计算机科学丛书

Mc Graw Hill Education

C语言程序设计

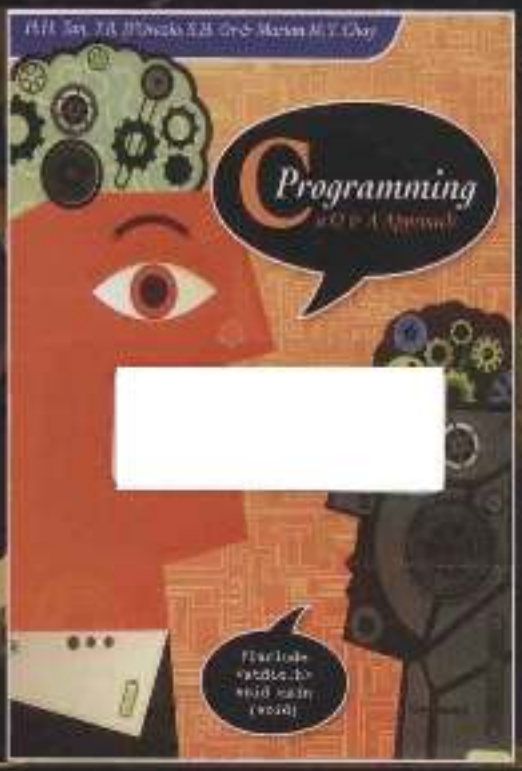
问题解答和实例解析方法

[美] H. H. 塔恩（H. H. Tan）
T. B. 多拉齐奥（T. B. D'Orazio）
何兆恒（S. H. Or）
玛丽安 M. Y. 周（Marian M. Y. Choy）
著

赵岩 译

C Programming

a Q & A Approach



机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

C语言程序设计

问题解答和实例解析方法

[美] H. H. 塔恩（H. H. Tan）
T. B. 多拉齐奥（T. B. D'Orazio）
柯兆恒（S. H. Or）
玛丽安 M. Y. 周（Marian M. Y. Choy）
著

赵岩 译

C Programming
a Q & A Approach

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

C语言程序设计：问题解答和实例解析方法 /（英）塔恩（H. H. Tan）等著；赵岩译．—北京：机械工业出版社，2016.7
（计算机科学丛书）
书名原文：C Programming: a Q & A Approach

ISBN 978-7-111-54334-3

I. C… II. ①塔… ②赵… III. C语言－程序设计 IV. TP312

中国版本图书馆CIP数据核字（2016）第167847号

本书版权登记号：图字：01-2012-4414

H. H. Tan, T. B. D'Orazio, S. H. Ur, Marian M. Y. Choy: C Programming: a Q & A Approach (ISBN 978-007-131116-8).

Copyright © 2012 by McGraw-Hill Education.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including without limitation photocopying, recording, taping, or any database, information or retrieval system, without the prior written permission of the publisher.

This authorized Chinese translation edition is jointly published by McGraw-Hill Education and China Machine Press. This edition is authorized for sale in the People's Republic of China only, excluding Hong Kong, Macao SAR and Taiwan.

Copyright © 2016 by McGraw-Hill Education and China Machine Press.

版权所有。未经出版人事先书面许可，对本出版物的任何部分不得以任何方式或途径复制或传播，包括但不限于复印、录制、录音、或通过任何数据库、信息或可检索的系统。

本授权中文简体字翻译版由麦格劳－希尔（亚洲）教育出版公司和机械工业出版社合作出版。此版本经授权仅限在中华人民共和国境内（不包括香港、澳门特别行政区和台湾）销售。

版权 © 2016 由麦格劳－希尔（亚洲）教育出版公司与机械工业出版社所有。

本书封面贴有 McGraw-Hill Education 公司防伪标签，无标签者不得销售。

本书以C语言作为工具，通过大量实例，详细介绍了基本程序设计的思想和技术。全书语言简练、图示有助于理解。同时兼顾读者经常关注的问题进行讲解，强调问题的分析和讨论，意在帮助读者认识程序设计的实质，理解从问题到程序的思考过程。

本书适合作为高等院校计算机及相关专业第一门程序设计课程的教材，也可供其他学习C程序设计的读者自学使用。

出版发行：机械工业出版社（北京市西城区百万庄大街22号 邮政编码100037）
责任编辑：和 静　　责任校对：殷 红
印　　刷：三河市宏图印务有限公司　　版　　次：2016年8月第1版第1次印刷
开　　本：185mm×260mm 1/16　　印　　张：25.5
书　　号：ISBN 978-7-111-54334-3　　定　　价：79.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066　　投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259　　读者信箱：hzjsj@hzbook.com

版权所有·侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所 韩光／邹晓东

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出，独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到"出版要为教育服务"。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brian W. Kernighan，Dennis Ritchie，Jim Gray，Alfred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以"计算机科学丛书"为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

"计算机科学丛书"的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，"计算机科学丛书"已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版"经典原版书库"作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：(010) 88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章教育

华章科技图书出版中心

译者序

C Programming: a Q & A Approach

C语言已经诞生了近半个世纪。在飞速发展的计算机领域，它完全称得上是一门古老的语言。虽然古老，但它和它的后继者C++以及Ojbective-C一起形成的C语言家族，依然是目前计算机行业最流行的开发语言之一，尤其在一些要求速度和效率的应用（如操作系统、编译器等）中，始终占据着不可动摇的统治地位。

作为计算机科学与技术相关专业的学生，全面系统地学习C语言是十分必要的。如果不能很好地掌握C语言，将在今后学习操作系统和编译原理时遇到更大的困难。但是不可否认，C语言由于在设计之初并不是一门教学语言，它的设计思想体现了很多底层的支持和对效率的考虑，如指针的概念、字符串的定义和操作、位操作等，这些内容对于刚刚接触计算机编程的学生，无论是学习还是掌握上，都具有一定的难度。

本书作者在长期的C语言教学中发现，作为一本实践性很强的学科，为了能让学生扎实地掌握C语言，必须要提供大量的编程实例，而不是在课堂上过多地讲解理论。通过实例学习编程是好，但是当学生完成这些实例的时候，他们不可避免地会提出很多的问题。所以本书又将所有实例中经常出现的问题和解答与实例一并给出。通过实例，再配合问答，学生可以快速地掌握C语言中的重要知识点。通过在大学里的教学实验发现，利用本书作为教材，学生反映学习的难度降低了，同时对C语言的掌握也更加扎实了。

我出版过一本C语言的书籍，也从事过两年的C语言教学工作，所以深知C语言对于计算机专业的重要性以及C语言教学的难度。当我看到这种基于实例和问答方式进行组织的C语言教材时，感觉到本书的翻译和出版将对C语言教学做出一些非常有益的探索和尝试，所以毫不犹豫地接受了翻译的任务。

我在深刻理解全书内容的基础上力求准确，对于发现的笔误和印刷错误进行了更正。在本书的翻译过程中，得到了出版社和家人的支持和帮助，出版社的编辑对译稿提出了很多中肯的意见和建议。在此一并向他们表示感谢！

限于水平，译文中疏漏和错误难免，敬请读者批评指正。如有任何建议，请发送邮件至zhaoyan.hrb@gmail.com。

译者

2016年4月

新生经常发现阅读计算机语言书很困难，书写本书的目的就是为了解决这个困难。如果能使学生深入到书本中，激发他们的兴趣，并使得他们思考C语言的用法和含义，那么我们就可以把学习的过程变得简单且有趣。为此，我们使用了Q&A的方式。在这个过程中，学生经常问的问题也会激发读者的思考。通过直接并清晰地回答这些问题，我们把读者的注意力集中到C编程中重要的概念上。

我们也观察到很多计算机语言书很少有图。因为可视化的图形在教学中非常有用，所以我们努力使图示既准确又容易理解。对于程序执行的操作，这些图有利于澄清概念，加强学生对概念的理解。特别地，我们用三维的图来描述循环和判断结构，从而让学生可以很快地掌握程序的流程。我们相信这些图是对标准流程图的加强。我们也意识到对于很多学生来说，指针是最困难的部分。指针图示建立在包含变量名字、类型、地址和值的表格的基础之上，并且表格出现在文本解释的前面。本书中，我们使用表格来说明一个内存单元的信息是如何与另一个内存单元的信息联系起来的。

很多书包含大量的代码，但是并没有给出充分的解释，大部分新生不能也不愿意在没有解释的情况下独立地理解哪怕是很简单的代码。本书在代码中引导学生了解操作以及生成代码的过程，目的就是使学生意识到哪些地方需要额外的想法，以及掌握正确细节的重要性。

这种独特的方式已经受到用过本书草稿的学生的热烈欢迎。本书也被推荐给其他学生并询问他们的意见。当和其他书比较时，学生会优先选择我们的书。我们相信你在教学和学习过程中也会发现本书的价值。

本书组织

第1章介绍编程基础，假设学生除了会使用计算机进行简单的文字处理以外，没有其他的计算机知识。第1章介绍了编程语言的概念，描述了硬件、信息在内存中的存储方法、计算机语言、编译器和软件工程。本章的目标是使学生了解计算机的工作方式和软件设计背后的概念。

第2章到第4章讨论了过程语言的基本概念、基本语法和控制结构，也介绍了C库函数和它们的用法。第5章介绍了用户定义函数，强调了模块化和可重用代码的概念，简单介绍了指针，并将它用在一个传递地址的函数中。本章最后，介绍了使用C语言用户定义函数的效果。

第6章关注数值型数组。

第7章描述了字符串和指针。由于字符串使用地址进行管理，这一章也非常适合用来解释如何利用指针修改内存。第8章覆盖了C语言中的结构及其在生成链表、堆栈、队列及二叉树中的用法。另外，本章也介绍了大型程序设计，因为工程问题通常都很大。使用C特性来处理大型程序是公司招聘学生开发商用软件产品时的一项重要考虑。

第9章是关于C++语言的介绍。因为已经介绍过C，所以更多地介绍C++中面向对象

编程的核心概念。我们用简单的术语介绍了类、封装和多态。这一章有很多演示，简单的语言和丰富的演示为学生提供了很多使用 C++ 基本特性的背景知识。

大部分章节被分为两个部分——课程和应用程序。课程部分学习语法、格式和基本构造，应用程序部分演示课程中教授的知识如何用于解决实际问题，演示了开发的流程，目标是使学生能遵循结构化方法来开发自己的程序。

特点

1）本书使用简单的问答方式，学生会发现这种方式比讲解的方式更加友好、更易于理解。这种方法式下，作者能够发现学生经常问的问题并能简洁地回答这些问题。

2）每一课都以一个样例程序开始：源代码并附有一些指示。学生根据指示观察代码的细节，从而了解 C 语言。下一步给出输出以及解释。解释环节给出一系列的问答以解释源代码做了什么。

3）应用程序部分演示了 C 语言如何用于解决工程和计算机科学中的问题。我们详细地解释了它们。例子主要涉及程序设计、软件工程、模块化和生成可重用代码。

4）给出大量的图来演示编程的概念。很多图都是独一无二的，能让学生快速地掌握概念。

5）在应用程序部分描述了四步结构化方法（引入了字符串和更复杂的数据结构后变成了五步结构化方法）。方法包括生成结构流程图和数据流程图。

6）应用程序部分也包括数值方法例子，这些例子用在把编程和数值方法结合起来的课程中。

7）课程部分包含注释代码，以帮助学生理解程序的细节和流程，使学生关注代码并把代码中的重要部分高亮显示。

8）我们意识到学生一般不会主动阅读多页代码，所以应用程序部分的每一段代码都只有 2 到 3 页，并有对应的解释。

9）指针的概念很难理解。为了让学生理解指针，可视化图形是非常有用的。盒子中一个指针指向另外一个盒子，这种图是不够的。使用表格和网格状的内存草图，可以降低指针的神秘性。我们发现阅读本书后，学生能够轻松地理解指针的概念。

10）应用程序部分后的练习可以用于实验课。教师可以让学生提前阅读特定的应用程序。上实验课时，可以指导学生做一些改动练习，后续的部分可以作为家庭作业。

11）新生通常会在调试的时候遇到困难，因为他们对这个过程很陌生。新生也会感到很沮丧，因为他们必须要调试自己的第一个程序。为此，我们在第 1 章介绍了一个详细的调试例子。初学者也发现调试循环是很困难的，本书中关注循环并演示了循环中值是如何变化的。学生将学习如何追踪循环并发现错误。另外，初学者经常会犯的错误也在本书相应的位置指出。

12）每课后面的判断题（有答案）可以让学生快速评价自己对基础知识的掌握程度。

13）每章后面的应用练习可以作为家庭作业。

14）本书中所有的程序都可以从 www.mheducation.asia/olc/cprogramming 获取[⊖]。学生可以修改并执行这些程序以理解它们是如何运行的。

⊖ 关于本书教辅资源，用书教师可登陆 www.mcgraw-hill.com.cn 申请。——编辑注

15）第9章是有关C++的介绍，不仅讨论了基础知识，阅读完本章后，学生还将学会使用面向对象编程的很多基本功能。

16）很多应用程序介绍了数值方法。

如何使用本书

对于学生

在第1章，你会理解什么可以保存在内存中，编译器如何工作，软件工程的步骤，最重要的是，编写自己的第一个C程序。其他章讨论了C语言编程。章节的课程都以一个简短的介绍开始，指导你关注源代码中一些重要的知识点。然后你可以阅读源代码和其中的注解。你甚至可以运行代码并观察程序的行为。完成这些后，确保你理解本课中主要的概念。之后阅读解释部分并完成判断题和简答题。如果做不好练习，应重新学习本课以消除疑问。

掌握了一章的课程后，开始学习应用程序部分，其目的是演示编写程序的一般过程以及C的实际用处。你会发现，当你写程序的时候会遇到很多应用程序中遇到的问题。在这一部分，关注学习方法论及理解每一个程序的逻辑。记住，编程中逻辑流是非常重要的。一个语句可能语法上正确但逻辑上却是错误的。掌握了每一个应用程序的来龙去脉，会让你在写自己程序的时候更加自信。不要只是读，要尝试每个程序，修改并试验它。它会帮助你掌握在课程中学习过的内容，进而解释程序的不同行为。利用这些知识完成教师布置的编程作业。

对于教师

作为一学期课程，本书的目的是为学生在后续课程中掌握高级编程奠定基础，例如C编程中带有C++的介绍，推荐你按照顺序讲完所有的内容。但是，按照不同的顺序讲解本书也是可以的。例如，课程3.2（单个字符数据）可以在第7章的课程前讲述。同时，如果需要的话，课程8.7（生成头文件）、课程8.8（使用多个源文件及存储类别）、类似函数的宏和条件包含可以在第5章介绍。

你也可以将第7、8和9章的部分课程延后，时间允许再讲解它们。例如，课程7.9（指针符号与数组符号）可以延后到指针的高级话题（第8章的附加材料，指向函数的指针和返回指针的函数，通过www.mheducation.asia/olc/cprogramming获取[⊖]）之前。

对于试图建立编程基础的一个学期课程，我们推荐你讲解到课程7.8，再加上课程7.10、8.1、8.2、8.3、8.4和8.5。

对于给学生一般的编程体验的短的课程，如果只讲解前6章，学生也会写出有价值且复杂的C程序。

本书提供了丰富的练习，课程部分后有判断题和简答题。学生应该独立完成这些练习。课程后的一些简单程序可以留作作业。一个星期的时间学生足可以完成一个程序。

应用程序部分后面的修改练习可以用于实验课。学生应该在实验之前学习相关的应用程

⊖ 关于本书教辅资源，读者可向麦格劳－希尔教育出版集团北京代表处申请，电话：010-57997600/59575582，电子邮件：instructorchina@mheducation.com。——编辑注

序。实验中，可以指导学生完成修改练习。一些练习比较容易，而另外一些很难，难的可以留作家庭作业。

大部分章末是应用练习。它们是本书中最有挑战性的练习，所以最适合留作家庭作业。根据不同的难度，需要 2 ~ 4 周的时间来完成它们。

另外，本书可以用作 ANSI C 的参考书，参考表格分散在本书正文中。

教师辅助材料

教师可在网站 www.mheducation.asia/olc/cprogramming[⊖]获得以下补充材料。

- 解答手册
- 教学课件
- 测试库
- 附加练习
- 附加阅读材料

⊖ 关于本书教辅资源，用书教师可登陆 www.mcgraw-hill.com.cn 申请。——编辑注

感谢 McGraw-Hill 出版社的 Eric Munson 和 Holly Stark。感谢他们对本书的兴趣、支持、鼓励以及有见地的建议。与他们一起工作非常愉快。感谢 Byron Gottfried（BEST 系列的编辑），感谢他的支持和宝贵的批评，以及 McGraw-Hill 的 Alisa Watson 和她的产品团队，使得本书版式优美。

我们有一些非常有想法的评论家。University of Houston 的 Betty Barr；University of Texas 的 Raymond Bell；Fayetteville State University 的 Tat W. Chan；Texas A&M University 的 Bart Childs；University of Minnesota 的 Chris J. Dovolis；Illinois State University 的 Janet Hartman；New Mexico State University 的 Elden W. Heiden；Purdue University 的 Elias Houstis；University of Minnesota 的 Joseph Konstan；University of Maryland 的 Jandelyn Plane；还有 WPI 的 Matthew Ward，他们都给出了非常有帮助的建议，感谢他们的贡献。

我们的计算机生涯开始于 University of Michigan 和 U. C. Berkeley，感谢 J. M. Duncan，John Lysmer 和 Raymond Canale 教授（我们第一年计算机课程的指导老师），他们鼓励我们解决一些复杂的问题，而由此带来的成就感和自信促使我们写这本书。

感谢 Suzanne Lacasse 和 Kaare Hoeg，他们分别为 Norwegian Geotechnical Institute 的主管和前主管。他们对我们的自信以及在开发地理应用程序时给予的资助，延展了计算机技术背景，为进一步开发程序所需要的技能打下了坚实的基础。感谢 Hui Xian Liu（Institute of Mechanical Engineering 中国哈尔滨地区的前主管），感谢他对 H.H. Tan 不断的支持、指导和鼓励。

最后要感谢我们的家庭。T.B. D'Orazio 的妻子 Elizabeth 虽然自己也很忙，但还是抽出时间鼓励和支持我们。正是有了她，辛苦的工作才有了乐趣。感谢 H.H. Tan 的妻子 Wei Huang，她一直是一个热心、含蓄、聪明的女人。感谢 H.H. Tan 的女儿 Sijing Tan，前四章的大部分图都是她绘制的。

H.H. Tan
T.B. D'Orazio

感谢 McGraw-Hill 的 Gerald Bok 和 Chris Cheung。Gerald 是这个项目的主要推动人，他处理了大量琐碎的事物，使得我们可以专注于本书的内容。Chris 负责沟通协作，也提供了很多帮助。也感谢 McGraw-Hill 的 Maureen Tan 的大力支持。

S.H. Or 要感谢香港中文大学的 John Lui 教授在担任计算机科学和工程系主任时给予的支持。同时也感谢自己的妻子 Pui-yee Lau，感谢她对孩子的细心照顾，使得自己可以专心于本书的编写。

Marian Choy 要感谢 Tim Lambert（在 University of New South Wales 她最受欢迎的讲师）让自己看到了编程的乐趣和创造性。也要感谢 Horace Ip 教授在香港城市大学给了她第一个教学的职位。同时也感谢香港大学工学院的 W.C. Chew 和 George L. Tham 教授，他们提供了免费的平台使自己可以探索和实验各种教学方式。最后，感谢所有学习过她的编程课的学生，感谢她的家庭、小组成员、朋友和同事，感谢他们一如既往的支持，以及对自己“固执地”追求高水平教学质量的包容。

S.H. Or

Marian M.Y. Choy

出版者的话
译者序
前言
致谢

第 1 章 编程基础 …… 1
课程 1.1 编程语言 …… 1
1.1.1 汇编语言 …… 1
1.1.2 高级语言 …… 1
课程 1.2 软件工程 …… 3
1.2.1 自顶向下模块化设计 …… 3
课程 1.3 C 语言、ANSI C 和 C 编译器 …… 5
1.3.1 C 和 ANSI C …… 5
1.3.2 程序开发 …… 5
课程 1.4 利用位表示字符、符号、整型数、实型数、地址和指令 …… 7
1.4.1 字符和符号 …… 7
1.4.2 整型数 …… 8
1.4.3 实型数 …… 9
1.4.4 十六进制和八进制表示 …… 9
课程 1.5 关于本书及如何充分利用本书 …… 10
1.5.1 课程 …… 10
1.5.2 应用程序 …… 11
课程 1.6 基本结构 …… 12
课程 1.7 格式化输出 …… 16
课程 1.8 其他转义字符 …… 18
课程 1.9 基本调试 …… 20
本章回顾 …… 22
第 2 章 变量、算术表达式和输入输出 …… 24
课程 2.1 变量：命名、声明、赋值和打印值 …… 24
课程 2.2 算术运算符和表达式 …… 30
课程 2.3 从键盘输入数据 …… 33
课程 2.4 常量宏及打印变量值的进一步讨论 …… 37
课程 2.5 混合类型的运算、复合赋值、运算符优先级和类型转换 …… 43
本章回顾 …… 52
第 3 章 C 语言基础：数学函数和字符文件输入输出 …… 53
课程 3.1 数学库函数 …… 53
课程 3.2 单个字符数据 …… 57
课程 3.3 从文件读入数据 …… 67
课程 3.4 输出到文件 …… 72
应用程序 3.1 面积计算——复合运算符和程序开发 …… 74
应用练习 …… 77
本章回顾 …… 79
第 4 章 初级决策和循环 …… 80
课程 4.1 if 控制结构和关系表达式 …… 80
课程 4.2 简单 if-else 控制结构 …… 84
课程 4.3 嵌套 if-else 控制结构 …… 87
课程 4.4 逻辑表达式 …… 90
课程 4.5 逻辑运算符的优先级 …… 92
课程 4.6 switch 和 if-else-if 控制结构 …… 96
课程 4.7 while 循环（1）…… 102
课程 4.8 while 循环（2）…… 105
课程 4.9 do-while 循环 …… 107
课程 4.10 简单 for 循环 …… 109
课程 4.11 嵌套 for 循环 …… 112
应用程序 4.1 梁交叉——if-else 控制结构 …… 116
应用程序 4.2 面积计算——for 循环 …… 118
应用程序 4.3 温度单位转换——for 循环 …… 120
应用程序 4.4 温度单位转换——循环和 if-else 控制结构 …… 121
应用程序 4.5 仿真 …… 123

应用程序 4.6 工程经济学——嵌套 for 循环……124
应用程序 4.7 解二次方程——if-else 控制结构（数值方法例子）……126
应用练习……128
本章回顾……131
第 5 章 函数……132
课程 5.1 不返回值的函数……134
课程 5.2 返回一个值的函数……142
课程 5.3 作用域和传值给函数的机制……146
课程 5.4 返回多个值的函数……151
课程 5.5 从函数返回多个值的机制——地址和指针变量……153
应用程序 5.1 使用带有复杂循环的函数处理网格（逻辑例子）……159
应用程序 5.2 模块化程序设计：平行四边形面积和平行六面体体积（数值方法例子）……164
应用练习……167
本章回顾……172
第 6 章 数值数组……173
课程 6.1 一维数组和打印数组元素介绍……174
课程 6.2 数组初始化……178
课程 6.3 基本数组输入输出……181
课程 6.4 多维数组……185
课程 6.5 函数和数组……192
课程 6.6 冒泡排序和最大交换排序……197
应用程序 6.1 将 16 个 1 位加法器组成 1 个 16 位加法器……202
应用程序 6.2 浪高的平均值和中位数（数值方法例子）……205
应用程序 6.3 矩阵－向量乘法（数值方法例子）……209
应用程序 6.4 搜索和文件压缩……212
应用练习……215
本章回顾……219
第 7 章 字符串和指针……220
课程 7.1 声明、初始化和输出字符串及理解内存布局……222
课程 7.2 确定字符串和字符信息及使用 printf……229
课程 7.3 二维字符数组……234
课程 7.4 从键盘和文件读入字符串……238
课程 7.5 指针变量与数组变量……245
课程 7.6 在声明中初始化……251
课程 7.7 将字符串传入用户自定义函数……256
课程 7.8 标准字符串函数……261
课程 7.9 指针符号与数组符号……272
课程 7.10 动态内存分配……279
应用程序 7.1 管流速、检查输入数据及模块化设计……285
应用程序 7.2 地震轶事报告分析、字符串操作和动态内存分配……294
应用练习……305
本章回顾……308
第 8 章 结构和大型程序设计……309
课程 8.1 结构……310
课程 8.2 结构成员……316
课程 8.3 指向结构的指针……318
课程 8.4 结构和函数……321
课程 8.5 结构数组……322
课程 8.6 带一个递归调用的函数……324
课程 8.7 生成头文件……329
课程 8.8 使用多个源文件及存储类别……331
课程 8.9 位操作……334
应用程序 8.1 排序——快速排序算法……342
本章回顾……350

第 9 章　C++ 介绍 ……351
课程 9.1　C++ 注释和基本输入输出流 ……351
课程 9.2　格式操纵符及格式化输出 ……354
课程 9.3　函数重载 ……357
课程 9.4　默认函数参数 ……360
课程 9.5　内联函数和变量声明的位置 ……363
课程 9.6　C++ 类和只有数据成员的对象 ……365
课程 9.7　带有数据和函数成员的类及封装 ……369
课程 9.8　构造函数和析构函数 ……375
课程 9.9　继承 ……379
应用程序 9.1　电子电路 ……385
应用练习 ……389
附录 A　ASCII 码 ……391
附录 B　ASCII 码描述 ……392

编程基础

本章目标

结束本章的学习后，你将可以：

- 编写一个完整的 C 程序。
- 使用 C 语句给出格式化的输出。

用计算机编程并不是一个简单的任务，但是那种伴随着自己程序成功运行的喜悦感也许是对程序员最大的奖励。在本章中你将学习如何编写一个简单的 C 程序，在屏幕上输出一些单词、数字和键盘符号。本章所介绍的示例程序本身的实际用途并不是很强，但是它们对学习 C 程序设计是很有帮助的。

在介绍第一个程序前，我们先简单了解一下编程的过程。

课程 1.1　编程语言

为什么要学习编程？这个问题需要详尽的阐述和回答。假设我们想控制计算机，第一件要做的事就是与这个可编程的机器进行交流。机器语言是计算机能够理解并运行的唯一语言，也就是说，它是唯一能够控制计算机的语言。机器语言由二进制的指令构成，即一系列 0 和 1 组成的代码，并只针对某种特定的处理器，例如 Intel 的 Pentium 系列处理器。计算机执行的每一步都需要用这些指令书写。因为机器语言非常繁琐，所以大部分程序都是先用其他的语言编写，然后被翻译成机器代码。

1.1.1　汇编语言

汇编语言被认为是比机器语言高一个层次的语言。在汇编语言中，所有的指令和对应的机器语言需要一一映射。例如，对某个特定的处理器来说，机器语言中的加法指令是“10010101”，而在汇编语言中，对应的指令代码是“ADD”。汇编语言中的指令不再是二进制码，而是英文单词，这种代码样式对人类来说更可读，这是汇编语言相对机器语言来说一个主要的优点。英文单词可以被语言翻译程序翻译成对应的机器语言。我们可以想象把英文单词直接转换成对应的二进制码并不是一件很困难的事。

虽然汇编语言比机器语言更好用，但是使用汇编语言的最大问题是它要求程序员对硬件有非常全面的了解。另外，为了完成诸如一些简单的计算和输出信息的任务，程序员需要书写大量的汇编语言代码。

1.1.2　高级语言

高级语言是为了进一步简化程序员需要书写的命令而开发的。例如，在机器语言或者汇编语言中，为了把两个数加到一起，我们需要在计算机的内部采取一系列的步骤，把信息从一个内存单元转移到另外一个内存单元中。而对人类来说，一个更简单的写法就是“a+b”。当这种语言被翻译后，用来把两个数加起来的一系列机器语言指令已经被写出并保存在内存

中。与机器语言不同，高级语言允许程序员不用关心运行程序的机器的内部设计情况。

高级语言需要满足一些规则使得自己能够被正确地翻译成机器语言。这些语言被设计用来简化可以解决特定问题的编程。例如，有些语言被开发用来编写能够解决科学计算问题的程序，而有些用来处理商业事务。

编程语言（见表1-1）可以被分成4种类型：

1）过程（或指令）语言

2）函数语言

3）声明语言

4）面向对象语言

目前，还不需要详细说明这四种语言之间的区别。但是需要指出的是，C语言是一个面向过程的语言。就像这个名字所暗示的，面向过程的语言需要程序员安排一系列的过程来解决问题。正如在本章的应用程序部分你将看到的：针对特定的问题，首先关注于开发一些过程，然后编写一个能够执行这些过程的程序。

在表1-1中还需要注意C++是一个面向对象的语言。抛开区别不说，C是C++的一个子集：这也就是说，本书中你学到的任何关于C语言的知识，在将来学习C++的时候都会用得到。

表1-1 高级语言总结

语言名称	语言类型	开发时间	语言名称	语言类型	开发时间
Fortran	过程语言	20世纪50年代中期	Smalltalk	面向对象语言	20世纪70年代中期
Basic	过程语言	20世纪60年代中期	Pascal	过程语言	20世纪70年代早期
Lisp	函数语言	20世纪50年代末期	C	过程语言	20世纪70年代中期
Prolog	声明语言	20世纪70年代早期	C++	面向对象语言	20世纪80年代中期
Ada	过程语言	20世纪70年代中期			

语言翻译器

语言翻译器是一类程序，负责把汇编语言或高级语言编写的指令转换成对应的机器语言（也叫目标码）。C语言编写的计算机程序需要用语言翻译器编译成机器语言。有三类语言翻译器：汇编器、编译器、解释器。

汇编器负责把用汇编语言写成的程序转换成目标码。因为汇编语言和机器语言非常类似，所以汇编器比解释器和编译器更简单。

编译器负责把整个用高级语言书写的程序转换成机器指令。在转换的过程中，高级语言代码中的错误可以被编译器检测到。编译器会查找那些违反了语言制定规则的问题，诸如不正确的标点符号或者冲突的声明；但是编译器并不负责检测出所有的错误，例如逻辑错误。通常情况下，人们的程序员通常会错误地假设经过编译器正确编译过的程序就一定会给出正确的结果。一个编译器编译时并没有给出任何错误信息的程序，就像一个没有任何语法错误的英文文章。它满足特定的规则，但是并不一定就有意义。

你将会一直用C语言编译器把C语言编写的指令翻译成机器语言代码。编译器会查找代码中的语法错误和不一致的地方，并将这些结果描述给你。在这种情况下，程序需要经历一个叫做调试（debugging）的过程，也就是说，修改程序中的错误以满足编译器的要求。然后需要运行并检验程序以确保它完成任务。如果程序并没有正确地运行，你需要修改、增加或删除一些C语言指令，重新编译整个程序并运行它。你需要一直重复调试的过程，直到

从程序得到令人满意的结果。

市场上有很多C语言的编译器，你的大学可能会提供一个编译器让你使用。你的老师会教会你如何使用这个编译器。你也可以自己从软件商店购买一个C语言编译器，或者使用互联网上的一些免费的编译器。这些编译器附带的指令会告诉你如何把C代码翻译成机器码以及如何正确使用编译器的其他特性。

翻译器也用于高级语言。与编译器不同，翻译器依次翻译并执行指令。换句话说，翻译器取出高级语言书写的一个指令，将它转换成机器语言并运行它。然后翻译器处理下一个指令，并一直重复这个过程直到所有的指令被执行完毕。

概念回顾

1）与计算机交流的语言可以分为：

- 高级语言——这种编程语言更加面向人，程序员不需要了解计算机内部的结构就可以使用它进行编程。
- 低级语言——编程过程以及把程序在计算机间进行移植更加困难。

2）C语言可以被认为是一种高级语言。

3）为了能在计算机上运行C程序，需要一个编译器把C语言程序转换成可执行的机器语言程序。

课程 1.2 软件工程

软件工程是一个用来描述软件开发过程的术语。它表明软件并不是被任意开发出来的。与此相反，软件的特性必须被仔细考虑、计划、构造和检测。这个术语反映了构造软件和构造硬件之间的类比性，也反映了建造软件和建造任何其他传统工程之间的类比性。

软件开发是一个精心计划的过程，软件的功能首先被定义。先要做出一个初始的计划，并且咨询所有软件开发相关方（用户、拥有者、程序员及其他）的建议。然后不断地修改，每个单独组件的设计也需要涉及。当软件进行整体的组装和测试时，需要计划和实施修改。软件的文档也要仔细地维护以便用户可以高效地使用以及方便日后的修改。同时，需要预测每一个开发步骤的时间和花费使得工程可按计划完成。

软件设计和开发的很多步骤在图1-1中给出。注意图中的“循环”，循环就是重复的步骤，也就是说，测试和修改的步骤。这些是非常重要的，因为软件会持续地经历修改和测试过程。

本文中，你会学习如何计划一个初始的程序结构，书写C语言的代码，遵循模块设计方案，测试程序，修改并书写文档。这些并没有包含所有的软件开发步骤，但是它们足够你计划和构造有用的C程序。

1.2.1 自顶向下模块化设计

自顶向下的设计方法从定义软件的主要功能开始，然后开发次要功能。每一个组成部分都更简单，并且可以被单独设计。所有的部分需要正确地组合在一起。每一个单独设计的组成部分都可以被认为是一个模块。

