6 (  14): **************************************
  7 (  14): **************************************
  8 (  14): **************************************
  9 (  20): ******************************************************
 10 (  13): ***********************************
 11 (  14): **************************************
 12 (   9): ************************
 13 (  13): ***********************************
 14 (  12): ********************************
 15 (  14): **************************************
 16 (  16): *******************************************
 17 (   9): ************************
 18 (  13): ***********************************
 19 (  15): ****************************************
 20 (  11): ******************************
 21 (  22): ************************************************************
 22 (  14): **************************************
 23 (   9): ************************
 24 (  10): ***************************
 25 (  15): ****************************************
 26 (  10): ***************************
 27 (  12): ********************************
 28 (  14): **************************************
 29 (  15): ****************************************
 30 (   9): ************************
104 items out of range
```

程序使用库程序memset初始化counters数组。这个程序在设置数组的所有值为0时具有很高的效率，下行使整个数组counters为零：

```
memset(counters, '\0', sizeof(counters));
```

Sizeof (counters) 确认整个数组都被赋成了0值。例18-3包括了*hist.c*的完整代码：

例 18-3: ia/hist.c

```
/********************************************************
 * hist -- Generates a histogram of an array of numbers.*
 *                                                      *
 * Usage                                                *
 *      hist <file>                                     *
 *                                                      *
 * Where                                                *
 *      file is the name of the file to work on.        *
 ********************************************************/
#include "ia.h"
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
/*
 * Define the number of lines in the histogram
 */
#define NUMBER_OF_LINES 30      /* Number of lines in the histogram */
const int DATA_MIN = 1;         /* Number of the smallest item */
const int DATA_MAX = 30;        /* Number of the largest item */
/*
 * WARNING: The number of items from DATA_MIN to DATA_MAX (inclusive)
 * must match the number of lines.
 */

/* number of characters wide to make the histogram */
const int WIDTH = 60;

static struct infinite_array data_array;
static int data_items;

int main(int argc, char *argv[])
{
    /* Function to read data */
    void read_data(const char name[]);

    /* Function to print the histogram */
    void  print_histogram(void);

    if (argc != 2) {
        fprintf(stderr, "Error:Wrong number of arguments\n");
        fprintf(stderr, "Usage is:\n");
        fprintf(stderr, "  hist <data-file>\n");
```

```
        exit(8);
    }
    ia_init(&data_array);
    data_items = 0;

    read_data(argv[1]);
    print_histogram();
    return (0);
}
/********************************************************
 * read_data -- Reads data from the input file into     *
 *              the data_array.                         *
 *                                                      *
 * Parameters                                           *
 *      name -- The name of the file to read.           *
 ********************************************************/
void read_data(const char name[])
{
    char  line[100];     /* line from input file */
    FILE *in_file;       /* input file */
    int data;            /* data from input */

    in_file = fopen(name, "r");
    if (in_file == NULL) {
        fprintf(stderr, "Error:Unable to open %s\n", name);
        exit(8);
    }
    while (1) {
        if (fgets(line, sizeof(line), in_file) == NULL)
            break;

        if (sscanf(line, "%d", &data) != 1) {
            fprintf(stderr,
              "Error: Input data not integer number\n");
            fprintf(stderr, "Line:%s", line);
        }
        ia_store(&data_array, data_items, data);
        ++data_items;
    }
    fclose(in_file);
}
/********************************************************
 * print_histogram -- Prints the histogram output.      *
 ********************************************************/
```

```
void  print_histogram(void)
{
    /* upper bound for printout */
    int   counters[NUMBER_OF_LINES];

    int   out_of_range = 0;/* number of items out of bounds */
    int   max_count = 0;/* biggest counter */
    float scale;         /* scale for outputting dots */
    int   index;         /* index into the data */

    memset(counters, '\0', sizeof(counters));

    for (index = 0; index < data_items; ++index) {
        int data;/* data for this point */

        data = ia_get(&data_array, index);
        if ((data < DATA_MIN) || (data > DATA_MAX))
            ++out_of_range;
        else {
            ++counters[data - DATA_MIN];
            if (counters[data - DATA_MIN] > max_count)
                max_count = counters[data - DATA_MIN];
        }
    }

    scale = ((float) max_count) / ((float) WIDTH);

    for (index = 0; index < NUMBER_OF_LINES; ++index) {
        /* index for outputting the dots */
        int   char_index;
        int   number_of_dots;   /* number of * to output */

        printf("%2d (%4d): ", index + DATA_MIN, counters[index]);

        number_of_dots = (int) (((float) counters[index]) / scale);
        for (char_index = 0;
             char_index < number_of_dots;
             ++char_index) {
            printf("*");
        }
        printf("\n");
    }
    printf("%d items out of range\n", out_of_range);
}
```

UNIX Generic C 的 Makefile

```
#-----------------------------------------------------#
#        Makefile for UNIX systems                    #
#    using a GNU C compiler.                          #
#-----------------------------------------------------#
CC=cc
CFLAGS=-g
#
# Compiler flags:
#       -g      -- Enable debugging

ia: ia.c
        $(CC) $(CFLAGS) -o ia ia.c

clean:
        rm -f ia
```

自由软件基金会的 gcc 的 Makefile

```
[File: ia/makefile.gcc]
#-----------------------------------------------------#
#        Makefile for UNIX systems                    #
#    using a GNU C compiler.                          #
#-----------------------------------------------------#
CC=gcc
CFLAGS=-g -Wall -D__USE_FIXED_PROTOTYPES__ -ansi

all:    hist

hist: hist.o ia.o
        $(CC) $(CFLAGS) -o  hist hist.o ia.o

hist.o: hist.c ia.h

ia.o: ia.c ia.h

clean:
        rm -f hist hist.o ia.o
```

Turbo C++ 的 Makefile

```
[File: ia/makefile.tcc]
#-----------------------------------------------#
#       Makefile for DOS systems                #
#    using a Turbo C++ compiler.                #
#-----------------------------------------------#
CC=tcc
CFLAGS=-v -w -ml

all:    hist.exe

hist.exe: hist.obj ia.obj ia.h
        $(CC) $(CFLAGS) -ehist hist.obj ia.obj

hist.obj: hist.c ia.h
        $(CC) $(CFLAGS) -c hist.c

ia.obj: ia.c ia.h
        $(CC) $(CFLAGS) -c ia.c

clean:
        del hist.exe hist.obj ia.obj
```

Borland C++ 的 Makefile

```
[File: ia/makefile.bcc]
#-----------------------------------------------#
#       Makefile for DOS systems                #
#    using a Borland C++ compiler.              #
#-----------------------------------------------#
CC=bcc
CFLAGS=-v -w -ml

all:    hist.exe

hist.exe: hist.obj ia.obj ia.h
        $(CC) $(CFLAGS) -ehist hist.obj ia.obj

hist.obj: hist.c ia.h
        $(CC) $(CFLAGS) -c hist.c
```

```
ia.obj: ia.c ia.h
        $(CC) $(CFLAGS) -c ia.c

clean:
        del hist.exe
        del hist.obj
        del ia.obj
```

Microsoft Visual C++ 的 Makefile

```
[File: ia/makefile.msc]
#-----------------------------------------------#
#       Makefile for DOS systems                #
#    Microsoft Visual C++ Compiler.             #
#-----------------------------------------------#
#
CC=cl
#
# Flags
#       AL -- Compile for large model
#       Zi -- Enable debugging
#       W1 -- Turn on warnings
#
CFLAGS=/AL /Zi /W1
SRC=hist.c ia.cpp
OBJ=hist.obj ia.obj

all: hist.exe

hist.exe: $(OBJ)
        $(CC) $(CFLAGS)  $(OBJ)

hist.obj: ia.h hist.c
        $(CC) $(CFLAGS) -c hist.c

ia.obj: ia.h ia.c
        $(CC) $(CFLAGS) -c ia.c

clean:
        erase hist.exe io.obj hist.obj
```

把一项任务分成模块

计算机编程是科学更是艺术，没有严格快速的规则可以告诉你怎样把任务划分成模块。知道哪些能产生好的模块或哪些不能，只能来自于经验和实践。

本节介绍一些模块划分的基本规则以及怎样把它们应用到现实世界的程序中。这里描述的技巧都很适合我，你可以采用适合你的技巧。

信息是任何程序的关键部分，对任何程序来讲，关键是决定使用什么样的信息及对它执行什么处理，设计开始之前应该先分析信息流。

模块设计应使模块间的信息传递量最小化。看看军队的组织，你可以发现，它们被划分成模块，这些模块是步兵、炮兵、坦克兵团等等。这些模块之间传递的信息量达到了最小，例如，如果一名步兵中士想让炮兵轰炸敌方某一阵地，就发出命令，"位置Y-94有一个碉堡，消灭它"。炮兵司令处理所有细节，他决定使用哪个炮兵连、根据其他任务的需要投入多少火力、后续支援，以及其他更详细的问题（注2）。

程序应该用同样的方法来组织。数据的隐蔽性是好程序的关键，一个模块可以是公用的函数，数据的数量应该最小化，界面越小就越简单，越简单就越容易使用，此外，一个简单的界面也比复杂的界面风险小、出错少。

小而简单的界面也比较容易设计、测试和维护。数据的隐蔽性和良好的界面设计是制作好模块的关键。

模块划分实例：文本编辑器

你已经熟悉了文本编辑器的使用，它是一个可以让用户显示和改动文本文件的程序。多数编辑器把大约25行当前文件定向并连续显示在屏幕上。文本编辑器还必须解释用户键入的命令，对这一信息进行语法分析从而使计算机能理解并执行它。

注2：　这是一个理想军队命令链的基本图表。实际的军队更复杂，分类级别更高，即使军队指挥官也不知道它是如何运行的。

单个命令很小且都执行类似的功能（如“delete line”非常类似于“delete character”）。在命令执行模块中，采用标准结构可以改善可读性和可靠性。

图 18-3 列示了组成一个文本编辑器的不同模块。

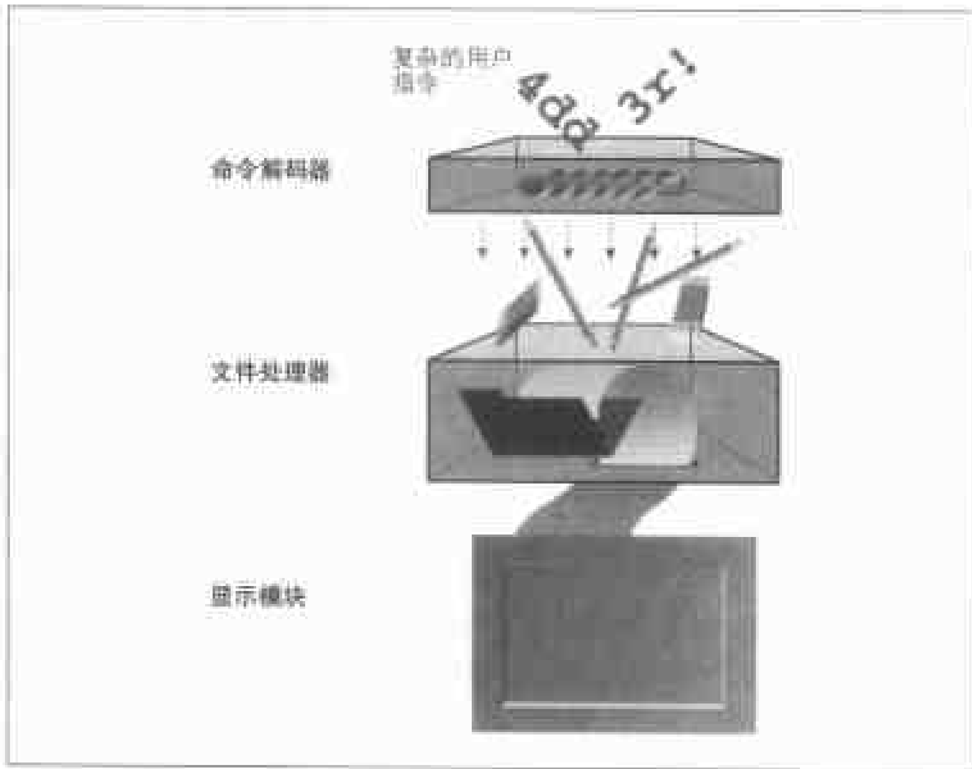


图 18-3　文本编辑器模块

模块间仅需要最少量的通信。显示器只需知道两件事：光标的位置及文件当前的样子。所有的文件处理器要做的就是读文件、写文件和记录修改，甚至涉及做变动的模块都可以最小化。所有编辑命令，不论多复杂都可以分成一系列的插入和删除。命令模块必须把复杂的用户命令变成简单的插入和删除，从而使文件处理器可以处理。

这种模块间命令的传递是最小的。实际上，在命令解码器和显示器之间就没有信息传递。

字处理器是一种奇异的文本编辑器，一个简单的编辑器可以只关注ASCII字符（一种字体，一种尺寸），字处理器则能处理不同大小和形状的字符。

编译器

在编译器中，要处理的信息是C代码。编译器的工作就是把这些信息从C源代码转换成机器能识别的目标代码。这个过程包括几个阶段：首先，预处理器要运行代码，把宏展开、注意条件编译，并读入引用文件；下一步，要处理的文件传给编译器的第一个阶段，词法分析器。

词法分析器把它的输入看作字符流，并返回一串记号。记号是一个字或运算符，例如，看下列英语命令：

