PROGRAMMER TO PROGRAMMER™



Beginning C: From Novice to Professional

Fourth Editon

C语言入门经典

(第4版)

(美) Ivor Horton 著

杨 浩 译

清华大学出版社

C 语言入门经典
(第 4 版)

(美) Ivor Horton 著

杨 浩 译

清华大学出版社

北 京

www.TopSage.com

EISBN：1-59059-735-4
Beginning C：From Novice to Professional，Fourth Edition
Ivor Horton
Original English language edition published by Apress L. P., 2560 Ninth Street, Suite 219, Berkeley, CA 94710 USA. Copyright ©2006 by Apress L.P. Simplified Chinese-Language edition copyright © 2008 by Tsinghua University Press. All rights reserved.

本书中文简体字版由 Apress 出版公司授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号　图字：01-2006-4779

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，翻印必究。举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

C 语言入门经典(第 4 版)/ (美) 霍顿(Horton，I.) 著；杨浩 译. —北京：清华大学出版社，2008.4
书名原文：Beginning C：From Novice to Professional，Fourth Edition
ISBN 978-7-302-17083-9
Ⅰ.C… Ⅱ.①霍…②杨… Ⅲ. ①C 语言—程序设计 Ⅳ.TP312
中国版本图书馆 CIP 数据核字(2008)第 021396 号

责任编辑：王　军　郑雪梅
装帧设计：孔祥丰
责任校对：胡雁翎
责任印制：孟凡玉
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：北京鑫海金澳胶印有限公司
经　　销：全国新华书店
开　　本：185×260　**印　张**：36.5　**字　数**：843 千字
版　　次：2008 年 4 月第 1 版　　**印　次**：2008 年 4 月第1 次印刷
印　　数：1～5000
定　　价：69.80 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770177 转 3103　　产品编号：023413－01

www.TopSage.com

前　言

欢迎使用《C 语言入门经典(第 4 版)》。研读本书，你就可以成为一位称职的 C 语言程序员。从许多方面来说，C 语言都是学习程序设计的理想起步语言。C 语言很简洁，因此无须学习大量的语法，就能够开始编写真正的应用程序。除了简明易学外，它还是一种功能非常强大的语言，至今仍被专业人士广泛使用。C 语言的强大之处主要体现在，它能够进行各种层次的程序设计，从硬件设备驱动程序和操作系统组件到大规模的应用程序，都能胜任。事实上，任何计算机都支持 C 语言编译器，因此，当我们学会了 C 语言，就可以在任何环境下进行程序设计。最后一点，掌握了 C 语言，就为理解面向对象的 C++语言奠定了良好的基础。

积极热情的程序员都必将面对三大障碍，即掌握适用于所有程序设计语言的术语，理解如何使用一种语言的元素(而不仅仅只知道它们的概念)以及领会如何在实际环境中应用这种语言，本书的目的就是将这些障碍降到最低。

术语是专业人士与优秀的业余人士们进行交流时必不可少的，因此掌握它们是必需的。本书会让你理解这些术语，并自如地在各种环境下使用它们。这样才能更有效地使用大多数软件产品附带的文档，且能轻松地阅读和学习大多数程序设计语言的相关文献。

显然，理解语言元素的语法和作用是学习一门语言的关键，不过认识语言的特性如何发挥作用和如何应用它们，也同等重要。在说明每种语言特性与特定问题的关系时，本书采用实际应用的程序示例，而不只是代码片断。这些示例提供了实践的基础，你可以任意改动它们，研究改动后的效果。

要理解在特定背景中的程序设计方法，需要理解应用独立语言元素的机理。为了帮助理解它们，本书每章最后都给出一个较复杂的程序，该程序应用了本章前面已经学习的知识。这些程序可帮助你获得开发程序的能力和信心，了解如何综合运用各种语言元素。最重要的是，它们能让你了解设计真实程序时会遇到的问题以及如何管理实际的代码。

学习任何程序设计语言，都要认识几件事情。首先，要学的东西很多，但是掌握了它们之后，你会有极大的成就感。其次，学习的过程很有趣，你将体会到这一点。第三，你只有通过动手实践才能学会程序设计。最后，学习程序设计语言比你想象的容易得多，所以你肯定能掌握它。

如何使用本书

作者认为动手实践是最好的方法，你应当立刻开始编写自己的第一个程序。每一章都有几个把理论应用于实践的程序，这些示例是学习本书的关键。建议读者输入并运行文中的示例，因为输入程序对记住语言元素有极大的帮助。此外，你还应该做每章后面

www.TopSage.com

的练习。当你第一次使一个程序运行起来，尤其是在试图解决自己的问题时，快速的进展会使你有很大的成就感。

刚开始，学习的进展不会太快，不过随着逐渐深入，我们会加快学习的速度。每一章都会涉及很多基础知识，因此在学习新的内容之前，需要花些时间，确保理解了前面学过的所有知识。实践各部分的代码，并尝试实现自己的想法，这是学习程序设计语言的一个重要部分。尝试修改书中的程序，看看还能让它们做什么，这是很有趣的。不要害怕尝试，如果不明白某一点如何使用，输入几种变体，看看会出现哪些情况。好的学习方法是先通读整章，全面了解其中介绍的内容，然后再实践其中的所有程序示例。

你可能会觉得某些章末尾的程序非常难。如果第一次读这样的程序没有完全理解，不必担心。第一次难免会觉得难以理解，因为它们通常都是把你所学的知识应用到了相当复杂的问题中。如果你真的不能理解，可以略过那些章末尾的程序，继续学习下一章，然后再回头研究这些程序。甚至可以在学完全书之后再来研究它们。之所以演示这些程序是因为即使读完了本书，它们对你来说仍是非常有用的资源。

本书读者对象

本书的目的是教你如何尽可能简单快速地编写有用的程序，如果你属于下列情况之一，那么本书就非常适合你：

- 刚接触程序设计，但想直接深入了解 C 语言，从头开始学习程序设计及编写 C 语言程序。
- 以前有一点程序设计经历，对其基本概念有一定了解，也许曾经使用过 BASIC 或 PASCAL。现在想学习 C 语言，进一步提高自己的程序设计技能。

本书并未假设此前你对程序设计的知识有所了解，不过本书会很快地从基本概念转入到实际应用。学完了本书，你就为自己的 C 语言程序设计奠定了全面的基础。

使用本书的条件

要使用本书，需要一台安装了 C 语言编译器和库的计算机，这样才能执行书中的示例，还需要一个程序文本编辑器，用于创建源代码文件。你使用的编译器要很好地支持 C 语言国际标准：ISO/IEC 9899。你还需要一个用于创建和修改代码的编辑器，可以采用任何纯文本编辑器创建源程序文件，如 Notepad 或 vi。不过，采用专为编辑 C 语言代码设计的编辑器更有帮助。

要最大限度地发挥本书的功效，你需要有学习的意愿、成功的渴望，当学习不顺利，觉得前途渺茫时，还要有坚持下去的决心。几乎每个人在初次学习程序设计时都会在某处觉得迷茫。当你发现自己艰难地掌握了 C 语言的某个方面时，要坚持下去，迷雾一定会消散，你会觉得为什么当初我不明白这一点呢？也许你明白要做到这些将会很难，不过相信你一定会惊讶自己能在较短的时间内取得很大进步。本书会帮助你开始自己的实践之旅，使你成为成功的程序设计员。

www.TopSage.com

本书采用的约定

本书的文本和布局采用了许多不同的样式，以便区分各种不同的信息。大多数样式表达的含义都很明显，其中程序代码以类似下面的样子出现：

```
int main(void)
{
printf("\nBeginning C");
return 0;
}
```

如果代码片段是从前面的实例修改而来的，修改过的代码行就用粗体显示，如下所示：

```
int main(void)
{
printf("\nBeginning C by Ivor Horton");
return 0;
}
```

程序代码中还使用了各种“括号”。它们之间的差别非常重要，不能互换。本书中称()为圆括号，{}为大括号，[]为方括号。

本书源代码下载

从 Apress 的站点可以下载本书中的所有代码和练习的解决方案：http://www.apress.com。也可以访问 www.tupwk.com.cn/downpage 下载本书中的所有代码和解决方案。

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

目　录

第 1 章　C 语言编程 ……1
1.1　创建 C 程序 ……1
1.1.1　编辑 ……1
1.1.2　编译 ……2
1.1.3　链接 ……2
1.1.4　执行 ……3
1.2　创建第一个程序 ……4
1.3　编辑第一个程序 ……4
1.4　处理错误 ……5
1.5　剖析一个简单的程序 ……6
1.5.1　注释 ……6
1.5.2　预处理指令 ……7
1.5.3　定义 main()函数 ……7
1.5.4　关键字 ……8
1.5.5　函数体 ……8
1.5.6　输出信息 ……9
1.5.7　参数 ……10
1.5.8　控制符 ……10
1.6　用 C 语言开发程序 ……12
1.6.1　了解问题 ……12
1.6.2　详细设计 ……12
1.6.3　实施 ……13
1.6.4　测试 ……13
1.7　函数及模块化编程 ……13
1.8　常见错误 ……17
1.9　要点 ……17
1.10　小结 ……18
1.11　习题 ……18

第 2 章　编程初步 ……19
2.1　计算机的内存 ……19
2.2　什么是变量 ……21
2.3　存储数值的变量 ……21
2.3.1　整数变量 ……21
2.3.2　变量的命名 ……25
2.3.3　变量的使用 ……26
2.3.4　变量的初始化 ……28
2.3.5　算术语句 ……28
2.4　变量与内存 ……34
2.5　整数变量类型 ……35
2.5.1　无符号的整数类型 ……35
2.5.2　使用整数类型 ……36
2.5.3　指定整数常量 ……37
2.6　浮点数 ……38
2.7　浮点数变量 ……38
2.8　使用浮点数完成除法运算 ……39
2.8.1　控制小数位数 ……40
2.8.2　控制输出的字段宽度 ……41
2.9　较复杂的表达式 ……41
2.10　定义常量 ……44
2.10.1　极限值 ……46
2.10.2　sizeof 运算符 ……49
2.11　选择正确的类型 ……50
2.12　强制类型转换 ……53
2.12.1　自动转换类型 ……53
2.12.2　隐式类型转换的规则 ……54
2.12.3　赋值语句中的隐式类型转换 ……54
2.13　再谈数值数据类型 ……55
2.13.1　字符类型 ……56
2.13.2　字符的输入输出 ……57
2.13.3　宽字符类型 ……60
2.13.4　枚举 ……60
2.13.5　存储布尔值的变量 ……63
2.13.6　复数类型 ……63
2.14　赋值操作的 op=形式 ……66

www.TopSage.com

2.15 数学函数 ····· 68
2.16 设计一个程序 ····· 69
2.16.1 问题 ····· 69
2.16.2 分析 ····· 69
2.16.3 解决方案 ····· 71
2.17 小结 ····· 75
2.18 练习 ····· 76

第 3 章 条件判断 ····· 79
3.1 判断过程 ····· 79
3.1.1 算术比较 ····· 80
3.1.2 涉及关系运算符的表达式 ··· 80
3.1.3 基本的 if 语句 ····· 81
3.1.4 扩展 if 语句：if-else ····· 84
3.1.5 在 if 语句中使用代码块 ····· 86
3.1.6 嵌套的 if 语句 ····· 87
3.1.7 更多的关系运算符 ····· 90
3.1.8 逻辑运算符 ····· 93
3.1.9 条件运算符 ····· 97
3.1.10 运算符的优先级 ····· 99
3.2 多项选择问题 ····· 103
3.2.1 给多项选择使用 else-if 语句 ····· 104
3.2.2 switch 语句 ····· 104
3.2.3 goto 语句 ····· 113
3.3 按位运算符 ····· 114
3.3.1 按位运算符的 op=用法 ····· 116
3.3.2 使用按位运算符 ····· 117
3.4 设计程序 ····· 120
3.4.1 问题 ····· 120
3.4.2 分析 ····· 120
3.4.3 解决方案 ····· 121
3.5 小结 ····· 124
3.6 练习 ····· 124

第 4 章 循环 ····· 127
4.1 循环 ····· 127
4.2 递增和递减运算符 ····· 128
4.3 for 循环 ····· 129
4.4 for 循环的一般语法 ····· 132
4.5 再谈递增和递减运算符 ····· 133
4.5.1 递增运算符 ····· 133
4.5.2 递增运算符的前置和后置形式 ····· 134
4.5.3 递减运算符 ····· 134
4.6 再论 for 循环 ····· 135
4.6.1 修改 for 循环变量 ····· 137
4.6.2 没有参数的 for 循环 ····· 138
4.6.3 循环内的 break 语句 ····· 138
4.6.4 使用 for 循环限制输入 ····· 141
4.6.5 生成伪随机整数 ····· 143
4.6.6 再谈循环控制选项 ····· 145
4.6.7 浮点类型的循环控制变量 ··· 146
4.7 while 循环 ····· 147
4.8 嵌套循环 ····· 150
4.9 嵌套循环和 goto 语句 ····· 153
4.10 do-while 循环 ····· 154
4.11 continue 语句 ····· 157
4.12 设计程序 ····· 157
4.12.1 问题 ····· 157
4.12.2 分析 ····· 157
4.12.3 解决方案 ····· 158
4.13 小结 ····· 170
4.14 习题 ····· 170

第 5 章 数组 ····· 173
5.1 数组简介 ····· 173
5.1.1 不用数组的程序 ····· 173
5.1.2 什么是数组 ····· 175
5.1.3 使用数组 ····· 176
5.2 内存 ····· 179
5.3 数组和地址 ····· 182
5.4 数组的初始化 ····· 184
5.5 确定数组的大小 ····· 184
5.6 多维数组 ····· 185
5.7 多维数组的初始化 ····· 187
5.8 设计一个程序 ····· 191

www.TopSage.com

5.8.1 问题 …… 192
5.8.2 分析 …… 192
5.8.3 解决方案 …… 193
5.9 小结 …… 200
5.10 习题 …… 200

第 6 章 字符串和文本的应用 …… 201
6.1 什么是字符串 …… 201
6.2 处理字符串和文本的方法 …… 203
6.3 字符串操作 …… 206
6.3.1 连接字符串 …… 206
6.3.2 字符串数组 …… 208
6.4 字符串库函数 …… 210
6.4.1 使用库函数复制字符串 …… 210
6.4.2 使用库函数确定字符串的长度 …… 211
6.4.3 使用库函数连接字符串 …… 212
6.4.4 比较字符串 …… 213
6.4.5 搜索字符串 …… 216
6.5 分析和转换字符串 …… 219
6.5.1 转换字符 …… 222
6.5.2 将字符串转换成数值 …… 225
6.7 使用宽字符串 …… 225
6.8 设计一个程序 …… 228
6.8.1 问题 …… 229
6.8.2 分析 …… 229
6.8.3 解决方案 …… 229
6.9 小结 …… 237
6.10 习题 …… 237

第 7 章 指针 …… 239
7.1 指针初探 …… 239
7.1.1 声明指针 …… 240
7.1.2 通过指针访问值 …… 241
7.1.3 使用指针 …… 244
7.1.4 指向常量的指针 …… 248
7.1.5 常量指针 …… 248
7.1.6 指针的命名 …… 249
7.2 数组和指针 …… 249
7.3 多维数组 …… 252
7.3.1 多维数组和指针 …… 255
7.3.2 访问数组元素 …… 257
7.4 内存的使用 …… 260
7.4.1 动态内存分配：malloc()函数 …… 260
7.4.2 分配内存时使用 sizeof 运算符 …… 261
7.4.3 用 calloc()函数分配内存 …… 265
7.4.4 释放动态分配的内存 …… 265
7.4.5 重新分配内存 …… 267
7.5 使用指针处理字符串 …… 268
7.5.1 更多地控制字符串输入 …… 268
7.5.2 使用指针数组 …… 269
7.6 设计程序 …… 280
7.6.1 问题 …… 280
7.6.2 分析 …… 281
7.6.3 解决方案 …… 281
7.7 小结 …… 291
7.8 习题 …… 291

第 8 章 程序的结构 …… 293
8.1 程序的结构 …… 293
8.1.1 变量的作用域和生存期 …… 294
8.1.2 变量的作用域和函数 …… 297
8.2 函数 …… 297
8.2.1 定义函数 …… 298
8.2.2 return 语句 …… 301
8.3 按值传递机制 …… 304
8.4 函数声明 …… 305
8.5 指针用作参数和返回值 …… 307
8.5.1 常量参数 …… 310
8.5.2 从函数中返回指针值 …… 318
8.5.3 在函数中递增指针 …… 322
8.6 小结 …… 322
8.7 习题 …… 323

www.TopSage.com

第 9 章 函数再探……325
9.1 函数指针……325
9.1.1 声明函数指针……325
9.1.2 通过函数指针调用函数…326
9.1.3 函数指针数组……329
9.1.4 作为变元的函数指针……331
9.2 函数中的变量……334
9.2.1 静态变量：函数内部的追踪……334
9.2.2 在函数之间共享变量……336
9.3 调用自己的函数：递归……338
9.4 变元个数可变的函数……341
9.4.1 复制 va_list……344
9.4.2 长度可变的变元列表的基本规则……344
9.5 main()函数……345
9.6 结束程序……346
9.7 函数库：头文件……347
9.8 提高性能……348
9.8.1 内联声明函数……348
9.8.2 使用 restrict 关键字……348
9.9 设计程序……349
9.9.1 问题……349
9.9.2 分析……349
9.9.3 解决方案……351
9.10 小结……367
9.11 习题……368

第 10 章 基本输入和输出操作……369
10.1 输入和输出流……369
10.2 标准流……370
10.3 键盘输入……371
10.3.1 格式化键盘输入……371
10.3.2 输入格式控制字符串…372
10.3.3 输入格式字符串中的字符……377
10.3.4 输入浮点数的各种变化…378
10.3.5 读取十六进制和八进制值……379
10.3.6 用 scanf()读取字符……381
10.3.7 scanf()的陷阱……383
10.3.8 从键盘上输入字符串……383
10.3.9 键盘的非格式化输入……384
10.4 屏幕输出……389
10.4.1 使用 printf()格式输出到屏幕……389
10.4.2 转义序列……391
10.4.3 整数输出……392
10.4.4 输出浮点数……394
10.4.5 字符输出……395
10.5 其他输出函数……398
10.5.1 屏幕的非格式化输出…398
10.5.2 数组的格式化输出……399
10.5.3 数组的格式化输入……400
10.6 打印机输出……400
10.7 小结……401
10.8 习题……401

第 11 章 结构化数据……403
11.1 数据结构：使用 struct……403
11.1.1 定义结构类型和结构变量……405
11.1.2 访问结构成员……405
11.1.3 未命名的结构……408
11.1.4 结构数组……408
11.1.5 表达式中的结构……411
11.1.6 结构指针……411
11.1.7 为结构动态分配内存…412
11.2 再探结构成员……414
11.2.1 将一个结构作为另一个结构的成员……414
11.2.2 声明结构中的结构……415
11.2.3 将结构指针用作结构成员……416
11.2.4 双向链表……420

www.TopSage.com

11.2.5 结构中的位字段……423
11.3 结构与函数……424
11.3.1 结构作为函数的变元……424
11.3.2 结构指针作为函数变元…425
11.3.3 作为函数返回值的结构…426
11.3.4 修改程序……430
11.3.5 二叉树……433
11.4 共享内存……442
11.4.1 联合……442
11.4.2 联合指针……444
11.4.3 联合的初始化……444
11.4.4 联合中的结构成员……444
11.5 定义自己的数据类型……446
11.5.1 结构与类型定义(typedef)功能……446
11.5.2 使用 typedef 简化代码…447
11.6 设计程序……448
11.6.1 问题……448
11.6.2 分析……448
11.6.3 解决方案……448
11.7 小结……459
11.8 习题……459

第 12 章 处理文件……461
12.1 文件的概念……461
12.1.1 文件中的位置……462
12.1.2 文件流……462
12.2 文件访问……462
12.2.1 打开文件……463
12.2.2 文件重命名……465
12.2.3 关闭文件……465
12.2.4 删除文件……466
12.3 写入文本文件……466
12.4 读取文本文件……467
12.5 将字符串写入文本文件……470
12.6 从文本文件中读入字符串…471
12.7 格式化文件的输入输出……474
12.7.1 格式化文件输出……474
12.7.2 格式化文件输入……475
12.8 错误处理……477
12.9 再探文本文件操作模式……478
12.10 二进制文件的输入输出……479
12.10.1 指定二进制模式……479
12.10.2 写入二进制文件……480
12.10.3 读取二进制文件……480
12.11 在文件中移动……488
12.11.1 文件定位操作……489
12.11.2 找出我们在文件中的位置……489
12.11.3 在文件中设定位置…490
12.12 使用临时文件……496
12.12.1 创建临时文件……496
12.12.2 创建唯一的文件名…496
12.13 更新二进制文件……497
12.13.1 修改文件的内容……502
12.13.2 从键盘读取记录……503
12.13.3 将记录写入文件……504
12.13.4 从文件中读取记录…505
12.13.5 写入文件……506
12.13.6 列出文件内容……507
12.13.7 更新已有的文件内容…508
12.14 文件打开模式小结……515
12.15 设计程序……516
12.15.1 问题……516
12.15.2 分析……516
12.15.3 解决方案……516
12.16 小结……522
12.17 习题……522

第 13 章 支持功能……523
13.1 预处理……523
13.1.1 在程序中包含头文件…523
13.1.2 外部变量及函数……524
13.1.3 替换程序源代码……525
13.1.4 宏替换……526
13.1.5 看起来像函数的宏……526

www.TopSage.com

13.1.6 多行上的预处理指令…… 528
13.1.7 字符串作为宏参数…… 528
13.1.8 结合两个宏展开式的结果 …… 529
13.2 预处理器逻辑指令…… 530
13.2.1 条件编译…… 530
13.2.2 测试指定值的指令…… 531
13.2.3 多项选择…… 531
13.2.4 标准预处理宏…… 532
13.3 调试方法 …… 533
13.3.1 集成的调试器…… 533
13.3.2 调试阶段的预处理器…… 533
13.3.3 使用 assert()宏 …… 537
13.4 其他库函数…… 539
13.4.1 日期和时间函数库 …… 539
13.4.2 获取日期…… 543
13.5 小结…… 549
13.6 习题…… 549
附录 A 计算机中的数学知识 …… 551
附录 B ASCII 字符代码定义 …… 559
附录 C C 语言中的保留字 …… 565
附录 D 输入输出格式指定符…… 567

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

第 1 章

C 语言编程

C 语言是一种功能强大、简洁的计算机语言，通过它可以编写程序，指挥计算机完成指定的任务。我们可以利用 C 语言创建程序(即一组指令)，并让计算机依指令行事。

用 C 语言编程并不难，本书将用浅显易懂的方法介绍 C 语言的基础知识，读完本章，读者就可以编写第一个 C 语言程序了，其实 C 语言很简单。

TopSage.com

本章的主要内容：

- 如何创建 C 程序
- 如何组织 C 程序
- 如何编写在屏幕上显示文本的程序

1.1 创建 C 程序

C 程序的创建过程有 4 个基本步骤或过程：

- 编辑
- 编译
- 链接
- 执行

这些过程很容易完成(就像翻转手臂一样简单，而且可以随时翻转)，首先介绍每个过程，以及它们对创建 C 程序的作用。

1.1.1 编辑

编辑过程就是创建和修改 C 程序的源代码——我们编写的程序指令称为源代码。有些 C 编译器带一个编辑器，可帮助管理程序。通常，编辑器是提供了编写、管理、开发与测试程序的环境，有时也称为集成开发环境(缩写为 IDE)。

也可以用其他编辑器来创建源文件，但它们必须将代码保存为纯文本，而没有嵌入附加的格式化数据。一般来说，如果编译器系统带有编辑器，就会提供很多更便于编写及组织程序的功能。它们通常会自动编排程序文本的格式，并将重要的语言元素以高亮颜色显示，这样不仅让程序容易阅读，还容易找到单词输入错误。

在 UNIX 或 Linux 上，最常用的文本编辑器是 vi，也可以使用 emacs 编辑器。

在 PC 上，可以使用许多免费(freeware)或共享(shareware)的程序设计编辑器。这些

www.TopSage.com

软件提供了许多功能，例如，高亮显示特殊的语法及代码自动缩进等功能，帮助减少代码的错误率。不要使用字处理器(例如微软的 Word)，不适合编写程序代码，因为它们在保存文本时，会附加一些格式化信息。当然，也可以购买支持 C 语言的专业编程开发环境，例如微软或 Borland 的相关产品，它们能大大提高代码编辑能力。不过，在付款之前，最好检查一下它们支持的 C 级别是否符合当前的 C 语言标准。因为现在很多编辑器产品主要面向 C++开发人员，C 语言只是一个次要目标。对于 Windows 平台来说，emacs 编辑器是一种不错的选择，很多专业程序员都使用它。

1.1.2 编译

编译器可以将源代码转换成机器语言，在编译的过程中，会找出并报告错误。这个阶段的输入是在编辑期间产生的文件，常称为源文件。

编译器能找出程序中很多无效或无法识别的错误，以及结构错误，例如程序的某部分永远不会执行。编译器的输出结果称为对象代码(object code)，存放它们的文件称为对象文件(object file)，这些文件的扩展名在 Windows 环境中通常是.obj，在 Linux/UNIX 环境中通常是.o。编译器可以在转换过程中找出几种不同类型的错误，它们大都会阻止对象文件的创建。

如果编译成功，就会生成一个文件，它与源文件同名，但扩展名是.o 或者.obj。

如果在 UNIX 系统下工作，在命令行上编译 C 程序的标准命令是 cc(若编译器是 GNU's Not UNIX(GNU)，则命令为 gcc)。下面是一个示例：

```
cc  -c  myprog.c
```

其中，myporg.c 是要编译的程序，如果省略了 - c 这个参数，程序还会自动链接。成功编译的结果是生成一个对象文件。

大多数 C 编译器都有标准的编译选项，在命令行(如 cc myprog.c)或集成开发环境下的菜单选项(Compile 菜单选项)里都可找到。

1.1.3 链接

链接器(linker)将源代码文件中由编译器产生的各种模块组合起来，再从 C 语言提供的程序库中添加必要的代码模块，将它们组合成一个可执行的文件。链接器也可以检测和报告错误，例如，遗漏了程序的某个部分，或者引用了一个根本不存在的库组件。

实际上，如果程序太大，可将其拆成几个源代码文件，再用链接器连接起来。因为很难一次编写一个很大的程序，也不可能只使用一个文件。如果将它拆成多个小源文件，每个源文件提供程序的一部分功能，程序的开发就容易多了。这些源文件可以分别编译，更容易避免简单输入错误的发生。再者，整个程序可以一点一点地开发，组成程序的源文件通常会用同一个项目名称集成，这个项目名称用于引用整个程序。

程序库提供的例程可以执行非 C 语言的操作，从而支持和扩展了 C 语言。例如，库中包含的例程支持输入、输出、计算平方根、比较两个字符串，或读取日期和时间信息

www.TopSage.com

等操作。

链接阶段出现错误，意味着必须重新编辑源代码；反过来，如果链接成功，就会产生一个可执行文件。在 Windows 环境下，这个可执行文件的扩展名为.exe；在 UNIX 环境下，没有扩展名，但它是一个可执行的文件类型。

多数 IDE 也有 Build(建立)选项，它可一次完成程序的编译和链接。在 IDE 中，这个选项通常在 Compile(编译)菜单项中，一些 IDE 有单独的 Build 菜单项。

1.1.4 执行

执行阶段就是当成功完成了前述 3 个过程后，运行程序。但是，这个阶段可能会出现各种错误，包括输出错误及什么也不做，甚至使计算机崩溃。不管出现哪种情况，都必须返回编辑阶段，检查并修改源代码。

在这个阶段，计算机最终会精确地执行指令。在 UNIX 和 Linux 下，只要键入编译和链接后的文件名，即可执行程序。在大多数 IDE 中，都有一个相应的菜单命令，来运行或者执行已编译的程序。这个 Run 或者 Execute 命令可能有自己的菜单，也可能位于 Compile 菜单项下。在 Windows 环境中，运行程序的.exe 文件即可，这与运行其他可执行程序一样。

在任何环境及任何语言中，开发程序的编辑、编译、链接与执行这 4 个步骤都是一样的。图 1-1 总结了创建 C 程序的各个过程。

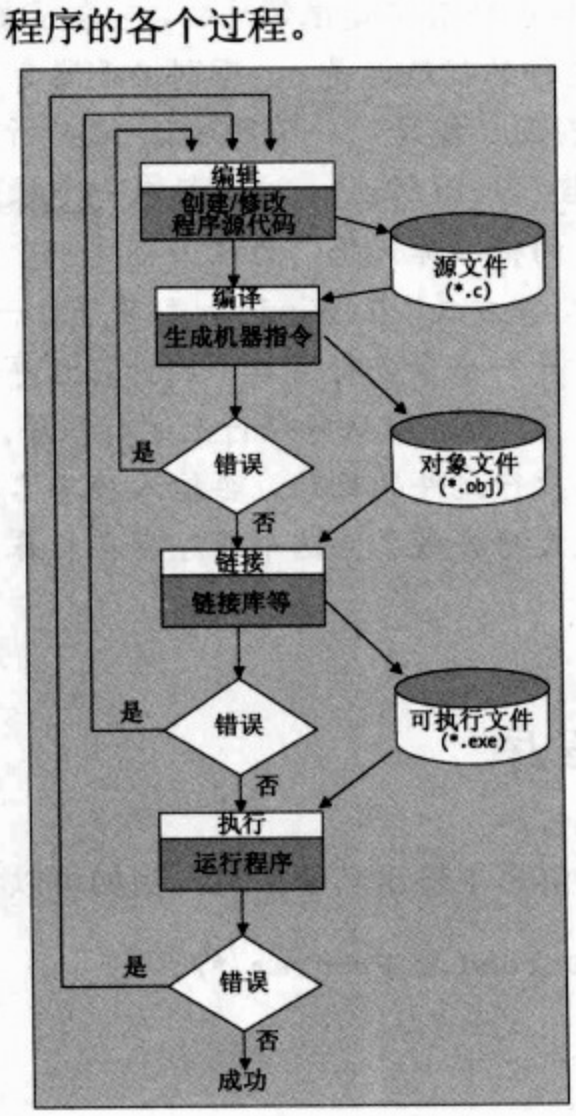


图 1-1 创建和执行程序

www.TopSage.com

1.2　创建第一个程序

本节先浏览一下创建 C 语言程序的流程，从输入代码到执行程序的所有 4 个步骤。在这个阶段，若不了解所键入的代码信息，别担心，笔者会解释每一个步骤。

试试看：C 程序示例

打开编辑器，输入下面的程序，请注意标点符号不要输错，第 4 行及最后一行的括号是大括号{}，而不是方括号[]或者圆括号()——这很重要。另外一定要键入斜杠(/)，以后也会用到反斜杠(\)。最后别忘了行末的分号(;)。

```
/*  Program 1.1 Your Very First C Program - Displaying Hello World */
#include  <stdio.h>

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

在输入了上面的源代码后，将程序保存为 hello.c。可以用任意名字替代 hello，但扩展名必须是.c。这个扩展名在 C 语言中是很常见的，它表示文件的内容是 C 语言源代码。大多数 C 编译器都要求源文件的扩展名是.c，否则编译器会拒绝处理它。

下面编译程序(如本章前面“编译”一节所述)，链接所有必要的内容，创建一个可执行程序(如本章前面“链接”一节所述)。这一般在一个操作中完成。源代码编译成功后，链接器就添加程序需要的标准库代码，为程序创建一个可执行文件。

最后，执行程序。这有几种方式，在 Windows 环境下，一般只需在 Windows Explorer 中双击.exe 文件，但最好打开一个命令行窗口，因为在程序执行完毕后，显示输出的窗口就会消失。在所有的平台上，都可以从命令行上运行程序。只需启动一个命令行会话，把当前目录改为包含程序可执行文件的目录，再输入程序名，就可以执行它了。

如果没有出现错误，就大功告成了。这个程序会在屏幕上输出如下信息：

```
Hello world!
```

1.3　编辑第一个程序

我们可以修改程序，在屏幕上输出其他信息，例如可以将程序改成：

```
/*Program 1.2 Your Second C Program */
#include <stdio.h>


int main(void)
{
```

www.TopSage.com

```
    printf("If at first you don\'t succeed, try, try, try again! ");
    return 0;
}
```

在要显示的文本中，\'序列称为转义序列(escape sequence)。这是在文本中包含单引号的特殊方式，因为单引号通常表示字符常量的开头和结尾。本章后面的“控制字符”一节将详细介绍转义序列。修改完源代码后，可以重新编译，链接后执行。反复练习，熟悉整个流程。

1.4 处理错误

犯错乃人之常情，没什么难为情的。幸好计算机一般不会出错，而且非常擅长于找出我们犯的错误。编译器会列出在源代码中找到的一组错误信息(甚至比我们想象的多)，通常会指出有错误的语句。此时，我们必须返回编辑阶段，找出有错误的代码并更正。

有时一个错误会使后面本来正确的语句也出现错误。这多半是程序的其他部分引用了错误语句定义的内容所造成的。当然，定义语句有错，但被定义的内容不一定有错。

下面看看源代码在程序中生成了一个错误时，会是什么样的情况。编辑第二个程序示例，将 printf()行最后的分号去掉，如下所示：

```
/*Program 1.2 Your Second C Program */
#include <stdio.h)

int main(void)
{
    printf("If at first you don\'t succeed, try, try, try again!")
    return 0;
}
```

编译这个程序后，会看到错误信息，具体信息随编译器的不同而略有区别。下面是一个比较常见的错误信息：

```
Syntax error : missing ';' before '}'
HELLO.C - 1 error(s), 0 warning(s)
```

编译器能精确地指出错误及其出处，在这里，printf()行的结尾处需要一个分号。在开始编写程序时，可能有很多错误是简单的拼写错误造成的。还很容易忘了逗号、括号，或按错了键。没关系，许多有经验的老手也常犯这种错误。

如前所述，有时一点小错误会造成大灾难，编译器会显示许多不同的错误信息。不要被错误的数量吓倒，仔细看过每一个错误信息后，返回并改掉错误部分，不懂的先不管它，然后再编译一次源文件，就会发现错误一次比一次少。

返回编辑器，重新输入分号，再编译，看看有没有其他错误，如果没有错误，程序就可以执行了。

www.TopSage.com

1.5 剖析一个简单的程序

编写并编译了第一个程序后，下面是另一个非常类似的例子，了解各行代码的作用：

```
/* Program 1.3 Another Simple C Program - Displaying a Quotation */
#include <stdio.h>

int main(void)
{
   printf("Beware the Ides Of March!");
   return 0;
}
```

这和第一个程序完全相同，这里把它作为练习，用编辑器输入这个示例，编译并执行。若输入完全正确，会看到如下输出：

```
Beware the Ides Of March!
```

1.5.1 注释

上述示例的第一行代码如下：

```
/* Program 1.3 Another Simple C Program - Displaying a Quotation */
```

这不是程序代码，因为它没有告诉电脑执行操作，它只是一个注释，告诉阅读代码的人，这个程序要做什么。位于/*和*/之间的任意文本都是注释。只要编译器在源文件中找到/*，就忽略它后面的内容，一直到表示注释结束的*/为止。/*可以和*/放在同一行代码上，也可以放在不同的代码行上。

应养成给程序添加注释的习惯，当然程序也可以没有注释，但在编写较长的程序时，可能会忘记这个程序的作用或工作方式。添加足够的注释，可确保日后自己(和其他程序员)能理解程序的作用和工作方式。

注释不一定单独占一行，无论/*和*/放在代码的什么地方，/*和*/之间的所有内容都是注释。下面给程序再添加一些注释：

```
/* Program 1.3 Another Simple C Program - Displaying a Quotation */
#include <stdio.h> /* This is a preprocessor directive */

int main(void)      /* This identifies the function main() */
{                   /* This marks the beginning of main() */
 printf("Beware the Ides of March!"); /* This line displays a quotation */
 return 0;          /* This returns control to the operating system */
}                   /* This marks the end of main() */
```

可以看出，使用注释是一种非常有效的方式，可以解释程序中要发生的事情。注释可以放在程序中的任意位置，说明代码的一般作用，指定代码是如何工作的。一个注释

www.TopSage.com

可以分散在多个代码行上，/*和*/之间的所有内容都是注释，编译器会忽略它们。下面使用一个注释说明代码的作者及版权所有：

```
/*
* Written by Ivor Horton
* Copyright 2006
*/
```

这个注释横跨 4 行。它用星号标记每个文本行的开头，但这不是必须的，而是注释中的内容。可以使用任意字符来提高注释的可读性，但必须将*/用作注释的结尾。

1.5.2 预处理指令

下面的代码行：

```
#include <stdio.h>  /* This is a preprocessor directive */
```

严格说来，它不是可执行程序的一部分，但它很重要，事实上程序没有它是不执行的。符号#表示这是一个预处理指令(preprocessing directive)，告诉编译器在编译源代码之前，要先执行一些操作。编译器在编译过程开始之前的预处理阶段处理这些指令。预处理指令相当多，大多放于程序源文件的开头。

在这个例子中，编译器要将 stdio.h 文件的内容包含进来，这个文件称为头文件(header file)，因为它通常放在程序的开头处。在本例中，头文件定义了 C 标准库中一些函数的信息，但一般情况下，头文件指定的信息应由编译器用于在程序中集成预定义函数或其他全局对象，所以有时需要创建自己的头文件，以用于程序。本例要用到标准库中的 printf()函数，所以须包含 stdio.h 头文件。stdio.h 头文件包含了编译器理解 printf()以及其他输入/输出函数所需要的信息。名称 stdio 是标准输入/输出(standard input/output)的缩写。C 语言中所有头文件的扩展名都是.h，本书的后面会用到其他头文件。

注意：

头文件名是不区分大小写的，但在#include 指令里，这些文件名通常是小写。

每个符合国际语言标准(ISO/IEC 9899)的 C 编译器都有一些标准的头文件。这些头文件主要包含了与 C 标准库函数相关的声明。所有符合该标准的 C 编译器都支持同一组标准库函数，有同一组标准头文件，但一些编译器有额外的库函数，它们提供的功能一般是运行编译器的计算机所专用的。

1.5.3 定义 main()函数

下面的 5 行指令定义了 main()函数：

```
int main(void)  /* This identifies the function main() */
{               /* This marks the beginning of main() */
```

www.TopSage.com

```
printf("Beware the Ides of March!"); /* This line displays a quotation */
return 0;       /* This returns control to the operating system */
}               /* This marks the end of main() */
```

函数是两个括号之间执行某组操作的一段代码。每个 C 程序都由一个或多个函数组成，每个 C 程序都必须有一个 main()函数——因为每个程序总是从这个函数开始执行。因此假定创建、编译、链接了一个名为 progname.exe 的文件。执行它时，操作系统会调用这个程序的 main()函数。

定义 main()函数的第一行代码如下：

```
int main(void)  /* This identifies the function main()*/
```

它定义了 main()函数的起始，注意这行代码的末尾没有分号。定义 main()函数的第一行代码开头是一个关键字 int，它表示 main()函数的返回值的类型，关键字 int 是表示 main()函数返回一个整数值。执行完 main()函数后返回的整数值表示返回给操作系统的一个代码，它表示程序的状态。在下面的语句中，指定了执行完 main()函数后要返回的值：

```
return 0; /* This returns control to the operating system */
```

这个 return 语句结束 main()函数的执行，把值 0 返回给操作系统。从 main()函数返回 0 表示，程序正常终止，而返回非 0 值表示异常，换言之，在程序结束时，发生了不应发生的事情。

紧跟在函数名 main 后的括号，带有函数 main()开始执行时传递给它的信息，在这个例子里，括号内是 void，表示没有给函数 main()传递任何数据，后面会介绍如何将数据传递给函数 main()或程序内的其他函数。

函数 main()可以调用其他函数，这些函数又可以调用其他函数。对于每个被调用的函数，都可以在函数名后面的括号中给函数传递一些信息。在执行到函数体中的 return 语句时，就停止执行该函数，将控制权返回给调用函数(对于函数 main()，则将控制权返回给操作系统)。

1.5.4 关键字

在 C 语言中，关键字是有特殊意义的字，所以在程序中不能将关键字用于其他目的。关键字也称为保留字。在前面的例子里，int 就是一个关键字，void 和 return 也是关键字。C 语言有许多关键字，我们在学习 C 语言的过程中，将逐渐熟悉这些关键字。附录 C 列出了完整的 C 语言关键字表。

1.5.5 函数体

main()函数的一般结构如图 1-2 所示：

www.TopSage.com

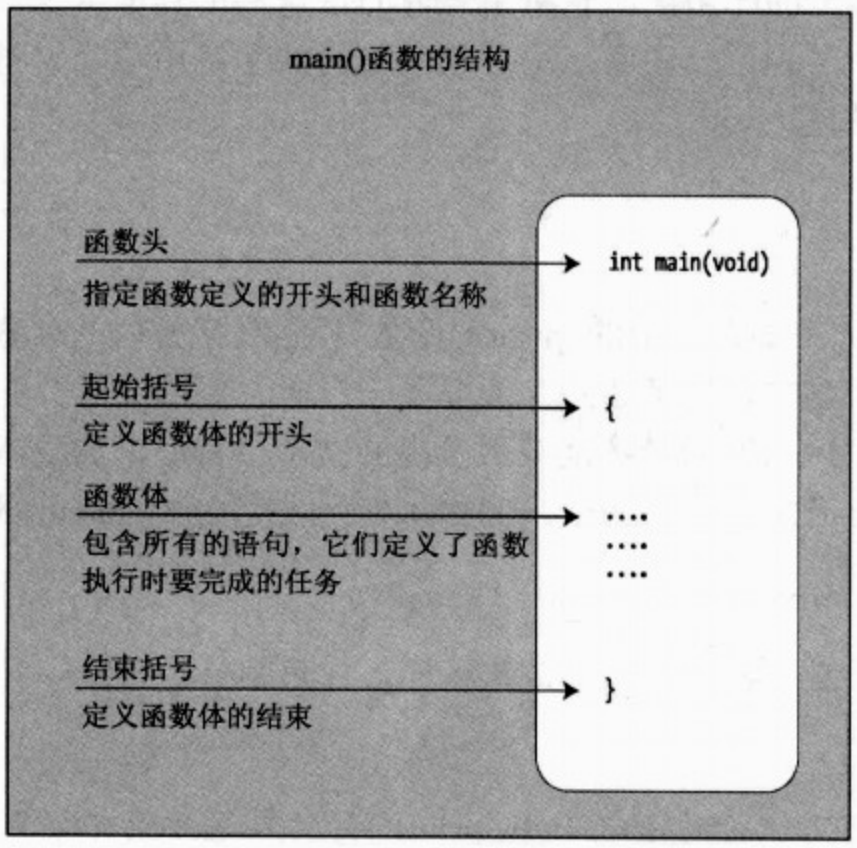


图 1-2 函数 main()的结构

函数体是在函数名称后面位于起始及结束两个大括号之间的代码块。它包含了定义函数功能的所有语句。这个例子的 main()函数体非常简单，只有两个语句：

```
{                    /* This marks the beginning of main() */
 printf("Beware the Ides of March!"); /* This line displays a quotation */
  return 0;          /* This returns control to the operating system */
}                    /* This marks the end of main() */
```

每个函数都必须有函数体，但函数体可以是空的，仅有起始及结束两个大括号，里面没有任何语句，在这种情况下，这个函数什么也不做。

这样的函数有什么用？事实上，在开发一个包含很多函数的程序时，这种函数是非常有用的。我们可以声明一些用来解决手头问题的空函数，确定需要完成的编程工作，再为每个函数创建程序代码。这个方法有助于条理分明地、系统地建立程序。

注意：

程序 1.3 将大括号单独排为一行，这么做可清楚地表示括号框起来的语句块从哪里起始和结束。大括号之间的语句通常缩进 2 个或多个空格，使大括号突出在前。这是个很好的编程格式，可以使语句块更容易阅读。

1.5.6 输出信息

例子中的 main()函数体包含了一个调用 printf()函数的语句：

```
printf("Beware the Ides of March!"); /* This line displays a quotation */
```

www.TopSage.com

printf()是一个标准的库函数，它将引号内的信息输出到屏幕上，在这个例子里，调用这个函数会显示引号内的一段警示语：双引号内的字符串称为字符串字面量。注意这行代码用分号作为结尾。

1.5.7 参数

包含在函数名(如上面语句中的 printf()函数)后的圆括号内的项称为参数，它指定要传送给函数的数据。当传送给函数的参数多于一个时，要用逗号分开。

在上面的例子中，函数的参数是双引号内的文本字符串。如果不喜欢例子中引号内的文本，可以改用自己想输出的句子。例如，使用马克白(Macbeth)的一句话：

```
printf("Out, damned Spot! Out I say!");
```

修改源代码后，必须再次编译及链接程序，才可执行。

注意：

与 C 语言中所有可执行的语句一样，printf()行的末尾必须有分号(这与定义语句或指令语句不同)。这是一个很容易犯的错误，尤其是初次使用 C 编程的人，老是忘了分号。

1.5.8 控制符

前面的程序可以改为输出两段句子。输入以下的代码：

```
/* Program 1.4 Another Simple C Program - Displaying a Quotation */
#include  <stdio. h>

int main(void)
{
   printf("\nMy formula for success?\nRise early, work late, strike oil.");
   return 0;
}
```

输出的结果是：

```
My formula for success?
Rise early, work late, strike oil.
```

在 printf()语句中，在文本的开头和第一句的后面，增加了字符\n，它代表一个字符：换行符。

反斜杠(\)在字符串里有特殊的意义，它表示转义序列的开始。转义序列可以在字符串中插入无法指定的字符，例如制表符及换行，或编译器在某些情况下会混淆的字符，例如双引号一般用于界定字符串。反斜杠后面的字符表示是哪种转义序列。在这个例子里，n 表示换行。还有其他许多转义序列。显然，反斜杠是有特殊意义的，所以需要一种方式在字符串中指定反斜杠。为此，应使用两个反斜杠(\\)。同样，如果要输出双引号，就用\"。

www.TopSage.com

输入以下的程序：

```
/* Program 1.5 Another Simple C Program - Displaying Great Quotations */
#include  <stdio.h>

int main(void)
{
   printf("\n\"It is a wise father that knows his own child.\"
     Shakespeare" );
   return 0;
}
```

输出的结果如下：

```
"It is a wise father that knows his own child."  Shakespeare
```

使用转义序列 \a 可以发出声音，说明发生了有趣或重要的事情。输入以下的程序并执行：

```
/* Program 1.6 A Simple C Program-Impontant */
#include  <stdio.h>

int main(void)
{
  printf("\nBe  careful!!\a");
  return 0;
}
```

这个程序的输出如下所示且带有声音。仔细聆听，电脑的扬声器会发出鸣响。

```
Be  careful!!
```

转义序列 \a 表示发出鸣响。表 1-1 是转义序列表。

表 1-1 转义序列

转 义 序 列	说　明
\n	换行
\r	回车键
\b	退后一格
\f	换页
\t	水平制表符
\v	垂直制表符
\a	发出鸣响
\?	插入问号(?)
\"	插入双引号(")
\'	插入单引号(')
\\	插入反斜杠(\)

www.TopSage.com

试着在屏幕上显示多行文本，在该文本中插入空格。使用 \n 可以把文本放在多个行上，使用 \t 可以给文本加上空格。本书将大量使用这些转义序列。

1.6　用 C 语言开发程序

如果读者从未写过程序，对 C 语言开发程序的过程就不会很清楚，但它和我们日常生活的许多事务是相同的，万事开头难。一般首先大致确定要实现的目标，接着把该目标转变成比较准确的规范。有了这个规范后，就可以制订达到最终目标的一系列步骤了。就好比光知道要盖房子是不够的，还得知道需要盖什么样的房子，它有多大，用什么材料，要盖在哪里。这种详细规划也需要运用到编写程序上。

下面介绍编写程序时需要完成的基本步骤。房子的比喻是很有帮助的，因此就利用这个比喻。

1.6.1　了解问题

第一步是弄清楚要做什么。在不清楚应提供什么设施：多少间卧房、多少间浴室、各房间多大等等之前就开始建造房子，会有不知所措之感。所有这些都会影响建造房子所需的材料和工作量，从而影响整个房子的成本。一般来说，在满足需求和完成项目的有限资金、人力及时间之间总会达成某种一致。

这和开发一个任意规模的程序是相同的。即使是很简单的问题，也必须知道有什么输入，对输入该做什么处理，要输出什么，以及输出哪种格式。输入可以来自键盘，也可以来自磁盘文件的数据，或来自电话或网络的信息。输出可以显示在屏幕上，或打印出来，也可以是更新磁盘上的数据文件。

对于较复杂的程序，须多了解程序的各个方面。清楚地定义程序要解决的问题，对于理解制订最终方案所需的资源与努力，是绝对必须的一部分。好好考虑这些细节，还可以确定项目是否切实可行。

1.6.2　详细设计

要建造房子，必须有详细的计划。这些计划能让建筑工人按图施工，并详细描述房子如何建造——具体的尺寸、要使用的材料等。还需要确定何时完成什么工作。例如，在砌墙之前须先挖地基，所以这个计划必须把工作分为可管理的单元，以便执行起来井然有序。

写程序也是一样。首先将程序分解成许多定义清楚且互相独立的小单元，描述这些独立单元相互沟通的方式，以及每个单元在执行时需要什么信息，从而开发出富有逻辑、相互独立的单元。把大型程序编写为一个大单元肯定是不可行的。

www.TopSage.com

1.6.3 实施

有了房子的详细设计，就可以开始工作了。每组建筑工人必须按照进度完成他们的工作。在下一阶段开始前，必须先检查每个阶段是否正确完成。省略了这些检查，将可能导致整栋房子倒塌。

当然，假使程序很大，可以一次编写一部分。一个部分完成后，再写下一部分。每个部分都要基于详细的设计规范，在进行下一个部分之前，应尽可能详细地检查每个部分的功能。这样，程序就会逐步完成预期的任务。

1.6.4 测试

房子完成了，还要进行许多测试：排水设备、水电设施、暖气等。任何部分都有可能出问题，这些问题必须解决。这有时是一个反复的过程，一个地方的问题可能会造成其他地方出问题。

这个机制与写程序是类似的。每个程序模块——组成程序的单元——都需要单独测试。若它们工作不正常，就必须调试。调试(Debugging)是一个找出程序中的问题及更正错误的过程。调试的由来有个说法，曾经有人在查找程序的错误时，使用计算机的电路图来跟踪信息的来源及其处理方式，竟然发现计算机程序出现错误，是因为一只虫子在电脑里，让里面的线路短路而发生的，后来，bug(虫子)这个词就成了程序错误的代名词。

对于简单的程序，通常只要检查代码，就可以找出错误。然而一般来说，调试过程通常会加入额外的程序代码，输出一些信息，来确定程序中事件的发生顺序以及即将生成的值。在大型的程序里，还需要联合测试各个程序模块，因为各个模块或许能正常工作，但并不保证它能和其他模块一起正常工作。在程序开发的这个阶段，有个专业术语叫集成测试(integration testing)。

1.7 函数及模块化编程

到目前为止，“函数”这个词已出现过好几次了，如 main()、printf()、函数体等。下面将深入研究函数是什么，为什么它们那么重要。

大多数编程语言(包含 C 语言)都提供了一种方法，将程序切割成多个段，各段都可以独立编写。在 C 语言中，这些段称为函数。一个函数的程序代码与其他函数是相互隔绝的。函数与外界有一个特殊的接口，可将信息传进来，也可将函数产生的结果传出去。这个接口在函数的第一行即在函数名的地方指定。

图 1-3 的程序例子由 4 个函数组成，用于分析棒球分数。

www.TopSage.com

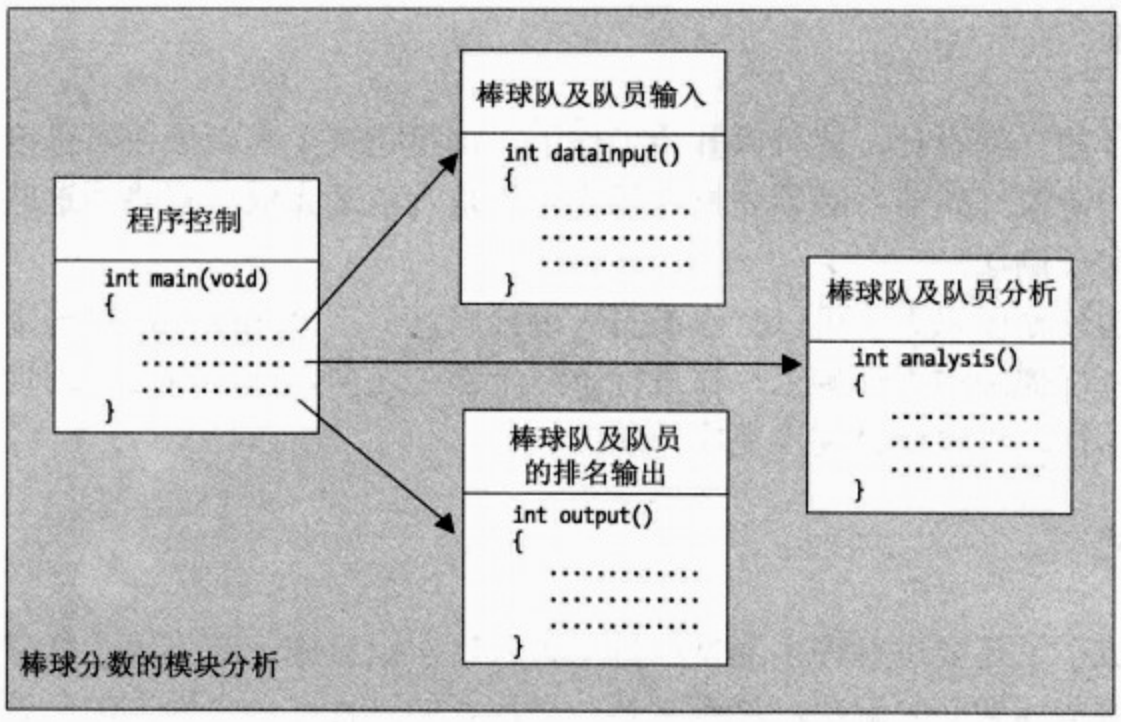


图 1-3　模块化编程

这 4 个函数都完成一个指定的、定义明确的工作。程序中操作的执行由一个模块 main()总体掌控。一个函数负责读入及检查输入数据，另一个函数进行分析。读入及分析了数据后，第 4 个函数就输出球队及球员的排名。

将程序分割成多个易于管理的小单元，对编程是非常重要的，其理由如下：

- 可以单独编写和测试每个函数，大大简化了使整个程序运转起来的过程。
- 几个独立的小函数比一个大函数更容易处理和理解。
- 库就是供人使用的函数集。因为它们是事先写好，且经过测试，能正常工作，所以可以放心地使用，无须细究它的代码细节。这就加快了开发程序的速度，因为我们只须关注自己的代码，这是 C 语言的一个基本组成部分。C 语言中丰富的函数库大大增强了 C 语言的能力。
- 也可以编写自己的函数库，应用于自己感兴趣的程序类型。如果发现经常编写某个函数，就可以编写它的通用版本，以满足自己的需求，并将它加入自己的库中。以后需要用到这个函数时，就可使用它的库版本了。
- 在开发包含几千到几百万行代码的大型程序时，可以由一些程序设计团队来进行，每个团队负责一个指定的函数子组，最后把它们组成完整的程序。

第 8 章将详细介绍 C 函数。C 程序的结构在本质上就是函数的结构，本章的第一个例子就用到一个标准的库函数 printf()。

试试看：将所学的知识用于实践

下面的例子将前面学到的知识用于实践。首先，看看下面的代码，检查自己是否理解它的作用。然后输入这些代码，编译、链接并执行，看看会发生什么。

```
/* Program 1.7 A longer program */
#include <stdio.h> /* Include the header file for input and output */
```

www.TopSage.com

```
int main(void)
{
   printf("Hi there!\n\n\nThis program is a bit");
   printf(" longer than the others.");
   printf("\nBut really it's only more text.\n\n\n\a\a");
   printf("Hey, wait a minute!! What was that???\n\n");
   printf("\t1.\tA bird?\n");
   printf("\t2.\tA plane?\n");
   printf("\t3.\tA control character? \n");
   printf("\n\t\t\b\bAnd how will this look when it prints out? \n\n");
   return 0;
}
```

输出如下:

```
Hi there!


This program is a bit longer than the others.
But really it's only more text.


Hey, wait a minute!! What was that???

1. A bird?
2. A plane?
3. A control character?

   And how will this look when it prints out?
```

代码的说明

这个程序看起来有点复杂，这只是因为括号内的文本字符串包含了许多转义序列。每个文本字符串都由一对双引号括起来。但这个程序只是连续调用 printf()函数，说明屏幕输出是由传送给 printf()函数的数据所控制。下面详细分析一下这个程序。

通过预处理指令包含了标准库中的 stdio.h 文件:

```
#include <stdio.h> /* Include the header file for input and output */
```

这是一个预处理指令，因为它以符号#开头。stdio.h 文件提供了使用 printf()函数所需的定义。

然后，定义 main()函数头，指定它返回一个整数值:

```
int main(void)
```

下一行的大括号表示其下是函数体:

```
{
```

下一行语句调用标准库函数 printf()，将“Hi there!”输出到屏幕上，接着空两行，

www.TopSage.com

输出 “This program is a bit”。

```
printf("Hi there!\n\n\nThis program is a bit");
```

空两行是由 3 个转义序列\n 生成的。转义序列\n 会把字符显示在新行上。第一个转义序列\n 结束了包含 “Hi there!” 的行，之后的两个转义序列\n 生成两个空行，文本 “This program is a bit” 显示在第 4 行上。这行代码在屏幕上生成了 4 行输出。

下一个 printf()生成的输出跟在上一个 printf()输出的最后一个字符后面。下面的语句输出文本 “ longer than the others.”，其中的第一个字符是一个空白:

```
printf(" longer than the others. ");
```

这个输出跟在上一个输出的后面，紧临 bit 中的 t。所以在文本的开头需要一个空格，否则计算机就会显示 “This program is a bitlonger than the others. ”，这不是我们希望的结果。

下一个语句在输出前会先换行，因为文本字符串的开头是\n:

```
printf("\nBut really itS only more text.\n\n\n\a\a");
```

显示完文本后会空两行(因为有 3 个\n 转义序列)，然后发出两次鸣响。下一个屏幕输出从空的第二行开始。

下一个输出语句如下:

```
printf("Hey, wait a minute!! What was that???\n\n");
```

输出文本后空一行。其后的输出在空的这行开始。

以下 3 行语句各插入一个制表符，显示一个数字后，再插入另一个制表符，之后是一些文本，结束后换行。这样，输出更容易阅读:

```
printf("\t1.\tA bird?\n");
printf("\t2.\rA plane?\n");
printf("\t3.\tA control character?\n");
```

这几个语句会生成 3 行带编号的输出。

下一语句先输出一个换行符，所以在前面输出的后面是一个空行，然后输出两个制表符和两个空格，接着退回两个空格，最后显示文本并换行:

```
printf("\n\t\t\b\bAnd how will this look when it prints out?\n\n");
```

函数体中的最后一个语句如下:

```
return 0;
```

这个语句结束 main()的执行，把 0 返回给操作系统。

结束大括号表示函数体结束:

```
}
```

www.TopSage.com

1.8 常见错误

错误是生活中的一部分。用 C 语言编写计算机程序时，必须用编译器将源代码转换成机器码，所以必须用非常严格的规则控制使用 C 语言的方式。漏掉一个该有的逗点，或添加不该有的分号，编译器都不会将程序转换成机器码。

即使实践了多年，程序中也很容易出现输入错误。这些错误可能在编译或链接程序时找出。但有些错误可能使程序执行时，表面上看起来正常，却不定时地出错，这就需要花很多时间来跟踪错误了。

当然，不是只有输入错误会带来问题，具体实施时也常常会发现问题。在处理程序中复杂的判断结构时，很容易出现逻辑错误。从语言的观点看，程序是正确的，编译及运行也正确，但得不到正确的结果。这类错误最难查找。

1.9 要点

温习第一个程序是个不错的方法，图 1-4 列出了重点。

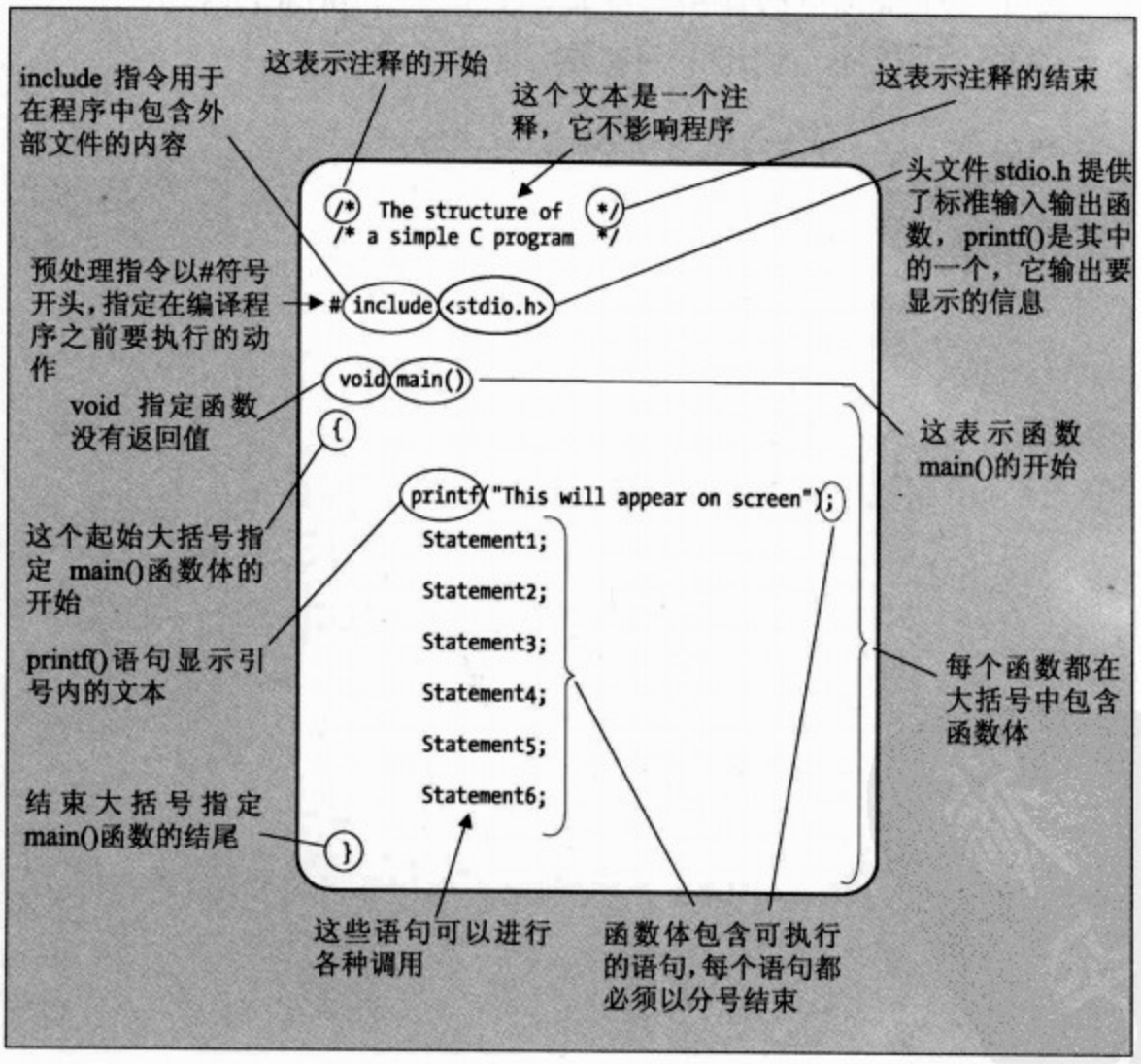


图 1-4 简单程序的要素

www.TopSage.com

1.10　小结

本章编写了几个 C 程序。我们学习了许多基础知识，本章的重点是介绍一些基本概念，而不是详细探讨 C 程序语言。现在读者应该对编写、编译及链接程序很有信心了。也许读者目前对如何构建 C 程序只有模糊的概念。以后学了更多的 C 语言知识，编写了一些程序后，就会清楚明白了。

下一章将学习较复杂的内容，而不只是用 printf()输出文本。我们要处理信息，得到更有趣的结果。另外，printf()不只是显示文本字符串，它还有其他用途。

1.11　习题

以下的习题能让读者测试本章所学的成果。如果有不懂的地方，可以翻看本章的内容，还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

练习 1.1　编写一个程序，用两个 printf()语句分别输出自己的名字及地址。

练习 1.2　将上一个练习修改成所有的输出只用一个 printf()语句。

练习 1.3　编写一个程序，输出下列文本，格式如下所示：

```
"It's  freezing  in  here," he said coldly.
```

www.TopSage.com

第2章

编程初步

现在读者一定很渴望编写程序，让计算机与外界进行实际的交互。我们不希望程序只能做打字员的工作，显示包含在程序代码中的固定信息。的确，编程的内涵远不止此。

理想情况下，我们应能从键盘上输入数据，让程序把它们存储在某个地方，这会让程序更具多样性。程序可以访问和处理这些数据，而且每次执行时，都可以处理不同的数据值。每次运行程序时输入不同的信息正是整个编程业的关键。在程序中存储数据项的地方是可以变化的，所以叫做变量(variable)，而这正是本章的主题。

本章相当长，介绍了许多基础知识。阅读完本章，就可以开始编写真正有用的程序了。

本章的主要内容：

- 内存的用法及变量的概念
- 在 C 中如何计算
- 变量的不同类型及其用途
- 强制类型转换的概念及其使用场合
- 编写一个程序，计算树木的高度

2.1 计算机的内存

首先看看计算机如何存储程序要处理的数据。为此，就要了解计算机的内存，在开始编写第一个程序之前，先简要介绍计算机的内存。

计算机执行程序时，组成程序的指令和程序所操作的数据都必须存储到某个地方。这个地方就是机器的内存，也称为主内存(main memory)，或随机访问存储器(Random Access Memory，RAM)。

计算机还包含另一种存储器，称为只读存储器(Read-Only Memory，ROM)。顾名思义，ROM 不能修改，只能读取其内容，或让计算机执行包含在 ROM 中的指令。ROM 中的信息是在制造机器时放进去的。这些信息主要是一些程序，用来控制连接到计算机上的各种设备的运作，如显示器、硬盘驱动器、键盘及软盘驱动器等。在 PC 上，这些程序称为计算机的基本输入输出系统(Basic Input/Output System，BIOS)。

本书不打算介绍 BIOS，只讨论 RAM——这是执行程序时存储程序及数据的地方。

可以将计算机的 RAM 想象成一排井然有序的盒子。每个盒子都有两个状态：满为 1，空为 0。因此每个盒子代表一个二进制数：0 或 1。计算机有时用真(true)和假(false)

TopSage.com

表示它们：1 是真，0 是假。每个盒子称为一个位(bit)，即二进制数(binary digit)的缩写。

注意：

如果读者不记得或从来没学过二进制数，可参阅附录 A。但如果不明白这些内容，不用担心，因为这里的重点是计算机只能处理 0 与 1，而不能直接处理十进制数。程序使用的所有数据(包括程序指令)都是由二进制数组成的。

为了方便起见，盒子或计算机中的位以 8 个为一组，每组的 8 位称为一个字节(byte)。为了使用字节的内容，每个字节用一个数字表示，第一个字节用 0 表示，第二个字节用 1 表示，直到计算机内存的最后一个字节。字节的这个标记称为字节的地址(address)。因此，每个字节在内存里都有一个和其他字节不同的地址。每栋房子都有一个唯一的街道地址。同样，字节的地址唯一地表示计算机内存中的字节。

总之，内存的最小单位是位(bit)，将 8 个位组合为一组，称为字节(byte)。位只能是 0 或 1，如图 2-1 所示。

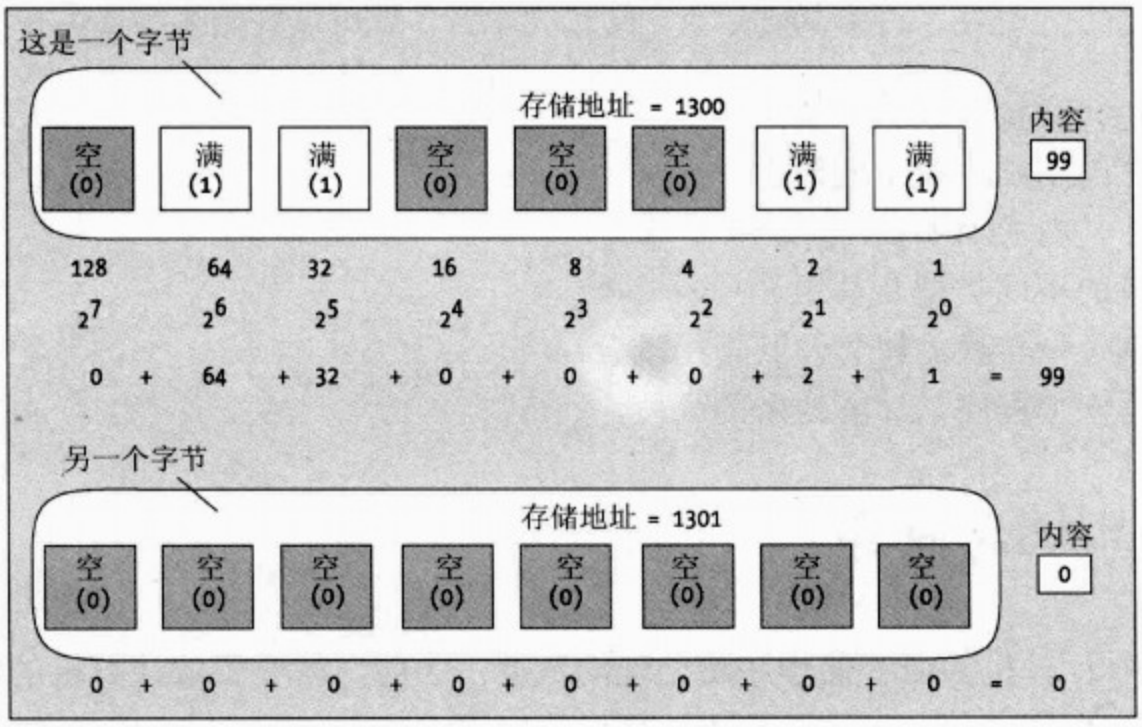


图 2-1 内存中的字节

计算机内存的常用单位是千字节(KB)、兆字节(MB)、千兆字节 (GB)。这些单位的意义如下：

- 1KB 是 1 024 字节。
- 1MB 是 1 024KB，也就是 1 048 576 字节。
- 1GB 是 1 024MB，也就是 1 073 741 841 字节。

为什么不使用更简单的整数，例如千、百万或亿？因为从 0 到 1023 共 1024 个数字，而在二进制中，1023 的 10 个位刚好全是 l：11 1111 1111，它是一个非常方便的二进制数。1000 是很好用的十进制数，但是在二进制的计算机里就不再那么方便了，它是 11 1110 1000。因此以 KB(1 024 字节)为单位，是为了方便计算机使用。同样，MB 需要 20 个位，GB 需要 30 个位。但是这有一个问题，特别是硬盘的容量。磁盘制造商常常宣称他们生

www.TopSage.com

产的磁盘的容量是 537MB 或 18.3GB，而实际上这两个数字表示 53 700 万字节及 183 亿字节。当然，53 700 万字节只有 512MB，而 183 亿字节只有 17GB，所以磁盘制造商给出的硬盘容量有误导作用。

有了字节的概念，下面看看如何在程序里使用这些内存。

2.2 什么是变量

变量是计算机里一块特定的内存，它是由一个或多个连续的字节所组成。每个变量都有一个名称，可以用该名称表示内存的这个位置，以提取它包含的数据或存储一个新数值。

下面编写一个程序，用第 1 章介绍的 printf()函数显示你的薪水。假设你的薪水是 10 000 元/月，则很容易编写这个程序。

```
/* Program 2.1 What is a Variable? */
#include <stdio.h>

int main(void)
{
    printf("My salary is $10000");
    return 0;
}
```

这个程序的工作方式不需要多做解释，它和第一章开发的程序差不多。如何修改这个程序，让它能够根据存储在内存中的值，定制要显示的信息？这有几种方法，它们有一个共同点：使用变量。

在这个例子里，可以分配一块名为 salary 的内存，把值 10 000 存储在该变量中。要显示薪水时，可以使用给变量指定的名称 salary；将存储在其中的值 10 000 显示出来。程序用到变量名时，计算机就会访问存储在其中的值。变量的使用次数是不受限制的。当薪水改变时，只需改变 salary 变量存储的值，整个程序就会使用新的值。当然，在计算机中，所有的值都存储为二进制数。

程序里变量的数量是没有限制的。在程序执行过程中，每个变量包含的值由程序的指令来决定。变量的值不是固定的，而可以随时改变，且没有次数的限制。

2.3 存储数值的变量

变量有几种不同的类型，每种变量都用于存储特定类型的数据。首先看看用于存储数值的变量。在程序里存放数字的方法很多，下面从最简单的方法开始。

2.3.1 整数变量

首先是存储整数的变量。整数是没有小数点的数字。例如：

www.TopSage.com

```
1
10,999,000,000
-1
```

这些数值是整数，但这对程序而言并不完全正确。整数是不能包含逗号的，所以第二个值在程序里应该写成 10999000000。

下面是一些不是整数的例子：

```
1.234
999.9
2.0
-0.0005
```

2.0 一般算作整数，但是计算机不将它算作整数，因为它带有小数点。在程序里，必须把这个数字写作 2，不带小数点。在 C 程序中，整数总是写成不带小数点的数字，如果数字中有小数点，就不是整数。在详细讨论变量之前，先看看程序里的一个简单的变量，学习变量的用法。

试试看：使用变量

回到“薪水”的例子。将前面的程序改为使用一个变量：

```
/*Program 2.2 Using a variable */
#include <stdio.h>

int main(void)
{
    int salary;      /* Declare a variable called salary          */
    salary = 10000; /* A simple arithmetic assignment statement */
    printf("My salary is %d.", salary);
    return 0;
}
```

输入这个例子，编译、链接并执行，会得到下面的结果：

```
My salary is 10000.
```

代码的说明

前三行和前一个例子相同，下面看看新的语句。

用来存放薪水的变量声明语句如下：

```
int salary; /* Declare a variable called salary */
```

这个语句称为变量声明，因为它声明了变量的名称。在这个程序中，变量名是 salary。

警告：

变量声明语句以分号结束。如果漏掉分号，程序编译时会产生错误。

变量声明也指定了这个变量存储的数据类型，这里使用关键字 int 指定，salary 用来

www.TopSage.com

存放一个整数。关键字 int 放在变量名称之前。如后面所述，声明存储其他数据类型的变量时，要使用另一个关键字指定数据类型，其方式大致相同。

注意：

关键字是特殊的 C 保留字，对编译器有特殊的意义。不能将它们用作变量名称，否则编译器会拒绝该变量名。

变量声明也称为变量的定义，因为它分配了一些存储空间，来存储整数值，该整数可以用变量名 salary 来引用。当然，现在还未指定变量 salary 的值，所以此刻该变量包含一个垃圾值，即上次使用这块内存空间时遗留在此的值。

下一个语句是：

```
salary = 10000;
```

这是一个简单的算术赋值语句，它将等号右边的数值存储到等号左边的变量中。这里声明了变量 salary，它的值是 10 000。将右边的值 10 000 存储到左边的变量 salary 中。等号“=”称为赋值运算符，它将右边的值赋予左边的变量。

然后是熟悉的 printf()语句，但这里的用法和之前稍有不同：

```
printf("My salary is %d.", salary);
```

括号内有两个参数，用逗号分开。参数是传递给函数的值。在这个程序语句中，传给 printf()函数的两个参数如下：

- 参数 1 是一个控制字符串，用来控制其后的参数输出以什么方式显示，它是放在双引号内的字符串，也称为格式字符串，因为它指定了输出数据的格式。
- 参数 2 是变量名 salary。这个变量值的显示方式是由第一个参数——控制字符串来确定。

这个控制字符串和前一个例子相当类似，都包含一些要显示的文本。但在本例的这个字符串中有一个%d，它称为变量的转换指定符(conversion specifier)。

转换指定符确定变量在屏幕上的显示方式。在本例中使用了 d，它是应用于整数值的十进制指定符，表示第二个参数 salary 输出为一个十进制数。

注意：

转换指定符总是以%字符开头，以便 printf()函数识别出它们。控制字符串中的%总是表示转换指定符的开头；所以如果要输出%字符，就必须用转义序列%%。

试试看：使用更多的变量

试试一个稍大的程序：

```
/* Program 2.3 Using more variables */
#include <stdio.h>

int main(void)
{
```

www.TopSage.com

```
    int brothers;    /* Declare a variable called brothers */
    int brides;      /* and a variable called brides       */

    brothers = 7;    /* Store 7 in the variable brothers   */
    brides = 7;      /* Store 7 in the variable brides     */

    /* Display some output */
    printf("%d brides for %d brothers", brides, brothers);
    return 0;
}
```

执行程序的结果如下:

```
7 brides for 7 brothers;
```

代码的说明

这个程序和前一个例子相当类似。首先声明两个变量 brothers 和 brides，语句如下:

```
int brothers; /* Declare a variable called brothers */
int brides;   /* and a variable called brides       */
```

两个变量都声明为 int 类型，都存储整数值。注意，它们在两个语句中声明。由于这两个变量的类型相同，故可以将它们放在同一行代码上声明:

```
int brothers, brides;
```

在一个语句中声明多个变量时，必须用逗号将数据类型后面的变量名分开，该语句要用分号结束。这是一种很方便的格式，但有一个缺点：每个变量的作用不很明显，因为它们全放在一行代码上，不能加入注释，来描述每个变量。因此可以将它们分成两行，语句如下:

```
int brothers,  /* Declare a variable called brothers */
    brides;    /* and a variable called brides       */
```

将语句分成两行，就可以加入注释了。这些注释会被编译器忽略，因此和最初没加入注释的语句相同。当然也可以编写两个声明语句。

注意，声明放在函数的可执行语句的开头。应该将要使用的所有变量的声明放在开头。

之后的两个语句给两个变量赋值 7:

```
brothers = 7; /* Store 7 in the variable brothers */
brides = 7;   /* Store 7 in the variable brides   */
```

注意，声明这些变量的语句放在上述语句之前。如果遗漏了某个声明，或把声明语句放在后面，程序就不会编译。

下一个语句调用 printf()函数，将一个控制字符串作为第一个参数传送给它，以显示一行文本。这个控制字符串中的两个转换指定符%d 会分别被 printf()函数的第二个参数 brides 和第三个参数 brothers 的值取代:

www.TopSage.com

```
printf("%d brides for %d brothers", brides, brothers);
```

转换指定符按顺序被 printf()函数的第二个参数 brides 和第三个参数 brothers 的值取代：变量 brides 的值对应第一个%d，变量 brothers 的值对应第二个%d。如果将设置变量值的语句改为如下所示，将会更清楚：

```
brothers = 8; /* Store 8 in the variable brothers */
brides = 4;   /* Store 4 in the variable brides   */
```

在这个比较明确的例子中，printf()语句会清楚地显示变量和转换指定符的对应关系，因为输出如下所示：

```
4 brides for 8 brothers
```

2.3.2 变量的命名

给变量指定的名称一般称为变量名。变量的命名是很有弹性的。它可以是一个或多个大写或小写字母、数字和下划线(_)(有时下划线也算做字母)。下面是一些正确的变量名：

```
Radius
diameter
Auntie_May
Knotted_Wool
D678
```

变量名不能以数字开头，所以 8_Ball 和 6_pack 都是不合法的名称。变量名只能包含字母、下划线和数字，所以 Hash!及 Mary-Lou 都不能用作变量名。Mary-Lou 是一个常见的错误，但是 Mary_Lou 就是可以接受的。变量名中不能有空格，所以 Mary Lou 会被视为两个变量名 Mary 和 Lou。以一或两个下划线开头的变量名常用在头文件中，所以在给变量命名时，不要将下划线用作第一个字符，以免和标准库里的变量名冲突。例如最好避免使用_this 和_that 这样的变量名。

可以在上述限制内随意指定变量名，但最好使变量名有助于了解该变量包含的内容，例如变量名 salary 很容易看出它包含的是薪水信息。

警告：

变量名可以包含的字符数取决于编译器，遵循 C 语言标准的编译器至少支持 31 个字符，只要不超过这个长度就没问题。建议变量名不要超过这个长度，因为这样的变量名比较繁琐，代码也难以理解。有些编译器会截短过长的变量名。

在给变量命名时，要特别注意的是，变量名是区分大小写的，即 Democrat 和 democrat 是两个完全不同的变量。修改 printf()函数，使其中一个变量名以大写字母开头，如下所示：

```
/* Program 2.3A Using more variables */
#include <stdio.h>
```

www.TopSage.com

```
int main(void)
{
   int brothers; /* Declare a variable called brothers */
   int brides;   /* and a variable called brides       */

   brothers = 7; /* Store 7 in the variable brothers   */
   brides = 7;   /* Store 7 in the variable brides     */

   /* Display some output */
   printf("%d brides for %d brothers", Brides, brothers),
   return 0;
}
```

编译这个版本的程序时，会得到一个错误信息。编译器把 brides 和 Brides 解释为两个不同的变量，所以它不理解 Brides 这个变量。这是一个常见的错误，如前所述，打字和拼写错误是出错的一个主要原因。

变量必须在使用之前声明，否则编译器就无法识别，将该语句标识为错误。

2.3.3 变量的使用

前面介绍了如何声明及命名变量，但这和在第一章学到的知识相比并没有太多用处。下面编写另一个程序，在产生输出前使用变量的值。

试试看：作一个简单的计算

这个程序用变量的值做简单的计算：

```
/* Program 2.4 Simple calculations */
#include <stdio.h>

int main(void)
{
   int Total_Pets;
   int Cats;
   int Dogs;
   int Ponies;
   int Others;

   /* Set the number of each kind of pet */
   Cats = 2;
   Dogs = 1;
   Ponies = 1;
   others = 46;

   /* Calculate the total number of pets */
   Total_Pets = Cats + Dogs + Ponies + Others;

   printf("We have %d pets in total", Total_Pets); /* Output the result */
```

www.TopSage.com

```
    return 0;
}
```

执行程序的结果如下：

```
We have 50 pets in total
```

代码的说明

与前面的例子一样，大括号中的所有语句都有相同的缩进量，这说明这些语句都包含在这对大括号中。应当仿效此法组织程序，使位于一对大括号之间的一组语句有相同的缩进量，使程序更易于理解。

首先，定义 5 个 int 类型的变量：

```
int Total_Pets;
int Cats;
int Dogs;
int Ponies;
int Others;
```

因为这些变量都用来存放动物的数量，它们肯定是整数，所以都声明为 int 类型。

也可以在一个语句中声明 5 个变量，并添加注释，如下所示：

```
int Total_Pets,  / * The total number of pets     */
    Cats,        /* The number of cats as pets   */
    Dogs,        /* The number of dogs as pets   */
    Ponies,      /* The number of ponies as pets */
    Others;      /* The number of other pets     */
```

这些都是很简单的注释，但它们说明了要点。这个语句分成好几行，以便用整齐的格式添加注释。注意要用逗号将每个变量名分开。编译器会忽略注释，所以这个语句和下面的语句等价：

```
int Total_Pets, Cats, Dogs, Ponies, Others;
```

可以将 C 语句分成好几行。分号决定语句的结束，而不是代码行的结束。

下面用 4 个赋值语句给变量指定特定的值：

```
Cats = 2;
Dogs = 1;
Ponies = 1;
Others = 46;
```

现在，变量 Total_Pets 还没有设定明确的值，它的值是使用其他变量进行计算的结果：

```
Total_Pets = Cats + Dogs + Ponies + Others;
```

在这个算术语句中，把每个变量的值加在一起，计算出赋值运算符右边的所有宠物数的总和，再将这个总和存储到赋值运算符左边的变量 Total_Pets 中。这个新值替代了

www.TopSage.com

存储在变量 Total_Pets 中的旧值。

printf()语句显示了变量 Total_Pets 的值，即计算结果:

```
printf("We have %d pets in total", Total_Pets);
```

试着改变某些宠物的值，或增加一些宠物。记住要声明它们，给它们设定数值，将它们加进变量 Total_Pets 中。

2.3.4 变量的初始化

在上面的例子，用下面的语句声明每个变量:

```
int Cats;    /* The number of cats as pets */
```

用下面的语句设定变量 Cats 的值:

```
Cats = 2;
```

将变量 Cats 的值设为 2。

这个语句执行之前，变量的值是什么？它可以是任何数。第一个语句创建了变量 Cats，但它的值是上一个程序在那块内存中留下的数值。其后的赋值语句将变量 Cats 的值设置为 2。但最好在声明变量时，就初始化它，语句如下所示:

```
int Cats = 2;
```

这个语句将变量 Cat 声明为 int 类型，并设定初值为 2。

声明变量时就初始化它一般是很好的做法。它可避免对初始值的怀疑，当程序运作不正常时，它有助于追踪错误。避免在创建变量时使用垃圾值，可以减少程序出错时计算机崩溃的机会。随意使用垃圾值可能导致各种问题，因此从现在起，就养成初始化变量的好习惯，即使是 0 也好。

2.3.5 算术语句

上面的程序是第一个真正做了些事情的程序。它非常简单，仅仅相加了几个数字，但这是非常重要的一步。它是运用算术语句进行运算的一个基本例子。下面介绍一些更复杂的计算。

1. 基本算术运算

在 C 语言中，算术语句的格式如下:

```
变量名 = 算术表达式;
```

赋值运算符右边的算术表达式指定使用变量中存储的值和/或明确给出的数字，以及算术运算符如加(+)、减(-)、乘(*)及除(/)进行计算。在算术表达式中也可以使用其他运算符，如后面所述。

www.TopSage.com

前面例子中的算术语句如下：

```
Total_Pets = Cats + Dogs + Ponies + Others;
```

这个语句先计算等号右边的算术表达式，再将所得的结果存到左边的变量中。

在 C 语言中，符号“=”定义了一个动作，而不是像数学中那样说明两边相等。它指定将右边表达式的结果存到左边的变量中。因此可以编写下面的语句：

```
Total_Pets = Total_Pets + 2;
```

以数学的观点来看，它是很荒唐的，但对编程而言它是正确的。假定重新编写程序，添加上面的语句。添加了这个语句的程序段如下：

```
Total_Pets = Cats + Dogs + Ponies + Others;
Total_Pets = Total_Pets + 2;
printf("The total number of pets is: %d", Total_Pets);
```

在执行完第一个语句后，Total_Pets 的值是 50。之后，第二行提取 Total_Pets 的值，给该值加 2，再将结果存储回变量 Total_Pets。因此最后显示出来的总数是 52。

注意：

在赋值运算中，先计算等号右边的表达式，然后将结果存到等号左边的变量中。新的值取代赋值运算符左边的变量中的原值。赋值运算符左边的变量称为 lvalue，因为在这个位置可以存储一个值。执行赋值运算符右边的表达式所得的值称为 rvalue，因为它是计算表达式所得的一个值。

计算结果是数值的表达式称为算术表达式，下面都是算术表达式：

```
3
1 + 2
Total_Pets
Cats + Dogs - Ponies
```

计算这些表达式，都会得到一个数值。注意，变量名也是一个表达式，它的计算结果是一个值，即该变量包含的值。稍后将详细讨论如何构建表达式，并学习运算规则。这里先用基本算术运算符做一些简单的例子。表 2-1 列出了这些算术运算符。

表 2-1 基本算术运算符

运 算 符	动 作
+	加
-	减
*	乘
/	除
%	取模(Modulus)

www.TopSage.com

前面没有提到过取模运算符。它用运算符左边的表达式值去除运算符右边的表达式值，并求出其余数，所以有时称为余数运算符。表达式 12 % 5 的结果是 2。因为 12 除以 5 的余数是 2。下一节将详细介绍。所有这些运算符的工作方式都与我们的常识相同，只有除法运算符例外，它应用于整数时有点不直观。下面进行一些算术运算。

试试看：减和乘

下面的基于食物的程序演示了减法和乘法：

```
/* Program 2.5 Calculations with cookies */
#include <stdio.h>

int main(void)
{
   int cookies = 5;
   int cookie_calories = 125; /* Calories per cookie */
   int total_eaten = 0;       /* Total cookies eaten */

   int eaten = 2;             /* Number to be eaten */
   cookies = cookies - eaten; /* Subtract number eaten from cookies */
   total_eaten = total_eaten + eaten;
   printf("\nI have eaten %d cookies. There are %d cookies left",
          eaten, cookies);
   eaten = 3;                 /* New value for cookies to be eaten */
   cookies = cookies - eaten; /* Subtract number eaten from cookies */
   total_eaten = total_eaten + eaten;
   printf("\nI have eaten %d more. Now there are %d cookies left\n",
          eaten, cookies);
   printf("\nTotal energy consumed is %d calories.\n",
          total_eaten*cookie_calories);
   return 0;
}
```

这个程序产生如下输出：

```
I have eaten 2 cookies. There are 3 cookies left
I have eaten three more. Now there are 0 cookies left

Total energy consumed is 625 calories.
```

代码的说明

首先声明并初始化 3 个 int 类型的变量：

```
int cookies = 5;
int cookie_calories = 125; /* Calories per cookie */
int total_eaten = 0;       /* Total cookies eaten */
```

在程序里，使用变量 total_eaten 计算吃掉的饼干总数，所以要将它初始化为 0。下一个声明并初始化的变量存储吃掉的饼干数，如下：

www.TopSage.com

```
int eaten = 2; /* Number to be eaten */
```

用减法运算符从 cookies 中减掉 eaten:

```
cookies = cookies - eaten; /* Subtract number eaten from cookies */
```

减法运算的结果存回 cookies 变量，所以 cookies 的值变成 3。因为吃掉了一些饼干，所以要给 total_eaten 增加吃掉的饼干数:

```
total_eaten = total_eaten + eaten;
```

将 eaten 变量的当前值 2 加到 total_eaten 的当前值 0 上，结果存储回变量 total_eaten。printf()语句显示剩下的饼干数:

```
printf("\nI have eaten %d cookies. There are %d cookies left",
          eaten,cookies);
```

这个语句在一行上放不下，所以在 printf()的第一个参数后的逗号后面，将该语句的其他内容放在下一行上。可以像这样分拆语句，使程序易于理解，或放在屏幕的指定宽度之内。

注意不能用这种方式拆分第一个字符串参数。不能在字符串的中间放置换行符。需要将字符串拆开成两行或多行时，一行上的每一段字符串必须有自己的一对双引号。例如，上面的语句可以写成:

```
printf("\nI have eaten %d cookies."
                " There are %d cookies left",
                                eaten, cookies);
```

如果两个或多个字符串彼此相邻，编译器会将它们连接起来，构成一个字符串。

用整数值的转换指定符%d 将 eaten 和 cookies 的值显示出来。在输出字符串中，eaten 的值取代第一个%d， cookies 的值取代第二个%d。字符串在显示之前会先换行，因为开头处有一个\n。

下一个语句将变量 eaten 的值设为一个新值:

```
eaten = 3; /* New value for cookies to be eaten */
```

新值 3 取代 eaten 变量中的旧值 2。然后完成和以前一样的操作序列:

```
cookies = cookies - eaten; /* Subtract number eaten from cookies */

 total_eaten = total_eaten + eaten;
printf("\nI have eaten %d more. Now there are %d cookies left\n",
       eaten, cookies);
```

最后，在执行 return 语句，结束程序前，计算并显示被吃掉饼干的卡路里数:

```
printf("\nTotal energy consumed is %d calories.\n",
                        total_eaten*cookie_calories);
```

www.TopSage.com

printf()函数的第二个参数是一个算术表达式，而不是变量。编译器会将表达式 total_eaten*cookie_calories 的计算结果存储到一个临时变量中，再把该值作为第二个参数传送给 printf()函数。函数的参数总是可以使用算术表达式，只要其计算结果是需要的类型即可。

下面看看除法和取模运算符。

试试看：除法和取模运算符

假设你有一罐饼干(其中有有 45 块饼干)和 7 个孩子。要把饼干平分给每个孩子，计算每个孩子可得到几块饼干，分完后剩下几块饼干。

```
/* Program 2.6 Cookies and kids */
#include <stdio.h>

int main(void)
{
   int cookies = 45;           /* Number of cookies in the jar */
   int children = 7;           /* Number of children           */
   int cookies_per_child = 0; /* Number of cookies per child  */
   int cookies_left_over = 0; /* Number of cookies left over  */

   /* Calculate how many cookies each child gets when they are divided up */
   cookies_per_child = cookies/children; /* Number of cookies per child */
   printf("You have %d children and %d cookies", children, cookies);
   printf("\nGive each child %d cookies.", cookies_per_child);

   /* Calculate how many cookies are left over */
   cookies_left_over = cookies%children;
   printf("\nThere are %d cookies left over.\n", cookies_left_over);
   return 0;
}
```

执行程序后的输出：

```
You have 7 children and 45 cookies
Give each child 6 cookies.
There are 3 cookies left over.
```

代码的说明

下面一步一步地解释这个程序。下面的语句声明并初始化 4 个整数变量：cookies、children、cookies_per_child、cookies_left_over:

```
int cookies = 45;           /* Number of cookies in the jar */
int children = 7;           /* Number of children           */
int cookies_per_child = 0; /* Number of cookies per child  */
int cookies_left_over = 0; /* Number of cookies left over  */
```

使用除号运算符“/”将饼干数量除以孩子的数量，得到每个孩子分得的饼干数：

```
cookies_per_child = cookies/children; /* Number of cookies per child */
```

下面两个语句输出结果，即 cookies/children 变量的值：

```
printf("You have %d children and %d cookies", children, cookies);
printf("\nGive each child %d cookies.", cookies_per_child);
```

从输出结果可以看出，cookies_per_child 的值是 6。这是因为当操作数是整数时，除法运算符总是得到整数值。45 除以 7 的结果是 6，余 3。下面的语句用取模运算符计算余数：

```
cookies_left_over = cookies % children;
```

赋值运算符右边的表达式计算 cookies 除以 children 得到的余数。最后一个语句输出余数：

```
printf("\nThere are %d cookies left over.\n", cookies_left_over);
```

2. 深入了解整数除法

当一个操作数是负数时，使用除法和模数运算符的结果是什么？在执行除法运算时，如果操作数不同号，结果就是负数。因此，表达式-45/7 和 45/-7 的结果相同，都是 - 6。如果操作数同号，都是正数或都是负数，结果就是正数。因此 45/7 和-45/-7 结果都是 6。至于模数运算符，其结果总是和左操作数的符号相同。因此 45%-7 等于 3，而-45/7 等于-3。

3. 一元运算符

前面使用的运算符都是二元运算符，因为它们都操作两个数据项。运算符操作的数据项称为操作数。例如，乘法运算符是一个二元运算符。因为它有两个操作数，其结果是一个操作数乘以另一个操作数。还有一些运算符是一元运算符，即它们只需一个操作数。后面将介绍更多的例子。但现在看看一个最常用的一元运算符。

4. 一元减号运算符

在比较复杂的程序里，一元减号运算符是很有用的，它可以把正数变成负数，把负数变成正数。要了解一元减号运算符的使用场合，考虑一下追踪银行账号。假定我们在银行存了 200 元。在簿子里用两列记录这笔钱的收支情况，一列记录付出的费用，另一列记录得到的收入，支出列是负数，收入列是正数。

我们决定购买一片价值 50 元的 CD 和一本价值 25 元的书。假使一切顺利，从银行的初始值中减掉支出的 75 元后，就得到了余额。表 2-2 说明这些项的记录情况。

表 2-2 收入与支出记录

项	收 入	支 出	存款余额
支票收入	$200		$200
CD		$50	$150
书		$25	$125
结余	$200	$75	$125

www.TopSage.com

如果将这些数字存储到变量中，可以将收入及支出都输入为正数，只有计算余额时，才会把这些数字变成负数。为此，可以将一个负号(-)放在变量名的前面。

要把总支出输出为负数，可编写如下语句：

```
int expenditure = 75;
printf("Your balance has changed by %d.", -expenditure);
```

这会产生如下结果：

```
Your balance has changed by -75.
```

负号表示花掉了这笔钱，而不是赚了。注意，表达式-expenditure 不会改变 expenditure 变量的值，它仍然是 75。这个表达式的值是-75。

在表达式-expenditure 中，一元减号运算符指定了一个动作，其结果是翻转 expenditure 变量的符号：将负数变成正数，将正数变成负数。必须执行程序里的指令，进行计算。这和编写一个负数(如-75 或-1.25)时使用的负号运算符是不同的。此时，负号不表示一个动作，程序执行时，不需要执行指令。它只是告诉编译器，在程序里创建一个负的常量。

2.4　变量与内存

前面介绍了整数变量，但未考虑过它们占用多少内存空间。每次声明变量时，计算机都会给它分配一块足够大的内存空间，来保存该类型的变量。相同类型的不同变量总是占据相同大小的内存(字节数)。但不同类型的变量需要分配的内存空间就不一样了。

注意：

在一台机器上，给定类型的变量占用的内存空间总是相同的。但在一些情况下，给定类型的变量在一台计算机上占用的内存比另一台计算机多。这是因为 C 语言规范让编译器的作者确定给定类型的变量占用多少内存空间。因此编译器的作者就可以选择变量的大小，以满足计算机的硬件结构要求。

本章的开头介绍了，计算机的内存组织为字节。每个变量都会占据一定数量的内存字节，那么存储整数需要几个字节？一个字节能存储-128~+127 的整数。这对于前面的例子而言已经足够，但是如何存储一双及膝的长筒袜上的平均针脚数？一个字节就不够了。因此在 C 语言中有不同类型的变量，来存储不同类型的数字，其中一个就是整数。整数变量还有几种不同的变体，以存储不同范围的整数。

下一节要叙述各种不同的类型。用一个表说明能存储的数值范围及变量所占的内存大小。本章的小结将这些总结成一个完整的变量类型表。

www.TopSage.com

2.5 整数变量类型

有 5 种基本的变量类型可以声明为存储带符号的整数值(无符号的整数值参见下一节)。每种类型都用不同的关键字或关键字组合来指定，如表 2-3 所示。

表 2-3 整数变量类型的名称

类 型 名 称	字 节 数	取 值 范 围
signed char	1	–128~+127
short int	2	–32 768~+32 767
int	4	–2 147 438 648~+2 147 438 647
long int	4	–2 147 438 648~+2 147 438 647
long long int	8	–9 223 372 036 854 775 808~+9 223 372 036 854 775 807

类型名称 short、long 和 long long 可以用作 short int、long int 和 long long int 的缩写，它们一般用缩写的形式。表 2-3 列出了每个整数变量类型的大小，但这些变量类型所占的内存空间取决于所使用的编译器。

注意：

在 C 语言的国际标准中，整数类型的唯一要求是，上表中的每种类型占用的内存空间都不少于其上的类型。类型 signed char 占用的内存空间与类型 char 相同，它有足够的内存空间来存储语言字符集中的任意字符，它一般是 1 字节，但也可以有更多字节。除了这些限制外，编译器的作者可以完全自由地、充分地利用执行编译器的机器上的硬件算术能力。

2.5.1 无符号的整数类型

对于每个存储带符号整数的类型，都有一个对应的类型来存储无符号的整数，它们占用的内存空间与无符号类型相同。每个无符号的类型名称都与带符号的类型名称相同，但要在前面加上关键字 unsigned。表 2-4 列出了可用的无符号整数类型。

表 2-4 无符号整数类型的名称

类 型 名 称	字 节 数	取 值 范 围
unsigned char	1	0~255
unsigned short int 或 unsigned short	2	0~65 535
unsigned int	4	0~4 294 967 295
unsigned long int 或 unsigned long	4	0~4 294 967 295
unsigned long long int 或 unsigned long long	8	0~18 446 744 073 709 551 615

www.TopSage.com

在处理不能为负的值时，使用无符号的整数类型，例如足球队的队员数，或者海滩上的鹅卵石数。如果位数给定，可以表示的数值就是固定的。32 位的变量可以表示 4 294 967 295 个不同的值。因此，使用无符号类型所提供的值不会多于对应的带符号类型，但其表示的数字比对应的带符号类型大一倍。

2.5.2 使用整数类型

在大多数情况下，int 或 long 类型的变量就能满足我们的需求，偶尔需要 unsigned int 或 unsigned long。下面是声明这些类型的例子：

```
unsigned int count = 10;
unsigned long inchesPerMile = 63360UL;
int balance = -500;
```

注意，long 类型的变量值尾部的 L，它表示 long 类型的常量，而不是 int 类型的常量，int 类型的常量不带后缀。unsigned long 类型的常量也有后缀 UL。本章后面的“指定整数常量”一节将介绍后缀。

int 变量类型的大小应最适合执行代码的计算机。例如下面的指令：

```
int cookies = 0;
```

这个语句声明的整数变量占据 4 个字节，但在另一个编译器上是 2 个字节。这似乎有点奇怪，但 int 类型要对应计算机上整数的大小，使计算机发挥其最大效能。这种字节数不仅在不同类型的机器上互不相同，甚至在相同的机器结构中，由于晶片技术的日新月异，也会出现不同。但最终由编译器确定类型的字节数。PC 的许多 C 编译器一度将 int 变量创建为 2 个字节，近来已普遍使用 4 个字节。这是因为现代的所有处理器在每次处理数据时，都至少使用 4 个字节。如果编译器比较旧，则即使硬件较适合使用 4 个字节，也可能给 int 类型使用 2 个字节。

注意：

这些类型的字节数取决于编译器。C 语言的国际标准要求，short 变量的字节数要小于等于 int 变量，int 变量的字节数要小于等于 long 变量。

如果使用 short 类型，就会得到 2 个字节的变量。之前的声明可以编写成：

```
short cookies = 0;
```

关键字 short 是 short int 的缩写，所以可以编写如下语句：

```
short int cookies = 0;
```

这和上面的语句相同。在变量声明中只编写 short，其实隐含了 int。大多数人喜欢使用这种形式，因为这比较清楚，还可以减少输入量。

www.TopSage.com

注意：

在一些机器上，short 和 int 类型占用相同的内存空间， 但它们是不同的类型。

如果需要取值范围较大的整数，例如存储一天平均卖出的汉堡数，就可以使用关键字 long：

```
long Big_Number;
```

类型 long 定义了一个 4 字节整数变量，它的取值范围是从–2 147 438 648 到 +2 147 438 647。如前所示，可以用 long int 代替 long，它们是相同的。

2.5.3 指定整数常量

整数变量有不同的类型，整数常量也有不同的类型。例如，如果将整数写成 100，它的类型就是 int。如果要确保它是 long 类型，就必须在这个数值的后面加上一个大写 L 或小写 l。所以，long 类型的整数 100 应写为 100L。虽然写为 100l 也是合法的，但应尽量避免，因为小写字母 l 与数字 1 很难辨别。

声明并初始化 Big_Number 的语句如下：

```
long Big_Number = 1287600L;
```

如果整数常量的数值超过类型 int 的范围，它的类型就是 long。因此，如果编译器用 2 个字节存储 int 值，数值 1000000 和 33000 的类型就默认为 long，因为它们在 2 个字节中放不下。

负整数常量的定义要用负号，例如：

```
int decrease = -4;
long below_sea_level = -100000L;
```

将整数常量指定为 long long 类型时，应添加两个 L：

```
long long ready_big_number = -123456789LL;
```

如前所述，将常量指定为无符号类型时，应添加 U，如下所示：

```
unsigned int count = 100U;
unsigned long value = 99999999UL;
```

也可以用十六进制编写整数，也就是说是以 16 为基底。十六进制的数字等价于十进制的 0~15，表示方式是 0~9 和 A~F(或 a~f)。因为需要一种方式区分十进制的 99 和十六进制的 99，所以就在十六进制数的前面加上 0x 或 0X。因此在程序中，十六进制的 99 可以编写成 0x99 或 0X99。

十六进制常量常用来表示位模式，因为每一个十六进制的数对应于 4 个二进制位。第 3 章介绍的按位运算符一般与十六进制常量一起用于定义掩码。如果不熟悉十六进制，可以参阅附录 A。

www.TopSage.com

注意：

以 0 开头的整数常量，例如 014，会被编译器看作八进制数——以 8 为基底。因此，014 等价于十进制的 12，而不是十进制的 14。所以，不要在整数中加上前导 0，除非要指定八进制数。

2.6 浮点数

浮点变量用来存储浮点数。浮点数包含的值带小数点，也可以表示分数和整数。下面是浮点数的例子：

```
1.6       0.00008        7655.899
```

由于浮点数的表示方式，它的位数是固定的。然而它的取值范围要比整数大得多。浮点数通常表示为一个小数值乘以 10 的次方。例如前面的每一个浮点数都可以采用表 2-5 的方式来表示。

表 2-5 浮点数表示法

数 值	使用指数表示法	在 C 语言中也可以写成
1.6	0.16×10^{1}	0.16E1
0.00008	0.8×10^{-4}	0.8E-4
7655.899	0.7655899×10^{4}	0.7655899E4

中间列说明，左列的数如何用指数表示法来表示，但在 C 语言中不使用这种方式：中间列采用一个替代方法来表示这些数值，以与右列对应。右列说明了中间列的数字在 C 语言中的表示法。这些数字中的 E 表示指数，也可以使用小写 e。当然在程序里编写这些数字时可以不用指数，而使用左列的方式，但对于非常大或非常小的数字，指数形式比较方便。0.5E-15 当然比 0.0 000 000 000 000 005 更好。

2.7 浮点数变量

表 2-6 是 3 种不同的浮点数变量。

表 2-6 浮点数变量类型

关 键 字	字 节 数	数 值 范 围
float	4	±3.4E38(精确到 6 位小数)
Double	8	±1.7E308(精确到 15 位小数)
long double	12	±1.19E4932(精确到 18 位小数)

www.TopSage.com

这些数所占用的字节数和取值范围与整数一样，取决于机器和编译器。在一些编译器上，类型 long double 和 double 相同。注意，小数的精确位数只是一个大约的数，因为浮点数在内部是以 2 进制方式存储的，十进制的浮点数在二进制中并不总是有精确的表示形式。

浮点数变量的声明方式和整数变量类似。只需给浮点数类型使用对应的关键字即可：

```
float Radius;
double Biggest;
```

如果需要存储至多有 7 位精确值的数(范围从 10^{-38} 到 10^{+38})，就应需要使用 float 类型的变量。类型 float 的值称为单精度浮点数。从表 2-6 中得知，它占用 4 个字节。使用类型 double 的变量可以存储双精度浮点数。类型 double 的变量占用 8 个字节，有 15 位精确值，范围从 10^{-308} 到 10^{+308}。它足以满足大多数的需求。但某些特殊的应用程序需要更精确、更大的范围，此时可以使用 long double。

编写一个类型为 float 的常量，需要在数值的末尾添加一个 f，以区别 double 类型。用下面的语句初始化前面的两个变量：

```
float Radius = 2.5f;
double Biggest = 123E30;
```

变量 Radius 的初值是 2.5，变量 Biggest 初始化为 123 后面加 30 个零。任何数，只要有小数点，就是 double 类型，除非加了 f，使它变为 float 类型。当用 E 或 e 指定指数值时，这个常量就不需要包含小数点。例如 1E3f 是 float 类型，3E8 是 double 类型。

要声明 long double 类型的常量，需要在数字的末尾添加一个大写 L 或小写 l，例如：

```
long double huge = 1234567.89123L;
```

2.8 使用浮点数完成除法运算

如前所见，除法运算使用的是整数操作数时，通常会得到整数结果。除非除法运算的左操作数刚好是右操作数的整数倍，否则其结果是不正确的。当然，在将饼干分给孩子们的例子中，整数除法运算的方式是没问题的，但将 10 尺长的厚板均分成 4 块时，就有问题了。这时就需要用到浮点数了。

使用浮点数进行除法运算，会得到正确的结果——至少是一个精确到固定位数的值。下一个例子说明如何使用 float 类型的变量进行除法运算。

试试看：使用 float 类型值的除法

这个例子用一个浮点数除以另一个浮点数，然后显示其结果：

```
/* Program 2.7 Division with float values */
#include <stdio.h>

int main(void)
```

www.TopSage.com

```
{
    float plank_length = 10.0f; /* In feet                      */
    float piece_count = 4.0f;   /* Number of equal pieces       */
    float piece_length = 0.0f;  /* Length of a piece in feet */

    piece_length = plank_length/piece_count;
    printf("A plank %f feet long can be cut into %f pieces %f feet long.",
            plank_length, piece_count, piece_length);
    return 0;
}
```

程序的结果输出如下:

```
A plank 10.000000 feet long can be cut into 4.000000 pieces 2.500000 feet
long.
```

代码的说明

如何平均切割木板是很容易理解的。注意，在 printf()语句里为 float 类型的值使用了新的格式指定符。

```
printf("A plank %f feet long can be cut into %f pieces %f feet long.",
        plank_length, piece_count, piece_length);
```

使用格式指定符%f 显示浮点数。格式指定符一般必须对应输出的值的类型。如果使用格式指定符%d 输出 float 类型的值，就会得到一个垃圾值。因为浮点数会解释为整数。同样，如果使用%输出整数类型的值，也会得到垃圾值。

2.8.1 控制小数位数

在上个例子的输出中有太多不必要的 0。擅长使用量尺和锯子，并不说明能用长度为 2.500000 量尺切割木板，更不用说用 2.500001 长度的量尺了。可以用格式指定符指定小数点后面的位数。例如，要使输出的小数点后有两位数，就可以使用格式指定符%.2f。如果小数点后需要有 3 位数，则可以使用%.3f。

可以修改上一个例子中的 printf()语句，生成更适当的结果：

```
printf("A plank %.2f feet long can be cut into %.0f pieces %.2f feet long.",
plank_length, piece_count, piece_length);
```

第一个格式指定符对应于变量 plank_length，其结果的小数点后有两位数。第二个格式指定符指定小数点后没有数字，这很合理，因为 piece_count 是整数。最后一个格式指定符和第一个相同。因此执行这个版本的例子，输出如下：

```
A plank 10.00 feet long can be cut into 4 pieces 2.50 feet long.
```

这样看起来舒服多了。

www.TopSage.com

2.8.2 控制输出的字段宽度

输出的字段宽度是输出值所使用的总字符数(包括空格)，它一般是默认的。printf()函数确定了输出值需要占用多少个字符位置，小数点后的位数由我们指定，并将它用作字段宽度。但我们可以自己确定字段宽度。如果要求输出一列排列整齐的数值，就应确定其字段宽度。如果让 printf()函数指定字段宽度，输出的数字列就不整齐。用于浮点数的格式指定符的一般形式是：

```
%[width][.precision][modifier]f
```

其中，方括号不包含在格式指定符中。它们包含的内容是可选的，所以可省略 width、.precision 或 modifier，或它们的任意组合。width 值是一个整数，指定输出的总字符数。precision 值也是一个整数，指定小数点后的位数。当输出值的类型是 long double 时，modifier 部分就是 L，否则就省略它。

可以重写上个例子的 printf()调用，指定字段宽度及小数点后的位数，例如：

```
printf("A %8.2f plank foot can be cut into %5.0f pieces %6.2f feet long.",
            plank_length, piece_count, piece_length);
```

上面的代码略微修改了文本，使之能放在书页上。现在，第一个值的字段宽度为 8，小数点后有 2 位数。第二个值是切割的总片数，其字段宽度为 5 个字符，且没有小数部分。第三个值的字段宽度为 6，小数点后有 2 位数。

指定字段宽度时，数值默认为右对齐。如果希望数值左对齐，只需在%的后面添加一个负号。例如，格式指定符%-10.4f 将输出一个左对齐的浮点数，其字段宽度为 10 个字符，小数点后有 4 位数。

注意，也可以对整数值指定字段宽度及对齐方式。例如%-15d 指定一个整数是左对齐，其字段宽度为 15 个字符。

还有其他格式指定符，以后会学习它们。用前面的例子试试各种不同的输出，尤其是看看字段宽度太小时会出现什么情况。

2.9 较复杂的表达式

算术要比两个数相除复杂得多。事实上，如果要进行复杂的算术运算，也可以使用笔和纸。有了加减乘除的工具，就可以开始进行一些真正复杂的计算了。

对于较复杂的的计算，需要更多地控制表达式的计算顺序。括号可以提供这方面的能力。当遇到错综复杂的情况时，括号还有助于使表达式更清晰。

在算术表达式中可以使用括号，其使用次数不受限制。包含在括号中的子表达式(Subexpressions)的计算顺序是：从最内层的括号开始计算到最外层的括号，对于运算符的优先级，一般规则是先乘除后加减。因此，表达式 2*(3+3*(5+4))的值是 60。首先计算表达式 5+4，得到 9。然后乘以 3，得到 27。之后加上 3，得到 30，最后乘以 2，得到 60。

www.TopSage.com

可以加入空格，将操作数和运算符分开，使算术表达式的可读性更高。需要使代码更紧凑时，则可以删除空格。无论采用哪种方式，编译器都不会受到影响，因为编译器会忽略空格。如果根据优先级规则，无法确定表达式的计算顺序，通常可以加进一些括号，确保生成需要的结果。

试试看：算术运算

这次要利用输入的直径计算一个圆桌的周长及面积。计算圆的周长及面积时，其数学公式要使用π或 pi(周长=2πr，面积=πr^2，其中 r 是半径)。如果不记得这些公式，也不用担心。这不是数学课本，所以只要理解程序是如何运作的即可。

```
/* Program 2.8 calculations on a table */
#include <stdio.h>

int main(void)
{
    float radius = 0.0f;          /* The radius of the table          */
    float diameter = 0.0f;        /* The diameter of the table        */
    float circumference = 0.0f;   /* The circumference of the table   */
    float area = 0.0f;            /* The area of a circle */
    float Pi = 3.14159265f;

    printf("Input the diameter of the table:");
    scanf("%f", &diameter);          /* Read the diameter from the keyboard */
    radius = diameter/2.0f;          /* Calculate the radius          */
    circumference = 2.0f*Pi*radius; /* Calculate the circumference  */
    area = Pi*radius*radius;         /* Calculate the area */
    printf("\nThe circumference is %.2f", circumference);
    printf("\nThe area is %.2f\n", area);
    return 0;
}
```

这个程序的输出如下：

```
Input the diameter of the table: 6

The circumference is 18.85
The area is 28.27
```

代码的说明

在第一个 printf()之前，这个程序看起来和以前的例子很类似：

```
float radius = 0.0f;          /* The radius of the table          */
float diameter = 0.0f;        /* The diameter of the table        */
float circumference = 0.0f;   /* The circumference of the table   */
float area = 0.0f;            /* The area of a circle             */
float Pi = 3.14159265f;
```

上述语句声明并初始化了 5 个变量，其中 Pi 有固定的数值。注意，所有的初值都在末尾添加了 f，因为这是 float 类型的初值。若没有 f 的话，它们的类型就是 double。不

www.TopSage.com

过在这里，它们仍然可行，但是编译器需要进行一些不必要的转换，将类型 double 转换为类型 float。

下一个语句输出一个从键盘上输入数据的提示:

```
printf("Input the diameter of the table: ");
```

下一个语句读取圆桌的直径。这需要使用一个新的标准库函数 scanf():

```
scanf("%f", &diameter); /* Read the diameter from the keyboard */
```

scanf()是另一个需要包含头文件 stdio.h 的函数。它专门处理键盘输入，提取通过键盘输入的数据，按照第一个参数指定的方式解释它，第一个参数是放在双引号内的一个控制字符串。在这里，这个控制字符串是%f。因为读取的值是 float 类型。scanf()将这个数存入第二个参数指定的变量 diameter 中。第一个参数是一个控制字符串，和 printf()函数的用法类似，但它控制的是输入，而不是输出。第 10 章将详细介绍 scanf()函数，附录 D 总结了所有的控制字符串。

注意，变量名 diameter 前的&是个新东西，它称为寻址运算符，它允许 scanf()函数将读入的数值存进变量 diameter 中。它的做法和将参数值传给函数是一样的。这里不详细解释它；第 8 章会详细说明。唯一要记住的是，使用函数 scanf()时，要在变量前加上寻址运算符&，而使用 printf()函数时不添加它。

在函数 scanf()的控制字符串中，%字符表示某数据项的格式指定符的开头。%字符后面的 f 表示输入一个浮点数。在控制字符串中一般有几个格式指定符，它们按顺序确定了函数中后面各参数的数据类型。本书的后面将介绍 scanf()函数的更多运用，下面的表 2-7 列出了读取各种类型的数据时所使用的格式指定符:

表 2-7 读取数据的格式指定符

操　作	需要的控制字符串
读取 short 类型的数值	%hd
读取 int 类型的数值	%d
读取 long 类型的数值	%ld
读取 float 类型的数值	%f 或%e
读取 double 类型的数值	%lf 或%le

在%ld 和%lf 格式指定符中，l 是小写的 L。别忘了一定要在接收输入值的变量名前加上&。另外，如果使用了错误的格式指定符，如使用%d 读取 float 类型的数据，变量中的数值就不正确，但系统不会提示存储了一个垃圾值。

接下来的 3 个语句计算结果:

```
radius = diameter/2.0f;          /* Calculate the radius          */
circumference = 2.0f*Pi*radius; /* Calculate the circumference */
area = Pi*radius*radius;         /* Calculate the area            */
```

www.TopSage.com

第一个语句计算半径，将输入的直径除以 2。第二个语句用计算出来的半径计算桌子的周长。第三个语句计算面积。注意，如果忘了 2.0f 中的 f，编译器就会显示一个警告消息。这是因为如果没有 f，常量的类型就是 double，于是在一个表达式里混用了不同的类型。后面会详细描述这个问题。

后面的两个语句输出计算后的数值:

```
printf("\nThe circumference is %.2f", circumference);
printf("\nThe area is %.2f\n", area);
```

这两个 printf()语句用格式指定符%.2f 输出变量 circumference 和 area 的值。如前所述，这两个语句里的格式控制字符串包含了要显示的文本，以及用于变量输出的格式指定符。这个格式指定符指定输出的值在小数点后面有两位数。默认的字段宽度足以容纳要显示的变量值。

当然，可以执行这个程序，给直径输入任意值。试着输入各种不同形式的浮点数，例如输入 1E1f。

2.10 定义常量

前面的例子将 Pi 定义为变量，但它是一个不会改变的常量，π的值是一个不循环的无限小数，其值总是固定不变。唯一的问题是，在指定它时精确到几位数。最好确保它的值在程序里保持不变，使之不会因错误而改变。

这有两种方法。第一是将Pi定义为一个符号，在程序编译期间用π的值取代它。此时，Pi不是一个变量，而是它表示的值的一个别名。

试试看：定义一个常量

下面将 PI 指定为一个数值的别名:

```
/* Program 2.9 More round tables */

#include <stdio.h>
#define PI 3.14159f /* Definition of the symbol PI */

int main(void)
{
  float radius = 0.0f;
  float diameter = 0.0f;
  float circumference = 0.0f;
  float area = 0.0f;

  printf("Input the diameter of the table:");
  scanf("%f", &diameter);
  radius = diameter/2.0f;
  circumference = 2.0f*Pi*radius;
  area = Pi*radius*radius;
```

www.TopSage.com

```
    printf("\nThe circumference is %.2f", circumference);
    printf("\nThe area is %.2f ", area);
    return 0;
}
```

这个输出和前面的例子完全相同。

代码的说明

在注释和头文件的#include 指令之后，有一个预处理指令:

```
#define PI 3.14159f      /*定义符号 PI*/
```

这里将 PI 定义为一个要被 3.14159f 取代的符号。使用 PI 而不是 Pi，是因为在 C 语言中有一个通用的约定：#define 语句中的标识符都是大写。只要在程序里的表达式中引用 PI，预处理器就会用#define 指令中的数值取代它。所有的取代动作都在程序编译之前完成。程序开始编译时，不再包含 PI 这个符号了，因为所有的 PI 都用#define 指令中的数值取代了。这些动作都是在编译器处理时在内部发生的，源程序没有改变，仍包含符号 PI。

第二种方法是将 Pi 定义成变量，但告诉编译器，它的值是固定的，不能改变。声明变量时，在变量名前加上 const 关键字，可以固化变量的值，例如：

```
const float Pi = 3.14159f; /*定义 Pi 的值是固定不变的*/
```

以这种方式定义 Pi 的优点是，Pi 现在定义为一个常量值。在前面的例子中，PI 只是一个字符序列，代码中的所有 PI 都会被替代。

在 Pi 的声明中添加关键字 const，会使编译器检查代码是否试图改变它的值。这么做的代码会被标记为错误，且编译失败。下面是它的一个例子。

试试看：定义一个其值固定的变量

在前面的例子中使用一个常量，但代码短一些:

```
 /*Program 2.10 Round tables again but shorter */
#include <stdio.h>

int main(void)
{
  float diameter = 0.0f;        /* The diameter of a table */
  float radius = 0.0f;          /* The radius of a table */
  const float Pi = 3.14159f;    /* Defines the value of Pi as fixed */

  printf("Input the diameter of the table: ");
  scanf("%f", &diameter);
  radius = diameter/2.0f;
  printf("\nThe circumference is %.2f", 2.0f*Pi*radius);
  printf("\nThe area is %.2f", Pi*radius*radius);
  return 0;
}
```

www.TopSage.com

代码的说明

下面是 Pi 变量的声明：

```
const float Pi = 3.14159f; /* Defines the value of Pi as fixed */
```

这个语句声明了变量 Pi，并给它定义一个数值；Pi 在这里还是变量，但它的初始值是不可改变的。这是 const 修饰符的功劳。它可应用在声明任何类型的变量的语句中，固化该变量的值。当然，变量值必须出现在声明语句中：在变量名称后、等号前。编译器会检查代码是否试图去改变声明为 const 的变量，如果发现有这种情况，编译器就会做出提示。可以设法骗过编译器，去改变 const 变量，但这违反了使用 const 的初衷。

下面两个语句输出程序的结果：

```
printf("\nThe circumference is %.2f", 2.0f*Pi*radius);
printf("\nThe area is %.2f", Pi*radius*radius);
```

在这个例子中，不再用变量存储周长及面积。现在这些表达式显示为 printf()函数的参数，它们的值会直接传给函数 printf()。

如前所述，传给函数的值可以是表达式的计算结果，而不是一个变量的值。编译器会创建一个临时变量，来存储这个值，再传给函数。之后这个临时变量就被删除。这很好，只要不在其他地方使用这些数值即可。

2.10.1 极限值

当然，一定要确定程序中给定的整数类型可以存储的极限值。头文件<limits.h>定义的符号表示每种类型的极限值。表 2-8 列出了对应于每种带符号类型的极限值符号名。

表 2-8 表示整数类型的极限值的符号

类 型	下 限	上 限
char	CHAR_MIN	CHAR_MAX
short	SHRT_MIN	SHRT_MAX
int	INT_MIN	INT_MAX
long	LONG_MIN	LONG_MAX
long long	LLONG_MIN	LLONG_MAX

无符号整数类型的下限都是 0，所以它们没有特定的符号。无符号整数类型的上限的符号分别是 UCHAR_MAX、USHRT_MAX、UINT_MAX、ULONG_MAX 和 ULLONG_MAX。

要在程序中使用这些符号，必须在源文件中添加<limits.h>头文件的#include 指令：

```
#include <limits.h>
```

可以用最大值初始化一个变量，如下所示：

www.TopSage.com

```
int number = INT_MAX;
```

这个语句把 number 的值设置为最大值，编译器会利用该最大值编译代码。

<float.h>头文件定义了表示浮点数的符号，其中一些的技术含量很高，所以这里只介绍我们感兴趣的符号。3 种浮点数类型可以表示的最大正值和最小正值如表 2-9 所示。

还可以使用 FLT_DIG、DBL_DIG 和 LDBL_DIG 符号，它们指定了对应类型的二进制尾数可以表示的小数位数。

下面用一个例子来说明如何使用表示整数和浮点数的符号。

表 2-9 表示浮点数类型的极限值的符号

类 型	下 限	上 限
float	FLT_MIN	FLT_MAX
double	DBL_MIN	DBL_MAX
long double	LDBL_MIN	LDBL_MAX

试试看：找出极限值

这个程序输出头文件中定义的符号的对应值。

```
/* Program 2.11 Finding the limits */
#include <stdio.h>   /* For command line input and output */
#include <limits.h>  /* For limits on integer types */
#include <float.h>   /* For limits on floating-point types */

int main(void)
{
  printf("Variables of type char store values from %d to %d",
            CHAR_MIN, CHAR_MAX);
  printf("\nVariables of type unsigned char store values from 0 to %u",
            UCHAR_MAX);
  printf("\nVariables of type short store values from %d to %d",
            SHRT_MIN, SHRT_MAX);
  printf("\nVariables of type unsigned short store values from 0 to %u",
            USHRT_MAX);
  printf("\nVariables of type int store values from %d to %d", INT_MIN,
            INT_MAX);
  printf("\nVariables of type unsigned int store values from 0 to %u",
            UINT_MAX);
  printf("\nVariables of type long store values from %ld to %ld",
            LONG_MIN, LONG_MAX);
  printf("\nVariables of type unsigned long store values from 0 to %lu",
            ULONG_MAX);
  printf("\nVariables of type long long store values from %lld to %lld",
            LLONG_MIN, LLONG_MAX);
  printf("\nVariables of type unsigned long long store values from 0 to
            %llu",ULLONG_MAX);
```

www.TopSage.com

```
    printf("\n\nThe size of the smallest non-zero value of type float is
            %.3e",FLT_MIN);
    printf("\nThe size of the largest value of type float is %.3e", FLT_MAX);
    printf("\nThe size of the smallest non-zero value of type double is %.3e",
            DBL_MIN);
    printf("\nThe size of the largest value of type double is %.3e", DBL_MAX);
    printf("\nThe size of the smallest non-zero value ~CCC
            of type long double is %.3Le", LDBL_MIN);
    printf("\nThe size of the largest value of type long double is %.3Le\n",
            LDBL_MAX);
    printf("\nVariables of type float provide %u decimal digits precision.",
            FLT_DIG);
    printf("\nVariables of type double provide %u decimal digits
            precision.",DBL_DIG);
   printf("\nVariables of type long double provide %u decimal digits
            precision.",LDBL_DIG);
    return 0;
}
```

结果如下所示:

```
Variables of type char store values from -128 to 127
Variables of type unsigned char store values from 0 to 255
Variables of type short store values from -32768 to 32767
Variables of type unsigned short store values from 0 to 65535
Variables of type int store values from -2147483648 to 2147483647
Variables of type unsigned int store values from 0 to 4294967295
Variables of type long store values from -2147483648 to 2147483647
Variables of type unsigned long store values from 0 to 4294967295
Variables of type long long store values ~CCC
from -9223372036854775808 to 9223372036854775807
Variables of type unsigned long long store values from 0 to
18446744073709551615

The size of the smallest non-zero value of type float is 1.175e-038
The size of the largest value of type float is 3.403e+038
The size of the smallest non-zero value of type double is 2.225e-308
The size of the largest value of type double is 1.798e+308
The size of the smallest non-zero value of type long double is 3.362e-4932
The size of the largest value of type long double is 1.190e+4932

Variables of type float provide 6 decimal digits precision.
Variables of type double provide 15 decimal digits precision.
Variables of type long double provide 18 decimal digits precision.
```

代码的说明

在一系列的 printf()函数调用中，输出<limits.h>和<float.h>头文件定义的符号的值。计算机中的数值总是受限于该机器可以存储的值域，这些符号的值表示每种数值类型的极限值。这里用指定符%u 输出无符号整数值。如果用%d 输出无符号类型的最大值，则

www.TopSage.com

最左边的位(带符号类型的符号位)为 1 的数值就得不到正确的解释。

对浮点数的极限值使用指定符%e，表示这个数值是指数形式。同时指定精确到小数点后的 3 位数，因为这里的输出不需要非常精确。printf()函数显示的值是 long double 类型时，需要使用 L 修饰符。L 必须是大写，这里没有使用小写字母 l。%f 指定符表示没有指数的数值，它对于非常大或非常小的数来说相当不方便。在这个例子中试一试，就会明白其含义。

2.10.2 sizeof 运算符

使用 sizeof 运算符可以确定给定的类型占据多少字节。当然，在 C 语言中 sizeof 是一个关键字。表达式 sizeof(int)会得到 int 类型的变量所占的字节数，所得的值是一个 size_t 类型的整数。size_t 类型在标准头文件<stddef.h>(和其他头文件如<stdio.h>)中定义，对应于一个基本整数类型。但是，与 size_t 类型对应的类型可能在不同的 C 库中有所不同，所以最好使用 size_t 变量存储 sizeof 运算符生成的值，即使知道它对应的基本类型，也应如此。下面的语句是存储用 sizeof 运算符计算所得的数值：

```
size_t size = sizeof(long long);
```

也可以将 sizeof 运算符用于表达式，其结果是表达式的计算结果所占据的字节数。通常该表达式是某种类型的变量。除了确定某个基本类型的值占用的内存空间之外，sizeof 运算符还可以确定每种类型占用的字节数。

试试看：确定给定类型占用的字节数

这个程序会输出每个数值类型占多少字节：

```
/* Program 2.12 Finding the size of a type */
#include <stdio.h>

int main(void)
{
  printf("\nVariables of type char occupy %d bytes", sizeof(char));
  printf("\nVariables of type short occupy %d bytes", sizeof(short));
  printf("\nVariables of type int occupy %d bytes", sizeof(int));
  printf("\nVariables of type long occupy %d bytes", sizeof(long));
  printf("\nVariables of type float occupy %d bytes", sizeof(float));
  printf("\nVariables of type double occupy %d bytes", sizeof(double));
  printf("\nVariables of type long double occupy %d bytes",
         sizeof(long double));

  return 0;
}
```

输出如下：

```
Variables of type char occupy 1 bytes
Variables of type short occupy 2 bytes
```

www.TopSage.com

```
Variables of type int occupy 4 bytes
Variables of type long occupy 4 bytes
Variables of type float occupy 4 bytes
Variables of type double occupy 8 bytes
Variables of type long double occupy 12 bytes
```

代码的说明

因为 sizeof 运算符的结果是一个整数，所以用%d 指定符输出它。注意，使用表达式 sizeof var_name 也可以得到变量 var_name 占用的字节数。显然，在关键字 sizeof 和变量名之间的空格是必不可少的。

现在已经知道编译器给每个数值类型指定的极限值和占用的字节数了。

2.11 选择正确的类型

必须仔细选择在计算过程中使用的变量类型，使之能包含我们期望的值。如果使用了错误的类型，程序就可能出现很难检测出来的错误。这最好用一个例子来说明。

试试看：变量的正确类型

下面的例子说明，如果给变量选择了不适当的类型，程序就会出错。

```
/* Program 2.13 Choosing the correct type for the job 1*/
#include <stdio.h>

int main(void)
{
  const float Revenue_Per_150 = 4.5f;
  short JanSold = 23500;    /* Stock sold in January */
  short FebSold = 19300;    /* Stock sold in February */
  short MarSold = 21600;    /* Stock sold in March */
  float RevQuarter = 0.0f; /* Sales for the quarter */

  short QuarterSold = JanSold+FebSold+MarSold;
            /* Calculate quarterly total */

  /* Output monthly sales and total for the quarter */
  printf("\nStock sold in\n Jan: %d\n Feb: %d\n Mar: %d",
       JanSold,FebSold,MarSold);
  printf("\nTotal stock sold in first quarter: %d",QuarterSold);

  /* Calculate the total revenue for the quarter and output it */
  RevQuarter = QuarterSold/150*Revenue_Per_150;
  printf("\nSales revenue this quarter is:$%.2f\n",RevQuarter);
  return 0;
}
```

这些都是相当简单的计算，一季度的总销售量应是 64 400，它只是将每个月的销售

www.TopSage.com

量加在一起。但运行这个程序，输出如下：

```
Stock sold in
Jan: 23500
Feb: 19300
Mar: 21600
Total stock sold in first quarter: -1136
Sales revenue this quarter is:$-31.50
```

显然，结果不正确。把 3 个较大的正数加在一起，不应得到一个负值。

代码的说明

首先，代码定义了一个要在计算中使用的常量：

```
const Revenue_Per_150 = 4.5f;
```

这个语句定义了每销售 150 个产品的收入。这没有什么错误。接着，声明 4 个变量，并给它们赋予初值：

```
short JanSold = 23500;    /* Stock sold in January */
short FebSold = 19300;    /* Stock sold in February */
short MarSold = 21600;    /* Stock sold in March */
float RevQuarter = 0.0f; /* Sales for the quarter */
```

前 3 个变量的类型是 short，足以存储初值了。RevQuarter 变量是 float 类型，因为我们希望季度收入在小数点后有两位数。

下一个语句声明 QuarterSold 变量，并存储每月销售量的总和：

```
short QuarterSold = JanSold+FebSold+MarSold;
       /* Calculate quarterly total */
```

注意，这个变量用表达式的结果初始化。这是可行的，因为编译器知道这些变量的值，所以这是一个常量表达式。如果表达式中的变量值要在程序执行过程中确定，例如要读取某个值，该程序就不会编译。编译器只能使用明确指定的初始值，或者编译器能计算的表达式的结果。

事实上，结果错误的原因是 QuarterSold 变量的声明错误。该变量声明为 short 类型，其初始值指定为 3 个月销售量的总和。这个总和是 64 400，而程序输出了一个负数。这个语句一定有错误。

问题的原因是，我们试图在 QuarterSold 变量中存储对 short 类型而言过大的数字。short 变量能存储的最大值是 32 767，计算机不能正确解释 QuarterSold 的值，所以输出了一个负值。这个问题的解决方法是使用 long 类型的变量来存储非常大的数字。

解决问题

修改程序，再次运行它。只需要修改 main()函数体中的两行代码。修改的新程序如下：

www.TopSage.com

```
/* Program 2.14 Choosing the correct type for the job 2 */
#include <stdio.h>

int main(void)
{
  const float Revenue_Per_150 = 4.5f;
  short JanSold =23500; /* Stock sold in January */
  short FebSold =19300; /* Stock sold in February */
  short MarSold =21600; /* Stock sold in March */
  float RevQuarter = 0.0f; /* Sales for the quarter */

  long QuarterSold = JanSold+FebSold+MarSold;
    /* Calculate quarterly total */

  /* Output monthly sales and total for the quarter */
  printf("Stock sold in\n Jan: %d\n Feb: %d\n Mar: %d\n",
        JanSold,FebSold,MarSold);
  printf("Total stock sold in first quarter: %ld\n",QuarterSold);

  /* Calculate the total revenue for the quarter and output it */
  RevQuarter = QuarterSold/150*Revenue_Per_150;
  printf("Sales revenue this quarter is:$%.2f\n",RevQuarter);
  return 0;
}
```

运行这个程序，这次的输出是正确的:

```
Stock sold in
Jan: 23500
Feb: 19300
Mar: 21600
Total stock sold in first quarter: 64400
Sales revenue this quarter is :$1930.50
```

一季度的销售量是正确的，收入也是正确的。注意，这里使用%ld 输出总销售量，这就告诉编译器，使用 long 类型输出这个值。检查程序，用计算器计算出收入。

得到的结果应是$1 932，少了$1.50，这个数字虽然不大，但对于会计而言，这就是一个错误，必须找到少了的$1.50。程序在计算收入值时，发生了什么?