在C语言中，模块被称为函数。这些函数主要分为两种类型：库函数和用户定义函数。库函数是一些已经被开发出来并包含在编译器中的模块。它们执行一些标准数学的操作，诸

如计算角度正弦值的 sin 函数。这些函数可以使得用户不再需要重复开发那些标准的操作。

程序员可以自行定制用户自定义的函数。它们执行 C 编译器自带库中没有的那些功能。例如，在课程注册列表程序中，一个用户自定义的程序用来比较申请加入该课程的学生列表和该课程的可用名额。

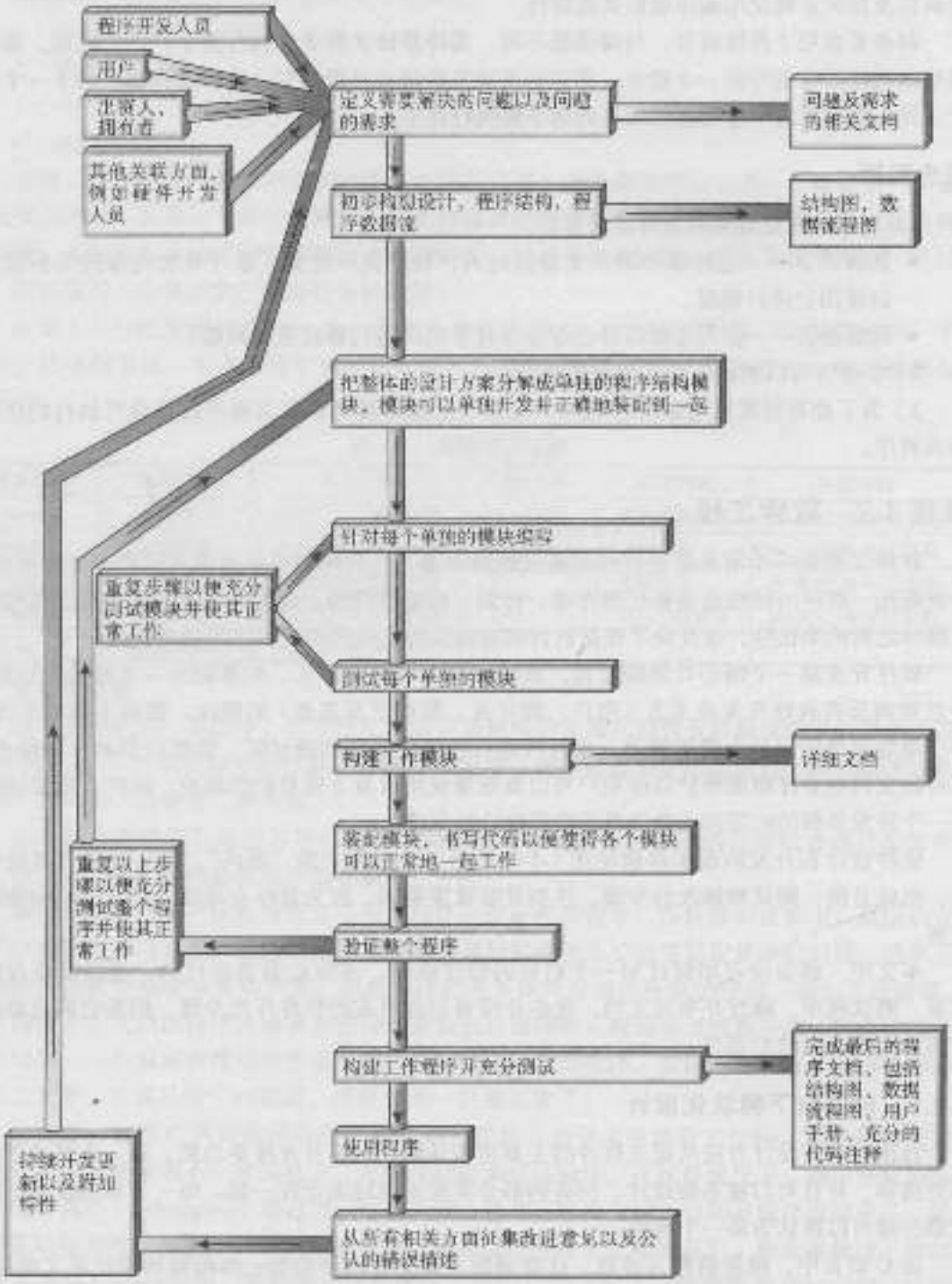


图 1-1 程序开发包含的步骤

数据从库以及用户自定义函数中传入和传出。程序员的责任在于保证数据流动的正确性及有效性。在本书中，你会学习如何编写用户定义的函数以及如何利用库函数。

课程 1.3 C 语言、ANSI C 和 C 编译器

1.3.1 C 和 ANSI C

C 语言是 Dennis Ritchie 于 20 世纪 70 年代早期在贝尔实验室（现在是朗讯科技的一部分）中发明的。这种语言最初被开发用来代替生成系统软件的汇编语言。它同时也是一种能够控制低端操作的高级语言。因为语言的结构中使用了很多的库函数，所以它具有很高的移植性和机器无关性。

为了保持 C 语言的标准，在 1989 年，美国国家标准学会（American National Standards Institute，ANSI）（X3J11 委员会）核准了一个与机器无关的 C 语言版本——ANSI C（American National Standard X3.159-1989）。标准的核准使得 C 语言编译器可以正确地处理所有用 ANSI C 标准编写的程序。这样，一个用 ANSI C 标准编写的程序可以正确地在任何有兼容 ANSI C 编译器的机器上编译并执行。本书中经常使用 ANSI C，在使用这个术语的任何上下文中，术语 ISO C 也是同样有效的。

在 1990 年（在 1994 年修改），制定了一个国际 C 语言标准（ISO/IEC 9899：1990，修订 1：1994），ISO C。除了很小的编辑类型的改变，ISO C 和 ANSI C 很相似。

20 世纪 90 年代后期，C 标准在 1999 年被进一步增强和修订，称为 C99。C99 中引入了一些新的特性，同时 C 语言的这个现代的版本也更加严格。随后，开发出新的在某种程度上实现了这些特性的软件工具，以便使得 C99 与以前的 C 语言标准（如 ANSI C）兼容。

为了应对现代计算发展的趋势，正在进行另外一次修订。在 2009 公布了一个标准的工作草稿，但是在修订成最后的版本之前还需要一些时间。

本文中，我们遵循 ANSI C 标准，因为它是目前最流行且被支持最多的标准。任何与 ANSI C 标准不同的地方将被标识出来。

1.3.2 程序开发

程序开发的主要目的在于生成一个可执行文件。可执行文件是一系列的机器语言指令（二进制格式），并且可以被用户运行。当你买一个商用软件的时候，你购买的是可以安装在电脑硬盘上的可执行文件（还有其他一些东西）。一旦安装完毕，你就可以在需要的时候运行它。通过阅读本书，你可以编写程序，然后由这些程序生成可执行文件。一旦生成了可执行文件，你就可以把它们保存在硬盘上，并在需要的时候运行，就像商用的软件。

本书的主要内容是演示如何生成源代码，通常来说，这也是程序开发流程最困难的部分。把源代码转换成可执行文件通常比编写源代码要简单很多，这是因为 C 语言的编译器完成了把语句转换成机器语言的大部分工作。把源代码转换成可执行文件的步骤如图 1-2 所示。

通常，现代的 C 语言开发环境要执行以下四种不同的操作：

1）允许用户编辑文本以便产生源代码，也就是用 C 语言编写程序。

2）预处理这些源代码（后面会详细介绍）。

3）编译源代码并指出任何可能的错误。

4）把库（后续章节中介绍）中的目标码和编译器产生的目标码链接起来。

注意，编译器并不仅仅编译（把C代码转换成机器语言或目标码）。这四步在图1-2中用方块来标识。

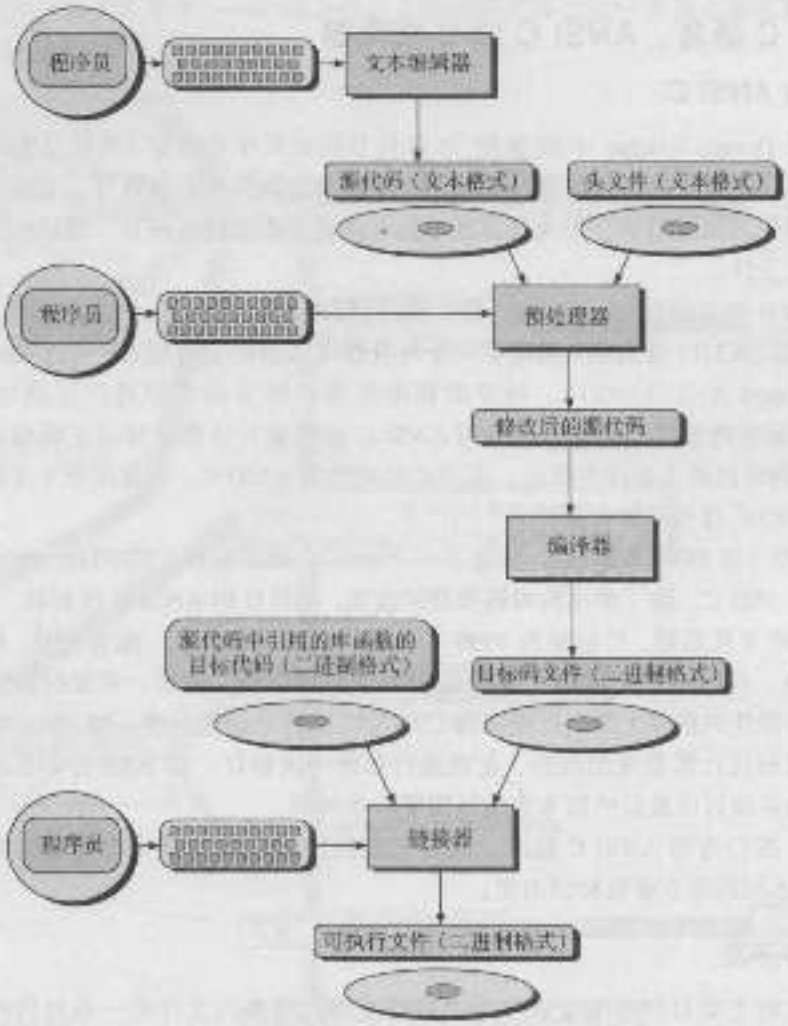


图1-2 程序员生成可执行文件需要的步骤。并不是所有的编译器和本图完全一致，但是大部分编译器遵循这一通用形式

为了产生可执行文件，需要完成以下4个步骤：

1）程序员需要使用一个文本编辑器（C编程环境的一部分）来产生源代码文件。程序员可以简单地从键盘输入文本，这种文本根据本书介绍的规则来书写。你将需要很多时间来学习如何正确地编写C代码。

2）一旦代码被正确地编写，程序员需要指示编译器开始编译过程。编译过程以预处理开始。预处理器把编译器内建的源代码和程序员产生的源代码结合起来。它同时根据程序给出的指令来修改源代码。

3）编译器的作用就是把经过预处理器修改过的源代码翻译成机器语言指令。注意，这个过程要求程序的语法都是正确的。编译器会生成目标码，并且把它保存在另外的中间文件中。

4）程序员调用C编译器中的链接器（有的编译器会自动执行这一过程）完成机器语言指令，并保证所有的指令被正确地装配。大部分情况下，程序员书写的源代码会调用其他内建的非程序员生成的代码——C库，来执行某些操作。链接器决定哪些函数被调用并且把那些函数与源代码翻译生成的目标代码组合起来。一旦链接器完成它的任务，编译过程结束，生成可执行程序。这个文件被保存在硬盘上。

市场上的编译器有不同的特性。有些有内建的文本编译器，简化了编写源代码和生成可执行文件的步骤。有些会通过简单的键盘敲击命令或单击鼠标来自动地执行所有描述过的编译器操作。有些编译器需要更多的操作，有时你需要手工完成那些操作。在本课堂上，你可以使用某个编译器，或者为你的个人电脑购买一个编译器。由于编译器种类很多，而本书篇幅有限，所以不会介绍所有的编译器。课堂指导老师会教你如何正确地使用C语言编译器。

如果你为了阅读本书而购买了一个编译器，请查阅其用户手册获得相关的信息，以便能够运行下面的任务：

- 命名源代码文件。
- 生成源代码文件（可能通过内建的文本编辑器）。
- 预处理、编译并链接文件。
- 执行程序（也可称为运行程序）。
- 重定向输入和输出（改变标准的输入和输出设备，默认情况下，标准输入设备是键盘，标准输出设备是屏幕）。

概念回顾

编写一个程序需要以下三步：

1）写C语言程序（源代码）。

2）把程序编译成可执行文件。

3）运行可执行文件以查看结果。

程序的生成可以再细分为以下步骤：

1）预处理源代码。

2）编译成目标代码。

3）链接目标代码以生成可执行文件。

课程1.4 利用位表示字符、符号、整型数、实型数、地址和指令

1.4.1 字符和符号

在数据传递的早期，广泛使用一种基于一系列划和点的表示字符和符号的二进制系统——摩斯代码（Morse Code）。计算机的到来使得我们可以开发出更加复杂的编码方式，但是它们都服务于相同的功能。与划和点不同，现在计算机的符号习惯上用0和1。ASCII（American standard code for Information Interchange）和Unicode是表示字符和符号的最常用的两种编码方式。ASCII通常用在个人计算机上，而Unicode是ASCII编码的国际版本，它包含了很多国家的字符集。如果不考虑那些国际字符集，Unicode中基本符号的编码值和ASCII中的编码值是一致的。涉及编程，我们将只使用英文字符集来发布命令，所以本书只讨论ASCII编码。

表1-2给出了大写字母A到F以及其他一些字符的编码值（128个字符的代码值的完整列表在附录中给出）。注意每个字符和代码用8位来表示，我们称之为一个字节。

表1-2 代码实例

字符	ASCII代码值	字符	ASCII代码值
A	01000001	F	01000110
B	01000010	?	00111111
C	01000011	>	01001110
D	01000100	{	01001101
E	01000101		

1.4.2 整型数

整型数于内存中被二进制数值系统所表示。二进制系统只用1和0来表示一个数，这和利用位来表示信息的系统非常符合。二进制系统与日常生活中的十进制系统不同。如果以10为基数，每个位代表10的幂，最右面的一位代表10^0（也就是1），从右向左每一个连续的位代表递增的10的幂。例如：五位数78326可以被翻译成$(7\times10^4)+(8\times10^3)+(3\times10^2)+(2\times10^1)+(6\times10^0)$。

在二进制中，每一位代表2的幂。最右面的一位代表2^0（也就是1），从右向左每一个连续的位代表递增的2的幂。如下图所示：

二进制位	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
位的十进制值	128	64	32	16	8	4	2	1

例：8位以2为基数的数字10010110对应的十进制数值是多少？

解：利用上表中对应的位，可以计算出数值。

二进制位	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
位的十进制值	128	64	32	16	8	4	2	1
二进制数字	1	0	0	1	0	1	1	0
二进制数字乘以十进制值	128	0	0	16	0	4	2	0

以10为基数的值可以通过把最后一行的数值相加得到，这个数值为128+16+4+2＝150。

例：利用上面介绍的二进制数的表示方法，计算出8位二进制数所能表示的最大和最小的数值。

解：最大的数值为所有的二进制位都是1，而最小的为所有的位都是0。由以上得知，最大的值为128+64+32+16+8+4+2+1=255（也就是2^8-1）。而最小的数就是0。注意，这种方案下不能表示负数。

假设想用8位来表示相等数目的正数和负数，我们该怎么做呢？一个简单的方法就是让其中一位成为符号位，其他的位如下所示：

符号位 2^6 2^5 2^4 2^3 2^2 2^1 2^0

我们规定如果符号位为1，那么这就是一个负数。（反之，符号位为零代表这是一个正数）。因此所能表示的范围为：

最大＝01111111＝－（64+32+16+8+4+2+1）＝127

最小＝11111111＝－（64+32+16+8+4+2+1）＝－127

通过利用符号位，把能表示的数值从全部是正数转换成了一半是正数，一半是负数。这种符号位的表示方法并不是计算机内部表示整型数的方法（计算机内部表示正数的方法叫做2的补码，本书不做详细的描述）。但是符号位的方案也演示了很多用来表示整型数的原则：

- 位数的个数直接决定了整型数所能表示的范围。
- 符号位占据一位，所以有无符号位的表示方法有不同的范围。

在C语言中会看到，可以指定用来表示整型数的位数的多少（这样就可以控制要使用的整型数的大小，以及是否在位中使用符号位。）

1.4.3 实型数

实型数和整型数、字符和符号一样，也是以二进制方式保存的。用来表示实型数的二进制代码和用来表示整型数的二进制代码是不一样的。

用来把实型数的二进制转换成十进制的方法，与整型数的转换方法很类似。小数点右面的每一位都是以下面的顺序表示的2的幂。

二进制位	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	等
位的十进制值	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	

下面这个例子演示了转换的方法。

例：8位二进制数100.10110的十进制表示是什么？

解：利用上表，可以计算如下：

二进制位	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
位的十进制值	4	2	1	0.5	0.25	0.125	0.0625	0.03125
二进制数字	1	0	0	1	0	1	1	0
二进制数字乘以十进制值	4	0	1	0.5	0	0.125	0.0625	0

答案就是把表中最后一行相加到一起，也就是4+1+0.5+0.125+0.0625= 5.6875。

为了能够有效地使用计算机的内存，实型数用科学计数法来保存。在十进制中，例如15230000可以被表示成1.52310^7。1.523叫做有效位，10叫做底，而7叫做指数。

在二进制中，你也可以使用科学计数法

$$101.01100 = 1.0101100 \times 2^2$$

$$-0.0001011101 = -1.011101 \times 2^{-4}$$

我们不会详细讨论实型数是如何被保存的，但是从这些简单的例子可以看出为了保存实型数必须要：

- 既保存有效位也保存指数。
- 有效位和指数的符号也需要内存中的空间。

用来表示整型数的位的个数限制了所能表示整型数的大小。而在实型数中，位数的个数不仅限制了所能保存的数的大小，也限制了精度（小数点后面的位数）。这是因为，为了保存一个实型数，所能使用的位数需要在保存有效位和保存指数上做一定比例的分配。

1.4.4 十六进制和八进制表示

现在，手工计算必须用纸和笔或其他方式才能写出位模式。但是对计算机来说，通常可以很简单地处理对人类来说很难应对的大量的1和0。例如，635 163 077的32位二进制表示为00100101110110111101000111000101。如果每天和这么一长串的字符或数字打交道，你会很快厌倦并犯错。同时二进制和十进制数之间缺乏直接的转换方法，也使得十进制数很难和二进制的位一起工作。作为一种解决方法，十六进制（16为基数）或八进制（8为基数）被用来表示一长串二进制位的缩写。表1-3给出了十进制、十六进制和八进制的位模式。

表 1-3 各种记法的比较

十进制	十六进制	八进制	位模式	十进制	十六进制	八进制	位模式
0	0	0	0000	8	8		1000
1	1	1	0001	9	9		1001
2	2	2	0010	10	A		1010
3	3	3	0011	11	B		1011
4	4	4	0100	12	C		1100
5	5	5	0101	13	D		1101
6	6	6	0110	14	E		1110
7	7	7	0111	15	F		1111

注意因为必须有一个符号作为位的占位符，所以大写的字符 A 到 F 用来表示十六进制中 10 到 15（十进制）的占位符。这个表可以生成八进制和十六进制的表示方法。八进制（0～7）每组由 3 个二进制位组成（忽略表 1-3 中位模式表示中最左边的 0）。十六进制（0～15）每组由 4 个二进制位组成。

例如，位 10100110011001100011011 的八进制表示方法为 51463073，如下所示。

二进制位	101	001	100	110	011	000	111	011
位的八进制值	5	1	4	6	3	0	7	3

而 00100101110110111101000111000101 的十六进制表示为 25DBD1C5，如下所示。

二进制位	0010	0101	1101	1011	1101	0001	1110	0101
位的十六进制值	2	5	D	B	D	1	C	5

你可以看出无论是八进制和十六进制，都比那一长串的二进制表示方法更好用。

课程 1.5 关于本书及如何充分利用本书

通常本书的每一章部分为两部分：课程部分和应用程序部分。每一章以一系列课程开始，这些课程会使你熟悉 C 语言的特性。课程结束后，应用程序部分会演示实际的应用以及一些特殊的编程技巧，这些技巧对你开发自己的程序非常有用。

1.5.1 课程

为了从本书课程中获益更多，你应该遵守循序渐进的过程：

1）阅读每一课程的介绍，在检查课程中的源代码时要遵循教授的内容。查看所有实例的源代码和教授的内容中提到的输出。回答源代码中的所有的问题，这是非常重要的一步。你应该尝试从源代码和程序的输出中尽可能多地收集信息，换句话说，把本书的这一部分当成一个要解决的问题。如果你能自己理解每一个程序要做什么以及为什么要这么做，那么你会从这个过程中学到很多。本书的这一部分经过特意的设计，以便给出足够的信息来引导你独立地推导出 C 代码的含义。

2）阅读课程的基本解释部分，以便更好地理解源代码要做什么，以及搞清楚在上一步骤中令你迷惑的地方。在这一步中，你会经常参考书中的源代码，尽量多地阅读并理解这些源代码是非常重要的。完成本书的学习后，你会“流利”地使用 C 语言而不再害怕面对老师

或庭主给你的很多页的 C 语言代码。

3）扩展解释部分被加到了基础解释部分后面，它介绍了更多的高级概念。扩展解释的内容主要帮助那些在尝试编写一个全新的程序前希望更好地理解高级概念的同学。不过如果你有一些冒险精神，可以先不看这一部分的内容，开始编写你的代码。如果在写代码的过程中遇到了问题，再回头看这一部分的内容。

4）在课程的结尾完成练习，确保学会本课程中的概念，并为下一课程做准备。

1.5.2 应用程序

当读应用程序时，你应该重点关注开发这些程序的方法和原则。为了演示应用程序的开发，我们使用了多步开发的过程。推荐你在开发自己的程序时也遵循这个流程。

在第三章的开始，这个过程描述如下：

1）收集相关公式。

2）对实例的问题，做一个手工计算。

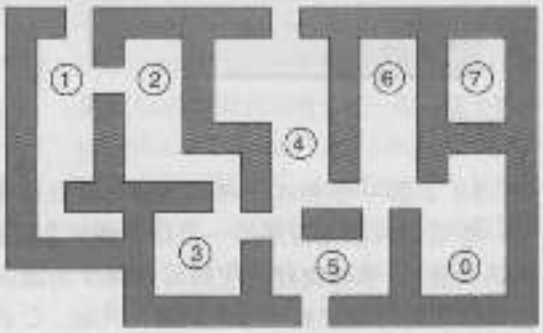
3）利用公式及手算的模式写一个算法（有时也叫伪码）。推荐你写一个不太正式的算法，可以只是一个程序要做什么的一行接一行的描述。应该用英文书写。随着水平的提高，你的准备工作会越来越接近最后要书写的源代码。

4）根据算法书写真实的源代码。

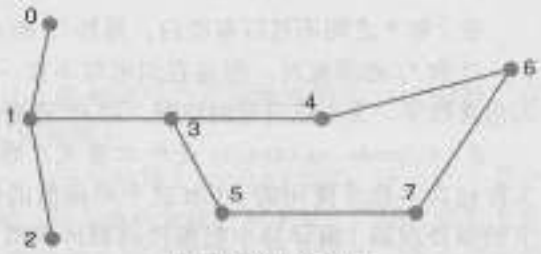
随着程序越来越复杂，将增加一些诸如开发结构图和数据流程图的步骤，以及规划程序中要使用的数据结构的步骤。虽然推荐你在开发自己的程序时遵守这些流程，不过我们也意识到，当你对编程越来越熟练时，可以跳过其中的一些步骤，开发出最适合自己的方法；或者使用老师推荐的方法。使用哪种方法开发并不重要，重要的是你要遵循一个标准的流程，而不是非常随意地去开发程序。

练习

1. 列举出二战以后开发出来的至少 5 种计算机语言。
2. 我们使用这个练习是因为你可能对计算机的游戏非常熟悉。很多计算机的游戏在屏幕上使用迷宫。故事很简单，一个英雄进入一个迷宫追寻、打败魔怪并救出人质。迷宫如图 1-3a 所示。物理形式的迷宫可以被建模成如图 1-3b 所示。技巧在于迷宫的每一个连接点都被标上一个数字，并被表示成一个节点。两个节点的通道被表示成图中的一个边。你的任务是：
   a. 给定图 1-3a 所表示的迷宫，画出迷宫的网抽象。
   b. 给定图 1-3a 所表示的迷宫，作一个表以建模迷宫。
   c. 画一个至少有十个模块的结构图使得你的游戏更加有趣。



a) 迷宫游戏



b) 含有节点和边的图

图 1-3

课程 1.6 基本结构

主题

- 写一个简单但完整的 C 程序。
- 利用 printf 函数在屏幕上显示输出。
- 简单 C 程序的结构。
- 书写 C 程序的基本原则。

本程序演示了 C 程序的基本的结构，当你执行这个程序时，语句“This is C!”显示并保留在你的屏幕上，直到后续的指令删除它或者把它向上滚动。

在你阅读解释部分之前，要仔细地检查源程序和输出，本节希望你先自己尝试解释那些源代码的含义。随着学习的深入，你会发现自己可以在查阅解释之前翻译出其中很多的源代码。

源代码

```
#include <stdio.h>      /* 这是一条包含指令*/
void main (void)
{
    /* 本程序的目的是在屏幕上打印出一个字符*/

    printf ("This is C!");
}/* 需要一个配套的右括号来结束这个程序 */
```

输出

程序被编译和运行后，在屏幕上显示如下：

```
This is C!
```

解释

1）在第一行中以 /* 开始的语句是什么含义？这一行代表的是 C 程序的注释语句。注释是一些笔记以描述程序的一部分要做什么以及是怎么做的（或者使得别人更加容易理解你的代码内容）。书写良好的注释可以减少犯错误的可能，因为修改程序的程序员可以通过阅读注释以便更好地理解程序是如何工作的。C 语言注释的语法是

```
/*    任何文本、数字、字符    */
```

在 / 和 * 之间不可以有空白。另外 /* 和 */ 必须成对出现。/* 和 */ 叫做注释分隔符。

/* 和 */ 必须配对，但是它们可以不在一行，因此注释语句可以有多行文本。一个多行的包含数字、文本或符号的注释，以 /* 开始，以 */ 结束。参见本课程代码中第二个注释。

2）#include <stdio.h> 是什么含义？现在不会讨论更多的细节，只说明 stdio.h 这个文件包含本程序使用的 printf 这个库函数的信息。指令 #include <stdio.h> 告诉 C 编译器中的预处理器（编译器中把源代码翻译成机器语言之前执行操作的那一部分）把 stdio.h 这个文件和书中所示的源代码组合在一起。这个操作如图 1-4 所示。因为源代码行 #include <stdio.h> 使得预处理开始工作，所以这个语句也叫做预处理指令。程序中最多只使用几个

预处理指令，因此预处理指令并不是编程过程中的主要部分。你可以在本书的其他程序中认出预处理指令，因为它们都是以#开始，并且出现在程序头几行。预处理指令在执行代码翻译之前，也执行其他一些有用的操作。

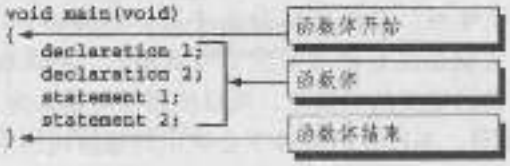
3）void main(void) 是什么含义？这一段给出了生成的程序体的名字。本例中，函数的名字叫做main，代表着这是要被执行的主程序。

main这个名字是强制要求的，程序员不能自己改动它。换句话说，即使你想利用自己的程序在屏幕上打印地址，也不能把程序命名成printmyaddress或编写命令void printfmyaddress(void)。

我们先不考虑本行中void的含义。本书会一直使用void main(void)到第5章。然后在第5章详细讨论这一话题。目前只需要记住这一行的样式，并在所有的程序中使用它就可以了。

4）程序中括号的含义是什么？函数名后面的代码就是函数体，它有以下特性：

- 以左花括号开始。
- 以右花括号结束。
- 括号用来包含一个代码块。我们经常使用一对花括号来形成一个代码块。本例中，花括号包围的代码块就是函数体。
- 函数体中包含C语言的声明（本书后面会介绍）和C语言的语句。这个典型的C函数如下所示：



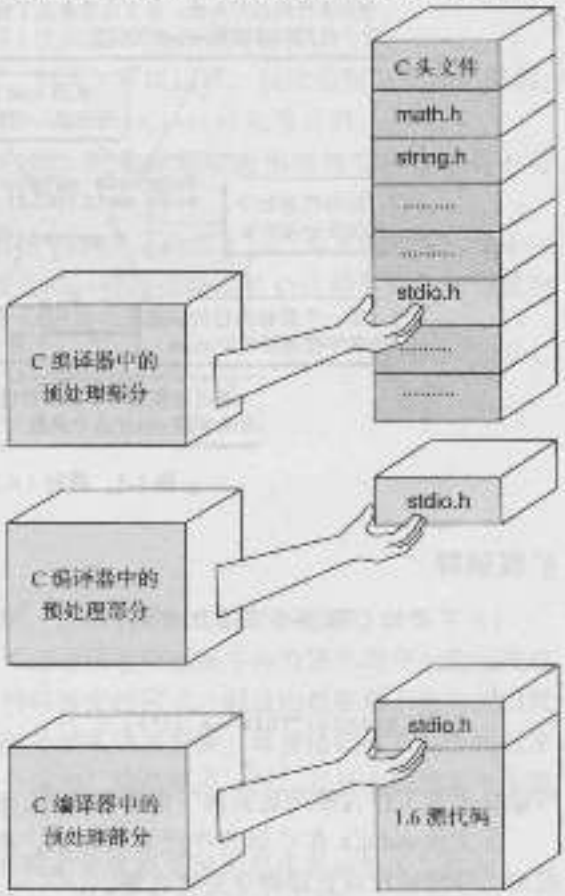


图1-4 C语言编译器中的预处理器把文件stdio.h和课程1.6中的源代码组合在一起。这个操作通过预处理指令#include <stdio.h>来完成。执行本操作后，代码可以成功地被翻译成机器语言，并且代码有足够的信息来正确使用源代码中出现的printf函数

5）printf("This is C!"); 是什么含义？这个语句调用C库函数中的printf函数。它代表我们要执行一些操作——本例中是把信息写到屏幕上。

6）为什么printf("This is C!"); 语句以一个分号结尾？printf("This is C!"); 是一个C语言语句。C语言语句必须以一个分号结尾，分号也叫做语句结束符。C语言语句末尾的分号就像我们每句话后面的句号一样。一个典型的C语言程序有很多语句，每个语句都以一个分号结束。

本课程代码的每一个组成部分的总结见图1-5。

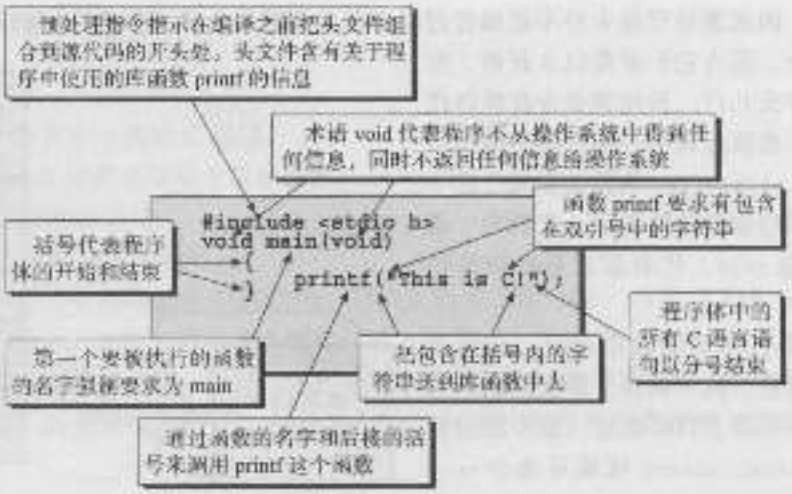


图 1-5 课程 1.6 程序的组成部分

扩展解释

1）下面的 C 程序会正常工作吗？

```
main()
{
        printf("This is C!");
}
```

可能会正常工作，但是这依赖于你的编译器。这个程序可能工作的原因如下：

①文件 stdio.h 在 C 语言中经常被用到，所以编译器自动地把这一文件和你的代码组合起来，即使你没有直接命令要这么做。

②即使你没有写出 void main(void)，也就是说 void 被忽略了，C 也会把默认的值当成 void。默认这个词在计算领域非常常用。一个默认的值就是当你不指定它的值的时候所使用的值。一个默认的类型就是当你不指定它的类型的时候所使用的类型，这里不再详述。当你忽略 void 时，C 会赋给一个默认的类型，所以程序会正常工作。但是我们不推荐你使用这种方式书写程序。为了更加清晰，推荐你使用本课程开始时所使用的格式。当然也有其他一些人使用刚刚介绍过的这种忽略 void 的形式，不过你要明白这种格式中使用了默认值。

2）C 语言中能同时使用大写和小写的字母吗？C 语言区分大小写。这样 printf 和 PRINTF、Printf 或者 PrintF 都是不一样的。也就是说 C 语言是大小写敏感的。本课程中，main 和 printf 都必须用小写字母。当你命名自己定义的函数时，可以使用任何你认为正确的大小写格式。C 传统上主要是用小写字母编写。不过有的条件下会使用其他格式的字母，这些条件在需要的时候，我们会介绍。

3）我们能写一个嵌套的注释语句吗？不可以，注释语句不能嵌套（也就是说在注释语句中写一个注释语句）。例如

```
/*/* This is an illegal comment because it is */ nested */
```

4）C 语言中可以使用哪些空白字符？C 语言中包含了一些词（Token）。一个 C 语言的词是编译器不能再进行分解的最小的单元。一个词可以是一个函数名如 main，或者是本章后面讨论的 C 的关键字。所有 C 语言的单词必须连续书写，例如表达式

```
void main(void)
```

是不合法的，因为单词 main 中的字符 a 和字符 i 之间是不允许有空格的。

在标识符之间，空白字符（空格、制表符、回车）可以出现，但这是可选而非必需的。例如，`void main(void)` 和 `void main (void)` 或 `void main (void)` 是等价的。

通常，在标识符之间加入空白字符是可以的，但是标识符内部增加空白字符是不可以的。

5）是否需要把程序写到不同的行里？当你在标识符之间按下一个回车的时候，这个时候产生的那个空白字符对 C 编译器来说是不可见的，所以你可以把 C 代码写到任何行或列中。C 编译器允许你把程序 L1_1.C 写成一行的方式

```
#include <stdio.h>void main(void){printf("This is C!");}
```

或者写成下面的方式

```
#include<stdio.h>void
main(  void          ) {
     printf                          ,
(  "This is C!"  )   ;     }
```

上面这种书写的方式会让你的程序难以理解，所以不要写出这样的程序。

在你的程序中并不需要加入很多的空白，不过你的老师或编辑会希望你遵守一些标准或其他可接受的样式。我们的示例程序演示了一种可接受的样式。但是出版限制不允许我们死板地遵守某一个样式。缩进和空格对一个程序的外观来说很重要，即使他们并不会影响程序的性能。为了让程序更加可读，每一行只写一个语句。括号要占一行，并且在程序指令中需要的地方加上空行。

6）为什么程序的外观如此重要？程序的外观非常重要是因为程序要持续经历修改。一个整洁且组织良好的程序是非常容易理解并修改的。那些遵守某些外观和组织样式的程序比起那些不遵守样式的程序，出错的机会将大大减少。

概念回顾

1）一个完整的 C 程序格式如下：

```
#include <stdio.h>
void main (void) {
   declaration statements;
   executable statements;
}
```

2）注释格式如下：

```
/*   任何文本、数字、字符   */
```

它们用作说明程序的文档。

3）printf 会把任何由双引号括起来的内容打印到屏幕上。

练习

1. 判断真假：

a. 通常，C 语言语句是大小写敏感的。

b. 默认情况下，C 语言语句是位置敏感的。

c. C 语言语句必须以句号结尾。

d. C 程序的主函数的名字必须是 main。

e. main() {} 是一个完整且正确的 C 程序。

f. printf 和 main 都是 C 语言的单词。

2. 在下面的语句中查找错误：

a. void main(void);

b. printf("Do we need parentheses here?");

c. printf("where do we need a blank space?");

d. print("Is any thing wrong here?")