```
Open the door.
```

这个命令中有14个字符，词法分析器把它转化为三个单词和一个句号。然后这些字符将被送给语法分析器，在那里，它们被汇编成句子。这一步将产生一个符号表，这样语法分析器可以知道程序中用到哪些变量。

现在编译器知道了程序打算要做什么，优化程序，检查指令，并试图找出使命令更高效的途径。这个步骤是可选项，如果不在命令行中指定 -O 标志，这一步将被略过。

代码生成器把高级语句转为机器特定的汇编代码。在汇编语言中，每个汇编语言语句都对应于一条机器指令，汇编器把汇编语言转变成机器可执行的二进制代码，编译器的一般信息流如图18-4所示。

C语言流行的一个主要原因就是很容易为新机器建立一个C编译器。Free Software Foundation为C编译器（*gcc*）分配了一个源程序，因为这个源程序是用模块风格写的，所以你可以通过改变代码生成器把它移植到新机器中，这些都是相对简单的任务（见第七章“编程过程”）。

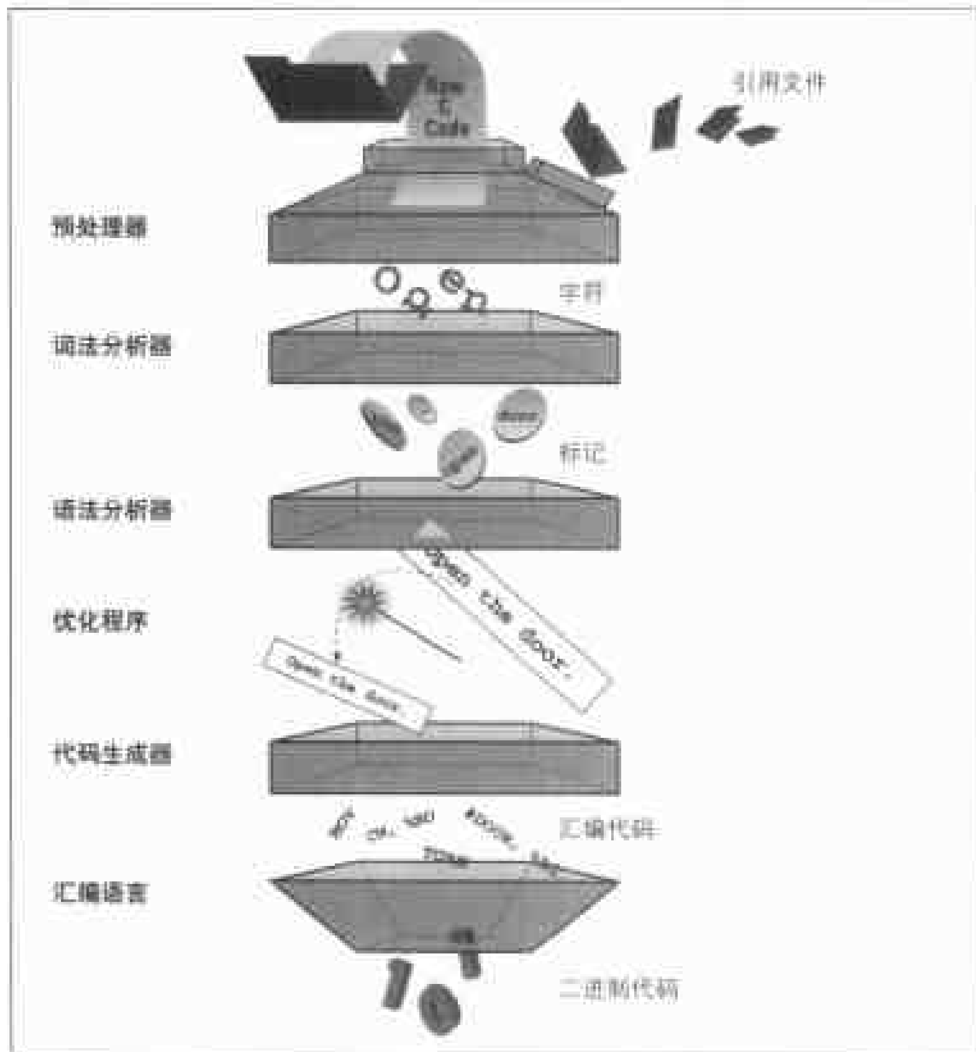


图 18-4　编译器模块

词法分析和语法分析很常见，并广泛地用于程序中。实用程序 lex 可以为一个程序生成词法分析器模块，给出程序所用标记的描述。另外一个实用程序，yacc 可以产生语法分析模块（注 3）。

注 3：　有关这些程序的描述，见O'Reilly 公司出版的《lex&yacc》，John Levine、Tony Mason 和 Doug Brown 著。

电子表格

简单的电子表格就是一个矩阵，矩阵元素是数和等式，并把结果显示在屏幕上，该程序管理的信息是等式和数据。

电子表格的核心是一套等式。要把等式修改为数，必须进行词法分析和语法分析，就像一个编译器一样。但和编译器不一样的是，这一过程不产生机器代码，而是要解释等式并计算结果。

算得的结果传送给显示器，把这些结果显示在屏幕上。另外还加上一个输入模块，它允许用户编辑并修改等式，得到一个电子表格，如图 18-5 所示。

模块设计准则

虽然在为程序划分模块时没有严格快速的规则，但有几个一般的规则：

- 模块中公用函数的数量要小。
- 模块间传递的信息要少。
- 模块中所有函数应执行相关的任务。

编程练习

练习 18-1：编写一个模块处理页面格式，包含下列函数：

`open_file (char *name)`	打开打印文件。
`define_header (char *heading)`	定义文本标题。
`print_line (char *line)`	向文件发送一行。
`page (void)`	开始新的一页。
`close_file (void)`	关闭打印文件。

练习 18-2：编写一个名为 search_open 的模块，它先接受一个文件名数组，在已有文件中进行查找，找到后，打开该文件。

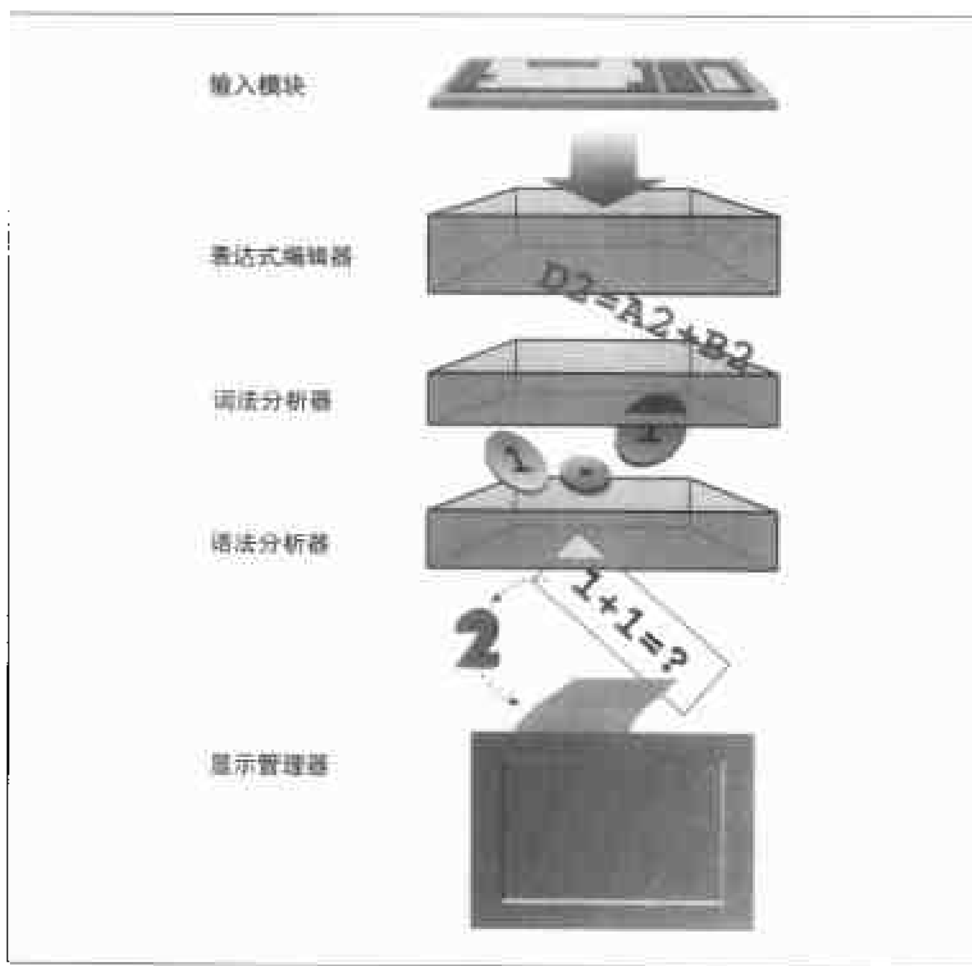


图18-5 电子表格模块

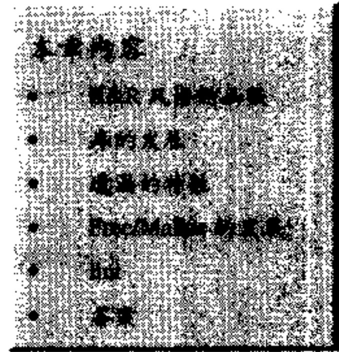
练习 18-3: 编写一个包含下列函数的符号表程序:

`void enter (char *name)`	输入符号表名称。
`int lookup (char *name)`	如果表中有名字返回 1、否则返回 0。
`void delete (char *name)`	从符号表中删除名字。

练习 18-4: 把第十七章"高级指针"的words程序和无限数组组合建立一个交叉引用程序。(顺便、引入C注释和串的概念以创建一个C交叉引用程序)。

第十九章

旧式编译器



几乎每个王国最古老的家族最初
都是王子的私生子。
—— Robert Burton

C的发展已有些年头了。最初它只是一些黑客们（Brian Kernigham 和 Dennis Ritchie）在地下室中使用计算机的工具，后来C编译器作为“便携式C编译器”得到研制发行。这种编译器的主要优点是可从一台机器移植到另一台机器上，所要做的只是写一个设备配置。实际上写设备配置很难，但这个任务要比写一个编译器程序容易得多。

便携式C编译器被广泛采用，不久就成为应用最广的C编译器。由于那时没有官方标准，便携式C编译器所编译的东西就成了“官方标准”。

本章讲述的就是这个“标准”。在后来的ANSI标准中没有对便携式C编译器的一些特性加以定义。许多新特性的加入使C程序更安全、更可靠。在ANSI C中编程很难，在老式的便携式C编译器中编程则像蒙着眼睛走钢丝绳。

K&R 风格的函数

K&R 风格的C编译器使用一个很老的定义函数，例如，ANSI 函数的定义是：

```
int process (int size, float data[], char *how)
```

在 K&R C 中将是：

```
int process (size, data, how)
int size;
float data[ ];
char *how;
{
    /* Rest of the function */
```

严格地讲，我们不需要定义“int size”，因为所有参数的类型都自动默认为**int**。但是我们在这里还是定义了**int**类型，因为定义所有的函数和变量而不取默认值是一种好的思想。

函数原型

函数原型在K&R风格的C中不做要求，可以略过。例如，想使用没有定义原型的函数draw：

```
draw (1, 8, 2, 20);
```

C会自动把这个函数定义成一个返回**int**、参数未知、类型未知的函数。显然地，无法进行参数的类型检查，所以写一个如例19-1的程序完全可能：

例19-1: area/area.c

```
#include <stdio.h>

float area (width, height)
int width;
float height;
{
    return  (width * height);
}

int main ()
{
    float size = area (3.0, 2);

    printf ("Area is %f.n", size);
    return  (0);
}
```

问题 19-1：当例 19-1 的程序运行时会产生什么结果？为什么？

K&R风格的C允许函数原型，但只能定义返回类型，参数列表必须是()。例如：

```
extern float atof ();
```

另外，() 表示这个函数的参量类型未知，参量数目也未知。

问题 19-2：例 19-2 显示什么？为什么？

例 19-2: ret/ret.c

```
#include <stdio.h>

int main ()
{
    /* Get the square of a number */
    int i = square (5);

    printf ("i is %d\n", i);
    return  (0);
}

float square (s)
int s;
{
    return  (s * s);
}
```

问题 19-3：例 19-3 显示什么？为什么？

例 19-3: sum/sum.c

```
#include <stdio.h>

int sum (i1, i2, i3)
{
   int i1;
   int i2;
   int i3;

   return  (i1 + i2 + i3);
```

```
}

int main ()
{
    printf ("Sum is %d\n", sum (1, 2, 3));
    return  (0);
}
```

问题 19-4：例19-4显示John'=3 而不是John Doe。为什么？（你的结果可能不同。）

例 19-4: scat/scat.c

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

char first[100];        /* First name of person */
char last[100];         /* Last name of person */

/* First and last name combined */
char full[100];

int main() {
    strcpy(first, "John");
    strcpy(last, "Doe");

    strcpy(full, first);
    strcat(full, ' ');
    strcat(full, last);

    printf("The name is %s\n", full);
    return (0);
}
```

原型对C编译器来讲是一个十分有用的诊断工具。没有它们，程序员所不了解的各种各样的错误都会发生，因此，人们从C++中将原型引入C中。

库的发展

像语言一样，库也有发展过程。C的“标准”库使用UNIX操作系统的内容，后来，UNIX操作系统分成两个族：BSD-UNIX和UNIX V系统，标准库也随之分化。

ANSI标准化C的同时，也标准化了库。不过，你还是能找到使用旧库调用的代码。主要区别是：

- 旧K&R C没有*stdlib.h*或*unistd.h*头文件。
- 一些旧的函数被重新命名或替换。表19-1列示了更新的函数。

表19-1 K&R对应的ANSI函数

K&R函数	ANSI等式	注解
bcopy	memcpy	拷贝一个数组或结构
bzero	memset	内存置零
bcmp	memcmp	比较两个部分的内存
index	strchr	在串中找字符
rindex	strrchr	从串尾开始找字符
char *sprintf	int sprintf	K&R函数给串返回指针。ANSI标准函数返回转化后的项目数

遗漏的特性

前文已经讲过，C语言已发展了一段时间。早期的一些编译器可能不具有近期编译器的某些特性。这些特性包括：

- **void** 类型
- **const** 修饰符
- **volatile** 修饰符（见第二十一章“C内的‘角落’”）

- *stdlib.h* 头文件或 *unistd.h* 头文件
- 枚举类型

Free/Malloc 的发展

在ANSI C中，malloc 函数的定义是：

```
void *malloc(unsigned long int size);
```

因为void * 表示"全局指针"，所以malloc的返回值和任何指针类型都匹配

```
struct person *person_ptr;      /* Define a pointer to a person */

/* This is legal in ANSI C */
person_ptr = malloc(sizeof(struct person))
```

因为一些 K&R C 编译器没有 **void** 类型，所以 **malloc** 的定义是：

```
char *malloc(unsigned long int size)
```

为了让编译器适应不同的指针类型，malloc 的输出应把它设置成正确的类型：

```
struct person *person_ptr; /* Define a pointer to a person */

/* This will generate a warning or error in K&R C */
person_ptr = malloc(sizeof(struct person));

/* This will fix that problem */
person_ptr = (struct person *)malloc(sizeof(struct person));
```

free 发生了同样的问题。ANSI C 中的 free 的定义是：

```
int free(void *);
```

K&R 中的定义是：

```
int free(char *);
```

所以你需要把参数分配给字符指针以避免警告信息。

lint

老式的C编译器缺乏现在较理想的一套错误检测系统。这个不足使得编程很困难。为了解决这个问题，人们写了一个叫lint（注1）的程序，这个程序可以检查一些普通的错误如调用函数时参数的错误、函数定义不一致、变量未初始化便被使用等等。

在程序上运行*lint*，可执行命令：

```
% lint -hpx prog.c
```

选项-h打开一些试探性的检查，选项-p检查可能的移植问题，选项-x检查虽被定义但从未使用的**extern**变量。注意：在UNIX V系统中，-h选项的函数是颠倒的，所以应该在这些系统中忽略掉这个选项。

答案

解答 19-1：问题是我们的函数把它的自变量当作整数或浮点数：

```
float area(width, height)
int width;
float height;
```

但我们可以用浮点数或整数来调用它：

```
float size = area(3.0, 2);
```

类型已经转化：函数（浮点数、整数）－调用（整数，浮点数）。但C没办法识别这种错位，因为我们使用的是K&R风格的C。结果是当程序从main向area传递参数时，参数被破坏了，我们的输出结果也发生错乱。

注1：　详细信息见Nutshell手册《用lint检查C程序》，Jan F.Darwin著。

问题19-5：例19-5包括“fixed”程序，现在使用两个浮点参数，“3.0”和“2.0”，但还是得到了错误的答案。为什么？

例 19-5: param2/param2.c

```
#include <stdio.h>
float area(width, height)
float width;
float height;
{
    return (width * height);
}
int main()
{
    float size = area(3.0 * 2.0);
    printf("Area is %f\n", size);
    return (0);
}
```

解答19-2：结果发生错乱。典型的输出是：

```
i is 1103626240
```

比52大一点。问题是没有原型，对square来说，即使是K&R风格的原型也可以。结果是C假定了缺省定义：函数返回一个带任意个参数的整数。

但这个函数返回了一个浮点数。因为返回的是浮点数而C却以为收到的是整数，所以得到了无用的数据。这个问题可以通过在程序开头加入K&r风格的原型来改正：

```
float square();
```

一个更好的解决办法是通过加入真正的原型和修改函数头文件，把这个程序转化成ANSI C。

解答19-3：这个程序在合计三个未经初始化的变量的基础上，显示了一个随机数。

问题出在函数头文件：

```
int sum(i1, i2, i3)
{
   int i1;
   int i2;
   int i3;
```

参数类型在函数的第一个括号({)之前定义。本例中，没有括号，所以i1,i2,i3默认为整数类型。

随后在函数开头定义了i1,i2,i3，这些对局部变量的定义和参数i1,i2,i3无关。但是，由于这些变量和参数同名，使参数变成了隐藏变量。这三个未经初始化的变量被合计，其随机结果返回给了调用者。

在ANSI（译注）中这类错误是非法的，但许多编译器还是接受这个代码。

解答 19-4: 函数strcat把两个串作为它的自变量。在语句stcat(full, ' ')中，第一个自变量full是一个串；第二个自变量' '，是一个字符。使用一个字符代替串是非法的。旧式的C编译器不检查参数类型，所以这种错误能通过编译器。字符' '应该由串" "代替。

解答 19-5: 问题在于写的是：

```
float size = area(3.0 * 2.0);
```

而应该写：

```
float size = area(3.0, 2.0);
```

第一个语句把表达式"3.0*2.0"或"6.0"作为第一个参数，不存在第二个参数。C不检查参数的数量，所以它给第二个变量一个随机值。

译注： ANSI: American National Standards Institute 的缩写，美国国家标准局。

本章内容

- 模块化
- 字大小
- 字节顺序问题
- 对齐问题
- NULL指针问题
- 文件名问题
- 文件类型
- 小结
- 答案

第二十章

移植问题

> 我说起最可怕的灾祸，
> 海上陆上惊人的奇遇
> 间不容发的脱险……
> ——莎士比亚，用于程序移植
> [《奥赛罗》，第1幕，第3场]

你已经完成了一件杰作，在Cray超级计算机上、用300MB内存和50GB硬盘空间，表示复杂的三维阴影图形的光线描绘程序。当某个人要求你把这个程序移植到仅有640K内存和100MB硬盘空间的IBM PC上时，你有何感想？“杀”了他，这不仅是非法的，而且，也不专业。你唯一的选择只有抱怨，然后开始移植。移植过程中，你会发现原来运行良好的程序会产生许多奇怪的不可思议的问题。

一般认为C程序是可移植的，但是C包含许多依赖机器的特性。而且，由于UNIX和MS-DOS/Windows之间存在巨大的不同，系统缺陷常常使许多程序存在移植问题。

本章将讨论写真正可移植的程序时有关的一些问题，也讨论可能会遇到的一些陷阱。

模块化

编写可移植程序的一个技巧是，把所有不可移植的代码放到一个模块中。例如，在MS-DOS/Windows和UNIX系统中，屏幕处理存在较大差异。要设计一个可移植程序，你只能为不同的机器编写各自的屏幕处理模块。

例如，HP-98752A终端有一组功能键，标记为F1到F8，PC也有一组功能键。问题是，在不同的系统中，这些键对应的代码不一样。HP中F1键发出的代码是“<esc>p<return>”，而PC机的F1发出的是“<null>”。在这种情况下，你会想写一个get_code程序，它从键盘得到一个字符（或是功能键串），并转换功能键。因为转换依机器的不同而不同，所以需要为每种机器编写一个依赖于机器的模块。对于HP机器，你可以把程序放到*main.cc*和*hp-tty.cc*中；而在PC中，你使用的可能是*main.cc*和*pc-tty.cc*。

字大小

长整数为32位字节，短整数为16位字节。一般而言，整数依据使用的机器可能是16位或32位，这种不一致性可能带来意想不到的问题。例如，下面的代码适用于32位的UNIX系统，但移植到MS-DOS/Windows上却运行不了：

```
int zip;
zip = 92126;
printf ("Zip code %d\n", zip);
```