```
RevQuarter = QuarterSold/150*Revenue_Per_150;
```

这个语句给 RevQuarter 赋值，该值是等号右边表达式的结果。根据本章前面介绍的优先级规则，一步步地计算该表达式。这是一个非常简单的表达式，只需从左向右计算，因为乘除的优先级相同。下面列出计算过程:

- QuarterSold/150 计算为 64400 / 150，结果应为 429.333。

这里有问题。QuarterSold 是一个整数，所以计算机将除法运算的结果四舍五入为一个整数，舍弃了.333。所以，在下一步计算中，结果会有出入。

www.TopSage.com

- 429*Revenue_Per_150 计算为 429 * 4.5，结果为 1930.50。

知道哪里有错误后，如何更正它？可以将所有的变量都改为浮点数类型，但这违背了使用整数的初衷。输入的数字是整数，所以将它们存储到整数变量中。有较简单的解决方法吗？当然有。重写如下的语句：

```
RevQuarter = Revenue_Per_150*QuarterSold/150;
```

这个语句先执行乘法运算，因为对混合的操作数执行算术运算时，编译器会自动把整数操作数转换为浮点数，所以结果是 float 类型。对该结果除以 150，该操作也在 float 值上执行，并将 150 转换为 150f。于是，结果就是正确的。

我们不仅需要理解在不同类型的操作数上如何执行算术运算，还要理解如何控制类型的转换。在 C 语言中，可以将一种类型显式地转换为另一种类型。这个过程称为强制类型转换。

2.12 强制类型转换

在程序 2.14 计算季度收入的表达式中，可以控制操作的执行，得到正确的结果：

```
RevQuarter = QuarterSold/150*Revenue_Per_150;
```

要使结果正确，必须修改这个语句，以浮点数的方式计算表达式。如果可以把 QuarterSold 的值转换为 float 类型，该表达式就会以浮点数的方式计算，问题就解决了。要把变量从一种类型转换为另一种类型，应把目标类型放在变量前面的括号中。因此，正确计算结果的表达式应如下所示：

```
RevQuarter = (float)QuarterSold/150.0f*Revenue_Per_150;
```

这就是我们需要的表达式：在正确的地方使用变量的正确类型。当希望保留除法结果的小数部分时，不应使用整数运算。一种类型显式转换为另一种类型的过程称为强制类型转换(cast)。

2.12.1 自动转换类型

该程序的第二个版本的输出如下：

```
Sales revenue this quarter is :$1930.50
```

即使表达式中没有显式转换类型，结果也是浮点数形式，但它仍是错误的。这是因为编译器在处理涉及不同类型的值操作时，会自动把其中一个操作数的类型转换为另一个操作数的类型。

只有两个操作数的类型相同，计算机才能执行二元算术操作(加、减、乘、除和取模)。在二元算术运算中使用不同类型的操作数，编译器就会把其中一个值域较小的操作数类型转换为另一个操作数的类型，这称为隐式类型转换(implicit conversion)。再看看前面计

www.TopSage.com

算收入的表达式：

```
QuarterSold / 150 * Revenue_Per_150
```

它计算为 64400 (int) / 150 (int),，结果是 429 (int)，再将 429(int 转换为 float)乘以 4.5 (float)，得到 1930.5 (float)。

当二元运算符处理不同类型的操作数时，总是会进行隐式类型转换。对于上述第一个操作，两个数字都是 int 类型，所以结果也是 int 类型。对于上述第二个操作，第一个值的类型是 int，第二个值的类型是 float。而 int 类型的值域小于 float 类型，所以自动将 int 类型的值转换为 float 类型。只要算术表达式中有混合类型的变量，C 编译器就会使用一组特殊的规则，确定表达式如何计算。下面就介绍这些规则。

2.12.2 隐式类型转换的规则

确定二元运算中的哪个操作数要转换为另一个操作数的类型时，其机制相当简单。其基本规则是，将值域较小的操作数类型转换为另一个操作数类型，但在一些情况下，两个操作数都要转换类型。

为了准确地表述这些规则，需要比上述更复杂的描述，所以可以忽略一些细节，在以后需要时再考虑它们。如果读者想了解全部规则，应继续阅读下去。

编译器按顺序采用如下规则，确定要使用的隐式类型转换：

(1) 如果一个操作数的类型是 long double，就把另一个操作数转换为 long double 类型。

(2) 否则，如果一个操作数的类型是 double，就把另一个操作数转换为 double 类型。

(3) 否则，如果一个操作数的类型是 float，就把另一个操作数转换为 float 类型。

(4) 否则，如果两个操作数的类型都是带符号的整数或无符号的整数，就把级别较低的操作数转换为另一个操作数的类型。无符号整数类型的级别从低到高为：

```
signed char, short, int, long, long long
```

每个无符号整数类型的级别都与对应的带符号整数类型相同，所以 unsigned int 类型的级别与 int 类型相同。

(5) 否则，如果带符号整数类型的操作数级别低于无符号整数类型的级别，就把带符号整数类型的操作数转换为无符号整数类型。

(6) 否则，如果带符号整数类型的值域包含了无符号整数类型所表示的值，就把无符号整数类型转换为带符号整数类型。

(7) 否则，两个操作数都转换为带符号整数类型对应的无符号整数类型。

2.12.3 赋值语句中的隐式类型转换

赋值运算符右边的表达式值与其左边的变量有不同的类型时，也可以进行隐式类型转换。在一些情况下，这会截短数值，丢失数据。例如，如果赋值操作将 float 或 double

www.TopSage.com

类型的值存储在 int 或 long 类型的变量中，float 或 double 的小数部分就会丢失，只存储整数部分。如下面的代码所示：

```
int number = 0;
float value = 2.5f;
number = value;
```

存储在 number 中的值是 2。这几行代码把 decimal 的值(2.5)赋予 int 类型的变量 number，就丢失了小数部分.5，只存储了 2。注意，在 2.5f 中使用了指定符 f。

赋值语句可能丢失信息，因为必须进行隐式类型转换，而编译器通常会为此发出一个警告。但是，代码仍可以编译，所以程序可能会得到不正确的结果。当需要在代码中进行可能导致丢失信息的类型转换时，最好使用显式类型转换。

下面的例子将说明赋值操作中的类型转换规则，代码如下：

```
double price = 10.0;      /* Product price per unit */
long count = 5L;          /* Number of items */
float ship_cost = 2.5F;   /* Shipping cost per order */
int discount = 15;        /* Discount as percentage */
long double total_cost =
          (count*price + ship_cost)*((100L - discount)/100.0F);
```

这些语句声明了 4 个变量，并根据给这些变量设置的值计算某个订单的总价。这里选择的类型主要用于演示隐式类型转换，它们不表示正常环境下的正确类型选择。下面看看最后一个语句如何计算 total_cost 的值：

(1) 先计算 count*price，再将 count 隐式转换为 double 类型，以进行乘法运算，结果是 double 类型，这源于第 2 个规则。

(2) 接着将 ship_cost 加到前一个操作的结果中。为此，要将 ship_cost 的值转换为前一个结果的类型 double。这个转换也源于第 2 个规则。

(3) 然后计算表达式 100L - discount，为此，要将 discount 的值转换为减法操作中另一个操作数的类型 long。这源于第 4 个规则，结果是 long 类型。

(4) 之后把上一个操作的结果(long 类型)转换为 float 类型，再除以 100.0F(float 类型)。这源于第 3 个规则，结果是 float 类型。

(5) 将第 2 步的结果除以第 4 步的结果，为此，要将上一个操作的 float 值转换为 double 类型，这源于第 3 个规则，结果是 double 类型。

(6) 最后，将上述结果存储在 total_cost 变量中，作为赋值操作的结果。当操作数的类型不同时，赋值操作总是要把右操作数的结果转换为左操作数的类型，所以上述操作的结果会转换为 long double 类型。编译器不会发出警告，因为 double 类型的所有值都可以表示为 long double 类型。

2.13 再谈数值数据类型

为了完整论述数值数据类型，下面讨论一些前面未提及的内容。第一个未涉及的类

www.TopSage.com

型是 char。char 类型的变量可以存储单个字符的代码。它只能存储一个字符代码(即一个整数)，所以被看作整数类型。可以像其他整数类型那样处理 char 类型存储的值，因此可以在算术运算中使用它。

2.13.1 字符类型

在所有数据的类型中，char 类型占用的内存空间最少。它一般只需一个字节。存储在 char 类型变量的整数可以表示为带符号或无符号的值，这取决于编译器。若表示为无符号的类型，则存储在 char 类型变量的值可以是 0～255。若表示为带符号的类型，则存储在 char 类型变量的值可以是 - 128～127。当然，这两个值域对应相同的位模式：0000 0000 到 1111 1111。对于无符号的值，这 8 位都是数据位，所以 0000 0000 对应于 0，1111 1111 对应于 255。对于带符号的值，最左边的 1 位是符号位，所以 - 128 的二进制值是 1000 0000，0 的二进制值是 0000 0000，127 的二进制值是 0111 1111。值 1111 1111 是一个带符号的二进制值，其对应的十进制值是 - 1。

从表示字符代码(位模式)的角度来看，char 类型是否带符号并不重要。重要的是何时对 char 类型的值执行算术运算。

char 变量可以包含任意单个字符，所以可以给 char 类型的变量指定字符常量，作为其初始值。字符常量是一个放在单引号中的字符。下面是一些例子：

```
char letter = 'A';
char digit = '9';
char exclamation = '!';
```

也可以使用转义序列指定字符常量，例如：

```
char newline = '\n';
char tab = '\t';
char single_quote = '\'';
```

当然，上面的每个语句都把变量设置为单引号内的字符代码。实际的代码值取决于计算机环境，但最常见的是美国标准信息交换码(ASCII)。ASCII 字符集参见附录 B。

还可以用整数值初始化 char 类型的变量，只要该值在编译器许可的 char 类型的值域内即可，如下面的例子：

```
char character = 74; /* ASCII code for the letter J */
```

char 类型的变量有双重性：可以把它解释为一个字符，也可以解释为一个整数。下面的例子对 char 类型的值进行算术运算：

```
char letter = 'C';    /* letter contains the decimal code value 67 */
letter = letter + 3;  /* letter now contains 70, which is 'F' */
```

因此，可以对 char 类型的值进行算术运算，同时仍把它当做一个字符。

www.TopSage.com

2.13.2 字符的输入输出

使用 scanf()函数和格式指定符%c，可以从键盘上读取单个字符，将它存储在 char 类型的变量中，例如：

```
char ch = 0;
scanf("%c", &ch); /* Read one character */
```

如前所述，在使用 scanf()函数的源文件中，必须给<stdio.h>头文件添加#include 指令：

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

要使用 printf()函数将单个字符输出到命令行上，也可以使用格式指定符%c：

```
printf("The character is %c", ch);
```

当然，也可以输出该字符的数值：

```
printf("The character is %c and the code value is %d", ch, ch);
```

这个语句会把 ch 的值输出为一个字符和一个数值。

试试看：字符的建立

编程新手可能想知道，计算机如何知道它处理的是字符还是整数？事实是计算机并不知道。这就好像 Alice 使用 Humpty Dumpty(矮胖的人)时，会说，“我使用这个单词时，就意味着我给它赋予了“矮胖的人”这个含义。”同样，内存中的一个数据项的含义是我们赋予它的。包含值 70 的字节是一个整数，把 70 看作字母 J 的代码也是正确的。

下面的例子会说明这一点。这个例子使用转换指定符%c，它指定将 char 类型的值输出为一个字符，而不是一个整数。

```
/* Program 2.15 Characters and numbers */
#include <stdio.h>

int main(void)
{
  char first = 'T';
  char second = 20;

  printf("\nThe first example as a letter looks like this - %c", first);
  printf("\nThe first example as a number looks like this - %d", first);
  printf("\nThe second example as a letter looks like this - %c", second);
  printf("\nThe second example as a number looks like this - %d\n", second);
  return 0;
}
```

这个程序的输出如下：

www.TopSage.com

```
The first example as a letter looks like this - T
The first example as a number looks like this - 84
The second example as a letter looks like this - ¶
The second example as a number looks like this - 20
```

代码的说明

这个程序首先声明了两个 char 类型的变量:

```
char first = 'T';
char second = 20;
```

把第一个变量初始化为一个字符处理，第二个变量初始化为一个整数。接下来的 4 个语句以两种方式输出每个变量的值:

```
printf("\nThe first example as a letter looks like this - %c",
       first_example);
printf("\nThe first example as a number looks like this - %d",
       first_example);
printf("\nThe second example as a letter looks like this - %c",
        second_example);
printf("\nThe second example as a number looks like this - %d\n",
        second_example);
```

%c 转换指定符将变量的内容解释为单个字符，%d 指定符把它解释为一个整数。输出的数值是对应字符的代码。这个例子中的这些代码都是 ASCII 码。在大多数情况下字符代码都是 ASCII 码，所以本书都使用 ASCII 码。

如前所述，并不是所有的计算机都使用 ASCII 字符集，所以可能会得到与上述不同的值。但只要给字符常量使用了符号字符，无论采用什么字符编码，都会得到所需的字符。

用格式指定符%x 替代%d，就可以把 char 类型变量的整数值输出为十六进制值。

试试看：用字符的对应整数值进行算术运算

下面的例子将算术运算应用于 char 类型的值:

```
/* Program 2.16 Using type char */
#include <stdio.h>

int main(void)
{
  char first = 'A';
  char second = 'B';
  char last = 'Z';

  char number = 40;

  char ex1 = first + 2;   /* Add 2 to 'A' */
  char ex2 = second - 1;  /* Subtract 1 from 'B' */
  char ex3 = last + 2;    /* Add 2 to 'Z' */
```

www.TopSage.com

```
    printf("Character values %-5c%-5c%-5c", ex1, ex2, ex3);
    printf("\nNumerical equivalents %-5d%-5d%-5d", ex1, ex2, ex3);
    printf("\nThe number %d is the code for the character %c\n",
           number, number);
    return 0;
}
```

运行这个程序，输出如下：

```
Character values       C     A     \
Numerical equivalents 67     65    92
The number 40 is the code for the character (
```

代码的说明

这个程序说明了如何对初始化为字符的 char 变量进行算术运算。main()函数体中的前 3 个语句如下：

```
char first = 'A';
char second = 'B';
char last = 'Z';
```

这些语句把变量 first、second 和 last 初始化为字符值。这些变量的数值是各个字符对应的 ASCII 码。它们可以看做数值和字符，所以可以对它们执行算术运算。

下一个语句用一个整数值初始化 char 类型的变量：

```
char number = 40;
```

初始值必须在单字节变量可以存储的值域内。对于笔者的编译器，char 是一个带符号的类型，所以其值必须在 - 128~127 之间。当然，也可以将该变量的内容解释为字符。在这个例子中，它是一个 ASCII 码为 40 的字符，即左括号。

接下来的 3 个语句又声明了 3 个 char 类型的变量：

```
char ex1 = first + 2;   /* Add 2 to 'A' */
char ex2 = second - 1;  /* Subtract 1 from 'B' */
char ex3 = last + 2;    /* Add 2 to 'Z' */
```

这些语句根据变量 first、second 和 last 中存储的值计算出新值，也就计算出了对应的新字符。这些表达式的结果存储在变量 ex1、ex2 和 ex3 中。

之后的两个语句以两种不同的方式输出 3 个变量 ex1、ex2 和 ex3：

```
printf("Character values %-5c%-5c%-5c", ex1, ex2, ex3);
printf("\nNumerical equivalents %-5d%-5d%-5d", ex1, ex2, ex3);
```

第一个语句使用%-5c 转换指定符把所存储的值解释为字符。它指定把值输出为字符，且左对齐，字符宽度为 5。第二个语句又输出了这些变量，但这次使用%-5d 指定符把这些值解释为整数。对齐方式和字符宽度与第一个语句相同，但%-5d 中的 d 指定输出是一个整数。在这两行输出中，第一行显示 3 个字符，第二行显示它们的

www.TopSage.com

ASCII 码。

最后一行代码将 number 变量输出为一个字符和一个整数:

```
printf("\nThe number %d is the code for the character %c\n", number,
        number);
```

变量要输出两次，只需编写两次即可——printf()函数的第二和第三个参数。它先输出一个整数，再输出一个字符。

对字符执行算术运算的功能是很有用的。例如，要把大写字母转换为小写，只需给大写字母加上 'a' -' A' 的结果(ASCII 码 32)即可。要把小写字母转换为大写，只需减去 'a' -' A'。附录 B 列出了字母字符的十进制 ASCII 值。当然，这个操作要求 a~z 和 A~Z 的字符代码是连续的整数。如果计算机使用的字符编码不是连续的整数，就不能这么做。一些 IBM 机器上使用的 EBCDIC 码就不能使用这个技巧，因为其字母的代码值不是连续的。

2.13.3 宽字符类型

wchar_t 类型的变量存储多字节字符码，一般占用两个字节。在处理 Unicode 字符时，就需要使用 wchar_t 类型。wchar_t 类型在标准头文件<stddef.h>中定义，所以需要在使用该类型的源文件中包含该头文件。在 char 类型的字符常量前面加上修饰符 L，就可以定义一个宽字符常量。例如，下面声明了一个 wchar_t 类型的变量，用大写字母 A 的代码初始化它:

```
wchar_t w_ch = L'A';
```

wchar_t 类型的操作与 char 类型的操作相同。wchar_t 类型是一个整数，所以可以对该类型的值进行算术运算。

要从键盘上把一个字符读入 wchar_t 类型的变量，可以使用%lc 格式指定符。使用这个格式指定符还可以输出 wchar_t 类型的值。下面的例子从键盘上读入一个字符，显示在下一行上:

```
wchar_t wch = 0;
scanf("%lc", &wch);
printf("You entered %lc", wch);
```

当然，这个代码段需要添加<stdio.h>的#include 指令，才能正确编译。

2.13.4 枚举

在编程时，常常希望变量存储一组可能值中的一个。例如一个变量存储表示当前月份的值。这个变量应只存储 12 个可能值中的一个，分别对应于 1~12 月。C 语言中的枚举(enumeration)就用于这种情形。

www.TopSage.com

利用枚举，可以定义一个新的整数类型，该类型变量的值域是我们指定的几个可能值。下面的语句定义了一个新的类型 Weekday：

```
enum Weekday {Monday, Tuesday, Wednesday,
              Thursday, Friday, Saturday, Sunday};
```

新类型的名称 Weekday 跟在关键字 enum 的后面，这个类型名称称为枚举的标记。Weekday 类型的变量值可以是类型名称后面的大括号中的名称指定的任意值。这些名称叫做枚举器(enumerator)或枚举常量(enumeration constant)，其数量可任意。每个枚举器都用我们赋予的唯一名称来指定，编译器会把 int 类型的整数值赋予每个名称。枚举是一个整数类型，因为指定的枚举器对应不同的整数值，这些整数默认从 0 开始，每个枚举器的值都比它之前的枚举器大 1。因此在这个例子中，Monday 到 Sunday 对应 0~6。

可以声明 Weekday 类型的一个新变量，并初始化它，如下所示：

```
enum Weekday today = Wednesday;
```

这个语句声明了一个变量 today，将它初始化为 Wednesday。由于枚举器有默认值，Wednesday 所以对应 2。用于枚举类型变量的整数类型是由实现代码确定的，选择什么类型取决于枚举器的个数。

也可以在定义枚举类型时，声明该类型的变量。下面的语句就定义了一个枚举类型和两个变量：

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday,
              Friday, Saturday, Sunday} today, tomorrow;
```

这个语句声明了枚举类型 Weekday，定义了该类型的两个变量 today 和 tomorrow。还可以在同一个语句中初始化变量，如下所示：

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday,
      Friday, Saturday, Sunday} today = Monday, tomorrow = Tuesday;
```

这个语句把变量 today 和 tomorrow 初始化为 Monday 和 Tuesday。枚举类型的变量是整数类型，所以可以在算术表达式中使用。前面的语句还可以写为：

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday,
     Friday, Saturday, Sunday} today = Monday, tomorrow = today + 1;
```

tomorrow 的初始值比 today 大 1。但是，在执行这个操作时，要确保算术运算的结果是一个有效的枚举值。

注意：

可以给枚举类型指定一组可能的值，但没有检查机制来确保程序只使用这些值。所以程序员要确保只为给定的枚举类型使用有效的枚举值。一种方式是只给枚举类型的变量赋予枚举常量名。

www.TopSage.com

1. 选择枚举值

可以给任意或所有枚举器明确指定自己的整数值。尽管枚举器使用的名称必须唯一，但枚举器的值不要求是唯一的。除非有特殊的原因让某些枚举器的值相同，否则一般应确保这些值也是唯一的。下面的例子定义了 Weekday 类型，使其枚举器的值从 1 开始：

```
enum Weekday {Monday=1, Tuesday, Wednesday,
        Thursday, Friday, Saturday, Sunday};
```

枚举器 Monday 到 Sunday 的对应值是 1～7。在明确指定了值的枚举器后面，枚举器会被赋予连续的整数值。这可能使枚举器有相同的值，如下面的例子所示：

```
enum Weekday {Monday=5, Tuesday=4, Wednesday,
     Thursday=10, Friday =3, Saturday, Sunday};
```

Monday、Tuesday、Thursday 和 Friday 明确指定了值，Wednesday 设置为 Tuesday+1，所以它是 5，Monday 与它相同。同样，Saturday 和 Sunday 设置为 4 和 5，所以它们的值也是重复的。完全可以这么做，但除非有很好的理由使一些枚举常量的值相同，否则这容易出现混淆。

只要希望变量有限定数量的可能值，就可以使用枚举。下面是定义枚举的另一个例子：

```
enum Suit{clubs = 10, diamonds, hearts, spades);
enum Suit card_suit = diamonds;
```

第一个语句定义了枚举类型 Suit，这个类型的变量可以有括号中的 4 个值的任意一个。第二个语句定义了 Suit 类型的一个变量，把它初始化为 diamonds，其对应的值是 11。还可以定义一个枚举，表示扑克牌的面值，如下所示：

```
enum FaceValue { two=2, three, four, five, six, seven,
       eight, nine, ten, jack, queen, king, ace};
```

在这个枚举中，枚举器的整数值匹配扑克牌的面值，其中 ace 的值最高。

在输出枚举类型的变量值时，会得到数值。如果要输出枚举器的名称，必须提供相应的程序逻辑，详见下一章的内容。

2. 未命名的枚举类型

在创建枚举类型的变量时，可以不指定标记，这样就没有枚举类型名了。例如：

```
enum {red, orange, yellow, green, blue, indigo, violet} shirt_color;
```

这里没有标记，所以这个语句定义了一个未命名的枚举类型，其可能的枚举器包括从 red 到 violet。该语句还声明了未命名类型的变量 shirt_color。

可以用通常的方式给 shirt_color 赋值：

```
shirt_color = blue;
```

www.TopSage.com

显然，未命名枚举类型的主要限制是，必须在定义该类型的语句中声明它的所有变量。由于没有类型名，因此无法在代码的后面定义该类型的其他变量。

2.13.5 存储布尔值的变量

_Bool 类型存储布尔值。布尔值一般是比较的结果 true 或 false；第 3 章将学习比较操作，并使用其结果做出判断。_Bool 类型的变量值可以是 0 或 1，对应于布尔值 false 和 true。由于值 0 和 1 是整数，所以_Bool 类型也被看作整数类型。声明_Bool 变量的方式与声明其他整数类型一样，例如：

```
_Bool valid = 1; /* Boolean variable initialized to true */
```

_Bool 并不是一个理想的类型名称。名称 bool 看起来更简洁、可读性更高，但布尔类型是最近才引入 C 语言的，所以选择类型名称_Bool，可以最大限度地减少与已有代码冲突的可能性。如果把 bool 选作类型名称，则在将 bool 作为一种内置类型的编译器上，使用 bool 名称的程序大都不会编译。

尽管如此，仍可以使用 bool 作为类型名称，只需在使用它的源文件中给<stdbool.h>标准头文件添加#include 指令即可。除了把 bool 定义为_Bool 的对应名称之外，<stdbool.h>头文件还定义了符号 true 和 false，分别对应 1 和 0。因此，如果在源文件中包含了这个头文件，就可以将上面的声明语句改写为：

```
bool valid = true; /* Boolean variable initialized to true */
```

这似乎比上面的版本清晰得多，所以最好包含<stdbool.h>头文件，除非有特殊的理由。

可以在布尔值和其他数值类型之间进行类型转换。非零数值转换为_Bool 类型时，会得到 1(true)，0 就转换为 0(false)。如果在算术表达式中使用_Bool 变量，编译器就会在需要时插入隐式类型转换。_Bool 类型的级别低于其他类型，所以在涉及_Bool 类型和另一个类型的操作中，_Bool 值会转换为另一个值的类型。

这里不详细介绍如何使用布尔变量，具体内容详见下一章。

2.13.6 复数类型

本节假定读者学过复数。如果读者未学过这个内容，可以跳过本节。如果对复数的概念比较模糊，这里将介绍它的基本特性。

复数的形式是 a + bi(在电子学中是 a + bj)，其中 i 是 - 1 的平方根，a 和 b 是实数。A 是实数部分，bi 是复数的虚数部分。复数可以看作实数(a, b)的有序对。

复数可以在复数面中表示，如图 2-2 所示。

www.TopSage.com

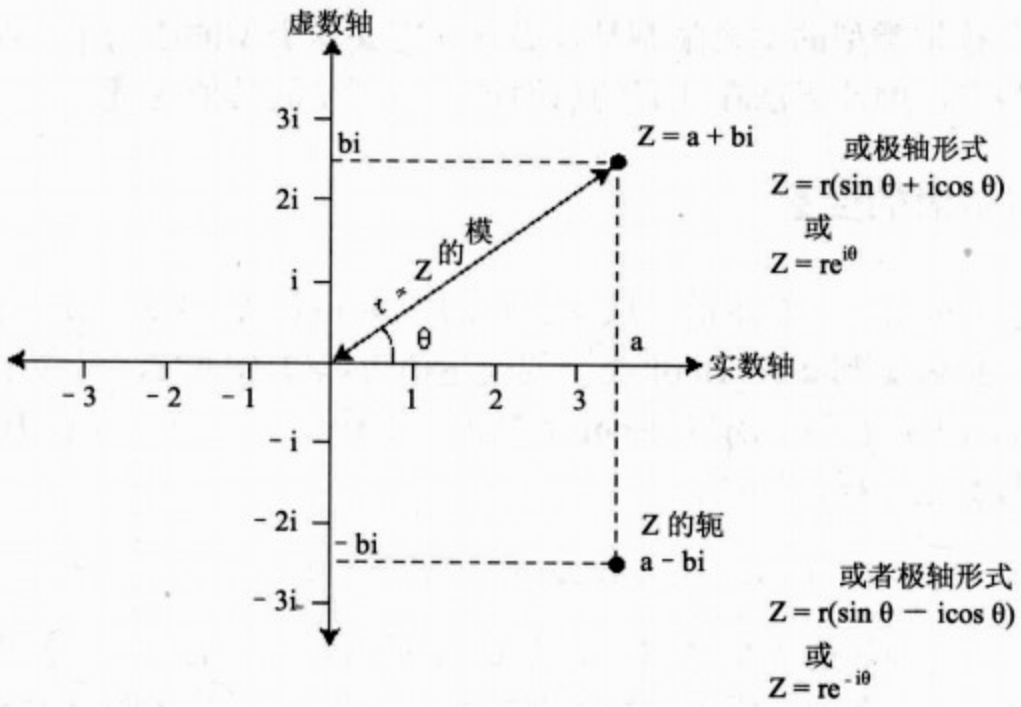


图 2-2　在复数面板中表示复数

复数可以执行如下操作。

- 模：a + bi 复数的模是$(a^2 + b^2)^{1/2}$。
- 相等：如果 a 等于 c，b 等于 d，则复数 a + bi 和 c + di 相等。
- 加：复数 a + bi 与 c + di 的和是(a + c) + (b + d)i。
- 乘：复数 a + bi 与 c + di 的积是(ac - bd) + (ad + bc)i。
- 除：复数 a + bi 与 c + di 的商是$(ac - bd) / (c^2 + d^2) + ((bc - ad)(c^2 + d^2))i$。
- 轭：复数 a + bi 的轭是 a - bi。注意，复数与其轭的积是$a^2 + b^2$。

复数也有极轴表示方式：r(sinθ+ icosθ)，还可以写为实数的有序对(r, θ)，其中 r 和θ如图 2-2 所示。在欧拉公式中，复数还可以表示为 reiθ。

下面将简要介绍 C 语言中存储复数的类型，因为这些类型的应用比较特殊。有 3 个类型可以存储复数。

- float _Complex：其实数和虚数部分为 float 类型。
- double _Complex：其实数和虚数部分为 double 类型。
- long double _Complex：其实数和虚数部分为 long double 类型。

声明一个存储复数的变量，如下所示：

```
double _Complex z1; /* Real and imaginary parts are type double */
```

为复数类型选择使用有点繁琐的_Complex 关键字，其原因与_Bool 类型相同：避免与已有的代码冲突。但<complex.h>头文件把 complex 定义为等价于_Complex，这个头文件还定义了处理复数的其他许多函数和宏。把<complex.h>头文件包含到源文件中，就可以使用 complex 代替_Complex，因此可以把 z1 变量声明为：

```
double complex z1; /* Real and imaginary parts are type double */
```

虚数单位 i 是 - 1 的平方根，用_Complex_I 关键字表示，在概念上是一个 float 类型的值。因此可以编写一个复数 2.0 + 3.0 * _Complex_I，其实数部分是 2.0，虚数部分是 3.0。

www.TopSage.com

<complex.h>头文件把 I 定义为等价于_Complex_I，所以只要在源文件中包含了这个头文件，就可以使用这个简单得多的表示方式。因此前面的复数可以表示为 2.0 + 3.0 * I.。下面的语句声明并初始化了变量 z1：

```
double complex z1 = 2.0 + 3.0*I;
   /* Real and imaginary parts are type double */
```

creal()函数返回 double complex 类型的值(该函数的参数)的实数部分，cimag()返回虚数部分，例如：

```
double real_part = creal(z1); /* Get the real part of z1 */
double imag_part = cimag(z1); /* Get the imaginary part of z1 */
```

在处理 float 类型的复数时，给这些函数名的末尾添加一个 f，即(crealf()和 cimagf())，处理 long double 类型的复数时，则添加小写的 L，即(creall()和 cimagl())。conj()函数返回其 double 参数的轭，处理其他两种类型的对应函数是 conjf()和 conjl()。

_Imaginary 关键字用于定义存储纯虚数的变量，换言之，该复数没有实数部分。虚数有 3 种类型，分别使用 float、double 和 long double 关键字，对应于 3 个复数类型。<complex.h>头文件把 imaginary 定义为_Imaginary 的可读性更高的形式，所以可以声明一个存储虚数的变量，如下：

```
double imaginary ix = 2.4*I;
```

把虚数值转换为复数，会生成一个实数部分为 0、虚数部分与已有的虚数相同的复数。把虚数类型的值转换为实数类型，而不是_Bool 类型，会得到 0。把虚数类型的值转换为_Bool 类型，若虚数值为 0，会得到 0，否则就得到 1。

包含复数和虚数值的算术表达式可以使用运算符+、*和/。下面举例说明。

试试看：使用复数

下面的简单例子会创建两个复数变量，执行一些简单的算术运算：

```
/* Program 2.17 Working with complex numbers
#include <complex.h>
#include <stdio.h>

int main(void)
{
  double complex cx = 1.0 + 3.0*I;
  double complex cy = 1.0 - 4.0*I;
  printf("Working with complex numbers:");
  printf("\nStarting values: cx = %.2f%+.2fi cy = %.2f%+.2fi",
       creal(cx), cimag(cx), creal(cy), cimag(cy));

  double complex sum = cx+cy;
  printf("\n\nThe sum cx + cy = %.2f%+.2fi",
        creal(sum),cimag(sum));
```

www.TopSage.com

```
    double complex difference = cx-cy;
    printf("\n\nThe difference cx - cy = %.2f%+.2fi",
           creal(difference),cimag(difference));

    double complex product = cx*cy;
    printf("\n\nThe product cx * cy = %.2f%+.2fi",
           creal(product),cimag(product));

    double complex quotient = cx/cy;
    printf("\n\nThe quotient cx / cy = %.2f%+.2fi",
           creal(quotient),cimag(quotient));

    double complex conjugate = conj(cx);
    printf("\n\nThe conjugate of cx = %.2f%+.2fi",
            creal(conjugate) ,cimag(conjugate));
    return 0;
}
```

这个例子的输出如下所示:

```
Working with complex numbers:
Starting values: cx = 1.00+3.00i cy = 1.00-4.00i

The sum cx + cy = 2.00-1.00i

The difference cx - cy = 0.00+7.00i

The product cx * cy = 13.00-1.00i

The quotient cx / cy = -0.65+0.41i

The conjugate of cx = 1.00-3.00i
```

代码的说明

代码非常简单。定义并初始化了变量 cx 和 cy 后，对它们使用 4 个算术运算符，并输出所有运算的结果。可以用_Complex 关键字替代 complex。

每个复数值的虚数部分都使用%+.2f 输出指定符，%后面的+指定总是输出符号。如果省略了+，符号就只有值为负时才输出。小数点后面的 2 指定输出时小数点后有两位。

在编译器提供的<complex.h>头文件中，包含许多处理复数值的函数。

2.14 赋值操作的 op=形式

C 语言是一种非常简洁的语言，提供了一些操作的缩写形式。考虑下面的代码:

```
number = number + 10;
```

这类赋值操作是给一个变量递增或递减一个数字，它非常常见，所以有一个缩写

www.TopSage.com

形式：

```
number += 10;
```

变量名后面的+=运算符是 op=运算符家族中的一员。这个语句等价于上面的语句，但输入量少了许多。op=中的 op 可以是任意算术运算符：

```
+ - * / %
```

如果 number 的值是 10，就可以编写如下语句：

```
number *= 3; /* number will be set to number*3 which is 30 */
number /= 3; /* number will be set to number/3 which is 3 */
number %= 3; /* number will be set to number%3 which is 1 */
```

op=中的 op 也可以是其他几个运算符：

```
<< >> & ^ |
```

第 3 章将介绍这些运算符。op=运算符的工作方式都相同。如果有如下形式的语句：

```
lhs op= rhs;
```

其中 rhs 表示 op=运算符右边的表达式，该语句的作用与如下形式的语句相同：

```
lhs = lhs op (rhs);
```

注意 rhs 表达式的括号，它表示 op 应用于整个 rhs 表达式的计算结果值。为了加强理解，下面看几个例子。下面的语句：

```
variable *= 12;
```

等价于：

```
variable = variable * 12;
```

现在给一个整数变量加 1 有两种方式。下面的两个语句都给 count 加 1：

```
count = count +1;
count += 1;
```

下一章将介绍这个操作的另一种方式。有这么多选择，使编写 C 程序的人数无法统计。op=运算符中的 op 应用于 rhs 表达式的计算结果，所以如下语句：

```
a /= b+1;
```

等价于：

```
a = a/(b+1);
```

到目前为止，我们的计算能力比较受限。现在只能使用一组非常基本的算术运算符。而使用标准库的功能可以大大提升计算能力。所以在进入本章的最后一个例子之前，先

www.TopSage.com

看看标准库提供的一些数学函数。

2.15 数学函数

math.h 头文件包含各种数学函数的声明。为了了解这些数学函数，下面介绍最常用的函数。所有的函数都返回一个 double 类型的值。

表 2-10 列出了各种用于进行数值计算的函数，它们都需要 double 类型的参数。

表 2-10 用于进行数值计算的函数

函　数	操　作
floor(x)	返回不大于 x(double 类型)的最大整数
ceil(x)	返回不小于 x(double 类型)的最小整数
fabs(x)	返回 x 的绝对值
log(x)	返回 x 的自然对数(底为 e)
log10(x)	返回 x 的对数(底为 10)
exp(x)	返回 e^x 的值
sqrt(x)	返回 x 的平方根
pow(x)	返回 x^y 的值

下面是使用这些函数的一些例子：

```
double x = 2.25;
double less = 0.0;
double more = 0.0;
double root = 0.0;
less = floor(x);  /* Result is 2.0 */
more = ceil(x);   /* Result is 3.0 */
root = sqrt(x);   /* Result is 1.5 */
```

还有一些三角函数，如表 2-11 所示。参数和返回值的类型也是 double，角度表示为弧度。

表 2-11 三 角 函 数

函　数	操　作
sin(x)	x(弧度值)的正弦
cos(x)	x 的余弦
tan(x)	x 的正切

如果使用三角法，这些函数的用法非常简单。下面是一些例子：

www.TopSage.com

```
double angle = 45.0;            /* Angle in degrees */
double pi = 3.14159265;
double sine = 0.0;
double cosine = 0.0;
sine = sin(pi*angle/180.0);    /*Angle converted to radians */
cosine = sin(pi*angle/180.0); /*Angle converted to radians */
```

180° 等于 1 弧度，所以以度数表示的角度除以 180，再乘以 PI 的值，就得到其弧度值，这些函数都要求使用弧度值。

还可以使用反三角函数：asin()、acos()和 atan()，以及双曲线函数 sinh()、cosh()和 tanh()。如果要使用这些函数，必须在程序中包含 math.h 头文件。如果不需要使用这些函数，就可以跳过本节。

2.16 设计一个程序

下面设计本章末的一个真实例子。在一个新程序中试用一些数值类型是一个很不错的想法。这里将从头开始编写一个程序，涉及编程的所有基本要素，包括问题的初始描述、问题的分析、解决方案的准备、编写程序、运行程序，以及测试它，确保它正常工作。该过程的每一步都会引入新问题，而不仅仅是纸上谈兵。

2.16.1 问题

许多人都对树的高度很感兴趣。如果将树砍倒，量出它的高度，就可以确定离树多远才是安全的。这对于患有神经衰弱的人来说非常重要。问题是如何不使用非常长的梯子，就可以确定树的高度，因为长梯也会对人和树枝带来危险。为了确定树的高度，可以向朋友求助，最好找一个个子比较矮的朋友。假定要测量的树比自己和朋友都高。比自己还矮的树很容易测量出其高度，除非这棵树长满了刺。

2.16.2 分析

现实问题很少能用适合于编程的方式来表达。在编写代码之前，需要确保完全理解了问题及其解决方式。只有这样，才能估计出创建解决方案所需的时间和精力。

分析阶段应增强对问题的理解，确定解决它的逻辑过程。一般这需要大量的工作，这包括找出问题阐述中模糊或遗漏的细节。只有全面理解了问题，才能开始以适合编程的形式表达解决方案。

我们打算用一个简单的图形和两个人(一高一矮)的身高来确定树的高度。首先给高个子命名为 Lofty，矮个子命名为 Shorty。为了得到比较精确的结果，高个子应明显高于矮个子。否则高个子可以考虑站在一个箱子上。图 2-3 给出了解决这个问题的思路。

www.TopSage.com

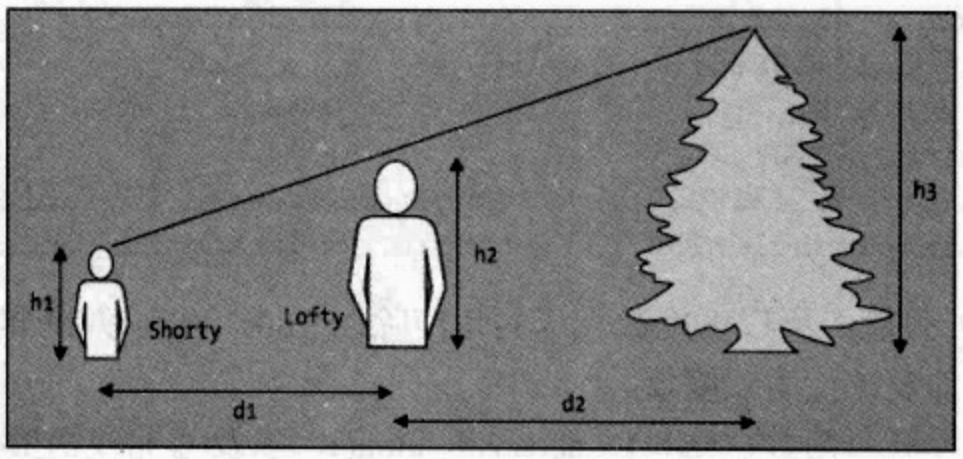


图 2-3 树的高度

确定树的高度是很简单的。如果知道图中 h1 和 h2 的值(它们分别是 Shorty 和 Lofty 的高度)以及 d1 和 d2(它们分别是 Shorty 与 Lofty 之间的距离和 Lofty 与树之间的距离)，就可以计算出树的高度。使用相似三角形的特性就可以求出树的高度，如图 2-4 所示。

因为三角形是相似的，所以 height1:distance1=height2:distance2。使用这个关系，就可以通过 Shorty 和 Lofty 的身高、以及他们与树之间的距离求出树的高度，如图 2-5 所示。

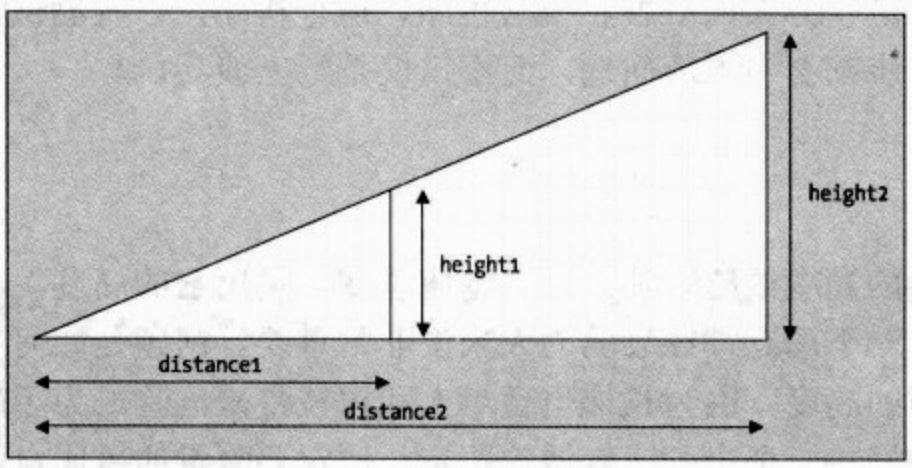


图 2-4 相似三角形

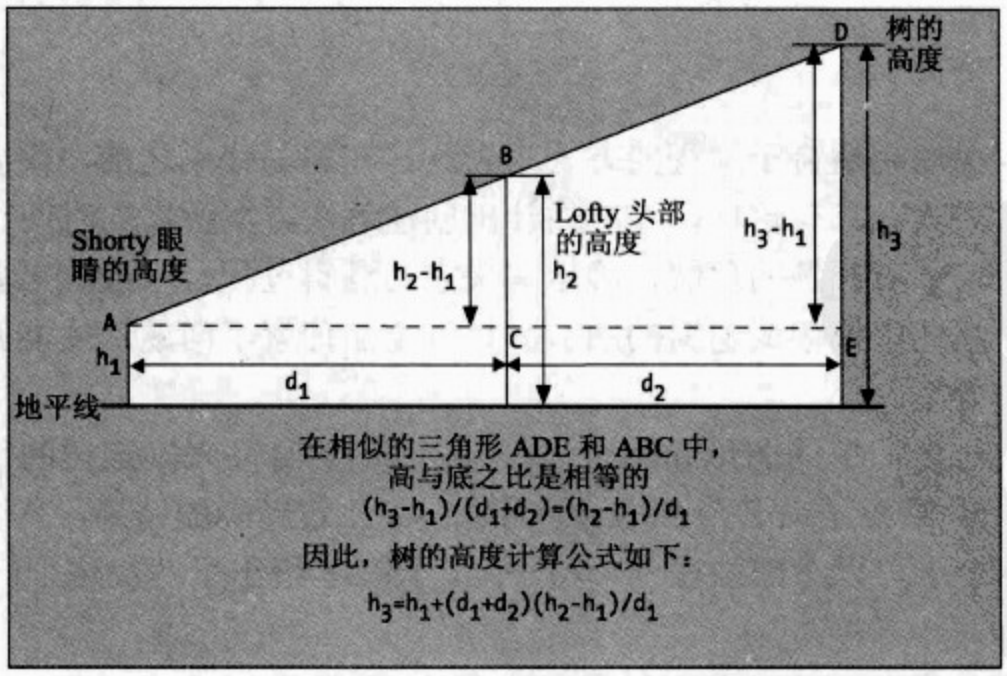


图 2-5 计算树的高度

三角形 ADE 和 ABC 与图 2-4 相同。由于这两个三角形相似，所以可以使用图 2-5 底部的等式计算出树的高度。

这说明，在程序中，可以使用如下 4 个值计算出树的高度：

- Shorty 与 Lofty 之间的距离，即图中的 d1。用 shorty_to_lofty 变量存储这个值。
- Lofty 与树之间的距离，即图中的 d2。用 lofty_to_tree 变量存储这个值。
- 从地平线到 Lofty 头部的高度，即图中的 h2，用 lofty 变量存储这个值。
- 从地平线到 Shorty 眼睛的高度，即图中的 h1，用 shorty 变量存储这个值。

接着，把这些值放在计算树高的等式中。首先要把这 4 个值输入计算机。接着使用其比值计算出树的高度，最后输出答案。步骤如下：

(1) 输入需要的值。

(2) 使用图中的等式计算树的高度。

(3) 显示答案。

2.16.3 解决方案

本节列出解决问题的步骤。

1. 步骤 1

第一步获取计算树高需要的值。这意味着必须包含 stdio.h 头文件，因为需要使用 printf()和 scanf()函数。接着确定存储这些值的变量。之后，就可以使用 printf()提示输入数字，使用 scanf()从键盘上读取值。

为了方便用户，把高个子和矮个子的身高输入为英尺英寸值。但在程序中，高度和距离使用相同的单位会更方便，所以应将所有的数字都转换为英寸值。我们需要两个变量存储 Shorty 和 Lofty 的身高(英寸值)，还需要一个变量存储 Shorty 和 Lofty 之间的距离，需要另一个变量存储 Lofty 与树之间的距离，当然，这两个距离值都以英寸为单位。

在输入过程中，首先将 Lofty 的身高输入为一个整数英尺值和一个英寸值，在此过程中要提示用户输入每个值。为此可以使用另外两个变量，一个存储英尺值，另一个存储英寸值。接着把它们转换为英寸值，将结果存储在为 Lofty 身高保留的变量中。对 Shorty 的身高进行相同的处理(但只输入从地平线到 Shorty 眼睛的高度)，最后处理他们之间的距离。对于 Lofty 与树之间的距离，可以只使用整数英尺值，因为这已经足够准确了——还要把距离转换为英寸值。对于每个输入的英尺值和英寸值，可以使用相同的变量。所以下面是程序的第一部分：

```
/* Program 2.18 Calculating the height of a tree */
#include <stdio.h>

int main(void)
{
  long shorty = 0L;        /* Shorty's height in inches */
  long lofty = 0L;         /* Lofty's height in inches */
  long feet = 0L;
```

www.TopSage.com

```
    long inches = 0L;
    long shorty_to_lofty = 0L; /* Distance from Shorty to Lofty in inches */
    long lofty_to_tree = 0L;   /* Distance from Lofty to the tree in inches */
    const long inches_per_foot = 12L;

    /* Get Lofty's height */
    printf("Enter Lofty's height to the top of his/her head,
          in whole feet: ");
    scanf("%ld", &feet);
    printf(" ...and then inches: ");
    scanf("%ld", &inches);
    lofty = feet*inches_per_foot + inches;

    /* Get Shorty's height up to his/her eyes */
    printf("Enter Shorty's height up to his/her eyes, in whole feet: ");
    scanf("%ld", &feet);
    printf("              ... and then inches: ");
    scanf("%ld", &inches);
    shorty = feet*inches_per_foot + inches;

    /* Get the distance from Shorty to Lofty */
    printf("Enter the distance between Shorty and Lofty, in whole feet: ");
    scanf("%ld", &feet);
    printf("              ... and then inches: ");
    scanf("%ld", &inches);
    shorty_to_lofty = feet*inches_per_foot + inches;

    /* Get the distance from Lofty to the tree */
    printf("Finally enter the distance to the tree to the nearest foot: ");
    scanf("%ld", &feet);
    lofty_to_tree = feet*inches_per_foot;

    /* The code to calculate the height of the tree will go here */

    /* The code to display the result will go here */
    return 0;
}
```

注意，代码进行了缩进，以便于阅读。这不是必须的，但如果要在未来修改程序，这么做更便于确定程序的工作方式。应总是给程序添加注释，以帮助理解程序。至少要清楚地说明变量的用途，解释程序的基本逻辑。

使用一个声明为 const 的变量将英尺转换为英寸。该变量的名称是 inches_per_foot，说明了它在代码中使用时会发生什么。这要比明确使用 12 这个数字好得多。这里处理的是英尺和英寸，大多数人都知道，12 英寸是 1 英尺。但在其他环境下，数值常量的重要性没有这么明显。如果在计算薪水的程序中使用 0.22，它的含义就不是很明显。因此，这个计算相当难理解。如果创建一个 const 变量 tax_rate，把它初始化为 0.22，就不会有理解障碍了。

www.TopSage.com

2. 步骤 2

有了需要的所有数据后，就可以计算树的高度了。只需利用变量的值，实现计算树高的等式即可。这里需要声明另一个变量来存储树的高度。

为此，添加如下粗体的代码：

```
/* Program 2.18 Calculating the height of a tree */
#include <stdio.h>

int main(void)
{
  long shorty = 0L;  /* Shorty's height in inches */
  long lofty = 0L;   /* Lofty's height in inches */
  long feet = 0L;    /* A whole number of feet */
  long inches = 0L;
  long shorty_to_lofty = 0; /* Distance from Shorty to Lofty in inches */
  long lofty_to_tree = 0;   /* Distance from Lofty to the tree in inches */
  long tree_height = 0;     /* Height of the tree in inches */
  const long inches_per_foot = 12L;

  /* Get Lofty's height */
  printf("Enter Lofty's height to the top of his/her head,
        in whole feet: ");
  scanf("%ld", &feet);
  printf("           ...and then inches: ");
  scanf("%ld", &inches);
  lofty = feet*inches_per_foot + inches;

  /* Get Shorty's height up to his/her eyes */
  printf("Enter Shorty's height up to his/her eyes, in whole feet: ");
  scanf("%ld", &feet);
  printf("           ... and then inches: ");
  scanf("%ld", &inches);
  shorty = feet*inches_per_foot + inches;

  /* Get the distance from Shorty to Lofty */
  printf("Enter the distance between Shorty and Lofty, in whole feet: ");
  scanf("%ld", &feet);
  printf("           ... and then inches: ");
  scanf("%ld", &inches);
  shorty_to_lofty = feet*inches_per_foot + inches;

  /* Get the distance from Lofty to the tree */
  printf("Finally enter the distance to the tree to the nearest foot: ");
  scanf("%ld", &feet);
  lofty_to_tree = feet*inches_per_foot;

  /* Calculate the height of the tree in inches */
  tree_height = shorty + (shorty_to_lofty + lofty_to_tree)
```

www.TopSage.com

```
  *(lofty-shorty)/shorty_to_lofty;

  /* The code to display the result will go here */
  return 0;
}
```

计算树高的语句与图中的等式相同。这有点繁琐，但直接转换为程序中的语句，以计算树高。

3. 步骤 3

最后，输出答案。为了以最容易理解的形式显示结果，应把存储在 tree_height 中的结果(英寸值)转换为英尺和英寸值：

```
/* Program 2.18 Calculating the height of a tree */
#include <stdio.h>

int main(void)
{
  long shorty = 0L;          /* Shorty's height in inches */
  long lofty = 0L;           /* Lofty's height in inches */
  long feet = 0L;
  long inches = 0L;
  long shorty_to_lofty = 0; /* Distance from Shorty to Lofty in inches */
  long lofty_to_tree = 0;   /* Distance from Lofty to the tree in inches */
  long tree_height = 0;     /* Height of the tree in inches */
  const long inches_per_foot = 12L;

  /* Get Lofty's height */
  printf("Enter Lofty's height to the top of his/her head,
        in whole feet: ");
  scanf("%ld", &feet);
  printf("            ... and then inches: ");
  scanf("%ld", &inches);
  lofty = feet*inches_per_foot + inches;

  /* Get Shorty's height up to his/her eyes */
  printf("Enter Shorty's height up to his/her eyes, in whole feet: ");
  scanf("%ld", &feet);
  printf("            ... and then inches: ");
  scanf("%ld", &inches);
  shorty = feet*inches_per_foot + inches;

  /* Get the distance from Shorty to Lofty */
  printf("Enter the distance between Shorty and Lofty, in whole feet: ");
  scanf("%ld", &feet);
  printf("            ... and then inches: ");
  scanf("%ld", &inches);
  shorty_to_lofty = feet*inches_per_foot + inches;
```

www.TopSage.com

```
  /* Get the distance from Lofty to the tree */
  printf("Finally enter the distance to the tree to the nearest foot: ");
  scanf("%ld", &feet);
  lofty_to_tree = feet*inches_per_foot;

  /* Calculate the height of the tree in inches */
  tree_height = shorty + (shorty_to_lofty + lofty_to_tree)*
  (lofty-shorty)/shorty_to_lofty;

  /* Display the result in feet and inches */
  printf("The height of the tree is %ld feet and %ld inches.\n",
         tree_height/inches_per_foot, tree_height% inches_per_foot);
  return 0;
}
```

程序的输出如下：

```
Enter Lofty's height to the top of his/her head, in whole feet first: 6
      ... and then inches: 2
Enter Shorty's height up to his/her eyes, in whole feet: 4
      ... and then inches: 6
Enter the distance between Shorty and Lofty, in whole feet : 5
      ... and then inches: 0
Finally enter the distance to the tree to the nearest foot: 20
The height of the tree is 12 feet and 10 inches.
```

2.17 小结

本章介绍了许多基础知识，讨论了 C 程序的构建方式、各种算术运算、如何选择合适的变量类型等。除了算术运算之外，还学习了输入输出功能，通过 scanf()将值输入变量，通过 Printf()函数把文本、字符值和数值变量输出到屏幕上。读者可能不能第一次就掌握所有这些内容，但可以在需要时复习本章。

下一章将开始学习如何根据输入值做出判断，控制程序的执行。这是创建有趣且专业化程序的关键。

表 2-12 总结了前面介绍的变量类型。在学习本书的过程中，可以随时复习这些内容。

表 2-12 变量类型和值域

类 型	字 节 数	值 域
char	1	-128~+127 或 0~+255
unsigned char	1	0~+255
short	2	-32 768~+32 767
unsigned short	2	0~+65,535
int	4	-32 768~+32 767 或 -2 147 438 648~+2 147 438 647

www.TopSage.com

(续表)

类　型	字 节 数	值　域
unsigned int	4	0~+65 535 或 0~+4 294 967 295
long	4	-2 147 438 648~+2 147 438 647
unsigned long	4	0~+4 294 967 295
long long	8	-9 223 372 036 854 775 808 到 +9 223 372 036 854 775 807
unsigned long long	8	0~+18 446 744 073 709 551 615
float	4	±3.4E38 (6 位)
double	8	±1.7E308 (15 位)
long double	12	±1.2E4932 (19 位)

存储复数的类型如表 2-13 所示。

表 2-13　复 数 类 型

类　型	说　明
float _Complex	存储一个复数，其实数和虚数部分为 float 类型
double _Complex	存储一个复数，其实数和虚数部分为 double 类型
long double _Complex	存储一个复数，其实数和虚数部分为 long double 类型
float _Imaginary	把虚数存储为 float 类型
double _Imaginary	把虚数存储为 double 类型
long double _Imaginary	把虚数存储为 long double 类型

<complex.h>头文件把 complex 和 imaginary 定义为_Complex 和_Imaginary 关键字的替代品，把 I 定义为表示 i，即 1 的平方根。

本章还介绍并使用了 printf()函数的数据输出格式指定符，完整的指定符列表请参见附录 D。附录 D 还描述了输入格式指定符，它们用于控制使用 scanf()函数从键盘上读取数据时这些数据的解释方式。当无法确定如何处理输入或输出数据时，可以参阅附录 D。

2.18　练习

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容。还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 2.1　编写一个程序，提示用户用英寸输入一个距离，然后将该距离值输出为码、英尺和英寸的形式。

www.TopSage.com

习题 2.2 编写一个程序，提示用户用英尺和英寸输入一个房间的长和宽，然后计算并输出面积，单位是平方码，精度为小数点后有两位数。

习题 2.3 一个产品有两种版本：其一是标准版，价格是$3.5，其二是豪华版，价格是$5.5。编写一个程序，使用学到的知识提示用户输入产品的版本和数量，然后根据输入的产品数量，计算并输出价格。

习题 2.4 编写一个程序，提示用户从键盘输入一个星期的薪水(以美元为单位)和工作时数，它们均为浮点数，然后计算并输出每个小时的平均薪水，输出格式如下所示：

```
Your average hourly pay rate is 7 dollars and 54 cents.
```



www.TopSage.com

www.TopSage.com

第3章

条件判断

第2章学习了如何在程序中执行计算。本章将在可以编写的程序种类和构建程序的灵活性方面迈出一大步。我们要学习一种非常强大的编程工具：比较表达式的值，根据其结果，选择执行某组语句。

也就是说，可以控制程序中语句的执行顺序。到目前为止，程序中的所有语句都严格按顺序执行。本章将改变这种状况。

TopSage.com

本章的主要内容：

- 根据算术比较的结果来判断
- 逻辑运算符的概念及其用法
- 再谈从键盘上读取数据
- 编写一个可用作计算器的程序

3.1 判断过程

这里将从程序中的基本元素开始。在程序中做出判断，就是选择执行一组程序语句，而不执行另一组程序语句。在现实生活中，我们总是要做判断。我们每天睡醒后，都要决定是否去工作。我们要回答如下问题：

感觉还好吗？
如果答案是否定的，就躺在床上不动。否则，就去工作。

可以把这些问题重写为：

如果感觉良好，就去工作。否则，就躺在床上不动。

这是一个很简单的判断。之后在吃早餐时，发现下雨了。于是决定：

如果雨下得和昨天一样大，就乘公交车。如果雨比昨天还大，就自己驾车。否则，就冒雨步行。

这是一个比较复杂的判断过程。这个判断根据雨的大小分为几级，可能有三种不同的结果。

www.TopSage.com

在这一天中，我们还要做出更多的判断。不进行这些判断，就只能执行一个动作。本书到目前为止，程序都只执行一个动作。所有的程序都沿着一条路线运行到定义好的终点，不做任何判断。这对于程序的功能而言是一个严重的限制。为了突破这个限制，下面先介绍一些基础知识。

3.1.1 算术比较

要做判断，就需要一种比较机制。这涉及到一些新运算符。由于要处理数字，比较数值就是做判断的基本操作。用于比较数值的三个基本关系运算符如下：

- 小于<
- 等于==
- 大于>

注意：

等于运算符是两个连续的等号(==)，使用一个等号会出错。这是很容易混淆的。下面说明其区别。如果输入 my_weight = your_weight，这就是一个赋值语句，将 your_weight 变量的值放在 my_weight 变量中。如果输入表达式 my_weight == your_weight，就是在比较两个数值：确定两个数值是否相同——而不是使它们相等。如果在本应使用==的地方使用了=，编译器就不知道这是否是个错误，因为它们都是有效的。

3.1.2 涉及关系运算符的表达式

看看下面的例子：

```
5 < 4        1 == 2        5 > 4
```

这些表达式称为逻辑表达式或布尔表达式，因为每个表达式都会得到两个结果之一：true 或 false。如上一章所述，1 表示值 true，0 表示 false。第一个表达式是 false，因为 5 不小于 4。第二个表达式也是 false，因为 1 不等于 2。第三个表达式是 true，因为 5 大于 4。

关系表达式会得到一个布尔结果，所以可以把其结果存储在_Bool 类型的变量中。例如：

```
_Bool result = 5 < 4; /* result will be false */
```

如果在源文件中用#include 指令包含头文件<stdbool.h>，就可以使用 bool，而不是关键字_Bool，所以可以编写如下语句：

```
bool result = 5 < 4; /* result will be false */
```

任何非零数值在转换为_Bool 类型时，都得到 true。这表示，可以把算术表达式的结果赋予_Bool 变量，如果它是非零值，就存储 true，否则就存储 false。

www.TopSage.com

3.1.3 基本的 if 语句

有了做判断的关系运算符后，就需要使用一个语句来做判断。最简单的语句就是 if 语句。如果要比较自己和他人的体重，并根据结果打印不同的句子，就可以编写如下程序：

```
if(your_weight > my_weight)
  printf("You are heavier than me.\n");

if(your_weight < my_weight)
  printf("I am heavier than you.\n");

if(your_weight == my_weight)
  printf("We are exactly the same weight.\n");
```

注意，每个 if 后面的语句都进行了缩进。这说明这些语句取决于 if 测试的结果。下面看看这些代码。第一个 if 测试 your_weight 的值是否大于 my_weight 的值。比较表达式位于 if 关键字后面的括号中。如果比较的结果是 true，就执行 if 后面的语句。输出如下信息：

```
You are heavier than me.
```

之后执行下一条 if 语句。

如果第一个 if 的括号中的表达式是 false，该怎么办？在这种情况下，就跳过 if 后面的语句，不显示信息。只有 your_weight 大于 my_weight，才显示信息。

第二个 if 的工作方式与第一个 if 相同。如果关键字 if 后面的括号中的表达式是 true，就执行下面的语句，输出如下信息：

```
I am heavier than you.
```

如果 your_weight 小于 my_weight，就执行输出信息的语句。否则就跳过该语句，不显示信息。第三个 if 也是这样。这些语句的作用都是根据 your_weight 是大于、小于还是等于 my_weight 来输出信息。这个程序只显示一条信息，因为只有其中一个 if 的结果是 true。

if 语句的一般形式或语法如下：

```
if(expression)
  Statement1;
Next_statement;
```

注意，进行测试(if)的表达式放在括号中，在第一行的末尾没有分号。这是因为 if 关键字所在的一行代码和其后的一行代码是联系在一起的。第二行代码可以写在第一行的后面，如下所示：

```
if(expression) Statement1;
```

www.TopSage.com

但为了简洁起见，一般应把 Statement1 放在新的一行上。括号中的 expression 可以是结果为 true 或 false 的任意表达式。如果表达式为 true，就执行 Statement1，之后程序继续执行 Next_statement。如果表达式为 false，就跳过 Statement1，直接执行 Next_statement，如图 3-1 所示。

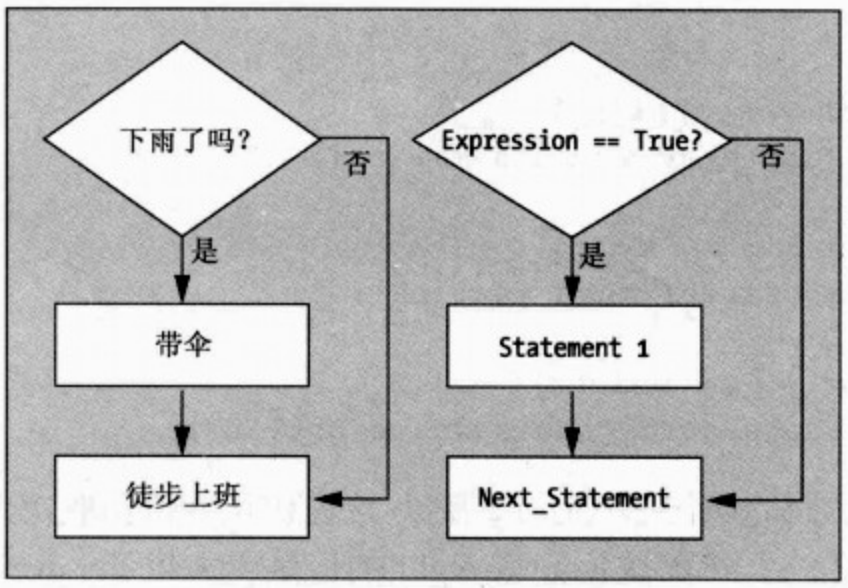


图 3-1　if 语句的执行过程

使用基本的 if 语句，可以在上一章计算树高的程序中添加一些不太礼貌的注释。例如，在计算了矮个子的身高后，添加如下代码：

```
if(Shorty < 36)
  printf("\nMy, you really are on the short side, aren't you?");
```

这里使用 if 语句添加了一个带有冒犯意味的评论，因为这个矮个子的身高小于 36 英寸。

前面提到，把一个数值转换为_Bool 类型时，会得到一个布尔结果。if 语句的控制表达式要生成一个布尔结果，所以编译器要将 if 表达式的数值结果转换为_Bool 类型。有时在程序中会使用它测试计算式的非零结果。如下面的语句所示：

```
if(count)
  printf("The value of count is not zero.");
```

只有 count 非零，才会输出结果，因为 count 的值是 0，表示 if 表达式为 false。

试试看：检查条件

下面看看 if 语句的执行情况。这个程序让用户输入一个 1～10 之间的数字，再确定该数字有多大。

```
/* Program 3.1 A simple example of the if statement */
#include <stdio.h>

int main(void)
{
  int number = 0;
  printf("\nEnter an integer between 1 and 10: ");
```

www.TopSage.com

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

    if(number > 5)
      printf("You entered %d which is greater than 5\n", number);

    if(number < 6)
      printf("You entered %d which is less than 6\n", number);
    return 0;
}
```

这个程序的输出如下:

```
Enter an integer between 1 and 10: 7
You entered 7 which is greater than 5
```

或

```
Enter an integer between 1 and 10: 3
You entered 3 which is less than 6
```

代码的说明

与往常一样，在开头包含一个注释，说明程序要做什么。包含 stdio.h 头文件是为了使用 printf()语句。接着是程序的 main()函数。这个函数没有返回值，因为它使用了关键字 void:

```
/* Program 3.1 A simple example of the if statement*/
#include <stdio.h>

int main(void)
{
```

在 main()函数体的前三条语句中，在提示用户输入数据后，从键盘上读取一个整数:

```
int number = 0;
printf("\nEnter an integer between 1 and 10: \n");
scanf("%d",&number);
```

这段代码声明一个整型变量 number，接着提示用户输入一个 1 ~ 10 之间的数字，使用 scanf()函数读取这个数值，并把该数字存储在变量 number 中。

下一条语句是一条测试输入值的 if 语句:

```
if(number > 5)
  printf("You entered %d which is greater than 5", number);
```

比较 number 变量的值和 5。如果 number 大于 5，就执行下一条语句，显示一条信息，然后进入程序的下一部分。如果 number 不大于 5，就跳过 printf()。printf()给整数值使用%d 转换指定符，输出用户键入的值。

接着是另一条 if 语句:

```
if(number < 6)
```

www.TopSage.com

```
printf("You entered %d which is less than 6", number);
```

这条 if 语句比较输入的值和 6，如果输入的值较小，就执行下一条语句，显示一条信息。否则，就跳过 printf()，结束程序。两条信息只可能显示其中一条，因为输入的数字要么小于 6，要么大于 5。

if 语句允许选择接受什么输入，以及如何处理它。例如，如果给变量的值添加特定的限制，则即使在程序中输入了较大的值，也可以编写如下语句：

```
if(x > 90)
  x = 90;
```

如果用户输入了大于 90 的值，程序就会将它自动更改为 90。如果程序只能处理某个范围内的值，这就是很有效的。还可以检查某个值是否低于某个给定的数字，如果是，就把它设置为该数字。这样，就可以确保该值在指定的范围内。

最后，使用 return 语句结束程序，将控制权返回给操作系统：

```
return 0;
```

3.1.4 扩展 if 语句：if-else

可以扩展 if 语句，提供更多的灵活性。假定昨天下雨了，就可以编写如下语句：

```
如果今天的雨比昨天还大，
我就带上雨伞。
否则
我就穿上夹克，
然后去上班。
```

这就是 if-else 语句提供的判断方式。if-else 语句的语法如下：

```
if(expression)
  Statement1;
else
  Statement2;

Next_statement;
```

这里有一个双重选择。根据 expression 的值是 true 还是 false，执行 Statement1 或 Statement2。

- 如果 expression 的值是 true，就执行 Statement1，之后程序继续执行 Next_statement。
- 如果 expression 的值是 false，就执行 Statement2，之后程序继续执行 Next_statement。

其执行过程如图 3-2 所示。

试试看：使用 if 语句分析数字

假定某个产品的售价是$3.50/个，当订购数量大于 10 时，就提供 5%的折扣。使用 if-else 语句可以计算并输出给定数量的总价。

www.TopSage.com

```
/* Program 3.2 Using if statements to decide on a discount */
#include <stdio.h>

int main(void)
{
  const double unit_price = 3.50; /* Unit price in dollars */
  int quantity = 0;
  printf("Enter the number that you want to buy:"); /* Prompt message */
  scanf(" %d", &quantity);          /* Read the input */

  /* Test for order quantity qualifying for a discount */
  if(quantity > 10)                 /* 5% discount */
    printf("The price for %d is $%.2f\n", quantity,
           quantity*unit_price*0.95);
  else                              /* No discount */
    printf("The price for %d is $%.2f\n", quantity, quantity*unit_price);
  return 0;
}
```

这个程序的输出如下:

```
Enter the number that you want to buy:20
The price for 20 is $66.50
```

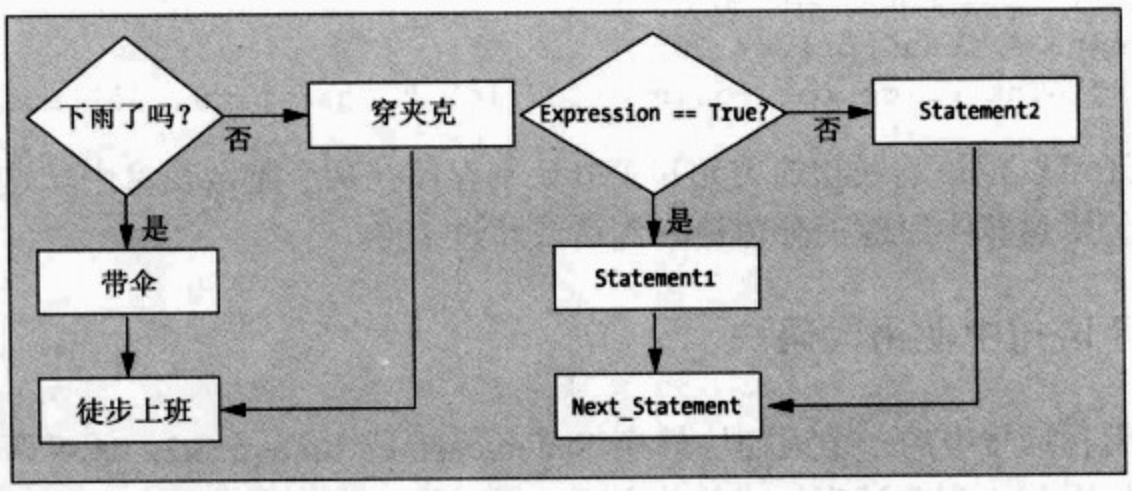


图 3-2 if-else 语句的执行过程

代码的说明

程序读取了订购数量后，if-else 语句将完成所有的工作:

```
if(quantity > 10)         /* 5% discount */
 printf("\nThe price for %d is $%.2f\n", quantity,
        quantity*unit_price*0.95);
else                      /* No discount */
  printf("\nThe price for %d is $%.2f\n", quantity, quantity*unit_price);
```

如果 quantity 大于 10，就执行第一个 printf()，应用 5%的折扣。否则，就执行第二个 printf()，不应用折扣。

这个主题有几个地方需要说明。首先，可以用简单的 if 语句替代 if-else 语句，来解

www.TopSage.com

决这个问题，如下面的代码所示：

```
double discount = 0.0; /* Discount allowed */
if(quantity > 10)
  discount = 0.05; /* 5% discount */
printf("\nThe price for %d is $%.2f\n", quantity,
          quantity*unit_price*(1.0-discount));
```

这大大简化了代码。现在我们只调用了一个 printf()来应用折扣，该折扣设置为 0 或 5%。用一个变量存储折扣值，还可以使代码更清晰。

第二，浮点变量不适合于涉及钱款的计算，因为浮点变量可能会取整。如果金额不是特别大，可以用整数值存储美分。例如：

```
const long unit_price = 350L; /* Unit price in cents */
int quantity = 0;
printf("Enter the number that you want to buy:"); /* Prompt message */
scanf(" %d", &quantity);      /* Read the input */

long discount = 0L;           /* Discount allowed */
if(quantity > 10)
  discount = 5L;              /* 5% discount */
long total_price = quantity*unit_price*(100-discount)/100;
long dollars = total_price/100;
long cents = total_price%100;
printf("\nThe price for %d is $%ld.%ld\n", quantity, dollars,cents);
```

当然，还可以把每笔钱款的美元和美分分别存储在两个整数变量中。这有点复杂，因为必须在算术运算中跟踪美分值何时到达或超过 100。

3.1.5　在 if 语句中使用代码块

还可以用{}括号中的一个语句块替换 Statement1 或 Statement2，或者两者都替换。这表示，在使用 if 语句测试表达式的值之后，可以把许多指令放在一对括号中，一起提供给计算机。下面用一个真实的例子来演示这个机制：

```
如果天气晴朗，
我就去公园，吃野餐，然后回家。
否则
就留在家中看足球赛，喝啤酒。
```

涉及语句块的 if 语句的语法如下：

```
if(expression)
{
  StatementA1;
  StatementA2;
  ...
}
```

www.TopSage.com

```
else
{
  StatementB1;
  StatementB2;
  ...
}

Next_statement;
```

如果 expression 等于 true，就执行 if 后面括号中的所有语句。如果 expression 等于 false，就执行 else 后面括号中的所有语句。在这两种情况下，程序都继续执行 Next_statement。下面看看代码的缩进。括号没有缩进，但括号中的语句缩进了。这使开闭括号中的所有语句非常清楚。

注意：

在 if 语句中，可以用一个语句块替代单个语句，这只是一般规则的一个应用例子。其实，只要可以使用单个语句的地方，都可以使用放在括号中的语句块。这也说明，可以把一个语句块嵌套在另一个语句块中。

3.1.6 嵌套的 if 语句

if 语句中也可以包含 if 语句，这称为嵌套的 if 语句。例如：

```
如果天气很好，
我就到院子里去。
如果天气很冷，
我就坐在太阳下。
否则
我就坐在树荫下。
否则
我就待在屋内，
然后喝一些柠檬水。
```

对应的程序代码如下：

```
if(expression1)    /* Weather is good? */
{
  StatementA;      /* Yes - Go out in the yard */
  if(expression2)  /* Cool enough? */
    StatementB;    /* Yes - Sit in the sun */
  else
    StatementC;    /* No - Sit in the shade */
}
else
  StatementD;      /* Weather not good - stay in */
Statement E;       /* Drink lemonade in any event */
```

其中，第二个 if 条件只有在第一个 if 条件 expression1 为 true 时才检查。包含

www.TopSage.com

StatementA 和第二个 if 的括号是必须的，以使两条语句都在 expression1 为 true 时执行。注意，else 与它所属的 if 对齐。其逻辑如图 3-3 所示。

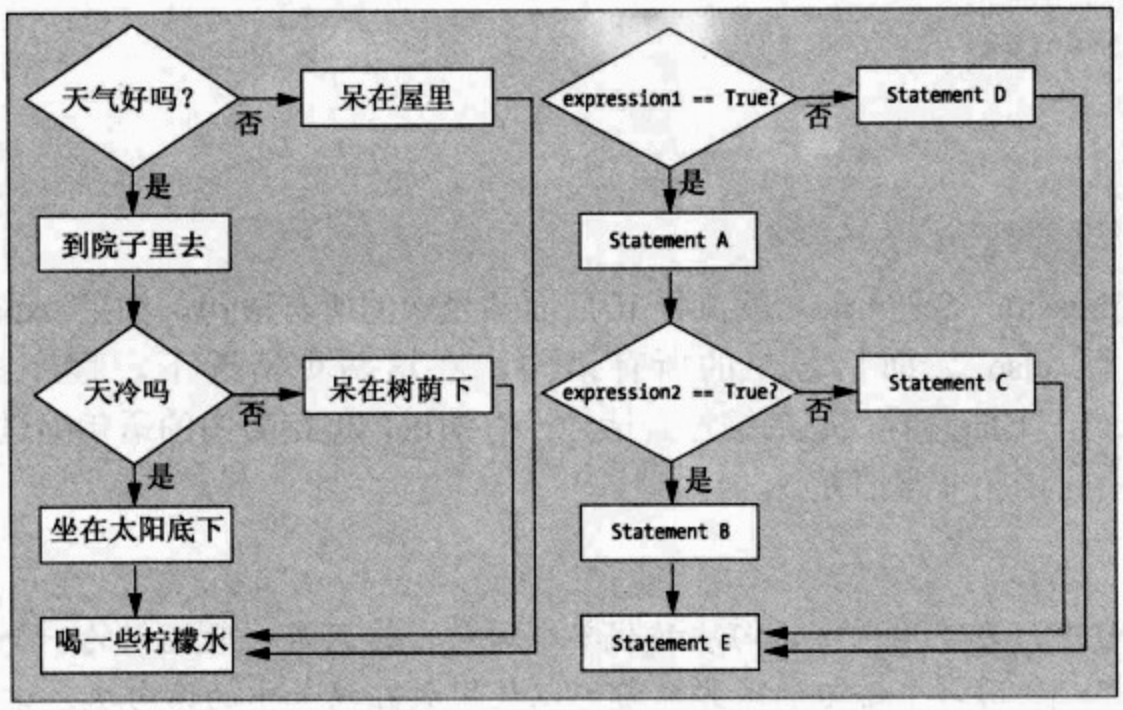


图 3-3 嵌套的 if 语句

试试看：分析数字

下面用另外几个例子练习 if 技巧。这个程序测试输入的数是偶数还是奇数，如果是偶数，就接着测试该数的一半是否还是偶数:

```
/* Program 3.3 Using nested ifs to analyze numbers */
#include <stdio.h>
#include <limits.h> /* For LONG_MAX */

int main(void)
{
  long test = 0L;   /* Stores the integer to be checked */

  printf("Enter an integer less than %ld:", LONG_MAX);
  scanf(" %ld", &test);

  /* Test for odd or even by checking the remainder after dividing by 2 */
  if(test % 2L == 0L)
  {
    printf("The number %ld is even", test);

    /* Now check whether half the number is also even */
    if((test/2L) % 2L == 0L)
    {
      printf("\nHalf of %ld is also even", test);
      printf("\nThat's interesting isn't it?\n");
    }
  }
  else
    printf("The number %ld is odd\n", test);
```

www.TopSage.com

```
  return 0;
}
```

输出如下所示：

```
Enter an integer less than 2147483647:20
The number 20 is even
Half of 20 is also even
That's interesting isn't it?
```

或者：

```
Enter an integer less than 2147483647:999
The number 999 is odd
```

代码的说明

提示输入时使用了在<limits.h>头文件中定义的 LONG_MAX 符号，它指定 long 类型的最大值。从输出可以看出，long 值的上限是 2147483647

第一个 if 条件测试输入是否为一个偶数：

```
if(test % 2L == 0L)
```

如果这里使用 0 代替 0L，编译器就会插入代码，将 int 类型的 0 转换为 long 类型，以便进行相等比较。使用 long 类型的常量 0L 可以避免这个不必要的操作。任何偶数除以 2 的余数均为 0，如果这个表达式为 true，就执行其后的代码块：

```
{
  printf("The number %ld is even", test);

  /* Now check whether half the number is also even */
  if((test/2L) % 2L == 0L)
  {
    printf("\nHalf of %ld is also even", test);
    printf("\nThat's interesting isn't it?\n");
  }
}
```

输出了表示输入值为偶数的信息后，执行另一个 if 语句。这称为嵌套的 if，因为它位于第一个 if 中。嵌套的 if 将初值除以 2，并使用与第一个 if 语句相同的机制，测试结果是否是偶数。在嵌套的 if 条件中，表达式 test/2L 有一对额外的括号，这不是必要的，但它们有助于使操作更清晰。使程序便于理解是优秀编程风格的本质。如果嵌套 if 条件的结果为 true，就执行其后代码块中的另外两条 printf()语句。

添加代码，使嵌套的 if 变成 if-else 语句，输出"Half of %ld is odd"。如果输入的初始值不是偶数，就执行 else 关键字后面的语句：

```
else
  printf("The number %ld is odd\n", test);
```

www.TopSage.com

注意：

可以在 if 语句的任意位置嵌套 if，但最好不要这么做，否则，程序就很难理解，还有可能出错。

为了使嵌套的 if 语句在条件为 false 时输出一条信息，需要在闭括号的后面插入如下代码：

```
else
  printf("\nHalf of %ld is odd", test);
```

3.1.7　更多的关系运算符

可以在 if 语句中添加更多的关系运算符来比较表达式。下面的 3 个运算符构成了完整的比较运算符系列：

- >=大于等于
- <=小于等于
- !=不等于

这些运算符都很简单，下面是一些例子，从几个算术例子开始：

```
6 >= 5   5 <= 5   4 <= 5   4 != 5   10 != 10
```

除了最后一个例子之外，其他例子的结果都是 true，最后一个例子的结果是 false，因为 10 肯定等于 10。这些运算符可以应用于 char、wchar_t 及其他数值类型。字符类型也有一个与其相关的数值。附录 B 中的 ASCII 表列出了所有标准 ASCII 字符及其数字码。表 3-1 节选了附录 B 中的几个 ASCII 字符，用于下面几个例子。

表 3-1　字符和 ASCII 码

字　符	ASCII 码(十进制)
A	65
B	66
P	80
Q	81
Z	90
b	98

char 值可以表示为整数或放在单引号中的字符，如'A'。存储为 char 类型的数值可以带符号或不带符号，这取决编译器实现该类型的方式。当 char 类型带符号时，其值为 - 128～+127。当 char 类型不带符号时，其值为 0～255。下面的几个例子比较 char 类型的值：

```
'Z' >= 'A'   'Q' <= 'P'   'B' <= 'b'   'B' != 66
```

www.TopSage.com

根据这些字符的 ASCII 值，第一个表达式为 true，因为'Z'的 ASCII 值是 90，'A'的 ASCII 值是 65。第二个表达式为 false，因为'Q'不在'P'前面。第三个表达式为 true，因为在 ASCII 码中，小写字母比对应的大写字母大 32。最后一个表达式是 false，值 66 是字符'B'的 ASCII 十进制值。

试试看：将大写字母转换为小写字母

这个例子练习使用新的逻辑运算符，将输入的大写字母转换为小写字母。

```
/* Program 3.4 Converting uppercase to lowercase */
#include <stdio.h>

int main(void)
{
  char letter = 0;                    /* Stores a character */

  printf("Enter an uppercase letter:"); /* Prompt for input */
  scanf("%c", &letter);               /* Read a character */

  /* Check whether the input is uppercase */
  if(letter >= 'A')                   /* Is it A or greater? */
    if(letter <= 'Z')                 /* and is it Z or lower? */
    { /* It is uppercase */
      letter = letter - 'A'+ 'a'; /* Convert from upper- to lowercase */
      printf("You entered an uppercase %c\n", letter);
    }
  else                                /* It is not an uppercase letter */
    printf("Try using the shift key, Bud! I want a capital letter.\n");
  return 0;
}
```

这个程序的输出如下：

```
Enter an uppercase letter:G
You entered an uppercase g
```

或者

```
Enter an uppercase letter:s
Try using the shift key, Bud! I want a capital letter.
```

代码的说明

在前三条语句中，声明了一个 char 类型的变量 letter，并提示用户输入一个大写字母，将输入的字母存储在变量 letter 中：

```
char letter = 0;                      /* Stores a character */

printf("Enter an uppercase letter:"); /* Prompt for input */
scanf("%c", &letter);                 /* Read a character */
```

www.TopSage.com

如果输入了大写字母，letter 变量中的字符必定在'A'~'Z'之间，所以下面的 if 语句检查该字符是否大于或等于'A':

```
if(letter >= 'A') /* Is it A or greater? */
```

如果该表达式为 true，就继续执行嵌套的 if 语句，测试 letter 变量是否小于或等于'Z':

```
if(letter <= 'Z') /* and is it Z or lower? */
```

如果该表达式为 true，就执行 if 后面的语句块，把该字母转换为小写，输出一条信息:

```
{                                    /* It is uppercase */
  letter = letter - 'A'+ 'a'; /* Convert from upper- to lowercase */
  printf("You entered an uppercase %c\n", letter);
}
```

要把该字母转换为小写，应从 letter 中减去'A'的字符码，再加上'a'的字符码。如果 letter 包含'A'，减去'A'就得到 0，再加上'a'，就得到'a'。如果 letter 包含'B'，减去'A'就得到 1，再加上'a'，就得到'b'。这个转换方式适用于所有的大写字母。注意，这个方式适用于 ASCII，但不适用于其他编码系统(如 EBCDIC)，因为该系统的字母没有连续的字符码。如果希望该转换方式适用于所有的字符码，可以使用标准库函数 tolower()。如果传入的参数是大写字母，它就把字母转换为小写，否则就返回原来的字符码值。要使用这个函数，需要在程序中包含 ctype.h 头文件。这个头文件还声明了另一个对应函数 toupper()，它将小写字母转换为大写。

如果表达式 letter <= 'Z'为 false，就执行 else 后面的语句，显示另一条信息:

```
else                                 /* It is not an uppercase letter */
  printf("Try using the shift key, Bud! I want a capital letter.\n");
```

但其中有错误。如果输入的字符小于'A'，该怎么办？第一个 if 没有 else 子句，所以程序会结束，不输出任何信息。为了更正这个错误，必须在程序的结尾添加另一条 else 子句。完整的嵌套 if 语句如下所示:

```
if(letter >= 'A')          /* Is it A or greater? */
  if(letter <= 'Z')        /* and is it Z or lower? */
  {                        /* It is uppercase */
    letter = letter - 'A'+ 'a'; /* Convert from upper- to lowercase */
    printf("You entered an uppercase %c\n", letter);
  }
  else                     /* It is not an uppercase letter */
    printf("Try using the shift key, Bud! I want a capital letter.\n");
else
  printf("You didn't enter an uppercase letter\n");
```

现在得到了一条信息。注意，代码的缩进表示哪个 else 属于哪个 if。这个缩进并不确定哪个 else 属于哪个 if，只是提供了一个可视化的线索。else 总是属于它之前、还没

www.TopSage.com

有 else 子句的那个 if。

如果使用的是宽字符，该如何处理？这没有什么区别：

```
/* Program 3.4A Converting uppercase to lowercase using wide characters */
#include <stdio.h>

int main(void)
{
  wchar_t letter = 0;                /* Stores a character */

  printf("Enter an uppercase letter:"); /* Prompt for input */
  scanf("%lc", &letter);             /* Read a character */

  /* Check whether the input is uppercase */
  if(letter >= L'A')                 /* Is it A or greater? */
    if(letter <= L'Z')               /* and is it Z or lower? */
    {                                /* It is uppercase */
      letter = letter - L'A'+ L'a'; /* Convert from upper- to lowercase */
      printf("You entered an uppercase %lc\n", letter);
    }
    else                             /* It is not an uppercase letter */
      printf("Try using the shift key, Bud! I want a capital letter.\n");
  return 0;
}
```

变量 letter 的类型现在是 wchar_t，字符常量的前面都加上了 L，以表示宽字符。唯一的区别是输入输出的格式指定符使用%lc，而不是%c。

当然，无论字符码是多少，从大写转换为小写的操作都会执行，但前面提及的 tolower()和 toupper()函数不能用于宽字符。然而，<wctype.h>头文件定义了 towlower()和 towupper()函数，只要包含这个头文件，就可以编写如下语句进行转换：

```
letter = towlower(letter); /* Convert from upper- to lowercase */
```

前面的例子使用一个嵌套的 if 语句检查两个条件，但如果需要检查的条件过多，就会引起混乱。C 语言允许使用逻辑运算符来简化这些检查。

3.1.8 逻辑运算符

有时执行一个测试不足以做出判断，而需要合并两个或多个检查，如果这些条件都是 true，才执行某个操作。或者如果一个或多个条件为 true，就执行一个计算。

例如，只有自己感觉良好，且当天是工作日，才去上班。仅感觉良好并不意味着要在周六或周日上班。另外，如果生病了或当天是周末，就可以呆在家中。这些都需要使用逻辑运算符。

www.TopSage.com

1. 逻辑与运算符&&

逻辑与运算符&&是一个二元运算符，因为它处理两个数据项。&&运算符合并两个逻辑表达式，即两个值为 true 或 false 的表达式。考虑下面的表达式：

```
Test1 && Test2
```

如果两个表达式 Test1 和 Test2 都等于 true，这个表达式就等于 true。如果运算符&&的一个或两个操作数是 false，该操作的结果就是 false。使用&&运算符的一个场合是 if 表达式。下面是一个例子：

```
if(age > 12 && age < 20)
  printf("You are officially a teenager.");
```

只有 age 的值在 13~19 之间(包含 13 和 19)，才执行 printf()语句。当然，&&运算符的操作数也可以是_Bool 变量。前面的语句可以替换为：

```
_Bool test1 = age > 12;
_Bool test2 = age < 20;
if(test1 && test2)
  printf("You are officially a teenager.");
```

两个检查 age 值的逻辑表达式的结果存储在变量 test1 和 test2 中。if 表达式现在比使用_Bool 变量作为操作数的情形简单得多。自然，也可以在一个表达式中使用多个这样的逻辑运算符：

```
if(age > 12 && age < 20 && savings > 5000)
  printf("You are a rich teenager.");
```

上面三个条件都必须是 true，printf()才会执行。即只有 age 的值在 13~19 之间(包含 13 和 19)，且 savings 的值大于 5000，才会执行 printf()。

2. 逻辑或运算符||

逻辑或运算符||用于两个或多个条件为 true 的情形。如果运算符||的一个或两个操作数是 true，其结果就是 true。只有两个操作数都是 false，结果才是 false。下面是使用这个运算符的例子：

```
if(a < 10 || b > c || c > 50)
  printf("At least one of the conditions is true.");
```

三个条件 a<10、b>c 和 c<50 中至少有一个是 true，就执行 printf()。例如，当 a、b 和 c 的值都是 9 时，就执行 printf()。当然，当三个条件中有两个或三个是 true 时，也会执行 printf()。

可以合并使用&&和||运算符，如下面的代码所示：

```
if((age > 12 && age < 20) || savings > 5000)
  printf ("Either you're a teenager, or you're rich, or possibly both.");
```

www.TopSage.com

如果 age 的值在 13~19 之间(包含 13 和 19)，或者 savings 的值大于 5000，就执行 printf() 语句。可以看出，在开始使用更多的运算符时，事情就变得复杂起来。在||运算符的左操作数中，表达式外面的括号并不是必要的，但加上括号，可以使条件更容易理解。使用布尔变量是有帮助的。可以用下面的代码替换上面的语句：

```
bool age_test1 = age > 12;
bool age_test2 = age < 20;
bool age_check = test1 && test2;
bool savings_check = savings > 5000;
if((age_check || savings_check)
  printf ("Either you're a teenager, or you're rich, or possibly both.");
```

这里使用 bool 声明了 4 个布尔变量，假定在源文件中包含了<stdbool.h>头文件。if 语句的工作方式与前面的测试相同。当然，也可以在一步中定义 age_check 的值，如下所示：

```
bool age_check = age > 12 && age < 20;
bool savings_check = savings > 5000;
if((age_check || savings_check)
  printf ("Either you're a teenager, or you're rich, or possibly both.");
```

这减少了要使用的变量个数，且代码仍很清晰。

3. 逻辑非运算符!

最后一个是逻辑非运算符，用“!”表示。! 运算符是一元运算符，因为它只有一个操作数。逻辑非运算符翻转逻辑表达式的值，使 true 变成 false，false 变成 true。假定有两个变量 a 和 b，其值分别是 5 和 2，则表达式 a>b 是 true。如果使用逻辑非运算符，表达式!(a>b)就是 false。尽量避免使用这个运算符，它会使代码难以理解。为了说明尽量避免使用逻辑非运算符的原因，下面重写前面的例子：

```
if((!(age >= 12) && !(age >= 20)) || !(savings <= 5000))
{
  printf("\nYou're either not a teenager and rich ");
  printf("or not rich and a teenager,\n");
  printf("or neither not a teenager nor not rich.");
}
```

可以看出，很难理解这些“!”的含义。

试试看：转换字母的一种更好方式

在本章前面的一个程序中，提示用户输入一个大写字母。程序使用一个嵌套的 if 语句确定输入值的类型正确，再在命令行上输出对应的小写字母，或者输出一条信息，说明输入的类型错误。

其中这些都不是必要的，因为使用下面的代码可以得到相同的结果：

```
/* Program 3.5 Testing letters the easy way */
```

www.TopSage.com

```
#include <stdio.h>
int main(void)
{
  char letter =0;                          /* Stores an input character */

  printf("Enter an upper case letter:"); /* Prompt for input */
  scanf(" %c", &letter); /* Read the input character */

  if((letter >= 'A') && (letter <= 'Z')) /* Verify uppercase letter */
  {
    letter += 'a'-'A';                     /* Convert to lowercase */
    printf("You entered an uppercase %c.\n", letter);
  }
  else
    printf("You did not enter an uppercase letter.\n");
  return 0;
}
```

输出与前面的例子类似。

代码的说明

输出类似于前面的程序，但不完全相同。在程序的这个更正版本中，当输入小于'A'时，将输出另一条信息。这个版本更好。比较一下两个程序中测试输入的机制，就可以看出第二个解决方案好在哪里了。下面是原来的版本:

```
if(letter >= 'A')
  if(letter <= 'Z')
```

下面是新版本:

```
if((letter >= 'A') && (letter <= 'Z')) /* Verify uppercase letter */
```

新版本不是使用容易混淆的嵌套if语句，而是在一条语句中检查输入的字符是否大于'A'、且小于'Z'。注意，在要检查的两个表达式外面都添加了额外的括号，它们不是必要的，但没有坏处，可以使程序员对执行顺序没有疑义。

转换为小写还有一种更简单的表达方式:

```
letter += 'a'-'A'; /* Convert to lowercase */
```

这里使用+=运算符将'a'和'A'之差加到 letter 存储的字符码中。如果在源文件中给<ctype.h>头文件添加了#include 指令，就可以使用 tolower()函数进行转换:

```
letter = tolower(letter);
```

tolower()函数返回的小写字母存储在 letter 变量中。在<ctype.h>中声明的 toupper()函数将参数转换为大写。

www.TopSage.com

3.1.9 条件运算符

条件运算符可用于测试数据，它根据一个逻辑表达式等于 true 还是 false，执行两个表达式中的一个。由于涉及到三个操作数——一个逻辑表达式和另外两个表达式——因此这个运算符也称为三元运算符。使用条件运算符的表达式的一般形式如下：

```
condition ? expression1 : expression2
```

注意运算符和操作数的相对位置。逻辑表达式 condition 后面的?将逻辑表达式与下一个操作数 expression1 分开，这个运算符用一个冒号与第三个操作数 expression2 分开。如果 condition 等于 true，就计算 expression1，生成该操作的结果，如果 condition 等于 false，就计算 expression2，生成该操作的结果。注意只计算 expression1 和 expression2 中的一个。一般情况下这不是很重要，但有时很重要。

可以在一条语句中使用条件运算符，如下：

```
x = y > 7 ? 25 : 50;
```

执行这条语句，如果 y 大于 7，x 就设置为 25，否则，x 就设置为 50。这是生成这一结果的一种快捷方式：

```
if(y > 7)
  x = 25;
else
  x = 50;
```

条件运算符可以简明地表达某些理念。使用它可以非常简单地编写出计算两个变量中较小值的表达式。例如，编写如下表达式，比较两份薪水，找出其中较大的那个：

```
your_salary > my_salary ? your_salary : my_salary
```

当然，可以在比较复杂的表达式中使用条件运算符。在前面的程序 3.2 中，曾使用 if-else 语句计算某产品的总价。该产品的单价是$3.50，当数量超过 10 时，提供 5%的折扣。使用条件运算符可以在一步中完成这个计算：

```
total_price = unit_price*quantity*(quantity > 10 ? 1.0 : 0.95);
```

试试看：使用条件运算符

这个折扣业务可以转换为一个小例子。假定产品的单价仍是$3.50，但提供三个级别的折扣：数量超过 50，折扣为 15%；数量超过 20，折扣为 10%；数量超过 10，折扣为 5%。下面是代码：

```
/* Program 3.6 Multiple discount levels */
#include <stdio.h>

int main(void)
{
```

www.TopSage.com

```
  const double unit_price = 3.50; /* Unit price in dollars */
  const double discount1 = 0.05;  /* Discount for more than 10 */
  const double discount2 = 0.1;   /* Discount for more than 20 */
  const double discount3 = 0.15;  /* Discount for more than 50 */
  double total_price = 0.0;
  int quantity = 0;

  printf("Enter the number that you want to buy:");
  scanf(" %d", &quantity);

  total_price = quantity*unit_price*(1.0 -
    (quantity > 50 ? discount3 : (
      quantity > 20 ? discount2 : (
        quantity > 10 ? discount1 : 0.0))));

  printf("The price for %d is $%.2f\n", quantity, total_price);
  return 0;
}
```

程序的输出如下:

```
Enter the number that you want to buy:60
The price for 60 is $178.50
```

代码的说明

比较有趣的是根据输入的数量计算产品总价的语句。该语句使用了三个条件运算符,所以有点难以理解:

```
total_price = quantity*unit_price*(1.0 -
  (quantity > 50 ? discount3 : (
    quantity > 20 ? discount2 : (
      quantity > 10 ? discount1 : 0.0))));
```

把它分解为各个部分,就容易理解它是如何得出正确结果的。总价是用表达式quantity*unit_price计算出来的,它只是将单价乘以订购数量。其结果必须乘以由数量决定的折扣因子。如果数量超过50,总价就必须乘以(1.0 - discount3),这用下面的表达式确定:

```
(1.0 - quantity > 50 ? discount3 : something_else)
```

这里,如果quantity大于50,表达式就乘以(1.0 - discount3),完成赋值运算符右边的计算。否则,表达式就乘以(1.0 - something_else),其中something_else是另一个条件运算符的结果。

当然,如果quantity不大于50,但仍大于20,something_else就应设置为discount2,这是由something_else所在的条件运算符决定的。

```
(quantity > 20 ? discount2 : something_else_again)
```

www.TopSage.com

如果 quantity 的值超过 20，something_else 就设置为 discount2，否则，就设置为 something_else_again。如果 quantity 超过 10，就把 something_else_again 设置为 discount1，否则就设置为 0。位于 something_else_again 的最后一个条件运算符如下所示：

```
(quantity > 10 ? discount1 : 0.0)
```

尽管其形式比较古怪，但条件运算符在 C 程序中使用得很频繁。这个运算符的一个方便应用是根据表达式的值改变信息的内容或提示信息。例如，如果要显示一条信息，指出某人拥有的宠物数，同时希望信息自动显示单词的单复数，就可以编写如下代码：

```
printf("You have %d pet%s.", pets, pets == 1 ? "" : "s" );
```

在输出一个字符串时，使用%s 指定符。如果 pets 等于 1，就在%s 的位置输出一个空字符串。否则就输出 “s”。因此，如果 pets 的值是 1，该语句就输出如下信息：

```
You have 1 pet.
```

如果 pets 变量是 5，就得到如下输出：

```
You have 5 pets.
```

使用这个机制可以根据表达式的值，以许多不同的方式修改输出的信息：she 代替 he，wrong 代替 right 等。

3.1.10 运算符的优先级

本章的例子都使用了括号，下面该探讨运算符的优先级了。运算符的优先级确定了表达式中运算符的执行顺序。运算符包括逻辑运算符&&、!和||，以及比较运算符和算术运算符。表达式中有多个运算符时，如何确定哪个运算符先执行？优先级顺序对表达式的结果有很大的影响。

例如，假定要处理求职申请，只接受 25 岁以上、毕业于哈佛或耶鲁的求职者。年龄条件可以用下面的条件表达式表示：

```
Age >= 25
```

假定毕业条件用变量 Yale 和 Harvard 表示，这两个变量可以是 true 或 false。现在可以把该条件编写为：

```
Age >= 25 && Harvard || Yale
```

可惜，这会带来许多抗议，因为这个表达式只接受 25 岁以下、毕业于耶鲁的求职者。事实上，这个语句会接受任意年龄的耶鲁毕业生。但如果求职者来自哈佛，就必须超过 25 岁。由于运算符有优先级，所以这个表达式的含义如下：

```
(Age >= 25 && Harvard) || Yale
```

所以它接受任意年龄的耶鲁毕业生。耶鲁毕业生会声称就应该使用这个表达式，但

www.TopSage.com

我们真正需要的是：

```
Age >= 25 && (Harvard || Yale)
```

由于运算符有优先级，所以必须加上括号，使操作按照我们希望的顺序执行。

一般情况下，表达式中运算符的优先级确定了是否需要加括号，才能得到希望的结果。但如果不知道运算符的优先级，加上括号也是无害的。表 3-2 列出了 C 语言中所有运算符的优先级，优先级最高的运算符排在最前面，优先级最低的运算符排在最后面。

表中有许多运算符都没有介绍过。本章后面的“按位运算符”一节将介绍运算符~、<<、>>、&、^和|。

表中同一行的所有运算符有相同的优先级。优先级相同的运算符的执行顺序由它们的相关性确定，相关性确定了运算符是从左至右还是从右至左执行。表达式中的括号一般是运算符列表中优先级最高的，因为它们用于重写已确定的优先级。

从表 3-2 可以看出，所有比较运算符的优先级都低于二元算术运算符，二元逻辑运算符的优先级低于比较运算符。因此，先执行算术运算，再比较，之后执行逻辑操作。赋值是列表中的最后一个，所以它们在其他运算都完成后执行。条件运算符的优先级高于赋值运算符。

注意！运算符在逻辑运算符中的优先级最高。因此，翻转逻辑表达式的值时，逻辑表达式外面的括号是必须的。

表 3-2 运算符的优先级

运 算 符	说 明	匹 配 规 则
()	带括号的表达式	从左至右
[]	数组下标	
.	按对象选择成员	
->	按指针选择成员	
+ -	一元+和 -	从右至左
++ --	前缀递增和前缀递减	
! ~	逻辑非和按位补	
*	取消引用	
&	寻址	
Sizeof	表达式或类型的字节数	
(type)	强制转换为 type，例如(int)或(double)，类型转换如(int) 或(double)	
* / %	乘、除、取模(取余数)	从左至右
+ -	加、减	从左至右
<< >>	按位左移、按位右移	从左至右
< <=	小于、小于等于	从左至右

www.TopSage.com

(续表)

运 算 符	说 明	匹 配 规 则
> >=	大于、大于等于	从左至右
== !=	等于、不等于	从左至右
&	按位与	从左至右
^	按位异或	从左至右
|	按位或	从左至右
&&	逻辑与	从左至右
||	逻辑或	从左至右
?:	条件运算符	从右至左
=	赋值	从右至左
+= -=	加法赋值、减法赋值	
/= *=	除法赋值、乘法赋值	
%=	取模赋值	
<<= >>=	按位左移赋值、按位右移赋值	
&= |=	按位与赋值、按位或赋值	
^=	按位异或赋值	
,	逗号运算符	从左至右

试试看：清楚地使用逻辑运算符

假定程序要为一家大型药厂面试求职者。该程序给满足某些教育条件的求职者提供面试机会。满足如下条件的求职者会接到面试通知：

(1) 25 岁以上，化学专业毕业生，但不是毕业于耶鲁。

(2) 耶鲁大学化学专业毕业生。

(3) 28 岁以下，哈佛大学经济学专业毕业生。

(4) 25 岁以上，耶鲁大学非化学专业毕业生。

实现该逻辑的程序如下：

```
/* Program 3.7 Confused recruiting policy */
#include <stdio.h>

int main(void)
{
  int age = 0;              /* Age of the applicant */
  int college = 0;          /* Code for college attended */
  int subject = 0;          /* Code for subject studied */
  bool interview = false;  /* true for accept, false for reject */

  /* Get data on the applicant */
```

www.TopSage.com

```
    printf("\nWhat college? 1 for Harvard, 2 for Yale, 3 for other: ");
    scanf("%d",&college);
    printf("\nWhat subject? 1 for Chemistry, 2 for economics,
          3 for other: ");
    scanf("%d", &subject);
    printf("\nHow old is the applicant? ");
    scanf("%d",&age);

    /* Check out the applicant */
    if((age>25 && subject==1) && (college==3 || college==1))
      interview = true;
    if(college==2 &&subject ==1)
      interview = true;
    if(college==1 && subject==2 && !(age>28))
      interview = true;
    if(college==2 && (subject==2 || subject==3) && age>25)
      interview = true;

    /* Output decision for interview */
    if(interview)
      printf("\n\nGive 'em an interview");
    else
      printf("\n\nReject 'em");
    return 0;
}
```

这个程序的输出如下:

```
What college? 1 for Harvard, 2 for Yale, 3 for other: 2
What subject? 1 for Chemistry, 2 for Economics, 3 for other: 1
How old is the applicant? 24

Give 'em an interview
```

代码的说明

这个程序非常简单。略复杂的仅是运算符的数量和需要找出候选人的 if 语句:

```
if((age>25 && subject==1) && (college==3 || college==1))
  interview =true;
if(college==2 &&subject ==1)
  interview = true;
if(college==1 && subject==2 && !(age>28))
  interview = true;
if(college==2 && (subject==2 || subject==3) && age>25)
  interview = true;
```

最后一个 if 语句指定是否要给求职者提供面试机会，它使用变量 interview:

```
if(interview)
  printf("\n\nGive 'em an interview");
```

www.TopSage.com

```
else
  printf("\n\nReject 'em");
```

变量 interview 初始化为 false，但如果满足其中一个条件，就给它赋予 true。if 表达式仅包含 interview 变量，所以当 interview 是 0 时，表达式就是 false，当 interview 是非零值时，表达式就是 true。

还可以更简单一些。下面看看获得面试机会的条件。每个条件都用一个表达式来表示，如表 3-3 所示：

表 3-3 选择候选人的表达式

条　件	表 达 式
25 岁以上，化学专业毕业生，但不是毕业于耶鲁	age>25 && college!=2
耶鲁大学化学专业毕业生	college==2 && subject==1
28 岁以下，哈佛大学经济学专业毕业生	college==1 && subject==2 && age<=28
25 岁以上，耶鲁大学非化学专业毕业生	college==2 && age>25 && subject!=1

只要 4 个表达式中的任意一个为 true，interview 变量就设置为 true。所以可以使用||运算符合并它们，设置 interview 变量的值：

```
interview = (age>25 && college!=2) || (college==2 && subject==1) ||
    (college==1 && subject==2 && age<=28) ||
       (college==2 && age>25 && subject!=1);
```

现在根本不需要 if 语句来检查条件，而只需存储合并这些表达式的逻辑结果 true 或 false。事实上，还可以将合并的表达式放在最后一个 if 中，删除变量 interview：

```
if((age>25 && college!=2) || (college==2 && subject==1) ||
    (college==1 && subject==2 && age<=28) ||
       (college==2 && age>25 && subject!=1))
  printf("\n\nGive 'em an interview");
else
  printf("\n\nReject 'em");
```

程序短了许多，但可读性略差。

3.2 多项选择问题

在编程时，常常会遇到多项选择问题。例如根据候选人是否来自 6 所不同大学中的一所，来选择一组不同的动作。另一个例子是根据某一天是星期几，来执行某组语句。在 C 语言中，有两种方式处理多项选择问题。一种是采用 else-if 形式的 if 语句，这是处理多项选择的最常见方式。另一种是 switch 语句，它限制了选择某个选项的方式，但在使用 switch 语句的场合中，它提供了一种非常简洁且便于理解的解决方案。下面先介绍

www.TopSage.com

else-if 语句。

3.2.1 给多项选择使用 else-if 语句

从一组选项中选择一项的 else-if 语句如下：

```
if(choice1)
  /* Statement or block for choice 1 */
else if(choice2)
  /* Statement or block for choice 2 */
else if(choice3)
  /* Statement or block for choice 2 */

/* ··· and so on ··· */
else
  /* Default statement or block */
```

每个 if 表达式可任意组成，只要其结果是 true 或 false 即可。如果第一个 if 表达式 choice1 是 false，就执行下一个 if。如果 choice2 是 false，就执行下一个 if。继续下去，直到找到一个结果为 true 的表达式为止。此时，就执行该 if 中的语句或语句块。然后结束这个执行序列，执行 else-if 语句后面的语句块。

如果所有的 if 条件都是 false，就执行最后一个 else 后面的语句或语句块。可以忽略这个 else，此时，如果所有的 if 条件都是 false，这个 else-if 语句序列就什么也不做。下面是一个例子：

```
if(salary<5000)
  printf("Your pay is very poor.");    /* pay < 5000 */
else if(salary<15000)
  printf("Your pay is not good.");     /* 5000 <= pay < 15000 */
else if(salary<50000)
  printf("Your pay is not bad.");      /* 15000 <= pay < 50000 */
else if(salary<100000)
  printf("Your pay is very good.");    /* 50000 <= pay < 100000 */
else
  printf("Your pay is exceptional."); /* pay > 100000 */
```

注意，在第一个 if 语句后，不需要测试 if 条件中的下限，因为如果执行到某个 if，前面的测试就一定是 false。

任意逻辑表达式都可以用作 if 条件，所以这个语句非常灵活，可以从任意多个选项中选择一项。switch 语句没有这么灵活，但在许多情况下使用起来更简单。下面看看 switch 语句。

3.2.2 switch 语句

switch 语句允许根据一个整数表达式的结果，从一组动作中选择一个动作。下面用

www.TopSage.com

一个简单的例子来说明其工作原理。假定在一家彩票销售点，数字 35 可赢得一等奖，数字 122 可赢得二等奖，数字 78 可赢得三等奖。使用 switch 语句可以检查购买彩票者是否获奖：

```
switch(ticket_number)
{
  case 35:
    printf("Congratulations! You win first prize!");
    break;
  case 122:
    printf("You are in luck - second prize.");
    break;
  case 78:
    printf("You are in luck - third prize.");
    break;
  default:
    printf("Too bad, you lose.");
}
```

在关键字 switch 的后面，括号中表达式的值是 ticket_number，它确定执行括号中的哪些语句。如果 ticket_number 的值与某个 case 关键字后面的指定值匹配，就执行该 case 后面的语句。例如，如果 ticket_number 的值是 122，就显示如下信息：

```
You are in luck - second prize.
```

printf()后面的 break 语句的作用是跳过括号中的其他语句，执行闭括号后面的语句。如果省略了某个 case 后面的 break 语句，就继续执行下一个 case 的语句。如果 ticket_number 的值不对应任何一个 case 值，就执行 default 关键字后面的语句，生成默认的信息。default 和 break 都是 C 语言中的关键字。switch 语句的一般形式如下：

```
switch(integer_expression)
{
  case constant_expression_1:
    statements_1;
    break;
    ....
  case constant_expression_n:
    statements_n;
    break;
  default:
    statements;
}
```

上述代码的测试基于 integer_expression 的值。如果该值对应于相关值 constant_expression_n 定义的某个 case 值，就执行该 case 值后面的语句。如果 integer_expression 的值不同于所有的 case 值，就执行 default 后面的语句。我们无法选择多个 case，所以所有的 case 值都必须互不相同。否则，在编译程序时就会得到一个错

www.TopSage.com

误信息。case 值必须是常量表达式，即可以在编译期间计算的表达式，这意味着 case 值不能依赖程序执行时确定的值。当然，测试表达式可以是任意的，只要它等于某个整数即可。

可以忽略 default 关键字及其相关的语句。如果没有 case 值匹配，就什么也不做。但要注意，constant_expression 对应的所有 case 值必须互不相同。break 语句会跳转到闭括号后面的语句上。

注意标点符号和格式。在第一个 switch 表达式的结尾处没有分号。语句体用括号括起来。case 的 constant_expression 值后跟一个冒号，后面的每条语句都以分号结束，这与一般的语句相同。

enumeration 类型是整数类型，所以可以使用 enumeration 类型的变量控制 switch。下面是一个例子：

```
enum Weekday {Monday, Tuesday, Wednesday,
        Thursday, Friday, Saturday, Sunday};
enum Weekday today = Wednesday;
switch(today)
{
  case Sunday:
    printf("Today is Sunday.");
    break;
  case Monday:
    printf("Today is Monday.");
    break;
  case Tuesday:
    printf("Today is Tuesday.");
    break;
  case Wednesday:
    printf("Today is Wednesday.");
    break;
  case Thursday:
    printf("Today is Thursday.");
    break;
  case Friday:
    printf("Today is Friday.");
    break;
  case Saturday:
    printf("Today is Saturday.");
    break;
}
```

这个 switch 语句选择对应于 today 变量值的 case，在本例中，显示的信息是“Today is Wednesday”。在这个 switch 语句中没有默认的 case，但可以添加一个，以防止出现 today 的无效值。

可以把多个 case 值与一组语句联系起来。还可以使用计算结果为 char 值的表达式作为 switch 的控制表达式。假定从键盘上将一个字符读入 char 类型的变量 ch 中，在 switch

www.TopSage.com

语句中测试这个字符，如下所示：

```
switch(tolower(ch))
{
  case 'a': case 'e': case 'i': case 'o': case 'u':
    printf("The character is a vowel.");
    break;
  case 'b': case 'c': case 'd': case 'f': case 'g': case 'h': case 'j':
  case 'k':case 'l': case 'm': case 'n': case 'p': case 'q': case 'r':
  case 's': case 't': case 'v': case 'w': case 'x': case 'y': case 'z':
    printf("The character is a consonant.");
    break;
  default:
    printf("The character is not a letter.");
    break;
}
```

这里使用了在<ctype.h>头文件中声明的 tolower()函数，将 ch 的值转换为小写，所以只需测试小写字母。如果 ch 包含的字符码表示一个元音，就输出一条信息。有 5 个 case 值对应元音，对它们要执行相同的 printf()语句。同样，当 ch 包含辅音时，也输出一条相应的信息。如果 ch 包含的字符码既不是辅音，也不是元音，就执行默认的 case。

注意，默认 case 后面的 break 语句，它不是必要的，但也是有作用的。总是把一个 break 语句放在最后一个 case 的末尾，可以确保以后在末尾添加一个新的 case 时，switch 总是正确工作。

使用另一个在<ctype.h>头文件中声明的 isalpha()函数，可以简化这个 switch。如果作为参数传入的字符是字母，isalpha()函数就返回一个非零整数(true)，否则就返回 0(false)。因此，下面的代码会生成与前面 switch 相同的结果：

```
if(!isalpha(ch))
  printf("The character is not a letter.");
else
  switch(tolower(ch))
  {
    case 'a': case 'e': case 'i': case 'o': case 'u':
      printf("The character is a vowel.");
      break;
    default:
      printf("The character is a consonant.");
      break;
  }
```

if 语句测试 ch 是否不是字母，如果不是，就输出一条信息。如果 ch 是一个字母，switch 语句就测试它是元音还是辅音。5 个元音 case 值会生成一个结果，默认的 case 生成另一个结果。在执行 switch 语句时，ch 包含的是一个字母，所以如果 ch 不是元音，就一定是辅音。

除了前面介绍的 tolower()、toupper()和 isalpha()函数之外，<ctype.h>头文件还声明了

www.TopSage.com

其他几个函数来测试字符，如表 3-4 所示。

表 3-4 测试字符的函数

函 数	测 试 内 容
islower()	小写字母
isupper()	大写字母
isalnum()	大写或小写字母
iscntrl()	控制字符
isprint()	可打印字符，包括空格
isgraph()	可打印字符，不包括空格
isdigit()	十进制数字('0'~'9')
isxdigit()	十六进制数字('0'~'9'，'A'~'F', 'a'~'f'))
isblank()	标准空白字符(空格，'\t')
isspace()	空位字符(空格，'\n', '\t', '\v', '\r', '\f')
ispunct()	Isspace() 和 isalnum()返回 false 的可打印字符

如果这些函数找到了它们希望的字符，就返回一个非零整数(表示 true)，否则返回 0(false)。

下面用一个例子来演示 switch 语句。

试试看：选择幸运数字

这个例子假定，在抽奖活动中有三个幸运数字，参与者要猜测一个幸运数字，switch 语句会结束这个猜测过程，给出参与者可能赢得的奖励。

```
/* Program 3.8 Lucky Lotteries */
#include <stdio.h>

int main(void)
{
  int choice = 0; /* The number chosen */

  /* Get the choice input */
  printf("\nPick a number between 1 and 10 and you may win a prize! ");
  scanf("%d",&choice);

  /* Check for an invalid selection */
  if((choice>10) || (choice <1))
    choice = 11; /* Selects invalid choice message */

  switch(choice)
  {
    case 7:
```

www.TopSage.com

```
      printf("\nCongratulations!");
      printf("\nYou win the collected works of Amos Gruntfuttock.");
      break; /* Jumps to the end of the block */

    case 2:
      printf("\nYou win the folding thermometer-pen-watch-umbrella.");
      break; /* Jumps to the end of the block */

    case 8:
      printf("\nYou win the lifetime supply of aspirin tablets.");
      break; /* Jumps to the end of the block */

    case 11:
      printf("\nTry between 1 and 10. You wasted your guess.");
      /* No break - so continue with the next statement */

    default:
      printf("\nSorry, you lose.\n");
      break; /* Defensive break - in case of new cases */
  }
  return 0;
}
```

这个程序的输出如下:

```
Pick a number between 1 and 10 and you may win a prize! 3
Sorry, you lose.
```

或:

```
Pick a number between 1 and 10 and you may win a prize! 7
Congratulations!
You win the collected works of Amos Gruntfuttock.
```

如果输入无效数字:

```
Pick a number between 1 and 10 and you may win a prize! 92
Try between 1 and 10. You wasted your guess.
Sorry, you lose.
```

代码的说明

开始的代码与前面的程序相同，也是声明一个整型变量 choice，接着要求用户输入一个 1~10 之间的数字，把该值存储在 choice 中:

```
int choice = 0; /* The number chosen */

/* Get the choice input */
printf("\nPick a number between 1 and 10 and you may win a prize! ");
scanf("%d",&choice);
```

www.TopSage.com

在执行其他操作之前，检查用户是否输入了一个 1~10 之间的数字:

```
/* Check for an invalid selection */
if((choice>10) || (choice <1))
  choice = 11;                    /* Selects invalid choice message */
```

如果值不在 1~10 之间，就自动把它改为 11，这不是必要的，但为了确保用户不出错，把 choice 变量设置为 11，对于这个 case 值，printf()语句会生成错误信息。

接着是 switch 语句，它根据 choice 的值从括号之间的 case 中选择:

```
switch(choice)
{
  ...
}
```

如果 choice 的值是 7，就执行该值对应的 case:

```
case 7:
  printf("\nCongratulations!");
  printf("\nYou win the collected works of Amos Gruntfuttock.");
  break;          /* Jumps to the end of the block */
```

执行两个 printf()调用，之后 break 语句跳到闭括号后面的语句上(这里是结束程序，因为闭括号后面没有语句了)。

下面的两个 case 也是这样:

```
case 2:
  printf("\nYou win the folding thermometer-pen-watch-umbrella.");
  break;      /* Jumps to the end of the block */

case 8:
  printf("\nYou win the lifetime supply of aspirin tablets.");
  break;     /* Jumps to the end of the block */
```

它们对应于 choice 的值是 2 或 8 的情况。

下一个 case 有点不同:

```
case 11:
  printf("\nTry between 1 and 10, you wasted your guess.");
  /* No break - so continue with the next statement */
```

它没有 break 语句，所以在显示了信息后，继续执行默认 case 的 printf()。其结果是，如果 choice 设置为 11，会得到两行输出。这对于本例完全合适，但一般应在每个 case 的最后添加 break 语句。从程序中删除 break 语句，再输入 7，看看结果如何。每个 case 都会得到所有的输出信息。默认 case 如下:

```
default:
  printf("\nSorry, you lose.\n");
  break;          /* Defensive break - in case of new cases */
```

www.TopSage.com

如果 choice 的值不对应任何一个 case 值，就选择这个默认 case。这里也有一个 break 语句，尽管它是不必要的，但许多程序员仍总是把 break 语句放在默认 case 语句的后面，或者 switch 语句的最后一个 case 后面，这便于以后提供更多的 case 语句。如果忘记在默认 case 的后面加上 break 语句，switch 语句就不会按照希望的那样执行。switch 语句中的 case 顺序可任意，default 不一定是最后一个 case。

试试看：是或否

下面的 switch 语句由用户输入的 char 变量值来控制。程序提示用户为一个动作输入值'y'或'Y'，为另一个动作输入'n'或'N'。这个程序其实没有什么用，但许多程序常常会问一个问题，再执行每个动作(例如保存一个文件)。

```
/* Program 3.9 Testing cases */
#include <stdio.h>

int main(void)
{
  char answer = 0;      /* Stores an input character */

  printf("Enter Y or N: ");
  scanf(" %c", &answer);

  switch(answer)
  {
    case 'y': case 'Y':
      printf("\nYou responded in the affirmative.");
      break;

    case 'n': case 'N':
      printf("\nYou responded in the negative.");
      break;

    default:
      printf("\nYou did not respond correctly...");
      break;
  }
  return 0;
}
```

这个程序的输出如下：

```
Enter Y or N: y
You responded in the affirmative.
```

代码的说明

把 answer 变量声明为 char 类型时，还把它初始化为 0。接着要求用户输入一个值，并存储该值：

www.TopSage.com

```
char answer = 0;        /* Stores an input character */

printf("Enter Y or N: ");
scanf(" %c", &answer);
```

switch 语句使用存储在 letter 中的字符选择 case：

```
switch(answer)
{
  ...
}
```

switch 语句中的第一个 case 要求用户输入 Y 的大写字母或小写字母：

```
case 'y': case 'Y':
  printf("\nYou responded in the affirmative.");
  break;
```

输入值'y'和'Y'，都会执行相同的 printf()。通常，可以把任意多个这样的 case 组合在一起。注意其标点符号：两个 case 放在一起，用一个冒号隔开。

否定的输入以相同的方式处理：

```
case 'n': case 'N':
  printf("\nYou responded in the negative.");
  break;
```

如果输入的字符不对应所有的 case 值，就选择默认的 case：

```
default:
  printf("\nYou did not respond correctly...");
  break;
```

注意默认 case 的 printf()语句后面的 break 语句以及合法的 case 值。与以前一样，break 语句会使执行过程在此中断，并从 switch 语句后面的语句继续执行。另外，没有 break 语句，会执行后续 case 中的语句，除非有效 case 的前面有 break 语句，否则就会执行后面的语句(包括 default 语句)。

当然，也可以使用 toupper()或 tolower()函数简化 switch 中的 case。使用这两个函数之一，可以使 case 个数减半：

```
switch(toupper(answer))
{
  case 'Y':
    printf("\nYou responded in the affirmative.");
    break;
  case 'N':
    printf("\nYou responded in the negative.");
    break;
  default:
    printf("\nYou did not respond correctly...");
```

www.TopSage.com

```
    break;
}
```

如果使用 toupper()函数，就需要用#include 指令包含<ctype.h>。

3.2.3 goto 语句

If 语句允许根据测试的结果选择执行两个语句块中的一个。这是一个强大的工具，可以改变程序的自然执行顺序，程序不再从 A 执行到 B，再到 C 和 D，而可以执行到 A，再决定是否跳过 B 和 C，直接执行 D。

goto 语句是一个比较生硬的指令，它可以无条件地改变程序流——不必通过 Go，也不必缴纳$200，而是直接进监狱。程序在遇到 goto 语句时，也会无条件地跳转。goto 语句会直接跳到某个指定的位置，无须检查任何值，或者要求用户考虑这是否是他希望执行的操作。

这里仅简要介绍 goto 语句，因为它并不像初看起来那么强大。goto 语句的问题是它看起来太简单了，这似乎不太恰当，但重要的是“看起来”这个词。goto 语句很简单，可以使用它到达任何地方，其实此时使用另一个语句会更好。goto 语句会产生非常难以理解的代码。

在使用 goto 语句时，会跳转到代码中用语句标签指定的位置。语句标签的定义方式与变量名相同，也是一组字母和数字，其中第一个字符必须是字母。语句标签后跟一个冒号(:)，将它与它标记的语句分开。它看起来类似于 switch 中的 case 标签。case 标签就是语句标签。

与其他语句一样，goto 语句也用分号结束：

```
goto there;
```

目标语句必须有与 goto 语句相同的标签，在上面的例子中，该标签是 there。如前所述，标签写在它所应用的语句之前，其后的冒号将该标签和语句的其他部分隔开，如下面的例子所示：

```
there: x=10;        /* A labeled statement */
```

goto 语句可以与 if 语句一起使用，如下面的例子所示：

```
...
if(dice == 6)
  goto Waldorf;
else
  goto Jail;        /* Go to the statement labeled Jail */

Waldorf:
  comfort = high;
  ...
  /* Code to prevent falling through to Jail */
```

www.TopSage.com

```
Jail:              /* The label itself. Program control is sent here */
  comfort = low;
  ...
```

这段代码在掷骰子。如果掷出了 6，就跳转到 Waldorf，否则就跳转到 Jail。这似乎很不错，但它很容易出现混淆。要理解执行的顺序，需要找出目标标签。假定代码使用了许多 goto 语句，就很难理解，甚至在出错时都无法修改。所以应尽可能避免使用 goto 语句。在理论上，总是可以避免使用 goto 语句，但在一两种情况下，这是一个有用的选项。第 4 章在介绍循环时会提到，从嵌套了许多层循环的最内层中退出时，使用 goto 语句比采用其他机制简单得多。

3.3 按位运算符

在进入本章的大型示例之前，还要先学习一组运算符，它们看起来类似于前面介绍的逻辑运算符，但实际上与逻辑运算符完全不同。这些运算符称为按位运算符，因为它们操作的是整数值中的位。按位运算符有 6 个，如表 3-5 所示。

表 3-5 按位运算符

运 算 符	说 明
&	按位与运算符
|	按位或运算符
^	按位异或(EOR)运算符
~	按位非运算符，也称为 1 的补位运算符
<<	按位左移运算符
>>	按位右移运算符

这些运算符都只能用于整数类型。~运算符是一元运算符，只处理一个操作数，其他都是二元运算符。

按位与运算符&合并操作数的对应位，如果两个位都是 1，结果位就是 1，否则，结果位就是 0。假定声明了如下变量：

```
int x = 13;
int y = 6;
int z = x&y;      /* AND the bits of x and y */
```

在执行第三条语句后，z 的值是 4(二进制为 100)，因为 x 和 y 的对应位的合并过程如下：

```
x    0 0 0 0 1 1 0 1
y    0 0 0 0 0 1 1 0
x&y  0 0 0 0 0 1 0 0
```

www.TopSage.com

显然，变量的位数要比这里显示的多，但其他位都是 0。变量 x 和 y 的对应位都是 1 的情况只有从右数的第三位，所以只有这一位的按位与结果为 1。

警告：
千万不要混淆按位运算符和逻辑运算符。表达式 x & y 生成的结果完全不同于 x && y。

如果对应位中有一个或两个位是 1，按位或运算符|就生成 1，否则就生成 0。下面看一个例子。如果在一个语句中合并相同的值：

```
int z = x|y;        /* OR the bits of x and y */
```

结果如下：

```
x   0 0 0 0 1 1 0 1
y   0 0 0 0 0 1 1 0
x|y 0 0 0 0 1 1 1 1
```

z 存储的值是 15(二进制的 1111)。

如果两个位是不同的，按位异或运算符^就生成 1，否则就生成 0。再使用相同的初始值，语句：

```
int z = x^y;        /*Exclusive OR the bits of x and y */
```

会使 z 包含值 11(二进制的 1011)，因为位的合并如下：

```
x   0 0 0 0 1 1 0 1
y   0 0 0 0 0 1 1 0
x^y 0 0 0 0 1 0 1 1
```

一元运算符~会翻转其操作数的位，将 1 变成 0，0 变成 1。如果把这个运算符应用于值为 13 的变量 x，并编写如下语句：

```
int z = ~x;         /* Store 1's complement of x */
```

z 的值就是 14，位的设置如下：

```
x  0 0 0 0 1 1 0 1
~x 1 1 1 1 0 0 1 0
```

在负整数的 2 的补码中，值 1111 0010 是 14。如果不熟悉 2 的补码形式，可以参阅附录 A。

移位运算符会把左操作数的位移动右操作数指定的位数。使用下面的语句可以指定左移位操作：

```
int value = 12;
int shiftcount = 3;                 /* Number of positions to be shifted */
int result = value << shiftcount; /* Shift left shiftcount positions */
```

变量 result 的值是 96，其二进制为 0000 1100。现在把其中的位向左移动 3 位，在右

www.TopSage.com

边补入 0，所以 value << shiftcount 的二进制值是 0110 0000。

右移位运算符会向右移位，但它比左移位复杂一些。对于不带符号的数值，向右移位时，会在左边的空位中填充 0。下面用一个例子来说明。假定声明一个变量：

```
unsigned int value = 65372U;
```

在两字节的变量中，这个值的二进制形式为：

```
1111 1111 0101 1100
```

假定现在执行如下语句：

```
unsigned int result = value >> 2;     /* Shift right two bits */
```

value 中的位向右移动两位，在左边补入 0，得到的值存储在 result 中。在二进制中，其值为 0，在十进制中，其值为 16 343。

```
0011 1111 1101 0111
```

对于带符号的负值，其最左一位是 1，则移位的结果取决于系统。在大多数情况下，符号位会扩散，所以向右移位时补入的是 1，但在一些系统上，补入的是 0。下面看看这对结果有什么影响。

假定用下面的语句定义一个变量：

```
int new_value = -164;
```

其位模式与前面使用的无符号值相同，这是该值的 2 的补码：

```
1111 1111 0101 1100
```

执行如下语句：

```
int new_result = new_value >> 2;  /* Shift right two bits */
```

这行语句将 new_value 的值向右移动两位，结果存储在 new_result 中。在通常情况下，如果扩散符号位，在向右移位时将 1 插入左边的空位，new_result 的值就是：

```
1111 1111 1101 0111
```

其十进制值是 - 41，这是我们希望的结果，因为 - 164/4 的结果应是 - 41。但在一些计算机上，如果不扩散符号位，new_result 的值就是：

```
0011 1111 1101 0111
```

在本例中向右移动两位，会把值 - 164 变成+16 343，这是一个意想不到的结果。

3.3.1 按位运算符的 op=用法

所有的二元按位运算符都可以在 op=形式的赋值语句中使用，但~运算符例外，它是

www.TopSage.com

一元运算符。如第 2 章所述，如下形式的语句：

```
lhs op= rhs;
```

等价于：

```
lhs = lhs op (rhs);
```

这说明，如果编写如下语句：

```
value <<= 4;
```

其作用是将整数变量 value 的内容向右移动 4 位。该语句与下面的代码等效：

```
value = value << 4;
```

其他二元运算符也可以这样使用。例如，可以编写如下语句：

```
value &= 0xFF;
```

其中 value 是一个整数变量，这个语句等价于：

```
value = value & 0xFF;
```

其作用是使最右边的 8 位保持不变，其他的位都设置为 0。

3.3.2 使用按位运算符

从学术的角度来看，按位运算符很有趣，但它们用于什么场合？它们不用于日常的编程工作，但在一些领域非常有效。按位与&、按位或|运算符的一个主要用途是测试并设置整数变量中的各个位。此时可以使用各个位存储涉及二选一的数据。例如，可以使用一个整数变量存储一个人的几个特性。在一个位中存储这个人是男性还是女性，使用 3 个位指定这个人是否会说法语、德语或意大利语。再使用另一个位记录这个人的薪水是否多于$50 000。在这 4 个位中，都记录了一组数据。下面看看这是如何实现的。

只有两个位都是 1，结果才是 1，此时可以使用&运算符选择整数变量的一个部分，甚至可以选择其中的一个位。首先定义一个值，它一般称为掩码，用于选择需要的位。在掩码中，希望保持不变的位置上包含 1，希望舍弃的位置上包含 0。接着对这个掩码与要从中选择位的值执行按位与操作。下面看一个例子。下面的语句定义了掩码：

```
unsigned int male       = 0x1;   /* Mask selecting first (rightmost) bit */
unsigned int french     = 0x2;   /* Mask selecting second bit */
unsigned int german     = 0x4;   /* Mask selecting third bit */
unsigned int italian    = 0x8;   /* Mask selecting fourth bit */
unsigned int payBracket = 0x10;  /* Mask selecting fifth bit */
```

在每条语句中，1 位表示该条件是 true。这些二进制掩码都选择一个位，所以可以定义一个 unsigned int 变量 personal_data 来存储一个人的 5 项信息。如果第一位是 1，这个人就是男性，如果是 0，这个人就是女性。如果第二位是 1，这个人就说法语，如果是 0，

www.TopSage.com

这个人就不说法语，数据值右边的 5 位都是这样。

因此，可以给一个说德语的人测试变量 personal_data，如下面的语句所示：

```
if(personal_data & german)
  /* Do something because they speak German */
```

如果 personal_data 对应掩码 german 的位是 1，表达式 personalData & german 的值就不是 0(true)，否则就是 0。

当然，也可以通过逻辑运算符合并多个使用掩码的表达式，选择各个位。下面的语句测试某个人是否是女性，是说法语还是说意大利语：

```
if(!(personal_data & male) && ((personal_data & french) ||
       (personal_data & italian)))
  /* We have a French or Italian speaking female */
```

可以看出，测试单个位或位的组合是很简单的。另一个需要理解的操作是如何设置各个位。此时可以使用按位或(OR)运算符。按位或运算符与测试位的掩码一起使用，就可以设置变量中的各个位。如果要设置变量 personal_data，记录某个说法语的人，就可以使用下面的语句：

```
personal_data |= french;       /* Set second bit to 1 */
```

上面的语句与如下语句等效：

```
personal_data = personal_data|french; /* Set second bit to 1 */
```

personal_data 中从右数的第二位设置为 1，其他位都不变。利用|运算符的工作方式，可以在一条语句中设置多个位：

```
personal_data |= french|german|male;
```

这条语句设置的位记录了一个说法语和德语的男子。如果变量 personal_data 以前曾记录这个人也说意大利语，则这一位仍会设置为 1，所以 OR 运算符是相加的。如果某个位已经设置为 1，它仍会设置为 1。

如何重置一个位？假定要将男性位设置为女性，这需要将一个位重置为 0，此时应使用!运算符和按位与(AND)运算符：

```
personal_data &= !male; /* Reset male to female */
```

这是可行的，因为!male 将表示男性的位设置为 0，其他位仍设置为 1。因此，对应于男性的位设置为 0，0 与任何值的与操作都是 0，其他位保持不变。如果另一个位是 1，则 1&1 仍是 1。如果另一个位是 0，则 0&1 仍是 0。

使用位的例子记录了个人数据的特定项。如果要使用 Windows 应用程序编程接口(API)编写 PC 程序，就会经常使用各个位来记录各种 Windows 参数的状态，在这种情况下，按位运算符非常有用。

www.TopSage.com

试试看：使用按位运算符

下面在一个略微不同的例子中使用一些按位运算符，但规则与前面相同。这个例子说明了如何使用掩码从变量中选择多个位。我们要编写的程序将在变量中设置一个值，再使用按位运算符翻转十六进制数字的顺序。下面是代码：

```
/* Program 3.10 Exercising bitwise operators */
#include <stdio.h>

int main(void)
{
  unsigned int original = 0xABC;
  unsigned int result = 0;
  unsigned int mask = 0xF; /* Rightmost four bits */

  printf("\n original = %X", original);

  /* Insert first digit in result */
  result |= original&mask; /* Put right 4 bits from original in result */

  /* Get second digit */
  original >>= 4;          /* Shift original right four positions */
  result <<= 4;            /* Make room for next digit */
  result |= original&mask; /* Put right 4 bits from original in result */

  /* Get third digit */
  original >>= 4;          /* Shift original right four positions */
  result <<= 4;            /* Make room for next digit */
  result |= original&mask; /* Put right 4 bits from original in result */
  printf("\t result = %X\n", result);
  return 0;
}
```

输出如下：

```
original = ABC result = CBA
```

代码的说明

这个程序使用了前面探讨的掩码概念。original 中最右边的十六进制数是通过表达式 original & mask 将 original 和 mask 的值执行按位与操作而获得的。这会把其他十六进制数设置为 0。因为 mask 的值的二进制形式为：

```
0000 0000 0000 1111
```

可以看出，只有右边的 4 位没有改变。在 original 中，这 4 位都是 1，在执行按位与操作的结果中，这 4 位仍是 1，其他位都是 0。这是因为 0 与任何值执行按位与操作，结果都是 0。选择了右边的 4 位后，用下面的语句存储结果：

```
result |= original&mask; /* Put right 4 bits from original in result */
```

www.TopSage.com

result 的内容与右边表达式生成的十六进制数进行或操作。为了获得 original 中的第二位，需要把它移动到第一个数字所在的位置。为此将 original 向右移动 4 位：

```
original >>= 4;          /* Shift original right four positions */
```

第一个数字被移出，且被舍弃。为了给 original 的下一个数字腾出空间，下面的语句将 result 的内容向左移动 4 位：

```
result <<= 4;            /* Make room for next digit */
```

现在要在 result 中插入 original 中的第二个数字，而当前这个数字在第一个数字的位置上，使用下面的语句：

```
result |= original&mask; /* Put right 4 bits from original in result */
```

要得到第三个数字，重复上述过程。显然，可以对任意多个数字重复这个过程。

3.4 设计程序

在第 3 章的最后，要应用所学的内容，建立一个有用的程序。

3.4.1 问题

我们要编写一个简单的计算器，进行加、减、乘、除操作，在执行除操作时，还要确定其余数。这个程序必须能以自然的方式进行计算，例如 5.6 * 27 或 3 + 6。

3.4.2 分析

本程序涉及的所有数学知识都很简单，但输入过程会增加复杂性。我们需要检查输入，确保用户没有要求计算机完成不可能的任务。还必须允许用户一次输入一个计算式，例如：

```
34.87 + 5
```

或者

```
9 * 6.5
```

编写这个程序的步骤如下：

(1) 获得用户要求计算机执行计算所需的输入。
(2) 检查输入，确保输入是可以理解的。
(3) 执行计算。
(4) 显示结果。

www.TopSage.com

3.4.3 解决方案

本节列出解决该问题的步骤。

1. 步骤 1

获得用户输入是很简单的，可以使用 printf()和 scanf()，所以需要<stdio.h>头文件。这里介绍的唯一的新知识是获得输入的方式。如前所述，可以让用户更自然地输入每个数字和要执行的操作，而不是逐个输入它们。可以这么做是因为 scanf()允许这么做，这里在列出程序的第一部分之后讨论其细节。下面是读取输入的程序代码：

```
/*Program 3.11 A calculator*/
#include <stdio.h>

int main(void)
{
  double number1 = 0.0;  /* First operand value a decimal number */
  double number2 = 0.0;  /* Second operand value a decimal number */
  char operation = 0;    /* Operation - must be +, -, *, /, or % */

  printf("\nEnter the calculation\n");
  scanf("%lf %c %lf", &number1, &operation, &number2);

  /* Plus the rest of the code for the program */
  return 0;
}
```

scanf()函数在读取数据方面相当聪明。其实并不需要在一行上输入每个数据项，只要在输入的每一项之间流出一个或多个空白即可。(按空格键、Tab 键或回车键，都可以创建空白字符)。

2. 步骤 2

接着，检查输入是否正确。最明显的检查是要执行的操作是否有效。有效的操作有+、-、/和 %，所以需要检查输入的操作是否是其中的一个。

还需要检查第二个数字，如果操作是/或 %，第二个数字就不能是 0。如果右操作数是 0，这些操作就是无效的。这些操作都可以使用 if 语句来完成，switch 语句则为此提供了一种更好的方式，因为它比一系列 if 语句更容易理解。

```
/*Program 3.11 A calculator*/
#include <stdio.h>

int main(void)
{
  double number1 = 0.0; /* First operand value a decimal number */
  double number2 = 0.0; /* Second operand value a decimal number */
```

www.TopSage.com

```
char operation = 0; /* Operation - must be +, -, *, /, or % */

printf("\nEnter the calculation\n");
scanf("%lf %c %lf", &number1, &operation, &number2);

/* Code to check the input goes here */
switch(operation)
{
  case '+':          /* No checks necessary for add */
    break;

  case '-':          /* No checks necessary for subtract */
    break;

  case '*':          /* No checks necessary for multiply */
    break;

  case '/':
    if(number2 == 0) /* Check second operand for zero */
      printf("\n\n\aDivision by zero error!\n");
    break;

  case '%':          /* Check second operand for zero */
    if((long)number2 == 0)
      printf("\n\n\aDivision by zero error!\n");
    break;

  default:           /* Operation is invalid if we get to here */
    printf("\n\n\aIllegal operation!\n");
    break;
  }
  /* Plus the rest of the code for the program */
  return 0;
}
```

当运算符是%时，将第二个操作数转换为一个整数，所以仅检查第二个操作数是否为 0 是不够的，还必须检查 number2 在转换为 long 时，其值是否为 0。

3. 步骤 3 和 4

检查了输入后，就可以计算结果了。这里有一个选择。可以在 switch 中计算每个结果，存储它们，在执行完 switch 后输出它们，也可以在每个 case 中输出结果。这里采用第二种方式。需要添加的代码如下：

```
/*Program 3.11 A calculator*/
#include <stdio.h>

int main(void)
{
```

www.TopSage.com

```
double number1 = 0.0; /* First operand value a decimal number */
double number2 = 0.0; /* Second operand value a decimal number */
char operation = 0;   /* Operation - must be +, -, *, /, or % */

printf("\nEnter the calculation\n");
scanf("%lf %c %lf", &number1, &operation, &number2);

/* Code to check the input goes here */
switch(operation)
{
  case '+':           /* No checks necessary for add */
    printf("= %lf\n", number1 + number2);
    break;

  case '-':           /* No checks necessary for subtract */
    printf("= %lf\n", number1 - number2);
    break;

  case '*':           /* No checks necessary for multiply */
    printf("= %lf\n", number1 * number2);
    break;

  case '/':
    if(number2 == 0)  /* Check second operand for zero */
      printf("\n\n\aDivision by zero error!\n");
    else
      printf("= %lf\n", number1 / number2);
    break;

  case '%':           /* Check second operand for zero */
    if((long)number2 == 0)
      printf("\n\n\aDivision by zero error!\n");
    else
      printf("= %ld\n", (long)number1 % (long)number2);
    break;

  default:            /* Operation is invalid if we get to here */
    printf("\n\n\aIllegal operation!\n");
    break;
  }
  return 0;
}
```

注意，在执行取模运算时，将两个数字从 double 转换为 long。这是因为在 C 语言中，%运算符只能用于整数。剩下的就是试运行代码了，下面是输出：

```
Enter the calculation
25*13
= 325.000000
```

www.TopSage.com

下面是另一个例子：

```
Enter the calculation
999/3.3
= 302.727273
```

下面是另一个例子：

```
Enter the calculation
7%0

Division by zero error!
```

3.5 小结

本章用一个相当复杂的例子来结束。在前两章中，仅介绍了 C 程序的基础知识，读者能完成一些有用的工作，但程序一旦开始，就不能控制程序的执行顺序。本章开始体验 C 语言的强大，并学习在执行过程中使用用户输入的数据或计算的结果，来确定下一步的操作。

本章学习了如何比较变量，再使用 if、if-else、else-if 和 switch 语句来影响结果。还学习了如何使用逻辑运算符合并变量之间的比较操作。现在读者应对如何做出判断有了很多认识，能使程序代码沿着不同的路径执行。

下一章将学习如何编写更强大的程序：程序可以重复执行一些语句，直到满足某个条件为止。学习了第 4 章后，就会知道本章的计算器程序其实非常小。

3.6 练习

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容。还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 3.1　编写一个程序，首先给用户以下两种选择：

(1) 将温度从摄氏度转换为华氏度。

(2) 将温度从华氏度转换为摄氏度。

接着，程序提示用户输入温度值，并输出转换后的数值。从摄氏度转换为华氏度，可以乘以 1.8 再加上 32。从华氏度转换为摄氏度，可以先减去 32 后，再乘以 5，除以 9。

习题 3.2　编写一个程序，提示用户输入 3 个整数值，分别代表月、日、年。例如用户输入了 12、31、2003，程序就以 31st December 2003 的格式输出该日期。

必须在日期值的后面加上 th、nd、st 和 rd。例如 1st、2nd、3rd、4th、11th、12th、13th、14th、21st、22nd、23rd、24th。

习题 3.3　编写一个程序，根据从键盘输入的一个数值，计算总价(单价是$5)，数值

www.TopSage.com

超过30的折扣是10%，数值超过50的折扣是15%。

习题3.4 修改本章最后的计算器例子，让用户选择输入y或Y，以执行另一个计算，输入n或N就结束程序。(注意：这需要使用goto语句，下一章将介绍一个更好的方法。)

www.TopSage.com

www.TopSage.com

第 4 章

循　环

上一章学习了如何比较数据项，并根据其结果进行判断。我们可以根据程序的输入选择计算机如何做出反应。本章将介绍如何重复执行一个语句块，直到满足某个条件为止，这称为循环。

语句块的执行次数可以简单地用一个计数器来控制，语句块重复执行指定的次数，或者还可以更复杂一些，重复执行一个语句块，直到满足某个条件为止，例如用户输入 quit。后者可以编写上一章的计算器示例，使计算过程重复需要的次数，而不必使用 goto 语句。

本章的主要内容：

- 使语句或语句块重复执行指定的次数
- 重复执行语句或语句块，直到满足某个条件为止
- 使用 for、while 和 do-while 循环
- 递增和递减运算符的作用及其用法
- 编写一个简单的 Simon 游戏程序

TopSage.com

4.1　循环

如前所述，使一系列语句重复执行指定的次数，或重复执行它们，直到满足某个条件为止的编程机制称为循环。循环和比较数据项是基本的编程工具。能比较数据值和重复执行语句块后，就可以合并这两个功能，控制语句块的执行次数。例如，可以重复执行一个操作，直到比较的两个数据项相同为止。当它们相同时，就可以执行另一个操作。

在第 3 章的程序 3.8 中，抽奖示例允许用户猜 3 次，换言之，可以让用户继续猜测，直到 number_of_guesses 变量等于 3 为止。这就涉及一个循环，它重复执行代码，从键盘上读取猜测的数字，检查输入值的准确性。图 4-1 显示了循环的工作过程。

我们要经常对不同的数据值应用相同的计算。没有循环，那么要处理多少组数据值，就必须重复编写多少组相同的指令，这非常繁琐。循环可以使用相同的程序码，处理输入的任何多个数据。

在讨论 C 语言中的各种循环之前，首先介绍 C 程序中常见的两个新算术运算符：递增运算符和递减运算符。这两个运算符经常用在循环中，这就是在这里讨论它们的原因。我们先简要介绍递增运算符和递减运算符，再用一个例子说明如何在循环中使用它们。在能灵活运用循环后，再回过头来了解递增运算符和递减运算符的一些特质。

www.TopSage.com

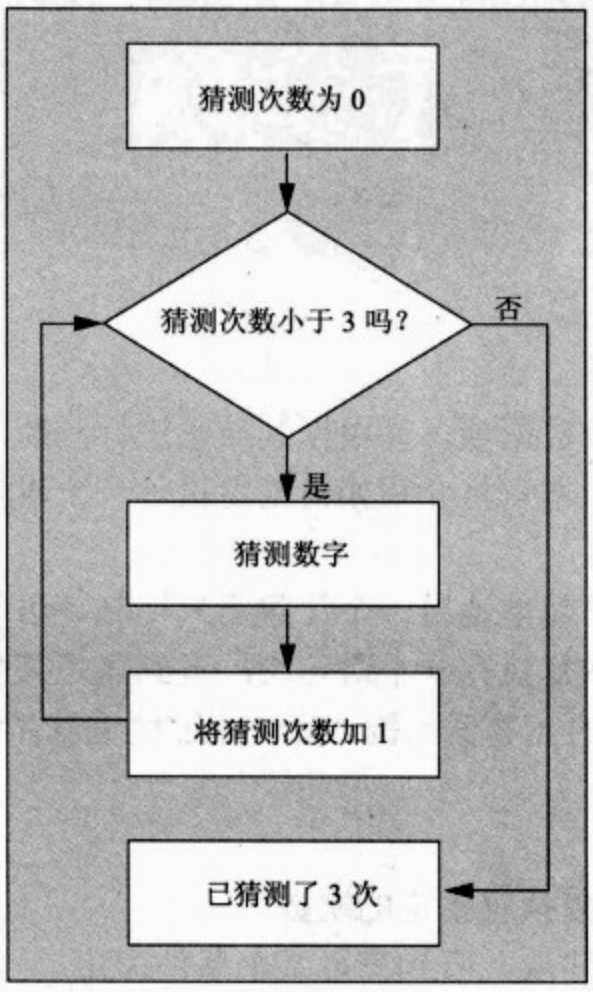


图 4-1　循环的工作过程

4.2　递增和递减运算符

递增运算符(++)和递减运算符(--)会将存储在整数变量中的值递增或递减 1。假设定义一个整数变量 number，它的当前值是 6。可以用下面的语句给它加 1：

```
++number; /* Increase the value by 1 */
```

执行完这个语句后，number 的值是 7。同样，可以用下面的指令给 number 减 1：

```
--number; /* Decrease the value by 1 */
```

这些运算符和前面介绍的其他算术运算符不一样。使用其他算术运算符时，会创建一个表达式，其计算结果为一个数值，该数值存储在一个变量中，或用作复杂表达式的一部分。它们不直接更改变量存储的值。假如 number 的值是+6，则表达式 - number 的结果是 - 6，但是存储在 number 中的值不会改变。而表达式--number 会更改 number 的值，这个表达式将 number 的值减 1，所以 number 的结果是 5。

递增和递减运算符还有更多需要了解的内容，这些将在后面讨论。现在回到主题，看看最简单的循环——for 循环。以后还会介绍其他类型的循环，这里用较多的篇幅介绍 for 循环，因为理解了这个循环，其他循环就容易掌握了。

www.TopSage.com

4.3 for 循环

使用 for 循环的基本形式可以使语句块重复执行指定的次数。假设要显示 1~10 之间的数字，可以不用编写 10 条 printf()语句，而可以这么写：

```
intcount = 0;
for(count = 1; count <= 10; ++count)
  printf("\n%d", count);
```

for 循环的操作由关键字 for 后面括号中的内容控制。如图 4-1 所示。每次重复执行循环时，都需要执行关键字 for 所在的第一行后面的语句。这里只有一条语句，但这和放在括号中的语句块是一样的。

图 4-2 说明了 3 个由分号分开的控制表达式，它们控制循环的执行。每个控制表达式的作用如图 4-2 所示，下面详细讨论循环执行的过程。

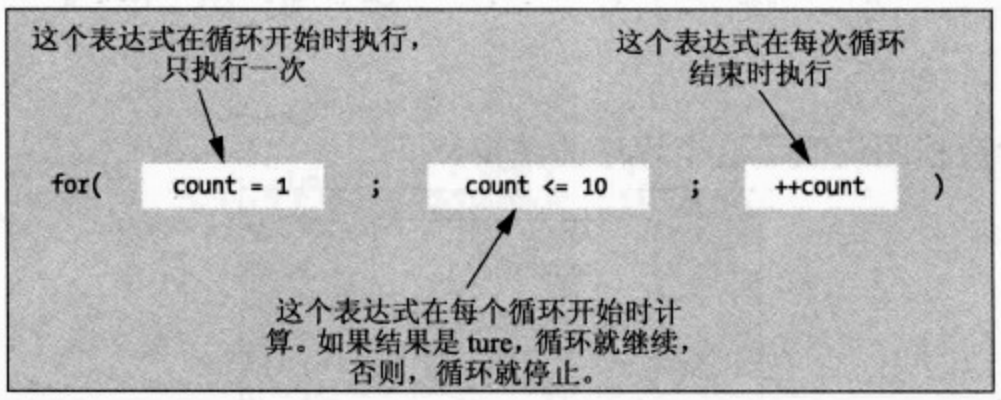


图 4-2　for 循环的控制表达式

- 第一个控制表达式在循环开始时执行，且只执行一次。在这个例子，第一个表达式设定一个变量 count 为 1。这个表达式是 count=1。
- 第二个控制表达式必须是一个逻辑表达式，其结果为 true 或 false；在这个例子，它是 count<=10。第二个表达式在每次循环迭代开始重复前计算。如果结果是 true，循环就继续；否则，循环就结束，程序继续执行循环块或循环语句后面的第一个语句。false 是 0，而非零值是 true，所以只要 count 小于或等于 10，这个循环例子就会执行 printf()语句。当 count 等于 11 时，循环结束。
- 第三个控制表达式++count 在每一次循环迭代结束时执行。这里使用递增运算符给 count 的值加 1：在第一次迭代时，count 是 1，所以 printf()输出 1。第二次迭代时，count 递增为 2，所以 printf()输出数值 2...一直到显示值 10 为止。在开始下一个迭代时，count 递增到 11，而第二个控制表达式的结果是 false，因此循环结束。

注意标点符号。for 循环的控制表达式包含在括号内，每个表达式用分号隔开。这些控制表达式均可以省略，但必须保留分号。例如，在循环外声明变量 count 并初始化为 1：

```
int count = 1;
```

现在不需要指定第一个控制表达式，for 循环如下所示：

www.TopSage.com

```
for( ; count <= 10 ; ++count)
  printf("\n%d", count);
```

在下面的小例子中，添加几行代码，把这个循环添加到一个真实的程序中：

```
/* Program 4.1 List ten integers */
#include <stdio.h>

int main(void)
{
  int count = 1;
  for( ; count <= 10 ; ++count)
    printf("\n%d", count);
  printf("\nWe have finished.\n");
  return 0;
}
```

这个程序将 1~10 的数字显示在不同的行上，接着输出如下信息：

```
We have finished.
```

图 4-3 中的流程图说明了这个程序的逻辑。

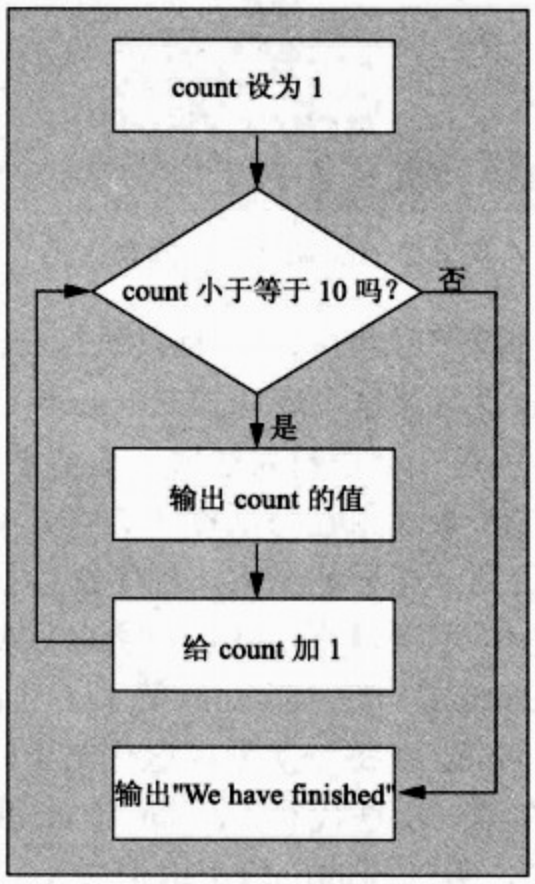


图 4-3　程序 4.1 的逻辑

在这个例子中，很容易看出变量 count 的起始点，所以这个代码没什么问题。但通常情况下，除非控制循环的变量非常靠近循环语句，否则最好在第一个控制表达式中初始化它。这样可以避免潜在的错误。也可以在第一个控制表达式中声明循环变量，此时该变量是循环的本地变量，循环结束后它就不存在了。main()函数可以编写如下：

```
int main(void)
```

www.TopSage.com

```
{
  for(int count = 1 ; count <= 10 ; ++count)
    printf("\n%d", count);
  printf("\nWe have finished.\n");
  return 0;
}
```

count 在第一个 for 循环表达式中声明。这说明，count 在循环结束后就不存在了，所以不能在循环结束后输出它的值。如果需要在循环的外部访问循环控制变量，就应在循环前面的一个语句中声明它，如程序 4-1 所示。

下面是一个略有不同的示例。

试试看：绘制一个盒子

假设要在屏幕上使用字符*绘制一个盒子。可以多次使用 printf()语句，但输入量很大。而使用 for 循环来绘制就容易多了。代码如下：

```
/* Program 4.2 Drawing a box */
#include <stdio.h>
int main(void)
{
  printf("\n***************");    /* Draw the top of the box */

  for(int count = 1 ; count <= 8 ; ++count)
    printf("\n*  *");             /* Draw the sides of the box */

  printf("\n***************\n"); /* Draw the bottom of the box */
  return 0;
}
```

这个程序的输出如下：

```
****************
*               *
*               *
*               *
*               *
*               *
*               *
*               *
*               *
****************
```

代码的说明

这个程序相当简单。第一条 printf()语句在屏幕上输出盒子的第一行：

```
printf("\n***************"); /* Draw the top of the box */
```

下一条语句是 for 循环：

```
for(int count = 1 ; count <= 8 ; ++count)
```

www.TopSage.com

```
  printf("\n* *"); /* Draw the sides of the box */
```

这里 printf()语句重复了 8 次，输出盒子的侧面。下面用专业术语来解释其执行过程。循环控制如下:

```
for(int count = 1 ; count <= 8 ; ++count)
```

循环的执行由关键字 for 后面括号中的 3 个表达式来控制。第一个表达式是:

```
int count = 1
```

这条语句创建并初始化了循环控制变量(或循环计数器)，在这个例子中，循环控制变量是整数变量 count。可以使用其他类型的变量，但是整数类型比较方便。下一个循环控制表达式是:

```
count <= 8
```

这是循环的继续条件。在每次循环迭代之前都要检查这个条件，确定循环是否应继续。如果该表达式是 true，循环就继续。否则，就结束循环，程序将继续执行循环后面的语句。在这个例子中，只要 count 变量小于等于 8，循环就继续。最后一个表达式是:

```
++count
```

这条语句在每一次循环迭代结束时，递增循环计数器的值。因此，输出盒子侧面的循环语句会执行 8 次。迭代 8 次后，变量 count 递增为 9，循环继续的条件是 false，所以循环结束。

程序继续执行循环后面的语句:

```
printf("\n**************\n"); /* Draw the bottom of the box */
```

这条语句在屏幕上输出盒子的底部。

注意:
只要需要多次重复执行某个语句块，就应使用循环，这通常可以节省时间和内存。

4.4 for 循环的一般语法

for 循环的一般形式如下:

```
for(starting_condition; continuation_condition ; action_per_iteration)
  Statement;

Next_statement;
```

重复执行的语句由 Statement 表示。通常这等价于包含在括号中的语句块(一组语句)。

starting_condition 通常(但不总是)设定循环控制变量的初值。循环控制变量一般(但非必要)是一个计数器，用来追踪循环重复的次数。也可以在这里声明并初始化相同类型的

www.TopSage.com

多个变量，各个声明用逗号隔开，此时所有的变量都是循环的本地变量，在循环结束后就不存在了。

continuation_conditon 是一个结果为 true 或 false 的逻辑表达式，用以决定循环是否继续执行。只要这个条件的值是 true，循环就继续。它一般检查循环控制变量的值，但任何逻辑表达式都可以放在这里，只要知道自己在做什么即可。

如前所述，continuation_conditon 在循环开始时测试，而不是在结束时测试。很明显，当 continuation_conditon 一开始就是 false 时，for 循环的语句就根本不执行。

action_per_iteration 在每次循环迭代结束时执行，通常(但不一定)是递增或递减一个或多个循环控制变量。在每次循环迭代时，都会执行 for 语句后面的语句或语句块。一旦 continuation_conditon 是 false，循环就结束，程序继续执行 Next_statement。

下面的循环示例在第一个循环控制条件中声明了两个变量：

```
for(int i = 1, j = 2 ; i<=5 ; i++, j = j+2)
  printf("\n %5d", i*j);
```

其输出是在每一行上输出 2、8、18、32 和 50。

4.5 再谈递增和递减运算符

前面的示例使用了递增运算符，下面深入探讨递增和递减运算符的作用。它们都是一元运算符，只使用一个操作数，用来将存储在整数类型变量中的值加 1 或减 1。

4.5.1 递增运算符

先看看递增运算符。它的形式是++，给所操作的变量加 1。例如，假如变量的类型是 int，下面的 3 条语句有相同的结果：

```
count = count + 1;
count += 1;
++count;
```

这些语句都给变量 count 加 1。最后一种形式最简洁。因此，如果声明了变量 count，将它初始化为 1：

```
int count = 1;
```

在一个循环中重复执行下面的语句 6 次：

```
++count;
```

循环结束时，count 的值就是 7。

也可以在表达式中使用递增运算符。这个运算符在表达式中的动作是递增变量的值，然后，在表达式中使用递增的值。例如，假设 count 的值是 5，执行如下语句：

www.TopSage.com

```
total = ++count + 6;
```

变量 count 会递增到 6，变量 total 的值为 12，这个指令改变两个变量。变量 count 的值是 5，加 1 后变成 6，然后给它加 6，得到 12，赋予赋值运算符右侧的表达式。这个值存储在 total 中。

4.5.2 递增运算符的前置和后置形式

前面将++运算符放在变量前面，这叫做前置形式。这个运算符也可以写在变量的后面，这称为后置形式。在表达式中使用前置和后置形式的效果大不相同。如果在表达式中编写的是 count++，则变量 count 的值在使用之后才递增。这看起来有点复杂。修改前面的例子：

```
total = 6 + count++;
```

count 的初值也是 5，但 total 的值变成了 11。这是因为这个语句使用 count 的初值计算等号右边的表达式(6+5)。变量 count 的值在表达式中使用后才递增。于是前面的语句等价于下面两个语句：

```
total = 6 + count;
++count;
```

然而请注意，在语句中使用递增运算符时(如上面的第二个指令，它递增了 count)，不论是前置还是后置形式，都会得到相同的结果。

若表达式是 a++ +b 或 a+++b，其含义就不是很明显，或编译器会得到什么结果并不容易确定。事实上这两个表达式是相同的，但是第二个表达式的含义可能是 a+ ++b，它是不同的，因为它比其他两个表达式多计算一次。

例如，假定 a=10，b=5，则语句：

```
x = a++ + b;
```

x 的值是 15(10+5)，因为 a 是在计算了表达式后才递增的。但下次使用变量 a 时，它的值是 11。另一方面，如果 a 和 b 的初值相同，执行如下语句：

```
y = a + (++b);
```

y 的值是 16(10+6)，因为 b 是在语句执行前递增的。为了避免混淆，最好使用括号。所以可以将这些语句编写成：

```
x = (a++) + b;
y = a + (++b);
```

4.5.3 递减运算符

递减运算符的操作和递增运算符完全相同。它的形式是—，作用是给它操作的变量减 1。它的使用方式和++完全相同。例如，假设变量 count 是 int 类型，下面 3 条语句会

www.TopSage.com

有相同的结果：

```
count = count - 1;
count -= 1;
--count;
```

这些语句都将变量 count 减 1。例如，如果 count 的值是 10，那么语句：

```
total = --count + 6;
```

其结果是：变量 total 的值为 15(9+6)。初值是 10 的变量 count 被减 1，所以它的值是 9。然后给它加 6，使赋值运算符右边表达式的值为 15。

递增运算符的前置和后置形式的使用规则也适用于递减运算符。例如，如果 count 的初值是 5，那么语句：

```
total = --count + 6;
```

total 的值是 10(4+6)，然而

```
total = 6 + count-- ;
```

total 的值是 11(6+5)。这两个运算符通常应用于整数，后面的章节也会介绍如何将它们应用于 C 语言的其他数据类型。

4.6 再论 for 循环

有了++和--的更多了解，下面看看另一个使用循环的例子。

试试看：数字汇总

这个程序比用*号画盒子要有用、有趣得多。假定想知道某条街上所有门牌号的总和是多少，这需要读入一个整数值，再使用 for 循环汇总所有的整数，从 1 加到输入的那个数值为止。

```
/* Program 4.3 Sum the integers from 1 to a user-specified number */
#include <stdio.h>

int main(void)
{
  long sum = 0L; /* Stores the sum of the integers */
  int count = 0; /* The number of integers to be summed */

  /* Read the number of integers to be summed */
  printf("\nEnter the number of integers you want to sum: ");
  scanf(" %d", &count);

  /* Sum integers from 1 to count */
  for(int i = 1 ; i <= count ; i++)
```

www.TopSage.com

```
    sum += i;

  printf("\nTotal of the first %d numbers is %ld\n", count, sum);
  return 0;
}
```

这个程序的输出如下:

```
Enter the number of integers you want to sum: 10

Total of the first 10 integers is 55
```

代码的说明

首先，声明和初始化两个计算过程中需要的变量:

```
long sum = 0L; /* Stores the sum of the integers */
int count = 0; /* The number of integers to be summed */
```

这里使用 sum 保存计算的最后结果。它的类型声明为 long，所以计算出来的总和最大可达整数的最大值。变量 count 存储输入的整数值，它也是要汇总的整数个数，在 for 循环中要使用这个值控制迭代的次数。

用以下语句处理输入:

```
printf("\nEnter the number of integers you want to sum: ");
scanf(" %d", &count);
```

在提示信息后，读取整数，它定义了需要的总和。例如，如果用户输入 4，程序将会计算 1、2、3 和 4 的总和。这个总和用下面的循环计算:

```
for(int i = 1 ; i <= count ; i++)
  sum += i;
```

循环变量 i 在 for 循环的开始条件中声明并初始化为 1。在每次迭代中，i 的值都会加到 sum 中，然后递增 i，所以值 1、2、3...一直到 count 的值，都会加到 sum 中。当 i 的值超过 count 的值时，循环就结束。

在叙述如何控制 for 循环时，本书使用了“非必要”这个词，暗示控制表达式的灵活性非常大。下一个程序将稍微缩短前面的例子，说明这个灵活性。

试试看：灵活的 for 循环

这个例子说明了如何在 for 循环的第 3 个控制表达式中完成一个计算:

```
/* Program 4.4 Summing integers - compact version */
#include <stdio.h>

int main(void)
{
  long sum = 0L; /* Stores the sum of the integers */
  int count = 0; /* The number of integers to be summed */
```

www.TopSage.com

```
  /* Read the number of integers to be summed */
  printf("\nEnter the number of integers you want to sum: ");
  scanf(" %d", &count);

  /* Sum integers from 1 to count */
  for(int i = 1 ; i<= count ; sum += i++ );

  printf("\nTotal of the first %d numbers is %ld\n", count, sum);
  return 0;
}
```

输出如下:

```
Enter the number of integers you want to sum: 6

Total of the first 6 numbers is 21
```

代码的说明

这个程序的执行和前一个例子完全相同。唯一的区别是，在循环的第 3 个控制表达式中汇总:

```
for(int i = 1 ; i<= count ; sum += i++ );
```

循环语句是空的: 在闭括号的后面只是一个分号。循环的第 3 个控制表达式将 i 的值加到 sum 中，再递增 i，准备下一次迭代。它是以这个方式运作的，因为这里使用了递增运算符的后置形式。如果在这里使用前置形式，将会得到错误的答案，因为 sum 的值包含了循环第一次迭代时的值 count+1，而不是 count。

4.6.1 修改 for 循环变量

当然，递增循环控制变量不是只能加 1。可以用任意值改变循环控制变量，正数或负数均可。例如，可以逆向计算前 *n* 个整数的总和，如下面的例子所示:

```
/* Program 4.5 Summing integers backward */
#include <stdio.h>

int main(void)
{
  long sum = 0L; /* Stores the sum of the integers */
  int count = 0; /* The number of integers to be summed */

  /* Read the number of integers to be summed */
  printf("\nEnter the number of integers you want to sum: ");
  scanf(" %d", &count);

  /* Sum integers from count to 1 */
  for(int i = count ; i >= 1 ; sum += i--);
```

www.TopSage.com

```
  printf("\nTotal of the first %d numbers is %ld\n", count, sum);
  return 0;
}
```

这个程序会产生和前一个例子相同的输出。唯一改变的是循环控制表达式。循环计数器初始化为 count，而不是 1，每次循环迭代时要递增它。因此是将 count、count - 1、count - 2...1 加在一起。同样，如果使用前置形式，答案会是错误的，因为这会从 count - 1 开始执行相加操作，而不是从 count 开始。

其实，使用循环汇总前 *n* 个整数不是必要的。下面这个简洁的公式可以更高效地计算出 1~n 的整数和，它与循环一点关系都没有：

```
n*(n+1)/2
```

4.6.2 没有参数的 for 循环

如前所述，不必在 for 循环语句内放置任何参数。for 循环的最简洁形式如下：

```
for(;;)
  statement;
```

和前面一样，这里的 statement 也可以是放在括号中的语句块，且通常是语句块。因为没有循环继续条件、初始条件以及循环计数器，因此循环将永不停止。除非希望计算机总是什么都不做，否则 statement 必须包含退出循环的方式。要停止循环，循环体必须包含两条语句：判断结束循环的条件是否已满足的语句，以及终止当前循环迭代并继续执行循环后面语句的语句。

4.6.3 循环内的 break 语句

第 3 章在 switch 语句里使用过 break 语句。它的作用是终止 switch 块中代码的执行，并继续执行跟在 switch 后的第一行语句。break 语句在循环体内的作用和 switch 基本相同。例如：

```
char answer = 0;
for( ;; )
{
  /* Code to read and process some data */
  printf("Do you want to enter some more(y/n): ");
  scanf("%c", &answer);
  if(tolower(answer) == 'n')
    break;                    /* Go to statement after the loop */
}
/* Statement after the loop */
```

这里有一个无限循环。scanf()语句将一个字符读入 answer，如果输入的字符是 *n* 或

www.TopSage.com

N，就执行 break 语句。结果是结束循环，继续执行循环后面的语句。下面用另一个例子来说明。

试试看：最小的 for 循环

这个例子计算了任意个数字的平均值：

```
/* Program 4.6 The almost indefinite loop - computing an average */
#include <stdio.h>
#include <ctype.h> /* For tolower() function */

int main(void)
{
  char answer = 'N';   /* Records yes or no to continue the loop */
  double total = 0.0;  /* Total of values entered */
  double value = 0.0;  /* Value entered */
  int count = 0;       /* Number of values entered */

  printf("\nThis program calculates the average of"
              " any number of values.");

  for( ;; )                /* Indefinite loop */
  {
    printf("\nEnter a value: "); /* Prompt for the next value */
    scanf(" %lf", &value);       /* Read the next value */
    total += value;              /* Add value to total */
    ++count;                     /* Increment count of values */

    /* check for more input */
    printf("Do you want to enter another value? (Y or N): ");
    scanf(" %c", &answer);       /* Read response Y or N */

    if(tolower(answer) == 'n')   /* look for any sign of no */
      break;                     /* Exit from the loop */
  }
  /* output the average to 2 decimal places */
  printf("\nThe average is %.2lf\n", total/count );
  return 0;
}
```

这个程序的输出如下：

```
This program calculates the average of any number of values.
Enter a value: 2.5
Do you want to enter another value? (Y or N): y

Enter a value: 3.5
Do you want to enter another value? (Y or N): y

Enter a value: 6
```

www.TopSage.com

```
Do you want to enter another value? (Y or N): n

The average is 4.00
```

代码的说明

图 4-4 显示了这个程序的一般逻辑。

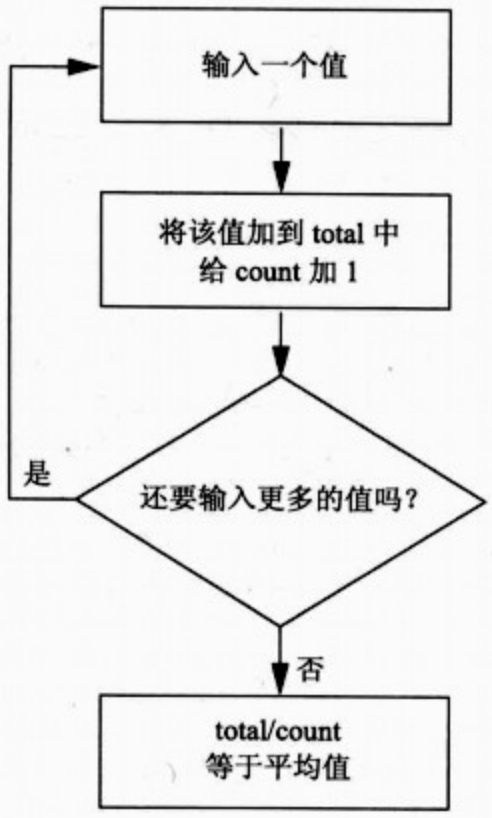


图 4-4 程序的一般逻辑

这个程序建立了一个无限循环，这个 for 循环没有指定结束条件——没有循环控制表达式:

```
for( ;; )                          /* Indefinite loop */
```

因此，括号内的语句块会无限重复下去。下面的语句在循环内显示一条提示信息，读入一个输入值:

```
printf("\nEnter a value: "); /* Prompt for the next value */
scanf(" %lf", &value);       /* Read the next value */
```

接下来将输入的值加到变量 total 中:

```
total += value;                /* Add value to total */
```

然后递增输入值的个数:

```
++count;                           /* Increment count of values */
```

读入一个值，加到 total 中之后，询问用户是否还要输入:

```
/* check for more input */
printf("Do you want to enter another value? (Y or N): ");
scanf(" %c", &answer); /* Read response Y or N */
```

www.TopSage.com

这个语句提示用户输入 Y 或 N。用户输入的字符用 if 语句检查:

```
if(tolower(answer) == 'n')       /* look for any sign of no */
  break; /* Exit from the loop */
```

保存在 answer 中的字符用<ctype.h>头文件中的 tolower()函数转换成小写，所以只需测试 n。如果用户输入了 N 或 n，表示已输入完数据，就执行 break 语句。在循环内执行 break 会立刻结束循环，继续执行跟在循环结束括号后的语句，如下所示:

```
printf("\nThe average is %.2lf\n", total/count);
```

这行语句用 total 除以输入值的个数来计算输入值的平均值，然后显示结果。

4.6.4 使用 for 循环限制输入

可以使用 for 循环限制用户输入的次数。循环的每次迭代都允许输入一个值。当循环完成指定的重复次数后，便结束循环，不允许再输入数据。下面编写一个简单的程序来演示，这个程序实现了一个数字猜谜游戏。

试试看：数字猜谜游戏

这个程序要求用户猜测该程序挑选出的幸运数字。它使用了一个 for 循环和许多 if 语句。还加进了条件运算符，提醒读者不要忘记如何使用它。

```
/* Program 4.7 A Guessing Game */
#include <stdio.h>

int main(void)
{
  int chosen = 15;                /* The lucky number */
  int guess = 0;                  /* Stores a guess */
  int count = 3;                  /* The maximum number of tries */

  printf("\nThis is a guessing game.");
  printf("\nI have chosen a number between 1 and 20"
        " which you must guess.\n");

  for( ; count>0 ; --count)
  {
    printf("\nYou have %d tr%s left.", count, count == 1 ? "y" : "ies");
    printf("\nEnter a guess: "); /* Prompt for a guess */
    scanf("%d", &guess);          /* Read in a guess */

    /* Check for a correct guess */
    if(guess == chosen)
    {
      printf("\nYou guessed it!\n");
      return 0;                   /* End the program */
```

www.TopSage.com

```
    }
    /* Check for an invalid guess */
    if(guess<1 || guess > 20)
      printf("I said between 1 and 20.\n ");
    else
      printf("Sorry. %d is wrong.\n", guess);
  }
  printf("\nYou have had three tries and failed. The number was %d\n",
            chosen);
  return 0;
}
```

输出如下:

```
This is a guessing game.
I have chosen a number between 1 and 20 which you must guess.