3. 输入、编译并运行下面程序：

```
main(        )
{ printf ("There is no class tomorrow!"  )  ;   }
```

改正任何你发现的错误。

4. 输入、编译并运行下面程序：

```
ma  in() PRINTF *, {'What is wrong?' }
```

改正任何你发现的错误。

5. 修改本课程的程序使得它在屏幕上打印你的姓名和地址。

答案

1. a. 真　b. 假　c. 假　d. 假　e. 真　f. 真

2. a. void main(void)

b. 没错误

c. 没错误

d. printf("Is any thing wrong here?");

课程 1.7　格式化输出

主题

- 格式化输出
- 回车

在 printf 函数双引号包含的文本字符串中，可以插入一些不会打印出来的符号，这些符号会被 printf 翻译成一些控制光标（也叫插入点）在屏幕上移动的指令。

源代码

```
行      代码
01      #include <stdio.h>
02      void main(void)
03      {
04          printf("Welcome to");
05          printf("London!");
06          printf("\nHow do we\njump\n\ntwo lines?\n");
07          printf("\n");
08          printf("It will rain\ntomorrow\n");
09      }
```

输出

```
Welcome toLondon!
How do we
jump

two lines?

It will rain
tomorrow
```

解释

换行操作可以通过在 printf 函数中的文本字符串加入 \n 符号来轻松地实现。符号 \n 包含两个字符，\（反斜杠，不要和斜杠/混淆）和 n，并且两个字符之间没有空格。在 C 语言中，\n 是很多转义字符中的一个，通常称之为换行。C 编译器把字符串文中的转移序列当成一个字符（而不是两个）。

假设我们想把

```
Welcome to
London!
```

以两行的方式显示在屏幕上，可以使用两个 C 语言语句

```
printf("Welcome to");
printf("London!");
```

来完成这个目的吗？不可以，这是因为 printf 函数在每次调用的时候并不会自动转到下一行进行输出。这样，第一个 printf 函数的输出会和第二个 printf 函数的输出连接到一起，从而在屏幕上 "welcome to" 和 "London" 会输出到同一行。

概念回顾

1）printf 函数中的转义字符会指定特殊的操作，转义字符都以反斜杠开始。

2）回车符 \n 会在屏幕的输出上执行换行的操作。

练习

1. 判断真假：

a. 语句 printf("\n\n\n"); 会生成三个空行。

b. 语句 printf("\nnn"); 会生成三个空行。

c. 语句 printf("\n\n\n"); 会生成三个空行。

d. 语句 printf("\n \n \n"); 会生成三个空行。

e. 语句 printf("\ n\ n\ n"); 会生成三个空行。

f. 转义字符代表两个字符。

2. 下面的语句中有些存在错误，请找出这些错误。

a. printf("I \n Love \n California \n");

b. printf("I \ n Love \ n California \ n");

c. printf("I n Love n New York \ n");

3. 编译并运行下面的程序：

```
main()
{printf("I \n Love \n California \n");
 printf("I \ n Love \ n California \ n");
```

```
 printf("I n Love n California n");
}
```

修改这个程序，使得它的输出有意义。

4. 写一段程序，在屏幕上输出一个10行的故事。
5. 修改练习。修改本课程的程序使得它完成下面的任务：
   a. 把所有的文本输出成两行。
   b. 打印下面的输出：

```
Welcome to London! How do we
jump
two lines?

It will rain   tomorrow
```

   c. 用两个 printf 语句打印下面的输出：

```
Welcome to London! How do we jump

two lines? It will rain

tomorrow
```

答案

1. a. 假　b. 假　c. 真　d. 真　e. 假　f. 假
2. a. 没错误
   b. 没错误，但是不会输出回车，字符 n 也会被输出
   c. 没错误，但是不会输出回车，字符 n 也会被输出

课程 1.8　其他转义字符

主题

- 产生声音。
- 连接 C 字符串文本。

\n 转义字符只是能用在字符串文本中的一个。所有的转义字符都是以一个反斜杠开始。下面程序中的转义序列可以发出嘟（beep）的一声，并且把光标移到一行中的不同位置。

源代码

```
#include <stdio.h>
void main (void)
{
    printf ("Listen to the beep now. \a");
    printf ("\nWhere is the 't' in cat\b?\n\n");

    printf ("I earned $50 \r Where is the money?\n");
    printf ("The rabbit jumps \t\t two tabs.\n\n");

    printf ("Welcome to\
            New York!\n\n");

    printf ("From "              "Russia \
            with "             "Love.\n");
    printf ("Print 3 double quotes      -\" \" \" \n");
}
```

输出

```
Listen to the beep now.
Where is the 't' in cat?

Where is the money?
The rabbit jumps                    two tabs.

Welcome to New York!

From Russia with Love.
Print 3 double quotes    -" " "
```

解释

1）如何发出嘟的一声？第4行

```
printf("Listen to the beep now. \a");
```

中的转义字符 \a 在打印出 "Listen to the beep now" 后，会发出嘟的一声。

2）如何执行回退的操作？可以使用回退转义字符 \b。第5行的语句

```
printf("\nWhere is the 't' in cat\b ?\n");
```

会把光标从单词 cat 的字母 t 后面回退一格，这样我们就看不到字母 t 了。

3）如何把光标移到本行的开头？第6行

```
printf("I earned $50 \r Where is the money?\n");
```

中的转义字符 \r 不会显示任何 \r 前面的字符。转义字符 \r 代表一个回车，并且把光标移到当前行的开始位置。

4）如何连接 C 语言的文本字符串？这里我们演示两种方法，第一种方法是在行的末尾使用反斜杠（第8行）：

```
printf("Welcome to \
```

它与下面的一行紧邻

```
New York!\n\n");
```

这个反斜杠代表文本字符串还没有结束，并且延伸到下一行。因为 C 编译器会忽略语句后面的所有空白符，所以下一行会正好连接到上一行的末尾处。第二种方法是把没有完成的字符串文本分别包含在双引号中（行10）：

```
printf("From " "Russia \
with " "love.\n");
```

等同于

```
printf("From Russia  with love.\n");
```

5）如何使用 printf 函数显示出双引号？双引号是一种特殊的字符，如果单独在文本字符串中使用会被错误地翻译。必须在双引号紧邻的左面放一个反斜杠以便显示它们。反斜杠和双引号之间不应该有任何空格。

printf 函数和其他的输出函数可以在格式化文本中使用转义字符，表1-4列出了 C 语言中使用的完整的转义字符。目前，你还不需要理解它们的全部含义。

表 1-4 转义字符

转义字符	含义	结果
\0	空字符	终止字符串
\a	警报/响铃	产生一个声音或可视的警告
\b	回退	把当前位置（对于控制台来说，指的是当前的光标位置）在当前行回退一个空格
\f	换页	把当前位置移到下一个逻辑页开始的位置
\n	换行	移到下一行的起始位置
\r	回车	移到当前行的起始位置
\t	水平制表	移到当前行中下一个水平制表符的位置
\v	垂直制表	移到下一个垂直制表符的起始位置
\0ddd	八进制常数	八进制的整型常数，ddd 代表一系列 0～7 之间的数
\xddd	十六进制常数	十六进制的整型常数，ddd 代表一列十进制的数，字母 a～f 或者 A～F 分别代表 10 到 15
\\	反斜杠	显示反斜杠
\'	单引号	显示单引号
\"	双引号	显示双引号
\%	百分号	显示百分号
\?	问号	防止对一些类似的三字母序列错误的翻译。例如，三字母序列 ??= 会翻译成字符 #，但是 \?\?= 会显示 ??=

概念回顾

1）转义字符包含一个反斜杠以及后面的一个字符、符号或者一个数字的组合。每一个字符代表一个特殊的含义，或者一个特殊的动作。

2）转义字符 \a 当在 printf 使用的时候会产生嘟的一声。

3）转义字符 \b 会使光标回退一格。

4）转义字符 \r 会使光标回退到第一列，也就是说，本行的开始位置。

5）转义字符 \ 会把当前行的右面和下一行连接到一起。

6）打印双引号使用 \"。

练习

1. 判断真假：

a. 语句 `printf("ABC\a\a");` 会打印出 ABC 并产生两声嘟嘟声。

b. 语句 `printf("ABC\b\b");` 会只打印出 ABC。

c. 语句 `printf("ABC\r\r");` 会只打印出 A。

d. 语句 `printf("ABC\t\t");` 会打印出 ABC。

答案

1. a. 真　　b. 假　　c. 假　　d. 真

课程 1.9 基本调试

主题

目前为止，我们已经看到了一个完整程序的基本方面。所有演示的程序都被仔细地检查过，所以都是没有错误的。但是，通常情况下并不是这样的。我们并不熟悉 C 语言，当自

已独立编写一个C程序的时候，错误会不时出现。在程序中，寻找那些令程序运行失败的错误和缺陷的过程叫做调试。在C语言中，有三种类型的错误：语法错误、运行时错误和逻辑错误。

语法错误是那些违反了C语言语法规则的错误。它们通常由打印错误，或者缺乏C语言语句格式的知识所引起。这类错误会在C编译器编译这个程序的时候被诊断出来。当你的编译器编译程序的时候会指出语法错误，同时编译器并不会把程序翻译成机器语言。在编译器能成功翻译你的程序之前，你必须修改这类错误。因此，当程序有语法错误的时候，它不会产生任何输出，即使语法错误非常小，或者位于程序的最后一行，也不会产生任何输出。

下面的程序语句中包含语法错误，你能找到它们吗？

```
01 #include <stdio.h>
02 void main (void)
03 {
04     printf ("Listen to the beep now. \a"),
05     printf ("\nWhere is the 't' in cat\b?\n\n")
    ;
```

通过仔细检查，你可以发现程序中的第4行，末尾的分号被错误地打印成了逗号。这个错误将禁止编译器成功编译你的程序。事实上，借助于编译器技术的进步，大部分这种错误会被编译器准确地定位出来。但你还是需要非常小心地避免这类错误，这是因为有的时候编译器生成的诊断信息对一个新手程序员来说，并不是非常容易理解。

运行时错误，也叫做语义错误或者逻辑的错误，是违反了程序运行时的规则所引起的。编译器在编译时不会识别出这类错误。但是程序运行的时候，计算机会显示一个信息，告诉你有些地方出错了，而且（通常）程序会结束运行。如果一个运行时错误在程序的末尾出现，你可能会得到程序的结果。计算机给出的信息会帮助你在源代码中定位出错误的源头。下面的例子演示了一个运行时错误。运行这个程序时，如果用户输入零，就会发生运行时错误。

```
printf ("Please input the value of x : ");
scanf("%d", &x);
printf ("Result is %d \n", 1 / x);
```

逻辑错误的识别和修改是最困难的，因为计算机并不会像指出程序有语法错误和运行时错误那样，指出你的程序有逻辑错误。完全依靠你自己去识别出这类错误，你需要去看程序的输出来判断程序是否有这类错误。换句话说，你的程序貌似已经成功地运行完毕，并给出了一个有意义的结果。但是这个结果是完全错误的。你必须能认识到它是错误的，并修改其中的源代码。（注意，随着你深入地学习本书，有的时候你会发现问题只是出现在输入数据中。当你在程序中花费很多时间去查找bug，最后却证明了程序是正确的，而输入数据是错误的。）

扩展解释

1）如何在你的程序中减少bug的数量？为了减少程序中bug的数量，你需要养成一个好习惯，并建立一个预防bug的策略。这包括：

- 写整洁的代码。
- 在自然的地方加入空行。
- 左括号和右括号独占一行。
- 加入正确的注释。

遵循以上步骤，会让你逐步避免这些 bug。本质上，你应该以一种结构化和组织化的方式工作。请记住计算机不会原谅或忽略任何错误。通过学习本书，你会注意到一些常见的错误。注意并关注到这些问题，你就会避免 bug。

2）如何调试程序？如果你的程序没有运行，不要失望，保持自信并平静。失望会让你不理智，并且把一些正确的东西改成错误的。调试程序和修理一台不工作的汽车是一样的。当你的车不启动，通常你会绕着它走一圈，然后去看看发动机罩下有没有什么线头松动，检查电池和其他一些东西，并不会去拆解你的发动机。不幸的是，很多没有经验的程序员，在程序不工作的时候，去拆解他们的代码并随意改动代码。

要全局考察整个程序，并提醒自己是不是打错了字符。例如，把 `priatf` 输入成了 `print`？是否正确地使用了 C 语言的标点符号？`void main(void)` 打成了 `void main(void);`？所有的括号是否成对出现？

换句话说，先去查看那些简单明显的错误。就像解决车的问题一样，识别出问题，并用它作为线索去发现源头。例如，如果车的挡风玻璃的雨刷不工作，你没必要去看车后面有哪些部件出了故障，而只需要去看雨刷电机和它的连接线。对一个程序来说，如果它没有计算正确，你只需去查看负责计算的那部分代码。图 1-6 总结了调试的过程。

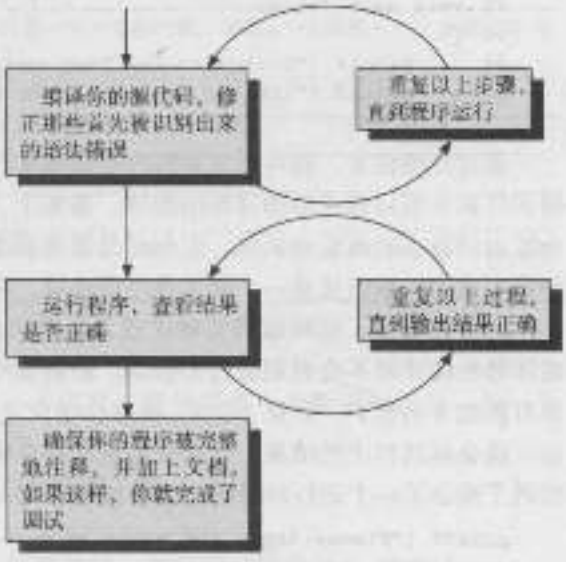


图 1-6 调试简洁描述

不要依赖 C 语言的编译器来定位错误。例如，在程序的开头，如果你只是简单地忘记写上一个注释语句的右界定符 */。这种情况下，编译器会把所有剩下的语句当成一个没有完成的注释语句。这个小错误会产生 30 个错误信息。不要被吓到了，记住典型的 C 语言编译器在发现语法错误方面，并不是那么准确和精密。你可能仅仅通过输入一个字符，就减少掉 100 个错误信息。

争取独立地解决问题，在试图得到帮助的时候要有选择性。通常，不要依赖别人来为你调试程序。你不应该去打扰那些正在试图使他们的程序正常工作的同学。自己先尝试解决问题，只有你努力地试图去解决问题但失败的时候，你才需要帮助。虽然这有点痛苦，但是通过只依赖自己的书本和计算机来解决问题，你才能收获最大并成为一个优秀的程序员。

本章回顾

一个 C 语言程序的通用格式如下所示：

```
# preprocessing directives

void main(void)
{
  statement 1;
  statement 2;
  statement 3;
  statement 4;
```

```
...
}
```

本章讲述了上面给出的C程序的完整结构。预处理指令出现在程序的开始，它会告诉编译器那些头文件需要被包含进来，以便能够正确地使用诸如printf的库函数。接下来是主函数声明，一对花括号{}用来界定所有的C语言语句。在这些语句中，变量声明语句必须位于语句的开头。典型的赋值语句和算术语句用来执行必要的流程。整个程序由右花括号结束，并且把控制权返还给操作系统。调试是一个程序员的重要工作。你需要有足够的C语言的知识，并且要坚持不懈地解决程序的各种问题。

第 2 章
C Programming: a Q & A Approach

变量、算术表达式和输入输出

本章目标

结束本章的学习后，你将可以：

- 声明 C 程序中使用的变量。
- 从键盘读取用户的输入。
- 利用 printf 语句控制输出的格式。
- 构建复杂的数学表达式。

为了让计算机程序有用，它必须有一些函数用来执行计算，并且能对用户的输入及时地反馈。本章要学习如何处理变量和执行数学计算。

课程 2.1　变量：命名、声明、赋值和打印值

主题

- 命名变量
- 声明数据类型
- 使用赋值语句
- 显示变量的值
- 基本的赋值语句

变量对所有的 C 程序都非常重要。在代数中你可能已经学习过变量，现在你会发现在 C 语言中几乎可以以同样的方式使用变量。

例如，假设给出下面的信息后，计算 10 000 个不同三角形的面积。

1）三个边长。

2）三个角的角度值。

为了写出计算面积的代数公式，首先需要命名变量，你可能选择下面的变量名：

1）三个边长：a，b，c

2）三个角度：α，β，γ

也可以这样命名变量：

1）三个边长：l_1，l_2，l_3

2）三个角度：θ_1，θ_2，θ_3

或者你可以给变量起完全不同的名字。这完全取决于你，并且你可能会基于某些原因把变量命名为自己喜欢的名字。

在 C 语言编程中，情形很类似。你选择一个变量名，最好选择一个自己喜欢的名字。一个典型的 C 程序和一个典型的代数表达式之间的主要区别在于，代数表达式中，变量的名字通常只包含一个或两个字符，也许带有上标或下标。C 程序中的变量通常是一个词，而不是单个的字符。为什么呢？这是因为 C 程序通常比较长，所以就会需要很多变量。因此

没有足够的单个字符用来命名所有需要的变量。另外，你会发现如果把变量命名为描述性的名字，自己和别人都会更容易理解程序。

例如，用于计算三角形面积的程序，可以使用以下三个变量：

1）三个边长：length1, length2, length3

2）三个角度：angle1, angle2, angle3

或者可以给出更加详尽的描述，像下面这样命名变量：

1）三个边长：side_length1, side_length2, side_length3

2）三个角度：angle_opposite_side1, angle_opposite_side2, angle_opposite_side3

这些变量的命名比起那些在代数公式中的命名减少了很多歧义。不好的方面在于表达式中使用这些变量比起使用单个字符会更麻烦，但这是我们必须要接受的一个缺点。

源代码

```
#include <stdio.h>
void main (void)
{
    int month;
    float expense, income;

    month = 12;
    expense = 111.1;
    income = 100.;

    printf ("Month=%2d, Expense=$%9.2f\n", month, expense);

    month = 11;
    expense = 82.1;

    printf ("For the %2dth month of the year\n"
            "the expenses were $%5.2f \n"
            "and the income was $%6.2f\n\n", month, expense,
            income);
}
```

输出

```
Month=12, Expense=$   111.10

For the 11th month of the year
the expenses were $82.10
and the income was $100.00
```

解释

1）如何声明变量？C语言中变量的名字必须在使用之前声明。语句

```
int month;
```

把变量month声明为int类型（代表这是一个整型数，并且必须要用小写字符）。一个int类型的数据不包括小数点。另外，必须在程序的开头声明所有的变量。事实上这意味着你需要列出所有的变量，并指出每一个变量的类型。相同类型的变量可以声明在同一个语句中，每

一个变量必须用逗号分隔。例如，语句

```
float expense, income;
```

声明了 float（必须用小写字符）类型的变量 expense 和 income。float 类型的变量包含一个小数点，有或者没有小数部分。例如，1.、1.0 和 0.6 都是 float 数据类型。当把一个没有小数点的数值赋值给一个 float 类型的变量时，编译器自动地在最后一个数字后加上小数点。

2）如何命名变量？C 程序中的变量是用名字标识的。变量名是一种标识符，所以命名变量必须遵循命名标识符的规则。例如，标识符的第一个字符不能是数字。具体要求如下：

组成部分	要求
标识符中第一个字符	必须是小写字母 a～z、大写字母 A～Z 或者下划线
标识符中其他字符	必须是小写字母 a～z、大写字母 A～Z、下划线或者数字 0～9

表 2-1 列出命名合法标识符的规则。

不合法的变量名字包括：1apple、interest rate、total、float、Tr. come 和 one.two。

这些是合法的变量名：apple1、interest_rate、xfloat、income 和 one_two。

3）什么是关键字？关键字是 C 语言中有特定用途的标识性单词。课程程序中用的关键字包括 int、float 和 void。因为这些单词在 C 语言中有特殊的意义，所以不能使用它们作为变量名。C 语言中关键字的数量很少，仅仅有 32 个，如表 2-1 所示。表中所给出的其他关键字的用途会在本书的后续部分介绍。

表 2-1 关于标识符的限制

主题	注释
内部标识符（函数内部标识符）最多的字符数	ANSI C 允许内部标识符最多可以有 31 个字符
C 保留字，也叫做关键字	下面这些保留字用来述说 C 的基本指令，不能在程序中用它们命名变量： auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while
使用标准标识符，如 printf	类似于函数名字的标准标识符可以被用作变量名。但是这种用法不推荐，因为会导致混淆
用大写或混合大小写字符	允许，但是很多程序员用小写字母作为变量的名字，大写字母作为常量的名字。用不同的字符来区分不同的标识符，而不要用不同的大小写来区分标识符
标识符内部使用空格	不允许，因为一个标识符是一个单词

4）什么是赋值语句？赋值语句就是把值赋给变量的指令。这也意味着赋值语句把值保存在变量的内存单元中。

在 C 语言中，一个简单的赋值语句如下所示：

```
variable_name=value;
```

这个语句把等号右边的值赋给等号左边的变量。注意赋值语句中的等号并不意味着相等。

5）如何在屏幕上显示变量或常量的值？printf 函数用来在屏幕上显示变量或常量的值，它的语法如下：

```
printf(format_string, argument_list);
```

其中，format_string 是一个格式控制字符串文本，包含三种类型的元素：

- 第一个类型是 ANSI C 定义的纯字符，它们会不被修改地显示在屏幕上。例如上例中的“This is C!”消息。
- 第二个是转换限定符，用来把 argument_list 中的参数进行转换，格式化并显示。
- 第三个是 printf 函数，用来控制光标和插入点位置转义序列。例如在第 1 章课程 1.7 中描述的“\n”。

每一个参数必须指定一个格式，否则结果是未定义的。例如，语句

```
printf("month=%5d \n",month);
```

中，格式控制字符串是“month=%5d \n”。纯文本 month= 会不经修改而直接显示，转换限定符 %5d 用来把参数 month 转换、格式化并输出到屏幕，转义序列 \n 用来把插入点移到下一行。

用于显示 int 和 float 类型的最简单的 printf 转换限定符（也叫格式限定符）有下面的格式：

int 类型：%[域宽]d　例如 %5d

float 类型：%[域宽][. 精度]f　例如 %9.2f

在 [] 之间的内容是可选的，而且 [和] 不是格式化字符串的一部分。域宽是一个整数，代表预留显示参数的最小的字符数（包括小数、小数点前面和后面的数以及符号）。精度是一个整数，代表小数点后面可以出现的最大位数。例如，%5d 会预留 5 个空白位置用来显示 int 类型的数，%9.2f 会为一个 float 类型预留 9 个空白位置，并且小数点后面显示 2 位。这些概念显示在图 2-1 中，如果事实上输入的数据在小数点后面有较少的位数，当显示它的时候 C 编译器会自动地在后面加上零。例如，语句

```
expense=111.1;
printf("the expenses were $%9.2f\n",expense);
```

中，C 语言编译器会在小数点后面加上 0，以满足其精度等于 2。这样，expense 的值会显示为 111.10。

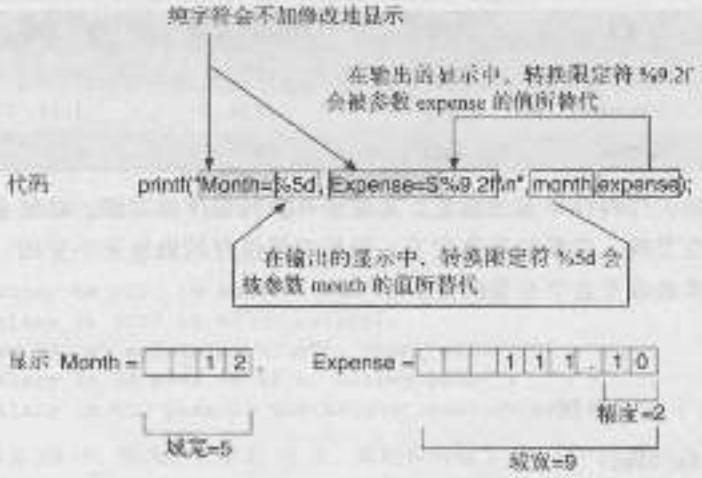


图 2-1　printf 中的格式限定符

扩展解释

1）声明一个变量的效果是什么？这会通知 C 语言编译器为保存这个变量在内存中预留

出多大的空间。根据变量的类型，C语言编译器会知道需要预留多少空间。虽然标准ANSI C并没有指定具体的数字，但是它隐式地定义了每一种数据类型所需的最少位数。例如，ANSI C要求int类型至少能容纳−32 767到32 767的数值。这就要求至少16位或者2字节的内存。因此当声明一个int类型的变量month的时候，就意味着16位或2字节的内存需要预留出来以保存这个变量的值。目前，大部分计算机都使用4字节用来保存一个int类型的变量。另一方面，float类型至少占据4字节或32位。因此当声明一个float类型的变量时，就意味着32位或4字节的内存需要预留出来以保存这个变量的值。

另外，C用不同的二进制编码来保存整型数和实型数。例如，以整型数保存的32的位模式和以浮点数保存的32的位模式完全不同。了解这一点非常重要，忽略这一点会非常容易导致错误。例如，如果printf试图去读一个包含int类型的内存单元，但是它却以float的格式显示，那么显示的结果是完全错误的。因为程序员会告诉printf函数以哪种格式去翻译内存单元，所以必须保证这种格式是正确的。本课程的后面会描述如何通知printf以何种格式去翻译内存单元。

2）当程序开始运行的时候，内存中发生了什么？理论上，会在内存中产生一个表格。这个表格包含变量的名字、类型、地址和值。名字、类型、地址会在编译的阶段建立。然后当执行的时候，内存空间会被分配出来，并且值会被保存在分配出来的内存空间中。例如，当课程中程序的前三个赋值语句被执行后，表如下所示。（注意内存单元地址用十六进制书写，在本书的后面也用十六进制来表示内存单元的地址。）

变量名	变量类型	内存单元地址	变量的值
month	int	FFF8	12
expense	float	FFF6	111.1
income	float	FFF2	100.