问题是在MS-DOS/Windows上，zip只有16位——不够存储92126。为解决这个问题，我们把zip定义成32位整数：

```
long int zip;
zip = 92126;
printf ("Zip code %d\n", zip);
```

现在zip是32位的变量，可以存储92126了。

问题20-1：为什么还有问题呢？zip在PC上显示不正确。

字节顺序问题

短整数包含两个字节。考虑一下数0x1234，2字节中的值分别是0x12和0x34，第一个字节中存储的是哪个值？回答是：这将依机器而定。

当你想写可移植的二进制文件时，这种不确定性会导致很大的麻烦。Motorola 68000系列机器使用的字节顺序为（ABCD），而Intel和DEC机器使用另外一种字节顺序（BADC）。

关于二进制移植问题的一种解决方法是避免使用它。在程序中放一个选项来读和写ASCII文件、ASCII的双重优势是更强的可移植性和可读性。

缺点则是文本文件较大。一些文件对于ASCII来说太大了，在这种情况下，文件开头所放的魔术数很有用。假定魔术数是0x11223344（虽是个坏魔术数，却是个好例子），当程序读取魔术数时，它能够检测出正确的数字和字节转换方案，程序可以自动解决这个问题：

```
const long int MAGIC      = 0x11223344L /* file identification number*/
const long int SWAP_MAGIC = 0x22114433L /* magic-number byte swapped */
FILE *in_file;                /* file containing binary data */
long int magic;               /* magic number from file */
in_file = fopen("data","rb");
fread((char *)&magic, sizeof(magic), 1, in_file);
switch (magic) {
    case MAGIC:
        /* No problem */
        break;
    case SWAP_MAGIC:
        printf("Converting file, please wait\n");
        convert_file(in_file);
        break;
    default:
        fprintf(stderr,"Error:Bad magic number %lx\n", magic);
        exit (8);
}
```

对齐问题

一些计算机限制整数和其它类型数据使用的地址。例如，68000系列机要求所有的整数要两字节两字节的对齐。如果你想用奇数地址访问一个数，会产生一个错误。有些处理器没有对齐规则，而有些则更严格些——求整数对齐四字节。

对齐规则不仅限于整数，浮点数和指针也必须正确对齐。

C向你隐藏对齐规则。例如，如果你在68000上定义下列结构：

```
struct funny {
    char     flag;    /* type of data following */
    long int value;  /* value of the parameter*/
};
```

C将为该结构分配内存如图20-1左所示：

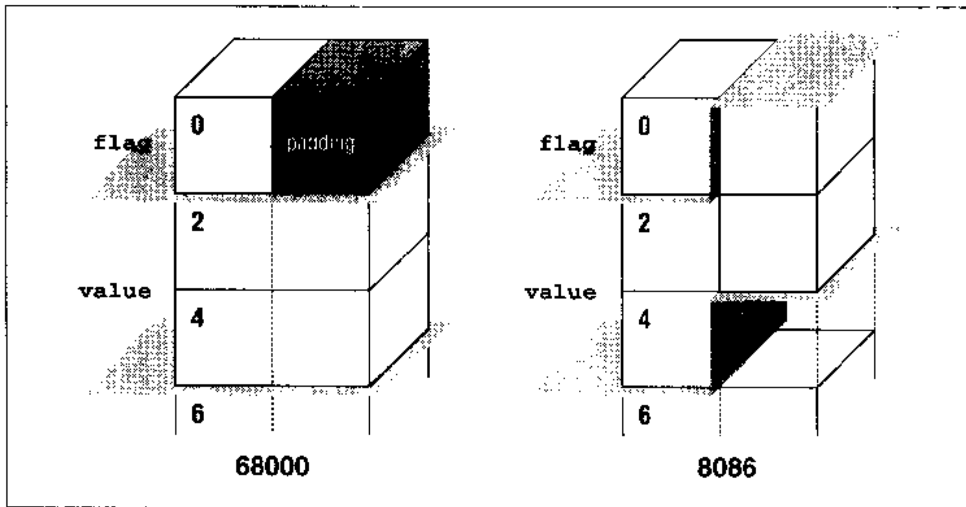


图20-1　在68000和8086体系结构下的结构

在8086系列机上，没有对齐规则，其内存分配如图20-1中右图所示。问题是结构的大小将依机器而改变。在68000上，结构大小为6字节；而在8086上是5字节。所以如果你在68000上写一个包含100个记录的二进制文件，它将有600个字节长；而在8086上，它只有500字节长。很明显，在这两种机器上写文件所用的不是同一种方法。

一种解决方法是使用ASCII文件，正如我们前面提到的，二进制文件存在许多问题。另一个解决方法是明确地定义一个pad字符：

```
struct new_funny {
        char    flag;   /* type of data following */
        char    pad;    /* not used */
        long int value; /* value of the parameter*/
};
```

这个pad字符使得在68000机器上正确地对齐字段值；而在8086机器上，能保证结构的大小正确。

使用pad字符很困难，也易出错。例如，虽然new_funny在对32位整数以1或2字节对齐的两种机器上是可移植的，但是，它不能移植到任何四字节对齐的机器上，如Sun SPARC系统。

NULL指针问题

许多程序和应用程序是使用UNIX在VAX计算机上书写的。在这样的计算机上，任何程序的第一个字节是0。在这个机器上，许多程序都包含同一个错误——它们把空指针当作一个串来使用。

例如：

```
#ifndef NULL
#define NULL  ((char *) 0)
#endif NULL

char *string;

string = NULL;
printf ("String is '%s'\n", string);
```

这段代码实际上是string的非法使用，空指针决不能被逆引用。在VAX机器上，这个错误不会导致什么问题，因为0字节的程序就是0。String指向一个空串，这个结果是凭运气，而不是故意设计的。

在VAX机器上，将输出下列结果：

```
String is ''
```

在Celerity计算机上，程序的第一个字是“Q”。当这个程序在C1200上运行时，结果是：

```
String is 'Q'
```

在其他计算机上，这种代码会产生出人意料的结果。从VAX机移植到Celerity机的许多实用程序都存在“Q”问题。

许多近期的编译器现在都能够检查NULL并显示：

```
String is (null)
```

这个信息并不是说显示NULL是对的，它的意思是这个错误太普通了，编译器制造者想给程序员一个安全网，即在错误发生时程序显示一些合理的东西，而不是崩溃掉。

文件名问题

UNIX规定文件名为*/root/sub/file*，而MS-DOS/Windows使用*\root\sub\file*。当从UNIX向MS-DOS/Windows移植时，文件名必须改变。例如：

```
#ifndef __MSDOS__
#include <sys/stat.h> /* UNIX version of the file */
#else __MSDOS__
#include <sys\stat.h> /* DOS version of the file */
#endif __MSDOS__
```

问题20-2：为什么下列程序在UNIX上能够运行，但当在MS-DOS/Windows上运行时得到下列信息：

```
oot
ew      able:   file not found.

FILE *in_file;
#ifndef __MSDOS__
```

```
const char NAME[] = "/root/new/table";
#else __MSDOS__
const char NAME[] = "\root\new\table";
#endif __MSDOS__
in_file = fopen(NAME, "r");
if (in_file == NULL) {
    fprintf(stderr,"%s: file not found\n", NAME);
    exit(8);
}
```

文件类型

在UNIX中只有一种文件类型。在MS-DOS/Windows有两种：文本和二进制。标志O_BINARY和O_TEXT用在MS-DOS/Windows中表示文件类型，UNIX中没定义这些标志。

当从MS-DOS/Windows向UNIX中移植时，需要对标志做点什么呢？如果标志还没有定义，答案之一就是使用预处理器来定义它们：

```
#ifndef O_BINARY            /* If we don't have a flag already */
#define O_BINARY 0          /* Define it to be a harmless value */
#define O_TEXT   0          /* Define it to be a harmless value */
#endif /* O_BINARY */
```

这种方法在UNIX和MS-DOS/Windows下允许使用同样的打开命令，然而用另一种方法可能会出问题，UNIX中文件就是文件，不需要多余的标志，通常一个标志也不设置，而当进入MS-DOS/Windows的时候，就需要额外的标志且要放到系统中来。

小结

在C中可以写一个可移植程序，不过，由于C在使用不同操作系统的许多不同类型的机器上都能运行，写可移植程序就不容易了。但如果在创建代码时你始终有移植观念，是可以使问题最小化的。

答案

解答 20-1：变量 zip 是长整型。Printf 的 %d 是对一般整数的说明，正确的说明应该是 %ld 表示一个长整数：

```
printf ("Zip code %ld\n", zip);
```

解答 20-2：问题是 C 使用反斜杠（\）作为消除字符。字符 \r 是回车键、\n 是换行符、\t 是水平制表符。名字的真正格式是：

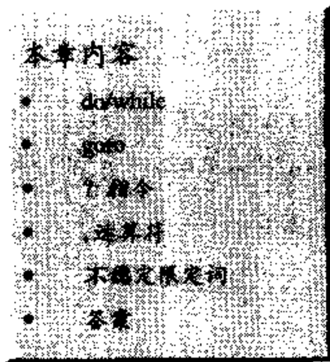
```
<return>oot<newline>ew<tab>able
```

名字应该被说明成：

```
const char NAME[] = "\\root\\new\\table";
```

注意：#include 使用文件名，而不是 C 串。在一个 C 串中可以使用双反斜杠（\\），而在一个 #include 文件中，应使用单斜杠（\）。下列两行都是正确的：

```
const char NAME[] = "\\root\\new\\table";
#include "\root\new\defs.h"
```



第二十一章

C内的“角落”

那就是他们，在其背后还有一个名字

——《圣经·传道书》44:8

本章讲述前面各章中尚未涉及到的C的其他一些特性。本章之所以称为角落，是因为在真正的程序中，这些语句几乎已不再使用。

do/while

do/while 语句有如下语法：

```
do {
    语句
    语句
}while (表达式);
```

程序执行循环，然后检查表达式的值，如果表达式为假（0），则停止循环。

注意：该命令至少执行一次。

C程序中，**do/while** 语句不常使用，因为大多数程序员更愿意使用 **while/break** 配对语句。

goto

本书的每一个样本程序都没用**goto**语句，在实际中，我发现每隔大约一年能使用一次**goto**语句。在极少数情况下，对那些还必须有**goto**语句的程序，它的语法为：

```
goto label;
```

这里，*label*是一个语句标号，语句标号与变量命名方法一样，给语句加标号的形式如下：

```
label: statement
```

例如：

```
for (x = 0; x < X_LIMIT; x++) {
    for (y = 0; y < Y_LIMIT; y++) {
        if (data[x][y] == 0)
            goto found;
    }
}
printf ("Not found\n");
exit (8);
```

发现：

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

问题21-1：当键入不正确的命令时，例21-1为什么没有打印出出错信息？

提示：这也正是我把这个例子放到**goto**一节中的原因。

例21-1: def/def.c

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

int main ()
{
    char  line[10];
```

```
    while (1) {
        printf ("Enter add (a) , delete (d) , quit (q): ");
        fgets (line, sizeof (line) , stdin);

        switch (line[0]) {
        case 'a':
            printf ("Add\n");
            break;
        case 'd':
            printf ("Delete\n");
            break;
        case 'q':
            printf ("Quit\n");
            exit (0);
        defualt:
            printf ("Error:Bad command %c\n", line[0]);
            break;
        }
    }
}
```

?: 指令

问号（?）和冒号（:）运算符与 **if/then/else** 工作方式相似，不一样的是?:指令可以用在表达式内部，一般形式是：

```
(表达式) ? 值1 : 值2
```

例如，下面的指令根据 balance 的金额，把 balance 的值或 0 赋给 amount_owed：

```
amount_owed = (balance < 0) ? 0 : balance;
```

下列宏返回两个自变量中最小者：

```
#define min(x,y) ((x) < (y) ? (x) : (y))
```

,运算符

逗号运算符(,)可以用在一组语句中,例如:

```
if (total < 0) {
    printf("You owe nothing\n");
    total = 0;
}
```

可以写成:

```
if (total < 0)
    printf("You owe nothing\n"),total = 0;
```

多数情况下用{}来代替逗号。大约只有一个地方使用逗号运算符,即在 **for** 语句中。下面的 **for** 循环将使两个计数变量 two 和 three 的值分别递增 2 和 3:

```
for (two = 0, three = 0;
     two < 10;
     two += 2, three += 3)
        printf("%d %d\n", two, three);
```

不稳定限定词

volatile(不稳定)关键字用来表示其值可在任何时刻变动的变量。这个关键字用于如存储映像 I/O 设备或实时控制应用,变量可以由中断信号例程改变。

存储映像设备驱动器、中断信号例程和实时控制都是极端先进的课题。你只有在远远超过本书范围的水平上编程时,才需要使用 **volatile**(不稳定)关键字。

答案

解答 21-1: 编译器看不见我们的默认行,因为我们错把 default 拼成了 defualt。这个错误没被标记,原因是 defualt 是有效的 **goto** 标签。

本章内容

- 需求
- 规范说明
- 代码设计
- 编码
- 功能描述
- 扩展
- 测试
- 修改
- 最后的警告
- 程序文件
- 编程练习

第二十二章

组合到一起

世界上没有哪一个工作，乞丐不知道和不能做的

—— 吉卜林（美国作家）

本章我们将创建一个完整的程序。该程序涵盖了从列出要求到测试结果的每一步。

需求

在开始之前，必须确定要干什么。这一步很重要，它可以缩短编程周期。

本章的程序必须满足若干要求。首先，它必须足够长从而能说明模块化程序设计方法，但同时又必须足够短，以便能放在一章的内容里。其次它必须足够复杂，以说明C众多的特性，但又必须足够简单，让C编程的人门者能够明白。

最后，它必须是有用的。有用性的定义并不简单，对一个人有用的程序可能对另一个人就没有用，我们决定提炼这个要求并重新表述成“程序对C程序员必须有用”。

选定的程序读取C源文件，并对嵌套括号的层数及代码中注释行的比率产生一个简单统计。

规范说明

我们的统计程序的规范说明如下:

C统计收集程序
初步文档说明
Steve Oualline

1996年2月10日

程序stat收集C源文件的统计数据并显示它们。命令行是:

```
stat <files..>
```

其中<files..>是源文件列表。下面显示的是对一个短的测试文件的运行结果:

```
[File: stat/stat.out]
  1  (0  {0  /*-*/
  2  (0  {0  /********************************************************
  3  (0  {0   * Name: Calculator (Version 2) .                      *
  4  (0  {0   *                                                     *
  5  (0  {0   * Purpose:                                            *
  6  (0  {0   *      Act like a simple four-function calculator.    *
  7  (0  {0   *                                                     *
  8  (0  {0   * Usage:                                              *
  9  (0  {0   *      Run the program.                               *
 10  (0  {0   *      Type in an operator (+ - * /) and a number.    *
 11  (0  {0   *      The operation will be performed on the current *
 12  (0  {0   *      result, and a new result will be displayed.    *
 13  (0  {0   *                                                     *
 14  (0  {0   *      Type 'Q' to quit.                              *
 15  (0  {0   *                                                     *
 16  (0  {0   * Notes: Like version 1 but written with a switch     *
 17  (0  {0   *      statement.                                     *
 18  (0  {0   ********************************************************/
 19  (0  {0  /*+*/
 20  (0  {0  #include <stdio.h>
 21  (0  {0  char  line[100];  /* line of text from input */
 22  (0  {0
 23  (0  {0  int   result;     /* the result of the calculations */
 24  (0  {0  char  operator;   /* operator the user specified */
 25  (0  {0  int   value;      /* value specified after the operator */
```

```
  26  (0  {0  int main()
  27  (0  {1  {
  28  (0  {1      result = 0;    /* initialize the result */
  29  (0  {1
  30  (0  {1      /* loop forever (or until break reached) */
  31  (0  {2      while (1) {
  32  (0  {2          printf("Result: %d\n", result);
  33  (0  {2          printf("Enter operator and number: ");
  34  (0  {2
  35  (0  {2          fgets(line, sizeof(line), stdin);
  36  (0  {2          sscanf(line, "%c %d", &operator, &value);
  37  (0  {2
  38  (0  {2          if ((operator == 'q') || (operator == 'Q'))
  39  (0  {2              break;
  40  (0  {3          switch (operator) {
  41  (0  {3          case '+':
  42  (0  {3              result += value;
  43  (0  {3              break;
  44  (0  {3          case '-':
  45  (0  {3              result -= value;
  46  (0  {3              break;
  47  (0  {3          case '*':
  48  (0  {3              result *= value;
  49  (0  {3              break;
  50  (0  {3          case '/':
  51  (0  {4              if (value == 0) {
  52  (0  {4                  printf("Error:Divide by zero\n");
  53  (0  {4                  printf("   operation ignored\n");
  54  (0  {3              } else
  55  (0  {3                  result /= value;
  56  (0  {3              break;
  57  (0  {3          default:
  58  (0  {3              printf("Unknown operator %c\n", operator);
  59  (0  {3              break;
  60  (0  {2          }
  61  (0  {1      }
  62  (0  {1      return (0);
  63  (0  {0  }
Total number of lines: 63
Maximum nesting of () : 2
Maximum nesting of {} : 4
Number of blank lines .................4
Number of comment only lines ..........20
Number of code only lines .............34
Number of lines with code and comments 5
Comment to code ratio 64.1%
```

代码设计

有几种代码设计流派。在结构化程序设计中，可以把代码分成模块，把模块分成子模块，再把子模块分成子子模块，依此类推。也有其他的方法，如状态表和转换图。

所有方法的基本原则都是一样的：“用尽可能最清楚最简单的方式整理程序的信息，然后再把它们转为C代码。”

我们的程序分为几个逻辑模块，首先是标记扫描模块，它读入原始的C代码并把它转成标记。这个分支又被分成三个更小的模块，第一个读取输入的文件，第二个确定字符类型，第三个把信息组合成标记。标记是组成一个单词、数字或符号的一组字符。

主模块收集标记，输出统计数据。这里这个模块又被分成两个更小的子模块：一个管理每个文件的 do_file 程序和一个每次统计时使用的子模块。

标记模块

程序扫描C源代码并用标记产生统计数据。例如，下行：