You have 3 tries left.
Enter a guess: 5
Sorry. 5 is wrong.

You have 2 tries left.
Enter a guess: 18
Sorry. 18 is wrong.

You have 1 try left.
Enter a guess: 7
Sorry. 7 is wrong.

You have had three tries and failed. The number was 15
```

代码的说明

首先，声明并初始化 3 个类型为 int 的变量: chosen、guess 和 count:

```
int chosen = 15;  /* The lucky number */
int guess = 0;    /* Stores a guess */
int count = 3;    /* The maximum number of tries */
```

这些变量分别存储幸运数字、用户猜测的数字和允许用户猜测的次数。注意，这里创建了一个变量来存储幸运数字。这个程序可以只使用 15 这个数字，而不使用变量，但使用变量更便于修改用户要猜的数字。使用变量 chosen 也可使代码的执行更明显。

给用户提供程序开始的说明:

```
printf("\nThis is a guessing game.");
printf("\nI have chosen a number between 1 and 20"
      " which you must guess.\n");
```

猜测的次数由这个循环控制:

```
for( ; count>0 ; --count)
```

www.TopSage.com

```
{
  ...
}
```

游戏所有的操作细节都在这个循环里实现，只要 count 是正数，循环就会继续下去，所以循环会重复 count 次。

下面的语句提示用户输入猜测的数字，然后读入该数字：

```
printf("\nYou have %d tr%s left.", count, count == 1 ? "y" : "ies");

printf("\nEnter a guess: "); /* Prompt for a guess */
scanf("%d", &guess);          /* Read in a guess */
```

第一个 printf()有点复杂，实际上，当 count 是 1 时，这个 printf()在输出的 tr 后面加上 y，而当 count 大于 1 时，在输出的 tr 后面加上 ies。我必须显示正确的单复数形式。

使用 scanf()读入 guess 值后，用下面的语句检查它是否正确：

```
/* Check for a correct guess */
if(guess == chosen)
{
  printf("\nYou guessed it!");
  return 0;                    /* End the program */
}
```

如果猜测的数值是正确的，就显示一条信息，执行 return 语句。return 语句会结束函数 main()，程序就结束了。第 8 章详细讨论函数时，会介绍 return 语句。

如果猜测的数值是错误的，程序就执行循环内的最后一条检查语句：

```
/* Check for an invalid guess */
if(guess<1 || guess > 20)
  printf("I said between 1 and 20.\n ");
else
  printf("Sorry. %d is wrong.\n", guess);
```

这组语句测试输入值是否在指定的范围内。如果不是，就显示一条信息，重申该范围。如果输入值是有效的，就显示一条猜错的信息。

循环在重复了 3 次，也就是猜了 3 次后结束。循环后的语句如下：

```
printf("\nYou have had three tries and failed. The number was %d\n",
            chosen);
```

这条语句只有在猜错 3 次后才执行。它显示一条信息，并透露正确的数值，然后程序结束。这个程序的设计更便于改变 chosen 变量的值。

4.6.5 生成伪随机整数

在前一个例子中，如果程序在每次执行时，可以生成要猜测的数字，该数字每次都

www.TopSage.com

不同。为此，可以使用在头文件<stdlib.h>中声明的函数 rand()：

```
int chosen = 0;
chosen = rand(); /* Set to a random integer */
```

每次调用 rand()函数，它都会返回一个随机整数，这个值在 0 到<stdlib.h>定义的 RAND_MAX 之间。由 rand()函数产生的整数称为伪随机数(pseudo-random)，因为真正的随机数只能在自然的过程中产生，而不能通过运算法则产生。

rand()函数使用一个起始的种子值生成一系列数字，对于一个特定的种子，所产生的序列数永远是相同的。如果使用这个函数和默认的种子值，如上面的代码所示，就总是得到相同的序列数，这会使这个游戏没什么意思，只是在测试程序时比较有用。C 语言提供了另一个标准函数 srand()，在调用这个函数时，可以用作为参数传递给函数的特定种子值来初始化序列数。这个函数也在<stdlib.h>头文件中声明。

乍看之下，这似乎并没有让猜数游戏改变多少，因为每次执行程序时，必须产生一个不同的种子值。此时可以使用另一个库函数：在<time.h>头文件中声明的函数 time()。time()函数会把自 1970 年 1 月 1 日起至今的总秒数返回为一个整数，因为时间永不停歇，所以每次执行程序时，都会得到不同的值。time()函数需要一个参数 NULL，NULL 是在<stdlib.h>中定义的符号，详见第 7 章。

因此，要在每次执行程序时得到不同的伪随机序列数，可以使用以下的语句：

```
srand(time(NULL)); /* Use clock value as starting seed */
int chosen = 0;
chosen = rand();   /* Set to a random integer 0 to RAND_MAX */
```

上限值 RAND_MAX 相当大，通常是类型 int 可以存储的最大值。如果需要更小范围的数值，可以按比例缩小 rand()的返回值，提供所需范围的值。假设要得到的数值在 0 到 limit(不包含 limit)的范围内，最简单的方法如下：

```
srand(time(NULL));      /* Use clock value as starting seed */
int limit = 20.0;       /* Upper limit for pseudo-random values */
int chosen = 0;
chosen = rand()%limit; /* 0 to limit-1 inclusive */
```

当然，如果数值需要在 1 到 limit 之间，可以编写如下语句：

```
chosen = 1+rand()%limit; /* 1 to limit inclusive */
```

这条语句在编译器和库中使用 rand()函数，运作得相当好。然而一般来说，最好不要限制伪随机数产生器产生的数值的范围。这是因为实质上，我们是将返回值的高位字节去掉，并假设剩余的字节也代表随机数。这不是必要的，试着在前面例子的一个变体中使用 rand()函数：

```
/* Program 4.7A A More Interesting Guessing Game */
#include <stdio.h>
#include <stdlib.h>       /* For rand() and srand() */
```

www.TopSage.com

```
#include <time.h>                /* For time() function */

int main(void)
{
  int chosen = 0;                /* The lucky number */
  int guess = 0;                 /* Stores a guess */
  int count = 3;                 /* The maximum number of tries */
  int limit = 20;                /* Upper limit for pseudo-random values */

  srand(time(NULL));             /* Use clock value as starting seed */
  chosen = 1 + rand()%limit;     /* Random int 1 to limit */

  printf("\nThis is a guessing game.");
  printf("\nI have chosen a number between 1 and 20"
        " which you must guess.\n");

  for( ; count>0 ; --count)
  {
    printf("\nYou have %d tr%s left.", count, count == 1 ? "y" : "ies");
    printf("\nEnter a guess: ");      /* Prompt for a guess */
    scanf("%d", &guess);              /* Read in a guess */

    /* Check for a correct guess */
    if(guess == chosen)
    {
      printf("\nYou guessed it!\n");
      return 0;                       /* End the program */
    }

    /* Check for an invalid guess */
    if(guess<1 || guess > 20)
      printf("I said between 1 and 20.\n ");
    else
      printf("Sorry. %d is wrong.\n", guess);
  }
  printf("\nYou have had three tries and failed. The number was %ld\n",
              chosen);
  return 0;
}
```

这个程序在大多数情况下会给出不同的猜测数字。

4.6.6 再谈循环控制选项

前面介绍了如何用++和--运算符递增或递减循环计数器。可以对循环计数器递增或递减任意数值。下面看一个例子：

```
long sum = 0L;
for(int n = 1 ; n<20 ; n += 2)
```

www.TopSage.com

```
  sum += n;
printf("Sum is %ld", sum);
```

前面代码段中的循环汇总 1~20 之间的所有奇数。第 3 个控制表达式在每次迭代时，将循环变量 n 递增 2。这个表达式可以是任意表达式，包含赋值语句。例如，要汇总 1~1000 之间彼此相隔 7 的所有整数，可以编写如下循环：

```
for(int n = 1 ; n<1000 ; n = n+7)
  sum += n;
```

现在循环控制表达式在每次迭代的最后，将 n 增加 7，所以得到的总和是 1+8+15+22+...1000。

循环控制表达式可能不只一个。可以重写第一个代码段中的循环，汇总 1~20 之间的奇数，如下：

```
for(int n = 1 ; n<20 ; sum += n, n += 2)
  ;
```

第 3 个控制表达式由逗号分开的两个表达式组成。它们会在每次循环迭代结束时依序执行，所以第一个表达式：

```
sum += n
```

会将 n 的当前值加到 sum 中。接着第二个表达式：

```
n += 2
```

给 n 增加 2。这些表达式的执行顺序是从左到右，所以必须依照这个顺序编写。如果颠倒了该顺序，结果就是错的。

表达式也并不是只能有两个，其个数是任意的，只要它们都有用逗号分开即可。当然，只应在有明显的优势时才这么做，不然程序会很难理解。

第一个和第二个控制表达式也可以由许多用逗号分开的表达式组成，但一般不需要这么做。

4.6.7 浮点类型的循环控制变量

循环控制变量也可以是一个浮点类型的变量。下面的循环汇总从 1/1~1/10 的分数：

```
double sum = 0.0;
for(double x = 1.0 ; x<11 ; x += 1.0)
  sum += 1.0/x;
```

这种情形并不常见。注意，分数值通常没有浮点数形式的精确表示，所以不应把相等判断作为结束循环的条件，例如：

```
for(double x = 0.0 ; x != 2.0 ; x+= 0.2) /* Indefinite loop!!! */
  printf("\nx = %.2lf",x);
```

www.TopSage.com

这个循环应输出 0.0~2.0 之间的 x 值，其递增量为 0.2，所以应该有 11 行输出。但 0.2 没有浮点数形式的二进制精确表示，所以这个循环会使计算机一直运行下去(除非在 Microsoft Windows 下使用 Ctrl+C 令它停止)。

4.7 while 循环

for 循环就介绍到这里。前面举了许多 for 循环的例子，现在探讨另一类循环：while 循环。在 while 循环中，只要某个逻辑表达式等于 true，就重复执行一组语句。这可以表示为：

```
While this condition is true
  Keep on doing this
```

下面是一个例子：

```
While you are hungry
  Eat sandwiches
```

其含义是，在吃下一个三明治之前，问自己“饿吗?”。如果答案为“是”，就吃一片三明治，然后问“还饿吗?”，就这样不断地吃，不断地问，直到答案为“否”为止，此时就去做其他事，例如喝咖啡。

while 循环的一般语法如下：

```
while( expression )
  Statement1;

Statement2;
```

和往常一样，Statementl 可以是语句块。while 循环的逻辑如图 4-5 所示。

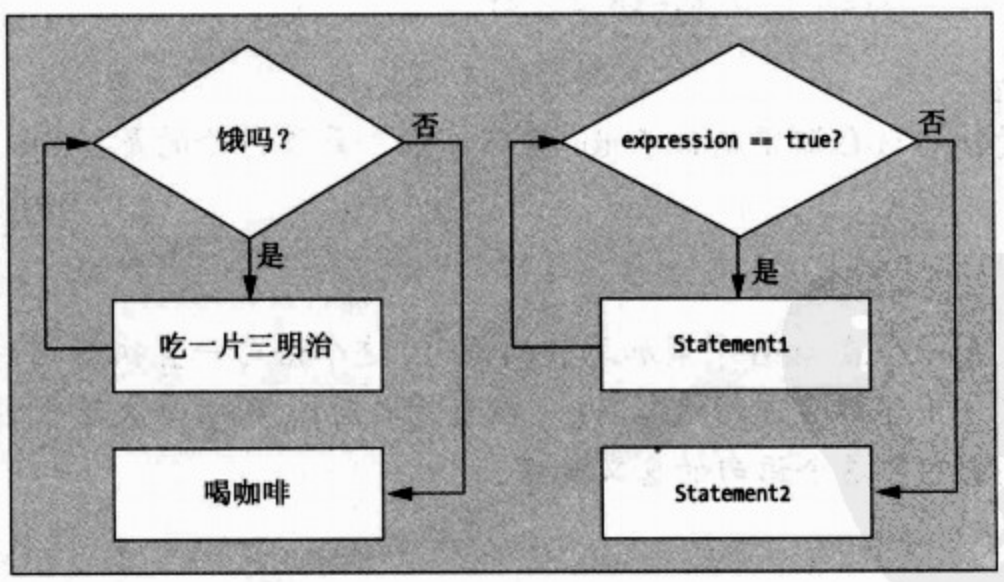


图 4-5 while 循环的逻辑

www.TopSage.com

与 for 循环一样，while 循环的继续条件也是在开始时测试，所以如果 expression 一开始就是 false，就不执行循环语句。如果第一个问题的回答是“不，不饿”，就不吃三明治，而是直接喝咖啡。

试试看：使用 while 循环

while 循环看起来相当简单，下面将它应用于前面编写的整数汇总程序：

```
ram 4.8 While programming and summing integers */
#include <stdio.h>

int main(void)
{
  long sum = 0L;  /* The sum of the integers */
  int i = 1;      /* Indexes through the integers */
  int count = 0;  /* The count of integers to be summed */

  /* Get the count of the number of integers to sum */
  printf("\nEnter the number of integers you want to sum: ");
  scanf(" %d", &count);

  /* Sum the integers from 1 to count */
  while(i <= count)
    sum += i++;

  printf("Total of the first %d numbers is %ld\n", count, sum);
  return 0;
}
```

这个程序序的输出如下：

```
Enter the number of integers you want to sum: 7
Total of the first 7 numbers is 28
```

代码的说明

这个例子的执行过程非常类似于 for 循环。唯一需要讨论的是 while 循环：

```
while(i <= count)
  sum += i++;
```

这个循环包含一个在 sum 中累加总数的语句。这个语句一直执行到 i 的值等于 count 为止。因为这里使用了后置递增运算符(++在变量之后)，所以每次迭代时，先用 i 的值计算 sum，再递增它。这个语句的含义如下：

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

所以在下一个迭代之前，sum 的值不受 i 的递增操作影响。下面试着用较简单的说法来解释：

www.TopSage.com

- 进入 while 循环：此时，i 是 1，count 的值就是用户输入的值(假定为 3)。当循环开始时，首先检查 i<=count 是否为 true。现在 i 是 1，而 1<=3，所以 i<=count 是 true，执行循环语句：

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

- 第一次执行 while 循环：首先，i 的值(为 1)加到变量 sum 中。原本变量 sum 是 0，现在等于 1。这里使用的是后置递增运算符，所以在 sum 中执行完计算后才递增变量 i。所以 i 现在是 2，然后回到循环的开始处。检查 while 表达式，确定 i 的值是否仍然小于等于 count，现在 i 是 2，而 2<3，所以再次执行循环语句。
- 第二次执行 while 循环：在第二次循环迭代中，将 i 的新值(现在是 2)加到 sum 的旧值(是 1)中，并将结果保存到 sum 中。变量 sum 现在等于 3。给 i 加 1，所以 i 现在是 3，然后回到循环的开始处，检查控制表达式是否仍然是 true。
- 第三次执行 while 循环：此时 i 等于 count，所以还可以继续循环。将 i 的新值(现在是 3)加到 sum 的旧值(也是 3)上，并将结果保存到 sum 中，现在它的值是 6。给 i 加 1，所以 i 现在是 4，然后返回，再次检查循环表达式。
- 最后一次执行 while 循环：现在 i 的值是 4，大于 count 的值 3，所以表达式 i<=count 是 false，因此退出循环。

这个例子使用了后置递增运算符。如何将前面的例子改为使用前置++运算符？其答案见下一节。

使用前置++运算符

要修改的代码是 while 循环：

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

在程序 4.8 中只修改这条语句。如果执行这个程序，会得到错误的答案：

```
Enter the number of integers you want to sum: 3
Total of the first 3 numbers is 9
```

这是因为++运算符在把值存储到 sum 之前，先给 i 加 1。变量 i 开始时是 1，在第一次迭代时递增为 2，并把这个 2 加到 sum 中。

要让第一个循环迭代正常运作，必须让 i 的初始值为 0。这样，第一次递增操作会将 i 设定为 1。所以必须修改 i 的声明：

```
int i = 0;
```

然而，程序还是不能正常运作，因为它一直计算到 i 的值大于 count 为止，多了一次迭代。所以必须修正控制表达式，使循环继续的条件变成 i 小于但不等于 count：

```
while(i < count)
```

现在程序可以生成正确的答案了。这个程序有助于理解这些运算符的后置和前置形式。

www.TopSage.com

4.8 嵌套循环

有时需要将一个循环放在另一个循环里面。例如计算某条街上每间房子的居住人数。这需要进入每间房子，计算每间房子的居住人数。统计所有的房子是一个外部循环，在外部循环的每次迭代中，都要使用一个内部循环来计算居住人数。了解嵌套循环的最好方法是举一个简单的例子。

试试看：使用嵌套循环

为了演示嵌套循环，下面的简单例子以汇总整数的程序为基础。原来的程序是计算从 1 到输入值之间的所有整数的和。现在要从第一间房子开始，一直到当前的房子为止，计算每间房子的居住人数。看看这个程序的输出，就会比较清楚。

```
/* Program 4.9 Sums of integers step-by-step */
#include <stdio.h>

int main(void)
{
  long sum = 0L; /* Stores the sum of integers */
  int count = 0; /* Number of sums to be calculated */

  /* Prompt for, and read the input count */
  printf("\nEnter the number of integers you want to sum: ");
  scanf(" %d", &count);

  for(int i = 1 ; i <= count ; i++)
  {
    sum = 0L;   /* Initialize sum for the inner loop */

    /* Calculate sum of integers from 1 to i */
    for(int j = 1 ; j <= i ; j++)
      sum += j;

    printf("\n%d\t%ld", i, sum); /* Output sum of 1 to i */
  }
  return 0;
}
```

输出如下：

```
Enter the number of integers you want to sum: 5

1       1
2       3
3       6
4       10
5       15
```

www.TopSage.com

可以看出，如果输入5，程序会分别计算1~1、1~2、1~3、1~4、1~5的整数和。

代码的说明

这个程序计算了1~1、1~2、1~3……1~count的整数和。这个嵌套循环的重点是，在外部循环的每次迭代中，内部循环要完成所有的迭代。因此外部循环设定的i值决定了内部循环的重复次数:

```
for(int i = 1 ; i <= count ; i++)
{
  sum = 0L;                          /* Initialize sum for the inner loop */

  /* Calculate sum of integers from 1 to i */
  for(int j = 1 ; j <= i ; j++)
    sum += j;

  printf("\n%d\t%ld", i, sum); /* Output sum of 1 to i */
}
```

外循环开始时将i初始化为1，之后递增i，执行该循环，直到i的值递增到count为止。对于外部循环的每次迭代，即对于i的每个值，sum都初始化为0，并执行内部循环，最后用printf()语句显示结果。内部循环会累加从1到i当前值之间的所有整数:

```
/* Calculate sum of integers from 1 to i */
for(int j = 1 ; j <= i ; j++)
  sum += j;
```

每次内部循环结束时，都会执行printf()，输出sum的值。然后回到外部循环的开始处，执行下一次迭代。

由输出结果可以看出嵌套循环的执行过程。第一个循环在每次返回开始处时，只是将变量sum设定为0，然后内部循环累加从1到i当前值之间的所有整数。可以修改这个嵌套循环，将while循环用作内部循环，并产生输出，使程序的执行过程更清楚。

试试看：在for循环内嵌套while循环

在前面的例子，在一个for循环内嵌套了for循环。在这个例子里，要在一个for循环内嵌套while循环。

```
/* Program 4.10 Sums of integers with a while loop nested in a for loop */
#include <stdio.h>

int main(void)
{
  long sum = 1L;  /* Stores the sum of integers */
  int j = 1;      /* Inner loop control variable */
  int count = 0;  /* Number of sums to be calculated */

  /* Prompt for, and read the input count */
  printf("\nEnter the number of integers you want to sum: ");
```

www.TopSage.com

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

  for(int i = 1 ; i <= count ; i++)
  {
    sum = 1L;                  /* Initialize sum for the inner loop */
    j=1;                       /* Initialize integer to be added */
    printf("\n1");

    /* Calculate sum of integers from 1 to i */
    while(j < i)
    {
      sum += ++j;
      printf("+%d", j);        /* Output +j - on the same line */
    }
    printf(" = %ld\n", sum); /* Output = sum */
  }
  return 0;
}
```

这个程序的输出如下:

```
Enter the number of integers you want to sum: 5

1 = 1

1+2 = 3

1+2+3 = 6

1+2+3+4 = 10

1+2+3+4+5 = 15
```

代码的说明

外部循环的里面有区别:

```
for(int i = 1 ; i <= count ; i++)
{
  sum = 1L;              /* Initialize sum for the inner loop */
  j=1;                   /* Initialize integer to be added */
  printf("\n1");

  /* Calculate sum of integers from 1 to i */
  while(j < i)
  {
    sum += ++j;
    printf("+%d", j);   /* Output +j    on the same line */
  }
  printf(" = %ld\n", sum); /* Output = sum */
}
```

www.TopSage.com

外部循环的控制和以前相同。不同之处是每次迭代过程。变量 sum 在外部循环中初始化为 1，因为 while 循环从 2 开始将值加到 sum 中。要相加的值存储在 j 中，它也初始化为 1。外部循环的第一个 printf()只是输出一个换行符，再输出 1，这是要累计的第一个整数。内部循环汇总从 2 到 i 的整数。对于 j 中每个要加到 sum 上的整数值，内部循环的 printf()都会输出+j，它与前面输出的 1 在同一行上。因此只要 j 小于 i，内部循环就会输出+2、+3 等。当然外部循环第一次迭代时，i 是 1，所以不执行内部循环，因为 j<i(1<1)是 false。

内部循环结束时，执行最后一个 printf()语句，它输出一个等号和 sum 的值。之后返回外部循环的开始处，执行下一次迭代。

4.9 嵌套循环和 goto 语句

前面学习了如何在一个循环内嵌套另一个循环，其实循环还可以嵌套任意多层。例如：

```
for(int i = 0 ; i<10 ; ++i)
  for(int j = 0 ; j<20 ; ++k)    /* Loop executed 10 times */
    for(int k = 0 ; k<30 ; ++k) /* Loop executed 10x20 times */
    {                            /* Loop body executed 10x20x30 times */
      /* Do something useful */
    }
```

由 i 控制的外部循环每次迭代时，都会执行一次由 j 控制的内部循环。由 j 控制的循环每次迭代时，都会执行一次由 k 控制的最内层循环。因此最内层的循环体会执行 6 000 次。

有时在这样的深层嵌套循环中，希望从最内层的循环跳到最外层循环的外面，执行最外层循环后面的语句。最内层循环中的 break 语句只能跳出这个最内层的循环，执行由 j 控制的循环。要使用 break 语句完全跳出嵌套循环，需要相当复杂的逻辑才能中断每一层循环，最后跳出最外层的循环。此时可以使用 goto 语句，因为它提供了一种避免复杂逻辑的方法。例如：

```
for(int i = 0 ; i<10 ; ++i)
  for(int j = 0 ; j<20 ; ++k)    /* Loop executed 10 times */
    for(int k = 0 ; k<30 ; ++k) /* Loop executed 10x20 times */
    {                            /* Loop body executed 10x20x30 times */
      /* Do something useful */
      if(must_escape)
      goto out;
    }
out: /*Statement following the nested loops */
```

www.TopSage.com

这段代码假定，可以在最内层的循环中修改 must_escape，发出应结束整个嵌套循环的信号。如果变量 must_escape 是 true，就执行 goto 语句，直接跳到有 out 标志的语句。这样就可以直接退出整个嵌套循环，不需要在外部循环中进行复杂的判断。

4.10 do-while 循环

第 3 种循环类型是 do-while。既然已经有 for 循环和 while 循环了，为什么还需要这个循环？do-while 和这两个循环有非常微妙的区别。它是在循环结束时测试循环是否继续，所以这个循环的语句或语句块至少会执行一次。

while 循环是在循环开始处进行测试。所以在任何动作发生之前，先检查表达式。下面的代码：

```
int number = 4;

while(number < 4)
{
  printf("\nNumber = %d", number);
  number++;
}
```

这段代码不会有任何输出。一开始控制表达式 number<4 就是 false，所以永远不会执行循环语句块。

然而，do-while 循环就不同了。可以看出，如果将前面的 while 循环用 do-while 循环取代，其他语句不变：

```
int number = 4;

do
{
  printf("\nNumber = %d", number);
  number++;
}
while(number < 4);
```

现在执行这个循环，会显示 number=4。这是因为表达式 number<4 在循环第一次迭代结束时检查。

do-while 循环的一般表示方式如下：

```
do
  Statement;
while(expression);
```

注意，分号在 do-while 循环中的 while 语句后面。在 while 循环没有这个符号。与往常一样，Statement 可以用括号内的语句块取代。在 do-while 循环中，如果 expression 的值是 true(非零)，循环就继续。当 expression 变成 false(零)时，这个循环就结束，如图 4-6

www.TopSage.com

所示。

在 do-while 循环中，是先吃一片三明治，再检查是否饿了，所以至少会吃一片三明治。这个循环不能用作卡路里控制食谱的一部分。

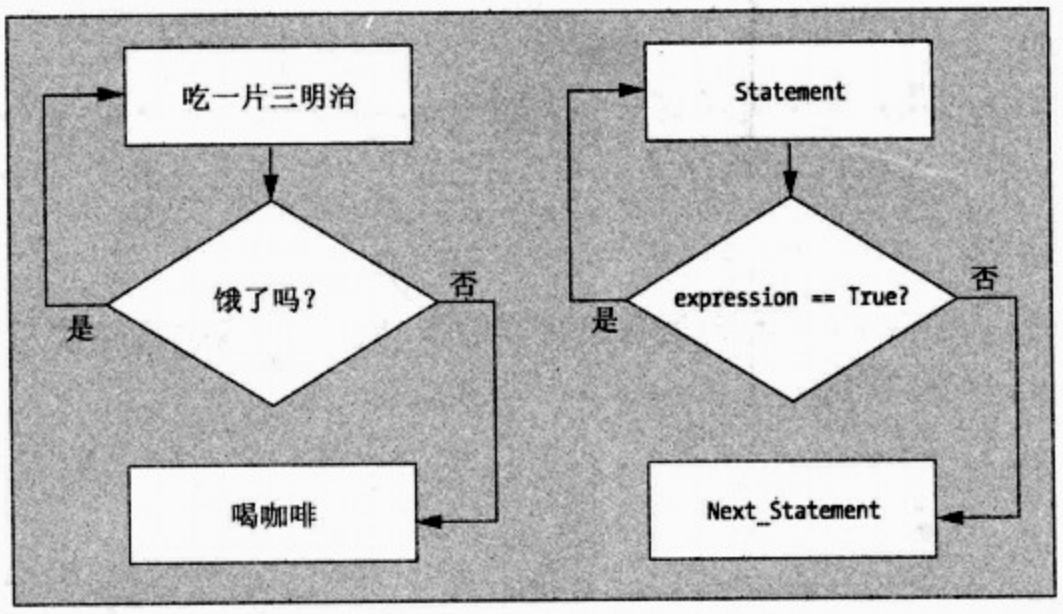


图 4-6 do-while 循环的操作

试试看：使用 do-while 循环

在一个小程序里使用 do-while 循环，将一个正整数中数字的顺序翻转过来：

```
/* Program 4.11 Reversing the digits */
#include <stdio.h>

int main(void)
{
  int number = 0;   /* The number to be reversed */
  int rebmun = 0;   /* The reversed number */
  int temp = 0;     /* Working storage */

  /* Get the value to be reversed */
  printf("\nEnter a positive integer: ");
  scanf(" %d", &number);

  temp = number;    /* Copy to working storage */

  /* Reverse the number stored in temp */
  do
  {
    rebmun = 10*rebmun + temp % 10; /* Add the rightmost digit */
    temp = temp/10;  /* Remove the rightmost digit */
  } while(temp);     /* Continue while temp>0 */

  printf("\nThe number %d reversed is %d rebmun ehT\n",
       number, rebmun );
  return 0;
}
```

www.TopSage.com

这个程序的输出如下:

```
Enter a positive integer: 43
The number 43 reversed is 34 rebmun ehT
```

代码的说明

说明执行过程的最好方法是通过一个小例子来解释。假设用户输入的数是 43。读取输入的整数，存储到变量 number 中后，程序将 number 的值复制到变量 temp 中:

```
temp = number; /* Copy to working storage */
```

这是必须的，因为翻转数字的过程会破坏原来的值，而我们希望将原来的数和翻转后的数一起输出，翻转数字是在 do-while 循环内完成:

```
do
{
  rebmun = 10*rebmun + temp % 10; /* Add the rightmost digit */
  temp = temp/10;                 /* Remove the rightmost digit */
} while(temp);                    /* Continue while temp>0 */
```

在这个程序中，do-while 循环是最适合的，因为任何数都至少有一位数字。用取模运算符%计算除以 10 的余数，可以得到 temp 变量值中的最右边的一位数字。temp 原本是 43，则 temp%10 结果是 3。将 10*rebmun+temp%10 的值赋予 rebmun。变量 rebmun 的初始值是 0，所以在第一次迭代时，rebmun 存储了数字 3。

将输入值最右边的数字 3 保存到 rebmun 中后，就可以给 temp 除以 10，去掉这个数字。temp 的初始值是 43，所以 temp/10 的结果会四舍五入为 4。

在循环结束时，检查 while(temp)条件，而 temp 的值是 4，所以该条件是 true。因此返回循环的开始处，执行另一个迭代。

注意:

任何非零整数都会转换为 true，布尔值 false 对应 0。

这次，存储在 rebmun 中的值与 10 相乘，得到 30，再加上 temp%10 的余数 4，所以 rebmun 的结果是 34。然后将 temp 除以 10，得到 0。现在，到达循环迭代的结尾时，temp 是 0，即 false，所以循环结束，完成了数字的翻转。这个程序也可以翻转有更多数字的数。下面输入一个比较长的数，该程序的输出如下:

```
Enter a positive integer: 1234

The number 1234 reversed is 4321 rebmun ehT
```

这个循环和其他两个循环比较起来，使用的机会相当少。尽管如此，也应记住它，当需要至少执行一次循环时，do-while 是最佳的选择。

www.TopSage.com

4.11 continue 语句

有时不希望结束循环，但要跳过目前的迭代，继续执行下一个迭代。循环体内的 continue 语句就有这个作用，它可以编写为：

```
continue;
```

当然，continue 是一个关键字，不能将它用于其他目的。下面是使用 continue 语句的一个例子：

```
for(int day = 1; day<=7 ; ++day)
{
  if(day == 3)
    continue;
  /* Do something useful with day */
}
```

这个循环用 day 的值 1~7 执行某个操作。当 day 的值为 3 时，会执行 continue 语句，跳过当前迭代的其他语句，之后 day 的值是 4，循环继续下一个迭代。

本书将在后面介绍更多使用 continue 的例子。

4.12 设计程序

现在，在一个比较大的编程问题上测试前面学习过的技巧，应用本章和前一章学到的东西。本节还会介绍几个新的标准库函数，它们非常有用。

4.12.1 问题

本节要编写一个简单的 Simon 游戏，这是一个记忆测试游戏。计算机会在屏幕上将一串数字显示很短的时间。玩家必须在数字消失之前记住他们，然后输入这串数字。每次过关后，计算机会显示更长的一串数字，让玩家继续玩下去。玩家应尽可能使这个过程重复更多的次数。

4.12.2 分析

程序必须产生一连串 0~9 的整数，使它们在屏幕上显示 1 秒钟，之后删除它们。玩家试着输入这串数字。这串数字会一次比一次长，直到玩家输入错误为止。根据成功的次数和所花的时间来计分。然后程序会询问玩家，是否继续玩。

这个程序的逻辑相当简单，可以用如图 4-7 的流程图来说明。

www.TopSage.com

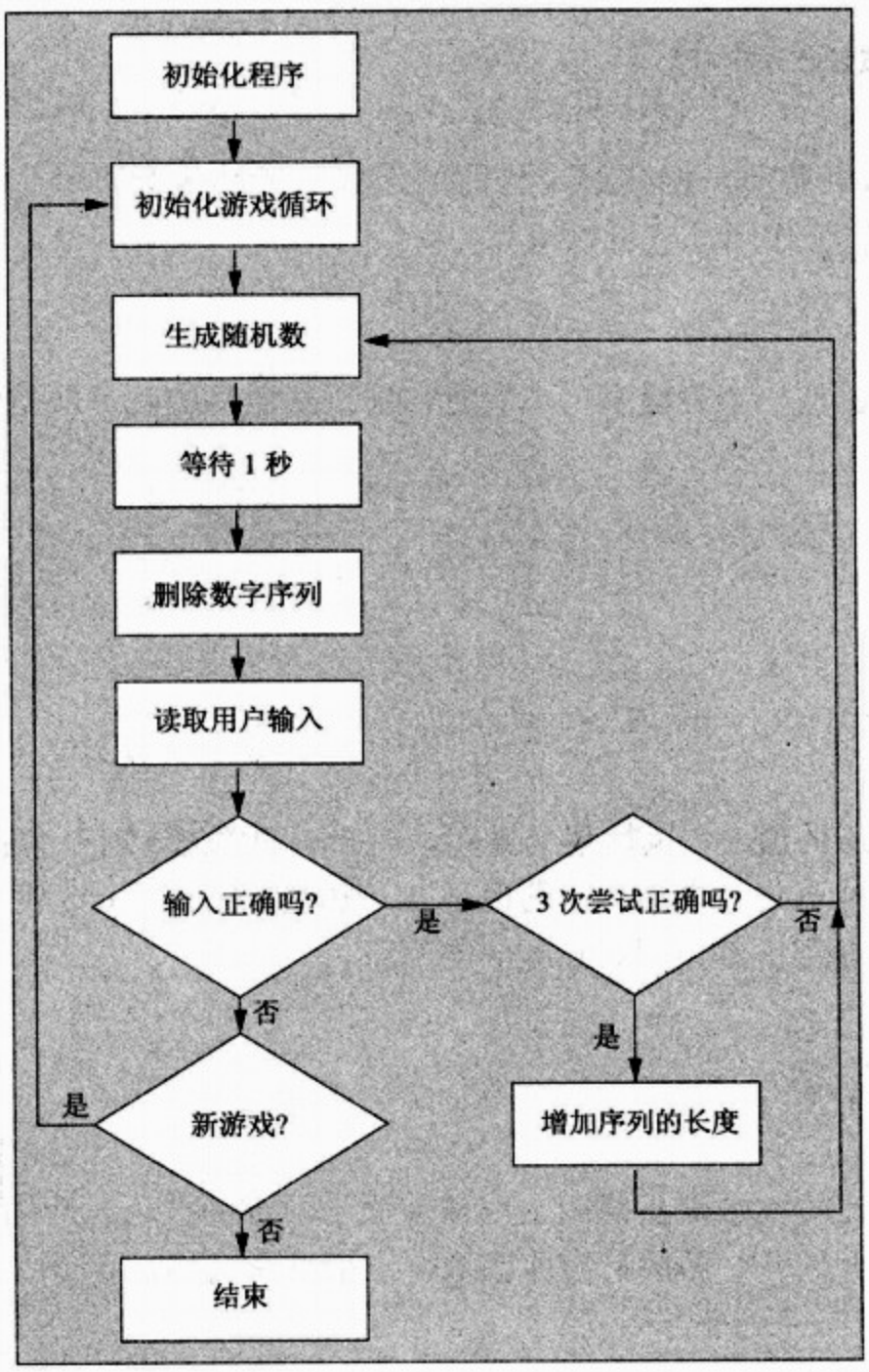


图 4-7 简单 Simon 游戏的基本逻辑

每个方块表示程序中的一个动作，菱形表示判断。下面将使用这个流程图作为编写程序的基础。

4.12.3 解决方案

本节列出解决该问题的步骤。

1. 步骤 1

首先为游戏编写一个主循环。玩家通常至少玩一次游戏，所以循环的检查应放在循环结束的地方。do while 循环在这里是最合适不过了。最初的程序代码如下：

```
/* Program 4.12 Simple Simon */
#include <stdio.h>     /* For input and output */
#include <ctype.h>     /* For toupper() function */
```

www.TopSage.com

```
int main(void)
{
  /* Records if another game is to be played */
  char another_game = 'Y';

  /* Rest of the declarations for the program */

  /* Describe how the game is played */
  printf("\nTo play Simple Simon, ");
  printf("watch the screen for a sequence of digits.");
  printf("\nWatch carefully, as the digits are only displayed"
         " for a second! ");
  printf("\nThe computer will remove them, and then prompt you ");
  printf("to enter the same sequence.");
  printf("\nWhen you do, you must put spaces between the digits. \n");
  printf("\nGood Luck!\nPress Enter to play\n");
  scanf("%c", &another_game);

  /* One outer loop iteration is one game */
  do
  {
    /* Code to play the game */

    /* Output the score when the game is finished */

    /* Check if a new game is required */
    printf("\nDo you want to play again (y/n)? ");
    scanf("%c", &another_game);
  } while(toupper(another_game) == 'Y');
  return 0;
}
```

只要玩家在一次游戏结束时输入 y 或 Y，就能再玩一次。注意，可以在 printf()语句中自动将两个字符串连接起来：

```
printf("\nWatch carefully, as the digits are only displayed"
       " for a second! ");
```

这个方法非常便于将一个很长的字符串拆成两行或好几行。只要将每一段字符串放在双引号内，编译器就会将它们组合成一个字符串。

2. 步骤 2

下一步，添加另一个变量 correct 的声明，程序需要记录玩家输入的数字项是否正确。这个变量用于控制玩一次游戏的循环次数。

```
/* Program 4.12 Simple Simon */
#include <stdio.h>      /* For input and output */
```

www.TopSage.com

```
#include <ctype.h>     /* For toupper() function */
#include <stdbool.h>   /* For bool, true, false */

int main(void)
{
  /* Records if another game is to be played */
  char another_game = 'Y';

  /* true if correct sequence entered, false otherwise */
  bool correct = true;

  /* Rest of the declarations for the program */

  /* Describe how the game is played */
  printf("\nTo play Simple Simon, ");
  printf("watch the screen for a sequence of digits.");
  printf("\nWatch carefully, as the digits are only displayed"
        " for a second! ");
  printf("\nThe computer will remove them, and then prompt you ");
  printf("to enter the same sequence.");
  printf("\nWhen you do, you must put spaces between the digits. \n");
  printf("\nGood Luck!\nPress Enter to play\n");
  scanf("%c", &another_game);

  /* One outer loop iteration is one game */
  do
  {
    correct = true; /* By default indicates correct sequence entered */

    /* Other code to initialize the game */

    /* Inner loop continues as long as sequences are entered correctly */
    while(correct)
    {
      /* Play the game */
    }

    /* Output the score when the game is finished */

    /* Check if new game required*/
    printf("\nDo you want to play again (y/n)? ");
    scanf("%c", &another_game);
  } while(toupper(another_game) == 'Y');
  return 0;
}
```

这里使用了_Bool变量correct，但因为<stdbool.h>给头文件添加了#include指令，所以可以将bool用作类型名。<stdbool.h>头文件还定义了符号true和false，它们分别对应1和0。

注意：

代码可以进行编译了，而且应该编译它，但还不应该执行它。当在开发程序时，应确保每步编写的代码都能编译。如果试图一口气编写出所有的程序代码，就可能出现上百个错误需要修正，而且更正了一个错误后，又出现其他更多的错误。这很令人气馁。每添加一些代码就检查程序，可以减少这个问题，有问题也比较容易处理。下面回过头来看看当前这个程序。如果执行它，计算机会完全由该程序接管，因为它包含一个无限循环。内部的 while 循环是个无限循环。这个循环的条件永远是 true，因为该循环没有对 correct 进行任何改变。稍后将在此添加一些代码。

3. 步骤 3

下面的这个工作稍有困难：生成一串随机数。这里要探讨两个问题。第一是生成一串随机数。第二是检查玩家的输入和计算机生成的数字串是否匹配。

生成一串数字的主要困难是：数字必须是随机的。每次调用标准函数 rand()时，它都会返回一个随机整数。要得到一个随机的数字，可以用%运算符计算 rand()的返回值除以 10 所得的余数。

为确保每次程序执行时，都能得到不同的数字串，还需要调用 srand()函数，用 time()函数的返回值初始化该数字串。函数 rand()和 srand()都需要在程序中包含<tdlib.h>头文件，time()函数需要用#include 指令在程序中包含<time.h>。

随机数字串需要显示两次：第一次是在程序开始时显示，之后删除它，第二次用于检查玩家的输入。可以将数字串存储为 unsigned long long 类型的一个整数。问题是如果遇到厉害的玩家，这串数字可能会非常长，超过 unsigned long long 类型的上限。还有一个可行的方法。只要每次调用 srand()得到相同的种子值，rand()函数就可以多次生成相同顺序的数字。这样就不需要存储数字串了，只要生成相同顺序的数字串两次即可。

现在给程序添加一些代码，生成随机数字串，并检查玩家的输入：

```
/* Program 4.12 Simple Simon */
#include <stdio.h>    /* For input and output */
#include <ctype.h>    /* For toupper() function */
#include <stdbool.h>  /* For bool, true, false */
#include <stdlib.h>   /* For rand() and srand() */
#include <time.h>     /* For time() function */

int main(void)
{
  /* Records if another game is to be played */
  char another_game = 'Y';

  /* true if correct sequence entered, false otherwise */
  bool correct = true;

  /* Number of sequences entered successfully */
  int counter = 0;
```

www.TopSage.com

```
int sequence_length = 0;/* Number of digits in a sequence */
time_t seed = 0;        /* Seed value for random number sequence */
int number = 0;         /* Stores an input digit */

/* Rest of the declarations for the program */

/* Describe how the game is played */
printf("\nTo play Simple Simon, ");
printf("watch the screen for a sequence of digits.");
printf("\nWatch carefully, as the digits are only displayed"
    " for a second! ");
printf("\nThe computer will remove them, and then prompt you ");
printf("to enter the same sequence.");
printf("\nWhen you do, you must put spaces between the digits. \n");
printf("\nGood Luck!\nPress Enter to play\n");
scanf("%c", &another_game);

/* One outer loop iteration is one game */
do
{
  correct = true; /* By default indicates correct sequence entered */
  counter = 0;    /* Initialize count of number of successful tries */
  sequence_length = 2; /* Initial length of a digit sequence */

  /* Other code to initialize the game */

/* Inner loop continues as long as sequences are entered correctly */
while(correct)
{
  /* On every third successful try, increase the sequence length */
  sequence_length += counter++%3 == 0;

  /* Set seed to be the number of seconds since Jan 1,1970 */
  seed = time(NULL);

  /* Generate a sequence of numbers and display the number */
  srand((unsigned int)seed); /* Initialize the random sequence */
  for(int i = 1; i <= sequence_length; i++)
    printf("%d ", rand() % 10); /* Output a random digit */

  /* Wait one second */

  /* Now overwrite the digit sequence */

  /* Prompt for the input sequence */

  /* Check the input sequence of digits against the original */
```

www.TopSage.com

```
      srand((unsigned int)seed); /* Restart the random sequence */
      for(int i = 1; i <= sequence_length; i++)
      {
        scanf("%d", &number);      /* Read an input number */
        if(number != rand() % 10) /* Compare against random digit */
        {
          correct = false;         /* Incorrect entry */
          break;                   /* No need to check further... */
        }
      }
      printf("%s\n", correct ? "Correct!" : "Wrong!");
    }

    /* Output the score when the game is finished */

    /* Check if new game required*/
    printf("\nDo you want to play again (y/n)? ");
    scanf("%c", &another_game);
  } while(toupper(another_game) == 'Y');
  return 0;
}
```

新添加的代码声明了5个要在while循环内用到的新变量，每当玩家成功过关，while循环继续执行。这个循环的每次迭代都显示一个玩家必须再次输入的数字串。变量counter记录玩家成功的次数，sequence_length记录当前数字串的长度。在声明这些变量时，已初始化了它们，但还必须在do-while循环中初始化这些值，以确保为每次游戏设置初始条件。上述代码还声明了一个long类型的变量seed，它作为参数传送给srand()，以初始化rand()函数返回的随机数字串。变量seed的值是在while循环内调用标准库函数time()得到的。

在while循环的开头，将表达式counter++%3 == 0的值加到sequence_length中。当counter是3的倍数时，这个表达式是1，否则是0。因此，玩家每成功3次，就将数字串的长度加1。在计算完这个表达式后，还将counter加1。

代码中还有一些其他要注意的事项。首先，将seed传给srand()函数时，其类型time_t要转换成unsigned int，因为srand()函数要求使用unsigned int类型，但是time()函数返回类型time_t的值。自1970年1月1日起至今的秒数超过了800 000 000，所以需要4字节的变量存储它。其次，rand()返回的随机整数除以10，得到的余数在0~9之间，这不是得到0~9之间的数字的最好方法，但很简单，足以满足这个程序的要求。尽管srand()函数生成的数字是随机分布的，但该数字中低位的十进制数字不一定是随机的。要获得随机数字，应将srand()函数生成的数字的整个取值范围分成10段，每一段对应0~9之间的每个数字。然后，根据每个数字所在的段，选择对应给定伪随机数的数字。

www.TopSage.com

数字串由 for 循环输出，它只输出 rand()返回值的低位十进制数字。然后，有一些注释指出还要添加一些代码，让数字串在屏幕上停留一秒后，才将其删除。之后的代码检查玩家输入的数字串。这段代码用最初使用的种子值调用函数 srand()，再次执行生成随机数的过程。每个输入的数字都和 rand()函数返回值的低位数字做比较。如果不一致，correct 就设定为 false，结束循环。当然，现在执行这个程序，数字串不会被清除，所以这个程序还不能使用。下一步要添加完成 while 循环的代码。

4. 步骤 4

必须在延迟 1 秒后，将数字串清除。如何让程序等待 1 秒？一种方法是使用另一个标准库函数 clock()，它返回从启动程序到当前的时间，单位是 tick。头文件<time.h>定义一个符号 CLOCKS_PER_SEC，它表示一秒有多少 tick。要使程序等待 1 秒，应等待函数 clock()的返回值递增到 CLOCKS_PER_SEC 为止，这表示过去了一秒。为此，可以存储函数 clock()返回的值，然后在一个循环内检查 clock()的返回值是否比先前存储的值大 CLOCKS_PER_SEC。用一个变量 now 存储当前的时间，这个循环的代码如下：

```
for( ;clock() - now < CLOCKS_PER_SEC; ); /* Wait one second */
```

还需要确定如何删除计算机生成的数字串。这其实非常简单。可以输出转义序列'\r'(回车键)，移到这行的开始处，然后输出足够的空格，覆盖掉数字串。下面填上 while 循环需要的代码：

```
/* Program 4.12 Simple Simon */
#include <stdio.h>    /* For input and output */
#include <ctype.h>    /* For toupper() function */
#include <stdbool.h>  /* For bool, true, false */
#include <stdlib.h>   /* For rand() and srand() */
#include <time.h>     /* For time() function */

int main(void)
{
  /* Records if another game is to be played */
  char another_game = 'Y';

  /* true if correct sequence entered, false otherwise */
  bool correct = true;

  /* Number of sequences entered successfully */
  int counter = 0;

  int sequence_length = 0; /* Number of digits in a sequence */
  time_t seed = 0;         /* Seed value for random number sequence */
  int number = 0;          /* Stores an input digit */

  /* Stores current time - seed for random values */
  time_t now = 0;
```

www.TopSage.com

```
/* Rest of the declarations for the program */

/* Describe how the game is played */
printf("\nTo play Simple Simon, ");
printf("watch the screen for a sequence of digits.");
printf("\nWatch carefully, as the digits are only displayed"
    " for a second! ");
printf("\nThe computer will remove them, and then prompt you ");
printf("to enter the same sequence.");
printf("\nWhen you do, you must put spaces between the digits. \n");
printf("\nGood Luck!\nPress Enter to play\n");
scanf("%c", &another_game);

/* One outer loop iteration is one game */
do
{
  correct = true; /* By default indicates correct sequence entered */
  counter = 0;    /* Initialize count of number of successful tries */
  sequence_length = 2; /* Initial length of a digit sequence */

  /* Other code to initialize the game */

  /* Inner loop continues as long as sequences are entered correctly */
  while(correct)
  {
    /* On every third successful try, increase the sequence length */
    sequence_length += counter++%3 == 0;

    /* Set seed to be the number of seconds since Jan 1,1970 */
    seed = time(NULL);

    now = clock();        /* record start time for sequence */

    /* Generate a sequence of numbers and display the number */
    srand((unsigned int)seed); /* Initialize the random sequence */
    for(int i = 1; i <= sequence_length; i++)
      printf("%d ", rand() % 10); /* Output a random digit */

    /* Wait one second */
    for( ;clock() - now < CLOCKS_PER_SEC; );

    /* Now overwrite the digit sequence */
    printf("\r");    /* go to beginning of the line */
    for(int i = 1; i <= sequence_length; i++)
      printf(" ");   /* Output two spaces */

    if(counter == 1) /* Only output message for the first try */
      printf("\nNow you enter the sequence - don't forget"
          " the spaces\n");
```

www.TopSage.com

```
      else
        printf("\r");   /* Back to the beginning of the line */

      /* Check the input sequence of digits against the original */
      srand((unsigned int)seed); /* Restart the random sequence */
      for(int i = 1; i <= sequence_length; i++)
      {
        scanf("%d", &number);      /* Read an input number */
        if(number != rand() % 10)  /* Compare against random digit */
        {
          correct = false;         /* Incorrect entry */
          break;                   /* No need to check further... */
        }
      }
      printf("%s\n", correct ? "Correct!" : "Wrong!");
    }

    /* Output the score when the game is finished */

    /* Check if new game required*/
    printf("\nDo you want to play again (y/n)? ");
    scanf("%c", &another_game);
  } while(toupper(another_game) == 'Y');
  return 0;
}
```

在输出数字串之前，先记录 clock()返回的时间。数字串显示后，for 循环会一直执行到 clock()的返回值比 now 变量值大 CLOCKS_PER_SEC 为止，CLOCKS_PER_SEC 表示 1 秒。

因为在显示数字串的过程中没有在屏幕上输出换行符，完成数字串的输出后，还在该数字串所在的行上。只输出回车符但不输出换行符，即只需输出'\r'，就可以将光标移到这行的开头。之后，给每个已显示的数字输出两个空格，用空白将它们覆盖掉。紧接着提示玩家输入刚刚显示的数字串。只在每一轮的第一次显示这条信息：否则会令人厌烦。在第二和第三轮，只需回到当前空白行的开始处，等待玩家的输入。

步骤 5

剩下就是一旦玩家出错，就生成并显示分数了。计分时要使用成功的次数和所花的时间。可以为正确输入的每个数字指定 100 点，再将总分除以所花的时间。这说明是答得越快，分数越高，正确输入的数字串越多，分数越高。

事实上，这个程序还有一个问题需要处理。如果玩家其中的一个数字输入错误，循环就结束，然后询问玩家是否要再玩一次。然而，如果不是最后一个数字输错了，就可以用下一个输入的数字作为问题“还要再玩一次吗(y/n)?”的答案，因为这些数字仍然在键盘的缓冲区内。所以必须删除还在键盘缓冲区内的数据。这有两个问题：第一，如何寻址键盘缓冲区；第二，如何清除缓冲区。

www.TopSage.com

注意：

键盘缓冲区是用来存储键盘输入的内存。scanf()函数是在键盘缓冲区查找输入数据，而不是直接从键盘上读取数据。

标准输入和输出——键盘和屏幕——有两个缓冲区：一个用于输入，另一个用于输出。标准输入输出流分别称为 stdin 和 stdout。要指定键盘输入缓冲区，只需使用名称 stdin。现在知道如何指定缓冲区了，该如何删除其中的信息？标准库函数 fflush()就是用于清除缓冲区的。这个函数多半用于文件，详见本书后面的内容，但事实上它可用于任何缓冲区。只要将流名称作为参数传送给该函数，就指定了要清除哪个缓冲区。所以清除输入缓冲区的内容可以使用这个语句：

```
fflush(stdin);          /* Flush the stdin buffer */
```

下面是完整的程序代码，包含了计算分数和清除输入缓冲区：

```
/* Program 4.12 Simple Simon */
#include <stdio.h>    /* For input and output */
#include <ctype.h>    /* For toupper() function */
#include <stdbool.h>  /* For bool, true, false */
#include <stdlib.h>   /* For rand() and srand() */
#include <time.h>     /* For time() and clock() */

int main(void)
{
  /* Records if another game is to be played */
  char another_game = 'Y';

  /* true if correct sequence entered, false otherwise */
  int correct = false;

  /* Number of sequences entered successfully */
  int counter = 0;

  int sequence_length = 0; /* Number of digits in a sequence */
  time_t seed = 0;         /* Seed value for random number sequence */
  int number = 0;          /* Stores an input digit */

  time_t now = 0; /* Stores current time - seed for random values */
  int time_taken = 0; /* Time taken for game in seconds */

  /* Describe how the game is played */
  printf("\nTo play Simple Simon, ");
  printf("watch the screen for a sequence of digits.");
  printf("\nWatch carefully, as the digits are only displayed"
     " for a second! ");
  printf("\nThe computer will remove them, and then prompt you ");
  printf("to enter the same sequence.");
  printf("\nWhen you do, you must put spaces between the digits. \n");
```

www.TopSage.com

```
printf("\nGood Luck!\nPress Enter to play\n");
scanf("%c", &another_game);

/* One outer loop iteration is one game */
do
{
  correct = true; /* By default indicates correct sequence entered */
  counter = 0;    /* Initialize count of number of successful tries*/
  sequence_length = 2;  /* Initial length of a digit sequence */
  time_taken = clock(); /* Record current time at start of game */

  /* Inner loop continues as long as sequences are entered correctly */
  while(correct)
  {
    /* On every third successful try, increase the sequence length */
    sequence_length += counter++%3 == 0;

    /* Set seed to be the number of seconds since Jan 1,1970 */
    seed = time(NULL);

    now = clock(); /* record start time for sequence */

    /* Generate a sequence of numbers and display the number */
    srand((unsigned int)seed); /* Initialize the random sequence */
    for(int i = 1; i <= sequence_length; i++)
      printf("%d ", rand() % 10); /* Output a random digit */

    /* Wait one second */
    for( ;clock() - now < CLOCKS_PER_SEC; );

    /* Now overwrite the digit sequence */
    printf("\r"); /* go to beginning of the line */
    for(int i = 1; i <= sequence_length; i++)
      printf(" "); /* Output two spaces */

    if(counter == 1) /* Only output message for the first try */
      printf("\nNow you enter the sequence - don't forget"
        " the spaces\n");
    else
      printf("\r"); /* Back to the beginning of the line */

    /* Check the input sequence of digits against the original */
    srand((unsigned int)seed);  /* Restart the random sequence */
    for(int i = 1; i <= sequence_length; i++)
    {
      scanf("%d", &number);      /* Read an input number */
      if(number != rand() % 10)  /* Compare against random digit */
      {
        correct = false;         /* Incorrect entry */
```

www.TopSage.com

```
            break;                    /* No need to check further... */
          }
        }
        printf("%s\n", correct? "Correct!" : "Wrong!");
      }

      /* Calculate total time to play the game in seconds)*/
      time_taken = (clock() - time_taken) / CLOCKS_PER_SEC;

      /* Output the game score */
      printf("\n\n Your score is %d", --counter * 100 / time_taken);

      fflush(stdin);

      /* Check if new game required*/
      printf("\nDo you want to play again (y/n)? ");
      scanf("%c", &another_game);

      } while(toupper(another_game) == 'Y');
    return 0;
}
```

函数 fflush() 需要的声明包含在<stdio.h>头文件中，它已经用#include 指令包含在程序中了。下面是程序的输出：

```
To play Simple Simon, watch the screen for a sequence of digits.
Watch carefully, as the digits are only displayed for a second!
The computer will remove them, and then prompt you to enter the same
sequence.
When you do, you must put spaces between the digits.

Good Luck!
Press Enter to play

Now you enter the sequence - don't forget the spaces
2 1 4
Correct!
8 7 1
Correct!
4 1 6
Correct!
7 9 6 6
Correct!
7 5 4 6
Wrong!

Your score is 16
Do you want to play again (y/n)? n
```

www.TopSage.com

4.13 小结

本章介绍了使用循环重复执行动作的所有知识。使用前面所学的强大的编程工具，就可以创建相当复杂的程序了。我们可以使用 3 个不同的循环来重复执行语句块：

- for 循环一般用于计算循环的次数，在该循环中，控制变量的值在每次迭代时递增或递减指定的值，直到到达某个最终值为止。
- while 循环只要给定的条件为 true 就继续执行。如果循环条件一开始就是 false，循环语句块就根本不执行。
- do-while 循环类似于 while 循环，但其循环条件在循环语句块执行后检查。因此循环语句块至少会执行一次。

下面重申前面提过的规则和建议：

- 开始编写程序前，先规划好过程和计算的逻辑，将它写下来，最好采用流程图的形式。试着从侧面思考问题，这也许比直接的方法更好。
- 理解运算符的优先级，以正确计算复杂的表达式。如果不能确定运算符的优先级，就应使用括号，确保表达式完成预期的操作，使用括号更便于理解复杂的表达式。
- 给程序加上注释，全面解释它们的操作和使用。要假设这些注释是为了方便别人阅读这个程序，并加以扩展与修改。声明变量时应说明它们的作用。
- 程序的可读性是最重要的。
- 在复杂的逻辑表达式中尽量避免使用!运算符。
- 使用缩进格式，可视化地表达出程序的结构。

采纳这些建议，就可以阅读下一章了。当然别忘了完成所有的习题。

4.14 习题

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容。还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 4.1 编写一个程序，生成一个乘法表，其大小由用户输入来决定。例如，如果表的大小是 4，该表就有 4 行 4 列。行和列标记为 1~4。表中的每一单元格都包含对应的行列之积，因此第 3 行第 4 列的单元格包含 12。

习题 4.2 编写一个程序，为 0~127 之间的字符码输出可打印的字符。输出每个字符码和它的符号，这两个字符占一行。列要对齐(提示：可以使用在 ctype.h 中声明的 isgraph() 函数，确定哪个字符是可以打印的)。

习题 4.3 扩展上一题，给每个空白字符输出对应的名称，例如 newline、space、tab 等。

习题 4.4 使用嵌套循环输出一个用星号绘制的盒子，与程序 4.2 类似，但是它的宽和高由用户输入。例如 10 字符宽、7 字符高的盒子如下：

www.TopSage.com

```
******
*    *
*    *
*    *
*    *
*    *
******
```

习题 4.5　修改程序 4.7 的猜谜游戏，在玩家猜错数字后，可以用一个选项让玩家继续玩下去，且想玩多久就玩多久。

www.TopSage.com

www.TopSage.com

第 5 章

数　组

我们经常需要在程序里存储某种类型的大量数据值。例如，如果编写一个程序，追踪一支篮球队的成绩，就要存储一个赛季的各场分数和各个球员的得分，然后输出某个球员的整季得分，或在赛事进行过程中计算出赛季的平均得分。我们可以利用前面所学的知识编写一个程序，为每个分数使用不同的变量。然而，如果一个赛季里有非常多的赛事，这会非常繁琐，因为有球赛的每个球员都需要许多变量。所有篮球分数的类型都相同，不同的是分值，但它们都是篮球赛的分数。理想情况下，应将这些分值组织在一个名称下，例如球员的名字，这样就不需要为每个数据项定义变量了。

本章将介绍如何在 C 程序中使用数组，然后探讨程序使用数组时，如何通过一个名称来引用一组数值。

本章的主要内容：

- 什么是数组
- 如何在程序中使用数组
- 数组如何使用内存
- 什么是多维数组
- 如何编写程序，计算帽子的尺寸
- 如何编写井字游戏

5.1　数组简介

说明数组的概念及其作用的最好方法，是通过一个例子，来说明使用数组后程序会变得非常简单。这个例子将计算某班学生的平均分数。

5.1.1　不用数组的程序

要计算某班学生的平均分数，假设该班只有 10 位学生(主要是避免键入太多的数字)。计算一组数字的平均值，要将它们全加起来，再除以数字的个数(在这里是除以 10)：

```
/* Program to Averaging ten numbers without storing the numbers */
#include <stdio.h>

int main(void)
{
```

www.TopSage.com

```
  int number = 0;  /* Stores a number */
  int count = 10;  /* Number of values to be read */
  long sum = 0L;   /* Sum of the numbers */
  float average = 0.0f; /* Average of the numbers */

  /* Read the ten numbers to be averaged */
  for(int i = 0; i < count; i ++)
  {
    printf("Enter grade: ");
    scanf("%d", &number);  /* Read a number */
    sum += number;         /* Add it to sum */
  }

  average = (float)sum/count; /* Calculate the average */

  printf("\nAverage of the ten numbers entered is: %f\n", average);
  return 0;
}
```

如果只对平均值感到兴趣，就不需要存储上面的分数。这个程序将所有的分数全部相加后，除以 count(其值是 10)。这个简单的程序只使用了一个变量 number，来存储循环中输入的每个分数。循环在 i 的值为 0, l, 2, 3...9 时执行，共迭代 10 次。本书前面也完成过这类任务，所以这个程序应该很简单。

假设要将这个程序开发成为一个更复杂的程序，需要在以后输入一些数值，输出每个人的分数，最后输出平均分。在上面的程序中，只有一个变量。每次加一个分数，旧的分值就被覆盖掉，不能再次使用。

我们可以声明 10 个整数变量来存储分数，但是不能用 for 循环输入这些数值。而必须添加代码，逐个读入这些数值。不过这样太繁琐。

```
/* Program 5.2 Averaging ten numbers - storing the numbers the hard way */
#include <stdio.h>

int main(void)
{
  int number0 = 0, number1 = 0, number2 = 0, number3 = 0, number4 = 0;
  int number5 = 0, number6 = 0, number7 = 0, number8 = 0, number9 = 0;

  long sum = 0L;         /* Sum of the numbers */
  float average = 0.0f; /* Average of the numbers */

  /* Read the ten numbers to be averaged */
  printf("Enter the first five numbers,\n");
  printf("use a space or press Enter between each number.\n");
  scanf("%d%d%d%d%d", &number0, &number1, &number2, &number3, &number4);
  printf("Enter the last five numbers,\n");
  printf("use a space or press Enter between each number.\n");
  scanf("%d%d%d%d%d", &number5, &number6, &number7, &number8, &number9);
```

www.TopSage.com

```
    /* Now we have the ten numbers, we can calculate the average */
    sum = number0 + number1+ number2 + number3 + number4+
          number5 + number6 + number7 + number8 + number9;
    average = (float)sum/10.0f;

    printf("\nAverage of the ten numbers entered is: %f\n", average);
    return 0;
}
```

这对只有 10 位学生没问题，但如果班里有 30、100 或 1 000 位学生，该怎么办？此时这个方法就不切实际，而应使用数组。

5.1.2 什么是数组

数组是一组数目固定、类型相同的数据项，数组中的数据项称为元素。数组的重要特性是：数组中的元素个数固定，每个数组的元素都是 int、long 或其他类型。所以可以有元素类型是 int 的数组，元素类型是 float 的数组，元素类型是 long 的数组等。

下面的数组声明非常类似于声明一个含有单一数值的正常变量，但要在名称后的方括号中放置一个数。

```
long numbers[10];
```

括号中的数字定义了要存放在数组中的元素个数，称为数组维 (array dimension)。这里有一个很重要的特性：存储在数组中的每个数据项都用相同的名称访问，在这个例子中，该名称就是 numbers。

如果只有一个变量名称，但存储了 10 个数值，要如何区分它们?数组中的每个值都由索引值(index value)来识别。索引值是一个整数，放在数组名称后的方括号内。数组中的每个元素都有一个不同的索引值，且索引值是从 0 开始的连续整数。前面 numbers 数组的元素索引值是 0~9。索引值 0 表示第一个元素，索引值 9 表示最后一个元素。要访问某个元素，只需在数组名称后的方括号中放入一个适当的索引值即可。因此数组元素可表示为 numbers[0]、numbers[1]、numbers[2]...numbers[9]。如图 5-1 所示。

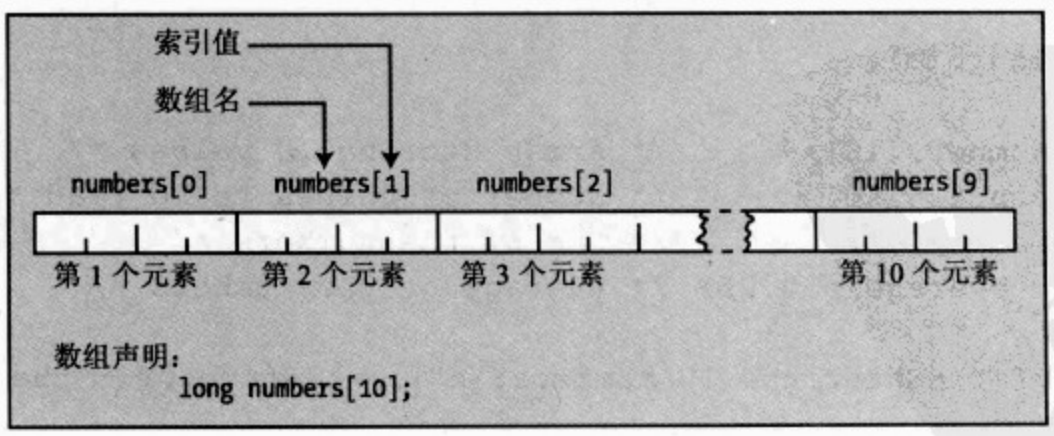


图 5-1　访问数组的元素

www.TopSage.com

索引值是从 0 开始，不是 1。第一次使用数组时，这是一个常犯的错误，有时这称为 off-by-one 错误。在一个 10 元素数组中，最后一个元素的索引值是 9。要访问数组中的第 4 个值，应使用表达式 numbers[3]。数组元素的索引值是与第 1 个元素的偏移量。第 1 个元素的偏移量是 0，第 2 个元素与第一个元素的偏移量是 1，第 3 个元素与第一个元素的偏移量是 2，依此类推。

要访问 numbers 数组元素的值，也可以在数组名称后的方括号内放置表达式，该表达式的结果必须是一个整数，对应于一个可能的索引值。例如 numbers[i-2]。如果 i 的值是 3，就访问数组中的第 2 个元素 numbers[1]。因此，有两种方法来指定索引值，以访问数组中的某个元素。其一，可以使用一个简单的整数，明确指定要访问的元素。其二，可以使用一个在执行程序期间计算的整数表达式。使用表达式的唯一限制是，它的结果必须是整数，该整数必须是对数组有效的索引值。

注意，如果在程序里使用的索引值超过了这个数组的合法范围，程序将不能正常运作。编译器检查不出这种错误，所以程序仍可以编译，但是执行是有问题的。在最好的情况下，是从某处提取了一个垃圾值，所以结果是错误的，且每次执行的结果都不会相同。在最糟的情况下，程序可能会覆盖重要的信息，且锁死计算机，需要重启计算机。有时，这对程序的影响比较微妙：程序有时能正常工作，有时不能，或者程序看起来工作正常，但结果是错误的，只是不明显。因此，一定要细心检查数组索引是否在合法范围内。

5.1.3 使用数组

讨论了许多理论内容后，下面需要解决平均分问题。这里将刚刚学到的数组知识用于下面的练习。

试试看：使用数组计算平均分

了解了数组后，就可以使用一个数组存储所有要平均的分数。即存储所有分数，以便重复使用它们。现在重写这个程序，计算 10 个分数的平均值：

```
/* Program 5.3 Averaging ten numbers - storing the numbers the easy way */
#include <stdio.h>

int main(void)
{
  int numbers[10];        /* Array storing 10 values */
  int count = 10;         /* Number of values to be read */
  long sum = 0L;          /* Sum of the numbers */
  float average = 0.0f;   /* Average of the numbers */

  printf("\nEnter the 10 numbers:\n"); /* Prompt for the input */

  /* Read the ten numbers to be averaged */
  for(int i = 0; i < count; i ++)
  {
```

www.TopSage.com

```
    printf("%2d> ",i+1);
    scanf("%d", &numbers[i]); /* Read a number */
    sum += numbers[i]; /* Add it to sum */
  }

  average = (float)sum/count; /* Calculate the average */

  printf("\nAverage of the ten numbers entered is: %f\n", average);
  return 0;
}
```

程序输出如下:

```
Enter the ten numbers:
1> 450
2> 765
3> 562
4> 700
5> 598
6> 635
7> 501
8> 720
9> 689
10> 527

Average of the ten numbers entered is: 614.700000
```

代码的说明

程序由常见的#include <stdio.h>开始,因为这里要使用 printf()和 scanf()函数。在 main()函数的一开始,声明一个包含 10 个整数的数组,然后是一些计算所需的变量:

```
int numbers[10];        /* Array storing 10 values */
int count = 10;         /* Number of values to be read */
long sum = 0L;          /* Sum of the numbers */
float average = 0.0f;   /* Average of the numbers */
```

然后,用下面的语句提示输入分数:

```
printf("\nEnter the 10 numbers:\n"); /* Prompt for the input */
```

接下来,用一个循环去读入数值且累加它们:

```
for(int i = 0; i < count; i ++)
{
  printf("%2d> ",i+1);
  scanf("%d", &numbers[i]); /* Read a number */
  sum += numbers[i];        /* Add it to sum */
}
```

for 循环采用首选格式,只要 i 不等于 count,循环就继续执行。一般应把 for 循环写

www.TopSage.com

成这种格式。循环的计数是从 0 到 9，而不是从 1 到 10，所以可以直接使用循环变量 i 访问数组的每个成员。print()输出 i+1 的当前值，之后输出>，结果如上面所示。这里使用格式指定符 %2d，确保每个值在两个字符宽的字段中输出，所以，这些分数排列得很整齐。如果使用 %d，这 10 个值的输出将会不整齐。

使用函数 scanf()将输入的每个值读入数组的元素 i 中：第 1 个值存储在 number[0]中，第 2 个输入值存储到 number[1]中，……第 10 个输入值存储到 number[9]中。在循环的每次迭代中，都会把读入的值加到 sum 中。

当循环结束时，用下面的语句计算并显示平均值：

```
average = (float)sum/count; /* Calculate the average */

printf("\nAverage of the ten numbers entered is: %f\n", average);
```

计算平均值的方法用 sum 除以 count，count 的值是 10。注意在 printf()中，告诉编译器将 sum 转换成类型 float(它声明的类型是 long)。这可以确保使用浮点数执行除法操作，因此不会舍弃结果的小数部分。

试试看：检索存储的数值

上面的例子可以稍微进行扩充，以展示数组的一个优点。这里对原来的程序作了一点修改(在下列代码中以粗体字显示)，现在这个程序显示所有输入的值。把这些值存储在数组中，就可以随时用各种不同的方法访问和处理它们。

```
/* Program 5.4 Reusing the numbers stored */
#include <stdio.h>

int main(void)
{
  int numbers[10];        /* Array storing 10 values */
  int count = 10;         /* Number of values to be read */
  long sum = 0L;          /* Sum of the numbers */
  float average = 0.0f;   /* Average of the numbers */

  printf("\nEnter the 10 numbers:\n"); /* Prompt for the input */

  /* Read the ten numbers to be averaged */
  for(int i = 0; i < count; i++)
  {
    printf("%2d> ",i+1);
    scanf("%d", &numbers[i]);   /* Read a number */
    sum += numbers[i];          /* Add it to sum */
  }

  average = (float)sum/count; /* Calculate the average */

  for(int i = 0; i < count; i++)
    printf("\nGrade Number %d was %d", i+1, numbers[i]);
```

www.TopSage.com

```
  printf("\nAverage of the ten numbers entered is: %f\n", average);
  return 0;
}
```

这个程序的输出如下:

```
Enter the ten numbers:
1> 56
2> 64
3> 34
4> 51
5> 52
6> 78
7> 62
8> 51
9> 47
10> 32

Grade No 1 was 56
Grade No 2 was 64
Grade No 3 was 34
Grade No 4 was 51
Grade No 5 was 52
Grade No 6 was 78
Grade No 7 was 62
Grade No 8 was 51
Grade No 9 was 47
Grade No 10 was 32
Average of the ten numbers entered is: 52.700001
```

代码的说明

这里只解释新增的部分，在循环中重用数组的元素:

```
for(int i = 0; i < count; i++)
  printf("\nGrade Number %d was %d", i+1, numbers[i]);
```

上述代码只添加了另一个循环，遍历数组中的元素，并输出每个值。使用循环控制变量作为每个元素对应的序号，并访问对应的数组元素。这些元素的数值显然对应输入的数字。要从1开始获取分数，可以在输出语句中使用表达式i+1，得到从1到10的分数，因为i是从0到9。

在深入探讨数组之前，需要研究变量如何存储到计算机的内存中，以及数组和变量有什么不同。

5.2 内存

下面快速复习一下第2章介绍的内存知识。计算机的内存可以看做一排很整齐的盒

www.TopSage.com

子，每个盒子都有两种状态：满(称为 1)和空(称为 0)。每个盒子都包含一个二进制数，称为位。图 5-2 显示了内存中的一个字节序列。

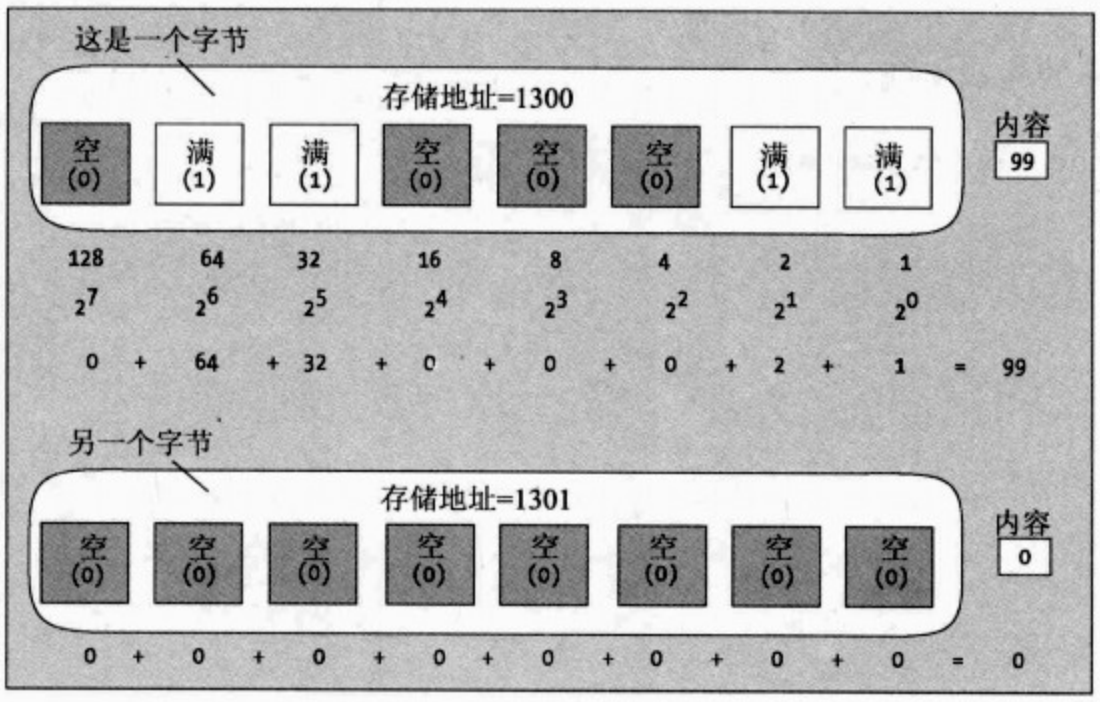


图 5-2　内存中的字节

为了方便起见，图 5-2 中的位组合为 8 个一组，每组称为字节。要辨识每个字节，以访问其内容，用数字标记字节，0 表示内存中的第一个字节，1 表示内存中的第 2 个字节，以此类推，直到内存中的最后一个字节。字节的标记称为地址。

前面使用了寻址运算符&，它广泛用于 scanf()函数。它放在变量名称之前，因为函数要将键盘输入的数据存入变量。只把这个变量名称用作函数的参数，函数就可以使用变量存储的值。而把寻址运算符放在变量名称之前，将它用作函数的参数，函数就可以利用这个变量的地址。这样，函数在这个地址中存储信息，修改在这个变量中存储的值。理解寻址运算符的最好方法是多使用它，下面就是一个例子。

试试看：使用寻址运算符

程序中使用的每个变量都会占用一定量的内存(以字节为单位)，占用的内存量取决于变量的类型。下面的程序要找出不同类型变量的地址：

```
/* Program 5.5 Using the & operator */
#include<stdio.h>

int main(void)
{
  /* declare some integer variables */
  long a = 1L;
  long b = 2L;
  long c = 3L;

  /* declare some floating-point variables */
  double d = 4.0;
  double e = 5.0;
  double f = 6.0;
```

www.TopSage.com

```
  printf("A variable of type long occupies %d bytes.", sizeof(long));
  printf("\nHere are the addresses of some variables of type long:");
  printf("\nThe address of a is: %p The address of b is: %p", &a, &b);
  printf("\nThe address of c is: %p", &c);
  printf("\n\nA variable of type double occupies %d bytes.",
  sizeof(double));
  printf("\nHere are the addresses of some variables of type double:");
  printf("\nThe address of d is: %p The address of e is: %p", &d, &e);
  printf("\nThe address of f is: %p\n", &f);
  return 0;
}
```

这个程序的输出如下:

```
A variable of type long occupies 4 bytes.
Here are the addresses of some variables of type long:
The address of a is: 0064FDF4  The address of b is: 0064FDF0
The address of c is: 0064FDEC

A variable of type double occupies 8 bytes.
Here are the addresses of some variables of type double:
The address of d is: 0064FDE4  The address of e is: 0064FDDC
The address of f is: 0064FDD4
```

读者得到的地址值肯定与上述的不同。得到什么地址值取决于所使用的操作系统及运行本程序的同时还运行了什么其他程序。实际的地址值由程序加载到内存的什么地方来决定，而每次执行程序时，这都是不同的。

代码的说明

声明 3 个 long 类型的变量和 3 个 double 类型的变量:

```
/* declare some integer variables */
long a = 1L;
long b = 2L;
long c = 3L;

/* declare some floating-point variables */
double d = 4.0;
double e = 5.0;
double f = 6.0;
```

接下来输出 long 变量占用的字节数，跟着输出这 3 个变量的地址:

```
printf("A variable of type long occupies %d bytes.", sizeof(long));
printf("\nHere are the addresses of some variables of type long:");
printf("\nThe address of a is: %p The address of b is: %p", &a, &b);
printf("\nThe address of c is: %p", &c);
```

www.TopSage.com

地址运算符&放在每个变量名称之前。这里还使用了一个新的格式指定符 %p，来输出变量的地址。这个格式指定符指定输出一个内存地址，其值为十六进制。内存地址一般是 16、32 或 64 位，地址的大小决定了可以引用的最大内存量。在本例使用的计算机上，内存地址是 32 位，表示为 8 个十六进制数；在其他机器上，这可能不同。

然后，输出 double 变量占用的字节数，接着输出这 3 个变量的地址：

```
printf("\n\nA variable of type double occupies %d bytes.", sizeof(double));
printf("\nHere are the addresses of some variables of type double:");
printf("\nThe address of d is: %p The address of e is: %p", &d, &e);
printf("\nThe address of f is: %p\n", &f);
```

事实上，程序本身不如输出那么有趣。看看显示出来的地址，地址值逐渐变小，呈等差排列，如图 5-3 所示。在本例使用的计算机上，地址 b 比 a 低 4，c 比 b 低 4。这是因为每个 long 类型的变量占用 4 个字节。变量 d、e、f 也是如此，但它们的差是 8。这是因为类型 double 的值用 8 个字节来存储。

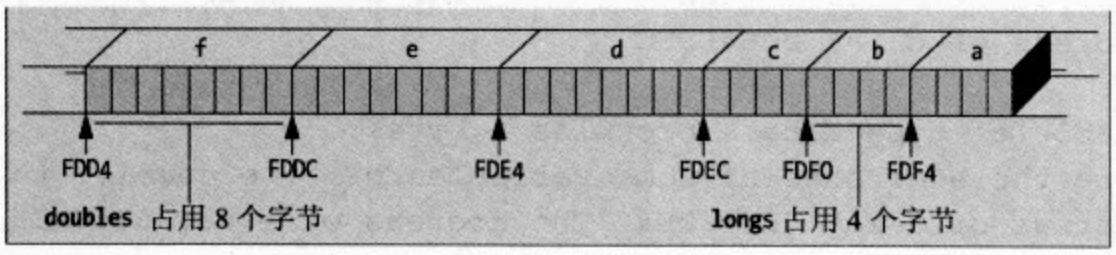


图 5-3　变量在内存中的地址

注意：

如果变量地址之间的间隔大于变量占用的字节数，可能是因为程序编译为调试版本。在调试模式下，编译器会配置额外的空间，以存储变量的其他信息，这些信息在程序以调试模式下执行时使用。

5.3　数组和地址

在下面的数组中，名称 number 指定了存储数据项的内存区域地址，把该地址和索引值组合起来就可以找到每个元素，因为索引值表示各个元素与数组开头的偏移量。

```
long number[10];
```

声明一个数组时，要给编译器提供为数组分配内存所需的所有信息，包括值的类型和元素的个数，而值的类型决定了每个元素需要的字节数。数组名称指定了数组从内存的什么地方开始存储，索引值指定了从开头到所需的元素之间有多少个元素。数组元素的地址是数组开始的地址加上元素的索引值乘以数组中每个元素类型所需的字节数。图 5-4 是数组变量保存在内存中的情形。

www.TopSage.com

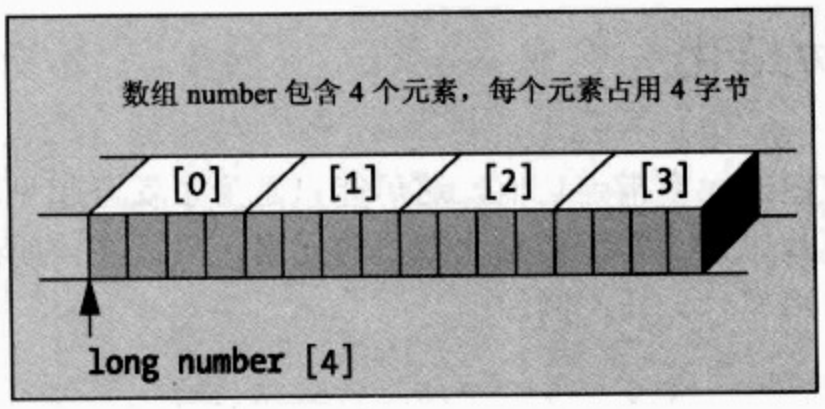


图 5-4　数组在内存中的组织方式

获取数组元素地址的方式类似于普通变量。对 value 整数变量，可以用以下语句输出它的地址：

```
printf("\n%p", &value);
```

要输出 number 数组的第 3 个元素的地址，可以编写如下代码：

```
printf("\n%p", &number[2]);
```

在方括号中使用 2 来访问第 3 个元素。这里用寻址运算符获得元素的地址。如果使用相同的语句但不用&，就会显示存储在数组第 3 个元素中的值，而不是它的地址。

下面用一些代码来说明。下面的代码段设置了数组中的元素值，然后输出了每个元素的地址和内容：

```
int data[5];
for(int i = 0 ; i<5 ; i++)
{
  data[i] = 12*(i+1);
  printf("\ndata[%d] Address: %p Contents: %d", i, &data[i], data[i]);
}
```

for 循环变量 i 遍历了 data 数组中的所有合法索引值。在这个循环中，位于索引位置 i 上的元素值设置为 12*(i+1)。输出语句显示了当前的元素及其索引值，由 i 的当前值决定的数组元素的地址，以及存储在元素中的值。如果这些代码放在一个程序中，则输出如下：

```
data[0] Address: 0x0012ff58 Contents: 12
data[1] Address: 0x0012ff5c Contents: 24
data[2] Address: 0x0012ff60 Contents: 36
data[3] Address: 0x0012ff64 Contents: 48
data[4] Address: 0x0012ff68 Contents: 60
```

i 的值显示在数组名后面的括号中。每个元素的地址都大于前一个元素，所以每个元素占用 4 个字节。

www.TopSage.com

5.4 数组的初始化

当然，可以给数组的元素指定初值，这可能只是为了安全起见。预先确定数组元素的初始值，更便于查找错误。为了初始化数组的元素，只需在声明语句中，在大括号中指定一列初值，它们用逗号分开，例如：

```
double values[5] = {1.5, 2.5, 3.5, 4.5, 5.5};
```

这个语句声明了一个包含 5 个元素的数组 value。values[0]的初值是 1.5，value[1]的初值是 2.5，依此类推。

要初始化整个数组，应使每个元素都有一个值。如果初值的个数少于元素数，没有初值的元素就设成 0。因此如果编写

```
double values[5] = {1.5, 2.5, 3.5};
```

前 3 个元素用括号内的值初始化，后两个元素初始化为 0。

如果初值的个数超过数组元素的个数，编译器就会报错。在指定一列初始值时，不必提供数组的大小，编译器可以从该列值中推断出元素的个数：

```
int primes[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

上述语句中的数组大小由列表中的初始值个数来确定，所以 primes 数组有 10 个元素。

5.5 确定数组的大小

sizeof 运算符可以计算出指定类型的变量所占用的字节数。对类型名称应用 sizeof 运算符，如下：

```
printf("\nThe size of a variable of type long is %d bytes.", sizeof(long));
```

sizeof 运算符后类型名称外的括号是必须的。如果漏了它，代码就不会编译。也可以对变量应用 sizeof 运算符，它会计算出该变量所占的字节数。例如，用下列语句声明变量 value：

```
double value = 1.0;
```

使用下面的语句输出 value 所占的字节数：

```
printf("\nThe size of value is %d bytes.", sizeof value);
```

注意，在这个例子，sizeof 的操作数可以不加括号，加上也无妨。这条语句会输出如下一行：

```
The size of value is 8 bytes.
```

www.TopSage.com

这是因为 double 类型的变量在内存中占用 8 个字节。当然可以存储应用 sizeof 运算符算出的字节数：

```
int value_size = sizeof value;
```

sizeof 运算符也可以用于数组。下面的语句声明一个数组：

```
double values[5] = {1.5, 2.5, 3.5, 4.5, 5.5};
```

现在可以用下面的语句输出这个数组所占的字节数：

```
printf("\nThe size of the array, values, is %d bytes.", sizeof values);
```

输出如下：

```
The size of the array, values, is 40 bytes.
```

也可以用表达式 sizeof values[0]计算出数组中一个元素所占的字节数。这个表达式的值是 8。当然，使用元素的合法索引值可以产生相同的结果。因此可以用 sizeof 运算符计算数组中元素的数目：

```
int element_count = sizeof values/sizeof values[0];
```

执行这条语句后，变量 element_count 就含有数组 values 中元素的数量。

可以将 sizeof 运算符应用于数据类型，所以可以重写先前的语句，计算数组元素的数量，如下所示：

```
int element_count = sizeof values/sizeof(double);
```

这会得到与前面相同的结果，因为数组的类型是 double，sizeof(double)会得到 double 值占用的字节数。有时偶尔会使用错误的类型，所以最好使用前一条语句。

sizeof 运算符应用于变量时不需要使用括号，但一般还是使用它们，所以前面的例子可以编写为：

```
int ElementCount = sizeof(values)/sizeof(values[0]);
printf("The size of the array is %d elements ", sizeof(values));
printf("and there are %d elements of %d bytes each",
      ElementCount, sizeof(values[0]);
```

这些语句的输出如下：

```
The size of the array is 40 bytes and there are 5 elements of 8 bytes each
```

5.6 多维数组

下面介绍二维数组。二维数组可以声明如下：

```
float carrots[25][50];
```

www.TopSage.com

这行语句声明了一个数组 carrots，它包含 25 行 50 个浮点数元素。同样，可以用以下的语句声明另一个二维浮点数数组：

```
float numbers[3][5];
```

与田里的蔬菜一样，使这些数组排成矩形会比较方便。把这个数组排成 3 行 5 列，它们实际上按行顺序存储在内存中，如图 5-5 所示。

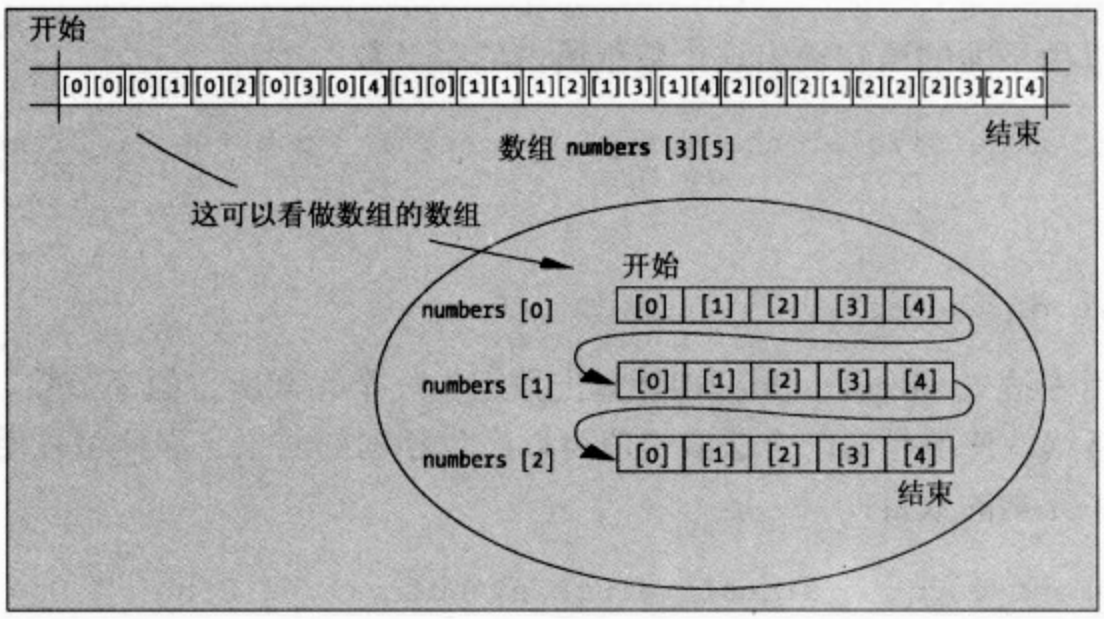


图 5-5　3 行 5 列元素数组在内存中的组织方式

很容易看出，最右边的索引变化得最快。图 5-5 也说明了如何将二维数组想象成一维数组，其中的每个元素本身是一个一维数组。可以将 number 数组视为 3 个元素的一维数组，数组中的每个元素都含有 5 个 float 类型的元素。第一行的 5 个 float 元素位于标记为 numbers[0]的内存地址上，第二行的 5 个 float 元素位于 numbers[1]，最后一行的 5 个元素位于 numbers[2]。

当然，分配给每个元素的内存量取决于数组所含的变量的类型。double 类型的数组需要的内存比 float 或 int 类型的数组多。图 5-6 说明了数组 numbers[4][10]的存储方式，该数组有 4 行 10 个 float 类型的元素。

因为数组元素的类型是 float，它在机器上占 4 个字节，这个数组占用的内存总数是 4×10×4 个字节，即 160 个字节。

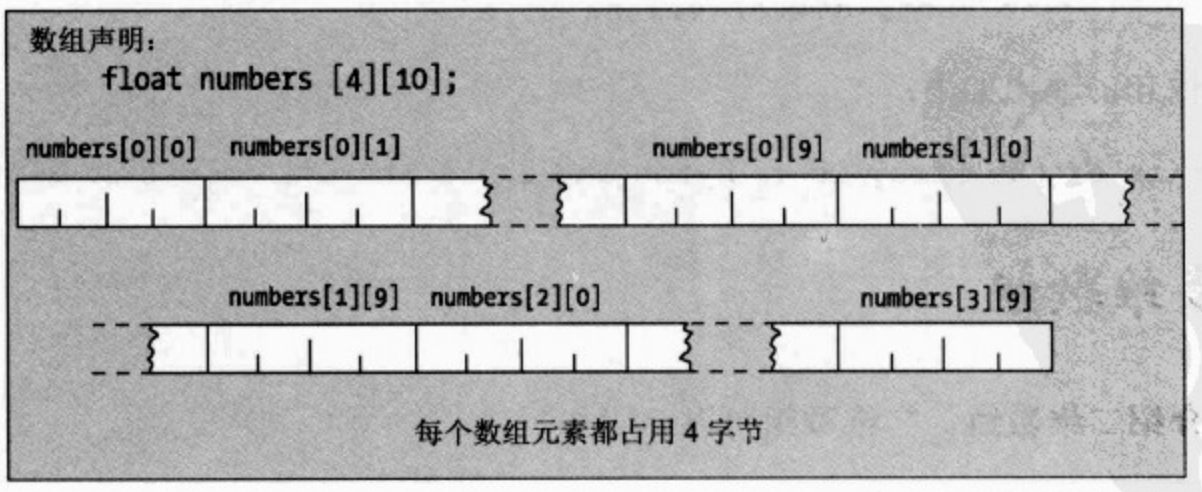


图 5-6　4×10 数组占用的内存

www.TopSage.com

5.7 多维数组的初始化

首先，考虑如何初始化二维数组。声明时初始化的基本结构和前面相同，但必须把每一行的初始值放在大括号{}中，例如：

```
int numbers[3][4] = {
                     { 10, 20, 30, 40 },  /* Values for first row */
                     { 15, 25, 35, 45 },  /* Values for second row */
                     { 47, 48, 49, 50 }   /* Values for third row */
                    };
```

初始化行中元素的每组值放在大括号中，所有的初始值则放在另一对大括号中。一行中的值以逗号分开，各行值也需以逗号分开。

如果指定的初值少于一行的元素数，这些值会从每行的第一个元素开始，依序赋予各元素，剩下未指定初值的元素则初始化为0。

对于三维或三维以上的数组，这个过程会被扩展。例如三维数组有3级嵌套的括号，内层的括号包含每行的初始值，例如：

```
int numbers[2][3][4] = {
                         {                          /* First block of 3 rows */
                              { 10, 20, 30, 40 },
                              { 15, 25, 35, 45 },
                              { 47, 48, 49, 50 }
                         },
                         {                          /* Second block of 3 rows */
                              { 10, 20, 30, 40 },
                              { 15, 25, 35, 45 },
                              { 47, 48, 49, 50 }
                         }
                       };
```

可以看到，初始化的值放在一个外层的大括号中，该外层括号由两个包含3行的块组成，每个块也放在括号中，各个块中的每一行也放在括号中，所以三维数组有3层嵌套括号。一般来说是这样的，例如六维数组用6层嵌套括号包含元素的初始值。可以省略每一行的括号，但给每一行的值加上括号比较安全，因为更不容易出错。当然，如果提供的初始值个数少于行中的元素数，就必须给每一行的值加上括号。

试试看：多维数组

下面介绍一个较实际的应用程序。在程序里可以用数组计算帽子的尺寸。对于这个程序，只需输入帽子的周长(英寸)，然后显示帽子的尺寸。

```
/* Program 5.6 Know your hat size - if you dare... */
#include <stdio.h>
#include <stdbool.h>
```

www.TopSage.com

```
int main(void)
{
  /* The size array stores hat sizes from 6 1/2 to 7 7/8 */
  /* Each row defines one character of a size value so */
  /* a size is selected by using the same index for each */
  /* the three rows. e.g. Index 2 selects 6 3/4. */
  char size[3][12] = { /* Hat sizes as characters */
    {'6', '6', '6', '6', '7', '7', '7', '7', '7', '7', '7', '7'},
    {'1', '5', '3', '7', ' ', '1', '1', '3', '1', '5', '3', '7'},
    {'2', '8', '4', '8', ' ', '8', '4', '8', '2', '8', '4', '8'}
    };

  int headsize[12] =       /* Values in 1/8 inches */
    {164,166,169,172,175,178,181,184,188,191,194,197};

  float cranium = 0.0;     /* Head circumference in decimal inches */
  int your_head = 0;       /* Headsize in whole eighths */
  int i = 0;               /* Loop counter */
  bool hat_found = false; /* Indicates when a hat is found to fit */

  /* Get the circumference of the head */
  printf("\nEnter the circumference of your head above your eyebrows "
     "in inches as a decimal value: ");
  scanf(" %f", &cranium);

  /* Convert to whole eighths of an inch */
  your_head = (int)(8.0*cranium);

  /* Search for a hat size */
  /* A fit is when your_head is greater that one headsize element */
  /* and less than or equal to the next. The size the the second */
  /* headsize value. */
  for (i = 1 ; i < 12 ; i++)
    /* Find head size in the headsize array */
    if(your_head > headsize[i-1] && your_head <= headsize[i])
    {
      hat_found = true;
      break;
    }

    if(your_head == headsize[0]) /* Check for min size fit */
    {
      i = 0;
      hat_found = true;
    }
    if(hat_found)
      printf("\nYour hat size is %c %c%c%c\n",
        size[0][i], size[1][i], (size[1][i]==' ') ? ' ' : '/',
```

www.TopSage.com

```
        size[2][i]);
    /* If no hat was found, the head is too small, or too large */
    else
    {
      if(your_head < headsize[0])    /* check for too small */
        printf("\nYou are the proverbial pinhead. No hat for"
            " you I'm afraid.\n");
      else                                /* It must be too large */
        printf("\nYou, in technical parlance, are a fathead."
            " No hat for you, I'm afraid.\n");
  }
  return 0;
}
```

这个程序的输出如下:

```
Enter the circumference of your head above your eyebrows in inches as a
decimal value: 22.5
Your hat size is 7 1/4
```

或:

```
Enter the circumference of your head above your eyebrows in inches as a
decimal value: 29
You, in technical parlance, are a fathead. No hat for you I'm afraid.
```

代码的说明

在开始讨论这个例子之前，注意不要用这个程序帮助足球运动员决定他们的帽子尺寸，除非他们很有幽默感。

这个例子有点复杂，因为它所解决的问题比较复杂。但它说明了数组的用法。

main()函数体的第一个声明如下:

```
char size[3][12] = {                    /* Hat sizes as characters */
    {'6', '6', '6', '6', '7', '7', '7', '7', '7', '7', '7', '7'},
    {'1', '5', '3', '7', ' ', '1', '1', '3', '1', '5', '3', '7'},
    {'2', '8', '4', '8', ' ', '8', '4', '8', '2', '8', '4', '8'}
    };
```

这个程序不是把帽子设计为单一尺寸，或设计为小、中、大，而是将帽子的尺寸设计为从6 1/2到7 7/8，其递增量为1/8。size数组是在程序中存储这些尺寸的一种方式。这个数组对应12个可能的帽子尺寸，每个尺寸由3个数值组成。对于每个帽子尺寸，要存储3个字符，以方便输出分数。最小的帽子尺寸是6 1/2，所以对应第一个尺寸的前3个字符是size[0][0]、size[1][0]、size[2][0]。它们含有字符6、1和2，代表尺寸6 1/2。最大的帽子尺寸是7 7/8，它存储在size[0][11]、size[1][11]和size[2] [11]中。

然后，声明数组headsize，在这个声明中，提供了头的参考尺寸:

```
int headsize[12] =          /* Values in 1/8 inches */
      {164,166,169,172,175,178,181,184,188,191,194,197};
```

www.TopSage.com

这个数组中的值都是 1/8 英寸的整数倍，它们对应含有帽子尺寸的 size 数组中的值。如果头的尺寸是 164 的 1/8 英寸(约 20.5 英寸)，帽子尺寸就是 6 1/2。而如果头的尺寸是 197 的 1/8 英寸，帽子尺寸就是 7 7/8。

注意，头的尺寸是不连续的。例如，如果头的尺寸是 171，就没有适合它的帽子尺寸。程序将在后面考虑这个问题，决定哪个帽子尺寸最接近这个头的大小。

声明数组之后，声明所有需要的变量:

```
float cranium = 0.0;     /* Head circumference in decimal inches */
int your_head = 0;       /* Headsize in whole eighths */
int i = 0;               /* Loop counter */
bool hat_found = false; /* Indicates when a hat is found to fit */
```

注意，cranium 声明成 float 类型，其他变量都声明为 int。这在后面很重要。变量 hat_found 声明为 bool 类型，所以用符号 false 初始化它。变量 hat_found 将记录找到的合适尺寸。

接着提示输入头的尺寸(英寸)，将这个值存储到变量 cranium 中(它的类型是 float，所以可以存储非整数值):

```
printf("\nEnter the circumference of your head above your eyebrows "
      "in inches as a decimal value: ");
scanf(" %f", &cranium);
```

接着使用下面的语句，将存储在 cranium 中的值转换成 1/8 英寸的整数倍:

```
your_head =(int)(8.0*cranium);
```

cranium 包含了头的周长(英寸)，所以乘以 8 就会得到其值的 1/8 英寸的整数倍。于是 your_head 的值和数组 headsize 的值有相同的单位。注意这里必须将它转换成 int 类型，以避免编译器发出警告信息。如果忽略这个类型转换，代码仍可以工作，但编译器必须插入该类型转换。但这个转换会潜在地丢失信息，所以编译器会发出警告。表达式(8.0*cranium)外的括号是必要的：没有括号，将只把 8.0 转换成 int 类型，而不是整个表达式。

使用存储在 your_head 中的值在数组 headsize 中查找不小于这个值且最接近它的值:

```
for (i = 1 ; i < 12 ; i++)
  /* Find head size in the headsize array */
  if(your_head > headsize[i-1] && your_head <= headsize[i])
  {
    hat_found = true;
    break;
  }
```

这个过程很简单，在 for 循环中完成。循环索引 i 从数组的第二个元素开始一直处理到最后一个元素，因为在 if 表达式中使用 i - 1 来索引数组。在每次循环迭代中，都比较头的尺寸和数组 headsize 中的一系列连续的值，找出第一个大于等于输入尺寸的元素值。

www.TopSage.com

找到的索引就对应于合适的帽子尺寸。

如果输入的尺寸正好对应于数组第一个元素的尺寸，则该尺寸就正合适，不需要在循环中继续查找了。因此在 if 语句中检查这个条件:

```
if(your_head == headsize[0]) /* Check for min size fit */
{
  i = 0;
  hat_found = true;
}
```

如果 your_head 的尺寸匹配数组 headsize 中的第一个元素，就找到了一个合适的帽子尺寸，所以把 i 设置为 0，hat_found 设置为 true。

如果 hat_found 的值是 true，就输出帽子尺寸:

```
if(hat_found)
  printf("\nYour hat size is %c %c%c%c\n",
    size[0][i], size[1][i], (size[1][i]==' ') ? ' ' : '/', size[2][i]);
```

前面提到，帽子尺寸在数组 size 中存储为字符，以简化分数的输出。这里 printf()利用条件运算符确定是打印一个空格，还是打印斜杠，以显示分数。数组 headsize 的第 5 个元素对应的帽子尺寸是 7，不需要输出 7 /，因此，应根据 size[1][i]元素是否包含''来定制 printf()。这样，当分数部分为空时，就省略斜杠。即使给数组添加新的尺寸，该程序也能正常工作。

当然，还有可能找不到合适的帽子尺寸，因为太大或太小的头都没有合适的帽子尺寸。用 if 语句的 else 子句处理这种情况，如果 hat_found 的值是 false，就执行 else 语句:

```
/* If no hat was found, the head is too small, or too large */
else
{
  if(your_head < headsize[0]) /* check for too small */
    printf("\nYou are the proverbial pinhead. No hat for"
       " you I'm afraid.\n");
  else                              /* It must be too large */
    printf("\nYou, in technical parlance, are a fathead."
       " No hat for you, I'm afraid.\n");
}
```

如果 your_head 的值小于第一个 headsize 元素，就表示头的尺寸太小，找不到合适的帽子，否则头的尺寸就太大。

注意，使用这个程序时，如果头的尺寸不正确，帽子将不合适。帽子尺寸只是圆形头的直径。因此，如果头的周长以英寸计，可以用这个数值除以π，产生帽子的尺寸。

5.8 设计一个程序

学习了数组后，将它们应用于一个比较大的问题上。下面编写另一个游戏。

www.TopSage.com

5.8.1 问题

编写一个程序，让两个人在计算机上玩井字游戏(也称为圈叉游戏)。

5.8.2 分析

井字游戏是一个 3×3 的方格。两个人轮流在方格中输入标记 X 或 O。谁先使自己的 3 个标记连接成水平、垂直或对角线，就是赢家。知道了这个游戏怎么玩，如何将它设计成程序？这需要：

- 一个 3×3 的方格，存储两个人交替输入的标记。这很简单，使用一个 3 行 3 列的二维数组即可。
- 轮到一个玩家输入标记时，需要一种方法标记选择出来的方格。可以用 1~9 的数字标记这 9 个方格。玩家只需输入要选择的方格数字。
- 有一种让两个玩家轮流输入标记的方法。可以将两个玩家识别为 1 和 2，编号 1 的玩家先玩。然后根据轮流的次数决定输入标记的玩家号码。轮到奇数号时，就由玩家 1 输入标记。轮到偶数号时，就由玩家 2 输入标记。
- 指定将玩家的标记放在哪个方格中，并检查它是否有效。一个有效的选择是 1~9 的数字。如果用 1、2、3 标记方格的第一行，用 4、5、6 标记第二行，用 7、8、9 标记第三行，就可以从方格数字中计算出列和行的索引。如果将玩家选择的方格数字减 1，方格数就是 0~8，如下图所示：

原来的方格

1	2	3
4	5	6
7	8	9

减 1 后的方格

0	1	2
3	4	5
6	7	8

表达式 choice/3 会得到行数，如下图所示：

原方格数减 1

0	1	2
3	4	5
6	7	8

除以 3

0	0	0
1	1	1
2	2	2

表达式 choice%3 会得到列数，如下图所示：

www.TopSage.com

原方格数减 1

0	1	2
3	4	5
6	7	8

除以 3 的系数

0	1	2
0	1	2
0	1	2

- 找出其中一位玩家获胜。每次轮完后，都需检查方格上的列、行或对角线是否有相同的标志。如果有，后一位玩家就赢了。
- 确定游戏的结束。因为板上有 9 个方格，所以游戏是在有人获胜或轮玩 9 次后结束。

5.8.3 解决方案

本节列出解决该问题的步骤。

1. 步骤 1

首先，添加主要的游戏循环和显示这个方格板的代码：

```
/* Program 5.7 Tic-Tac-Toe */
#include <stdio.h>

int main(void)
{
  int player = 0;        /* Player number - 1 or 2 */
  int winner = 0;        /* The winning player */

  char board[3][3] = {  /* The board */
      {'1','2','3'},    /* Initial values are reference numbers */
      {'4','5','6'},    /* used to select a vacant square for */
      {'7','8','9'}     /* a turn. */
      };

  /* The main game loop. The game continues for up to 9 turns */
  /* As long as there is no winner */
  for(int i = 0; i<9 && winner==0; i++)
  {
    /* Display the board */
    printf("\n\n");
    printf(" %c | %c | %c\n", board[0][0], board[0][1], board[0][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[1][0], board[1][1], board[1][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[2][0], board[2][1], board[2][2]);
```

www.TopSage.com

```
    player = i%2 + 1; /* Select player */

    /* Code to play the game */
  }
  /* Code to output the result */
  return 0;
}
```

这里声明了以下变量：i 是循环变量；player，存储目前玩家的识别码 1 或 2；winner，含有获胜者的识别码：数组 board，它的类型是 char。因为这个数组把标记 X 或 O 放在方格中。这个数组用方格的识别数字作为初始值。游戏的主循环只要循环条件为 true，就会继续执行。如果 winner 的值不等于 0(表示找到获胜者)，或循环计数器的值大于等于 9(表示方格板上的 9 格全部填满)，循环条件就是 false。

在循环中显示方格板时，使用“|”和“_”字符绘制方框。玩家选择了一个方格时，玩家的标志将会取代这个字符。

2. 步骤 2

接下来，编写代码，让玩家选择一个方格，并确定那个方格是否有效：

```
/* Program 5.7 Tic-Tac-Toe */
#include <stdio.h>

int main(void)
{
  int player = 0;        /* Player number - 1 or 2 */
  int winner = 0;        /* The winning player */
  int choice = 0;        /* Square selection number for turn */
  int row = 0;           /* Row index for a square */
  int column = 0;        /* Column index for a square */

  char board[3][3] = {  /* The board */
      {'1','2','3'},    /* Initial values are reference numbers */
      {'4','5','6'},    /* used to select a vacant square for */
      {'7','8','9'}     /* a turn. */
      };

  /* The main game loop. The game continues for up to 9 turns */
  /* As long as there is no winner */
  for(int i = 0; i<9 && winner==0; i++)
  {
    /* Display the board */
    printf("\n\n");
    printf(" %c | %c | %c\n", board[0][0], board[0][1], board[0][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[1][0], board[1][1], board[1][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[2][0], board[2][1], board[2][2]);
```

www.TopSage.com

```
    player = i%2 + 1; /* Select player */

    /* Get valid player square selection */
    do
    {
      printf("\nPlayer %d, please enter the number of the square "
          "where you want to place your %c: ",
          player,(player==1)?'X':'O');
      scanf("%d", &choice);

      row = --choice/3;           /* Get row index of square */
      column = choice%3;          /* Get column index of square */
    }while(choice<0 || choice>9 || board[row][column]>'9');

    /* Insert player symbol */
    board[row][column] = (player == 1) ? 'X' : 'O';

    /* Code to check for a winner */
  }
  /* Code to output the result */
  return 0;
}
```

在 do-while 循环中提示当前的玩家输入标记，并且把方格数字读入类型声明为 int 的 choice 变量。这个值将用于计算数组中列和行的索引值。行和列的索引值分别保存在变量 row 和 column 中，然后用前面的表达式计算这些值。do-while 的循环条件确认选择的方格是有效的。有 3 种可能导致选择无效：输入的方格数小于 1 或大于 9，或选择已包含 X 或 O 的方格。在后一种情况下，方格的内容将大于字符'9'，因为 X 和 O 的字符码都大于 9 的字符码。如果输入的 choice 属于以上任一种情况，都需要求玩家再选择一个方格。

3. 步骤 3

添加检查获胜线的代码，这必须在每次轮完后执行：

```
/* Program 5.7 Tic-Tac-Toe */
#include <stdio.h>

int main(void)
{
  int player = 0;          /* Player number - 1 or 2 */
  int winner = 0;          /* The winning player */
  int choice = 0;          /* Square selection number for turn */
  int row = 0;             /* Row index for a square */
  int column = 0;          /* Column index for a square */
  int line=0;              /* Row or column index in checking loop */
```

www.TopSage.com

```
char board[3][3] = {  /* The board */
    {'1','2','3'}, /* Initial values are reference numbers */
    {'4','5','6'}, /* used to select a vacant square for */
    {'7','8','9'} /* a turn. */
    };

/* The main game loop. The game continues for up to 9 turns */
/* As long as there is no winner */
for(int i = 0; i<9 && winner==0; i++)
  {
  /* Display the board */
  printf("\n\n");
  printf(" %c | %c | %c\n", board[0][0], board[0][1], board[0][2]);
  printf("---+---+---\n");
  printf(" %c | %c | %c\n", board[1][0], board[1][1], board[1][2]);
  printf("---+---+---\n");
  printf(" %c | %c | %c\n", board[2][0], board[2][1], board[2][2]);

  player = i%2 + 1;        /* Select player */

  /* Get valid player square selection */
  do
  {
    printf("\nPlayer %d, please enter the number of the square "
        "where you want to place your %c: ",
        player,(player==1)?'X':'O');
    scanf("%d", &choice);

    row = --choice/3;      /* Get row index of square */
    column = choice%3;     /* Get column index of square */
  }while(choice<0 || choice>9 || board[row][column]>'9');

  /* Insert player symbol */
  board[row][column] = (player == 1) ? 'X' : 'O';

  /* Check for a winning line - diagonals first */
  if((board[0][0]==board[1][1] && board[0][0]==board[2][2]) ||
       (board[0][2]==board[1][1] && board[0][2]==board[2][0]))
    winner = player;
  else
    /* Check rows and columns for a winning line */
    for(line = 0; line <= 2; line ++)
      if((board[line][0]==board[line][1] &&
          board[line][0]==board[line][2])||
          (board[0][line]==board[1][line] &&
          board[0][line]==board[2][line]))
        winner = player;
}
/* Code to output the result */
```

www.TopSage.com

```
  return 0;
}
```

检查获胜线时，可以用线上的一个元素比较线上的其他两个元素，确定它们是否相同。如果 3 个都相同，就有一个获胜线。用 if 表达式检查 board 数组中的两个对角线，如果任一条对角线中的 3 个标志完全相同，就把 winner 设定成当前的玩家。当前的玩家用 player 识别，他一定是赢家，因为他是最后一个在方格中放置标志的。如果两个对角线都没有相同的标志，就在 else 子句中用一个 for 循环去检查列和行。这个 for 循环含有一个 if 语句，它检查列和行是否有相同的元素。如果有，就把 winner 设定成当前的玩家。循环变量 line 的每个值都用于索引行和列。因此 for 循环会用索引值 0 检查列和行，即第一列和第一行，然后用索引值 1 检查第二列和第二行，最后用索引值 2 检查第三列和第三行。当然，如果 winner 设定为一个值，主循环的条件就是 false，所以结束循环，继续执行主循环后的代码。

4. 步骤 4

最后的任务是显示格子上最后各个标记的位置，显示比赛结果。如果 winner 是 0，这局就是平手；否则 winner 含有获胜者的号码。

```
/* Program 5.7 Tic-Tac-Toe */
#include <stdio.h>

int main(void)
{
  int player = 0;        /* Player number - 1 or 2 */
  int winner = 0;        /* The winning player */
  int choice = 0;        /* Square selection number for turn */
  int row = 0;           /* Row index for a square */
  int column = 0;        /* Column index for a square */
  int line=0;            /* Row or column index in checking loop */

  char board[3][3] = {  /* The board */
      {'1','2','3'},    /* Initial values are reference numbers */
      {'4','5','6'},    /* used to select a vacant square for */
      {'7','8','9'}     /* a turn. */
      };

  /* The main game loop. The game continues for up to 9 turns */
  /* As long as there is no winner */
  for(int i = 0; i<9 && winner==0; i++)
  {
    /* Display the board */
    printf("\n\n");
    printf(" %c | %c | %c\n", board[0][0], board[0][1], board[0][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[1][0], board[1][1], board[1][2]);
    printf("---+---+---\n");
```

www.TopSage.com

```
    printf(" %c | %c | %c\n", board[2][0], board[2][1], board[2][2]);

    player = i%2 + 1;     /* Select player */

    /* Get valid player square selection */
    do
    {
      printf("\nPlayer %d, please enter the number of the square "
          "where you want to place your %c: ",
          player,(player==1)?'X':'O');
      scanf("%d", &choice);

      row = --choice/3;   /* Get row index of square */
      column = choice%3;  /* Get column index of square */
    }while(choice<0 || choice>9 || board[row][column]>'9');

    /* Insert player symbol */
    board[row][column] = (player == 1) ? 'X' : 'O';

    /* Check for a winning line - diagonals first */
      if((board[0][0]==board[1][1] && board[0][0]==board[2][2]) ||
        (board[0][2]==board[1][1] && board[0][2]==board[2][0]))
        winner = player;
      else
        /* Check rows and columns for a winning line */
        for(line = 0; line <= 2; line ++)
          if((board[line][0]==board[line][1] &&
              board[line][0]==board[line][2])||
              (board[0][line]==board[1][line] &&
              board[0][line]==board[2][line]))
            winner = player;
    }
    /* Game is over so display the final board */
    printf("\n\n");
    printf(" %c | %c | %c\n", board[0][0], board[0][1], board[0][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[1][0], board[1][1], board[1][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[2][0], board[2][1], board[2][2]);

    /* Display result message */
    if(winner == 0)
      printf("\nHow boring, it is a draw\n");
    else
      printf("\nCongratulations, player %d, YOU ARE THE WINNER!\n",
          winner);
    return 0;
}
```

www.TopSage.com

这个程序的输出如下：

```
1  | 2 | 3
---+---+---
4  | 5 | 6
---+---+---
7  | 8 | 9

Player 1, please enter your go: 1

X  | 2 | 3
---+---+---
4  | 5 | 6
---+---+---
7  | 8 | 9

Player 2, please enter your go: 2

X  | O | 3
---+---+---
4  | 5 | 6
---+---+---
7  | 8 | 9

Player 1, please enter your go: 5

X  | O | 3
---+---+---
4  | X | 6
---+---+---
7  | 8 | 9

Player 2, please enter your go: 3

X  | O | 0
---+---+---
4  | X | 6
---+---+---
7  | 8 | 9

Player 1, please enter your go: 9

X  | O | 0
---+---+---
4  | X | 6
---+---+---
7  | 8 | X

Congratulations, player 1, YOU ARE THE WINNER!
```

5.9 小结

本章详细探讨了数组。数组是一组数目固定、类型相同的元素，使用数组名和一个或多个索引值，就可以访问数组中的任意元素。数组的索引值是从 0 开始的整数值，每一维数组都有一个索引。

将数组和循环合并使用，提供了一种非常强大的编程技术。使用数组可以在循环中处理类型相同的大量数据值，无论有多少数据值，操作所需的代码量都是相同的。还可以用多维数组组织数据。建立这样的数组，每一维数组都用某个特性来选择一组元素，例如与某个时间或地点相关的数据。给多维数据应用嵌套的括号，可以用非常少的代码处理所有的数组元素。

前面的章节主要介绍了数字的处理，但没有处理过文本。下一章将编写可以处理和分析字符串的程序。

5.10 习题

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容。还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 5.1 编写一个程序，从键盘上读入 5 个 double 类型的值，将它们存储到一个数组中。计算每个值的倒数(值 x 的倒数是 1.0/x)，将结果存储到另一个数组中。输出这些倒数，并计算和输出倒数的总和。

习题 5.2 定义一个数组 data，它包含 100 个 double 类型的元素。编写一个循环，将以下的数值顺序存储到数组的对应元素中：

```
1/(2*3*4) 1/(4*5*6) 1/(6*7*8) … up to 1/(200*201*202)
```

编写另一个循环，计算：

```
data[0]-data[1]+data[2]-data[3]+… -data[99]
```

将这个结果乘以 4.0，加 3.0，输出最后的结果。

习题 5.3 编写一个程序，从键盘上读入 5 个值，将它们存储到一个 float 类型的数组 amounts 中。创建两个包含 5 个 long 元素的数组 dollars 和 cents。将 amounts 数组元素的整数部分存储到 dollars 的对应元素中，amounts 数组元素的小数部分存储到 cents 中，只保存两位数字(例如：amounts[1]的值是 2.75，则把 2 存储到 dollars[1]中，把 75 存储 cents[1]中)。以货币格式输出这两个 long 类型数组的值(如$2.75)。

习题 5.4 定义一个 double 类型的二维数组 data[12][5]。用 2.0~3.0 的值初始化第一列元素(每步增加 0.1)。如果行中的第一个元素值是 x，该行的其他元素值分别是 1/x，x^2、x^3 和 x^4。输出数组中的值，每一行放在一行上，每一列要有标题。

www.TopSage.com

第 6 章

字符串和文本的应用

上一章介绍数组时，说明了如何使用数值数组方便地完成许多编程工作。本章将探讨如何使用字符数组，以扩展数组知识。我们经常需要将文本字符串用作一个实体，不过 C 语言没有提供字符串数据类型，这与其他编程语言不同。C 语言使用 char 类型的数组元素存储字符串。

本章将介绍如何创建和处理字符串变量，标准库函数如何简化字符串的处理。

本章的主要内容：

- 如何创建字符串变量
- 如何连接两个或多个字符串，形成一个字符串
- 如何比较字符串
- 如何使用字符串数组
- 如何使用宽字符串
- 哪些库函数能处理字符串，如何应用它们
- 编写一个简单的密码保护程序

6.1 什么是字符串

字符串常量的例子非常常见。字符串常量是放在一对双引号中的一串字符或符号。一对双引号之间的任何内容都会被编译器视为字符串，包括特殊字符和嵌入的空格。每次使用 printf()显示信息时，就将该信息定义成字符串常量了。以下的语句是用这种方法使用字符串的例子：

```
printf("This is a string");
printf("This is on\ntwo lines");
printf("\"r \"you write \"");
```

这 3 个字符串例子如图 6-1 所示。存储在内存中的字符码的十进制值显示在这些字符的下方。

www.TopSage.com

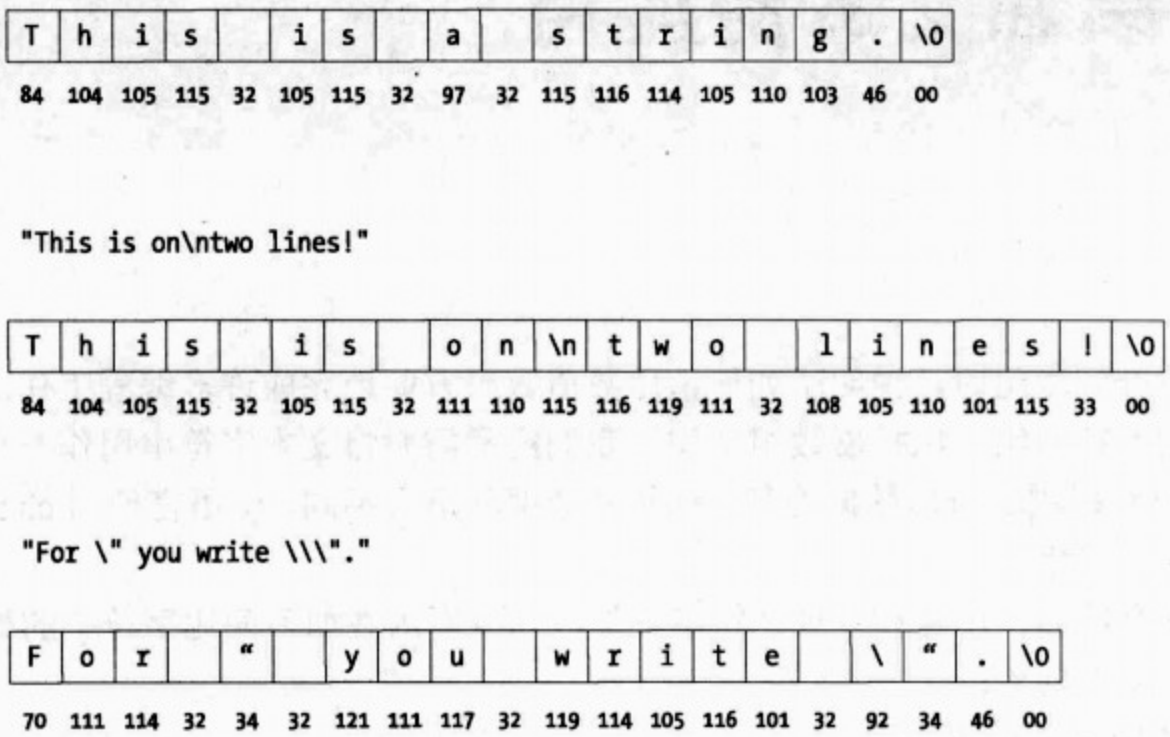


图 6-1 内存中的字符串例子

如图 6-1 所示，第一个字符串是一系列字符后跟一个句号。printf()函数会把这个字符串输出为：

```
This is a string.
```

第二个字符串有一个换行符\n，所以字符串显示在两行上：

```
This is on
two lines!
```

第三个字符串有点难以理解，但 printf()函数的输出很清楚：

```
For " you write \".
```

必须把字符串中的双引号写为转义序列\"，因为编译器会把双引号看作字符串的结尾。要在字符串中包含反斜杠，也必须使用转义序列\\，因为字符串中的反斜杠总是表示转义序列的开头。

如图 6-1 所示，每个字符串的末尾都添加了代码值为 0 的特殊字符，这个字符称为空字符(不要和 NULL 搞混)，写做\0。

注意：

C 中的字符串总是由\0 字符结束，所以字符串的长度永远比字符串中的字符数多 1。

可以自己将\0 字符添加到字符串的结尾，但是这会使字符串的末尾有两个\0 字符。下面的程序说明了空字符\0 是如何运作的：

```
/* Program 6.1 Displaying a string */
```

www.TopSage.com

```
#include <stdio.h>

int main(void)
{
  printf("The character \0 is used to terminate a string.");
  return 0;
}
```

编译并执行这个程序，会得到如下输出：

```
The  character
```

这可不是我们期望的结果：仅显示了字符串的第一部分。这个程序显示了前两个字符后就结束输出，是因为 printf()函数遇到第一个空字符\0 时，就会停止输出。即使在字符串非末尾还有另一个\0，也永远不会执行它。在遇到第一个\0 时，就表示字符串结束了。

6.2 处理字符串和文本的方法

与其他编程语言不同，C 语言对变量存储字符串的语法没有特殊的规定，而且 C 根本就没有字符串变量，也没有处理字符串的特殊运算符。但这不成问题，因为我们可以使用前面介绍的工具处理字符串。

如本章开头所述，可以使用 char 类型的数组保存字符串。这是字符串变量的最简单的形式。char 数组变量的声明如下：