当第4个和第5个赋值语句执行后，表变成如下所示。

变量名	变量类型	内存单元地址	变量的值
month	int	FFF8	11
expense	float	FFF6	111.1
income	float	FFF2	82.1

现在你不需要关注内存单元的地址，当编译和执行程序的时候，地址会被自动设定。通过比较这两个表会发现，变量的值改变了，但是变量保存的地址是不变的。你会发现程序的主要目的就是持续地改变表中变量的值。

概念回顾

1）变量用以下方式声明：

```
type variable_list;
```

例如：

```
int month;
```

2）变量名必须遵循以下规则：

- 组成变量名的字符只能是字母、数字和下划线。

- 第一个字符必须是字母或者下划线，推荐使用字母。

3）赋值语句遵循下面的格式：

```
variable_name = value;
```

右边的值会赋给左边的变量。

4）变量的值可以用 printf 函数以下面的方式打印出来：

```
printf(format_string, argument_list);
```

例如，

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

会把变量 month 的值打印到屏幕上。

练习

1. 判断真假：

a. 下面的 int 类型的变量是合法的：

```
1cat, 2dogs, 3pears, %area
```

b. 下面的 float 类型的变量是合法的：

```
cat, dogs2, pears3, cat_number
```

c. 用于 int 类型的变量或常量的格式限定符 %d 和 %8D 是合法的。

d. 用于 float 类型的变量或常量的格式限定符 6.3f 和 %10.1F 是合法的。

e. 下面两个语句是完全等同的：

```
int ABC, DEF;
int abc, def;
```

2. 下面哪些变量的名字不正确，为什么？

```
enum, KMUM, lotus123, A+B23, A(b)c, AaBbCc, Else, αβχ, pi, π
```

3. 下面哪些是不正确的 C 语言语句，为什么？

```
year = 1967
1967 = oldyear;
day = 24 hours;
while = 32;
```

4. 假设 year 是一个 int 类型的变量，salary 是一个 float 类型的变量，下面哪个 printf ( ) 语句是不可以接受的，为什么？

```
printf("My salary in 2007 is $2000", salary);
printf("My salary in 2007 is %d\n",salary);
printf(In year %d, my salary is %f\n"), year, salary;
printf("My salary in %d year is %f\n, salary,year");
printf("My salary in %5d year is %10.2f\n\n",year,salary);
```

5. 一个苹果的价格是 50 分，鸭梨的价格是 35 分，瓜的价格是 2 元。写一个程序显示如下的价目表：

```
*****  ON SALE  *****
Fruit type       Price
Apple            $ 0.50
Pear             $ 0.35
Melon            $ 2.00
```

答案

1. a. 假　　b. 真　　c. 假　　d. 假　　e. 假

2. enum（保留字），a=123（不应该有 - 号），a(b)c（不应该有 ()）

3. year = 1967（没分号）

   1967=oldyear;（1967 是常量）

   while=32;（while 是保留字）

4. 只有 printf("My salary in %5d year is %10.2f\n\n", year, salary); 是可接受的。

课程 2.2　算术运算符和表达式

主题

- 运算数
- 算术运算符和它们的特点
- 算术表达式

C 语言中的算术表达式和你写的代数表达式非常类似。本课程第一部分的实例程序演示了能在 C 语言算术表达式中执行的操作。

注意本节程序中的下面两个语句：

i = i+1 和 j=j+1

很显然，如果这两个语句出现你的数学课上，那么它们是没有什么意义的。但是在 C 语言中这两个语句（这种类型的语句）不仅有意义，而且使用得非常普遍。回忆一下提到过的赋值语句，它把赋值语句右面的值保存到赋值语句左面的变量中去。

在这个程序的第二部分，是一些带有运算符的表达式，它们的功能并不是非常明显。参看这些语句以及相关的输出。% 符号非常有技巧，看看你能否发现它是做什么的。（提示：它和除法操作有关。）

同时需要注意，++ 和 -- 是操作符，这些语句中没有等号。但是它们会影响这些操作符前面或者后面的变量的值。查看程序的输出，看看它们对变量有什么影响。

源代码

```
#include <stdio.h>
void main (void)
{
    int i,j,k,p,m,n;
    float a,b,c,d,e,f,g,h,x,y;

    i=5; j=5;
    k=11; p=3;
    x=3.0; y=4.0;
    printf (".....  Initial values .....\n");
    printf ("i=%4d, j=%4d\nk=%4d, p=%4d\nx=%4.2f, y=%4.2f\
            n\n",i,j,k,p,x,y);

    /*--------------- Section 1 -------------------*/
    a=x+y;
    b=x-y;
    c=x*y;
    d=x/y;
    e=d+3.0;
```

```
    f=d+3;
    i=i+1;
    j=j+1;
    printf ("...... Section 1 output ......\n");
    printf ("a=%5.2f, b=%5.2f\nc=%5.2f, d=%5.2f\n"
            "e=%5.2f, f=%5.2f\ni=%5d, j=%5d \n\n", a,b, c,d,
            e,f, i,j);

    /*---------------- Section 2 -------------------*/
    m=k%p;
    n=p%k;
    i++;
    ++j;
    e--;
    --f;

    printf ("...... Section 2 output ......\n");
    printf ("m=%4d, n=%4d\ni=%4d, j=%4d\n"
            "e=%4.2f, f=%4.2f\n",m,n, i,j, e,f);
}
```

输出

```
...... Initial values ......
i=   5, j= 5
k=  11, p= 3
x=3.00, y=4.00

...... Section 1 output ......
a= 7.00, b=-1.00
c=12.00, d= 0.75
e= 3.75, f=3.75
i=    6, j=  6

...... Section 2 output ......
m=  2,  n=  3
i=  7,  j=  7
e=2.75, f=2.75
```

解释

1）算术表达式的组成部分是什么？一个算术表达式包含一系列用于计算数值的运算数和运算符。例如，表达式 -x + y 包含两个运算数 x 和 y 以及两个运算符 + 和 -。

2）运算符 ++、-- 和 * 的含义是什么？++ 是自增运算符，可以放到变量的前面或者后面（不能同时放到变量的前面和后面）。这个运算符会把这个变量的值增加 1。例如，假设一个变量 i 等于 1。当执行完下面的语句

```
i++;
```

或者

```
++i;
```

以后，i 的值就变为了 2。（i++ 和 ++i 并不完全一致，具体的细节见课程 2.5。）

注意 C 语句

```
i++;
```

或者

```
++i;
```

可以被理解成

```
i=i+1;
```

这个语句也把变量 i 的值增加 1。类似地，-- 是自减运算符，把 i 的值减去 1。注意 C 语句 i-- 或者 --i 可以被理解成

```
i=i-1;
```

运算符 % 是求余运算符。这个运算符必须放到两个整型变量或常量的中间。假设 k 和 p 是两个整型变量，那么 k%p 的结果是 k 除以 p 以后的余数。例如，如果 k=11，p=3，那么 k%p 相当于 11%3，也就等于 2（因为 11 减去 3 个 3 以后，余下一个 2）。% 操作符的发音是 mod。在本例中为 k mod p。ANSI C 指出，如果求余的两个操作数中的一个是负数，那么最后的值的符号由 C 语言编译器的设计者来决定。例如，根据不同的编译器，-20%6 或者 20%(-6) 可能是 2 或者是 -2。

3）算术表达式是完整的 C 语句吗？如何在 C 赋值语句中使用算术表达式？算术表达式并不是完整的 C 语句，只是一个语句的组成部分。被表达式计算出来的值可以保存在一个赋值语句的变量中。例如，算术表达式 x/y 只是以下 C 赋值语句的一部分：

```
d = x/y;
```

这个语句把右边算术表达式计算出来的值赋给左边的变量。赋值语句

```
i=i+1;
```

虽然不是一个合法的代数表达式，但却是一个合法的 C 语言赋值语句。算术表达式 i+1 生成了一个新值，这个新值比当前变量 i 的值大 1。然后，赋值语句把这个新值赋给 i。

注意我们不能写出下面两个赋值语句：

```
x/y = d;
i + 1 = i;
```

这是因为赋值语句的左边只能有一个变量，而不是一个表达式。单个的变量可以是左值，意味着它们可以出现在赋值语句的左边。表达式是右值，它们只能出现在赋值语句的右边。

4）一个单独的变量可以被认为是一个表达式吗？可以，例如一个出现在赋值语句右边的单个变量可以被认为是表达式。我们会遇到其他一些情况，单个的变量也可以被认为是表达式。

5）当我们试图除以 0 的时候会发生什么？通常，会引发一个运行时错误，然后你的程序会被终止。一个包含 overflow 的错误信息会显示在计算机的屏幕上，这是因为当除以 0 或一个接近 0 的数的时候，会产生一个很大的数，这个很大的数无法保存在分配的内存单元中，所以产生了一个溢出的错误。

概念回顾

1）C 语言中的算术表达式就像正常的算术表达式一样构建。例如，+、-、* 和 / 分别代表加、减、乘和除。

2）++ 和 -- 分别称为自增和自减运算符，它们分别把变量增加或减少 1。这两个运算

符可以放到变量的前面和后面。

3）% 是求余运算符，它会返回两个整数相除后的余数。

练习

1. 判断真假：

a. a+b 是一个正确的算术表达式。

b. x=a+b; 是一个完整的 C 语句。

c. 如果 a=5，那么当执行 a++ 后，a 等于 6，但是，当执行 ++a 后，a 还等于 5。

d. 5%3 等于 2，3%5 等于 3。

e. % 的操作数必须是整数。

f. 在下面的语句中

```
a = x+y;
```

等号的意义是相等，也就是说，a 等于 x+y。

2. 假设 a、b、c 是 int 变量，而 x、y、z 是 float 变量，下面哪些语句是不正确的 C 语句？

```
a+b = c;
a+x =y;
c = a%b;
a/b = x+y;
x = a*3;
z= x+y;
```

3. 写一个程序，计算你本学期 GPA 的平均成绩，并把结果显示在屏幕上。

答案

1. a. 真　b. 真　c. 假　d. 真　e. 真　f. 假

2.
```
a+b = c;
a+x = y;
a/b = x+y;
```

课程 2.3　从键盘输入数据

主题

- 使用 scanf() 函数
- 从键盘输入数据
- 地址操作符 &
- double 数据类型

上面课程中的程序在运行的时候并没有输入，只有输出，输出的设备是屏幕（显示器）。通常情况下，你的程序会同时有输入和输出。在下列情形中，输入是非常重要的。写一个程序，根据作业和考试的成绩计算学生的成绩级别。在程序中可以通过赋值语句输入一个学生的成绩，但是这个程序在需要计算另外一个同学的成绩级别的时候，只能重新进行编译。很明显，一个更好的办法就是这个程序在运行的时候能够接受用户的输入。你的程序告诉计算机从各种不同的输入设备中接收数据，这些输入设备包括：

- 键盘
- 鼠标
- 硬盘
- USB 盘

本课程的程序演示了如何从键盘读入数据。

通常从键盘输入数据的程序运行的时候，在用户和程序之间生成一个对话框。

源代码

```
#include <stdio.h>
void main (void)
{
    float income;
    double expense;
    int month, hour, minute;

    printf ("What month is it?\n");
    scanf ("%d", &month);
    printf ("You have entered month=%5d\n",month);

    printf ("Please enter your income and expenses\n");
    scanf ("%f %lf",&income,&expense);
    printf ("Entered income=%8.2f, expenses=%8.2lf\n",
            income,expense);

    printf ("Please enter the time, e.g.,12:45\n");
    scanf ("%d : %d",&hour,&minute);
    printf ("Entered Time = %2d:%2d\n",hour,minute);
}
```

输出

```
                What month is it?
键盘输入        12
                You have entered month=  12
                Please enter your income and expenses
键盘输入        32  43
                Entered income =  32.00, expenses =  43.00
                Please enter the time, e.g., 12:45
键盘输入        12:15
                Entered Time = 12:15
```

解释

1）如何从键盘输入数据？从键盘输入数据的最简单方法是使用 scanf 函数。这个函数的语法是：

```
scanf (format_string, argument_list);
```

format_string 用于把输入的字符转换成特定数据类型的值。argument_list 包含保存这些值的变量的地址，逗号用来分隔每一个参数，例如：语句

```
scanf("%f%lf",&income,&expense);
```

会把从键盘输入的第一个数据根据 %f 这个转换限定符转换成一个 float 类型的值，并把这个值保存到内存中为变量 income 预留的内存单元中。同样把从键盘输入的第二个数据根据 %lf 这个转换限定符转换成一个 double 类型的值，并把这个值保存到内存中为变量 expense 预留的内存单元中。注意，必须在每一个变量名字前面加上一个 &。因为 scanf 函数中的参数使用的是变量的地址（&income 代表的是保存变量 income 的内存单元的地址）。

同样，&expense 代表的是保存变量 expense 的内存单元的地址。这个 scanf 语句一共把三个参数传递给了 scanf 函数：第一个是包含转换限定符的字符串文本，第二个是变量 income 的地址，第三个是变量 expense 的地址。

图 2-2 演示了函数 scanf 的调用。如果想读入一个 int 数据，用 %d 而不是 %f 或 %lf 作为转换限定符。注意 %lf 中的 l 是英文小写字母。%f 在这里代表着用一个标准的浮点类型变量保存输入的值。%lf 代表的是双精度浮点类型（也用来保存实型数，但是有多一倍的存储空间）。双精度浮点类型会在第 3 章中详细地讨论。

2）能更详细地介绍 & 符号吗？如果观察更多的 C 语言库函数，我们会发现有些函数需要输入变量的值（用变量的名字代表），而有的函数需要输入变量的地址。可以通过在变量的前面放上取址操作符 & 来把变量的地址传入到函数中去。

scanf 函数需要内存单元的地址，这是因为它需要知道把从键盘输入的值放到具体哪个地方。通过传入 scanf 地址，程序会知道把这个值放到内存的哪里，如图 2-2 所示。

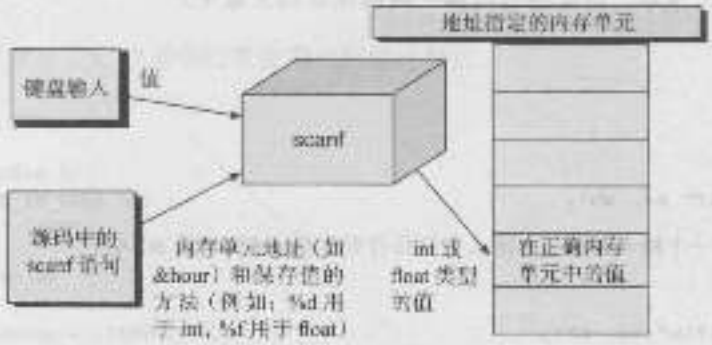


图 2-2 scanf 在变量读入的时候使用地址

3）scanf 函数中的格式控制字符串有哪些组成部分？格式控制字符串包含格式转换限定符（如 %d 或者 %lf）、空格和输入的字符。如果格式控制字符串包含字符，当你从键盘输入的时候必须匹配这些字符，例如语句

```
scanf("%d : %d",&hour,&minute);
```

在格式字符串中包含一个冒号（ : ）。如果你想输入 hour=12 和 minute=34，合法的输入如下：

```
12 : 34
```

如果你忽略了冒号，数据就会读入错误。通常 scanf 中的格式字符串应该尽量保持简单以减少输入的错误。本书的大多数情况，在格式字符串中我们只使用转换限定符，而不使用其他纯字符。

4）使用 scanf 函数时常见的错误有哪些？通常你会在变量的前面忘记 & 符号。记住 printf 不使用 & 而 scanf 使用 &。

如果在 scanf 语句前面不使用 printf 语句提示用户的话，不算什么错误。因为 scanf 语句会暂停程序的运行以等待从键盘的输入。如果不用 printf 语句在屏幕上打印一些信息来告诉用户现在该输入了，计算机会一直等待键盘的输入。因此，如果没有 printf 语句的提示，用户会不知道什么时候输入或输入什么。所以，记住在 scanf 语句前面要使用 printf 语句。

同样重要的是，格式限定符和变量类型一定要匹配。换句话说，必须使用 %lf 输入 double（当保存成一个 double 类型的数据时，%f 不会工作，它只会保存一些垃圾信息），使用 %f 输入 float，使用 %d 输入 int。

概念回顾

1）double 是一个增强版的 float 数据类型，它比 float 提供更高的精度和更大的范围。

2）scanf 用来从键盘输入数据，格式如下：

```
scanf(format_string, argument_list);
```

例如，

```
scanf("%f%lf",&income,&expense);
```

分别读入两个浮点数，并把它们保存到 income 和 expense 中去。

必须要使用符号 &，以便值可以被正确地保存到变量中。

练习

1. 判断真假：

a. 语句

```
scanf("%f %lf" &a, &b);
```

会把键盘的第一个输入和第二个输入分别保存到变量 &a 和变量 &b 中。

b. 语句

```
scanf("%f %lf\n",&a, &b);
```

包含四个参数。

c. 语句

```
scanf("%d:%f:%lf\n",&a, &b,&c);
```

包含 4 个参数 "%d:%f:%lf\n"、&a、&b 和 &c。

2. 基于语句

```
int cat, dog;
double weight;
```

指出下面语句中的错误：

```
a. scanf("%d %d"),cat,dog;
b. scanf(%d %d,cat,dog);
c. scanf("%d %f",cat,dog);
d. scanf("%d %d",&cat,&dog);
e. scanf("%d, %lf",&cat,&weight);
```

3. 写一个程序，从键盘输入你上学期所有的成绩，然后在屏幕上打印你的平均 GPA。

答案

1. a. 假　　b. 假　　c. 真

2. a. `scanf("%d %d",&cat,&dog);`

b. `scanf("%d %d",&cat,&dog);`

c. `scanf("%d %d",&cat,&dog);`

d. 没错误。

e. 没错误，但是必须在两个值之间输入一个逗号。

课程 2.4 常量宏及打印变量值的进一步讨论

主题

- 用 define 指令来定义常量
- 更多的转换限定符及组成
- 科学计数法
- 转换限定符中的标志

在编写程序的时候，你会发现经常需要使用一些在程序中并不会改变的数值。例如，我们都知道 π 大约为 3.14159。对于一个计算圆的程序来说，在公式中使用字符 PI 要比使用数字 3.14159 方便很多。在 C 语言中可以通过使用常量宏来达到这个目的。常量宏通过用预处理指令来生成。

另外，在一些涉及很大或很小的数的算术运算中，科学计数法也是非常方便的。例如，为了表示 57 650 000，科学计数法的表示是 5.765×10^7。C 语言中会表示成 5.765e+007 或者 5.765E+007。

下面的程序演示了 C 语言中的常量宏和科学计数法。

源代码

```
#include <stdio.h>
#define DAYS_IN_YEAR 365
#define PI 3.14159

void main (void)
{
    float income = 1234567890.12;

    printf ("CONVERSION SPECIFICATIONS FOR INTEGERS \n\n");
    printf ("Days in year = \n"
            "[[%1d]] \t(field width less than actual)\n"
            "[[%9d]] \t(field width greater than actual)\n"
            "[[%d]]  \t(no field width specified)  \n\n\n",
            DAYS_IN_YEAR, DAYS_IN_YEAR, DAYS_IN_YEAR);

    printf ("CONVERSION SPECIFICATIONS FOR REAL NUMBERS\n\n");
    printf ("Cases for precision being specified correctly \n");
    printf ("PI = \n"
            "[[%1.5f]] \t\t(field width less than actual) \n"
            "[[%15.5f]] \t(field width greater than actual)\n"
            "[[%.5f]] \t\t(no field width specified) \n\n",
            PI,PI,PI);

    printf ("Cases for field width being specified correctly \n");
    printf ("PI = \n"
            "[[%7.2f]] \t\t(precision less than actual) \n"
            "[[%7.8f]] \t\t(precision greater than actual)\n"
            "[[%7.f]] \t\t(no precision specified) \n\n",
            PI,PI,PI);
    printf ("PRINTING SCIENTIFIC NOTATION \n\n");
    printf ("income = \n"
            "[[%10.2e]] \t(field width large, precision small) \n"
            "[[%8.5e]] \t(field width and precision medium size)\n"
            "[[%4.1e]]  \t\t(field width and precision small) \n"
            "[[%e]      \t(no specifications)  \n\n",
            income, income, income, income);
```

```
    printf ("USING A FLAG IN CONVERSION SPECIFICATIONS \n\n");
    printf ("Days in year= \n"
            "[[%-9d]] \t\t[field width large, flag included]\n",
            DAYS_IN_YEAR);
}
```

输出

```
CONVERSION SPECIFICATIONS FOR INTEGERS

Days in year =
[[365]]                 (field width less than actual)
[[      365]]           (field width greater than actual)
[[365]]                 (no field width specified)

CONVERSION SPECIFICATIONS FOR REAL NUMBERS

Cases for precision being specified correctly
PI =
[[3.14159]]             (field width less than actual)
[[        3.14159]]     (field width greater than actual)
[[3.14159]]             (no field width specified)

Cases for field width being specified correctly
PI =
[[  3.141]]             (precision less than actual)
[[3.14159000]]          (precision greater than actual)
[[3.141590]]            (no precision specified)

PRINTING SCIENTIFIC NOTATION

income =
[[          1.23e+09]]  (field width large, precision small)
[[1.23457e+09]]         (field width and precision medium size)
[[1.2e+09]]             (field width and precision small)
[[1.234568e+09]]        (no specifications)

USING A FLAG IN CONVERSION SPECIFICATIONS

Days in year =
[[365      ]]           (field width large, flag included)
```

解释

1）如何生成常量宏？我们使用预处理指令来生成常量宏。在C语言中预处理指令以符号#开始。在末尾一定不要使用分号，并且保证预处理指令写在同一行。例如

```
#define DAYS_IN_YEAR 365
```

是一个叫做define指令的预处理指令。这里定义DAYS_IN_YEAR等于常数365。

2）预处理器如何处理预处理指令？define指令的格式如下：

```
#define symbolic_name  replacement
```

其中，symbolic_name是生成的常量宏的名字，replacement用来代替symbolic_name的值。回忆在课程1.6中讲过，预处理器是编译器的一部分，它在把源代码翻译成机器语言前自动执行一些操作。给定define指令，预处理器会把源程序中出现的每一个symbolic_name替换成给定的replacement（除了出现在注释和字符串文本中的）。

例如，在本课的程序中，语句

```
printf("Days in year=%5d\n",DAYS_IN_YEAR);
```

中的符号DAYS_IN_YEAR会在程序被翻译成机器语言之前替换成365。换句话说，上面的语句会被预处理器重写为

```
printf("Days in year=%5d\n",365);
```

注意每一行只能定义一个常量宏。常量宏不能放到赋值语句的左边，这意味着我们不能在程序的后续部分给常量宏赋一个新值。想一下这个操作的含义，你就理解为什么不可以了。例如，如果在程序中写下了 DAYS_IN_YEAR=365.25 这样一个赋值语句，那么预处理器会在把源代码翻译成目标码之前，把它转换成 365=365.25。这个语句没什么意义，只是为了演示为什么不能把一个常量宏放到赋值语句的左边。也就是说，常量宏不能是左值。它不能放到赋值语句的左边。可以认为常量宏是一个右值，这意味着它可以放到赋值语句的右边而不是左边。

3）完整的 int 和 float 的格式限定符是什么？完整的格式限定符是：

```
%[flag][field width][.precision]type
```

所有在 [] 的格式限定符里面的内容是可选的。[] 字符不是格式字符串的一部分。更多细节见图 2-3。

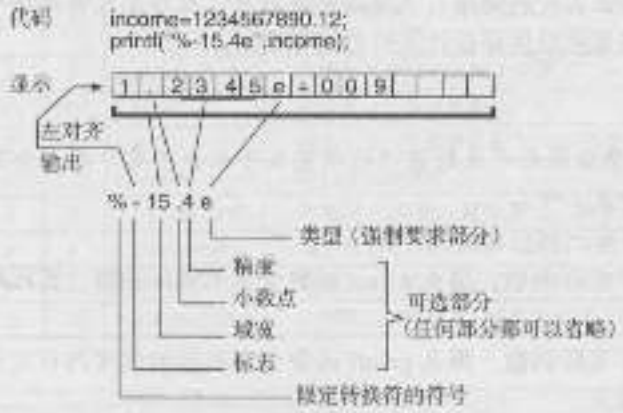


图 2-3 格式字符串中不同组成部分的含义

这些组件的意义可能会随编译器的不同而稍有不同。所以应该参考编译器的手册。在 ANSI C 下使用的部分标志和类型见表 2-2。

表 2-2 ANSI C 的标志和类型

组件	用法
flag = -	这个标志使得输出在给定的域宽内左对齐
flag = +	这个标志使得输出在给定的域宽内右对齐，并且如果显示的是一个正数，前面会有一个正号
flag = 0	这个标志使得输出的前面显示 0 以填充最小的域宽，如果它和标志位一起使用，那么这个标志会被忽略
field width	这个参数代表为了显示整个输出所需的最少的字符空白数（包括小数点、小数点前后的数以及符号）。如果指定的域宽没有输出或者小于实际的域宽，域宽被自动扩展以容纳需要显示的数值。混合使用域宽和精度来限定小数点的前面和后面显示多少个数字
precision	对浮点数类型，精度指定小数点后面显示多少位。对于 float 类型默认的精度是 6。精度也可以用在整型数据中，用来指定最小的显示位数。如果要显示的数字比指定的精度低，C 语言编译器会在输出的左边加上前导的零
type = d	用于整型数
type = f	输出被转换成小数形式 [符号]ddd.dddd，其中小数点后的位数等于指定的精度
type = e 或者 E	输出被转换成科学计数法的形式 [符号]d.dddd e[符号]ddd，其中小数点前面的位数为 1，小数点后面的位数等于指定的精度。指数至少是 2。如果值为 0，则指数为 0

在使用 printf 语句的时候，你应该阅读表 2-2，并了解转换限定符中可以使用的特性。

4）ANSI C 如何在输出时把一个浮点数转换成科学计数法？

它把一个浮点数转换成如下格式的科学计数法：

```
[sign]d.ddd  e[sign]ddd
```

其中 d 代表一个数字，注意这种格式表示的数据等同于

```
[sign] d.ddd × 10[sign]ddd
```

例如，当使用格式 %15.4e 的时候，表示宽度为 15，精度为 4。把 123456789.12 这个数显示为科学计数法的格式会得到

```
␣␣␣␣1.2346e+009
```

这等同于

1.2346×10^{9} 或者 123460000.00

在显示的时候会丢失一些精度，这是因为我们并没有指定足够高的精度。但是这仅仅影响显示部分，完整的值依然保存在计算机的内存中。

扩展解释

1）如果指定的整型数或浮点数显示的域宽大于或小于实际数，会显示什么？如果我们不指定显示域宽，会显示什么？

从本课程的程序输出你会观察到：

- 如果域宽小于实际的数，那么 printf 函数会显示实际的数，实际的宽度等于显示的数的位数。
- 如果域宽大于实际的数，那么 printf 函数会在指定的宽度内右对齐，数左边的多余的空位用空格填充。
- 如果不指定，就按实际的宽度，即显示的数的位数。

图 2-4 演示了使用 %d 格式来显示一个整型数并指定了宽度时的效果。这个图显示了数的宽度大于指定的位宽。对实型数（%f 或 %e），指定的位宽和指定的精度同时影响显示的时候有多少空间。但是，对实型数，指定的宽度是第二位重要的，因为 printf 函数会优先利用指定的精度来进行显示。

图 2-4 printf 函数显示整型数时指定域宽的效果

2）对于实型数，如果指定的精度大于或小于实际的数，如何进行显示？如果不指定，如何进行显示？

从本课程的程序输出你会观察到：

- 如果小于实际的数，那么只有指定精度的位数才会被显示出来。（被截断的位数并没有在内存中丢失，它们只是没有显示出来。）

- 如果大于实际的数，printf 函数会在末尾加上零，使得显示的精度等于指定的精度。
- 如果不指定，那么 printf 函数默认显示 6 位精度（或者在末尾加 0 或者截断小数位以获得 6 位精度，这些操作都不会影响保存在内存中的值）。

3）“-”标志有什么用？改变默认的右对齐方式为左对齐。

4）不同的转换限定符如何显示 365、3.1416 和 1234567890.12？表 2-3 和图 2-5 中演示了一些实例，这些实例使用 %d 显示 365，使用 %f 显示 3.1416，使用 %e 或 %E 显示 1234567890.12。

表 2-3 样例显示

转换	标志	域宽	类型	精度	显示（␣代表空格）	注解
%+5d	+	5	d	无	␣+365	右对齐输出，显示 + 号，一共显示 5 个字符
%-5d	-	5	d	无	365␣␣	标志是 -，所以输出左对齐
%1d	无	1	d	无	365	指定的域宽小于实际的宽度，所以显示数值，没有截断
%0.5d	0	0	d	5	00365	标志是 0，输出的前面加零，显示五位数
%d	无	无	d	无	365	域宽未定义，所有的位被显示，没有截断。没有前面空格，左对齐
%+9.5f	+	9	f	5	␣+3.14160	共输出 9 个字符，加上空格
%-9.5f	-	9	f	5	3.14160␣␣	标志是 -，左对齐输出
%1.3f	无	1	f	3	3.142	精度是 3，注意结果是 3.142，不是 3.141
%f	无	无	f	无	3.141600	使用默认的精度 6
%-12.4e	1	15	e	4	␣1.2346e-009	共输出 12 位数，域宽为 15，包含 e 和 +，精度是 4
%-12.4e	2	15	e	4	1.2346e-009␣	与上例类似，但是有一标志，所以左对齐
%5.2e	无	5	e	2	1.23e+009	精度是 2，域宽太小了，所以 C 用最小域宽输出
%E	无	无	E	无	1.234568E1009	C 用默认的精度 6，域宽太小了，所以 C 用最小域宽输出

图 2-5 使用不同的格式控制字符串来打印浮点类型数

注意，如果显示的宽度没有指定，或者小于要显示的实际的数，那么它们会显示实际的数值。也就是说，显示的宽度不会截断要显示的数。

5）给定相同的数值和显示格式，如果使用不同的编译器，那么输出会完全相同吗？不会，这是因为不同的编译器遵循不同的ANSI C标准。所以，通常即使给定相同的数值和显示格式，如果使用不同的编译器，它们的显示也会稍有不同。

6）如果使用%f来显示一个int类型的值，或者使用%d显示一个float类型的值，会发生什么事情？如果这么做，在屏幕上会显示完全没有任何意义的数，或者一个零。这是新程序员（也包括有经验的程序员）经常犯的一个错误。这会非常让人失望，因为程序中的其他部分都是正确的，仅仅因为错误地使用了%d而不是%f，使得程序看起来出现了重大的问题。你会花费很多时间去检查逻辑，然后做手工验算。最后发现，仅仅是一个小的格式转换符引发了这个错误。如果你的程序输出了没有意义的值或者零，在调查其他一些比较难以追踪的错误源之前，应该先检查格式转换符。

7）对于工程类型的程序，使用%f类型的格式显示数据是否有一定的风险？

如果数字非常小，有的时候它们会打印成零。例如，需要一个非常小的量度，想打印3.5×10^{-12}。如果使用%f打印，你会得到0.000000。所以当处理很小的数时，应该使用%e来显示有意义的结果。

8）为什么我们给printf函数如此多的关注？这里有两个原因。一个原因在于你会经常使用printf语句。如果你能熟练地使用printf语句，那么编程的其他领域对你来说也会很简单。熟练使用printf函数是一个很好的建议，这样你就可以很容易地进行其他的编程事务。另外一个原因是，不能正确使用printf语句是很多新手程序员经常犯的一个错误。如果你理解了printf语句，会在很大程度上减少编程的错误。

概念回顾

1）define指令的格式如下：

```
#define symbolic_name  replacement
```

其中，在程序中出现的symbolic_name会被预处理器在编译程序之前替换成replacement。

2）格式限定符的完整格式如下：

```
%[flag][field width][.precision]type
```

格式限定符中所有在[]内的内容是可选的。

3）浮点数的科学计数法输出是：

```
[sign]d.ddd  e[sign]ddd
```

其中d代表一个数字，注意这种格式表示的数据等同于

```
[sign] d.ddd x 10^[sign]ddd
```

练习

1. 判断真假：

a. 语句printf("%-3d",123);显示-123

b. 语句printf("%+2d",123);显示+12

c. 语句printf("%-2f",123);显示12.0

d. 语句printf("%+f.3",123);显示.123

e. 用于int类型的格式限定符不应该包含小数点和精度，例如%8.2d是非法的

2. 如果下面的语句有错误，请指出。

```
a. #DEFINE PI 3.1416
b. #define PI 3.1416;
c. #define PI=3.14; More_AccuratePI=3.1416;
d. printf("%f",123.4567);
e. printf("%d %d %f %f",1,2,3.3,4.4);
```

3. 下面那句是不正确的 define 预处理指令，为什么？

```
define speed of light 30000
#define long 12345678901234567890
#define SHORT 0.01;
#DEFINE RADIUS 30
```

4. 假设 rate 是一个 float 变量，year 是一个 int 变量，如果在下面的语句中发现错误，请修改它们。

```
printf("The interest rate in %-.d year is %+.F%\n", year, rate);
printf("In year %#d5, the interest rate was 8.2f%\n", rate, year, rate);
printf("In %+5.8d, the interest rate will be %010.18e%\n", year, rate);
```

5. 写一个程序显示如下的输出：

```
123456789012345678901234567890123 45
income      expense      Name
+111.1      -999.99      Tom
+222.2      -888.88      Dennis
+333.3      -777.77      Jerry
```

6. 使用四种不同的标志但是相同的域宽和精度，四种不同的域宽但是相同的标志和精度，四种不同的精度但是相同的标志和域宽（一共 12 种格式限定）来显示一个 int 类型变量 A 和一个 float 类型变量 B，其中 A=12345 且 B=9876.54322。

答案

1. a. 假　　b. 假　　c. 假　　d. 假　　e. 假

2. a. #define PI 3.1416
   b. #define Pi 3.1416
   c. #define PI 3.14
      #define More_AccuratePI 3.1416
   d. 没有错误
   e. 没有错误

3.
```
#define speed_of_light 30000
#define LONG 12345678901234567890
#define SHORT 0.01
#define RADIUS 30
```

4.
```
printf("The interest rate in %d year is %+.f%\n",
year, rate);
printf("In year %d, the interest rate was %8.2f\n",
year, rate);
printf("In %8d, the interest rate will be %10.1e%\n",
year, rate);
```

课程 2.5　混合类型的运算、复合赋值、运算符优先级和类型转换

主题

- 算术运算符的优先级
- 初始化变量

- 算术运算中的陷阱
- 在算术表达式中混合使用整型数和实型数
- 类型转换
- 副作用

当在一个算术表达式中使用变量的时候，必须首先赋予它一个数值。给变量赋第一个数值叫做初始化。后续会学习几种不同的方法来进行初始化。

在数学课上你可能已经学习了括号能用在表达式中来改变运算的顺序。在C语言中，也可以使用括号来改变运算的顺序。同时C语言对加减乘除等操作有非常严格的顺序规定。这些规定建立在运算符的优先级上。优先级高的运算符要比运算符低的先执行。当两个运算符具有相同的优先级，那么左边的先执行。

阅读下面的程序和输出，程序中演示了变量初始化的概念、混合算术运算和运算优先级。

源代码

```
#include <stdio.h>
void main (void)
{
    int i=1, j=1,
        k1=10, k2=20, k3=30, k4=40, k5=50,
        k, h, m, n;
    float a=7, b=6, c=5, d=4,
          e, p, q, x, y, z;

    printf ("Before increment, i=%2d, j=%2d\n",i,j);

    k=i++;
    h=++j;

    printf ("After increment, i=%2d, j=%2d\n"
            "                 k=%2d, h=%2d \n\n",i,j,k,h);

    m=6/4;
    p=6/4;
    n=6/4.0;
    q=6/4.0;

    printf ("m=%2d, p=%3.1f\nn=%2d, q=%3.1f\n\n",m, p, n, q);
    printf ("Original k1=%2d, k2=%2d, k3=%2d, k4=%2d,
            k5=%2d\n",k1,k2,k3,k4,k5);

    k1 += 2;
    k2 -= i;
    k3 *= (8/4);
    k4 /= 2.0;
    k5 %= 2;

    printf ("New  k1=%2d,k2=%2d,k3=%2d,k4=%2d,k5=%2d\n\n",
            k1,k2,k3,k4,k5);

    e= 3;
    x= a + b -c /d *e;
    y= a +(b -c)/d *e;
    z=((a + b)-c /d)*e;
```

```
    printf("a=%3.0f,b=%3.0f,c=%3.0f\nd=%3.1f,e=%3.1f\n\n",
           a,b,c,d,e);

    printf ("x=  a + b -c  /d *e = %10.3f \n"
            "y=  a +(b -c) /d *e = %10.3f \n"
            "z=((a + b)-c  /d)*e = %10.3f\n", x,y,z);
}
```