```
answer = (123 + 456) / 89;  /* Compute some sort of result */
```

包括的标记是：

```
T_ID            单词 "answer"
T_OPERATOR      字符 "="
T_L_PAREN       左括号
T_NUMBER        数字 123
T_OPERATOR      字符 "+"
T_NUMBER        数字 456
T_R_PAREN       右括号
T_OPERATOR      除法运算符
T_NUMBER        数字 89
T_OPERATOR      分号
T_COMMENT       注释
T_NEW_LINE      行尾字符
```

那么怎样识别标记呢？在多数时候，仅看第一个字符就行了，例如，*字符代表一个运算符标记，而A字符开始一个标识符。但一些字符比较含糊，例如，/字符可以是除法运算符，也可以是注释的开始符号。为识别这个字符，我们需要看前一个字符，所以对输入模块的要求之一就是它允许看前一个字符。

标记模块从多组字符中建立标记。例如，一个标识符被定义成一个字母或下划线，后面跟任何数目的字母或数字，所以标记识别程序要包含下列伪代码：

```
if the current character is a letter, then
      scan until we get a character that's not a letter or digit
```

（如果当前字符是一个字母，那么扫描，直到我们得到一个不是字母或数字的字符）

从伪代码中你可以看到，标记识别程序在很大程度上依赖于字符类型，所以需要一个模块来帮助我们区分类型信息。

输入模块

输入模块要做两件事：第一，需要把当前和后面的字符提供给标记模块；第二，把整行数据存入缓冲区供以后显示用。

如何不必设计一个输入模块

有时软件设计在最后编码之前会几经修改，输入模块也一样。我仔细设计了这个模块，又检查了我的设计，最后决定把它扔掉。

不过，写出来的东西不会全是垃圾，于是我把第一个设计作为一个反例，并看怎样才能改进它。

第一个设计包含一个公用结构：

```
struct input_file {
    FILE *file;          /* File we are reading */
    char line[LINE_MAX];/* Current line */
    char *char_ptr;      /* Current charactor on the line */

    int cur_char;        /* Current character (can be EOF) */
```

```
    int next_char;       /* Next character (can be EOF) */
};
```

操作这个结构的函数是：

```
extern void in_open (struct input_file *in_file, const char name[]);
extern void in_read_char (struct input_file *in_file);
extern void in_flush (struct input_file *in_file);
```

要使用这个包，调用者需要调用in_open以打开文件，然后检查in_file.file看文件是否已被打开。在C中，这些操作实现如下：

```
struct input_file in_file;     /* File for input */
/* ... */
in_open (&in_file, name);
if (in_file.file == NULL) {
    fprintf (stderr,"Error: Could not open input file: %s\n", name);
```

标记模块需要看到当前和下一个字符。例如，当它看见斜杠（/）和星号（*）时，它知道它正在看一个注释。当前字符存储于in_file.cur_char中，下一个字符存储于in_file.next_char中。检查注释的C代码如下：

```
if ((in_file.cur_char == '/') && (in_file.next_char == '*')) {
    /* Handle a comment */
```

要前移一个字符，用户调用in_read_char使输入提前一个字符。

最后，文件完成时，用户用下列语句关闭文件：

```
fclose(in_file.file);
```

在良好的模块设计中：

- 使用模块的人需要的信息量应该最小化。
- 使用模块的用户必须遵循的而且能正确使用模块的规则的数量应该小。
- 模块应易于扩展。

输入模块的设计，要求用户知道相当多的模块设计知识。为打开一个文件，用户必须知道调用in_open函数，然后检查in_file结构的file字段错误，因此用户必须知道in_file结构，甚至还要知道in_file结构的内部运行。

在其他情况下，用户还需要知道struct in_file的内部，比如访问当前字符(cur_char)或下一字符(next_char)。此外，用户还需要使用数据结构的其他内容来手工关闭文件。

所以这种设计要求用户知道模块的许多内部运行问题，而且要正确访问。好的设计对用户的要求少得多。

改善输入模块

新模块设计去除了in_file结构（和用户相关的部分），并提供了下列函数：

```
extern int in_open(const char name[]);
extern void in_close(void);
extern void in_read_char(void);
extern int in_cur_char(void);
extern int in_next_char(void);
extern void in_flush(void);
```

这些函数把处理输入文件必须的管理操作隐藏了起来，而且，文件的打开简化了，我们可以在同一个函数调用中打开文件及检查错误。

这个设计的一大优点是调用者无须知道输入文件的结构。实际上，该结构已经完全从头文件中去除了，这种设计在很大程序上简化了调用者使用模块所需的信息。

这种设计也有几个缺点。模块中的函数要比前一种设计多得多，而且模块一次只允许打开一个文件，这种限制存在的原因是，从头文件中删除的结构在模块中被替换了。单文件的限定制约了我们的灵活性，不过，我们并不需要打开多个文件，所以这种特性此时并不需要。本例中，权衡得失，界面简化和灵活性降低相比还是值得的。

字符类型模块

字符类型模块的设计目的是读取数据并判定它们的类型。一些类型是重叠的，例如，C_ALPHA_NUMERIC包含了C_NUMERIC的字符集。

这个模块把多数类型的信息储存在一个数组中，且对处理如C_ALPHA_NUMERIC之类的特殊类型只需要进行一些逻辑上的判断。

这个模块中的函数有:

```
extern int is_char_type(int ch, enum CHAR_TYPE kind);
extern enum CHAR_TYPE get_char_type(int ch);
```

一个问题出现了：怎样初始化字符型数组呢？我们可以要求用户在调用任何函数之前，先对它进行初始化，如:

```
main() {
    /* .... */
    init_char_type();

    /* ...... */
    type_info = ch_to_type(ch);
```

另一种方法是每个函数开头放一个检查语句，如果需要就进行初始化:

```
int is_char_type(int ch, enum CHAR_TYPE kind)
    if (!ch_setup) {
        init_char_type();
        ch_setup = 0;
    }
```

第二种方法要求代码更多一些。它也有几个优点，首先，它让用户的日子好过了，他不必记住要初始化一个字符型模块。此外，不易发生错误，如果用户不用做初始化，他就不会忘记做，(也就不会导致由此发生的错误)。最后，这种方法把内部管理操作问题隐藏在字符类型模块中，致使用户不用担心它。

统计子模块

每个统计子模块都可以访问输入标记的内容，并产生对它的统计。例如，括号计数器统计括号嵌套。有些统计在逐行的基础上进行报告，如当前括号嵌套。其他的统计在文件末尾报告，如括号嵌套的最大数。

我们收集了四种统计，分别是行数统计、括号（）嵌套统计、括号{}嵌套统计、无注释和有注释行数统计。因为每个统计子模块执行类似的函数，所以给它们命以相似的名字（*xx* 即下列的子模块标识符）。

`xx_init`
: 初始化统计。这个函数在每个文件开头调用。

`xx_take_token`
: 接收一个标记并基于它更新统计。

`xx_line_start`
: 在每行开头输出统计值，有时无值。

`xx_eof`
: 文件末尾显示统计信息。

标识符中 *xx* 的部分是子模块标识符。它是：

lc
: 行计数器子模块

pc
: 括号计数器子模块

bc
: 花括号计数器子模块

cc
: 注释／非注释行计数器子模块

编码

编码过程相对简单，唯一的问题是要注意行尾的正确性。

功能描述

本节描述程序中所有的模块和主要函数，更完整更详细的描述见本章末尾的列表。

ch_type 模块

ch_type模块计算字符类型。这个计算功能的大部分通过一个叫type_info的表完成。一些类型如C_ALPHA_NUMERIC包括两个不同的字符类型：C_ALPHA和C_DIGIT，所以除表以外，我们还需要为特殊情况准备一点代码。

In_file 模块

这个模块从输入文件中读取数据，一次一个字符。它缓冲一行，并在需要时输出。

标记模块

我们需要一个输入的标记流，所需要的开头是包含符号的输入流。这个类的主函数next_token把字符转成标记。实际上，标记识别模块相当简单，因为我们不必处理一个完整C标记识别模块所必须处理的大部分细节。

除了把多行注释分成了一系列的T_COMMENT和T_NEW_LINE标记以外，这个函数的代码相对简明。

行计数器子模块（lc）

我们所收集的最简单统计是当前所处理的行数统计，这一观念通过行计数器子模块实现，它所关注的唯一标记是T_NEW_LINE。每行开头，输出行数（T_NEW_

LINE标记的当前统计)。在文件末尾，这个子模块什么也不输出。考虑到连续性，我们定义了一个lc_of函数，但该函数什么也不做。

括号{}计数器子模块（bc）

该子模块跟踪括号{}的嵌套层数。通过bc_take_token函数给子模块一个标记流，这个函数跟踪左和右括号并忽略任何其他符号：

```
void bc_take_token(enum TOKEN_TYPE token) {
    switch (token) {
        case T_L_CURLY:
            ++bc_cur_level;
            if (bc_cur_level > bc_max_level)
                bc_max_level = bc_cur_level;
            break;
        case T_R_CURLY:
            --bc_cur_level;
             break;
        default:
            /* Ignore */
            break;
    }
}
```

这个统计的结果在两处显示。第一个在每行开头，第二个在文件结尾。定义两个函数显示这些统计：

```
static void bc_line_start(void) {
    printf("{%-2d ", bc_cur_level);
}

static void bc_eof(void) {
   printf("Maximum nesting of {} : %d\n", bc_max_level);
}
```

括号（）计数器模块（pc）

这个子模块和括号{}计数器子模块很相似，实际上它是通过复制括号{}计数器子模块和执行几个简单编辑来建立的。

我们或许应该把括号()计数器子模块和括号{}计数器子模块结合成一个子模块，并用参数告诉它统计什么。接下来介绍下一个模块。

注释统计子模块（cc）

在这些函数中，我们跟踪有注释的行，有代码的行，既有注释又有代码的行和既没有注释又没有代码的行。统计结果在文件末尾显示。

do_file 程序

do_file 程序读取文件，一次读一个标记，且为每个统计类把标记发送给 take_token 程序：

```
while (1) {
cur_token = next_token();

    lc_take_token(cur_token);
    pc_take_token(cur_token);
    bc_take_token(cur_token);
    cc_take_token(cur_token);
```

扩展

在设计任何软件时，头脑中都应保持扩展观念。换句话说，当有人要往你的程序中加内容时，该怎么办呢？假设某人想向我们的程序中加入新统计，他该做什么呢？

假定他要加入的是一个字统计子模块（wc），他需要定义四个程序：wc_init、wc_take_token、wc_line_start、wc_eof。

应在正确的地方调用这些程序，但他们怎么知道把调用程序放在哪儿呢？答案是他们可以用编辑器找出使用了注释计数器子模块程序的每个地方，并复制注释计数器调用。这种方法不是做事情的最好方法，特别是做跨几个文件的调用时。但是，在C的限制范围内，这种方法是最好的。

C++ 就没有这种限制。在《Practical C++ Programming》(《实用 C++ 编程》)一书中，我们设计了相似的使用 C++ 类的程序，结果是没有使用程序类的多列表，而用的是一个类列表。单表使得扩展性和可维护性都提高了。

测试

为测试这个程序，我们采用了一个包含各种可能遇到的不同类型符号的 C 程序，测试的结果如例 22-1。

例 22-1: stat/test.out

```
  1 (0  {0  /* This is a single line comment */
  2 (0  {0  /*
  3 (0  {0   * This is a multiline
  4 (0  {0   * comment.
  5 (0  {0   */
  6 (0  {0  int main()
  7 (0  {1  {
  8 (0  {1      /* A procedure */
  9 (0  {1      int i;      /* Comment / code line */
 10 (0  {1      char foo[10];
 11 (0  {1
 12 (0  {1      strcpy(foo, "abc");     /* String */
 13 (0  {1      strcpy(foo, "a\"bc");    /* String with special character */
 14 (0  {1
 15 (0  {1      foo[0] = 'a';           /* Character */
 16 (0  {1      foo[1] = '\'';          /* Character with escape */
 17 (0  {1
 18 (0  {1      i = 3 / 2;              /* Slash that's not a commment */
 19 (0  {1      i = 3;                  /* Normal number */
 20 (0  {1      i = 0x123ABC;           /* Hex number */
 21 (0  {1
 22 (1  {1      i = ((1 + 2) *          /* Nested () */
 23 (0  {1           (3 + 4));
 24 (0  {1
 25 (0  {2      {
 26 (0  {2          int j;              /* Nested {} */
 27 (0  {1      }
 28 (0  {1      return (0);
 29 (0  {0  }
 30 (0  {0
```

```
Total number of lines: 30
Maximum nesting of () : 2
Maximum nesting of {} : 2
Number of blank lines .................6
Number of comment only lines ..........6
Number of code only lines .............8
Number of lines with code and comments 10
Comment to code ratio 88.9%
```

修改

目前，程序进行了一套非常有限的统计，你可以在每个程序中加入像平均标识符长度、每个程序的统计情况等。设计程序时，扩展的需要应该牢记在心。

统计数据只收集了四种类型的统计，因为我们已经完成了说明一套先进的C指令的使命。我们没有加入更多的C指令，因为过多的指令会使程序太复杂而不适合于本章。从整体上来讲，本程序运行良好。

最后的警告

不要因为你可以得到统计结果，就认为它是有用的。

程序文件

in_file 文件

```
/********************************************************
 * input_file -- Data from the input file.              *
 *                                                      *
 * The current two characters are stored in             *
 *      cur_char and next_char.                         *
 * Lines are buffered so that they can be output to     *
 * the screen after a line is assembled.                *
 *                                                      *
 * Functions:                                           *
```

```
 *      in_open -- Opens the input file.                   *
 *      in_close -- Closes the input file.                 *
 *      read_char   -- Reads the next character.           *
 *      in_char_char -- Returns the current character.     *
 *      in_next_char  - Returns the next character.        *
 *      in_flush -- Sends line to the screen.              *
 ********************************************************/

/********************************************************
 * in_open -- Opens the input file.                     *
 *                                                      *
 * Parameters                                           *
 *      name -- Name of disk file to use for input.     *
 *                                                      *
 * Returns                                              *
 *      0 -- Open succeeded.                            *
 *      nonzero -- Open failed.                         *
 ********************************************************/
extern int in_open(const char name[]);

/********************************************************
 * in_close -- Closes the input file.                   *
 ********************************************************/
extern void in_close(void);

/********************************************************
 * in_read_char -- Read the next character from the     *
 *      input file.                                     *
 ********************************************************/
extern void in_read_char(void);

/********************************************************
 * in_cur_char -- Gets the current input character.     *
 *                                                      *
 * Returns                                              *
 *      current character.                              *
 ********************************************************/
extern int in_cur_char(void);

/********************************************************
 * in_next_char -- Peeks ahead one character.           *
 *                                                      *
 * Returns                                              *
```

```
 *      next character.                                 *
 ********************************************************/
extern int in_next_char(void);

/********************************************************
 * in_flush -- Flushes the buffered input line to the   *
 *              screen.                                 *
 ********************************************************/
extern void in_flush(void);
```

in_file.c 文件

```
/********************************************************
 * infile module                                        *
 *      Handles opening, reading, and display of        *
 *      data from the input file.                       *
 *                                                      *
 * Functions:                                           *
 *      in_open -- Opens the input file.                *
 *      in_close -- Closes the input file.              *
 *      read_char   -- Reads the next character.        *
 *      in_char_char -- Returns the current character.  *
 *      in_next_char -- Returns the next character.     *
 *      in_flush -- Sends line to the screen.           *
 ********************************************************/
#include <stdio.h>
#include <errno.h>

#include "in_file.h"

#define LINE_MAX 500    /* Longest possible line */

struct input_file {
    FILE *file;         /* File we are reading */
    char line[LINE_MAX];/* Current line */
    char *char_ptr;     /* Current character on the line */

    int cur_char;       /* Current character (can be EOF)*/
    int next_char;      /* Next character (can be EOF)   */
};

/* Input file that we are reading */
```

```
static struct input_file in_file = {
    NULL,               /* file */
    "",                 /* line */
    NULL,               /* char_ptr */
    '\0',               /* cur_char */
    '\0',               /* next_char */
};