```
char  saying[20];
```

在这条语句中声明的 saying 变量可以存储一个至多包含 19 个字符的字符串，因为必须给终止字符提供一个数组元素。当然也可以使用这个数组存储 20 个字符，那就不是一个字符串了。

警告：

声明存储字符串的数组时，其大小至少要比所存储的字符数多 1，因为编译器会自动在字符串常量的末尾添加\0。

也可以用以下的声明初始化前面的字符串变量：

```
char  saying[]  =  "This  is  a  string." ;
```

这里没有明确定义这个数组的大小。编译器会指定一个足以容纳这个初始化字符串常量的数值。在这个例子中它是 18，其中 17 个元素用于存储字符串中的字符，再加上一个额外的终止字符\0。当然可以指定这个数值，但是如果让编译器指定，可以确保它一定正确。

也可以用一个字符串初始化 char 类型数组的部分元素，例如：

www.TopSage.com

```
char str[40] = "TO be";
```

这里编译器会使用指定字符串的字符初始化从 str[0]到 str[4]的前 5 个元素，而 str[5]含有空字符'\0'。当然，数组的所有 40 个元素都会被分配空间，可以以任意方式使用。

初始化一个 char 数组，将它声明为常量，是处理标准信息的好方法：

```
const char message[] = "The end of the world is nigh";
```

将 message 声明成常量，它就不会在程序中被显式更改。只要试图更改它，编译器都会产生错误信息。当标准信息在程序中的许多地方使用时，这种定义标准信息的方法特别有用。它可以防止在程序的其他部分意外地修改这种常量。当然，假使必须改变这条信息，就不应将它指定为 const。

要引用存储在数组中的字符串时，只需使用数组名即可。例如，如果要用 printf()函数输出存储在 message 中的字符串，可以编写：

```
printf("\nThe message is: %s", message);
```

这个%s 指定符用于输出一个用空字符终止的字符串。函数 printf()会在第一个参数的%s 位置，输出 message 数组中连续的字符，直到遇到\0 字符为止。当然，char 数组的执行方式与其他类型的数组一样，所以可以用相同的方式使用它。字符串处理函数唯一需要特别考虑的是'\0'字符，所以从外表看来，包含字符串的数组没有什么特别的。

使用 char 数组存储各种不同的字符串的主要缺点是浪费内存。因为根据定义，数组的长度是固定的，必须用足以容纳要存储的最大字符串长度来声明数组的大小。在大多数情况下，一般的字符串都会小于这个最大值，所以总是会浪费内存。我们一般使用数组存储不同长度的字符串，所以确定字符串的长度是很重要的，特别是要给字符串添加更多的字符。下面用一个例子来说明：

试试看：确定字符串的长度

这个例子将初始化两个字符串，然后确定每个字符串有多少个字符，不包含空字符：

```
/* Program 6.2 Lengths of strings */
#include <stdio.h>

int main(void)
{
  char str1[] = "To be or not to be";
  char str2[] = ",that is the question";
  int count = 0;                /* Stores the string length */
  while (str1[count] != '\0')/* Increment count till we reach the string */
    count++;                    /* terminating character. */
  printf("\nThe length of the string \"%s\" is %d characters.", str1,
          count);
  count = 0;                    /* Reset to zero for next string */
  while (str2[count] != '\0') /* Count characters in second string */
    count++;
  printf("\nThe length of the string \"%s\" is %d characters.\n",
```

www.TopSage.com

```
        str2, count);
  return 0;
}
```

这个程序的输出是:

```
The length of the string "To be or not to be" is 18 characters.
The length of the string ",that is the question" is 21 characters.
```

代码的说明

首先声明一些变量:

```
char str1[] = "To be or not to be";
char str2[] = ",that is the question";
int count = 0;          /* Stores the string length */
```

声明两个 char 类型的数组，每个数组都初始化了一个字符串。编译器把每个数组的大小都设置为能容纳其字符串和终止字符。接着声明并初始化一个计数器 count，在程序的循环中使用。当然可以不指定每个数组的大小，让编译器计算出需要的空间，如前所述。

接下来，用 while 循环确定第一个字符串的长度:

```
while (str1[count] != '\0') /* Increment count till we reach the string */
  count++;                  /* terminating character. */
```

以这种方式使用循环，在使用字符串的程序中是很常见的。为了确定长度，只需在 while 循环中不断地递增计数器，直到到达字符串尾为止。这个循环的继续条件检查是否到达终止字符。循环结束时，变量 count 就包含了字符串的字符个数，但不包含终止字符。

while 循环比较 str1[count]元素的值和'\0'，所以确定字符串末尾的机制非常清晰。这个循环一般写作:

```
while (str1[count])
  count++;
```

'\0'字符的 ASCII 码是 0，对应于布尔值 false。其他 ASCII 码都不是 0，对应布尔值 true。因此，只要 str1[count]不是'\0'，循环就继续执行。

确定了长度后，用下面的语句显示字符串:

```
printf("\nThe length of the string \"%s\" is %d characters.", str1, count);
```

这也显示了字符串包含的字符数，但不包含终止字符。注意使用新的格式指定符%s，%s 在前面已介绍过了。这会输出字符串中的字符，直到遇到终止字符为止。如果没有终止字符，它将一直输出字符，直到在内存某处找到一个终止字符为止。在某些情况下，输出会非常多。上面的语句也使用转义序列\"在字符串中包含一个双引号。如果没有在双引号前加上反斜杠，编译器会把它当做 printf()函数的第一个参数中的字符

www.TopSage.com

串结尾，因此该语句会产生一条错误信息。

确定第二个字符串的长度和显示其结果的方式，与第一个字符串完全相同。

6.3 字符串操作

上例中的代码说明了确定字符串长度的机制，但还没有编写过代码。如后面所述，标准库中的 strlen()函数可以确定字符串的长度。知道了如何确定字符串的长度后，该如何使用它?

可惜，不能使用赋值运算符以处理 int 或 double 变量的方式来复制字符串。要对字符串执行算术赋值操作，必须逐个元素地把一个字符串复制到另一个字符串中。事实上，字符串变量的任何操作都不同于前面介绍的数值变量。下面介绍字符串的一些常见操作。

6.3.1 连接字符串

把一个字符串连接到另一个字符串的尾部是很常见的需求。例如，把两个或多个字符串合成为一条信息。在程序中，将错误信息定义为几个基本的文本字符串，然后给它们添加另一个字符串，使之变成针对某个错误的信息。下面用一个例子来说明其工作原理。

试试看：连接字符串

重做上一个例子，将第二个字符串连接到第一个字符串上:

```
/* Program 6.3 Joining strings */
#include <stdio.h>

int main(void)
{
  char str1[40] = "To be or not to be";
  char str2[] = ",that is the question";
  int count1 = 0;          /* Length of str1 */
  int count2 = 0;          /* Length of str2 */

  /* find the length of the first string */
  while (str1[count1] )  /* Increment count till we reach the string */
    count1++;            /* terminating character. */

  /* Find the length of the second string */
  while (str2[count2])   /* Count characters in second string */
    count2++;

  /* Check that we have enough space for both strings */
  if(sizeof str1 < count1 + count2 + 1)
    printf("\nYou can't put a quart into a pint pot.");
  else
```

www.TopSage.com

```
    { /* Copy 2nd string to end of the first */
    count2 = 0; /* Reset index for str2 to 0 */
    while(str2[count2]) /* Copy up to null from str2 */
      str1[count1++] = str2[count2++];
    str1[count1] = '\0'; /* Make sure we add terminator */
    printf("\n%s\n", str1 ); /* Output combined string */
  }
  return 0;
}
```

这个程序的输出如下所示:

```
To be or not tO be, that is the question
```

代码的说明

这个程序首先确定两个字符串的长度，然后检查 str1 是否有足够的空间容纳两个字符串和终止字符:

```
if(sizeof str1 < count1 + count2 + 1)
  printf("\nYou can't put a quart into a pint pot.");
```

注意，这里以数组名为参数，使用 sizeof 运算符确定数组的总字节数。表达式 sizeof str1 的结果是这个数组保存的字符数，因为每个字符都占据一个字节。

如果这个数组太小，不能容纳两个字符串，就显示一条信息，然后结束这个程序。这是很重要的，因为不能在数组中放置数组不能容纳的过多字符，这会覆盖某些可能含有重要数据的内存，使程序崩溃。在没有先检查数组是否有足够空间容纳新字符之前，是不能给字符串添加字符的。

只有两个字符串在数组中都能放得下，才会执行到 else 块。else 块中的下述语句把变量 count2 重置为 0，将第二个字符串复制到第一个数组中:

```
else
  { /* Copy 2nd string to end of the first */
  count2 = 0;                /* Reset index for str2 to 0 */
  while(str2[count2])        /* Copy up to null from str2 */
    str1[count1++] = str2[count2++];
  str1[count1] = '\0';       /* Make sure we add terminator */
  printf("\n%s\n", str1 ); /* Output combined string */
}
```

确定第一个字符串 str1 长度的循环执行完后，变量 countl 就包含了该长度值。这就是使用两个变量计算两个字符串的字符数的原因。数组的索引从 0 开始，所以 count1 的值指向第一个字符串末尾包含'\0'的元素。因此，使用 countl 索引数组 str1 时，就从这条信息的尾部开始，用第二个字符串的第一个字符覆盖掉空字符。

然后，将字符从 str2 复制到 str1 中，直到遇到 str2 中的'\0'字符为止。还需要给 str1 添加一个终止字符'\0'，因为它没有从 str2 复制过来。这个操作结束后，str2 的内容就添加到 strl 的尾部，覆盖了 str1 的终止字符，并在组合字符串的尾部添加了一个终止字符。

www.TopSage.com

可以用一个比较简洁的方法替换 3 行复制语句:

```
while  ((str1[count1++] =  str2[count2++]));
```

这取代了程序中的循环以及在 str1 的末尾添加'\0'字符的语句。这条语句会把 str2 中的'\0'复制到 str1 中，因为复制操作在循环的继续条件中执行。下面看看各个步骤:

(1) 将 str2[count2]的值赋予 str1[count1]。赋值表达式的值存储在赋值运算符的左操作数中。在这个例子中，它是复制到 str1[count1]中的字符。

(2) 使用++运算符的后置形式，给每个计数器加 1。

(3) 检查赋值表达式的值——存储在 str1 中的最后一个字符——是 true 还是 false。字符'\0'复制到 str1 中后，赋值表达式的值就是 false，所以结束循环。

6.3.2 字符串数组

可以使用 char 类型的二维数组存储字符串，数组的每一行都用来存储一个字符串。这样，就可以存储一整串字符串，通过一个变量名来引用它们，例如:

```
char  sayings[3][32]  =  {
                         "Manners  maketh  man."
                         "Many  hands  make  light  work."
                         "Too  many  cooks  spoil  the  broth. "
                         };
```

这条语句创建了一个数组，它包含 3 行，每行 32 个字符。括号中的字符串按顺序指定数组的 3 行 sayings[0]、sayings[1]和 sayings[3]。注意，不需要用括号将每个字符串括起来。编译器能推断出每个字符串初始化数组的一行。最后一维指定为 32，刚好能容纳最长的字符串(包含\0 终止字符)。第一维指定字符串的数目。

当在引用数组的元素时，例如 sayings[i][j]，第一个索引 i 指定数组中的行，第二个索引 j 指定该行中的一个字符。要引用数组中包含一个字符串的一整行，只需在方括号中包含一个索引值。例如 sayings[1]引用数组的第二个字符串，"Many hands make light work."。

在字符串数组中，必须指定最后一维的大小，也可以让编译器计算数组有多少个字符串:

```
char  sayings[][32]  =  {
                        "Manners  maketh  man."
                        "Many  hands  make  light  work."
                        "Too  many  cooks  spoil  the  broth. "
                        };
```

这条语句省略了数组第一维的大小，编译器要从括号内的初始字符串中推算出它的大小。因为有 3 个初始字符串，编译器会将数组的第一维大小指定为 3。当然，还必须确保最后一维的空间足以容纳最长的字符串，包含终止字符。

可以用下列代码输出 3 句格言:

www.TopSage.com

```
for(i = 0; i < 3;  i++)
  printf("\n%s", sayings[i]);
```

表达式 sayings[i]使用一个索引来引用数组中的一行，这等价于访问 sayings 数组中索引位置为 i 的一维数组。

将上面的例子改为使用一个二维数组。

试试看：字符串数组

更改前面的例子，在一个数组中存储两个初始的字符串，并用比较简洁的代码确定字符串的长度，并复制字符串：

```
/* Program 6.4 Arrays of strings */
#include <stdio.h>

int main(void)
{
  char str[][40] = {
                    "To be or not to be" ,
                    ", that is the question"
                   };
  int count[] = {0, 0};       /* Lengths of strings */

  /* find the lengths of the strings */
  for(int i = 0 ; i<2 ; i++)
    while (str[i][count[i]])
      count[i]++;

  /* Check that we have enough space for both strings */
  if(sizeof str[0] < count[0] + count[1] + 1)
    printf("\nYou can't put a quart into a pint pot.");
  else
    {                              /* Copy 2nd string to first */
    count[1] = 0;
    while((str[0][count[0]++] = str[1][count[1]++]));

    printf("\n%s\n", str[0]);  /* Output combined string */
  }
  return 0;
}
```

这个程序的输出如下：

```
To be or not to be, that is the question
```

代码的说明

这个例子声明一个二维 char 数组，而不是声明两个一维数组：

```
char str[][40] = {
                  "To be or not to be" ,
```

www.TopSage.com

```
", that is the question"
};
```

第一个初始字符串用 str[0]来存储，第二个初始字符串用 str[1]来存储。当然，可以在括号中添加任意多个初始字符串，编译器会调整数组第一维的大小，以容纳它们。

字符串长度现在存储为 count 数组的元素。由于 count 是一个数组，所以可以在一个循环中确定两个字符串的长度:

```
for(int i = 0 ; i<2 ; i++)
  while (str[i][count[i]])
    count[i]++;
```

外层的 for 循环迭代两个字符串，内层的 while 循环迭代当前字符串中的字符。这个方法显然可应用于 str 数组中有任意多个字符串的情况，自然，count 数组中的元素个数必须与字符串的个数相同。该方法的缺点是，如果字符串远远少于 40 个字符，就会浪费内存。下一章将学习如何避免浪费空间，以最高效的方式存储每个字符串。

6.4 字符串库函数

前面的例子很费力地把字符串从一个变量复制到另一个变量中，下面看看字符串标准函数库如何执行这个操作。我们只需了解使用这些库函数时，会发生什么。

字符串函数在<string.h>头文件中声明，所以如果要使用它们，需要在程序的开始处添加如下代码:

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

事实上，该库含有相当多的函数，编译器提供的字符串处理能力比 C 标准需要的还要高。这里只讨论几个基本函数，以说明基本概念，其余函数留给读者自己去发掘。

6.4.1 使用库函数复制字符串

首先，看看如何把字符串从一个数组复制到另一个数组中，它使用了字符串的赋值操作。前面使用了一个精心创建的 while 循环来完成这个工作。还可以使用如下语句:

```
strcpy(string1, string2);
```

函数 strcpy()的参数是 char 数组名，这个函数的作用是将第二个参数指定的字符串复制到第一个参数指定的字符串中，所以就这个例子而言，string2 会复制到 string1 中，取代先前存储在 string1 中的字符串。这个复制操作包含终止字符'\0'。但必须确保数组 string1 的空间要足以容纳 string2。函数 strcpy()不检查数组的大小，所以如果发生错误，一定是因为我们没有检查数组的大小。很明显，sizeof 运算符在这里很重要，因为可以用以下方式去检查:

```
if(sizeof(string2) <=  sizeof(string1))
```

www.TopSage.com

```
strcpy(stringl, string2);
```

只有 string2 数组的长度小于等于 stringl 数组的长度，才执行 strcpy()函数。

另一个可用的函数 strncpy()将一个字符串中的前 n 个字符复制到另一个字符串中。第一个参数是目标字符串，第二个参数是源字符串，第三个参数是一个 size_t 类型的整数，它指定了要复制的字符数。下面的例子说明了该函数的用法：

```
char destination[] = "This string will be replaced";
char source[] = "This string will be copied in part";
size_t n = 26;      /* Number of characters to be copied */
strncpy(destination, source, n);
```

在执行这些语句后，destination 会包含字符串"This string will be copied"，因为它们对应于 source 中的前 26 个字符。终止字符'\0'会附加到最后一个复制的字符后面。如果 source 少于 26 个字符，该函数会添加'\0'字符，补足 26 个字符。

注意，源字符串的长度大于要复制的字符数时，strncpy()函数就不会在目标字符串中添加终止字符'\0'。因此，目标字符串没有终止字符'\0'，以后使用目标字符串进行其他操作时，会带来很大的问题。

6.4.2 使用库函数确定字符串的长度

要确定字符串的长度，可以使用 strlen()函数，它会把字符串的长度返回为一个 size_t 类型的整数。要确定程序 6.3 中字符串的长度，可以编写：

```
while (str2[count2])
  count2++;
```

代码有点长，可以简化如下：

```
count2 = strlen(str2);
```

现在，确定字符串的结尾所需的所有计算和搜索操作都由函数来完成，我们不用再操心了。注意，它返回的长度不包含'\0'字符，这一般是最方便的结果。另外，它返回的 size_t 值对应于无符号的整数类型，所以还要声明一个变量，把结果存储为 size_t。否则，编译器就会发出警告信息。

类型 size_t 在标准库头文件<stddef.h>中定义，它也是运算符 sizeof 的返回类型。类型 size_t 定义为一个无符号的整数类型 unsigned int。之所以要这么做，是考虑到代码的可移植性。在各种 C 语言中，sizeof 和 strlen()函数的返回类型互不相同，这由编译器的作者来决定。把这个类型定义为 size_t，并把 size_t 定义放在头文件中，非常便于在代码中包容这种依赖性。上面的例子只要把 count2 定义为类型 size_t，即使 size_t 的定义在各种 C 语言中互不相同，该代码也可以在所有的标准 C 语言中正常运作。

为了创建可移植性高的代码，应该以如下方式编写：

```
size_t count2  =  0;
count2  =  strlen(str2);
```

www.TopSage.com

只要给<string.h>和<stddef.h>添加了#include 指令，代码就会用 ISO/IEC 标准 C 编译器来编译。

6.4.3 使用库函数连接字符串

程序 6.3 使用如下相当复杂的代码，将第二个字符串复制到第一个字符串的末尾：

```
count2 = 0;
while(str2[count2])
  str1[count1++] = str2[count2++];
str1[count1] = '\0';
```

这里字符串库也做了一些简化。使用一个函数可以把一个字符串连接到另一个字符串上。使用下面极其简单的语句，也可以达到相同的结果：

```
strcat(str1, str2); /* Copy str2 to the end of str1 */
```

这个函数会将 str2 复制到 str1 的末尾。函数 strcat()之所以如此命名，是因为它连接了字符串，换言之，函数 strcat()不但把 str2 添加到 strl 的末尾，还返回了 str1。

如果只把源字符串的一部分附加到目标字符串上，可以使用 strncat()函数。这个函数需要第三个 size_t 类型的参数，指定要复制的字符数，例如：

```
strncat(str1, str2, 5);
/* Copy 1st 5 characters of str2 to the end of str1 */
```

与将一个字符串复制到另一个字符串的所有操作一样，这个函数也需要确保目标数组的空间足够容纳复制后的字符串，否则，这个函数与其他字符串复制函数一样，也会覆盖目标数组之后的内存空间。

这些字符串函数都会返回目标字符串，以便在其他字符串操作中使用返回值，例如：

```
size_t length  =  0;
length =  strlen(strncat(str1, str2, 5));
```

这里的 strncat()函数把 str2 中的 5 个字符复制到 str1 的末尾，返回数组 str1，所以它作为参数传送给 strlen()函数。strlen()函数会返回加上 str2 的 5 个字符后的 strl 新版本的长度。

试试看：使用字符串库

现在有足够的工具重写程序 6.3 了：

```
/* Program 6.5 Joining strings - revitalized */
#include <stdio.h>
#include <string.h>
#define STR_LENGTH 40

int main(void)
{
```

www.TopSage.com

```
  char str1[STR_LENGTH] = "To be or not to be";
  char str2[STR_LENGTH] = ",that is the question";

  if(STR_LENGTH > strlen(str1) + strlen(str2)) /* Enough space ? */
   printf("\n%s\n", strcat(str1, str2)); /* yes, so display joined string */
  else
   printf("\nYou can't put a quart into a pint pot.");
  return 0;
}
```

这个程序会产生和之前完全一样的输出。

代码的说明

库使问题变得简单了。这个程序使用#define 指令给数组的大小定义了一个符号。如果稍后要改变这个数组的尺寸，只需要修改 STR_LENGTH 的定义。用 if 语句检查数组是否有足够的空间：

```
if(STR_LENGTH > strlen(str1) + strlen(str2)) /* Enough space ? */
 printf("\n%s\n", strcat(str1, str2)); /* yes, so display joined string */
else
 printf("\nYou can't put a quart into a pint pot.");
```

如果有足够的空间，就用 strcat()函数连接字符串，并将它作为 printf()的参数。strcat()函数会返回 strl，所以 printf()会显示连接字符串后的结果。如果 strl 太短，就显示一条信息。注意，比较操作使用了>运算符——这是因为数组长度至少要比这两个字符串的总长度大 1，以便添加'\0'终止字符。

6.4.4 比较字符串

字符串库提供的函数还可以比较字符串，确定一个字符串是大于还是小于另一个字符串。字符串使用“大于”和“小于”这样的术语听起来有点奇怪，但是其结果相当简单。两个字符串的比较是基于它们的字符码，如图 6-2 所示，图中的字符码显示为十六进制数。

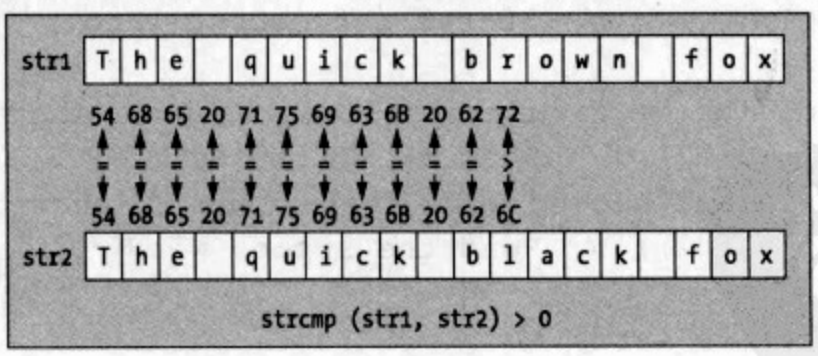


图 6-2 比较两个字符串

如果两个字符串是相同的，它们就是相等的。要确定第一个字符串是小于还是大于第二个字符串，应比较两个字符串中第一对不同的字符。例如，如果第一个字符串中某字符的字符码小于第二个字符串中的对应字符，第一个字符串就小于第二个字符串。以字母次序安排字符串时，这种比较机制一般符合我们的预期。

www.TopSage.com

函数 strcmp(str1, str2)比较两个字符串，返回一个小于、等于或大于 0 的 int 值，分别对应 str1 小于、等于和大于 str2。图 6-2 中的比较可以用如下的程序段表示：

```
char str1[] = "The quick brown fox";
char str2[] = "The quick black fox" ;
if(strcmp(str1, str2) < 0)
  printf("str1 is less than str2");
```

只有 strcmp()函数返回一个负数，才会执行 printf()语句，此时，strcmp()函数会在这两个字符串中找到一对不相同的字符，str1 的字符码小于 str2 的字符码。

strncmp()函数会比较两个字符串的前 n 个字符。它的前两个参数和 strcmp()函数相同，第三个类型为 size_t 的整数参数指定要比较的字符数。如果要处理的字符串中表示零件号或序列号的前 10 个字符，就可以使用这个函数。使用 strncmp()函数可以比较两个字符串的前 10 个字符，决定哪个字符串放在前面：

```
if(strncmp(str1, str2, 10) <= 0)
  printf("\n%s\n%s", str1, str2);
else
  printf("\n%s\n%s",  str2,   str1);
```

这些语句会根据两个字符串中的前 10 个字符，按升序输出字符串 str1 和 str2。下面用一个实例比较字符串。

试试看：比较字符串

这个例子比较两个从键盘输入的词，说明了字符串的比较方法：

```
/* Program 6.6 Comparing strings */
#include <stdio.h>
#include <string.h>

int main(void)
{
  char word1[20];          /* Stores the first word */
  char word2[20];          /* Stores the second word */

  printf("\nType in the first word (less than 20 characters):\n1: ");
  scanf("%19s", word1); /* Read the first word */
  printf("Type in the second word (less than 20 characters):\n 2: ");
  scanf("%19s", word2); /* Read the second word */

  /* Compare the two words */
  if(strcmp(word1,word2) == 0)
    printf("You have entered identical words");
  else
    printf("%s precedes %s",
           (strcmp(word1, word2) < 0) ? word1 : word2,
           (strcmp(word1, word2) < 0) ? word2 : word1);
  return 0;
```

www.TopSage.com

```
}
```

这个程序会读取两个词，按照字母顺序说明哪个词在前，如下：

```
Type in the first word (less than 20 characters):
1: apple
Type in the second word (less than 20 characters):
2: banana
apple precedes banana
```

代码的说明

程序先用#include 指令包含用于标准输入输出和处理字符串的头文件：

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

在 main()函数体中，先声明两个字符数组，以存储两个从键盘读入的词：

```
char word1[20];     /* Stores the first word */
char word2[20];     /* Stores the second word */
```

数组的大小设置为 20。这对一个例子来说足够了，但还是有点风险。与 strcpy()函数一样，必须给用户输入的词分配足够的空间。如果用格式指定符指定了宽度，函数 scanf()就可以限制字符数。这可以确保不超过数组的上限，但超出该宽度的所有字符会留在输入流中，由下一个输入流操作读入。

下一个工作是从用户处得到两个词，所以在提示后，使用 scanf()两次，从键盘读入两个词：

```
printf("\nType in the first word (less than 20 characters):\n1: ");
scanf("%19s", word1);                    /* Read the first word */
printf("Type in the second word (less than 20 characters):\n 2: ");
scanf("%19s", word2);                    /* Read the second word */
```

宽度指定为 19 个字符，可以确保不超过数组的大小，即 20 个元素。在这个例子中，没有在 scanf()函数的参数变量前使用&运算符，因为数组名本身就是一个地址，它对应于数组中第一个元素的地址。显式使用&运算符的代码应如下所示：

```
scanf("%s", &word1[0]];
```

因此，&word1[0]等于 word1，详见下一章。最后使用 strcmp()函数比较这两个输入的词：

```
if(strcmp(word1,word2) == 0)
  printf("You have entered identical words");
else
  printf("%s precedes %s",
    (strcmp(word1, word2) < 0) ? word1 : word2,
    (strcmp(word1, word2) < 0) ? word2 : word1);
```

www.TopSage.com

如果 strcmp()函数返回 0，这两个字符串就相等，显示这个结果的信息。否则，就输出一条信息，说明哪个词在另一个词之前。这里用条件运算符指定先输出哪个词，后输出哪个词。

6.4.5 搜索字符串

头文件<string.h>声明了几个字符串搜索函数，但是在探讨它们之前，先了解下一章的主题——指针，这里需要这些基础知识，以理解如何使用字符串搜索函数。

1. 指针的概念

如下一章所述，C 提供了一个非常有用的变量类型，叫做指针。指针是含有地址的变量，它含有内存中另一个包含数值的位置的引用。函数 scanf()就使用了地址。下面的第二条语句就定义了指针 pNumber：

```
int Number = 25;
int *pNumber = &Number;
```

图 6-3 展示了执行这两条语句的过程。

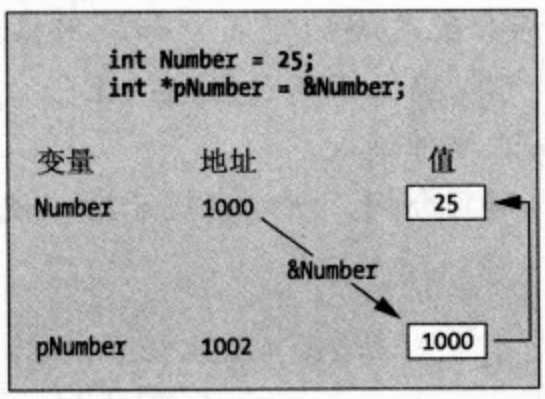


图 6-3 指针示例

这两条语句声明一个变量 Number(其值为 25)和一个指针 pNumber(它含有 Number 的地址。现在，可以在表达式*pNumber 中使用变量 pNumber，得到 Number 中包含的值。*是取消引用运算符，其作用是访问指针指定的地址中存储的数据。

之所以提早介绍这个概念，是因为下一节讨论的函数返回指针，所以如果不先解释指针，读者就会很迷惑。但如果还是不明白，别担心，这些会在下一章中详细说明。

2. 搜索字符串中的一个字符

函数 strchr()在字符串中搜索给定的字符。它的第一个参数是要搜索的字符串(是 char 数组的地址)，第二个参数是要查找的字符。这个函数会从字符串的开头开始搜索，返回在字符串中找到的第一个给定字符的地址。这是一个在内存中的地址，其类型为 char*，表示“char 的指针”。所以要存储这个返回值，必须创建一个能存储字符地址的变量。如果没有找到给定的字符，函数就会返回 NULL，它相当于 0，表示这个指针没有指向任何对象。

www.TopSage.com

函数 strchr()的用法如下：

```
char str[] = "The quick brown fox"; /* The string to be searched */
char c = 'q';                       /* The character we are looking for */
char *pGot_char = NULL;             /* Pointer initialized to zero */
pGot_char = strchr(str, c); /* Stores address where c is found */
```

用 char 类型变量 c 定义要查找的字符。strchr()函数的第二个参数是 int 类型，所以编译器在将它传给函数之前，先把 c 的值转换为 int 类型。

也可以将 c 定义成 int 类型，如下：

```
int c = 'q'; /* Initialize with character code for q */
```

函数经常要求将字符作为 int 类型参数传入，因为 int 类型比 char 类型更易用。图 6-4 说明了使用 strchr()函数搜索的结果。

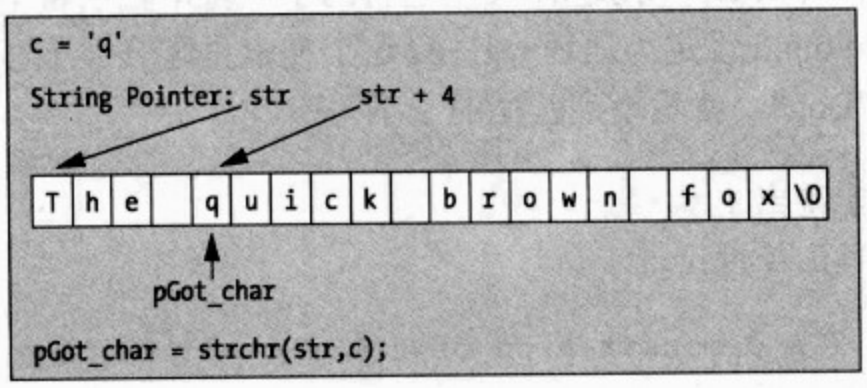


图 6-4 搜索一个字符

在字符串中，第一个字符的地址是数组名称 str 指定的。'q'是字符串中的第 5 个字符，所以它的地址是 str+4，与第一个字符偏移 4 字节。因此变量 pGot_char 将含有地址 str+4。

在表达式中使用变量名称 pGot_char 可以访问地址。如果要访问存储该地址中的字符，就必须取消对这个指针的引用。为此，在指针变量名之前使用取消引用运算符*，例如：

```
printf("Character found was %C. ", *pGot_char);
```

下一章将详细介绍取消引用运算符。当然，我们要查找的字符不一定在字符串中，所以不要试图取消对 NULL 指针的引用，否则，程序会崩溃。只要使用 if 语句就可以避免这种情况，如下：

```
if(pGot_char != NULL)
  printf("Character found was %c.", *pGot_char);
```

现在，只要变量 pGot_char 不是 NULL，就执行 printf()语句。

函数 strrchr()非常类似于 strchr()，但后者从字符串的末尾开始查找字符。因此，它返回字符串中的最后一个给定字符的地址，如果找不到给定字符，就返回 NULL。

www.TopSage.com

3. 在字符串中查找子字符串

strstr()函数是<string.h>头文件声明的所有搜索函数中最有用的函数，它查找一个字符串中的子字符串，返回找到的第一个子字符串的位置指针。如果找不到匹配的子字符串，就返回 NULL。所以如果返回值不是 NULL，就说明这个函数找到了所需的子字符串。这个函数的第一个参数是要搜索的字符串，第二个参数是要查找的子字符串。下面有一个使用 strstr()函数的例子：

```
char text[] = "Every dog has his day";
char word[] = "dog";
char *pFound = NULL;
pFound = strstr(text, word);
```

这些语句在字符串 text 中寻找 word 中包含的子字符串。字符串"dog"出现在 text 的第 7 个位置，所以 pFound 设定为地址 text+6。这个搜索是区分大小写的，所以如果在 text 字符串中查找的是"Dog"，就找不到匹配的子字符串。

试试看：搜索字符串

下面的部分代码已经讨论过了：

```
/* Program 6.7 A demonstration of seeking and finding */
#include <stdio.h>
#include <string.h>

int main(void)
{
  char str1[] = "This string contains the holy grail.";
  char str2[] = "the holy grail";
  char str3[] = "the holy grill";

  /* Search str1 for the occurrence of str2 */
  if(strstr(str1, str2) == NULL)
    printf("\n\"%s\" was not found.", str2);
  else
    printf("\n\"%s\" was found in \"%s\"",str2, str1);

  /* Search str1 for the occurrence of str3 */
  if(strstr(str1, str3) == NULL)
    printf("\n\"%s\" was not found.", str3);
  else
    printf("\nWe shouldn't get to here!");
  return 0;
}
```

这个程序会产生如下输出：

```
"the holy grail" was found in "This string contains the holy grail."
```

www.TopSage.com

```
"the holy grill" was not found.
```

代码的说明

注意，要使用任何字符串处理函数，必须使用#include 指令添加<string.h>头文件。接着定义 3 个字符串 str1、str2 和 str3:

```
char str1[] = "This string contains the holy grail.";
char str2[] = "the holy grail";
char str3[] = "the holy grill";
```

在第一个 if 语句中，使用库函数 strstr()搜索在第一个字符串中出现的第二个字符串:

```
if(strstr(str1, str2) == NULL)
  printf("\n\"%s\" was not found.", str2);
else
  printf("\n\"%s\" was found in \"%s\"",str2, str1);
```

比较 strstr()的返回值和 NULL，显示一条对应的信息。如果返回值等于 NULL，就表示在第一个字符串中没有找到第二个字符串，显示对应的信息。如果找到了第二个字符串，就执行 else，显示找到字符串的信息。

然后，在第二个 if 语句中重复这个过程，在第一个字符串中查找第三个字符串:

```
if(strstr(str1, str3) == NULL)
  printf("\n\"%s\" was not found.", str3);
else
  printf("\nWe shouldn't get to here!");
```

如果第一个或最后一个 printf()生成了结果，就表示程序有严重的错误。

6.5 分析和转换字符串

如果需要检查字符串内部的内容，可以使用在头文件<ctype.h>(详见第 3 章)中声明的标准库函数。这些都是非常灵活的分析函数，可以测试有什么样的字符。它们还独立于计算机上的字符码。表 6-1 中的函数可以测试各种不同的字符种类。

表 6-1 字符分类函数

函 数	测 试 内 容
islower()	小写字母
isupper()	大写字母
isalpha()	大写或小写字母
isalnum()	大写或小写字母，或数字
iscntrl()	控制字符
isprint()	可打印字符，包括空格

www.TopSage.com

(续表)

函　数	测 试 内 容
isgraph()	可打印字符，不包括空格
isdigit()	十进制数字('0'~'9')
isxdigit()	十六进制数字('0'~'9'、'A'~'F'、'a'~'f')
isblank()	标准空白字符(空格、'\t')
isspace()	空白字符(空格、'\n'、'\t'、'\v'、'\r'、'\f')
ispunct()	isspace()和 isalnum()返回 false 的可打印字符

这些函数的参数是要测试的字符。如果这个字符在该函数的测试内容范围之内，所有这些函数都返回一个非零的 int 值,；否则返回 0。当然，这些返回值可以转换为 true 或 false，以便用作布尔值。下面使用这些函数测试一个字符串中的字符。

试试看：使用字符分类函数

下面的例子判断从键盘输入的一个字符串中有多少个数字和字母:

```
/* Program 6.8 Testing characters in a string */
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char buffer[80];      /* Input buffer */
  int i = 0;            /* Buffer index */
  int num_letters = 0;  /* Number of letters in input */
  int num_digits = 0;   /* Number of digits in input */

  printf("\nEnter an interesting string of less than 80 characters:\n");
  gets(buffer);         /* Read a string into buffer */

  while(buffer[i] != '\0')
  {
    if(isalpha(buffer[i]))
      num_letters++;     /* Increment letter count */
    if(isdigit(buffer[i++]))
      num_digits++;      /* Increment digit count */
  }
  printf("\nYour string contained %d letters and %d digits.\n",
       num_letters, num_digits);
  return 0;
}
```

这个程序的输出如下:

```
Enter an interesting string of less than 80 characters:
```

www.TopSage.com

```
I was born on the 3rd of October 1895

Your string contained 24 letters and 5 digits.
```

代码的说明

这个例子相当简单。用以下语句将字符串读入数组 buffer:

```
gets(buffer);
```

这个语句使用新的标准库函数 gets()将输入的字符串读入数组 buffer。前面只使用 scanf()接受键盘输入，但是它不适合于读入字符串，因为它会将空格解释为输入值的末尾。而 gets()函数的优点是可以从键盘读入所有字符(包含空白)，直到按下回车键为止。然后，将字符串存储到其参数指定的区域中，在这个例子中，是存储到数组 buffer 中。在字符串的末尾会自动附加一个'\0'字符。

与输入或输出操作一样，gets()函数也有可能出错。如果 gets()函数在读入输入时发生错误，就会返回 NULL(通常它会返回传给它的参数的地址，这个例子是 buffer)。因此，可以使用下面的代码检查输入操作是否成功:

```
if(gets(buffer)  ==  NULL)
{
  printf("Error  reading  input.");
  return 1;                          /* End the program */
}
```

如果输入操作失败，就输出一条信息，并结束程序。键盘输入错误很少见，所以在读取键盘输入时，没有在例子中包含这个检查。但是如果读入一个文件，确认读入是否成功就非常重要了。

gets()函数的缺点是，它会读取任意长度的字符串，并存储在 buffer 中，但没有检查 buffer 是否有足够的空间存储该字符串，所以有可能使程序崩溃。为了避免这种情况，可以使用 fgets()函数，它允许指定输入字符串的最大长度。这个函数可用于任意种类的输入字符串，而 gets()只能读取标准输入流 stdin; 所以还必须指定 fgets()的第三个参数，说明要读取的输入流。下面使用 fgets()从键盘上读取一个字符串:

```
if(fgets(buffer, sizeof(buffer), stdin) == NULL)
{
  printf("Error reading input.");
  return 1;                          /* End the program */
}
```

fgets()函数读取的字符数比第二个参数指定的字符数大 1，接着在内存中把\0 字符附加到字符串的末尾，所以这个例子中的第二个参数是 sizeof(buffer)。注意，fgets()和 gets()还有另一个重要的区别。这两个函数都读取一个换行符来结束输入过程，但在输入换行符时，fgets()会存储一个'\n'字符，而 gets()不会。因此，如果从键盘上读取字符串，fgets()读取的字符串比 gets()读取的字符串多一个字符。另外，在输入时按下回车键，gets()读取的输入会附加一个空字符串\0，而 fgets()读取的输入会附加\n\0。本章的下一个程序 6.9

www.TopSage.com

将使用 fgets()，在这个例子中，要计算字符串中存储了多少个换行符，第 12 章将详细介绍 fgets()函数。

分析字符串的语句如下：

```
while(buffer[i] != '\0')
{
  if(isalpha(buffer[i]))
    num_letters++;       /* Increment letter count */
  if(isdigit(buffer[i++]))
    num_digits++;        /* Increment digit count */
}
```

在 while 循环中逐个字符地检查输入的字符串。在两个 if 语句中检查字母和数字。找到字母或数字时，就给对应的计数器加 1。注意，在第二个 if 中递增 buffer 数组的索引。这里使用了前置形式的递增运算符，所以检查时使用了 i 的当前值，之后递增 i。

也可以不用 if 语句，如下：

```
while(buffer[i] != '\0')
{
  num_letters += isalpha(buffer[i]) != 0;
  num_digits += isdigit(buffer[i++]) != 0;
}
```

如果参数在测试字符组内，测试函数就返回一个非零值(不一定是 1)。如果字符属于所测试的类别，赋值运算符右边的逻辑表达式的值就是 true，否则是 false。

这个例子的编码方法不是很有效率，因为即使已知当前字符是一个字母，也要测试它是否是数字。读者可以试着改善它。

6.5.1 转换字符

标准库<ctype.h>还包含两个转换函数。函数 toupper()将小写字母转换成大写，函数 tolower()将大写字母转换成小写。这两个函数都返回转换后的字符，如果字母的大小写形式是正确的，就返回原来的字符。因此，以下这些语句可以一个字符串转换成大写：

```
for(int i = 0 ; (buffer[i] = toupper(buffer[i])) != '\0' ; i++);
```

这个循环会一次一个字符地遍历字符串，将整个字符串转换成大写：将小写字母转换成大写，原本已是大写的字母不变。到达终止字符'\0'时，循环就结束。这种在循环控制表达式中完成所有工作的方式在 C 语言中很常见。

下面的例子对一个字符串应用这些函数。

试试看：转换字符

使用函数 toupper()和函数 strstr()可以确定一个字符串是否出现在另一个字符串中(忽略大小写)。

www.TopSage.com

```
/* Program 6.9 Finding occurrences of one string in another */
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main(void)
{
  char text[100];      /* Input buffer for string to be searched */
  char substring[40]; /* Input buffer for string sought */

  printf("\nEnter the string to be searched (less than 100
        characters):\n");
  fgets(text, sizeof(text), stdin);

  printf("\nEnter the string sought (less than 40 characters):\n");
  fgets(substring, sizeof(substring), stdin);

  /* overwrite the newline character in each string */
  text[strlen(text)-1] = '\0';
  substring[strlen(substring)-1] = '\0';

  printf("\nFirst string entered:\n%s\n", text);
  printf("\nSecond string entered:\n%s\n", substring);

  /* Convert both strings to uppercase. */
  for(int i = 0 ; (text[i] = toupper(text[i])) ; i++);
  for(int i = 0 ; (substring[i] = toupper(substring[i])) ; i++);

  printf("\nThe second string %s found in the first.",
    ((strstr(text, substring) == NULL) ? "was not" : "was"));
  return 0;
}
```

这个例子的输出如下:

```
Enter the string to be searched(less than 100 characters):
Cry havoc, and let slip the dogs of war.

Enter the string sought (less than 40 characters ):
The Dogs of War

First string entered:
Cry havoc, and let slip the dogs of war

Second string entered:
The Dogs of War

The second string was found in the first.
```

www.TopSage.com

代码的说明

这个程序有三个明显的阶段：获取输入字符串，将两个字符串转换成大写，在第一个字符串中搜索第二个字符串。

首先，使用 printf()提示用户输入字符串，然后使用上一个例子中介绍的 fgets()函数，将输入放在变量 text 和 substring 中：

```
printf("\nEnter the string to be searched (less than 100 characters):\n");
fgets(text, sizeof(text), stdin);
printf("\nEnter the string sought (less than 40 characters):\n");
fgets(substring, sizeof(substring), stdin);
```

这里使用 fgets()函数，是因为它可以从键盘上读入任何字符串(包含空格)，按下回车键后，输入就终止。第一个字符串 text 至多输入 99 个字符，第二个字符串 substring 至多输入 39 个字符。如果输入了过多的字符，就忽略它们，所以该程序的操作是安全的。

fgets()函数还存储了结束输入过程的换行符，对于第一个字符串而言，这并不重要，但对于要在第一个字符串中查找的第二个字符串来说，就有重要影响了。例如，如果要查找的字符串是 dogs，fgets()函数就会存储 dogs\n，两者并不完全相同。因此要用'\0'字符覆盖\n，去掉两个字符串中的换行符：

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

换行符是每个字符串中的倒数第二个字符，该位置的索引是字符串的长度减 1。当然，如果放大了输入限制，字符串就会被截断，结果也就不正确了。这可以从下述语句生成的两个字符串看出：

```
printf("\nFirst string entered:\n%s\n", text);
printf("\nSecond string entered:\n%s\n", substring);
```

用以下的语句将两个字符串转换成大写：

```
for(int i = 0 ; (text[i] = toupper(text[i])) ; i++);
for(int i = 0 ; (substring[i] = toupper(substring[i])) ; i++);
```

这些语句使用 for 循环进行转换，所有的工作都在循环的控制表达式中完成。第一个 for 循环将 i 初始化成 0，然后在循环的条件式中将 text 的第 i 个字符转换成大写，将结果存回 text 中原来的位置。只要第二个循环控制表达式中 text[i]的字符码不是 0，即除 NULL 之外的任意字符，循环就继续执行。索引 i 在循环的第三个控制表达式中递增。这样可保证递增 i 时不会造成混乱。第二个循环以完全相同的方法将 substring 转换成大写。

两个字符串转换成大写后，就可以检查 text 中是否有 substring，且无须考虑它们的大小写形式。这个测试在报告结果的输出语句里完成。

```
printf("\nThe second string %s found in the first.",
   ((strstr(text, substring) == NULL) ? "was not" : "was"));
```

条件运算符根据 strstr()函数是否返回 NULL，选择 was not 或 was 作为输出字符串的

www.TopSage.com

一部分。当第二个参数指定的字符串不在第一个字符串中时，strstr()函数就返回 NULL，否则，它返回找到的字符串的地址。

6.5.2 将字符串转换成数值

头文件<stdlib.h>声明了一些能将字符串转换成数值的函数。表 6-2 中的每个函数都需要一个指针参数，指向一个字符串或包含字符串的字符数组，该字符串代表一个数值。

表 6-2 将字符串转换成数值的函数

函　数	返 回 值
atof()	从字符串参数中生成的 double 类型的值
atoi()	从字符串参数中生成的 int 类型的值
atol()	从字符串参数中生成的 long 类型的值
atoll()	从字符串参数中生成的 long long 类型的值

这些函数的用法很简单，例如：

```
char value_str[] = "98.4";
double value = 0;
value = atof(value_str); /* Convert string to floating-point */
```

数组 value_str 含有 double 类型值的字符串表示。将数组名作为参数传给 atof()函数，就可以把它转换成 double 类型。其他三个函数的用法与此类似。

这些函数在需要以字符串格式读取数值时特别有用。在数据输入的顺序不确定时，需要去分析这个字符串，以决定它含有什么数据。一旦知道这个字符串代表哪种数值，就可以使用适当的库函数去转换它。

6.7 使用宽字符串

使用宽字符串与使用前面的字符串一样简单。宽字符串存储在 wchar_t 类型的数组中，宽字符串常量只需在其前面加上 L 修饰符。因此，可以用如下方式声明和初始化宽字符串：

```
wchar_t proverb[] = L"A nod is as good as a wink to a blind horse.";
```

如第 2 章所述，wchar_t 字符占用 2 个字节。proverb 字符串包含 44 个字符和终止空字符，所以它占用 90 个字节。

如果要使用 printf()将字符串输出到屏幕上，必须使用%S 格式指定符，而不是用于 ASCII 字符串的%s。如果使用了%s，printf()函数就假定字符串包含单字节字符，这样结果就不正确了。下面的语句会正确输出宽字符串：

www.TopSage.com

```
printf("The proverb is:\n%S", proverb);
```

宽字符串的操作

<wchar.h>头文件声明了一些函数来操作宽字符串，它们对应于处理一般字符串的函数。表 6-3 列出的宽字符串处理函数在<wchar.h>中声明，它们对应于本章前面介绍的常规字符串处理函数。

表 6-3 处理宽字符串的函数

函　数	说　明
wcslen(const wchar_t* ws)	返回类型的值，它表示宽字符串 ws 的长度。该长度不包含终止字符 L'\0'
wcscpy(wchar_t* destination, const wchar_t source)	将宽字符串 source 复制到宽字符串 destination 中，该函数返回 source
wcsncpy(wchar_t* destination, const wchar_t source, size_t n)	将宽字符串 source 中的前 n 个字符复制到宽字符串 destination 中，如果 source 包含的字符数少于 n，就给 destination 补足字符 L'\0'。该函数返回 source
wcscat(whar_t* ws1, whar_t* ws2)	将 ws2 的副本添加到 ws1 的尾部。ws2 的第一个字符覆盖 ws1 尾部的终止空字符。该函数返回 ws1
wcsncmp(const wchar_t* ws1, const wchar_t* ws2)	比较 ws1 指向的宽字符串和 ws2 指向的宽字符串，返回一个 int 类型的值。如果字符串 ws1 小于、等于或大于字符串 ws2，该 int 值就小于、等于或大于 0
wcscmp(const wchar_t* ws1, const wchar_t* ws2, size_t n)	比较 ws1 指向的宽字符串的前 n 个字符和 ws2 指向的宽字符串的前 n 个字符，返回一个 int 类型的值。如果字符串 ws1 的前 n 个字符小于、等于或大于字符串 ws2 的前 n 个字符，该 int 值就小于、等于或大于 0
wcschr(const wchar_t* ws, wchar_t wc)	在 ws 指向的宽字符串中查找并返回第一个 wc 宽字符串的指针。如果在 ws 中没有找到 wc，就返回 NULL 指针值
wcsstr(const wchar_t* ws1, const wchar_t* ws2)	在 ws1 宽字符串中查找并返回第一个 ws2 宽字符串的指针。如果在 ws1 中没有找到 ws2，就返回 NULL 指针值

从表中可以看出，所有这些函数的工作方式都与前面介绍的字符串函数相同。参数类型的说明中包含 const 关键字时，表示该参数不能由函数修改。这迫使编译器检查函数是否没有改变该参数。在探讨如何创建自己的函数时，将详细探讨这些内容(详见

www.TopSage.com

第 7 章)。

<wchar.h>头文件还声明了 fgetws()函数，它从 stdin 等流中读取宽字符串，stdin 默认对应键盘输入。必须给 fgetws()函数提供 3 个参数，与读取单字节字符串的 fgets()函数类似：

- 第一个参数是包含字符串的 wchar_t 数组指针。
- 第二个参数是 size_t 类型的值 n，表示可以存储在数组中的最大字符数。
- 第三个参数是读取数据的流，从键盘上读取字符串时，该流是 stdin。

该函数从流中至多读取 n - 1 个字符，把它们存储在数组中，并附加一个 L'\0'。从流中读取的字符数少于 n - 1 时，若读取了一个换行符，表示输入结束。该函数返回一个包含字符串的数组指针。

测试并转换宽字符

<wchar.h>头文件还声明了测试宽字符子集的函数，类似于前面测试 char 类型字符的函数，如表 6-4 所示。

表 6-4 宽字符分类函数

函 数	测 试 内 容
iswlower()	小写字母
iswupper()	大写字母
iswalnum()	大写或小写字母
iswcntrl()	控制字符
iswprint()	可打印字符，包括空格
iswgraph()	可打印字符，不包括空格
iswdigit()	十进制数字(L'0'~ L '9')
iswxdigit()	16 进制数字(L '0'~ L '9'、L 'A'~ L 'F'、L 'a'~ L 'f)
iswblank()	标准空白字符(空格、L'\t')
iswspace()	空白字符(空格、L'\n'、L'\t'、L'\v'、L'\r'、L'\f)
iswpunct()	iswspace()和 iswalnum()返回 false 的可打印字符

大小写转换函数 towlower() 和 towupper()分别返回 wchar_t 参数的大小写形式。程序 6.9 的宽字符版本演示了一些宽字符函数的用法。

试试看：转换宽字符

这个例子使用 fgets()、toupper()和 wcsstr()的宽字符版本。程序 6.9 中有改动的代码以粗体显示：

```
/* Program 6.9A Finding occurrences of one wide character string in another */
#include <stdio.h>
```

www.TopSage.com

```
#include <wchar.h>

int main(void)
{
  wchar_t text[100];      /* Input buffer for string to be searched */
  wchar_t substring[40]; /* Input buffer for string sought */

  printf("\nEnter the string to be searched(less than 100
        characters):\n");
  fgetws(text, 100, stdin);

  printf("\nEnter the string sought (less than 40 characters):\n");
  fgetws(substring, 40, stdin);

  /* overwrite the newline character in each string */
  text[wcslen(text)-1] = L'\0';
  substring[wcslen(substring)-1] = L'\0';

  printf("\nFirst string entered:\n%S\n", text);
  printf("\nSecond string entered:\n%S\n", substring);

  /* Convert both strings to uppercase. */
  for(int i = 0 ; (text[i] = towupper(text[i])) ; i++);
  for(int i = 0 ; (substring[i] = towupper(substring[i])) ; i++);

  printf("\nThe second string %s found in the first.",
    ((wcsstr(text, substring) == NULL) ? "was not" : "was"));
  return 0;
}
```

输出与前面的例子相同。

代码的说明

这个程序与前面的例子完全相同，只是把输入存储到宽字符串中，使用了宽字符函数。由于这个例子与前面的程序非常类似，没有必要再解释什么。当然。数组元素的类型是 wchar_t，函数名略有不同。从键盘上将输入读取到宽字符数组中是使用 fgetws()函数完成的，在该函数中，为第二和第三个参数指定了可以存储的字符个数和流的名称。每个字符串中的换行符用终止空字符的宽字符版本 L'\0'替代。在字符字面量的前面加上 L，使其类型变成 wchar_t。当然，输出字符串的语句使用了%S，因为这里要输出宽字符串。

6.8 设计一个程序

本章就要结束了。剩下的就是利用前面所学的知识完成一个较大的程序。

www.TopSage.com

6.8.1 问题

开发一个程序，从键盘上读取任意长度的一段文本，确定该文本中每个单词的出现频率(忽略大小写)。该段文本的长度不完全是任意的，因为我们要给程序中的数组大小指定一个限制，但可以使该数组存储任意大小的文本。

6.8.2 分析

要从键盘上读取一段文本，需要读取任意长度的输入行，把它们合并为一个最终包含整个段落的字符串。我们不希望截断输入行，所以 fgets()似乎是输入操作的首选函数。如果在代码的开头定义一个符号，指定存储该段落的数组大小，则只要改变该符号的定义，就可以改变程序的容量。

这段文本将包含标点符号，如果能将各个单词分隔开，就必须以某种方式处理它们。如果每个单词都用一个或多个空格相互隔开，从文本中提取单词就非常简单。为此，可以用空格替换单词中没有出现的字符。我们还要从文本段中删除所有的标点符号和其他古怪的字符。不需要保留原来的文本，但如果要保留它，可以在删除标点符号之前，进行复制。

分隔单词是很简单的，只需提取每段连续的、没有空格的字符，作为一个单词。可以把这些单词存储在另一个数组中。我们要计算单词的出现次数(忽略大小写)，所以可以把每个单词存储为小写形式。找到一个新的单词时，必须将它与已找到的所有单词进行比较，确定它以前是否出现过。只有单词以前未出现过，才把它存储在数组中。要记录每个单词的出现次数，需要另一个数组来存储单词的出现次数。这个数组需要包含的元素个数与程序找到的单词个数相同。

6.8.3 解决方案

本节列出解决问题的步骤。程序包含一系列相互独立的步骤。现在，实现该程序的方法受到目前已掌握的知识的限制，在学到第 9 章时，可以更高效地完成这个程序。

1. 步骤 1

第一步从键盘上读取段落。输入行数是任意的，所以需要使用一个无限循环。首先，定义用于实现输入机制的变量：

```
/* Program 6.10 Analyzing text */
#include <stdio.h>
#include <string.h>

#define TEXTLEN 10000    /* Maximum length of text */
#define BUFFERSIZE 100   /* Input buffer size */

int main(void)
```

www.TopSage.com

```
{
  char text[TEXTLEN+1];
  char buffer[BUFFERSIZE];
  char endstr[] = "*\n";    /* Signals end of input */

  printf("Enter text on an arbitrary number of lines.");
  printf("\nEnter a line containing just an asterisk to end input:\n\n");

  /* Read an arbitrary number of lines of text */
  while(true)
  {
    /* A string containing an asterisk followed by newline */
    /* signals end of input */
    if(!strcmp(fgets(buffer, BUFFERSIZE, stdin), endstr))
      break;

    /* Check if we have space for latest input */
    if(strlen(text)+strlen(buffer)+1 > TEXTLEN)
    {
      printf("Maximum capacity for text exceeded. Terminating program.");
      return 1;
    }
    strcat(text, buffer);
  }

  /* Plus the rest of the program code ... */

  return 0;
}
```

编译并运行这段代码。符号 TEXTLEN 和 BUFFERSIZE 分别指定 text 和 buffer 数组的大小。text 数组存储整个段落，buffer 数组存储一行输入。我们需要某种方式，让用户告诉程序他已经输入完文本了。如最初的输入提示所示，在一行上输入一个星号表示输入结束。fgets()函数将单个星号输入读取为字符串"*\n"，因为在按下回车键时，该函数会存储换行符。endstr 数组存储表示输入结束的字符串，以便将每个输入行与这个数组作比较。

在输入提示后，在无限的 while 循环中进行整个输入过程，在 if 语句中读取一行输入：

```
if(!strcmp(fgets(buffer, BUFFERSIZE, stdin), endstr))
  break;
```

函数从 stdin 中读取的最大字符数是 BUFFERSIZE - 1。如果用户输入的文本行超过了这个长度，不要紧。超过 BUFFERSIZE - 1 的字符会留在输入流中，在下一个循环迭代中读入。将 BUFFERSIZE 设置为 10，输入超过 10 个字符的文本行，就可以检查这个输入机制是否有效。

fgets()函数返回一个指针，该指针指向传送为第一个参数的字符串，所以可以把 fgets()

www.TopSage.com

用作 strcmp()函数的参数，比较读取的字符串和 endstr。因此，if 语句不仅读取一行输入，还检查输入是否已结束。

在将新的输入行添加到 text 中之前，应检查 text 中是否有足够的自由空间容纳新的输入行。要添加新的输入行，只需使用 strcat()库函数，把 buffer 中的字符串连接到 text 中已有的字符串上。

下面是执行这个输入操作的结果：

```
Enter text on an arbitrary number of lines.
Enter a line containing just an asterisk to end input:

Mary had a little lamb,
Its feet were black as soot,
And into Mary's bread and jam,
His sooty foot he put.
*
```

2. 步骤 2

读取了所有的输入文本后，就可以用空格替换标点符号和 fgets()函数记录的换行符了。下面的代码放在 main()尾部的 return 语句前面：

```
/* Replace everything except alpha and single quote characters by spaces */
for(int i = 0 ; i < strlen(text) ; i++)
{
  if(text[i] == quote || isalnum(text[i]))
    continue;
  text[i] = space;
}
```

这个循环遍历 text 数组存储的字符串中的字符。假定单词只包含字母、数字和单引号，所以不在这个集合中的所有字符都会被空格字符替换。在<ctype.h>头文件中声明的 isalnum()函数对字母或数字返回 true，所以必须在程序中为这个头文件添加#include 指令。还需要在 endstr 的声明后面，添加变量 quote 和 space 的声明：

```
const char space = ' ';
const char quote = '\'';
```

当然，可以在代码中直接使用字符字面量，但像这样定义变量，有助于使程序更容易理解。

3. 步骤 3

下一步是从 text 数组中提取单词，把它们存储在另一个数组中。首先，添加两个符号的定义，它们与存储单词的数组相关。这些定义放在 BUFFERSIZE 的定义后面：

```
#define MAXWORDS 500  /* Maximum number of different words */
#define WORDLEN 15    /* Maximum word length */
```

www.TopSage.com

现在可以为其他数组和变量添加声明了，以便从 text 中提取单词，这些语句可以放在 main()开头处已有声明的后面：

```
char words[MAXWORDS][WORDLEN+1];
int nword[MAXWORDS];      /* Number of word occurrences */
char word[WORDLEN+1];     /* Stores a single word */
int wordlen = 0;          /* Length of a word */
int wordcount = 0;        /* Number of words stored */
```

words 数组至多存储长度为 WORDLEN 的 MAXWORDS 个单词(不包含终止字符)。nword 数组存储 words 数组中各个单词的出现次数。每找到一个新单词，就把它存储在 words 数组中的下一个空闲元素中，将 nword 数组中索引位置相同的元素设置为 1。找到一个已在 words 中的单词时，只需递增 nword 数组中相应的元素。

在另一个无限的 while 循环中提取 text 数组中的单词，因为事先不知道这段文本中有多少个单词。这个循环的代码很多，所以这里一点一点地添加它们。下面是最初的循环代码：

```
/* Find unique words and store in words array */
int index = 0;
while(true)
{
  /* Ignore any leading spaces before a word */
  while(text[index] == space)
    ++index;

  /* If we are at the end of text, we are done */
  if(text[index] == '\0')
    break;

  /* Extract a word */
  wordlen = 0;          /* Reset word length */
  while(text[index] == quote || isalpha(text[index]))
  {
    /* Check if word is too long */
    if(wordlen == WORDLEN)
    {
      printf("Maximum word length exceeded. Terminating program.");
      return 1;
    }
    word[wordlen++] = tolower(text[index++]); /* Copy as lowercase */
  }
  word[wordlen] = '\0';                    /* Add string terminator */
}
```

这段代码放在 main()函数中现有代码的后面，在最后的 return 语句之前。

index 变量记录了 text 数组中的当前字符位置。外层循环中的第一个操作是跳过前面所有的空格，使 index 指向单词中的第一个字符。为此，在内层的 while 循环中，只要当

www.TopSage.com

前字符是空格，就递增 index。

因为有可能已经到达了 text 中字符串的尾部，所以接下来要检查一下。如果 index 位置上的当前字符是'\0'，就退出循环，因为已经提取了所有的单词。

提取单词只需复制字母、数字或单引号。如果所提取的字符不是字母、数字或单引号，就表示到了该单词的末尾。在另一个 while 循环中把组成单词的字符复制到 word 数组中，之后使用标准库中的 tolower()函数把每个字符转换为小写。在把字符存储到 word 中之前，要检查单词的长度是否超出了数组的大小。复制完成后，只需给 word 数组中的字符添加终止字符。

在循环中，下一个操作是确定刚才提取的单词是否已在 words 数组中。下面的代码完成了这个任务，它们放在 while 循环的闭括号之前：

```
/* Check for word already stored */
bool isnew = true;
for(int i = 0 ; i< wordcount ; i++)
  if(strcmp(word, words[i]) == 0)
  {
    ++nword[i];
    isnew = false;
    break;
  }
```

isnew 变量记录了 words 数组中是否已有这个单词，该变量最初设置为最新提取的单词是一个新单词。在 for 循环中，使用库函数比较 word 和 words 数组中的字符串。如果两个字符串相同，该函数就返回 0；此时把 isnew 设置为 false，递增 nword 数组中的对应元素，并退出 for 循环。

从 text 中提取单词的无限循环中，最后一个操作是把新单词存储到 words 数组中，如下面的代码所示：

```
if(isnew)
{
  /* Check if we have space for another word */
  if(wordcount >= MAXWORDS)
  {
    printf("\n Maximum word count exceeded. Terminating program.");
    return 1;
  }

  strcpy(words[wordcount], word); /* Store the new word */
  nword[wordcount++] = 1;         /* Set its count to 1 */
}
```

这段代码也放在上述代码的后面，在无限的 while 循环的闭括号之前。如果 isnew 指示器是 true，就要存储一个新单词，但首先要确认 words 数组中还有自由空间。strcpy()函数把 word 中的字符串复制到根据 wordcount 选择的 words 数组元素中。接着设置 nword 数组的对应元素值，使之包含 text 中某单词的出现次数。

www.TopSage.com

4. 步骤 4

最后一段代码输出单词及其出现次数。下面是本程序的完整代码，其中第 3 和第 4 步的代码显示为粗体：

```
/* Program 6.10 Analyzing text */
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

#define TEXTLEN 10000  /* Maximum length of text */
#define BUFFERSIZE 100 /* Input buffer size */
#define MAXWORDS 500   /* Maximum number of different words */
#define WORDLEN 15     /* Maximum word length */

int main(void)
{
  char text[TEXTLEN+1];
  char buffer[BUFFERSIZE];
  char endstr[] = "*\n"; /* Signals end of input */

  const char space = ' ';
  const char quote = '\'';

  char words[MAXWORDS][WORDLEN+1];
  int nword[MAXWORDS];   /* Number of word occurrences */
  char word[WORDLEN+1]; /* Stores a single word */
  int wordlen = 0;       /* Length of a word */
  int wordcount = 0;     /* Number of words stored */

  printf("Enter text on an arbitrary number of lines.");
  printf("\nEnter a line containing just an asterisk to end input:\n\n");

  /* Read an arbitrary number of lines of text */
  while(true)
  {
    /* A string containing an asterisk followed by newline */
    /* signals end of input */
    if(!strcmp(fgets(buffer, BUFFERSIZE, stdin), endstr))
      break;

    /* Check if we have space for latest input */
    if(strlen(text)+strlen(buffer)+1 > TEXTLEN)
    {
      printf("Maximum capacity for text exceeded. Terminating program.");
      return 1;
    }
    strcat(text, buffer);
```

www.TopSage.com

```
}

/* Replace everything except alpha and single quote characters by spaces */
for(int i = 0 ; i < strlen(text) ; i++)
{
  if(text[i] == quote || isalnum(text[i]))
    continue;
  text[i] = space;
}

/* Find unique words and store in words array */
int index = 0;
while(true)
{
  /* Ignore any leading spaces before a word */
  while(text[index] == space)
    ++index;

  /* If we are at the end of text, we are done */
  if(text[index] == '\0')
    break;

  /* Extract a word */
  wordlen = 0; /* Reset word length */
  while(text[index] == quote || isalpha(text[index]))
  {
    /* Check if word is too long */
    if(wordlen == WORDLEN)
    {
      printf("Maximum word length exceeded. Terminating program.");
      return 1;
    }
    word[wordlen++] = tolower(text[index++]); /* Copy as lowercase */
}
word[wordlen] = '\0'; /* Add string terminator */

/* Check for word already stored */
bool isnew = true;
for(int i = 0 ; i< wordcount ; i++)
  if(strcmp(word, words[i]) == 0)
  {
    ++nword[i];
    isnew = false;
    break;
  }

if(isnew)
{
  /* Check if we have space for another word */
```

www.TopSage.com

```
      if(wordcount >= MAXWORDS)
      {
        printf("\n Maximum word count exceeded. Terminating program.");
        return 1;
      }

      strcpy(words[wordcount], word); /* Store the new word */
      nword[wordcount++] = 1;         /* Set its count to 1 */
      }
    }

    /* Output the words and frequencies */
    for(int i = 0 ; i<wordcount ; i++)
    {
      if( !(i%3) )                    /* Three words to a line */
        printf("\n");
      printf(" %-15s%5d", words[i], nword[i]);
    }
    return 0;
}
```

粗体显示的 7 行代码输出了单词及其出现次数。这是在一个 for 循环中遍历单词而实现的。循环代码在每一行上输出 3 个单词及其出现次数，如果 i 的当前值是 3 的倍数，就给 stdout 写入一个换行符。当 i 是 3 的倍数时，表达式 i%3 是 0，这个值对应布尔值 false，所以!(i%3)表达式是 true。

这个程序的 main()函数超过了 100 条语句。在学习完 C 语言时，可以将这个程序的代码放在几个比较短的函数中，以完全不同的方式组织这个程序。第 9 章将完成这个任务，读者在学习完第 9 章时，再看看这个例子。下面是这个程序的输出：

```
Enter text on an arbitrary number of lines.
Enter a line containing just an asterisk to end input:

When I makes tea I makes tea, as old mother Grogan said.
And when I makes water I makes water.
Begob, ma'am, says Mrs Cahill, God send you don't make them in the same
pot.
*

when          2 i           4 makes         4
tea           2 as          1 old           1
mother        1 grogan      1 said          1
and           1 water       2 begob         1
ma'am         1 says        1 mrs           1
cahill        1 god         1 send          1
you           1 don't       1 make          1
them          1 in          1 the           1
same          1 pot         1
```

www.TopSage.com

6.9 小结

本章应用前几章所学的技术处理字符串的一般问题。字符串的问题和数值数据类型不同，或许稍微困难一些。

本章主要用数组处理字符串，也提到了指针。这些方法在处理字符串时更有弹性，下一章将介绍许多其他的方法。

6.10 习题

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容。还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 6.1　编写一个程序，从键盘上读入一个小于 1 000 的正整数，然后创建并输出一个字符串，说明该整数的值。例如，输入 941，程序产生的字符串是"Nine hundred and forty one"。

习题 6.2　编写一个程序，输入一系列单词，单词之间以逗号分隔，然后提取这些单词，并将它们分行输出，删除头尾的空格。例如，如果输入是

```
John  ,  Jack  ,  Jill
```

输出将是：

```
John
Jack
Jill
```

习题 6.3　编写一个程序，从一组至少有 5 个字符串的组里，输出任意挑选的一个字符串。

习题 6.4　回文是正读反读均相同的句子，忽略空白和标点符号。例如，“Madam，I' m Adam”和“Are we no drawn onward，we few? Drawn onward to new era?”都是回文。编写一个程序，确定从键盘输入的字符串是否是回文。

www.TopSage.com

www.TopSage.com

第 7 章

指　针

上一章已提到过指针，还给出使用指针的提示。本章深入探索这个主题，了解指针的功用。

本章将介绍许多新概念，所以可能需要多次重复某些内容。本章很长，需要花一些时间学习其内容，用一些例子体验指针。指针的基本概念很简单，但是可以应用它们解决复杂的问题。指针是用 C 语言高效编程的一个基本元素。

本章的主要内容：

- 指针的概念及用法
- 指针和数组的关系
- 如何将指针用于字符串
- 如何声明和使用指针数组
- 如何编写功能更强的计算器程序

7.1　指针初探

指针是 C 语言里最强大的工具之一，它也是最容易令人困惑的主题，所以一定要在开始时正确理解其概念，在深入探讨指针时，要对其操作有清楚的认识。

第 2 和第 5 章讨论内存时，谈到计算机如何为声明的变量分配一块内存。在程序中使用变量名引用这块内存，但是一旦编译执行程序，计算机就使用内存位置的地址来引用它。这是计算机用来引用“盒子(其中存储了变量值)”的值。

请看下面的语句：

```
int number =5
```

这条语句会分配一块内存来存储一个整数，使用 number 名称可以访问这个整数。值 5 存储在这个区域中。计算机用一个地址引用这个区域。存储这个数据的地址取决于所使用的计算机、操作系统和编译器。在源程序中，这个变量名是固定不变的，但地址在不同的系统上是不同的。

可以存储地址的变量称为指针(pointers)，存储在指针中的地址通常是另一个变量，如图 7-1 所示。指针 P 含有另一个变量 number 的地址，变量 number 是一个值为 5 的整数变量。存储在 P 中的地址是 number 第一个字节的地址。

www.TopSage.com

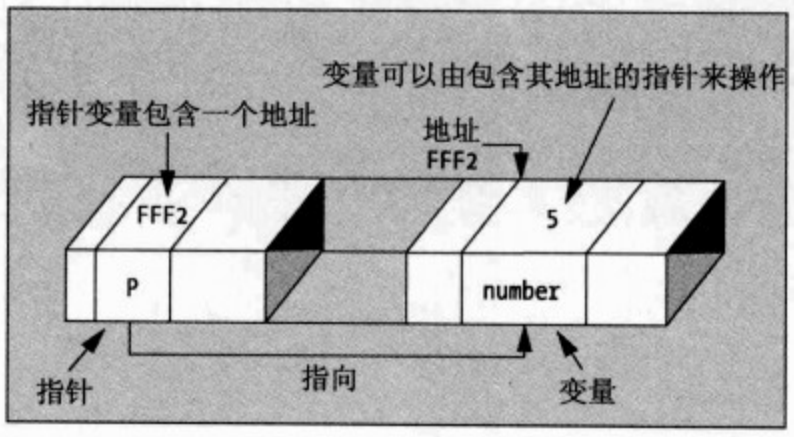


图 7-1 指针的工作原理

首先，知道变量 P 是一个指针是不够的，更重要的是，编译器必须知道它所指的变量类型。没有这个信息，根本不可能知道如何处理它所指的内存的内容。char 类型值的指针指向占有一个字节的值，而 long 类型值的指针通常指向占有 4 个字节的值。因此，每个指针都和某个变量类型相关联，也只能用于指向该类型的变量。所以如果指针的类型是整数，就只能指向 int 类型的变量，如果指针的类型是 float，就只能指向 float 类型的变量。一般给定类型的指针写做 type*，其中 type 是任意给定的类型。

类型名 void 表示没有指定类型，所以 void*类型的指针可以包含任意类型的数据项地址。类型 void*常常用做参数类型，或以独立于类型的方式处理数据的函数的返回值类型。任意类型的指针都可以传送为 void*类型的值，在使用它时，再将其转换为合适的类型。例如，int 类型变量的地址可以存储在 void*类型的指针变量中。要访问存储在 void*指针所指地址中的整数值，必须先把指针转换为 int *类型。本章后面介绍的 malloc()库函数返回 void*类型的指针。

7.1.1 声明指针

以下语句可以声明一个指向 int 类型变量的指针：

```
int  *pointer;
```

pointer 变量的类型是 int *，它可以存储任意 int 类型变量的地址。这条语句创建了 pointer，但没有初始化它。未初始化的指针是非常危险的，所以应总是在声明指针时对它进行初始化。重写刚才的声明，初始化 pointer，使它不指向任何对象：

```
int  *pointer  =  NULL;
```

NULL 是在标准库中定义的一个常量，对于指针它表示 0。NULL 是一个不指向任何内存位置的值。这表示，使用不指向任何对象的指针，不会意外覆盖内存。NULL 在头文件<stddef.h>、<stdlib.h>、<stdio.h>、<string.h>、<time.h>、<wchar.h>和<locale.h>中定义，必须在源文件中至少包含这些头文件中的一个，编译器才能识别 NULL。

如果用已声明的变量地址初始化 pointer 变量，可以使用寻址运算符&，例如：

www.TopSage.com

```
int  number   =   10;
int  *pointer  =   &number;
```

pointer 的初值是 number 变量的地址。注意，number 的声明必须在 pointer 的声明之前。否则，代码就不能编译。编译器需要先分配好空间，才能使用 number 的地址初始化 pointer 变量。

指针的声明没有什么特别之处。可以用相同的语句声明一般的变量和指针，例如：

```
double  value , *pVal, fnum;
```

这条语句声明了两个双精度浮点数变量value和fnum，以及一个指向double的变量pVal。从该语句中可以看出，只有第 2 个变量 pVal 是指针，考虑如下语句：

```
int *p, q;
```

上述语句声明了一个指针 p 和一个变量 q，两者都是 int 类型。把 p 和 q 都当做指针是一个很常见的错误。

7.1.2 通过指针访问值

使用间接运算符*可以访问指针所指的变量值。这个运算符也称为取消引用运算符(dereferencing operator)，因为它用于取消对指针的引用。假设声明以下的变量：

```
int  number  =  15;
int  *pointer =  &number;
int  result =  0;
```

pointer 变量含有 number 变量的地址，所以可以在表达式中使用它计算一个新的汇总值，如下：

```
result  =  *pointer  +5;
```

表达式*pointer 等于存储在 pointer 中的地址的值。这是存储在 number 中的值 15，所以 result 是 15+5，等于 20。

这完全符合理论。下面的小程序将凸显指针变量的某些特性。

试试看：声明指针

这个例子将声明一个变量和一个指针，然后给出它们的地址和它们所含的值。

```
/* Program 7.1 A simple program using pointers */
#include <stdio.h>

int main(void)
{
  int number = 0;        /* A variable of type int initialized to 0 */
  int *pointer = NULL;   /* A pointer that can point to type int */
```

www.TopSage.com

```
  number = 10;
  printf("\nnumber's address: %p", &number); /* Output the address */
  printf("\nnumber's value: %d\n\n", number); /* Output the value */

  pointer = &number;     /* Store the address of number in pointer */

  printf("pointer's address: %p", &pointer); /* Output the address */
  printf("\npointer's size: %d bytes", sizeof(pointer));
  /* Output the size */
  printf("\npointer's value: %p", pointer);
  /* Output the value (an address) */
  printf("\nvalue pointed to: %d\n", *pointer); /* Value at the address */
  return 0;
}
```

这个程序的输出如下所示。注意，实际地址在不同的计算机上是不同的。

```
number's address: 0012FEE4
number's value: 10

pointer's address: 0012FEE0
pointer's size: 4 bytes
pointer's value: 0012FEE4
value pointed to: 10
```

代码的说明

首先，声明一个 int 变量和一个指针:

```
int number = 0; /* A variable of type int initialized to 0 */
int *pointer = NULL; /* A pointer that can point to type int */
```

指针 pointer 是 int 类型指针。指针的声明和其他变量一样。声明指针 pointer 时，在变量名称前添加一个星号(*)。这个星号将 pointer 定义成一个指针，它的类型是 int，表示整数变量的指针。变量 pointer 的初值是 NULL，对于指针 NULL 表示 0——它没有指向任何对象。

声明之后，在变量 number 存储值 10，然后用以下语句输出它的地址和值:

```
number  = 10;
printf("\nnumber's address: %p", &number); /* Output the address */
printf("\nnumber's value: %d\n\n", number); /* Output the value */
```

要输出变量 number 的地址，应使用输出格式指定符%p，它以十六进制格式输出内存的地址。

下一条语句使用寻址运算符&获取变量 number 的地址，将该地址存储到 pointer 中:

```
pointer = &number; /* Store the address of number in pointer */
```

www.TopSage.com

注意，在 pointer 中只能存储地址。

接下来有 4 个 printf()语句，分别输出 pointer 的地址(pointer 所占的内存位置的第一个字节)、pointer 所占的字节数、存储在 pointer 的值(它是 number 的地址)，以及在 pointer 所含的地址内存储的值(它是存储在 number 中的值)。

为了解释清楚，下面逐行解释这些代码。第一条输出语句如下:

```
printf("pointer's address: %p", &pointer);
```

这条语句输出 pointer 的地址。指针本身也有一个地址，就像一般的变量一样。使用%p 作为转换指定符，以显示一个地址，然后用&(寻址)运算符引用 pointer 变量的地址。

接着，输出这个指针的字节数:

```
printf("\npointer's size: %d bytes", sizeof(pointer));
/* Output the size */
```

可以像其他变量一样，使用 sizeof 运算符获得指针所占的字节数，在某台机器上，一个指针占用 4 个字节，所以该机器上的内存地址是 32 位。

下一条语句输出存储在 pointer 中的值:

```
printf("\npointer's value: %p", pointer);
```

存储在 pointer 中的值是 number 的地址。因为这是一个地址，所以用%p 显示它，用变量名 pointer 访问这个地址值。

最后一条输出语句如下所示:

```
printf("\nvalue pointed to: %d\n", *pointer);
```

这里使用 pointer 访问存储在 number 中的值。*运算符的作用是访问存储在 pointer 中的地址的数据。使用%d 是因为它是一个整数。变量 pointer 存储 number 的地址，所以可以使用该地址访问存储在 number 中的数值。如前所述，*运算符称为间接运算符，有时也称为取消引用运算符。

所显示的地址在不同的机器上是不同的，在同一台机器上，如果程序的运行时间不同，所显示的地址也不相同。后者是因为程序不会每次都加载到相同的内存位置。number 和 pointer 的地址是变量在这台计算机上存放的地方。它们的值存储在该地址中。number 变量是一个整数(10)，但 pointer 变量是 number 的地址。使用*pointer 可以访问 number 的值，即间接地使用 number 变量的值。

间接运算符*也是乘的符号，编译器不会混淆它们。编译器会根据星号出现的位置确定它是间接运算符还是乘号。

图 7-2 说明了指针的用法。

www.TopSage.com

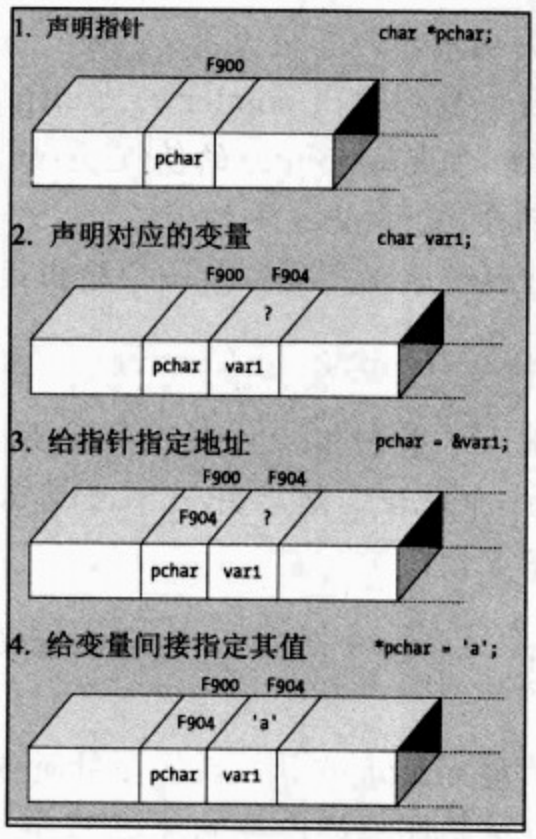


图 7-2 使用指针

7.1.3 使用指针

可以通过指针 pointer 访问 number 的内容，所以可以在算术语句中使用取消引用的指针，例如：

```
*pointer += 25;
```

上述语句将变量 pointer 所指向的地址中的值增加 25。星号*表示访问 pointer 变量所指向的内容。这里它是变量 number 的内容。

变量 pointer 能存储任何 int 变量的地址。这表示可以用下面的语句改变 pointer 指向的变量：

```
pointer = &another_number;
```

重复之前的语句：

```
*pointer += 25;
```

该语句操作的是新的变量 another_number。这表示指针可以包含同一类型的任意变量的地址，所以使用一个指针变量可以改变其他许多变量的值，只要它们的类型与指针相同。

试试看：使用指针

下面的例子使用指针递增存储在其他变量中的值。

```
/* Program 7.2 What's the pointer */
#include <stdio.h>
```

www.TopSage.com

```
int main(void)
{
  long num1 = 0L;
  long num2 = 0L;
  long *pnum = NULL;

  pnum = &num1;   /* Get address of num1 */
  *pnum = 2;      /* Set num1 to 2 */
  ++num2;         /* Increment num2 */
  num2 += *pnum;  /* Add num1 to num2 */

  pnum = &num2;   /* Get address of num2 */
  ++*pnum;        /* Increment num2 indirectly */

  printf("\nnum1 = %ld num2 = %ld *pnum = %ld *pnum + num2 = %ld\n",
         num1, num2, *pnum, *pnum + num2);
  return 0;
}
```

执行这个程序，会得到如下输出：

```
num1 = 2 num2 = 4 *pnum = 4 *pnum + num2 = 8
```

代码的说明

printf()后面的注释使这个程序比较容易理解。首先，在 main()函数体中有这些声明：

```
long  num1   =   0;
long  num2   =   0;
long  *pnum  =   NULL;
```

两个变量 num1 和 num2 的初值设置为 0。第三个语句声明了一个整数指针 pnum，它初始化为 NULL。

警告：

声明指针时，一定要初始化它们。使用未初始化的指针存储数据项是很危险的。在使用指针存储一个值时，谁也不知道会覆盖什么内容。

下一条语句是赋值：

```
pnum = &num1;     /* Get address of num1 */
```

指针 pnum 设定为指向 num1，因为该语句使用寻址运算符获取 num1 的地址，并将它保存在 pnum 中。

下两行语句是：

```
*pnum = 2;    /* Set num1 to 2 */
++num2;       /* Increment num2 */
```

www.TopSage.com

第一条语句利用了指针的新功能，为 pnum 取消引用，间接设定了 numl 的值 2。然后，变量 num2 以正常方式用递增运算符加 1。

之后的语句：

```
num2 += *pnum; /* Add num1 to num2 */
```

这条语句把 pnum 指向的变量内容加到 num2 上。pnum 仍指向 num1，所以给 num2 加上 num1 的值。

下两条语句是：

```
pnum = &num2;  /* Get address of num2 */
++*pnum;       /* Increment num2 indirectly */
```

首先，指针重新指向 num2。然后，通过指针间接地递增变量 num2。表达式++*pnum 递增了 pnum 指向的值。但如果要使用后置形式，必须写成(*pnum)++。括号很重要，它指定要递增的是数值，而不是地址。如果省略括号，就会递增 pnum 所含的地址。这是因为运算符++和一元运算符*(和一元运算符&)的优先级相同，且都是从右到左计算的。编译器会先给 pnum 应用运算符++，递增地址，然后取消对它的引用，得到它包含的值。这是一个通过指针递增数值的常见错误，所以最好在任何情况下都使用括号。

最后，在结束程序的 return 语句之前，有一条 printf()语句：

```
printf("\nnum1 = %ld num2 = %ld *pnum = %ld *pnum + num2 = %ld\n",
        num1, num2, *pnum, *pnum + num2);
```

它会显示 numl、num2、num2 通过 pnum 加 1 的结果，最后是以 pnum 形式出现的 num2 和 num2 的值之和。

第一次遇到指针时，很可能会弄不清楚。指针有多层意义，这就是混乱的根源。我们可以使用地址、数值、指针或变量，有时很难搞清楚到底是怎么回事。最好编写短一点的程序，使用指针得到数值，改变值，打印地址等。这是能有信心用好指针的唯一方法。

这里又一次提到运算符优先级的重要性。C 语言中所有运算符的优先级可参阅第 3 章的表 3-2，如果不清楚某个运算符的优先级，可以参阅该表。

下面的例子说明了指针如何用于键盘输入。

试试看：scanf()与指针的使用

前面使用 scanf()输入数值时，使用了&运算符获取传给函数的地址。有了一个含有地址的指针后，只需使用这个指针的名字作为参数。如下面的例子：

```
/* Program 7.3 Pointer argument to scanf */
#include <stdio.h>

int main(void)
{
  int value = 0;
  int *pvalue = NULL;
```

www.TopSage.com

```
  pvalue = &value;                    /* Set pointer to refer to value */

  printf ("Input an integer: ");
  scanf(" %d", pvalue);              /* Read into value via the pointer */

  printf("\nYou entered %d\n", value); /* Output the value entered */
  return 0;
}
```

这个程序只是输出了输入的信息。输出如下:

```
Input an integer:  10
You entered 10
```

代码的说明

scanf()语句中的每个参数都很清晰:

```
scanf(" %d", pvalue);
```

这条语句将用户输入的值存储到变量的地址中。就这个例子而言，可以使用&value。但是这里使用指针 pvalue 将 value 的地址传递给 scanf()。下面的赋值语句将 value 的地址存储到 pvalue 中:

```
pvalue = &value;              /* Set pointer to refer to value */
```

pvalue 和&value 是相同的，所以用任何一个都可以。

然后，显示 value:

```
printf("\nYou entered %d\n", value); /* Output the value entered */
```

这是一个没什么意义的例子，但它说明了指针和变量可以一起使用。

测试 NULL 指针

在上面的例子中，指针声明如下：

```
int  *pvalue  =  NULL;
```

这个语句用 NULL 初始化 pvalue。如前所述，NULL 在 C 语言中是一个特殊的常量，它是相当于数字 0 的指针。NULL 的定义包含在<stdio.h>和其他许多头文件中，所以如果使用它，一定要包含其中一个头文件。

给指针赋予 0 时，就等于将它设为 NULL，所以可以编写如下语句：

```
int  *pvalue = 0;
```

因为 NULL 等于 0，如果要测试指针 pvalue 是否为 NULL，可以编写如下语句：

```
if(!pvalue)
{
```

www.TopSage.com

```
  ...
}
```

pvalue 是 NULL，则!pvalue 就是 true，所以这段语句只有在 pvalue 是 NULL 时才会执行。也可以将这个测试写成如下语句：

```
if(pvalue ==  NULL)
{
 ...
}
```

7.1.4 指向常量的指针

声明指针时，可以使用 const 关键字指定，该指针指向的值不能改变。下面是声明 const 指针的例子：

```
long value = 9999L;
const long *pvalue = &value; /* Defines a pointer to a constant */
```

把 pvalue 指向的值声明为 const，所以编译器会检查是否有语句试图修改 pvalue 指向的值，并将这些语句标记为错误。例如，下面的语句就会让编译器生成一条错误信息：

```
*pvalue = 8888L; /* Error - attempt to change const location */
```

pvalue 指向的值不能改变，但可以对 value 进行任意操作。

```
value = 7777L;
```

改变了 pvalue 指向的值，但不能使用 pvalue 指针做这个改变。当然，指针本身不是常量，所以仍可以改变它指向的值：

```
long number = 8888L;
pvalue = &number;      /* OK - changing the address in pvalue */
```

这会改变指向 number 的 pvalue 中的地址，仍然不能使用指针改变它指向的值。可以改变指针中存储的地址，但不允许使用指针改变它指向的值。

7.1.5 常量指针

当然，也可以使指针中存储的地址不能改变。此时，在指针声明中使用 const 关键字的方式略有区别。下面的语句可以使指针总是指向相同的对象：

```
int count = 43;
int *const pcount = &count; /* Defines a constant */
```

第二条语句声明并初始化了 pnumber，指定该指针存储的地址不能改变。编译器会检查代码是否无意中把指针指向其他地方，所以下面的语句会在编译时生成一条错误信息：

www.TopSage.com

```
int item = 34;
pcount = &item; /* Error - attempt to change a constant pointer */
```

但使用 pcount，仍可以改变 pcount 指向的值：

```
*pcount = 345; /* OK - changes the value of count */
```

这条语句通过指针引用了存储在 count 中的值，并将其改为 345。还可以直接使用 count 改变这个值。

可以创建一个常量指针，它指向一个常量值：

```
int item = 25;
const int *const pitem = &item;
```

pitem 是一个指向常量的常量指针，所以所有的信息都是固定不变的。不能改变存储在 pitem 中的地址，也不能使用 pitem 改变它指向的内容。

7.1.6 指针的命名

我们已经开始编写相当大的程序了。程序越来越大，就越难记住哪个是一般变量，哪个是指针。因此，最好将 p 作为指针名的第一个字母。如果严格遵循这个命名方法，肯定很清楚哪个变量是指针。

7.2 数组和指针

下面复习一下什么是数组，什么是指针：

- 数组是相同类型的对象集合，可以用一个名称引用。例如，数组 scores[50]可以含有 50 场篮球季赛的比分。使用不同的索引值可以引用数组中的每个元素。scores[0]是第一个分数，scores[49]是最后一个分数。如果每个月有 10 场比赛，就可以使用多维数组 scores[12][10]。如果一月开始比赛，则五月的第 3 场比赛用 scores[5][2]引用。
- 指针是一个变量，它的值是给定类型的另一个变量或常量的地址。使用指针可以在不同的时间访问不同的变量，只要它们的类型相同即可。

数组和指针似乎完全不同，但它们有非常密切的关系，有时还可以互换。下面考虑字符串。字符串是 char 类型的数组。如果用 scanf()输入一个字符，可以使用如下语句：

```
char  single;
scanf("%c", &single);
```

这里，scanf()需要寻址运算符，因为 scanf()需要存储输入数据的地址。然而，如果读入字符串，可以编写如下代码：

```
char  multiple[10];
scanf("%s",  multiple);
```

www.TopSage.com

这里不需要使用&运算符，而使用了数组名称，就像指针一样。如果以这种方式使用数组名称，而没有带索引值，它就引用数组的第一个元素的地址。

但数组不是指针，它们有一个重要区别：可以改变指针包含的地址，但不能改变数组名称引用的地址。

下面通过几个例子来了解数组和指针如何一起使用。这些例子串在一起，构成一个完整的练习。通过这些练习，很容易掌握指针的基本概念及其和数组的关系。

试试看：数组和指针

这个例子进一步说明了，数组名称本身引用了一个地址，执行以下程序：

```
/* Program 7.4 Arrays and pointers - A simple program*/
#include <stdio.h>

int main(void)
{
  char multiple[] = "My string";

  char *p = &multiple[0];
  printf("\nThe address of the first array element : %p", p);

  p = multiple;
  printf("\nThe address obtained from the array name: %p\n", p);
  return 0;
}
```

在某台计算机上的输出如下所示：

```
The address of the first array element : 0x0013ff62
The address obtained from the array name: 0x0013ff62
```

代码的说明

可以从这个程序的输出中得到一个结论：&multiple[0]会产生和 multiple 表达式相同的值。这正是我们期望的，因为 multiple 等于数组第一个字节的地址，&multiple[0]等于数组第一个元素的第一个字节，如果它们不同，才令人惊讶。如果 p 设置为 multiple，而 multiple 的值与&multiple[0]相同，那么 p+1 等于什么。试试下面的例子。

试试看：数组和指针(续)

这个程序说明了将一个整数值加到指针上的结果：

```
/* Program 7.5 Arrays and pointers taken further */
#include <stdio.h>

int main(void)
{
  char multiple[] = "a string";
  char *p = multiple;

  for(int i = 0 ; i<strlen(multiple) ; i++)
```

www.TopSage.com

```
    printf("\nmultiple[%d] = %c *(p+%d) = %c &multiple[%d] = %p p+%d = %p",
          i, multiple[i], i, *(p+i), i, &multiple[i], i, p+i);
  return 0;
}
```

输出如下：

```
multiple[0] = a *(p+0) = a &multiple[0] = 0x0013ff63 p+0 = 0x0013ff63
multiple[1] =   *(p+1) = &multiple[1] = 0x0013ff64 p+1 = 0x0013ff64
multiple[2] = s *(p+2) = s &multiple[2] = 0x0013ff65 p+2 = 0x0013ff65
multiple[3] = t *(p+3) = t &multiple[3] = 0x0013ff66 p+3 = 0x0013ff66
multiple[4] = r *(p+4) = r &multiple[4] = 0x0013ff67 p+4 = 0x0013ff67
multiple[5] = i *(p+5) = i &multiple[5] = 0x0013ff68 p+5 = 0x0013ff68
multiple[6] = n *(p+6) = n &multiple[6] = 0x0013ff69 p+6 = 0x0013ff69
multiple[7] = g *(p+7) = g &multiple[7] = 0x0013ff6a p+7 = 0x0013ff6a
```

代码的说明

注意输出中右边的地址列表。p设置为multiple的地址，p+n就等于multiple+n，所以multiple[n]与*(multiple+n)是相同的。地址加上了1，对于元素占用一个字节的数组来说，这正是我们期望的。从输出的两列中可以看出，*(p+n)是给p中的地址加上整数n，再对得到的地址取消引用，就计算出了与multiple[n]相同的结果。

试试看：不同类型的数组

这很有趣，计算机可以将多个数字加在一起。下面改变数组的类型，看看会发生什么：

```
/* Program 7.6 Different types of arrays */
#include <stdio.h>

int main(void)
{
  long multiple[] = {15L, 25L, 35L, 45L};
  long * p = multiple;

  for(int i = 0 ; i<sizeof(multiple)/sizeof(multiple[0]) ; i++)
  printf("\naddress p+%d (&multiple[%d]): %d *(p+%d) value: %d",
         i, i, p+i, i, *(p+i));
  printf("\n Type long occupies: %d bytes\n", sizeof(long));
  return 0;
}
```

编译并执行这个程序，得到完全不同的结果：

```
address p+0 (&multiple[0]): 1310552 *(p+0) value: 15
address p+1 (&multiple[1]): 1310556 *(p+1) value: 25
address p+2 (&multiple[2]): 1310560 *(p+2) value: 35
address p+3 (&multiple[3]): 1310564 *(p+3) value: 45
Type long occupies: 4 bytes
```

www.TopSage.com

代码的说明

这里将 printf()函数的第二个参数及后面的参数用空格分隔开，以便于看出格式指定符和参数之间的对应关系。这次，指针 p 设置为 multiple 的地址，而 multiple 是 long 类型的数组。该指针最初包含数组中第一个字节的地址，也就是元素 multiple[0]的第一个字节。地址用指定符%d 显示，所以它们都是十进制值，这将易于看出后续地址的区别。

注意看输出。在这个例子中，p 是 1 310 552 ，p+1 是 1 310 556 ，而 1 310 556 比 1 310 552 大 4，但我们仅给 p 加上了 1。这并没有错。编译器知道，给地址值加 1 时，就表示要访问该类型的下一个变量。这就是为什么声明一个指针时，必须指定该指针指向的变量类型。char 类型存储在一个字节中，long 变量一般占用 4 个字节。在计算机上声明为 long 的变量占 4 个字节，给 long 类型的指针加 1，结果是给地址加 4，因为 long 类型值占 4 个字节。如果计算机在 8 个字节中存储 long 类型，则给指向 long 的指针加 1，会给地址值加 8。

注意，这个例子可以直接使用数组名称。编写 for 循环，如下所示:

```
for(int i = 0 ; i<sizeof(multiple)/sizeof(multiple[0]) ; i++)
  printf(
    "\naddress multiple+%d (&multiple[%d]): %d *(multiple+%d) value: %d",
    i, i, multiple+i, i, *(multiple+i));
```

这个循环可以执行，因为表达式 multiple 和 multiple+i 都等于一个地址。我们输出这些地址的值，再使用*运算符输出这些地址存储的值。地址的算术运算规则与指针 p 相同。给 multiple 加 1，会得到数组中下一个元素的地址，即内存中 multiple 后面的 4 个字节。但注意，数组名称是一个固定的地址，而不是一个指针。

7.3 多维数组

前面讨论的都是一维数组，它和二维或多维数组是否相同？它们在某种程度上是相同的。然而，指针和数组名称之间的差异变得更为明显。考虑第 5 章末尾在井字程序中使用的数组。数组声明如下：

```
char board[3][3] = {
                     {'1','2','3'},
                     {'4','5','6'},
                     {'7','8','9'}
                   };
```

本节的例子将使用这个数组，探讨多维数组和指针的关系。

试试看：使用二维数组

这个例子说明了地址和数组 board 的关系:

```
/* Program 7.7 Two-Dimensional arrays and pointers */
#include <stdio.h>
```

www.TopSage.com

```
}

int main(void)
{
  char board[3][3] = {
                       {'1','2','3'},
                       {'4','5','6'},
                       {'7','8','9'}
                     };
  printf("address of board         : %p\n", board);
  printf("address of board[0][0]   : %p\n", &board[0][0]);
  printf("but what is in board[0] : %p\n", board[0]);
  return 0;
}
```

输出如下:

```
address of board          : 0x0013ff67
address of board[0][0]    : 0x0013ff67
but what is in board[0]   : 0x0013ff67
```

代码的说明

可以看到，3 个输出值都是相同的，从中可以得到什么推论？答案相当简单：声明一维数组时，[n1]放在数组名称之后，告诉编译器它是一个有 n1 个元素的数组。声明二维数组时，在第一维[n1]的后面放置第二维[n2]，编译器就会创建一个大小为 n1 的数组，它的每个元素是一个大小为 n2 的数组。

如第 5 章所述，声明二维数组时，就是在创建一个数组的数组。因此，用数组名称和一个索引值访问这个二维数组时，例如 board[0]，就是在引用一个子数组的地址。仅使用二维数组名称，就是引用该二维数组的开始地址，它也是第一个子数组的开始地址。

总之，board、board[0]和&board[0][0] 的数值相同，但它们并不是相同的东西。

也就是说，表达式 board[1]和 board[1][0]的地址相同。这很容易理解，因为 board[1][0]是第二个子数组 board[1]的第一个元素。

但是，用指针记号获取数组中的值时，就会出问题。仍然必须使用间接运算符，但要非常小心。如果改变上面的例子，显示第一个元素的值，就知道原因了:

```
/* Program 7.7 A Two-Dimensional arrays */
#include <stdio.h>

int main(void)
{
  char board[3][3] = {
                       {'1','2','3'},
                       {'4','5','6'},
                       {'7','8','9'}
                     };

  printf("value of board[0][0] : %c\n", board[0][0]);
  printf("value of *board[0] :   %c\n", *board[0]);
```

www.TopSage.com

```
  printf("value of **board :      %c\n", **board);
  return 0;
}
```

这个程序的输出如下:

```
value of board[0][0] : 1
value of *board[0]   : 1
value of **board     : 1
```

可以看到，如果使用 board 获取第一个元素的值，就需使用两个间接运算符**board。前面的程序可以只使用一个*，是因为那是一维数组。如果只使用一个*，只会得到子数组的第一个元素，即 board[0]引用的地址。

多维数组和它的子数组之间的关系如图 7-3 所示。

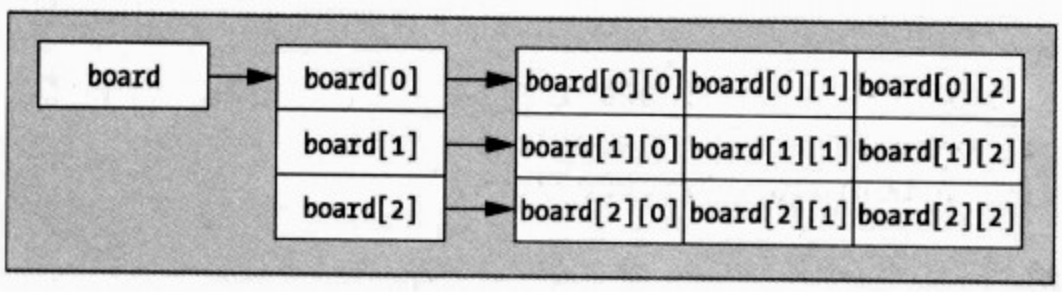


图 7-3 引用数组、其子数组和元素

如图 7-3 所示，board 引用子数组中第一个元素的地址，而 board[0]、board[1]和 board[2]引用对应子数组中第一个元素的地址。用两个索引值访问存储在数组元素中的值。明白了多维数组是怎么回事，下面看看如何使用 board 得到数组中的所有值。

试试看：得到二维数组中的所有值

这个例子用 for 循环进一步改进前一个例子:

```
/* Program 7.8 Getting the values in a two-dimensional array */
#include <stdio.h>

int main(void)
{
  char board[3][3] = {
                       {'1','2','3'},
                       {'4','5','6'},
                       {'7','8','9'}
                      };

  /* List all elements of the array */
  for(int i = 0; i < 9; i++)
    printf(" board: %c\n", *(*board + i));
  return 0;
}
```

程序的输出如下:

```
board:  1
```

www.TopSage.com

```
board:  2
board:  3
board:  4
board:  5
board:  6
board:  7
board:  8
board:  9
```

代码的说明

这个程序要注意在循环中取消引用 board 的方法:

```
printf(" board:  %c\n",  *(*board + i));
```

可以看到，使用表达式*(*board+i)可以得到一个数组元素的值。括号中的表达式 *board+i 会得到数组中偏移量为 i 的元素的地址。取消对它的引用，会得到这个地址中的值。括号在这里是很重要的。省略它们会得到 board 所指向的值(即存储在 board 中的地址所引用的值)再加上 i 的值。因此，如果 i 的值是 2，*board+i 会得到数组的第一个元素值加 2。我们真正想要的是将 i 的值加到 board 中的地址，然后对这个新地址取消引用，得到一个值。

下面去掉例子中的括号，看看会发生什么。改变数组的初值，使字符变成从'9'到'1'。如果去掉 printf()函数调用中表达式的括号:

```
printf(" board: %c\n", **board + i);
```

会得到如下输出:

```
board: 9
board: :
board: ;
board: <
board: =
board: >
board: ?
board: @
board: A
```

这是因为 i 的值加到数组 board 中的第一个元素上。在 ASCII 表中，得到的字符是从'9'到'A'。

另外，如果使用表达式**(board+i)，一样会导致错误的结果。此时，**(board+0)指向 board[0][0]，而**(board+1)指向 board[1][0]，**(board+2)指向 board[2][0]。如果增加的数值过大，就会访问数组以外的内存位置，因为这个数组没有第 4 个元素。

7.3.1 多维数组和指针

前面通过指针的表示法用数组名称引用二维数组，现在学习使用声明为指针的变量。

www.TopSage.com

如前所述，这有非常大的区别。如果声明一个指针，给它指定数组的地址，就可以用该指针访问数组的成员。

试试看：多维数组和指针

这个例子使用了多维数组和指针:

```
/* Program 7.9 Multidimensional arrays and pointers*/
#include <stdio.h>

int main(void)
{
  char board[3][3] = {
                       {'1','2','3'},
                       {'4','5','6'},
                       {'7','8','9'}
                      };
  char *pboard = *board; /* A pointer to char */

  for(int i = 0; i < 9; i++)
    printf(" board: %c\n", *(pboard + i));
  return 0;
}
```

输出和前一个例子相同:

```
board:   1
board:   2
board:   3
board:   4
board:   5
board:   6
board:   7
board:   8
board:   9
```

代码的说明

这里用数组中第一个元素的地址初始化指针，然后用一般的指针算术运算遍历整个数组:

```
char *pboard = *board;     /* A pointer to char */

for(int i = 0; i < 9; i++)
  printf(" board: %c\n", *(pboard + i));
```

注意，取消了对 board 的引用(*board)，得到了需要的地址，因为 board 是数组 board[0] 的地址，而不是一个元素的地址。可以用以下的方式初始化指针 pboard:

```
char *pboard = &board[0][0];
```

效果相同。用下面的语句初始化指针 pboard：

```
pboard = board;     /* Wrong level of indirection! */
```

这是错误的。如果这么做，至少会得到编译器的警告。严格地讲，这是不合法的，因为 pboard 和 board 有不同的间接级别。这个专业术语的意思是 pboard 指针引用的地址包含一个 char 类型的值，而 board 引用一个地址，那个地址引用另一个含有 char 类型值的地址。board 比 pboard 多了一级。因此，pboard 指针需要一个*，以获得地址中的值，而 board 需要两个*。一些编译器允许这么用，但是会给出一条警告信息。然而，这是很糟的用法，不应这么用!

7.3.2 访问数组元素

可以使用几种方法访问二维数组的元素。表 7-1 列出了访问 board 数组的方法。最左列包含 board 数组的行索引值，最上面的一行包含列索引值。表中对应于给定行索引和列索引的项列出了引用该元素的各种表达式。

表 7-1　访问数组元素的指针表达式

board	0	1	2
0	board[0][0] *board[0] **board	board[0][1] *(board[0]+1) *(*board+1)	board[0][2] *(board[0]+2) *(*board+2)
1	board[1][0] *(board[0]+3) *board[1] *(*board+3)	board[1][1] *(board[0]+4) *(board[1]+1) *(*board+4)	board[1][2] *(board[0]+5) *(board[1]+2) *(*board+5)
2	board[2][0] *(board[0]+6) *(board[1]+3) *board[2] *(*board+6)	board[2][1] *(board[0]+7) *(board[1]+4) *(board[2]+1) *(*board+7)	board[2][2] *(board[0]+8) *(board[1]+5) *(board[2]+2) *(*board+8)

下面看看如何把前面所学的指针知识应用于前面没有使用指针编写的程序中，然后就可以看出基于指针的实现方式有什么不同了。第 5 章编写了一个计算帽子尺寸的例子，下面用另一种方式完成这个例子。

试试看：帽子尺寸的另一种计算方法

使用指针表示法重写帽子尺寸的例子：

```
/* Program 7.10 Understand pointers to your hat size - if you dare */
```

www.TopSage.com

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

int main(void)
{
  char size[3][12] = {                    /* Hat sizes as characters */
      {'6', '6', '6', '6', '7', '7', '7', '7', '7', '7', '7', '7'},
      {'1', '5', '3', '7', ' ', '1', '1', '3', '1', '5', '3', '7'},
      {'2', '8', '4', '8', ' ', '8', '4', '8', '2', '8', '4', '8'}
                    };

  int headsize[12] =                              /* Values in 1/8 inches */
     {164,166,169,172,175,178,181,184,188,191,194,197};

  char *psize = *size;
  int *pheadsize = headsize;

  float cranium = 0.0;      /* Head circumference in decimal inches */
  int your_head = 0;        /* Headsize in whole eighths */
  bool hat_found = false;   /* Indicates when a hat is found to fit */
  bool too_small = false;   /* Indicates headsize is too small */

  /* Get the circumference of the head */
  printf("\nEnter the circumference of your head above your eyebrows"
      " in inches as a decimal value: ");
  scanf(" %f", &cranium);
  /* Convert to whole eighths of an inch */
  your_head = (int)(8.0*cranium);
  /* Search for a hat size */
  for(int i = 0 ; i < 12 ; i++)
  {
    /* Find head size in the headsize array */
    if(your_head > *(pheadsize+i))
      continue;

    /* If it is the first element and the head size is */
    /* more than 1/8 smaller then the head is too small */
    /* for a hat */
    if((i == 0) && (your_head < (*pheadsize)-1))
    {
      printf("\nYou are the proverbial pinhead. No hat for"
             "you I'm afraid.\n");
      too_small = true;
      break;              /* Exit the loop */
    }

    /* If head size is more than 1/8 smaller than the current */
    /* element in headsize array, take the next element down */
    /* as the head size */
```

www.TopSage.com

```
      if( your_head < *(pheadsize+i)-1)
      i--;

      printf("\nYour hat size is %c %c%c%c\n",
          *(psize + i),          /* First row of size */
          *(psize + 1*12 + i), /* Second row of size */
          (i==4) ?' ' : '/',
          *(psize+2*12+i));     /* Third row of size */
      hat_found=true;
      break;
    }
    if(!hat_found && !too_small)
      printf("\nYou, in technical parlance, are a fathead."
            " No hat for you, I'm afraid.\n");
    return 0;
}
```

这个程序的输出和第 5 章相同，所以不再重复。这里关心的是代码本身，下面看看这个程序的新元素。

代码的说明

这个程序的执行过程和第 5 章相同。其区别是这个实现代码使用了指针 pheadsize 和 psize，它们分别包含 headsize 数组和 size 数组的开始地址。your_head 的值和数组的值用下面的语句作比较:

```
if(your_head > *(pheadsize+i))
  continue;
```

比较运算符右侧的表达式*(pheadsize+i)等于数组表示法中的 headsize[i]。括号内的表达式把 i 加到数组开始的地址上。给地址加一个整数值，会给该地址加上元素长度的 i 倍值。因此，括号内的子表达式会产生对应于索引值为 i 的元素的地址。然后，使用取消引用运算符*，得到这个元素的内容，将它和变量 your_head 的值进行比较。

如果在中间执行 printf (), 可以看到访问某行一个元素的指针表达式对二维数组的执行结果:

```
printf("\nYour hat size is %c %c%c%c\n",
    *(psize + i),          /* First row of size */
    *(psize + 1*12 + i), /* Second row of size */
    (i==4) ?' ' : '/',
    *(psize+2*12+i));     /* Third row of size */
```

第一个表达式是*(psize+i)，它访问 size 数组中第一行的第 i 个元素，等于 size[0][i]。第二个表达式是*(psize + 1*12 + i)，它访问 size 数组中第二行的第 i 个元素，等于 size[1][i]。这个表达式说明了第二行的开始位置可以通过给 psize 加上行的大小来得到。接着给该结果加上 i，就得到了第二行中的元素。要得到 size 数组中第三行的元素，可以使用表达式*(psize+2*12+i)，它等于 size[2][i]。

www.TopSage.com

7.4 内存的使用

指针是一个非常灵活且强大的编程工具，有非常广泛的应用。大多数 C 程序都在某种程度上使用了指针。C 语言还进一步增强了指针的功能，为在代码中使用指针提供了很强的激励机制，它允许在执行程序时动态分配内存。只有使用指针，才能动态分配内存。

第 5 章的一个程序计算一组学生的平均分，当时它只处理 10 个学生。假设要编写一个程序，但事先不知道要处理多少个学生，若使用动态内存分配(dynamic memory allocation)，所使用的内存就不会比指定的学生分数所需的内存多。可以在执行时创建足以容纳所需数据量的数组。

在程序的执行期间分配内存时，内存区域中的这个空间称为堆(heap)。还有另一个内存区域，称为堆栈(stack)，其中的空间分配给函数的参数和本地变量。在执行完该函数后，存储参数和本地变量的内存空间就会释放。堆中的内存是由程序员控制的。如本章后面所述，在分配堆上的内存时，由程序员跟踪所分配的内存何时不再需要，并释放这些空间，以便于以后重用它们。

7.4.1 动态内存分配：malloc()函数

在运行时分配内存的最简单的标准库函数是 malloc()。使用这个函数时，需要在程序中包含头文件<stdlib.h>。使用 malloc()函数需指定要分配的内存字节数作为参数。这个函数返回所分配内存的第一个字节的地址。因为返回的是一个地址，所以这里可以使用指针。

动态内存分配的一个例子如下：

```
int *pNumber = (int *)malloc(100);
```

这条语句请求 100 个字节的内存，并把这个内存块的地址赋予 pNumber。只要不修改它，任何时间使用这个变量 pNumber，它都会指向所分配的 100 个字节的第一个 int 的位置。这个内存块能保存 25 个 int 值，每个 int 占 4 个字节。

注意，类型转换(int*)将函数返回的地址转换成 int 类型的指针。这么做是因为 malloc()是一般用途的函数，可为任何类型的数据分配内存。这个函数不知道要这个内存作什么用，所以它返回的是一个 void 类型的指针，写做 void*。类型 void*的指针可以指向任意类型的数据，然而不能取消对 void 指针的引用，因为它指向未具体说明的对象。许多编译器会把 malloc()返回的地址自动转换成适当的类型，且不会伤害具体指定的对象。

可以请求任意数量的字节，字节数仅受制于计算机中未用的内存以及 malloc()的运用场合。如果因某种原因而不能分配请求的内存，malloc()会返回一个 NULL 指针。这个指针等于 0。最好先用 if 语句检查请求动态分配的内存是否已分配，再使用它。就如同金钱，没钱又想花费，会带来灾难性的后果。因此，应编写如下语句：

```
if(pNumber == NULL)
```

www.TopSage.com

```
{
  /*Code  to  deal  with  no  memory  allocated  */
}
```

如果指针是 NULL，最好执行适当的操作。例如，至少可以显示一条信息“内存不足”，然后中止程序。这比允许程序继续执行，使之使用 NULL 地址存储数据导致崩溃要好得多。然而，在某些情况下，可以释放在别的地方使用的内存，以便程序有足够的内存继续执行下去。

7.4.2 分配内存时使用 sizeof 运算符

前一个例子很不错，但我们不常处理字节，而常常处理 int、double 等数据类型。例如给 75 个 int 类型的数据项分配内存，可以使用以下的语句：

```
pNumber = (int *) malloc(75*sizeof(int));
```

如前所述，sizeof 是一个运算符，它返回一个 size_t 类型的无符号整数，该整数是存储它的参数需要的字节数。它把关键字如 int 或 float 等作为参数，返回存储该类型的数据项所需的字节数。它的参数也可以是变量或数组名。把数组名作为参数时，sizeof 返回存储整个数组所需的字节数。前一个例子请求分配足以存储 75 个 int 数据项的内存。以这种方式使用 sizeof，可以根据不同的 C 编译器为 int 类型的值自动调整所需的内存空间。

试试看：动态内存分配

下面使用指针来计算质数，将动态内存分配的概念应用于实践。质数是只能被 1 和这个数本身整除的整数。

查找质数的过程非常简单。首先，由观察得知，2、3 和 5 是前三个质数，因为它们不能被除了 1 以外更小的数整除。其他质数必定都是奇数(否则它们可以被 2 整除)，所以要找出下一个质数，可以从最后一个质数开始，给它加 2。检查完这个数后，再给它加 2，继续检查。

检查一个数是否为质数，而不只是奇数，可以用这个数除以比它小的所有奇数。其实不需要这么麻烦。如果一个数不是质数，它必定能被比它小的质数整除。我们要按顺序查找质数，所以可以把已经找到的质数作为除数，确定所检查的数是否为质数。

这个程序将使用指针和动态内存分配：

```
/* Program 7.11 A dynamic prime example */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main(void)
{
  unsigned long *primes = NULL; /* Pointer to primes storage area */
  unsigned long trial = 0;      /* Integer to be tested */
```

www.TopSage.com

```
bool found = false;              /* Indicates when we find a prime */
size_t total = 0;                /* Number of primes required */
size_t count = 0;                /* Number of primes found */

printf("How many primes would you like - you'll get at least 4? ");
scanf("%u", &total); /* Total is how many we need to find */
total = total<4U ? 4U:total; /* Make sure it is at least 4 */

/* Allocate sufficient memory to store the number of primes required */
primes = (unsigned long *)malloc(total*sizeof(unsigned long));
if(primes == NULL)
{
  printf("\nNot enough memory. Hasta la Vista, baby.\n");
  return 1;
}

/* We know the first three primes */
/* so let's give the program a start. */
*primes = 2UL;       /* First prime */
*(primes+1) = 3UL;   /* Second prime */
*(primes+2) = 5UL;   /* Third prime */
count = 3U;          /* Number of primes stored */
trial = 5U;          /* Set to the last prime we have */

/* Find all the primes required */
while(count<total)
{
  trial += 2UL;                  /* Next value for checking */

  /* Try dividing by each of the primes we have */
  /* If any divide exactly - the number is not prime */
  for(size_t i = 0 ; i < count ; i++)
    if(!(found = (trial % *(primes+i))))
      break; /* Exit if no remainder */

  if(found) /* we got one - if found is true */
    *(primes+count++) = trial; /* Store it and increment count */
}

/* Display primes 5-up */
for(size_t i = 0 ; i < total ; i ++)
{
  if(!(i%5U))
    printf("\n");                /* Newline after every 5 */
  printf ("%12lu", *(primes+i));
}
printf("\n");                    /* Newline for any stragglers */
return 0;
}
```

www.TopSage.com

程序的输出如下：

```
How many primes would you like - you'll get at least 4? 25

 2     3     5     7    11
13    17    19    23    29
31    37    41    43    47
53    59    61    67    71
73    79    83    89    97
```

代码的说明

在这个例子中，可以输入要程序产生的质数个数。指针变量 primes 引用一块用于存储所计算的质数的内存区。然而，在程序中没有一开始就定义内存。这块空间是在输入质数个数后分配的：

```
printf("How many primes would you like - you'll get at least 4? ");
scanf("%u", &total); /* Total is how many we need to find */
total = total<4U ? 4U:total; /* Make sure it is at least 4 */
```

在提示后，输入的值存储在 total 中。下一行语句确保 total 至少是 4。这是因为程序将定义并存储已知的前三个质数。

然后，使用 total 的值分配适当数量的内存来存储质数：

```
primes = (unsigned long *)malloc(total*sizeof(unsigned long));
if(primes == NULL)
{
  printf("\nNot enough memory. Hasta la Vista, baby.\n");
  return 1;
}
```

质数的大小增长得比其数量快，所以把它们存储在 unsigned long 类型中。但如果要指定可以处理的最大质数，可以使用 unsigned long long 类型。程序把每个质数存储为类型 long，所以需要的字节数是 total*sizeof(unsigned long)。如果 malloc()函数返回 NULL，就不分配内存，而是显示一条信息，并结束程序。

可以指定最大的质数个数取决于计算机的可用内存和编译器使用 malloc()一次能分配的内存量，前者是主要的限制。malloc()函数的参数是 size_t 类型，所以 size_t 对应的整数类型限制了可以指定的字节数。如果 size_t 对应 4 字节的无符号整数，则一次至多可以分配 4 294 967 295 个字节。

一旦有了分配给质数的内存，就定义前三个质数，将它们存储到 primes 指针指向的内存区的前三个位置：

```
*primes = 2UL;      /* First prime */
*(primes+1) = 3UL;  /* Second prime */
*(primes+2) = 5UL;  /* Third prime */
```

可以看到，引用连续的内存位置是很简单的。primes 是 unsigned long 类型的指针，

www.TopSage.com

所以 primes+1 引用第二个位置的地址——这个地址是 primes 加上存储一个 unsigned long 类型数据项所需的字节数。使用间接运算符存储每个值；否则就要修改这个地址本身。

有了三个质数，就把 count 变量设定为 3，用最后一个质数 5 初始化变量 trial:

```
count = 3U; /* Number of primes stored */
trial = 5U; /* Set to the last prime we have */
```

开始查找下一个质数时，给 trial 中的值加 2，得到下一个要测试的数。所有的质数都在 while 循环内查找:

```
while(count<total)
{
 ...
}
```

在循环内每找到一个质数，就递增 count 变量，当它到达 total 值时，循环就结束。

在 while 循环内，首先将 trial 的值加 2UL，然后测试它是否是质数:

```
trial += 2UL; /* Next value for checking */

/* Try dividing by each of the primes we have */
/* If any divide exactly - the number is not prime */
for(size_t i = 0 ; i < count ; i++)
  if(!(found = (trial % *(primes+i))))
    break; /* Exit if no remainder */
```

for 循环用于测试。在这个循环内，把 trial 除以每个质数的余数存放到 found 中。如果除尽，余数就是 0，因此 found 设置为 false。如果余数是 0，就表示 trial 中的值不是质数，可以继续测试下一个数。

赋值表达式的值存储到赋值运算符左边的变量中。因此，表达式(found= (trial % *(primes+i))) 的结果存储到 found 中。如果除尽，found 就是 false，表达式!(found=(trial % *(primes+i)))将是 true，执行 break 语句。因此，如果 trial 能整除任一个先前存储的质数，for 循环就会结束。

如果没有一个质数除 trial 是整除，当所有的质数都试过后，就结束 for 循环，found 的结果是把最后一个余数(它是某个正整数)转换为 bool 类型的值。如果 trial 能被某个质数整除，循环会通过 break 语句结束，found 会含有 false。因此，可以在完成 for 循环时，使用存储在 found 中的值确定是否找到一个新的质数：

```
if(found)                          /* we got one - if found is true */
  *(primes+count++) = trial;  /* Store it and increment count */
```

如果 found 是 true，就将 trial 的值存储到内存区的下一个位置上。下一个位置的地址是 primes+count。第一个位置是 primes，所以当有 count 个质数时，最后一个质数所占的位置是 primes+count - 1。这个语句存储了新的质数后，递增 count 的值。

while 循环重复这个过程，直到找出所有的质数为止。然后，以 5 个一行输出质数：

www.TopSage.com

```
for(size_t i = 0 ; i < total ; i ++)
{
  if(!(i%5U))
    printf("\n"); /* Newline after every 5 */
  printf ("%12lu", *(primes+i));
}
printf("\n");      /* Newline for any stragglers */
```

for 循环会输出 total 个质数。printf()函数在当前行上显示每个质数，但 if 语句在 5 次迭代后输出一个换行符，所以每行显示 5 个质数。因为质数的个数不会刚好是 5 的倍数，所以在结束循环后，输出一个换行符，以确保在输出的最后至少有一个换行符。

7.4.3 用 calloc()函数分配内存

在<stdlib.h>头文件中声明的 calloc()函数与 malloc()函数相比有两个优点。第一，它把内存分配为给定大小的数组，第二，它初始化了所分配的内存，所有的位都是 0。calloc()函数需要两个参数：数组的元素个数和数组元素占用的字节数，这两个参数的类型都是 size_t。该函数也不知道数组元素的类型，所以所分配区域的地址返回为 void *类型。

下面的语句使用 calloc()为包含 75 个 int 元素的数组分配内存：

```
int *pNumber = (int *) calloc(75, sizeof(int));
```

如果不能分配所请求的内存，返回值就是 NULL，也可以检查分配内存的结果，这非常类似于 malloc()，但 calloc()分配的内存区域都会初始化为 0。

将程序 7.11 改为使用 calloc()代替 malloc()来分配需要的内存，只需修改一条语句，如下面的粗体显示，其他代码不变：

```
/* Allocate sufficient memory to store the number of primes required */
primes = (unsigned long *)calloc(total, sizeof(unsigned long));
if (primes == NULL)
{
  printf("\nNot enough memory. Hasta la Vista, baby.\n");
  return 1;
}
```

7.4.4 释放动态分配的内存

在动态分配内存时，应总是在不需要该内存时释放它们。堆上分配的内存会在程序结束时自动释放，但最好在使用完这些内存后立即释放，甚至是在退出程序之前，也应立即释放。在比较复杂的情况下，很容易出现内存泄漏。当动态分配了一些内存时，没有保留对它们的引用，就会出现内存泄漏，此时无法释放内存。这常常发生在循环内部，由于没有释放不再需要的内存，程序会使用越来越多的内存，最终占用所有内存。

当然，要释放用 malloc()或 calloc()分配的内存，必须使用函数返回的引用内存块的地址。要释放动态分配的内存，而该内存的地址存储在 pNumber 指针中，可以使用下面

www.TopSage.com

的语句：

```
free(pNumber);
```

free()函数的形参是 void *类型，所有指针类型都可以自动转换为这个类型，所以可以把任意类型的指针作为参数传送给这个函数。只要 pNumber 包含分配内存时 malloc()或 calloc()返回的地址，就会释放所分配的整个内存块，以备以后使用。

如果给 free()函数传送一个空指针，该函数就什么也不做。应避免两次释放相同的内存区域，因为在这种情况下，free()函数的操作是不确定的，因此也就无法预料。如果多个指针变量引用已分配的内存，就有可能两次释放相同的内存，所以要特别小心。

下面修改前面的例子，使用 calloc()，并在程序的最后释放内存。

试试看：释放动态分配的内存

这个程序使用指针和动态分配的内存：

```
/* Program 7.11A Allocating and freeing memory */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main(void)
{
  unsigned long *primes = NULL; /* Pointer to primes storage area */
  unsigned long trial = 0;      /* Integer to be tested */

  bool found = false;           /* Indicates when we find a prime */
  size_t total = 0;             /* Number of primes required */
  size_t count = 0;             /* Number of primes found */

  printf("How many primes would you like - you'll get at least 4? ");
  scanf("%u", &total); /* Total is how many we need to find */
  total = total<4U ? 4U:total; /* Make sure it is at least 4 */

  /* Allocate sufficient memory to store the number of primes required */
  primes = (unsigned long *)calloc(total, sizeof(unsigned long));
  if (primes == NULL)
  {
    printf("\nNot enough memory. Hasta la Vista, baby.\n");
    return 1;
  }

  /* Code to determine the primes as before...*/
  /* Display primes 5-up */
  for(int i = 0 ; i < total ; i ++)
  {
    if(!(i%5U))
      printf("\n"); /* Newline after every 5 */
    printf ("%12lu", *(primes+i));
```

www.TopSage.com

```
  }
  printf("\n"); /* Newline for any stragglers */

  free(primes); /* Release the memory */
  return 0;
}
```

如果输入相同，这个程序的输出与上一个版本相同。只要两行粗体显示的代码与上一个版本不同。程序现在使用 calloc()来分配内存，该函数的第一个参数是 long 类型的字节数，第二个参数是 total，即需要的质数个数。在结束程序的 return 语句之前，用 primes 作为参数调用 free()函数，释放了分配的内存。

7.4.5 重新分配内存

realloc()函数可以重用前面通过 malloc()或 calloc() (或 realloc())分配的内存。函数需要两个参数：一个是指针，它包含前面调用 malloc()、calloc()或 realloc()返回的地址，另一个是要分配的新内存的字节数。

realloc()函数释放第一个指针参数引用的之前分配的内存，然后重新分配该内存区域，以满足第二个参数指定的新请求。显然，第二个参数的值不应超过以前分配的字节数。否则，新分配的内存将与以前分配的内存区域大小相同。

下面的代码演示了如何使用 realloc()函数：

```
long *pData = NULL;     /* Stores the data */
size_t count = 0;       /* Number of data items */
size_t oldCount = 0;    /* previous count value */
while(true)
{
  oldCount = count;     /* Save previous count value */
  printf("How many values would you like? ");
  scanf("%u", &count); /* Total is how many we need to find */

  if(count == 0)        /* If none required, we are done */
  {
    if(!pData)          /* If memory is allocated */
      free(pData);      /* release it */
    break;              /* Exit the loop */
  }

  /* Allocate sufficient memory to store count values */
  if((pData && (count <= oldCount) /* If there's big enough old memory... */
    pData = (long *)realloc(pData, sizeof(long)*count); /* reallocate it. */
  else
    {                   /* There wasn't enough old memory */
    if(pData)           /* If there's old memory... */
      free(pData);      /* release it. */
```

www.TopSage.com

```
      /* Allocate a new block of memory */
      pData = (long *)calloc(count, sizeof(long));
    }
    if (pData == NULL)   /* If no memory was allocated... */
    {
      printf("\nNot enough memory.\n");
      return 1;          /* abandon ship! */
    }
    /* Read and process the data and output the result... */
  }
```

很容易通过注释理解这段代码。循环读取任意个由用户提供的数据项，如果以前分配过内存空间，且该空间足以满足新请求，就再次使用该空间。如果以前没有分配过内存空间，或空间不够大，代码就使用 calloc()分配一块新内存。

从这段代码中可以看出，重新分配内存需要做许多工作，因为一般需要确保已有的内存块足以满足新请求。在大多数情况下，最好明确释放旧内存块，再分配一块全新的内存。

下面是使用动态分配的内存的基本规则：

- 避免分配大量的小内存块。分配堆上的内存有一些系统开销，所以分配许多小的内存块比分配几个大内存块的系统开销大。
- 仅在需要时分配内存。只要使用完堆上的内存块，就释放它。
- 总是确保释放已分配的内存。在编写分配内存的代码时，就要确定在代码的什么地方释放内存。
- 在释放内存之前，确保不会无意中覆盖堆上分配的内存的地址，否则程序就会出现内存泄漏。在循环中分配内存时，要特别小心。

7.5 使用指针处理字符串

前面使用 char 类型的数组变量存储字符串，也可以使用 char 类型的指针变量引用字符串。这个方法在处理字符串时非常灵活。下面的语句声明了一个 char 类型的指针变量：

```
char  *pString = NULL;
```

注意，指针只是一个存储另一个内存位置的地址的变量。前面只创建了指针，没有指定一个存储字符串的地方。要存储字符串，需要分配一些内存。可以声明一块内存，来存储字符串数据，然后使用指针追踪这块存储字符串的内存。

7.5.1 更多地控制字符串输入

在读取文本时，常需要比 scanf()函数更多的控制。在<stdio.h>中声明的 getchar()函数提供了非常基本的操作，一次只读取一个字符，但是它可控制何时停止读入字符。这样，就可以确保不会超过为存储输入而分配的内存。

www.TopSage.com

getchar()函数从键盘读入一个字符，并以 int 类型返回。可以把一个结尾是'\n'的字符串读入数组 buffer，如下：

```
char buffer[100];          /* String input buffer */
char *pbuffer = buffer;  /* Pointer to buffer */
while((*pbuffer++ = getchar()) != '\n');

*pbuffer = '\0';           /* Add null terminator */
```

所有的输入都在 while 循环的条件式中完成。getchar()函数读取一个字符，并将它存储在 pbuffer 的当前地址中。然后，递增 pbuffer 中的地址，以指向下一个字符。赋值表达式(*pbuffer++ = getchar())的值存储在这个操作中。只要存储的字符不是'\n'，循环就会继续。循环结束后，将'\0'字符添加到下一个可用的位置上。注意，这将'\n'字符当作字符串的一部分。如果不希望这么做，则可以调整存储'\0'的地址，覆盖掉'\n'。

输入的内容有可能超过数组可用的 100 个字节，所以这只能安全地用于数组足够大的情况下。然而，可以重写循环，检查该条件：

```
size_t index = 0;
for(; index<sizeof(buffer) ; i++)
  if((*(pbuffer+index) = getchar()) == '\n')
  {
    *(pbuffer + index++) = '\0';
    break;
  }
if( (index ==sizeof(buffer) &&( (*(pbuffer+index-1) != '\0) )
{
  printf("\nYou ran out of space in the buffer.");
  return 1;
}
```

index 变量指向 buffer 数组的下一个可用元素。读取操作现在在 for 循环中进行，当到达 buffer 数组的尾部，或读取并存储了'\n'字符，就停止该循环。在循环中，'\n'字符用'\0'字符替换。注意，index 在存储了'\0'后递增。这就可以确保 index 仍引用 buffer 数组的下一个可用元素。当然，如果填满了 buffer 数组，就超出了数组的最后一个元素。

循环结束时，必须确定其原因：可能因为字符串已读取完，也可能因为用尽了 buffer 数组的空间。在用尽了 buffer 数组的空间时，index 等于 buffer 数组的元素个数，buffer 数组的最后一个元素不是终止字符。因此，如果填满了 buffer，if 表达式中&&操作的左操作数就是 true，如果 buffer 数组的最后一个元素不是终止字符，该操作的右操作数就是 true。我们读取的字符串可能刚好填满 buffer 数组，此时最后一个元素是终止字符，if 表达式应是 false。

7.5.2 使用指针数组

当然，处理多个字符串时，可以在堆上使用指针数组存储对字符串的引用。假定从

www.TopSage.com

键盘上读取 3 个字符串，将它们存储在 buffer 数组中。可以创建一个指针数组，存储这 3 个字符串的位置：

```
char *pS[3] = { NULL };
```

这条语句声明了一个数组 pS，它包含 3 个指针。第 5 章提到，如果在数组初始化列表中提供的初始值个数少于数组的元素个数，剩下的元素就初始化为 0。因此，上述语句的初始化列表中只有一个值 NULL，它将任意大小的指针数组中的所有元素都初始化为 NULL。

下面看一个例子。

试试看：指针数组

下面的例子重写了前一个程序，说明了如何使用数组指针达到相同的目的：

```
/* Program 7.12 Arrays of Pointers to Strings */
#include <stdio.h>
const size_t BUFFER_LEN = 512; /* Size of input buffer */

int main(void)
{
  char buffer[BUFFER_LEN];      /* Store for strings */
  char *pS[3] = { NULL };       /* Array of string pointers */
  char *pbuffer = buffer;       /* Pointer to buffer */
  size_t index = 0;             /* Available buffer position*/

  printf("\nEnter 3 messages that total less than %u characters.",
      BUFFER_LEN-2);

  /* Read the strings from the keyboard */
  for(int i=0 ; i<3 ; i++)
  {
    printf("\nEnter %s message\n", i>0? "another" : "a" );
    pS[i] = &buffer[index];             /* Save start of string */
    /* Read up to the end of buffer if necessary */
    for( ; index<BUFFER_LEN ; index++) /* If you read \n ... */
      if((*(pbuffer+index) = getchar()) == '\n')
      {
        *(pbuffer+index++) = '\0';      /* ...substitute \0 */
        break;
      }

      /* Check for buffer capacity exceeded */
      if((index == BUFFER_LEN) && ((*(pbuffer+index-1) != '\0') || (i<2)))
      {
        printf("\nYou ran out of space in the buffer.");
        return 1;
    }
  }
```

www.TopSage.com

```
  printf("\nThe strings you entered are:\n\n");
  for(int i = 0 ; i<3 ; i++)
    printf("%s\n", pS[i]);

  printf("The buffer has %d characters unused.\n",
      BUFFER_LEN-index);
  return 0;
}
```

这个程序的输出如下:

```
Enter a message
Hello World!

Enter another message
Today is a great day for learning about pointers.

Enter another message
That's all.

The strings you entered are:
Hello World!
Today is a great day for learning about pointers.
That's all.
The buffer has 437 characters unused.
```

代码的说明

在这个例子中，首先使用全局变量 BUFFER_LEN 指定 buffer 数组的大小:

```
const size_t BUFFER_LEN = 512; /* Size of input buffer */
```

这个变量必须声明为 const，才能用来指定数组的大小；数组的大小只能用常量表达式指定。

main()函数开始处的声明如下所示:

```
char buffer[BUFFER_LEN]; /* Store for strings */
char *pS[3] = { NULL };  /* Array of string pointers */
char *pbuffer = buffer;  /* Pointer to buffer */
size_t index = 0;        /* Available buffer position*/
```

buffer 数组的类型是 char，有 BUFFER_LEN 个元素。pS 数组有 3 个指针，存储 buffer 数组中字符串的地址。pbuffer 指针用 buffer 数组中的第一个字节的地址初始化。在填充输入字符时，要使用 pbuffer 遍历 buffer 数组。index 变量记录 buffer 数组中当前未用的元素的位置。

第一个 for 循环读取 3 个字符串。循环中的第一条语句如下:

```
printf("\nEnter %s message\n", i>0? "another" : "a" );
```

这里通过一种简洁的方式使用条件运算符，在 for 循环的第一次迭代后修改提示。

www.TopSage.com

该语句在第一次迭代时输出 a，在后续的迭代中输出 another。

下一条语句将当前保存在 pbuffer 中的地址存储到指针数组中：

```
pS[i] = &buffer[index];          /* Save start of string */
```

上述赋值语句把指针 pbuffer 中的地址保存到指针数组 pS 的一个元素中。

读取字符串并添加字符串终止符的语句如下：

```
for( ; index<BUFFER_LEN ; index++) /* If you read \n ... */
  if((*(pbuffer+index) = getchar()) == '\n')
  {
    *(pbuffer+index++) = '\0';      /* ...substitute \0 */
    break;
  }
```

这个 for 循环就是上一节用于读取到 buffer 数组末尾的循环。如果读入一个'\n'，就用'\0'替代它，并结束循环。循环结束后，检查 buffer 数组是否还没有到达字符串的末尾就已满：

```
if((index == BUFFER_LEN) && ((*(pbuffer+index-1) != '\0') || (i<2)))
{
  printf("\nYou ran out of space in the buffer.");
  return 1;
}
```

这段代码在上一节解释过了。

使用 getchar()读取字符串，可以对输入过程进行很多控制。这种方法并不仅限于读取字符串，还可以用于读取逐个处理字符的所有输入过程。可以从输入中删除空格，或者查找特定的字符，例如用于分隔各个输入值的逗号。

```
printf("\nThe strings you entered are:\n\n");
for(int i = 0 ; i<3 ; i++)
  printf("%s\n", pS[i]);
```

在循环中，输出 pS 指向的每个元素中的字符串。

可以进一步开发这个例子，允许输入任意多个信息，其个数仅受限于为数组提供的字符串指针个数。

在最后一个 printf()中，输出字符串中剩下的字符个数：

```
printf("The buffer has %d characters unused.\n",
  BUFFER_LEN-index);
```

从 buffer 数组的元素个数中减去 index，得到未使用的元素个数。

本章开头将指针初始化为 NULL。还可以将指针初始化为常量字符串的地址：

```
char *pString = "To be or not to be";
```

这条语句给字符串分配了足够多的空间，将常量字符串放在所分配的内存中，将

www.TopSage.com

指针 pS 的值设置为字符串第一个字节的地址。该语句的问题是可以用如下语句修改字符串:

```
*(pString+3) = 'm';
```

const 关键字对此没有作用。如果将 pString 声明为 const，仅指定了指针是常量指针，它指向的内容并不是常量。

试试看：字符串输入的扩展

重写上面的例子，扩展字符串输入示例。可以扩充这个程序，读取任意数量的字符串，直到指定的数量为止，并确保读入的字符串不超过所提供的空间。程序如下:

```
/* Program 7.13 Generalizing string input */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const size_t BUFFER_LEN = 128; /* Length of input buffer */
const size_t NUM_P = 100;      /* maximum number of strings */

int main(void)
{
  char buffer[BUFFER_LEN];      /* Input buffer */
  char *pS[NUM_P] = { NULL };   /* Array of string pointers */
  char *pbuffer = buffer;       /* Pointer to buffer */
  int i = 0;                    /* Loop counter */

printf("\nYou can enter up to %u messages each up to %u characters.",
    NUM_P, BUFFER_LEN-1);

for(i = 0 ; i<NUM_P ; i++)
{
  pbuffer = buffer ; /* Set pointer to beginning of buffer */
  printf("\nEnter %s message, or press Enter to end\n",
        i>0? "another" : "a");

  /* Read a string of up to BUFFER_LEN characters */
  while((pbuffer - buffer < BUFFER_LEN-1) &&
     ((*pbuffer++ = getchar()) != '\n'));

  /* check for empty line indicating end of input */
  if((pbuffer - buffer) < 2)
    break;

  /* Check for string too long */
  if((pbuffer - buffer) == BUFFER_LEN && *(pbuffer-1)!= '\n')
  {
    printf("String too long - maximum %d characters allowed.",
          BUFFER_LEN);
```

www.TopSage.com

```
      i--;
      continue;
    }
    *(pbuffer - 1) = '\0';                    /* Add terminator */

    pS[i] = (char*)malloc(pbuffer-buffer); /* Get memory for string */
    if(pS[i] == NULL)                         /* Check we actually got some…*/
    {
      printf("\nOut of memory - ending program.");
      return 1;                               /* …Exit if we didn't */
    }

    /* Copy string from buffer to new memory */
    strcpy(pS[i], buffer);
  }

  /* Output all the strings */
  printf("\nIn reverse order, the strings you entered are:\n");
  while(--i >= 0)
  {
    printf("\n%s", pS[i] ); /* Display strings last to first */
    free(pS[i]);            /* Release the memory we got */
    pS[i] = NULL;           /* Set pointer back to NULL for safety*/
  }
  return 0;
}
```

输出和前两个例子非常相似:

```
Enter a message, or press Enter to end
Hello

Enter another message, or press Enter to end
World!

Enter another message, or press Enter to end

In reverse order, the strings you entered are:
World!
Hello
```

代码的说明

与最初的版本相比，这个程序稍有扩展，但涵盖了相当多的内容。现在可以处理任意数量的字符串，能处理的最大字符串数是数组 pS 中的指针数。这个数组的大小在程序起始处定义，以便于修改。

```
const size_t NUM_P = 100;     /* maximum number of strings */
```

如果要改变这个程序能处理的最大字符串数，只需改变这个变量值。在 main()函数

www.TopSage.com

开始处，声明一些需要的变量:

```
char buffer[BUFFER_LEN];      /* Input buffer */
char *pS[NUM_P] = { NULL };  /* Array of string pointers */
char *pbuffer = buffer;       /* Pointer to buffer */
int i = 0;                    /* Loop counter */
```

buffer 数组只是一个输入缓冲区，含有每个读入的字符串。因此，#define 指令将 BUFFER_LEN 定义为能接受的字符串最大长度。然后，声明指针数组的长度 NUM_P 和指针 pbuffer，以用于 buffer 数组。最后是两个循环控制变量。

下面显示一条信息，说明输入的限制:

```
printf("\nYou can enter up to %u messages each up to %u characters.",
     NUM_P, BUFFER_LEN-1);
```

输入信息的最大长度允许加上终止字符。

第一个 for 循环读入字符串并存储它们。这个循环控制如下:

```
for  (i = 0; i < NUM_P; i++)
```

这能确保输入的字符串数不超过前面声明的指针数量。一旦输入的字符串数到达最大字符串数，循环就会结束，进入程序的输出部分。

在循环中，字符串输入使用类似 getchar()的机制，但是多了一个额外的条件:

```
/* Read a string of up to BUFFER_LEN characters */
while((pbuffer - buffer < BUFFER_LEN-1) &&
   ((*pbuffer++ = getchar()) != '\n'));
```

整个过程发生在 while 循环的条件式中。由 getchar()得到的字符存储在 pbuffer 指向的地址中，pbuffer 最初保存的是 buffer 的地址。然后递增 pbuffer 指针，指向下一个可用的空间，这个赋值语句所存储的字符与'\n'比较，若该字符是'\n'，就结束循环。如果 pbuffer-buffer<BUFFER _LEN-1 是 false，循环也会结束。即如果下一个要存储的字符占据了 buffer 数组的最后一个位置，循环也会结束。

输入过程结束后，用下面的语句进行检查:

```
if((pbuffer  =  buffer) < 2)
  break;
```

这个语句检测空行，因为如果只按下回车键，就只输入一个字符'\n'。此时，break 语句立即结束循环，开始输出过程。

下一个 if 语句检查是否试图输入超过 buffer 容量的字符串:

```
if((pbuffer - buffer) == BUFFER_LEN && *(pbuffer-1)!= '\n')
{
  printf("String too long - maximum %d characters allowed.",
    BUFFER_LEN);
  i--;
```

www.TopSage.com

```
    continue;
  }
```

因为使用了 buffer 数组的最后一个位置时，会结束 while 循环，如果试图输入超过 buffer 数组容量的字符，表达式 pbuffer-buffer 就等于 BUFFER_LEN。当然如果输入一个刚好等于 buffer 数组容量的字符串，也会出现这种情况。所以也必须检查 buffer 的最后一个字符，确定它是不是'\n'。如果不是，表示输入了太多的字符，所以在显示一个信息后，递减循环记数器，进入下一次迭代。

下一条语句是:

```
*(pbuffer - 1) = '\0'; /* Add terminator */
```

这条语句把'\0'放在'\n'字符的位置上，因为 pbuffer 指向 buffer 数组中第一个未用的元素。输入了字符串后，就使用 malloc()函数请求足够的内存，保存这个字符串:

```
pS[i] = (char*)malloc(pbuffer-buffer); /* Get memory for string */
if(pS[i] == NULL)                      /* Check we actually got some…*/
{
  printf("\nOut of memory - ending program.");
  return 0;                            /* …Exit if we didn't */
}
```