输出

```
Before increment, i= 1, j= 1
After increment,  i= 2, j= 2,
                  k= 1, h= 1
m= 1, p=1.0
n= 1, q=1.5

Original k1=10, k2=20, k3=30, k4=40, k5=50
New      k1=12, k2=18, k3=60, k4=20, k5= 0

a= 7, b= 6, c= 5
d=4.0, e=3.0

x= a + b -c /d *e  =  9.250
y= a +(b -c) /d *e =  7.750
z=((a + b)-c /d)*e = 35.250
```

解释

1）如何初始化变量？有两个方法初始化变量。

- 方法 1：使用赋值语句来初始化变量，例如

```
e=3;
```

- 方法 2：在声明变量的时候初始化，例如

```
float a=7, b=6;
```

2）假设 int 类型的变量 i 和 j 都等于 1，k=i++ 是否等同于 h=++j？不等同。第一个语句中，i 的值首先被赋给了变量 k，当赋值结束后，变量 i 被后自增运算符从 1 增加到 2。因此语句

```
k=i++;
```

执行后，i = 2，k = 1。但是对于 h = ++j，j 的值被前自增运算符从 1 增加到 2。增加结束后，新的 j 等于 2，赋给了变量 h。因此语句

```
h=++j;
```

执行后，j=2，h=2。换句话说，语句

```
k=i++;
```

等价于下面两句：

```
k=i;
i=i+1;
```

而语句

```
h=++j;
```

等价于下面两句

```
j=j+1;
h=j;
```

下面是这些运算符的运算规则：

- 如果增加运算符或减少运算符出现在变量的前面，变量被首先增加或减少1。然后把这个新值用在表达式的运算中。
- 如果增加运算符或减少运算符出现在变量的后面，表达式首先被运算，然后变量被增加或减少1。

你必须记住以上两个规则。

3）6/4的值是多少？这里，一个整型数被另一个整型数除。如果两个运算数都是正数或负数，那么结果中的小数部分会被舍弃。因此6/4不是等于1.5，而是等于1，如图2-6的中间部分所示。

4）6.0/4.0的值是多少？由于两个运算数都是实型数，所以结果也是实型数的1.5，如图2-6的开始部分所示。

5）6/4.0的值是多少？对于这个运算，一个运算数是实型数，而另外一个运算数是整型数。当这种情况发生在C语言中的时候，C语言暂时把这个整型数转换为一个实型数（这意味着6转换成6.0）。然后运算并得到实型数。因此，6/4.0最后等于1.5。整个过程如图2-6的底部所示。

6）如果把一个实型数的值赋给一个声明为整型数的变量，那么会发生什么？C把小数点的部分去掉，把剩下的部分转换为一个int类型的值，然后在变量的单元格内以二进制的补码方式来保存这个变量。例如，语句

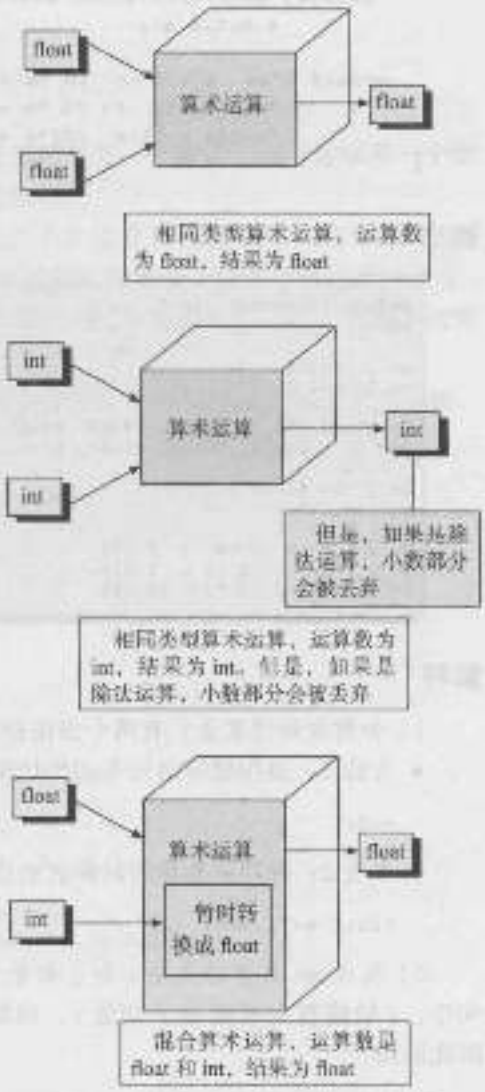


图2-6 混合及相同数据类型的计算

```
n = 6 / 4.0;
```

把实型值（6/4.0这里是1.5）去掉0.5而剩下1.0。然后把1.0转换成1（没有小数点，意味着它是以2的补码方式保存的），并保存在变量n的内存单元中。

7）在算术运算中，可以修改C使用的类型吗？可以，本课程的程序并没有演示这么做，但是C语言有转换符，可以用来转换表达式的类型（回忆一个单独的变量可以是一个表达式）。因此我们可以在赋值语句的左边使用转换符来修改算术表达式结果的类型。一个转换符由括号内的类型的名字构成。

例如，已经声明了变量

```
    int aa=5, bb=2, cc;
float xx, yy=12.3, zz=18.8;
```

转换符可以如下使用：

```
xx = ((float) aa) / ((float) bb);
     cc = (int)yy +(int) zz;
```

为了理解这些语句中的操作，我们需要意识到，在执行算术运算的时候，C语言拷贝变量的值，然后一直用这个拷贝工作。当运算结束以后，得到一个表达式的最后的值。如果表达式在赋值语句的右边，那么发生赋值运算。

于是，转换符（float）和（int）生成了aa的拷贝5.0，bb的拷贝2.0，yy的拷贝12，zz的拷贝18。表2-4总结了这些操作。因为运算是基于这些拷贝的，所以我们可以清晰地看到，保存到赋值语句左边的变量的值为：

```
XX = 5.0/2.0 = 2.5
CC = 12 + 18 = 30
```

如果这段程序没有使用转换符，那么最后的值如表2-5所示。

表2-4 转换运算符的行为

变量名和类型	初始值	转换操作	用在算术运算中的拷贝的值
int aa	5	(float)5	5.0
int bb	2	(float)2	2.0
float yy	12.3	(int)12.3	12
float zz	18.8	(int)18.8	18

表2-5 不用转换运算符的表达式

没有转换运算符的表达式	保存的值
xx=aa/bb;	xx=5/2=2.0（因为xx是一个浮点数，但是操作数都是int）
cc=yy+zz;	cc=12.3+18.8=31（因为cc是一个整型数，而操作数都是float）

我们可以看到转换符确实更改了变量xx和cc的值。

8）转换符的一般形式是什么？转换符的一般形式如下：

```
(type) expression
```

转换符必须包含在一个括号内。type可以是任何合法的C语言的类型。本文后面我们会看到更多合法的C语言的类型。

9）+=、-=、*=、/=和%=运算符的含义是什么？这些运算符叫做复合赋值运算符。它们都执行一个算术运算和一个赋值运算。这些运算符需要两个运算数。左边的运算数必须是变量，右边的可以是常量、变量或表达式。通常，两个运算数可以是整型数或者实型数。但是，%=运算符的两个运算数必须都是整型数。

例如，k1+=2;（不要写成k1=+2）的含义可以理解为

```
k1=k1+2;
```

如果k1原始的值等于20，那么新的值将会是20+2或22。同样，当我们把+号换成-、*、/或%的时候，语句也是合法的。例如，

```
k1*=2;
```

等同于

```
k1=k1*2;
```

10）如何控制算术表达式中的优先级？括号可以用来控制优先级。括号内的算术运算符比括号外面的有更高的优先级。当表达式中有多对括号的时候，最内部的括号有最高的优先级。例如，下面语句中的如号

```
z = ((a+b)-c/d);
```

比 - 和 / 有更高的优先级，所以 a+b 会被优先运算。

11）哪些运算符可以用在算术表达式中？表 2-6 给出了可以用在算术表达式中的运算符以及它们的性质。

表 2-6 算术运算符

运算符	名字	运算数数目	位置	结合性	优先级
(	括号	单目	前置	左到右	1
)	括号	单目	后置	左到右	1
+	正号	单目	前置	右到左	2
-	负号	单目	前置	右到左	2
++	后置增 1	单目	后置	左到右	2
--	后置减 1	单目	后置	左到右	2
++	前置增 1	单目	前置	右到左	2
--	前置减 1	单目	前置	右到左	2
+=	加赋值	双目	中置	右到左	2
-=	减赋值	双目	中置	右到左	2
*=	乘赋值	双目	中置	右到左	2
/=	除赋值	双目	中置	右到左	2
%=	求余赋值	双目	中置	右到左	2
%	求余	双目	中置	左到右	3
*	乘	双目	中置	左到右	3
/	除	双目	中置	左到右	3
+	加	双目	中置	左到右	4
-	减	双目	中置	左到右	4
=	赋值	双目	中置	右到左	5

12）参考表 2-6，解释以下概念：运算数的数目、运算数的位置、结合性和优先级？运算数的数目是指运算符需要的运算数的数目。双目运算符（像 /）需要两个运算数，而单目运算符（像 ++）只需要一个运算数。图 2-7 显示了这两个运算符的概念。

图 2-7 单目和双目运算符

运算数的位置是一个运算符相对于运算数的位置。对于单目运算符，如果运算符放到变量的前面，那么它的位置就是前置，如果运算符放到变量的后面，那么就是后置。对于双目运算符，它的位置是中置，因为它总是放在两个运算数的中间。例如，-x 中的负号就是前置，y++ 中的后自增运算符是后置的，而求余运算符的位置就是中置的。

结合性指的是相同优先级的运算符求值时的方向性。例如，运算符 + 和 - 有相同的优先级。它们的结合性都是从左到右。那么，1+2-3 以 (1+2)-3 运算而不是 1+(2-3) 的方式。这个概念在图 2-8 中解释。

图 2-8 结合性的含义

优先级指的是运算符和它们的运算数运算时的顺序。具有高优先级的运算符先计算，例如：* 比 - 有更高的优先级，所以 1-2*3 运算为 1-(2*3)，而不是 (1-2)*3。注意，在这个表达式中 - 号代表的是一个减号，优先级是 4。- 也可以被当成负号运算符，这时它是一个单目运算符，优先级是 2。例如，-2+3*4 运算为 (-2)+(3*4)，而不是 -2(+(3*4))。

13）什么是副作用？计算一个表达式的主要目的是得到这个表达式的值。除了这个目的之外，任何在计算表达式时发生的事情都可以被认为是副作用。例如，C 语句（假设 i 的初始值是 7）

```
j = i++;
```

的主要目的是计算出赋值语句右边的表达式的值为 7。而副作用是，i 的值被增加了 1（使得 i 现在等于 8）。考虑下面的 C 语句：

```
j = (i=4) + (k=3)-(m=2);
```

这个语句的主要目的是得到赋值语句右边的表达式的值为 5=（4+3-2）。三个副作用是

- i 等于 4
- k 等于 3
- m 等于 2

扩展解释

1）如果把一个整型数赋值给一个浮点数，会发生什么？C 语言在后面加上一个小数点，然后转换成浮点数的表达方式，并保存在一个变量的内存单元中。因此

```
p = 6 / 4;
```

接受一个整型数值（6/4，由于要截断结果的小数部分，所以最后结果等于 1），转换成 1.0（有小数点就意味着它是一个指数的二进制形式了），然后保存到变量 p 的内存单元中。

2）本课程没有在程序中演示，表达式 -6/5 的最后结果是什么？ANSI C 标准给出的结果是“实现的定义”。也就是说，ANSI C 没有硬性定义取整的方向性。所以，结果取决于你使用的编译器，也许是 -1，也许是 -2。注意，正确的结果是 -1.2。编译器可以选择向上取整或者向下取整。

3）C 语言做算术运算的方式如何影响你写自己的程序？

当你在程序中写算术语句的时候，需要注意以下几点：

- 当进行除法运算的时候，避免使用整型数的两个变量或常量，除非你真的想把小数点部分截断。记住，当使用除法的时候，检查运算数的类型。
- 在你的代码中，如果赋值语句的左边出现一个浮点类型的变量，为了安全起见，我们推荐你在赋值语句右边的常量中使用小数点。不用小数点有的时候也会得到正确的结果，不过，在你对 C 语言的混合类型运算的规则很熟悉之前，推荐加上小数点。
- 如果赋值语句的左边出现一个 int 类型的变量，保证赋值语句右边的表达式也会生成一个 int 类型的数。如果生成一个实型数，那么小数点的部分会被截断。
- 当你试图把一个实型变量赋给一个整型变量的时候，几乎所有现代的 C 语言编译器都会给出警告信息。因此可以通过查看是否有表达式和赋值语句的警告信息来保证一切都正确。

4）如果全部运算符都有相同的优先级，会发生什么？如果所有算术运算符有相同的优先级，那么最左边的运算符被首先执行。

5）可以在表达式中连续使用两个算术运算符吗？不能在算术语句中使用两个连续的运算符，除非使用括号。例如，x/-y 是不允许的（虽然一些编译器允许你这么做），但是 x/(-y) 是允许的。

6）转换运算符如何影响优先级？转换运算符要求放在括号内，拥有最高的优先级，所以会被首先执行。于是表达式

```
(float) aa / (float) bb
```

等同于

```
((float) aa) / ((float)bb)
```

因为转换运算符比除法运算符有更高的优先级。

7）转换运算符如何影响图 2-6 中演示的 C 语言中的算术运算规则？对 C 语言中的算术运算规则的影响表现在，有的时候并不需要大量地使用转换运算符。例如有以下的声明

```
int a, b;
float x;
```

语句 x = (float)a+b; 和 x = (float)a +(float)b; 会给出相同的结果。这是因为，第一个语句中，把 float 和 int 类型相加的时候，C 语言会生成 int 的一个 float 类型的拷贝，然后把两个 float 类型进行相加。

8）副作用有什么危险？目前副作用有点令人迷惑，对于语句

```
k = (k=4) * (j=3);
```

最后的结果会是 12，而不是 4。最好不要写这种带有副作用的语句。除非使用它们的最简单的形式，如：

```
i = j++;
```

或者

```
i = j = k = 5;
```

注意，因为赋值运算符的结合性是从右到左的，上面那种复合赋值语句可以依次写成下面的语句，操作的顺序是

```
①k = 5
②j = k
③i = j
```

同样，表达式

```
i = j = k = 2 + n + 1;
```

按下面的顺序进行运算：

```
①k = 2 + n + 1;
②j = k;
③i = j;
```

这是因为加法运算符比赋值运算符有更高的优先级。

概念回顾

1）变量可以在声明的时候初始化。例如

```
float a=7, b=6;
```

2）自增和自减运算符可以被分为前自增/自减运算符和后自增/自减运算符两种。

- 如果自增运算符（++）或者自减运算符（--）放到了一个变量的前面，变量首先被增加或减少1，然后把这个变量的新值用到其所在的表达式中。
- 如果自增运算符（++）或者自减运算符（--）放到了一个变量的后面，这个变量的值首先用到其所在的表达式中，然后变量被增加或减少1。

3）C的转换运算符形式如下：

```
(type) expression
```

其中，把expression的值暂时转换为type的类型。转换运算符用在涉及不同数据类型的操作中，以确保最后的精度不会丢失。

4）复合赋值运算符+=、-=、*=、/=、%=执行一个算术操作和一个赋值操作，被简写成x op = y，等价于x = x op y。

例如，x+ 2在运行的时候执行x = x+2。

5）运算符的优先级指定了运算符和运算数执行操作的顺序。高优先级的运算符首先被运算。当运算符的优先级和结合性一起考虑时，将决定一个复杂表达式的运算顺序。优先级和结合性见表2-6。

6）副作用表述了在一个语句中执行了多余一个操作的情况，这种情况是由使用了一些类似的自增运算符引起的。例如，i = j++; 中j被增加了1，这就是赋值语句i =j之外的副作用。

练习

1. 基于

```
int i=10, j=20, k, m,n;
float  a,b,c,d,e=12.0;
```

确定下面语句是真是假：

a. i -=2; 是一个合法的语句

b. i %=a; 是一个合法的语句

c. i *=(i+j*a/123.45); 是一个合法的语句

d. 执行k=i/j;，k等于0.5

e. 执行i+=j;，i等于30，j等于20

f. k=1/3+1/3+1/3; 等于1

g. d=1/3.+1.0/3+1.0/3 等于1.0

h. a=1/3+1/3+1/3 等于1.0

i. a=1./3+1/3.+1.0/3.0 等于1.0

j. i+2/3*3/2 等于11

k. i+3/2*2/3 等于10

l. ++i=j; i 等于21

m. ++i++; i 等于31

2. 把下面的公式换成C语言的表达式。

a. $1+\frac{1}{3}+\frac{1}{5}+\frac{1}{7}$

$1+\frac{1}{2.0}+\frac{1}{3.0}+\frac{1}{4.0}$

b. $\frac{\pi R^2}{\frac{(a+b)^2}{(a-b)^3}}$

c. $a+\frac{\frac{b}{c}}{d+\frac{e}{\frac{f}{gh^2}}}$

3. 假设a、b、c是int变量，并且有下列的值：a=10，b=20，c=30。在下面每一列的语句后面分别计算a、b、c的值。

```
a.a=c;      b.a=c++;      c.a=++c;      d.a+=c;
  b=a;        b=a++;        b=++a;        b/=a;
  c=b;        c=b++;        c=++b;        c%=b;
```

4. 假设a、b、c是上述定义的变量。写一个程序交换它们的值，使得a有c的值，b有a的值，c有b的值。

5. 手工计算下面程序中x、y和z的值，并且运行程序检查你的结果。

```
#include <stdio.h>
void main(void)
{
      float a=2.5,b=2,c=3,d=4,e=5,x,y,z;
      x=  a *  b -  c + d  /e ;
      y=  a * (b -  c)+ d  /e ;
      z=  a * (b - (c + d) /e) ;
      printf("x= %10.3f, y= %10.3f, z=%10.3f",x,y,z);
}
```

6. 计算下面每一个算术表达式：

```
13/36, 36/4.5, 3.1*4, 3-2.6, 12%5, 32%7
```

答案

1. a. 假 b. 假 c. 真 d. 假 e. 真 f. 假 g. 假 h. 假 i. 真 j. 假 k. 真 l. 假 m. 假

3. a. 30，30，30

b. 31，31，30

c. 32，33，33

d. 程序因为除以零而崩溃

6. 0，8.0，12.4，0.4，2，4

本章回顾

本章中学习了如何使用格式控制符控制程序中变量的输出，讨论了如何在程序中声明变量，以及如何使用算术运算符处理数据。然后使用scanf从键盘读入数值，并利用printf把这些变量打印到屏幕上。最后学习了C语言表达式中相关的数学运算问题。

现在你可以用本章学到的知识来写一个执行复杂科学运算任务的程序了。

C 语言基础：数学函数和字符文件输入输出

本章目标

结束本章的学习后，你将可以：

- 使用 C 数学库中的函数。
- 从用户输入中读入字符。
- 执行文件输入 / 输出操作。
- 基于给定的问题描述，构建工作程序。

为了让计算机程序有用，它必须有一些函数用来执行计算，并且能对用户的输入及时地反馈。

本章要学习如何使用各种不同的数学函数来执行复杂的计算任务。同时学习处理字符输入的函数。另外，你会学习如何从计算机硬盘或其他的外部设备上的文件读入 / 写出数据。这是除了使用键盘和屏幕以外另外的一种处理数据的方法。使用文件是一种重要的方法，因为它提供了一种永久保存数据和程序结果的方法。否则，所用的数据在电脑关闭或断电的时候就会丢失。

课程 3.1　数学库函数

主题

- 使用标准的数学头文件
- double 和 float 数据类型的不同
- 其他数据类型

计算器会让你通过它的一个按键来轻松地执行诸如 sin、log 或开平方的操作。同样，C 语言编译器通过提供可以在程序中使用的数学库函数来执行这些操作。本课会演示如何使用这些库函数。C 语言编译器在库中包含这些函数，但是为了高效地使用它们，你需要告诉编译器必要的信息。

虽然没有在源代码中显示，但是你要知道，C 语言中除了 int、float、double 以外，还有其他的数值数据类型。它们也可以用 2 的补码或者科学计数法来表示。本课中讨论的数据类型占据了不同数量的内存。有的时候，其他的数值类型对我们也很有用。本课程演示了 double 和 float 之间的区别。

源代码

```
#include <math.h>  ← 包含用于数学函数的头文件
#include <stdio.h>
void main(void)
{
        double x=3.0, y=4.0, a,b,c,d,e,f;
        float g;
```

```
    a=sin(x);
    b=exp(x);
    c=log(x);

    d=sqrt(x);
    e=pow(x,y);
    f=sin(y)+exp(y)-log10(y)*sqrt(y)/pow(3.2,4.4);
    g=log(x);

    printf("x=%4.1f    y=%4.1f \n\n\r\
           a=sin(x)    = %11.4f\n\r\
           b=exp(x)    = %11.4f\n\r\
           c=log(x)    = %11.9f\n\n\r\
           d=sqrt(x)   = %11.4f\n\r\
           e=pow(x,y)  = %11.4f\n\r\
           f=sin(y)+exp(y)log10(y)*sqrt(y)/pow(3.2,4.4)= %11.4f\n\n\r\
           g=log(x)    = %11.9f\n",x,y, a,b,c,d,e,f,g);
}
```

在表达式中使用的标准的C数学函数

输出

```
x= 3.0  y= 4.0

a=sin(x)     =      0.1411
b=exp(x)     =     20.0855
c=log(x)     = 1.098612289

d=sqrt(x)    =      1.7321
e=pow(x,y)   =     81.0000
f=sin(y)+exp(y)-log10(y)*sqrt(y)/pow(3.2,4.4)=   53.8341

g=log(x)     = 1.098612309
```

解释

1）double 和 float 之间有哪些不同？我们已经在课程 2.3 中介绍过了 double 类型。事实上，这两种数据类型都以指数二进制的形式来保存数值。但是 double 比 float 占据更多的内存，这是使用 double 变量的一个缺点。什么时候需要使用一个更多位数的数字呢？在执行大数操作的时候，使用一个更多位数的数字就很重要了。

下面的例子演示了位数在计算中的效果。你可以先在计算器上试验一下。假设你把一个数，例如 π 乘 100 次，实际上是在计算 π^{100}，此时有效位对计算的影响如下。假设你使用五位有效位，那么结果是：

```
(3.1416)^100 = 5.189061599 * 10^49
```

当使用八位有效位的时候，结果是：

```
(3.1415926)^100 = 5.1897839464 * 10^49
```

第一个结果使用了五位有效位，但是仅仅前三位数字是对的。这演示了当执行大量数学运算后，精度会降低。因为计算机会轻易地执行 1 百万次的操作，所以你现在理解了更多的位数是多么的重要了。

我们可以从本程序中计算 log(3) 的值中看出差别。利用 double，自然对数是

```
c=log(3)    = 1.098612289
```

而使用 float 的时候，我们得到

```
g=log(3)    = 1.098612309
```

如果和计算器的结果比较，你会发现 double 类型的值是更准确的。你应该意识到计算器其实有 12 或更多的位。而 float 仅仅有六位有效位。因此如果你想让计算程序和计算器一样地准确，那么就不能使用 float。

2）C 语言中还有没有其他的数据类型用来保存实型数？有，除了 float、double、long double 也可用于保存实型数。long double 比 double 占据更多的内存。这意味着 long double 有更多的有效位，并比 float 和 double 保存更大的数。

float、double 和 long double 在表 3-1 中进行了比较，ANSI C 标准中并没有特别指定每一种类型需要占据多少内存。所以表中只是通常使用的标准。你可以通过阅读编译器手册来决定每一种类型到底占据了多少内存。为了检查编译器，你可以查看 float.h 头文件。其中有一个常数宏 DBL_MAX_10_EXP，也许它会给出 308 这个数，这代表 double 所能容纳的最大的指数。

表 3-1 数据类型

项	float	double	long double
内存使用	4 字节 =32 位	8 字节 =64 位	10 字节 =80 位
值的范围	1.17549435E-38 到 3.40282346E+38	2.22507385E-308 到 1.79769313E+308	大约 1.0E-4931 到 1.0E-4932
精度	6	15	19
显示形式	%f%e,%E,%g,%G	%lf,%e,%E,%g,%G	%Lf,%Le,%LE,%Lg,%LG

由于 float 类型的精度比较低，我们推荐你在程序中使用 double 数据类型。注意，最简单的 double 数据类型的格式是 %lf。这个格式在本书中被广泛使用。在使用 printf 函数的时候，%f 也是可以接受的，但是对于 scanf 函数，一定不要使用 %f 格式。scanf 函数中一定对于 double 使用 %lf，且对于 long double 使用 %Lf。

3）有不同的整型数据类型吗？是的，不同的整型数据类型如表 3-2 所示。

表 3-2 整型数据类型

	short int		int、long int	
项	signed short int	unsigned short int	signed int, signed long int	unsigned int, unsigned long int
内存使用	2 字节 =16 位	2 字节 =16 位	4 字节 =32 位	4 字节 =32 位
值的范围	−32768 到 32767	0 到 65535	−2147483648 到 −2147483647	0 到 4294967295
显示形式	%d，也许需要用 %hd 表示 short int	%d，也许需要用 %hd 表示 unsigned short int	%ld	%ld

4）经常使用 long int 或者 unsigned long int 吗？通常不会，经常在程序中使用 int 数据类型来计数。如果你不涉及一个很大的数，int 通常都是够用的。但是你必须意识到 int 在某些机器的的界限大约在 32 000。本书的后面会演示一个使用 unsigned long int 的例子。

5）本课程中的函数是什么含义？C 数学库函数的意义在表 3-3 中给出（注意这些函数的输入参数和返回类型都是 double 数据类型）。注意 sin 函数的参数（类似还有 tan、cos 等用角度作为输入函数）要求输入的是弧度的值，而不是角度值。所以，如果你想在程序中计算 30 度的正弦值，应手工写代码，通过乘以 π/180 来把角度转换为弧度。例如，代码可能如下：

```
angle = 30.;
x = angle * 3.141592654/180.;
y = sin(x);
```

表 3-3 常用 C 语言数学函数

函数名字	描述	函数名字	描述
sin(x)	计算 x 的正弦值，x 为弧度	sqrt(x)	计算 x 的平方根
exp(x)	计算 x 的自然指数	pow(x,y)	计算 x 的 y 次幂
log(x)	计算 x 的自然对数		

同时应该使用有更多有效位的数据类型来保证计算的精度。

其他的 C 数学函数会接受不同的数据作为输入，并返回不同的数据作为结果。通常 C 数学库函数会随着不同的编译器而有些差别。检查你的编译器来获得更多的细节。表 3-4 给出了更多的 C 语言的数学函数。

表 3-4 更多的数学库函数

函数名字	例子	描述
abs(x)	y=abs(x);	计算 int 类型变量 x 的绝对值，y 也为 int 类型
fabs(x)	y=fabs(x);	计算 double 类型变量 x 的绝对值，y 也为 double 类型
sin(x)	y=sin(x);	计算弧度单位的角 x 的正弦值，x 和 y 为 double 类型
sinh(x)	y=sinh(x);	计算弧度单位的角 x 的双曲正弦值，x 和 y 为 double 类型
cos(x)	y=cos(x);	计算弧度单位的角 x 的余弦值，x 和 y 为 double 类型
cosh(x)	y=cosh(x);	计算弧度单位的角 x 的双曲余弦值，x 和 y 为 double 类型
tan(x)	y=tan(x);	计算弧度单位的角 x 的正切值，x 和 y 为 double 类型
tanh(x)	y=tanh(x);	计算弧度单位的角 x 的双曲正切值，x 和 y 为 double 类型
log(x)	y=log(x);	计算 x 的自然对数值，x 和 y 为 double 类型
log10(x)	y=log10(x);	计算 x 的以 10 为底的对数值，x 和 y 为 double 类型

6）当使用数学库函数的时候，应该包含那个头文件？为了使用这些库函数，必须包含以下头文件：

```
#include <math.h>
```

概念回顾

1）doube 数据类型比 float 数据类型占据多一倍的内存空间。double 有更高的精度（有效位）和更大的存储范围（见表 3-1）。在使用 double 的时候，使用 %lf。

2）long double 在需要更大的精度和存储范围的时候使用。在使用 long double 的时候，使用 %Lf。

3）C 语言中一共有 6 种不同的 int 数据类型（见表 3-2）。通常在大部分的操作中使用 int（有符号位）。

4）C 在数学库函数中提供了大量的有用的数学函数，在使用这些函数之前，需要在文件的开头包含下面的头文件：

```
#include <math.h>
```

比较常见的数学函数见表 3-3 和表 3-4。

练习

1. 判断真假：

a. #include<Math.h> 是一个正确的 C 预处理指令。

b. 头文件必须放到C文件的开头。

c. 在C语言中，sin(30) 等于0.5。

d. 在C语言中，log(100) 等于2.0。

答案

1. a. 假　　b. 真　　c. 假　　d. 假

课程 3.2　单个字符数据

主题

- 字符集
- 单个字符的输入和输出
- 把字符当成一个整型数
- 字符输入和输出函数
- 输入缓冲区
- 清空输入缓冲区

单个字符类型 char，有些时候可以像数字类型（int、float、double）一样使用。术语字符并不仅仅代表大写和小写的字母，还有一些图形符号，例如 !,#,^，转义字符 \n 和 \r 等，也被认为是单个的字符。

甚至于数字0到9也可以被当成是字符。当我们想保存电话号码的时候，把它们当成字符明显更方便。电话号码并不是执行算术操作的数字（把两个电话号码相加没有什么意义）。正因为这样，如果把电话号码保存成整数会难以处理。所以我们发现把那些不需要执行算术操作的数字保存成字符会更方便一些。

本课程中的程序使用了11个字符。这个程序只是简单地读入并打印字符。

源代码

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

void main (void)
{
     char  c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11;

/*************************************************************
**          SECTION 1
*************************************************************/

     printf("\n***SECTION 1*****************************************\n");

     c1 = 'g';
     c2 = '<';
     c3 = '\n';
     printf ("%c %c\n", c1, c2);
     putchar(c1);
     putchar(32) ;
     putchar(c2) ;
     putchar(c3);
```

函数 getche 的头文件，这不是 ANSI C 的头文件。我们只是演示 ANSI C 之外的函数能做什么，这是本书中为数不多这么做的地方

声明字符变量

把字符赋值给字符变量

利用 printf 打印单个或者更多字符，注意转换的规则

利用 putchar 每次打印一个字符，注意"空格"和"回车"也是字符

```
    printf("\n***SECTION 2 ************************************\n");

/*************************************************************
**          SECTION 2
*************************************************************/

    printf ("Enter two characters (without spaces), then press return:\n");
    scanf  ("%c%c", &c4, &c5);
    putchar(c4) ;
    putchar(c5);
    printf ("\n%d %d\n", c4, c5);

    printf("\n***SECTION 3 ************************************\n");
/*************************************************************
**          SECTION 3
*************************************************************/

    printf ("Enter two more characters (without spaces), then press return\n");
    getchar();
    c6 = getchar();
    c7 = getchar();
    putchar(c6);
    putchar(c7);
    putchar('\n') ;

    printf("\n***SECTION 4 ************************************\n");
/*************************************************************
**          SECTION 4
*************************************************************/

    printf ("Enter two more characters (without spaces) then press return:\n");
    fflush(stdin);
    c8 = getchar();
    c9 = getchar()
    fflush(stdin);
    printf ("%c%c \n", c8, c9);

    printf("\n***SECTION 5 ************************************\n");
/*************************************************************
**          SECTION 5
*************************************************************/


    printf ("Enter two more characters (without spaces) DO NOT press return:\n");
    c10 = getche();
    c11 = getche();
    putchar('\n');
    putchar(c10) ;
    putchar(c11);

}
```

利用 scanf 从键盘读入字符。注意在文本字符串中没有空格

从键盘读入字符，但是并没有保存在任何变量中

使用 getchar 函数读入键盘输入的单个字符，注意没有参数列表（所以括号里面是空的）。赋值语句使得输入的字符保存在为 c6 和 c7 预留出的内存单元内

fflush 函数清空输入缓冲区，这使得接下来的两个输入的字符可以正确地保存在 c8 和 c9 两个变量中

getche 函数与 getchar 操作类似，但是它避免了输入缓冲区的问题。因为它不是 ANSI C，所以在你的编译器上可能不工作。这里演示的目的就是告诉你可能会遇到一些非 ANSI C 的函数

输出

```
***SECTION 1*************************************
g <
g <

***SECTION 2 ************************************
Enter two characters (without spaces), then press return:
xpreturn
xp
120 112
```

键盘输入

```
                ***SECTION 3 ****************************************
                Enter two more characters (without spaces), then press return
键盘输入        qkreturn
                qk

                ***SECTION 4 ****************************************
                Enter two more characters (without spaces), then press return
键盘输入        vsreturn
                vs

                ***SECTION 5 ****************************************
                Enter two more characters (without spaces), DO NOT press return
键盘输入        wq
                wq
```

解释

1）如何声明字符变量？字符变量使用关键字 char 来声明，形式如下：

```
char variable1, variable2, variable3,...;
```

例如，本课程程序中的声明语句

```
char c1, c2, c3, c4, c5, c6, c7, c8, c9;
```

声明了 c1、c2、c3、c4、c5、c6、c7、c8、c9 为字符变量。

2）如何对这些字符变量写赋值语句？为了把字符值赋给字符变量，需要把字符值（常量）包含在单引号中。例如，赋值语句