/********************************************************
 * in_open -- Opens the input file.                     *
 *                                                      *
 * Parameters                                           *
 *      name -- Name of disk file to use for input.     *
 *                                                      *
 * Returns                                              *
 *      0 -- Open succeeded.                            *
 *      nonzero -- Open failed.                         *
 ********************************************************/
int in_open(const char name[])
{
   in_file.file = fopen(name, "r");
   if (in_file.file == NULL)
       return (errno);

    /*
     * Initialize the input file and read the first two
     * characters.
     */
    in_file.cur_char = fgetc(in_file.file);
    in_file.next_char = fgetc(in_file.file);
    in_file.char_ptr = in_file.line;
    return (0);
}

/********************************************************
 * in_close -- Closes the input file.                   *
 ********************************************************/
void in_close(void)
{
    if (in_file.file != NULL) {
        fclose(in_file.file);
        in_file.file = NULL;
    }
```

```
}

/*****************************************************
 * in_cur_char -- Gets the current input character.  *
 *                                                   *
 * Returns                                           *
 *      current character.                           *
 *****************************************************/
int in_cur_char(void)
{
   return (in_file.cur_char);
}

/*****************************************************
 * in_next_char -- Peeks ahead one character.        *
 *                                                   *
 * Returns                                           *
 *      next character.                              *
 *****************************************************/
int in_next_char(void)
{
   return (in_file.next_char);
}

/*****************************************************
 * in_flush -- Flushes the buffered input line to the *
 *             screen.                               *
 *****************************************************/
void in_flush(void)
{
    *in_file.char_ptr = '\0';           /* End the line */
    fputs(in_file.line, stdout);        /* Send the line */
    in_file.char_ptr = in_file.line;    /* Reset the line */
}
/*****************************************************
 * in_read_char -- Reads the next character from the *
 *      input file.                                  *
 *****************************************************/
void in_read_char(void)
{
    *in_file.char_ptr = in_file.cur_char;
    ++in_file.char_ptr;
```

```
    in_file.cur_char = in_file.next_char;
    in_file.next_char = fgetc(in_file.file);
};
```

ch_type.h 文件

```
/********************************************************
 * char_type -- Character type module.                  *
 ********************************************************/
enum CHAR_TYPE {
    C_EOF,       /* End of file character */
    C_WHITE,     /* Whitespace or control character */
    C_NEWLINE,   /* A newline character */
    C_ALPHA,     /* A Letter (includes _) */
    C_DIGIT,     /* A Number */
    C_OPERATOR,  /* Random operator */
    C_SLASH,     /* The character '/' */
    C_L_PAREN,   /* The character '(' */
    C_R_PAREN,   /* The character ')' */
    C_L_CURLY,   /* The character '{' */
    C_R_CURLY,   /* The character '}' */
    C_SINGLE,    /* The character '\'' */
    C_DOUBLE,    /* The character '"' */
    /* End of simple types, more complex, derrived types follow */
    C_HEX_DIGIT,/* Hexidecimal digit */
    C_ALPHA_NUMERIC/* Alpha numeric */
};

/********************************************************
 * is_char_type -- Determines if a character belongs to *
 *                 a given character type.              *
 *                                                      *
 * Parameters                                           *
 *      ch -- Character to check.                       *
 *      kind -- Type to check it for.                   *
 *                                                      *
 * Returns:                                             *
 *      0 -- Character is not of the specified kind.    *
 *      1 -- Character is of the specified kind.        *
 ********************************************************/
extern int is_char_type(int ch, enum CHAR_TYPE kind);

/********************************************************
 * get_char_type -- Given a character, returns its type.*
```

```
 *                                                      *
 * Note: We return the simple types.  Composite types   *
 * such as C_HEX_DIGIT and C_ALPHA_NUMERIC are not      *
 * returned.                                            *
 *                                                      *
 * Parameters:                                          *
 *      ch -- Character having the type we want.        *
 *                                                      *
 * Returns                                              *
 *      character type.                                 *
 ********************************************************/
extern enum CHAR_TYPE get_char_type(int ch);
```

ch_type.c 文件

```
/********************************************************
 * ch_type package                                      *
 *                                                      *
 * This module is used to determine the type of         *
 * various characters.                                  *
 *                                                      *
 * Public functions:                                    *
 *      init_char_type -- Initializes the table.        *
 *      is_char_type -- Is a character of a given type? *
 *      get_char_type -- Given char, returns type.      *
 ********************************************************/
#include <stdio.h>

#include "ch_type.h"

/* Define the type information array */
static enum CHAR_TYPE type_info[256];
static int ch_setup = 0;        /* True if character type info setup */
/********************************************************
 * fill_range -- Fills in a range of types for the      *
 *      character type class.                           *
 *                                                      *
 * Parameters                                           *
 *      start, end -- Range of items to fill in.        *
 *      type -- Type to use for filling.                *
 ********************************************************/
static void fill_range(int start, int end, enum CHAR_TYPE type)
{
```

```
    int cur_ch; /* Character we are handling now */

    for (cur_ch = start; cur_ch <= end; ++cur_ch) {
        type_info[cur_ch] = type;
    }
}

/**********************************************************
 * init_char_type -- Initializes the char type table.     *
 **********************************************************/
static void init_char_type(void)
{
    fill_range(0, 255, C_WHITE);

    fill_range('A', 'Z', C_ALPHA);
    fill_range('a', 'z', C_ALPHA);
    type_info['_'] = C_ALPHA;

    fill_range('0', '9', C_DIGIT);

    type_info['!'] = C_OPERATOR;
    type_info['#'] = C_OPERATOR;
    type_info['$'] = C_OPERATOR;
    type_info['%'] = C_OPERATOR;
    type_info['^'] = C_OPERATOR;
    type_info['&'] = C_OPERATOR;
    type_info['*'] = C_OPERATOR;
    type_info['-'] = C_OPERATOR;
    type_info['+'] = C_OPERATOR;
    type_info['='] = C_OPERATOR;
    type_info['|'] = C_OPERATOR;
    type_info['~'] = C_OPERATOR;
    type_info[','] = C_OPERATOR;
    type_info[':'] = C_OPERATOR;
    type_info['?'] = C_OPERATOR;
    type_info['.'] = C_OPERATOR;
    type_info['<'] = C_OPERATOR;
    type_info['>'] = C_OPERATOR;

    type_info['/'] = C_SLASH;
    type_info['\n'] = C_NEWLINE;

    type_info['('] = C_L_PAREN;
    type_info[')'] = C_R_PAREN;
```

```
    type_info['{'] = C_L_CURLY;
    type_info['}'] = C_R_CURLY;

    type_info['"'] = C_DOUBLE;
    type_info['\''] = C_SINGLE;
}

/**********************************************************
 * is_char_type -- Determines if a character belongs to *
 *              a given character type.                  *
 *                                                       *
 * Parameters                                            *
 *      ch -- Character to check.                        *
 *      kind -- Type to check it for.                    *
 *                                                       *
 * Returns:                                              *
 *      0 -- Character is not of the specified kind.     *
 *      1 -- Character is of the specified kind.         *
 **********************************************************/
int is_char_type(int ch, enum CHAR_TYPE kind)
{
    if (!ch_setup) {
       init_char_type();
       ch_setup = 1;
    }

    if (ch == EOF) return (kind == C_EOF);

    switch (kind) {
        case C_HEX_DIGIT:
            if (type_info[ch] == C_DIGIT)
                return (1);
            if ((ch >= 'A') && (ch <= 'F'))
                return (1);
            if ((ch >= 'a') && (ch <= 'f'))
                return (1);
            return (0);
        case C_ALPHA_NUMERIC:
            return ((type_info[ch] == C_ALPHA) ||
                    (type_info[ch] == C_DIGIT));
        default:
            return (type_info[ch] == kind);
    }
};
```

```
/*******************************************************
 * get_char_type -- Given a character, returns its type.*
 *                                                      *
 * Note: We return the simple types.  Composite types   *
 * such as C_HEX_DIGIT and C_ALPHA_NUMERIC are not      *
 * returned.                                            *
 *                                                      *
 * Parameters:                                          *
 *      ch -- Character having the type we want.        *
 *                                                      *
 * Returns                                              *
 *      character type.                                 *
 *******************************************************/
enum CHAR_TYPE get_char_type(int ch) {
    if (!ch_setup) {
       init_char_type();
       ch_setup = 1;
    }

    if (ch == EOF) return (C_EOF);

    return (type_info[ch]);
}
```

token.h 文件

```
/*******************************************************
 * token --Token handling module.                       *
 *                                                      *
 * Functions:                                           *
 *     next_token --Gets the next token from the input. *
 *******************************************************/
/*
 * Define the enumerated list of tokens.
 */
enum TOKEN_TYPE {
   T_NUMBER,            /* Simple number (floating point or integer */
   T_STRING,            /* String or character constant */
   T_COMMENT,   /* Comment */
   T_NEWLINE,   /* Newline character */
   T_OPERATOR,  /* Arithmetic operator */
   T_L_PAREN,   /* Character "(" */
   T_R_PAREN,   /* Character ")" */
```

```
    T_L_CURLY,    /* Character "{" */
    T_R_CURLY,    /* Character "}"*/
    T_ID,                 /* Identifier */
    T_EOF                 /* End of File */
};

/*
 * We use #define here instead of "const int" because so many old
 * software packages use #define.  We must have picked
 * up a header file that uses #define for TRUE/FALSE.  Consequently,
 * we protect against double defines as well as against using #define
 * ourselves.
 */
#ifndef TRUE
#define TRUE 1          /* Define a simple TRUE/FALSE values */
#define FALSE 0
#endif /* TRUE */

/*********************************************************
 * next_token -- Reads the next token in an input stream.*
 *                                                       *
 * Parameters                                            *
 *      in_file -- File to read.                         *
 *                                                       *
 * Returns                                               *
 *      next token.                                      *
 *********************************************************/
extern enum TOKEN_TYPE next_token(void);
```

token.c 文件

```
/*********************************************************
 * token -- Token handling module.                       *
 *                                                       *
 * Functions:                                            *
 *      next_token -- Gets the next token from the input.*
 *********************************************************/
#include <stdio.h>
#include <stdlib.h>

#include "ch_type.h"
#include "in_file.h"
#include "token.h"
```

```
static int in_comment = FALSE;  /* True if we're in a comment */

/*******************************************************
 * read_comment --Reads in a comment.                  *
 *                                                     *
 * Returns                                             *
 *      Token read.  Can be a T_COMMENT or T_NEW_LINE,  *
 *      depending on what we read.                     *
 *                                                     *
 *      Multiline comments are split into multiple     *
 *      tokens.                                        *
 *******************************************************/
static enum TOKEN_TYPE read_comment(void)
{
    if (in_cur_char() == '\n') {
        in_read_char();
        return (T_NEWLINE);
    }
    while (1) {
        in_comment = TRUE;
        if (in_cur_char() == EOF) {
            fprintf(stderr, "Error: EOF inside comment\n");
            return (T_EOF);
        }
        if (in_cur_char() == '\n')
            return (T_COMMENT);
        if ((in_cur_char() == '*') &&
            (in_next_char() == '/')) {
            in_comment = FALSE;
            /* Skip past the ending */
            in_read_char();
            in_read_char();
            return (T_COMMENT);
        }
        in_read_char();
    }
}
/*******************************************************
 * next_token -- Reads the next token in an input stream.*
 *                                                     *
 * Returns                                             *
 *      next token.                                    *
 *******************************************************/
enum TOKEN_TYPE next_token(void)
```

```
{
    if (in_comment)
        return (read_comment());

    while (is_char_type(in_cur_char(), C_WHITE)) {
        in_read_char();
    }
    if (in_cur_char() == EOF)
        return (T_EOF);

    switch (get_char_type(in_cur_char())) {
        case C_NEWLINE:
            in_read_char();
            return (T_NEWLINE);
        case C_ALPHA:
            while (is_char_type(in_cur_char(), C_ALPHA_NUMERIC))
                in_read_char();
            return (T_ID);
        case C_DIGIT:
            in_read_char();
            if ((in_cur_char() == 'X') || (in_cur_char() == 'X')) {
                in_read_char();
                while (is_char_type(in_cur_char(), C_HEX_DIGIT))
                    in_read_char();
                return (T_NUMBER);
            }
            while (is_char_type(in_cur_char(), C_DIGIT))
                in_read_char();
            return (T_NUMBER);
        case C_SLASH:
            /* Check for  '/', '*' characters  */
            if (in_next_char() == '*') {
                return (read_comment());
            }
            /* Fall through */
        case C_OPERATOR:
            in_read_char();
            return (T_OPERATOR);
        case C_L_PAREN:
            in_read_char();
            return (T_L_PAREN);
        case C_R_PAREN:
            in_read_char();
            return (T_R_PAREN);
```

```
        case C_L_CURLY:
            in_read_char();
            return (T_L_CURLY);
        case C_R_CURLY:
            in_read_char();
            return (T_R_CURLY);
        case C_DOUBLE:
            while (1) {
                in_read_char();
                /* Check for end of string */
                if (in_cur_char() == '"' )
                    break;

                /* Escape character, then skip the next character */
                if (in_cur_char() == '\\')
                    in_read_char();
            }
            in_read_char();
            return (T_STRING);
        case C_SINGLE:
            while (1) {
                in_read_char();
                /* Check for end of character */
                if (in_cur_char() == '\'')
                    break;

                /* Escape character, then skip the next character */
                if (in_cur_char() == '\\')
                    in_read_char();
            }
            in_read_char();
            return (T_STRING);
        default:
            fprintf(stderr, "Internal error: Very strange character\n");
            abort();
    }
    fprintf(stderr, "Internal error: We should never get here\n");
    abort();
    return (T_EOF);     /* Should never get here either */
                        /* But we put in the return to avoid a compiler */
                        /* warning. */
}
```

stat.c 文件

```
/********************************************************
 * stat                                                 *
 *      Produces statistics about a program.            *
 *                                                      *
 * Usage:                                               *
 *      stat [options] <file-list>                      *
 *                                                      *
 ********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

#include "ch_type.h"
#include "in_file.h"
#include "token.h"

/********************************************************
 ********************************************************
 ********************************************************
 * line_counter -- Handles line number / line count     *
 *              stat.                                   *
 *                                                      *
 * Counts the number of T_NEW_LINE tokens seen and      *
 * outputs the current line number at the beginning     *
 * of the line.                                         *
 *                                                      *
 * At EOF, it will output the total number of lines.    *
 ********************************************************/
static int cur_line;            /* Current line number */

/********************************************************
 * lc_init -- Initializes the line counter variables.   *
 ********************************************************/
static void lc_init(void)
{
    cur_line = 0;
};

/********************************************************
 * lc_take_token -- Consumes tokens and looks for       *
 *              end-of-line tokens.                     *
 *                                                      *
```

```
 * Parameters                                       *
 *      token -- The token coming in from the input   *
 *                       stream.                    *
 ******************************************************/
static void lc_take_token(enum TOKEN_TYPE token) {
    if (token == T_NEWLINE)
    ++cur_line;
}

/******************************************************
 * lc_line_start -- Outputs the per-line statistics,  *
 *                namely the current line number.     *
 ******************************************************/
static void lc_line_start(void) {
    printf("%4d ", cur_line);
}

/******************************************************
 * lc_eof -- Outputs the eof statistics.              *
 *               In this case, the number of lines.   *
 ******************************************************/
static void lc_eof(void) {
    printf("Total number of lines: %d\n", cur_line);
}

/******************************************************
 ******************************************************
 ******************************************************
 * paren_count -- Counts the nesting level of ().     *
 *                                                    *
 * Counts the number of T_L_PAREN vs T_R_PAREN tokens  *
 * and writes the current nesting level at the beginning*
 * of each line.                                      *
 *                                                    *
 * Also keeps track of the maximum nesting level.     *
 ******************************************************/
static int pc_cur_level;
static int pc_max_level;

/******************************************************
 * pc_init -- Initializes the () counter variables.   *
 ******************************************************/
void pc_init(void) {
    pc_cur_level = 0;
```

```
    pc_max_level = 0;
};

/*******************************************************
 * pc_take_token -- Consumes tokens and looks for      *
 *              () tokens.                             *
 *                                                     *
 * Parameters                                          *
 *      token -- The token coming in from the input    *
 *                      stream.                        *
 *******************************************************/
void pc_take_token(enum TOKEN_TYPE token) {
    switch (token) {
        case T_L_PAREN:
            ++pc_cur_level;
            if (pc_cur_level > pc_max_level)
                pc_max_level = pc_cur_level;
            break;
        case T_R_PAREN:
            --pc_cur_level;
            break;
        default:
            /* Ignore */
            break;
    }
}

/*******************************************************
 * pc_line_start -- Outputs the per-line statistics,   *
 *              namely the current () nesting.         *
 *******************************************************/
static void pc_line_start(void) {
   printf("(%-2d ", pc_cur_level);
}

/*******************************************************
 * pc_eof -- Outputs the eof statistics.               *
 *              In this case, the max nesting of ().   *
 *******************************************************/
void pc_eof(void) {
   printf("Maximum nesting of () : %d\n", pc_max_level);
}
```

```
/********************************************************
 ********************************************************
 ********************************************************
 * brace_counter -- Counts the nesting level of {}.     *
 *                                                      *
 * Counts the number of T_L_CURLY vs T_R_CURLY tokens   *
 * and writes the current nesting level at the beginning*
 * of each line.                                        *
 *                                                      *
 * Also, keeps track of the maximum nesting level.      *
 *                                                      *
 * Note: brace_counter and paren_counter should         *
 * probably be combined.                                *
 ********************************************************/
static int bc_cur_level;        /* Current nesting level */
static int bc_max_level;        /* Maximum nesting level */