所需的字节数是 pbuffer 当前指向的地址(即 buffer 中的第一个空元素)和 buffer 中第一个元素的地址之差。从 malloc()返回的指针转换成 char 类型后，存储到 pS 数组的当前元素中。如果 malloc()返回一个 NULL 指针，就显示一条信息，并结束程序。

使用下面的语句，把这个字符串从 buffer 复制到新得到的内存中:

```
strcpy(pS[i], buffer);
```

这条语句使用了库函数 strcpy()，将 buffer 的内容复制到 pS[i]指向的内存中。注意，使用 strcpy()函数时不要混淆其参数。第二个参数是复制操作的源内容，第一个参数是目的地。混淆它们通常非常危险，因为复制操作会一直持续到找到'\0'为止。

结束循环后，不论是因为输入了一个空字符串，或是使用了 pS 数组中的所有指针，都会产生输出:

```
printf("\nIn reverse order, the strings you entered are:\n");
while(--i >= 0)
{
  printf("\n%s", pS[i] ); /* Display strings last to first */
  free(pS[i]);            /* Release the memory we got */
  pS[i] = NULL;           /* Set pointer back to NULL for safety*/
}
```

索引 i 的值比输入字符串的个数多 1。因此，在检查第一个循环条件后，可以使用它索引最后一个字符串。这个循环会递减这个值，在最后一次迭代时，i 是 0，索引第一个字符串。

www.TopSage.com

可以使用表达式*(pS+i)代替 pS[i]，但使用数组表示法比较简洁。

在最后的 printf()之后使用 free()函数。这个函数和 malloc()是互补的，它释放了 malloc()分配的内存。它只需把所分配的内存指针作为参数。虽然内存在程序结束时会自动释放，但是内存最好在不需要时立即释放。当然，一旦用这个方式释放内存后，就不能再使用它，所以最好立刻将指针设定成 NULL。

警告：

指针错误会产生灾难性的结果。如果使用一个没有指定地址值的未初始化指针存储值，该指针使用的地址就是存储在该指针位置的任何内容，这可能是内存中的任何一个位置。

试试看：使用指针对字符串排序

下面的例子使用在<string.h>头文件中声明的函数，介绍如何使用指针进行简单的排序：

```
/* Program 7.14 Sorting strings */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#define BUFFER_LEN 100          /* Length of input buffer */
#define NUM_P 100               /* maximum number of strings */

int main(void)
{
  char buffer[BUFFER_LEN];      /* space to store an input string */
  char *pS[NUM_P] = { NULL };   /* Array of string pointers */
  char *pTemp = NULL;           /* Temporary pointer */
  int i = 0;                    /* Loop counter */
  bool sorted = false;          /* Indicated when strings are sorted */
  int last_string = 0;          /* Index of last string entered */

  printf("\nEnter successive lines, pressing Enter at the"
       " end of each line.\nJust press Enter to end.\n\n");
  while((*fgets(buffer, BUFFER_LEN, stdin) != '\n') && (i < NUM_P))
  {
    pS[i] = (char*)malloc(strlen(buffer) + 1);
    if(pS[i]==NULL) /* Check for no memory allocated */
    {
      printf(" Memory allocation failed. Program terminated.\n");
      return 1;
    }
    strcpy(pS[i++], buffer);
  }
  last_string = i; /* Save last string index */

  /* Sort the strings in ascending order */
```

www.TopSage.com

```
  while(!sorted)
  {
    sorted = true;
    for(i = 0 ; i<last_string-1 ; i++)
      if(strcmp(pS[i], pS[i + 1]) > 0)
      {
        sorted = false;     /* We were out of order */
        pTemp= pS[i];       /* Swap pointers pS[i] */
        pS[i] = pS[i + 1]; /* and */
        pS[i + 1] = pTemp; /* pS[i + 1] */
      }
  }

  /* Displayed the sorted strings */
  printf("\nYour input sorted in order is:\n\n");
  for(i = 0 ; i<last_string ; i++)
  {
    printf("%s\n", pS[i] );
    free( pS[i] );
    pS[i] = NULL;
  }
  return 0;
}
```

假定输入相同的数据，这个程序的输出如下所示:

```
Enter successive lines, pressing Enter at the end of each line.
Just press Enter to end.

Many a mickle makes a muckle.
A fool and your money are soon partners.
Every dog has his day.
Do unto others before they do it to you.
A nod is as good as a wink to a blind horse.

Your input sorted in order is:

A fool and your money are soon partners.
A nod is as good as a wink to a blind horse.
Do unto others before they do it to you.
Every dog has his day.
Many a mickle makes a muckle.
```

代码的说明

这个例子从谷壳里分类小麦。它使用输入函数 fgets()读入一个完整的字符串，直到按下回车键为止，然后在末尾加上'\0'。使用 fgets()而不使用 gets()，可以确保不超过 buffer 的容量。第一个参数是一个指针，它指向存储字符串的内存区域，第二个参数是可以存储的最大字符数，第三个参数是要读取的流，这里是标准输入流。它的返回值是存储输

www.TopSage.com

入字符串的地址，在这个例子中是 buffer。如果出错，返回值就是 NULL。fgets()与 gets()的区别是，gets()存储终止输入的换行符，之后加上终止符'\0'，而+gets()不存储换行符。

这个程序的整个操作是非常简单的，它包含 3 个不同的动作：

- 读入全部的输入字符串。
- 给输入的字符串排序。
- 以字母顺序显示输入字符串。

显示开始的提示行后，下面的语句就处理输入过程：

```
while((*fgets(buffer, BUFFER_LEN, stdin) != '\n') && (i < NUM_P))
{
  pS[i] = (char*)malloc(strlen(buffer) + 1);
  if(pS[i]==NULL) /* Check for no memory allocated */
  {
    printf(" Memory allocation failed. Program terminated.\n");
    return 1;
  }
  strcpy(pS[i++], buffer);
}
```

输入过程一直持续到输入一个空行或用完指针数组的空间为止。使用 fgets()函数将每一行读入 buffer。这在 while 循环条件中完成，只要 fgets()没有读入'\0'，且输入的总行数不超过指针数组的大小，循环就会继续。没有输入任何文本就按下回车键，就会输入一个'\0'。使用*获取 fgets()返回的指针地址的内容。当然这和取消 buffer 的引用是相同的。

把每个输入行放到 buffer 中后，就用 malloc()函数分配能容纳这行字符的内存量。使用 strlen()函数可以得到所需要的字节数，因为尾部要加上'\0'，所以再给该字节数加 1。确认获得了分配的内存后，使用库函数 strcpy()将字符串从 buffer 复制到新的内存中。

然后，存储最后一个字符串的索引：

```
last_string = i; /* Save last string index */
```

这是因为要重用循环记数器 i，需要记录有多少个字符串。

安全地保存了所有的字符串后，就使用最简单的、效率最低的但很容易理解的方法给它们排序，如下面的语句：

```
while(!sorted)
{
  sorted = true;
  for(i = 0 ; i<last_string-1 ; i++)
    if(strcmp(pS[i], pS[i + 1]) > 0)
    {
      sorted = false;     /* We were out of order */
      pTemp= pS[i];       /* Swap pointers pS[i] */
      pS[i] = pS[i + 1];  /* and */
      pS[i + 1] = pTemp;  /* pS[i + 1] */
    }
}
```

www.TopSage.com

排序在 while 循环内部进行：只要 sorted 是 false，循环就继续。在排序时，要在 for 循环内使用 strcmp()函数比较连续的一对字符串。如果第一个字符串大于第二个字符串，就交换指针值。使用指针是交换顺序的一种非常经济的方法。这些字符串本身仍旧在它原来的内存中，而只是在指针数组 pS 中交换它们的地址顺序。这个机制如图 7-4 所示。交换指针所需的时间是移动所有字符串所需时间的一部分。

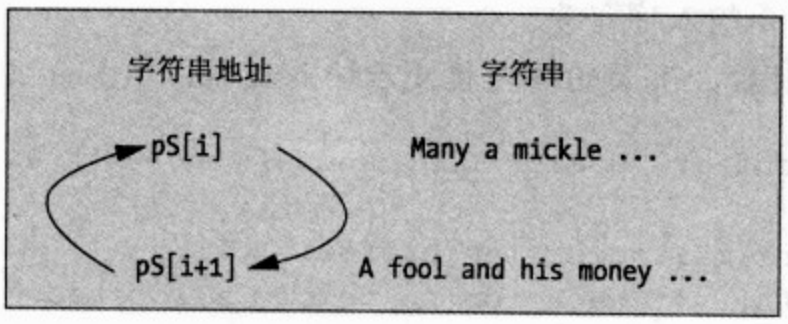


图 7-4 使用指针来排序

对所有的字符串指针执行这个交换过程。如果在遍历字符串时需要交换它们，就把 sorted 设定为 false，重复整个 for 循环。如果重复 for 循环时没有互换任何字符串，就表示字符串已完成排序了。用 bool 变量 sorted 跟踪其状态。它在每个循环的开始设为 true，但是只要发生互换操作，就将它设回 false。如果退出循环时，sorted 仍然是 true，就表示没有发生互换操作，所以每个字符串都已排好序；因而退出 while 循环。

这个排序不是很好的原因是，每次遍历所有的项时，仅将一个值移动一个位置。最坏的情况是，第一项在最后的位置上，此时重复这个过程的次数是列表里的项数减 1。这个效率低却很有名的方法叫做冒泡排序(bubble sort)。

以这种方式使用指针处理字符串和其他类型的数据，是 C 语言中一个相当强大的机制。可以将基本数据(在这个例子是字符串)以任意顺序放在一块内存里，然后只需改变指针，就可以用任意顺序处理它们，而完全不用移动它们。可以使用这个例子的方法作为排序任何文本的基础。然而，最好使用一个比较好的排序方法。

7.6 设计程序

前面介绍了 C 语言中一个比较难的部分，现在运用学过的内容编写一个应用程序。本节将依循惯例，先分析、设计，然后一步步编写代码。这是本章最后一个程序。

7.6.1 问题

要处理的问题是用一些新的特性重写第 3 章的计算器程序，但这次要使用指针。主要改进如下：

- 允许使用有符号的小数，包含带 - 或+符号的小数和有符号的整数。
- 允许表达式组合多个运算式，如 2.5+3.7 - 6/6。
- 添加运算符^，计算幂，因此 2^3 会得到 8。

www.TopSage.com

- 允许使用前一个运算的结果。如果前一个运算的结果是 2.5，那么=*2+7 会得到 12。任何以赋值运算符开头的输入行都自动假设左操作数是前一个运算的结果。

不考虑运算符的优先级，只简单地从左到右计算输入的表达式，将每个运算符应用于前一个结果和右操作数。所以下面的表达式：

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

会以如下方式计算：

```
((1 + 2)*3 - 4)*(-5)
```

7.6.2 分析

我们事先并不知道表达式有多长或有多少个操作数。用户会输入一个完整的字符串，然后分析它，确定它包含什么数值和运算符。只要一个运算符有左右操作数，就计算其中间结果。

步骤如下：

(1) 读入用户输入的字符串，如果它是 quit，就退出。

(2) 检查=运算符，如果有一个=运算符，就查找第一个操作数。

(3) 寻找跟在操作数后的运算符，依次执行每一个运算符，直到输入字符串结束为止。

(4) 显示结果，并回到步骤 1。

7.6.3 解决方案

本节列出解决问题的步骤。

1. 步骤 1

如本章前面所述，scanf()函数不允许读入含有空格的完整字符串，而是在第一个空白字符处停止读取。因此，这里使用在头文件<stdio.h>中声明的 gets()函数读取输入的表达式。这个函数会读入整行输入，包含空格。可以将输入和整个程序的循环结合在一起，如下：

```
/* Program 7.15 An improved calculator */
#include <stdio.h>          /* Standard input/output */
#include <string.h>         /* For string functions */
#define BUFFER_LEN 256      /* Length of input buffer */

int main(void)
{
  char input[BUFFER_LEN]; /* Input expression */
  while(strcmp(fgets(input, BUFFER_LEN, stdin), "quit\n") != 0)
  {
    /* Code to implement the calculator */
  }
```

www.TopSage.com

```
  return 0;
}
```

可以这么做，是因为函数 strcmp()的参数是一个指向字符串的指针，而函数 fgets()会返回一个指向用户输入的字符串的指针，在这个例子中是&input[0]。strcmp()函数会比较输入的字符串和"quit\n"，如果它们相等，就返回 0。然后，结束循环。

设定输入的字符串长度是 256。这应该足够了，因为大多数计算机的键盘缓冲区是 255 个字符(这也是按下回车键之前，所能输入的最大字符数)。

有了字符串，就可以开始分析它，但最好从字符串中删除空格。因为输入的字符串是定义好的，不需要用空格分隔运算符和操作数。下面在 while 循环中添加代码，删除空格：

```
/* Program 7.15 An improved calculator */
#include <stdio.h>          /* Standard input/output */
#include <string.h>         /* For string functions */
#define BUFFER_LEN 256      /* Length of input buffer */

int main(void)
{
  char input[BUFFER_LEN];   /* Input expression */
  unsigned int index = 0;   /* Index of the current character in input */
  unsigned int to = 0;      /* To index for copying input to itself */
  size_t input_length = 0; /* Length of the string in input */
  while(strcmp(fgets(input, BUFFER_LEN, stdin), "quit\n") != 0)
  {
    input_length = strlen(input); /* Get the input string length */
    input[--input_length] = '\0'; /* Remove newline at the end */

    /* Remove all spaces from the input by copy the string to itself */
    /* including the string terminating character */
    for(to = 0, index = 0 ; index<=input_length ; index++)
    if(*(input+index) != ' ')          /* If it is not a space */
      *(input+to++) = *(input+index); /* Copy the character */

    input_length = strlen(input);      /* Get the new string length */
    index = 0;                         /* Start at the first character */

    /* Code to implement the calculator */
  }
  return 0;
}
```

上述代码声明了一些额外的变量。变量 input_length 声明为类型 size_t，是为了和 strlen()函数返回的类型兼容，以避免编译器产生警告信息。

按下回车键结束输入行后，fgets()函数会存储一个换行符。分析这个字符串的代码不希望看到换行符，所以用'\0'覆盖它。input 数组的索引表达式递减 strlen()函数返回的长度值，使用其结果引用包含换行符的元素。

www.TopSage.com

将存储在 input 中的字符串复制到它本身，以删除空格。在复制循环中需要跟踪两个索引：一个是下一个要复制的非空格字符在 input 中的位置，另一个是下一个要复制的字符的位置。在这个循环中不复制空格；只递增 index，以移动到下一个字符。索引 to 只有在复制字符时才递增。进入循环后，把新字符串长度存储到 input_length 中，并重置索引，以引用 input 的第一个字符。

可以用数组表示法来编写这个循环：

```
for(to = 0, index = 0 ; index<=input_length ; index++)
  if(input[index] != ' ')          /* If it is not a space */
    input[to++] = input[index];  /* Copy the character */
```

使用数组表示法比较整洁，但下面使用指针表示法来实践。

2. 步骤 2

输入的表达式有两种可能的形式。它可以由一个赋值运算符开始，表示最后的结果要用作左操作数，它也可以由一个有符号或无符号的数值开始。可以先寻找“=”字符来区分这两种情况。如果找到一个“=”字符，左操作数就是前一个运算的结果。

下面添加到 while 循环中的代码将查找“=”字符，如果没找到，就寻找一个数值子字符串，它应该是左操作数：

```
/* Program 7.15 An improved calculator */
#include <stdio.h>          /* Standard input/output */
#include <string.h>         /* For string functions */
#include <ctype.h>          /* For classifying characters */
#include <stdlib.h>         /* For converting strings to numeric values */
#define BUFFER_LEN 256      /* Length of input buffer */

int main(void)
{
  char input[BUFFER_LEN];   /* Input expression */
  char number_string[30];   /* Stores a number string from input */
  unsigned int index = 0;   /* Index of the current character in input */
  unsigned int to = 0;      /* To index for copying input to itself */
  size_t input_length = 0;  /* Length of the string in input */
  unsigned int number_length = 0;
                            /* Length of the string in number_string */
  double result = 0.0;      /* The result of an operation */

  while(strcmp(fgets(input, BUFFER_LEN, stdin), "quit\n") != 0)
  {
    input_length = strlen(input); /* Get the input string length */
    input[--input_length] = '\0'; /* Remove newline at the end */

    /* Remove all spaces from the input by copying the string to itself */
    /* including the string terminating character */
    for(to = 0, index = 0 ; index<=input_length ; index++)
```

www.TopSage.com

```
if(*(input+index) != ' ')                    /* If it is not a space */
  *(input+to++) = *(input+index);            /* Copy the character */
input_length = strlen(input);                /* Get the new string length */
index = 0;                                   /* Start at the first character */

if(input[index]== '=')                       /* Is there =? */
 index++;                                    /* Yes so skip over it */
else
{                                            /* No - look for the left operand */
 /* Look for a number that is the left operand for */
 /* the first operator */

 /* Check for sign and copy it */
 number_length = 0;                          /* Initialize length */
 if(input[index]=='+' || input[index]=='-') /* Is it + or -? */
       *(number_string+number_length++) = *(input+index++);
                                             /* Yes so copy it */

 /* Copy all following digits */
 for( ; isdigit(*(input+index)) ; index++) /* Is it a digit? */
  *(number_string+number_length++) = *(input+index);
                                             /* Yes - Copy it */

 /* copy any fractional part */
 if(*(input+index)=='.')                     /* Is it decimal point? */
 { /* Yes so copy the decimal point and the following digits */
  *(number_string+number_length++) = *(input+index++);
                                             /* Copy point */

   for( ; isdigit(*(input+index)) ; index++) /* For each digit */
    *(number_string+number_length++) = *(input+index); /* copy it */
  }
  *(number_string+number_length) = '\0'; /* Append string terminator */

  /* If we have a left operand, the length of number_string */
  /* will be > 0. In this case convert to a double so we */
  /* can use it in the calculation */
  if(number_length>0)
   result = atof(number_string); /* Store first number as result */
 }

 /* Code to analyze the operator and right operand */
 /* and produce the result */
}
return 0;
}
```

为字符分析函数包含<ctype.h>头文件，为使用 atof()函数包含<stdlib.h>头文件，atof()函数将一个字符串参数转换成浮点值。这里添加了相当多的代码，但它们都是由一些容

www.TopSage.com

易理解的步骤组成。

if语句检查“=”是否为input中的第一个字符：

```
if(input[index]== '=')  /* Is there =? */
  index++;              /* Yes so skip over it */
```

如果找到一个“=”，就递增index，跳过它，直接找寻操作数。否则，就执行else，查找数值型的左操作数。

把组成这个数的所有字符复制到number_string数组中。这个数可能由一元运算符'-'或'+'开头，所以在else块中首先检查第一个字符。如果找到该一元运算符，就用以下的语句把它复制到number_string中：

```
if(input[index]=='+' || input[index]=='-')          /* Is it + or -? */
 *(number_string+number_length++) = *(input+index++); /* Yes so copy it */
```

如果没有找到符号，index值就保持不变，index值记录了当前要在input中分析的字符。如果找到符号，就将它复制到number_string中，然后递增index的值，指向下一个字符。

接下来应有一个或多个数字，所以用一个for循环将所有的数字复制到number_string中：

```
for( ; isdigit(*(input+index)) ; index++)          /* Is it a digit? */
  *(number_string+number_length++) = *(input+index); /* Yes - Copy it */
```

这将复制整数的所有数字，随后相应递增index的值。当然，如果没有数字，index的值不会改变。

这个数可能不是整数。在这种情形下，下一个必定是小数点，它可能后跟许多数字。if语句检查小数点。如果有一个小数点，就复制这个小数点和其后的数字：

```
if(*(input+index)=='.')                          /* Is it decimal point? */
  { /* Yes so copy the decimal point and the following digits */
  *(number_string+number_length++) = *(input+index++); /* Copy point */

  for( ; isdigit(*(input+index)) ; index++)          /* For each digit */
    *(number_string+number_length++) = *(input+index);    /* copy it */
}
```

现在复制完表示第一个操作数的字符串，所以给number_string附加一个字符串终止字符：

```
*(number_string+number_length) = '\0'; /* Append string terminator */
```

也可能没有找到数值，如果在number_string中复制了代表一个数值的字符串，number_length的值就必然是正的，因为它至少要有一个数字。因此，将number_length的值作为有一个数值的指示器：

```
if(number_length>0)
  result = atof(number_string);      /* Store first number as result */
```

www.TopSage.com

atof()函数把这个字符串转换成 double 类型的浮点数。注意，字符串的值存储到 result 中。后面会使用这个变量存储运算的结果。这可确保 result 始终含有运算的结果，包含计算完整个字符串的结果。如果不在这里存储值，由于没有左操作数，result 将含有前一个输入字符串的值。

3. 步骤 3

此时，输入字符串后的内容已定义得非常清楚。它必然是一个运算符跟一个数值。这个运算符把之前找到的数或前一个的 result 用作左操作数。这个运算符和数值(op-number)的组合也可能后跟另一个运算符和数值组合，所以可能会有连续的运算符和数值组合，直到字符串结束。可以用一个循环查找这些组合：

```
/* Program 7.15 An improved calculator */
#include <stdio.h>      /* Standard input/output */
#include <string.h>     /* For string functions */
#include <ctype.h>      /* For classifying characters */
#include <stdlib.h>     /* For converting strings to numeric values */
#define BUFFER_LEN 256 /* Length of input buffer */

int main(void)
{
  char input[BUFFER_LEN];   /* Input expression */
  char number_string[30];   /* Stores a number string from input */
  char op = 0;              /* Stores an operator */

  unsigned int index = 0;   /* Index of the current character in input */
  unsigned int to = 0;      /* To index for copying input to itself */
  size_t input_length = 0; /* Length of the string in input */
  unsigned int number_length = 0;
  /* Length of the string in number_string */
  double result = 0.0; /* The result of an operation */
  double number = 0.0; /* Stores the value of number_string */

  while(strcmp(fgets(input, BUFFER_LEN, stdin), "quit\n") != 0)
  {
  input_length = strlen(input); /* Get the input string length */
  input[--input_length] = '\0'; /* Remove newline at the end */

  /* Remove all spaces from the input by copying the string to itself */
  /* including the string terminating character */
  /* Code to remove spaces as before... */

  /* Code to check for '=' and analyze & store the left operand as before */

  /* Now look for 'op number' combinations */
  for(;index < input_length;)
  {
    op = *(input+index++); /* Get the operator */
```

www.TopSage.com

```
/* Copy the next operand and store it in number */
number_length = 0;      /* Initialize the length */

/* Check for sign and copy it */
if(input[index]=='+' || input[index]=='-') /* Is it + or -? */
  *(number_string+number_length++) = *(input+index++);
                                          /* Yes - copy it. */


/* Copy all following digits */
for( ; isdigit(*(input+index)) ; index++)      /* For each digit */
*(number_string+number_length++) = *(input+index); /* copy it. */

/* copy any fractional part */
if(*(input+index)=='.')                  /* Is it a decimal point? */
  { /* Copy the decimal point and the following digits */
  *(number_string+number_length++) = *(input+index++); /* Copy point */
  for( ; isdigit(*(input+index)) ; index++)    /* For each digit */
    *(number_string+number_length++) = *(input+index); /* copy it. */
 }
 *(number_string+number_length) = '\0'; /* terminate string */

 /* Convert to a double so we can use it in the calculation */
 number = atof(number_string);
}
/* code to produce result */
}
return 0;
}
```

这里没有重复相同的代码，而是用一些注释指出在哪里添加上述代码。下次给程序添加代码时，将列出完整的代码。

for 循环会持续到到达输入串的末尾为止，此时 index 等于 input_length。循环在每次迭代时，都把运算符存储到 char 类型变量 op 中：

```
op = *(input+index++); /* Get the operator */
```

执行这个运算符后，提取组成下一个数的字符，这是运算符的右操作数。这里没有验证这个运算符是否有效，所以此时程序不立即断定它是无效的运算符。

从字符串中提取右操作数的过程和提取左操作数完全相同。重复相同的代码。然而这次，操作数的 double 值存储到 number 中：

```
number = atof(number_string);
```

现在左操作数存储在 result 中，运算符存储在 op 中，右操作数存储在 number 中。下面准备执行这个计算式：

```
Resule=(result  op  number]
```

为此添加代码后，这个程序就完整了。

www.TopSage.com

4. 步骤 4

用一个 switch 语句根据操作数选择要执行的运算。这和之前的计算器代码相同。显示输出，并且程序开头添加一个提示，说明如何使用这个计算器。下面是这个程序的完整代码，新加的代码用粗体字显示：

```
/* Program 7.15 An improved calculator */
#include <stdio.h>   /* Standard input/output */
#include <string.h>  /* For string functions */
#include <ctype.h>   /* For classifying characters */
#include <stdlib.h>  /* For converting strings to numeric values */
#include <math.h>    /* For power() function */
#define BUFFER_LEN 256 /* Length of input buffer */

int main(void)
{
char input[BUFFER_LEN];   /* Input expression */
char number_string[30];   /* Stores a number string from input */
char op = 0;              /* Stores an operator */
unsigned int index = 0;   /* Index of the current character in input */
unsigned int to = 0;      /* To index for copying input to itself */
size_t input_length = 0;  /* Length of the string in input */
unsigned int number_length = 0; /* Length of the string in number_string */
double result = 0.0;      /* The result of an operation */
double number = 0.0;      /* Stores the value of number_string */

printf("\nTo use this calculator, enter any expression with"
       " or without spaces");
printf("\nAn expression may include the operators:");
printf("\n +, -, *, /, %%, or ^(raise to a power).");
printf("\nUse = at the beginning of a line to operate on ");
printf("\nthe result of the previous calculation.");
printf("\nUse quit by itself to stop the calculator.\n\n");

/* The main calculator loop */
while(strcmp(fgets(input, BUFFER_LEN, stdin), "quit\n") != 0)
{
  input_length = strlen(input); /* Get the input string length */
  input[--input_length] = '\0'; /* Remove newline at the end */

  /* Remove all spaces from the input by copying the string to itself */
  /* including the string terminating character */
  for(to = 0, index = 0 ; index<=input_length ; index++)
  if(*(input+index) != ' ')          /* If it is not a space */
    *(input+to++) = *(input+index); /* Copy the character */

  input_length = strlen(input);      /* Get the new string length */
  index = 0;                         /* Start at the first character */
```

www.TopSage.com

```
if(input[index]== '=')                    /* Is there =? */
  index++;                                /* Yes so skip over it */
else
{                                         /* No - look for the left operand */
/* Look for a number that is the left operand for the 1st operator */

/* Check for sign and copy it */
number_length = 0;                                  /* Initialize length */
if(input[index]=='+' || input[index]=='-')  /* Is it + or -? */
  *(number_string+number_length++) = *(input+index++);
                                                    /* Yes so copy it */

/* Copy all following digits */
for( ; isdigit(*(input+index)) ; index++) /* Is it a digit? */
  *(number_string+number_length++) = *(input+index); /* Yes - Copy it */

/* copy any fractional part */
if(*(input+index)=='.')                           /* Is it decimal point? */
  { /* Yes so copy the decimal point and the following digits */
  *(number_string+number_length++) = *(input+index++); /* Copy point */

  for( ; isdigit(*(input+index)) ; index++)       /* For each digit */
    *(number_string+number_length++) = *(input+index); /* copy it */
}
*(number_string+number_length) = '\0'; /* Append string terminator */

/* If we have a left operand, the length of number_string */
/* will be > 0. In this case convert to a double so we */
/* can use it in the calculation */
if(number_length>0)
  result = atof(number_string); /* Store first number as result */
}

/* Now look for 'op number' combinations */
for(;index < input_length;)
{
  op = *(input+index++); /* Get the operator */
  /* Copy the next operand and store it in number */
  number_length = 0;      /* Initialize the length */

  /* Check for sign and copy it */
  if(input[index]=='+' || input[index]=='-') /* Is it + or -? */
    *(number_string+number_length++) = *(input+index++);
                                                    /* Yes - copy it. */

  /* Copy all following digits */
  for( ; isdigit(*(input+index)) ; index++) /* For each digit */
    *(number_string+number_length++) = *(input+index); /* copy it. */
```

www.TopSage.com

```
    /* copy any fractional part */
    if(*(input+index)=='.') /* Is it a decimal point? */
      { /* Copy the decimal point and the following digits */
        /* Copy point */
      *(number_string+number_length++) = *(input+index++);
      for( ; isdigit(*(input+index)) ; index++) /* For each digit */
        *(number_string+number_length++) = *(input+index); /* copy it. */
}
*(number_string+number_length) = '\0'; /* terminate string */

/* Convert to a double so we can use it in the calculation */
number = atof(number_string);

/* Execute operation, as 'result op= number' */
switch(op)
{
  case '+':      /* Addition */
    result += number;
    break;
  case '-':      /* Subtraction */
    result -= number;
    break;
  case '*':      /* Multiplication */
    result *= number;
    break;
  case '/':      /* Division */
    /* Check second operand for zero */
    if(number == 0)
      printf("\n\n\aDivision by zero error!\n");
    else
      result /= number;
    break;
  case '%':      /* Modulus operator - remainder */
    /* Check second operand for zero */
    if((long)number == 0)
      printf("\n\n\aDivision by zero error!\n");
    else
      result = (double)((long)result % (long)number);
    break;
  case '^':       /* Raise to a power */
    result = pow(result, number);
    break;
  default:        /* Invalid operation or bad input */
    printf("\n\n\aIllegal operation!\n");
    break;
 }
 }
 printf("= %f\n", result); /* Output the result */
}
```

www.TopSage.com

```
return 0;
}
```

switch 语句和前一个计算器程序相同，但多了一些 case。因为这个程序用幂函数 pow() 计算 resultnumber，所以必须使用#include 指令包含头文件<math.h>。这个计算器程序的输出如下：

```
To use this calculator, enter any expression with or without spaces
An expression may include the operators:
   +, -, *, /, %, or ^(raise to a power).
Use = at the beginning of a line to operate on
the result of the previous calculation.
Use quit by itself to stop the calculator.

2.5+3.3/2
= 2.900000
= *3
= 8.700000
= ^4
= 5728.976100
1.3+2.4-3.5+-7.8
= -7.600000
=*-2
= 15.200000
= *-2
= -30.400000
= +2
= -28.400000
quit
```

7.7 小结

本章涵盖了许多基础知识，详细探讨了指针。读者现在应该了解指针和数组(一维和多维数组)间的关系了，并掌握了它们的用法。本章介绍了动态分配内存的函数 malloc()、calloc()和 realloc()，它们给程序提供了足够的内存，以执行数据处理。函数 free()用来释放先前由 malloc()、calloc()和 realloc()分配的内存。读者应该很清楚如何给字符串使用指针，以及如何使用指针数组。

本章讨论的主题是本书以后许多章节的基础，对编写 C 程序是很有帮助的，所以在进入下一章之前，一定要熟练掌握这些内容。下一章将介绍程序的结构。

7.8 习题

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容。还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，

www.TopSage.com

但这应是最后一种方法。

习题 7.1 编写一个程序，计算从键盘输入的任意个浮点数的平均值。将所有的数存储到动态分配的内存中，之后计算并显示平均值。用户不需要事先指定要输入多少个数。

习题 7.2 编写一个程序，从键盘读入任意个谚语，并将它们存储到执行期间分配的内存中。然后，将它们以字长顺序由短到长地输出。

习题 7.3 编写一个程序，从键盘读入一个字符串，显示删除了所有空格和标点符号的字符串。所有的操作都使用指针完成。

习题 7.14 编写一个程序，读入任意天数的浮点温度记录值，每天有 6 个记录。温度记录存储在动态分配内存的数组中，数组的大小刚好等于输入的温度数。计算出每天的平均温度，然后输出每天的记录，在单独一行上输出平均值，该平均值精确到小数点后一位。

www.TopSage.com

第 8 章

程序的结构

如第 1 章所述，将程序分成适度的自包含单元是开发任一程序的基本方式。当工作很多时，最明智的做法就是把它分成许多便于管理的部分，使每一部分能很轻松地完成，并确保正确完成整个工作。如果仔细设计各个代码块，就可以在其他程序中重用其中的一些代码块。

C 语言中的一个重要观念是，每个程序都应切割成许多小的函数。前面的所有例子都编写成一个函数 main()，还涉及到了其他函数，因为这些例子还使用各种标准库函数进行输入输出、数学运算和处理字符串。

本章将介绍如何使程序更有效率，利用更多自己的函数更方便地开发程序。

本章的主要内容：

- 数据如何传给函数
- 函数如何返回结果
- 如何定义自己的函数
- 函数使用指针参数的优势

8.1 程序的结构

如概述所言，C 程序是由许多函数组成的，其中最重要的就是函数 main()，它是执行的起点。本书介绍库函数 printf()或 scanf()时，说明了一个函数可以调用另一个函数，完成特定的工作，然后调用函数继续执行。不考虑存储在全局变量中的数据的负面影响，程序中的每个函数都是一个执行特定操作的自包含单元。调用一个函数时，就执行该函数体内的代码，这个函数执行结束后，控制权就回到调用该函数的地方。如图 8-1 所示，C 程序由 5 个函数组成时的执行顺序，它并未显示任何语句细节。

这个程序以正常的方式按顺序执行语句，当遇到调用一个函数的语句时，就从该函数的起始点开始执行，即该函数体的第一个语句。这个函数会一直执行，在遇到 return 语句或到达这个函数体的结束括号时，就返回调用它的那个位置之后。

这些组成程序的函数通过函数调用及其 return 语句链接在一起，完成各种工作，以达到程序的目标。图 8-1 中的每个函数在程序中只执行一次。实际上，每个函数可以执行多次，且可以从程序中多个地方调用。前面的例子中就多次调用函数 printf()和 scanf()。

TopSage.com

在详细了解如何定义自己的函数之前，必须解释变量的一个重要方面，这个方面一直未提及。

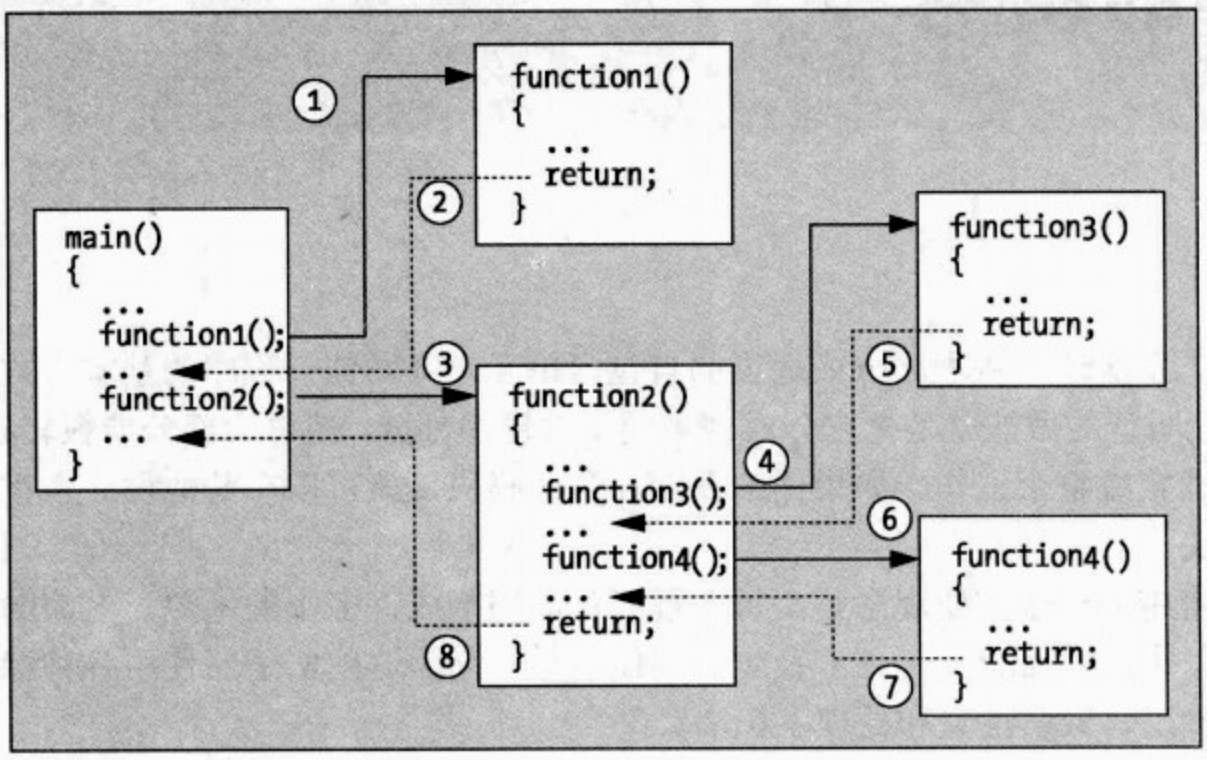


图 8-1 由 5 个函数组成的程序的执行过程

8.1.1 变量的作用域和生存期

在前面所有的例子中，都是在定义 main()函数体的起始处声明程序的变量。事实上，可以在任何代码块的起始处定义变量。这有什么不同吗？这是绝对不同的。变量只存在于定义它们的块中。它们在声明时创建，在遇到下一个闭括号时就不存在了。

在一个块内的其他块中声明的变量也是这样。在外部块的起始处声明的变量也存在于内部块中。这些变量可以随意使用，只要内部块中没有同名的变量即可。

变量在一个块内声明时创建，在这个块结束时删除，这种变量称为自动变量，因为它们是自动创建和删除的。给定变量可以在某个程序代码块中访问和引用，这个程序代码块称为变量的作用域。在作用域内使用变量是没问题的。但是如果试图在变量的作用域外部引用它，编译程序时就会得到一条错误信息，因为这个变量在它的作用域之外不存在。例如下面的代码：

```
{
  int a = 0;                                  /* Create a */
  /* Reference to a is OK here */
  /* Reference to b is an error here */
  {
    int b = 10;                               /* Create b */
    /* Reference to a and b is OK here */
  }                                           /* b dies here */
  /* Reference to b is an error here */
  /* Reference to a is OK here */
}                                             /* a dies here */
```

www.TopSage.com

在一个块内声明的所有变量，在这个块的结束括号之后，它们就不再存在。变量 a 可在内外两个块内访问，因为它是在外部块中声明的。变量 b 只能在内部块中访问，因为它是在内部块中声明的。

执行程序时，会创建变量，并给它分配内存。有时，在一个块中声明了自动变量，在这个块结束时，该变量占用的内存就会返回给系统。当然，在执行块内调用的函数时，变量仍存在，只有执行到创建变量的块的尾部时，才删除变量。变量存在的时间称为变量的生存期。

下面通过一个例子来了解变量作用域的含义。

试试看：了解作用域

下面的简单例子在循环体中包含一个嵌套块：

```
/* Program 8.1 A microscopic program about scope */
#include <stdio.h>

int main(void)
{
  int count1 = 1;        /* Declared in outer block */
  do
  {
    int count2 = 0;      /* Declared in inner block */
    ++count2;
    printf("\ncount1 = %d count2 = %d", count1,count2);
  } while( ++count1 <= 8 );

  /* count2 no longer exists */

  printf("\ncount1 = %d\n", count1);
  return 0;
}
```

这个程序的输出如下所示：

```
count1 = 1    count2 = 1
count1 = 2    count2 = 1
count1 = 3    count2 = 1
count1 = 4    count2 = 1
court1 = 5    count2 = 1
count1 = 6    count2 = 1
count1 = 7    count2 = 1
count1 = 8    count2 = 1
count1 = 9
```

代码的说明

包含 main()函数体的块内，有一个内部块，即 do-while 循环。在这个循环块内声明且定义了 count2:

```
do
{
  int count2 = 0;     /* Declared in inner block */
  ++count2;
  printf("\ncount1 = %d count2 = %d", count1,count2);
} while( ++count1 <= 8 );
```

因此，count2 的值永远不超过 1。在循环的每次迭代中，都创建、初始化、递增和删除 count2 变量。它只存在于从声明它的语句到这个循环的结束括号为止。另一方面，变量 count1 位于 main()块中。当递增它时，它仍然存在，所以最后的 printf()输出 9。

修改这个程序，使最后的 printf()输出 count2 的值。此时程序不能编译，而是得到一条错误信息，因为在执行最后的 printf()时，count2 已经不存在了。由此可以看出，在使用之前没有初始化自动变量，会导致混乱，因为它们所占的内存可能在它们不再存在后重新分配给其他变量。结果，下一次执行程序时，未初始化的变量可能含有我们不希望的内容。

试试看：深入了解作用域

稍微修改一下上一个程序：

```
/* Program 8.2 More scope in this one */
#include <stdio.h>

int main(void)
{
  int count = 0;   /* Declared in outer block */
  do
  {
    int count = 0; /* This is another variable called count */
    ++count;       /* this applies to inner count */
    printf("\ncount = %d ", count);
  }
  while( ++count <= 8 ); /* This works with outer count */
                         /* Inner count is dead, this is outer */
  printf("\ncount = %d\n", count);
  return 0;
}
```

现在，main()块和循环块使用相同的变量名称 count。编译并执行程序的结果如下：

```
count = 1
count = 1
count = 1
count = 1
count = 1
count = 1
count = 1
```

www.TopSage.com

```
count = 1
count = 9
```

代码的说明

这个输出很无聊，也很有趣。事实上有两个叫 count 的变量，但是在循环块的内部，本地会掩盖 main()块中的 count。当使用名称 count 时，编译器会假设使用的是当前块中声明的那个变量。在 while 循环内，只有 count 的本地版本，所以递增这个变量。循环块内的 printf()显示本地变量 count 的值，它永远是 1，原因之前已解释过了。一旦退出这个循环，外部的 count 就可以访问了，最后一个 printf()显示它在循环结束时的值 9。

很明显，控制循环的变量是在 main()开始时声明的那个变量。这个小例子演示了为什么最好不要在一个函数中对两个不同的变量使用相同的名称，但这是合法的。在最好的情况下，它会混淆视听。在最坏的情况下，会遇到大麻烦。

8.1.2 变量的作用域和函数

在讨论创建函数的细节之前，最后要讨论的是，每个函数体都是一个块(当然，它可能含有其他块)。因此，在一个函数内声明的自动变量是这个函数的本地变量，在其他地方不存在。所以在一个函数内部声明的变量完全独立于在其他函数内声明的变量。可以在不同的函数内使用相同的变量名称，它们是完全独立的。

处理很大的程序时，这很重要，因为此时确保变量的唯一性是比较困难的。当然，最好避免在不同的函数中使用不必要或易引起误解的相同的变量名。应该尽量使用便于理解程序的名称。在 C 里深入探讨 C 语言的函数时，将介绍这方面的更多内容。

8.2 函数

本书的程序广泛使用了内置函数，例如 printf()或 strcpy()。还介绍了在按名称引用内置函数时如何执行它们，如何通过函数名称后括号内的参数，给函数传递信息。例如 printf()函数的第一个参数通常是一个字符串，其后的参数(可能没有)是一系列变量或要显示其值的表达式。

可以通过两种方法接收函数返回的信息。第一种方法是使用函数括号内的一个参数。通过函数的一个参数提供变量的地址，这个函数会在该变量中放置一个值。例如，使用 scanf()从键盘上读取数据时，输入会存储到一个作为参数提供的地址中。第二种方法是通过返回值接收函数传回的信息。例如对于 strlen()函数，当调用该函数时，该函数会返回程序代码作为参数传送给它的字符串的长度。因此，如果在表达式 2*strlen(str)中，str 是字符串"example"，strlen()函数返回的值就是 7，接着用值 7 取代表达式中的函数调用。因此，这个表达式是 2*7。函数会返回一个特定类型的值，所以在能使用该类型变量的任意表达式中，都可以使用函数调用，作为表达式的一部分。

www.TopSage.com

所有的程序都必须编写函数 main()，所以我们已经具备函数构成的基本知识。下面详细讨论函数的构成。

8.2.1 定义函数

创建一个函数时，必须指定函数头作为函数定义的第一行，跟着是这个函数放在括号内的执行代码。函数头后面放在括号内的代码块称为函数体。

- 函数头定义了函数的名称、函数参数(换句话说，即调用函数时传给函数的值的类型)和函数返回值的类型。
- 函数体决定函数对传给它的值执行什么操作。

函数的一般形式和 main()相同，如下所示：

```
Return_type Function_name( Parameters - separated by commas )
{
  Statements;
}
```

函数体内可以没有语句，但是大括号必须有。如果函数体内没有语句，返回类型必须是 void，此时函数没有任何作用。void 类型表示“不存在任何类型”，所以这里它表示函数没有返回值。没有返回值的函数必须将返回类型指定为 void，而返回类型不是 void 的函数都在函数体中有一个 return 语句，返回一个指定返回类型的值。

没有函数体的函数通常在复杂程序的测试阶段使用。这允许在执行程序时，只用选定的函数执行一些操作，然后逐步增加函数体中的代码，直到完成整个工作。

括号中的参数是变元值的占位符，调用一个函数时，必须指定参数的值。术语“参数”表示函数定义中的一个占位符，指定了调用函数时传送给函数的值的类型。参数包含在函数体内用来表示函数执行时使用的数据类型和名称。术语“变元”表示调用函数时提供的对应于参数的值。本章后面的“函数的参数”一节将详细介绍参数。

注意：

函数体内的语句也可能含有嵌套的语句块。但不能在一个函数体内定义另一个函数。

调用函数的一般形式是：

```
Function_name(List of Arguments - separated by commas)
```

使用函数名后跟括号内一连串以逗号分隔的变元，与调用 printf()和 scanf()函数一样。函数调用可以显示在一行中，如下：

```
printf("A fool and your money are soon partners,");
```

像这样调用的函数可以是有返回值的函数。在这个例子中，返回值会被丢弃。返回类型定义为 void 的函数只能这样调用。有返回值的函数通常会出现在表达式中，例如：

www.TopSage.com

```
result= 2.0*sqrt(2.0);
```

sqrt()函数(在头文件<math.h>中声明)返回的值乘以 2.0，结果存储到变量 result 中。很明显，返回类型为 void 的函数不返回任何值，所以不可能成为表达式的一部分。

1. 函数的命名

在 C 语言中，函数的名称可以是任何合法的名字，但不能是保留字(如 int、double 和 sizeof 等)，也不能和程序中其他函数的名称相同。注意，不要使用与任何标准库函数相同的名称，以避免混淆。

区别自己的函数和标准库函数的一个方法是，函数名用一个大写字母开头，但一些程序员觉得这相当受限。合法的函数名与变量名的形式相同：一串字母和数字，第一个必须是字母。与变量名称一样，下划线字符算是一个字母。除此之外，函数的名称可以任意，但是最好能说明函数的作用。有效的函数名称示例如下：

```
cube_root  FindLast  Explosion  Back2Front
```

通常，将函数名称(和变量名称)定义为包含多个单词。有两个常见的方法可以采用：

- 在函数名称中用下划线分开每个单词。
- 将每个单词的第一个字母大写。

这两种方法都很好，采用哪一个取决于程序员，但最好在选择了一种方法后就固定使用它。当然，可以对函数使用一种方法，对变量使用另一种方法。在本书中这两种方法都有使用，阅读完本书后，读者就会对使用哪种方法有自己的看法了。

2. 函数的参数

函数的参数在函数头中定义，是调用函数时必须指定的变元的占位符。函数的参数是一列变量名称和它们的类型，参数之间以逗号分隔。整个参数列表放在函数名称后的括号中。

参数提供了调用函数给被调用函数传递信息的方法。这些参数名对于函数而言是本地的，调用函数时给它们指定的值称为“变元”。然后使用这些参数在函数体中编写计算操作，当函数执行时，参数使用变元的值。当然，函数也可以在函数体中声明本地定义的自动变量。最后当计算完成时，退出函数，将一个适当的值返回给原来的调用语句。

函数头的示例如表 8-1 所示。

表 8-1 函数头示例

函 数 头	说 明
bool SendMessage(char *text)	该函数有一个参数 text，其类型是 char 指针类型，该函数返回一个 bool 类型的值
void PrintData(int count, double *data)	该函数有两个参数，一个是 int 类型，另一个是 double 指针类型，该函数没有返回值

www.TopSage.com

(续表)

函 数 头	说 明
char message GetMessage(void)	这个函数没有参数，返回一个 char 类型的指针

调用函数时，要使用函数名称，后跟放在括号中的变元。在程序的某部分引用函数以调用它，在调用时指定的变元会取代函数中的参数。因此，函数执行时，会使用为变元提供的值进行计算。调用函数时指定的变元的类型、个数和顺序必须和函数头中指定的参数一致。调用函数和被调用函数的关系与传送的信息如图 8-2 所示。

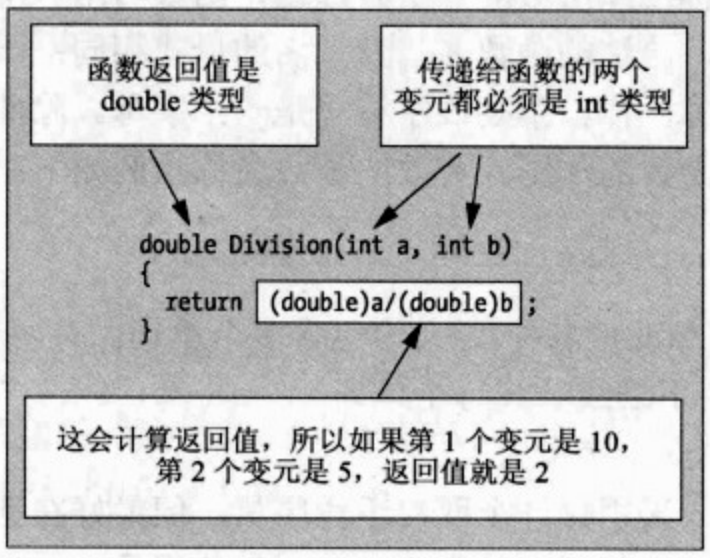


图 8-2 给参数传送变元

如果函数的变元类型不匹配对应参数的类型，编译器就会插入一个类型转换操作，将变元值的类型转换为参数的类型。这可能会截断变元值，例如把 double 类型的值传送给 int 类型的参数，所以这是一个危险的操作。如果编译器不能把变元转换为需要的类型，就会得到一条错误信息。

如果函数没有参数，就可以把参数列表指定为 void，如 main()函数所示。

3. 指定返回值的类型

另一个常见的函数形式如下：

```
Return_type Function_name(List of Parameters - separated by commas)
{
  Statements;
}
```

Return_type 指定了函数返回值的类型。如果在表达式中使用函数，或函数在赋值语句的右侧使用，则函数的返回值会取代该函数。函数的返回值可以指定为 C 语言中任何合法的类型，包括指针。该类型也可以指定为 void，表示没有返回值。如前所述，返回类型为 void 的函数不能用在表达式中，或用在赋值语句中的任何地方。

返回类型也可以是 void*，表示指向 void 的指针。此时，返回值是一个地址值，但没有指定类型。希望返回一个能灵活用于各种目的的指针时，就可以使用这个类型，例

www.TopSage.com

如分配内存的 malloc()函数。最常见的返回类型如表 8-2 所示。

表 8-2 常见的返回类型

类 型	含 义	类 型	含 义
int	整数，2 个或 4 个字节	int *	指向 int 的指针
short	整数，2 个字节	short*	指向 short 的指针
long	整数，4 个字节	long*	指向 long 的指针
long long	整数，8 个字节	long long*	指向 long long 的指针
char	字符，1 个字节	char*	指向 char 的指针
float	浮点数，4 个字节	float*	指向 float 的指针
double	浮点数，8 个字节	double*	指向 double 的指针
long double	浮点数，12 个字节	long double*	指向 long double 的指针
void	无返回值	void*	指向 void 的指针

当然，也可以把函数的返回类型指定为无符号的整数类型或指向无符号整数类型的指针。返回类型也可以是枚举类型或指向枚举类型的指针。如果函数的返回类型指定成 void 以外的类型，它就必须返回一个值。执行 return 语句可以返回这个值。

注意：

第 11 章将介绍 structs 对象，它提供了把几个数据项作为一个单元来处理的方式。函数的参数可以是结构或指向结构的指针，也可以返回一个结构或指向结构的指针。

8.2.2 return 语句

return 语句允许退出函数，从调用函数中发生调用的那一点继续执行。return 语句最简单的形式如下：

```
return;
```

这个形式的 return 语句用于返回类型声明为 void 的函数，它不返回任何数值。然而，较常见的 return 语句形式是：

```
return expression;
```

这个形式的 return 语句必须用于返回类型没有声明为 void 的函数，返回给调用程序的数值是计算 expression 的结果。

警告：

如果函数的返回类型定义为 void，却试图返回一个值，编译程序时就会得到一条错误信息。如果返回类型定义为除 void 之外的其他类型的函数，没有使用 return，编译器也会生成一条错误信息。

www.TopSage.com

返回语句的 expression 可以是任何表达式，但是它生成的值的类型应该和函数头声明的返回值相同。如果它们的类型不同，编译器会将 return expression 的类型转换成需要的类型(如果可能的话)，如果不能转换，编译器就生成一条错误信息。

一个函数中可能有多个 return 语句，但每个 return 语句都必须提供一个可以转换为函数头中为返回值指定的类型的值。

注意：

调用函数不必识别或处理被调用函数返回的值。程序员负责确定如何使用函数调用的返回值。

试试看：使用函数

使用例子总是更容易理解新的概念，所以下面演示一个包含两个函数的程序。编写一个函数，计算两个浮点变量的平均值，并在 main()中调用这个函数。这个例子主要说明编写和调用函数的机制，而不是说明它们的用法。

```
/* Program 8.3 Average of two float values */
#include <stdio.h>

/* Definition of the function to calculate an average */
float average(float x, float y)
{
  return (x + y)/2.0f;
}

/* main program - execution always starts here */
int main(void)
{
  float value1 = 0.0F;
  float value2 = 0.0F;
  float value3 = 0.0F;

  printf("Enter two floating-point values separated by blanks: ");
  scanf("%f %f", &value1, &value2);
  value3 = average(value1, value2);
  printf("\nThe average is: %f\n", value3);
  return 0;
}
```

程序的输出如下：

```
Enter two floating-point values separated by blanks: 2.34 4.567

The average is: 3.453500
```

代码的说明

下面一步步地讨论这个例子。图 8-3 显示了这个例子的执行顺序。

www.TopSage.com

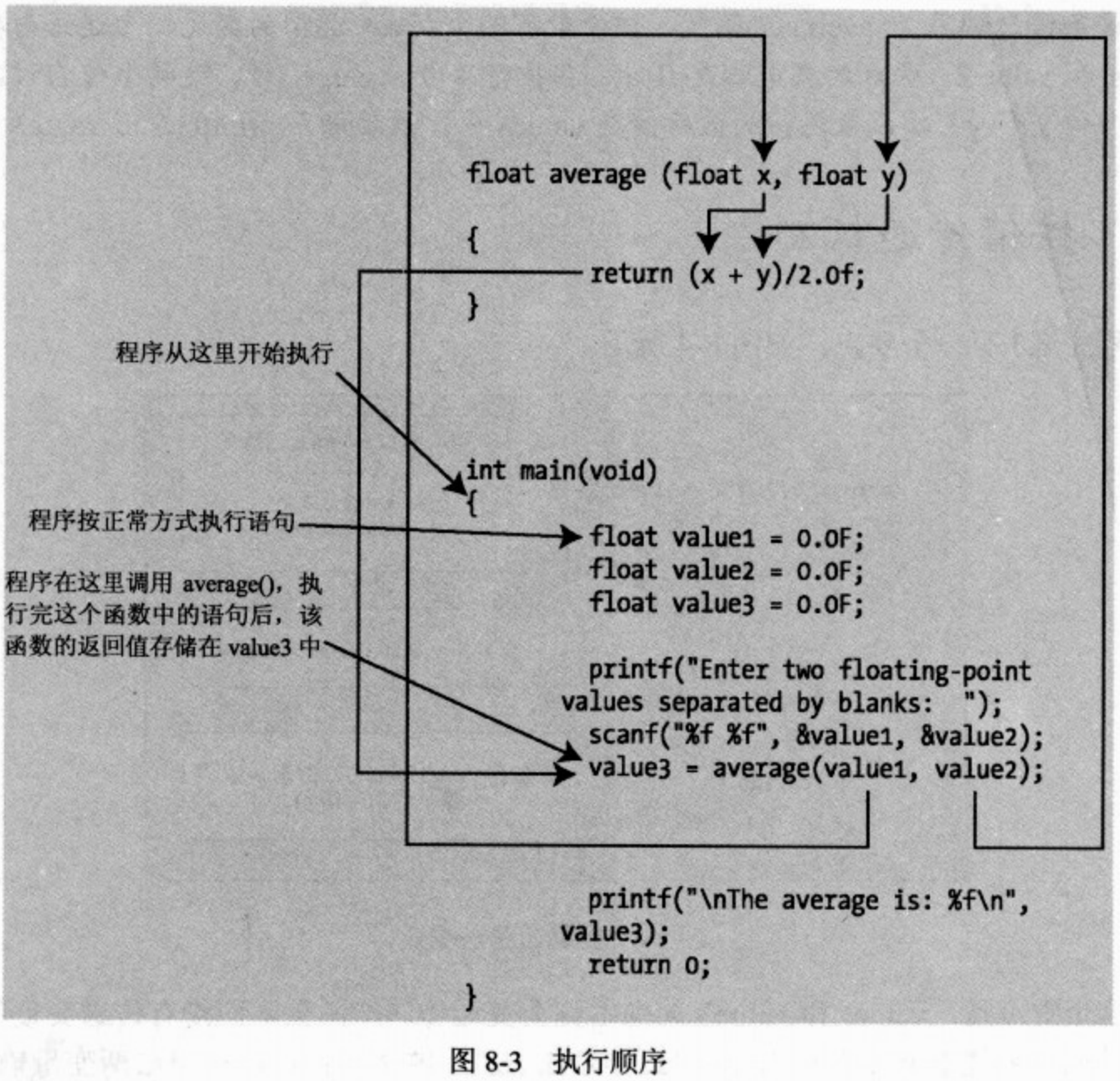


图 8-3 执行顺序

程序从 main()函数的第一个可执行语句开始。main()函数体的第一个语句是:

```
printf("Enter two floating-point values separated by blanks: ");
```

这里面没什么新东西，用一个字符串变元调用函数 printf()。传送给 printf()的值是一个含有字符串起始地址的指针，该字符串是指定的变元。然后，printf()函数显示作为变元提供的字符串。

下一条语句也很熟悉:

```
scanf("%f %f", &value1, &value2);
```

这条语句调用输入函数 scanf()。这个函数有 3 个变元，第一个变元是一个字符串，它是一个有效的指针，如同前一条语句一样。第二个变元是第一个变量的地址，它也是一个有效的指针。第三个变元是第二个变量的地址，它同样是一个有效的指针。如前所述，scanf()必须有最后两个变元的地址，才能把输入的数据存储到它们中。本章稍后会说明原因。

读入这两个值后，就执行赋值语句:

```
value3 = average(value1, value2);
```

www.TopSage.com

这条语句调用了 average()函数，该函数有两个 float 类型的变元，上述语句提供了 value1 和 value 2，两者的类型都是 float。在执行函数 average()时，这两个值在该函数体中被当作 x 和 y。该函数返回的值存储在 value3 中。最后调用 printf()输出 value3 的值。

8.3 按值传递机制

程序 8.3 有一个要点，如图 8-4 所示。

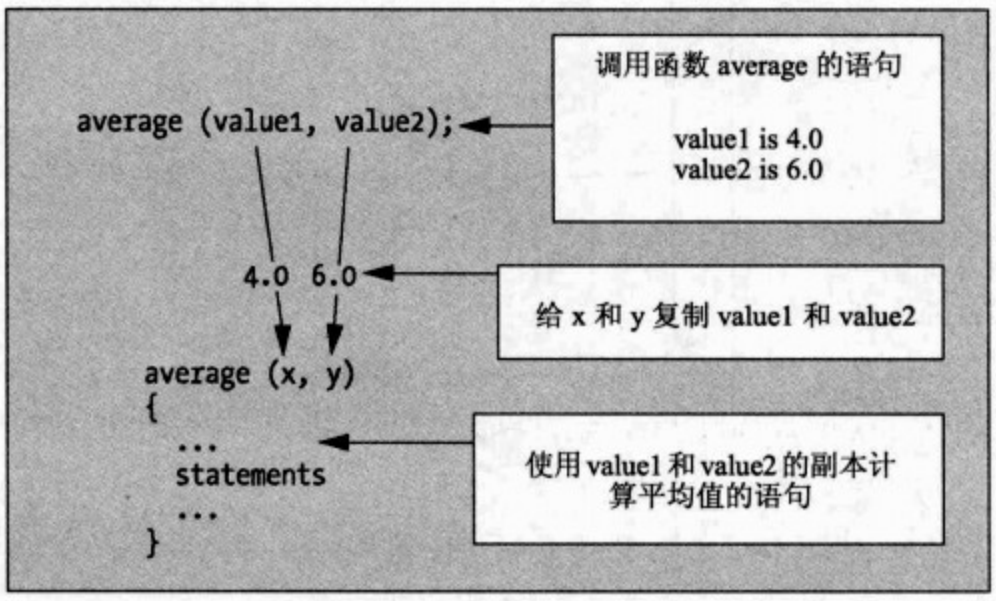


图 8-4 给函数传送变元

这个重点是：value1 和 value2 的副本作为变元传送给函数，而没有传送变量本身。也就是说，函数不能改变存储在 value1 或 value2 中的值。例如，如果给两变量输入 4.0 和 6.0，编译器会在堆栈上创建这两个值的副本，在调用 average()函数时，average()函数会访问这些副本。这个机制是 C 语言中给函数传送变元值的方式，称为按值传递(pass-by-value)机制。

被调用函数要改变属于调用函数的变量，只能将变量的地址作为接收的变元值。将地址作为变元传递给函数时，它还是传给函数地址的副本，而不是原始的地址。然而，该副本仍然是原始变量的地址。本章的“将指针用作变元和返回值”一节将讨论这一点。

执行 average()函数时，要使用 value1 和 value2 的值来取代函数中的参数 x 和 y。这个函数用下面的语句定义：

```
float average(float x, float y)
{
  return(x + y)/2.0f;
}
```

函数体只有一条语句，即 return 语句。但是该 return 语句完成需要的工作。它包含的表达式计算了 x 和 y 的平均值。然后，这个值替代了 main()里赋值语句中的 average()函数。所以实际上，返回一个值的函数的作用与和返回值类型相同的变量一样。

注意，如果常量 2.0 没有 f，这个值就是 double 类型，整个表达式应计算为 double

www.TopSage.com

类型。接着，编译器会将它转换为返回值的 float 类型，在编译过程中会输出一条警告信息，因为值从，double 类型转换成 float 类型时，可能会遗失数据。

如果不将函数的结果赋予 value3，可以将最后的 printf()语句改为：

```
printf("\nThe average is: %f", average(value1, value2));
```

这里函数 average()接收变元值的副本。这个函数的返回值会作为变元直接提供给 printf()，而没有创建一个变量来存储它。编译器会分配一些内存来存储 average()返回的值。它也会把这个副本作为变元传送给函数 printf()。但一旦 printf()执行完毕，就无法访问 average()函数的返回值了。

另一个选择是在函数调用中使用其返回值。现在重写前文的 printf()语句：

```
printf("\nThe average is: %f", average(4.0, 6.0));
```

如果 4.0 和 6.0 不存在，编译器一般会创建一个内存位置，以存储 average()的每个常量变元，然后给函数提供这些数值的副本(和之前一样)。然后，将函数 average()返回值的副本传递给 printf()。

图 8-5 说明了从 main()中调用 useful()函数，得到返回值的过程。

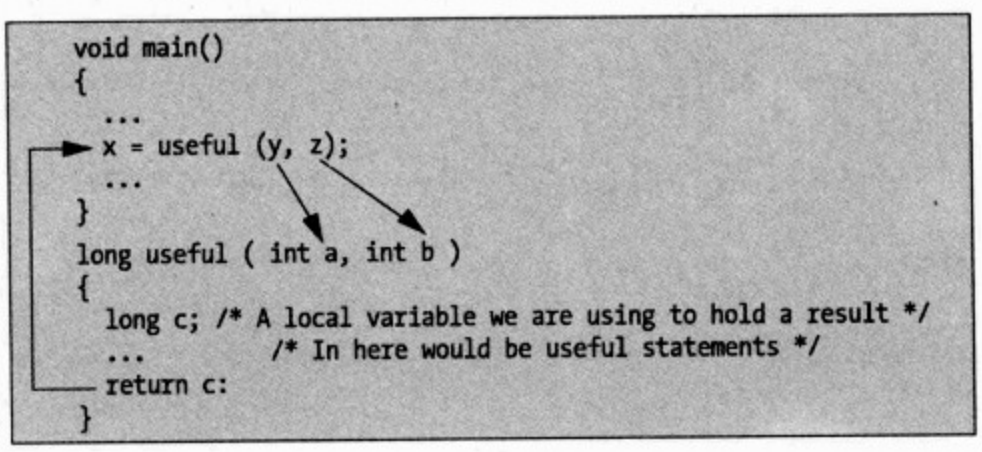


图 8-5 传送的变元和返回值

图中的箭头表示函数 main()和 useful()的对应值。注意，useful()函数返回的值最后存储在 main()函数的变量 x 中；useful()函数返回的值存储在该函数的本地变量 c 中。useful()函数执行完时，这个变量也就不存在了，但是返回值仍然存在，因为返回的是值的副本本，而不是原来存储在 c 里的值。因此，从函数返回一个值和传递变元值的方式是相同的。

8.4 函数声明

在程序 8.3 的变体中，先定义 main()函数，再定义 average()函数：

```
#include <stdio.h>

int main(void)
{
```

www.TopSage.com

```
  /* Code in main() ···*/
}

float average(float x, float y)
{
  return (x + y)/2.0;
}
```

这段代码不会编译。编译器在遇到 average()函数的调用时，不知道该如何处理，因为那时 average()函数还没有声明。为了编译这段代码，必须在 main()的定义之前添加代码，告诉编译器 average()函数的信息。

函数声明是一个定义函数基本特性的语句，它定义了函数的名称、返回值的类型和每个参数的类型。事实上，可以将它编写的和函数头一模一样，只是要在尾部加一个分号。函数声明也叫做函数原型，因为它提供了函数的所有外部规范。函数原型能使编译器在使用这个函数的地方创建适当的指令，检查是否正确地使用它。在程序中包含头文件时，这个头文件就会在程序中为库函数添加函数原型。例如，头文件<stdio.h>含有 printf()和 scanf()的函数原型。

为了使程序 8.3 的变体可以编译，只需在函数 main()的定义前面添加 average()函数的原型：

```
#include <stdio.h>
float average(float, float); /* Function prototype */

int main(void)
{
  /* Code in main() ··· */
}

float average(float x, float y)
{
  return (x + y)/2.0;
}
```

现在，编译器可以编译 main()中的 average()函数调用了，因为编译器知道该函数的所有特性，例如名称、参数类型和返回类型。在技术上，可以把 average()函数的声明放在 main()函数体中，只是 average()函数的声明必须放在该函数的调用之前，但事实上这种做法并不可行。函数原型一般放在源文件的开头处，而且在所有函数的定义之前。另外，在源文件中，函数原型在所有函数的外部，函数的作用域是从其声明处开始一直到源文件的结尾。因此无论函数的定义放在什么地方，源文件中的任意函数都可以调用该文件中的其他函数。

无论在什么地方调用，最好总是把函数的声明放在程序的源文件中。这有助于程序与设计保持一致，还可以防止在程序的另一部分调用函数时出错。当然，main()函数不需要函数原型，因为在程序开始执行时，这个函数会由主机环境调用。

www.TopSage.com

8.5 指针用作参数和返回值

前面介绍了如何将指针作为变元传递给函数。另外，如果函数修改在调用函数中定义的变量值，也需要使用指针变元。事实上这是唯一的方法。下面用另一个实例来探讨。

试试看：使用初始变量的函数

首先介绍一个不使用指针变元的函数例子。该函数试图改变变量的内容，方法是把变量当成变元传给函数，改变它，然后返回它。最后输出函数中变量的值及它返回 main() 后的值。

```
/* Program 8.4 The change that doesn't */
#include <stdio.h>

int change(int number); /* Function prototype */

int main(void)
{
  int number = 10;       /* Starting Value */
  int result = 0;        /* Place to put the returned value */
  result = change(number);
  printf("\nIn main, result = %d\tnumber = %d", result, number);
  return 0;
}

/* Definition of the function change() */
int change(int number)
{
  number = 2 * number;
  printf("\nIn function change, number = %d\n", number);
  return number;
}
```

程序的输出如下：

```
In function change, number = 20
In main, result = 20 number = 10
```

代码的说明

这个例子说明，没有指针，就不能改变变量，只能改变该变量在函数内的值。在这个例子中，首先要注意，函数 change()的原型和#include 指令一起，放在 main()外部：

```
#include <stdio.h>

int change(int number); /* Function prototype */
```

这使函数声明为全局函数，如果这个例子有其他函数，它们将可以使用这个函数。

在 main()函数中，用初始值 10 建立一个整数变量 number。第二个变量 result 用来存

www.TopSage.com

储从函数 change()返回的值。

```
int number = 10; /* Starting Value */
int result = 0; /* Place to put the returned value */
```

然后，调用函数 change()，给它传递变量 number 的值:

```
result = change(number);
```

函数 change()的定义如下:

```
int change(int number)
{
  number = 2 * number;
  printf("\nIn function change, number = %d\n", number);
  return number;
}
```

在 change()函数体内，第一条语句把从 main()函数传进来的变元值翻倍。这里甚至使用和 main()函数中相同的变量名，来加强要改变原始值的观念。change()函数显示 number 的新值，之后使用 return 语句将它返回 main()函数。

在 main()函数中也显示 change()返回的值和 number 的值:

```
printf("\nIn main, result = %d\tnumber = %d", result, number);
```

从输出可以看出，这里将变量传递给函数，试图以此改变变量的值。很明显，change()函数中变量 number 的值是 20。它显示在 change()函数中，而在 main()函数中显示为返回值。尽管两个变量的名称相同，但 main()和 change()函数中的两个变量 number 是完全独立的，改变其中一个并不会影响另一个。

试试看：在函数中使用指针

现在使用指针修改上一个例子，应该可以成功地修改 main()中的变量值。

```
/* Program 8.5 The change that does */
#include <stdio.h>

int change(int *pnumber); /* Function prototype */

int main(void)
{
  int number = 10;          /* Starting Value */
  int *pnumber = &number;   /* Pointer to starting value */
  int result = 0;           /* Place to put the returned value */

  result = change(pnumber);
  printf("\nIn main, result = %d\tnumber = %d", result, number);
  return 0;
}
```

www.TopSage.com

```
/* Definition of the function change() */
int change(int *pnumber)
{
  *pnumber *= 2;
  printf("\nIn function change, *pnumber = %d\n", *pnumber );
  return *pnumber;
}
```

程序输出如下:

```
In function change, *pnumber = 20
In main, result = 20 number = 20
```

代码的说明

本程序对上一个例子几乎没做什么改变，只是在 main()函数中定义了一个指针 pnumber，把它初始化成变量 number 的地址，number 包含了起始值:

```
int change(int *pnumber); /* Function prototype */
```

修改函数 change()的原型，把参数指定为一个指针:

```
int change(int *pnumber); /* Function prototype */
```

change()函数的定义也改为使用指针:

```
int change(int *pnumber)
{
  *pnumber *= 2;
  printf("\nIn function change, *pnumber = %d\n", *pnumber );
  return *pnumber;
}
```

指针 pnumber 和 main()中的指针同名，然而这对程序的运作是无足轻重的。给它指定什么名称都可以，只要它是一个正确类型的指针，因为这是 change()函数的一个本地变量。

在函数 change()中，算术语句改成:

```
*pnumber* = 2;
```

使用*=运算符严格说来不是必要的，但如果知道*=的含义，这行语句可减少混乱。它和下面的语句完全相同:

```
*pnumber = 2*(*pnumber);
```

输出证明了指针机制可以正确运作，函数 change()的确修改了函数 main()中的 number 变量值。当然，将一个指针作为变元传送时，仍然是按值传递。因此，编译器不传递原来的指针，而是将指针变量中的地址副本传送给函数。因为这个副本含有和原来相同的地址，仍然引用变量 number，所以所有的操作都正常。

如果怀疑这点，可以证明如下：在函数 change()中添加一行语句，以修改指针

www.TopSage.com

pnumber。例如，将它设成 NULL。然后，在 main()函数中检查 pnumber 是否仍然指向 number。当然，必须改变 change()函数中的 return 语句，才能得到正确的结果。

8.5.1 常量参数

可以使用 const 关键字修饰函数参数，这表示函数将传送给参数的变元看做一个常量。由于变元是按值传送的，所以只有参数是一个指针时，这个关键字才有效。一般将 const 关键字应用于指针参数，指定函数不修改该指针指向的值。换言之，函数体中的代码不修改指针变元指向的值。下面是带一个 const 参数的函数示例：

```
bool SendMessage(const char* pmessage)
{
  /* Code to send the message */
  return true;
}
```

编译器将确认函数体中的代码没有使用 pmessage 指针修改信息文本。也可以把指针本身指定为 const，但这没有意义，因为地址是按值传送的，所以不能改变调用函数中的原始指针。

将指针参数指定为 const 有另一个用途。const 修饰符表示，函数不修改指针指向的数据，编译器知道，指向常量数据的指针变元应是安全的。另一方面，如果不给参数使用 const 修饰符，对编译器而言，函数就可以修改变元指向的数据。将指向常量数据的指针作为变元传送给未声明为 const 的参数时，C 编译器至少应给出一条警告信息。

提示：

如果函数不修改指针参数指向的数据，就把该函数参数声明为 const。这样，编译器就会确认，函数的确没有改变该数据。将指向常量的指针传送给函数时，还可以避免出现警告或错误信息。

要将指针包含的地址声明为 const，应在指针类型声明中将 const 关键字放在“*”的后面。下面的代码包含了两个指针声明的例子，说明了指向常量的指针和常量指针之间的区别：

```
int value1 = 99;
int value2 = 88;

const int pvalue = &value1;   /* pointer to constant */
int const cpvalue = &value1; /* Constant pointer */

pvalue = &value2;             /* OK: pointer is not constant */
*pvalue = 77;                 /* Illegal: data is constant */

cpvalue = &value2;            /* Illegal: pointer is constant */
*cpvalue = 77;                /* OK: data is not constant */
```

www.TopSage.com

如果希望 SendMessage()函数的参数是一个常量指针，应编写下面的代码：

```
bool SendMessage(char *const pmessage)
{
  /* Code to send the message */
  /* Can change the message here */
  return true;
}
```

现在，函数体可以修改信息，但不能修改 pmessage 中的地址。如前所述，pmessage 包含地址的副本，所以可以使用 const 声明该指针。

试试看：使用指针传输数据

下面用更实际的方式练习使用指针给函数传递数据的方法，并复习第 7 章中排序字符串函数的修改版本。这里列出了完整的代码，后面将讨论它的细节。源代码除了 main()函数之外，还定义了三个函数。

```
/* Program 8.6 The functional approach to string sorting*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

bool str_in(char **);                    /* Function prototype for str_in */
void str_sort(const char *[], int); /* Function prototype for str_sort */
void swap( void **p1, void **p2);    /* Swap two pointers */
void str_out(char *[], int);             /* Function prototype for str_out */

const size_t BUFFER_LEN = 256;
const size_t NUM_P = 50;

/* Function main - execution starts here */
int main(void)
{
  char *pS[NUM_P];                       /* Array of string pointers */
  int count = 0;                         /* Number of strings read */

  printf("\nEnter successive lines, pressing Enter at the end of"
         " each line.\nJust press Enter to end.\n");

  for(count = 0; count < NUM_P ; count++) /* Max of NUM_P strings */
    if(!str_in(&pS[count]))              /* Read a string */
      break;                             /* Stop input on 0 return */

  str_sort( pS, count);                  /* Sort strings */
  str_out( pS, count);                   /* Output strings */
  return 0;
}
```

www.TopSage.com

```
/*********************************************************
* String input routine                                   *
* Argument is a pointer to a pointer to a constant *
* string which is const char** *
* Returns false for empty string and returns true *
* otherwise. If no memory is obtained or if there *
* is an error reading from the keyboard, the program *
* is terminated by calling exit(). *
*********************************************************/
bool str_in(char **pString)
{
  char buffer[BUFFER_LEN];     /* Space to store input string */

  if(gets(buffer) == NULL )    /* NULL returned from gets()? */
  {
    printf("\nError reading string.\n");
    exit(1); /* Error on input so exit */
  }

  if(buffer[0] == '\0')        /* Empty string read? */
    return false;

  *pString = (char*)malloc(strlen(buffer) + 1);

  if(*pString == NULL)         /* Check memory allocation */
  {
    printf("\nOut of memory.");
    exit(1);                   /* No memory allocated so exit */
  }

  strcpy(*pString, buffer);    /* Copy string read to argument */
  return true;
}

/*********************************************************
* String sort routine                                    *
* First argument is array of pointers to constant    *
* strings which is of type const char*[].            *
* Second argument is the number of elements in the   *
* pointer array - i.e. the number of strings         *
*********************************************************/
void str_sort(const char *p[], int n)
{
  char *pTemp = NULL;    /* Temporary pointer */
  bool sorted = false;   /* Strings sorted indicator */
  while(!sorted)         /* Loop until there are no swaps */
  {
    sorted = true;       /* Initialize to indicate no swaps */
      for(int i = 0 ; i<n-1 ; i++ )
```

www.TopSage.com

```
      if(strcmp(p[i], p[i + 1]) > 0)
      {
        sorted = false;        /* indicate we are out of order */
        swap(&p[i], &p[i+1]); /* Swap the pointers */
      }
  }
}

/*********************************************
* Swap two pointers                          *
* The arguments are type pointer to void*    *
* so pointers can be any type*.              *
*********************************************/
void swap( void **p1, void **p2)
{
  void *pt = *p1;
  *p1 = *p2;
  *p2 = pt;
}

/*********************************************************
* String output routine                                  *
* First argument is an array of pointers to strings      *
* which is the same as char** *
* The second argument is a count of the number of        *
* pointers in the array i.e. the number of strings       *
*********************************************************/
void str_out(char *p[] , int n)
{
  printf("\nYour input sorted in order is:\n\n");
  for(int i = 0 ; i<n ; i++)
  {
    printf("%s\n", p[i]); /* Display a string */
    free(p[i]);           /* Free memory for the string */
    p[i] = NULL;
  }
  return;
}
```

这个程序的输出如下:

```
Enter successive lines, pressing Enter at the end of each line.
Just press Enter to end.
Mike
Adam
Mary
Steve
```

www.TopSage.com

```
Your input sorted in order is:

Adam
Mary
Mike
Steve
```

代码的说明

这个例子的执行过程和第 7 章的排序例子类似。它看起来很大，实际上注释占据了许多空间。使用许多函数的大程序最好添加一些注释，以便确定每个函数的用途。

源文件全部由函数组成。在程序源文件的开头，定义 main()函数之前，#include 语句包含了要使用的库和函数原型。每个库的使用范围都是从它们出现开始到文件的结尾，因为它们在所有函数的外面定义。因此，它们可用于所有的函数。

这个程序由 4 个函数外加 main()函数组成，函数的原型定义如下：

```
bool str_in(char **);                    /* Function prototype for str_in */
void str_sort(const char *[], int); /* Function prototype for str_sort */
void swap( void **p1, void **p2);    /* Swap two pointers */
void str_out(char *[], int);             /* Function prototype for str_out */
```

str_sort()函数的第一个参数指定为 const，所以编译器会确认该函数体没有改变该参数指向的值。当然，函数定义中的参数也指定为 const，否则代码就不能编译。这里没有指定参数名，在函数原型中没有必要指定参数名，但通常最好指定它们。这个例子省略它们，是为了说明可以省略，但建议在程序中包含它们。也可以在函数原型中使用不同于函数定义中的参数名。

注意：

可以在函数原型中使用较长的解释性名字，而在函数定义中使用较短的名字，以保持代码的简洁。

这个例子中的原型声明函数 str_in()读入所有的字符串，函数 str_sort()排序字符串，函数 str_out()输出排序后的字符串。函数 swap()交换存储在两个指针中的地址。每个函数原型都声明了参数和返回值的类型。

第一个是对 str_in()的声明，它声明参数的类型是 char **，即指向 char 的指针的指针。有点复杂是吗？如果仔细研究这个例子，就会发现其实它很简单。

在 main()函数中，其变元是&pS[i]，即 pS[i]的地址，换句话说，就是指向 pS[i]的指针。而 pS[i]是一个指向 char 的指针。将这些放在一起，类型就声明为 char **，表示指向 char 的指针的指针。必须这么声明它，因为要在函数 str_in()中修改 pS 数组元素的内容。这是 str_in()函数访问 pS 数组的唯一方法。如果在参数类型定义中只使用一个*，只使用 pS[i]作为变元，这个函数就会收到 pS[i]中的内容，那可不是我们希望的。这个机制如图 8-6 所示。

www.TopSage.com

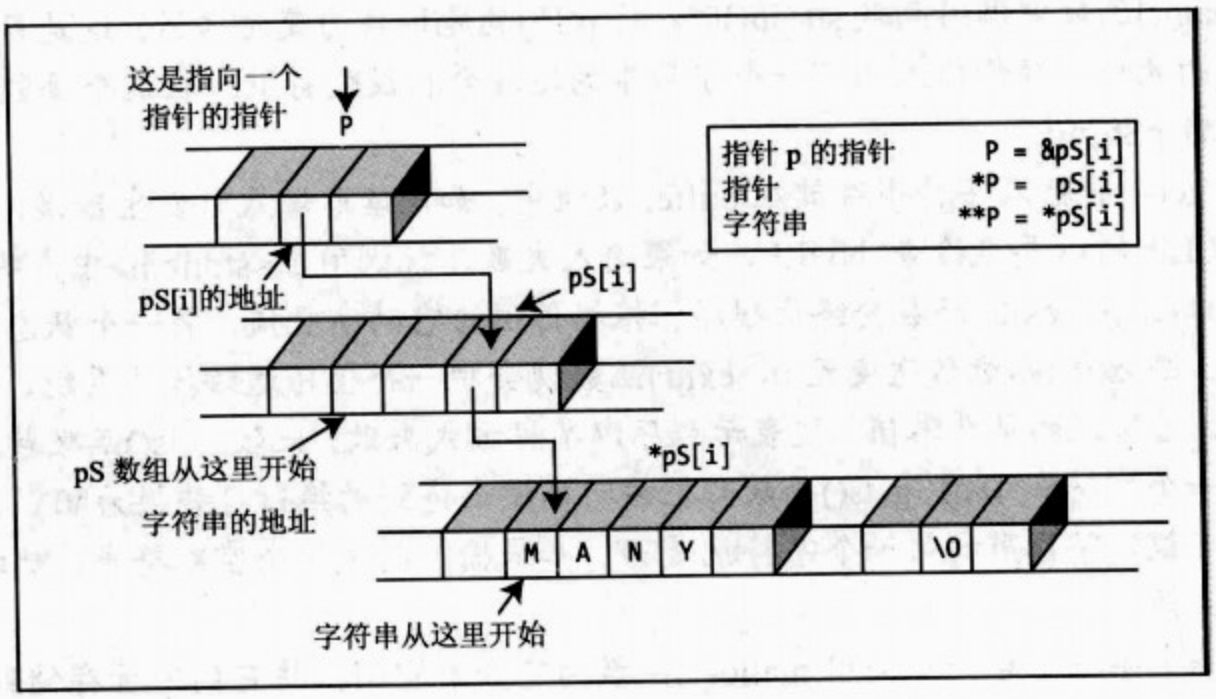


图 8-6 确定指针的类型

当然，类型 const char**与类型 const char*[]相同，const char*[]是 const char*类型的数组。这两种类型都可以使用。

现在看看函数的内部工作。

STR_IN()函数

首先，注意开头的注释。最好在函数的开头处强调它的基本功能。函数定义如下：

```
bool str_in(char **pString)
{
  char buffer[BUFFER_LEN];  /* Space to store input string */

  if(gets(buffer) == NULL)  /* NULL returned from gets()? */
  {
    printf("\nError reading string.\n");
    return false;           /* Read error */
  }

  if(buffer[0] == '\0')     /* Empty string read? */
    return false;

  *pString = (char*)malloc(strlen(buffer) + 1);

  if(*pString == NULL)      /* Check memory allocation */
  {
    printf("\nOut of memory.");
    exit(1);                /* No memory allocated so exit */
  }

  strcpy(*pString, buffer); /* Copy string read to argument */
  return true;
}
```

www.TopSage.com

从 main()函数中调用函数 str_in()时，将 pS[i]的地址作为变元传送。这是目前为空的数组元素的地址，我们输入的下一个字符串的地址会在该处存储。在这个函数中，该地址称为参数 pString。

函数 gets()将输入字符串存储在 buffer 数组中。如果读取输入时发生错误，该函数就返回 NULL，所以要先检查 NULL。如果输入失败，就调用在<stdlib.h>中声明的 exit()函数，终止程序。exit()函数会终止程序，根据传送给它的参数值，将一个状态值返回给操作系统。给 exit()函数传送变元 0，exit()函数就会把一个值传送给操作系统，表示程序成功结束。若传送的是非零值，就表示程序以某种方式失败。比较 gets()函数获取的字符串中的第一个字符和'\0'。 gets()函数会把按回车键而得到的换行符替换为'\0'，所以如果按下了回车键，字符串的第一个字符就是'\0'。如果输入的是一个空字符串，就返回 false 给 main()函数。

读入一个字符串后，就使用 malloc()函数为它分配空间，将它的地址存储到*pString 中。在检查确实分配了内存后，就把 buffer 数组的内容复制到所分配的内存中。如果 malloc()函数分配内存失败，就显示一条信息，调用 exit()函数。

函数 str_in()在 main()的循环中调用:

```
for(count = 0; count < NUM_P ; count++) /* Max of NUM_P strings */
  if(!str_in(&pS[count]))               /* Read a string */
    break;                              /* Stop input on 0 return */
```

所有的工作都已在 str_in()函数中完成，现在只需继续循环，直到从这个函数中得到 false，执行 break 语句为止，或直到 count 到达 NUM_P 的值，这表示指针数组 pS 已填满，因此结束循环。这个循环还在 count 中计算了输入字符串的数目。

安全地存储了所有字符串后，main()函数用下面的语句调用函数 str_sort()，给这些字符串排序:

```
str_sort( pS, count ); /* Sort strings */
```

第一个变元是数组名称 pS，所以把数组第一个位置的地址传给函数。第二个变元是字符串的数目，告诉函数要排序多少个字符串。现在看看 str_sort()函数是如何工作的。

str_sort()函数

函数 str_sort()用下面的语句定义:

```
void str_sort(const char *p[], int n)
{
  char *pTemp = NULL;   /* Temporary pointer */
  bool sorted = false;  /* Strings sorted indicator */
  while(!sorted)        /* Loop until there are no swaps */
  {
    sorted = true;      /* Initialize to indicate no swaps */
      for(int i = 0 ; i < n-1 ; i++ )
      if(strcmp(p[i], p[i + 1] ) > 0)
      {
```

www.TopSage.com

```
            sorted = false; /* indicate we are out of order */
            swap(&p[i], &p[i+1]); /* Swap the pointers */
        }
    }
}
```

在这个函数中，参数变量 p 定义成指针数组。调用函数时，p 被作为变元传入函数的 pS 的地址取代。这里没有指定 p 的大小。这不是必要的，因为这是一维数组。该地址传入函数，作为第一个变元，而第二个变元定义了要处理的元素数目。如果数组大于一维，就必须指定所有维的大小，但第一维除外。这是让编译器了解数组的概况所必需的。第一个参数是 const，因为这个函数不修改字符串，只是重新安排了它们的地址。

在 str_sort()函数中将第二个参数 n 声明为 int 类型，调用函数时，它的值为变元 count 的值。声明一个变量 sorted，其值为 true 或 false，表示字符串排序是否已完成。在函数体中声明的所有变量都是这个函数的本地变量。

字符串在 for 循环中使用 swap()函数排序，这个函数交换两个指针中的地址，其执行过程如图 8-7 所示。注意，只改变了指针。在这个图中，使用了之前见过的输入数据，这些输入数据刚好在一次迭代中完成了排序工作。然而，如果在纸上给次序与此不同的输入字符串排序，通常需要多次迭代才能完成排序工作。

注意，str_sort()函数定义中没有返回语句。在函数执行过程中到达函数体的尾部与执行一个没有返回表达式的返回语句，其结果是一样的。

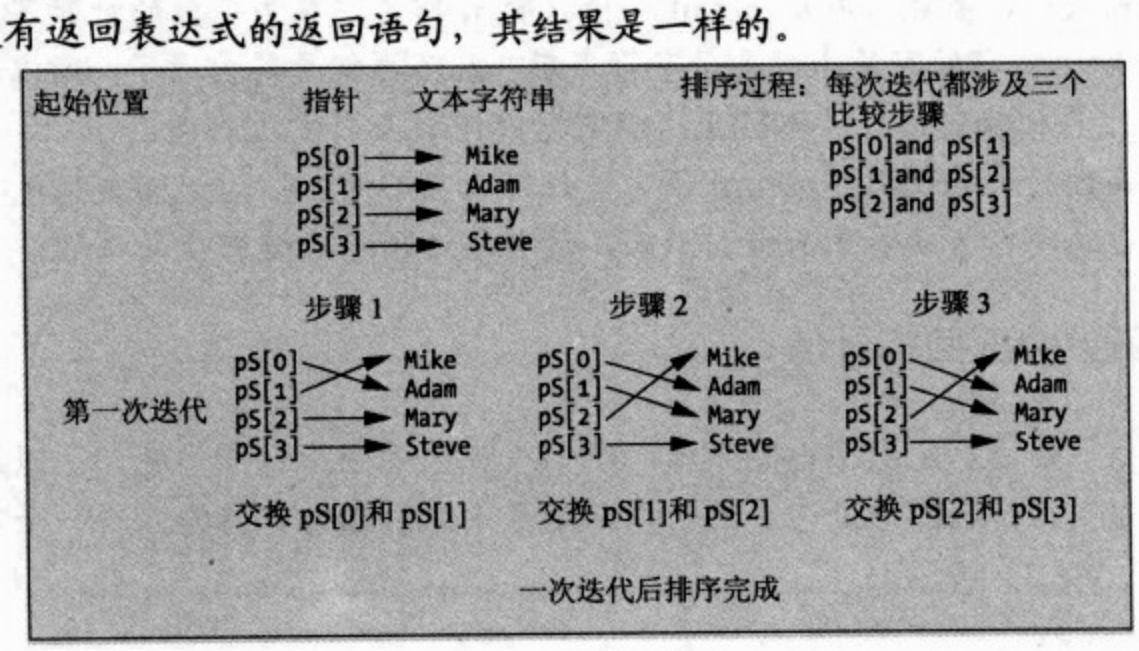


图 8-7 字符串排序

swap()函数

由 sort()函数调用的 swap()函数是一个比较短的实用函数，它交换两个指针：

```
void swap( void **p1, void **p2)
{
  void *pt = *p1;
  *p1 = *p2;
  *p2 = pt;
}
```

www.TopSage.com

指针是按值传送的，与其他类型的变元一样，所以要改变指针，必须把指针的指针作为函数的变元传送。这里的参数是类型 void**，表示执行 void*的指针。type*形式的指针都会转换为 void*类型，所以这个函数可以交换两个任意给定类型的指针。交换过程很简单。先把存储在 p1 位置的地址放在 pt 中，再把 p2 存储的地址传送给 p1，最后把 p2 设置为 p1 的原地址，即 pt 中的地址。

str_out()函数

最后一个被 main()函数调用的函数是 str_out()，它显示排序后的字符串。

```
void str_out(char *p[] , int n)
{
  printf("\nYour input sorted in order is:\n\n");
  for(int i = 0 ; i<n ; i++)
  {
    printf("%s\n", p[i]); /* Display a string */
    free(p[i]);           /* Free memory for the string */
    p[i] = NULL;
  }
  return;
}
```

在这个函数中，参数 n 接收 count 的值，将 n 作为字符串个数的计数器来显示字符串。for 循环输出所有的字符串。显示字符串后，就不再需要其内存了，所以调用库函数 free()释放它。再把指针设定为 NULL，以避免因错误而引用该内存。

在这个函数的最后使用了 return 语句，但可以省略它。因为返回类型是 void，执行到函数体最后的闭括号就等于 return(然而，这并不适用于有返回值的函数) 。

8.5.2 从函数中返回指针值

前面介绍了如何从函数中返回数值，学习了如何使用指针作为变元，以及如何将指针存储到作为变元传送的地址中。还可以从函数中返回指针。首先，看一个非常简单的例子。

试试看：从函数中返回数值

这里使用加薪作为这个例子的基础，因为它是一个大众化的主题。

```
/* Program 8.7 A function to increase your salary */
#include <stdio.h>

long *IncomePlus(long* pPay);  /* Prototype for increase function */

int main(void)
{
  long your_pay = 30000L;      /* Starting salary */
  long *pold_pay = &your_pay;  /* Pointer to pay value */
```

www.TopSage.com

```
  long *pnew_pay = NULL; /* Pointer to hold return value */
  pnew_pay = IncomePlus( pold_pay );
  printf("\nOld pay = $%ld", *pold_pay);
  printf(" New pay = $%ld\n", *pnew_pay);
  return 0;
}

/* Definition of function to increment pay */
long *IncomePlus(long *pPay)
{
  *pPay += 10000L; /* Increment the value for pay */
  return pPay;     /* Return the address */
}
```

执行这个程序，输出如下：

```
Old pay = $40000 New pay = $40000
```

代码的说明

在 main()函数中，为变量 your_pay 设置一个初始值，定义两个用于 IncomePlus()函数的指针，IncomePlus()函数用来增加 your_pay。一个指针初始化为 your_pay 的地址，另一个初始化为 NULL，因为它接收 IncomePlus()函数返回的地址。

输出看起来不错，但不正确。如果不知道原来的薪水是$30 000，这个输出看起来好像薪水一点都没有增加。因为函数 IncomePlus()通过指针 pold_pay 修改了 your_pay 的值，原来的值已经改变了。很明显，两个指针 pold_pay 和 pnew_pay 引用相同的位置：your_pay。这是函数 IncomePlus()的下述语句的结果：

```
return pPay;
```

这会返回函数调用时接收到的指针值，即 pold_pay 内的地址。结果是原来的薪水增长了——这就是指针的作用。

试试看：使用本地存储器

如果在函数 IncomePlus()中使用本地存储器存储返回值，会有什么结果？对这个例子做如下修改：

```
/* Program 8.8 A function to increase your salary that doesn't */
#include <stdio.h>

long *IncomePlus(long* pPay); /* Prototype for increase function */

int main(void)
{
  /* Code as before··· */
  return 0;
}
```

www.TopSage.com

```
/* Definition of function to increment pay */
long *IncomePlus(long *pPay)
{
  long pay = 0;          /* Local variable for the result */

  pay = *pPay + 10000;  /* Increment the value for pay */
  return &pay;          /* Return the address */
}
```

代码的说明

编译这个例子，可能会得到一条警告信息。但运行程序，得到的结果如下(由于机器不同，得到的结果可能不同，但该结果可能是正确的):

```
Old pay = $30000    New pay = $27467656
```

pay 的值$27 467 656 让人吃惊。在抱怨此类错误前可能会犹豫。如前所述，在不同的机器上可能会得到不一样的结果，这次可能是正确的结果。编译这个版本的程序，应该会得到一个警告。例如“令人怀疑的指针转换”。这是因为这个程序返回了变量 pay 的地址，它超出了函数 IncomePlus()的作用域，使 pay 的新值非常大——这个值是一个垃圾值，是其他程序遗留下来的。这是很容易犯的错误，如果编译器没有提出警告，就很难找出这个错误。

将 main()函数中的两个 printf()语句合并成一个语句:

```
printf("\nOld pay = $%ld New pay = $%ld\n", *pold_pay, *pnew_pay);
```

现在的输出如下:

```
Old pay = $30000   New pay = $40000
```

这看起来是正确的，但事实上程序中有一个严重的错误。虽然变量 pay 超出了作用域，因此不再存在，但它所占的内存尚未被重新使用。在这个例子中，显然某些东西使用了 pay 变量使用过的这个内存，生成了巨大的输出值。使用如下定律可以避免这类问题。

定律：绝不返回函数中本地变量的地址。

如何实现 IncomePlus()函数？如果要求函数修改传递给它的地址，第一个实现方式就很好，但如果不想改变地址，就应只返回 pay 的新值，而不是指针。调用程序必须存储这个返回值，而不是地址。

如果要将 pay 的新值存储到另一个位置中，函数 IncomePlus()就可以用 malloc()函数为它分配空间，并返回这个内存的地址。然而，应该注意调用函数必须释放该内存。最好给函数传送两个变元，一个变元是初始 pay 的地址，另一个是存储新 pay 的地址。这样，调用函数就可以支配内存了。

将执行期间分配内存和释放内存分开，有时会造成内存泄漏。在循环中重复调用的函数动态分配内存后，却没有释放它，就会出现内存泄漏。结果，越来越多可用的内存

www.TopSage.com

被占据了，当没有内存可用时，程序就会崩溃。应尽可能使分配内存的函数在使用完内存后就释放它。如果不能由函数释放内存，就要编写代码，释放动态分配的内存。

修改程序 8.6，就返回指针。这个例子改写了函数 str_in()，如下所示：

```
char *str_in(void)
{
  char buffer[BUFFER_LEN]; /* Space to store input string */
  char *pString = NULL;    /* Pointer to string */

  if(gets(buffer) == NULL) /* NULL returned from gets()? */
  {
    printf("\nError reading string.\n");
    exit(1);               /* Error on input so exit */
  }

  if(buffer[0] == '\0')    /* Empty string read? */
    return NULL;

  pString = (char*)malloc(strlen(buffer) + 1);

  if(pString == NULL)      /* Check memory allocation */
  {
    printf("\nOut of memory.");
    exit(1);               /* No memory allocated so exit */
  }
  return strcpy(pString, buffer); /* Return pString */
}
```

当然，必须修改函数原型：

```
char *str_in(void);
```

现在这个函数没有参数，因为上述语句将参数列表声明成 void，返回值是一个指向字符串的指针，而不是指向整数的指针。

还必须修改函数 main()中调用这个函数的 for 循环：

```
for(count=0; count < NUM_P ; count++)  /* Max of NUM_P strings */
  if((pS[count] = str_in())== NULL)    /* Stop input on NULL return */
    break;
```

现在比较函数 sir_in()返回的指针(存储在 pS[count]中)和 NULL，这表示输入了一个空字符串，或字符串因读取错误而未读入。这个例子的运作还和以前一样，但内部输入机制稍有不同。现在这个函数返回分配给字符串副本的内存地址。不能用 buffer 的地址来代替，因为 buffer 是函数的本地变量，一旦从这个函数返回后，它就超出了作用域。

选择函数 str_in()的哪个版本，就某种程度而言是个人的喜好，但平心而论，后一个版本比较好，因为它使用比较简单的参数定义，较容易理解。然而注意，内存在分配后一定要在程序中的某处释放。必须这么做时，最好在编写内存分配函数后，再编写内存

www.TopSage.com

释放函数，以减少内存泄漏的风险。

8.5.3 在函数中递增指针

使用数组名称作为函数的变元时，会把数组起始地址的副本传给函数。因此，可以把接收的数值看成指针，然后递增或递减它。例如，重写程序 8.6 中的 str_out()函数，如下所示：

```
void str_out(char *p[] , int n)
{
  printf("\nYour input sorted in order is:\n\n");
  for(int i = 0 ; i<n ; i++)
  {
    printf("%s\n", *p); /* Display a string */
    free(*p);           /* Free memory for the string */
    *p++ = NULL;
  }
  return;
}
```

这段代码在 printf()和 free()函数调用中用指针取代了数组。这不适用于在函数体内声明的数组，但这段代码有原始数组地址的副本，所以可以这么做。可以把这个参数当作一般的指针。这里的地址是 main()函数中原地址的副本，所以不妨碍原始的数组地址 pS。

函数的这个版本和原版本没有区别，前一个版本使用数组，较容易理解。然而用指针表示的操作通常执行起来比用数组快。

8.6 小结

本章尚未完成函数的讨论，所以第 9 章的最后将通过另一个例子，介绍使用函数的更多内容。下面总结创建和使用函数时的重点：

- C 程序由一个或多个函数组成，其中一个是 main()函数。该函数永远是执行的起点，操作系统通过一个用户命令调用它。
- 函数是程序中独立的一块代码。函数的名称采用标识符名称的形式，由一系列字母和数字组成，第一个字符必须是字母(下划线算是字母)。
- 函数定义由函数头和函数体组成。函数头定义了函数的名称、函数返回值的类型及函数中所有参数的类型和名称。函数体含有函数的可执行语句，定义了这个函数的功能。
- 在函数中声明的所有变量都是函数的本地变量。

www.TopSage.com

- 函数原型是一个以分号终止的声明语句，用以定义函数的名称、返回类型和函数的参数类型。在可执行代码中，如果函数调用出现在函数定义之前，就需要函数原型给编译器提供相关信息。
- 在源文件中使用函数之前，应该先定义这个函数，或用函数原型声明这个函数。
- 将指针参数指定为 const，就告诉编译器，这个函数不改变该参数指向的数据。
- 函数变元的类型必须符合函数头中对应的参数。如果指定参数的类型是 int，但传送了 double 类型的值，该值就会被截断，删除小数部分。
- 有返回值的函数可以用在表达式中，就如同它是一个与返回值类型相同的数值一样。
- 在调用函数中，是将变元值的副本传给函数，而不是传送原始值。这种给函数传送数据的方式称为按值传递机制。
- 如果函数要修改在调用函数中定义的变量，就需要将这个变量的地址作为变元传送。

这些涵盖了创建定制函数的重点。第 9 章将介绍使用函数的其他技巧，在真实的例子中使用函数。

8.7 习题

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容。还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 8.1　定义一个函数，给函数传送任意多个浮点数，计算出这些数的平均值。从键盘输入任意个值，并输出平均值，以说明这个函数的执行过程。

习题 8.2　定义一个函数，返回其整数变元的字符串表示。例如，如果这个变元是 25，函数就返回"25"。如果变元是 - 98，函数就返回" - 98"。用适当的 main()版本说明函数的执行过程。

习题 8.3　扩展为上一题定义的函数，使函数接受第二个变元，以指定结果的字段宽度，使返回的字符串表示右对齐。例如，如果第一个变元的值是-98，字段宽度变元是 5，返回的字符串就应是" - 98"。用适当的 main()版本说明函数的执行过程。

习题 8.4　定义一个函数，其参数是一个字符串，返回该字符串中的单词数(单词以空格或标点符号来分隔。假设字符串不含单双引号，也就是说没有像 isn't 这样的单词)。定义第二个函数，它的第一个参数是一个字符串，第二个参数是一个数组，该函数将第一个字符串变元分割成单词，把这些单词存储在第二个数组变元中，最后返回存储在数组中的单词。定义第三个函数，其参数是一个字符串，返回该字符串中的字母数。使用这些函数实现一个程序，从键盘读入含有文本的字符串，输出文本中的所有单词，输出顺序是按照单词中的字母数，由短到长。

www.TopSage.com

www.TopSage.com

第 9 章

函 数 再 探

学习了第 8 章后，读者就应具备创建和使用函数的基础知识了。本章将以此为基础，介绍函数的使用和操作，尤其是如何通过指针访问函数，也会使用一些更灵活的方法在函数之间通信。

本章的主要内容：

- 函数指针的概念及其用法
- 如何在函数内使用静态变量
- 如何在函数之间共享变量
- 函数如何调用自己，但不陷入无限循环
- 编写一个五子棋游戏(也称为 Reversi)

9.1 函数指针

指针对于操作数据和含有数据的变量是一个非常有用的工具。只要一把火钳就可处理所有火热的东西，同样，使用指针也可以操作函数，函数的内存地址存储了函数开始执行的位置(起始地址)，存储在函数指针中的内容就是这个地址。

不过，仅有地址还不够。如果函数通过指针来调用，还必须提供变元的类型和个数，以及返回值的类型。编译器不能仅通过函数的地址来推断这些信息。这说明，声明函数指针比声明数据类型指针复杂一些。指针包含了地址，而且必须定义一个类型，同样，函数指针也包含了地址，也必须定义一个原型。

9.1.1 声明函数指针

函数指针的声明看起来有点奇怪，容易混淆，所以下面从一个简单的例子开始：

```
int  (*pfunction)  (int);
```

这是一个函数指针的声明，它不指向任何东西——该语句只定义了指针变量。这个指针的名称是 pfunction，指向一个参数是 int 类型，返回值是 int 类型的函数。而且，这个指针只能指向有这些特质的函数。如果函数接受 float 变元，返回 float 值，就需要声明另一个有这些特质的指针。图 9-1 说明了声明的各个成分。

TopSage.com

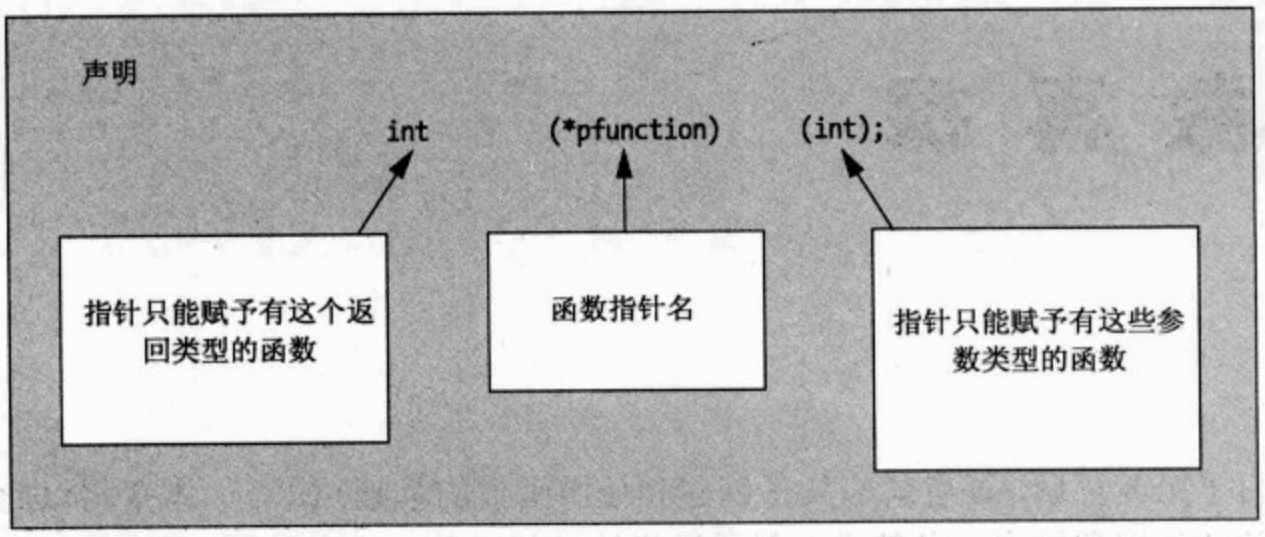


图 9-1 声明函数指针

在函数指针的声明中有许多括号。在这个例子中，声明*pfunction 的部分必须放在括号中。

注意：

如果省略了括号，就变成 pfunction()函数的声明了，这个函数返回一个指向 int 的值，这可不是我们希望的结果。

在函数指针的声明中，必须在括号中添加*和指针名称。第二对括号包含参数列表，这与标准函数声明相同。函数指针只能指向特定的函数，该函数有特定的返回类型、特定的参数个数和特定类型的参数，唯一可变的是函数名称。

9.1.2 通过函数指针调用函数

假定定义如下函数原型：

```
int sum(int a, int b); /* Calculates a+b */
```

这个函数有两个 int 类型的参数，返回值的类型是 int，所以可以把它的地址存储在如下函数指针中：

```
int (*pfun)(int, int) = sum;
```

这条语句声明了一个函数指针 pfun，它存储函数的地址，该函数有两个 int 类型的参数，返回值的类型是 int。该语句还用 sum()函数的地址初始化 pfun。要提供初始值，只需使用有所需原型的函数名。

现在可以通过函数指针调用 sum()函数：

```
int result = pfun(45, 55);
```

这条语句通过 pfun 指针调用变元值为 45 和 55 的 sum()函数，将 sum()的返回值用作 result 变量的初始值，因此 result 是 100。注意，像使用函数名那样使用函数指针名调用该指针指向的函数，不需要取消引用运算符。

www.TopSage.com

假定定义了有如下原型的另一个函数：

```
int product(int a, int b); /* Calculates a*b */
```

就可以使用下面的语句在 pfun 中存储 product()的地址：

```
pfun = product;
```

pfun 包含 product()的地址，所以可以通过指针调用 product()：

```
result = pfun(5, 12);
```

执行了这条语句后，result 就包含 60。
下面以一个简单的例子来说明函数指针是如何运作的。

试试看：使用函数指针

这个例子将定义 3 个函数，它们有相同的参数和返回类型，再使用函数指针轮流调用它们。

```
/* Program 9.1 Pointing to functions */
#include <stdio.h>

/* Function prototypes */
int sum(int, int);
int product(int, int);
int difference(int, int);

int main(void)
{
  int a = 10;                /* Initial value for a */
  int b = 5;                 /* Initial value for b */
  int result = 0;            /* Storage for results */
  int (*pfun)(int, int);     /* Function pointer declaration */

  pfun = sum;                /* Points to function sum() */
  result = pfun(a, b);       /* Call sum() through pointer */
  printf("\npfun = sum result = %d", result);

  pfun = product;            /* Points to function product() */
  result = pfun(a, b);       /* Call product() through pointer */
  printf("\npfun = product result = %d", result);

  pfun = difference;         /* Points to function difference() */
  result = pfun(a, b);       /* Call difference() through pointer */
  printf("\npfun = difference result = %d\n", result);
  return 0;
}
```

www.TopSage.com

```
int sum(int x, int y)
{
  return x + y;
}

int product(int x, int y)
{
  return x * y;
}

int difference(int x, int y)
{
  return x - y;
}
```

这个程序的输出结果如下:

```
pfun =  sum         result = 15
pfun =  product     result = 50
pfun =  difference  result =  5
```

代码的说明

这个例子声明并定义了3个不同的函数,以返回两个整数变元的和、积和差。在main()函数中,使用下面的语句声明一个函数指针:

```
int (*pfun)(int, int); /* Function pointer declaration */
```

这个指针可以赋予任何带两个 int 参数且返回 int 值的函数。注意给指针赋值的方式:

```
pfun = sum;                /* Points to function sum() */
```

这只是一般的赋值语句,等式右边只有函数名称,不需要添加参数列表或其他数据。如果添加了其他数据,就是错误的,因为这就变成函数调用了,而不是一个地址,此时编译器会报错。在这里,函数的用法非常类似于数组。如果需要的是数组的地址,只要使用数组名即可。同样,如果需要的是函数的地址,也是只使用函数名即可。

在 main()中,依次将每个函数的地址赋予函数指针 pfun,然后使用 pfun 指针调用每个函数,并显示结果。下面的语句说明了如何使用指针调用函数:

```
result = pfun(a, b); /* Call sum() through pointer */
```

在此将指针名当成函数名来使用,后面跟随放在括号中的变元列表。而将函数指针变量名当做原来的函数名,则变元列表必须对应函数头的参数列表,如图 9-2 所示。

www.TopSage.com

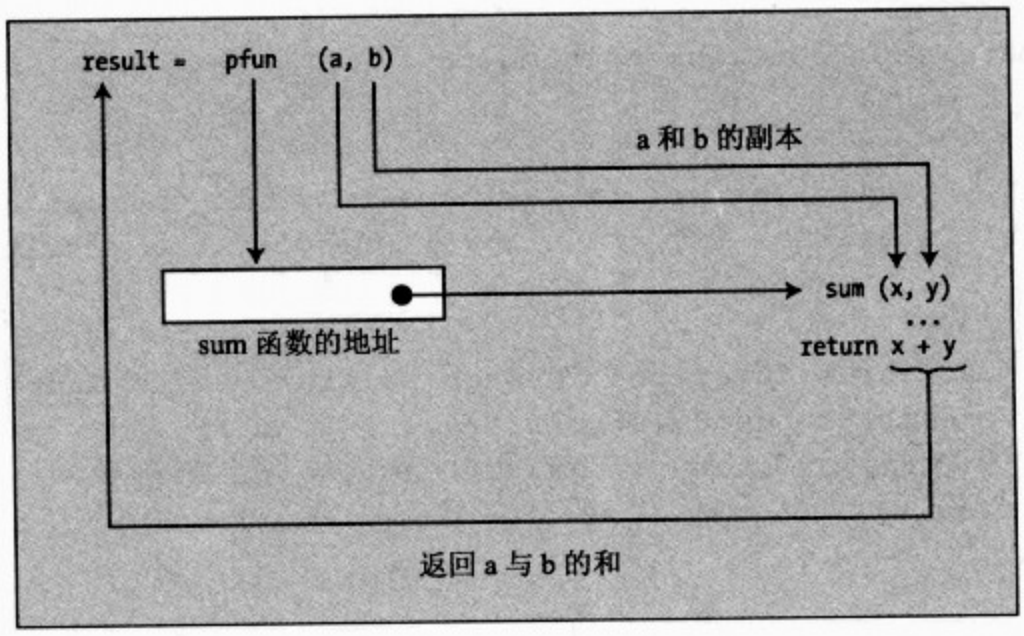


图 9-2 通过指针调用函数

9.1.3 函数指针数组

函数指针和一般的变量是一样的，所以可创建函数指针的数组。要声明函数指针数组，只需将数组的大小放在函数指针数组名之后。例如：

```
int  (*pfunctions[10])  (int);
```

这条语句声明了一个包含 10 个元素的 pfunctions 数组。这个数组里的每个元素都能存储一个函数的地址，该函数有两个 int 类型的参数，返回类型是 int。下面看一个实例。

试试看：函数指针数组

对上一个例子做一些变化，以说明如何使用函数指针数组：

```
/* Program 9.2 Arrays of Pointers to functions */
#include <stdio.h>

/* Function prototypes */
int sum(int, int);
int product(int, int);
int difference(int, int);

int main(void)
{
  int a = 10;                          /* Initial value for a */
  int b = 5;                           /* Initial value for b */
  int result = 0;                      /* Storage for results */
  int (*pfun[3])(int, int); /* Function pointer array declaration */

  /* Initialize pointers */
  pfun[0] = sum;
  pfun[1] = product;
  pfun[2] = difference;
```

www.TopSage.com

```
  /* Execute each function pointed to */
  for(int i = 0 ; i < 3 ; i++)
  {
    result = pfun[i](a, b); /* Call the function through a pointer */
    printf("\nresult = %d", result); /* Display the result */
  }

  /* Call all three functions through pointers in an expression */
  result = pfun[1](pfun[0](a, b), pfun[2](a, b));
  printf("\n\nThe product of the sum and the difference = %d\n",
         result);
  return 0;
}

/* Definitions of sum(), product() and difference() as before··· */
```

程序的输出如下:

```
result =  15
result = 50
result =  5

The  product of  the  sum  and  the  difference = 75
```

代码的说明

这个程序和前一个例子的主要差异是指针数组，它的声明如下:

```
int (*pfun[3])(int, int); /* Function pointer array declaration */
```

这类似于前面对一个指针变量的声明，只是指针名称后面多了放在方括号中的数组大小。如果需要的是二维数组，就应有两对方括号，如同声明一般的数组类型一样。参数列表仍然要放在括号内，这与单个指针的声明相同。另外，和一般的数组一样，函数指针数组的所有元素都是相同的类型，都只能接受指定的变元列表。因此在此例中，这些指针都只能指向带两个 int 参数，返回 int 值的函数。

给数组中的指针赋值时，语句和一般的数组元素相同:

```
pfun[0]  =  sum;
```

除了等号右侧的函数名称之外，这就是一个正常的数据数组，其用法也完全相同。可以在声明中初始化指针数组的所有元素:

```
int  (*pfun[3])(int, int)  =  {  sum, product, difference  };
```

这条语句初始化了 3 个元素，所以不再需要执行初始化的赋值语句。事实上，也可以去掉数组的大小，由初始化列表确定数组的大小:

```
int  (*pfun[])(int, int)  =  {  sum, product, difference  };
```

www.TopSage.com

大括号内的初始值个数确定了数组中的元素数目。因此，函数指针数组的初始化列表与其他数组的初始化列表的作用相同。

调用数组元素所指的函数时，可以用下列方法表示：

```
result = pfun[i](a, b); /* Call the function through a pointer */
```

这与前一个例子类似，只是指针名的后面多了放在方括号中的索引值。用循环变量 i 调用这个数组，这与前面使用一般数据数组的方式相同。

在输出中，前三行在 for 循环内生成，在该循环中，函数 sum()、product()和 difference()依次通过指针数组中的对应元素调用。最后一行输出是在下面的语句中使用 result 值创建的：

```
result = pfun[1](pfun[0](a, b), pfun[2](a, b));
```

这行语句说明，可以通过指针将函数调用合并到表达式中，这和使用一般函数调用的方式相同。这里通过指针调用两个函数，将它们的结果用作通过指针调用的第三个函数的变元。pfun 数组元素依次对应函数 sum()、product()和 difference()，因此这行语句相当于下面的语句：

```
result = product(sum(a, b), difference(a, b));
```

这行语句处理的事件顺序如下：

(1) 执行 sum(a，b)和 difference(a，b)，然后存储返回值。

(2) 使用步骤 1 的返回值作为变元，执行函数 product()，然后存储返回值。

(3) 将步骤 2 所得的值存储到变量 result 中。

9.1.4 作为变元的函数指针

也可以将函数指针作为变元来传递，这样就可以根据指针所指向的函数，而调用不同的函数了。

试试看：作为变元的函数指针

修改上一个例子，将函数指针作为变元传入函数：

```
/* Program 9.3 Passing a Pointer to a function */
#include <stdio.h>

/* Function prototypes */
int sum(int,int);
int product(int,int);
int difference(int,int);
int any_function(int(*pfun)(int, int), int x, int y);

int main(void)
{
  int a = 10;              /* Initial value for a */
```

www.TopSage.com

```
  int b = 5;                    /* Initial value for b */
  int result = 0;               /* Storage for results */
  int (*pf)(int, int) = sum; /* Pointer to function */

  /* Passing a pointer to a function */
  result = any_function(pf, a, b);

  printf("\nresult = %d", result );

  /* Passing the address of a function */
  result = any_function(product,a, b);

  printf("\nresult = %d", result );

  printf("\nresult = %d\n", any_function(difference, a, b));
  return 0;
}

/* Definition of a function to call a function */
int any_function(int(*pfun)(int, int), int x, int y)
{
  return pfun(x, y);
}

/* Definition of the function sum */
int sum(int x, int y)
{
  return x + y;
}

/* Definition of the function product */
int product(int x, int y)
{
  return x * y;
}

/* Definition of the function difference */
int difference(int x, int y)
{
  return x - y;
}
```

程序的输出结果如下:

```
result = 15
result = 50
result = 5
```

www.TopSage.com

代码的说明

将函数指针作为变元的函数是 any_function()，它的函数原型如下：

```
int  any_function(int(*pfun)(int, int), int  x, ant  y);
```

any_function()函数有 3 个参数，第一个参数是一个函数指针，它指向的函数接受两个整数参数并且返回整数。any_function()函数的后两个参数都是整数，在调用第一个参数指定的函数时使用。any_function()函数返回一个整数，而这个整数是调用第一个变元指定的函数得到的。

在 any_function()函数的定义里，指针变元指定的函数在 return 语句中调用：

```
int  any_function(int(*pfun)(int, int), int  x, int  y)
{
  return  pfun(x, y);
}
```

这个定义使用了指针名称 pfun，后跟的另外两个参数用作被调用函数的变元。pfun 的值和另外两个参数 x 和 y 的值都来自于 main()。

注意在 main()中声明的函数指针 pf 是如何初始化的：

```
int (*pf)(int, int) = sum; /* Pointer to function */
```

将函数 sum()的名称作为初始化值放在等号的后面，如前所述，只要将函数名作为初始化值，就可以将函数指针初始化为指定函数的地址。

any_function()的第一个调用给 any_function()传递了指针 pf、变量 a 及 b 的值：

```
result =  any_function(pf, a, b);
```

指针 pf 和平常一样用作变元，any_function()返回的值存储到变量 result 中。pf 的初始值是 sum()函数的地址，所以在 any_function()内调用了 sum()函数，因此返回值是 a 和 b 的和。

any_function()的下一个调用是：

```
result = any_function(product, a, b);
```

这里明确指定函数名 product 作为第一个变元，所以在 any_function()中调用函数 product，并将 a 和 b 的值作为变元。在此例中，编译器会创建一个指向 product 函数的内部指针，并传给函数 any_function()。

any_function()的最后一个调用放在 printf()函数调用的变元中：

```
printf("\nresult  =  %d\n", any_function(difference, a, b));
```

在这行语句里，也明确指定函数名 difference 作为 any_function()的一个变元。编译器从 any_function()的原型中了解到，该函数的第一个变元应该是一个函数指针。这里将函数名 difference 指定为变元，所以编译器会创建一个指向这个函数的指针，并将它传给 any_function()。最后，将 any_function()返回的值作为变元传递给函数 printf()。执行完这

www.TopSage.com

行语句，会显示 a 和 b 的差。

注意，不要混淆了把函数的地址作为变元传送给函数和传递函数的返回值的概念，例如下面的表达式是把函数的地址作为变元传送给函数：

```
any_function(product, a, b)
```

下面的语句是传递函数的返回值：

```
printf("\n%d", product(a, b));
```

前一条语句是将函数 product()的地址作为变元传送，该函数是否会调用以及何时调用取决于 any_function()函数体。后一条语句是调用 printf()之前先调用函数 product()，然后将 product()返回的结果作为变元传递给 printf()。

9.2 函数中的变量

将程序分解成函数，不仅简化了开发程序的过程，还增强了程序语言解决问题的能力。设计优良的函数常常可以重用，使新应用程序的开发变得更快、更简单。标准库就证明了可重用函数的威力。函数中变量的特性以及 C 语言在声明变量时提供的一些额外功能进一步增强了程序语言的力量。下面介绍函数中的变量。

9.2.1 静态变量：函数内部的追踪

前面使用的所有变量在执行到定义它的块尾时就超出了作用域，它们在堆栈上分配的内存会被释放，以供另一个函数使用。这些变量称为自动变量，因为它们是在声明时自动创建的，在程序退出声明它的块后自动删除。这是一种非常高效的过程，因为只要正在执行的语句在声明变量的函数内，函数中包含数据的内存就会一直保存该数据。

然而在某些情况下，要求在退出一个函数调用后，该调用中的数据可以在程序的其他函数中使用。例如保留函数中的某种计数器，如函数的调用次数或输出行数。这使用自动变量是做不到的。

不过，C 语言提供了静态变量，可以达到这个目的。例如用下面的语句声明一个静态变量 count：

```
static int count = 0;
```

上述语句中的 static 是 C 的一个关键字，该语句声明的变量和自动变量有两点不同。第一，虽然它在函数的作用域内定义，但当执行退出该函数后，这个静态变量不会删除。第二，自动变量每次进入作用域时，都会初始化一次，但是声明为 static 的变量只在程序开始时初始化一次。静态变量只能在包含其声明的函数中可见，但它是一个全局变量，因此可以用全局变量的方式使用它。

www.TopSage.com

注意：
可以在函数内创建任何类型的静态变量。

试试看：使用静态变量
下面这个简单的例子演示了静态变量的用法：

```
/* Program 9.4 Static versus automatic variables */
#include <stdio.h>

/* Function prototypes */
void test1(void);
void test2(void);

int main(void)
{
  for(int i = 0; i < 5; i++ )
  {
    test1();
    test2();
  }
  return 0;
}

/* Function test1 with an automatic variable */
void test1(void)
{
  int count = 0;
  printf("\ntest1 count = %d ", ++count );
}

/* Function test2 with a static variable */
void test2(void)
{
  static int count = 0;
  printf("\ntest2 count = %d ", ++count );
}
```

程序的输出结果如下：

```
test1   count = 1
test2   count = 1
test1   count = 1
test2   count = 2
test1   count = 1
test2   count = 3
test1   count = 1
test2   count = 4
test1   count = 1
test2   count = 5
```

代码的说明

可以看出，这两个 count 变量是完全不同的，其值的变化清楚地说明了它们是相互独立的。静态变量 count 在函数 test2()内声明，如下：

```
static int count = 0;
```

可以给这个变量指定初始值，但这里将它初始化为 0，因为将它声明为静态变量。

注意：

所有的静态变量都会初始化为 0，除非将它们初始化为其他值。

静态变量 count 用于计算函数的调用次数。当程序开始执行时初始化它，程序退出函数后，它的当前值仍然保留。该变量没有在函数的后续调用中重新初始化。由于该变量声明为 static，所以编译器只将它初始化一次。初始化操作是在程序开始之前进行的，所以总是可以确保静态变量在使用时初始化。

自动变量 count 在函数 testl()内的声明如下：

```
int count = 0;
```

这是自动变量，所以它不会在程序开始执行时初始化。如果不给它指定初始值，它将会含有一个垃圾值。这个变量会在每次执行函数时初始化为 0，在每次退出 test1()后删除，因此它永远不会大于 1。

只要程序开始执行，静态变量就一直存在，但是它只能在声明它的范围内可见，不能在该作用域的外部引用。

9.2.2 在函数之间共享变量

也可以在所有的函数之间共享变量。常量在程序文件的开头声明，所以常量位于组成程序的所有函数的外部)，同样，也可以采用这种方式声明变量，这种变量称为全局变量(global variables)，因为它们可以在任意位置访问。它的声明方式和一般变量相同，但声明它的位置非常重要，这个位置确定了变量是否为全局变量。

试试看：使用全局变量

修改前一个例子，在函数之间共享 count 变量。

```
/* Program 9.5 Global variables */
#include <stdio.h>

int count = 0;              /* Declare a global variable */

/* Function prototypes */
void test1(void);
void test2(void);

int main(void)
```

www.TopSage.com

```
{
  int count = 0;              /* This hides the global count */

  for( ; count < 5; count++)
  {
    test1();
    test2();
  }
  return 0;
}

/* Function test1 using the global variable */
void test1(void)
{
printf("\ntest1 count = %d ", ++count);
}

/* Function test2 using a static variable */
void test2(void)
{
  static int count;           /* This hides the global count */
  printf("\ntest2 count = %d ", ++count);
}
```

程序的输出结果如下：

```
test1 count = 1
test2 count = 1
test1 count = 2
test2 count = 2
test1 count = 3
test2 count = 3
test1 count = 4
test2 count = 4
test1 count = 5
test2 count = 5
```

代码的说明

在这个例子中，有3个count变量。第一个是全局变量count，它在程序的开头声明：

```
#include <stdio.h>

int count = 0;
```

这不是静态变量(也可以把它声明成静态变量)，而是全局变量，所以如果没有初始化它，它就默认为0。从声明该全局变量到程序结束的任何函数中访问它。

第二个count是自动变量，在main()函数中声明：

```
int count = 0; /* This hides the global count */
```

www.TopSage.com

它和全局变量同名，所以在 main()函数中不能访问全局变量 count。在 main()函数中使用的 count 都是在 main()函数体中声明的自动变量。本地变量隐藏了全局变量。

第三个 count 是静态变量，在函数 test2()里声明:

```
static int count; /* This hides the global count */
```

这是一个静态变量，所以默认初始化为 0。这个变量也隐藏了同名的全局变量，所以在 test2()内只能访问静态变量 count。

函数 testl()使用的是全局变量 count。函数 main()和 test2()使用的是 count 的本地版本，因为本地声明隐藏了同名的全局变量。

显然，main()内的 count 变量从 0 递增到 4，因为调用了 5 次 test1()和 test2()。在 test1()及 test2()内，count 变量是不同的，否则程序就不会输出 1~5 的值。

删除 test2()内对静态变量 count 的声明，可以进一步证实这个事实。这会使 test1()和 test2()共享全局变量 count，显示出的值会变成 1~10。如果将 test2()内的 count 变量改成已初始化的自动变量:

```
int  count  =  0;
```

test1()会输出 1~5，而 test2()的输出始终是 1，这是因为该变量现在是自动变量，每次执行函数时，都会重新初始化。

全局变量可以取代函数变元及返回值，完全取代自动变量似乎很吸引人，但应少使用全局变量，全局变量可以简化并缩短某些程序，但过度使用会使程序容易出错。主要原因是很容易修改全局变量，却忘记它对整个程序带来的后果。程序越大，避免错误引用全局变量的难度就越大。而本地变量可以有效地隔离各个函数，避免这些函数互相干扰。删除程序 9.5 中 main()的本地变量 count，看看输出结果会如何。

注意:

在 C 语言中，最好不要给本地变量和全局变量使用相同的名字。这不但没有好处，反而有坏处，如上面的例子。

9.3 调用自己的函数: 递归

函数调用自己称为递归，递归在程序设计中不常见，所以本节仅介绍概念，不过在某些情况下，这是一个效率很高的技巧，可以显著简化解决特定问题所需的代码。递归也有几个坏处，但这里也不涉及。

显然，函数调用自己时，一个问题是如何停止递归过程。下面的函数示例就陷入了一个无限循环:

```
void Looper(void)
{
  printf("\nLooper function called.");
  Looper(); /* Recursive call to Looper() */
```

www.TopSage.com

```
}
```

调用这个函数会输出无数行结果，因为在执行 printf()调用后，函数会调用它自己。代码中没有停止该过程的机制。

这就类似于一个无限循环，解决方法也很类似：一个调用自己的函数必须包含停止处理的方式，下面说明这个方式。

试试看：递归

递归的主要用途是解决复杂的问题，所以很难用简单的例子说明其工作原理。因此这里使用标准证明方式：计算整数阶乘。所谓整数阶乘，就是从 1 到该整数的所有整数之积。

```
/* Program 9.6 Calculating factorials using recursion */
#include <stdio.h>

unsigned long factorial(unsigned long);

int main(void)
{
  unsigned long number = 0L;
  printf("\nEnter an integer value: ");
  scanf(" %lu", &number);
  printf("\nThe factorial of %lu is %lu\n", number, factorial(number));
  return 0;
}

/* Our recursive factorial function */
unsigned long factorial(unsigned long n)
{
  if(n < 2L)
    return n;

  return n*factorial(n - 1);
}
```

程序的输出结果如下：

```
Enter an integer value: 4

The factorial of 4 is 24
```

代码的说明

一旦理出头绪，事情就会变得很简单。下面讨论一个具体的例子，假设输入 4，计算过程如图 9-3 所示。

www.TopSage.com

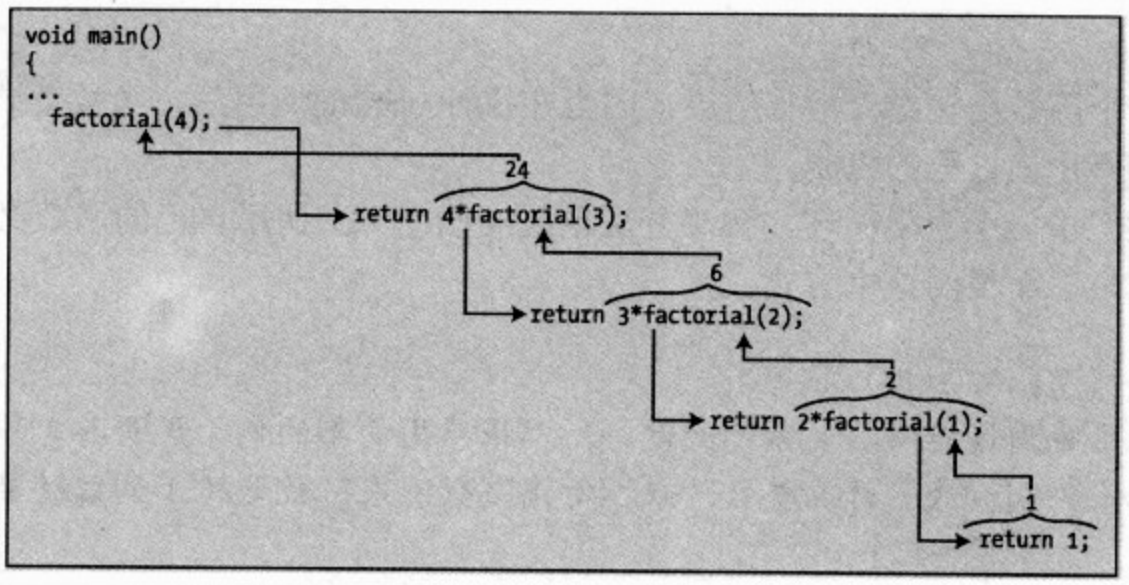


图 9-3 递归函数调用

在下面的语句中:

```
printf("\nThe factorial of %lu is %lu\n", number, factorial(number));
```

在 main()中调用函数 factorial()时，给它传递值 4 作为变元。

在 factorial()函数中，因为变元大于 1，所以执行下面的语句:

```
return  n*factorial(n  -  1);
```

这是函数内的第二个 return 语句，它在算术表达式中用数值 3 再次调用 factorial()函数，这个表达式也不能计算，return 语句不能执行，除非用变元 3 调用的 factorial()函数返回了其值。

这个调用过程会继续下去，如图 9-3 所示，直到 factorial()函数最后一次调用的变元是 1 为止。此时，执行第一个 return 语句:

```
return n;
```

将值 1 返回给前面的调用点。实际上，这个调用点在 factorial()函数的第二个 return 语句中，它现在可以计算 2*1，返回给前一个调用。

以这种方式执行整个过程，直到将需要的值返回给显示结果的 main()函数为止。对任意给定的值 n，函数 factorial()会调用 n 次。每次调用都创建变元的一个副本，并存储返回的地址。如果递归的层次很多，这是相当浪费内存的。而使用循环来完成，不但节省内存，而且速度较快。如果非用递归不可，一定要有停止处理的方法。换句话说，就是要有停止递归调用的机制。在上面的例子中，检查变元是否等于 1，就是停止递归调用 factorial()函数的机制。

注意，阶乘值会很快变得非常大，即使不太大的输入值，计算出来的阶乘值也会超过 unsigned long 的容量，导致错误的结果。

www.TopSage.com

9.4 变元个数可变的函数

在标准库中，某些函数的变元数是可变的，例如函数 printf()和 scanf()。有时需要这么做，所以标准库<stdarg.h>提供了编写这种函数的例程。

编写参数个数可变的函数时，第一个问题是如何指定它的原型。假设要创建一个函数，计算两个或多个 double 值的平均值。显然，计算少于两个数的平均值是没有意义的。它的原型可以这么编写：

```
double  average(double  v1,  double  v2,  …);
```

第二个参数类型后的 3 个点(省略号)表示，在前两个固定的变元后面，可以有数量可变的变元。至少要有一个固定的变元，其他内容和一般的函数原型一样，前两个变元是 double 类型，返回的结果也是 double。

变元个数可变的第二个问题是，在编写函数时，如何引用变元？我们不知道有多少个变元，所以不可能给它们指定名字。唯一的方法是通过指针间接地指定变元。<stdarg.h>头文件提供了通常实现为宏的例程，宏的外观和操作都类似于函数，所以将它们作为函数来讨论。要实现变元个数可变的函数时，必须同时使用 3 个宏：va_start()、va_arg()、va_end()。第一个宏的形式如下：

```
void  va_start(va_list  parg,  last_fixed_arg);
```

这个宏的名称来源于 variable argument start。这个函数接受两个变元： va_list 类型的指针 parg 和为函数指定的最后一个固定参数的名字。va_list 类型也在<stdarg.h>头文件中定义，用于存储支持可变参数列表的例程所需的信息。

以 average()函数为例，可以将该函数编写成：

```
double average(double v1, double v2,...)
{
  va_list parg;                  /* Pointer for variable argument list */
  /* More code to go here... */
  va_start( parg, v2);
  /* More code to go her. . . */
}
```

首先，声明一个 va_list 类型的变量 parg。然后，用 parg 作为第一个变元，指定最后一个固定参数 v2 作为第二个变元，调用 va_start()。调用 va_start()的结果是将变量 parg 设定为指向传送给函数的第一个可变变元。此时并不知道这个值的类型，标准库对此也无能为力，但必须确定每一个可变变元的类型，例如假设所有的可变变元都是同一种特定的类型，或从固定变元包含的信息推断每个变元的类型。

average()函数处理 double 类型的变元，所以确定可变变元的类型不成问题。现在必须知道如何访问每个可变变元的值，下面完成 average()函数：

```
/* Function to calculate the average of a variable number of arguments */
```

www.TopSage.com

```
double average( double v1, double v2,...)
{
  va_list parg;          /* Pointer for variable argument list */
  double sum = v1+v2;    /* Accumulate sum of the arguments */
  double value = 0;      /* Argument value */
  int count = 2;         /* Count of number of arguments */

  va_start(parg,v2); /* Initialize argument pointer */
  while((value = va_arg(parg, double)) != 0.0)
  {
    sum += value;
    count++;
  }
  va_end(parg);          /* End variable argument process */
  return sum/count;
}
```

在声明 parg 后，将变量 sum 声明为 double 类型，同时用前两个固定变元 v1 和 v2 的和来初始化 sum。所有变元值的和都会累加到 sum 中，所以下一个变量 value 声明成 double，用于存储可变变元的值。然后声明计数器 count，用来存储变元的数目，并将计数器 count 初始化为 2，因为至少有两个固定变元。在调用 va_start()初始化 parg 后，在下面的 while 循环内执行所有的动作：

```
while((value = va_arg(parg, double)) != 0.0)
```

循环条件调用了<stdarg.h>头文件中的另一个函数 va_arg()。va_arg()的第一个变元是通过调用 va_start()初始化的变量 parg，第二个变元是期望确定的变元类型的说明。va_arg()函数会返回 parg 指定的当前变元值，并将它存储到 value 中。同时会更新 parg 指针，使之根据调用中指定的类型，指向列表中的下一个变元。必须有某种方式来确定可变变元的类型，因为如果指定的类型不正确，就不能正确得到下一个变元。这个例子编写函数时，假设所有的变元都是 double 类型。另一个假设是除了最后一个变元外，其他变元都是非零值。这反映在循环继续条件中，即 value 不等于 0。在循环中，在 sum 中累计总和，并递增 count。

变元值等于 0 时，就结束循环，执行下一行语句：

```
va_end(parg); /* End variable argument process */
```

调用 va_end()函数，处理该过程的剩余工作。它将 parg 重置为指向 NULL。如果省掉这个调用，程序就不会正常工作。整理完成后，就可以用下面的语句返回需要的结果了：

```
return  sum/count;
```

试试看：使用可变的变元列表

编写完函数 average()后，最好用一个小程序确保它可以正常工作：

www.TopSage.com

```
/* Program 9.7 Calculating an average using variable argument lists */
#include <stdio.h>
#include <stdarg.h>

double average(double v1 , double v2,...); /* Function prototype */

int main(void)
{
  double Val1 = 10.5, Val2 = 2.5;
  int num1 = 6, num2 = 5;
  long num3 = 12, num4 = 20;

  printf("\n Average = %lf", average(Val1, 3.5, Val2, 4.5, 0.0));
  printf("\n Average = %lf", average(1.0, 2.0, 0.0));
  printf("\n Average = %lf\n", average( (double)num2, Val2,(double)num1,
         (double)num4,(double)num3, 0.0));
  return 0;
}

/* Function to calculate the average of a variable number of arguments */
double average( double v1, double v2,...)
{
  va_list parg;          /* Pointer for variable argument list */
  double sum = v1+v2;    /* Accumulate sum of the arguments */
  double value = 0;      /* Argument value */
  int count = 2;         /* Count of number of arguments */

  va_start(parg,v2);     /* Initialize argument pointer */

  while((value = va_arg(parg, double)) != 0.0)
  {
    sum += value;
    count++;
  }
  va_end(parg);          /* End variable argument process */
  return sum/count;
}
```

编译并运行程序，输出如下：

```
Average = 5.250000
Average = 1.500000
Average = 9.100000
```

代码的说明

这是用不同数目的变元调用 3 次 average( )的结果。可变的变量必须转换成 double 类型，因为这是函数 average()函数假设的变元类型。可以用任何数目的变元调用 average()函数，但最后一个变元必须是 0。

printf()如何处理混合类型？ printf()的第一个变元是带有格式字符的控制字符串，它

www.TopSage.com

提供的信息确定了其后变元的类型和个数。第一个变元后面的变元个数必须匹配控制字符串中格式指定符的数目，这些变元的类型也必须符合对应的格式指定符隐含的类型。如果为要输出的变量指定了错误的类型，输出的结果就不正确。

9.4.1 复制 va_list

有时需要多次处理可变的变元列表。<stdarg.h>头文件为此定义了一个复制已有 va_list 的例程。假定在函数中使用 va_start()创建并初始化了一个 va_list 对象 parg，现在要复制 parg：

```
va_list parg_copy;
copy(parg_copy, parg);
```

第一条语句创建了一个新的 va_list 变量 parg_copy，下一条语句将 parg 的内容复制到 parg_copy 中。接着可以独立地处理 parg 和 parg_copy，使用 va_arg()和 va_end()提取变元值。

注意，copy()例程复制 va_list 对象时，不需要考虑它所处的状态，所以如果用 parg 执行 va_arg()，从列表中提取变元值，之后执行 copy()例程，parg_copy 的状态就与已经提取出来的一些变元值相同。另外注意，在给 parg_copy 执行 pa_end()之前，不能将 va_list 对象 parg_copy 用作另一个复制过程的目的地。

9.4.2 长度可变的变元列表的基本规则

以下是编写变元数目可变的函数的基本规则：

- 在变元数目可变的函数中，至少要有一个固定变元。
- 必须调用 va_start()初始化函数中可变变元列表指针的值。变元指针的类型必须声明为 va_list 类型。
- 必须有确定每个变元类型的机制。可以假设默认的类型，或用一个参数来指定变元的类型。例如，在 average()函数中，可以有另一个固定的变元，它的值是 0 时，表示变元的类型是 double，它的值是 1 时，表示变元的类型是 long。如果在 va_arg()调用中指定的变元类型不对应于调用函数时指定的变元值，函数就不能正常工作。
- 必须有确定何时终止变元列表的方法。例如，在可变的变元列表中，最后一个变元有固定的值，称为“哨兵”值，它可以检测，因为它不同于其他变元的值。或者在某个固定的变元中包含变元的个数或变元列表中的可变变元个数。
- va_arg()的第二个变元指定了变元值的类型，这个指针类型可以在类型名的后面加上*来指定。最好检查一下编译器的文档说明，了解其他限制。
- 在退出变元数目可变的函数前，必须调用 va_end()，否则，函数将不会正常运作。

www.TopSage.com

可以试着修改程序 9.7，更好地了解这个过程。在 average()函数中输出一些信息，看看改变了某些数据后会发生什么。例如，可以在 average()函数的循环中显示 value 和 count，再修改 main()，使用非 double 类型的变元，或调用最后一个变元不是 0 的函数。

9.5 main()函数

main()函数是程序执行的起点。这个函数有一个参数列表，在命令行中执行程序时，可以给它传递变元。main()函数可以有两个参数，也可以没有参数。

main()函数有参数时，第一个参数的类型是 int，表示在命令行中执行 main()函数的参数个数，包含程序名在内。第二个参数是一个字符串指针数组。因此，如果在语句行中，在程序名称的后面添加两个变元，main()函数的第一个变元值就是 3，第二个参数是一个包含 3 个指针的数组，第一个指针指向程序的名称，第二和第三个指针是指向在命令行上输入的两个变元。

```
/* Program 9.8 A program to list the command line arguments */
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("Program name: %s\n", argv[0]);
  for(int i = 1 ; i<argc ; i++)
    printf("\nArgument %d: %s", i, argv[i]);
  return 0;
}
```

argc 的值至少是 1，因为执行程序时，必须输入程序名称。argv[0]是程序名称，argv 数组中的后续元素是在命令行下输入的变元。上述程序在 for 循环中依序输出它们。

这个程序的源文件是 Program9_08，所以输入如下命令来执行它：

```
Program9_08 first second_arg "Third is this"
```

注意，使用双引号包含有空格的变元。这是因为空格一般被看做分隔符。可以将变元放在双引号中，确保将它当作一个变元。

上述命令会创建下面的输出：

```
Program name:   Program9_08

Argument 1:   first
Argument 2:   second_arg
Argument 3:   Third is this
```

最后一个变元放在双引号中，确保将它看做一个变元，而不是 3 个变元。

所有命令行变元都以字符串读入，如果在命令行上输入数值，就需要把包含数值的字符串转换成适当的数值类型。为此可以使用表 9-1 中的函数，这些函数在<stdlib.h>头文件中声明。

www.TopSage.com

表 9-1 将字符串转换为数值的函数

函 数	说 明
atof()	将作为变元传送的字符串转换为 double 类型
atoi()	将作为变元传送的字符串转换为 int 类型
atol()	将作为变元传送的字符串转换为 long 类型

例如，如果需要将一个命令行变元用作整数，可以用下面的方式处理：

```
int arg_value = 0;  /* Stores value of command line argument */
if(argc>1)          /* Verify we have at least one argument */
  arg_value = atoi(argv[1]);
else
{
  printf("Command line argument missing.");
  return 1;
}
```

注意检查变元的个数，在处理命令行变元前，先检查变元的数目是很重要，因为很容易忘记输入变元。

9.6 结束程序

结束程序的方法有几种。执行到 main()函数体的结尾，就等于执行 main()中的 return 语句，结束程序。另外，调用两个在<stdlib.h>头文件中声明的标准库函数可以结束程序，一个是 abort()函数，它可以立即终止程序，并表示程序操作是非正常结束，所以对于正常结束的程序不应使用这个函数。调用 abort()函数结束程序的方式如下：

```
abort(); /* Abnormal program termination */
```

这行语句可以在程序的任何地方使用。

另一个是 exit()函数，它可使程序正常结束。这个函数需要一个整数变元返回给操作系统。一般 0 代表正常结束，其他值以某种方式代表程序的状态。exit()函数返回的值由操作系统来定。exit()函数的调用方式如下：

```
exit(1); /* Normal program end - status is 1 */
```

这行语句可以在程序的任何地方使用。

也可以在 main()函数中用一个整数执行 return 语句，来结束程序。例如：

```
return  1;
```

return 语句在 main()函数中有特殊的意义(其他函数没有)，这相当于使用 return 语句中指定的值调用 exit()函数。因此，return 语句中的值会返回给操作系统。

www.TopSage.com

9.7 函数库：头文件

编译器提供了许多在头文件中声明的标准函数。头文件也称为包含文件，它们是开发应用程序时必要的资源。前面已经见过一些头文件，因为头文件是 C 编程的一个基本组成部分。前面使用过的头文件如表 9-2 所示。

表 9-2 前面使用过的标准头文件

头 文 件	函 数
<stdio.h>	输入输出函数
<stdarg.h>	支持变元个数可变的函数的宏
<math.h>	数学浮点函数
<stdlib.h>	内存分配函数
<string.h>	字符串处理函数
<stdbool.h>	bool 类型和布尔值 true 和 false
<complex.h>	支持复数
<ctype.h>	字符分类函数
<wctype.h>	宽字符转换函数

表 9-2 中的所有头文件包含了各种函数的声明以及各种常量的声明。它们都是 ISO/IEC 标准库，所以所有遵循该标准的编译器都支持它们，至少提供了基本的函数集，但它们一般支持更多的函数。要全面讨论 ISO/IEC 标准库头文件及函数的内容，需要两本书的篇幅，所以在此只介绍最重要的标准头文件，其余内容可参阅编译器的文档说明。

头文件<stdio.h>含有大量高级输入/输出(I/O)函数的声明，所以第 10 章将专门探讨它们，尤其是处理文件的输入/输出函数。

除了内存分配函数外，<stdlib.h>也提供了将字符串转换为 ASCII 表中对应数值的工具。其中的函数还可以排序和搜寻，以及生成随机数。

第 6 章使用过<string.h>头文件，它提供了处理终止空字符串的函数。

<ctype.h>头文件提供了大量与字符串处理相关的函数，这个头文件也在第 6 章中介绍过。<ctype.h>提供的函数可以将字母从大写转换为小写，从小写转换为大写，以及检查字母、数字等字符。这些头文件提供了大量工具，可以分析字符串的内容，对处理用户输入特别有用。

<wctype.h>头文件提供了宽字符转换函数，<wchar.h>提供了多字节字符实用函数。

建议读者花些时间熟悉编译器提供的这些头文件以及库的内容，这样，在用 C 语言开发应用程序时就能轻松许多。

www.TopSage.com

9.8 提高性能

有两个工具可以使编译器生成性能更高的代码。其中一个与短函数调用的编译方式相关，另一个涉及指针的使用。但不能保证其效果，而是取决于编译器的实现方式。这里先探讨短函数。

9.8.1 内联声明函数

C 语言的功能结构要求将程序分解为许多函数，函数有时可以非常短。短函数的每次调用可以用实现该函数功能的内联代码替代，提高执行性能。要采用这种技术，可以内联指定短函数，下面是一个例子：

```
inline double bmi(double kg_wt, double m_height)
{
  return kg_wt/(m_height*m_height);
}
```

这个函数根据成人的体重(Kg)及身高(m)计算其 Body Mass Index。这个操作可以定义为一个函数，也可以使用调用的内联实现方式，因为其代码非常简单。要采用后一种方式，需要在函数头中使用 inline 关键字来指定。但一般不保证编译器能识别声明为 inline 的函数。

9.8.2 使用 restrict 关键字

专业的 C 编译器可以优化对象代码的性能，这涉及到改变在代码中为操作指定的计算顺序。为了优化代码，编译器必须确保操作的这种重新排序不影响计算的结果，并用指针指出这方面的错误。为了优化涉及指针的代码，编译器必须能肯定指针是没有别名的——换言之，每个指针引用的数据项都没有在给定范围内以其他方式引用。关键字 restrict 就可以告诉编译器，何时出现这种情况，并允许应用代码优化功能。

下面是一个在<string.h>中声明的函数：

```
char *strcpy(char * restrict s1, char * restrict s2)
{
  /* Implementation of the function to copy s2 to s1 */
}
```

这个函数将 s2 复制到 s1 中，关键字 restrict 应用于两个参数，表示在函数体中，s1 和 s2 引用的字符串仅通过这两个指针引用，所以编译器可以优化为该函数生成的代码。关键字 restrict 仅将信息告知编译器，但不保证进行优化。当然，如果在条件不具备的代码上应用了关键字 restrict，代码就会生成不正确的结果。

在大多数情况下，不需要使用关键字 restrict，只有代码进行大量计算，进行代码优化才有显著的效果，而这还取决于编译器。

www.TopSage.com

9.9 设计程序

到此函数已经介绍完毕，我们的 C 语言学习之旅也已过半，一些不太复杂的问题应该都可以解决。接下来的这个程序将用到目前学过的各种 C 元素。

9.9.1 问题

现在要编写一个游戏。选择编写游戏程序有几个理由。首先，游戏比其他类型的程序复杂，即使是比较简单的游戏程序。其次，游戏比较有趣!