```
c1 = 'g';
```

把字符 g 赋给变量 c1，一个常见的错误是使用双引号而不是单引号。不要犯这个错误。为了正确处理单个字符和字符函数，必须使用单引号。

3）什么是 ANSI C 的字符常量，C 语言如何处理它们？一个完整的 ANSI C 字符常量和它们的 ASCII（American Standard Code for Information Interchange，ASCII 字母表用来在计算机内部对不同的符号进行编码）十进制码值在表 3-5 中给出。注意这个表中并没有包含所有的 ASCII 字符集。所以你会发现表中并没有包含诸如 $ 和 α 的值。这样就带来了一个问题，下面这个语句

表 3-5 ANSI C 字符和它们的 ASCII 码值（十进制）

字符	ASCII 值	字符	ASCII 值	字符	ASCII 值
\a	7	<	60	_	95
\b	8	=	61	`	96
\t	9	>	62	a	97
\n	10	?	63	b	98
\v	11	A	65	c	99
\f	12	B	66	d	100
\r	13	C	67	e	101
space	32	D	68	f	102
!	33	E	69	g	103
"	34	F	70	h	104
#	35	G	71	i	105

（续）

字符	ASCII 值	字符	ASCII 值	字符	ASCII 值
%	37	H	72	j	106
&	38	I	73	k	107
'	39	J	74	l	108
(	40	K	75	m	109
)	41	L	76	n	110
*	42	M	77	o	111
+	43	N	78	p	112
,	44	O	79	q	113
-	45	P	80	r	114
.	46	Q	81	s	115
/	47	R	82	t	116
0	48	S	83	u	117
1	49	T	84	v	118
2	50	U	85	w	119
3	51	V	86	x	120
4	52	W	87	y	121
5	53	X	88	z	122
6	54	Y	89	{	123
7	55	Z	90	|	124
8	56	[	91	}	125
9	57	\	92	~	126
:	58	]	93		
;	59	^	94		

```
printf("The first Greek character is %c\n", 'α');
```

有可能被编译，也有可能不被编译。那些超出 ANSI C 标准的编译器可能会编译并运行这个语句。即使有的编译器可以编译，还是推荐你不要在源码中使用这些字符，这样可以使得你的程序更加具有移植性。

C 实质上在字符函数和字符操作中使用字符的整型值。就像接下来会看到的，这允许我们在某些操作中使用整型值来代替字符。

4）为什么转义字符（诸如 \n 和 \r）也被包含在字符集中？ C 语言把转义字符当成一个单独的字符。当你使用它们的时候，确保在 \ 的后面不要有空格，因为它会被认为是一个单独的字符。赋值语句 c3 = '\n' 是合法的。

5）如何使用 printf 函数来打印字符？我们使用 %c 转换标准，就像处理整型和浮点型变量那样处理字符型变量。例如，为了打印字符变量 c1 和 c2，语句

```
printf ("%c %c \n", c1, c2);
```

会完成任务。但是因为 C 语言把单个的字符当成整型数，如果使用 %d 转换格式，字符的整数值会显示出来，例如

```
printf ("\n%d %d\n", c4, c5);
```

会把 c4 和 c5 的整型数值打印出来。如果给定 c4 为字符 x，而 c5 为字符 p，它们的 ASCII

码值分别为 120 和 112，这两个值会被打印出来。

6）函数 putchar 是如何工作的？函数 putchar 函数在标准输出设备上打印或显示通过参数传入的字符。格式如下：putchar (character)；例如，语句 putchar(c2)；把字符变量 c2 的值打印到屏幕。putchar 函数也可以打印字符常量。例如，语句

```
putchar ('y');
```

会把字符 y 打印到屏幕上。

7）函数 putchar(32) 的作用是什么？因为 C 语言在函数中使用字符的整数值，所以 putchar(32) 会把 32 所代表的那个字符打印到屏幕上。在表 3-5 中可以看出 32 代表的是字符空格。这意味着一个空格会被打印到屏幕上。

因此，printf 语句

```
printf ("%c %c\n", c1, c2);
```

在两个 %c 之间有一个空格，并且末尾有一个 \n。它和下面的 putchar 语句

```
putchar(c1);
putchar(32);
putchar(c2);
putchar(10);
```

执行相同的任务。

8）为了打印单独的字符，应该使用 printf 函数还是 putchar 函数？两个都可以，但是很多程序员使用 putchar 函数，因为它是专门为打印字符而设计的，在打印单个字符方面它更有效率。

9）怎样使用 scanf 函数从键盘输入单个的字符？我们使用 %c 转换标准。于是，

```
scanf ("%c%c", &c4, &c5);
```

从键盘读入两个字符，并且把它们保存在为 c4 和 c5 变量分配的内存单元中。注意在字符串文本中间没有空格。后续我们会更仔细地研究 scanf 函数，并解释为什么在字符串文本中有一个空格会造成读入单个字符的困难。

10）getchar 函数如何工作？它返回从标准输入设备（键盘）读入的一个字符。格式如下：

```
getchar();
```

括号中为空。这个函数经常放到一个赋值语句的右边，例如

```
c6 = getchar();
```

从键盘读入输入的一个字符，然后把这个字符赋值给 c6。这和赋值语句 y=log(x) 是一样的。不同的是 getchar 函数不需要传入参数，因为没有信息需要传入 getchar 函数。

11）getchar 直接从键盘得到输入的字符吗？不是，getchar 函数和输入缓冲区一起工作来得到从键盘输入的各种信息。

12）什么是输入缓冲区，它如何和 getchar 一起工作？缓冲区是一部分内存，它用来暂时保存将要被传送的信息。这个内存被顺序地读取，也就是说，读完一个内存单元，然后再去读下一个内存单元。一个位置指示器来追踪读到的具体的位置。每读一个单元，计数器加 1，以便下一个内存单元被读取。

getchar 函数利用缓冲区内的位置指示器来从缓冲区内取得下一个字符，并递增位置指示器。当 getchar 函数被调用，它从缓冲区的下一个单元读入字符；如果下一个单元为空，那么就暂停程序的运行，等待用户的输入。这个时候字符可以被输入。当你按下 return 或者

enter 键的时候，getchar 被重新激活。

换句话说，当一个键被敲击，缓冲区得到并保存你敲击的那个键的值。这个过程会一直持续直到你按下了 enter 键。然后 getchar 函数会从缓冲区内取得一个字符（或者下个字符，如果你持续地使用 getchar 函数）。

使用 getchar 函数的一个困难在于：getchar 函数是由按下 return 或 enter 键所激活的。例如，对于本课程的程序，最初输入的几个键如下：

xp$_{return}$

这意味着缓冲区如下所示：

scanf 函数已经读取了 x 和 p，所以位置指示器如本课程序的 Secton 2 的位置。

然后在 Section 3 部分，执行下面的语句：

```
getchar();
c6 = getchar();
c7 = getchar();
```

运行以下过程：

①语句 getchar 被执行。程序从缓冲区读入 \n，但是这个字符并没有被保存在任何变量中，因为这个语句并不是一个赋值语句。这个语句的唯一目的就是使得缓冲区内的位置指针前进一格。这是因为当读入一个字符的时候，位置指针会前进一格，达到下一个位置。

②因为位置指示器所指的缓冲区为空，所以 c6 = getchar();，使得程序暂停下来，等待用户从键盘输入。用户输入 vs 使得缓冲区如下：

然后用户按下了 enter，这带来了两个效果：\n 被输入到缓冲区内，同时程序的运行被重新激活。

③ c6=getchar; 和 c7=getchar(); 这两个语句被执行（因为相应的缓冲区位置不空）。缓冲区和位置指针如下：

13）总体来说，使用 getchar 函数的困难在哪里？因为 getchar 函数需要 enter 来配合工作，这样 \n 会被输入到缓冲区，并且你必须要处理掉它。另外缓冲区可能包含多余的其他字符。例如，如果用户被提示输入两个字符，他一不小心输入了三个字符，然后按下了回车。第三个字符会在缓冲区内而不会被丢弃掉。这样下一个 getchar 会读入那个多输入的字符，使得整个程序的运行和你期望的不一样。

14）如何处理缓冲区内多余的字符？当我们得到想要的字符以后，可以清空缓冲区。这可以通过使用 fflush 函数来完成。为了清空输入缓冲区，格式如下：

```
fflush(stdin);
```

在本课程中，语句

```
fflush(stdin);
c8 = getchar();
c9 = getchar();
fflush(stdin);
```

在两个 getchar 函数调用之前或之后，清空了输入的缓冲区。如果在 c8 的赋值语句之前没有清空缓冲区，那么 c8 可能被赋值为一个 \n，这是因为下一个字符可能就是 \n（取决于缓冲区内的位置指示器）。在两个语句后面执行 fflush 语句可以确保不读入多余的字符。

15）什么是 stdin？它和文件的输入输出有关，我们会在后面的课程中讨论。现在你需要知道 stdin 被定义在 <stdio.h> 这个头文件中，指向标准输入流。因此，fflush(stdin); 会清空 stdin 所指向的标准输入流——从键盘输入的输入缓冲区。

注意，因为 stdio 在 <stdio.h> 中定义，你不能在程序中使用 stdin 作为一个标识符的名字，如果这样做，编译器会报告错误。

把 stdin 当成一个标识符名字的错误很少见，因为 stdin 并没有明显的声明语句。本课程中，你会发现声明了所有源代码中使用的标识符。

16）为什么不能只使用 scanf 而不用 getchar？可以使用 scanf 函数，但是在处理字符类型的数据时会遇到另外的问题。在我们输入整型数的时候这个问题不用担心。现在，值得我们一起来讨论 scanf 这个函数如何工作了。

当 scanf 函数被激活的时候，暂停程序的执行，用户可以输入信息，然后传递到输入缓冲区。当用户按下 enter 时重新开始执行，并使得 scanf 从输入缓冲区读入数据。

就像我们已经看到的，scanf 基于字符串文本中给定的格式转换符来翻译和转换输入缓冲区内的信息。scanf 把字符串文本中的内容分成三类：

①转换格式限定符（以 % 开始）

②空白字符（对 scanf 来说，空白字符不仅包含空格，还包含 tab 和回车）

③非空白字符

例如，在本课程的程序中，scanf 函数调用如下：

```
scanf ("%c%c", &c4, &c5);
```

文本字符串有两个转换限定符，没有空白字符和非空白字符。这个字符串文本意味着两个输入缓冲区单元会被读入，然后转换成字符类型。

但是如果字符串文本中有空白字符，或者用户的输入中有空白字符，那么 scanf 函数在处理字符的时候就很会麻烦。这是因为，像我们已经讨论过的那样，空白字符也可以被当成一个字符。（在以前这不是问题，因为空白字符不能翻译成一个数值。）

当在文本字符串中有空格的时候，空格使得 scanf 会忽略掉在输入缓冲区内的字符而得到下一个非空白的字符。因此，当读入字符时，你必须确保在文本字符串中不会出现空白字符。否则，你不会从输入中读入任何的空白字符。

例如，scanf 语句

```
scanf ("%c_%c", &c1, &c2);
```

不会接受以下输入：

```
u u
```

其中 u 代表通过按 spacebar 而得到的字符。试一下，scanf 不会返回，即使你按下了很多 enter 键。因为 enter 也是一个空白字符。直到一个非空白被输入，scanf 函数才会继续。但是

```
scanf ("%c%c%c", &c1, &c2, &c3);
```

会使得 c1=u，c2=u，c3=u。

```
scanf ("%c_%c_", &c1, &c2);
```

也会造成同样的问题，因为在文本字符串的末尾有一个空白。另外，本例中，scanf 函数需要输入一个非空白字符（当两个已经被输入以后）才能得以继续执行。当你输入第三个非空白字符的时候，它会被放到输入缓冲区里，但是不会被 scanf 读取。但是，scanf 后续的语句会不小心地读入这个数据。

因为 \n 被认为是一个空白字符，你可能想到通过在文本字符串的前面放一个空格来使得 scanf 忽略掉输入缓冲区内的多余的 \n。我们并不推荐你这么做，因为这会造成其他的一些问题。

如果非空白字符被用在 scanf 的格式字符串中，那么 scanf 希望在给定的位置出现一个匹配的字符串。如果 scanf 得到了，那么这个字符被丢弃，如果没有得到，scanf 终止。例如：

```
scanf ("%c-%c", &c1, &c2);
```

期待下面的输入

```
B-2
```

使得 c1=B，c2=2。如果你输入了 B2，scanf 函数会终止这个程序。这个特性在你的代码中非常有欺骗性。例如

```
scanf ("%cd%c", &c1, &c2);
```

%cd 不是一个新的转换符，它只是说期望在两个字符输入的中间有一个字母 d。所以输入 3d2 会继续运行。

scanf 另外一个特性是允许一个未知的元素被读入并丢弃。如果你在 % 和格式限定字母中间使用星号，那么这个元素会被读入并丢弃掉。例如，

```
scanf ("%c%*c%c", &c1, &c2);
```

会读入 B#2，并且赋值 c1=B，c2=2，# 可以是任何字符，它被读入并丢弃。

17）当 scanf 处理数字类型的输入数据时，还有那些空白字符？当处理数字类型的数据时，输入缓冲区内的空白字符被不同的对待。对于数字类型的数据，一个或多个空格、tab 和回车被要求用来分隔输入的元素。例如，

```
scanf ("%d%d%f", &a1, &a2, &a3);
```

会接受输入

```
1 2

3.14159
```

使得 a1=1, a2=2, a3=3.1415926。输入

```
1 2  3.14159
```

会产生相同的结果。

扩展解释

1）putchar(32) 在所有的计算机上都会打印出一个空格吗？并不总是，只有那些使用 ASCII 编码的计算机（包含大部分个人电脑）会把 32 翻译成空格。所以，不同的电脑会给出不同的结果。为了使得你的程序更具有移植性，我们推荐你使用 putchar(' ') 而不是使用 putchar(32)。

2）有没有方法避免 getchar 函数带来的问题？有。它并不是 ANSC C 的一个标准，但是函数 getche 在很多编译器上都有，尤其是专门为个人电脑开发的编译器。这个函数并不工作在缓冲区上，也就是说，产生无缓冲的输入。它直接和操作系统打交道，并不需要通过按 enter 键传递字符码。因此，每次按键都会被立即响应，而不通过输入缓冲区。这样就不用担心多余的字符，也不需要清空缓冲。它的语法如下：

```
getche();
```

注意在括号之间是空白的。

更多的时候，getche 被用来生成和用户之间一个交互的字符输入。但是，如果使用 getche 函数，你也就丢失了程序的一些移植性，因为它并不是 ANSI C 兼容的。这里介绍它的目的是让你意识到有些非 ANSI C 函数可以在编译器上工作，如果你的麻主同意你使用它们，你可以使用。

3）关于字符型数据，还有更多的事情需要学习吗？是的，我们仅仅接触了字符输入的表面，第 7 章会更深入地介绍字符处理的其他方面。

概念回顾

1）字符变量如下声明：

```
char variable1, variable2, variable,...;
```

2）赋值语句可以用来修改字符变量，例如

```
c1 = 'g';
```

转换限定符 %c 被用于输入输出的格式控制中。

3）C 语言中，字符可以被当成一个整数使用，整数的值由字符的编码值决定。ASCII 编码系统被用在大部分个人电脑上。

4）putchar 函数把字符参数打印到标准输出设备（屏幕）上，格式如下：

```
putchar (character);
```

5）getchar 返回一个从标准输入设备读入的字符，格式如下：

```
getchar();
```

6）缓冲区是为了暂时保存需要传送的信息而分配的一块内存区域。位置指示器可以追踪信息的位置。每读入一个单元，位置指示器要前进一个单元，以便下一个单元可以被读取。

getchar 利用缓冲区内的位置指示器来读取其中的下一个字符，当 getchar 函数被调用的时候，它或者读取下一个字符，或者暂停程序的运行以等待用户的输入。此时，一个或多个字符可以被输入。当你按下 return 或者 enter 键，getcahr 函数重新被激活。

因为 getchar 函数需要 enter 键才能工作，所以一个多余的 \n 会输入到缓冲中，你必须处理它。这可以通过 fflush 函数来完成。为了清空输入缓冲区，格式如下：

```
fflush(stdin);
```

练习

1. 判断真假，假设

```
char c1, c2, c3, c4;
```

a. 语句 c1=g; 把字符 g 赋值给字符 c1。

b. 语句 putchar(2); 把数字 2 打印到屏幕。

c. 语句 getchar(c4); 把下一个输入的字符值给字符 c4。

d. 语句 c2=getchar(c4); 把下一个输入的字符值给字符 c2。

e. 语句 scanf("%c%c",c3,c4); 把下两个输入的字符值给字符 c3 和 c4。

2. 判断下面的语句是否有错，如果有错，指出它们。

```
a.fflush(input);
b.putchar(\t);
c.c4='$';
d.putchar(47);
e.getchar(c4);
f.scanf(" %c %c ", &c1,&c3);
g.printf("%c %c ",c3, c4);
```

3. 写一个程序读入下列字符串：

```
&gt 891><
-raw {[]}
```

并把它打印出如下：

```
{98we[-gt] -&rl<>}
```

4. 写一个程序显示下列输出：

```
ABCD$
 BCD$
  CD$
   D$
    $
```

5. 写一个程序，要求用户输入 0 到 255 之间的五个整数，然后把它们转换成字符，分别显示这些字符和对应的数字。

答案

1. 判断真假：

a. 假，应该是 c1='g'；

b. 假，应该是putchar('2');

c. 假，应该是c4=getchar();

d. 真

e. 假，应该是scanf("%c%c",&c3,&c4);

2. a. fflush(stdin);

b. putchar('\t');

c. 没错误，但是语句并不是ANSI C标准，不能用在所有的系统上。

d. 没错误，会显示7。

e. c4=getchar();

f. 没错误，但是在c1和c3被输入后，scanf会等待直到再键入一个非空白字符。

g. 没错误

课程 3.3 从文件读入数据

主题

- 打开和关闭文件
- 从一个文件读入数据
- 使用fscanf函数

如果你输入的数据很长，并且计划多次运行程序，那么从键盘输入数据是很不方便的。尤其是当你多次运行程序的时候，你只是想对输入的数据进行很小的更改。

例如，你的输入是每月的收入。每月重复输入相同的数字会很繁琐。建立一个包含收入数据的文件更方便。你的程序可以从文件中读取数据，而不需要从键盘输入。如果你想使用不同的数据来运行程序，只需要先修改你的数据文件，然后再运行程序就可以了。

文件是以电子格式保存的一个信息的集合。它可以包含你的个人数据、CIA的秘密文档或好莱坞的电影。文件被保存在外部设备上，像磁盘、CD盘、或者电脑的硬盘。与数字不同，例如11234，可以用尺寸和符号来定义。文件更复杂，包含更多的特性。例如，一个文件必须有一个名字，这样计算机才能够识别它。文件可以被打开用来读出（从里面得到数据）或写入（把数据保存在里面）。文件可以是文本格式，就像程序的源代码；也可以是二进制格式，就像程序的可执行代码。另外，文件需要暂时的存储区域，以便它可以被电脑的操作系统正确地载入。

本课程的程序演示了如何从一个输入文件读取输入。在这个程序中，文件名为c3_3.IN。你必须记住，当用编辑器生成输入文件的时候，给这个文件指定和代码中相同的名字。当执行程序的时候，电脑会搜索那个名字的文件并读入。如果文件不存在，程序将不会运行。

源代码

```
#include <stdio.h>
void main(void)
{
        double xx ;
        int   ii, kk;
        FILE *inptr;
```

变量声明

声明一个指针用于函数fopen和fscanf

```
inptr=fopen ("C3_3.IN","r");
fscanf(inptr,"%d",&ii);
fscanf(inptr,"%d %lf",&kk,&xx);

fclose(inptr);

printf("ii=%5d\nkk=%5d\nxx=%9.3lf\n",ii, kk, xx);
}
```

调用 fopen 函数允许程序存取磁盘文件 C3_3.IN

fscanf 函数与 scanf 函数以类似方式工作，但是它用了一个 inptr 文件指针

关闭 inptr 指针指向的文件

输入文件 C3_3.IN

```
36
123 456.78
```

输出

```
ii=  36
kk= 123
xx= 456.780
```

解释

1）经常使用什么函数从一个文件中读入数据？在 C 语言中，通常使用 fscanf 函数来从一个文件读入数据，fscanf 的语法如下：

```
fscanf(file_pointer, format_string, argument_list);
```

fscanf 函数从 file_pointer 指定的文件读入内容，根据 format_string 转化格式信息，把读入的内容放到 arugment_list 给出的地址中。例如，语句

```
fscanf(inptr, "%d %lf",&kk,&xx);
```

格式转换符为 "%d %lf"。文件指针 inptr 表示两个数值从 inptr 指定的文件中被读入。第一个是 int（根据 %d），第二个是 double（根据 %lf）。读入的值被放到了 argument_list 指定的 &kk 和 &xx 地址中。函数 scanf 和 fscanf 有密切的联系。我们已经在 scanf 函数中学习了格式字符串和地址列表，这部分知识可以用在 fscnaf 函数中，这里不再重复了。

2）什么是文件指针？指针会在第 7 章中讨论。现在你只需要记住指针是一个变量，其中保存的是一个地址，而不是诸如 int、float、double 那样的数值。（回忆在第 2 章 2.1 和 2.2 节中讨论的内容。）这个地址给出了存取保存在硬盘上的文件的钥匙。为了声明一个保存地址的变量——文件指针，我们必须在声明语句中以 FILE 开头，并且把 * 号放到变量名字的前面。例如，语句

```
FILE *inptr;
```

声明了一个文件指针变量 inptr。这意味着 C 语言希望一个地址被保存在这个单元内。

3）命名一个文件指针有哪些习惯？命名文件指针的习惯和命名其他的 C 语言标识符是一样的。合法的和不合法的命名如下：

合法：FILE *apple, *IBM93, *HP7475;

不合法：FILE *+apple, *93IBM, *75HP75;

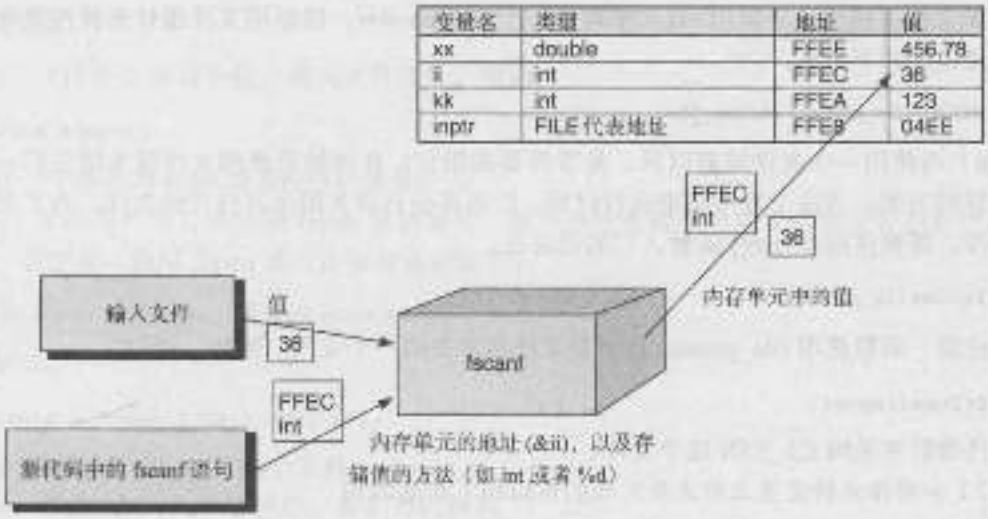


图 3-1 fscanf 函数的操作

4）声明一个文件指针后，如何使得文件可读？可以使用 C 库函数 fopen。这个函数允许我们把一个磁盘文件和一个文件指针联系起来。一旦这个联系被建立起来，我们就可以通过在程序中使用文件指针来读取和文件指针关联的文件了。

使用 fopen 建立连接的形式如下：

```
file_pointer = fopen (file_name, access_mode);
```

在这个语句中

```
inptr = fopen ("C3_3.IN","r");
```

被操作系统引用的文件名是 C3_3.IN。文件指针的名字是 inptr，access_mode 是 'r'，代表这个文件以读文本的模式被打开。你可以给 file_pointer 起任何合法的名字。注意 file_name 和 access_mode 都是字符串文本，必须包含在一对引号中。

注意到这个语句是一个赋值语句。在这里使用赋值语句是因为调用 fopen 会产生一个表达式。表达式的值是一个地址值，通过这个地址值就可以存取文件。于是，赋值语句把这个地址赋给 inptr，代表着 inptr 是一个地址。当赋值语句完成后，我们可以使用 inptr 来存储文件 C3_3.IN。

5）能更多介绍一下 FILE 吗？ FILE 是保持磁盘文件相关信息的数据类型。为了把这些特征保存在一个地方，C 语言发明了一种新的名为 FILE 的数据类型。这个数据类型和我们学过的 int 和 float 类似，但是要更复杂一些。

FILE 是定义在头文件 stdio.h 中的一个派生类型。这个头文件必须包含在程序的开头，以便你可以在程序中使用 FILE。如果不包含这个头文件，C 编译器无法理解 FILE 是什么含义，就会产生错误。

当你想操作一个磁盘文件的时候，可以用 C 数据类型 FILE 来声明一个 file_pointer，然后使用 file_pointer 来处理文件。这意味着在 C 数据类型 FILE 和实际的文件之间没有直接联系。你不能使用下面的语句

```
FILE "C3_3.IN";
```

来声明文件，相反，必须用 FILE 来声明一个 file_pointer，然后用文件指针来操控文件。整个处理过程如下所示：

```
FILE → file_pointer → actual_file
```

6）当使用一个文件结束以后，是否需要关闭它？在你结束使用文件后关闭它们，这是一个好的习惯。当这个程序结束运行以后，C 语言会自动关闭所有打开的文件。为了手工关闭文件，需要使用 fclose() 函数。它的语法是：

```
fclose(file_pointer);
```

注意，函数使用 file_pointer 而不是文件名来关闭一个文件。例如，语句

```
fclose(inptr);
```

用文件指针来关闭 C3_3.IN 这个文件。

7）如何指定特定盘上的文件？我们可以用下面的语句

```
inptr = fopen ("C:\\C3_3.IN", "r");
```

指定 C 盘上的文件 C3_3.IN。

观察到语句里有两个反斜杠，这是因为在字符文本中，反斜杠有特殊的含义，正如我们在第 1 章课程 1.7 中描述的那样。因此在那些只需要一个反斜杠的地方其实需要两个反斜杠。

8）scanf 和 fscanf 函数之间的关系是什么？ fscanf 函数和 scanf 函数几乎一样，除了输入流不同（从一个特定的文件而不是标准输入流）。我们介绍过的所有 scanf 的内容都适用于 fscanf。这样，你就会更好地理解输入文件如何被 fscanf 函数翻译成数字和字符输入。例如，你可以看到，没必要把所有的数字输入放到同一行。fscanf 函数会去下一行去读取更多的非空白字符。

9）为什么介绍 scanf 和 fscanf 函数如此多的细节？我们介绍如此多的细节是因为，你应该准确知道程序得到的信息，这是非常重要的。输入数据和 scanf 语句之间的不匹配是一个非常常见的错误，尤其对一个新手程序员来说。很多情况下，程序的其他部分完全正确，而错误只发生在输入部分。所以当你调试程序的时候，不要忘记检查输入数据以及 scanf 和 fscanf 语句。

回忆一下前两次课的声明，

```
FILE *inptr;
FILE *myfile;
```

我们用 inptr 作为文件指针指向一个生成的输入文件，用 myfile 文件指针指向我们生成的另一个用于输出的文件。stdin 在程序中并没有被显式声明，但是它同样是一个文件指针。它被声明在 stdio.h 头文件中，已经通过预编译指令 #include<stdio.h> 包含在程序中了。在 stdio.h 头文件中，stdin 被定义成指向一个标准输入流的指针。因此

```
fflush(stdin);
```

会把 stdin 指向的缓冲区清空，也就是标准输入设备——键盘所指向的输入缓冲区清空。我们也可以把其他的文件指针名字作为参数传递给 fflush。但是目前，暂时不介绍这么做会带来哪些效果。

概念回顾

1）文件在C语言中被声明为文件指针。例如

```
FILE *inptr;
```

声明了一个名字为inptr的文件指针变量。

2）文件指针用C库函数fopen来初始化，这个函数在磁盘上的文件和文件指针之间建立起一种连接。使用fopen建立连接的格式如下：

```
file_pointer = fopen (file_name,access_mode);
```

例如，

```
inptr = fopen ("C3_1.IN","r");
```

会以读模式打开C3_1.IN这个文件。

3）当成功打开一个文件后，我们可以使用

```
fscanf(file_pointer, format_string,variable_list);
```

来读入数据。所有scanf中的格式特性都可以用于fscanf函数中。

一旦结束了操作，可以使用fclose命令来关闭文件：

```
fclose(file_pointer);
```

练习

1. 判断真假：
   a. 使用fscanf从键盘读入输入。
   b. 使用fscanf从文件读入输入。
   c. 在读入输入数据之前，必须在外部磁盘和文件指针之间建立一个连接。
   d. 当你不再需要存取文件的时候，最好把它关闭。
   e. 文件指针是int类型，并可以被声明为其他的int类型变量。
2. 判断下面的语句是否有错，如果有错请指出。
   a. #INCLUDE <stdio.h>
   b. file myfile;
   c. *myfile = fopen (C3_1.DAT,r);
   d. fscanf("myfile","%4d %5d\n",WEEK,YEAR);
   e. close("myfile");
3. 写程序从一个叫做GRADE.REP的文件中读入你上学期的成绩，文件中仅有包含四个数据的一行（没有字符），例如

```
4.0 3.3 2.7 3.7
```

计算平均GPA并将结果输出到屏幕。

答案

1. a. 假　b. 真　c. 真　d. 真　e. 假

2. a. #include <stdio.h>
   b. FILE *myfile;
   c. myfile = fopen ("C7_1.DAT","r");
   d. fscanf(myfile,"%4d %5d\n",&WEEK,&YEAR);
   e. fclose(myfile);

课程 3.4 输出到文件

主题

- 向文件写入数据
- 使用 fprintf 函数

以前的程序把所有的输出都显示在屏幕上。有的时候这很方便；但是一旦屏幕滚动或者清空，输出就丢失了。

大部分情况下，你想长期保存输出，这可以通过把输出写入到文件而不是屏幕上做到。一旦输出是一个文件，你就可以使用文件编辑器来查看它，并把结果通过打印机打印出来。

本课程的程序演示了如何把结果输出到一个文件中。就像输入文件一样，输出文件

1）可以有任何可接受的文件名。

2）在使用之前必须和一个文件指针相关联。

3）使用之前必须打开。

4）使用以后必须关闭。

源代码

```
#include <stdio.h>
void main(void)
{
        double  income=123.45, expenses=987.65;
        int week=7, year=2006;
        FILE *myfile;

        myfile = fopen("L3_4.OUT","w");
        fprintf(myfile,"Week=%5d\nYear=%5d\n",week,year);
        fprintf(myfile,"Income  =%7.2lf\n Expenses=%8.3lf\n",
                        income,expenses);
        fclose(myfile);
}
```

fprintf 函数，与 print 用法类似，这里，它和一个指向 myfile 的文件指针配合使用

输出文件 3_4.OUT

```
Week=   7
Year= 2006
Income  =123.45
Expenses=987.650
```

解释

1）使用哪个函数把数据写入文件？C 语言中使用 fprintf 函数把数据写入文件中。通常 fprintf 函数的语法如下：

```
fprintf(file_pointer, format_string, argument_list);
```

fprintf 函数把 argument_list 中的值，利用 format_string 写入文件指针关联的文件中去。例如语句

```
fprintf(myfile," Week = %5d\n Year = %5d\n",week,year);
```

把 argument_list 中的值 week 和 year 以 "week = %5d \n year = %5d\n" 的格式写入 myfile 文件指针指定的文件中。

2）使用什么函数打开文件，并把打开的文件和文件指针建立连接？在数据写到外部文件之前，文件必须用 fopen 函数打开，它的语法如下：

```
file_pointer = fopen (file_name,access_mode);
```

其中，file_pointer 和 file_name 与输入文件中的作用是一致的。但是写入的 access_mode 是 'w'，这意味着文件被以文本的模式写入。例如语句

```
myfile = fopen ("L3_4.OUT","w");
```

打开写入文件 L3_4.OUT，并在文件和文件指针 myfile 之间建立起连接。如果在语句中没有 fclose 语句，C 语言会在程序执行完毕以后自动关闭打开的文件。我们也可以使用 fclose 函数来手工关闭一个输出文件。

概念回顾

1）使用 fopen 函数来打开一个文件并以 "w" 的打开模式写入：