/********************************************************
 * pc_init -- Initialize the {} counter variables.      *
 ********************************************************/
void bc_init(void) {
    bc_cur_level = 0;
    bc_max_level = 0;
};

/********************************************************
 * bc_take_token -- Consumes tokens and looks for       *
 *               {} tokens.                             *
 *                                                      *
 * Parameters                                           *
 *      token -- The token coming in from the input     *
 *                      stream.                         *
 ********************************************************/
void bc_take_token(enum TOKEN_TYPE token) {
    switch (token) {
        case T_L_CURLY:
            ++bc_cur_level;
            if (bc_cur_level > bc_max_level)
                bc_max_level = bc_cur_level;
            break;
        case T_R_CURLY:
            --bc_cur_level;
            break;
        default:
```

```
            /* Ignore */
            break;
    }
}

/********************************************************
 * bc_line_start -- Outputs the per-line statistics,    *
 *              namely the current {} nesting.          *
 ********************************************************/
static void bc_line_start(void) {
    printf("{%-2d ", bc_cur_level);
}

/********************************************************
 * bc_eof -- Outputs the eof statistics.                *
 *              In this case, the max nesting of {}.    *
 ********************************************************/
static void bc_eof(void) {
   printf("Maximum nesting of {} : %d\n", bc_max_level);
}

/********************************************************
 ********************************************************
 ********************************************************
 * comment_counter -- Counts the number of lines        *
 *      with and without comments.                      *
 *                                                      *
 * Outputs nothing at the beginning of each line, but   *
 * will output a ratio at the end of file.              *
 *                                                      *
 * Note: This class makes use of two bits:              *
 *      CF_COMMENT  -- a comment was seen               *
 *      CF_CODE     -- code was seen                    *
 * to collect statistics.                               *
 *                                                      *
 * These are combined to form an index into the counter *
 * array so the value of these two bits is very         *
 * important.                                           *
 ********************************************************/
static const int CF_COMMENT = (1<<0);   /* Line contains comment */
static const int CF_CODE    = (1<<1);   /* Line contains code */
/* These bits are combined to form the statistics */
/*      0                   -- [0] Blank line */
/*      CF_COMMENT          -- [1] Comment-only line */
```

```
/*      CF_CODE             -- [2] Code only line */
/*      CF_COMMENT|CF_CODE -- [3] Comments and code on this line */

static int counters[4]; /* Count of various types of stats. */
static int flags;       /* Flags for the current line */

/******************************************************
 * cc_init -- Initializes the comment counter variables.*
 ******************************************************/
static void cc_init(void) {
    memset(counters, '\0', sizeof(counters));
    flags = 0;
};

/******************************************************
 * cc_take_token -- Consumes tokens and looks for       *
 *              comments tokens.                        *
 *                                                      *
 * Parameters                                           *
 *      token -- The token coming in from the input     *
 *                      stream.                         *
 ******************************************************/
void cc_take_token(enum TOKEN_TYPE token) {
    switch (token) {
        case T_COMMENT:
            flags |= CF_COMMENT;
            break;
        default:
            flags |= CF_CODE;
            break;
        case T_NEWLINE:
            ++counters[flags];
            flags = 0;
            break;
    }
}

/******************************************************
 * cc_line_start -- Outputs the per-line statistics.    *
 ******************************************************/
static void cc_line_start(void)
{
   /* Do nothing */
}
```

```
/****************************************************
 * cc_eof -- Outputs the eof statistics.            *
 *              In this case, the comment/code ratios.  *
 ****************************************************/
static void cc_eof(void) {
    printf("Number of blank lines .................%d\n",
            counters[0]);
    printf("Number of comment only lines ..........%d\n",
            counters[1]);
    printf("Number of code only lines .............%d\n",
            counters[2]);
    printf("Number of lines with code and comments %d\n",
            counters[3]);
    printf("Comment to code ratio %3.1f%%\n",
       (float)(counters[1] + counters[3]) /
       (float)(counters[2] + counters[3]) * 100.0);
}

/****************************************************
 * do_file -- Processes a single file.              *
 *                                                  *
 * Parameters                                       *
 *      name -- The name of the file to process.    *
 ****************************************************/
static void do_file(const char *const name)
{
    enum TOKEN_TYPE cur_token;    /* Current token type */

    /*
     * Initialize the counters
     */
    lc_init();
    pc_init();
    bc_init();
    cc_init();

    if (in_open(name) != 0) {
        printf("Error: Could not open file %s for reading\n", name);
        return;
    }
    while (1) {
        cur_token = next_token();

        lc_take_token(cur_token);
```

```
        pc_take_token(cur_token);
        bc_take_token(cur_token);
        cc_take_token(cur_token);

        switch (cur_token) {
            case T_NEWLINE:
                lc_line_start();
                pc_line_start();
                bc_line_start();
                cc_line_start();
                in_flush();
                break;
            case T_EOF:
                lc_eof();
                pc_eof();
                bc_eof();
                cc_eof();
                in_close();
                return;
            default:
                /* Do nothing */
                break;
        }
    }
}

int main(int argc, char *argv[])
{
    char *prog_name = argv[0];  /* Name of the program */

    if (argc == 1) {
        printf("Usage is %s [options] <file-list>\n", prog_name);
        exit (8);
    }

    for (/* argc set */; argc > 1; --argc) {
       do_file(argv[1]);
       ++argv;
    }
    return (0);
}
```

用于CC（一般Unix系统）的UNIX Makefile

```
# File: stat/makefile.unx

#
# Makefile for the UNIX standard cc compiler
#
CC=cc
CFLAGS=-g
OBJS= stat.o ch_type.o token.o in_file.o

all: stat.out stat test.out

test.out: test.c stat
	stat test.c >test.out

# This generates a test output based on another example
# in this book.
stat.out: stat
	stat ../calc3/calc3.c >stat.out

stat: $(OBJS)
	$(CC) $(CFLAGS) -o stat $(OBJS)

stat.o: stat.c token.h
	$(CC) $(CFLAGS) -c stat.c

ch_type.o: ch_type.c ch_type.h
	$(CC) $(CFLAGS) -c ch_type.c

token.o: token.c token.h ch_type.h in_file.h
	$(CC) $(CFLAGS) -c token.c

in_file.o: in_file.c in_file.h
	$(CC) $(CFLAGS) -c in_file.c

clean:
	rm -f stat stat.o ch_type.o token.o in_file.o
```

用于gcc的UNIX Makefile

```
# File: stat/makefile.gcc
```

```
#
# Makefile for the Free Software Foundations g++ compiler
#
CC=gcc
CFLAGS=-g -Wall -D__USE_FIXED_PROTOTYPES__
OBJS= stat.o ch_type.o token.o in_file.o

all: stat.out stat test.out

test.out: test.c stat
	stat test.c >test.out

# This generates a test output based on another example
# in this book.
stat.out: stat
	stat ../calc3/calc3.c >stat.out
stat: $(OBJS)

	$(CC) $(CFLAGS) -o stat $(OBJS)

stat.o: stat.c token.h
	$(CC) $(CFLAGS) -c stat.c
ch_type.o: ch_type.c ch_type.h

	$(CC) $(CFLAGS) -c ch_type.c
token.o: token.c token.h ch_type.h in_file.h

	$(CC) $(CFLAGS) -c token.c

in_file.o: in_file.c in_file.h
	$(CC) $(CFLAGS) -c in_file.c
clean:
	rm -f stat stat.o ch_type.o token.o in_file.o
```

Turbo C++ 的 Makefile

```
# File: stat/makefile.tcc

#
# Makefile for Borland's Borland-C++ compiler
#
CC=tcc
#
```

```
# Flags
#       -N  -- Check for stack overflow.
#       -v  -- Enable debugging.
#       -w  -- Turn on all warnings.
#       -ml -- Large model.
#
CFLAGS=-N -v -w -ml
OBJS= stat.obj ch_type.obj token.obj in_file.obj

all: stat.out stat.exe test.out

test.out: test.c stat.exe
        stat test.c >test.out

# This generates a test output based on another example
# in this book.
stat.out: stat.exe
        stat ..\calc3\calc3.c >stat.out

stat.exe: $(OBJS)
        $(CC) $(CFLAGS) -estat $(OBJS)

stat.obj: stat.c token.h
        $(CC) $(CFLAGS) -c stat.c

in_file.obj: in_file.c in_file.h
        $(CC) $(CFLAGS) -c in_file.c

ch_type.obj: ch_type.c ch_type.h
        $(CC) $(CFLAGS) -c ch_type.c

token.obj: token.c token.h ch_type.h
        $(CC) $(CFLAGS) -c token.c


clean:
        erase stat.exe
        erase stat.obj
        erase ch_type.obj
        erase in_file.obj
        erase token.obj
```

Borland C++ 的 Makefile

```
# File: stat/makefile.bcc

#
# Makefile for Borland's Borland C++ compiler
#
CC=bcc
#
# Flags
#       -N  -- Check for stack overflow.
#       -v  -- Enable debugging.
#       -w  -- Turn on all warnings.
#       -ml -- Large model.
#
CFLAGS=-N -v -w -ml
OBJS= stat.obj ch_type.obj token.obj in_file.obj

all: stat.out stat.exe test.out

test.out: test.c stat.exe
        stat test.c >test.out

# This generates a test output based on another example
# in this book.
stat.out: stat.exe
        stat ..\calc3\calc3.c >stat.out

stat.exe: $(OBJS)
        $(CC) $(CFLAGS) -estat $(OBJS)

stat.obj: stat.c token.h
        $(CC) $(CFLAGS) -c stat.c

in_file.obj: in_file.c in_file.h
        $(CC) $(CFLAGS) -c in_file.c

ch_type.obj: ch_type.c ch_type.h
        $(CC) $(CFLAGS) -c ch_type.c

token.obj: token.c token.h ch_type.h
        $(CC) $(CFLAGS) -c token.c
```

```
clean:
        erase stat.exe
        erase stat.obj
        erase ch_type.obj
        erase in_file.obj
        erase token.obj
```

Microsoft Visual C++ 的 Makefile

```
# File: stat/makefile.msc
#
# Makefile for Microsoft Visual C++
#
CC=cl
#
# Flags
#       AL -- Compile for large model.
#       Zi -- Enable debugging.
#       W1 -- Turn on warnings.
#
CFLAGS=/AL /Zi /W1
OBJS= stat.obj ch_type.obj token.obj in_file.obj

all: stat.out stat.exe test.out

test.out: test.c stat.exe
        stat test.c >test.out

# This generates a test output based on another example
# in this book.
stat.out: stat.exe
        stat ..\calc3\calc3.c >stat.out

stat.exe: $(OBJS)
        $(CC) $(CCFLAGS)  $(OBJS)

stat.obj: stat.c token.h
        $(CC) $(CCFLAGS) -c stat.c

ch_type.obj: ch_type.c ch_type.h
        $(CC) $(CCFLAGS) -c ch_type.c
```

```
token.obj: token.c token.h ch_type.h
        $(CC) $(CCFLAGS) -c token.c

in_file.obj: in_file.c
        $(CC) $(CCFLAGS) -c in_file.c


clean:
        erase stat.exe
        erase stat.obj
        erase ch_type.obj
        erase token.obj
        erase in_file.obj
```

编程练习

练习 22-1: 写一个程序，用来检查文本文件中的双单词（例如“in the *the* file”）。

练习 22-2: 写一个程序，用来删除一个文件中所有由四个字母组成的单词，并把它们替换为可以接受的字。

练习 22-3: 写一个邮件列表程序。该程序能进行读、写、排序并输出邮件标签

练习 22-4: 更新本章统计程序，加入一个交叉引用功能。

练习 22-5: 写一个程序，用来得到一个文本文件，并把每个长行划分为两个短一些的行。如果可能的话，分隔点应该在句子的结束处；如果句子太长，分隔点在字与字之间。

第二十三章

程序设计格言

本章内容

- 概述
- 设计
- 定义
- switch 语句
- 预处理器
- 风格
- 编译
- 最后的注解
- 答案

三思而行。

——欧里庇德（古希腊戏剧家）

概述

- 注释、注释、注释。在你的程序中加入大量注释，它们告诉其他程序员你做过的处理，同时也告诉你自己所做过的处理。
- 使用“KISS”原则。（Keep It Simple, Stupid. 保持程序简单。）清晰和简单比复杂和玄妙更好。
- 避免副作用。使用 ++ 和 -- 要单独占一行。
- 使用前缀版的 ++ 和 --（++x，--x），不要用后缀版（x++，x--）。这条格言在 C 中没有用处，但当你开始用 C++ 时，它就会有用了。
- 决不要把赋值语句放在条件中。
- 决不要把赋值语句放在任何其他语句中。
- 了解 = 和 == 的区别。使用 = 代替 == 是常见的错误且很难发现。
- 永远不要不声不响地做“空事”。

  ```
  /* Don't program like this */
  for (index = 0; data[index] < key; ++index);
  /* Did you see the semicolon at the end of the last line? */
  ```

  总是放一个注释或语句。

```
for (index = 0; data[index] < key; ++index)
        continue;
```

设计

- 设计程序时，头脑中始终应有“平淡无奇法则”，强调的是程序应该以一种给用户最少惊奇的方式运行。
- 令用户界面尽可能简单一致。
- 给用户尽可能多的帮助。
- 所有的错误信息用单词“error”清楚地标识，并尽量给用户一些修改错误的提示。

定义

- 一个变量定义占一行，并注释它们。
- 让变量名足够长，以易于理解，但不要太长，免得键入困难。通常两到三个单词就足够了。
- 决不要使用默认定义。如果函数返回一个整数，把它的类型定义为**int**。
- 所有的函数参数应该定义和注释，决不要用默认定义。
- 总要把 main 定义成：

```
int main(void)                        /* Correct declaration */
int main(int argc, char *argv[])     /* Also correct */
```

决不要定义成：

```
void main()                           /* never program like this */
void main(int ac, char *av[])        /* never use names like this */
```

switch 语句

- 在**switch**语句中一定要放一个缺省情况，即使它什么也不做，也要放进去。

```
switch (expression) {
    /* ... */
    default:
            /* do nothing */
}
```

- switch中的任何情况都应该以**break**结束或以 /*Fall through*/ 注释。

预处理器

- 在预处理器中，由**#define**指令定义的常量表达式应总是用括号括起来。

```
#define BOX_SIZE (3*10) /* size of the box in pixels */
```

- 定义时尽可能地用**const**代替**#define**。

- 给带参数的宏中的每个自变量加括号( )。

```
#define SQUARE(x) ((x) * (x))
```

- 包含完整语句的宏应加括号({})。

```
/* A fatal error has occurred. Tell user and abort */
#define DIE(msg) {printf(msg);exit(8);}
```

- 在条件编译中使用**#ifdef/#endif**指令时，把**#define**和**#undef**放在程序顶端并注释它们。

风格

- 包含在{}中的一个代码块不应长过两页纸，过大的程序可以分为几个较小较简单的程序。

- 当你的代码开始遇到右边界时，应该把程序分成几个小而简单的程序语句。

编译

- 总是建立一个*Makefile*，使其他人知道如何编译你的程序。

- 打开所有的警告标志，使你的程序对错误有最强的敏感性。

最后的注解

当你认为已知道了C能做的所有事情的时候——再想一想，还会有更惊奇的发现。

问题 23-1：为什么例23-1把所有的值都认为是2？(由此而产生最后一条格言。)

例 23-1: not2/not2.c

```
#include <stdio.h>
int main()
{
    char line[80];
    int number;
    printf("Enter a number: ");
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &number);
    if (number =! 2)
        printf("Number is not two\n");
    else
        printf("Number is two\n");
    return (0);
}
```

答案

解答 23-1：语句 (number =! 2) 不是关系等式，而是一个赋值语句。它等价于：