这个游戏和五子棋或 Microsoft Windows 3.0 的 Reversi 有相同的性质。这个游戏要两位玩家在棋盘上轮流放置不同颜色的棋子，一位玩家使用黑子，另一位玩家使用白子。棋盘是一个偶数边的正方形，图 9-4 显示了从开始位置到连续下五子的过程。

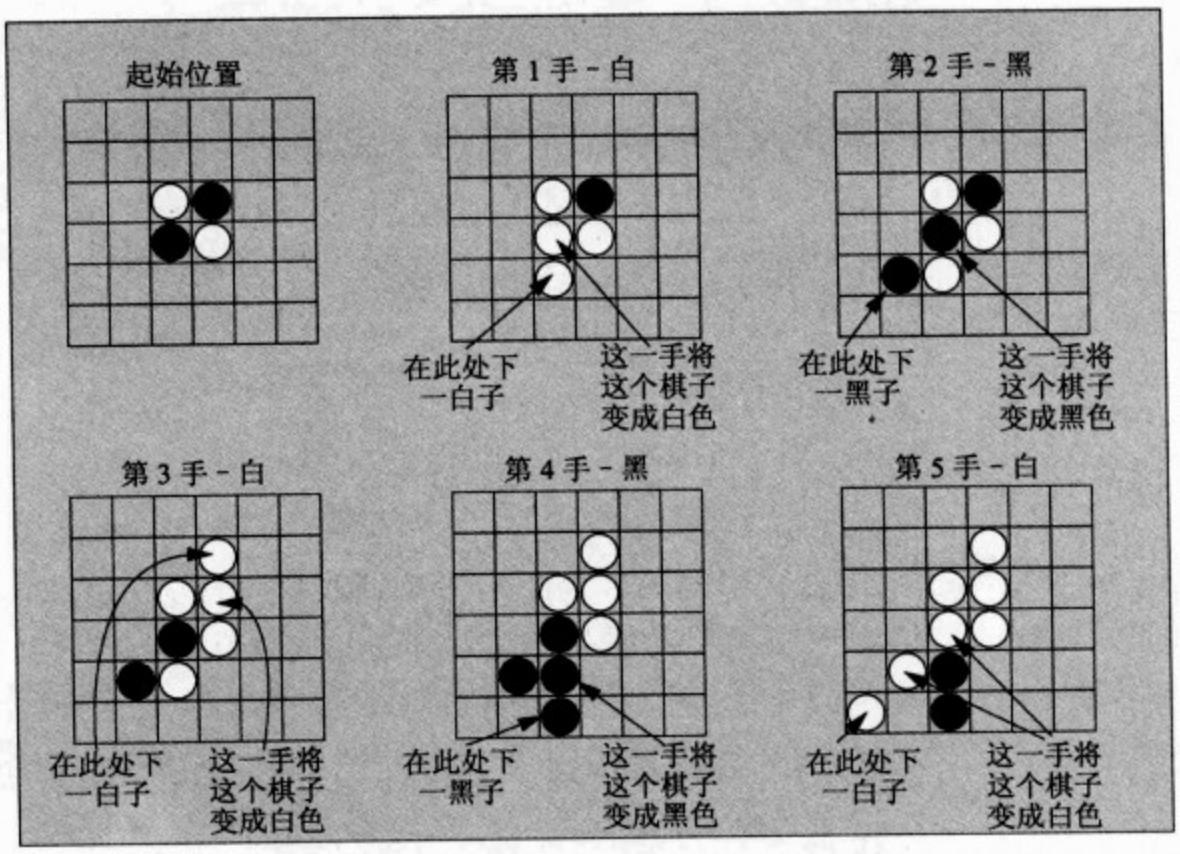


图 9-4 Reversi 中的起始位置和最初的几步

只能将一个棋子放在对手的棋子旁，使对手在对角线、水平线或垂直线上的棋子被自己的棋子包围住，这样对手的棋子就变成自己的棋子了。游戏结束时，棋子多的玩家就获胜。如果所有的方格都放置了棋子，游戏就结束，或者没有玩家在放下棋子后，能将对方的棋子变成自己的，这局也算结束。

这个游戏可以使用任何大小的棋盘，这里使用 6×6 的棋盘，并使一个玩家和计算机对奕。

9.9.2 分析

这个问题的分析和以前所见的稍有不同。本章介绍的重点是结构化编程，换句话说，就是将一个大问题分解成许多小问题逐一解决，这就是为什么要花这么多时间介绍函数

www.TopSage.com

的原因。

最好先用一个图来进行分析。首先有一个方框，它代表整个程序或 main()函数。下一层是要在 main()函数中直接调用的函数，并说明这些函数的功能。再下一层，是这些函数要使用的更小的函数。不必编写出函数，只要写出它们必须完成的工作即可。然而这些工作就是函数要做的工作，所以这是设计程序的一个好方法。图 9-5 显示了程序要执行的任务。

现在可以开始思考动作或函数的执行顺序了。图 9-6 是一个流程图，它不仅描述了这组函数，还描述了这些函数的执行顺序与确定其执行顺序的逻辑。这更精确地说明了程序的运作。

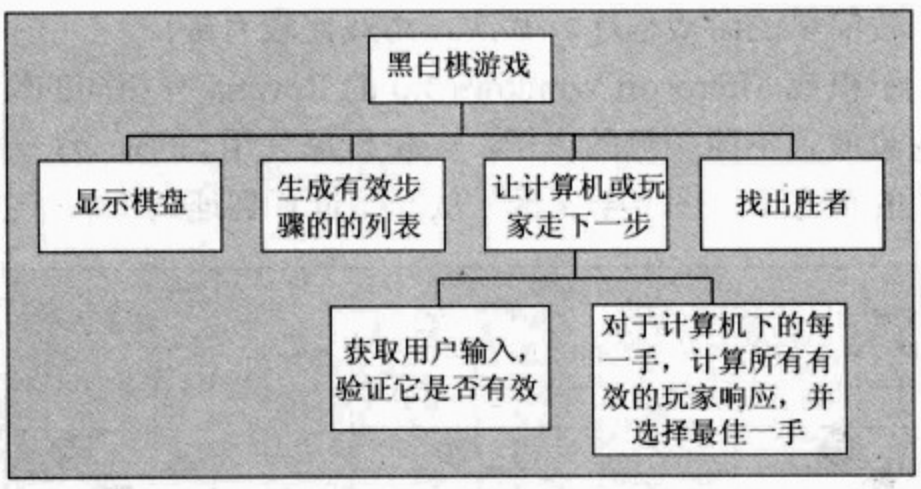


图 9-5 Reversi 程序的任务

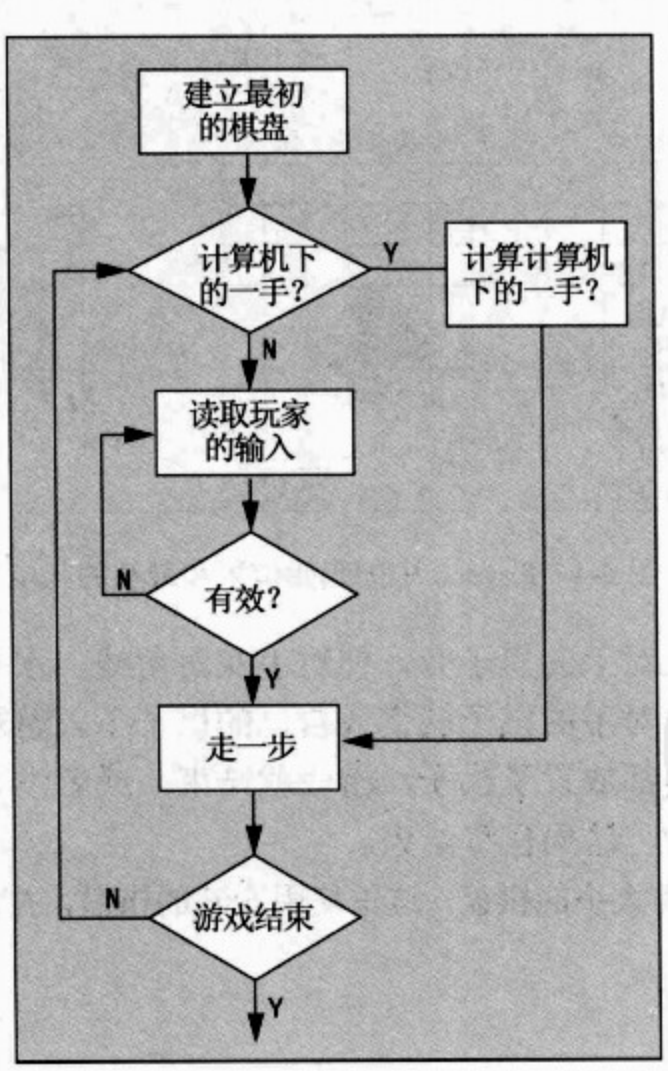


图 9-6 Reversi 程序的基本逻辑

www.TopSage.com

当然，这还没有完成，还必须详细填入许多细节。这种图可以帮助理清程序的逻辑，进而对程序的运作进行更详细的定义。

9.9.3 解决方案

本节列出解决问题的步骤。

1. 步骤 1

首先，建立并显示棋盘。为了使游戏程序比较短，使用比较小的棋盘(6×6)。但这里在程序中通过一个预处理器指令将棋盘的大小设置为一个符号，以便在以后改变棋盘的大小。使用一个独立的函数显示棋盘，因为这是一个自包含的动作。

从声明、初始化及显示棋盘的代码开始。计算机使用@作为棋子，玩家使用 O 作为棋子：

```
/* Program 9.9 REVERSI An Othello type game */
#include <stdio.h>

const int SIZE = 6;          /* Board size - must be even */
const char comp_c = '@';     /* Computer's counter */
const char player_c = 'O'; /* Player's counter */

/* Function prototypes */
void display(char board[][SIZE]);

int main(void)
{
  char board [SIZE][SIZE] = { 0 }; /* The board */
  int row = 0;                     /* Board row index */
  int col = 0;                     /* Board column index */

  printf("\nREVERSI\n\n");
  printf("You can go first on the first game, then we will take turns.\n");
  printf(" You will be white - (%c)\n I will be black - (%c).\n",
           player_c, comp_c);
  printf("Select a square for your move by typing a digit for the row\n "
         "and a letter for the column with no spaces between.\n");
  printf("\nGood luck! Press Enter to start.\n");
  scanf("%c", &again);

  /* Blank all the board squares */
  for(row = 0; row < SIZE; row++)
    for(col = 0; col < SIZE; col++)
      board[row][col] = ' ';

  /* Place the initial four counters in the center */
  int mid = SIZE/2;
```

www.TopSage.com

```
  board[mid - 1][mid - 1] = board[mid][mid] = player_c;
  board[mid - 1][mid] = board[mid][mid - 1] = comp_c;
  display(board); /* Display the board */
  return 0;
}

/*************************************************
* Function to display the board in its           *
* current state with row numbers and column      *
* letters to identify squares.                   *
* Parameter is the board array.                  *
*************************************************/
void display(char board[][SIZE])
{
  /* Display the column labels */
  char col_label = 'a'; /* Column label */
  printf("\n ");                    /* Start top line */
  for(int col = 0 ; col<SIZE ;col++)
    printf(" %c", col_label+col); /* Display the top line */
  printf("\n");                     /* End the top line */

  /* Display the rows... */
  for(int row = 0; row < SIZE; row++)
  {
    /* Display the top line for the current row */
    printf(" +");
    for(int col = 0; col<SIZE; col++)
      printf("---+");
    printf("\n%2d|",row + 1);

    /* Display the counters in current row */
    for(int col = 0; col<SIZE; col++)
      printf(" %c |", board[row][col]); /* Display counters in row */
    printf("\n");
    }

    /* Finally display the bottom line of the board */
    printf(" +");                      /* Start the bottom line */
    for(int col = 0 ; col<SIZE ; col++)
      printf("---+");                  /* Display the bottom line */
    printf("\n");                      /* End the bottom line */
  }
```

display()函数输出棋盘，行用 1~6 表示，列用字母 a~f 表示。这是玩家选择在何处落子的参考系统。

代码并不如看起来那样复杂。第一个循环输出包含列标 a~f 的顶行，下一个循环输出可放棋子的方格，一次输出一行，每行以该行的行号开头。最后一个循环输出最后一行。注意，将 board 数组作为变元传给 display()函数，而不把 board 声明成全局变量，以

www.TopSage.com

避免其他函数无意中更改 board 的内容。这个函数可以输出任何大小的棋盘。

2. 步骤 2

需要一个函数生成当前玩家所有可能的走法。这个函数有两个功用：第一，检查玩家的输入是否有效。第二，确定计算机要走哪一步。但首先必须确定如何表示和存储每次的走法。

需要存储哪些信息，有什么选项？前面定义了一个棋盘，其中的每个棋格都可以用行号和列字母引用。因此，可以把每一步走法存储为包含一个数字和一个字母的字符串。接着需要一个能容纳长度可变的一组走法的空间，允许棋盘的尺寸变成 10×10 或更大。

一个简单的方法是，创建和棋盘大小相同的第二个 bool 类型元素数组，如果在棋盘上的某个棋格中放下的一子是有效的，就在对应的 bool 数组元素中存储 true，否则存储 false。因此函数需要 3 个参数：board 数组、moves 数组和当前玩家的标识。其中 board 数组可以检查是否有空的棋格，moves 数组记录有效的每一步，当前玩家的标识是玩家使用的棋子字符。

其策略是：对每一空格搜寻四周有对手棋子的棋格，找到后，沿着对手棋子所在行的方向(水平、垂直或对角线)，查找自己的棋子。如果找到，就表示可以在这个空格上落下自己的一子。

在 display()函数定义的后面添加这个函数的定义：

```
/* Program 9.9 REVERSI An Othello type game */
#include <stdio.h>
#include <stdbool.h>

const int SIZE = 6;          /* Board size - must be even */
const char comp_c = '@';     /* Computer's counter */
const char player_c = 'O';   /* Player's counter */

/* Function prototypes */
void display(char board[][SIZE]);
int valid_moves(char board[][SIZE], bool moves[][SIZE], char player);

int main(void)
{
  char board [SIZE][SIZE] = { 0 };     /* The board */
  bool moves[SIZE][SIZE] = { false }; /* Valid moves */
  int row = 0;                         /* Board row index */
  int col = 0;                         /* Board column index */

  /* Other code for main as before... */
}

/* Code for definition of display() as before... */
```

www.TopSage.com

```
/************************************************
* Calculates which squares are valid moves     *
* for player. Valid moves are recorded in the  *
* moves array - true indicates a valid move,   *
* false indicates an invalid move.             *
* First parameter is the board array           *
* Second parameter is the moves array          *
* Third parameter identifies the player        *
* to make the move.                            *
* Returns valid move count.                    *
************************************************/
int valid_moves(char board[][SIZE], bool moves[][SIZE], char player)
{
  int rowdelta = 0;          /* Row increment around a square */
  int coldelta = 0;          /* Column increment around a square */
  int x = 0;                 /* Row index when searching */
  int y = 0;                 /* Column index when searching */
  int no_of_moves = 0;       /* Number of valid moves */

  /* Set the opponent */
  char opponent = (player == player_c) ? comp_c : player_c;

  /* Initialize moves array to false */
  for(int row = 0; row < SIZE; row++)
    for(int col = 0; col < SIZE; col++)
      moves[row][col] = false;

/* Find squares for valid moves. */
/* A valid move must be on a blank square and must enclose */
/* at least one opponent square between two player squares */
for(int row = 0; row < SIZE; row++)
  for(int col = 0; col < SIZE; col++)
  {
    if(board[row][col] != ' ') /* Is it a blank square? */
      continue;                /* No - so on to the next */

    /* Check all the squares around the blank square */
    /* for the opponents counter */
    for(rowdelta = -1; rowdelta <= 1; rowdelta++)
      for(coldelta = -1; coldelta <= 1; coldelta++)
      {
        /* Don't check outside the array, or the current square */
        if(row + rowdelta < 0 || row + rowdelta >= SIZE ||
           col + coldelta < 0 || col + coldelta >= SIZE ||
               (rowdelta==0 && coldelta==0))
          continue;

        /* Now check the square */
```

www.TopSage.com

```
if(board[row + rowdelta][col + coldelta] == opponent)
{
  /* If we find the opponent, move in the delta direction */
  /* over opponent counters searching for a player counter */
  x = row + rowdelta; /* Move to */
  y = col + coldelta; /* opponent square */

  /* Look for a player square in the delta direction */
  for(;;)
  {
    x += rowdelta; /* Go to next square */
    y += coldelta; /* in delta direction*/

    /* If we move outside the array, give up */
    if(x < 0 || x >= SIZE || y < 0 || y >= SIZE)
      break;

    /* If we find a blank square, give up */
    if(board[x][y] == ' ')
      break;

    /* If the square has a player counter */
    /* then we have a valid move */
    if(board[x][y] == player)
    {
      moves[row][col] = true; /* Mark as valid */
      no_of_moves++;          /* Increase valid moves count */
      break;                  /* Go check another square */
    }
  }
 }
}
return no_of_moves;
}
```

增加了 valid_moves()的函数原型，并在 main()函数中声明了 moves 数组。因为棋子是 player_c 或 comp_c，所以可以在 valid_moves()函数中将对手的棋子设定为不是自己的棋子，为此可以使用条件运算符，然后在第一个嵌套循环中将 moves 数组设定成 false，这样就只需将有效的位置设定 true，第二个嵌套循环遍历棋盘里所有的棋格，查找其中的空格，找到一个空格后，在内层循环中寻找对手的棋子：

```
/* Check all the squares around the blank square */
/* for the opponents counter */
for(rowdelta = -1; rowdelta <= 1; rowdelta++)
  for(coldelta = -1; coldelta <= 1; coldelta++)
  ...
```

www.TopSage.com

这会遍历空格四周的所有棋格，包含空格本身，所以用下面的 if 语句跳过当前的空格以及棋盘外的位置：

```
/* Don't check outside the array, or the current square */
if(row + rowdelta < 0 || row + rowdelta >= SIZE ||
   col + coldelta < 0 || col + coldelta >= SIZE ||
     (rowdelta==0 && coldelta==0))
  continue;
```

如果通过了这个检查，表示在棋盘上找到一个非空的棋格。如果它含有对手的棋子，就在该棋子所在的方向移动，寻找对手或自己的棋子。如果找到自己的棋子，就可以在原来的空格上落子，并记录这一步。如果找到的是空格，或出了棋盘，表示它是无效的一步，应继续查找另一个空格。

这个函数会返回有效走法的个数，可以使用这个返回值表示函数是否返回了有效的落子处。注意，正整数表示 true，0 表示 false。

3. 步骤 3

现在可以在 main()函数的游戏循环中生成含有所有有效走法的数组了。根据前面的流程图，需要加入两个嵌套的 do-while 循环：外面的循环初始化每一次游戏，里面的循环让玩家和计算机轮流下子。

```
/* Program 9.9 REVERSI An Othello type game */
#include <stdio.h>
#include <stdbool.h>

const int SIZE = 6;          /* Board size - must be even */
const char comp_c = '@';     /* Computer's counter */
const char player_c = 'O';   /* Player's counter */

/* Function prototypes */
void display(char board[][SIZE]);
int valid_moves(char board[][SIZE], bool moves[][SIZE], char player);

int main(void)
{
  char board [SIZE][SIZE] = { 0 };     /* The board */
  bool moves[SIZE][SIZE] = { false }; /* Valid moves */
  int row = 0;                         /* Board row index */
  int col = 0;                         /* Board column index */
  int no_of_games = 0;                 /* Number of games */
  int no_of_moves = 0;                 /* Count of moves */
  int invalid_moves = 0;               /* Invalid move count */
  int comp_score = 0;                  /* Computer score */
  int user_score = 0;                  /* Player score */
  char again = 0;                      /* Replay choice input */
```

www.TopSage.com

```
/* Player indicator: true for player and false for computer */
bool next_player = true;

/* Prompt for how to play - as before */

/* The main game loop */
do
{
  /* On even games the player starts; */
  /* on odd games the computer starts */
  next_player = !next_player;
  no_of_moves = 4; /* Starts with four counters */

  /* Blank all the board squares */
  for(row = 0; row < SIZE; row++)
    for(col = 0; col < SIZE; col++)
     board[row][col] = ' ';

  /* Place the initial four counters in the center */
  int mid = SIZE/2;
  board[mid - 1][mid - 1] = board[mid][mid] = player_c;
  board[mid - 1][mid] = board[mid][mid - 1] = comp_c;
  /* The game play loop */
  do
  {
    display(board); /* Display the board */
    if(next_player = !next_player)
    { /* It is the player's turn */
      /* Code to get the player's move and execute it */
    }
    else
    { /* It is the computer's turn */
      /* Code to make the computer's move */
    }
}while(no_of_moves < SIZE*SIZE && invalid_moves<2);

/* Game is over */
display(board); /* Show final board */

/* Get final scores and display them */
comp_score = user_score = 0;
for(row = 0; row < SIZE; row++)
  for(col = 0; col < SIZE; col++)
  {
    comp_score += board[row][col] == comp_c;
    user_score += board[row][col] == player_c;
  }
printf("The final score is:\n");
```

www.TopSage.com

```
    printf("Computer %d\n User %d\n\n", comp_score, user_score);

    printf("Do you want to play again (y/n): ");
    scanf(" %c", &again);          /* Get y or n */
  }while(tolower(again) == 'y'); /* Go again on y */

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

/* Code for definition of display() */

/* Code for definition of valid_moves() */
```

现在还不能运行这个程序，因为还没有编写代码，处理用户或计算机下的子。此时，循环是无限的，能输出棋盘，但没有下新的子，接下来就完成这个部分。

变量 player 确定轮到谁下子了。当 Player 是 false 时，就该计算机下子了，当 player 是 true 时，就该玩家下子了。player 最初设置为 true，在 do-while 循环中将 player 设置为! player，就可以使玩家和计算机轮流下子。为了确定下一个是计算机还是玩家，翻转变量 player 值，在 if 语句中测试其结果，该结果会自动让玩家和计算机轮流下子。

变量 no_of_moves 中的计数器值到达棋盘上所有方格的总数 SIZE*SIZE 时，或者变量 invalid_moves 的值到达 2 时，游戏结束。每走一步，就将 invalid_moves 的值设定成 0，每次某步走法无效时，就递增该值。因此，只要连续两子无效，invalid_moves 的值就到达 2，表示两个玩家都不能再下子了。游戏结束后，输出最后的棋盘和结果，并提供继续游戏的选择。

现在，在 main()函数中添加代码，让玩家和计算机轮流下子：

```
/* Program 9.9 REVERSI An Othello type game */
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h>
#include <string.h>

const int SIZE = 6;          /* Board size - must be even */
const char comp_c = '@';     /* Computer's counter */
const char player_c = 'O'; /* Player's counter */

/* Function prototypes */
void display(char board[][SIZE]);
int valid_moves(char board[][SIZE], bool moves[][SIZE], char player);
void make_move(char board[][SIZE], int row, int col, char player);
void computer_move(char board[][SIZE], bool moves[][SIZE], char player);

int main(void)
{
  char board [SIZE][SIZE] = { 0 };     /* The board */
```

www.TopSage.com

```
bool moves[SIZE][SIZE] = { false }; /* Valid moves */
int row = 0;                        /* Board row index */
int col = 0;                        /* Board column index */
int no_of_games = 0;                /* Number of games */
int no_of_moves = 0;                /* Count of moves */
int invalid_moves = 0;              /* Invalid move count */
int comp_score = 0;                 /* Computer score */
int user_score = 0;                 /* Player score */
char y = 0;                         /* Column letter */
int x = 0;                          /* Row number */
char again = 0;                     /* Replay choice input */

/* Player indicator: true for player and false for computer */
bool next_player = true;

/* Prompt for how to play - as before */

/* The main game loop */
do
  {
  /* The player starts the first game */
  /* then they alternate */
  next_player = !next_player;
  no_of_moves = 4;                  /* Starts with four counters */

/* Blank all the board squares */
for(row = 0; row < SIZE; row++)
  for(col = 0; col < SIZE; col++)
    board[row][col] = ' ';

/* Place the initial four counters in the center */
board[SIZE/2 - 1][SIZE/2 - 1] = board[SIZE/2][SIZE/2] = 'O';
board[SIZE/2 - 1][SIZE/2] = board[SIZE/2][SIZE/2 - 1] = '@';

/* The game play loop */
do
{
  display(board);                   /* Display the board */
  if(next_player=!next_player) /* Flip next player */
    { /* It is the player's turn */
      if(valid_moves(board, moves, player_c))
      {
        /* Read player moves until a valid move is entered */
        for(;;)
        {
          printf("Please enter your move (row column): ");
          scanf(" %d%c", &x, &y);  /* Read input */
          y = tolower(y) - 'a';    /* Convert to column index */
```

www.TopSage.com

```
      x--; /* Convert to row index */
      if( x>=0 && y>=0 && x<SIZE && y<SIZE && moves[x][y])
      {
        make_move(board, x, y, player_c);
        no_of_moves++; /* Increment move count */
        break;
      }
      else
        printf("Not a valid move, try again.\n");
      }
    }
    else              /* No valid moves */
    if(++invalid_moves<2)
    {
      printf("\nYou have to pass, press return");
      scanf("%c", &again);
    }
    else
      printf("\nNeither of us can go, so the game is over.\n");
    }
    else
    { /* It is the computer's turn */
      if(valid_moves(board, moves, '@')) /* Check for valid moves */
      {
        invalid_moves = 0;  /* Reset invalid count */
        computer_move(board, moves, '@');
        no_of_moves++;      /* Increment move count */
      }
      else
      {
      if(++invalid_moves<2)
        printf("\nI have to pass, your go\n"); /* No valid move */
      else
        printf("\nNeither of us can go, so the game is over.\n");
      }
    }
  }while(no_of_moves < SIZE*SIZE && invalid_moves<2);

  /* Game is over */
  display(board); /* Show final board */

  /* Get final scores and display them */
  comp_score = user_score = 0;
  for(row = 0; row < SIZE; row++)
    for(col = 0; col < SIZE; col++)
    {
        comp_score += board[row][col] == comp_c;
        user_score += board[row][col] == player_c;
    }
```

```
            printf("The final score is:\n");
            printf("Computer %d\n User %d\n\n", comp_score, user_score);

            printf("Do you want to play again (y/n): ");
            scanf(" %c", &again);  /* Get y or n */
    }while(tolower(again) == 'y'); /* Go again on y */

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

/* Code for definition of display() */

/* Code for definition of valid_moves() */
```

处理下子的代码使用了两个新函数，并给它们添加了函数原型。make_move()函数表示下一子，computer_move()函数计算计算机下的一子。对于玩家，使用 if 语句为有效的走法计算 moves 数组：

```
if(valid_moves(board, moves, player_c))
...
```

如果返回值是正的，就表示走法有效，因此读入玩家所选方格的行号和列字母：

```
printf("Please enter your move: "); /* Prompt for entry */
scanf(" %d%c", &x, &y);             /* Read input */
```

将行号减 1，列字母减 a，将行号和列字母转换成索引值。调用 tolower()函数，以保证在 y 中输入的值是小写字母。当然，必须为这个函数包含头文件<ctype.h>对于有效的走法，索引值必须在数组的范围内，且 moves[x][y]必须是 true：

```
if( x>=0 && y>=0 && x<SIZE && y<SIZE && moves[x][y])
...
```

如果找到一个有效的棋格，就调用函数 make_move()在该棋格中下一子，这个函数稍后编写(注意目前这个程序还不能编译，因为程序还没有定义这个函数)。

如果玩家选择的是一个无效的棋格，就递增变量 invalid_moves 的值。如果它的值仍然小于 2，就输出“这一步不能走”的信息，继续下一个迭代，让计算机下子。如果 invalid_moves 的值不小于 2，此时 do-while 循环中控制游戏的条件会变成 false，于是输出一条信息，并结束游戏。

如果计算机下的子是有效的，就调用 computer_move()函数走这一步，并递增移动次数。计算机选择无效棋格的处理方法和玩家一样。

接下来添加 make_move()函数的定义。要下一子，必须在所选的棋格上放置一个适当的棋子，并将被当前这个玩家棋子围住的对手棋子翻转过来。现在在源文件中添加这个函数的代码，这里没有重复其他代码：

www.TopSage.com

```
/*****************************************************************
* Makes a move. This places the counter on a square and reverses  *
* all the opponent's counters affected by the move.               *
* First parameter is the board array.                             *
* Second and third parameters are the row and column indices.     *
* Fourth parameter identifies the player.                         *
*****************************************************************/
void make_move(char board[][SIZE], int row, int col, char player)
{
  int rowdelta = 0;           /* Row increment */
  int coldelta = 0;           /* Column increment */
  int x = 0;                  /* Row index for searching */
  int y = 0;                  /* Column index for searching */

  /* Identify opponent */
  char opponent = (player == player_c) ? comp_c : player_c;

  board[row][col] = player; /* Place the player counter */

  /* Check all the squares around this square */
  /* for the opponents counter */
  for(rowdelta = -1; rowdelta <= 1; rowdelta++)
    for(coldelta = -1; coldelta <= 1; coldelta++)
    {
      /* Don't check off the board, or the current square */
      if(row + rowdelta < 0 || row + rowdelta >= SIZE ||
         col + coldelta < 0 || col + coldelta >= SIZE ||
         (rowdelta==0 && coldelta== 0))
       continue;

/* Now check the square */
if(board[row + rowdelta][col + coldelta] == opponent)
{
  /* If we find the opponent, search in the same direction */
  /* for a player counter */
  x = row + rowdelta;       /* Move to opponent */
  y = col + coldelta;       /* square */

  for(;;)
  {
    x += rowdelta;          /* Move to the */
    y += coldelta;          /* next square */

    /* If we are off the board give up */
    if(x < 0 || x >= SIZE || y < 0 || y >= SIZE)
      break;

    /* If the square is blank give up */
    if(board[x][y] == ' ')
```

www.TopSage.com

```
            break;

          /* If we find the player counter, go backward from here */
          /* changing all the opponents counters to player */
          if(board[x][y] == player)
          {
            while(board[x-=rowdelta][y-=coldelta]==opponent)
             /* Opponent? */
            board[x][y] = player;  /* Yes, change it */
            break;                 /* We are done */
        }
      }
    }
  }
}
```

这个函数的逻辑和检查走法是否有效的 valid_moves()函数很类似。第一步是搜寻参数 row 和 col 索引的方格的四周，找出对手的棋子。这用以下的嵌套循环来完成：

```
for(rowdelta = -1; rowdelta <= 1; rowdelta++)
  for(coldelta = -1; coldelta <= 1; coldelta++)
  {
  ...
  }
```

找到一个对手棋子时，就在一个无限 for 循环中沿着该棋子所在的方向上寻找自己的棋子。如果出了棋盘或找到空的棋格，就跳出 for 循环，在外层循环中移动到所选棋格的下一个棋格上。如果找到一个自己的棋子，就将所有对手的棋子变成自己的棋子。

```
/* If we find the player counter, go backward from here */
/* changing all the opponents counters to player */
if(board[x][y] == player)
{
  while(board[x-=rowdelta][y-=coldelta]==opponent)  /* Opponent? */
  board[x][y] = player;                             /* Yes, change it */
  break;                                            /* We are done */
}
```

break 语句可以中断 for 无限循环。

有了这个函数后，就可以进入程序中最难的部分：编写让计算机下子的函数。这里采用一个相当简单的策略来确定计算机的走法。这个策略就是算出计算机所有可能的有效走法，对于计算机每个有效的走法，都要判断玩家可能采用哪个走法，并确定该走法的分数。然后选择计算机走哪一步，能使玩家的走法分数最低。

在编写 computer_move()之前，要先编写两个辅助函数。辅助函数可帮助实现一个操作，这里是实现计算机的下子。第一个辅助函数是 get_score()，它计算棋盘上特定位置的分数。在源文件的最后添加如下代码：

www.TopSage.com

```
/*****************************************************************
* Calculates the score for the current board position for the    *
* player. player counters score +1, opponent counters score -1   *
* First parameter is the board array                             *
* Second parameter identifies the player                         *
* Return value is the score.                                     *
*****************************************************************/
int get_score(char board[][SIZE], char player)
{
  int score = 0; /* Score for current position */

  /* Identify opponent */
  char opponent = (player == player_c) ? comp_c : player_c;

  /* Check all board squares */
  for(int row = 0; row < SIZE; row++)
    for(int col = 0; col < SIZE; col++)
    {
      score -= board[row][col] == opponent; /* Decrement for opponent */
      score += board[row][col] == player;  /* Increment for player */
    }
  return score;
}
```

这个函数相当简单。分数的计算是给棋盘上每个玩家的棋子加 1，给每个对手的棋子减 1。

下一个辅助函数是 best_move()，它计算并返回玩家当前有效走法中最佳走法的分数。代码如下：

```
/*****************************************************************
* Calculates the score for the best move out of the valid moves  *
* for player in the current position.                            *
* First parameter is the board array                             *
* Second parameter is the moves array defining valid moves.      *
* Third parameter identifies the player                          *
* The score for the best move is returned                        *
*****************************************************************/
int best_move(char board[][SIZE], bool moves[][SIZE], char player)
{
  /* Identify opponent */
  char opponent = (player == player_c) ? comp_c : player_c;

  char new_board[SIZE][SIZE] = { 0 }; /* Local copy of board */
  int score = 0;                      /* Best score */
  int new_score = 0;                  /* Score for current move */

  /* Check all valid moves to find the best */
  for(int row = 0 ; row<SIZE ; row++)
    for(int col = 0 ; col<SIZE ; col++)
```

www.TopSage.com

```
    {
    if(!moves[row][col])                /* Not a valid move? */
      continue;                         /* Go to the next */

    /* Copy the board */
    memcpy(new_board, board, sizeof(new_board));

    /* Make move on the board copy */
    make_move(new_board, row, col, player);

    /* Get score for move */
    new_score = get_score(new_board, player);
    if(score<new_score)   /* Is it better? */
      score = new_score;  /* Yes, save it as best score */
  }
  return score;           /* Return best score */
}
```

必须在main()函数之前，为这两个辅助函数添加函数原型：

```
/* Function prototypes */
void display(char board[][SIZE]);
int valid_moves(char board[][SIZE], bool moves[][SIZE], char player);
void make_move(char board[][SIZE], int row, int col, char player);
void computer_move(char board[][SIZE], bool moves[][SIZE], char player);
int best_move(char board[][SIZE], bool moves[][SIZE], char player);
int get_score(char board[][SIZE], char player);
```

4. 步骤4

程序的最后一部分是computer_move()函数的实现代码，如下：

```
/*************************************************************
* Finds the best move for the computer. This is the move for  *
* which the opponent's best possible move score is a minimum. *
* First parameter is the board array.                         *
* Second parameter is the moves array containing valid moves. *
* Third parameter identifies the computer.                    *
*************************************************************/
void computer_move(char board[][SIZE], bool moves[][SIZE], char player)
{
  int best_row = 0;                 /* Best row index */
  int best_col = 0;                 /* Best column index */
  int new_score = 0;                /* Score for current move */
  int score = 100;                  /* Minimum opponent score */
  char temp_board[SIZE][SIZE];      /* Local copy of board */
  bool temp_moves[SIZE][SIZE];      /* Local valid moves array */

  /* Identify opponent */
  char opponent = (player == player_c) ? comp_c : player_c;
```

www.TopSage.com

```
/* Go through all valid moves */
for(int row = 0; row < SIZE; row++)
  for(int col = 0; col < SIZE; col++)
  {
    if( !moves[row][col] )
      continue;

    /* First make copies of the board array */
    memcpy(temp_board, board, sizeof(temp_board));

    /* Now make this move on the temporary board */
    make_move(temp_board, row, col, player);

    /* find valid moves for the opponent after this move */
    valid_moves(temp_board, temp_moves, opponent);

    /* Now find the score for the opponent's best move */
    new_score = best_move(temp_board, temp_moves, opponent);

    if(new_score<score)   /* Is it worse? */
    {                     /* Yes, so save this move */
      score = new_score;  /* Record new lowest opponent score */
      best_row = row;     /* Record best move row */
      best_col = col;     /* and column */
    }
  }
  /* Make the best move */
  make_move(board, best_row, best_col, player);
}
```

这两个辅助函数并不难，它们选择的走法要让对手后面的最佳走法分数最低。

主循环用计数器 row 和 col 控制，玩家和计算机轮流在棋盘的副本上下子，当前棋盘的副本存储在本地数组 temp_board 上。每次下子后，都调用 valid_moves()函数计算对手在该位置上的有效走法，并将结果保存到 temp_moves 数组中。然后，调用 best_move()函数，从 temp_moves 数组存储的有效走法中得到对手最佳走法的分数。如果该分数低于之前任何一个分数，就存储这个分数，把该棋格的行和列索引作为计算机的最佳走法。

变量 score 初始化为高于任何可能的分数，并设法使这个变量最小化(因为这是对手下一步的分数)，以找到计算机的最佳走法移动。试完了计算机的所有有效走法后，best_row 和 best_col 包含的行和列索引就会使对手下一步的分数最小，然后调用 make_move()函数，让计算机用最佳走法下子。

现在可以编译并执行游戏程序了。程序开始时游戏如下：

www.TopSage.com

```
  a   b   c   d   e   f
 +---+---+---+---+---+---+
1|   |   |   |   |   |   |
 +---+---+---+---+---+---+
2|   |   |   |   |   |   |
 +---+---+---+---+---+---+
3|   |   | O | @ |   |   |
 +---+---+---+---+---+---+
4|   |   | @ | O |   |   |
 +---+---+---+---+---+---+
5|   |   |   |   |   |   |
 +---+---+---+---+---+---+
6|   |   |   |   |   |   |
 +---+---+---+---+---+---+
Please enter your move: 3e
   a   b   c   d   e   f
  +---+---+---+---+---+---+
1 |   |   |   |   |   |   |
  +---+---+---+---+---+---+
2 |   |   |   |   |   |   |
  +---+---+---+---+---+---+
3 |   |   | O | O | O |   |
  +---+---+---+---+---+---+
4 |   |   | @ | O |   |   |
  +---+---+---+---+---+---+
5 |   |   |   |   |   |   |
  +---+---+---+---+---+---+
6 |   |   |   |   |   |   |
  +---+---+---+---+---+---+
```

计算机这方玩得不是很好，因为它只会向前移动一步，不会在边缘和角落下子。

棋盘只有 6×6，如果要改变棋盘的大小，可以将 SIZE 改为另一个偶数。程序仍然可以运行。

9.10 小结

如果读者到目前为止都没有遇到什么大问题，说明您将成为一位有能力的 C 程序员。本章和前一章介绍了编写结构优秀的 C 程序所需的所有知识，函数结构是 C 语言的核心，要尽量使函数短小精悍、意图明确。这是优秀 C 代码的本质。现在读者应该能够使用函数结构去处理自己的编程问题了。

指针非常灵活，可大大简化许多编程问题，应常用它们作为函数变元和返回值，把这作为一个习惯。如果还不是很有自信，就复习本章的程序，因为熟能生巧。之后就可以处理自己的一些问题。

C 语言中还有一个新的领域未介绍，即数据处理和数据的结构化。这部分将在第 11

www.TopSage.com

章详细介绍。在这之前，还必须详细探讨输入和输出。处理输入和输出是很重要的、很吸引人的编程方面，这是下一章的主题。

9.11 习题

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容。还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 9.1 函数原型：

```
double power(double x, int n);
```

会计算并返回 x^n。因此 power(5.0，4)会计算 5.0*5.0*5.0*5.0，它的结果是 625.0。将 power()函数实现为递归函数，再用适当的 main()版本演示它的操作。

习题 9.2 函数原型：

```
double add(double a, double b);       /* Returns a+b */
double subtract(double a, double b); /* Returns a-b */
double multiply(double a, double b); /* Returns a*b */
double array_op(double array[], int size, double (*pfun)(double,double));
```

array_op()函数的参数是：要运算的数组、数组元素数目以及一个函数指针，该函数指针指向的函数定义了在连续几个元素上进行的操作。在实现 array_op()函数时，将 subtract()函数传送为第三个参数，subtract()函数会用交替符号组合这些元素。因此，对于有 4 个元素 x1、x2、x3、x4 的数组，subtract()函数会计算 x1 - x2+x3 - x4 的值。

用适当的 main()版本演示这些函数的运作。

习题 9.3 定义一个函数，它的参数是字符串数组指针，返回一个将所有字符串合并起来的字符串指针，每个字符串都用换行符来终止。如果输入数组中的原字符串将换行符作为最后一个字符，函数就不能给字符串添加另一个换行符。编写一个程序，从键盘读入几个字符串，用这个函数输出合并后的字符串。

习题 9-4 一个函数的原型是：

```
char *to_string(int count, double first, ...);
```

这个函数返回一个字符串，这个字符串含有第二及其后参数的字符串表示，每个参数都有两位小数，参数间用逗号隔开。第一个参数是从第二个参数算起的参数个数。编写一个 main()函数，演示这个函数的运作。

www.TopSage.com

第 10 章

基本输入和输出操作

本章将详细介绍键盘输入、屏幕输出和打印机输出。本章的内容相当简单，但要记住许多东西。不过不用熟记，需要时可以再回来参考本章的内容。

与大多数现代编程语言一样，C 语言也没有输入输出的能力，所有这类操作都由标准库中的函数提供。前面各章介绍的许多这类函数提供了键盘输入和屏幕输出的功能。

本章将按顺序总结输入输出的内容，并介绍前面没有解释的内容，再加入一些打印的内容，因为打印通常是程序的重要部分。本章的内容非常简单，不需要用一个程序解决方案来演示。

本章的主要内容：

- 如何从键盘读入数据
- 如何将数据格式化后输出到屏幕上
- 如何处理字符输出
- 如何把数据输出到打印机上

10.1 输入和输出流

前面章节主要使用 scanf()函数从键盘输入数据，使用 printf()函数将数据输出到屏幕上。事实上，使用这些函数指定从哪里输入或输出到哪里去的方式没有什么特别。因为 scanf()函数可以从任何地方接收信息，只要这些信息是字符流即可。同样，printf()函数也可以将数据输出到任何能接收字符流的地方去。这并不是巧合：C 语言的标准输入输出函数都是独立于设备的，程序员不需要考虑如何在特定设备上传入传出数据。C 语言的库函数和操作系统会确保在特定设备上的操作完全正常。

C 语言中的每个输入源和输出目的地都称为流(stream)。输入流是可读入程序的数据源，而输出流是程序输出数据的终点。流和设备的实体(如屏幕或键盘)相互独立。程序使用的每个设备通常都有一个或多个相关的流，这取决于它是简单的输入设备(如键盘)、输出设备(如打印机)，还是可输入输出的设备(如磁盘驱动器)。如图 10-1 所示。

磁盘驱动器可以有多个输入输出流，因为它可以含有多个文件。流和文件一一对应，而不是流和设备一一对应。一个流可以和磁盘里的一个文件关联，与文件关联的流是一个输入流，所以可以从这个文件中读取数据。如果这个流是输出流，就可以在这个文件中写入数据；如果这个流允许输入和输出，就可以对这个文件进行读取和写入。很明显，

www.TopSage.com

如果和文件关联的流是输入流，这个文件就曾经被写入过，所以包含一些数据。流也可以和光盘驱动器里的文件关联，因为光盘驱动器一般是只读的，因此这个流必然是一个输入流。

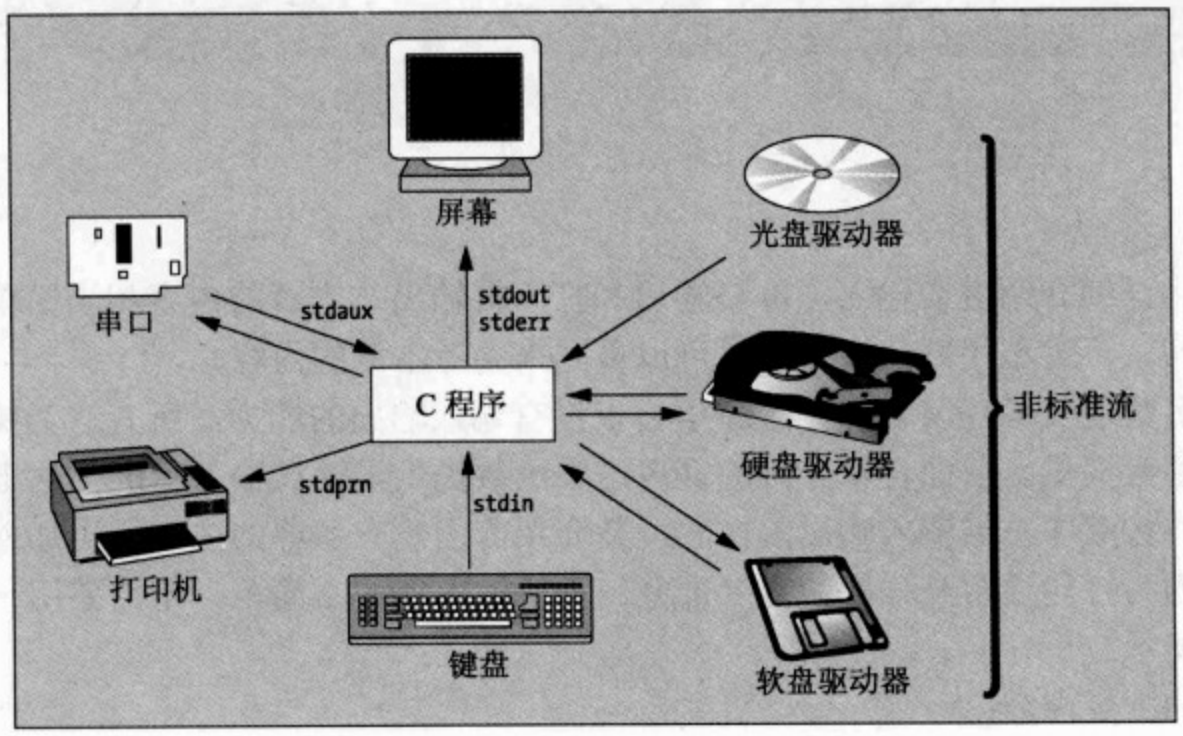


图 10-1 标准流和非标准流

还有两种流：字符流(character streams，也称为文本流)和二进制流(binary streams)。它们之间的主要差异是，在字符流中传入传出的数据是一系列字符，可以根据格式规范由库例程修改。在二进制流中传入传出的数据是一系列字节，不能以任何方式修改。在第 12 章讨论读写磁盘文件时，会讨论二进制流。

10.2 标准流

C 语言有 3 个在<stdio.h>头文件中预定义的标准流，程序只要包含了这个头文件，就可以使用这些流。这 3 个标准流分别是 stdin、stdout 和 stderr。在一些系统中还可以使用另外两个流 stdprn 和 stdaux，但它们不是 C 语言的标准流，所以编译器不支持它们。

使用这些流不需做任何准备，只是要使用适当的库函数给它们发送数据，它们都预先赋予了特定的物理设备，如表 10-1 所示。

表 10-1 标 准 流

流	设 备
stdin	键盘
stdout	显示屏幕
stderr	显示屏幕

www.TopSage.com

(续表)

流	设　备
stdprn	打印机
stdaux	串口

本章主要介绍标准输入流 stdin、标准输出流 stdout 以及打印机流 stdprn。

stderr 流只是将来自 C 库的错误信息传送出去，也可以将自己的错误信息传送给 stderr。stderr 和 stdout 之间的主要差别是，输出到 stdout 的流在内存上缓存，所以写入 stdout 的数据不会马上送到设备上。而 stderr 不缓存，所以写入 stderr 的数据会立刻传送到设备上。对于缓存的流，程序会在内存中传入或传出缓存区域的数据，在物理设备上传入或传出数据可以异步进行。这使输入输出操作更高效。为错误信息使用不缓存的流，其优点是可以确保错误信息显示出来，但输出操作是低效的。缓存的流比较高效，但如果程序因某种原因而失败，缓存的流就不会刷新，所以输出可能永远不会显示出来。这里不进一步探讨它们，只是要知道 stderr 只能输出到屏幕上，而不能重定向到其他设备上。stdaux 流将数据传送到串口上，但它超出了本书的范围，限于篇幅(而不是复杂性)，本书不讨论它。

使用操作系统命令，stdin 和 stdout 都可以重定向到文件上，而不是默认的键盘和屏幕上。这就带来了极大的灵活性。如果要在测试过程中用相同的数据多次运行程序，就可以把这些数据放在一个文本文件中，将 stdin 重定向到这个文件上。这样每次运行程序时，就无须输入这些数据了。将程序的输出重定向到文件上，很容易保留这些输出，以便以后参考，也可以使用文本编辑器访问或搜索它。

10.3 键盘输入

前面介绍过，stdin 上的键盘输入有两种形式：一种是格式化输入，主要由 scanf()函数提供；另一种是非格式化输入，通过 getchar()等函数接收原始的字符数据。这两种形式都很常见，下面详细介绍它们。

10.3.1 格式化键盘输入

函数 scanf()从 stdin 流中读入字符，并根据格式控制字符串中的格式指定符，将它们转换成一个或多个值。scanf()函数的原型如下：

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

格式控制字符串参数的类型是 char*，即字符串指针。在函数调用时，它通常显示为显式的变元，如下：.

```
scanf("%lf",  &variable);
```

www.TopSage.com

也可以写成：

```
char  str[] = "%lf";
scanf(str, &variable);
```

scanf()函数使用第 9 章介绍的参数个数可变的功能。格式控制字符串基本上描述了scanf()如何将传入的字符流转换成所需的值。在格式控制字符串的后面可以有一个或多个可选参数，每个参数都是存储转换后的输入值的一个地址。这意味着，每个参数都必须是一个指针，或带&前缀的变量名，来定义变量的地址，而不是变量值。

scanf()函数从 stdin 中读入数据，直到格式控制字符串结束为止，或某个错误条件停止了输入过程为止。这种错误一般是读入的数据不匹配当前格式指定符所致。注意，scanf()的返回值是读入的输入值个数。因此，比较 scanf()的返回值和期望的输入值个数，就可以检测是否发生了错误。

wscanf()函数与 scanf()的功能相同，只是 wscanf()函数的第一个参数是格式控制字符串，它必须是 wchar_t *类型的宽字符串。

因此，可以使用 wscanf()从键盘上读取一个浮点值，如下所示：

```
wscanf(L"%lf", &variable);
```

第一个参数是宽字符串常量，该函数的其他方面都与 scanf()相同。如果省略表示宽字符串字面量的 L，编译器就会生成一条错误信息，因为变元不匹配第一个参数的类型。当然，也可以编写如下代码：

```
wchar_t wstr[] = L"%lf";
wscanf(wstr, &variable);
```

10.3.2　输入格式控制字符串

在 scanf()或 wscanf()函数中使用的格式控制字符串不完全类似于 printf()中的格式控制字符串。在格式控制字符串中添加一个或多个空白字符，如空格' '、制表符'\t'或换行符'n'，scanf()会忽略空白字符，直接读入输入中的下一个非空白字符。在格式控制字符串中只要出现一个空白字符，就会造成无数个连续的空白字符被忽略。因此，可以在格式字符串内加入任意多个的空白字符，使输入易于理解。注意，scanf()默认忽略空白字符，但使用%c、%[]或%n 指定符读取数据时除外。

scanf()会读入任何非空白字符(除了%外)，但不会存储这个连续出现的字符。例如，scanf()需要忽略输入中分隔各个值的逗号，只需在格式字符串的前面加上逗号。还有其他区别，详见本章后面的“屏幕输出”一节。

格式指定符最常见的形式如图 10-2 所示。

www.TopSage.com

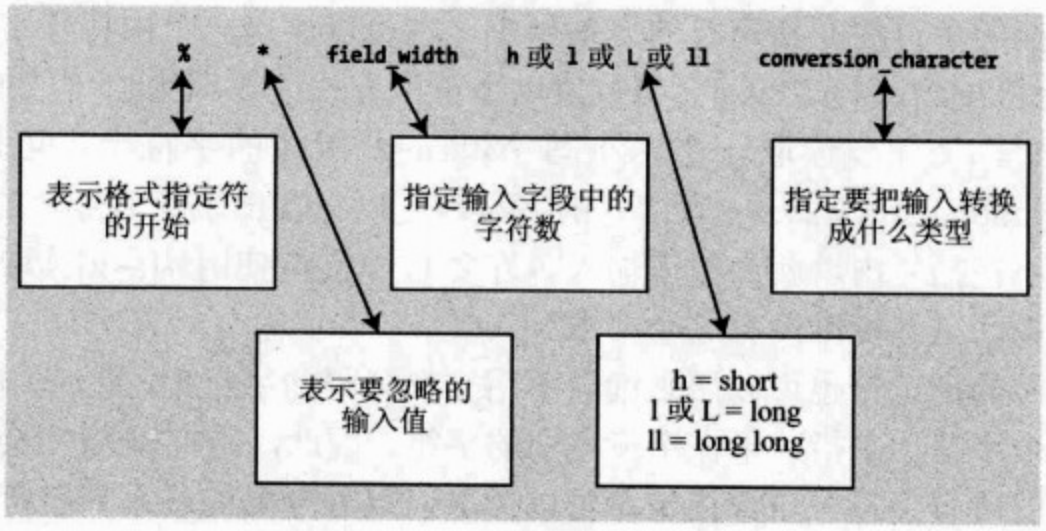


图 10-2 输出指定符的一般形式

以下是一般格式的各部分说明：

- %表示格式指定符的开头，不能省略。
- *是可选的。如果包含它，就表示忽略下一个输入值。这在键盘输入中不常用，但如果 stdin 重定向转到文件上，就可以使用它忽略文件中不想处理的值。
- 字段宽度也是可选的。它是一个整数，指定了 scanf()读入的字符数。它可以使输入的一连串数字中没有空白。它常用于读入文件。
- 下一个字符也是可选的，它可以是 h、L、l(L 的小写字母)或 ll(两个小写 L)。如果它是 h，就只能使用整数转换指定符(d、I、o、u 或 x)，表示输入要转换为 short 类型。如果它是 l，在 int 转换指定符之前表示 long，在 float 转换指定符之前表示 double。在 c 指定符的前面加上 l 表示宽字符转换，所以输入会读入为 wchar_t 类型。给 e、E、f、g 或 G 加上前缀 L，表示输入值的类型是 long double。给整数转换指定 ll 前缀，表示输入应存储为 long long 类型。
- 最后一个转换字符指定要将输入流转换为什么类型，因此它是不可缺少的。转换字符及其含义如表 10-2 所示。

表 10-2 转换字符及其含义

转 换 字 符	含　义
d	将输入转换为 int
i	将输入转换为 int。如果加了前缀 0，就输入八进制数。如果加了前缀 0x 或 0X，就输入十六进制数
o	将输入转换为 int，并假定所有的数字都是八进制数
u	将输入转换为 unsigned int
x	将输入转换为 int，并假定所有的数字都是十六进制数
c	将下一个字符读入为 char 类型(包含空白)。如果在读入单个字符时要忽略空白，就在格式指定符的前面加上一个空白字符
s	从下一个非空白字符开始，输入一串连续的非空白字符
e、f 或 g	将输入转换为 float 类型，输入中的小数点和指数是可选的
n	不读入任何输入，但前面读到此处为止的数字字符存储在对应的 int*类型参数中

www.TopSage.com

将所有可能的字符放在指定符的方括号中，也可以读入一个由特殊字符组成的字符串，例如，指定符%[0]23456789.-]会将数值以字符串方式读入。因此，假如输入是 - 1.25，就读成" - 1.25"。又如要读取一个由小写字母 a~z 组成的字符串，可以使用指定符%[abcdefghijklmnopqrstuvwxyz]。它读入的字符串可以包含出现在括号内的任何字符，如果输入的字符不在括号内，就会结束输入。许多 C 库支持使用%[a~z]读取由任意小写字母组成的字符串，虽然标准没有这个要求。

使用方括号的指定符也可以用来读取不用空白分隔的字符串，此时，使用^字符作为字符集的第一个字符，来指定不在字符串中的字符。因此，指定符%[^,]会包含在字符串中除逗号之外的所有字符，这个指定符可以读取以逗号分隔的一系列字符串。

表 10-3 列出了应用各个选项的例子。

表 10-3　转换指定符应用示例

指　定　符	说　　明
%lf	将下一个值读取为 double 类型
%*d	读入下一个整数值，但不存储它
%lc	将下一个字符读取为 wchar_t 类型
%\nc	将下一个字符读取为 char 类型，并忽略空白字符
%10lld	将接下来的 10 个字符读取为 long long 类型的整数值
%5d	将接下来的 5 个字符读取为一个整数
%Lf	将下一个值读取为 long double 类型的浮点数
%hu	将下一个值读取为 unsigned short 类型

下面用实际的例子练习这些格式控制字符串。

试试看：格式输入

读入各种数据并输出结果：

```
/* Program 10.1 Exercising formatted input */
#include <stdio.h>

const size_t SIZE = 20;      /* Max characters in a word */

int main(void)
{
  int value_count = 0;       /* Count of input values read */
  float fp1 = 0.0;           /* Floating-point value read */
  int i = 0;                 /* First integer read */
  int j = 0;                 /* Second integer read */
  char word1[SIZE] = " ";    /* First string read */
  char word2[SIZE] = " ";    /* Second string read */
  int byte_count = 0;        /* Count of input bytes read */
```

www.TopSage.com

```
  value_count = scanf("%f %d %d %[abcdefghijklmnopqrstuvwxyz] %*1d %s%n",
            &fp1, &i , &j, word1, word2, &byte_count);
  printf("\nCount of bytes read = %d\n", byte_count);
  printf("\nCount of values read = %d\n", k);
  printf("\nfp1 = %f i = %d j = %d", fp1, i, j);
  printf("\nword1 = %s word2 = %s\n", word1, word2);
  return 0;
}
```

程序输出的结果:

```
-2.35 15 25 ready2go

Count of bytes read = 20

Count of values read = 5

fp1 = -2.350000 i = 15 j = 25
word1 = ready word2 = go
```

代码的说明

前 3 个输入值以直接的方式读入。第 4 个输入值使用指定符%[abcdefghijklmnopqrstuvwxyz]读入，它会将一串小写字母读入为一个字符串。结果它读入了"ready"字符串，因为其后的字符是'2'，它不属于括号内的字符。输入中的'2'使用指定符"%*1d"读入，指定符中的*将只读取输入，但不存储它，且字符宽度是 1。接着使用"%s"指定符读取"go"，并存储到 word2 中。%n 指定符不从输入流中提取数据，只是将 scanf()当前从输入流中读取的字节数存储在 byte_count 中。

value_count 保存了 scanf()返回的数值个数，这个值反映了已存储的数值个数，但不包括"%*1d"指定符读取的值。

数据不一定在同一行上输入，也可以在输入两个值后按下回车键，scanf()函数会在下一行等待用户输入下一个值。

现在修改程序中的一行语句，用下面的语句替换输入语句:

```
value_count = scanf(
   "%4f %d %d %*d %[abcdefghijklmnopqrstuvwxyz] %*1d %[^o]%n",
   &fp1, &i , &j, word1, word2, &byte_count);
```

输入和以前一样的字符串，得到的输出如下:

```
-2.35 15 25 ready2go

Count of bytes read = 19

Count of values read = 5

fp1 = -2.300000  i = 5  j = 15
word1 = ready  word2 = g
```

www.TopSage.com

在这行语句中，给浮点数指定的字符宽度是 4，所以提取前 4 个字符，作为第一个输入变量的值，要输入的下一个整数值是 5，这是读入 -2.3 后的一个数字。存储在 j 中的整数是 15。接著指定符"%*d"读取值 25，并忽略它。最后读入的字符串是 g，因为指定符"%[^o]"要求字符串中不能有字符'o'。由于没有读取字母'o'，所以 byte_count 是 19。

再做一点变化，将语句改成:

```
value_count = scanf(
    "%4f %4d %d %*d %[abcdefghijklmnopqrstuvwxyz] %*1d %[^o]%n",
    &fp1, &i , &j, word1, word2, &byte_count);
```

输入相同，输出如下:

```
-2.35 15 25 ready2go

Count of bytes read = 19

Count of values read = 5

fp1 = -2.300000 i = 5 j = 15
word1 = ready word2 = g
```

从这个结果得到什么结论？第一个浮点数定义为输入的前 4 个字符，接着是两个整数 5 和 15。显然，给第二个整数指定的字段宽度 4 并未起作用，这是因为数字 5 的后面是空白字符，停止读取扫描正在读取的值。所以，无论给字符宽度设置什么值，只要碰到空格，就会马上停止扫描给定值的输入行。将整数值的指定符改成%12d，输出的结果就与给定的输入相同。

用稍微不同的输入运行上述例子的最后一个版本，进一步演示数值输入处理。

```
-2.3A 15 25 ready2go

Count of bytes read = 0

Count of values read = 1

fp1 = -2.300000  i = 0  j = 0
word1 =   word2 =
```

输入值的个数是 1，即只给变量 fpl 输入了值。读入的字节数是 0，这显然不正确，因为我们没有在 byte_count 中存储值。输入流中的 A 在数值输入时是无效的，整个处理因此而停止，变量 i、j、word1 及 word2 都没有值，byte_count 也没有存储值。这说明了 scanf()是多么无情，只因为输入流中有一个无效的字符，就禁止程序读取数据。如果希望在输入无效时仍能恢复输入，可以使用 scanf()的返回值，判断是否正确处理了所有的输入，并包含一些代码，在必要时重新读取数据。

最简单的方法是显示一条错误信息，然后要求重复全部输入。但在这种情况下，要避免代码中的错误使程序陷入无限循环。如果希望创建健壮的程序，就必须考虑到各种

www.TopSage.com

可能出错的情况。

还可以使用 wscanf()读取输入。唯一的区别是必须将格式字符串指定为宽字符串:

```
value_count = wscanf(L"%f %d %d %[abcdefghijklmnopqrstuvwxyz] %*1d %s%n",
        &fp1, &i , &j, word1, word2, &byte_count);
```

要编译上述语句，需要给<wchar.h>头文件添加#include 指令。第一个参数中的 L 前缀指定，字符串常量是一个宽字符串，所以它占用的空间是一般字符串的两倍。如果在程序中使用宽字符串，就要使用 wscanf()，还可以将读取的字符串存储为宽字符串。下面的代码说明了如何把字符串读取为宽字符串:

```
wchar_t wword1[SIZE] = L" ";
wchar_t wword2[SIZE] = L" ";
value_count = wscanf(L"%l[abcdefghijklmnopqrstuvwxyz] %*1d %ls%n",
     wword1, wword2, &byte_count);
printf("\nwword1 = %ls wword2 = %ls\n", wword1, wword2);
```

这段代码中的两个数组存储了 wchar_t 类型的字符串。在两个字符串的格式指定符中，类型指定符将 l(小写的 L)作为前缀，所以输入读取为宽字符串。如果输入 ready2go，ready 和 go 就在 wword1 和 wword2 中存储为宽字符串，它们之间的字符会被忽略。要输出字符串，printf()函数应将%ls 用作格式指定符，因为函数需要知道它们是宽字符串，如果使用%s，输出就不正确。读者可以自己试一试。

10.3.3 输入格式字符串中的字符

可以在输入格式字符串中包含一些不是格式转换指定符的字符。为此，必须指定输入中有这些字符，且 scanf()函数应读取它们，但不存储它们。但这些非格式转换字符必须和输入流的字符完全相同，只要有一个不同，scanf()就会终止输入。

试试看：输入格式字符串中的字符

下面的程序在输入格式字符串中包含一些字符:

```
/* Program 10.2 Characters in the format control string */
#include <stdio.h>

int main(void)
{
  int i = 0;
  int j = 0;
  int value_count = 0;
  float fp1 = 0.0;

  printf("Input:\n");
  value_count = scanf("fp1 = %f i = %d %d", &fp1, &i , &j);

  printf("\nOutput:\n");
```

www.TopSage.com

```
  printf("\nCount of values read = %d", value_count);
  printf("\nfp1 = %f\ti = %d\tj = %d\n", fp1, i, j);
  return 0;
}
```

程序输出如下:

```
Input:
fp1 = 3.14159 i = 7 8

Output:

Count of values read = 3
fp1 = 3.141590  i = 7  j = 8
```

代码的说明

输入时，空白是在等于号(=)的前面还是后面并不重要，因为它们是空白字符，会被忽略。重要的是，格式控制字符串中的字符的出现顺序和位置都必须和输入完全相同。试试另一个输入:

```
Input:
fp1 = 3.14159 i = 7 j = 8

Output:

Count of values read = 2
fp1 = 3.141590 i = 7 j = 0
```

这次只读入了两个数值。因为字符 j 使输入停止，变量 j 也没有存储值。scanf()函数在处理字符输入时是区分大小写的，如果输入的是 Fpl=而不是 fpl=，就不会读取任何值，因为大写 F 的不匹配，使整个输入终止。

10.3.4 输入浮点数的各种变化

使用 scanf()函数读取格式化的浮点数时，不仅可以选择格式指定符，而且可以输入不同形式的数。看看下面这个简单的例子。

试试看：输入浮点数

这个例子尝试了各种形式的指定符和各种不同的输入方式:

```
/* Program 10.3 Floating-Point Input */
#include <stdio.h>

int main(void)
{
  float fp1 = 0.0f;
  float fp2 = 0.0f;
  float fp3 = 0.0f;
```

www.TopSage.com

```
    int value_count = 0;

    printf("Input:\n");
    value_count = scanf("%f %f %f", &fp1, &fp2, &fp3);

    printf("\nOutput:\n");
    printf("Return value = %d", value_count);
    printf("\nfp1 = %f fp2 = %f fp3 = %f\n", fp1, fp2, fp3);
    return 0;
}
```

下面是同一个输入值使用 3 种不同方法的输出结果:

```
Input:
3.14.314E1.0314e+02

Output:
Return value = 3
fp1 = 3.140000  fp2 = 3.140000  fp3 = 3.140000
```

代码的说明

这个例子示范了同一个值的 3 种不同输入方法。第一个方法是输入一般的小数值。第二个是输入指数值，E1 表示.314 要乘以 10。第三个也是输入指数值，其中 e+02 表示.0314 要乘以 100。因此，使用指定符"%f"读取浮点数时，可以选择是否包含指数。如果包含指数，就可以把指数定义为以大写 E 或小写 e 开头，当然也可以给指数加上+或-符号。

将 scan()语句改成:

```
value_count = scanf("%e %g %f", &fp1, &fp2, &fp3);
```

下面是该语句的输出:

```
Input:
3.14.314E1.0314e+02

Output:
Return value = 3
fp1 = 3.140000  fp2 = 3.140000  fp3 = 3.140000
```

显然，这 3 个格式指定符可以用于各种输入形式，唯一的差异是将它们用于 printf()函数的时间。建议尝试各种不同的输入，特别是尝试浮点数和用于读取整数的字段宽度指定符。

10.3.5 读取十六进制和八进制值

前面曾经提过，可以使用格式指定符%x 从输入流中读取十六进制值，使用格式指定符%o 读取八进制值。这很简单，下面是一个实际的例子。

www.TopSage.com

试试看：读取十六进制和八进制值

练习下面的例子:

```
/* Program 10.4 Reading hexadecimal and octal values */
#include <stdio.h>

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

  printf("Input:\n");
  n = scanf(" %d %x %o", &i , &j, &k );

  printf("\nOutput:\n");
  printf("%d values read.", n);
  printf("\ni = %d j = %d k = %d\n", i, j, k );
  return 0;
}
```

以下是一些输出:

```
Input:
12 12 12

Output:
3 values read.
i = 12  j = 18  k = 10
```

代码的说明

这个例子读入 3 个 12。第一个 12 用十进制格式指定符%d 读入，第二个 12 用十六进制格式指定符%x 读入，第三个 12 用八进制格式指定符%o 读入。十六进制的 12 输出为十进制的 18，八进制的 12 输出为十进制的 10。

十六进制数据项常用于输入位模式(一连串的 1 和 0)，因为这些 1 和 0 更容易用十六进制指定，而不是十进制。每个十六进制数对应 4 位，所以 16 位对应 4 个十六进制数。八进制很少使用，这里介绍它主要是因为以前曾使用它。

注意下面的输出:

```
Input:
18 18 18

Output:
3 values read.
i = 18  j = 24   k = 1
```

前两个值都正确读入为 18，因为十六进制的 18 表示十进制的 24。然而，第三个值

www.TopSage.com

读成 1，这是因为 8 不是合法的八进制数。八进制数字是 0~7。

在十六进制中，10 以上的数字用 A~F 或 a~f 表示，大小写可以混用。下面是另一个输出：

```
Input:
12 aA 17

Output:
3 values read.
i = 12  j = 170   k = 15
```

十六进制的 aA 表示为十进制是 10 × 16+10，结果是 170。八进制的 17 表示为十进制是 1 × 8+7，结果是 15。

在 scanf()中使用"%X"和"%x"没有区别，但在 printf()中它们是不同的。现在将最后一个 printf()改成：

```
printf("\ni = %x j = %X k = %d\n", i, j, k );
```

上述语句以十六进制输出前两个值。对于下面的输入，输出如下：

```
Input:
26 AE 77

Output:
3 values read.
i = 1a   j = AE   k = 63
```

%x 使用十六进制数 a~f 输出结果，而"%X"使用十六进制数 A~F 输出结果。

10.3.6 用 scanf()读取字符

第一个例子尝试过读入字符串，读入字符串还有其他方法。有 3 个格式指定符用于读取一个或多个单字节字符。使用格式指定符%c 可以读取一个字符，并将它存储为 char 类型，如果使用的是%lc，就存储为 wchar_t 类型。对于字符串，可以使用指定符%s 或%[]，如果要将输入存储为宽字符，就使用%ls 或%l[]，其中转换指定符的前缀是 L 的小写。此时要给存储的字符串追加终止字符'\0'，作为最后一个字符。使用格式符%[]或%l[]读入的字符串必须只包含方括号内的字符，如果方括号中的第一个字符是^时，则读入的字符串不能包含方括号内^字符后面的任何字符，例如%[aeiou]读入的字符串只能包含元音。碰到不是元音的字符就停止输入。而%[^aeiou]读入的字符串不能包含元音。碰到元音就停止输入。

要注意的是%[]指定符可以读入含有空格的字符串，但指定符%s 不能。使用%[]指定符时，只需在方括号中包含空格字符。

试试看：读入字符和字符串

下面的程序可以读入字符：

www.TopSage.com

```
/* Program 10.5 Reading characters with scanf() */
#include <stdio.h>

int main(void)
{
  char initial = ' ';
  char name[80] = { 0 };
  char age[4] = { 0 };
  printf("Enter your first initial: ");
  scanf("%c", &initial );
  printf("Enter your first name: " );
  scanf("%s", name );

  if(initial != name[0])
    printf("\n%s,you got your initial wrong.", name);
  else
    printf("\nHi, %s. Your initial is correct. Well done!", name );
  printf("\nEnter your full name and your age separated by a comma:\n" );
  scanf("%[^,] , %[0123456789]", name, age );
  printf("\nYour name is %s and you are %s years old\n", name, age );
  return 0;
}
```

程序的输出如下:

```
Enter your first initial: I
Enter your first name: Ivor

Hi, Ivor. Your initial is correct. Well done!
Enter your full name and your age separated by a comma:
Ivor Horton , 99

Your name is
Ivor Horton and you are 99 years old
```

代码的说明

首先程序要求输入姓名的第一个字母和名字，然后检查名字的第一个字母是否与输入的第一个字母相同，这很简单，如输出所示。

接着要求输入全名和年龄，中间用逗号分隔。读入操作由下面的语句完成:

```
scanf("%[^,] , %[0123456789]", name, age );
```

这里在输入数据中加入了空格，所以很容易看出，第一个输入指定符%[^,]读入除逗号外的任何字符，包含空格。在最后一行输出中，姓名的后面有多余的空格。接着，控制字符串中的逗号让 scanf()从输入中读入逗号(或连续几个逗号)，但不会存储它。年龄用指定符%[0123456789]读入为一个字符串，这会把连续几个数字读入为字符串。

注意，输入字符串中的逗号对正确的读取输入是很重要的。如果省略了这个逗号，scanf()会将逗号读取为年龄的一部分，但因为逗号不是数字，所以会终止年龄的输入，

www.TopSage.com

让年龄变成一个空字符串。

如果先输入一个空格，再输入姓名的第一个字母，程序会将空白当成 initial 的值，输入的单个字符当成 name。根据控制字符串的定义方式，使用指定符%c 输入的第一个字符就是要提取的字符。如果不接受空格作为姓名的第一个字符，可以修改输入语句：

```
scanf(" %c",  &initial );
```

控制字符串中的第一个字符是空格，因此 scanf()会读入且忽略所有空格，将不是空白的第一个字符读入为 initial。

10.3.7 scanf()的陷阱

使用 scanf()常犯的两个错误如下：

- 变元必须是指针，最常犯的错误是将变量指定为 scanf()的变元时，忘记在变量名的前面加上&符号，不过使用 printf()时不需要这个&字符。此外，如果变元是数组名或指针变量，也不需要&符号。
- 在读字符串时，要确保有足够的空间存放读入的字符串，这个字符串需包含终止字符'\0'，否则，会覆盖内存中的数据，甚至是程序代码。

10.3.8 从键盘上输入字符串

<stdio.h>头文件中的 gets()函数可以将一整行的文本作为字符串读入。它的函数原型如下：

```
char  *gets(char  *str);
```

这个函数会将连续的字符读入指针 str 所指的内存中，直到按下回车键为止。它会用终止字符'\0'取代按下回车键时读入的换行符。其返回值与变元相同，即存储字符串的地址。下面是一个演示 gets()函数的例子：

试试看：使用 gets()读入字符串

下面是使用 gets()函数的例子：

```
/* Program 10.6 Reading a string with gets() */
#include <stdio.h>

int main(void)
{
  char initial[2] = {0};
  char name[80] = {0};

  printf("Enter your first initial: ");
  gets(initial);
  printf("Enter your name: " );
  gets(name);
```

www.TopSage.com

```
  if(initial[0] != name[0])
    printf("\n%s,you got your initial wrong.\n", name);
  else
    printf("\nHi, %s. Your initial is correct. Well done!\n", name);
  return 0;
}
```

程序的输出如下:

```
Enter your first initial:  M
Enter your name:  Mephistopheles

Hi,  Mephistopheles. Your initial is correct. Well done!
```

代码的说明

这个范例使用 gets()读入姓名的第一个字符和名字。这个函数使用起来很简单，而且不涉及格式指定符。它一直读入字符，直到按下回车键为止，因此也可以输入全名。

当然，gets()的缺点是无法控制存储的字符数，必须创建一个数组来接收数据，该数组必须有足够的空间来存储可能输入的最大长度的字符串。当需要确保数组的长度足够大时，可以使用 fgets()函数，用下面的语句管理输入:

```
printf("Enter your first initial: ");
fgets(initial, sizeof(initial), stdin); /* Read 1 character max */
fflush(stdin); /* Flush the newline */

printf("Enter your name: " );
fgets(name, sizeof(name), stdin); /* Read max name-1 characters */
size_t length = strlen(name);
name[length-1] = name[length]; /* Overwrite the newline */
```

fgets()函数读取的字符数比第二个变元指定的字符数少 1，因为还需要添加终止字符'\0'。这样，就不会超过作为第一个变元传送的数组长度。注意，fgets()函数在输入字符串中存储一个换行符来对应按下的回车键，而 gets()函数不是这样。在第一个读取操作中，输入缓存区中会剩下换行符，所以调用 fflush()刷新 stdin，并删除它。否则，换行符会成为 name 的输入。name 中空字符前的最后一个字符是换行符，所以将终止字符复制到这个位置，以覆盖它。

对于字符串输入，使用 gets()或 fgets()通常是首选方式，除非要控制字符串的内容，此时可以使用%[]。当支持非标准的%[a-z]格式时，使用%[]指定符比较方便。但注意这是非标准的，所以代码不像使用%[abcdefghijklmnopqrstuvwxyz]标准形式读取小写字符串那样是可移植的。

10.3.9 键盘的非格式化输入

getchar()函数可以从 stdin 中一次读一个字符，它在<stdio.h>中定义，语法如下:

```
int  getchar(void);
```

www.TopSage.com

getchar()函数不需要变元，它会返回从输入流中读入的字符。注意，这个字符返回为 int 类型，并显示在屏幕上。

在 C 语言的许多版本中，通常都包含了头文件<conio.h>，它提供了额外的字符输入输出函数，其中的 getch()函数最有用，这个函数可以从键盘上读入一个字符，但不会显示在屏幕上。不希望他人看到键盘输入时，就可以使用它，例如输入密码。

标准头文件<stdio.h>也声明了 ungetc()函数，它允许把刚才读取的一个字符放回输入流。这个函数需要两个参数，第一个是要放回输入流的字符，第二个是流的标识符，对于标准输入流，它就是 stdin。ungetc()返回一个 int 类型的值，对应放回输入流的字符，如果操作失败，就返回一个特殊的字符 EOF (end-of-file，文件结束)。

原则上，可以把一连串字符放回输入流，但只能保证一个字符有效。前面说过，未能将一个字符放回输入流，函数就会返回 EOF，所以如果要将几个字符返回输入流，就可以检查这个 EOF。

逐个字符地读取输入，但不知道数据由多少个字符组成时，可以使用 ungetc()函数。例如，读入一个整数值，但不知道这个整数有多少个数字。此时可以使用 getchar()函数读取一连串字符，当读入的字符不是数字时，就使用 ungetc()函数把它返回给输入流。使用 getchar()和 ungetc()函数可以忽略标准输入流中的空格和制表符：

```
void eatspaces(void)
{
  char ch = 0;
  while(isspace(ch = getchar()));  /* Read as long as there are spaces */
  ungetc(ch, stdin);               /* Put back the nonspace character */
}
```

变元是一个空格字符时，在<ctype.h>头文件中声明的 isspace()函数就返回 true。只要读取的字符是空格或制表符，while 循环就会继续，并把每个字符存储到 ch 中。读入一个非空格字符时，循环就结束，并将该字符放在 ch 中。调用 ungetc()函数会把非空格字符返回到输入流中，以便将来处理。

下面在一个示例中使用 getchar()和 ungetc()函数。

试试看：读取和不读取字符

这个例子假定键盘输入包含随机顺序的整数和名字：

```
/* Program 10.7 Reading and unreading characters */
#include <stdio.h>
#include <ctype.h>
#include <stdbool.h>
#include <string.h>

const size_t LENGTH = 50;

/* Function prototypes */
void eatspaces(void);
bool getinteger(int *n);
```

www.TopSage.com

```
char *getname(char *name, size_t length);
bool isnewline(void);

int main(void)
{
  int number;
  char name[LENGTH];
  printf("Enter a sequence of integers and alphabetic names:\n");
  while(!isnewline())
    if(getinteger(&number))
      printf("\nInteger value:%8d", number);
    else if(strlen(getname(name, LENGTH)) > 0)
      printf("\nName: %s", name);
    else
    {
      printf("\nInvalid input.");
      return 1;
    }
  return 0;
}

/* Function to check for newline */
bool isnewline(void)
{
  char ch = 0;
  if((ch = getchar()) == '\n')
    return true;

  ungetc(ch, stdin);
  return false;
}

/* Function to ignore spaces from standard input */
void eatspaces(void)
{
  char ch = 0;
  while(isspace(ch = getchar()));
  ungetc(ch, stdin);
}

/* Function to read an integer from standard input */
bool getinteger(int *n)
{
  eatspaces();
  int value = 0;
  int sign = 1;
  char ch = 0;

  /* Check first character */
```

www.TopSage.com

```
  if((ch=getchar()) == '-') /* should be minus */
    sign = -1;
  else if(isdigit(ch))      /* ...or a digit */
    value = 10*value + (ch - '0');
  else if(ch != '+')        /* ...or plus */
  {
    ungetc(ch, stdin);
    return false;           /* Not an integer */
  }

  /* Find more digits */
  while(isdigit(ch = getchar()))
    value = 10*value + (ch - '0');

  /* Push back first nondigit character */
  ungetc(ch,stdin);
  *n = value*sign;
  return true;
}

/* Function to read an alphabetic name from input */
char *getname(char *name, size_t length)
{
  eatspaces();                  /* Remove leading spaces */
  size_t count = 0;
  char ch = 0;
  while(isalpha(ch=getchar())) /* As long as there are letters */
  {
    name[count++] = ch;         /* store them in name */
    if(count == length-1)
      break;
  }

  name[count] = '\0';           /* Append string terminator */
  if(count < length-1)
    ungetc(ch, stdin);          /* Return nonletter to stream */
  return name;
}
```

下面是该程序的一个输出:

```
Enter a sequence of integers and alphabetic names:
12              Jack Jim 234 Jo Janet 99 88

Integer value:    12
Name: Jack
Name: Jim
Integer value:    234
Name: Jo
Name: Janet
```

www.TopSage.com

```
Integer value:     99
Integer value:     88
```

下面是该程序的另一个输出：

```
Enter a sequence of integers and alphabetic names:
Jim     Jo Will Bert

Name: Jim
Name: Jo
Name: Will
Name: Bert
```

代码的说明

有 4 个函数使用了 getchar()和 ungetc()函数从 stdin 中读取输入。上一节介绍了 eatspaces()函数。isnewline()函数只是从键盘上读取一个字符，如果该字符是换行符，就返回 true。它用于控制 main()函数中输入何时停止。

getinteger()函数从键盘上读取一个任意长度的整数，该整数的前面可以有符号。第一步是调用 eatspaces()函数删除前导空格。检查第一个字符是符号还是数字后，这个函数在循环中继续读取数字：

```
while(isdigit(ch = getchar()))
  value = 10*value + (ch - '0');
```

数字是从左到右地读取，所以最后读取的数字是该整数值中的低位数字。从当前数字的编码值中减去 0 的编码值，就得到了当前数字的值。这是因为数字的编码值是递增的。要插入数字，应给当前累加的值乘以 10，再加上新的数值。当然，必须将结果存储为 int 类型。可以编写一个函数，将值存储为 long long 类型，以存储更大范围的值。还可以包含代码，检查得到的数字有多大，如果它不能存储在 int 类型中，就输出一条错误信息。

如果读取的字符不是数字，就结束循环，这个字符会返回到流中，以便再次读取。

getname()函数从键盘上读取由字母组成的名字。其参数是存储名字的数组和该数组的长度，这样函数才能确保没有超出数组的容量。该函数将字符串的第一个字节的地址返回给调用程序。在原则上，该过程与 getinteger()函数相同。只要读取的字符是字母，这个函数就会在循环中继续读取字符。

```
while(isalpha(ch=getchar()))   /* As long as there are letters */
{
  name[count++] = ch;          /* store them in name */
  if(count == length-1)
  break;
}
```

count 变量跟踪存储在 name 数组中的字符数，当只剩下一个空闲元素时，循环结束。之后，代码将终止字符'\0'添加到字符串中。当然，当 count 的值达到 length 1 时，读取的最后一个字符必须是字母，并存储在数组中。因此，当最后一个字符不是字母时，就

www.TopSage.com

调用 ungetc()把这个字符返回到输入流中。

main()函数在循环中读取随机顺序的名字和整数:

```
while(!isnewline())
  if(getinteger(&number))
    printf("\nInteger value:%8d", number);
  else if(strlen(getname(name, LENGTH)) > 0)
    printf("\nName: %s", name);
  else
  {
    printf("\nInvalid input.");
    return 1;
  }
```

只要当前字符不是表示当前行结束的换行符，循环就继续。程序希望在每次迭代时读取整数或名字。循环首先调用 getinteger()函数尝试读取整数。如果没有找到整数，这个函数就返回 false，此时调用 getname()函数读取名字。如果没有找到名字，说明输入既不是名字也不是整数，程序就在输出一条信息后结束。

10.4 屏幕输出

将数据输出到屏幕的命令行上要比从键盘上读取数据容易多了，因为我们知道要输出什么数据，而输入时可能输入错误的数据。将格式化数据输出到 stdout 流的主要函数是 printf()。printf()函数可以提供许多不同的格式输出，其格式指定符远多于 scanf()。

10.4.1 使用 printf()格式输出到屏幕

printf()函数在头文件<stdio.h>中定义，它的一般形式如下：

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

第一个参数是格式控制字符串。通常这个参数传递给函数的变元是一个明确的字符串常量，如前面的例子所示，但也可以是一个指针，指向在其他地方指定的字符串。该函数的可选变元是要输出的值，它们的数目以及类型必须与第一个字符串参数后面的格式转换指定符相符。当然，如果输出只是控制字符串中的文本时，除了第一个变元外，就不需要其他的变元了。但如果要输出多个变元值，则变元的个数必须对应格式指定符的个数，否则会产生不可预知的后果。如果变元多于格式指定符，就会忽略多余的变元。因为该函数使用格式字符串确定要输出的变元个数和类型。

注意：

事实上，格式字符串只用于确定如何解释数据，所以用指定符%d 输出一个 long long 变元时，会得到错误的结果。

www.TopSage.com

头文件<stdio.h>还声明了 wprintf()函数。与 scanf()相同，wprintf()函数的工作方式与 printf()完全相同，只是它的第一个参数是宽字符串。两个函数的格式指定符完全相同。

printf()和 wprintf()的格式转换指定符比 scanf()和 wscanf()复杂得多。输出格式指定符的一般形式如图 10-3 所示。

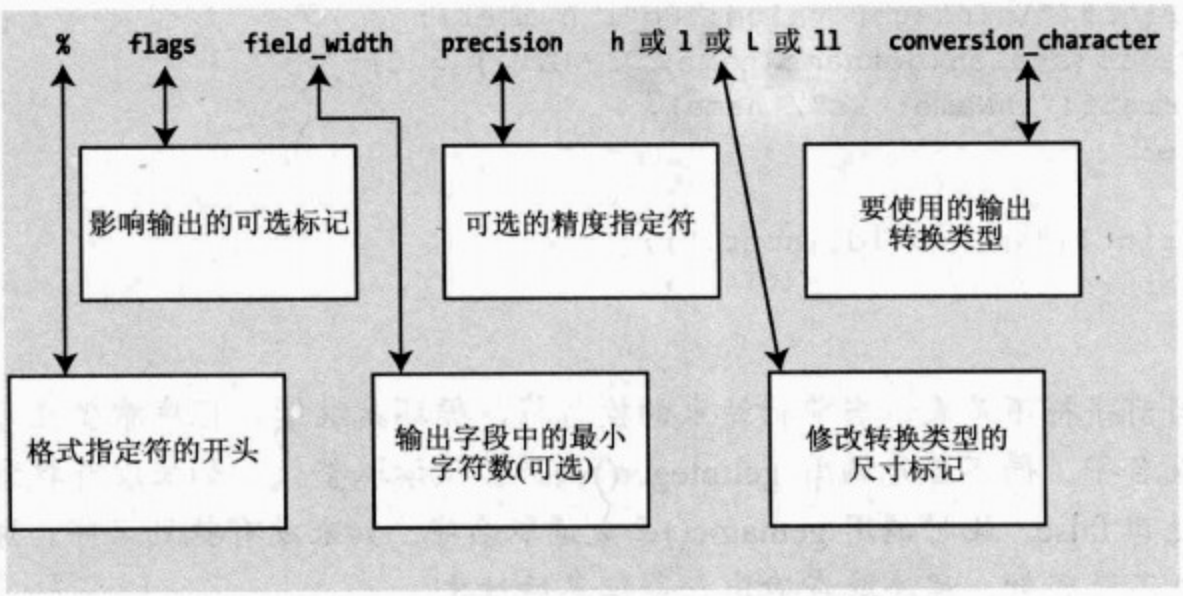


图 10-3　printf()和 wprintf()的格式指定符

前面已经介绍了其中的大多数细节，下面快速浏览一下这个一般格式指定符的元素：

- %符号表示用于输出的指定符由此开始。
- 可选择的标志字符有+、－、#和空格。这些会影响输出，如表 10-4 所示。

表 10-4　输出指定符中可选标志字符的作用

字　符	用　途
+	对于有符号的输出值，这个字符确保输出值的前面总是有一个符号+或－。默认情况下，只有负值有－号
-	指定输出值在输出字段中左对齐，右边用空格填充。输出的默认对齐方式是右对齐
0	前置一个 field_width 值，指定在输出值的前面填充 0，以填满字段宽度
#	指定将 0 放在八进制的输出值前面，将 0x 或 0X 放在十六进制的输出值前面，或者浮点数包含小数点。对于 g 或 G 浮点转换字符，则忽略尾部的 0
空格	指定在正数或 0 输出值前面放置一个空格，而不是+号

- 可选的 field_width 指定输出值的最少字符数。如果输出值需要更多的字符，它会自动增加。如果输出值需要的字符数少于指定的最少字符数，多余的位置会填充空白，除非字段宽度用前导 0 指定，如 09，此时在左边会补入 0。
- 精度指定符也是可选的。它通常用于浮点数的输出，指定符.n 表示输出值精确到小数点后 n 位。如果输出的小数位数多于 n，就四舍五入或舍掉。
- 在对应的类型转换符前面加上 h、l(L 的小写)、ll 或 L 修饰符，指定分别将输出转换为 short、long、long long 和 long double。l 修饰符还应用于 c 类型指定符，表示字符应存储为 wchar_t 类型。

www.TopSage.com

- 转换字符用来定义如何将输出值转换为指定的类型。转换字符如表 10-5 所示。

表 10-5 输出指定符中的转换字符

转 换 字 符	生成的输出
可应用于整数	
d	带符号的十进制整数值
o	不带符号的八进制整数值
u	不带符号的十进制整数值
x	不带符号的十六进制整数值，使用小写的十六进制数 a、b、c、d、e、f
X	与 x 相同，但使用大写的十六进制数 A、B、C、D、E、F
应用于浮点数	
F	带符号的小数值
e	带符号和指数的小数值
E	与 e 相同，但用 E 表示指数，而不是 e
g	与 e 或 f 相同，取决于值的大小和精度
G	与 g 相同，但用 E 表示指数
应用于字符	
c	单个字符
s	在'\0'之前的所有字符或已输出的 precision 个字符

这里只列出了最重要的输出选项。更多的输出选项可参阅编译器的文档说明。

10.4.2 转义序列

在 printf()和 wprintf()函数的格式控制字符串中可以包含空白字符。空白字符有换行符、回车符、换页符、空格和制表符。它们用以\开头的转义序列表示。表 10-6 列出了一些最常见的转义序列。

表 10-6 常见的转义序列

转 义 序 列	说　　明
\a	鸣响(计算机上的一次鸣响，目前不常用)
\b	退格
\f	换页
\n	换行
\r	回车(用于打印机)，在屏幕输出中，就是移动到当前行的开头
\t	水平制表符

www.TopSage.com

输出反斜杠\时，必须在格式控制字符串中使用\\。否则，就不会输出反斜杠，因为反斜杠会被误认为转义序列的开头。要将%写入 stdout，应使用%%，不能使用%，因为它会被看做格式指定符的开头。

注意：
当然，可以在任何字符串中使用转义序列，而不仅仅只在 printf()函数的格式字符串中。

10.4.3 整数输出

下面介绍一些前面未提及的变化，其中字符宽度和精度指定符最有趣。

试试看：输出整数
首先是一个整数输出格式的例子：

```
/* Program 10.8 Integer output variations */
#include <stdio.h>

int main(void)
{
  int i = 15;
  int j = 345;
  int k = 4567;
  long li = 56789L;
  long lj = 678912L;
  long lk = 23456789L;

  printf("\ni = %d j = %d k = %d i = %6.3d j = %6.3d k = %6.3d\n",
         i ,j, k, i, j, k);
  printf("\ni = %-d j = %+d k = %-d i = %-6.3d j = %-6.3d k ="
          " %-6.3d\n",i ,j, k, i, j, k);
  printf("\nli = %d lj = %d lk = %d\n", li, lj, lk);
  printf("\nli = %ld lj = %ld lk = %ld\n", li, lj, lk);
  return 0;
}
```

程序输出如下：

```
i = 15  j = 345  k =  4567  i =  015  j =  345  k =  4567

i = 15  j = +345 k = 4567   i = 015   j = 345   k = 4567

li = -8747  lj = 23552  lk = -5099

li = 56789  lj = 678912  lk = 23456789
```

代码的说明
这个例子演示了整数输出的许多选项。比较这些语句输出的前两行，可以看出 - 标

www.TopSage.com

志的作用:

```
printf("\ni = %d j = %d k = %d i = %6.3d j = %6.3d k = %6.3d\n",
      i ,j, k, i, j, k);
printf("\ni = %-d j = %+d k = %-d i = %-6.3d j = %-6.3d k ="
     " %-6.3d\n",i ,j, k, i, j, k);
```

标志 - 使输出左对齐。字符宽度的作用是在这 6 个输出的后 3 个输出中有明显的空白，注意预设的字符宽度只是提供了足够的空间来放置要输出的数字，所以第二行的 - 标志没有起作用。

第二行输出 j 时利用标志修饰符添加了一个前导+号——可以添加任意多个标志修饰符。对于第二个 i 值的输出，插入了一个前导 0，因为最小精度指定为 3。如果在格式指定符的最小字符宽度前放一个 0，也可以得到相同的结果。

第三行输出由如下语句生成:

```
printf("\nli = %d lj = %d lk = %d\n", li, lj, lk);
```

输出 long 类型的整数时插入 l(L 的小写)修饰符，会得到垃圾值，因为输出值假定为 2 个字节的整数。当然，如果系统将 int 类型表示为 4 字节的整数，结果就是正确的。只有 long 和 int 是不同的，才会出问题。如果不匹配，编译器就会发出警告。输出 long long 类型的值时，如果使用了错误的类型转换，也会出现这类问题。

下面的语句会得到正确的结果:

```
printf("\nli = %ld lj = %ld lk = %ld\n", li, lj, lk);
```

给输出值指定的字符宽度及精度不够大，会产生意想不到的结果。用下面的例子来测试一下有什么样的结果。

试试看：单一整数的变化

用一个整数例子来测试所有可能的输出:

```
/* Program 10.9 Variations on a single integer */
#include <stdio.h>

int main(void)
{
  int k = 678;

  printf("%%d %%o %%x %%X"); /* Display format as heading */
  printf("\n%d %o %x %X", k, k, k, k ); /* Display values */

  /* Display format as heading then display the values */
  printf("\n\n%%8d %%-8d %%+8d %%08d %%-+8d");
  printf("\n%8d %-8d %-+8d %08d %-+8d\n", k, k, k, k, k );
  return 0;
}
```

www.TopSage.com

代码的说明

这个程序乍看之下会令人困惑。因为每一对 printf()语句中的第一个 printf()语句显示了用于输出数字的格式。%%指定符仅输出%字符。执行这个例子，会得到如下结果:

```
%d   %o    %x   %X
678  1246  2a6  2A6

%8d   %-8d   %+8d   %08d       %-+8d
678   678    +678   00000678   +678
```

第一行输出值由下面的语句产生:

```
printf("\n%d %o %x %X", k, k, k, k ); /* Display values */
```

它使用预设的宽度输出 678 的十进制、八进制和两种十六进制。每个值对应的格式显示在其上方。

下一行输出由下面的语句产生:

```
printf("\n%8d %-8d %-+8d %08d %-+8d\n", k, k, k, k, k );
```

这行语句包含了标志设置的各种变化，其字符宽度指定为 8。第一个是数值在字段内默认右对齐。第二个使用了 - 标志，所以是左对齐；第三个使用了+标志，所以输出+符号。第四个的字符宽度指定为 08，所以输出值有前导 0，它还使用了+标志，所以输出+符号。最后一个输出值在指定符中使用了所有的标志%-+8d，所以输出的结果左对齐，并且带符号。

提示:

要在屏幕上输出多行值，可以使用字段宽度指定符——和制表符—— 使数值排列整齐。

10.4.4 输出浮点数

前面介绍了的输出整数的选项，接下来看看输出浮点数的选项。

试试看：输出浮点数

请看下面的例子:

```
/* Program 10.10 Outputting floating-point values */
#include <stdio.h>

int main(void)
{
  float fp1 = 345.678f;
  float fp2 = 1.234E6f;
  double fp3 = 234567898.0;
  double fp4 = 11.22334455e-6;
```

www.TopSage.com

```
    printf("\n%f %+f %-10.4f %6.4f\n", fp1, fp2, fp1, fp2);
    printf("\n%e %+E\n", fp1, fp2);
    printf("\n%f %g %#+f %8.4f %10.4g\n", fp3,fp3, fp3, fp3, fp4);
    return 0;
}
```

代码的说明

在一个编译器中，得到如下输出：

```
345.678009      +1234000.000000     345.6780      1234000.0000

3.456780e+002    +1.234000E+006

234567898.000000   2.34568e+008    +234567898.000000   234567898.0000

1.122e-005
```

读者可能不会得到完全相同的输出，不过应该很接近。大部分输出都演示了前面讨论的格式转换指定符的作用，但要注意以下几点：

fp1 的第一个输出和赋予变量的值有点不同。这是浮点值从十进制数转换成二进制数时常见的微小差异。小数的十进制值通常不会刚好等于其二进制浮点值。

第一行输出语句如下：

```
printf("\n%f %+f %-10.4f %6.4f\n", fp1, fp2, fp1, fp2);
```

fp1 的第二个输出值说明如何限制小数点后的位数。这里的输出在字段宽度内左对齐。为 fp2 的第二个输出指定的字符宽度太小，放不下小数位数，因此会舍弃多余的部分。

第二个 printf()语句如下：

```
printf("\n%e %+E\n", fp1, fp2);
```

它以指数格式输出相同的浮点值。指数指示器使用大写 E 还是小写 e，取决于指定格式符中是用的是大写 E 还是小写 e。

最后一行，用 g 指定 fp3 四舍五入后的输出。

注意：

使用 prinf()会得到很多可能的输出。读者可以给相同的数据尝试各种选项。

10.4.5 字符输出

学习了输出数值的各种选项后，接下来看看字符的输出。printf()和 wprintf()函数可以使用 4 个输出指定符输出字符数据：单个字符和字符串使用%c 和%s，单个宽字符和宽字符串使用%lc 和%ls。前面介绍过%lc 和%ls，下面用一个例子说明单个字符的输出。

www.TopSage.com

试试看：输出字符数据

这个例子输出所有可打印的字符，然后是所有大小写字母。

```
/* Program 10.11 Outputting character data */
#include <stdio.h>
#include <limits.h>
#include <wchar.h>
#include <ctype.h>
#include <wctype.h>

int main(void)
{
  int count = 0;
  char ch = 0;
  printf("\nThe printable characters are the following:\n");

  /* Iterate over all values of type char */
  for(int code = 0 ; code <= CHAR_MAX ; code++)
  {
    ch = (char)code;
    if(isprint(ch))
    {
      if(count++ % 32 == 0)
        printf("\n");
      printf("%c", ch);
    }
  }

  /* Use wprintf() to output wide characters */
  count = 0;
  wchar_t wch = 0;
  wprintf(L"\n\nThe alphabetic characters
         and their codes are the following:\n");

  /* Iterate over the lowercase wide character letters */
  for(wchar_t wch = L'a' ; wch <= L'z' ; wch++)
  {
    if(count++ % 3 == 0)
      wprintf(L"\n");

    wprintf(L" %lc %#x %lc %#x", wch, (long)wch, towupper(wch),
          (long)towupper(wch));
  }
  return 0;
}
```

这个程序的输出如下：

```
The printable characters are the following:
```

www.TopSage.com

```
!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~

The alphabetic characters and their codes are the following:
a  0x61   A   0x41   b   0x62   B   0x42 c   0x63  C  0x43
d  0x64   D   0x44   e   0x65   E   0x45 f   0x66  F  0x46
g  0x67   G   0x47   h   0x68   H   0x48 i   0x69  I  0x49
j  0x6a   J   0x4a   k   0x6b   K   0x4b l   0x6c  L  0x4c
m  0x6d   M   0x4d   n   0x6e   N   0x4e o   0x6f  O  0x4f
p  0x70   P   0x50   q   0x71   Q   0x51 r   0x72  R  0x52
s  0x73   S   0x53   t   0x74   T   0x54 u   0x75  U  0x55
v  0x76   V   0x56   w   0x77   W   0x57 x   0x78  X  0x58
y  0x79   Y   0x59   z   0x7a   Z   0x5a
```

代码的说明

输出的第一个可打印字符块在 for 循环中生成:

```
for(int code = 0 ; code <= CHAR_MAX ; code++)
{
  ch = (char)code;
  if(isprint(ch))
  {
    if(count++ % 32 == 0)
      printf("\n");
    printf("%c", ch);
  }
}
```

首先，注意循环控制变量 code 的类型。读者可能在这里尝试使用 char 类型，但这是一个严重的错误，因为循环会无限继续下去。其原因是在递增了 code 的值后，要检查结束循环的条件。在最后一次正确的迭代中，code 的值是 CHAR_MAX，即 char 类型能存储的最大值。如果 code 是 char 类型，给 CHAR_MAX 加 1，会得到 0，因此循环会继续，而不是结束。

在循环中将 code 的值转换为 char 类型，将结果存储在 ch 中。这里不使用显式类型转换，代码也会编译，因为编译器会插入这个类型转换，这里使用显式转换是故意的。接着使用在<ctype.h>头文件中声明的 isprint()函数，测试可打印字符。如果 isprint()返回 true，就使用%c 格式指定符输出该字符。每次输出 32 个字符后，就输出一个换行符，这样输出就不会在换到新行上时丢失字符了。

第二个循环使用 wprintf()函数输出宽字母字符及其编码值:

```
for(wchar_t wch = L'a' ; wch <= L'z' ; wch++)
{
  if(count++ % 3 == 0)
```

www.TopSage.com

```
        wprintf(L"\n");

    wprintf(L" %lc %#x %lc %#x", wch, (long)wch, towupper(wch),
        (long)towupper(wch));
}
```

这次可以使用 wchar_t 类型的循环控制变量 wch，迭代从 L'a'到 L'z'的编码值。这个循环使用的技巧与上一个循环一样，在每一行上输出 3 个连续的字符组。wprintf()使用%lc 指定符输出字符，使用%ld 输出编码值。要输出编码值，应将 wch 的值转换为 long 类型，使用%ld 格式指定符显示它。使用在<wctype.h>头文件中声明的 towupper()函数得到 wch 的大写形式。

wprintf()和 printf()函数的唯一区别是，前者的第一个参数是宽字符串。最后的输出操作也可以使用 printf()完成，只需从第一个参数的格式字符串中去掉 L 前缀即可。

10.5 其他输出函数

除了 printf()和 wprintf()函数有输出字符串的功能外，在<stdio.h>头文件中声明的 puts()函数也可以输出字符串。puts()函数与 gets()函数互补。这个函数的名称来自于其用途：放置(put)字符串。puts()函数的一般形式如下：

```
int  puts(const char  *string);
```

puts()函数接受字符串指针作为变元，将字符串写入标准输出流 stdout。其字符串必须用字符'\0'终止。puts()函数的参数是 const，所以该函数不能修改传送给它的字符串。如果输出错误，它会返回一个负整数，否则就返回非负数。puts()函数用于输出单行信息，例如：

```
puts("Is there no end to input and output?");
```

这行语句输出作为变元传入的字符串，然后将光标移动到下一行。使用 printf()函数必须在字符串的尾部加入'\n'，才能达到这样的效果。

注意：

puts()函数会在作为参数传入的字符串尾部添加'\n'字符，输出多行数据。

10.5.1 屏幕的非格式化输出

函数 putchar()也是包含在<stdio.h>头文件中，与函数 getchar()互补。putchar()函数的一般形式如下：

```
int  putchar(int  c);
```

putchar()函数将单个字符 c 输出到 stdout 上，并返回所显示的字符。它可以输出信息，

www.TopSage.com

一次显示一个字符，这会使程序比较大，但能控制是否输出某些字符。例如，下面的语句输出一个字符串：

```
char string[] = "Beware the Jabberwock, \nmy son!";
puts(string);
```

也可以编写下面的语句：

```
char string[] = " Beware the Jabberwock, \nmy son!";
int i = 0;
while( string[i] != '\0')
  if(string[i] != '\n')
    putchar(string[i++]);
```

第一段语句在两行上输出字符串，如下所示：

```
Beware the Jabberwock,
my son!
```

第二段语句跳过字符串中的换行符，所以输出如下：

```
Beware the Jabberwock, my son!
```

使用 putchar()函数不仅可以输出这么简单的信息，还可以从字符串的中间选择输出给定界定符指定的一串字符，或者转换字符串中的某些字符，之后输出，例如将制表符转换为空格。

10.5.2 数组的格式化输出

使用在<stdio.h>头文件中声明的 sprintf()函数，可以将格式化数据写入 char 类型的数组中。这个函数的原型如下：

```
int sprintf(char *str, const char *format, . . .);
```

这个函数根据第二个格式字符串参数输出第三个参数和后续参数指定的数据，其工作方式与 printf()相同，只是将数据写入函数第一个参数指定的字符串中。它返回的整数是写入 str 的字符数，不包括终止字符。下面的代码演示了这个函数：

```
char result[20];      /* Output from sprintf */
int count = 4;
int nchars = sprintf(result, "A dog has %d legs.", count);
```

这段代码使用第二个参数指定的格式字符串将 count 的值写入 result。所以，result 包含字符串"A dog has 4 legs."。nchars 变量的值是 17，与执行 sprintf()调用后 strlen(result)的返回值相同。如果在操作中出现了编码错误，sprintf()函数就返回一个负整数。

sprintf()函数的一个用途是以编程方式创建格式字符串。第 12 章的程序 12.8 是该函数的一个简单例子。

www.TopSage.com

10.5.3 数组的格式化输入

sscanf()函数与 sprintf()函数互补，因为 sscanf()函数可以在格式字符串的控制下，从 char 类型的数组元素中读取数据。函数的原型如下：

```
int sscanf(const char *str, const char *format, . . .);
```

根据格式字符串 format，数据从 str 读入第三个参数和后续参数指定的变量中。这个函数返回读取的数据项个数。如果读取和存储数据值之前出现了错误，就返回 EOF。字符串用文件结束条件来表示其结束，所以如果在转换值之前到达了 str 字符串的末尾，就返回 EOF。

下面演示了 sscanf()函数的用法：

```
char *source = "Fred 94";
char name[10];
int age = 0;
int items = sscanf(source, " %s %d", name, age);
```

执行这段代码的结果是：name 包含字符串"Fred"，age 的值是 94。items 变量的值是 2，因为从 source 中读取了两项。

sscanf()函数的一个用途是尝试读取同一数据的各种方式。可以把输入行读入一个数组中，作为字符串，之后使用 sscanf()从数组中再次读取这个输入行，但使用不同的格式字符串。

10.6 打印机输出

将数据写入打印机不是 C 语言的一个标准，但一些 C 库支持它，所以这里简单讨论一下。要将输出写入默认打印机，可以使用比 printf()更常见的函数 fprintf()。这个函数可以将格式化输出传送到任何流中，比较多见的是传送到磁盘上的文件中。不过本章只介绍打印机输出功能。fprintf()的一般形式如下：

```
fprintf(stdprn, format_string, argument1, argument2, ..., argumentn);
```

这个函数会返回 int 类型的值，即输出到 stdprn 上的字符数。除了第一个变元和函数名中的 f 外，fprintf()和 printf()完全相同。如果编译器和库没有定义 stdprn，应查看文档说明，了解如何处理打印，但许多 C 库都定义了 stdprn。使用 fprintf()函数将数据输出到打印机上时，所使用的格式字符串和指定符与 printf()函数完全相同，但其中有几个地方要注意，下面用一个例子说明。

试试看：打印机输出

这个程序是一个输出到打印机的例子：

```
/* Program 10.12 Printing on a printer - where else? */
```

www.TopSage.com

```
#include <stdio.h>

int main(void)
{
   fprintf(stdprn, "The barber shaves all those who do not"
           " shave themselves.");
  fprintf(stdprn, "\n\rQuestion: Who shaves the barber?\n\r");
  fprintf(stdprn, "\n\rAnswer: She doesn't need to shave.\f");
  return 0;
}
```

代码的说明

唯一要说明的是转义序列\r 和\f。在打印机中，\n\r 相当于换行/回车，\f 是换页符，使打印机换一页纸。

10.7 小结

本章选择介绍了前面讨论过的各种格式指定符，但还有许多未介绍的格式指定符。熟悉它们的唯一方式是实践，最好在真实环境下实践。理解各种编码并不等同于熟悉，要熟练运用它们，必须在实际的程序中多次使用它们。附录 D 提供了快速参考。

10.8 习题

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容，还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 10.1 编写一个程序，读入、存储以及输出下列 5 种类型的字符串，每个字符串占一行，字符串间不能有空格。

- 类型 1：一串小写字母，后跟一个数字(如 numberl)
- 类型 2：两个单词，每个单词的第一个字母大写，单词间用-分隔(如 Seven-Up)
- 类型 3：小数(如 7.35)
- 类型 4：一串大小写字母以及空格(如 Oliver Hardy)
- 类型 5：一串除了空格及数字外的任何字符(如 floating-point)

以下是这 5 种输入类型的例子，要分开读入这些字符串：

```
babylon5John-Boy3.14159Stan  Laurel'Winner!'
```

习题 10.2 编写一个程序，读入以下数值，并输出它们的和：

```
$3.50, $ 0.75, $9.95, $2. 50
```

习题 10.3 定义一个函数，其参数是一个 double 类型的数组，输出该数组和数组中

www.TopSage.com

的元素个数。这个函数的原型如下：

```
void show(double array[], int array_size, int field_width);
```

输出的值 5 个一行，每个值有两位小数，字符宽度是 12。在程序中使用这个函数输出从 1.5 到 4.5 的值，每次增加 0.3(如：1.5、1.8、2.1、…、4.5)。

习题 10.4 定义一个函数，使用 getchar()函数从 stdin 中读入一个字符串，这个字符串用特定的字符终止，这个特定的终止字符作为第二个变元传给这个函数。因此，函数的原型如下：

```
char *getString(char *buffer, char end_char);
```

返回值是一个指针，它是这个函数的第一个变元。编写一个程序，使用这个函数从键盘上读取并输出 5 个以冒号终止的字符串。

www.TopSage.com

第 11 章

结构化数据

前面学习了如何声明和定义变量，使之包含各种类型的数据，如整数、浮点数和字符等。学习了如何创建这些类型的数组及指针数组，这些指针指向包含可用数据类型的内存位置。这些很有用，但是许多应用程序还需要一些更灵活的功能。

例如，要编写一个处理马匹数据的程序，就需要每匹马的名字、出生日期、颜色、高度和它的父母等。在这些数据中，一些项是字符串，一些项是数值。因此，必须为每一种数据类型建立数组并存储它们。但这是有限制的，例如不能方便地引用 Dobbin 的生日或 Trigger 的身高。必须通过一个通用索引将数据项关联起来，使数组同步。C 语言在这方面提供了相当好的方法，这也是本章将要讨论的主题。

本章的主要内容：

- 什么是数据结构
- 如何声明并定义数据结构
- 如何使用结构和结构指针
- 如何将指针作为结构的成员
- 如何在变量间共享内存
- 如何定义自己的数据类型
- 如何编写程序，根据数据生成条形图

11.1 数据结构：使用 struct

关键字 struct 能定义各种类型的变量集合，称为结构(structure)，并把它们视为一个单元。下面是一个简单的结构声明例子：

```
struct  horse
{
  int  age;
  int  height;
} $lver;
```

这个例子声明了一个结构 horse。horse 不是一个变量名，而是一个新的类型，这个类型名称通常称为结构标记符(structure tag)或标记符名称(tag name)。结构标记符的命名方式和我们熟悉的变量名相同。

www.TopSage.com

注意：

结构标记符可以和变量使用相同的名称，但最好不要这么做，因为这会使代码难以理解。

结构内的变量名称 age 和 height 称为结构成员(structure members)。在这个例子中，它们都是 int 类型。结构成员出现在结构标记符名称 horse 后的大括号内。

在这个结构例子中，结构的一个实例 Silver 是在定义结构时声明的。它是一个 horse 类型的变量，只要使用变量名称 Silver，它都包含两个结构成员：age 和 height。

下面是 horse 结构类型的稍微复杂的声明：

```
struct  horse
{
  int  age;
  int  height;
  char  name[20];
  char  father[20];
  char  mother[20];
}  Dobbin  =  {
        24, 17, "Dobbin", "Trigger", "Flossie"
        };
```

结构内的成员可以是任何类型的变量，包含数组在内。在 horse 结构类型的这个版本中，有 5 个成员：整数成员 age 和 height，数组成员 name、father 和 mother。每个成员的声明方式和一般变量的声明方式相同，都是先声明类型，然后是名称，最后用分号结束。注意，初始化值不能放在这里，因为现在是定义 horse 类型的成员，而不是在声明变量。结构类型是一种说明或一种蓝图，可以用于定义该类型的变量——就这个例子而言，类型是 horse。

在 horse 结构定义的闭括号后定义了一个实例变量 Dobbin。给 Dobbin 赋予初始值的方式和数组类似，所以在定义 horse 类型的实例时，可以指定初始值。

在 Dobbin 变量的声明中，最后一对大括号内的值按顺序赋予成员变量 age(24)、height(17)、name("Dobbin")、father("Trigger")和 mother("Flossie")。该语句用分号结束。变量 Dobbin 现在引用了结构内所有的成员。结构 Dobbin 占用的内存如图 11-1 所示(假定 int 类型的变量占 4 个字节)。通常可以使用 sizeof 运算符计算出结构占用的内存量。

结构变量 Dobbin

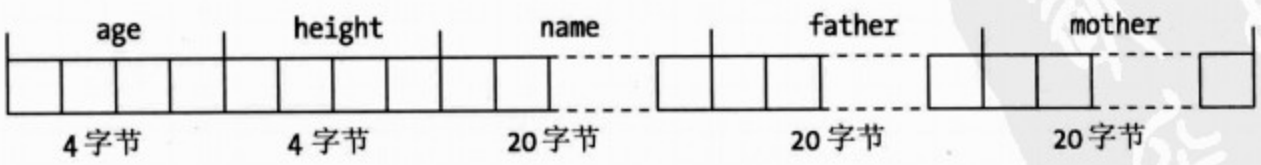


图 11-1　Dobbin 占用的内存

www.TopSage.com

11.1.1 定义结构类型和结构变量

可以将结构的声明和结构变量的声明分开。取代前面例子的语句如下：

```
struct  horse
{
  int  age;
  int  height;
  char  name[20];
  char  father[20];
  char  mother[20];
};

struct  horse  Dobbin  =  {
                          24, 17, "Dobbin", "Trigger", "Flossie"
                          };
```

现在有两个分开的语句。第一个定义结构标记符 horse，第二个声明该类型的变量 Dobbin。结构定义和结构变量声明语句都用分号结束。在 Dobbin 结构成员的初始值中，Dobbin 的父亲是 Trigger，母亲是 Flossie。

也可以给前面两个例子添加第三条语句，定义另一个 horse 类型的变量：

```
struct  horse  Trigger  =  {
                           30, 15, "Trigger", "Smith", "Wesson"
                           };
```

现在有一个变量 Trigger，它包含 Dobbin 父亲的数据，显然，Trigger 的父母是 Smith 和 Wesson。

当然，也可以在一行语句中声明多个结构变量。声明的方式和声明 C 语言标准类型的多个变量一样。例如：

```
struct  horse  Piebald, Bandy;
```

这行语句声明了两个 horse 类型的变量。比起标准类型的声明，这个声明只增加了关键字 struct。为了使这行语句简单，没有初始化变量，不过一般应初始化变量。

11.1.2 访问结构成员

现在知道如何定义结构及声明结构变量了，还必须引用结构的成员。结构变量的名称不是一个指针，所以需要特殊的语法访问这些成员。

要引用结构成员，应在结构变量名称的后面加上一个句点，再加上成员变量名称。例如，发现 Dobbin 隐瞒了它的年龄，事实上它比初始化的值年轻，就可以将值修正如下：

```
Dobbin.age  =  12;
```

结构变量名称和成员名称间的句点是一个运算符，称为成员选择运算符。这行语句

www.TopSage.com

将 Dobbin 结构的 age 成员设定成 12。结构成员和相同类型的变量完全一样，可以给它们设定值，也可以在表达式中像使用一般变量一样使用它们。

试试看：使用结构

尝试将前面所学的 horse 结构用于一个简单的例子：

```
/* Program 11.1 Exercising the horse */
#include <stdio.h>

int main(void)
{
  /* Structure declaration */
  struct horse
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
  };

  struct horse My_first_horse; /* Structure variable declaration */

  /* Initialize the structure variable from input data */
  printf("Enter the name of the horse: " );
  scanf("%s", My_first_horse.name ); /* Read the horse's name */

  printf("How old is %s? ", My_first_horse.name );
  scanf("%d", &My_first_horse.age ); /* Read the horse's age */

  printf("How high is %s ( in hands )? ", My_first_horse.name );
  scanf("%d", &My_first_horse.height ); /* Read the horse's height */

  printf("Who is %s's father? ", My_first_horse.name );
  scanf("%s", My_first_horse.father ); /* Get the father's name */

  printf("Who is %s's mother? ", My_first_horse.name );
  scanf("%s", My_first_horse.mother ); /* Get the mother's name */

  /* Now tell them what we know */
  printf("\n%s is %d years old, %d hands high,",
       My_first_horse.name, My_first_horse.age, My_first_horse.height);
  printf(" and has %s and %s as parents.\n", My_first_horse.father,
      My_first_horse.mother );
  return 0;
}
```

www.TopSage.com

根据输入的数据，得到的输出如下：

```
Enter the name of the horse: Neddy
How old is Neddy? 12
How high is Neddy ( in hands )? 14
Who is Neddy's father? Bertie
Who is Neddy's mother? Nellie

Neddy is 12 years old, 14 hands high, and has Bertie and Nellie as parents.
```

代码的说明

引用结构成员的方式使这个例子非常容易理解。用下面的语句定义 horse 结构：

```
struct horse
{
  int age;
  int height;
  char name[20];
  char father[20];
  char mother[20];
};
```

这个结构有两个整数成员 age 和 height，以及三个字符数组成员 name、father 和 mother。在闭括号的后面仅是一个分号，还没有声明 horse 类型的变量。在定义完 horse 结构后，具有如下语句：

```
struct horse My_first_horse; /* Structure variable declaration */
```

这行语句声明 My_first_horse 是一个 horse 类型的变量，没有指定初值。

然后，使用下面的语句为 My_first_horse 结构的成员 name 读入数据：

```
scanf("%s", My_first_horse.name ); /* Read the horse's name */
```

这里不需要使用寻址运算符(&)，因为结构的成员 name 是一个数组，所以将数组第一个元素的地址隐式传送给函数 scanf()。要引用结构成员，应使用结构名称 My_first_horse，后跟一个句点和成员的名称 name。访问结构成员时，除了表示方法不同外，其他的和一般变量完全相同。

接下来给 horse 的成员 age 读入数值：

```
scanf("%d", &My_first_horse.age ); /* Read the horse's age */
```

由于这个成员是 int 类型的变量，所以必须使用&运算符传递这个结构成员的地址。

注意：

对 struct 对象的成员使用寻址运算符时，要将&放在成员的引用之前，而不是放在成员名称之前。

后面的语句使用相同的方式为结构的其他成员读入数据，并对每个输入显示提示。

www.TopSage.com

输入完成后，就使用下面的语句将读入的数值输出到一行上。

```
printf("\n%s is %d years old, %d hands high,",
  My_first_horse.name, My_first_horse.age, My_first_horse.height);
printf(" and has %s and %s as parents.\n", My_first_horse.father,
  My_first_horse.mother );
```

引用结构成员的名字很长，使语句看起来很复杂，其实它是相当简单的。程序使用变量成员的名字作为函数的第一个参数，这是以前介绍的格式控制字符串的标准形式。

11.1.3 未命名的结构

不一定要给结构指定标记符名字。用一条语句声明结构和该结构的实例时，可以省略标记符名字。在上一个例子中，声明了 horse 类型和该类型的实例 My_first_horse，也可以改为：

```
struct
{                    /* Structure declaration and... */
  int age;
  int height;
  char name[20];
  char father[20];
  char mother[20];
} My_first_horse; /* ...structure variable declaration combined */
```

使用这种方法的最大缺点是不能在其他语句中定义这个结构的其他实例。这个结构类型的所有变量必须在一行语句中定义。

11.1.4 结构数组

保存马匹数据的基本方法就是这样，但在处理 50 或 100 匹马如此大量的数据时会比较麻烦，此时需要一个更可靠的方法去处理大量的马匹数据。使用变量也会遇到这个问题。此时解决方法是使用数组，这里也可以声明一个 horse 数组。

试试看：使用结构数组

扩展前一个例子，以处理几匹马的数据：

```
/* Program 11.2 Exercising the horses */
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  struct horse /* Structure declaration */
  {
    int age;
```

www.TopSage.com

```
  int height;
  char name[20];
  char father[20];
  char mother[20];
};

struct horse My_horses[50]; /* Structure array declaration */
int hcount = 0;             /* Count of the number of horses */
char test = '\0';           /* Test value for ending */

for(hcount = 0; hcount<50 ; hcount++ )
{
  printf("\nDo you want to enter details of a%s horse (Y or N)? ",
          hcount?"nother " : "" );
  scanf(" %c", &test );
  if(tolower(test) == 'n')
    break;

  printf("\nEnter the name of the horse: " );
  scanf("%s", My_horses[hcount].name ); /* Read the horse's name */

  printf("\nHow old is %s? ", My_horses[hcount].name );
  scanf("%d", &My_horses[hcount].age ); /* Read the horse's age */

  printf("\nHow high is %s ( in hands )? ", My_horses[hcount].name );
  /* Read the horse's height*/
  scanf("%d", &My_horses[hcount].height );

  printf("\nWho is %s's father? ", My_horses[hcount].name );
  /* Get the father's name */
  scanf("%s", My_horses[hcount].father );

  printf("\nWho is %s's mother? ", My_horses[hcount].name );
  /* Get the mother's name */
  scanf("%s", My_horses[hcount].mother );
 }

 /* Now tell them what we know. */
 for(int i = 0 ; i<hcount ; i++ )
 {
  printf("\n\n%s is %d years old, %d hands high,",
      My_horses[i].name, My_horses[i].age, My_horses[i].height);
  printf(" and has %s and %s as parents.", My_horses[i].father,
      My_horses[i].mother );
 }
 return 0;
}
```

这个程序的输出和前一个只处理一匹马的例子有点不同。输入每匹马的数据时，都

www.TopSage.com

会显示提示。50 匹马的数据输入完后，程序就输出所有数据的小结。整个机制是稳定的，运行良好(几乎总是成功)。

代码的说明

在这个马匹数据处理版本中，首先声明 horse 结构，如下:

```
struct horse /* Structure declaration */
```

这条语句声明变量 My_horses 是一个有 50 个 horse 结构的数组。除了关键字 struct 外，这个数组声明与其他的数组声明都相同。

然后是一个用变量 hcount 控制的 for 循环:

```
for(hcount = 0;  hcount  <  50  ;  hcount++  )
{
 ...
}
```

这个循环让程序读入 50 匹马的数据。循环控制变量 hcount 用来累加 horse 结构的总数。循环内的第一个动作是:

```
printf("\nDo you want to enter details of a%s horse (Y or N)? ",
      hcount?"nother " : "" );
scanf(" %c", &test );
if(tolower(test) == 'n')
  break;
```

每次迭代都要求用户输入 Y 或 N，指定是否输入另一匹马的数据。在第一次之后的每次迭代中，printf()语句都使用条件运算符在输出中插入"nother"。在使用 scanf()读完用户输入的字符后，如果用户的响应是否定的，if 语句就会执行 break 语句，跳出循环。

接下来的一串 printf()同 scanf()同以前一样，但有两点需要注意，如下面的语句:

```
scanf("%s", My_horses[hcount].name ); /* Read the horse's name */
```

从上述语句可以看出，引用结构数组的一个元素成员的方法非常简单。这个结构数组名称将索引放在方括号内，后跟句点和成员名。如果想引用这个结构第 4 个元素的 name 数组的第 3 个元素，可以使用:

```
My_horses[3].name[2]
```

注意:

结构数组的索引与其他类型的数组一样，也是从 0 开始，所以结构数组的第 4 个元素的索引值是 3，而其成员数组的第 3 个元素的索引值是 2。

现在看看下面的语句:

```
scanf("%d", My_horses[hcount].age); /* Read the horse's age */
```

注意，如果传给 scanf()的变元是字符串数组变量，就不需要寻址运算符，如 My_horses

www.TopSage.com

[hcount].name。但如果变元是整数，如 My_horses[hcount].age 和 My_horses[hcount].height，就必须使用寻址运算符。在读入变量值时很容易忘掉&，所以要特别注意。

前面介绍的结构不仅适用于马匹数据的应用程序，也适用于猪或驴等的数据处理。

11.1.5 表达式中的结构

结构中的成员可以像一般变量那样用于表达式。以程序 11.2 中的结构为例，可以将它们用在下面的表达式中：

```
My_horses[1].height = (My_horses[2].height + My_horses[3].height]/2;
```

一匹马的高度是另两匹马的平均高度是没什么道理的，但这是一个合法的语句。也可以在赋值语句中使用整个结构元素。

```
My_horses[1]  =  My  horses[2];
```

这行语句会将结构 My_horses[2]的所有成员复制到结构 My_horses[1]中，使这两个结构完全相同。使用整个结构的另一个操作是使用&运算符提取地址。但是不能对整个结构执行加、比较或其他操作。为此，必须编写定制的函数。

11.1.6 结构指针

要获得结构的地址，就需要使用结构的指针。由于需要的是结构的地址，因此需要声明结构的指针。结构指针的声明方式和声明其他类型的指针变量相同，例如：

```
struct  horse  *phorse;
```

这条语句声明了一个 phorse 指针，它可以存储 horse 类型的结构地址。现在可以将 phorse 设置为一个特定结构的地址值，使用的方法和其他类型的指针完全相同，例如：

```
phorse = &My_horses[1];
```

现在 phorse 指向结构 My_horses[1])。可以通过 phorse 指针引用这个结构的元素。因此，如果要显示这个结构成员的名字，可以编写如下语句：

```
printf("\nThe  name  is  %s.",   (*phorse).name);
```

取消引用指针的括号是非常重要的，因为成员选择运算符(句点)的优先级高于取消引用指针运算符*。这个操作还有另一种方法，且更容易理解。将上面的语句改写成：

```
printf("\nThe  name  is  %s.",   phorse->name);
```

这就不需要括号或星号了。->运算符是一个负号后跟一个大于号。这个运算符有时也称为成员指针运算符。这个表示法几乎可用于取代通常的取消引用指针表示法。因为这个运算符使程序更容易理解。

www.TopSage.com

11.1.7 为结构动态分配内存

可以利用前面掌握的各种工具重写程序 11.2，以更经济的方式使用内存。程序 11.2 的最初版本为包含 50 个 horse 结构的数组分配了内存，而实际上并不需要这么多内存。

要为结构动态分配内存，可以使用结构指针数组，其声明非常简单，如下所示：

```
struct  horse  *phorse[50];
```

这行语句声明了 50 个指向 horse 结构的指针数组。该语句只给指针分配了内存。还需要分配一些内存来存储每个结构的成员。

试试看：使用结构指针

下面的例子演示了如何为结构动态分配内存：

```
/* Program 11.3 Pointing out the horses */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>    /* For malloc() */

int main(void)
{
  struct horse          /* Structure declaration */
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
};

struct horse *phorse[50]; /* pointer to structure array declaration */
int hcount = 0;           /* Count of the number of horses */
char test = '\0';         /* Test value for ending input */

for(hcount = 0; hcount < 50 ; hcount++ )
{
  printf("\nDo you want to enter details of a%s horse (Y or N)? ",
        hcount?"nother " : "" );
  scanf(" %c", &test );
  if(tolower(test) == 'n')
    break;

  /* allocate memory to hold a structure */
  phorse[hcount] = (struct horse*) malloc(sizeof(struct horse));

  printf("\nEnter the name of the horse: " );
  scanf("%s", phorse[hcount]->name ); /* Read the horse's name */
```

www.TopSage.com

```
    printf("\nHow old is %s? ", phorse[hcount]->name );
    scanf("%d", &phorse[hcount]->age ); /* Read the horse's age */

    printf("\nHow high is %s ( in hands )? ", phorse[hcount]->name );
    scanf("%d", &phorse[hcount]->height ); /* Read the horse's height */

    printf("\nWho is %s's father? ", phorse[hcount]->name );
    scanf("%s", phorse[hcount]->father ); /* Get the father's name */

    printf("\nWho is %s's mother? ", phorse[hcount]->name );
    scanf("%s", phorse[hcount]->mother ); /* Get the mother's name */
  }

  /* Now tell them what we know. */
  for(int i = 0 ; i < hcount ; i++ )
  {
    printf("\n\n%s is %d years old, %d hands high,",
          phorse[i]->name, phorse[i]->age, phorse[i]->height);
    printf(" and has %s and %s as parents.",
          phorse[i]->father, phorse[i]->mother);
    free(phorse[i]);
  }
  return 0;
}
```

输入和程序 11.2 相同的数据，则输出也相同。

代码的说明

这和前一个版本非常类似，但是其运行并不相同。一开始没有为任何结构分配内存。下面的声明：

```
struct horse *phorse[50]; /* pointer to structure array declaration */
```

仅定义了 50 个 horse 类型的结构指针，还要将结构放在指针指向的地址中，如下：

```
phorse[hcount] = (struct horse*) malloc(sizeof(struct horse));
```

这行语句会给每个结构分配内存空间。malloc()函数会分配变元指定的字节数，并将所分配内存块的地址返回为 void 类型的指针。这个例子使用 sizeof 运算符计算需要的字节数。

使用 sizeof 运算符可以计算出结构所占的字节数，其结果不一定对应于结构中各个成员所占的字节数总和，如果自己计算，就很容易出错。

除了 char 类型的变量之外，2 字节变量的起始地址常常是 2 的倍数，4 字节变量的起始地址常常是 4 的倍数，依此类推。这称为边界调整(boundary alignment)，它和 C 语言无关，而是硬件的要求。以这种方式在内存中存储变量，可以更快地在处理器和内存之间传递数据，但不同类型的成员变量之间会有未使用的字节。这些未使用的字节也必须算在结构的字节数中，如图 11-2 所示。

www.TopSage.com

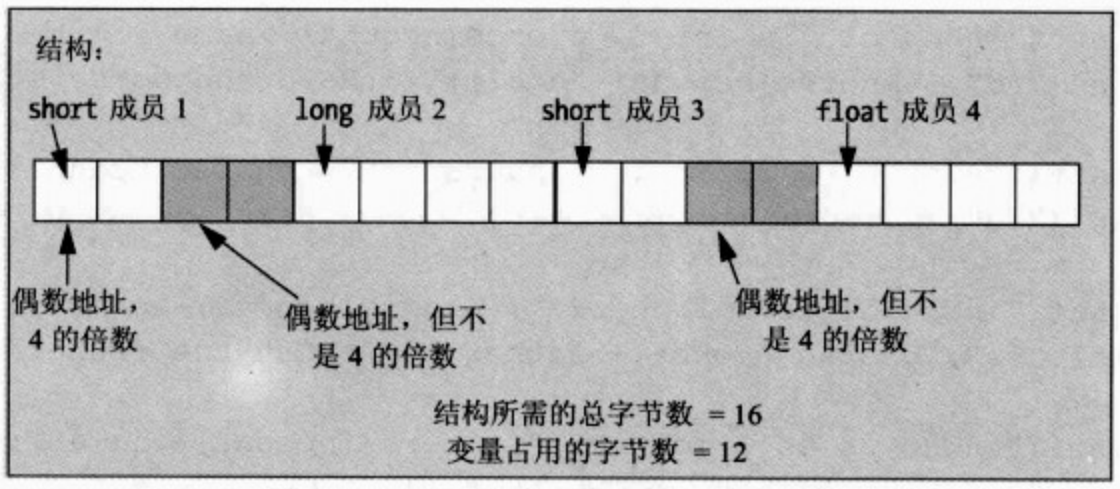


图 11-2　边界调整在内存分配上的影响

malloc()函数返回的值是一个 void 指针，因此必须用表达式(struct horse*)将它转换成所需要的类型。这样，这个指针在必要时就可以正确地递增或递减了。

```
scanf("%s", phorse[hcount]->name ); /* Read the horse's name */
```

这行语句使用新的表示法，通过指针选择结构的成员。它比 (*phorse[hcount]).name 清楚得多。以后引用 horse 结构的成员都使用这种新的表示法。

最后，程序给每匹马的输入数据显示一个总结，然后释放内存。

11.2　再探结构成员

前面说过，所有基本数据类型(包含数组)都可以成为结构的成员。除此之外，还可以把一个结构作为另一个结构的成员，不仅指针可以是结构的成员，结构指针也可以是结构的成员。

使用结构为编程打开了一个全新领域的大门，同时也增加了潜在的危机。下面逐一探讨这些内容，深入了解结构成员的组成。

11.2.1　将一个结构作为另一个结构的成员

本章的开头为满足马饲养员的需要，设计了一个程序，处理每匹马的各种数据，包括名字、身高和生日等，但程序 11.1 用年龄代替了生日。其部分原因是日期处理起来比较麻烦，要用 3 个数值表示，还要处理闰年的问题。现在准备将一个结构作为另一个结构的成员来处理日期。

可以定义一个用于保存日期的结构类型。下面的语句用标记符名称 Date 定义了这个结构：

```
struct  Date
{
  int day;
  int month;
  int year;
```

www.TopSage.com

```
};
```

现在定义结构 horse，其中包含出生日期变量，如下所示：

```
struct  horse
{
  struct  Date  dob;
  int  height;
  char  name[20];
  char  father[20];
  char  mother[20];
};
```

现在结构中有一个变量成员，它代表马的出生日期的结构。接下来用通常的语句定义一个 horse 结构的实例，如下所示：

```
struct  horse  Dobbin;
```

用与前面相同的语句为成员 height 设定值：

```
Dobbin.height = 14;
```

要在一系列赋值语句中设定出生日期，可以使用下面的逻辑：

```
Dobbin.dob.day  =  5;
Dobbin.dob.month  =  12;
Dobbin.dob.year  =  1962;
```

这是一匹很老的马，表达式 Dobbin.dob.day 引用了 int 类型的变量，所以可以将它用于算术表达式或比较表达式。但如果使用 Dobbin.dob，就会引用一个 Date 类型的结构变量。Date 不是一个基本类型，而是一个结构，所以只能使用下面的方式赋值：

```
Trigger.dob  =  Dobbin.dob;
```

这行语句表示两匹马是双胞胎，但不能保证事实如此。

可以将第一个结构用作第二个结构的成员，再将第二个结构作为第三个结构的成员，依此类推。但 C 编译器只允许结构最多有 15 层。如果结构有这么多层，则引用最底层的成员时，需要输入所有的结构成员名称。

11.2.2 声明结构中的结构

可以在 horse 结构的定义中声明 Date 结构，如下：

```
struct  horse
{
  struct  Date
  {
    int day;
    int month;
```

www.TopSage.com

```
    int year;
  } dob;
  int  height;
  char  name[20];
  char  father[20];
  char  mother[20];
};
```

这个声明将 Date 结构声明放在 horse 结构的定义内，因此不能在 horse 结构的外部声明 Date 变量。当然，每个 horse 类型的变量都包含 Date 类型的成员 dob。但下面的语句：

```
struct  Date  my_date;
```

会导致编译错误。错误信息会说明 Date 结构类型未定义。如果需要在 horse 结构的外部使用 Date，就必须将它定义在 horse 结构之外。

11.2.3 将结构指针用作结构成员

任何指针都可以是结构的成员，包含结构指针在内。结构成员指针可以指向相同类型的结构。例如，horse 类型的结构可以含有一个指向 horse 类型结构的指针。

试试看：将结构指针用作结构成员

修改前一个例子，让结构含有指向同类型结构的指针：

```
/* Program 11.4 Daisy chaining the horses */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int main(void)
{
  struct horse                    /* Structure declaration */
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
    struct horse *next;           /* Pointer to next structure */
  };

  struct horse *first = NULL;     /* Pointer to first horse */
  struct horse *current = NULL;   /* Pointer to current horse */
  struct horse *previous = NULL; /* Pointer to previous horse */

  char test = '\0';               /* Test value for ending input */
```

www.TopSage.com

```
for( ; ; )
{
  printf("\nDo you want to enter details of a%s horse (Y or N)? ",
        first != NULL?"nother " : "" );
  scanf(" %c", &test );
  if(tolower(test) == 'n')
    break;

  /* Allocate memory for a structure */
  current = (struct horse*) malloc(sizeof(struct horse));

  if(first == NULL)
    first = current;                /* Set pointer to first horse */

  if(previous != NULL)
    previous -> next = current; /* Set next pointer for previous horse */

  printf("\nEnter the name of the horse: ");
  scanf("%s", current -> name); /* Read the horse's name */

  printf("\nHow old is %s? ", current -> name);
  scanf("%d", &current -> age); /* Read the horse's age */

  printf("\nHow high is %s ( in hands )? ", current -> name );
  scanf("%d", &current -> height); /* Read the horse's height */

  printf("\nWho is %s's father? ", current -> name);
  scanf("%s", current -> father); /* Get the father's name */

  printf("\nWho is %s's mother? ", current -> name);
  scanf("%s", current -> mother); /* Get the mother's name */

  current->next = NULL;             /* In case it's the last... */
  previous = current;               /* Save address of last horse */
}

/* Now tell them what we know. */
current = first;                    /* Start at the beginning */

while (current != NULL)             /* As long as we have a valid pointer */
{ /* Output the data*/
  printf("\n\n%s is %d years old, %d hands high,",
      current->name, current->age, current->height);
  printf(" and has %s and %s as parents.", current->father,
       current->mother);
  previous = current; /* Save the pointer so we can free memory */
  current = current->next; /* Get the pointer to the next */
  free(previous);                   /* Free memory for the old one */
}
```

www.TopSage.com

```
  return 0;
}
```

如果输入相同，这个例子会产生和程序 11.3 相同的输出。

代码的说明

这次不但没有为结构分配空间，而且只定义了三个指针。这些指针用下面的语句声明和初始化:

```
struct horse *first = NULL;      /* Pointer to first horse */
struct horse *current = NULL;    /* Pointer to current horse */
struct horse *previous = NULL;   /* Pointer to previous horse */
```

每个指针都定义成 horse 结构的指针。first 指针仅用于存储第一个结构的地址。第二和第三个指针是工作用的存储器: current 存储了正在处理的 horse 结构的地址，previous 跟踪前一个处理过的结构的地址。

horse 结构中新增的 next 成员是指向 horse 结构的指针。每个 horse 结构中的 next 都指向下一个 horse 的地址，以链接所有的 horse 结构。但最后一个 horse 结构例外，它的 next 设定成 NULL。这个结构的其他方面与前面相同，如图 11-3 所示。