```
myfile = fopen ("L3_4.OUT","w");
```

2）数据可以通过使用 fprintf 函数来写入文件。

```
fprintf(file_pointer, format_string, argument_list);
```

练习

1. 判断真假：
   a. 使用 fprintf 函数把输出写到屏幕上。
   b. 使用 fprintf 函数把输出写到外部文件里。
   c. 在把输出写到外部文件之前，必须在外部文件和文件指针之间建立起连接。
   d. 不使用外部文件的时候，最好关闭它。
   e. 打开输出文件之前，必须声明一个文件指针。
2. 下面的语句中是否有错误，如果有，请指出它们。
   a. #include <stdio.h>
   b. File myfile;
   c. *myfile = fopen (TEST.OUT,w);
   d. fprintf(*myfile,"Week=%4d\nYear=%5d\n",&week,&year);
   e. fclose("myfile);
3. 写一个程序，从键盘输入你上学期所有的成绩，然后把所有的成绩和平均 GPA 输出到屏幕上的同时，写入到一个名字为 MYGRADE.REP 的文件中。

答案

1. a. 假　b. 真　c. 真　d. 真　e. 真

2. a. FILE*myfile;
   b. myfile = fopen ("TESTOUT", "w");
   c. fprintf(myfile, "..", week, year);
   d. fclose(myfile);

应用程序 3.1 面积计算——复合运算符和程序开发

问题描述

这个程序并不是出于实用的目的，只是为了演示如何识别模式并基于这些模式开发一个程序。另外，程序演示了前面介绍过的开发程序的方法论。这里没有介绍新的概念以免扰乱我们对模式和方法论的关注。这个例子在概念上力求简单，它的功能只是让你熟悉写算术运算程序的一些基本逻辑。

写程序计算四个直角三角形的面积。三个直角三角形如图 3-2 所示，你可以从这三个三角形演示的模式中推导出第四个三角形的信息，然后用这个信息来写你的程序。

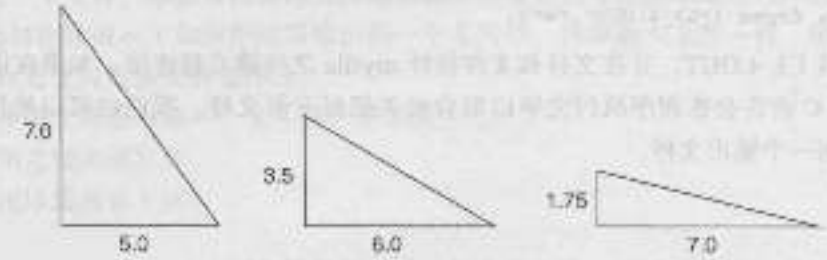


图 3-2 三个直角三角形

解决方法

这里用第 1 章介绍的过程来开发应用程序。重复一下，在开发程序的时候，遵循一定的流程比随意地开发更重要。如果你不选择跟随我们介绍的方法论，那么就遵循其他方法论。

1. 相关公式

注意，边长有一定的模式，水平边的长度分别为 5，5+1=6，6+1=7，并且垂直边为 7，7/2=3.5，3.5/2=1.75。因此，可以看出，第四个三角形的水平边为 7+1=8，垂直边为 1.75/2 = 0.875。

水平边可以用下面的公式来计算：

$$L_{h2} = L_{h1} + 1$$
$$L_{h3} = L_{h2} + 1$$
$$L_{h4} = L_{h3} + 1$$

其中，L_{h1} 是第一个三角形的水平边的长度

L_{h2} 是第二个三角形的水平边的长度

L_{h3} 是第三个三角形的水平边的长度

L_{h4} 是第四个三角形的水平边的长度

垂直边的长度是：

$$L_{v2} = L_{v1}/2$$
$$L_{v3} = L_{v2}/2$$
$$L_{v4} = L_{v3}/2$$

其中，L_{v1} 是第一个三角形的垂直边的长度

L_{v2} 是第二个三角形的垂直边的长度

L_{v3} 是第三个三角形的垂直边的长度

L_{v4}是第四个三角形的垂直边的长度

注意三角形的面积公式为：

$$A = 0.5L_1L_2$$

2. 特殊的例子

针对这个特殊的程序，最后的结果可以轻松地用计算器计算出来。但对很多真实的程序，需要执行非常大量的计算，所以不能用计算器来处理。下面的公式显示了边长和对应的面积。

三角形 1

$$L_{h1} = 5$$
$$L_{v1} = 7$$
$$A_1 = (0.5)(5)(7) = 17.50$$

三角形 2

$$L_{h2} = 5 + 1 = 6$$
$$L_{v2} = \frac{7}{2} = 3.5$$
$$A_2 = (0.5)(6)(3.5) = 10.50$$

三角形 3

$$L_{h3} = 6 + 1 = 7$$
$$L_{v3} = \frac{3.5}{2} = 1.75$$
$$A_3 = (0.5)(7)(1.75) = 6.125$$

三角形 4

$$L_{h4} = 7 + 1 = 8$$
$$L_{v4} = \frac{1.75}{2} = 0.875$$
$$A_4 = (0.5)(8)(0.875) = 3.50$$

3. 算法

执行样例计算的目的是勾画出得到正确且完整的结果所需的所有步骤。这个样例计算用来指导下面的算法：

开始

声明变量

初始化第一个三角形的水平边的长度
初始化第一个三角形的垂直边的长度
计算第一个三角形的面积

计算第二个三角形的水平边的长度
计算第二个三角形的垂直边的长度
计算第二个三角形的面积

计算第三个三角形的水平边的长度

计算第三个三角形的垂直边的长度
计算第三个三角形的面积

计算第四个三角形的水平边的长度
计算第四个三角形的垂直边的长度
计算第四个三角形的面积

在屏幕上打印结果

结束

源代码

```
#include <stdio.h>
void main(void)
{
    double horizleg, vertleg, area1, area2, area3, area4;

    horizleg  = 5.0;
    vertleg   = 7.0;
    area1     = 0.5 * horizleg * vertleg;

    horizleg  += 1.0;
    vertleg   /= 2.0;
    area2      = 0.5 * horizleg * vertleg;

    horizleg  += 1.0;
    vertleg   /= 2.0;
    area3      = 0.5 * horizleg * vertleg;

    horizleg  += 1.0;
    vertleg   /= 2.0;
    area4      = 0.5 * horizleg * vertleg;

    printf ("  \n\
    First  triangle area = %6.2f   \n\
    Second triangle area = %6.2f   \n\
    Third  triangle area = %6.2f   \n\
    Fourth triangle area = %6.2f   \n",
    area1, area2, area3, area4);
}
```


这里的源代码直接来源于算法。查看每一行以确保你理解它们的含义。你可以参考算法来一行一行地理解源代码。

输出

```
First  triangle area   = 17.50
Second triangle area   = 10.50
Third  triangle area   =  6.13
Fourth triangle area   =  3.50
```

注释

这个程序演示了编程中如何使用模式。你可以想象，遵循同一模式可以写出计算 50 个

三角形面积的程序。随着介绍更多的编程技巧，可以用更少的语句写出这个程序。

这个特殊的例子是故意为之，就是要建立一种模式。你会发现实际的问题其实也包含模式。编写高级程序的技能之一就是识别出模式，并利用模式写出高效率的代码。

修改练习

修改上述程序完成以下任务：

1. 计算遵循同一模式的10个三角形的面积。
2. 把结果打印到文件。
3. 仅用三个变量（horizleg、vertleg和area）来产生同样的输出。
4. 每次计算时，不是把垂直边的长度减半，而是加倍。

应用练习

用本章介绍的4步过程写出下列程序：

3.1 写程序生成奥林匹克田径距离的一个表格，分别用米、公里、码和英里作为单位。这里采用下面的距离：

100米

200米

400米

800米

利用距离的转换模式来构建你的程序。（1米=0.001公里=1.094码=0.0006215英里。）

输入规范：没有外部输入（不需要从键盘和文件输入数据）。所有的距离用实型数。

输出规范：把结果遵循下面的方式输出到屏幕上。

奥林匹克田径距离表			
米	公里	码	英里
100	—	—	—
200	—	—	—
400	—	—	—
800	—	—	—

表中的数字右对齐。

3.2 写一个程序，根据5个直角三角形的两个直角边，计算斜边的长度。

输入规范：从键盘输入数据，并相应下面的方式来提示用户。

屏幕输出	输入5个直角三角形的直角边的长度	
键盘输入	leg1	leg2
键盘输入	leg1	leg2
键盘输入	leg1	leg2
键盘输入	leg1	leg2
键盘输入	leg1	leg2

所有的输入都是实型数。

输出规范：遵循下面的模式在屏幕上打印结果：

五个三角形的斜边长度			
三角形序号	leg1 长度	leg2 长度	leg3 长度
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—

表中所有的数字右对齐。

3.3 下表给出太阳距离最近的四个行星的距离。写一个程序将公里距离换算成厘米和英寸。

行星	距离（百万公里）
水星	58
金星	108.2
地球	149.5
火星	227.8

输入规范：不需要外部输入，这些距离可以在源代码中初始化。

输出规范：利用下面表格的样式，把结果打印到屏幕。

	到太阳的距离		
行星	百万公里	厘米	英寸
水星	58		
金星	108.2		
地球	149.5		
火星	227.8		

注意，如果要在表中正确显示出结果，应该采用科学计数法。

3.4 稳定电流的欧姆定律可以写成：

$$V = IR$$

其中，V = 导体的电压差

I = 导体中流过的电流

R = 导体的电阻

写一个程序，填充下面表格中的空白处。

情形	V（伏特）	I（安培）	R（欧姆）
1	10	2	—
2	—	5	7
3	3	—	4

输入规范：输入数据来自键盘并被当成一个实型数。应该用下面的模式来提示用户：

```
"For case 1, enter the voltage and current."
"For case 2, enter the current and resistance."
"For case 3, enter the voltage and resistance."
```

输出规范：在屏幕上打印完整的表格。

3.5 一个简单的钟摆的摆动周期定义如下：

$$T = 2\pi\sqrt{\frac{l}{g}}$$

其中（采用国际标准单位），T = 周期（秒）

l = 钟摆的长度（米）

g = 重力加速度（9.81 米/秒^2）

写一个程序填充下面的表格：

长度（米）	周期（秒）
0.5	—
1.0	—
—	10
—	20
0.32	—

输入规范：用练习 3.4 中介绍的类似的提示样式，提示用户输入数据。

输出规范：在屏幕上打印完整的表格。

3.6 一个运动物体的动能定义如下：

$$k=\frac{1}{2}mv^2$$

其中，k = 物体的动能

m = 物体的质量

v = 物体的速度

在物体运动方向上施加一个推力，这个推力所做的功为：

$$W=Fs$$

其中，W = 力所做的功

F = 施加在物体上的力

s = 物体被推动过程中移动的距离

当在水平方向上推一个物体的时候，$K=W$。所以

$$Fs=\frac{1}{2}mv^2$$

假设一个人的推力为 0.8kN，并且有一辆 m=1000kg 的汽车。写一个程序填充下面的表格。

推动的距离（米）	最后的速度（米/秒）	需要多少人去推
5	10	—
—	10	15
20	—	8

输入规范：提示用户从键盘输入数据。

输出规范：在屏幕上打印完整的表格。

本章回顾

本章中，学习了如何在 C 语言表达式中写算术运算，如何使用内建的 C 语言的数学库。我们介绍了另外一个基本的数据类型：字符型。通过学习输入和输出操作，我们了解了 C 程序中处理字符时的注意事项。最后，讨论了如何利用文件进行输入输出。文件的输入输出在编程中是非常重要的，因为文件可以永久地保存任何程序的执行结果。

但是，我们目前使用的复杂程序需要更多的高级技术。下一章，我们会学习编程的另一个重要方面——流程控制。

第 4 章
C Programming: a Q & A Approach

初级决策和循环

本章目标

结束本章的学习后，你将可以：

- 写一个表达式以描述一个检查点或一个计算语句。
- 区分不同的选择和循环方式。
- 当写判定语句和表达式的时候，观察它们正确的布局。
- 针对不同的判定过程构建逻辑流。

在前面的章节中已经学习了程序编写的基础知识，实践了书写简单的表达式以及输入和输出语句。目前，我们已经了解到计算机程序中的指令都是顺序地一行一行执行的。

本章将学习如何写指令使得程序能够根据不同的条件决定后续执行什么。例如，老师可能想把学生在测验中得到的分数转换成评分等级，他让你帮忙来完成这个任务。这可以通过手工来完成，你画一个转换表格，其中包含了分数的区间和对应的评分等级，然后根据这个信息来给出每一个学生的评分等级。下面的表给出了这样的一个转换实例。

分数	等级
0 ~ 45	F
50 ~ 59	D
60 ~ 69	C
70 ~ 79	B
80 ~ 100	A

你的程序必须能够根据学生的分数来决定他的等级。本章你会学习如何使你的程序能够做出这种决定。利用这个分数－等级转换程序，当老师键入一个学生的分数时，计算机会输出对应的字母表示的等级。现在的问题是，为了能够转换一个班的 40 个同学的成绩，老师需要运行这个程序 40 次！本章会描述一种不用重复写语句就可以反复操作的方法，这个方法叫做循环。

课程 4.1　if 控制结构和关系表达式

主题

- 关系运算符和表达式
- 简单 if 语句
- if 语句块
- 控制程序流程

你的程序有的时候需要做决定。if 语句通常用于这个目的。if 语句的格式非常简单。一个关系表达式（例如比较两个变量值的表达式）被包含在 if 语句中；如果关系表达式为真，那么“真组”的语句会运行，如果表达式为假，那么语句不会运行。

本课的程序中有 4 个 if 语句。这个程序是一个简单的游戏，用户试图猜出赌注的数目，本程序假设用户并不知道变量 jackpot 的值。

第一个 if 语句中，在关系表达式中比较了那两个变量？对于输入的等于 3 的试猜，这个关系表达式是真还是假？后续的 printf 函数是否被执行了（看最后输出的结果）？第二个 if 语句中，关系表达式是真还是假？后续的 printf 函数是否被执行了？其他 if 语句的情况如何？

源代码

```
#include <stdio.h>
void main(void)
{
    int i,guess,jackpot=8;

    printf("Try to guess the jackpot number \nbetween 1 and 10!\n");
    printf("Please type a number.\n");
    scanf("%d",&guess);

    if  (guess<jackpot)
        printf("Try a bigger number\n");

    if (guess>jackpot)
        printf("Try a smaller number\n");

    if (guess==jackpot)
            printf("Verify your guess by typing it one more time\n");

     scanf ("%d", &guess);

    if (guess==jackpot)
    {
            printf("You hit the \n");
            printf("JACKPOT!\n");
    }

    if (guess!=jackpot)
        printf("You have not guessed correctly.\nYou lose.\n");
}
```

关系表达式，它的结果只能为真或者假

等于关系运算符有两个等号

这是另外一个关系运算符

输出

```
Try to guess the jackpot number
between 1 and 10!
Please type a number.
3
Try a bigger number
7
You have not guessed correctly.
You lose.
```

键盘输入

键盘输入

解释

1）guess<jackpot 语句什么含义？表达式

```
guess<jackpot
```

是用来比较两个算术表达式值的关系表达式。一个关系表达式是一种只产生真或假两种结果

的逻辑表达式。这里它比较变量 guess 的值是否比变量 jackpot 的值小。它的通用语法是

```
左操作数    关系运算符    右操作数
```

其中，左操作数和右操作数可以是变量，就像本课程序中使用的 guess，也可以是任何的算术表达式。关系运算符用来比较两个操作数的值。C 语言里面有 6 个关系运算符。

关系运算符	含义
<	小于
<=	小于等于
==	等于
>	大于
>=	大于等于
!=	不等于

观察表中 != 的含义，这是你以前没有接触过的一个符号组合。在键盘上没有办法输入 ≠，所以 C 语言中使用 != 来代表不等于。

2）`if(guess<jackpot) printf("Try a bigger number \n");` 是什么含义？这是一个简单的 if 语句，可以归纳为如下格式：

```
if (表达式) 语句;
```

其中，如 `guess<jackpot` 这样的表达式代表一个逻辑表达式，printf 语句是一个可执行的语句。注意任何可执行的语句都可以被当成语句，包括 if 和其他控制语句。一个逻辑表达式产生一个或真或假的结果。如果逻辑表达式为真，那么 if 表达式后面的语句会执行；如果逻辑表达式为假，那么 if 表达式后面的语句不会执行。包含在 if 语句中的表达式叫做条件。

3）什么是 if 语句块？语句

```
if (guess==jackpot)    {statements... }
```

叫做 if 语句块。如果逻辑表达式为真，那么在“真”块中的那些语句（一对花括号之间的那些语句）会被执行，否则整个语句块都会被忽略。if 语句块的通常样式如下：

```
if (expression)
{
    executable statement 1;
    executable statement 2;
    ...
}
```

包含 if 语句的程序中有一些缩进，这是为了使得程序有更好的可读性。整个代码块对于关键字 if 来说，应该缩进一个 tab（至少三个空格）。本书的后续部分还会提到，缩进是使得你的程序对其他人来说更加易于理解的一个重要的方法。虽然对于缩进没有明确的规定，但是它已经变成了编程的一个惯例。

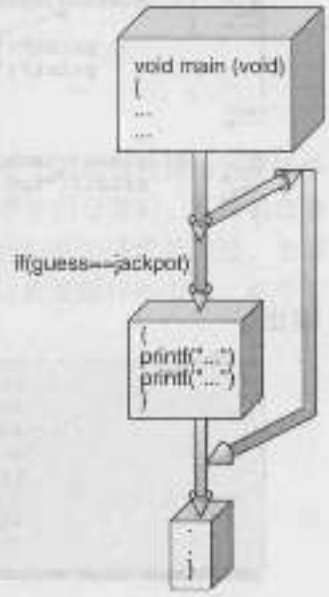


图 4-1 本课代码中使用的简单 if 语句块

本程序中第 4 个 if 语句的逻辑演示见图 4-1。图中显示控制结构使得代码块或者被执行，或者被忽略。

4）= 和 == 运算符有什么不同？= 是赋值运算符，千万不要和关系运算符 == 搞混了。一个新手程序员最容易犯的错误就是在关系表达式中使用 = 运算符。这么做会在你的程序中造成严重的错误。记住在关系表达式中使用 ==。

扩展解释

1）推荐使用 == 来比较两个 double 或者 float 类型变量的值吗？不推荐，在大部分工程编程的任务中，不推荐使用 == 来比较任何的实型变量，虽然这么做 C 编译器也不会报告错误。不推荐这么做的原因在于典型的 C 编译器将实型数表示成具有很多有效位的形式（具体有效位的位数取决于保存实型数的字节数）。如果两个 double 进行 == 比较，即使有效位的最后一位是不同的，那么 == 比较也会返回假。例如，如果 a=12.3456789123456789 和 b=12.3456789123456788 比较，a==b 的结果也是假。通常在工程领域，计算的时候都是用一个近似的值和另外一个近似的值进行比较。大部分情况下我们只对两个数是否相近或非常相近感兴趣。如果两个数非常近似，我们希望比较它们的时候返回真值。因为 == 不符合这个要求，所以一般不使用它。

另外，十进制数的二进制表示也需要近似。例如，需要很多位二进制位来准确的表示 5.3 这个数。因为这个特性，基于十进制计算出来的数不可能利用有限的二进制位来准确地表示出来。也就是说，基于十进制利用手工的计算可能不会利用有效的二进制位来准确地表示出来，所以我们不使用 == 比较两个实型数以避免这种不准确近似带来的问题。

2）如果不使用 == 比较 double、long double 或者 float，那应该怎么比较两个实型数？比较两个实型数的一种方法是使用 fabs 函数和 < 符号。例如，比较上面例子中的变量 a 和 b 的值，下面的语句将返回真值：

```
if (fabs (a-b) < 1.0e-10)
```

这里随机选取了一个常数 1.0e-10 作为一个很小的数。当你书写自己的程序时，可以根据程序的具体要求来决定需要使用一个多小的值。通常你需要使用上面的方式来比较两个实型数。

概念回顾

1）关系运算符的语法和在数学中学到的语法差别很大。

2）赋值运算符 = 和关系运算符 == 差别很大。关系运算符用于比较，而赋值运算符用于把一个值赋给一个变量。

3）块中的语句必须包含在 {} 中。在块中的任何内容都被认为是一个整体。也就是说，整个块或者被执行，或者被忽略。

练习

1. 找出下面语句中的错误。

```
a. if (today = 7) printf("Go to the park");
b. if (today = 7);
c. if (today == 7);
d. if (today ==7) if (Money>100) printf("Dine out!");
e. if (today == 7) j=i;
f. if (today == 7)
   {
     j=i+1; k=100/j;
   }
```

2. 把下面的文本用 C 语言写出：

a. 如果 $b^2-4ac<0$

b. 如果 n 等于 0，把 100 赋给 x

c. 如果 n 不等于 0，计算 $1.0/n$

3. 从键盘上输入你的分数和平均的 GPA。如果你的 GPA 分数小于 2.0，在屏幕上输出 20 行的警告信息。如果你的分数高于 3.9，那么产生 10 声"嘟"作为祝贺。

答案

1. a. `if (today == 7) printf("Go to the park");`

b. `if (today == 7);`

c. 没有错，但是语句不做任何事。

d. 没有错。

e. 没有错。

f. 没有错。

课程 4.2 简单 if-else 控制结构

主题

- 简单的 if-else 控制结构
- if-else 控制结构的简写方式

if 语句的另外一种方式就是 if-else 格式。当一组语句在逻辑表达式为假而需要执行的时候使用，我们使用这种结构。

本课的程序基于收入和支出情况来判断你是否在存钱。

看第 1 行的 if 语句，如果这一行中的逻辑表达式为假，哪个语句块被执行了（看具体的输出）？

本课程中介绍了 ?: 运算符。它是 C 语言中唯一的三目运算符，这意味着需要三个运算数。看看赋值语句右边的 ?: 表达式，这个表达式中的三个运算数都是什么？冒号把两个运算数分开了。看看变量 interest 的输出。一个运算数的值被输出，你能猜到是哪个运算数被输出吗？

源代码

```
#include<math.h>
 void main(void)
 {
     double  income, expenses, savings, deficit, interest;

     printf ("Enter your income and expenses:\n");
     scanf  ("%lf %lf", &income, &expenses);
     printf ("\n\n");

     if (income >expenses)
     {
          savings = income - expenses ;
          printf ("\nYou are saving money.                \n"
                  "Your savings for this month are: $%8.2f",savings);
     }

     else
     {
          deficit = expenses - income ;
          printf ("\nYou are running a deficit.            \n"
                  "Your deficit for this month is : $%8.2f",deficit);
     }
```

if-else 控制结构

true

false

关系表达式

```
    interest = (deficit > 0.0) ? (0.05*deficit) : (0.0) ;
    printf ("\nThe interest you owe on your debt is $%f2f \n",
                interest);
}
```

?:运算符需要三个运算数

输出

```
                Enter your income and expenses
键盘输入        3500 4500

                You are running a deficit.
                Your deficit for this month is: $ 1000.00
                The interest you owe on your debt is $50.00
```

解释

1）简单的 if-else 语句的语法是什么？简单的 if-else 语句的语法是

```
if (表达式)
{
   执行语句 1a;
   执行语句 1b;
...
}
else
{
   执行语句 2a;
   执行语句 2b;
...
}
```

执行语句 1a、1b 属于“真”块，而执行语句 2a、2b 属于“假”块。如果表达式为真，那么真块的语句得以运行，如果表达式为假，控制被转移到假块。如果语句块（无论真还是假）中的语句多于 1 个，那么语句块必须被一对花括号包围。有时括号是可忽略的，例如，

```
if (test>=0)
{
   真语句块…
}
else
   单独一行语句
```

这里真语句块包含多于一个语句，所以必须放到一对大括号中。假语句块中只有一个语句，所以花括号可以忽略。

如果假语句块中没有执行的语句，那么假语句块可以被忽略。没有假语句块的语法如下：

```
if (表达式)
{
   执行语句 1a;
   执行语句 1b;
   ...
}
```

if-else 控制结构的概念演示见图 4-2。注意使用这样的控制结构会使得执行一个语句块的同时，忽略掉另外一个语句块。

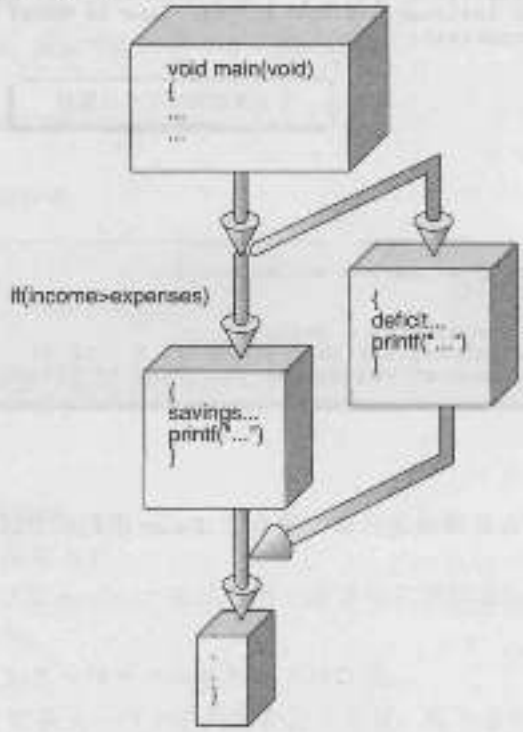


图 4-2 本课中简单的 if-else 控制结构

扩展解释

1）?：运算符如何工作？这个运算符要求三个运算数，遵循下面的格式：

```
表达式1 ? 表达式2 : 表达式3
```

如果表达式 1 是真，那么表达式 2 被计算；如果表达式 1 是假，那么表达式 3 被计算。整个?：表达式的值就等于被计算的表达式值。

在本课程序中，表达式 1 " deficit>0.0" 是真，因此计算表达式 2 "0.05*deficit"，而且赋值语句右边的值就等于表达式 2 的值。这样，interest 等于 0.05*deficit。

另外一个例子演示了?：如何找到两个数中较小的那个数，语句

```
x = (y<z) ? y : z;
```

把 y 和 z 中较小的那个数赋值给 x。

?:语句是 if-else 这种比较长的控制结构的一种简写方式。注意对于那个没有计算的表达式，没有副作用发生。

概念回顾

1）?:是一种简单的 if-else 形式，例如

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

2）if-else 语句中的 else 语句是可选的。

练习

1. 下面的语句中是否有错误，如果有，请指出它们。

```
a. if (today = 7) printf("Go to the park");
   else            printf("Go to work");
b. if (today == 7) ; else  printf("Go to work");
c. y = ? x>x : a w;
```

2. 写一个程序，从键盘输入一个数 x，如果这个数大于 0，计算它的平方根，否则计算 x*x。

答案

1. a.
```
if (today == 7) printf("Go to the park");
else            printf("Go to work");
```
b. 没错误

c.
```
y = x>x ? a : w;
```

课程 4.3　嵌套 if-else 控制结构

主题

- 嵌套的 if-else 控制结构

if-else 控制结构可以嵌套，这意味着一个 if-else 控制结构可以包含在另外一个 if-else 控制结构中。

假设你有兴趣写一个程序，告诉自己在一周某天的特定时间需要做什么。程序逻辑和你头脑中构想的具体做什么的过程是类似的。

假设你想让程序遵循如下时间表：

工作日

（周一到周五）

0:00 ~ 9:00	睡觉
9:01 ~ 19:00	工作
19:01 ~ 23:59	休息

周末

（周六、周日）

0:00 ~ 11:00	睡觉
11:01 ~ 23:59	玩

检查本课程程序，看看如何使用 if-else 结构。注意那些嵌套的 if-else 语句。

源代码

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

    printf (" Type the day and time of interest\n\n");
    scanf  (" %d %d ", &day, &time);
```

```
if (day<= 5)
{
    if (time<= 900)
        printf ("Sleep\n\n");
    else
    {
        if (time<= 1900)
            printf ("Work\n\n");
        else
            printf ("Relax\n\n");
    }
}
else
{
    if (time<=1100)
        printf ("Sleep\n\n");
    else
        printf ("Have fun\n\n");
}
}
```

if-else 控制结构嵌包含在另外一个 if-else 控制结构中

输出

```
                Type the day and time of interest
任意输入        3 1900
                Relax
```

解释

1）if-else 控制结构可以嵌套吗？可以，在C语言中不同层次的if-else控制结构可以嵌套。但是当if-else控制结构嵌套的时候，理解程序会变得更困难。像以前提过的，可理解性是一个程序的重要特征。提高程序的可读性可以使得程序对别人来说更易理解。传统的增加可读性的方法就是使用缩进。推荐缩进配对的if-else语句，这样内层的else与内层的if相配对，外层的else与外层的if相配对。例如

```
if (outer)
{...            */ 如果外层是真，执行这一块
    if (inner_1)
    {... }      */ 如果inner_1是真，执行这一块
    else
    {... }      */ 如果inner_1是假，执行这一块

    if (inner_2)
    {... }      */ 如果inner_2是真，执行这一块
    else
    {... }      */ 如果inner_2是假，执行这一块
}
else
{... }          */ 如果外层是假，执行这一块
```

本课程序中所用的嵌套控制结构的逻辑演示见图4-3，注意那些嵌套产生出来的分支。

2）if和else的数目必须配对吗？不是，在嵌套的if-else语句中，if的总数量会比else的数量多或者持平，但是不会比else的数目少。默认情况下else语句与前面最近的一个if语句配套，如果之间没有别的else语句的话。花括号可以用来标识出配对的if和else语句。

概念回顾

1）根据条件，if和else与不同的执行部分相关联。因此嵌套的if和else语句会表示一些不同的执行路径。

2）当需要的时候，使用 {} 把语句包围起来形成一个语句块。

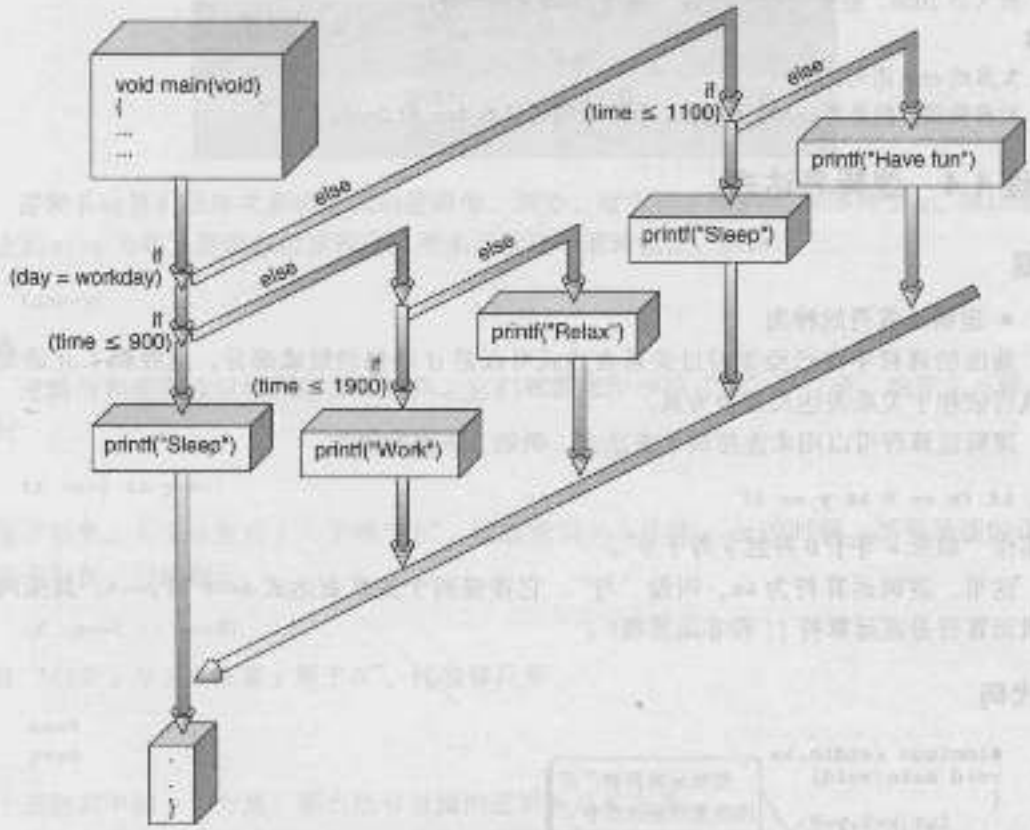


图 4-3 本课程中用到的嵌套的 if-else 控制结构

练习

1. 下面的语句中是否有错误，如果有，请指出它们。

```
a. if (i>100) printf("Hot\n");
   else       printf("Warm\n");
   else       printf("Cool\n");

b. if (i>100)  printf("Hot\n");
   if (i==100) printf("Warm\n");
   else        printf("Cool\n");
```

2. 写一个程序，从文件读入以下 10 个收入的值。

```
$187
$768
$1974
$373
$66733
$437892
$593
$8091
$40903
$1839
```

然后根据下列公式计算每个收入的税金：

a. 如果收入 <1000，不上税。

b. 如果 1000 ≤收入 <2000，25% 税金。

c. 收入 ≥ 2000，税金 =500+30% × （超出 2000 元的部分）。

答案

1. a. 太多的 else 语句。

b. 没有错误，但是当 i=100 的时候，程序会同时显示 Hot 和 Cool。

课程 4.4 逻辑表达式

主题

- 逻辑运算符的种类

前面的课程中你已经学习过关系表达式可以是 if 语句的组成部分，也看到了 if 语句是否执行依赖于关系表达式是否为真。

逻辑运算符可以用来连接两个表达式。例如，下面的语句

```
if (x == 0 && y == 0)
```

被读作“如果 x 等于 0 并且 y 等于 0”。

这里，逻辑运算符为 &&，叫做“与”。它连接两个关系表达式 x==0 和 y==0。其他两个逻辑运算符是或运算符 || 和非运算符 !。

源代码

```
#include <stdio.h>
void main(void)
{
    int x=5,y=0;

    printf("x=%2d, y=%2d\n\n",x,y);

    if (x>0 && y>=0)
        printf("x is greater than 0 and"
               " y is greater than or equal to 0\n\n");

    if (x==0 || y==0)
        printf("x equals 0 or y equals 0\n\n");

    if (! (x==y))
        printf("x is not equal to y\n");
}
```