```
number = (!2);
```

因为 2 是非零值，所以 !2 是零。

程序员常常把不等于弄颠倒，!= 于是变成了 =!。语句应该写成：

```
if (number != 2)
```

第四部分

其他语言特性

附录收集了本书涉及的一些较为深奥的内容，但这些内容如果包括在参考资料里将更合适。

- 附录一“ASCII表”，列出了现在差不多全世界通用的ASCII字符集的八进制、十六进制和十进制表示法。
- 附录二“范围和参数传递转换”，列出了用不同大小的内存分配来处理数据时可能会出现的问题。
- 附录三“运算符优先规则”，列出了一些很难记忆的规则，当你遇到粗心人写的未使用足够括号的代码时，它会给你提供帮助。
- 附录四“使用幂级数计算正弦函数的程序”，列出了浮点（实）数的操作，这是在本书其它章节中未给予足够重视的一个问题。
- “词汇表”定义了本书中使用的一些技术性的专业词语。

附录一

ASCII表

表A-1　ASCII字符表

十进制	八进制	十六进制	字符	十进制	八进制	十六进制	字符
0	000	00	NUL	25	031	19	EM
1	001	01	SOH	26	032	1A	SUB
2	002	02	STX	27	033	1B	ESC
3	003	03	ETX	28	034	1C	FS
4	004	04	EOT	29	035	1D	GS
5	005	05	ENQ	30	036	1E	RS
6	006	06	ACK	31	037	1F	US
7	007	07	BEL	32	040	20	SP
8	010	08	BS	33	041	21	!
9	011	09	HT	34	042	22	"
10	012	0A	NL	35	043	23	#
11	013	0B	VT	36	044	24	$
12	014	0C	NP	37	045	25	%
13	015	0D	CR	38	046	26	&
14	016	0E	SO	39	047	27	'
15	017	0F	SI	40	050	28	(
16	020	10	DLE	41	051	29	)
17	021	11	DC1	42	052	2A	*
18	022	12	DC2	43	053	2B	+
19	023	13	DC3	44	054	2C	,
20	024	14	DC4	45	055	2D	-
21	025	15	NAK	46	056	2E	.
22	026	16	SYN	47	057	2F	/
23	027	17	ETB	48	060	30	0
24	030	18	CAN	49	061	31	1

表 A-1　ASCII 字符表（续）

十进制	八进制	十六进制	字符	十进制	八进制	十六进制	字符
50	062	32	2	89	131	59	Y
51	063	33	3	90	132	5A	Z
52	064	34	4	91	133	5B	[
53	065	35	5	92	134	5C	\
54	066	36	6	93	135	5D	]
55	067	37	7	94	136	5E	^
56	070	38	8	95	137	5F	_
57	071	39	9	96	140	60	`
58	072	3A	:	97	141	61	a
59	073	3B	;	98	142	62	b
60	074	3C	<	99	143	63	c
61	075	3D	=	100	144	64	d
62	076	3E	>	101	145	65	e
63	077	3F	?	102	146	66	f
64	100	40	@	103	147	67	g
65	101	41	A	104	150	68	h
66	102	42	B	105	151	69	i
67	103	43	C	106	152	6A	j
68	104	44	D	107	153	6B	k
69	105	45	E	108	154	6C	l
70	106	46	F	109	155	6D	m
71	107	47	G	110	156	6E	n
72	110	48	H	111	157	6F	o
73	111	49	I	112	160	70	p
74	112	4A	J	113	161	71	q
75	113	4B	K	114	162	72	r
76	114	4C	L	115	163	73	s
77	115	4D	M	116	164	74	t
78	116	4E	N	117	165	75	u
79	117	4F	O	118	166	76	v
80	120	50	P	119	167	77	w
81	121	51	Q	120	170	78	x
82	122	52	R	121	171	79	y
83	123	53	S	122	172	7A	z
84	124	54	T	123	173	7B	{
85	125	55	U	124	174	7C	|
86	126	56	V	125	175	7D	}
87	127	57	W	126	176	7E	~
88	130	58	X	127	177	7F	DEL

附录二

范围和参数传递转换

范围

表 B-1 和表 B-2 列示了不同变量类型的范围。

表 B-1　32 位 UNIX 机器

名称	位	下限值	上限值	精度
整	32	-2147483648	2147483647	
短整	16	-32768	32767	
长整	32	-2147483648	2147483647	
无符号整	32	0	4294967295	
无符号短整	16	0	65535	
无符号长整	32	0	4294967295	
字符	8	依系统而定		
无符号字符	8	0	255	
浮点	32	-3.4E+38	3.4E+38	6 位
双	64	-1.7E+308	1.7E+308	15 位
长双	64	-1.7E+308	1.7E+308	15 位

表 B-2 Turbo C++, Borland C++ 和其他 16 位系统

名称	位	下限值	上限值	精度
整	16	-32768	32767	
短整	16	-32768	32767	
长整	32	-2147483648	2147483647	
无符号整	16	0	65535	
无符号短整	16	0	65535	
无符号长整	32	0	4294967295	
字符	8	-128	127	
无符号字符	8	0	255	
浮点	32	-3.4E+38	3.4E+38	6 位
双	64	-1.7E+308	1.7E+308	15 位
长双	80	-3.4E+4932	1.7E+308	17 位

传递参数时自动类型转换的使用

为了消除向函数传递参数时可能发生的问题,C对下列函数自变量执行自动转换、如表 B-3。

表 B-3 自动转换

类型	转换为
字符	整
短整	整
整	整
长整	长整
浮点	双
双	双
长双	双
数组	指针

附录三

运算符优先规则

标准规则

表C-1中，越靠上面的运算符优先级越高。

表C-1　C优先规则

优先等级	运算符				
1.	()	[]	->	.	
2.	!	~	++	--	(类型转换运算符)
	-（负号运算符）	*（指针运算符）			
	&（地址运算符）	（长度运算符）			
3.	*（乘法运算符）	/	%		
4.	+	-			
5.	<<	>>			
6.	<	<=	>	>=	
7.	==	!=			
8.	&（按位与运算符）				
9.	^				
10.	|				
11.	&&				
12.	||				
13.	?:				
14.	=	+=	-=	etc.	
15.	,				

实用子集

表 C-2　优先规则，实用子集

优先等级	运算符		
1.	*（乘法运算符）	/	%
2.	+	-	

在所有该加括号的地方都加上括号。

附录四

使用幂级数计算正弦函数的程序

本程序的设计目的是使用幂级数计算正弦函数，一个很有限的浮点数格式用来说明使用浮点数时会发生的问题。

程序由下列语句调用:

```
sine value
```

其中 *value* 是一个弧度角。

程序计算幂级数中的每一项，并显示结果。一直计算下去，直到最后一项小得对最终结果没有影响。

为比较起见，我们把库函数 sin 的结果和计算的正弦结果一起列出。

sine.c 程序

例 D-1: sine/sine.c

```
[File: sine/sine.c]
/********************************************************
 * sine -- Computes sine using very simple floating     *
 *      arithmetic.                                     *
```

```
 *                                                        *
 * Usage:                                                 *
 *      sine <value>                                      *
 *                                                        *
 *      <value> is an angle in radians                    *
 *                                                        *
 * Format used in f.fffe+X                                *
 *                                                        *
 * f.fff is a 4-digit fraction                            *
 *      + is a sign (+ or -)                              *
 *      X is a single digit exponent                      *
 *                                                        *
 * sine(x) = x  - x**3 + x**5 - x**7                      *
 *                -----  -----  ----- . . . .             *
 *                  3!     5!     7!                      *
 *                                                        *
 * Warning: This program is intended to show some of the  *
 *      problems with floating-point. It is not intended  *
 *      to be used to produce exact values for the        *
 *      sine function.                                    *
 *                                                        *
 * Note: Even though we specify only one digit for the    *
 *       exponent, two are used for some calculations.    *
 *       We have to do this because printf has no         *
 *       format for a single digit exponent.              *
 **********************************************************/
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

/**********************************************************
 * float_2_ascii -- Turns a floating-point string         *
 *      into ascii.                                       *
 *                                                        *
 * Parameters                                             *
 *      number -- Number to turn into ascii.              *
 *                                                        *
 * Returns                                                *
 *      Pointer to the string containing the number.      *
 *                                                        *
 * Warning: Uses static storage, so later calls           *
 *              overwrite earlier entries.                *
 **********************************************************/
static char *float_2_ascii(float number)
```

```
{
    static char result[10]; /*place to put the number */
    sprintf(result,"%8.3E", number);
    return (result);
}
/********************************************************
 * fix_float -- Turns high-precision numbers into        *
 *              low-precision numbers to simulate a      *
 *              very dumb floating-point structure.      *
 *                                                       *
 * Parameters                                            *
 *      number -- Number to take care of.                *
 *                                                       *
 * Returns                                               *
 *      Number accurate to five places only.             *
 *                                                       *
 * Note: This works by changing a number into ascii and  *
 *       back. Very slow, but it works.                  *
 ********************************************************/
float fix_float(float number)
{
    float   result; /* result of the conversion */
    char    ascii[10];      /* ascii version of number */
    sprintf(ascii,"%8.4e", number);
    sscanf(ascii, "%e", &result);
    return (result);
}
/********************************************************
 * factorial -- Computes the factorial of a number.      *
 *                                                       *
 * Parameters                                            *
 *      number -- Number to use for factorial.           *
 *                                                       *
 * Returns                                               *
 *      Factorial(number) or number!                     *
 *                                                       *
 * Note: Even though this is a floating-point routine,   *
 *       using numbers that are not whole numbers        *
 *       does not make sense.                            *
 ********************************************************/
float factorial(float number)
{
    if (number <= 1.0)
        return (number);
```

```
    else
        return (number *factorial(number - 1.0));
}

int main(int argc, char *argv[])
{
    float   total;   /* total of series so far */
    float   new_total;/* newer version of total */
    float   term_top;/* top part of term */
    float   term_bottom;/* bottom of current term */
    float   term;    /* current term */
    float   exp;     /* exponent of current term */
    float   sign;    /* +1 or -1 (changes on each term) */
    float   value;   /* value of the argument to sin */
    int     index;   /* index for counting terms */

    if (argc != 2) {
        fprintf(stderr,"Usage is:\n");
        fprintf(stderr,"  sine <value>\n");
        exit (8);
    }
    value = fix_float(atof(&argv[1][0]));

    total = 0.0;
    exp = 1.0;
    sign = 1.0;

    for (index = 0; /* take care of below */ ; ++index) {
        term_top = fix_float(pow(value, exp));
        term_bottom = fix_float(factorial(exp));
        term = fix_float(term_top / term_bottom);
        printf("X**%d     %s\n", (int)exp,
                        float_2_ascii(term_top));
        printf("%d!       %s\n", (int)exp,
                        float_2_ascii(term_bottom));
        printf("X**%d/%d! %s\n", (int)exp, (int)exp,
                float_2_ascii(term));
        printf("\n");
        new_total = fix_float(total + sign * term);
        if (new_total == total)
            break;
        total = new_total;
        sign = -sign;
        exp = exp + 2.0;
```