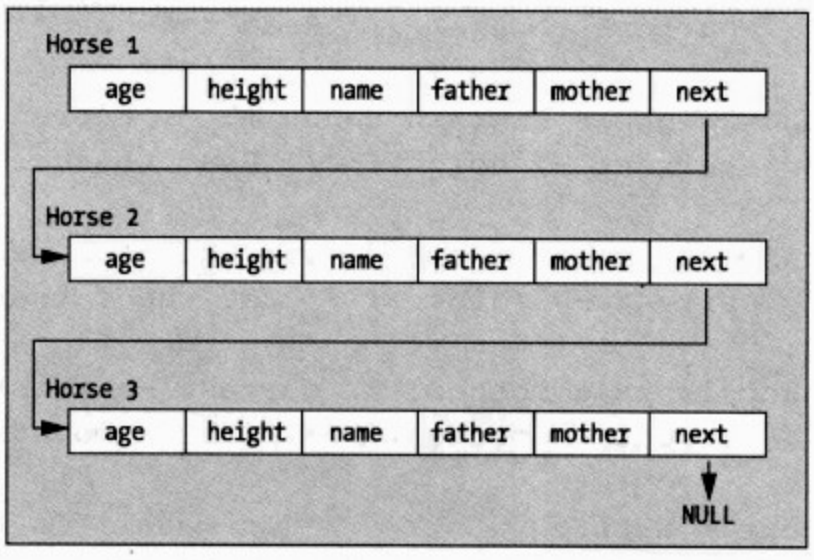


图 11-3 链接起来的 horse 结构

输入循环如下:

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

因为没有使用数组，不需要考虑索引，所以输入循环是一个无限循环。也不需要计算读入了多少组数据，所以不需要使用变量 hcount 及循环变量 i。因为给每个 horse 结构分配了内存，所以只需接受输入的数据。

循环中的开始语句如下:

```
printf("\nDo you want to enter details of a%s horse (Y or N)? ",
```

www.TopSage.com

```
      first != NULL?"nother " : "" );
    scanf(" %c", &test );
    if(tolower(test) == 'n')
      break;
```

在提示后，如果回答是 N 或 n，就结束循环。否则，就准备接受另一组结构成员。first 指针只有在第一次迭代时是 NULL，所以在第二次以后的迭代中，其提示信息会和第一次稍有不同。

回答了循环开头的问题后，就执行下面的语句：

```
current = (struct horse*) malloc(sizeof(struct horse));

if(first == NULL)
  first = current;                    /* Set pointer to first horse */

if(previous != NULL)
  previous -> next = current;  /* Set next pointer for previous horse */
```

每次迭代时，都为当前的结构分配必要的内存。为了精简程序，没有检查 malloc()函数是否返回了 NULL，但在实际使用时应检查。

如果指针 first 等于 NULL，就表示是第一次迭代，即开始输入第一个结构。因此，将 first 指针设置为 malloc()函数返回的指针值，即 current 变量存储的值。first 中的地址也是访问链中第一个 horse 结构的关键。可以从 first 中的地址开始，利用成员 next 指针得到下一个结构的地址，再依序访问下一个结构，从而到达任何一个 horse 结构。

如果有下一个结构，就必须将 next 指针指向这个结构，但只要有下一个结构，就可以确定其地址。因此，在第二次和后续的迭代中，应将当前结构的地址存储到前一个结构的 next 成员中，前一个结构的地址存放到 previous 指针中。在第一次迭代中，previous 的指针是 NULL，所以什么也不做。

在完成了所有的输入语句，到循环的最后，有下面两行语句：

```
current->next = NULL;   /* In case it's the last…*/
previous = current;     /* Save address of last horse */
```

在 current 指向的结构中，next 指针设定成 NULL，表示这是最后一个结构，没有下一个结构了。如果有下一个结构，指针 next 会在下一次迭代时修改。指针 previous 设定成 current，然后进入下一次迭代，此时 current 指向的结构就是 previous 指向的结构了。

这个程序的优点是生成了 horse 结构链，在这个链中，每个结构的 next 成员都指向下一个结构。最后一个结构例外，因为再也没有下一个 horse 结构了，所以 next 指针包含 NULL，这称为链表。

horse 数据放在链表中后，就可以从第一个结构开始，通过指针成员 next 访问下一个结构。指针 next 是 NULL 时，就到达了链表的末尾。这就是为所有输入生成输出表的方式。

在需要处理数量未知的结构的应用程序中，链表非常有用。链表的主要优点是内存的使用和便于处理。存储和处理链表所占用的内存量最少。即使所使用的内存比较分散，

www.TopSage.com

也可以从一个结构进入下一个结构。因此，链表可以用于同时处理几个不同类型的对象，每个对象都可以用它自己的链表来处理，以优化内存的使用。但链表也有一个小缺点：数据处理的速度比较慢，尤其是要随机访问数据时，速度更慢。

输出过程说明了如何遍历链表，以访问它，语句如下：

```
current = first;        /* Start at the beginning */

while (current != NULL) /* As long as we have a valid pointer */
{ /* Output the data*/
  printf("\n\n%s is %d years old, %d hands high,",
     current->name, current->age, current->height);
  printf(" and has %s and %s as parents.", current->father,
     current->mother);
  previous = current; /* Save the pointer so we can free memory */
  current = current->next;   /* Get the pointer to the next */
  free(previous);            /* Free memory for the old one */
}
```

输出循环由 current 指针控制，它开始时设定成 first。而 first 指针包含链表中第一个结构的地址。循环会遍历链表，显示每个结构的成员，之后把 current 赋予指向下一个结构的成员 next。

结构显示过后就释放其内存。这是很重要的，一旦不再需要引用结构，就释放其内存。但是不能在输出当前结构的所有成员后，马上调用 free()函数。必须先引用当前结构的 next 成员，得到下一个 horse 结构的指针。

在链表的最后一个结构中，next 指针是 NULL，因而结束循环。

11.2.4 双向链表

前一个例子创建的链表有一个缺点：只能往前走。其实，只需小小的修改，就可以得到双向链表(doubly linked list)，可以双向遍历链表。方法是除了指向下一个结构的指针外，在每个结构中再添加一个指针，存储前一个结构的地址。

试试看：双向链表

修改程序 11.4，改成双向链表：

```
/* Program 11.5 Daisy chaining the horses both ways */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int main(void)
{
  struct horse /* Structure declaration */
  {
    int age;
```

www.TopSage.com

```
  int height;
  char name[20];
  char father[20];
  char mother[20];
  struct horse *next;          /* Pointer to next structure */
  struct horse *previous;      /* Pointer to previous structure */
};

struct horse *first = NULL;    /* Pointer to first horse */
struct horse *current = NULL; /* Pointer to current horse */
struct horse *last = NULL;     /* Pointer to previous horse */

char test = '\0';              /* Test value for ending input */

for( ; ; )
{
  printf("\nDo you want to enter details of a%s horse (Y or N)? ",
         first == NULL?"nother " : "");
  scanf(" %c", &test );
  if(tolower(test) == 'n')
    break;

  /* Allocate memory for each new horse structure */
  current = (struct horse*)malloc(sizeof(struct horse));

  if( first == NULL )
  {
    first = current;             /* Set pointer to first horse */
    current->previous = NULL;
  }
  else
  {
    last->next = current;        /* Set next address for previous horse
*/
    current->previous = last; /* Previous address for current horse */
  }

  printf("\nEnter the name of the horse: ");
  scanf("%s", current -> name ); /* Read the horse's name */

  printf("\nHow old is %s? ", current -> name);
  scanf("%d", &current -> age); /* Read the horse's age */

  printf("\nHow high is %s ( in hands )? ", current -> name);
  scanf("%d", &current -> height); /* Read the horse's height */

  printf("\nWho is %s's father? ", current -> name);
  scanf("%s", current -> father); /* Get the father's name */
```

www.TopSage.com

```
    printf("\nWho is %s's mother? ", current -> name);
    scanf("%s", current -> mother); /* Get the mother's name */

    current -> next = NULL; /* In case it's the last horse..*/
    last = current;         /* Save address of last horse */
  }

  /* Now tell them what we know. */
  while(current != NULL) /* Output horse data in reverse order */
  {
    printf("\n\n%s is %d years old, %d hands high,",
       current->name, current->age, current->height);
    printf(" and has %s and %s as parents.", current->father,
        current->mother);
    last = current; /* Save pointer to enable memory to be freed */
    current = current->previous; /* current points to previous in list */
    free(last);                  /* Free memory for the horse we output */
  }
  return 0;
}
```

如果输入相同的数据，这个程序会产生和前一个例子相同的结果，只是显示的顺序相反。

代码的说明

开始的指针声明如下：

```
struct horse *first = NULL;     /* Pointer to first horse */
struct horse *current = NULL;   /* Pointer to current horse */
struct horse *last = NULL;      /* Pointer to previous horse */
```

把在循环的上一个迭代中输入的 horse 结构指针名称改成 last。这么做并不是必需的，但有助于避免和 horse 结构中的成员 previous 混淆。

horse 结构的声明如下：

```
struct horse /* Structure declaration */
{
  int age;
  int height;
  char name[20];
  char father[20];
  char mother[20];
  struct horse *next;      /* Pointer to next structure */
  struct horse *previous;  /* Pointer to previous structure */
};
```

现在 horse 结构有两个指针，一个是往前的指针称为 next，另一个是往后的指针称为 previous。这样就可以双向遍历链表，这也是在程序的最后可以反向输出数据的原因。

www.TopSage.com

除了输出之外，程序的唯一变化是在输入循环的开头添加了使用结构成员指针 previous 的语句：

```
if( first == NULL )
{
  first = current;           /* Set pointer to first horse */
  current->previous = NULL;
}
else
{
  last->next = current;      /* Set next address for previous horse */
  current->previous = last;  /* Previous address for current horse */
}
```

这里用 if-else 取代上一个例子中的两个 if 语句。唯一的区别是设定了结构成员 previous 的值。第一个结构的 previous 设定成 NULL，其后的结构都把 previous 设定成 last，last 的值是在前一次迭代中存储的。

另一个改变是在输入循环的最后。

```
last = current; /* Save address of last horse */
```

添加这行语句，是为了把下一个结构的 previous 指针设定成相应的值，即变量 last 中存储的 current 结构。

输出过程基本上和前一个例子相同，只是从链表中的最后一个结构开始，遍历到第一个结构而已。

11.2.5 结构中的位字段

位字段(bit-fields)提供的机制允许定义变量来表示一个整数中的一个或多个位，这样，就不需要为每个位明确指定成员名称了。

注意：

位字段常用在必须节省内存的情况下。这种情况目前比较少见。与标准类型的变量相比，位字段会明显降低程序执行的速度。因此，必须在节省内存和程序执行速度之间作一个抉择。在大多数情况下，不需要使用位字段，使用它甚至是不理想的，但读者应了解它。

下面是一个声明位字段的例子：

```
struct
{
  unsigned  int  flag1 :   1;
  unsigned  int  flag2 :   1;
  unsigned  int  flag3 :   2;
  unsigned  int  flag4 :   3;
} indicators;
```

www.TopSage.com

上述语句定义了 indicators 变量，它是匿名结构的一个实例，包含 4 个位字段，分别是 flagl~flag4。它们全部存储在一个字符组(word)中，如图 11-4 所示。

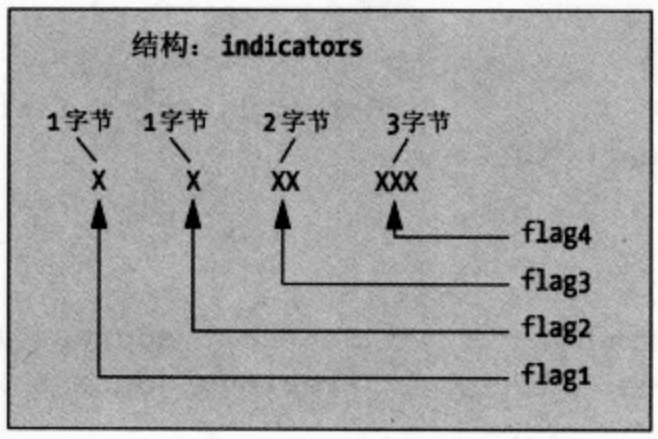


图 11-4　结构中的位字段

前两个位字段在定义中指定为 1，表示它们是一个位，其值是 0 或 1。第三个位字段 flag3 有两个位，其值是 0~3。最后一个 flag4 有三个位，其值是 0~7。引用这些位字段的方式和引用一般结构成员的方式相同。例如：

```
indicators.flag4  =  5;
indicators.flag3  =  indicators.flag1  =  1;
```

几乎没什么机会用到这个功能，这里介绍它只是为了讨论完整性，如果哪天缺乏内存，就可以考虑使用它。

11.3　结构与函数

结构表示 C 语言的一个强大特性，因此它与函数并用非常重要。现在探讨如何把结构当成变元传递给函数，以及如何从函数中返回结构。

11.3.1　结构作为函数的变元

将结构作为变元传给函数和传递一般变量没有什么不同。创建类似于 horse 的结构，如下：

```
struct  family
{
  char  name[20];
  int  age;
  char  father[20];
  char  mother[20];
};
```

然后，建立一个函数，检查两个 family 类型的成员是否为兄弟，如下：

```
bool  siblings(struct  family  member1, struct  family  member2)
```

www.TopSage.com

```
{
  if(strcmp(member1.mother, member2.mother) ==  0)
    return  true;
  else
    return  false;
}
```

这个函数有两个结构变元，该函数比较这两个结构中的 mother 成员。如果它们相同，就表示是兄弟，返回 true。否则就返回 false。这里忽略了离婚、人工受精和克隆等其他可能性。

11.3.2 结构指针作为函数变元

在调用函数时，传送给函数的是变元值的副本。如果变元是一个非常大的结构，就需要相当多的时间，并占用结构副本所需的内存。在这种情况下，应该使用结构指针作为变元。这可以避免占用内存，节省复制的时间，因为只需复制指针。函数可以通过指针直接访问原来的结构。另外，使用指针给函数传送结构，也提高了效率。重写 siblings() 函数，如下：

```
bool  siblings(struct  family  *member1, struct  family  *member2)
{
  if(strcmp(member1->mother, member2->mother) == 0)
    return  true;
  else
    return  false;
}
```

这有一个缺点。按值传递机制禁止在被调用的函数中意外地更改变元值。如果使用指针，就丧失了这个优点。而如果不需要更改指针变元的值(只是访问并使用它们)，把指针传送给函数还是可以获得某种程度的保护。

在上一个 siblings()函数中，不需要修改传给它的结构，它只是比较两个成员而已。因此可以重写它，如下所示：

```
bool siblings(struct family const *pmember1, struct family const *pmember2)
{
  if(strcmp(pmember1->mother,   pmember2->mother) == 0)
    return  true;
  else
    return  false;
}
```

本书在前面介绍了 const 修饰符，它用于将变量变成常量。这个函数声明将参数的类型指定为 family 结构的常量指针。这意味着，传递给函数的结构指针在函数中被视为常量。试图改变结构，会在编译期间产生错误信息。当然，这不会影响它们在调用程序中的状态，因为 const 关键字仅在执行 siblings()函数时应用于指针值。

www.TopSage.com

注意下面这个函数和前一个函数的差异：

```
bool siblings(struct family *const pmember1, struct family *const pmember2)
{
  if(strcmp(pmember1->mother,   pmember2->mother) == 0)
    return  true;
  else
    return  false;
}
```

每个参数定义中的间接运算符在关键字 const 的前面，而不是在指针名称的前面。这有什么差别吗？这里的参数是“指向 family 结构类型的常量指针”，而不是“指向常量结构的指针”，因此可以在函数中随意改变结构，但是不能改变存储在指针内的地址。因为这里保护的是指针，而不是指针指向的结构。

11.3.3 作为函数返回值的结构

函数返回结构和返回一般数值一样，只是在函数原型中，要以正常的方式指出函数返回的是结构，例如：

```
struct  horse  my_fun(void);
```

这个函数原型说明，它是一个没有变元的函数，返回 horse 类型的结构。

可以像这样从函数返回一个结构，但比较方便的做法是返回结构指针。下面在实例中探讨其细节。

试试看：返回结构指针

为了返回结构指针，可以重写前面的 horse 结构例子，把马换成人，并将输入部分放在一个函数中。

```
/* Program 11.6 Basics of a family tree */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

struct Family *get_person(void); /* Prototype for input function */

struct Date
{
  int day;
  int month;
  int year;
};

struct Family /* Family structure declaration */
{
  struct Date dob;
```

www.TopSage.com

```
  char name[20];
  char father[20];
  char mother[20];
  struct Family *next;            /* Pointer to next structure */
  struct Family *previous;        /* Pointer to previous structure */
};

int main(void)
{
  struct Family *first = NULL;    /* Pointer to first person */
  struct Family *current = NULL; /* Pointer to current person */
  struct Family *last = NULL;     /* Pointer to previous person */
  char more = '\0';               /* Test value for ending input */

  for( ; ; )
  {
    printf("\nDo you want to enter details of a%s person (Y or N)? ",
        first != NULL?"nother" : "");
    scanf(" %c", &more);
    if(tolower(more) == 'n')
      break;

    current = get_person();

    if(first == NULL)
    {
      first = current;            /* Set pointer to first Family */
      last = current;             /* Remember for next iteration */
    }
    else
    {
      last->next = current;       /* Set next address for previous Family */
      current->previous = last;   /* Set previous address for current */
      last = current;             /* Remember for next iteration */
    }
}

/* Now tell them what we know */

/* Output Family data in reverse order */
while (current != NULL)
{
  printf("\n%s was born %d/%d/%d, and has %s and %s as parents.",
    current->name, current->dob.day, current->dob.month,
    current->dob. year, current->father, current->mother );
  last = current; /* Save pointer to enable memory to be freed */
  current = current->previous; /* current points to previous list */
  free(last);     /* Free memory for the Family we output */
}
```

www.TopSage.com

```
  return 0;
}

/* Function to input data on Family members */
struct Family *get_person(void)
{
  struct Family *temp; /* Define temporary structure pointer */

  /* Allocate memory for a structure */
  temp = (struct Family*) malloc(sizeof(struct Family));

  printf("\nEnter the name of the person: ");
  scanf("%s", temp -> name ); /* Read the Family's name */

  printf("\nEnter %s's date of birth (day month year); ", temp->name);
  scanf("%d %d %d", &temp->dob.day, &temp->dob.month, &temp->dob.year);

  printf("\nWho is %s's father? ", temp->name );
  scanf("%s", temp->father ); /* Get the father's name */

  printf("\nWho is %s's mother? ", temp->name );
  scanf("%s", temp -> mother );  /* Get the mother's name */

  temp->next = temp->previous = NULL; /* Set pointers to NULL */

  return temp; /* Return address of Family structure */
}
```

代码的说明

代码很多，但很简单，执行的方式和前一个例子类似，只是用了两个函数，而不是一个函数。

第一个结构的声明如下：

```
struct  Date
{
  int  day;
  int  month;
  int  year;
);
```

用三个整数成员 day、month 和 year 来定义 Date 结构。目前这个结构还没有实例。这个定义放在源文件的所有函数之前，因此可以在文件的所有函数中访问。

下一个结构的声明如下：

```
struct Family /* Family structure declaration */
{
  struct Date dob;
  char name[20];
  char father[20];
```

www.TopSage.com

```
    char mother[20];
    struct Family *next;      /* Pointer to next structure */
    struct Family *previous; /* Pointer to previous structure */
};
```

上述语句定义了结构类型 Family，它的第一个成员是 Date 类型的结构。然后是三个 char 数组成员。最后两个成员是结构指针，分别指向表中的下一个结构和上一个结构，将该结构变成一个双向链表。

这和前一个例子的最大差别是，这两个结构都在所有函数的外部声明，因此是全局结构。这是必需的，因为 main()函数和 get_person()函数要定义 Family 结构的变量。

注意：

只有结构类型的声明可以全局访问，在每个函数内声明的 Family 类型变量的作用域是声明它们的函数。

函数 get_person()的原型如下：

```
struct Family *get_person(void); /* Prototype for input function */
```

它指出这个函数没有变元，但返回一个 Family 结构的指针。

其过程与程序 11.5 的操作相同，区别是使用了全局的结构类型声明，并将结构放在一个独立的函数中。

检查用户在 more 中的响应，确认用户要输入数据后，main()函数调用 get_person()函数。在函数 get_person()内声明如下指针：

```
temp = (struct Family*) malloc(sizeof(struct Family));
```

这是“Family 类型结构的指针”，并且是本地变量。结构类型的声明是全局性的，但它与结构实例的作用域无关。每个实例的作用域取决于其声明在程序中的位置。

函数 get_person()内的第一个动作是：

```
temp = (struct Family*) malloc(sizeof(struct Family));
```

调用 malloc()函数给 Family 类型的结构分配了足够的内存，并将返回的地址存储到指针变量 temp 中。temp 是本地变量，在 get_person()函数结束时，temp 就不存在了，但是 malloc()函数分配的内存是永久的，可以在程序的某个地方释放该内存，或在退出程序时释放它。

函数 get_person()会读取每个人的所有基本数据，并将它存储到 temp 所指的结构中。这个函数会接受任何数值的日期，但在实际情况下应该检查数据的有效性，例如要确认月份应是 1~12，日期值对于该月也应是有效的。由于输入的是出生日期，因此应验证该日期不是未来的某个日期。

get_person()函数的最后一行语句是：

```
return temp; /* Return address of Family structure */
```

www.TopSage.com

这行语句会返回结构指针的副本。虽然返回后 temp 就不存在了，但是它所指的内存地址仍然有效。

回到 main()函数中，返回的指针存储到 current 变量中。如果这是第一次迭代，该指针也会存储到 first 变量中。这么做的目的是不希望失去链表中第一个结构的记录。另外也将 current 指针存到 last 变量中，因此下一次迭代时，可以将它填入当前结构的指针成员 previous 中。

读完所有输入的数据后，程序以类似前一个例子的方式，以反向顺序将小结输出到屏幕上。

11.3.4 修改程序

下面创建一个例子，将结构指针作为变元和返回值。修改前一个例子(程序 11.6)，在 Family 类型的结构中声明一些额外的指针 p_to_pa 和 p_to_ma，如下所示：

```
struct Family                /* Family structure declaration */
{
  struct Date dob;
  char name[20];
  char father[20];
  char mother[20];
  struct Family *next;       /* Pointer to next structure */
  struct Family *previous;   /* Pointer to previous structure */
  struct Family *p_to_pa;    /* Pointer to father structure */
  struct Family *p_to_ma;    /* Pointer to mother structure */
};
```

现在可以在指针成员 p_to_pa 和 p_to_ma 中记录相关结构的地址了。必须在 get_person()函数的 return 语句之前添加如下语句，将它们设定成 NULL。

```
temp->p_to_pa = temp->p_to_ma = NULL; /* Set pointers to NULL */
```

现在可以扩展程序，使用两个额外的函数，在输入完每个人的数据后，给指针 p_to_pa 和 p_to_ma 填入适当的数值。第一个函数是 set_ancestry()，它把 Family 结构的指针作为变元，检查第二个变元是否为第一个变元的父亲或母亲。如果是，就更改相应的指针，以反映这个结果，然后返回 true。否则返回 false。代码如下：

```
bool set_ancestry(struct Family *pmember1, struct Family *pmember2)
{
  if(strcmp(pmember1->father, pmember2->name) == 0)
  {
    pmember1->p_to_pa = pmember2;
    return true;
  }
  if( strcmp(pmember1->mother, pmember2->name) == 0)
  {
    pmember1->p_to_ma = pmember2;
```

www.TopSage.com

```
    return true;
  }
  else
    return false;
}
```

第二个函数检查两个 Family 结构间的关系，如下：

```
/* Fill in pointers for mother or father relationships */
bool related (struct Family *pmember1, struct Family *pmember2)
{
  return set_ancestry(pmember1, pmember2) ||
      set_ancestry(pmember2, pmember1);
}
```

函数 related()在 return 语句中调用 set_ancestry()函数两次，以测试所有可能的关系。如果两次调用 set_ancestry()中有一次返回 true，related()函数的返回值就是 true。调用程序可以用这个返回值判断指针是否已经填入了数值。

注意：

这里使用了库函数 strcmp()，所以必须在程序的开头用#include 指令包含头文件<string.h>。还需要给<stdbool.h>添加#include 指令，因为还使用了 bool 类型和 true、false 值。

现在要给程序 11.6 中的 main()函数添加一些代码，以使用 related()函数，在包含有效地址的所有结构内填入指针。在输入初始数据的循环后，可以将以下代码插入 main()函数中：

```
current = first;

while(current->next != NULL) /* Check for relation for each person in */
{                            /* the list up to second to last */
  int parents = 0;           /* Declare parent count local to this block */
  last = current->next;      /* Get the pointer to the next */

  while(last != NULL) /* This loop tests current person */
  {                          /* against all the remainder in the list */
    if(related(current, last)) /* Found a parent ? */
      if(++parents == 2)       /* Yes, update count and check it */
        break;                 /* Exit inner loop if both parents found */

    last = last->next;         /* Get the address of the next */
  }
  current = current->next;   /* Next in the list to check */
}

/* Now tell them what we know etc. */
/* rest of output code etc. ...*/
```

这是一段相当独立的代码块，用来填入双亲的指针。它从第一个结构开始，检查后

www.TopSage.com

续的结构之间是否存在父母关系。如果找到两个双亲(说明两个指针中已填入数据)，或到达链表的尾部，就停止检查。

一些结构的指针值可能没有更改。因为这不是一个永无止尽的链表，可能有些人的双亲数据不齐全。所以需要遍历链表中的每个结构，将它与后续的所有结构比较，确定它们是否有关系，至少要到找到两个双亲的结构为止。

当然，还必须在程序的开头、函数 get_person()的原型后面插入 related()和 set_ancestry()的函数原型。它们的原型如下：

```
bool related(struct Family *pmember1, struct Family *pmember2);
bool set_ancestry(struct Family *pmember1, struct Family
                  *pmember2);
```

为了说明指针已成功插入，可以扩展最后的输出，在最后一个 printf()函数的后面加入一些语句，显示每个人的双亲信息。也可以修改输出循环，从 first 开始，所以输出循环如下所示：

```
/* Output Family data in correct order */
current = first;

while (current != NULL) /* Output Family data in correct order */
{
  printf("\n%s was born %d/%d/%d, and has %s and %s as parents.",
    current->name, current->dob.day, current->dob.month,
    current->dob. year, current->father, current->mother);
  if(current->p_to_pa != NULL )
    printf("\n\t%s's birth date is %d/%d/%d ",
           current->father, current->p_to_pa->dob.day,
           current->p_to_pa->dob.month,
           current->p_to_pa->dob.year);
  if(current->p_to_ma != NULL)
    printf("and %s's birth date is %d/%d/%d.\n ",
           current->mother, current->p_to_ma->dob.day,
           current->p_to_ma->dob.month,
           current->p_to_ma->dob.year);

  current = current->next; /* current points to next in list */
}
```

只有将双亲的结构指针设定成有效的地址，才能为每个人生成双亲的出生日期。不过，不能在循环内释放内存，否则，当双亲结构出现在链表的前面时，输出父母出生日期的语句就会生成垃圾值。因此在输出完毕后，必须在 main()函数的最后添加一个循环，以释放内存。

```
/* Now free the memory */
current = first;
while(current != NULL)
{
```

www.TopSage.com

```
  last = current; /* Save pointer to enable memory to be freed */
  current = current->next; /* current points to next in list */
  free(last);              /* Free memory for last */
}
```

把所有的片段集合成一个相当大的新程序。它的输出如下：

```
Do you want to enter details of a person (Y or N)? y

Enter the name of the person: Jack

Enter Jack's date of birth (day month year); 1 1 65

Who is Jack's father? Bill

Who is Jack's mother? Nell

Do you want to enter details of another person (Y or N)? y

Enter the name of the person: Mary

Enter Mary's date of birth (day month year); 3 3 67

Who is Mary's father? Bert

Who is Mary's mother? Moll

Do you want to enter details of another person (Y or N)? y

Enter the name of the person: Ben

Enter Ben's date of birth (day month year); 2 2 89

Who is Ben's father? Jack

Who is Ben's mother? Mary

Do you want to enter details of another person (Y or N)? n

Jack was born 1/1/65, and has Bill and Nell as parents.
Mary was born 3/3/67, and has Bert and Moll as parents.
Ben was born 2/2/89, and has Jack and Mary as parents.
Jack's birth date is 1/1/65 and Mary's birth date is 3/3/67.
```

可以修改这个程序，以日期的顺序输出每个人，或计算出每个人有多少后代。

11.3.5 二叉树

二叉树是组织数据的一种非常有效的方式，因为二叉树中的数据可以以有序的方式

www.TopSage.com

安排。二叉树也是一种非常有趣的机制，它在本质上非常简单。实现二叉树涉及到递归和动态分配内存，还要使用指针传送结构。

二叉树包含一系列相互关联的元素，称为节点。起始节点是树的根，称为根节点，如图 11-5 所示。

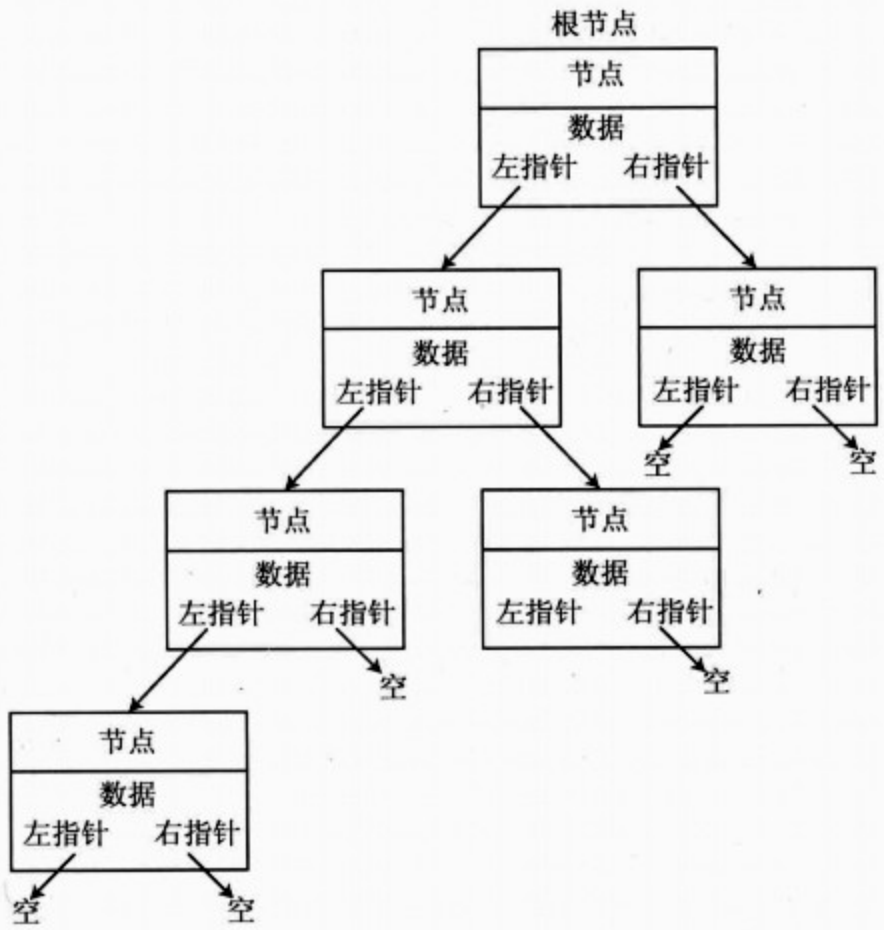


图 11-5 二叉树的结构

每个节点一般包含一个数据项，以及两个指向后续节点(左节点和右节点)的指针。如果有一个后续节点不存在，对应的指针就是 NULL。节点还可以包含一个计数器，记录树中何时有重复的数据项。

结构很容易表示二叉树的节点。下面的结构示例就定义了存储 long 类型整数的节点：

```
struct Node
{
  long item;            /* The data item */
  int count;            /* Number of copies of item */
  struct Node *pLeft;   /* Pointer to left node */
  struct Node *pRight;  /* Pointer to right node */
};
```

将数据项添加到二叉树中，且该项已存在于二叉树中时，就不创建新节点，而是把已有节点的 count 成员递增 1。本章的下一个程序示例 11.7 在创建二叉树时，就利用前面的结构定义表示一个包含整数的节点。

www.TopSage.com

1. 对二叉树中的数据排序

构建二叉树的方式确定了树中数据项的顺序。将一个数据项添加到二叉树中，需要比较要添加的项和树中已有的项。一般在添加数据项时，要求使左子节点的数据项小于当前节点的数据项，右子节点的数据项大于当前节点的数据项。图 11-6 中的示例二叉树包含随机顺序的整数。

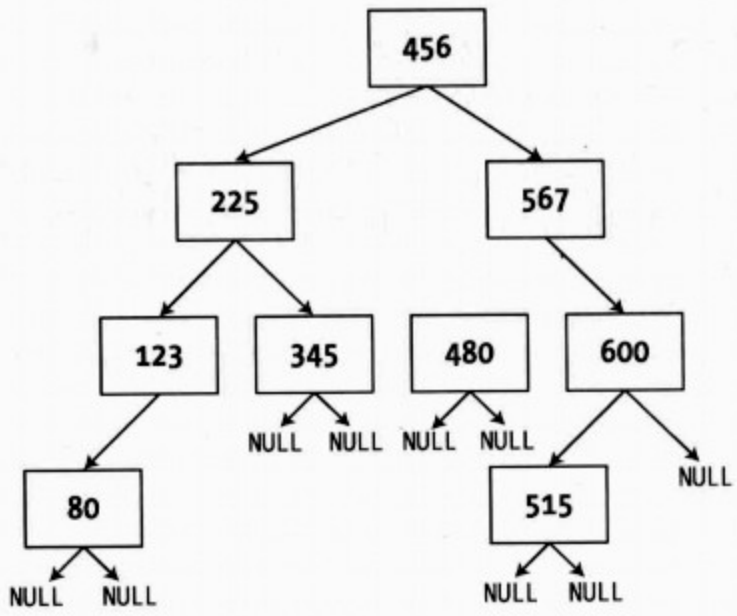


图 11-6 存储整数的二叉树

树的结构取决于数据项添加到树中的顺序。添加一个新项时，需要从树中的根节点开始，比较树中节点的值和新项。如果新项比给定的节点小，就查看给定节点的左子节点；反之，如果新项比给定的节点大，就查看给定节点的右节点。这个过程一直继续下去，直到找到一个与新项的值相同的节点为止，此时就更新这个节点的 count。如果到达一个左节点或右节点的指针为 NULL，就把新节点放在这里。

2. 构建二叉树

首先是创建根节点。所有节点的创建方式都相同，所以第一步是定义一个函数，从数据项中创建节点。假定创建一个存储整数的二叉树，可以使用前面的结构定义。下面是创建节点的函数定义：

```
struct Node *createnode(long value)
{
  /* Allocate memory for a new node */
  struct Node *pNode = (struct Node *)malloc(sizeof(struct Node));
  pNode->item = value;                    /* Set the value */
  pNode->count = 1;                       /* Set the count */
  pNode->pLeft = pNode->pRight = NULL; /* No left or right nodes */
  return pNode;
}
```

这个函数给新的 Node 结构分配内存，将 item 成员设置为 value。count 成员是节点中值的重复次数，所以对于第一个节点，这个 count 成员是 1。现在还有后续节点，所以

www.TopSage.com

pLeft 和 pRight 成员设置为 NULL。这个函数返回指向新建 Node 对象的指针。

要为新的二叉树创建根节点，可以使用这个函数，如下所示：

```
long newvalue;
printf("Enter the node value: ");
scanf(" %ld", newvalue);
struct Node *pRoot = createnode(newvalue);
```

从键盘上读取了要存储的值后，就调用 createnode()函数，在堆上创建一个新节点。当然，不要忘了使用完毕后释放节点的内存。

二叉树是应用递归的一个领域。插入节点的过程涉及到以相同的方式查看一系列节点，这是使用递归的一个强烈的暗示。用下面的函数在树中添加一个已有的节点：

```
/* Add a new node to the tree */
struct Node *addnode(long value, struct Node* pNode)
{
  if(pNode == NULL)              /* If there's no node */
    return createnode(value); /* …create one and return it */

  if(value ==pNode->item)
  {                              /* Value equals current node */
    ++pNode->count;              /* …so increment count and */
    return pNode;                /* …return the same node */
  }

  if(value < pNode->item)     /* If less than current node value */
  {
    if(pNode->pLeft == NULL)  /* and there's no left node */
    {
      pNode->pLeft = createnode(value); /* create a new left node and */
      return pNode->pLeft;              /* return it. */
    }
    else                           /* If there is a left node…*/
      return addnode(value, pNode->pLeft); /* add value via the left node */
  }
  else                             /* value is greater than current */
  {
    if(pNode->pRight == NULL) /* so the same process with */
    {                              /* the right node. */
      pNode-> pRight = createnode(value);
      return pNode-> pRight;
    }
    else
      return addnode(value, pNode-> pRight);
  }
}
```

第一次调用 addnode()函数时，它的变元是存储在树中的值和根节点的地址。如果将

www.TopSage.com

NULL 作为第二个变元传送，该函数就创建并返回一个新节点，所以也可以使用这个函数创建根节点。把根节点作为第二个变元传送时，有如下三种情况。

(1) 如果 value 等于当前节点的值，就不需要创建新节点，只递增当前节点中的计数器，并返回该节点。

(2) 如果 value 小于当前节点的值，就需要查看左子节点。如果左节点的指针是 NULL，就创建一个包含 value 的新节点，使之成为左子节点。如果左节点存在，就递归调用 addnode()函数，把指向左子节点的指针作为第二个变元。

(3) 如果 value 大于当前节点的值，就以与左节点相同的方式查看右节点。

无论调用递归函数时执行了什么，该函数都返回一个插入值的节点的指针。这可能是一个新节点，也可以是一个其值已存在于树中的节点。

下面的代码构建了一个完整的二叉树，以存储任意多个整数。

```
long newvalue = 0;
struct Node *pRoot = NULL;
char answer = 'n';
do
{
  printf("Enter the node value: ");
  scanf(" %ld", &newvalue);
  if(pRoot == NULL)
    pRoot = createnode(newvalue);
  else
    addnode(newvalue, pRoot);

  printf("\nDo you want to enter another (y or n)? ");
  scanf(" %c", &answer);
} while(tolower(answer) == 'y');
```

do-while 循环构建了一个包含根节点的完整的二叉树。在第一次迭代中，pRoot 是 NULL，所以创建根节点。所有后续的迭代给已有的树中添加节点。

3. 遍历二叉树

可以遍历二叉树，用升序或降序方式提取其内容。下面讨论如何以升序方式提取数据，读者可以以类似的方式获得降序排序的数据。从二叉树中提取数据初看上去是一个复杂的问题，因为二叉树的结构是随意的，但使用递归，这个问题就很简单了。

从很明显的地方开始：左子节点的值总是小于当前节点，当前节点的值总是小于右子节点，因此，提取值的顺序就应是左子节点、当前节点、右子节点。当然，如果子节点还有孙节点，也必须按“左子节点、当前节点、右子节点”的顺序处理。下面用一些代码来说明。

假定要以升序方式列出二叉树中的整数值。完成该操作的函数如下：

```
/* List the node values in ascending sequence */
void listnodes(struct Node *pNode)
```

www.TopSage.com

```
{
if(pNode->pLeft != NULL)
  listnodes(pNode->pLeft); /* List nodes in the left subtree */

for(int i = 0; i<pNode->count ; i++)
  printf("\n%10ld", pNode->item); /* Output the current node value */

if(pNode->pRight != NULL)
  listnodes(pNode->pRight); /* List nodes in the right subtree */
}
```

该函数包含如下三步：

(1) 如果存在左子节点，就为该节点递归调用 listnodes()函数。

(2) 输出当前节点的值。

(3) 如果存在右子节点，就为该节点递归调用 listnodes()函数。

如果根节点存在左子节点，就重复第 1 步，因此，输出左子树的所有信息后，再输出当前节点的值。当前节点的值在输出中重复 count 次，以表示该节点的重复次数。在输出当前节点的值后，输出根节点的整个右子树中的值。树中的每个节点都要进行这样的处理，所以值以升序方式输出。只需将根节点指针作为变元，调用 listnodes()函数即可，如下所示：

```
listnodes(pRoot); /* Output the contents of the tree */
```

这个简单的函数可以输出任意二叉树的所有整数值，下面探讨其工作过程。

试试看：用二叉树排序

这个示例将前面的代码合并起来，如下：

```
/* Program 11.7 Sorting integers using a binary tree */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

/* Function prototypes */
struct Node *createnode(long value); /* Create a tree node */
struct Node *addnode(long value, struct Node* pNode);
                                     /* Insert a new node */
void listnodes(struct Node *pNode);  /* List all nodes */
void freenodes(struct Node *pNode);  /* Release memory */

/* Defines a node in a binary tree sotring integers */
struct Node
{
  long item;            /* The data item */
  int count;            /* Number of copies of item */
  struct Node *pLeft;   /* Pointer to left node */
  struct Node *pRight;  /* Pointer to right node */
```

www.TopSage.com

```
};

/* Function main - execution starts here */
int main(void)
{
  long newvalue = 0;
  struct Node *pRoot = NULL;
  char answer = 'n';
  do
  {
    printf("Enter the node value: ");
    scanf(" %ld", &newvalue);
    if(pRoot == NULL)
       pRoot = createnode(newvalue);
    else
      addnode(newvalue, pRoot);
    printf("\nDo you want to enter another (y or n)? ");
    scanf(" %c", &answer);
  } while(tolower(answer) == 'y');

  printf("The values in ascending sequence are: ");
  listnodes(pRoot);    /* Output the contents of the tree */
  freenodes(pRoot);    /* Release the heap memory */

  return 0;
}

struct Node *createnode(long value)
{
  struct Node *pNode = (struct Node *)malloc(sizeof(struct Node));
  pNode->item = value;   /* Set the value */
  pNode->count = 1;      /* Set the count */
  pNode->pLeft = pNode->pRight = NULL; /* No left or right nodes */
  return pNode;
}

/* Add a new node to the tree */
struct Node *addnode(long value, struct Node* pNode)
{
  if(pNode == NULL)            /* If there's no node */
    return createnode(value); /* ...create one and return it */

  if(value ==pNode->item)
  { /* Value equals current node */
    ++pNode->count;            /* ···so increment count and */
    return pNode;              /* ···return the same node */
  }
  if(value < pNode->item)      /* If less than current node value */
  {
```

www.TopSage.com

```
    if(pNode->pLeft == NULL)                /* and there's no left node */
    {
      pNode->pLeft = createnode(value);     /* create a new left node and */
      return pNode->pLeft;                  /* return it. */
    }
    else                                    /* If there is a left node... */
      return addnode(value, pNode->pLeft); /* add value via the left node */
  }
  else                                      /* value is greater than current */
  {
    if(pNode->pRight == NULL) /* so the same process with */
    {                                       /* the right node. */
      pNode-> pRight = createnode(value);
      return pNode-> pRight;
    }
    else
      return addnode(value, pNode-> pRight);
  }
}

/* List the node values in ascending sequence */
void listnodes(struct Node *pNode)
{
  if(pNode->pLeft != NULL)
    listnodes(pNode->pLeft);

  for(int i = 0; i<pNode->count ; i++)
    printf("\n%10ld", pNode->item);

  if(pNode->pRight != NULL)
    listnodes(pNode->pRight);
}

/* Release memory allocated to nodes */
void freenodes(struct Node * pNode)
{
  if(pNode == NULL)             /* If there's no node... */
    return;                     /* we are done. */

  if(pNode->pLeft != NULL)      /* If there's a left sub-tree */
    freenodes(pNode->pLeft);    /* free memory for those nodes. */

  if(pNode->pRight != NULL)     /* If there's a right sub-tree */
    freenodes(pNode->pRight);   /* free memory for those nodes. */

  free(pNode);                  /* Free current node memory */
}
```

程序的输出如下:

```
Enter the node value: 56

Do you want to enter another (y or n)? y
Enter the node value: 33

Do you want to enter another (y or n)? y
Enter the node value: 77

Do you want to enter another (y or n)? y
Enter the node value: -10

Do you want to enter another (y or n)? y
Enter the node value: 100

Do you want to enter another (y or n)? y
Enter the node value: -5

Do you want to enter another (y or n)? y
Enter the node value: 200

Do you want to enter another (y or n)? n
The values in ascending sequence are:
    -10
     -5
     33
     56
     77
    100
    200
```

代码的说明

main()函数中的 do-while 循环利用前面讨论的方式从输入的值中构建出了二叉树。只要在提示时输入 y 或 Y，该循环就继续。调用 listnodes()函数时，将根节点的地址作为变元，就会以升序方式输出树中的所有值。接着，调用 freenodes()函数，释放为树中节点分配的内存。

freenodes()函数是本例中唯一的新东西。这是另一个递归函数，其工作方式类似于 listnodes()函数。本例在释放节点的内存之前，先删除每个节点的子节点的内存。因为一旦释放了内存块，其他程序就可以使用它了。也就是说，一旦释放了内存，子节点的地址就无效了。因此，在释放当前节点的内存之前，listnodes()函数总是为非空的子节点指针调用 freenodes()函数。

可以构建二叉树来存储任意类型的数据，包括结构对象和字符串。如果要在二叉树中组织字符串，就可以在每个节点中使用指针引用字符串，而无须复制树中的字符串。

www.TopSage.com

11.4 共享内存

前面讨论了如何使用位字段节省内存，这一般应用于逻辑变量。C 语言还有另一个功能，可以将几个变量放在相同的内存区。这个功能在内存短缺时比位字段应用得更广，因为实际上，我们常常使用几个变量，但其中只有一个变量在任意给定的时刻都有有效值。

多个变量共享内存的另一种情形是，程序处理许多不同类型的数据，但是一次只能处理一种，要处理的类型在执行期间确定。第三种可能是，要在不同的时间访问相同的数据，但在不同的情况下该数据的类型是不同的。例如对于一组数值类型的变量，要把它们当成 char 类型的数组，以便能将它们作为一块数据来移动。

11.4.1 联合

在 C 语言中允许在多个不同变量共享同一内存区的功能称为联合(union)。声明联合的语法类似于结构，给联合指定标记名称的方式通常也是类似的。定义联合要使用关键字 union。例如下面的语句声明一个联合被三个变量共享。

```
union  u_example
{
  float  decval;
  int  *pnum;
  double  my_value;
}U1;
```

上述语句用标记符名称 u_example 声明一个联合，它由浮点值 decval、整数指针 pnum 和双精度浮点变量 my_value 共享。该语句定义了一个联合的实例，即变量 U1。也可以用下面的语句声明这个联合的其他实例：

```
union  u-example  U2, U3;
```

联合成员的访问方式和结构成员完全相同。例如，要指定 U1 和 U2 成员的值，可以编写：

```
U1.decval = 2.5;
U2.decval = 3.5  *  U1.decval;
```

试试看：使用联合

下面是一个使用联合的例子：

```
/* Program 11.8 The operation of a union */
#include <stdio.h>

int main(void)
{
  union u_example
```

www.TopSage.com

```
{
  float decval;
  int pnum;
  double my_value;
} U1;

U1.my_value = 125.5;
U1.pnum = 10;
U1.decval = 1000.5f;
printf("\ndecval = %f pnum = %d my_value = %lf",
       U1.decval, U1.pnum, U1.my_value );
printf("\nU1 size = %d\ndecval size = %d pnum size = %d my_value"
       " size = %d",sizeof U1, sizeof U1.decval,
       sizeof U1.pnum, sizeof U1.my_value);
return 0;
}
```

代码的说明

这个例子示范了联合的结构和基本操作。U1 联合的声明如下:

```
union u_example
{
  float decval;
  int pnum;
  double my_value;
}U1;
```

联合的这三个成员的类型是不同的，它们需要的存储空间也不同(假设编译器给 int 类型指定两个字节)。

在赋值语句中，给联合实例 U1 的每个成员赋值如下:

```
U1.my_value = 125.5;
U1.pnum = 10;
U1.decval = 1000.5f;
```

注意，引用联合成员的方式和引用结构成员的方法相同。

下面的两行语句输出这三个成员的值，联合 U1 占用的字节数及每个成员占用的字节数。输出如下所示(如果机器给 int 类型的变量指定 4 个字节，输出就与此类似):

```
decval = 1000.500000 pnum = 8192 my_value = 125.50016
U1 slze = 8
decval slze = 4 pnum size = 2 my_value size = 8
```

首先要注意，只有最后一个变量的值是正确的，其他两个都是错的。这在意料之中，因为它们共享同一块内存空间。其次，my_value 成员没有被毁坏。这是因为只修改了 my_value 中最不重要的部分。正常情况下，这么小的错误很容易被忽略掉，但最后的结果可能很糟。在使用联合时，要特别注意不要使用无效的数据。

www.TopSage.com

注意：
从输出可知，联合所占的字节数是其最大的成员所占的空间。

11.4.2 联合指针

也可以用下列语句定义联合指针：

```
union  u_example  *pU;
```

有了指针后，就可以修改联合的成员了。如下列语句：

```
pU  =  &U2;
U1.decval  =  pU->decval;
```

第二行赋值语句中，等号右边的表达式 pU - >decval 等于 U2.decval。

11.4.3 联合的初始化

声明联合时，若需要初始化联合的实例，只能用和联合中第一个变量相同类型的常量初始化。以 u_example 为例，只能用 float 常量去初始化，如下所示：

```
union  u_example  U4  =  3.14f;
```

可以重新安排联合中成员的顺序，将要初始化的成员作为第一个成员。联合中成员的顺序并不重要，因为它们都重叠在同一个内存区中。

11.4.4 联合中的结构成员

结构和数组可以是联合的成员。反之，联合也可以是结构的成员。例如：

```
struct  my_structure
{
  int  num1;
  float  num2;
  union
  {
    int  *pnum;
    float  *pfnum;
  }  my_U;
}  samples[5];
```

这里声明了一个结构类型 my_structure，它包含一个没有标记符名称的联合，所以这个联合的实例只能存在于结构的实例之中。这种联合常常称为匿名联合。上面的语句还定义了一个包含 5 个结构实例的数组 samples。结构中的联合由两个指针共享。要使用联合成员，其表示法和嵌套的结构相同，例如，要访问结构数组中第三个元素的 int 指针，要使用如下语句中左边的表达式：

www.TopSage.com

```
samples[2].my_U.pnum = &my_num;
```

首先，假设变量 my_num 已声明为 int 类型。使用存储在联合中的值时，总是会提取上次赋予联合的值。这似乎很明显，但实际上，很容易将最近存储为整数的值当成 float，有时错误很微小，例如程序 11.7 输出的 my_value 值。自然，这样通常会得到垃圾值。常常采用的一种方法是将联合嵌入结构中，该联合也有一个成员，指定在联合中当前存储的值的类型。例如：

```
/* Type code for data in union */
#define TYPE_LONG   1
#define TYPE_FLOAT  2
#define TYPE_CHAR   3

struct Item
{
  int u_type;
  union
  {
    long integer;
    float floating;
    char ch;
  } u;
} var;
```

这段代码定义了 Item 结构类型，它包含两个成员：int 类型的值 u_type 和匿名联合的一个实例 u。这个联合可以存储 long 类型、float 类型或 char 类型的值，u_type 成员用于记录当前存储在 u 中的类型。

为 var 设置一个值：

```
var.u.floating = 2.5f;
var.u_type = TYPE_FLOAT;
```

在处理 var 时，需要检查它存储了什么类型的值。下面的例子说明了具体的做法：

```
switch(var.u_type)
{
  case TYPE_FLOAT:
    printf("\nValue of var is %10f", var.u.floating);
    break;
  case TYPE_LONG:
    printf("\nValue of var is %10ld", var.u.integer);
    break;
  case TYPE_CHAR:
    printf("\nValue of var is %10c", var.u.ch);
    break;
  default:
    printf("\nInvalid union type code.");
    break;
```

www.TopSage.com

```
}
```

使用联合，主要是为了便于把这样的代码放在函数中。

11.5 定义自己的数据类型

使用结构非常类似定义自己的数据类型。其实这并不完全正确，因为声明结构变量必须使用 struct 关键字。而声明内置类型的变量要容易一些。但是 C 语言的一个特性允许使用内置类型的语法来定义结构类型的变量。这个特性可以用于简化派生于内置类型的类型，在这里使用结构定义自己的数据类型。

11.5.1 结构与类型定义(typedef)功能

假设有一个结构用三个坐标 x、y、z 表示一个点。它的定义如下：

```
struct  pts
{
  int  x;
  int  y;
  int  z;
};
```

现在用关键字 typedef 声明这个结构，如下：

```
typedef  struct  pts  Point;
```

这条语句指定，名称 Point 是 struct pts 的同义字。声明 pts 结构的一些实例时，可以用下列语句：

```
Point  start_pt;
Point  end_pt;
```

这里声明了两个结构变量 start_pt 和 end_pt。声明结构变量不再需要关键字 struct 了。声明的方式与 float 或 int 变量一样。可以组合 typedef 和结构声明，如下：

```
typedef  struct  pts
{
  int  x;
  int  y;
  int  Z;
} Point;
```

不要把上述语句与基本的结构声明混淆。Point 不是结构变量名称，而是类型的名称。声明结构变量时，可以使用以下语句：

```
Point  my_pt;
```

www.TopSage.com

对一个结构类型，可以定义几种类型，但这在某些情况下会造成混淆。typedef 的一种有助于理解程序的应用是，对某个数值使用基本的类型，再使用类型名称，以反映变量的类型。例如，假设程序涉及到不同种类的重量，如部件的重量和装配体的重量，此时应把类型名称 weight 定义为 double 的同义词。语句如下：

```
typedef  double  weight;
```

现在就可以声明 weight 类型的变量：

```
weight  piston  =  6.5;
weight  valve  =   0.35;
```

当然，这些变量都是 double 类型，因为 weight 只是 double 的同义字。还可以像平常一样声明 double 类型的变量。

11.5.2 使用 typedef 简化代码

typedef 的另一个应用是简化复杂的类型。假设要经常定义结构指针 pts，就可以定义一个类型，如下：

```
typedef  struct  pts  *pPoint;
```

现在声明某些指针，可以编写：

```
pPoint  pfirst;
pPoint  plast;
```

这两个变量都声明为 pts 类型的结构指针。这样的声明可以减少编写错误，也易于了解。

在第 9 章中讨论过函数指针，它的声明方式更复杂。下面是一个声明函数指针的例子：

```
int(*pfun)(int, int); /* Function pointer declaration */
```

如果期望在程序中使用几个这类函数指针，可以使用 typedef 为该声明定义一般类型，如下：

```
typedef int (*function_pointer)(int, int); /* Function pointer type */
```

这不是声明函数指针的变量，而是将 function_pointer 声明为类型名称，以用来声明函数指针。所以，可以用下面的语句取代原来 pfun 的声明：

```
function_pointer  pfun;
```

代码明显简化了。如果要声明几个这样的指针，简化的优点就更突出了。下面的语句声明了三个函数指针：

```
function_pointer  pfun1;
function_pointer  pfun2;
function_pointer  pfun3;
```

www.TopSage.com

当然，也可以初始化它们。假设有函数 sum()、difference()及 product()，可以声明并初始化它们，如下：

```
function_pointer  pfun1  =   sum;
function_pointer  pfun2  =  difference;
function_pointer  pfun3  =  product;
```

这里定义的类型名称只能应用于在 typedef 语句中指定的那种类型的函数。如果要应用于其他函数，就必须定义另一个类型。

11.6 设计程序

在本章的最后，通过以下的案例实践本章学到的知识。

11.6.1 问题

数值数据用图表表示通常更容易理解。现在要处理的问题是编写一个程序，从一组数值中生成柱状图。选择柱状图的理由有如下三个：

(1) 实践结构的用法。

(2) 了解如何在有效的空间中放置并显示柱状图。

(3) 柱状图在实际应用中很常见。

11.6.2 分析

无须对纸张大小、列数甚至图的比例作任何假设。只需编写一个函数，它将纸张大小作为参数，使柱状图能放在该纸张上。这可以使函数适用于任何的情况。我们将数值存放在链表的一系列结构里。这样，就只需将第一个结构传递给函数，函数就能够从链表中得到所有的结构。这个结构非常简单，但以后可以用自己设计的信息去修饰它。

假定柱状图中的竖条显示顺序和数据输入的顺序相同，因此无须排序数据。这个程序有两个函数：生成柱状图的函数和 main()函数，用来练习柱状图生成过程。

以下是所需的步骤：

(1) 编写柱状图函数。

(2) 编写 main()函数，测试柱状图函数。

11.6.3 解决方案

本节列出了解决问题的步骤。

www.TopSage.com

1. 步骤 1

很明显，这个程序将使用结构，因为这是本章讨论的主题。第一步是设计程序要使用的结构。这里将使用 typedef，以避免重复使用关键字 struct。

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>

typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;

int main(void)
{
 /* Code for main */
}

/* Definition of the bar-chart function */
```

barTAG 结构用它的值定义一个竖条。注意将该结构中的指针定义为指向下一个结构。这样就可以把竖条存储为链表，其优点是在分配内存时不会浪费内存。链表适合于本例，因为我们只想按顺序遍历所有的竖条，从第一个到最后一个。接着按输入值的顺序创建竖条，将新建的竖条追加到前一个竖条之后。然后遍历链表中的结构，生成柱状图的可视化表示。

这里必须使用 struct barTAG 定义结构，而不能使用类型名称 bar，因为此时编译器还没有完成 typedef 的处理，所以 bar 还未定义。换句话说，编译器先分析 barTAG 结构，再利用 typedef 定义 bar 的意义。

现在为柱状图函数指定函数原型，给这个函数定义添加框架。它需要的参数有指向链表中第一个竖条的指针、页高、页宽及图表的标题。

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>

#define PAGE_HEIGHT 20
#define PAGE_WIDTH 40

typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;

typedef unsigned int uint; /* Type definition*/
```

www.TopSage.com

```
/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width,
              uint page_height, char *title);

int main(void)
{
  /* Code for main */
}

int bar_chart(bar *pfirstbar, uint page_width,
              uint page_height,char *title)
{
  /* Code for function…*/
  return 0;
}
```

添加一条 typedef 语句，将 uint 定义为 unsigned int 的同义词。这样可以缩短用 unsigned int 声明变量的语句长度。

接下来为柱状图需要的基本数据添加一些声明和代码。需要竖条的最大和最小值、图表的垂直高度(即最大值和最小值之差)。另外，需要根据纸张的宽度及竖条的数量计算竖条的宽，并调整高度，以包含水平轴和标题。

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40

typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;

typedef unsigned int uint; /* Type definition */

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width,
              uint page_height, char *title);

int main(void)
{
  /* Code for main */
}

int bar_chart(bar *pfirstbar, uint page_width,
              uint page_height,char *title)
{
  bar *plastbar = pfirstbar; /* Pointer to previous bar */
```

www.TopSage.com

```
double max = 0.0;              /* Maximum bar value */
double min = 0.0;              /* Minimum bar value */
double vert_scale = 0.0;       /* Unit step in vertical direction */
uint bar_count = 1;            /* Number of bars - at least 1 */
uint barwidth = 0;             /* Width of a bar */
uint space = 2;                /* spaces between bars */

/* Find maximum and minimum of all bar values */

/* Set max and min to first bar value */
max = min = plastbar->value;

while((plastbar = plastbar->pnextbar) != NULL)
{
  bar_count++; /* Increment bar count */
  max = (max < plastbar->value)? plastbar->value : max;
  min = (min > plastbar->value)? plastbar->value : min;
}
vert_scale = (max - min)/page_height; /* Calculate step length */

/* Check bar width */
if((barwidth = page_width/bar_count - space) < 1)
{
  printf("\nPage width too narrow.\n");
  return -1;
}

  /* Code for rest of the function…*/
  return 0;
}
```

space 变量存放两个竖条间的空格数，初值设为 2。柱状图在输出时是一次显示一行，因此需要一个字符串来对应一行一行地绘制该竖条时所使用的区段，还需要另一个相同长度的字符串，包含页面上特定位置没有竖条时所使用的空格数。下面添加如下代码：

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>
#include <stdlib.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40

typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;
```

www.TopSage.com

```
typedef unsigned int uint; /* Type definition */

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width,
              uint page_height, char *title);

int main(void)
{
  /* Code for main */
}

int bar_chart(bar *pfirstbar, uint page_width,
              uint page_height,  char *title)
{
  bar *plastbar = pfirstbar; /* Pointer to previous bar */
  double max = 0.0;          /* Maximum bar value */
  double min = 0.0;          /* Minimum bar value */
  double vert_scale = 0.0;   /* Unit step in vertical direction */
  uint bar_count = 1;        /* Number of bars - at least 1 */
  uint barwidth = 0;         /* Width of a bar */
  uint space = 2;            /* spaces between bars */
  uint i = 0;                /* Loop counter */
  char *column = NULL;       /* Pointer to bar column section */
  char *blank = NULL;        /* Blank string for bar+space */

  /* Find maximum and minimum of all bar values */

/* Set max and min to first bar value */
max = min = plastbar->value;

while((plastbar = plastbar->pnextbar) != NULL)
{
  bar_count++; /* Increment bar count */
  max = (max < plastbar->value)? plastbar->value : max;
  min = (min > plastbar->value)? plastbar->value : min;
}
vert_scale = (max - min)/page_height; /* Calculate step length */

/* Check bar width */
if((barwidth = page_width/bar_count - space) < 1)
{
  printf("\nPage width too narrow.\n");
  return -1;
}

/* Set up a string that will be used to build the columns */

/* Get the memory */
if((column = malloc(barwidth + space + 1)) == NULL)
```

www.TopSage.com

```
{
  printf("\nFailed to allocate memory in barchart()"
         " - terminating program.\n");
  exit(1);
}
for(i = 0 ; i<space ; i++)
  *(column+i)=' '; /* Blank the space between bars */
for( ; i<space+barwidth ; i++)
  *(column+i)='#'; /* Enter the bar characters */
*(column+i) = '\0'; /* Add string terminator */

/* Set up a string that will be used as a blank column */

/* Get the memory */
if((blank = malloc(barwidth + space + 1)) == NULL)
{
  printf("\nFailed to allocate memory in barchart()"
       " - terminating program.\n");
  exit(1);
}

for(i = 0 ; i<space+barwidth ; i++)
  *(blank+i) = ' '; /* Blank total width of bar+space */
*(blank+i) = '\0';  /* Add string terminator */

/* Code for rest of the function···*/
free(blank);        /* Free memory for blank string */
free(column);       /* Free memory for column string */
return 0;
}
```

这里用字符'#'绘制竖条。画竖条时，要先编写一个字符串，使之含有 space 个空格和 barwidth 个'#'字符。之后使用库函数 malloc()为它动态分配内存，因此必须给头文件<stdlib.h>添加#include 指令。用来画竖条的字符串是 column，blank 是包含空格的且长度与 column 相同的字符串。画完柱状图后，在退出之前要释放 column 和 blank 所占的内存。

接下来，添加画竖条图的最后一段代码：

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40

typedef struct barTAG
{
```

www.TopSage.com

```
  double value;
  struct barTAG *pnextbar;
}bar;

typedef unsigned int uint; /* Type definition */

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width,
              uint page_height, char *title);

int main(void)
{
  /* Code for main */
}

int bar_chart(bar *pfirstbar, uint page_width, ,
      uint page_height char *title)
{
  bar *plastbar = pfirstbar;   /* Pointer to previous bar */
  double max = 0.0;            /* Maximum bar value */
  double min = 0.0;            /* Minimum bar value */
  double vert_scale = 0.0;     /* Unit step in vertical direction */
  double position = 0.0;       /* Current vertical position on chart */
  uint bar_count = 1;          /* Number of bars - at least 1 */
  uint barwidth = 0;           /* Width of a bar */
  uint space = 2;              /* spaces between bars */
  uint i = 0;                  /* Loop counter */
  uint bars = 0;               /* Loop counter through bars */
  char *column = NULL;         /* Pointer to bar column section */
  char *blank = NULL;          /* Blank string for bar+space */
  bool axis = false;           /* Indicates axis drawn */

  /* Find maximum and minimum of all bar values */

  /* Set max and min to first bar value */
  max = min = plastbar->value;
  while((plastbar = plastbar->pnextbar) != NULL)
  {
    bar_count++; /* Increment bar count */
    max = (max < plastbar->value)? plastbar->value : max;
    min = (min > plastbar->value)? plastbar->value : min;
  }
  vert_scale = (max - min)/page_height; /* Calculate step length */

  /* Check bar width */
  if((barwidth = page_width/bar_count - space) < 1)
  {
    printf("\nPage width too narrow.\n");
    return -1;
```

www.TopSage.com

```
}

/* Set up a string that will be used to build the columns */

/* Get the memory */
if((column = malloc(barwidth + space + 1)) == NULL)
{
  printf("\nFailed to allocate memory in barchart()"
         " - terminating program.\n");
 exit(1);
}

for(i = 0 ; i<space ; i++)
  *(column+i)=' '; /* Blank the space between bars */
for( ; i < space+barwidth ; i++)
  *(column+i)='#'; /* Enter the bar characters */
*(column+i) = '\0'; /* Add string terminator */

/* Set up a string that will be used as a blank column */

/* Get the memory */
if((blank = malloc(barwidth + space + 1)) == NULL)
{
  printf("\nFailed to allocate memory in barchart()"
         " - terminating program.\n");
  exit(1);
}

for(i = 0 ; i<space+barwidth ; i++)
  *(blank+i) = ' ';        /* Blank total width of bar+space */
*(blank+i) = '\0';         /* Add string terminator */

printf("^ %s\n", title); /* Output the chart title */
/* Draw the bar chart */
position = max;
for(i = 0 ; i <= page_height ; i++)
{
  /* Check if we need to output the horizontal axis */
  if(position <= 0.0 && !axis)
  {
    printf("+");          /* Start of horizontal axis */
    for(bars = 0; bars < bar_count*(barwidth+space); bars++)
      printf("-");        /* Output horizontal axis */
    printf(">\n");
    axis = true;          /* Axis was drawn */
    position -= vert_scale;/* Decrement position */
    continue;
  }
  printf("|");            /* Output vertical axis */
```

www.TopSage.com

```
    plastbar = pfirstbar; /* start with the first bar */

    /* For each bar…*/
    for(bars = 1; bars <= bar_count; bars++)
    {
      /* If position is between axis and value, output column */
      /* otherwise output blank */
      printf("%s", position <= plastbar->value &&
                   plastbar->value >= 0.0 && position > 0.0 ||
                   position >= plastbar->value &&
                   plastbar->value <= 0.0 &&
                   position <= 0.0 ? column: blank);
      plastbar = plastbar->pnextbar;
    }
    printf("\n");              /* End the line of output */
    position -= vert_scale; /* Decrement position */
  }
  if(!axis) /* Have we output the horizontal axis? */
  {                           /* No, so do it now */
    printf("+");
    for(bars = 0; bars < bar_count*(barwidth+space); bars++)
      printf("-");
    printf(">\n");
  }

  free(blank);      /* Free memory for blank string */
  free(column);     /* Free memory for column string */
  return 0;
}
```

for 循环会输出 page_height 行字符。每行表示垂直轴上一段由 vert_scale 代表的距离。这个值是 page_height 除以最大值和最小值之差得到的。因此，第一行输出对应的 position 是 max，在每次迭代中，position 都会递减 vert_scale，直到到达 min 为止。

在输出每一行之前，都要先确定是否要输出水平轴。当 position 小于或等于 0，且尚未显示水平轴时，就需要输出水平轴。

除了水平轴之外，还必须确定每个竖条位置显示什么内容。这是为每个竖条执行的内层 for 循环内决定的。printf()函数内的条件运算符会选择输出 column 或 blank。如果 position 的值介于竖条最大值和 0 之间，则输出 column，否则输出 blank。在输出完整的一行后，输出'\n'以结束这行，并递增 position 的值。

所有的竖条可能都是正的，此时需要确保在循环结束后输出水平轴，因为它在循环内没有输出。

2. 步骤 2

现在需要实现 main()函数，调用 bar_chart()函数：

www.TopSage.com

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define PAGE_HEIGHT 20
#define PAGE_WIDTH  40

typedef struct barTAG       /* Bar structure */
{
  double value;             /* Value of bar */
  struct barTAG *pnextbar;  /* Pointer to next bar */
}bar;                       /* Type for a bar */

typedef unsigned int uint;  /* Type definition */

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width,
              uint page_height, char *title);

int main()
{
  bar firstbar;             /* First bar structure */
  bar *plastbar = NULL;     /* Pointer to last bar */
  char value[80];           /* Input buffer */
  char title[80];           /* Chart title */

  printf("\nEnter the chart title: ");
  gets(title);              /* Read chart title */

  for( ;; )                 /* Loop for bar input */
  {
    printf("Enter the value of the bar, or use quit to end: ");
    gets(value);

    if(strcmp(value, "quit") == 0)  /* quit entered? */
      break;                        /* then input finished */

    /* Store in next bar */
    if(!plastbar)                   /* First time? */
    {
      firstbar.pnextbar = NULL; /* Initialize next pointer */
      plastbar = &firstbar;     /* Use the first */
    }
    else
    {
      /* Get memory */
```

www.TopSage.com

```
      if(!(plastbar-> = malloc(sizeof(bar))))
      {
        printf("Oops! Couldn't allocate memory\n");
        return -1;
      }
      plastbar = plastbar->pnextbar; /* Old next is new bar */
      plastbar->pnextbar = NULL;     /* New bar next is NULL */
      }
    plastbar->value = atof(value); /* Store the value */
  }

  /* Create bar-chart */
  bar_chart(&firstbar, PAGE_WIDTH, PAGE_HEIGHT, title);

  /* We are done, so release all the memory we allocated */
  while(firstbar.pnextbar)
  {
    plastbar = firstbar.pnextbar;           /* Save pointer to next */
    firstbar.pnextbar = plastbar->pnextbar; /* Get one after next */
    free(plastbar); /* Free next memory */
  }
  return 0;
}

int bar_chart(bar *pfirstbar, uint page_width, uint page_height, char *title)
{
  /* Implementation of function as before…*/
}
```

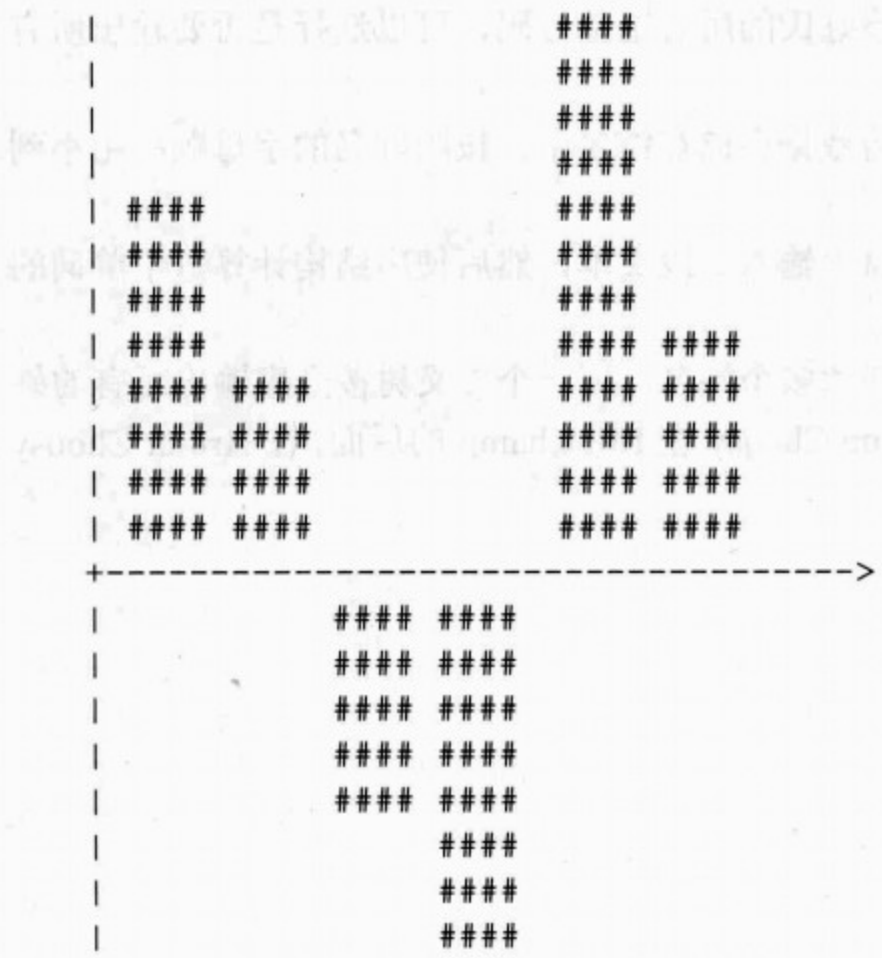
使用 gets()函数读取柱状图的标题后，就在 for 循环内读入连续的数值。除了第一个值外，其他值都要用于给新的竖条结构分配内存，之后存储该值。当然，可以跟踪第一个结构，因为它链接了其他结构，跟踪上一个新增结构中的指针，在添加下一个结构时更新它的指针 pnextbar。输入了所有的值后，就调用 bar_chart()函数，生成柱状图。最后，释放竖条结构的内存。注意，不能删除 firstbar，因为它的内存不是动态分配的。给头文件<string.h>包含#include 指令，因为使用了 gets()函数。

之后在 main()函数中添加一行代码，根据输入的值生成柱状图。程序的输出如下：

```
Enter the chart title: Trial Bar Chart
Enter the value of the bar, or use quit to end: 6
Enter the value of the bar, or use quit to end: 3
Enter the value of the bar, or use quit to end: -5
Enter the value of the bar, or use quit to end: -7
Enter the value of the bar, or use quit to end: 9
Enter the value of the bar, or use quit to end: 4
Enter the value of the bar, or use quit to end: quit

^ Trial Bar Chart
```

www.TopSage.com



11.7 小结

本章很长，但其主题很重要。如果想高效地使用 C 语言，就必须熟练掌握结构，了解指针与函数的重要性。

许多实际的应用程序主要处理的是人、车或物料，这些都需要用几个不同的值来表示。而 C 语言中的结构是处理这类复杂对象的最佳工具。虽然某些操作看起来相当复杂，但用于处理的是复杂的实体，所以复杂的不是编程本身，而是要解决的问题。

第 12 章将介绍如何将数据存储到外部文件中。当然这包括存储到结构中。

11.8 习题

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容。还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 11.1　定义一个结构类型 Length，它用码、英尺及英寸表示长度。定义一个 add()函数，它相加两个 Length 变元，返回 Length 类型的总和。定义第二个函数 show()，显示其 Length 变元的值。编写一个程序，从键盘输入任意个单位是码、英尺以及英寸的长度，使用 Length 类型、add()和 show()函数去汇总这些长度，并输出总长。

习题 11.2　定义一个结构类型，它含有一个人的姓名及电话号码。在程序中使用这个结构，输入一个或多个姓名及对应的电话号码，将输入的数据项存储在一个结构数组

www.TopSage.com

中。程序允许输入姓氏，输出对应于该姓氏的所有电话号码，可以选择是否要输出所有的姓名及他们的电话号码。

习题 11.3　修改上一题的程序，将数据存储在链表中，按照姓名的字母顺序由小到大排序。

习题 11.4　编写一个程序，从键盘上输入一段文本，然后使用结构计算每个单词的出现次数。

习题 11.5　编写一个程序，读取任意多个姓名。用一个二叉树按升序输出所有的姓名，排序时先排姓氏，后排名字(例如 Ann Choosy 在 Bill Champ 的后面，在 Arthur Choosy 的前面)。

www.TopSage.com

第 12 章

处 理 文 件

如果计算机只能处理存储在主内存中的数据，则应用程序的适用范围和多样性就会受到相当大的限制。事实上，所有重要的商业应用程序所需的数据量远远大于主内存所能提供的数据量，常常需要具备处理外部设备(例如固定磁盘)所存储的数据的能力。本章将了解如何处理外部设备上的文件数据。

C 语言在头文件<stdio.h>中提供了一系列读写外部设备的函数。用于存储和检索数据的外部设备一般是固定磁盘，但不仅仅是固定磁盘。而 C 语言中用于处理文件的库函数都独立于设备，所以它们可以应用到任何外部存储设备上。而本章的例子假定处理的是磁盘文件。

本章的主要内容：

- C 语言中的文件
- 如何处理文件
- 如何读写格式化文件及二进制文件
- 如何在文件中直接随机存取数据
- 如何在程序中使用临时文件
- 如何更新二进制文件
- 如何编写文件查看器程序

12.1 文件的概念

在前面的所有例子中，用户在执行程序时输入的任何数据，在程序结束后都会消失。此时如果用户要用相同的数据执行程序，就必须重新输入一遍。这种方式不仅不方便，还使编程任务无法完成。

例如，如果程序要维护一组姓名、地址以及电话号码，但每次执行时都必须输入一遍姓名、地址以及电话号码，这个程序就不会有人愿意使用了。解决方法是将这些数据存储到一个即使关掉计算机后数据也不会消失的存储设备中。该存储设备就叫做文件，文件通常存储到硬盘上。

如果读者对硬盘的基本工作机制略知一二，这些知识将有助于了解何时适合使用文件，何时不适合使用文件。不过，就算对硬盘的文件机制没有任何概念，也不用担心。因为 C 语言的文件处理概念与物理存储设备的知识完全无关。

文件其实是一系列的字节，如图 12-1 所示。

www.TopSage.com

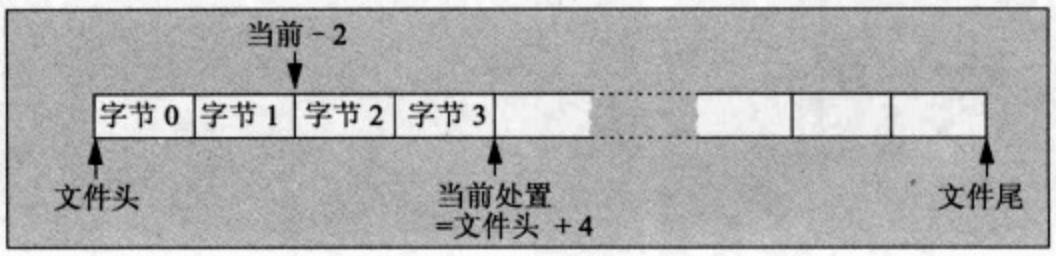


图 12-1　文件的结构

12.1.1　文件中的位置

文件有开头和结尾，还有一个当前位置，通常定义为从文件头到当前位置有多少个字节数，如图 12-1 所示。当前位置就是发生文件操作(读写文件的动作)的地方。当前位置可以移动到文件的其他地方去。新的当前位置可以指定为距离文件开头的偏移量，或在某些情况下，指定为从前一个当前位置算起的正或负偏移量。

12.1.2　文件流

C 库提供了读写数据流的函数。流是外部数据源或数据目的地的抽象表示，所以键盘、显示器上的命令行和磁盘文件都是流。因此，可以使用输入输出函数读写映射为流的任意外部设备。

将数据写入流(即磁盘文件)有两种方式。首先，可以将数据写入文本文件，此时数据写入为字符，这些字符组织为数据行，每一行都用换行符结束。显然，二进制数据，例如 int 或 double 类型的值，必须先转换为字符，才能写入文本文件。前面介绍了如何使用 printf()函数完成这个格式化。其次，可以将数据写入二进制文件。写入二进制文件的数据总是写入为一系列字节，与它在内存中的表示形式相同，所以 double 类型的值就写入为 8 个字节，与其内存表示形式相同。

当然。可以将任意数据写入文件，但数据一旦写入文件，磁盘上的文件都只包含一系列字节。无论是将数据写入二进制文件还是写入文本文件，不论它们是什么样的数据，这些数据最终都是一系列字节。也就是说，读取文件时，程序必须知道这个文件包含什么种类的数据。一系列字节代表的意义完全取决于我们怎么解释它们。二进制文件中的一串 12 个字节可以表示 12 个字符、12 个 8 位有符号的整数、12 个 8 位无符号的整数、6 个 16 位有符号的整数，或 1 个 32 位整数，后跟一个 8 字节的浮点数等。以上这些解释都是正确的，因此程序在读取文件时，必须正确地假设文件是如何写入的。

12.2　文件访问

磁盘上的每个文件都有一个名称，文件命名规则由操作系统确定。如果一个处理文件的程序只能处理特殊名字的文件，就不是很方便，需要为每个要处理的文件编写不同的程序。因此，在 C 语言中处理文件时，程序通过文件指针来引用文件。文件指针是一

www.TopSage.com

个抽象指针，关联到一个特定的文件上，所以程序可以在不同的情况下处理不同的文件。文件指针指向表示流的结构。本章的示例使用微软的 Windows 文件名。如果使用其他操作系统(例如 UNIX)就需要适当地调整文件的名称。

如果要同时使用几个文件，就需要对每个文件使用不同的文件指针，但使用完一个文件后，可以将文件指针关联到另一个文件上。因此，如果要处理多个文件，但一次只处理一个，一个文件指针就够了。

12.2.1 打开文件

将内部文件指针变量关联到一个特定的外部文件名称上的过程称为打开文件。调用标准库函数 fopen()就可以打开文件，该函数返回特定外部文件的文件指针。fopen()函数在<stdio.h>中定义，它的原型如下：

```
FILE  *fopen(char  *name,   char*  mode);
```

函数的第一个变元是一个字符串指针，它是要处理的外部文件名称。可以将该文件名明确指定为变元，也可以使用数组或一个 char 类型变量指针，它包含了文件名字符串的地址。文件名的获得一般需要采用一些外部方式，例如程序开始执行时的命令行或从键盘读入。如果程序处理的是同一个文件，也可以在程序的开头将文件名定义成一个常量。

函数 fopen()的第二个变元也是一个字符串，称为文件模式，它指定对文件进行什么处理。它有相当多的选项，但这里只介绍 3 种文件模式(包含了文件的基本操作)。表 12-1 列出了这 3 种文件模式。

表 12-1 文 件 模 式

模　式	说　明
"w"	打开一个文本文件，进行写入操作。如果文件存在，就删除其当前内容
"a"	打开一个文本文件，进行追加操作。写入的数据放在文件尾
"r"	打开一个文本文件，进行读取操作

注意：
文件模式说明是一个带双引号的字符串，而不是单引号中的单个字符。

这 3 种模式仅应用于文本文件，即将数据写入为字符的文件。也可以使用写入为一系列字节的二进制文件，详见本章后面的“二进制文件的输入输出”。如果成功地调用 fopen()，它会返回一个 File *类型的指针，通过该指针可以引用文件，使用其他库函数执行进一步的输入输出操作。如果文件因为某种原因打不开，fopen()就返回一个空指针。

注意：
fopen()函数返回的指针称为文件指针(file pointer)，或流指针(stream pointer)。

www.TopSage.com

因此，调用 fopen()，可以创建一个文件指针，指定程序要处理的磁盘文件，并确定在程序中能对它执行什么操作。

fopen()所返回的指针是 FILE *类型，或指向 FILE 的指针，其中 FILE 是通过 typedef 在头文件<stdio.h>中预定义的结构类型。文件指针指向的结构包含文件的信息，例如指定的打开模式、内存中用于数据的缓存区地址和指向文件中当前位置的指针，以用于下一个操作。实际上，不需要操心这个结构的内容，它们由输入输出函数负责。然而，如果想了解 FILE 结构，可以浏览库的头文件。

如前所述，要同时打开几个文件时，必须为每个文件声明各自的文件指针变量，并分别调用 fopen()函数，将其返回值存储在各自的文件指针中。一次能打开的文件数由<stdio.h>中定义的常量 FOPEN_MAX 确定。常量 FOPEN_MAX 是一个整数，指定了一次可以打开的最大流数。C 语言标准规定，FOPEN_MAX 的值至少是 8，包括 stdin、stdout 和 stderr。因此，我们至少一次可以处理 5 个文件。

如果要写入文本文件 myfile.txt，可以使用下列这些语句：

```
FILE *pfile = fopen("myfile.txt", "w");
/* Open file myfile.txt to write it */
```

上述语句打开文件，将物理文件 myfile.txt 关联到内部的文件指针 pfile 上。因为指定的模式是"w"，所以只能写入文件，而不能读取。第一个字符串变元的最大字数限制是<stdio.h>中定义的 FILENAME_MAX。这个数值相当大，不会成为真正的限制。

如果文件 myfile.txt 不存在，上一条语句中的 fopen()函数就会创建它。因为 fopen()函数的第一个变元只提供文件名，没有指定路径，所以这个文件在当前目录下，如果在当前目录下没有找到它，就在这个目录下创建它。也可以指定一个包含完整路径和文件名的字符串，如果在该路径下没有找到文件，就在该位置上创建文件。注意，如果包含那个文件的目录不存在，fopen()函数就不会创建目录，也不会创建文件，而是失败。如果 fopen()函数调用失败，就返回 NULL。而如果试图使用 NULL 文件指针，程序会终止。

注意：

现在可以创建新的文本文件了。只要调用 fopen()，文件模式指定为"w"，第一个变元指定新文件名即可。

打开文件，以用于执行写入操作时，文件指针会位于已有数据的开头。也就是说，在开始任何写入操作时，前一次写入文件的数据会被覆盖掉。

如果要在已有的文本文件中添加数据，而不是覆盖数据，可以要指定模式"a"，它是操作的附加模式。将文件指针放在前一次写入的数据的末尾。如果文件不存在，与模式"w"一样，也会创建新文件。使用之前声明的文件指针，打开文件，将数据添加到文件末尾，语句如下：

```
pfile = fopen("myfile.txt", "a"); /* Open file myfile.txt to add to it */
```

在追加模式中打开文件时，所有的写入操作都在文件的数据末尾执行。换言之，所有的写入操作都会把数据追加到文件中，在这种模式下不能更新已有的内容。

www.TopSage.com

如果要读入文件，声明了文件指针后，就可以使用下面的语句：

```
pfile = fopen("myfile.txt", "r");
```

模式变元指定为"r"，表示要读取文件，所以不能写入文件。文件位置设定在文件中数据的开头。如果要读入文件，文件就必须存在。如果要读取的文件不存在，fopen()会返回 NULL。因此，最好用 if 语句检查 fopen()的返回值，以确保可以访问文件。

12.2.2 文件重命名

在许多情况下都需要对文件进行重命名。例如更新文件的内容，创建一个新的、更新过的文件。这需要在创建新的文件后，给它指定一个临时的文件名，然后删除旧文件，再将这个临时的文件名更改成被删掉的文件名。文件重命名非常简单，只需使用 rename()函数，它的原型如下：

```
int rename(const char *oldname, const char *newname);
```

如果文件名更改成功，就返回整数 0，否则返回非零值。调用 rename()函数时，文件必须关闭，否则操作会失败。

下面是使用 rename()函数的例子：

```
if(rename( "C:\\temp\\myfile.txt", "C:\\temp\\myfile_copy.txt"))
  printf("Failed to rename file.");
else
  printf("File renamed successfully.");
```

这个例子会将 C 盘 temp 目录下的 myfile.txt 文件改名为 myfile_copy.txt。结果是一条改名是否成功的信息。显然，如果文件的路径有错或文件不存在，重命名操作会失败。

警告：

注意文件路径字符串中的两个斜杠。如果在指定微软 Windows 文件路径时没有使用斜杠的转义序列，就得不到希望的文件名。

12.2.3 关闭文件

使用完文件后，需告诉操作系统释放文件指针，这称为关闭文件。这个动作通过调用函数 fclose()来完成。这个函数将文件指针作为变元，返回 int 类型的值。如果成功关闭文件，就返回 0，否则返回 EOF。函数 fclose()的使用方式如下：

```
fclose(pfile); /* Close the file associated with pfile */
```

执行这行语句的结果是断开指针 pfile 和物理文件名间的连接，因此 pfile 不能再用于访问它表示的文件。如果文件在执行写入操作，就将输出缓冲区的内容写到文件中，以确保数据不会遗失。在重命名或删除文件时，也必须关闭文件。

www.TopSage.com

注意：

EOF 是一个特殊的字符，称为文件结束字符。EOF 符号在<stdio.h>中定义，一般等于 -1。但并不总是这样，所以应在程序中使用 EOF，而不是 -1。EOF 一般表示不能再从流中获得数据了。

使用完文件后，最好马上关闭文件。这可以避免输出数据的遗失，当程序的其他部分出错，会使程序不正常结束，从而遗失数据。这可能是因为文件没有正确地关闭，而遗失了输出缓冲区的数据。

注意：

另一个使用完马上关闭文件的理由是，操作系统通常会限制一次可以打开的文件数。使用完后马上关闭文件，可以使其与操作系统发生冲突的机会降到最低。

<stdio.h>中有一个函数可以迫使留在缓冲区内的数据写入文件。这个函数是 fflush()，前面的章节用它清除输入缓冲区。假定文件指针是 pfile，用以下语句将输出缓冲区内的数据写入文件：

```
fflush(pfile);
```

fflush()函数会返回一个 int 类型的数值，正常的返回值是 0，如果有错误，则返回 EOF。

12.2.4 删除文件

现在可以在代码中创建文件，有时也要编程删除文件。此时可以使用在<stdio.h>中声明的函数 remove()，其用法如下：

```
remove("pfile.txt");
```

这行语句会从当前目录中删除 pfile.txt 文件。在调用函数 remove()删除文件时，文件不应是打开的，否则，调用函数 remove()的动作取决于具体的 C 实现方式，请参阅库文档说明。

文件的任何动作都需要检查两次，尤其是删除文件的动作。

12.3 写入文本文件

打开一个文件以用于写入数据后，就可以在程序的任何地方给它写入数据，只要可以访问 fopen()为文件设置的文件指针即可。如果要在包含多个函数的任意位置访问文件，就需要确保文件指针有全局作用域，或可以作为变元传送给访问文件的函数。

注意：

要确保文件指针有全局作用域，应将它的声明放在所有函数的外部，通常在源文件

www.TopSage.com

的开头。

最简单的写入操作由函数 fputc()提供，它将一个字符写入外部文件。其原型如下：

```
int fputc(int c, FILE *pfile);
```

函数 fputc()将第一个变元指定的字符写入第二个变元(文件指针)指定的文件中。如果写入操作成功，就返回写入的字符。否则，返回 EOF。

实际上，字符不是一个一个地写入物理文件的，这样做的效率极低。在程序和输出例程的监控下，输出字符写入内存中的缓存区，缓存区累积到一定的数量后，就一次将它们写入文件，如图 12-2 所示。

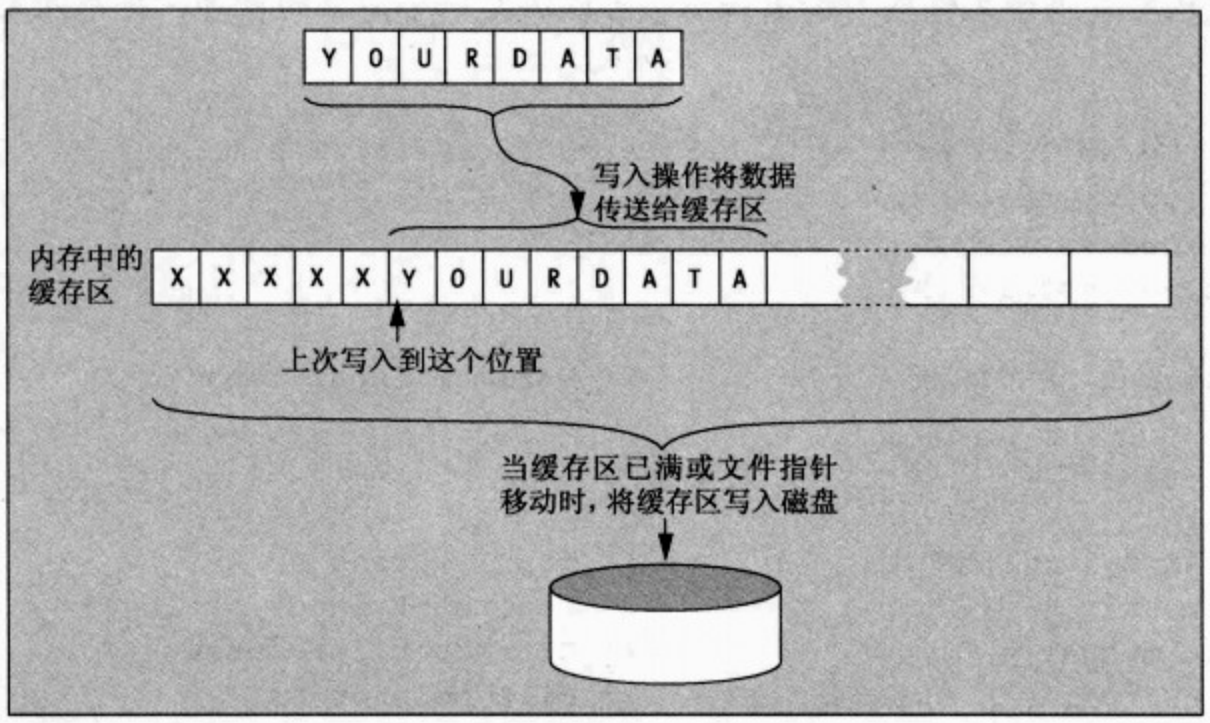


图 12-2 写入文件

注意，putc()函数等价于 fputc()，它们需要的变元和返回类型都相同，区别是 putc()在标准库中实现为一个宏，而 fputc()定义为一个函数。

12.4 读取文本文件

fgetc()函数与 fputc()函数互补，fgetc()从打开的文本文件中读取一个字符。它将文件指针作为唯一的变元，如果读取操作成功，就把读取的字符返回为 int 类型；否则，返回 EOF。fgetc()函数的一般用法如下面的语句所示：

```
mchar = fgetc(pfile); /* Reads a character into mchar */
```

这里假定 mchar 变量已声明为 int 类型。在后台，读取文件的机制与写入文件正好相反。在一次操作中将一整块字符写入缓存区。接着一次将一个字符传送给程序，直到缓存区空为止，此时再读取另一个块。读取操作非常快，因为大多数 fgetc()操作都不涉及读取磁盘，只是将一个字符从主内存的缓存区移动到指定存储它的位置上。

www.TopSage.com

注意，getc()函数等价于 fgetc()，getc()需要一个 FILE*类型的参数，将读取的字符返回为 int 类型，所以与 fgetc()完全相同，它们的唯一区别是 getc()实现为一个宏，而 fgetc()是一个函数。

警告：

不要混淆 getc()和 gets()，它们的操作完全不同。getc()从其变元指定的流中读取一个字符，而 gets()从标准输入流（键盘）中读取一整行输入，前面的章节使用 gets()函数从键盘上读取一个字符串。

试试看：使用简单的文件

现在已有足够的文件输入输出知识，可以编写一个简单的程序，先写入文件，再读出它。

```
/* Program 12.1 Writing a file a character at a time */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

const int LENGTH = 80;          /* Maximum input length */

int main(void)
{
  char mystr[LENGTH];           /* Input string */
  int lstr = 0;                 /* Length of input string */
  int mychar = 0;               /* Character for output */
  FILE *pfile = NULL;           /* File pointer */
  char *filename = "C:\\myfile.txt";
  printf("\nEnter an interesting string of less than 80 characters:\n");
  fgets(mystr, LENGTH, stdin); /* Read in a string */

  /* Create a new file we can write */

  if(!(pfile = fopen(filename, "w")))
  {
    printf("Error opening %s for writing. Program terminated.", filename);
    exit(1);
  }

  lstr = strlen(mystr);
  for(int i = lstr-1 ; i >= 0 ; i--)
    fputc(mystr[i], pfile); /* Write string to file backward */

  fclose(pfile);              /* Close the file */

  /* Open the file for reading */
  if(!(pfile = fopen(filename, "r")))
  {
```

www.TopSage.com

```
    printf("Error opening %s for reading. Program terminated.", filename);
    exit(1);
  }

  /* Read a character from the file and display it */
  while((mychar = fgetc(pfile)) != EOF)
    putchar(mychar); /* Output character from the file */
  putchar('\n');     /* Write newline */

  fclose(pfile);     /* Close the file */
  remove(filename);  /* Delete the physical file */
  return 0;
}
```

程序的输出如下:

```
Enter an interesting string.
Too many cooks spoil the broth.
.htorb eht liops skooc ynam ooT
```

代码的说明

要使用的文件名称在下面的语句中定义:

```
char  *filename  =  "C: \\myfile.txt";
```

这条语句使用微软的 Windows 文件名表示法定义 C 盘上的文件 myfile.txt。如前所述，必须使用转义序列'\\'得到反斜杠字符。如果只使用一个反斜杠，编译器会认为这是转义序列'\m'，这是一个无效的字符。

在执行这个程序之前(以及处理文件的所有例子)，要确保不存在同名、同路径的文件。如果有这么一个文件，就应改变程序中 filename 的初始值，否则已有的文件会被覆盖。

显示提示后，程序从键盘读入一个字符串。然后，执行下面的语句:

```
if(!(pfile = fopen(filename, "w")))
{
  printf("Error opening %s for writing. Program terminated.", filename);
  exit (1);
}
```

这段 if 语句中的条件调用 fopen()在 C 盘上创建新文件 myfile.txt，并打开它，以备写入数据，然后把返回的指针存储在 pfile 中。fopen()的第二个变元指定写入文件。如果 fopen()返回 NULL，就执行语句块，显示一条信息，调用<stdlib.h>中声明的 exit()函数，使程序非正常结束。

使用 strlen()得到字符串的长度后，将它存储到 lstr 变量中，然后是一个 for 循环，如下:

```
for(int i = lstr-1 ; i >= 0 ; i--)
```

www.TopSage.com

```
    fputc(mystr[i], pfile); /* Write string to file backward */
```

这个循环的索引是从字符串的最后一个字符 lstr - 1 往回到 0。因此，循环内的 puts()函数是以反向顺序将字符逐个写入新文件。要写入的文件由这个函数的第二个指针变元 pfile 指定。

在调用 fclose()关闭文件后，用下面的语句以读入模式再次打开：

```
if(!(pfile = fopen(filename, "r")))
{
  printf("Error opening %s for reading. Program terminated.", filename);
  exit(1);
}
```

模式说明符"r"指定要读入文件，所以文件的位置设定成文件的开头。这里也检查返回值是否为 NULL，和写入文件一样。

接着在 while 循环条件内使用 getc()函数从文件中读入字符：

```
while((mychar = fgetc(pfile)) != EOF)
  putchar(mychar); /* Output character from the file */
```

文件是一个字符一个字符地读入。读入操作发生在循环的继续条件式中。每读入一个字符，就用 putc()函数将它显示在屏幕上。getc()在文件的结尾返回 EOF 时，就停止这个过程。

在 main()函数的 return 语句之前，最后两行语句如下：

```
fclose(pfile);     /* Close the file */
remove(filename);  /* Delete the physical file */
```

在处理完文件后，这两行语句提供了必要的整理动作。关闭文件后，程序调用 remove()函数，删除变元定义的文件。这样可以避免磁盘中有乱七八糟的文件。如果要检查使用文本编辑器编写的文件内容，只需输出或注释掉 remove()调用。

12.5 将字符串写入文本文件

puts()函数将字符串写入 stdout，而函数 fputs()会将字符串写入文本文件。它的原型如下：

```
int  fputs(char  *pstr,  FILE  *pfile);
```

第一个变元是要写入文件的字符串指针，第二个变元是文件指针。这个函数的动作有点古怪，它会将字符串写入文件，直到碰到'\0'字符为止，但是'\0'不会写入文件。用 fputs()写入文件的不定长的字符串，可以用 fgets()将它读取出来。这是因为它是一个字符写入操作，不是二进制写入操作，所以它希望写入的一行文本以换行符结束。这个函数不需要换行符，但是读取文件(使用互补函数 fgets())时，换行符会非常有用。

如果发生错误，fputs()函数会返回 EOF，如果正常，就返回 0。其使用方式与 puts()

www.TopSage.com

相同，如下：

```
fputs("The higher the fewer", pfile);
```

这会将第一个变元的字符串输出到 pfile 指向的文件中。

12.6 从文本文件中读入字符串

fputs()的互补函数是 fgets()，它可以从文本文件中读入一个字符串。其函数原型如下：

```
char *fgets(char *pstr, int nchars, FILE *pfile);
```

fgets()有 3 个参数。它会从 pfile 所指向的文件将字符串读入 pstr 所指向的内存。该函数会一直从文件中读取字符串，直到读到了'\n'字符或读入 nchars - 1 个字符为止。

如果读到换行符，它会保留在字符串中。字符'\0'会附加到字符串的末尾。如果没有错误，fgets()会返回 pstr 指针；否则返回 NULL。函数的第二个变元可以确保输入不会超过指定给它的内存。为了避免输入的数据过量，只需在函数的第二个变元中指定接收输入的区域或数组的长度。

试试看：在文本文件中传入传出字符串

下面练习使用在文本文件中传入传出字符串的函数，用追加模式写入文件：

```
/* Program 12.2 As the saying goes...it comes back! */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

const int LENGTH = 80; /* Maximum input length */

int main(void)
{
  char *proverbs[] =
      { "Many a mickle makes a muckle.\n",
        "Too many cooks spoil the broth.\n",
        "He who laughs last didn't get the joke in"
        " the first place.\n"
      };

  char more[LENGTH];                    /* Stores a new proverb */
  FILE *pfile = NULL;                   /* File pointer */
  char *filename = "C:\\myfile.txt";

  /* Create a new file( if myfile.txt does not exist */
  if(!(pfile = fopen(filename, "w"))) /* Open the file to write it */
  {
```

www.TopSage.com

```
    printf("Error opening %s for writing. Program terminated.", filename);
    exit(1);
  }

  /* Write our first three sayings to the file. */
  int count = sizeof proverbs/sizeof proverbs[0];
  for(int i = 0 ; i < count ; i++)
    fputs(proverbs[i], pfile);

  fclose(pfile);                    /* Close the file */

  /* Open the file to append more proverbs */
  if(!(pfile = fopen(filename, "a")))
  {
    printf("Error opening %s for writing. Program terminated.", filename);
    exit(1);
  }

  printf("Enter proverbs of less than 80 characters or press Enter to
          end:\n");
  while(true)
  {
    fgets(more, LENGTH, stdin);          /* Read a proverb */
    if(more[0] == '\n')                  /* If its empty line */
      break;                             /* end input operation */
    fputs(more, pfile);                  /* Write the new proverb */
  }

  fclose(pfile);                         /* Close the file */

  if(!(pfile = fopen(filename, "r")))    /* Open the file to read it */
  {
    printf("Error opening %s for writing. Program terminated.", filename);
    exit(1);
  }

  /* Read and output the file contents */
  printf("The proverbs in the file are:\n\n");
  while(fgets(more, LENGTH, pfile)) /* Read a proverb */
    printf("%s", more);             /* and display it */
  fclose(pfile);                    /* Close the file */
  remove(filename);                 /* and remove it */
  return 0;
}
```

程序的输出如下:

```
Enter proverbs of less than 80 characters or press Enter to end:
Least said, soonest mended.
A nod is as good as a wink to a blind horse.
```

www.TopSage.com

```
The proverbs in the file are:

Many a mickle makes a muckle.
Too many cooks spoil the broth.
He who laughs last didn't get the joke in the first place.
Least said, soonest mended.
A nod is as good as a wink to a blind horse.
```

代码的说明

用下列语句初始化数组指针 proverbs[]:

```
char *proverbs[] =
    { "Many a mickle makes a muckle.\n",
      "Too many cooks spoil the broth.\n",
      "He who laughs last didn't get the joke in"
      " the first place.\n"
    };
```

以三个谚语作为数组元素的初始值，会使编译器分配足够的空间来存储这些字符串。再声明一个数组，存储从键盘上读取的谚语:

```
char more[LENGTH]; /* Stores a new proverb */
```

上述语句用另一个谚语初始化了一个传统的 char 数组。同样，在这个字符串的最后包含'\n'字符。

在 C 盘上创建并打开一个用于写入的文件后，程序就在一个循环中将最初的三个谚语写入文件:

```
int count = sizeof proverbs/sizeof proverbs[0];
for(int i = 0 ; i < count ; i++)
  fputs(proverbs[i], pfile);
```

在 for 循环内使用函数 fputs()将 proverbs[]数组元素所指的内存区的内容写入文件。这个函数非常简单，第一个变元是字符串指针，第二个变元是文件指针。

数组中的谚语个数用以下的表达式计算:

```
sizeof  proverbs/sizeof  proverbs[0]
```

表达式 sizeof proverbs 等于整个数组所占的字节数，sizeof proverbs[0]是数组中一个指针元素所占的字节数。因此，整个式子等于指针数组的元素数目。

当然，也可以手动算出共有多少个初始字符串，但是用这个方法可以自动算出正确的迭代次数，即使数组因增加初始字符串而改变大小时，这个表达式仍然正确。

写入第一组谚语后，就关闭文件，然后重新打开文件，如下:

```
if(!(pfile = fopen(filename, "a")))
{
  printf("Error opening %s for writing. Program terminated.", filename);
```

www.TopSage.com

```
    exit(1);
  }
```

模式指定为"a"，所以文件以追加模式打开。注意，在这个模式下，文件的当前位置自动设定在文件的末尾，所以后续的写入动作会追加到文件中已有数据的后面。

输入提示后，从键盘上读取更多的谚语，将它们写入文件，如下面的语句所示:

```
while(true)
{
  fgets(more, LENGTH, stdin); /* Read a proverb */
  if(more[0] == '\n')         /* If its empty line */
    break;                    /* end input operation */
  fputs(more, pfile);         /* Write the new proverb */
}
```

存储在 more 数组中的谚语也使用 fputs()写入文件。可以看到，fputs()函数将数组用作指针。因为文件使用了追加模式，新的谚语会添加到文件中已有数据的后面。当输入一个空行时，循环结束。空行是只包含'\n'和字符串终止字符的字符串。

写入文件后，关闭文件，再次使用模式指定符"r"打开它，以进行读取。然后，是一个 while 循环:

```
while(fgets(more, LENGTH, pfile)) /* Read a proverb */
  printf("%s", more);             /* and display it */
```

在循环继续条件式中成功地将字符串从文件读入 more 数组中。每读完一个字符串，就在循环中调用 printf()函数将它显示在屏幕上。fgets()在读入每个谚语时，只要检测到'\n'字符，就停止读取。fgets()函数返回 NULL 时，循环就会结束。

最后关闭文件，然后使用函数 remove()删除文件，其方法与上一个例子相同。

12.7 格式化文件的输入输出

将字符及字符串写入文件比较顺利，但是在程序中一般有许多其他的数据类型。例如，要将数值数据写入文件，就需要更多的操作；要使文件中的内容能让人看得懂，还需要数值数据的字符表示。而格式化文件的输入输出函数提供了这样的机制。

12.7.1 格式化文件输出

在第 10 章讨论标准流时，就遇到过格式化文件输出的函数。它和 printf()语句相同，但有一个额外的参数，名称也稍有不同。它的用法如下：

```
fprintf(pfile, "%12d112d%14f", num1, num2, fnum1);
```

可以看到，这个函数名称比 printf()多了一个 f(文件)，第一个变元是指定输出目的地的文件指针。显然，文件指针需要通过调用 fopen()来设定。其余的变元和 printf()相同。

www.TopSage.com

这个例子根据第二个变元指定的格式字符串，将 3 个变量 num1、num2 及 num3 的值写入文件指针 pfile 所指定的文件。因此，前两个 int 类型的变量用字段宽度 12 写入文件，第 3 个 float 类型的变量用字段宽度 14 写入文件。

12.7.2 格式化文件输入

使用 fscanf()函数可以得到格式化文件输入。例如，从文件 pfile 读入 3 个变量值，可以使用如下语句：

```
fscanf(pfile, "%12d%12d%14f", &num1, &num2, &fnum1);
```

这个函数的操作和 scanf()对 stdin 的操作完全相同，只是要从第一个变元指定的文件中得到输入。scanf()函数的使用规则也适用于这个函数的格式字符串和操作。如果发生错误，没有读取输入，函数会返回 EOF，否则将读取的值的个数返回为 int 类型的值。

试试看：使用格式化输入及输出函数

现在用一个例子演示格式化输入及输出函数，并说明这些操作对数据带来了什么影响。

```
/* Program 12.3 Messing about with formatted file I/O */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  long num1 = 234567L; /* Input values... */
  long num2 = 345123L;
  long num3 = 789234L;

  long num4 = 0L;       /* Values read from the file... */
  long num5 = 0L;
  long num6 = 0L;

  float fnum = 0.0f;    /* Value read from the file */
  int ival[6] = { 0 }; /* Values read from the file */
  FILE *pfile = NULL;   /* File pointer */
  char *filename = "C:\\myfile.txt";

  pfile = fopen(filename, "w"); /* Create file to be written */
  if(pfile == NULL)
  {
    printf("Error opening %s for writing. Program terminated.", filename);
    exit(1);
  }
  fprintf(pfile, "%6ld%6ld%6ld", num1, num2, num3);  /* Write file */
  fclose(pfile);                                     /* Close file */
  printf("\n %6ld %6ld %6ld", num1, num2, num3);
```

www.TopSage.com

```
    /* Display values written */

  pfile = fopen(filename, "r"); /* Open file to read */
  fscanf(pfile, "%6ld%6ld%6ld", &num4, &num5 ,&num6); /* Read back */
  printf("\n %6ld %6ld %6ld", num4, num5, num6); /* Display what we got */

  rewind(pfile); /* Go to the beginning of the file */
  fscanf(pfile, "%2d%3d%3d%3d%2d%2d%3f", &ival[0], &ival[1],
      &ival[2], &ival[3], &ival[4] , &ival[5], &fnum); /* Read it again */
  fclose(pfile);     /* Close the file and */
  remove(filename);  /* delete physical file. */

  /* Output the results */
  printf("\n");
  for(int i = 0 ; i < 6 ; i++ )
    printf("%sival[i] = %d", i == 4 ? "\n\t" : "\t", i, ival[i]);
  printf("\nfnum = %f\n", fnum);
  return 0;
}
```

这个例子的输出如下:

```
234567 345123 789234
234567 345123 789234
       ival[i] = 0   ival[i] = 1   ival[i] = 2   ival[i] =3
       ival[i] = 4   ival[i] = 5
fnum = 234.000000
```

代码的说明

这个例子将 3 个变量 num1、num2、num3 的值写入 C 盘的 myfile.txt 文件。这 3 个变量是在声明中定义和初始化的。文件名是通过文件指针 pfile 指定的。这个文件先关闭，然后再次打开，以读取数据，从文件中读取的数据采用写入时的格式。之后是下面的语句:

```
rewind(pfile);
```

这行语句调用 rewind()函数，将当前位置放回到文件的开头，以便再次读取。也可以关闭文件，再打开文件，来达到相同的目的，但是使用 rewind()只需使用一个函数调用，速度也快得多。

文件重新定位后，使用下面的语句再次读取文件:

```
fscanf(pfile, "%2d%3d%3d%3d%2d%2d%3f", &ival[0], &ival[1],
  &ival[2], &ival[3], &ival[4] , &ival[5], &fnum); /* Read it again */
```

这条语句将相同的数据读入数组 ival[]和变量 fnum，但是使用的格式和写入文件时不同。其结果是，这个文件只由字符串组成，和 printf()输出到屏幕上的结果相同。

www.TopSage.com

注意：

如果输出的格式指定符指定的精度小于存储数值的精度，就会丢失信息。

从文件读回的值，取决于所使用的格式字符串和在 fscanf()函数中指定的变量列表。

写入文件时，不需要维护内部的源信息。把数据放在文件中，它就只是一串字节，这些字节的意义取决于我们如何解释它们。这种情形在这个例子中可以很清楚地看到，本例将原来的 3 个值转换成 8 个新值。

最后和前几个例子一样，关闭文件，用 remove()函数删除文件。

12.8 错误处理

本书的例子都只包含最起码的错误检查及报告，因为完整的错误检查和报告代码会占用很大的篇幅，使程序看起来相当复杂。然而在实际的程序中，应尽可能地检查及报告错误。

一般来说，应将错误信息写入 stderr，stderr 可自动用于程序，且总是指向显示屏幕。stdout 可以通过一个操作系统语句重定向到文件上，但是 stderr 仍是指向屏幕。在读取文件之前，检查文件是否存在是很重要的，前面的例子就是这么做的，当然还可以做得更多。首先，可以将错误信息写入 stderr，而不是 stdin，例如：

```
char *filename = "C:\\MYFILE.TXT"; /* File name */
FILE *pfile = NULL;                /* File pointer */

if(!(pfile = fopen(filename, "r")))
{
  fprintf(stderr, "\nCannot open %s to read it.", filename);
  exit(1);
}
```

写入 stderr 的优点是，输出总是要显示出来，且立即写到显示设备上。这就是说，无论程序中发生什么，输出都会写入 stderr。而 stdin 流会缓存，如果程序崩溃，数据就有可能留在缓存区中，不显示出来。调用 exit()函数终止程序，可以确保刷新输出流缓存区，使输出写入最终的目的地。流 stdin 可以重定向到文件上，但 stderr 不能重定向，以确保总是有输出。

知道会发生某类错误是很有用的，但是还可以做得更多。perror()函数可以输出将变元传给它的字符串，以及与所发生错误对应的系统定义的错误信息。因此，可以将之前的代码段改写为：

```
if(!(pfile = fopen(myfile, "r")))
{
  perror(strcat("Error opening ", filename));
  exit(1);
}
```

www.TopSage.com

这将输出信息，其中包含 strcat()的第一个变元、文件名及与错误相关的系统信息。输出会写入 stderr。

如果读取文件时发生错误，可以检查是否到达了文件尾。如果到达了文件尾，feof()函数就会返回一个非零整数。因此，可以使用下面的语句来检查：

```
if(feof(pfile))
  printf("End of file reached.");
```

注意，这里没有将信息写入 stderr，因为到达文件尾不算错误。

如果传给 ferror()函数的文件指针所指定的流在运作时发生错误，ferror()函数就返回一个非零整数。调用这个函数可以建立一个确定有错误发生的机制。<errno.h>头文件定义了一个值 errno，它指定发生了哪一种错误。读者应查阅 C 语言的文档说明，了解有关的信息。errno 的值是为错误设定的，而不只是文件操作。

应总是在程序中包含一些基本的错误检查及报告代码。编写一些程序后，会发觉为每种操作错误包含标准的检查代码并不是很辛苦。如果使用标准方法，就可以在程序之间复制大多数错误检查代码。

12.9 再探文本文件操作模式

前面使用的文本模式都是打开文件的默认操作模式。在 C 的早期版本中，可以明确指定文件以文本模式打开。为此，只需在已有的指定符后面加上“t”。因此，除了原来的 3 个模式之外，还有 3 个模式指定符"wt"、"rt"和"at"。这里提及它们，因为读者可能在其他 C 程序中遇到它们。尽管大多数编译器支持它们，但它们不是当前 C 标准的内容，所以最好不要在代码中使用它们。

也可以使用指定符"r+"，打开文件，进行更新，即读写文件。如果要读写一个新文件，或在开始之前删除已有文件的原始内容，还可以将打开模式指定为"w+"。用"w+"打开文件，会将已有文件的长度设置为 0，所以应在删除当前文件的内容时使用这个模式。在旧式程序中，还可能遇到"rt+"、"r+t"、"wt+"和"w+t"模式。

如前所述，在更新模式下，可以读写文本文件。但不能在写入文件后就马上读取，或在读取文件后马上写入，除非到达 EOF，或采用某种方法改变了文件中的当前位置(例如调用 rewind()函数或其他函数，来改变文件位置)。这是因为将数据写入文件，并不表示把数据写到外部设备上，而只是传送到缓冲区中，直到缓冲区满了，或发生了其他事件，才将数据写入文件。同样，第一次读取文件时，会将读取的数据填入内存的缓冲区，后续的读取操作会从缓冲区中传出数据，直到缓冲区空了，才会再次从文件中读取数据，以填满缓冲区，如图 12-3 所示。

这意味着，如果从写入模式立即切换到读取模式，数据就会遗失，因为它们会留在缓冲区内。从读取模式切换到写入模式时，文件的当前位置可能和想象的不一样，导致覆盖文件中的数据。因此，在读取模式和写入模式之间切换时，需要一个刷新缓冲区的中介事件。fflush()函数会将留在输出缓冲区内的字节写到输出文件中。

www.TopSage.com

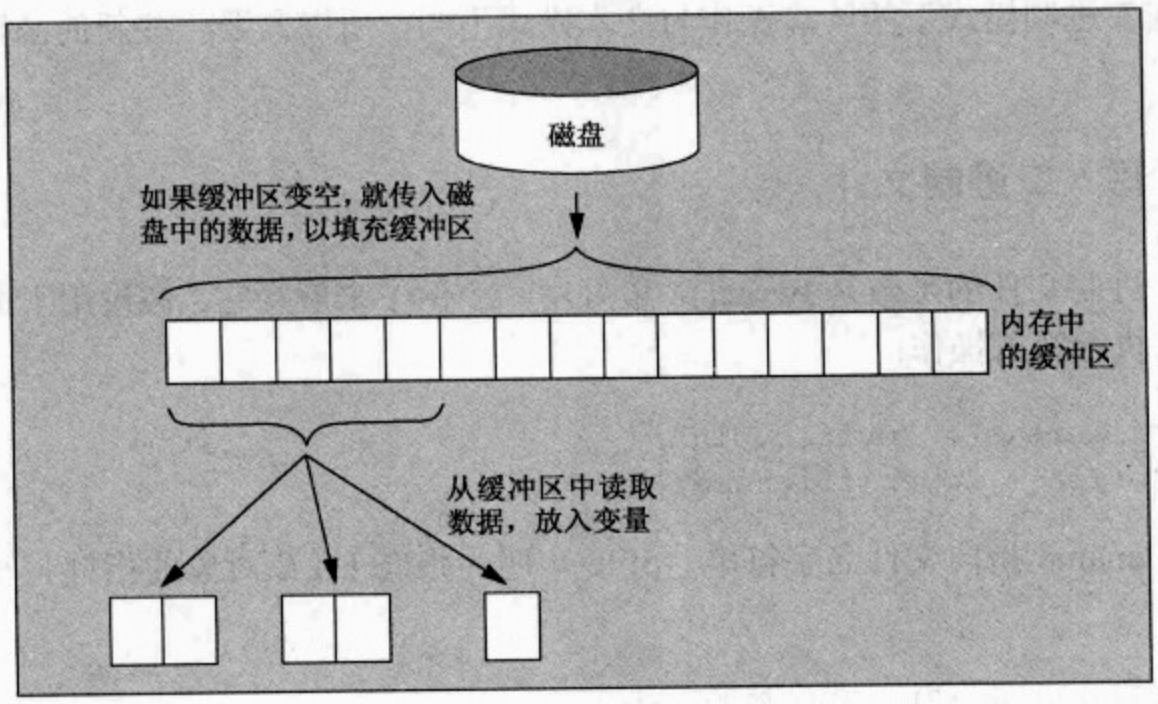


图 12.3 缓存的输入操作

12.10 二进制文件的输入输出

文件操作除了文本模式外，还有一个二进制模式。在这个模式下，不转换数据，也不需用格式字符串控制输入输出，所以它比文本模式简单。二进制模式将内存的数据直接传送到文件中。文本模式下具有特殊意义的字符，如'\n'和'\0'，在二进制模式下就没有意义了。

二进制模式的优点是没有数据转换，也没有精度的损失。而文本模式因为有格式化过程，有数据转换和精度损失。另外，二进制模式比文本模式的速度快。图 12-4 是这两种模式的比较。

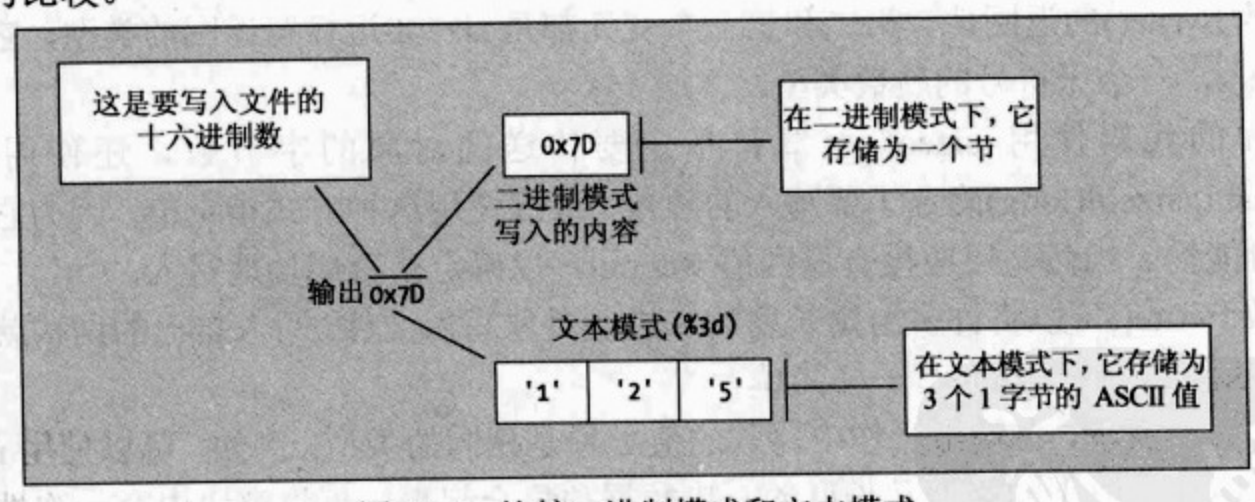


图 12-4 比较二进制模式和文本模式

12.10.1 指定二进制模式

要指定二进制模式，只需在基本打开模式指定符后附加 b。因此，打开模式指定符"wb"表示写入二进制文件，"rb"表示读取二进制文件，"ab"表示将数据追加到二进制文件的末尾，"rb+"表示读写二进制文件。

www.TopSage.com

数据在二进制模式下的读写操作与文本模式不同，所以需要一些新的函数执行输入输出。

12.10.2 写入二进制文件

写入二进制文件的函数是 fwrite()。最好用一个例子来解释它。假设用下面的语句打开文件，以执行写入操作：

```
char *filename = "myfile.bin";
FILE *pfile = fopen(filename, "wb");
```

变量 filename 指向文件名字符串，pfile 声明为指向 FILE 对象的指针。写入文件的语句如下：

```
long pdata[] = {2L, 3L, 4L};
int num_items = sizeof(pdata)/sizeof(long);
FILE *pfile = fopen(filename, "wb");
size_t wcount = fwrite(pdata, sizeof(long), num_items, pfile);
```

fwrite()函数将指定字节数的数据项写入文件，其中每个数据项都是指定数量的 long 字节。第一个变元 pdata 是一个指针，包含要写入的数据项在内存中的开始位置。第二个变元 size 指定每个要写入的项的字节数。第三个变元 num_items 指定了要在文件中写入多少项。最后一个变元 pfile 是写入数据的文件。函数 fwrite()将实际写入的数据项的个数返回为 size_t 类型的值。如果操作因为某种原因而失败，这个值就小于 num_items。

注意，在调用函数 fwrite()时，没有检查是否在二进制模式下打开文件。写入操作会把二进制数据写入以文本模式打开的文件。当然，也可以把文本数据写入二进制文件。这会使结果很难理解。

函数 fwrite()的返回值、第二和第三个变元都是 sizeof 运算符返回的类型，它定义为类型 size_t，一个无符号的整数类型。

上面的代码使用 sizeof 运算符指定要传送的对象的字节数，还使用表达式 sizeof(pdata)/sizeof(long)确定了要写入的项数。最好使用这种方式指定值，因为它减少了出错的可能性。当然，还应检查返回值 wcount，以确定是否成功地写入。

函数 fwrite()可以将许多给定长度的二进制对象写入文件。写入自己的结构就像写入 int 值、double 值或一串字节一样简单。

但这并不表示，在输出操作中写入的值必须是相同的类型。例如，可以使用 malloc()分配一些内存，将不同类型及长度的一串数据项集合起来，再将整块内存一次性写出为一串字节。当然，读取它们时，必须知道文件中值的顺序及类型，才能使它们有意义。

12.10.3 读取二进制文件

二进制文件以读取模式打开后，就可以使用 fread()函数读取它。使用和写入二进制文件例子中相同的变量读取文件，语句如下：

www.TopSage.com

```
size_t wcount = fread( pdata, sizeof(long), num_items, pfile);
```

这个操作和写入操作正好相反。函数从 data 指定的地址开始，读取 num_items 个对象，每个对象的字节数由第二个变元指定。这个函数返回读取的项数。如果读取不完全成功，这个项数就小于要求读取的数目。

试试看：读入二进制文件

可以将二进制文件的操作应用于第 7 章中计算质数的程序 7.11。这次使用磁盘文件作为缓冲区，计算更多的质数。如果存储质数的数组不足以存储所要求的质数数目，可以让程序把质数自动存储在文件中。在这个寻找质数的程序版本中，要改进检查过程。

除了 main()函数(该函数包含寻找质数的循环)之外，再编写一个函数 test_prime()，测试某个值是否是质数，用辅助函数 check()比较给定的值和已有的质数，用 put_primes()函数从文件中提取质数，并显示它们。

这个程序有几个函数，使用全局变量，所以下面一点一点地讨论。首先是函数原型和全局数据：

```
/* Program 12.4 A prime example using binary files */
#include <stdio.h>
#include <stdlib.h>
#include <math.h > /* For square root function sqrt() */

/* Function prototypes */
int test_prime(unsigned long long N);
void put_primes(void);
int check(unsigned long long buffer[], size_t count, unsigned long long N);

/* Global data */
const unsigned int MEM_PRIMES = 100;
/* Count of number of primes in memory */

struct
{
  char *filename;                              /* File name for primes */
  FILE *pfile;                                 /* File stream pointer */
  int nrec;                                    /* Number of file records */
  unsigned long long primes[MEM_PRIMES]; /* Array to store primes */
  size_t index;                  /* Index of free location in array primes */
} global = { "C:\\myfile.bin",    /* Physical file name */
             NULL,                /* File pointer value */
             0,                   /* File record count */
             {2ULL, 3ULL, 5ULL}, /* primes array values */
             3                    /* Number of primes */
             };

int main(void)
{
  /* Code for main()... */
```

www.TopSage.com

```
}

/*****************************************************************
* Function to test if a number, N, is prime using primes in      *
* memory and on file                                             *
* First parameter N - value to be tested                         *
* Return value - a positive value for a prime, zero otherwise    *
*****************************************************************/
int test_prime(unsigned long long N)
{
  /* Code for test_prime()... */
}

/*****************************************************************
* Function to check whether an integer, N, is divisible by any   *
* of the elements in the array pbuffer up to the square root of N. *
* First parameter buffer - an array of primes                    *
* second parameter count - number of elements in pbuffer         *
* Third parameter N - the value to be checked                    *
* Return value - 1 if N is prime, zero if N is not a prime,      *
*                -1 for more checks                              *
*****************************************************************/
int check(unsigned long long buffer[], size_t count, unsigned long long N)
{
  /* Code for check()... */
}

/*********************************************
* Function to output primes from the file    *
*********************************************/
void put_primes(void)
{
  /* Code for put_primes()... */
}
```

在#include 语句后，是 3 个在程序中使用的函数原型：

```
int test_prime(unsigned long long N);
void put_primes(void);
int check(unsigned long long buffer[], size_t count, unsigned long long N);
```

原型中只有参数的类型，没有参数名。函数原型有没有参数名称都没关系，但是必须指定参数的类型。通常最好有参数的名称，因为它们可以提示参数的作用。原型中的参数名可以和函数定义中的参数名不一样，但这只应在有助于理解程序时才这么编写。为了得到数量最多的质数，将质数存储为 unsigned long long 类型。

需要在几个函数中访问变量时，将这些变量定义为全局变量通常比较方便。这可以避免函数有很长的参数列表。每次调用函数时，都需要给函数传送数据作为变元。但是，全局变量声明得越多，就越有可能与程序中的本地变量或标准库中使用的名字发生冲突。

www.TopSage.com

所以最好使全局变量的个数降到最低。一种方式是将全局变量放在结构中，这里定义了一个没有类型名的结构，还定义了一个结构变量 global。因此，定义为该结构成员的所有变量都没有全局作用域，因为它们必须用 global 来修饰。

结构的前 3 个成员是 filename、pfile 和 nrec，filename 是存储质数的文件名，pfile 是文件流指针，nrec 变量存储文件中的记录数。接着，用 primes 数组在内存中存储至多 MEM_PRIMES 个值，之后就需要把质数写入文件。index 变量记录了 primes 数组中的当前空闲元素。

这里用一个初始化列表初始化结构的成员。primes 数组中的 3 个元素初始值显示在括号中；由于现在数组只有 3 个值，其他元素都设置为 0。

下面是 main()函数的定义：

```
int main(void)
{
  unsigned long long trial = 5ULL; /* Prime candidate */
  unsigned long num_primes = 3UL;  /* Prime count */
  unsigned long total = 0UL;       /* Total required */

  printf("How many primes would you like? ");
  scanf("%lu", &total);            /* Total is how many we need to find */
  total = total<4UL ? 4UL:total;   /* Make sure it is at least 4 */

  /* Prime finding and storing loop */
  while(num_primes < total)        /* Loop until we get total required */
    {
    trial += 2ULL;                 /* Next value for checking */
    if(test_prime(trial))          /* Check if trial is prime */
    {                              /* Positive value means prime */
      global.primes[global.index++] = trial; /* so store it */
      num_primes++;                /* Increment total number of primes */

      if(global.index == MEM_PRIMES) /* Check if array is full */
      {
        /* File opened OK? */
        if(!(global.pfile = fopen(global.filename, "ab")))
        { /* No, so explain and end the program */
          printf("\nUnable to open %s to append\n", global.filename);
          exit(1);
        }
        /* Write the array */
        fwrite(global.primes, sizeof(unsigned long long),
                    MEM_PRIMES, global.pfile);

        fclose(global.pfile); /* Close the file */
        global.index = 0U;    /* Reset count of primes in memory */
        global.nrec++;        /* Increment file record count */
      }
    }
```

www.TopSage.com

```
    }

    if(total>MEM_PRIMES)  /* If we wrote some to file */
      put_primes();       /* Display the contents of the file */
    if(global.index)      /* Display any left in memory */
      for(size_t i = 0; i<global.index ; i++)
      {
        if(i%5 == 0)
          printf("\n");                    /* Newline after five */
        printf("%12llu", global.primes[i]); /* Output a prime */
      }

  if(total>MEM_PRIMES)            /* Did we need a file? */
    if(remove(global.filename))   /* then delete it. */
      printf("\nFailed to delete %s\n", global.filename);
                                  /* Delete failed */
    else
      printf("\nFile %s deleted.\n", global.filename); /* Delete OK */
  return 0;
}
```

代码的说明

main()函数的定义放在全局声明的后面。执行程序时，输入要查找的质数的个数，这个值用于控制查找质数的循环。查找质数是函数 test_prime()完成的，它在循环的 if 语句中调用。如果所测试的值是质数，该函数就返回 1，否则返回 0。如果找到了一个质数，就执行如下语句：

```
global.primes[global.index++] = trial; /* so store it */
num_primes++;                          /* Increment total number of primes */
```

第一行语句把找到的质数存储在 global.primes[]数组中。用 num_primes 变量记录找到的质数个数。结构成员变量 global.index 记录在任意给定时刻内存中的质数个数。

每次找到一个质数，将它添加到 primes[]数组中时，都进行如下检查：

```
if(global.index == MEM_PRIMES) /* Check if array is full */
{
  /* File opened OK? */
  if(!(global.pfile = fopen(global.filename, "ab")))
  { /* No, so explain and end the program */
    printf("\nUnable to open %s to append\n", global.filename);
    exit(1);
  }
  /* Write the array */
  fwrite(global.primes, sizeof(unsigned long long),
        MEM_PRIMES, global.pfile);

  fclose(global.pfile); /* Close the file */
  global.index = 0U;    /* Reset count of primes in memory */
```

www.TopSage.com

```
    global.nrec++; /* Increment file record count */
  }
```

如果 global.primes 数组已满，if 条件就是 true，执行相应的语句块。这里是以二进制模式打开文件，以追加数据。第一次写入数据时，会创建一个新文件。以后调用 fopen()函数时，会打开已有的文件，将其当前位置设置为在文件中已有数据的后面，准备写入下一个块。写入一个块后，就关闭文件。在检查质数的函数中，需要再次打开文件，以进行读取。

在这组语句的最后，内存中的质数个数重置为 0，因为它们已经安全地移出了，并递增写入文件的质数块个数。

找到足够多的质数后，就用下面的语句显示质数:

```
if(total>MEM_PRIMES)  /* If we wrote some to file */
  put_primes();       /* Display the contents of the file */
if(global.index)      /* Display any left in memory */
  for(size_t i = 0; i<global.index ; i++)
  {
    if(i%5 == 0)
      printf("\n");                          /* Newline after five */
    printf("%12llu", global.primes[i]);      /* Output a prime */
  }
```

如果所请求的质数个数可以放在内存中，就根本不会写入文件。因此，在调用 put_primes()函数，将质数输出到文件中之前，必须检查 total 是否超过 MEM_PRIMES。如果 global.index 的值是正的，global.primes 数组就有未写入文件的质数。此时在 for 循环中显示它们，每 5 个一行。

最后，在 main()中，用下面的语句从磁盘中删除文件:

```
if(total>MEM_PRIMES)              /* Did we need a file? */
  if(remove(global.filename))     /* then delete it. */
    printf("\nFailed to delete %s\n", global.filename); /* Delete failed */
  else
    printf("\nFile %s deleted.\n", global.filename); /* Delete OK */
```

第一个 if 语句确保在未创建文件的情况下，不删除它。

检查某个值是否是质数的函数如下:

```
int test_prime(unsigned long long N)
{
  unsigned long long buffer[MEM_PRIMES];
   /* local buffer for primes from file */

  int k = 0;

  if(global.nrec > 0) /* Have we written records? */
  {
    if(!(global.pfile = fopen(global.filename, "rb")))
```

www.TopSage.com

```
    /* Then open the file */
    {
      printf("\nUnable to open %s to read\n", global.filename);
      exit(1);
    }

    for(size_t i = 0; i < global.nrec ; i++)
    { /* Check against primes in the file first */
      /* Read primes */
      fread(buffer, sizeof( long long), MEM_PRIMES, global.pfile);
      if((k = check(buffer, MEM_PRIMES, N)) >= 0) /* Prime or not? */
      {
        fclose(global.pfile); /* Yes, so close the file */
        return k;             /* 1 for prime, 0 for not */
      }
    }
    fclose(global.pfile);     /* Close the file */
  }

  /* Check against primes in memory */
  return check(global.primes, global.index, N);
}
```

test_prime()函数将要测试的值作为变元，如果它是质数，就返回 1，否则返回 0。

如果已将数据写入文件，global.nrec 的值就是正的。此时，文件中的质数需要先用作除数，因为它们比当前内存中的值小。

注意：

质数是只能被 1 和它本身整除的数。要检查一个数是否是质数，只需检查该数是否能用小于该数的平方根的质数整除。这来自于一个简单的逻辑——任何大于平方根的除数都可以被小于该平方根的数整除。

函数中读取文件的语句如下：

```
fread(buffer, sizeof( long long), MEM_PRIMES, global.pfile);
```

这行语句会从文件中将一个质数块读入数组缓冲区。第二个变元定义每个要读取的对象的大小，MEM_PRIMES 定义要读入多少个指定大小的对象。

读取一个块后，执行下列检查语句：

```
if((k = check(buffer, MEM_PRIMES, N)) >= 0) /* Prime or not? */
{
  fclose(global.pfile); /* Yes, so close the file */
  return k;             /* 1 for prime, 0 for not */
}
```

在 if 的条件式中，调用 check()函数确定是否有数组元素可以整除这个测试值。如果能整除，函数就返回 0，表示这个测试值不是质数。如果从 2 到这个测试值的平方根都

www.TopSage.com

不能整除这个数，就返回 1，表示这个测试值是质数。不论 check()返回什么值，结束后都要关闭文件，并给 main()函数返回相同的值。

如果不能整除这个测试值，check()函数就返回 - 1，但是它不会超过这个测试值的平方根，不需要检查返回值 - 1，因为 check()返回的值不是 0 就是 1。因此，在 for 循环的下次迭代中，如果有下一个质数块，就从文件中读取它。

如果文件的内容已读取完，仍未确定 N 是否是质数，for 循环就结束，并关闭文件，然后执行下面的语句：

```
return check(global.primes, global.index, N);
```

这次 check()函数使用内存中 primes 数组的质数来寻找质数。如果找到质数，check()就返回 1，否则返回 0。而 check()返回的值会返回给 main()函数。

check()函数的代码如下：

```
int check(unsigned long long buffer[], size_t count, unsigned long long N)
{
  /* Upper limit */

  unsigned long long root_N = (unsigned long long)(1.0 + sqrt(N));

  for(size_t i = 0 ; i<count ; i++)
  {
    if(N % buffer[i] == 0ULL ) /* Exact division? */
      return 0;                /* Then not a prime */

    if(buffer[i] > root_N)     /* Divisor exceeds square root? */
      return 1;                /* Then must be a prime */
  }
  return -1;                   /* More checks necessary... */
}
```

这个函数的作用是检查传送为第一个变元的 buffer 数组中是否有质数可以整除第二个变元，即测试值 N。函数中的本地变量声明如下：

```
/* Upper limit */
unsigned long long root_N = (unsigned long long)(1.0 + sqrt(N));
```

整数变量 root_N 是检查测试值的除数上限，只测试小于测试值 N 的平方根的除数。在 for 循环中完成测试：

```
for(size_t i = 0 ; i<count ; i++)
{
  if(N % buffer[i] == 0ULL ) /* Exact division? */
    return 0;                /* Then not a prime */

  if(buffer[i] > root_N) /* Divisor exceeds square root? */
    return 1;                /* Then must be a prime */
}
```

www.TopSage.com

这个循环会遍历 buffer 数组中的每一个除数。如果有一个除数能整除 N，就结束函数，并返回 0，表示这个值不是质数。如果遍历到大于 root_N 的除数，就表示已经测试完小于这个数的所有质数，所以 N 一定是质数，函数返回 1。如果循环结束时没有执行 return 语句，就表示没有找到能整除 N 的数，但此时尚未测试完所有小于 root_N 的质数，函数返回 -1，表示还需要检查。

最后一个函数将文件中的所有质数显示出来:

```
void put_primes(void)
{
  unsigned long long buffer[MEM_PRIMES]; /* Buffer for a block of primes */
  if(!(global.pfile = fopen( global.filename, "rb"))) /* Open the file */
  {
    printf("\nUnable to open %s to read primes for output\n",
                  global.filename);
    exit(1);
  }

  for (size_t i = 0U ; i< global.nrec ; i++)
  {
    /* Read a block of primes */
    fread(buffer, sizeof( unsigned long long), MEM_PRIMES, global.pfile);

    for(size_t j = 0 ; j<MEM_PRIMES ; j++) /* Display the primes */
    {
      if(j%5 == 0U)                        /* Five to a line */
         printf("\n");
      printf("%12llu", buffer[j]);         /* Output a prime */
    }
  }
  fclose(global.pfile);                    /* Close the file */
}
```

put_primes()函数的操作非常简单。打开文件后，将质数块读入 buffer 数组中，在读取每块质数时，for 循环会将它们以 5 个一行、字段宽度为 12 的方式显示在屏幕上。在读完所有数据后，就关闭文件。

要执行这个程序，必须将所有的函数集合起来，放在一个文件中，并编译它。利用这个程序可以得到任意多个质数。

这个程序的缺点是质数的数目很大时，由于输出的速度过快，来不及看清楚质数。为了克服这个缺点，可以将它们输出到打印机上，作为永久的记录，而不是写到屏幕上。或者让程序显示一个提示，等用户按任意键后显示下一页。

12.11 在文件中移动

在许多应用程序中，需要能随机访问文件中的数据，而不是按顺序访问它们。某些信息存储在文件的中央，因此必须从文件的开头读起，直到找到需要的信息为止。但如

www.TopSage.com

果文件包含几百万项，就要花相当多的时间。

当然，要随机访问数据，需要知道要提取的数据存储在文件的什么地方。这一般是一个相当复杂的主题。有许多不同的方法可以建立指针或索引，快速而简易地直接访问文件中的数据。其基本概念类似于书本的目录。用一个键表指出文件中每个记录的内容，每个键都定义了一个关联的位置，记录在文件中存储数据的地方。

下面看看库中处理这类输入输出的基本工具。

注意：

不能在追加模式下更新文件。在移动文件位置时，不论涉及到什么操作，所有的写入操作都在已有数据的后面发生。

12.11.1 文件定位操作

文件定位有两个方面：找出当前我们在文件中的位置，然后移动到文件中某个特定的位置。前者是后者的基础：如果不知道在文件的什么地方，就不可能知道如何到达要去的地方。

无论文件是以二进制模式还是以文本模式打开，都可以访问文件的随机位置。然而，使用文本模式文件在某些环境下是比较复杂的，尤其是微软的 Windows 环境。事实上，文件记录的字符数比实际写入的多，因为内存中的换行符'\n'在写入文本模式的文件时，会转换成两个字符(回车 CR 和换行 LF)。当然，读取数据的 C 库函数会还原这个字符。问题是假定某个位置距离文件开头有 100 个字节，以文本模式在文件中写入 100 个字符，它们是否在文件中就占据 100 个字节，取决数据中是否包含换行符。如果随后将内存中相同长度的不同数据写入文件，则只有该数据含有相同数量的'\n'字符，它们在文件中才会有相同的长度。

因此，最好避免随机写入文本文件。这里不讨论文本文件的随机写入操作，而专注于比较有用且简单的二进制文件的随机访问。

12.11.2 找出我们在文件中的位置

有两个函数可以指出我们在文件中的位置，它们相当类似，但并不相同。它们是互补的位置函数。第一个函数是 ftell()，它的原型是：

```
long  ftell(FILE  *pfile);
```

这个函数将一个文件指针作为变元，返回一个 long 整数值，指定文件中的当前位置。这个函数可以使用之前使用的文件指针 pfile 所指向的文件，语句如下：

```
fpos =  ftell(pfile);
```

www.TopSage.com

变量 fops 的类型是 long，它包含文件中的当前位置，所以可以在以后的函数调用中使用它返回这个位置。这个值是距离文件开头的偏移字节数。

第二个提供目前文件位置信息的函数复杂一些。它的原型如下：

```
int  fgetpos(FILE  *pfile,   fpos_t  *position);
```

第一个参数是文件指针。第二个参数是定义在<stdio.h>中的 fpos_t 指针。fpos_t 不是能记录文件中的每个位置的数组类型。它一般是一个整数类型，在这里它是 long 类型。读者可查看<stdio.h>头文件，了解 fpos_t 在自己的系统上的类型。

fgetpos()函数与稍后介绍的位置函数 fsetpos()一起使用。如果操作成功，函数 fgetpos()会在 position 中存储当前位置和文件状态信息，并返回 0，否则返回非零整数。用下面的语句声明一个 fpos_t 类型的变量：

```
fpos_t  here  =  0;
```

现在可以用如下语句记录文件中的当前位置：

```
fgetpos(pfile, &here);
```

这行语句将当前的文件位置记录到变量 here 中。

警告：

必须声明 fpos_t 类型的变量，而不要声明 fpos_t*类型的指针，因为它不会分配任何内存来存储位置数据。

12.11.3 在文件中设定位置

ftell()的互补函数是 fseek()，它的原型如下：

```
int  fseek(FILE  *pfile,  long offset, int origin);
```

第一个参数是要重新定位的文件指针。第二和第三个参数定义文件中要到达的位置。第二个参数是距离第三个参数指定的参考点的位移。参考点可以是以下 3 种情况：SEEK_SET 指定了文件的开头，SEEK_CUR 指定了文件的当前位置，SEEK_END 指定了文件的末尾。当然，这 3 个值都在头文件<stdio.h>中定义。对于文本文件，如果要避免丢失失败，第二个参数必须是 ftell()返回的值。第三个参数对于文本文件必须是 SEEK_SET。因此，对于文本文件模式，fseek()函数的所有操作都以文件的开头作为参考点。

对于二进制文件，offset 参数是一个相对的字节数。因此，参考点指定为 SEEK_CUR 时，可以给 offset 提供正数或负数。

与 fgetpos()函数配对的函数是 fsetpos()。它的原型相当简单：

```
int  fsetpos(FILE  *pfile,   fpos_t  *position);
```

www.TopSage.com

第一个参数是使用 fopen()打开的文件的指针，第二个参数是 fpos_t 类型的指针，它的值是调用 fgetpos()得到的。使用 fsetpos()函数的方法如下：

```
fsetpos(pfile, &here);
```

变量 here 是前面调用 fgetpos()设定的。与 fgetpos()函数一样，如果出错，fsetpos()就返回非零值。这个函数要使用 fgetpos()返回的值，所以只能用它得到之前文件中的某个位置，而 fseek()函数允许到达任何位置。

注意，seek 这个动词用来表示将磁盘的读/写头移动到文件中的特定位置。这就是 fseek()的名字来历。

例如，使用模式"rb+"或"wb+"打开要更新的文件，不论之前对这个文件执行了什么动作，在执行完定位函数 fsetpos()或 fseek()后，都可以安全地执行读写操作。

试试看：随机访问文件

要练习刚学到的文件处理技巧，可以修改前一章的程序，记录家庭成员的信息。因此，要创建一个含有所有家庭成员数据的文件，然后处理这个文件，输出每个成员及其双亲的数据。所使用的结构仅用于包含家庭中最少的成员。当然还可以扩展它，使之包含亲戚的信息。

首先是 main()函数：

```
/* Program 12.5 Investigating the family.*/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

/* Global Data */
const int NAME_MAX = 20;

struct
{
  char *filename;  /* Physical file name */
  FILE *pfile;     /* File pointer */
} global = {"C:\\myfile.bin", NULL };

/* Structure types */
struct Date       /* Structure for a date */
{
  int day;
  int month;
  int year;
};

typedef struct family /* Structure for family member */
{
```

www.TopSage.com

```
  struct Date dob;
  char name[NAME_MAX];
  char pa_name[NAME_MAX];
  char ma_name[NAME_MAX];
}Family;

/* Function prototypes */
bool get_person(Family *pfamily); /* Input function */
void getname(char *name);         /* Read a name */
void show_person_data(void);      /* Output function */
void get_parent_dob(Family *pfamily); /* Function to find pa & ma */

int main(void)
{
  Family member; /* Stores a family structure */

  if(!(global.pfile = fopen(global.filename, "wb")))
  {
    printf("\nUnable to open %s for writing.\n", global.filename);
    exit(1);
  }

  while(get_person(&member)) /* As long as we have input */
    fwrite(&member, sizeof member, 1, pfile); /* write it away */

  fclose(global.pfile); /* Close the file now its written */

  show_person_data();    /* Show what we can find out */

  if(remove(global.filename))
    printf("\nUnable to delete %s.\n", global.filename);
  else
    printf("\nDeleted %s OK.\n", global.filename);
  return 0;
}
```

代码的说明

在#include 语句后，有一些全局变量以及结构定义。这些在前面的例子中已经见过了。Family 不是变量，而是声明为 family 结构的类型名称。这样在声明 Family 类型的对象时，可以不使用关键字 struct。在结构声明后，有三个函数原型。

本例要练习文件定位函数，而不只是基本的文件读写操作。在 get_person()函数中一次获得一个人的输入信息，将得到的数据存储到 member 结构对象中。接收到数据后，将每个结构写入文件中，当 get_person()函数返回 0 时，就停止输入处理。

while 循环结束后，关闭输入文件，调用 show_person_data()函数。在这个函数中要使用文件位置的获取和设定函数。最后，使用 remove()函数从磁盘中删除文件。

输入函数 get_person()的代码如下：

www.TopSage.com

```
/* Function to input data on Family members */
bool get_person(Family *temp)
{
  static char more = '\0'; /* Test value for ending input */

  printf("\nDo you want to enter details of a%s person (Y or N)? ",
                  more != '\0'?"nother " : "" );
  scanf(" %c", &more);

  if(tolower(more) == 'n')
     return false;

  printf("\nEnter the name of the person: ");
  getname(temp->name);              /* Get the person's name */
  printf("\nEnter %s's date of birth (day month year); ", temp->name);
  scanf("%d %d %d", &temp->dob.day, &temp->dob.month, &temp->dob.year);

  printf("\nWho is %s's father? ", temp->name);
  getname(temp->pa_name);           /* Get the father's name */
  printf("\nWho is %s's mother? ", temp->name);
  getname(temp->ma_name);           /* Get the mother's name */
  return true;
}
```

这个函数相当简单，没有新的机制。它用一个变量 more 控制是否继续读入数据，该变量由第一个提示后的输入来设定。这个变量定义成静态变量，所以它的值在程序的多次 get_person()调用中保持不变。因此，可以在第二次和以后的迭代中正确提示稍有不同的信息。

如果没有数据可以输入，就在函数的第一个提示中键入 N 或 n，函数会返回 false。反之，就将数据输入适当的结构成员中，并返回 true。

读取姓名的 getname()函数如下：

```
/* Read a name from the keyboard */
void getname(char *name)
{
  fflush(stdin);              /* Skip whitespace */
  fgets(name, NAME_MAX, stdin);
  int len = strlen(name);
  if(name[len-1] == '\n') /* If last char is newline */
     name[len-1] = '\0'; /* overwrite it */
}
```

这个函数使用 fgets()读取姓名，允许输入中包含空白，还能确保不超过存储姓名的容量。如果输入超过了 NAME_MAX 个字符(包括终止字符)，就截断姓名。fgets()函数存储按下回车键而生成的换行符，所以当换行符是字符串中的最后一个字符时，就用终止字符替换它。

www.TopSage.com

下一个函数是为每一个人产生输出(包含父母亲的生日)。这个函数的代码如下:

```
/* Function to output data on people on file */
void show_person_data(void)
{
  Family member;       /* Structure to hold data from file */
  fpos_t current = 0;  /* File position */

  /* Open file for binary read */
  if(!(global.pfile = fopen(global.filename, "rb")))
  {
    printf("\nUnable to open %s for reading.\n", global.filename);
    exit(1);
  }

  /* Read data on person */
  while(fread(&member, sizeof member, 1, global.pfile))
  {
    fgetpos(global.pfile, &current); /* Save current position */
    printf("\n\n%s's father is %s, and mother is %s.",
              member.name, member.pa_name, member.ma_name);
    get_parent_dob(&member);          /* Get parent data */
    fsetpos(global.pfile, &current); /* Position file to read next */
  }
  fclose(global.pfile);              /* Close the file */
}
```

这个函数在文件中依序处理每个结构。声明一个 Family 类型的变量后，用下面的语句声明一个变量 current:

```
fpos_t current = 0; /* File position */
```

这个语句将 current 声明为 fpos_t 类型。该变量用于记录文件中的当前位置。函数 get_parent_dob()在函数的后面调用，它也访问这个文件。因此，在调用 get_parent_dob()函数之前，必须记录要读取的下一个结构的文件位置，以便在下次迭代中读取数据。

在以二进制读取模式打开文件后，就在一个循环中进行所有的处理:

```
while(fread(&member, sizeof member, 1, global.pfile))
```

这行语句在循环条件式中读取文件，使用函数 fread()返回的值判断循环是否继续。如果函数返回 1，循环就继续，若返回 0，循环就结束。

在循环内有这些语句:

```
fgetpos(global.pfile, &current); /* Save current position */
printf("\n\n%s's father is %s, and mother is %s.",
          member.name, member.pa_name, member.ma_name);
get_parent_dob(&member);         /* Get parent data */
fsetpos(global.pfile, &current); /* Position file to read next */
```

www.TopSage.com

首先存储文件的当前位置，然后显示 member 中存储的人的父母。之后调用函数 get_parent_dob()，在文件中搜索双亲的项。函数返回后，文件的位置就变成未知的了，所以调用 fsetpos()，移动到下一个要读取的结构上。处理完所有的结构后，就结束 while 循环，并关闭文件。

找出某个人的父母生日的函数如下：

```
/* Function to find parents' dates of birth. */
void get_parent_dob(Family *pmember)
{
  Family relative;    /* Stores a relative */
  int num_found = 0;  /* Count of relatives found */

  rewind(global.pfile); /* Set file to the beginning */

  /* Get the stuff on a relative */
  while(fread(&relative, sizeof(Family), 1, global.pfile))
  {
    if(strcmp(pmember->pa_name, relative.name) == 0) /*Is it pa? */
    { /* We have found dear old dad */
      printf("\n Pa was born on %d/%d/%d.",
             relative.dob.day, relative.dob.month, relative.dob.year);

      if(++num_found == 2) /* Increment parent count */
         return;           /* We got both so go home */
    }
    else
      if(strcmp(pmember->ma_name, relative.name) == 0) /*Is it ma? */
      { /* We have found dear old ma */
         printf("\n Ma was born on %d/%d/%d.",
            relative.dob.day, relative.dob.month, relative.dob.year);

      if(++num_found == 2)  /* Increment parent count */
          return;           /* We got both so go home */
    }
  }
}
```

文件已被调用程序打开，所以在处理之前，只需使用 rewind()函数，回到文件的开头。然后依序读取文件，用父母亲的名字匹配每个读取的结构。搜寻父亲的机制使用了下列语句：

```
if(strcmp(pmember->pa_name, relative.name) == 0) /*Is it pa? */
{ /* We have found dear old dad */
  printf("\n Pa was born on %d/%d/%d.",
       relative.dob.day, relative.dob.month, relative.dob.year);

  if(++num_found == 2) /* Increment parent count */
     return; /* We got both so go home */
```

www.TopSage.com

```
}
```

pmember 指向的这个人的父亲名字和结构对象 relative 的成员名进行比较。如果检查失败，这个函数就继续对母亲做相同的检查。

如果找到双亲，就显示其生日信息。将一个计数存储在找到父母亲的计数器 num_found 中，如果父母亲都找到了，就结束这个函数。函数在读完文件中所有的结构后结束。

要执行这个程序，必须将 main()和其他函数合并到一个文件中，然后编译并执行。当然，这个例子也可以使用 ftell()和 fseek()作为定位函数。

如本章的前几个例子所示，程序使用了特定的文件名，因为这些例子都假设程序在运行时，这个文件不存在。在 C 语言中可以创建临时文件，来避开这个问题。

12.12 使用临时文件

程序执行时，常需要一个工作文件来存储中间结果，程序结束后，就删除它。本章计算质数的程序就是一个例子，文件仅在计算过程中需要。

使用临时文件的函数有两个，它们各有优缺点。

12.12.1 创建临时文件

第一个函数会自动创建临时文件。它的原型如下：

```
FILE  *tmpfile(void);
```

这个函数没有参数，返回临时文件的指针。如果因某种原因不能创建这个文件，例如磁盘满了，这个函数会返回 NULL。这个文件会以更新方式创建并打开，所以可以读写它。这个文件在程序结束后会自动删除，所以不需要任何整理操作。我们永远不知道这个文件叫什么。

这个函数的缺点是文件在关闭时被删除。因此，不能在程序的某个部分写完数据后关闭它，然后在程序的另一部分重新打开它，以读取数据。只要还需访问数据，就必须使这个文件处于打开状态。以下的语句就创建了一个临时文件：

```
FILE pfile;          /* File pointer */
pfile = tmpfile();  /* Get pointer to temporary file */
```

12.12.2 创建唯一的文件名

第二个方法是使用一个可以提供唯一文件名的函数，这个临时的文件名由程序员指定。函数的原型如下：

www.TopSage.com

```
char  *tmpnam(char  *filename);
```

如果传给函数的变元是 NULL，文件名会在内部的静态对象中生成，并返回该对象的指针。如果文件名应存储在自己声明的 char 数组中，它的长度就至少是 L_tmpnam 个字符，L_tmpnam 是一个定义在<stdio.h>中的常量。此时，这个函数的变元就是存储在数组中的文件名，函数返回指向数组的指针。如果函数不能创建唯一的文件名，就返回 NULL。

对于第一种情况，可以用以下语句创建唯一的文件：

```
FILE *pFile = NULL;
char *filename = tmpnam(NULL);
if(filename != NULL)
   pfile = fopen(filename, "wb+");
```

这里声明了文件指针 pfile 和指针 filename，filename 用 tmpnam()函数返回的临时文件名的地址来初始化。由于 tmpnam()的变元是 NULL，所以文件名生成为一个内部静态对象，其地址放在指针 filename 中。只要 filename 不是 NULL，就调用 fopen()，用"wb+"模式创建文件。当然，也可以创建临时文本文件。

千万不能编写如下语句：

```
pfile = fopen(tmpnam(NULL), "wb+"); /* Wrong!! */
```

tmpnam()函数可能返回 NULL，而且不能再访问该文件名，所以也不能使用 remove()删除这个文件。

如果要创建一个数组来包含文件名，可以使用下面的语句：

```
FILE *pfile = NULL;
char filename[L_tmpnam];
if(tmpnam(filename) != NULL)
   pfile = fopen(filename, "wb+");
```

这个标准库函数只提供了唯一的文件名，程序员必须自己删除文件。

注意：

这个函数在程序中能创建唯一名字的数目是有限的，最大值由<stdio.h>中定义的 TMP_MAX 指定。

12.13 更新二进制文件

有 3 个打开模式可用于更新二进制文件：

- 模式"r+b"(也可以写做"rb+")打开已有的二进制文件，以进行读写。使用这个打开模式，可以读写文件中任意位置的数据。
- 模式"w+b"(也可以写做"wb+")将已有二进制文件的长度截断为 0，删除其内容。接着可以执行读写操作，但由于文件的长度是 0，所以必须先写入一些数据，才能读取文件，如果文件不存在，在用"w+b"模式调用 fopen()时，会创建一个新文件。

www.TopSage.com

- 第三个模式"a+b"(或"ab+")打开已有的文件，进行更新。这个模式只允许在文件末尾执行写入操作。

可以用两种方式以更新二进制文件的打开模式写入数据，但最好总是在模式的末尾加上“+”，因为“+”非常重要，它意味着更新。下面的例子使用"wb+"模式创建一个新文件，然后使用其他模式进行更新。

试试看：用更新模式写入二进制文件

文件包含人们的姓名和年龄，这些数据从键盘上读取。姓名存储为一个字符串，其中包含姓氏和名字。代码给 C 盘的 temp 目录指定了完整的文件路径，读者应检查自己的系统，确保有这个路径，或者修改该路径，使之满足本例的要求。注意，如果路径中的目录不存在，程序就会失败。下面是本例的代码:

```
/* Program 12.6 Writing a binary file with an update mode */
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

const int MAXLEN = 30;          /* Size of name buffer */

void listfile(char *filename); /* List the file contents */

int main(void)
{
  const char *filename = "C:\\temp\\mydata.bin";
  char name[MAXLEN];            /* Stores a name */
  size_t length = 0;            /* Length of a name */
  int age = 0;                  /* Person's age */
  char answer = 'y';

  FILE *pFile = fopen(filename, "wb+");

  do
  {
    fflush(stdin);              /* Remove whitespace */

    printf("\nEnter a name less than %d characters:", MAXLEN);
    gets(name);                 /* Read the name */

    printf("Enter the age of %s: ", name);
    scanf(" %d", &age);         /* Read the age */

    /* Write the name & age to file */
    length = strlen(name);                          /* Get name length */
    fwrite(&length, sizeof(length), 1, pFile);  /* Write name length */
    fwrite(name, sizeof(char), length, pFile);  /* then the name */
    fwrite(&age, sizeof(age), 1, pFile);        /* then the age */
```

www.TopSage.com

```
  printf("Do you want to enter another(y or n)? " );
  scanf("\n%c", &answer);
} while(tolower(answer) == 'y');

  fclose(pFile);       /* Close the file */

  listfile(filename); /* List the contents */
  return 0;
}

/* List the contents of the binary file */
void listfile(char *filename)
{
  size_t length = 0;       /* Name length */
  char name[MAXLEN];       /* Stores a name */
  int age = 0;
  char format[20];         /* Format string */

  /* Create the format string for names up to MAXLEN long */
  sprintf(format, "\n%%-%ds Age:%%4d", MAXLEN);

  FILE *pFile = fopen(filename, "rb"); /* Open to read */
  printf("\nThe contents of %s are:", filename);

  /* Read records as long as we read a length value */
  while(fread(&length, sizeof(length), 1, pFile) == 1)
  {
    if(length+1>MAXLEN)
    {
      printf("\nName too long.");
      exit(1);
    }
    fread(name, sizeof(char), length, pFile); /* Read the name */
    name[length] = '\0';                      /* Append terminator */
    fread(&age, sizeof(age), 1, pFile);       /* Read the age */
    printf(format, name, age);                /* Output the record */
  }
 fclose(pFile);
}
```

程序的输出如下：

```
Enter a name less than 30 characters:Bill Bloggs
Enter the age of Bill Bloggs: 21
Do you want to enter another(y or n)? y

Enter a name less than 30 characters:Yolande Dogsbreath
Enter the age of Yolande Dogsbreath: 27
Do you want to enter another(y or n)? y
```

www.TopSage.com

```
Enter a name less than 30 characters:Ned Nudd
Enter the age of Ned Nudd: 33
Do you want to enter another(y or n)? y

Enter a name less than 30 characters:Binkie Huckerback
Enter the age of Binkie Huckerback: 18
Do you want to enter another(y or n)? y

Enter a name less than 30 characters:Mary Dunklebiscuit
Enter the age of Mary Dunklebiscuit: 29
Do you want to enter another(y or n)? n

The contents of C:\temp\mydata.bin are:
Bill Bloggs                    Age: 21
Yolande Dogsbreath             Age: 27
Ned Nudd                       Age: 33
Binkie Huckerback              Age: 18
Mary Dunklebiscuit             Age: 29
```

代码的说明

将模式指定为"rb+"，打开文件，以更新二进制文件。在这个模式下，文件内容会被覆盖，因为文件的长度会被截断为 0。如果该文件不存在，就创建一个。数据从键盘上读取，在 do-while 循环中把数据写入文件。循环中的第一条语句刷新 stdin:

```
fflush(stdin); /* Remove whitespace */
```

这是必需的，因为在循环条件中读取单个字符的操作，会在除第一次迭代之外的所有迭代中，将一个换行符留在 stdin 中。如果不删除这个字符，读取姓名的操作就不能正确执行，它会把换行符读取为一个空的姓名字符串。

从键盘上读取姓名和年龄后，就使用如下语句将这些信息写入文件，作为二进制数据:

```
length = strlen(name);                          /* Get name length */
fwrite(&length, sizeof(length), 1, pFile); /* Write name length */
fwrite(name, sizeof(char), length, pFile); /* then the name */
fwrite(&age, sizeof(age), 1, pFile);            /* then the age */
```

姓名的长度各不相同，这有两种处理方式。每次可以把整个姓名数组写入文件，不考虑姓名字符串的长度。这种方法的编码很简单，但文件中有许多多余的数据。本例采用另一种方法。在姓名的前面先写入该姓名字符串的长度，这样在读取文件时，首先读取该长度，再从文件中读取该数量的字符，作为姓名。注意，'\0'终止字符串不写入文件，因此在读取文件时，要把它添加到每个姓名字符串的末尾。

这个循环允许在文件中添加任意个记录，因为只要在提示时输入 Y 或 y，循环就会继续。循环结束时，关闭文件，调用 listfile()函数，在 stdout 上列出文件的内容。

www.TopSage.com

listfile()函数用模式"rb"打开文件，以执行二进制读取操作。在这个模式下，文件指针定位在文件的开头，而且只能读取文件。

姓名的最大长度由 MAXLEN 符号指定，所以可以使用%-MAXLENs 格式输出姓名，它会使姓名左对齐，且字符宽度是姓名字符串的最大字符数。这样姓名就会排列整齐，总是能放在字段中。当然，不能把 MAXLEN 放在格式字符串中，因为 MAXLEN 符号中的字母会被解释为一系列字母，而不是一个符号的值。为了获得需要的结果，listfile()函数使用 sprintf()函数写入 format 数组，以创建格式字符串：

```
sprintf(format, "\n%%-%ds Age:%%4d", MAXLEN);
```

sprintf()函数与 printf()类似，但其输出会写入其第一个参数指定的 char 数组中。因此，这个操作会使用如下格式字符串把 MAXLEN 的值写入 format 数组中：

```
"\n%%-%ds Age:%%4d"
```

在换行符\n 的后面，%%在输出中指定一个%符号。之后是-和用%d 指定符格式化的 MAXLEN 值。然后是 s、一个空格和 Age:。最后是一个%字符和 4d。MAXLEN 符号定义为 30，所以执行了 sprintf()函数后，format 数组包含如下字符串：

```
"\n%-30s Age:%d"
```

在 while 循环中读取文件，将其内容列在 stdout 上，该循环用从文件中读取姓名长度的表达式的值控制：

```
while(fread(&length, sizeof(length), 1, pFile) == 1)
{
...
}
```

fread()函数将一个 sizeof(length)字节读入&length 指定的位置。如果该操作成功，fread()函数就返回读取的项数，但如果到达文件末尾，该函数返回的数就小于请求的数，因为不再有要读取的数据了。因此，到达文件末尾时，循环就结束。

确定是否到达文件末尾的另一种方式是编写如下循环：

```
while(true)
{
  fread(&length, sizeof(length), 1, pFile);
  /* Now check for end of file */
  if(feof(pFile))
    break;
  ...
}
```

feof()函数为变元指定的流测试文件尾指示器，如果设置了该指示器，就返回非零值。因此在到达文件尾时，就执行 break，结束循环。

www.TopSage.com

从文件中读取了长度值后，就使用下面的语句检查是否有足够的空间容纳该姓名：

```
if(length+1>MAXLEN)
{
  printf("\nName too long.");
  exit(1);
}
```

文件中的姓名没有终止字符'\0'，所以必须在 name 数组中加上它。因此，比较 length+1 和 MAXLEN。

用下面的语句从文件中读取姓名和年龄：

```
fread(name, sizeof(char), length, pFile);  /* Read the name */
name[length] = '\0';                       /* Append terminator */
fread(&age, sizeof(age), 1, pFile);        /* Read the age */
```

最后在循环中，使用通过 sprintf()函数创建的格式字符串，将姓名和年龄写入 stdout：

```
printf(format, name, age); /* Output the record */
```

12.13.1 修改文件的内容

扩展上一个例子，以使用另外两个二进制更新模式。本节要添加功能，以更新文件中的已有记录，添加记录或删除文件。这个程序相当复杂，所以最好将其操作分解到函数中。文件仍包含姓名记录，这样由姓名和年龄组成的记录的长度就互不相同。在修改文件的内容时，可以看到因此带来的复杂性。

为了突出本节的主题，先看看程序的大纲。该程序包含如下 9 个函数：

main()：控制程序的整体操作，允许用户从一系列文件操作中选择。

listfile()：将文件的内容输出到 stdin 中。

writefile()：在两种模式下操作，将从 stdin 中读取的记录写入新文件，或把记录追加到已有的文件中。

getrecord()：从 stdin 中读取记录。

getname()：从 stdin 中读取姓名。

writerecord()：将记录写入文件。

readrecord()：从文件中读取记录。

findrecord()：在文件中查找姓名与输入匹配的记录。

duplicatefile()：重新创建文件来包含一个更新的记录。当新记录与原记录的长度不同时，这个函数用于更新记录。

图 12-5 显示了应用程序中调用函数的层次结构。

www.TopSage.com

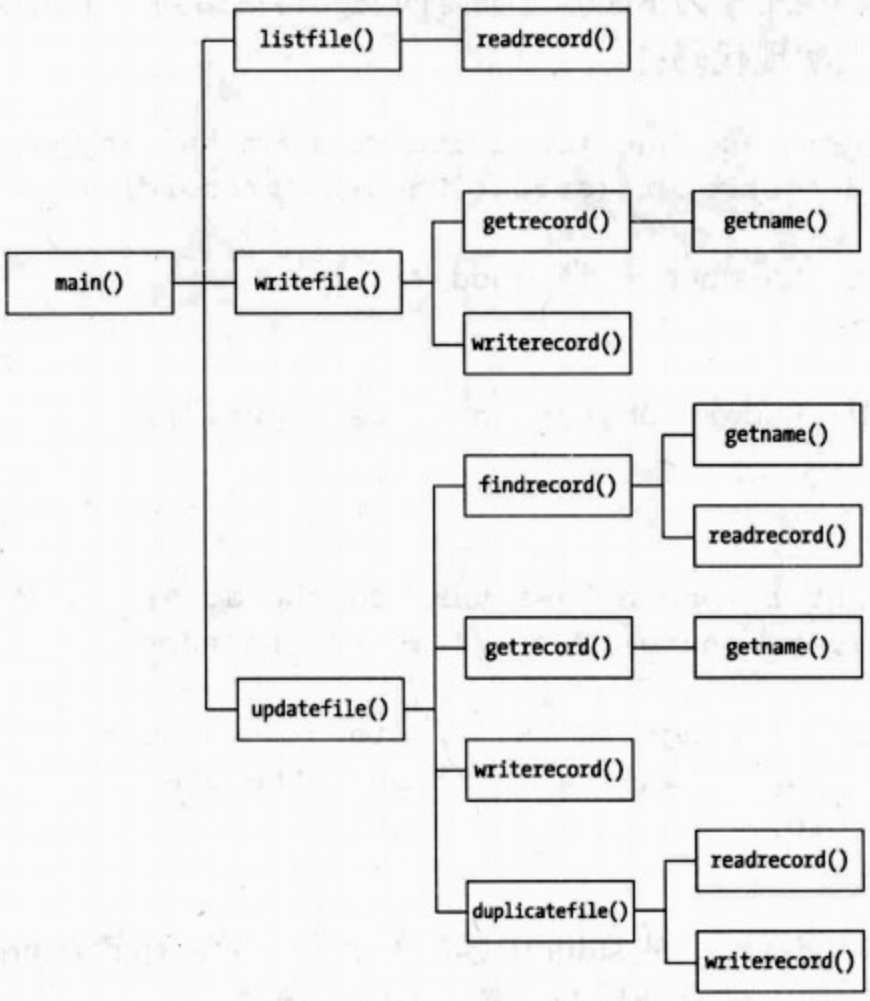


图 12-5　程序 12.8 调用的函数的层次结构

main()调用的 3 个函数实现了程序的基本功能。它们右边的函数有助于简化 3 个主要函数的功能。如果定义一个结构，在函数之间传送姓名和年龄，就可以简化代码。

```
struct Record
{
  char name[MAXLEN];
  int age;
};
```

可以将 Record 类型的对象写入文件，但需要每次写入包含 MAXLEN 个元素的 name 数组，使文件为字符数少于 MAXLEN 的姓名包含大量多余的字节。然而，这个结构非常便于将姓名和相关的年龄值传送给函数。新例子中有几个函数，下面先介绍每个函数的代码，再把它们组合起来。可以把下面几小节的函数代码组合到一个源文件中。

12.13.2　从键盘读取记录

编写一个函数，从 stdin 中读取姓名字符串和年龄值，将它们存储在 Record 对象中。该函数的原型如下：

```
struct Record *getrecord(struct Record *precord);
```

这个函数的参数是指向已有 Record 结构对象的指针，其返回值是该对象的地址。因

www.TopSage.com

此，可以把该函数的调用作为 Record *类型的变元传送给另一个函数。

下面是该函数的实现代码：

```
/* Read the name and age for a record from the keyboard */
struct Record *getrecord(struct Record *precord)
{
  /* Verify the argument is good */
  if(!precord)
  {
    printf("No Record object to store input.");
    return NULL;
  }

  printf("\nEnter a name less than %d characters:", MAXLEN);
  getname(precord->name);      /* readf the name */

  printf("Enter the age of %s: ", precord->name);
  scanf(" %d", &precord->age); /* Read the age */
  return precord;
}
```

这是一个很简单的操作，从 stdin 中读取姓名和年龄，存储到 precord 指向的 Record 对象的相应成员中。读取姓名的辅助函数 getname()如下：

```
/* Read a name from the keyboard */
void getname(char *pname)
{
  fflush(stdin);
  fgets(pname, MAXLEN, stdin); /* Read the name */
  int len = strlen(pname);
  if(pname[len-1] == '\n')     /* if there's a newline */
     pname[len-1] = '\0';      /* overwrite it */
}
```

getname()中唯一略复杂的部分是需要处理 fgets()函数中存储的'\n'。如果输入的字符数超过了 MAXLEN，'\n'就留在输入缓冲区中，不存储到 pname 指向的数组中。程序需要在多个地方读取姓名，所以把该操作打包到 getname()函数中比较方便。

12.13.3 将记录写入文件

现在，定义一个函数，将 Record 对象的成员写入 FILE *指针指向的文件中，该函数的原型如下：

```
void writerecord(struct Record *precord, FILE *pFile);
```

第一个参数是指向 Record 结构的指针，该结构的姓名和年龄成员要写入文件。第二个参数是文件指针。

这个函数的实现代码如下：

www.TopSage.com

```
/* Write a new record to the file at the current position */
void writerecord(struct Record *precord, FILE *pFile)
{
  /* Verify the arguments are good */
  if(!precord)
  {
    printf("No Record object to write to the file.");
    return;
  }
  if(!pFile)
  {
    printf("No stream pointer for the output file.");
    return;
  }

  /* Write the name & age to file */
  size_t length = strlen(precord->name);          /* Get name length */
  fwrite(&length, sizeof(length), 1, pFile);     /* Write name length */
  fwrite(precord->name, sizeof(char), length, pFile); /* then the name */
  fwrite(&precord->age, sizeof(precord->age), 1, pFile);/* then the age */
}
```

这个函数检查文件指针是否存在，并将数据写入文件的当前位置。因此，调用它的函数要确保文件已经以正确的模式打开，且文件位置的设置是正确的。该函数首先将字符串的长度写入文件，之后写入字符串，但要去掉终止字符'\0'。这样读取文件的代码才能先确定姓名字符串有多少个字符。最后将年龄值写入文件中。

12.13.4 从文件中读取记录

下面是从文件中读取一个记录的函数的原型：

```
struct Record *readrecord(struct Record *precord, FILE *pFile);
```

要读取的文件用第二个参数指定，即文件指针。为了方便，返回值是作为第一个参数传送的地址。

readrecord()函数的实现代码如下：

```
/* Reads a record from the file at the current position */
struct Record * readrecord(struct Record *precord, FILE *pFile)
{
  /* Verify the arguments are good */
  if(!precord)
  {
    printf("No Record object to store data from the file.");
    return NULL;
  }
  if(!pFile)
  {
```

www.TopSage.com

```
    printf("No stream pointer for the input file.");
    return NULL;
  }

  size_t length = 0;                              /* Name length */
  fread(&length, sizeof(length), 1, pFile);       /* Read the length */
  if(feof(pFile))                                 /* If it's end file */
     return NULL;                                 /* return NULL */

  /* Verify the name can be accommodated */
  if(length+1>MAXLEN)
  {
    fprintf(stderr, "\nName too long. Ending program.");
    exit(1);
  }

  fread(precord->name, sizeof(char), length, pFile); /* Read the name */
  precord->name[length] = '\0';                      /* Append terminator */
  fread(&precord->age, sizeof(precord->age), 1, pFile); /* Read the age */

  return precord;
}
```

与 writerecord()函数相同，readrecord()函数也假定文件已经用正确的模式打开，并尝试从当前位置读取记录。每个记录都以长度值开头。当然，文件位置可以是文件的末尾，所以在读取操作后，要通过文件指针调用 feof()，检查 EOF。如果到达了文件尾，feof()函数就返回一个非零整数，此时应返回 NULL，以告诉调用函数，到达了 EOF。

接着，该函数检查姓名的长度是否超过了 name 数组的长度。如果是，就在标准错误流中输出一条信息，之后结束程序。

如果一切正常，就从文件中读取姓名和年龄，并存储在 record 对象的成员中。必须将'\0'添加到姓名字符串的末尾，以避免以后处理字符串时出现灾难性的后果。

12.13.5 写入文件

下面是将任意个记录写入文件的函数原型，其中记录是从键盘上输入的：

```
void writefile(char *filename, char *mode);
```

第一个参数是要写入的文件名，这表示该函数要打开文件。第二个参数是要使用的文件打开模式。将模式指定为"wb+"，writefile()函数就会删除文件的原始内容，再将数据写入文件。如果文件不存在，就用指定的名称创建一个文件。如果模式指定为"ab+"，记录就追加到已有的文件中，如果文件不存在，就创建一个新文件。

该函数的实现代码如下：

```
/* Write to a file */
void writefile(char *filename, char *mode)
```

www.TopSage.com

```
{
  char answer = 'y';

  FILE *pFile = fopen(filename, mode); /* Open the file */
  if(pFile == NULL)                    /* Verify file is open */
  {
    fprintf(stderr, "\n File open failed.");
    exit(1);
  }
  do
  {
    struct Record record; /* Stores a record name & age */

    writerecord(getrecord(&record), pFile);
    /* Get record & write the file */

    printf("Do you want to enter another(y or n)? " );
    scanf("\n%c", &answer);
    fflush(stdin);                          /* Remove whitespace */
  } while(tolower(answer) == 'y');

  fclose(pFile);                            /* Close the file */
}
```

用第二个参数指定的模式打开文件后，该函数就会在 do-while 循环中将数据写入文件。从 stdin 中读取数据和写入文件在一个语句中完成，该语句调用了 writerecord()，并把 getdata()的调用作为第一个参数。getdata()返回的指向 Record 对象的指针直接传送给 writerecord()函数。用户输入 n 或 N 时，表示没有要输入的数据了，所以结束操作，关闭文件，退出函数。

12.13.6 列出文件内容

将文件中的记录输出到标准输出流中的函数原型如下：

```
void listfile(char *filename);
```

其参数是文件名，所以函数要先打开文件，操作完成后关闭它。下面是其实现代码：

```
/* List the contents of the binary file */
void listfile(char *filename)
{
  /* Create the format string for names up to MAXLEN long */
  /* format array length allows up to 5 digits for MAXLEN */
  char format[15];                        /* Format string */
  sprintf(format, "\n%%-%ds Age:%%4d", MAXLEN);

  FILE *pFile = fopen(filename, "rb"); /* Open file to read */
  if(pFile == NULL)                    /* Check file is open */
```

www.TopSage.com

```
    {
      printf("Unable to open %s. Verify it exists.\n", filename);
      return;
    }
    struct Record record; /* Stores a record */
    printf("\nThe contents of %s are:", filename);

    while(readrecord(&record, pFile) != NULL) /* As long as we have records */
       printf(format, record.name, record.age); /* Output the record */
    printf("\n");                               /* Move to next line */

    fclose(pFile);                              /* Close the file */
}
```

这个函数生成一个格式字符串，将姓名的字段宽度调整为 MAXLEN 个字符。sprintf()函数将格式字符串写入 format 数组。

文件以二进制读取模式打开，所以初始位置在文件的开头。如果文件成功打开，就在 while 循环中调用前面定义的 readrecord()函数，从文件中读取记录。readrecord()函数的调用放在循环条件中，所以返回 NULL 就表示到达文件尾，结束循环。在循环中，使用前面创建的 format 数组中的字符串，将 readrecord()读取的 Record 对象的成员写入 stdout。读取了所有的记录后，就通过文件指针调用 fclose()函数，关闭文件。

12.13.7 更新已有的文件内容

更新文件中的已有记录增加了复杂性，因为文件中的姓名有不同的长度。不能简单地覆盖已有的记录，因为用于替换的记录可能在原来的空间中放不下。如果新记录的长度与原记录相同，就可以覆盖它，如果不同，就只能将数据写入一个新文件。下面是更新文件的函数的原型：

```
void updatefile(char *filename);
```

唯一的参数是文件名，所以该函数会查找出要更新的记录，并打开和关闭文件。下面是代码：

```
/* Modify existing records in the file */
void updatefile(char *filename)
{ char answer = 'y';

  FILE *pFile = fopen(filename, "rb+"); /* Open the file for update */
  if(pFile == NULL)                     /* Check file is open */
  {
    fprintf(stderr, "\n File open for updating records failed.");
    return;
  }
  struct Record record;                  /* Stores a record */
  int index = findrecord(&record, pFile); /* Find the record for a name */
```

www.TopSage.com

```
  if(index<0)                          /* If the record isn't there */
  {
    printf("\nRecord not found.");     /* ouput a message */
    return;                            /* and we are done. */
  }

  printf("\n%s is aged %d,", record.name, record.age);
  struct Record newrecord;             /* Stores replacement record */
  printf("\nYou can now enter the new name and age for %s.", record.name);
  getrecord(&newrecord);               /* Get the new record */
  /* Check if we can update in place */
  if((strlen(record.name) == strlen(newrecord.name)))
  { /* Name lengths are the same so we can */
    /* Move to start of old record */
    fseek(pFile,
        -(long)(sizeof(size_t)+strlen(record.name)+sizeof(record.age)),
        SEEK_CUR);
    writerecord(&newrecord, pFile); /* Write the new record */
    fflush(pFile);                   /* Force the write */
  }
  else
    duplicatefile(&newrecord, index, filename, pFile);

  printf("File update complete.\n");
}
```

这个函数包含许多代码。但其步骤相当简单：

(1) 打开要更新的文件。

(2) 找到要更新的记录的索引(第一个记录的索引是 0)。

(3) 获取记录的数据，以替换旧记录。

(4) 检查记录是否可以替换。当姓名的长度相同时，就可以替换记录。此时要将当前位置往回移动旧记录的长度，将新记录写入旧文件。

(5) 如果姓名的长度不同，就复制文件，在复制文件中用新记录替换旧记录。

(6) 打开要更新的文件后，函数就为要更新的记录读取姓名。稍后探讨的 findrecord() 函数为要更新的记录读取姓名，然后返回该记录(假定该记录存在)的索引值(第一个记录的索引是 0)。如果记录不存在，findrecord()函数就返回 - 1。

如果新旧姓名的长度相同，就调用 fseek()，将文件位置往回移动旧记录的长度。接着将新记录写入文件，刷新输出缓冲区。调用 fflush()会迫使新记录传送到文件中。

如果新旧姓名的长度不同，就调用 duplicatefile()，复制文件，在复制文件中用新记录替换旧记录。该函数的实现代码如下：

```
/* Duplicate the existing file replacing the record to be update */
/* The record to be replaced is index records from the start     */
void duplicatefile(struct Record *pnewrecord, int index,
                   char *filename, FILE *pFile)
{
```

www.TopSage.com

```
/* Create and open a new file */
char tempname[L_tmpnam];
if(tmpnam(tempname) == NULL)
{
  printf("\nTemporary file name creation failed.");
  return;
}
char tempfile[strlen(dirpath)+strlen(tempname)+1];
strcpy(tempfile, dirpath);  /* Copy original file path */
strcat(tempfile, tempname); /* Append temporary name */
FILE *ptempfile = fopen(tempfile, "wb+");

/* Copy first index records from old file to new file */
rewind(pFile);              /* Old file back to start */
struct Record record;       /* Store for a record */
for(int i = 0 ; i<index ; i++)
 writerecord(readrecord(&record, pFile), ptempfile);

writerecord(pnewrecord, ptempfile);  /* Write the new record */
readrecord(&record,pFile);           /* Skip the old record */

/* Copy the rest of the old file to the new file */
while(readrecord(&record,pFile))
 writerecord(&record, ptempfile);

/* close the files */
if(fclose(pFile)==EOF)
  printf("\n Failed to close %s", filename);
if(fclose(ptempfile)==EOF)
  printf("\n Failed to close %s", tempfile);

if(!remove(filename)) /* Delete the old file */
{
  printf("\nRemoving the old file failed. Check file in %s", dirpath);
  return;
}
/* Rename the new file same as original */
if(!rename(tempfile, filename))
  printf("\nRenaming the file copy failed. Check file in %s", dirpath);
}
```

这个函数通过以下步骤完成更新：

(1) 在旧文件所在的目录下创建名称唯一的新文件。dirpath 是一个全局变量，包含原文件的路径。

(2) 将要更新的记录之前的所有记录从旧文件复制到新文件中。

(3) 将新记录写入新文件，跳过旧文件中要更新的记录。

(4) 将其他记录从旧文件写入新文件。

(5) 关闭新旧两个文件。

www.TopSage.com

(6) 删除旧文件，把新文件重命名为旧文件名。

使用 tmpnam()生成的名字创建新文件后，就把记录从旧文件复制到新文件中，只有要更新的记录例外，它在新文件中用新记录替换。前 index 个记录的复制在 for 循环中完成，在该循环中，读取旧文件的 readrecord()返回一个指针，该指针作为参数传送给写入新记录的 writerecord()函数。要更新的记录后面的记录在 while 循环中复制。这个循环继续复制记录，直到到达旧文件的末尾为止。最后关闭两个文件，删除旧文件，将新文件重命名为旧文件名。如果希望操作更安全，可以重命名旧文件，而不是删除它，例如在已有的文件名后面加上"_old"。之后就可以重命名新文件了。这样，目录中有一个备份文件，如果更新出了问题，就可以使用该备份。

updatefile()调用 findrecord()函数，查找与输入的姓名匹配的记录的索引，findrecord()函数的实现代码如下：

```
/* Find a record                                    */
/* Returns the index number of the record  */
/* or -1 if the record is not found.          */
int findrecord(struct Record *precord, FILE *pFile)
{
  char name[MAXLEN];
  printf("\nEnter the name for the record you wish to find: ");
  getname(name);

  rewind(pFile);    /* Make sure we are at the start */
  int index = 0;    /* Index of current record */

  while(true)
  {
    readrecord(precord, pFile);
    if(feof(pFile))  /* If end-of-file was reached */
      return -1;     /* record not found */
    if(!strcmp(name, precord->name))
      break;
    ++index;
  }
  return index;      /* Return record index */
}
```

这个函数为要更新的记录读取姓名，然后读取文件中的记录，查找与所输入姓名匹配的姓名。如果到达文件尾时未找到该姓名，就返回 - 1，告诉调用程序，这个记录不在文件中。如果找到了匹配的姓名，函数就返回匹配记录的索引值。

现在可以合并完成整个例子了。

试试看：读取、写入和更新二进制文件

这里不重复前面介绍的函数。把下面的代码添加到包含函数代码的源文件开头，而不是添加到 main()函数的开头：

www.TopSage.com

```
/* Program 12.7 Writing, reading and updating a binary file */
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

const int MAXLEN = 30;                  /* Size of name buffer */
const char *dirpath = "C:\\temp\\";     /* Directory path for file */
const char *file = "mydata.bin";        /* File name */

/* Structure encapsulating a name and age */
struct Record
{
  char name[MAXLEN];
  int age;
};

void listfile(char *filename);    /* List the file contents */
void updatefile(char *filename); /* Update the file contents */
struct Record *getrecord(struct Record *precord);
/* Read a record from stdin */
void getname(char *pname);                      /* Read a name from stdin */
void writefile(char *filename, char *mode);  /* Write records to a file */
void writerecord(struct Record *precord, FILE *pFile);
struct Record * readrecord(struct Record *precord, FILE *pFile);
int findrecord(struct Record *precord, FILE *pFile);
void duplicatefile(struct Record *pnewrecord,
                   int index, char *filename, FILE *pFile);

int main(void)
{
  char filename[strlen(dirpath)+strlen(file)+1]; /* Stores file path */
  strcpy(filename, dirpath);                     /* Copy directory path */
  strcat(filename, file);                         /* and append file name */

  /* Choose activity option */
  char answer = 'q';
  while(true)
  {
    printf("\nChoose from the following options:"
           "\nTo list the file contents enter L"
           "\nTo create a new file enter C"
           "\nTo add new records enter A"
           "\nTo update existing records enter U"
           "\nTo delete the file enter D"
           "\nTo end the program enter Q\n : ");
    scanf("\n%c", &answer);
```

www.TopSage.com

```
switch(tolower(answer))
{
  case 'l':          /* List file contents */
    listfile(filename);
    break;
  case 'c':          /* Create new file */
    writefile(filename,"wb+");
    printf("\nFile creation complete.");
    break;
  case 'a':          /* Append records */
    writefile(filename, "ab+");
    printf("\nFile append complete.");
    break;
  case 'u':          /* Update existing records */
    updatefile(filename);
    break;
  case 'd':
    printf("Are you sure you want to delete %s (y or n)? ", filename);
    scanf("\n%c", &answer);
    if(tolower(answer) == 'y')
      remove(filename);
    break;
  case 'q':          /* Quit the program */
    printf("\nEnding the program.", filename);
    return 0;
      default:
        printf("Invalid selection. Try again.");
        break;
      }
  }
  return 0;
}
```

使用程序提供的主要选项，得到的输出如下：

```
Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
: c

Enter a name less than 30 characters:Bill Bloggs
Enter the age of Bill Bloggs: 22
Do you want to enter another(y or n)? y

Enter a name less than 30 characters:Kitty Malone
```

www.TopSage.com

```
Enter the age of Kitty Malone: 23
Do you want to enter another(y or n)? n
File creation complete.

Choose from the following options:
To list the file contents enter   L
To create a new file enter        C
To add new records enter          A
To update existing records enter  U
To delete the file enter          D
To end the program enter          Q
: l

The contents of C:\temp\mydata.bin are:
Bill Bloggs                    Age: 22
Kitty Malone                   Age: 23

Choose from the following options:
To list the file contents enter    L
To create a new file enter         C
To add new records enter           A
To update existing records enter   U
To delete the file enter           D
To end the program enter           Q
: a

Enter a name less than 30 characters:Jack Flash
Enter the age of Jack Flash: 30
Do you want to enter another(y or n)? n
File append complete.

Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
: l

The contents of C:\temp\mydata.bin are:
Bill Bloggs                    Age: 22
Kitty Malone                   Age: 23
Jack Flash                     Age: 30

Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
```

www.TopSage.com

```
To delete the file enter          D
To end the program enter          Q
: u

Enter the name for the record you wish to find: Kitty Malone

Kitty Malone is aged 23,
You can now enter the new name and age for Kitty Malone.
Enter a name less than 30 characters:Kitty Moline
Enter the age of Kitty Moline: 24
File update complete.

Choose from the following options:
To list the file contents enter   L
To create a new file enter        C
To add new records enter          A
To update existing records enter  U
To delete the file enter          D
To end the program enter          Q
: l

The contents of C:\temp\mydata.bin are:
Bill Bloggs                       Age: 22
Kitty Moline                      Age: 24
Jack Flash                        Age: 30

Choose from the following options:
To list the file contents enter   L
To create a new file enter        C
To add new records enter          A
To update existing records enter  U
To delete the file enter          D
To end the program enter          Q
: q
```

代码的说明

main()函数有许多代码，但它非常简单。全局字符串 dirpath 和 file 指定了目录和包含数据的文件名，它们在 main()函数中与 filename 存储的结果连接起来。

无限 while 循环提供了一系列选项，输入的选择在 switch 语句中确定。根据输入的字符，调用前面为程序开发的一个函数。输入 Q 或 q，就结束程序。

12.14 文件打开模式小结

掌握文件打开模式字符串需要一定的练习。表 12-2 总结了这些字符串，以备参考。

www.TopSage.com

表 12-2 文 件 模 式

模　式	说　明
"w"	打开或创建一个文本文件，以执行写入操作
"a"	打开一个文本文件，以执行追加操作，将数据添加到文件末尾
"r"	打开一个文本文件，以执行读取操作
"wb"	打开或创建一个二进制文件，以执行写入操作
"ab"	打开一个二进制文件，以执行追加操作
"rb"	打开一个二进制文件，以执行读取操作
"w+"	打开或创建一个文本文件，以执行更新操作。已有的文件内容会被删除
"a+"	打开或创建一个文本文件，以执行更新操作。将数据添加到文件末尾
"r+"	打开一个文本文件，以执行更新操作。可以在任意位置执行读写操作
"w+b" 或 "wb+"	打开或创建一个二进制文件，以执行更新操作。已有的文件内容会被删除
"a+b"或 "ab+"	打开一个二进制文件，以执行更新操作。将数据添加到文件末尾
"r+b" 或 "rb+"	打开一个二进制文件，以执行更新操作。可以在任意位置执行读写操作

12.15 设计程序

本章的最后将前面所学的内容应用于最后一个程序。这个程序比前面的例子短，但很有趣。

12.15.1 问题

需要解决的问题是编写一个文件查看器程序，它可以将文件显示为十六进制和字符方式。

12.15.2 分析

这个程序以二进制只读模式打开文件，将信息显示在两列中，第一列是文件中表示为十六进制的字节，第二列是显示为字符的字节。文件名作为一个命令行参数提供，如果没有提供文件名，程序就要求输入文件名。

步骤如下：

(1) 如果没有提供文件名，就要求用户输入。

(2) 打开文件。

(3) 读取并显示文件的内容。

12.15.3 解决方案

本节列出解决问题的步骤。

www.TopSage.com

1. 步骤 1

检查函数 main()的参数，就可以确定文件名是否出现在命令行中。前面都忽略了给 main()函数传送参数，但是这里可以使用它作为识别文件的方法。调用 main()函数时，会给它传入两个参数。第一个参数是一个整数，指出命令行上单词的数目，第二个参数是一个字符串指针数组。第一个字符串是在命令行中用于启动程序的名称，其余字符串代表参数。当然，可以在命令行中输入任意个值，并传给 main()函数。

如果 main()函数的第一个变元是 1，则命令行上只包含程序名，所以必须提示输入文件名：

```
/* Program 12.8 Viewing the contents of a file */
#include <stdio.h>

const int MAXLEN = 256;   /* Maximum file path length */

int main(int argc, char *argv[])
{
  char filename[MAXLEN]; /* Stores the file path */

  if(argc == 1)           /* No file name on command line? */
  {
    printf("Please enter a filename: "); /* Prompt for input */
    fgets(filename, MAXLEN, stdin); /* Get the file name entered */

    /* Remove the newline if it's there */
    int len = strlen(filename);
    if(filename[len-1] == '\n')
      filename[len-1] = '\0';
  }
  return 0;
}
```

文件路径的最大长度为 256 个字符。

2. 步骤 2

如果 main()函数的第一个变元不是 1，就至少有一个变元，假设它是文件名。因此，将 argv[1]所指的字符串复制到变量 openfile 中。假设这是个有效的文件名，可以打开它，开始读取：

```
/* Program 12.8 Viewing the contents of a file */
#include <stdio.h>

const int MAXLEN = 256; /* Maximum file path length */

int main(int argc, char *argv[])
{
  char filename[MAXLEN]; /* Stores the file path */
```

www.TopSage.com

```
  FILE *pfile;              /* File pointer */

  if(argc == 1)             /* No file name on command line? */
  {
    printf("Please enter a filename: "); /* Prompt for input */
    fgets(filename, MAXLEN, stdin); /* Get the file name entered */

    /* Remove the newline if it's there */
    int len = strlen(filename);
    if(filename[len-1] == '\n')
      filename[len-1] = '\0';
  }
  else
    strcpy(filename, argv[1]); /* Get 2nd command line string */

  /* File can be opened OK? */
  if(!(pfile = fopen(filename, "rb")))
  {
    printf("Sorry, can't open %s", filename);
    return -1;
  }
  fclose(pfile);            /* Close the file */
  return 0;
}
```

在程序的结尾调用 fclose()函数，以关闭文件，以免后面忘记关闭文件。另外，程序发生错误时，就返回 - 1。

3. 步骤 3

现在可以输出文件的内容了。一次读取一个字节，将它存储到缓冲区中。当缓冲区满了或到达文件的末尾时，就以指定的格式输出这个缓冲区。数据输出为字符时，首先必须检查字符是否可以打印，否则屏幕会显示乱码。为此使用在<ctype.h>中声明的 isprint()函数。如果这是不可打印的字符，就输出一个句点。以下是完整的代码：

```
/* Program 12.8 Viewing the contents of a file */
#include <stdio.h>
#include <ctype.h>
#include <string.h>

const int MAXLEN = 256;       /* Maximum file path length */
const int DISPLAY = 80;       /* Length of display line */
const int PAGE_LENGTH = 20;   /* Lines per page */

int main(int argc, char *argv[])
{
  char filename[MAXLEN];                  /* Stores the file path */
  FILE *pfile;                            /* File pointer */
  unsigned char buffer[DISPLAY/4 - 1];    /* File input buffer */
```

www.TopSage.com

```
int count = 0;                          /* Count of characters in buffer */
int lines = 0;                          /* Number of lines displayed */

if(argc == 1)                          /* No file name on command line? */
{
  printf("Please enter a filename: "); /* Prompt for input */
  fgets(filename, MAXLEN, stdin); /* Get the file name entered */

  /* Remove the newline if it's there */
  int len = strlen(filename);
  if(filename[len-1] == '\n')
    filename[len-1] = '\0';
}
else
  strcpy(filename, argv[1]); /* Get 2nd command line string */

/* File can be opened OK? */
if(!(pfile = fopen(filename, "rb")))
{
  printf("Sorry, can't open %s", filename);
  return -1;
}
while(!feof(pfile)) /* Continue until end of file */
{
  if(count < sizeof buffer)              /* If the buffer is not full */
    buffer[count++] = (unsigned char)fgetc(pfile); /* Read a character */
  else
  { /* Output the buffer contents, first as hexadecimal */
    for(count = 0; count < sizeof buffer; count++)
      printf("%02X ", buffer[count]);
    printf("| ");                       /* Output separator */

    /* Now display buffer contents as characters */
    for(count = 0; count < sizeof buffer; count++)
      printf("%c", isprint(buffer[count]) ? buffer[count]:'.');
    printf("\n");                       /* End the line */
    count = 0;                          /* Reset count */

    if(!(++lines%PAGE_LENGTH)) /* End of page? */
      if(getchar()=='E')               /* Wait for Enter */
        return 0;                       /* E pressed */
    }
  }

/* Display the last line, first as hexadecimal */
for(int i = 0; i < sizeof buffer; i++)
  if(i < count)
    printf("%02X ", buffer[i]); /* Output hexadecimal */
  else
```

www.TopSage.com

```
      printf(" ");                    /* Output spaces */
    printf("| ");                     /* Output separator */

    /* Display last line as characters */
    for(int i = 0; i < count; i++)
      /* Output character */
      printf("%c",isprint(buffer[i]) ? buffer[i]:'.');

    /* End the line */
    printf("\n");
    fclose(pfile);                    /* Close the file */
    return 0;
}
```

符号 DISPLAY 指定屏幕上一行输出的宽度，符号 PAGE_LENGTH 指定一页的行数。显示一页后，要等待用户按下回车键，再显示下一页，这样可以避免显示太快而无法阅读。

文件输入缓冲区的声明如下：

```
unsigned char buffer[DISPLAY/4 - 1]; /* File input buffer */
```

计算数组大小的表达式来源于在屏幕上需要 4 个字符来显示文件中的每个字符，再加上一个分隔符。每个字符都显示为两个十六进制数和一个空格，因此一个字符需要用 4 个字符来显示。

只要 while 循环的条件为 true，就继续读取：

```
while(!feof(pfile)) /* Continue until end of file */
```

如果读到文件的结尾，即 EOF，库函数 feof()就返回 true，否则返回 false。在 if 语句中，用文件中的字符填充 buffer 数组：

```
if(count < sizeof buffer)                  /* If the buffer is not full */
  buffer[count++] = (unsigned char)fgetc(pfile); /* Read a character */
```

当 count 超过 buffer 的容量时，就执行 else 子句，输出缓冲区的内容：

```
else
{ /* Output the buffer contents, first as hexadecimal */
  for(count = 0; count < sizeof buffer; count++)
    printf("%02X ", buffer[count]);
  printf("| "); /* Output separator */

  /* Now display buffer contents as characters */
  for(count = 0; count < sizeof buffer; count++)
    printf("%c", isprint(buffer[count]) ? buffer[count]:'.');
  printf("\n");          /* End the line */
  count = 0;             /* Reset count */

  if(!(++lines%PAGE_LENGTH)) /* End of page? */
    if(getchar()=='E') /* Wait for Enter */
```

www.TopSage.com

```
        return 0; /* E pressed */
}
```

第一个 for 循环将缓冲区的内容输出为十六进制数，然后输出分隔符，执行下一个 for 循环，将相同的数据输出为字符。printf()第二个变元中的条件运算符确保将不可打印的字符输出为句点。

if 语句递增行的计数器 lines，每输出 PAGE_LENGTH 行，就等待输入。如果按下回车键，就显示下一页，如果键入 E 并按下回车键，程序就结束。这样，当文件过大时，可以退出输出操作。

最后两个 for 循环和前面的类似。唯一不同的是为不含有文件字符的数组元素输出空格。输出的例子如下，它显示了程序 12.8 的源文件，从输出中可以推断出，文件路径输入为一个命令行参数。

```
2F 2A 20 50 72 6F 67 72 61 6D 20 31 32 2E 38 20 56 69 65 | /* Program 12.8 Vie
77 69 6E 67 20 74 68 65 20 63 6F 6E 74 65 6E 74 73 20 6F | wing the contents o
66 20 61 20 66 69 6C 65 20 2A 2F 0D 0A 23 69 6E 63 6C 75 | f a file */..#inclu
64 65 20 3C 73 74 64 69 6F 2E 68 3E 0D 0A 23 69 6E 63 6C | de <stdio.h>..#incl
75 64 65 20 3C 63 74 79 70 65 2E 68 3E 0D 0A 23 69 6E 63 | ude <ctype.h>..#inc
6C 75 64 65 20 3C 73 74 72 69 6E 67 2E 68 3E 0D 0A 0D 0A | lude <string.h>....
63 6F 6E 73 74 20 69 6E 74 20 4D 41 58 4C 45 4E 20 3D 20 | const int MAXLEN =
32 35 36 3B 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 | 256;
20 20 20 20 20 2F 2A 20 4D 61 78 69 6D 75 6D 20 66 69 6C |      /* Maximum fil
65 20 70 61 74 68 20 6C 65 6E 67 74 68 20 20 20 20 20 20 | e path length
2A 2F 0D 0A 63 6F 6E 73 74 20 69 6E 74 20 44 49 53 50 4C | */..const int DISPL
41 59 20 3D 20 38 30 3B 20 20 20 20 20 20 20 20 20 20 20 | AY = 80;
20 20 20 20 20 20 20 20 20 2F 2A 20 4C 65 6E 67 74 68 20 |          /* Length
20 20 20 20 20 20 20 20 20 2F 2A 20 4C 65 6E 67 74 68 20 |
6F 66 20 64 69 73 70 6C 61 79 20 6C 69 6E 65 20 20 20 20 | of display line
20 20 20 20 2A 2F 0D 0A 63 6F 6E 73 74 20 69 6E 74 20 50 |     */..const int P
41 47 45 5F 4C 45 4E 47 54 48 20 3D 20 32 30 3B 20 20 20 | AGE_LENGTH = 20;
20 20 20 20 20 20 20 20 20 20 20 20 20 20 2F 2A 20 4C 69 6E |               /* Lin
65 73 20 70 65 72 20 70 61 67 65 20 20 20 20 20 20 20 20 | es per page
20 20 20 20 20 20 20 20 2A 2F 0D 0A 0D 0A 69 6E 74 20 6D |         */....int m
61 69 6E 28 69 6E 74 20 61 72 67 63 2C 20 63 68 61 72 20 | ain(int argc, char
```

这个输出的最后部分是：

```
66 65 72 5B 69 5D 3A 27 2E 27 29 3B 0D 0A 20 20 2F 2A 20 | fer[i]:'.');..  /*
45 6E 64 20 74 68 65 20 6C 69 6E 65 20 20 20 20 20 20 20 | End the line
20 20 20 2A 2F 0D 0A 20 20 70 72 69 6E 74 66 28 22 5C 6E |    */..  printf("\n
22 29 3B 0D 0A 20 20 66 63 6C 6F 73 65 28 70 66 69 6C 65 | ");..  fclose(pfile
29 3B 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 | );
20 20 20 20 20 20 20 20 20 20 2F 2A 20 43 6C 6F 73 65 20 |          /* Close
74 68 65 20 66 69 6C 65 20 20 20 20 20 20 20 20 20 20 20 | the file
20 20 20 20 20 2A 2F 0D 0A 20 20 72 65 74 75 72 6E 20 30 |      */..  return 0
3B 0D 0A 7D 0D 0A 0D 0A FF                               | ;..}.....
```

www.TopSage.com

12.16 小结

本章介绍了编写各种文件函数所需的所有基本工具。例子示范的函数相当有限，还有很多应用这些工具的方式，提供了管理及提取文件信息的更复杂的方法。例如，可将索引信息写入文件，该索引可以放在文件中已知的地方，通常是文件的开头，也可以是数据块中的位置指针，例如链表中的指针。读者应练习文件操作，以理解其机制。

本章讨论的函数包含了大部分技巧，另外编译器提供的输入输出库还有许多函数，为文件处理提供了更多的选择。

12.17 习题

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容。还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 12.1 编写一个程序，将任意数目的字符串写入文件。字符串由键盘输入，程序不能删除这个文件，因为下一题还要使用这个文件。

习题 12.2 编写一个程序，读取上一题创建的文件。每次都以反向的顺序读取一个字符串，然后按照读取顺序将它们写入一个新文件。例如，程序读取最后一个字符串，将它写入新文件，再读取倒数第二个字符串，将它写入新文件，依此类推。

习题 12.3 编写一个程序，从键盘读入姓名和电话号码，将它们写入一个文件。如果这个文件不存在，就写入一个新文件。如果文件已存在，就将它们写入该文件。这个程序需提供列出所有数据的选项。

习题 12.4 扩展上一题的程序，提取对应指定的姓的所有电话号码。这个程序允许进一步查询，添加新的姓名和电话号码，删除已有的项。

www.TopSage.com

第 13 章

支 持 功 能

前面已完整介绍了C语言以及重要的库函数。读者应能使用C语言编写各种程序了。如果不行，就需要多加练习。学习了一门语言的基本要素之后，所需要做的就是实践，实践，再实践。

最后一章将探讨一些尚未解决的问题，深入讨论预处理器的功能和一些很有用的库函数。

本章的主要内容：

- 预处理器及其操作
- 如何编写预处理器宏
- 逻辑预处理器指令的概念及用法
- 条件编译的概念及用法
- 调试方法
- 使用一些有用的库函数

13.1 预处理

源代码在编译成机器指令之前，要进行预处理。预处理阶段可以根据预处理指令(它的第一个字符是#)执行一系列服务。预处理阶段可以在编译之前处理及修改 C 源代码。完成预处理阶段，并分析及执行了所有预处理指令后，这些指令就不再出现在源代码中。编译器开始编译阶段，生成与程序对应的机器码。

所有的例子都使用了预处理指令，包括#include 和#define 指令。还有许多其他指令，为指定程序的方式提供了相当灵活的方法。注意，这些预处理操作发生在编译程序之前。它们会修改程序的语句，但不会干涉程序的执行。

13.1.1 在程序中包含头文件

头文件是外部的文件，使用#include 预处理指令可以将它的内容包含在程序中。下面的语句：

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

这会将支持输入输出操作的标准库头文件放在程序中。将标准库包含到程序中的一

www.TopSage.com

般语句如下：

```
#include <standard_library_file_name>
```

尖括号中可以包含任何库头文件名称。如果包含了未使用的头文件，除了使读程序的人感到困惑外，唯一的作用是增加编译的时间。

也可以将自己的源文件用略微不同的#include 语句包含到程序中。下面是一个典型的例子：

```
#include "myfile.h"
```

这行语句会将双引号内的文件内容引入程序，替代#include 指令。任何文件的内容都可以通过这个方法包含到程序中，只要在引号中指定文件名即可。

也可以使用操作系统允许的任何文件名。理论上，不一定要使用扩展名.h，但它是大多数 C 程序员惯用的扩展名，所以最好使用它。使用这种形式和尖括号的区别是指定的源文件不同。具体的操作与编译器相关，在编译器的文档说明中有详细解释，但通常第一种形式是在默认的头文件目录中搜索需要的文件，而第二种形式是先在当前源文件的目录下搜索，再搜索默认的头文件目录。

头文件不能包含实现代码，即可执行代码。头文件可以包含声明，但不能包含函数定义或初始化的全局数据。函数定义和初始化的全局数据应放在扩展名为.c 的源文件中。可以使用头文件机制将程序的源代码放在几个文件中，当然，要管理自己编写的每个库函数的声明。一个常用的技巧是创建一个头文件，它含有程序中所有函数的原型以及类型声明。然后，将这些作为一个独立的单元来管理，并放在程序源文件的开头。如果源文件包含多个头文件，必须避免信息的重复。重复的代码会造成编译错误。本章稍后的“条件编译”一节将介绍如果不小心将任意给定的代码块包含了多次，如何使它们只在程序中出现一次。

注意：

#include 指令引入程序的文件，可能也含有其他#include 指令。预处理器处理第二个#include 的方式和第一个完全相同，也是用对应文件的内容取代该指令，直到程序中没有#include 指令为止。

13.1.2 外部变量及函数

一个由几个源文件组成的程序，常需要使用在其他文件内定义的全局变量。为此，可以使用关键字 extern 将它们声明为外部变量。例如，使用如下语句在其他文件内定义了一个全局变量(是在任何函数之外)：

```
int number = 0;
double in_to_mm = 2.54;
```

然后，要在一个函数中访问它们，可以使用以下语句指定这些变量名是外部的：

www.TopSage.com

```
extern  int  number;
extern  double in_to_mm;
```

这些语句不会创建这些变量，只是告诉编译器，这些名称在文件外定义，但可以应用于源文件的其他地方。指定为 extern 的变量在程序的外部声明和定义，通常是在另一个源文件中。如果要让当前文件中的所有函数都可访问这些外部变量，必须在文件的开头，在任何函数的定义之前将它们声明为外部变量。程序是由几个文件组成的，可以把所有已初始化的全局变量放在一个文件的开头，将所有的 extern 语句放在另一个头文件中。使用 include 语句包含该头文件，所有的 extern 语句就合并到需要访问这些变量的程序文件中。

注意：

每个全局变量在一个文件中只允许声明一次。当然，全局变量可以在许多文件中声明为外部变量。

13.1.3 替换程序源代码

程序在编译之前，预处理器指令会替换源代码中的符号。最简单的符号替换前面已介绍过。例如，使用预处理器指令将字符串 PI 替换为特定值：

```
#define  PI  3.14159265;
```

标识符 PI 看起来像是变量，但它不是变量，与变量一点关系也没有。PI 是一个标志，有点像凭证，在预处理阶段用来替换在#define 指令中指定的一串数字。当预处理完成后，准备编译程序时，PI 字符串已经被它的定义取代，不再出现了。这类预处理器指令的一般形式如下：

```
#define  identifier  sequence_of_characters
```

这里的 identifier 符合 C 语言中标识符的一般定义；一串字母和数字，但第一个字符必须是字母或下划线。注意，取代 identifier 的是 sequence_of_characters，即一串字符，不一定是数字。

#define 指令的一个常见用法是定义数组的大小，允许用一个标志确定多个数组的大小。只要修改程序中的一个指令，就可以改变多个数组的大小。这有助于在需要进行这类修改时减少错误，如下面的例子所示：

```
#define MAXLEN 256
char *buffer[MAXLEN];
char *str[MAXLEN];
```

这两个数组的大小可以通过修改#define 指令而改变，当然受影响的数组声明可以放在程序文件的任何地方。当程序涉及几十个甚至上百个函数时，这个方法的优点非常明显。它不仅便于修改，还可以确保在程序中只使用同一个值。当几个程序员合作开发大项目时，这一点特别重要。

www.TopSage.com

当然，也可以将 MAXLEN 定义为 const 常量：

```
const size_t MAXLEN = 256;
```

这种方法与使用#define 指令的区别是，MAXLEN 不再是一个标志，而是一个指定类型的变量，其名称是 MAXLEN。源文件完成了预处理后，#define 指令中的 MAXLEN 就不再存在，因为代码中的 MAXLEN 都替换为 256。

最后两个例子使用了数值替换，但这个用法是没有限制的。例如可以编写：

```
#define  Black  White
```

这行语句使程序中的所有 Black 被 White 取代。用来取代标志的字符串可以是任何东西。

13.1.4 宏替换

宏基于前面的#define 指令，但它的适用范围比较大，允许进行多个参数化替换。不仅可以用固定的字符串替换标志符，还允许指定一些参数，而这些参数可以被变元的值取代。

下面是一个例子：

```
#define  Print(My_var)  printf(" %d",  My_var)
```

这个指令提供了两层替换。一个是用其后的字符串替换 Print(My_var)，另一个是对 My_var 的替换。例如下面的语句：

```
Print(ival);
```

这行语句在预处理时，会转换成：

```
printf("%d", ival);
```

可以使用这个指令在程序的不同地方给 printf()语句指定不同的整数变量。这种宏的一般作用是用比较简单的方式表示复杂的函数调用，以提高程序的可读性。

13.1.5 看起来像函数的宏

这类替换语句的一般形式是：

```
#define macro_name( list_of_identifiers ) substitution_string
```

在一般情况下，可以有许多参数，因此可定义比较复杂的替换。为了说明如何使用，下面定义一个宏，使用下面的语句找出两个数中的较大值：

```
#define  max(x,  y)  x> y  ?  x  :  y
```

然后，将下面的语句放在程序中：

www.TopSage.com

```
result = max(myval, 99);
```

在预处理期间，该语句会展开成：

```
result = myval>99 ? myval : 99;
```

注意，这里执行的是替换，不要把它看做函数。否则会得到奇怪的结果，特别是替换标志符中使用显式或隐式的赋值时。例如，将前一个例子稍稍修改一下，就会得到错误的结果：

```
result = max(myval++, 99);
```

替换过程会产生以下语句：

```
result = myval++>99 ? myval++ : 99;
```

这个结果是，如果 myval 的值大于 99，myval 会递增两次。注意，在此情况下使用括号也是没用的。如果改写成：

```
result = max((myval++), 99);
```

预处理的结果是：

```
result = (myval++>99) ? (myval++) : 99;
```

在编写生成表达式的宏时，要特别小心。除了刚才看到的多重替换的陷阱外，优先级规则也会使代码出错。例如，编写一个宏，获得两个参数的积：

```
#define product(m, n) m*n
```

然后，在下面的语句使用这个宏：

```
result = product(X, y + 1);
```

当然，宏切换过程是正常的，只是得不到所要的结果，因为这个宏展开成：

```
result = x*y + 1;
```

这可能要花很长时间才发现，没有得到这两个参数的积，因为光从外观根本看不出发生了什么事，只知道程序就是有点小错误。解决方案非常简单。如果使用宏生成表达式，则将所有的东西加上括号。因此，可以将这个例子重写为：

```
#define product(m, n)((m)*(n))
```

现在一切都正常了。外层的括号看起来似乎是多余的，但由于不知道宏会展开成什么样子，所以最好加上括号。如果编写一个宏来计算其参数的总和，就会很容易看出，没有外面的括号，在许多情况下得不到预期的结果。甚至有了括号，在展开的表达式中重复使用一个参数时，例如前面使用条件运算符的表达式，如果这个参数使用了递增或递减运算符，也不能正常操作。

www.TopSage.com

13.1.6 多行上的预处理指令

预处理指令必须在一个逻辑行中，但可以使用续行符“\”将它分成许多行。可以编写如下语句：

```
#define  min(x,   y)  \
         ((x)<(y)  ?  (x)  :   (y))
```

这里，语句定义在第二行的第一个非空白字符处继续，因此只要觉得这样的安排比较好，可以将文本放在第二行。注意，\必须是这行的最后一个字符，其后是回车符。

13.1.7 字符串作为宏参数

使用宏时，字符串常量是一个潜在的混乱根源。最简单的字符串替换是单层的定义，例如：

```
#define  MYSTR  "This  string"
```

假设编写了下面的语句：

```
printf("%s",   MYSTR);
```

在预处理期间，这行语句会转换成：

```
printf("%s",   "This  string" );
```

结果正确无误。但在#define 指令中定义替换字符串时没有加上双引号，它就不会加上双引号。例如，假设编写：

```
#define  MYSTR  This  string
  ...
printf("%s",   "MYSTR"  );
```

本例中，printf()函数内的 MYSTR 不会被替换。程序中双引号里的内容都被假设成文本字符串，所以在预处理期间不会分析它。

有一个特殊的方法可以指定替换宏的字符串参数。例如，可以指定一个宏，使用 printf()函数显示字符串，如下：

```
#define  PrintStr(arg)  printf("%s",   #arg)
```

出现在宏中的 arg 参数前的#字符指出，这个变元放在双引号中，就会被替换。因此，如果在程序中编写了如下语句：

```
PrintStr(Output);
```

在预处理期间，它会转换成：

```
printf("%s", "Output");
```

www.TopSage.com

为什么将这个看起来很复杂的机制引入预处理阶段？其实，没有这个功能，就不能在宏定义中使用可变的字符串。如果希望将宏的参数放在双引号中，它就不会被解读为变量；而只是一个带双引号的字符串。换句话说，如果宏展开式放在双引号中，双引号中的字符串就不解读为参数的标识符，而只是一个字符串常量。因此，乍看之下不必要的复杂机制其实是允许在宏中创建带双引号的字符串的重要工具。

这个机制的常见用法是将变量名称转换成字符串，如下：

```
#define  show(var)  printf("\n%s  =  %d",  #var,   var);
```

如果这么编写：

```
show(number);
```

就会产生如下语句：

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

也可以显示带双引号的字符串。假设定义了之前显示的宏 PrintStr，然后编写如下语句：

```
PrintStr("Output");
```

它会预处理成：

```
printf("%s",   "\"Output\"");
```

这是可能的，因为预处理阶段知道需要在两端放置\"，以正确显示双引号。

13.1.8 结合两个宏展开式的结果

有时希望宏生成两个结果，并将它们结合在一起，其中没有空格。假设定义如下的宏：

```
#define  join(a,   b)  ab
```

这不能正常工作。这个展开式的定义会解读成 ab，而不是参数 a 后跟参数 b。如果用空格将它们分开，结果也会有一个空格，这不是希望的结果。预处理阶段提供了一个运算符来解决这个问题。指定宏的方法如下：

```
#define  join(a,   b)  a##b
```

这个运算符由两个字符##组成，用来分隔参数，表示两个替换的结果。例如，下面的语句：

```
strlen(join(var, 123));
```

会替换成如下语句：

```
strlen(var123);
```

www.TopSage.com

这可以用于合成变量名称，或是从两个或多个宏参数中生成一个格式控制字符串。

13.2 预处理器逻辑指令

上一个例子看起来好像相当有限，实在很难想象在什么情形下需要把 var 和 123 连接起来。毕竟，总是可以使用一个参数，将变元编写成 varl23。但是预处理的一个作用是，允许前一个例子进行多个宏的替换，即一个宏中的变元派生于另一个宏中定义的替换。在上一个例子中，join()宏的两个变元都可以由其他#define 替换或宏生成。预处理也支持提供逻辑 if 功能的指令，它极大地扩展了预处理阶段所能处理的范围。

13.2.1 条件编译

第一个要讨论的逻辑指令测试前一个#define 指令创建的标识符是否存在。它的形式如下：

```
#if  defined  identifier
```

如果定义了指定的 identifier，则#if 和 #endif 之间的语句就包含到程序代码中。如果没有定义该标识符，就跳过#if 和#endif 之间的语句。这和 C 编程中使用的逻辑过程相同，只是这里将程序语句包含或不包含在源文件中。

也可以测试标识符是否不存在。事实上，这个机制的使用率比上一个高得多。它的一般形式如下：

```
#if  !defined  identifier
```

如果没有定义 identifier，#if 和#endif 间的所有语句就包含到程序中。这可以避免在包含多个文件的程序中重复定义函数、其他代码块和指令，或当程序处理了#include 语句后，确保不重复可能在不同库中重复出现的代码。避免重复代码块的机制如下：

```
#if !defined block1
  #define block1
  /* Block of code you do not */
  /* want to be repeated.     */
#endif
```

如果没有定义标识符 block1，就包含并处理#/if 和#endif 间的代码块，定义 block1。#endif 后面的代码块也会包含在程序中。以后出现的相同语句块不会包含，因为 block1 已经存在。

这里的#define 指令不需要指定替换值。要执行条件编译，将 block1 放在#define 指令中就足够了。现在可以将这块代码包含到需要它们的任何地方去，并保证在程序中绝不会有重复。预处理指令确保不会有重复。

www.TopSage.com

注意：

最好总是以这种方式保护自己的库中的代码。将自己的函数放在几个库中后，很容易出现重复的代码块。

使用#if 预处理器指令不仅能测试一个值，还可测试是否定义了多个标识符。例如，下面的语句：

```
#if  defined  block1  &&  defined  block2
```

如果之前定义了 block1 和 block2，该表达式就等于 true，因此这个指令后的代码不会包含进来。

条件预处理器指令的另一个应用是取消前面定义的标识符，这需要使用#undef block1 指令。

如果已经定义了 block1，在这个指令后它就不再被定义。这两个指令组合起来使用可以取得很好的效果。

这些指令有另外一个比较简洁的写法。选用哪种形式取决于个人的喜好。指令#ifdef block 和#if defined block1 等效，指令#ifndef block 和#if !defined block1 等效。

13.2.2 测试指定值的指令

也可以使用#if 指令的一种形式测试常量表达式的值。如果常量表达式的结果不是 0，这条语句和下一个#endif 之间的所有语句就都包含到程序代码中。如果常量表达式的结果是 0，就跳过这条语句和下一个#endif 之间的所有语句。#if 指令的一般形式如下：

```
#if  constant_expression
```

它经常用于测试前面的预处理器语句赋予标识符的指定值。例如，下面的语句：

```
#if CPU == Pentium4
  printf("\nPerformance should be good." );
#endif
```

如果标识符 CPU 在之前的#define 指令中定义为 Pentium4，printf()语句就包含到程序中。

13.2.3 多项选择

为了补足#if 指令，可以使用#else 指令。它的作用和 else 语句完全相同：当#if 条件失败时，就执行一组指令或包含一些语句。例如：

```
#if  CPU ==  Pentium4
 printf("\nPerformance  should  be  good."  );
#else
 printf("\nPerformance  may  not  be  so  good."  );
#endif
```

www.TopSage.com

在这个例子中，将哪一个 printf()语句包含到程序中，取决于 CPU 是否定义成 Pentium4。

预处理阶段也为多项选择提供了一个特殊的#if 形式，只将几个语句中的一个包含到程序中。这个指令是#elif。它的一般形式是：

```
#elif  constant_expression
```

使用这个指令的例子如下：

```
#define US 0
#define UK 1
#define Australia 2
#define Country US
#if Country == US
  #define Greeting "Howdy, stranger."
#elif Country == UK
  #define Greeting "Wotcher, mate."
#elif Country == Australia
  #define Greeting "G'day, sport."
#endif
printf("\n%s", Greeting );
```

在这串指令中，printf()语句的输出取决于赋予标识符 Country 的值，在这个例子中它是 US。

13.2.4 标准预处理宏

在编译器的文档说明中，通常定义了大量的标准预处理宏。这里只介绍其中两个比较常用的宏。

宏__DATE__提供日期的字符串表示法，在程序中调用它时，它的格式是 Mmm dd yyyy。其中 Mmm 是月份，如 Jan、Feb 等，dd 是日期，即 1~31 的数字，如果是一个数字，就在该数字前面加上空白。yyyy 是 4 位数字的年份，例如 2006。

宏__TIME__提供了包含时间值的字符串，在程序中调用它时，它的格式是 hh:mm:ss，代表时、分、秒。每个都有两位数字，用冒号隔开。注意这是编译器执行的时间，不是程序执行的时间。

可以使用这个宏记录程序最后一次的编译时间，如下：

```
printf("\nProgram last compiled at %s on %s", __TIME__, __DATE__ );
```

注意，__DATE__和__TIME__的下划线在开头和末尾。编译含有这行语句的程序时，printf()输出的值会固定不变，直到下次编译程序为止。以后程序执行时，会输出当前的时间和日期。不要把这两个宏和本章后面的“日期和时间函数库”一节讨论的时间函数搞混。

www.TopSage.com

13.3 调试方法

第一次编写完程序时，程序大都有一些错误。从程序中删除这些错误大致和编写程序所花的时间成正比。程序越大、越复杂，包含的错误就越多，使程序正常运行所需的时间也就越多。一些非常大的程序，如操作系统，或复杂的应用程序，如字处理系统，甚至 C 程序开发系统，都因为过于复杂，不可能将错误完全消除。读者也许对此有一些经验。通常这类残余的错误相当轻微，能与系统一起运行。

编写程序的方法对将来测试程序的难度有显著的影响。结构优秀的程序由简洁的函数组成，每个函数的目的都定义得很明确，所以比没有这些特性的函数更容易测试。给程序的操作和函数的作用加上详细的注释，使用恰当的变量及函数名称，也使错误的查找变得比较容易。使用缩排语句的布局也对测试及错误查找有帮助。

调试虽不在本书所讨论的范围，但是本节将介绍必须注意的基本观念。

13.3.1 集成的调试器

许多编译器都在程序开发环境中嵌入了大量的调试工具。这些工具的功能很强大，可以显著减少使程序正常运行所需的时间。一般它们提供了各种帮助测试程序的工具，具体如下：

- 追踪程序流：这个功能可以一次执行一行语句。每执行完一行语句，就暂停，用户按下特定的键后，就继续执行下一行语句。调试环境中的其他工具通常能显示信息，暂时中止执行，以便了解程序中的数据发生了什么事。
- 设定断点：在执行很大、很复杂的程序时，一次执行一行是很痛苦的。有时这是不可能的，例如一个循环需要执行 10 000 次。然而，断点提供了绝佳的替代方案。使用断点，可以在程序中选择一些特定的语句，程序执行到该处时就会暂停，以便检查当时的状况。按下指定的键后，继续执行到下一个断点。
- 设定观看窗口：这个功能可以在执行过程中追踪变量的值。所选变量的值在程序的每个暂停处显示出来。如果一步步地执行程序，就可以看到变量值变化的情形，或者没有像期望的那样变化。
- 检查程序元素：有时也要检查许多程序成员的情况。例如，在断点处显示函数的细节，如返回类型以及它的变元。还可以根据指针的地址查看指针的细节，它含有的地址以及指针地址所存储的数据。也可以查看表达式的值，修改变量。修改变量可以绕过问题区域，让其他的区域使用正确的数据执行，即使之前的程序部分不能正常工作。

13.3.2 调试阶段的预处理器

使用条件预处理器指令，可以将代码块包含到程序中，以帮助测试。许多 C 语言开发系统的调试功能非常强大，但添加自己的追踪代码仍旧有用。可以控制所显示的数据

www.TopSage.com

的格式，甚至根据程序中的条件或关系，输出各种不同的数据以用于调试。

试试看：使用预处理器指令

为了说明如何使用预处理器指令控制程序的执行，打开或关闭调试功能，下面编写一个程序，通过函数指针数组随机调用函数：

```
/* Program 13.1 Debugging using preprocessing directives */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* Macro to generate pseudo-random number from 0 to NumValues */
#define random(NumValues)
     ((int)(((double)(rand())*(NumValues))/(RAND_MAX+1.0)))

#define iterations 6
#define test        /* Select testing output */
#define testf       /* Select function call trace */
#define repeatable  /* Select repeatable execution */

/* Function prototypes */
int sum(int, int);
int product(int, int);
int difference(int, int);

int main(void)
{
  int funsel = 0;     /* Index for function selection */
  int a = 10, b = 5;  /* Starting values */
  int result = 0;     /* Storage for results */

  /* Function pointer array declaration */
  int (*pfun[])(int, int) = {sum, product, difference};

  /* Conditional code for repeatable execution */
  #ifdef repeatable
  srand(1);
  #else
  srand((unsigned int)time(NULL)); /* Seed random number generation */
  #endif

  /* Execute random function selections */
  int element_count = sizeof(pfun)/sizeof(pfun[0]);
  for(int i = 0 ; i < iterations ; i++)
  {
    /* Generate random index to pfun array */
    funsel = random(element_count);
    if( funsel>element_count-1 )
    {
```

www.TopSage.com

```
      printf("\nInvalid array index = %d", funsel);
      exit(1);
    }

    #ifdef test
      printf("\nRandom index = %d", funsel);
    #endif

    result = pfun[funsel](a , b);     /* Call random function */
    printf("\nresult = %d", result );
  }
  return 0;
}

/* Definition of the function sum */
int sum(int x, int y)
{
  #ifdef testf
    printf("\nFunction sum called args %d and %d.", x, y);
  #endif

return x + y;
}

/* Definition of the function product */
int product( int x, int y )
{
  #ifdef testf
    printf("\nFunction product called args %d and %d.", x, y);
  #endif

  return x * y;
}

/* Definition of the function difference */
int difference(int x, int y)
{
  #ifdef testf
    printf("\nFunction difference called args %d and %d.", x, y);
  #endif

  return x - y;
}
```

代码的说明

程序一开始定义了一个宏:

```
#define random(NumValues)
    ((int)(((double)(rand())*(NumValues))/(RAND_MAX+1.0)))
```

www.TopSage.com

上述语句定义了宏 random()，其中的函数 rand()在<stdlib.h>中声明，它会生成 0~RAND_MAX 之间的随机数。RAND_MAX 是一个在<stdlib.h>中定义的常量。这个宏会映射这个范围的值，产生 0~NumValues－1 之间的值。将 rand()函数产生的值转换成 double，确保使用 double 类型进行计算，最后因为程序的需要，再将计算的结果转换成 int 类型。<stdlib.h>版本里已经有完成该操作的宏。编译器不允许宏有两种不同的定义，因此程序编译时会产生错误。此时可以将该定义从程序中删除。

将 random()定义为宏，是为了声明如何定义它，但最好把它定义为一个函数，因为这可以避免宏的变元值带来的问题。

接着，定义符号的 4 个指令:

```
#define iterations 6
#define test        /* Select testing output */
#define testf       /* Select function call trace */
#define repeatable  /* Select repeatable execution */
```

第一个指令定义的符号指定了循环的迭代次数，该循环会随机执行 3 个函数中的一个。后 3 个符号都用于控制包含在程序中的代码。定义 test 符号，所包含的代码会输出用于选择函数的索引值。定义 testf 会把跟踪函数调用的代码包含在程序中。定义 repeatable 时，用一个固定的种子值调用 srand()函数，所以 rand()函数总是生成相同的伪随机序列，在运行程序时会得到相同的结果。在程序的测试过程中有可重复的结果，会使测试过程更容易。如果删除了定义 repeatable 符号的代码，就用当前的时间值作为变元来调用 srand()函数，因此每次执行程序使种子值都是不同的，得到的结果也不同。

建立了在 main()函数中使用的初始变量后，用下面的语句声明和初始化 pfun 数组:

```
int (*pfun[])(int, int) = {sum, product, difference};
```

这条语句声明了一个函数指针数组，该指针指向的函数有两个 int 类型的参数，返回类型是 int。数组用 3 个函数名初始化，所以数组包含 3 个元素。

接着根据是否定义了 repeatable 符号，用一个指令包含两个可选语句中的一个:

```
#ifdef repeatable
  srand(1);
#else
  srand((unsigned int)time(NULL)); /* Seed random number generation */
#endif
```

如果定义了 repeatable，用变元值 1 调用 srand()的语句就包含在源代码中，进行编译。这样，每次执行程序时，都会得到相同的结果。否则，就包含把 time()函数作为变元的语句，每次执行程序时，结果都不同。

main()函数中的循环如下：迭代次数由迭代符号的值确定，这里它是 6。循环中的第一个操作如下:

```
funsel = random(element_count);
if( funsel>element_count-1 )
```

www.TopSage.com

```
{
  printf("\nInvalid array index = %d", funsel);
  exit(1);
}
```

这段代码将 element_count 作为变元调用 random()宏。element_count 是 pfun 数组的元素个数，在循环执行之前计算。预处理器会在编译代码之前替换宏展开式中的 element_count。为了安全起见，检查 pfun 数组是否有有效的索引值。

之后的 3 个语句如下:

```
#ifdef test
  printf("\nRandom index = %d", funsel);
#endif
```

定义了 text 符号后，就把 printf()语句包含在代码中。如果删除了定义 text 的指令，printf()语句在编译时就不包含在程序中。

循环中的最后两条语句通过 pfun 数组中的一个指针调用函数，输出调用的结果:

```
result = pfun[funsel](a , b); /* Call random function */
printf("\nresult = %d", result );
```

下面看看一个被调用的函数 product():

```
int product( int x, int y )
{
  #ifdef testf
    printf("\nFunction product called args %d and %d.", x, y);
  #endif

  return x * y;
}
```

如果定义了 testf 符号，函数定义就包含一个输出语句。因此，可以控制#ifdef 块中的语句是否包含进来，它与 main()中由 test 控制的输出块相互独立。在定义了 test 和 testf 后，可以得到生成的随机索引值，以及每个函数调用时的信息，以了解程序调用的顺序。

可以定义任意多个不同的符号常量。也可以使用条件指令的#if defined 形式将它们组合到逻辑表达式中。

13.3.3 使用 assert()宏

assert()宏在标准库的头文件<assert.h>中定义。这个宏能在程序中插入测试用的任意表达式，如果表达式是 false(等于 0)，程序就中止，并输出一条诊断信息。assert()宏的变元是一个结果为整数的表达式，例如:

```
assert(a ==  b);
```

如果 a 等于 b，表达式的结果就是 true(非零)。如果 a 不等于 b，表达式的结果就是

www.TopSage.com

false，程序就输出一条相关的断言信息，然后中止。程序的中止是调用 abort()实现的，所以是不正常结束。调用 abort()时，程序会立即终止。流输出缓冲区是否刷新，打开的流是否关闭，临时文件是否删除，都取决于 C 的实现方式，所以应参阅编译器的文档说明。

在程序 13.1 中，可以使用断言验证 funsel 是否有效：

```
assert(funsel<element_count);
```

如果 funsel 不小于 element_count，表达式就是 false，所以程序终止。断言的一般输出如下：

```
Assertion failed: funsel<element_count d:\examples\program13_01.c 44
```

在#include <assert.h>语句之前定义 NDEBUG 符号，就可以关闭断言功能：

```
#define NDEBUG          /* Switch off assertions */
#include <assert.h>
```

这会忽略所有的断言。在某些系统中，assert 功能默认为关闭，此时可以取消 NDEBUG 符号的定义，打开该功能：

```
#undef NDEBUG         /* Switch on assertions */
#include <assert.h>
```

包含取消 NDEBUG 定义的指令，可以保证源文件开启断言功能。#undef 指令必须放在#include <assert.h>语句之前。

下面用一个简单例子去示范这个功能。

试试看：示范 assert()宏

这是一个使用 assert()宏的程序：

```
/* Program 13.2 Demonstrating assertions */
#undef NDEBUG /* Switch on assertions */
#include <stdio.h>
#include <assert.h>

int main(void)
{
  int y = 5;
  for(int x = 0 ; x < 20 ; x++)
  {
    printf("\nx = %d y = %d", x, y);
    assert(x<y);
  }
  return 0;
}
```

编译后执行的结果如下：

www.TopSage.com

```
x = 0   y = 5
x = 1   y = 5
x = 2   y = 5
x = 3   y = 5
x = 4   y = 5
x = 5   y = 5
Assertion failed: x<y , file Prog13_02.C, line 12
```

代码的说明

除了 assert()语句外，这个程序无须多做解释，它只是在循环中显示 x 和 y 的值。

条件 x<y 是 false 时，assert()宏就马上中止程序。从输出中可以看出，此时 x 的值是 5。这个宏将输出显示到 stderr 上，即屏幕。不仅会得到失败的条件，还会得到文件名和该文件中失败的行数。这对由许多文件组成的程序特别有用，因为它可以确切地指出错误的来源。

断言通常用于程序中的重要条件，如果不满足某个条件，就会出现灾难性后果。如果发生这类错误，程序就不应继续执行。

可以用#define NDEBUG 指令替换#undef 指令，关闭例子中的断言机制。该指令必须放在#include <assert.h>指令之前，但断言功能默认为关闭。程序 13.2 的开头由于有这个#define，所以会得到 x 从 0 到 19 的所有输出，不会出现诊断信息。

13.4 其他库函数

库函数是 C 语言强大的基础。前面已经介绍了许多标准库函数，但讨论所有的标准库函数超出了本书的范围。但是本节打算介绍一些目前还未使用的常用函数。

13.4.1 日期和时间函数库

时间是一个很重要的参数，所以 C 的标准库<time.h>包含了一些处理时间与日期的函数。它们可以根据计算机的硬件计时器提供各种不同格式的输出。

最简单的函数原型如下：

```
clock_t  clock(void);
```

这个函数返回程序自开始执行后的处理器时间(不是消逝的时间)，其类型是 clock_t。计算机一般在任意给定的时刻执行多个过程，处理器时间是处理器执行调用 clock()函数的过程所用的总时间。类型 clock_t 在<time.h>中定义，等价于 size_t 类型。clock()函数返回的值的单位是 tick。为了将它转换成秒，需要将这个值除以<time.h>中定义的宏 CLOCKS_PER_SEC 生成的值。执行 CLOCKS_PER_SEC 的结果是一秒内的 tick 数，且是 clock_t 类型。如果有错误，clock()函数就返回 - 1。

要确定执行过程所用的处理器时间，需要记录过程开始执行的时间，从中减去过程结束时返回的时间。例如：

www.TopSage.com

```
clock_t start, end;
double cpu_time;
start = clock();

/* Execute the process for which you want the processor time */

end = clock();
cpu_time = ((double)(end-start)/CLOCKS_PER_SEC;
```

这段代码将所使用的总处理器时间存储在 cpu_time 中。在最后一条语句中需要把它的类型转换为 double，以获得正确的结果。

time()函数可把日历时间返回为 time_t 类型的值。目前的日历时间是从一个固定的时间及日期算起的秒数。这个固定的时间及日期是 1970 年 1 月 1 日格林威治时间 0 点 0 分 0 秒。这也是一般的时间定义。

time()函数的原型如下：

```
time_t  time(time_t  *time_t);
```

如果变元不是 NULL，目前的日历时间就存储在 time_t 变元指向的位置中。类型 time_t 在<time.h>头文件中定义，等价于 long。

要以秒为单位计算连续调用两次 time()函数的时间差，可以使用函数 difftime()，它的原型是：

```
double  difftime(time_t  T2,  time_t  T1);
```

这个函数会返回 T2_T1 的数值，其类型是 double，单位是秒。这是两个 time()函数调用的时间差，这两个函数分别生成了 time_t 类型的值 T1 和 T2。

试试看：使用时间函数

可以使用<time.h>中的函数，定义一个函数，记录连续调用的时间差和处理器时间，如下：

```
/* Program 13.3 Test our timer function */
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <ctype.h>

int main(void)
{
  time_t calendar_start = time(NULL); /* Initial calendar time */
  clock_t cpu_start = clock();        /* Initial processor time */
  int count = 0;                      /* Count of number of loops */
  const int iterations = 1000000;     /* Loop iterations */
  char answer = 'y';

  printf("Initial clock time = %lu Initial calendar time = %lu\n",
```

www.TopSage.com

```
        cpu_start, calendar_start);

while(tolower(answer) == 'y')
{
  for(int i = 0 ; i<iterations ; i++)
  {
    double x = sqrt(3.14159265);
  }
  printf("\n%ld square roots completed.", iterations*(++count));
  printf("\nDo you want to run some more(y or n)? ");

  scanf("\n%c", &answer);
}

clock_t cpu_end = clock(); /* Final cpu time */
time_t calendar_end = time(NULL); /* Final calendar time */

printf("\nFinal clock time = %lu Final calendar time = %lu\n",
        cpu_end, calendar_end);

printf("\nCPU time for %ld iterations is %.2lf seconds\n",
    count*iterations, ((double)(cpu_end-cpu_start))/CLOCKS_PER_SEC );

printf("\nElapsed calendar time to execute the program is %8.2lf\n",
   difftime(calendar_end, calendar_start));

return 0;
}
```

得到如下的输出:

```
Initial clock time = 0 Initial calendar time = 1155841882

1000000 square roots completed.
Do you want to run some more(y or n)? y

2000000 square roots completed.
Do you want to run some more(y or n)? n

Final clock time = 7671 Final calendar time = 1155841890

CPU time for 2000000 iterations is 7.67 seconds

Elapsed calendar time to execute the program is 8.00
```

代码的说明

这个程序演示了函数 clock()、time()和 difftime()的用法。time()函数返回当前时间(单位是秒)，因此不会得到比 1 小的值。根据计算机的速度，可以调整循环的迭代次数，以减少或增加执行这个程序的时间。注意，函数 clock()不能精确地确定程序所用的处理器时间。然而，可以使用 time()函数确定消逝的时间。

www.TopSage.com

使用下列语句记录并显示处理器时间及日历时间的初始值，并设置循环的控制变量:

```
time_t calendar_start = time(NULL); /* Initial calendar time */
clock_t cpu_start = clock();        /* Initial processor time */
int count = 0;                      /* Count of number of loops */
const int iterations = 1000000;     /* Loop iterations */
char answer = 'y';
printf("Initial clock time = %lu Initial calendar time = %lu\n",
    cpu_start, calendar_start);
```

在 C 库中，类型 time_t 和 clock_t 都在<time.h>头文件中定义为 unsigned long 类型。读者应查阅自己的库中这些类型的定义，调整这些值的格式指定符。

接下来的循环用 answer 中的字符控制，只要希望循环继续，它就会继续执行:

```
while(tolower(answer) == 'y')
{
  for(int i = 0 ; i<iterations ; i++)
  {
    double x = sqrt(3.14159265);
  }

  printf("\n%ld square roots completed.", iterations*(++count));
  printf("\nDo you want to run some more(y or n)? ");

  scanf("\n%c", &answer);
}
```

内层循环调用在<math.h>头文件中定义的 sqrt()函数 iterations 次，所以这会占用一些处理器时间。如果对提示输入的响应不是 y，就会延长消逝的时间。注意 scanf()中第一个变元开头的换行符转义序列。如果遗漏了它，程序就会无限循环下去，因为 scanf()不会忽略输入流缓冲区中的空白字符。

最后，输出 clock()及 time()返回的值，并计算处理器及日历时间的间隔:

```
clock_t cpu_end = clock();          /* Final cpu time */
time_t calendar_end = time(NULL);   /* Final calendar time */

printf("\nFinal clock time = %lu Final calendar time = %lu\n",
  cpu_end, calendar_end);

printf("\nCPU time for %ld iterations is %.2lf seconds\n",
  count*iterations, ((double)(cpu_end-cpu_start))/CLOCKS_PER_SEC );

printf("\nElapsed calendar time to execute the program is %8.2lf\n",
  difftime(calendar_end, calendar_start));
```

C 编译器包含的库可能有其他非标准的函数来获得处理器时间，它比 clock()可靠。

警告:

处理器时钟可能会重置为 0，所测量的处理器时间的精度随硬件平台的不同而不同。

www.TopSage.com

例如，如果处理器时钟是 32 位值，其精度为 1 毫秒，该时钟会每隔 72 分钟重置为 0。

13.4.2 获取日期

有一个自 25 年前开始算起的时间(秒)是很重要的，但使今天的日期显示为字符串更方便。为此，可以使用函数 ctime()，它的原型如下：

```
char *ctime(const time_t  *timer);
```

这个函数接受一个 time_t 变量的指针作为变元，它含有 time()函数返回的日历时间值。它返回一个指向 26 个字符的字符串的指针，其中有星期、日期、时间以及年，最后用一个换行符和'\0'终止。

返回的字符串如下：

```
"Mon  Aug  25  10:45:56  2003\n\0"
```

使用 ctime()函数：

```
time_t calendar = 0;
calendar = time(NULL);                    /* Store calendar time */
printf("\n%s", ctime(&calendar)); /* Output calendar time as date string */
```

也可使用库函数 localtime()得到日历时间中的时间和日期的各个组成部分。这个函数的原型如下：

```
struct tm *localtime(const time_t *timer);
```

这个函数接受一个 time_t 值的指针，并返回结构类型 tm 的指针，结构类型 tm 在<time.h>头文件中定义。这个结构包含的成员列在表 13-1 中。

表 13-1 tm 结构的成员

成　员	说　明
tm_sec	24 小时制，分钟后的秒数
tm_min	24 小时制，小时后的分钟(0~59)
tm_hour	24 小时制中的小时(0~23)
tm_mday	月份中的日(1~31)
tm_mon	月份(0~11)
tm_year	年份(当前年份减去 1900)
tm_wday	星期(星期天是 0，星期六是 6)
tm_yday	年份中的日(0~365)
tm_isdst	白天存储标记，正数表示白天存储标记，0 表示非白天存储标记，－1 表示未知

www.TopSage.com

结构的所有成员都是 int 类型。每次调用 localtime()函数，都会返回一个指向该结构的指针，在每次调用中，所有的结构成员都会被覆盖。如果要保留它们的值，必须在下次调用 localtime()之前，将它们复制到别的地方去，或者创建自己的 tm 结构，然后存储所有成员的值。

localtime()函数生成的时间是本地时间。如果要使 tm 结构的时间反映 UTC(世界调整时间)，就可以使用 gmtime()函数，它也需要一个 time_t 类型的变元，返回一个 tm 结构指针。

以下是输出 tm 结构成员的天及日期的程序片段：

```
time_t calendar = 0;                              /* Holds calendar time */
struct tm *time_data;                          /* Holds address of tm struct */
const char *days[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
                      "Thursday", "Friday", "Saturday" };
constchar *months[] = {"January", "February", "March",
                       "April", "May", "June",
                       "July", "August", "September",
                       "October", "November", "December" };
calendar = time(NULL);                  /* Get current calendar time */
printf("\n%s", ctime(&calendar));
time_data = localtime(&calendar);
printf("Today is %s %s %d %d\n",
       days[time_data->tm_wday], months[time_data->tm_mon],
       time_data->tm_mday, time_data->tm_year+1900);
```

这段代码定义了字符串数组来存储星期以及月。然后，调用 localtime()函数，设置适当的 tm 结构成员。可以直接使用该结构的月份和年份中的日，也可以扩展程序，输出时间。

试试看：获取日期

从 localtime()函数返回的 tm 结构中输出成员是很容易的。以下是一个示例：

```
/* Program 13.4 Getting date data with ease */
#include <stdio.h>
#include <time.h>

int main(void)
{
  const char *Day[7] = {
                        "Sunday" , "Monday", "Tuesday", "Wednesday",
                        "Thursday", "Friday", "Saturday"
                        };
  const char *Month[12] = {
                        "January", "February", "March", "April",
                        "May", "June", "July", "August",
                        "September", "October", "November", "December"
                        };
  const char *Suffix[4] = { "st", "nd", "rd", "th" };
```

www.TopSage.com

```
enum sufindex { st, nd, rd, th } sufsel = th; /* Suffix selector */
 struct tm *OurT = NULL;          /* Pointer for the time structure */
time_t Tval = 0;                  /* Calendar time */

Tval = time(NULL);                /* Get calendar time */
OurT = localtime(&Tval);          /* Generate time structure */

switch(OurT->tm_mday)
{
  case 1: case 21: case 31:
    sufsel= st;
    break;
  case 2: case 22:
    sufsel= nd;
    break;
  case 3: case 23:
    sufsel= rd;
    break;
  default:
    sufsel= th;
    break;
}

printf("Today is %s the %d%s %s %d", Day[OurT->tm_wday],
      OurT->tm_mday, Suffix[sufsel], Month[OurT->tm_mon], 1900 +
      OurT->tm_year);
  printf("\nThe time is %d : %d : %d",
      OurT->tm_hour, OurT->tm_min, OurT->tm_sec );
  return 0;
}
```

下面是程序的输出:

```
Today is Friday the 18th August 2006
The time is 11 : 49 : 53
```

代码的说明

在这个例子里，main()函数中的声明如下:

```
const char *Day[7] = {
                      "Sunday" , "Monday", "Tuesday", "Wednesday",
                      "Thursday", "Friday", "Saturday"
                      };
const char *Month[12] = {
                      "January", "February", "March", "April",
                      "May", "June", "July", "August",
                      "September", "October", "November", "December"
                      };
const char *Suffix[] = { "st", "nd", "rd", "th" };
```

www.TopSage.com

每个数组都定义成字符指针数组。第一个数组包含星期的天，第二个数组含有年的月份，第三个数组包含表示日期时的日的尾字符。前两个数组的大小本来可以留给编译器计算，但这个例子可以保证这些数字不会有问题，所以填入这些数字，以避免错误。const 修饰符指定，指针指向的字符串是常量，不能在代码中修改。

枚举提供了从 Suffix 数组中选择一个元素的机制:

```
enum sufindex { st, nd, rd, th } sufsel = th; /* Suffix selector */
```

枚举常量 st、nd、rd 和 th 默认赋予 0~3 的值，所以可以将 sufsel 变量用作索引，来访问 Suffix 数组中的元素。

还声明了一个结构变量，具体如下:

```
struct tm *OurT = NULL; /* Pointer for the time structure */
```

这里提供了空间来存储函数 local_time()返回的结构指针。

首先，使用函数 time()得到当前时间，存储到 Tval 中。然后，利用这个值为 localtime()函数返回的结构生成成员的值。如果要保留结构中的数据，必须在再次调用 localtime()之前先复制它们，因为它们会被覆盖。从 localtime()得到结构后，就执行以下的 switch:

```
switch( OurT->tm_mday )
{
  case 1: case 21: case 31:
    sufsel= st;
    break;
  case 2: case 22:
    sufsel= nd;
    break;
  case 3: case 23:
    sufsel= rd;
    break;
  default:
    sufsel= th;
    break;
}
```

这里唯一的目的是选择哪个尾字符添加到日期值的后面。输出日期时，会根据 tm_mday 成员，将 sufsel 变量设置为对应的枚举常量值，在 switch 中选择 Suffix[]数组的一个索引。

根据结构成员的值去索引适当的数组，就得到了日及月的字符串，然后显示星期、日期以及时间。给成员 tm_year 加 1900，是因为这个值是相对 1900 年估算出来的。

还可以使用 mktime()函数确定给定日期是星期几。该函数的原型如下:

```
time_t mktime(struct tm *ptime);
```

给该函数传送 tm 结构的指针，并将 tm_mon、tm_day 和 tm_year 值设置为需要的日期。忽略结构中 tm_wkday 和 tm_yday 成员的值，如果操作成功，就用所提供日期的正

www.TopSage.com

确值替代这些值。如果操作成功，该函数就把日历时间返回为 time_t 类型的值，如果日期不能表示为 time_t 值，使操作失败，该函数就返回 - 1。下面看一个例子。

试试看：确定某一天是星期几

从 localtime()函数返回的 tm 结构中输出成员是很容易的。以下是一个示例：

```
/* Program 13.5 Getting the day for a given date */
#include <stdio.h>
#include <time.h>

int main(void)
{
  const char *Day[7] = {
                        "Sunday" , "Monday", "Tuesday", "Wednesday",
                        "Thursday", "Friday", "Saturday"
                        };
  const char *Month[12] = {
                        "January", "February", "March", "April",
                        "May", "June", "July", "August",
                        "September", "October", "November", "December"
                        };
  const char *Suffix[4] = { "st", "nd", "rd", "th" };
  enum sufindex { st, nd, rd, th } sufsel = th; /* Suffix selector */

  int day = 0;          /* Stores a day... */
  int month = 0;        /* month... */
  int year = 0;         /* and year for a date */

  struct tm birthday; /* A birthday time structure */

  /* Set the structure members we don't care about */
  birthday.tm_hour = birthday.tm_min = 0;
  birthday.tm_sec = 1;
  birthday.tm_isdst = -1;

  printf("Enter your birthday as integers, day month year."
     "\ne.g. Enter 1st February 1985 as 1 2 1985. : ");
  scanf(" %d %d %d", &day, &month, &year);

  birthday.tm_mon = month-1;
  birthday.tm_mday = day;
  birthday.tm_year = year-1900;

  if(mktime(&birthday) == (time_t)-1)
  {
    printf("\nOperation failed.");
    return 0;
  }
```

www.TopSage.com

```
    switch(birthday.tm_mday)
    {
        sufsel= st;
        break;
      case 2: case 22:
        sufsel= nd;
        break;
      case 3: case 23:
        sufsel= rd;
        break;
      default:
        sufsel= th;
        break;
    }

    printf("\nYour birthday, the %d%s %s %d, was a %s",
        birthday.tm_mday, Suffix[sufsel], Month[birthday.tm_mon],
        1900 + birthday.tm_year, Day[birthday.tm_wday]);

    return 0;
}
```

程序的输出如下：

```
Enter your birthday as integers, day month year.
e.g. Enter 1st February 1985 as 1 2 1985. : 15 6 1985

Your birthday, the 15th June 1985, was a Saturday
```

代码的说明

这个例子为月份中的日和日期的尾字符创建常量字符串数组，与程序 13.4 相同。接着创建一个 tm 结构，初始化它的一些成员：

```
struct tm birthday; /* A birthday time structure */

/* Set the structure members we don't care about */
birthday.tm_hour = birthday.tm_min = 0;
birthday.tm_sec = 1;
birthday.tm_isdst = -1;
```

将 tm_isdst 成员的值设置为 - 1，因为不知道它是否应用于输入的日期。

输出提示后，从键盘上读取某个生日的日、月和年，在 birthday 结构中设置输入值：

```
printf("\nEnter your birthday as integers, day month year."
       "\ne.g. Enter 1st February 1985 as 1 2 1985. : ");
scanf(" %d %d %d", &day, &month, &year);

birthday.tm_mon = month-1;
birthday.tm_mday = day;
birthday.tm_year = year-1900;
```

www.TopSage.com

设置了日期后，就可以调用 mktime()函数，得到设置的 tm_wday 和 tm_yday 成员：

```
if(mktime(&birthday) == (time_t)-1)
{
  printf("\nOperation failed.");
  return 0;
}
```

if 语句检查函数是否返回-1，这表示操作失败。此时输出一条信息，终止程序。最后，显示所输入的生日日期是星期几，方法与上一个例子相同。

13.5 小结

本章讨论的预处理器指令可以在编译之前处理和转换源文件中的代码。标准库的头文件是编写预处理器指令的绝佳例子。可以用任何文本编辑器浏览这些例子。预处理器的所有功能都在库里使用了，还有许多 C 源代码。这也有助于熟悉库的内容，因为库的文档说明没有许多必要的描述。例如，如果不知道 clock_t 类型是什么，可以查看<time.h>头文件中的定义。

预处理器提供的调试功能很有用，但许多 C 编程系统提供的调试工具更强大。对于大型程序开发，调试工具与编译器的性能一样重要。

如果读者此时对所学的内容很有自信，可以理解及应用它们，说明现在您应该是一个训练有素的 C 程序设计师了。还应勤加练习。练习是把程序写好的不二法门，所编写的程序类型越多越好。我们可以提高技巧，但永远也不可能到达完美的境界，因为不可能一开始就能写出任意大小、没有任何错误的程序。然而，每当完成一小部分新程序时，它所带来的兴奋与满足是很值得期待的。好好享受编程吧!

13.6 习题

以下的习题可测试读者对本章的掌握情况。如果有不懂的地方，可以翻看本章的内容。还可以从 Apress 网站 http://www.apress.com 的 Source Code/Download 部分下载答案，但这应是最后一种方法。

习题 13.1　定义一个 COMPARE(x，y)宏。如果 x < y，就返回 - 1，如果 x == y，就返回 0，如果 x > y 就返回 1。编写一个例子，说明这个宏可以正常工作。这个宏会优于完成相同任务的函数吗?

习题 13.2　定义一个函数，返回含有当前时间的字符串，如果变元是 0，它的格式就是 12 小时制(a.m./p.m.)，如果变元是 1，它的格式就是 24 小时制。编写一个程序，说明这个函数可以正常工作。

习题 13.3　定义一个 print_value(expr)宏，在新的一行上输出 exp = result，其中 result 的值由 expr 算出。编写一个例子，说明这个宏可以正常工作。

www.TopSage.com

www.TopSage.com

附录 A

计算机中的数学知识

在本书中，作者已经对计算机中的数学知识做了简单的讨论。不过，它是很重要的，所以本附录将快速地浏览一遍这个主题。如果你对自己的数学知识信心十足，可以略过。如果你觉得数学很难，那么本附录会向你展示它实际上是多么容易。

大学网 TopSage.com

二进制数

首先，让我们好好地思考一下一个常见的、每天都在使用的十进制数，如 324 或 911。很明显，它是“三百二十四”或“九百一十一”。更精确地说，它是：

324 是指 $3\times10^2+2\times10^1+4\times10^0$，也可以是 $3\times10\times10+2\times10+4$
911 是指 $9\times10^2+1\times10^1+1\times10^0$，也可以是 $9\times10\times10+1\times10+1$

我们称之为十进制符号，因为它构建在 10 的幂上。这是从中世纪拉丁语 decimalis 演变而来的，表示“十分之一的”，即税收的 10%。

用这种方式表示数字，对于 10 个手指、10 个脚趾或其他任何 10 种附属肢体的人来说，非常方便。但是，对于主要由开关(用 on 和 off 表示)来实现操作的计算机来说，就没那么方便了。让计算机累积计算到 2 还没有问题，但让它累积计算到 10 就不那么容易了。这就是为什么计算机采用 2 为基数表示数字，而不是以 10 为基数的主要原因。以 2 为基数表示数字，称之为二进制计数系统。如果是以 10 为基数表示数字，阿拉伯数字是 0~9，而二进制数中的数字只能是 0 或 1，对于只有开/关状态的开关来说，表示其开关状态是非常方便的。按照十进制数的计数方式，二进制数 1101 可以被分割为下面的形式：

$1\times2^3+1\times2^2+1\times2^1+1\times2^0$ 即 $1\times2\times2\times2+1\times2\times2+0\times2+1$

在十进制系统中，它等于 13。在表 A-1 中，可以看到用 8 位二进制数字(每个二进制数字通常叫做一位)能表示的所有十进制数。

表 A-1　8 位二进制数对应的十进制数

二 进 制 数	十 进 制 数	二 进 制 数	十 进 制 数
0000 0000	0	1000 0000	128
0000 0001	1	1000 0001	129
0000 0010	2	1000 0010	130
…	…	…	…

www.TopSage.com

(续表)

二 进 制 数	十 进 制 数	二 进 制 数	十 进 制 数
0001 0000	16	1001 0000	144
0001 0001	17	1001 0001	145
…	…	…	…
0111 1100	124	1111 1100	252
0111 1101	125	1111 1101	253
0111 1110	126	1111 1110	254
0111 1111	127	1111 1111	255

注意，用前 7 位可以表示 0~127 的数，共 2^7 个，用 8 位可以得到 256 个或 2^8 个数。一般说来，如果有 n 位，就能表示 2^n 个整数，值为 0~2^n - 1。

在计算机中对二进制数执行加法，非常容易。因为对相应的位执行加法生成的借位只能是 0 或 1，用一个很简单的电路就能处理这个过程。图 A-1 展示了两个 8 位二进制数值的加法操作是如何执行的。

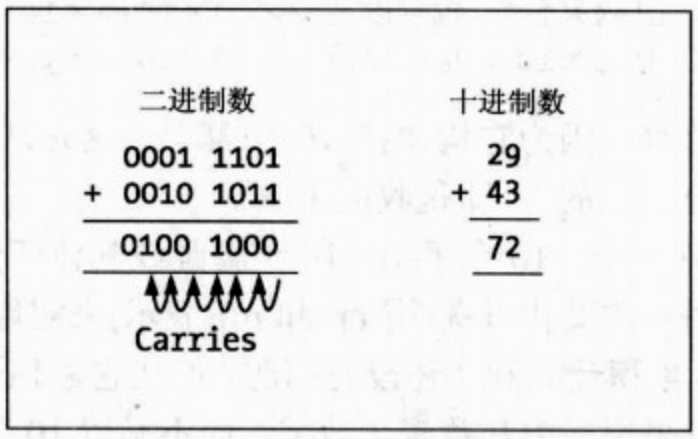


图 A-1 二进制数值的加法

十六进制数

在处理较大的二进制数时，会出现一些小问题。看看下面这个数：

```
1111 0101  1011 1001  1110  0001
```

在实际应用中，二进制符号显得有此累赘，尤其是当你考虑要用这个二进制数的十进制数时，它只是 16 103 905，只有 8 个十进制数字。显然，我们需要一种更经济的方法来书写这样的数，但是十进制并不总是适用的。有时(如你在第 3 章所见)你可能需要把从右边算起的第 10 位和第 24 位数字设置为 1，而不需要大费周折地用二进制符号写出所有的数字。用十进制整数完成这种要求也非常困难，而且很可能会出错。一种容易得多的方法是使用十六进制符号，其中的数字是以 16 为基数表示的。

以 16 为基数的算术方法是相当方便的选择，比二进制更加适合。每个十六进制数的

www.TopSage.com

值可以是 0~15(10~15 的数字由字母 A~F 表示，如表 A-2 所示)的任一个数，0~15 这个范围中的值恰好对应于 4 位二进制数字表示的值的范围。

表 A-2 十六进制数字及其对应的二进制和十进制数字

十六进制数	十进制数	二进制数
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

由于一个十六进制数字对应于 4 位二进制数字，所以只需要把一个二进制数从右边开始分为 4 位一组，然后用对应的十六进制数字代替每组二进制数字，就可以用十六进制数表示一个较大的二进制数。例如下面的二进制数：

```
1111 0101 1011 1001 1110 0001
```

依次每 4 位一组，把每一组替换为表 A-2 中对应的十六进制数字，这个数用十六进制符号表示的形式如下所示：

```
F  5  B  9  E  1
```

此时得到 6 个十六进制数字，对应于 6 组 4 位二进制数字。为了证明这种做法没有欺骗性，可以把这个数直接转换成十进制数。下面对这个十六进制数进行几次转换。

F5B9E1 的十进制值是通过下面的表达式得来的：

$$15\times16^5+5\times16^4+11\times16^3+9\times16^2+14\times16^1+1\times16^0$$

它可以转换为：

www.TopSage.com

```
15,728,640 + 327,680 + 45,056 + 2,304 + 224 + 1
```

这个表达式的结果与对应的二进制转换成十进制数所得的结果相同，都是 16 103 905。

二进制负数

关于二进制算术，还有一点需要了解，即负数。到现在，我们都是假定所有事物都是正面的；乐观主义者认为，只要你愿意，杯子总是半满的。但是生活的负面是无法避免的，悲观主义者认为，杯子已经空了一半了。那么在计算机中怎么表示负数呢？在计算机中，你所能用的只有二进制数字，因此解决方案也只能用一位二进制数字表示一个数是正的还是负的。

如果允许有负值的数(称之为有符号数)，首先必须决定它的长度(简而言之，就是它的二进制位数)，然后指派最左边的二进制数字作为符号位。必须指定这种数的长度，以免搞不清楚哪一位是符号位。

由于计算机内存是由 8 位的字节构成的，所以二进制数要存储在多个(通常是 2 的幂)8 位的字节中。因此，有的数是 8 位的，有的数是 16 位的，而有的数是 32 位(或更多位)的，只要你知道每种情况下采用的位数，就能找到符号位，就是最左边的位。如果符号位是 0，这个数是正数，如果符号位是 1，该数就为负数。

这种方法看起来解决了问题，在某些计算机上也确实解决了问题。每个数由符号位和其他数位构成，符号位为 0 的，就是正数值，符号位为 1 的，就是负数值，其他数位指定了这个数的绝对值，换句话说，就是无符号的值。将+6 改成 - 6 只需要把符号位从 0 改成 1。遗憾的是，要执行用这种数值表示法构成的算术运算，需要复杂的电路才能解决。因此，大多数计算机采用的都是另外一种方法。

在理想情况下，要对两个整数执行加法运算，计算机不必关心某个或两个运算数是正数还是负数。只要使用简单的“加法”电路，无论运算数的符号是什么，加法运算将把对应的二进制数字加在一起，生成正确的数字位作为结果，如果有进位，就与下一个数位进行运算。如果把二进制数 - 8 加到+12 上，那么你一定想用计算+3 加+8 的电路来得到答案+4。

如果用你所想的最简单的解决方法，即把正数的符号位设置为 1，使它成为负数，然后用传统的进位法执行算术运算，结果就不正确了：

以二进制表示+12 是 0000 1100；

以二进制表示 - 8(你的假设)是 1000 1000；

如果将它们相加，结果是 1001 0100。

这意味着结果是 - 20，根本不是你想要的。它绝对不是+4，+4 对应的二进制值是 0000 0100。这时，你一定会说，“你不能把符号位也作为数字处理。”但是，上面的所为正是你想做的。

让我们看看计算机是如何用+4 减去+12 来表示 - 8 的，它应该能给你正确答案了：

以二进制表示+4 是 0000 0100；

以二进制表示+12 是 0000 1100；

www.TopSage.com

前者减去后者的结果是 1111 1000。

从右边数第 4 位数字开始，就需要通过借位来执行减法，就像执行普通的十进制算术运算一样。结果应该是 - 8，即使看起来不像，但它的确是 - 8。试试看用它与二进制的+12 或+15 相加，就能看到结果正确。

当然，如果你想生成 - 8，用 0 减+8 就可以办到。

那么用+4 减去+12 或者用 0 减去+8，得到的到底是什么呢?结果，这里你所得到的是 2 的补码形式，它表示二进制负数；用简单的心算就可以从二进制正数生成这种补码形式。这里，我需要你相信我所讲的，以避免解释这种方法为什么有效。我只能向你展示如果从一个正数值构成一个负数的二进制补码形式，你可以自己证明这种方法是可行的。让我们再看看上一个例子，其中需要 - 8 的二进制补码表示法。

从+8 的二进制形式着手：

```
0000 1000
```

现在对每个二进制数字求反，将 0 变成 1，1 变成 0：

```
1111 0111
```

这叫做 1 的二进制的补码形式，如果给这个数加 1，就能得到 2 的二进制补码形式：

```
1111 1000
```

这与用+4 减去+12 得到的 - 8 的表示完全相同。为了确保万无一失，让我们来看看把 - 8 加到+12 上的普通运算：

以二进制表示+12 是 0000 1100；

以二进制表示 - 8 是 1111 1000；

如果将它们加在一起，结果为 0000 0100

神奇吧！答案是 4。这种方法是有效的。进位会传递到最左边的所有 1 上，把它们都设置回 0。有一位会超过界限，不过不用担心，它可能弥补了在做减法得到 - 8 时末尾所借的数位。事实上，这里有一个假设，即符号位 1 或 0 会重复出现，直到最左边的数位。试做练习，你会发现这种方法总是有效的。使用负数的二进制补码形式的真正伟大之处在于大大简化(并加快)了计算机的算术运算。

Big-Endian 和 Little-Endian 系统

前面说过，整数通常以二进制的形式存储在内存中连续的字节序列里，以 2 个、4 个或者 8 个字节为一组。字节出现的序列非常重要，它就是那种看似无关紧要，但是到用的时候，就会发现它是真的很重要的东西。

让我们看看以 4 个字节的二进制值存储的十进制值 262 657。之所以选择这个值，是因为它的二进制值恰好是：

```
0000 0000 0000 0100 0000 0010 0000 0001
```

www.TopSage.com

这样每个字节都有一种位模式，区分起来较为容易。

如果你使用的是 Intel PC，这个数字将被存储为：

```
字节地址:  00          01          02          03
数据位:    0000 0001   0000 0010   0000 0100   0000 0000
```

如你所见，最有效的 8 位数值全都是 0，存储在地址最高的字节(也就是最后的字节)中，最低效的 8 位数值存储在地址最低的字节中，即最左边的字节。这种摆放方法叫做 little-endian(小尾数法)。

如果使用的是大型机，如 RISC 工作站或基于 Motorola 处理器的 Mac 机，同样的数据，在内存中摆放的序列如下：

```
字节地址:  00          01          02          03
数据位:    0000 0000   0000 0100   0000 0010   0000 0001
```

现在，字节是按照逆序摆放的，最有效的 8 位存储在最左边的字节中，即地址最低的字节。这种摆放方法叫做 big-endian(大尾数法)。

注意：

在每个字节中，无论字节顺序是 big-endian 还是 little-endian，最有效的位摆放在左边，最低效的位摆放在右边。

你也许会说，这一点非常有趣，但为什么它会造成麻烦呢?大多数时候，它并不构成问题。通常，在编写 C 语言程序时，都不需要知道执行这些代码的计算机是 big-endian 的，还是 little-endian 的。不过，在处理来自另一台机器的二进制数据时，这一点就很重要。二进制数据会被写入一个文件，然后通过网络以字节序列的形式传递。至于如何解释它，则由你决定。如果数据源所用的计算机采用的字节摆放方式与运行代码的计算机所采用的不同，那么必须对每个二进制值执行逆序操作。如果不这样做，所得到的数据则都是垃圾。

注意：

有些人对背景信息非常好奇。术语 big-endian 和 little-endian 来自于 Jonathan Swift 所写的 Gulliver's Travels 一书。在这本书的故事中，小人国的国王命令他所有的随从在吃蛋时都先打破蛋的小端。这是因为国王的儿子在以传统的方式先打破蛋的大端时割破了手指。遵纪守法的小人国居民都从较小的一端打破蛋，被称为 Little Endian。Big Endian 则是小人国王国的叛军，他们坚持从大端打破蛋。他们中的许多人都被判了死刑。

浮点数

我们经常需要处理非常大的数，例如，宇宙的质子数，大概需要 79 位十进制数字。显然，在很多情况下，需要的十进制数位都超过了 10 个，而 4 字节的二进制数只能表示 10 位以内的十进制数。同样，还有一些非常小的数。处理这两种情况采用的是同一种机

www.TopSage.com

制，即浮点数。

用十进制符号表示数的浮点形式是具有固定位数的小数值乘以 10 的幂，生成你想要的值。用例子来解释更容易理解，所以让我们来看一些示例。用普通的十进制符号表示的 365 可以写成如下的浮点形式：

```
0.3650000E03
```

E 表示指数，它位于 10 的幂之前，0.3650000(尾数)乘以 10 的幂，可以得到需要的值，即：

```
0.3650000×10×10×10
```

显然，它等于 365。

这个数中的尾数有 7 位十进制数字。浮点数中数字的精度是由分配给它的内存决定的。单精度浮点值占用 4 个字节，基本上可以提供 7 位十进制数字的精度。之所以说“基本上”是因为在计算机中这些数是以二进制浮点形式存储的，一个 23 位的二进制小数并不能完全对应于一个 7 位数字的十进制小数。

现在让我们来看一个比较小的数：

```
0.3650000E-04
```

它等于 0.365×10^{-4}，即 0.0000365。

假设有一个像 2134 311 179 这样大的数，那么用浮点数表示是什么样的呢？正好如下所示：

```
0.2134311E10
```

它们并不完全一样。这里丢失了最低的 3 位数字，把原始值变成了 2 134 311 000。这是处理这么大的数时需要付出的小代价，通常可以表示的数的范围是 10^{-308}~10^{+38}，可以是正数也可以是负数，还有扩展的表示法，可以表示的范围是 10^{-308} 到 10^{+308}。它们之所以会被称为“浮点数”，一个显而易见的原因在于小数点是浮动的，它的位置是由指数值决定的。

除了固定的精度限制外，还有一点需要注意。在对两个大小差别很大的数执行加法或减法运算时，要格外小心。用一个简单的例子就能说明这个问题。首先考虑 0.365E-3 加 0.365E+7，下面用十进制和的形式写这个算式：

```
0.000365+3,650,000.0
```

这个算式生成的结果如下：

```
3,650,000.000365
```

把它转换成具有 7 位精度的浮点数，将变为：

```
0.3650000E+7
```

www.TopSage.com

也许这对你并不构成问题。这个问题是直接由仅能保留 7 位精度引起的。较大的数的 7 位数字不会受到影响，因为舍去的都是右边的数位。当数字几乎相等时，必须格外小心。如果计算两个几乎相等的数的差，那么得到的结果可能只有 1 位或 2 位的精度。在这种情况下，计算得到的数很可能都是垃圾数。

www.TopSage.com

附录 B

ASCII 字符代码定义

美国国家信息交换标准代码(ASCII)的前 32 个字符提供了控制功能。有许多字符在本书里没有提及到，下面将它们汇总出来，以供参考。表 B-1 只包含前 128 个字符。其余的 128 个字符包含特殊符号及公共通用字符集。

表 B-1　ASCII 字符代码值

十 进 制	十六进制	字　符	控 制 符
000	00	Null	NUL
001	01	☺	SOH
002	02	●	STX
003	03	♥	ETX
004	04	♦	EOT
005	05	♣	ENQ
006	06	♠	ACK
007	07	•	BEL(发出哔声)
008	08	--	Backspace
009	09	--	HT
010	0A	--	LF(换行)
011	0B	--	VT(垂直跳位)
012	0C	--	FF(进纸)
013	0D	--	CR(回车)
014	0E	--	SO
015	0F	--	S1
016	10	--	DLE
017	11	--	DCl
018	12	--	DC2
019	13	--	DC3
020	14	--	DC4
021	15	--	NAK

www.TopSage.com

(续表)

十 进 制	十六进制	字 符	控 制 符
022	16	--	SYN
023	17	--	ETB
024	18	--	CAN
025	19	--	EM
026	1A	→	SUB
027	1B	←	ESC(回退)
028	1C	└	FS
029	1D	--	GS
030	1E	--	RS
031	1F	--	US
032	20	--	Space
033	21	!	--
034	22	"	--
035	23	#	--
036	24	$	--
037	25	%	--
038	26	&	--
039	27	'	--
040	28	(	--
041	29	)	--
042	2A	*	--
043	2B	+	--
044	2C	,	--
045	2D	-	--
046	2E	.	--
047	2F	/	--
048	30	0	--
049	31	1	--
050	32	2	--
051	33	3	--
052	34	4	--
053	35	5	--

www.TopSage.com

(续表)

十 进 制	十六进制	字 符	控 制 符
054	36	6	--
055	37	7	--
056	38	8	--
057	39	9	--
058	3A	:	--
059	3B	;	--
060	3C	<	--
061	3D	=	--
062	3E	>	--
063	3F	?	--
064	40	@	--
065	41	A	--
066	42	B	--
067	43	C	--
068	44	D	--
069	45	E	--
070	46	F	--
071	47	G	--
072	48	H	--
073	49	I	--
074	4A	J	--
075	4B	K	--
076	4C	L	--
077	4D	M	--
078	4E	N	--
079	4F	O	--
080	50	P	--
081	51	Q	--
082	52	R	--
083	53	S	--
084	54	T	--
085	55	U	--

www.TopSage.com

(续表)

十 进 制	十六进制	字 符	控 制 符
086	56	V	--
087	57	W	--
088	58	X	--
089	59	Y	--
090	5A	Z	--
091	5B	[	--
092	5C	\	--
093	5D	]	--
094	5E	^	--
095	5F	_	--
096	60	'	--
097	61	a	--
098	62	b	--
099	63	c	--
100	64	d	--
101	65	e	--
102	66	f	--
103	67	g	--
104	68	h	--
105	69	i	--
106	6A	j	--
107	6B	k	--
108	6C	l	--
109	6D	m	--
110	6E	n	--
111	6F	o	--
112	70	p	--
113	71	q	--
114	72	r	--
115	73	s	--
116	74	t	--
117	75	u	--

www.TopSage.com

(续表)

十 进 制	十六进制	字 符	控 制 符
118	76	v	--
119	77	w	--
120	78	x	--
121	79	y	--
122	7A	z	--
123	7B	{	--
124	7C	|	--
125	7D	}	--
126	7E	~	--
127	7F	Delele	--

注意，这个表中的空字符不必使用 NULL，它是在 C 语言库中定义的，其值是独立执行的。

www.TopSage.com

www.TopSage.com

附录 C

C 语言中的保留字

下面所列的单词都是 C 语言中的保留字，绝不可以使用它们作为变量名或函数名。

```
Auto        for         struct
break       goto        switch
case        if          typedef
char        inline      union
const       int         unsigned
continue    long        void
default     register    volatile
do          restrict    while
double      return      _Bool
else        short       _Complex
enum        signed      _Imaginary
extern      sizeof
float       static
```

www.TopSage.com

www.TopSage.com

附录 D

输入输出格式指定符

本附录总结流输入输出的所有格式指定符，它们用于标准流 stdin、stdout、stderr 和文本文件流。

D.1 输出格式指定符

格式化输出有 3 个标准库函数：printf()函数将输出写到标准输出流 stdout 上(默认为命令行)，sprintf()函数将输出写到字符串中，fprintf()函数将输出写到文件中。这些函数的格式如下：

```
int printf(const char* format_string, . . .);
int sprintf(char* source_string, const char* format_string, . . .);
int fprintf(FILE* file_stream, const char* format_string, . . .);
```

参数列表末尾的省略号表示，可以提供 0 个或多个变元。这些函数返回写出的字节数，如果出现错误，就返回一个负值。格式字符串可以包含写到输出中的原始字符(包括转义序列)和用于输出后续变元值的格式指定符。

输出格式指定符总是以%字符开头，其一般形式如下：

```
%[flags][width][.precision][size_flag]type
```

方括号中的项都是可选的，所以唯一的必选项是开头的%字符和指定转换类型的 type 指定符。

每个可选部分的作用如下：

- [flags]是 0 个或多个转换标志，控制输出的显示方式。可以使用的标志如表 D-1 所示。

表 D-1 转换标志

标 志	说 明
+	在输出中包含符号，如+和 - 。例如，%+d 会输出带正负符号的十进制整数
空格	符号用空格或 - 表示，即正值的前面有一个空格。当一列输出中包含正值和负值时，使用空格可以使输出排列整齐。例如，% d 会输出带符号的十进制整数，正值的前面有一个空格
-	输出在字段宽度中左对齐，如果需要，在右边添加空格。例如，% - 10d 会输出十进制整数，其字符宽度为 10 个字符，且左对齐。% - +10d 指定符会输出带正负符号的十进制整数，其字符宽度为十个字符，且左对齐

www.TopSage.com

(续表)

标　志	说　明
#	十六进制的输出值用 0X 或 0x 作为前缀(分别对应转换类型指定符 X 和 x)，八进制的输出值用 0 作为前缀
0	在右对齐的输出中，用 0 作为左边的填充字符。例如%012d 会输出十进制整数，其字符宽度为 12 个字符，且右对齐，左边用 0 填充

- [width]指定输出值的最小字符宽度。如果值在指定的最小宽度中放不下，就扩展指定的宽度。例如%15u 输出无符号的整数值，其字符宽度为 15 个字符，且右对齐，左边用空格填充。
- [.precision]指定浮点数输出中小数点后的位数。例如，%15.6f 输出一个浮点数，其最小字符宽度为 15 个字符，小数点后有 4 位数。
- [size_flag]是值的大小指定符，它改变了类型指定符的含义。大小指定符有 l(L 的小写)、L、ll(两个 L 的小写)和 h。可以使用的大小指定符取决于所使用的类型指定符，如表 D-2 所示。
- type 是一个字符，指定了应用于输出值的转换类型，如表 D-2 所示。

表 D-2　转换类型和大小指定符

转 换 类 型	说　明
d, i	值假定是 int 类型，输出一个十进制整数 使用 h 大小指定符(hd 或 hi)，变元就假定为 short 类型 使用 l 大小指定符(ld 或 li)，变元就假定为 long 类型 使用 ll 大小指定符(lld 或 lli)，变元就假定为 long long 类型
u	值假定是 unsigned int 类型，输出一个无符号的十进制整数 使用 h 大小指定符(hu)，变元就假定为 unsigned short 类型 使用 l 大小指定符(lu)，变元就假定为 unsigned long 类型 使用 ll 大小指定符(llu)，变元就假定为 unsigned long long 类型
o	值假定是 unsigned int 类型，输出一个无符号的八进制整数 使用 h 大小指定符(ho)，变元就假定为 unsigned short 类型 使用 l 大小指定符(lo)，变元就假定为 unsigned long 类型 使用 ll 大小指定符(llo)，变元就假定为 unsigned long long 类型
x 或 X	值假定是 unsigned int 类型，输出一个无符号的十六进制整数。如果使用小写形式的类型转换指定符，就使用 16 进制数 a~f，否则，就使用 16 进制数 A~F 使用 h 大小指定符(ho)，变元就假定为 unsigned short 类型 使用 l 大小指定符(lo)，变元就假定为 unsigned long 类型 使用 ll 大小指定符(llo)，变元就假定为 unsigned long long 类型
c	值假定是 char 类型，输出一个字符 使用 l 大小指定符(lc)，变元就假定为宽字符类型 wchar_t

www.TopSage.com

(续表)

转换类型	说明
e 或 E	值假定是 double 类型，输出一个用科学计数法表示的浮点数(带指数)。使用小写形式的类型转换 e 时，输出中的指数值跟在 e 的后面，否则跟在 E 的后面 使用 L 大小指定符(Le 或 LE)，变元就假定为 long double 类型
f 或 F	值假定是 double 类型，输出一个一般形式的浮点数(不带指数) 使用 L 大小指定符(Lf 或 LF)，变元就假定为 long double 类型
g 或 G	值假定是 double 类型，输出一个一般形式的浮点数(不带指数)，如果指数值大于精度(默认为 6)或小于 -4，输出就用科学计数法表示 使用 L 大小指定符(Lg 或 LG)，变元就假定为 long double 类型
s	变元假定是 char 类型的字符串，用终止字符\0 结尾，输出该字符串，直到终止字符为止，如果有精度指定符，就输出到精度指定符指定的字符为止。可选的精度指定符表示可以输出的最大字符数
S	与 printf()一起使用时，变元假定是 wchar_t 类型的字符串，用终止字符\0 结尾，输出该字符串，直到终止字符为止，如果有精度指定符，就输出到精度指定符指定的字符为止。可选的精度指定符表示可以输出的最大字符数
p	变元假定是一个指针，因为输出的是地址，所以它是一个十六进制值
n	变元假定是一个 int*类型的指针(指向 int)，输出中的字符数存储在由该变元指向的地址中 如果使用 h 指定符(hn)，变元就假定为 short*类型(指向 short) 如果使用 l 指定符(ln)，变元就假定为 long*类型(指向 long) 如果使用 ll 指定符(lln)，变元就假定为 long long*类型(指向 long long)
%	没有变元，输出%字符

D.2 输入格式指定符

scanf()函数从标准输入流 stdin(默认为键盘)中读取数据，sscanf()函数从内存的一个字符串中读取数据，fscanf()函数从文件中读取数据，从源中读取数据是由一个格式字符串控制的，这个格式字符串是上述函数的一个变元。这些输入函数的形式如下：

```
int scanf(const char* format_string, pArg1, ...);
int sscanf(const char* destination_string, const char* format_string, ...);
int fscanf(FILE* file_stream, const char* format_string, ...);
```

上述每个函数都返回读取的数据项的个数。参数列表末尾的省略号表示这里可以有 0 个或多个变元。格式字符串后面的变元必须是指针。将不是指针的变量作为这些输入函数的一个变元是一个常见错误。

控制输入过程的格式字符串可以包含空格、其他字符和格式指定符，且以%字符开头。

www.TopSage.com

格式字符串如果只有一个空白，函数就会忽略输入中的后续空白，将第一个非空白字符解释为下一个数据项的第一个字符。输入中的换行符后跟一个要读取的值时，例如，使用%c 格式指定符从键盘上读取一个字符时，输入的换行符、制表符或空格字符都会当做输入字符。在重复读取一个字符时尤为明显，而将按下回车键所产生的换行符留在缓冲区中。如果希望函数在这种情形下忽略空白，就可以在格式字符串的%c 前面加上至少一个空白字符，迫使函数跳过空白。

还可以在输入格式字符串中包含非空白字符，但该非空白字符不是格式指定符的一部分。这些非空白字符必须与输入中的字符完全匹配。否则输入操作就会结束。

数据项的格式指定符使用如下形式：

```
%[*][width][size_flag]type
```

方括号中的项是可选的。格式指定符中的必选部分是表示格式指定符开头的%字符和末尾的 type 转换类型指定符。可选部分的含义如下：

- [*]表示对应这个格式指定符的输入数据项应被读取，但不存储。例如，%*d 会读取一个整数值，然后删除它。
- [width]指定从输入值中读取的最大字符数。如果在达到这个字符数之前遇到了空白，就结束当前数据项的读取。例如，%2d 读取 2 个字符，作为一个整数值。宽度指定符可用于读取多个没有用空白字符分隔开的输入。把"%2d%2d%2d%2d"作为格式字符串，12131415 可以读取为值 12、13、14 和 15。
- [size_flag]修改指定符中 type 部分指定的输入类型。该指定符可以是 h、l(L 的小写)、ll(两个 L 的小写)和 L，使用哪个指定符取决于所使用的类型指定符，如表 D-3 所示。

表 D-3 输入转换类型指定符和修饰符

转换类型	说明
c	将一个字符读取为 char 类型 使用 l 修饰符(%lc)，单个字符就读取为 wchar_t 类型 也可以在 c 或 lc 指定符的前面加上一个十进制数字 m，从没有用空字符终止的字符串中读取 m 个连续的字符。例如%20c 会连续读取 20 个字符。对应的变元应是一个指向字符数组的指针，该数组有足够的元素来容纳所读取的字符
d	将连续的十进制数读取为 int 类型的值 使用 h 修饰符(%hd)，就读取连续的数字，并解释为 short 类型 使用 l 修饰符(%ld)，就读取连续的数字，并解释为 long 类型 使用 ll 修饰符(%lld)，就读取连续的数字，并解释为 long long 类型
u	将连续的十进制数读取为 unsigned int 类型的值 使用 h 修饰符(%hu)，就读取连续的数字，并解释为 unsigned short 类型 使用 l 修饰符(%lu)，就读取连续的数字，并解释为 unsigned long 类型 使用 ll 修饰符(%llu)，就读取连续的数字，并解释为 unsigned long long 类型

www.TopSage.com

(续表)

转换类型	说　明
o	将连续的八进制数读取为 unsigned int 类型的值 使用 h 修饰符(%ho)，就读取连续的八进制数字，并解释为 unsigned short 类型 使用 l 修饰符(%lo)，就读取连续的八进制数字，并解释为 unsigned long 类型 使用 ll 修饰符(%llo)，就读取连续的八进制数字，并解释为 unsigned long long 类型
x 或 X	将连续的十六进制数读取为 unsigned int 类型的值 使用 h 修饰符(%hx 或%hX)，就读取连续的十六进制数字，并解释为 unsigned short 类型 使用 l 修饰符(%lx 或%lX)，就读取连续的十六进制数字，并解释为 unsigned long 类型 使用 ll 修饰符(%llx 或%llX)，就读取连续的十六进制数字，并解释为 unsigned long long 类型
s	读取连续的字符，直到遇到空白为止，将所得的用终止符结束的字符串的地址存储在对应的变元中 如果使用 l 修饰符(%ls)，就读取字符，并存储为用终止符结束的宽字符串
n	不读取输入，但把到目前为止读取的字符数存储为一个整数，放在对应变元指定的地址中，变元的类型是 int*

注意，如果要读取包含空白字符的字符串，应使用%[set_of_characters]形式的指定符。使用这个指定符，只要把需要的字符放在方括号中，就可以从输入源中读取这些字符。因此，指定符%[abcdefghijklmnopqrstuvwxyz]可以读取任意序列的小写字母和空格，作为一个字符串。该指定符更有用的一种变体是在字符集合的前面加上字符^，例如，%[^set_of_characters]字符集合表示，遇到方括号中的字符会结束字符串输入。例如，指定符%[^,!]会读取一系列字符，遇到逗号或感叹号时，就结束字符串输入。

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

PROGRAMMER TO PROGRAMMER™

Beginning C: From Novice to Professional

Fourth Editon

尊敬的读者：

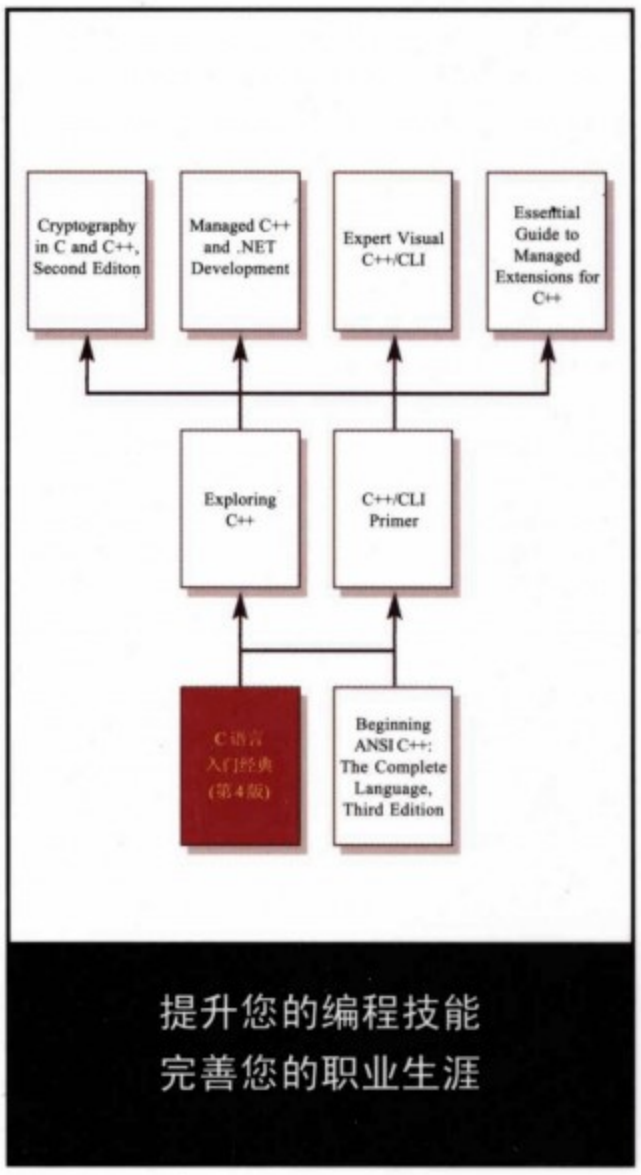
C 语言是一种非常优秀的程序设计的入门语言。它比其他大部分语言都简洁易学，所以在开始使用 C 语言编写真正的应用程序之前并不需要学习太多语法。C 语言也是功能相当强大的语言，很多操作系统都是用 C 语言编写的。大多数计算机环境都支持 C 语言，因此当你学会了 C 语言后，你将有能力在各种环境下进行程序设计。

本书的目标是使你在C语言程序设计方面由一位初学者成为一位称职的程序员。书中包含了 C 语言的全部基础知识，并将教会你如何进行程序设计。除了本书之外，你需要的东西只有一样，那就是一个得到广泛支持的免费的或者商业的标准 C 编译器，有了它即可开始编写实际的 C 程序了。

本书从第一个编程原理开始，使用各种程序示例解释 C 语言的所有元素，不要求你之前拥有任何编程知识。通过编写能够运行的完整 C 应用程序，在实际的环境中运用所学的知识，可帮助你提高自己的程序设计能力。你也可以在创建和执行自己的程序示例的过程中获得自信。

学习 C 语言也有一定的难度，但是我确信你会感到乐趣无穷，也会发现自己劳有所获。只要你有热情和信心读完本书，你将会掌握大量有价值的知识。从此，你将步入 C 程序设计的殿堂，开始自己的 C 程序开发之旅。

Ivor Horton



提升您的编程技能
完善您的职业生涯

a! Apress™
www.apress.com

Recommended Computer Book Categories	Programming Languages
	C

ISBN 978-7-302-17083-9
9 787302 170839 >
定价：69.80 元

www.TopSage.com