```
        printf("  total   %s\n", float_2_ascii(total));
        printf("\n");
    }
    printf("%d term computed\n", index+1);
    printf("sin(%s)=\n", float_2_ascii(value));
    printf("  %s\n", float_2_ascii(total));
    printf("Actual sin(%G):%G\n",
           atof(&argv[1][0]), sin(atof(&argv[1][0])));
    return (0);
}
```

词汇表

! 逻辑非运算符。

!= 不等于关系运算符。

" 见“双引号”词条。

% 取模运算符。

& 1)按位与运算符。

2)放在变量名前的符号(如&x)，意思是变量的地址（x的地址）。用来把一个值赋给一个指针变量。

&& 逻辑与运算符(用于比较运算)。

' 见“单引号”词条。

* 1)乘法运算符。

2)用来放在指针变量名字前面的符号，意思是“得到由指针变量指向的地址里储存的值。”(*x意思是“得到x中储存的值”)。有时也叫逆引用运算符或间接运算符。

+ 加运算符。

++ 增量运算符。

, 逗号字符是一个模糊C运算符，可以用来把两条语句联结成一条语句。

- 减号运算符。

-- 减量运算符。

-> 用来从一个类或结构指针得到内容。

/ 除法运算符。

< 小于关系运算符。

<< 左移运算符。

<=　小于或等于关系运算符。

==　等于关系运算符。

>　大于关系运算符。

>=　大于或等于关系运算符。

>>　右移运算符。

?:　C 运算符，允许在一个表达式中放一个条件，很少使用。

^　按位异或运算符。

\　用于串中标记一个特殊字符的字符。

\b　退格字符（在多数输出设备上把光标回移一位）。

\f　换页符。（在多数打印机上，走一页纸。在许多终端上这个字符将清屏。）

\n　换行符。把光标移到下一行开头。

{}　见“卷括号”词条

|　按位或运算符

||　逻辑或运算符

~　按位补运算符。所有位取反。

'\0'　串尾符(NULL 字符)

#define

　C 预处理器命令，为一个名字定义替换文本。

#endif

　预处理器宏块的关闭括号。开始是 #ifdef 指令。

#ifdef

　预处理器指令，检查是否定义一个宏名。如果定义了，则后续代码加入源文件。

#ifndef

　预处理器命令，检查是否宏名没有定义。如果没有定义，则后续代码将加到扩展的宏中。

#include

　预处理器命令，它使指定文件插入到 #include 的位置。

#undef

　预处理器命令，它取消 #define。

_ptr　本书使用的习惯，所有的指针变量以扩展名 _ptr 结尾。

Accuracy 精度

　衡量实数表示中内在误差的一个量。

Address 地址

表示内存中存储位置的值。

and 与

布尔运算，如果两个操作数中有一个为0，则结果为0；如果两个操作数均为1，则结果为1。

ANSI-C

符合美国标准研究院（American National Standards Institute）X3J委员会规范的任何一个C版本。

ANSI-C++

符合美国标准研究院（American National Standards Institute）规范的任何一个C++版本。到本书写作时，标准只有草案形式，还有许多细节仍在探讨。

API 应用程序接口（Application Programming Interface）

由系统提供给程序员的一套函数调用。通常用于MS-DOS/Windows编程中。

Archive 档案库

见“库”。

Array 数组

一维或多维的按下标排列的数据元素集合。在C中，数组存储于连续内存中。

ASCII

美国信息交换用标准代码（American Standard Code for Information Interchange），一种表示字符的编码。

Assignment statment 赋值语句

把值存储在变量中的操作。

auto

C关键字，用来建立一个临时变量。由于很少使用，被默认为自动变量。

Automatic variable 自动变量

见“临时变量”。

Base 基类

见“基数”。

Bit 位

二进制数字；可以为数字0或1。

Bit field 位字段

作为一个整体的一组连续的位，C++语言的这个特征允许访问独立的位。

Bit flip 位反向

反转一个操作数中的所有位，又见“补”。

Bit-mapped graphics 位图

计算机图形，图形输出设备中的每个像素都由一个位或一组位来控制。

Bit operators 位运算符

见“按位运算符”。

Bitwise operator 按位运算符

对两个操作数执行布尔操作的运算符。它把操作数中的每一位看作是独立的位，一位一位地执行相应的运算。

Block 块

包含在卷括号{}中的一段代码。

Boolean 布尔运算

一种运算或值，可以返回真或假的结果。

Borland C++

由Borland公司开发的用于个人计算机的一个C++语言版本。它是Borland公司的Turbo-C++产品的更高级版本。这个产品可以处理C和C ++代码。

Boxing (a comment) 框（注释）

使用垂直和水平星号及其它印刷字符在注释周围所画的框，为的是把它和代码分开。

break 中断

for、while、switch和do/while语句最内层执行的一种语句。

Breakpoint 断点

程序中的一个位置。程序执行到此时，挂起正常的执行，控制回到调试程序中。

Buffered I/O 缓冲I/O

在I/O流的源和目标之间使用中间存储(一个缓冲区)的输入/输出。

Byte 字节

八个位组成的组。

C

由Dennis Ritchie于1974年在贝尔实验室开发的通用计算机程序设计语言。C被认为是一种中级到高级之间的语言。

C++

基于C的一种语言，它是Bjarne Stroustrup于1980年开发的，最初叫“带类的C”，它已经发展成自己的语言。

C code C代码

用C语言编写的计算机指令。

C compiler C编译器

把C源代码翻译为机器代码的软件。

C syntax C语法

见"语法"

Call by value 值调用

一种过程调用。其中，通过传递参数的值来传递参数。

case

当作switch语句中可选项之一的标号。

Cast 替换

通过指明变量的类型，把一个变量从一种类型转换为另一种类型。

CGA

彩色图形适配器。用于IBM PC的一种常见彩色图形卡。

char

C关键字，用来说明表示字符或小整数的变量。

Class (of a variable) （变量的）类

见"存储类"。

Clear a bit 清除一个位

把一个独立的位设置为0的操作。在C++中没有定义这个操作。

Code design 代码设计

用一般术语描述程序如何完成它的功能的一个文档。

Coding 编码

用一种计算机语言编程的行为。

Command-line options 命令行选项

控制像编译器这样的程序运行的选项，它从计算机控制台键入。

Comment 注释

包含在计算机程序中的文本，它仅仅是为了提供程序的有关信息。注释是一个程序员对自己及未来程序员的注解。这些文本被编译器所忽略。

Comment block 注释块

一组相关的注释，它表达一个程序或一段程序的一般信息。

Compilation 编译

把源代码翻译为机器代码。

Compiler 编译器

完成编译功能的一个系统程序

Complement 补

算术或逻辑运算。逻辑补等于取反运算、或非运算。

Computer language 计算机语言
见"程序设计语言"。

Conditional compilation 条件编译
通过测试代码中的条件命令、根据条件的真假、有选择地编译程序中一部分代码的能力

continue
一种流控制语句，使下一次循环开始执行。

Control statements 控制语句
根据条件测试结果，判定将执行哪个语句的一种语句。

Control variable 控制变量
在循环执行期间由系统修改的一种变量。当变量达到一个预定值时，循环结束。

Conversion specification 转换规格说明
C语言中用于printf系列函数的一个串，它规定一个变量的打印格式。

Curly braces 大括号
字符 { 或 } 之一。它们用在C中，标定作为一个整体的元素组。

Debugging 调试
找出并删除程序中错误的过程。

Decision statement 判断语句
测试由程序建立的条件，根据这个判断来改变程序流的一个语句。

Declaration 定义
程序中使用的变量类型和名字的规范说明

default
在switch语句中，如果没有匹配的case的值，就把它看作是一个case标号。

Define statement define语句
见"#define"。

Dereferencing operator 逆引用运算符
表示访问由一个指针变量或是一个地址表达式所指向的值的运算符。又见"*"

Directive 伪指令
给预处理器的一个命令（不同于产生机器代码的一条语句）。

double
C语言的一个关键字，定义一个能存储实数的变量。该数通常需要两倍于float类型的存储空间。

Double linked list 双向链表
一种带有向前和向后指针的链表，又见"链表"。

Double quote(") 双引号

ASCII 字符 34，在 C 中用来标定字符串。

EGA

增强图形适配器，IBM PC 中常见的图形卡。

else

if 语句中的一个子句，用来指明当 if 条件失败时将执行什么语句。

enum

C 的一个关键字，定义一个枚举数据类型。

Enumerated data type 枚举数据类型

含有一组有名值的一种数据类型，C 编译器给其中的每元素都指定一个整数值。

EOF

定义在 stdio.h 中的文件结束字符（End-of-file）。

Escape character escape 字符

一个特殊的字符，它改变了其后所跟字符的意思，在 C 中表示为反斜杠 \。

Exclusive OR 异或

布尔运算，如果两个操作数相同，结果为 0；如果两个操作数不同，结果为 1。

Executable file 可执行文件

含有机器代码的一个文件，它已经被连接了，而且可以在一台计算机上运行。

Exponent 指数

浮点数的一部分，表示整数幂，基数乘上幂次才能表示数本身。

Exponent overflow 指数上溢

由浮点数运算引起的一种情况，指数太大以至于在分配给指数的位域中装不下。

Exponent underflow 指数下溢

由浮点数运算引起的一种情况，此时，得到了一个很大的负值指数，以至于在分配给指数的位域中装不下。

extern

C 关键字，用来说明定义在当前文件之外的变量或是函数。

Fast prototyping 快速原型

自顶向下的程序设计方法，先编写规范中的最小部分，让其可以运行，然后再扩大之

fclose

关闭文件函数。

fflush

强制输出缓冲区函数。

fgetc

读单个字符函数。

fgets

读一单行的输入流库函数。

FILE

定义在stdio.h中的一个宏，它说明一个文件变量。

File　文件

看作一个整体的一组相关记录。

float

C关键字，定义可以存储一个实数的一个变量。

Floating point　浮点数

一种记数系统，表示为一个小数部分和一个指数。这种系统可以处理非常大和非常小的数。

Floating-point exception (core dumped)　浮点数异常

因除0或其它非法算术操作而导致的一个错误，因为它是由（核心崩溃）　整 数和浮点数错误引起的，所以有时很令人费解。

Floating-point hardware　浮点数硬件

不必借助于软件，可以直接完成浮点数运算的电路。在个人计算机中，它就是数学协处理器。像80486这样的较先进的处理器中都有内置的浮点运算单元。

fopen

打开文件用于I/O流的函数。

fprintf

把二进制数据转换为字符数据并写入一个文件中的函数。

fputc

写一单个字符的函数。

fputs

写一单行的函数。

fread

二进制I/O输入函数。

free

把数据返回给内存的C函数。

Free Software Foundation　自由软件基金会

一个程序员小组，他们编制并奉献了高质量的免费软件。在他们的产品中，包括有编辑器emacs和C++编译器gcc。他们的地址是：Free Software Foundation，Inc，675 Massa-chusetts Ave., Cambridge, MA 02139。电话是：（617）876-3296。ftp地址是：prep.ai.mit.edu:/pub/gnu。也可在WWW的http://www.gnu.org找到他们。

fscanf
类似于 scanf 的一个输入程序。

function 函数
返回一个值的程序。

fwrite
一个二进制 I/O 输出函数。

Generic pointer 通用指针
一个可以指向任何变量、不用强制转换为变量类型的指针。不用考虑内容而指向存储的一个指针。

ghostscript
类似于 PostScript 的解释器。它可以从自由软件基金会免费得到。

Global variable 全局变量
在整个程序中都知道的变量。

Guard digit 监视位
浮点数计算中使用的一个额外精度数字，它确保不丢失精度。

Header file 头文件
见“包含文件”。

Heap 堆
malloc 使用的内存中的一部分，分配新结构和数组使用的空间。使用 free 函数可把空间返回给这个内存。

Hexadecimal number 十六进制数
基数为 16 的数。

High-level language 高级语言
计算机语言的一个级别，界于机器语言和自然（人类）语言之间。

I/O manipulators I/O 操作符
一组函数，虽然它的“输出”或“输入”没有引起 I/O 操作，但设置了各种转换标志或参数。

IEEE floating-point standard IEEE 浮点标准
IEEE 浮点标准 754，它规定了浮点数格式、精度和一些非分数值的标准。

if
在条件为真值基础上选择执行程序的一部分的语句。

Implementation dependence 实现依赖
是指因为计算机系统的变化性，使得从计算机操作或是软件得到的结果不统一这样一种情形。在其他系统中运行一个特殊操作会产生不同结果。

include file 包含文件

一个文件，执行预处理器命令#include后，该文件可以与源代码合并在一起。又叫头文件。

Inclusive OR 相容析取

见“或”。

Index 下标

一个值、变量或是表达式，选取数组中的一个具体元素。

Indirect operator 间接运算符

见“逆引用运算符”。

Information hiding 信息隐藏

一种代码设计系统，它使得模块之间传递的信息量达到最小。这种思想是尽可能地把信息隐藏在模块内部，只有当绝对需要时，才共享信息。

Instruction 指令

定义由计算机完成的一个操作的一组位或是字符。

int

C中定义一个整数的关键字。

Integer 整型

一个整数。

Interactive debugger 交互调试器

用来调试程序的一个程序。

Inver: operator 反运算符

执行非操作的逻辑运算符。

Left shift 左移

移位操作，在位域内向左移动指定的位数，空出的位填0。

Library 库

文件集合，通常包含和程序相关的目标代码，也叫档案库。

Linked list 链表

数据节点集合，每个节点含有一个值和一个指向链表中下一项的指针。

Local include files 局部包含文件

专用库中的文件。使用预处理器命令#include“文件名”，可以把它插入到源代码中。

Local variable 局部变量

其作用域局限于定义它的块中的变量。

Logical AND 逻辑与

布尔运算，如果两个自变量都为真，则结果为真。

Logical operator 逻辑运算符

C运算符，它对两个操作数进行逻辑运算，返回真值或假值。

Logical OR 逻辑或

布尔运算。如果两个自变量中有一个为真，则结果为真。

long

说明长于正常精度的一种数据类型的修饰符。

Machine code 机器代码

二进制格式的机器指令，它不需要转换就可由机器直接识别。

Machine language 机器语言

见“机器代码”。

Macro 宏

一小段文本或文本模板，它可以被扩充到更长的文本中。

Macro processor 宏处理器

产生代码的一个程序，它把值放到已定义模板中的相应位置。

Magnitude of the number （数）量

不考虑符号时一个数的值。

Maintenance (of a program) （程序）维护

因为计算机系统外部条件的改变而进行的程序修改。

make

UNIX 和 MS-DOS/Windows 系统下的一个实用程序，它管理程序的编译过程。

Makefile

含有用于 make 实用程序命令的文件。

malloc

管理内存堆的 C 程序。

Mask 屏蔽

位的一种模式，用来控制另一组位的留与去。

member 成员

结构的一个元素。也叫字段。

Module 模块

程序的一个逻辑部分。

MS-DOS

微软公司开发的用于 IBM 及其兼容的个人计算机的一个操作系统。

Newline character 换行符

一个字符，它使得输出设备定位到下一行的开始位置。('\n')

Nonsignificant digits 无意义数字

前导数字，对一个数的值不起作用（在补码格式下，0用于正数，1用于负数）。

Normalization 标准化

移动浮点数的小数部分（并调整幂次），这样，在小数部分中就没有前导0了。

NOT 非

布尔运算，它得到操作数的逻辑反。NOT 1 得 0，NOT 0 得 1。

Not a number

IEEE754 中定义的一个特殊值，它表示浮点数运算得到了一个无效结果。

NULL

一个常量，其值为 0，表示指向空。

Null pointer 空指针

其位类型全为 0 的一个指针，它表示该指针不指向任何有效数据。

Object-Oriented Design 面向对象的设计

一种设计方法，程序员把其设计建立在数据对象（类）和它们之间的联系上。

Octal number 八进制数

基数为 8 的数。

Ones complement 补

把一个整数中的所有位反转的操作。1 变为 0，0 变为 1。

Operator 运算符

表示要执行的一个动作的符号。

OR 或

布尔运算：如果两个操作数中有一个为 1，则结果为 1；如果两个操作数均为 0，则结果为 0。

Overflow error 上溢错误

因算术运算的结果大于计算机为结果而提供的存储空间而导致的一种算术运算错误。

Packed structure 压缩结构

按需分配位域的大小而不考虑字边界的一种数据结构技术。

Pad byte 填充字节

加到一个结构中的一个字节，它的唯一目的是确保内存对齐。

Parameter 参数

一个数据项，可以给它赋一个值；常常是指调用函数和被调用函数之间传递的自变量。

Parameterized macro 参数宏

含有一个模板的宏，模板中带有参数的插入点。

Parameters of a macro **宏的参数**

要插入到宏中定义的参数位置的值。在扩展宏时才实施插入操作。

Permanent variables **永久变量**

在程序启动之前建立和初始化的一个变量，在整个程序的运行期间始终保留它的内存。

Pixel **像素**

最小的显示元素，它可以单独地被赋以亮度和颜色。本词来自于Picture Element（图元）。

Pointer **指针**

一种可以存储内存地址的数据类型。

Pointer arithmetic **指针算术运算**

C允许对指针进行三种算术运算：1)指针加上一个数，2)指针减去一个数，3)一个指针可以减去另一个指针。

Portable C compiler **可移植的C编译器**

Stephen Johnson编写的一个C编译器，将它改为不同计算机体系结构下的编译器相对要容易一些。

Precision **精度**

衡量差不多相等的两个值之间差别的一种机制

Preprocessor **预处理器**

执行预处理的一个程序，它的目的是把宏代码模板扩展为C代码。

Preprocessor directive **预处理器指令**

预处理器的一个命令。

printf

产生格式输出的一个C库程序

Procedure **过程**

可从不同程序或程序的不同部分调用的一个程序段。它不返回值（void类型函数

Program **程序**

让计算机执行一系列操作的一组指令。

Program header **程序头**

程序开头的注释块。

Programmer **程序员**

为计算机写程序的人。

Programming (a computer) **(计算机)程序设计**

用代表计算机指令的语言来描述一个问题的解决方案的过程。

Programming language 程序设计语言
用于表示计算机程序的一种形式记号体系。

Program specification 程序规范说明
描述程序功用的文档。

Pseudo code 伪代码
一种编码技术。过程的明确描述由易读的语言指令来书写，而不必考虑计算机语言的语法规则。

Qualifier 修饰符
修改数据说明含义的一个字

Radix 基数
正整数。数位的权乘上基数得到下一个更高数位的权。

Real number 实数
在固定基数计数系统中，由有穷或无穷数表示的一个数。

Recursion 递归
当一个函数直接或间接地调用它本身时，发生递归。

Redirect 重定向
命令行选项“>file”允许用户把程序的结果输出到一个文件中，而不是输出到屏幕上。类似的选项 <file 将从文件中取输入信息，而不是从键盘上。

Reduction in strength 约简
用廉价（快速）操作替代昂贵（缓慢）操作的过程。

Relational operator 关系运算符
比较两个操作数，根据其关系的真或假得出真假值的运算符。

Release 交付
编程项目完成，即此时程序备好可以使用了。

Replay file 重放文件
用来替代键盘数据标准输入的文件。

Return statement return 语句
标志函数结束，并使控制返回调用者的语句。

Revision 修正
对程序的明显改动。

Right shift 右移
移位操作，在位域内向右移动指定的位数

Round 四舍五入
在位置表示中，删去或忽略一个或多个最小的有效数字，然后用某些特定的规则来调整剩余的部分，以使误差最小。

Rounding error 截断误差

四舍五入的截断时产生的误差。

Save file 保存文件

一个调试工具、用户键入的所有内容都保存到一个文件中以备用，又见重放文件。

scanf

直接从键盘读数据的库输入函数，很难使用．在大多数情况下、使用 fgets/sscanf 两个函数。

Scope 作用域

变量的作用域是程序中知道变量名的那一部分。

Segmentation violation 段违例

程序试图访问其地址空间之外的内容而引起的一个错误、因逆引用一个错误指针而引起。

Set a bit 设置一个位

把一个指定位设置为 1 的操作、这不是 C 中定义的操作。

Shift 移位

在位域内向左或向右移动位的操作。

short

与整数相同或更小的一种算术数据类型．

Side effect 副作用

一个语句中主操作之外执行的一个操作．如赋值语句中的变量增量操作：result=begin++-end;

Significand 有效数

浮点数中最有效数字，不考虑其基点位置。

Significant digits 有效数字

保持给定精度的一个数字。

Single quote(') 单引号

ASCII 字符 39．在 C 中用在单字符两侧。

sizeof

返回一种数据类型变量大小（字节数）的运算符。

Source code 源代码

在被计算机翻译之前的原码形式。

Source file 源文件

包含源代码的文件。

Specification 规范说明

描述程序功用的文档。

sprintf

类似于 fprintf，但它使用串输出。

sscanf

库输入程序。

Stack 栈

内存的一个区域，用来临时保存一串数据和指令。

Stack variable 栈变量

见“临时变量”。

Stack overflow 栈上溢

因程序中的变量使用了太多的临时空间（栈空间）而引起的一个错误，由大程序或无限递归引起。

static

一种存储类属性。在一对卷括号{}内时，它表示一个永久变量。在一对卷括号{}外时，它表示一个局部文件变量。

stderr

预定义的标准错误文件。

stdin

预定义的输入源。

stdio.h

原来C风格的I/O包 stdio

stdout

预定义的标准输出。

Storage class 存储类

一种变量定义属性，它控制变量在内存中是如何存储的。

String 串

字符序列或是字符数组。

struct

C关键字，表示结构数据类型。

Structure 结构

一组名字，表示可能有不同属性的一组数据项

Style sheet 风格单

描述编程风格的文档，它由具体的公司或团体来使用。

Sunview

用在SUN工作站上的图形和窗口系统。

switch
多路分支、根据索引表达式的值，把控制转到几个case语句之一。

Syntax 语法
构造语句所遵从的规则。

Syntax error 语法错误
构造C表达式时的错误。

Temporary variable 临时变量
从栈中分配存储的变量，每次进入定义它的块时，初始化临时变量。它只存在于那个块的执行期间。

Test a bit 测试一个位
判定某具体的位是否被设置的操作，它不是C中定义的操作。

Test plan 测试计划
测试的详细说明，程序必须按照它的说明来运行。

Text editor 文本编辑器
用来建立或修改文本文件的软件。

Translation 翻译
用一种替代语言来建立新程序，保证在逻辑上等价于源语言写的程序。

Tree 树
一种层次数据结构。

Truncation 截断
丢弃实数中小数部分的一种操作。

Turbo C++
由Borland公司开发的用于个人计算机的一个C语言版本。

Typecast 类型转换
见“转换”。

typedef
从已有类型建立新类型的运算符。

typing statement 类型语句
建立变量特征的语句。

Unbuffered I/O 非缓冲I/O
每一次的读写都导致一次系统调用的I/O操作。

union
允许对同一存储位置指定不同数据名和数据类型的数据类型。

UNIX

通用多用户操作系统，最初由贝尔实验室的Ken Thompson和Dennis Ritchie开发成功。

unsigned

指定int和char变量不包含负号的修饰符。

Upgrading 升级

程序的修改，以改进其性能或提供新特性。

value 值

赋给一个常量的内容。

Variable 变量

给一个值取的名字。由变量名表示的数据在程序运行的不同时刻，可以取不同的值。

Variable name 变量名

给用来存储一个变量的一段内存取的符号名。

Version 版本

用来标识软件特定版的一个术语，通常包括一个版本号。整数部分表示主要的修改，分数部分表示较小的修改或是问题的修正。

void

C中的一种数据类型，当用作函数调用中的一个参数时，它表示没有返回值，void *表示返回一个通用指针值。

volatile

C关键字，表示一个变量或常量的值可随时改变。该属性用在内存映射I/O、共享内存应用程序和其它高级程序设计中。

while

重复语句。当给定的条件为真时，它就重复地执行一条语句。

Windows

也被称为MS-Windows。由微软公司为个人计算机开发的一种操作系统。也是一个使用相似用户界面及如Windows NT和Windows 95之类API的新操作系统。

X Window system X Window系统

一个图形和窗口系统，可从X Consortium公司得到，目前它运行在许多计算机系统上。

Zero-based counting 基于0的计数

一种计数系统，第一个对象计为0而不是计为1。

［Ｇｅｎｅｒａｌ　Ｉｎｆｏｒｍａｔｉｏｎ］
书名＝实用Ｃ语言　第三版
作者＝ＢＥＸＰ
页数＝４８０
下载位置＝ｈｔｔｐ：／／ｂｏｏｋ８．ｓｓｒｅａｄｅｒ．ｃｏｍ／ｄｉｓｋｊｓｊ／ｊｓ９５／１２／！００００１．ｐｄｇ