逻辑运算符被广泛用在关系表达式中

输出

```
x= 5, y= 0,

x is greater than 0 and y is greater than or equal to 0

x equals 0 or y equals 0

x is not equal to y
```

解释

C 语言中有多少逻辑运算符？C 语言有三个逻辑运算符：&&、|| 和 !。&& 和 || 是双目运算符，它们用在两个关系表达式的中间。! 是单目运算符，它只出现在一个关系表达式的

前面。这些关系运算符的意义如下：

运算符	名字	操作	操作符类型
!	逻辑非	取反	单目
&&	逻辑与	逻辑合取	双目
||	逻辑或	逻辑析取	双目

逻辑非运算符反转关系表达式的逻辑值。例如，对于 x=5 和 y=0，x 不等于 y，所以逻辑表达式 x==y 为假。逻辑非运算符可以用来反转这个逻辑假值。因此

```
!(x==y)
```

为真。

逻辑与和逻辑或以相同的方式工作，它们就像数学中的“与”和“或”的意义一样。C 语句

```
if (x>0 && y>=0)
```

读作“如果 x 大于 0 并且 y 大于等于 0”，代表着当 x>0 并且 y>=0 的时候，括号里面的逻辑表达式为真。与此相反

```
if (x==0 || y==0)
```

读作“如果 x 等于 0 或者 y 等于 0”，代表着只要

```
x==0
y==0
```

两个表达式中的一个为真，那么括号里面的逻辑表达式为真。

在下面的表中总结了所有逻辑表达式的可能结果，其中 A 和 B 分别代表逻辑表达式。

A	B	A&&B	A||B	!A	!B
真	真	真	真	假	假
真	假	假	真	假	真
假	真	假	真	真	假
假	假	假	假	真	真

逻辑表达式被广泛用在 if 语句中。如果逻辑表达式为真，那么一系列 C 语句被执行。如果逻辑表达式为假，那么另外一系列 C 语句被执行。

概念回顾

1）有很多方法可以表达一个条件。

2）逻辑与是 &&，逻辑或是 ||。在 C 语言中，运算符 & 和 | 与 && 和 || 是完全不一样的。

3）取反（!）作用于单个运算数。

练习

1. 给定 x=200 和 y = -400，判断下面这些逻辑表达式是真是假。注意 a 和 b 两道题中，只需要计算逻辑表达式的前半部分，就可以确定出整个结果的逻辑值了，为什么？

a. (x<y && x!=y)
b. (x>y || x==y)
c. !(x>y)

2. 把下面的文本用C语言写出：
   a. 如果 $(a/b)>100$ 并且 $a<b$
   b. 如果 $(a+b)$ 不等于200并且 $b \geq 300$
   c. 如果 $(a+b) \leq 2200$ 或者 $(a-b) \times 4$ 等于500
3. 假设某个地区的C语言程序员的年需求为 D，年供应为 S，且符合下面公式的定义：

$S=1000+50 \times (Y-1990)$ $(1990 \leq Y \leq 2010)$

$D=1200$ $(1990 \leq Y \leq 1995)$

$D=1200+60 \times (Y-1995)$ $(1995 \leq Y \leq 2010)$

写出程序执行下面的任务：
   a. 输出从1990到2010年的C程序员的供应量和需求量。
   b. 找出那些年中没有足够的C程序员。
   c. 计算出1990到2010年所有失业的C程序员。

答案

1. a. 假（因为第一个表达式是假，而且逻辑运算符为&&，因此无论第二个逻辑表达式是什么，最后的结果就是假）
   b. 真（因为第一个表达式是真，而且逻辑运算符为||，因此无论第二个逻辑表达式是什么，最后的结果就是真）
   c. 假
2. a. `if ( (a/b)>100 && a<b)`
   b. `if ( (a+b)! = 200 && b>=300)`
   c. `if ( (a+b)<=-200 ||(a-b)*4==500 )`

课程4.5 逻辑运算符的优先级

主题

- 运算符的优先级和结合性
- 逻辑值和关系表达式

就像给数学表达式一个数字值一样，C语言也给关系表达式一个数字值。如果关系表达式为假，C语言赋给0。如果为真，C语言赋给1（你也可以把它当作非零）。C语言编译器也会以相反的模式进行操作。如果关系表达式的值为0，那么它知道结果为假，如果表达式的结果非0，那么结果为真。

使用变量的情形与此类似，如果一个变量的值为0，那么它可以被当成假。如果一个变量的值不是0，那么它可以被当成真。

检查源代码中前三个if语句，考虑其中变量的值后，你能合理地解释这些if语句所对应的输出吗？另外，注意程序中的逻辑取反（!）可以用在一个变量上。如果不看程序的输出，您能猜到!a的结果吗？

前面介绍过C语言已经制定了算术运算符的优先级顺序。同样，C也制定了关系和逻辑运算符的优先级顺序。从源代码的两个复合逻辑表达式和输出中，你能判断出逻辑运算符的优先级吗？

源代码

```
#include <stdio.h>
 void main(void)
 {
     int  a=4,b=-2, c=0 ,x;

     if(a) printf("a=%2d, !a=%2d\n",a,!a);
     if(b) printf("b=%2d, !b=%2d\n",b,!b) ;

     if(c) printf("Never gets printed\n");
     else  printf("c=%2d, !c=%2d\n\n",c,!c);

     if ( a>b   &&   b>c   ||   a==b )  printf("Answer is TRUE\n");
     else                               printf("Answer is FALSE\n");

     x=  a>b   ||   b>c   &&   a==b;
     printf("x=%2d, !x=%2d\n",x,!x);
 }
```

单个变量可以根据它的值被当成真或者假

就像算术运算符一样，关系运算符的优先级也会影响运算的顺序

逻辑表达式的结果可以赋给一个整型的变量

输出

```
a= 4, !a= 0
b=-2, !b= 0
c= 0, !c= 1

Answer is FALSE
x= 1, !x= 0
```

解释

1）逻辑、关系和算术运算符的优先级和结合性是什么？这些运算符的优先级和结合性如下：

运算符	名称	结合性	优先级
()	括号	左到右	1（最高）
++, --	后增（减）	左到右	2
++, --	前增（减）	右到左	2
!	逻辑非	左到右	3
+, -	正负号	左到右	3
-=, +=, *=, /=, %=	复合赋值	右到左	3
*, /	乘除	左到右	4
+, -	加减	左到右	5
<=, >=, <, >, ==, !=	关系运算符	左到右	6
&&	逻辑与	左到右	7
||	逻辑或	左到右	8
=	赋值	右到左	9（最低）

这个表显示了括号有最高的优先级，紧随其后的是单目自增/自减运算符和逻辑非运算符。通常，算术运算符（包括加、减、乘、除等）比关系运算符有更高的优先级。然后就是逻辑运算符，其中逻辑与比逻辑或有更高的优先级。赋值运算符有最低的优先级。除了前自增/自减运算符、复合赋值运算符和赋值运算符，运算符都是从左到右进行计算的（结合性）。

例如，假设 a=4，b=22 且 c=0，表达式

```
x = ! a>b || b>c  &&  a==b !
```

等同于（注意括号被加到了相应的位置以标识出优先级的层次）

```
x = ( ( a>b ) || ( ( b>c ) && ( a==b ) ) )
```

表达式最后的结果是真。在C语言中，假被定义为0，而真被定义为非0（任何正的或负的整数）。当我们把上面的表达式赋值给一个整型数x的时候，x的值为1，因为上面表达式的结果为真。如果x为真，!x就为假，输出!x会输出0。

2）单个变量的逻辑值是什么？如图4-4中演示，如果单个变量的值为0，那么它的逻辑值为假。如果单个变量为非0，那么它的逻辑值为真。例如，本课中c的逻辑值为假，因为c等于0。但是a和b的逻辑值为真，因为a(=4)和b(=22)都非0。另外，!a和!b都是假，所以输出它们会输出0。与此同时，!c会输出1。

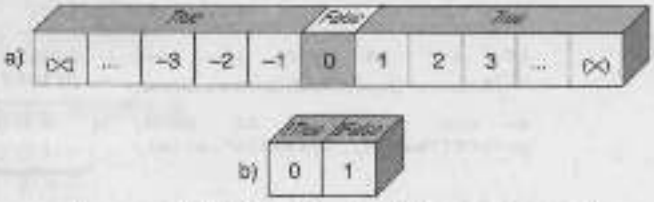


图4-4 a）整数值的真假，b）非真和非假的整数值

3）可以用a>b==c的方式来使用关系表达式吗？可以，也不可以。由于所有的运算符有相等的优先级，所以这个表达式从左到右运行。例如，如果a=4，b=22且c=5，这个表达式的结果为假。运算的步骤如下：

a>b为真，a>b这个表达式的值被赋值为1，接下来1==c逻辑值为假。

如果你的本意是a>b&&b==c，那么写成a>b==c是错误的。

概念回顾

1）C语言中的关系表达式和数学中的关系表达式写法不一样。例如在数学中a<b<c的含义是清晰的。但是在C语言中却要用（a<b&&b<c）来表达出相同的含义。

2）0值被翻译成逻辑假，任何非零值都被翻译成逻辑真。

3）当你不确定优先级的顺序时，使用括号来正确表述你想要表达的优先级顺序。

练习

1. 假设a=100，确定下面的逻辑表达式是真是假：

```
a.a==100 && a>100 && !a
b.a==100 || a>100 && !a
c.a==100 && a>100 || !a
d.a==100 || a>100 || !a
```

2. 假设a=1，b=2，c=3，d=4，下面的语句中是否有错误，如果有，请指出它们。

```
a.if (a>b)
      printf("This is an arithmetic if_statement\n")

b.if (a>b)
  {
      printf("This is a block if_statement\n");
      other-statements...
  }

c.if (a>b == c)
      printf("This is a block if_else statement\n");
      other-statements...
```

```
  }
  else
  {
      printf("This is a block if_else statement\n");
    other-statements...
  }

d.if (a>b) if (c>d)
      printf("This is a nested if_statement\n");

e.if (a);
  {
      printf("if and else are matched");
      other-statements...
  }
  else
      if(b>a)
      {
          printf("if and else are matched");
          other-statements...
      }
      else
      {
          printf("if and else are matched");
          other-statements...
      }

f.if (a>b)
  if (c<d)
  {
      printf("More if than else");
      other-statements...
  }
  else
  {
      printf("This else is associated with the last if");
      other-statements...
  }

g.if (a>b)
  {
   if (c<d)
   {
      printf("More if than else");
      other-statements...
   };
  };
  else
  {
      printf("This else is associated with the 1st if");
      printf("since we use braces to block the last if");
  };
```

3. 在本课的程序中，解释为什么表达式 (a>b&&b>c||a==b) 得到假值。

答案

1. a. 真 && 假 && 假 = 假

   b. 真 || 假 && 假 = 真

   c. 真 && 假 || 假 = 假

   d. 真 || 假 || 假 = 真

2. a. 在 printf 后面需要一个分号。
   b. 没错误。
   c. 在第一个 printf 语句的前面需要一个左括号 (，表达式从左向右进行计算。
   d. 没错误，这是一个嵌套的 if 语句。
   e. 在 if (a) 的后面不应该有一个分号。
   f. 没错误。
   g. 在三个右括号) 后面，不应该有分号。

课程 4.6 switch 和 if-else-if 控制结构

主题

- if-else-if 控制结构
- 使用 switch 控制语句
- switch 和 if-else-if

本课程给出两段代码，它们用不同的方法来执行相同的任务。第一段代码使用 if-else-if 控制结构，第二段代码使用 switch 控制结构。

利用 if 控制结构的知识，你可以读懂程序 1 的代码和输出，并追踪程序的流程。与程序 2 的代码进行比较。

对程序 2 来说，查看那些有关键字 switch 的程序行。什么符号跟随 switch 关键字？case 被用了三次。每一次都跟随着一个不同的整型常量。这些整型常量和跟在 switch 语句后面的符号有什么关系？什么标志被用在了常量的后面？

关键字 break 使用了三次。跟随着程序的流程，你能看出 break 语句引导程序运行到什么地方了吗？这里也使用了关键字 default。它的目的是什么？

源代码 1

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

   printf("Please type 1, 2, or 3\n"
          "1. Breakfast\n"
          "2. Lunch\n"
          "3. Dinner\n");
   scanf("%d",&option);

   if(option==1)
   {
          printf("Good morning\n");
          printf("Order breakfast\n");
   }
   else if (option==2)
   {
          printf("Order lunch\n");
   }
   else if (option==3)
   {
          printf("Order dinner\n");
   }
   else
   {
          printf("Order nothing\n");
   }
}
```

if-else-if 控制结构中，只有一个语句块（被包含在一对括号中）被执行了

源代码 2

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

 printf("Please type 1, 2, or 3\n"
        "1. Breakfast\n"
        "2. Lunch\n"
        "3. Dinner\n");
 scanf("%d",&option);

 switch(option)
 {
 case 1: printf("Good morning\n");
         printf("Order breakfast\n");
         break;

 case 2: printf("Order lunch\n");
          break ;

 case 3: printf("Order dinner\n");
          break;

 default:
         printf("Order nothing\n");
 }

}
```

标记后面要跟一个冒号

switch 控制结构

break 语句使得流程跳出 switch 控制结构

输出

```
Please type 1, 2, or 3
1. Breakfast
2. Lunch
3. Dinner
2
Order lunch
```

解释

1）if-else-if 控制结构如何工作？ if-else-if 控制结构通过一系列的语句块来逐步转换程序的控制。控制在关系表达式为真的时候会停下，然后去执行对应的语句块。当执行完这个语句块以后，控制被转移到整个控制结构的末尾。如果这些关系表达式都不为真，那么最后一个语句块会被执行。在本课的程序中 option 的值为 2，所以第一个语句块不会被执行。因为 option==2 这个关系表达式为真，所以第二个语句块会被执行。第三个和第四个语句块会被跳过，控制直接跳转到整个控制结构的末尾。

if-else-if 控制结构的组成如下：

```
if (关系表达式1)
{
        语句块1
}
else if (关系表达式2)
{
        语句块2
}
```

```
.
.
.
.
.
else if  (关系表达式b)
{
      语句块b
}
else
{
      语句块
}
```

图 4-5 演示了本课中的 if-else-if 控制结构。注意不同的 option 值会有不同的执行分支。

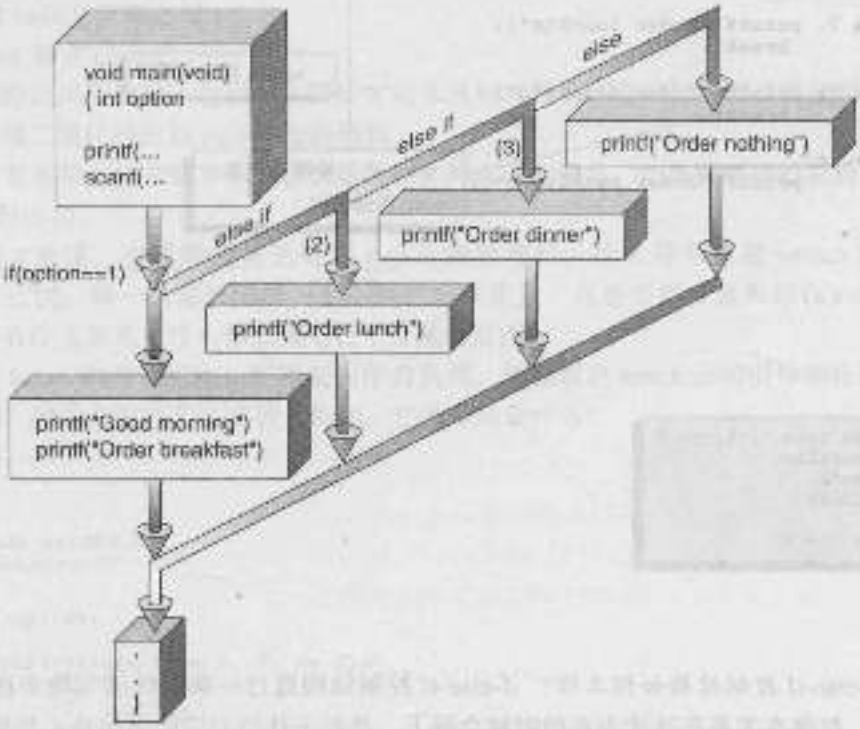


图 4-5 本课中的 if-else-if 控制结构，将 switch 和嵌套 if-else 控制结构的演示进行比较

2）switch 语句有什么用？switch 语句或者 switch 控制结构与 if-else-if 控制结构构建的方式通常是类似的。它用来进行转换控制。语法如下：

```
switch(表达式)
{
case 常数1:
      语句1a
      语句1b
      ...
case 常数2:
      语句2a
      语句2b
      ...
      ...
```

```
    default:
        语句
}
```

其中表达式必须包含在一对括号内，并且当程序流程进入 switch 块中的时候，表达式必须是一个整型的数值。一个 switch 块必须用一对括号包围起来。术语常数 1、常数 2 等也必须是整型类型。注意这些常数的表达式后面都跟着一个冒号。所有的常数表达式必须是唯一的。这意味着没有两个常数表达式可以相等。虽然不是必需的，但是通常情况下，最后一个 case 是关键字 default。如果表达式的值与任何一个常数的值都不匹配，那么在 default 下的语句被执行。default 是可选的，如果没有 default 语句，并且表达式的值与任何一个常数的值都不匹配，那么整个 switch 块将被忽略。

图 4-6 演示了本课中使用的 switch 控制结构，与图 4-3 和图 4-5 进行比较。注意 if-else-if、嵌套 if 和 switch 控制结构的相同点。在所有演示的例子中，控制结构只选择一个语句块执行，而忽略掉其他的语句块。

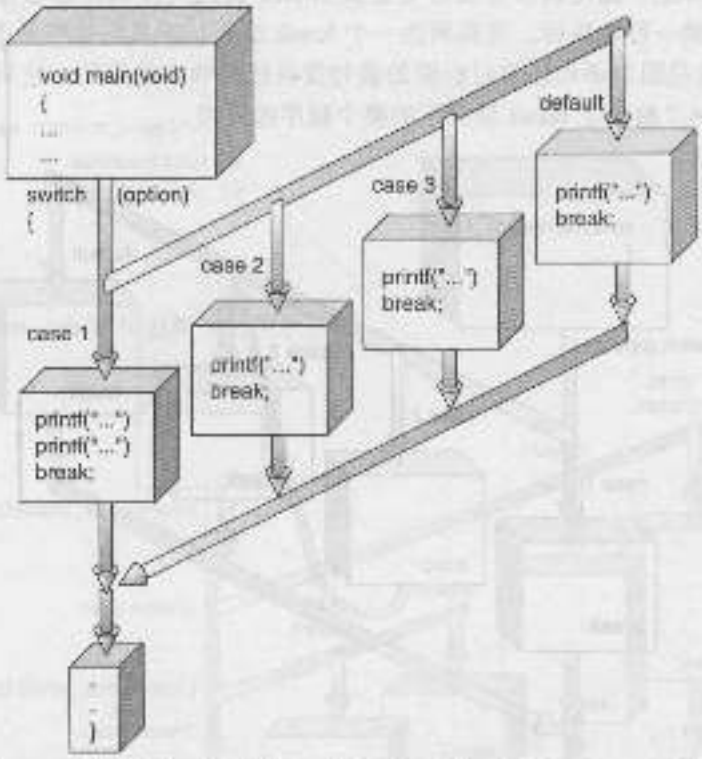


图 4-6 本课程序的 switch 控制结构。注意其 break 语句的重要性。将它与 if-else-if 和嵌套 if 结构进行比较

3）case 是什么？这是一个只能用在 switch 控制结构中的关键字。它被用来形成一个 case 标签。case 标签是一个常量跟随着一个冒号。标签并不影响其后语句的执行。在 switch 控制结构中，C 查看 switch 表达式和 case 标签中的表达式是否相等，然后执行相等的那个 case 标签中的语句。例如，对于上面给出的格式，如果 switch 表达式和常量 1 相等，那么程序流程被转换至 case 常量 1，然后语句 1a 和语句 1b 被执行。因为 switch 语句只是查看相等关系，所以它和 if-else-if 有些不同，后者可以使用不同的关系运算符。

4）什么是 break 语句？switch 控制结构中的 break 语句会终止一个 switch 选择片段的执行。终止意味着控制转移到 switch 控制结构的右结束括号处。

5）跟随 case 标签的一系列语句必须以 break 语句结尾吗？不是，虽然大部分时候，最后一句是 break 语句，因为它结束了当前 case 标签分支的执行，并离开 switch 控制结构。如果没有 break 语句，那么下一个 case 标签的内容会被执行，程序演示如下：

```
switch (option)
{
 case (1): printf("Entering case 1\n");
           break;
 case (2): printf("Entering case 2\n");

 case (3): printf("Entering case 3\n");
           break;
}
```

如果 option=1，那么 "Entering case 1" 会被显示在屏幕上。如果 option=3，"Entering case 3" 会被显示。但是如果 option=2，那么 "Entering case 2" 和 "Entering case 3" 会被同时显示在屏幕上。这是因为首先 C 会查找 switch 表达式和 case 标签相同的分支，找到后程序会被一行接一行地执行，直到到达一个 break 语句，或者到达整个语句块的末尾（右括号来标识）。这是因为语句标签对后面的语句没有任何约束的作用，它只是一个流程能够达到的标记。图 4-7 给出了 break 缺失下的整个程序的流程。

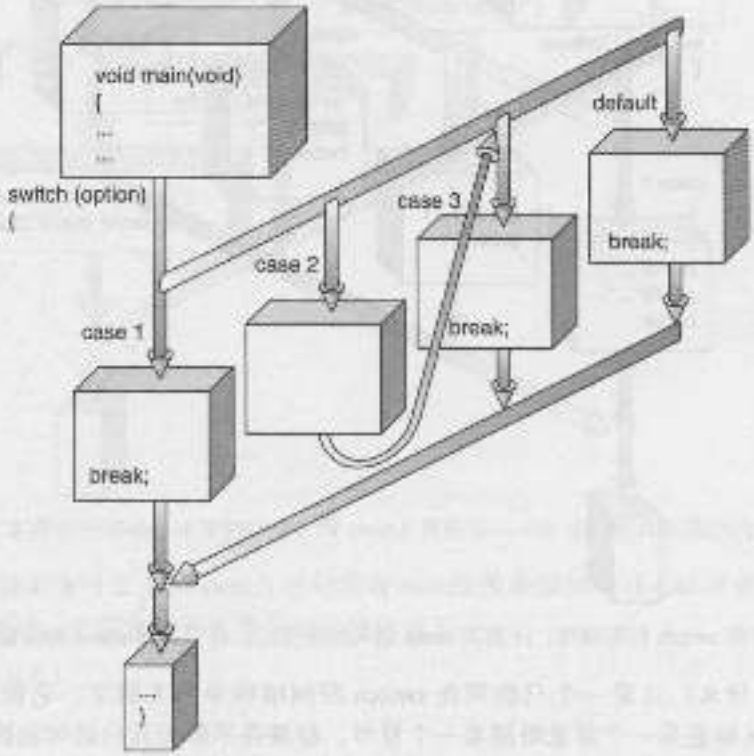


图 4-7 本课程序中，case 2 语句块中没有 break 语句的程序控制流程。注意 break 语句会使得流程离开 switch 结构，但是如果没有 break，那么程序会继续执行下一个 case

6）什么是 default？这是一个只用在 switch 控制结构中的关键字。如果没有 case 标签能和 switch 表达式匹配，那么控制会跳转到 default 标签。default 是一个关键字，不要把它理解成一个用户定义的标签。

7）switch 控制结构可以嵌套吗？可以，嵌套的控制结构组成如下：

```
switch (outer_expression)
{
        case constant_outer1:
                switch (inner_expression)
                {
                        case constant_inner1:
                                statement inner_1a
                                statement inner_1b
                                ..
                                ..
                        case constant_inner2:
                                statement inner_2a
                                ..
                                ..
                }
        case constant_outer2:
                statement outer_2a
                statement outer_2b
                ..
                ..
        case constant_outer3:
                statement outer_3a
                statement outer_3b
                ..
                ..
}
```

一个嵌套的 switch 结构如图 4-8 所示。

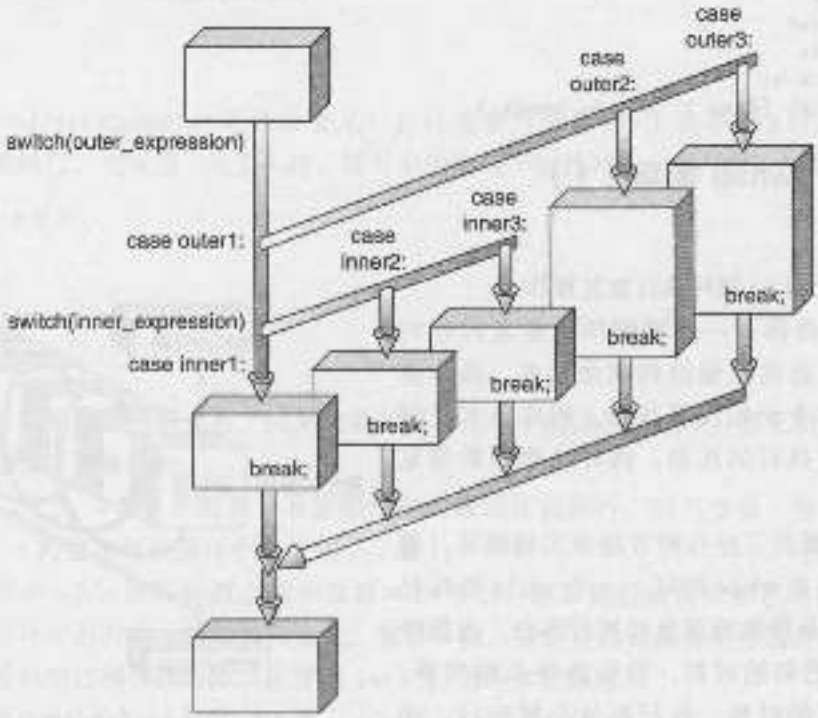


图 4-8 嵌套的 switch 结构和 break 语句

概念回顾

1）当需要检查的表达式可以计算成单一数值的时候，switch 结构可以用来代替 if-else-if。

2）switch 并不支持每一个数据类型，例如 double 数据类型就不能用在 switch 结构中。

练习

1. 判断真假：

a. Switch 语句中的 case 常量必须按顺序排列，例如 101、102、103 等。

b. 一个 switch 语句可以被一个 if-else-if 语句替换。

c. 一个 switch 语句必须包含一个 default case 部分。

2. 假设 $a=1$，$b=2$。如果下面的语句有错误，请指出。

```
a.default:
b.switch (a);
c.case 123;
d.switch {a+b}
e.switch (a): {case 1: b=a+2; break;}
```

3. 基于给定的收税办法，利用 switch 语句写一个程序来计算需要缴纳的税：

tax=income × 20%　　(income<1000)

tax=income × 30%　　(1000<=income<2000)

tax=income × 40%　　(income>=2000)

4. 利用 switch 语句完成练习 3。（提示：引入一个整型变量 A=income/1000 作为一个 switch 变量。）

答案

1. a. 假　　b. 真　　c. 假

2. a. 没错

```
b.switch (a)
c.case 123;
d.switch (a+b)
e.switch (a) {case 1: b=a+2; break;}
```

课程 4.7　while 循环（1）

主题

- 利用 while 循环执行重复操作

C 语言提供了一些能够执行重复操作的控制结构，这些控制结构叫做循环。循环使得一个或多个语句反复执行。程序员来控制语句被重复执行的次数。循环效果的解释见图 4-9。

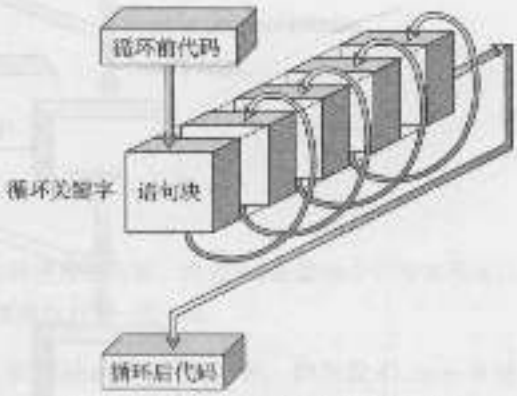


图 4-9　循环（一个语句块的重复执行）

C 语言提供了好几种方法来实现循环。最简单的方法是 while 循环。一个 while 循环包含两部分：条件检验部分和执行部分。当程序到达 while 语句的时候，检验条件会被判断。当条件为真的时候，执行部分会被运行，并

被一直执行到检验条件为假。当检验条件为假时，执行部分会被越过，程序控制被转移到 while 循环的结束部分。

查看在源代码中含有 while 关键词的文本行。从你了解的关于语句块及条件表达式的知识，你能确定出哪个表达式代表着检验条件吗？哪些语句在执行部分呢？查看输出。循环被执行了多少次？为什么？

源代码

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

     i = 1;

     while (i<= 5 )
     {
        printf (" Loop number %d in the while_loop\n",i);
        i++;
     }
}
```

检验表达式

语句块被反复执行直到检验表达式为假

增加计数变量

输出

```
Loop number 1 in the while_loop
Loop number 2 in the while_loop
Loop number 3 in the while_loop
Loop number 4 in the while_loop
Loop number 5 in the while_loop
```

解释

`while(i<=5){statements}` 是什么含义？它代表着当变量 i 小于或等于 5 时，括号中的语句被反复地执行。当变量 i 大于 5 时，括号中的语句不被执行。while 循环的结构是：

```
while (表达式)
{
   语句1
   语句2
   ...
}
```

其中表达式的结果为真或者为假，如果结果为真，括号中的语句被执行。如果循环体中只有一个语句，可以不需要括号。

当 i=1、2、3、4 和 5 的时候，本课程序中的循环体被执行，而当变量 i 等于 6 时，循环体被越过。i 的值在每次循环中被语句 `i++` 加 1。

本课程序中 while 循环的概念演示如图 4-10 所示。追踪那些被箭头标示的路径，并观察经过每次循环体的时候 i 的值如何变化。基于 i 值，检验表达式或者把控制交给循环执行体，或者把控制越过循环执行体。注意当 i=6 时，循环体被越过。

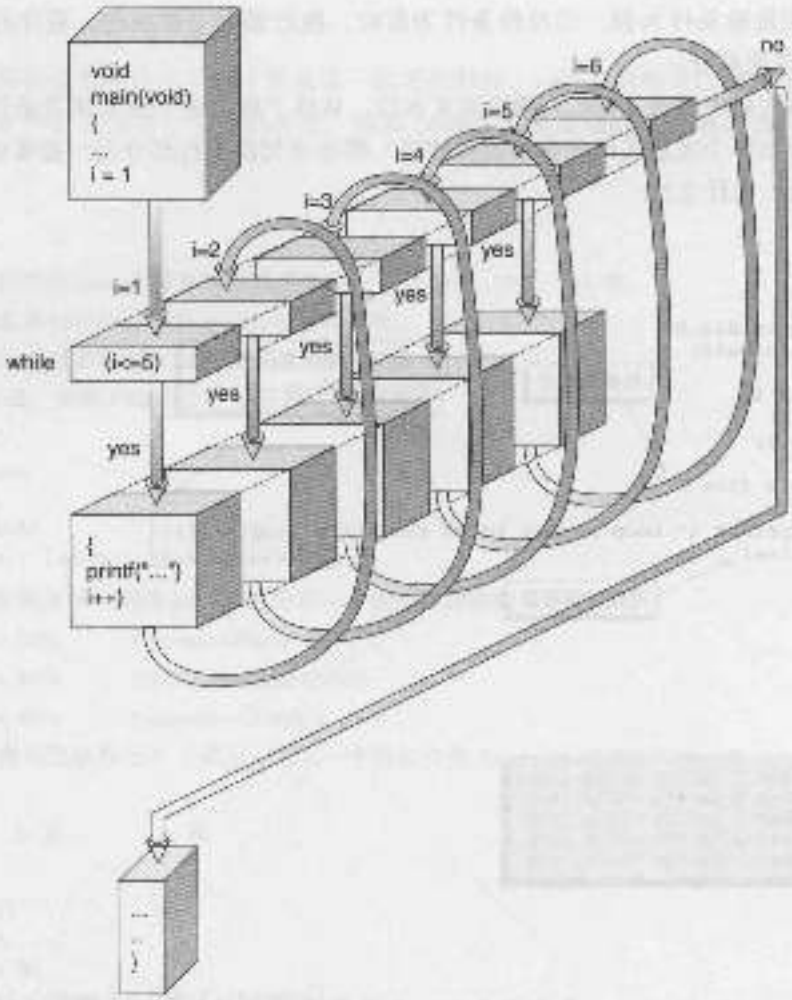


图 4-10

概念回顾

1）循环允许一个语句块被反复执行。

2）与 while 循环关联的检验表达式控制循环是否重复或终止。

练习

1. 在下面的语句中如果存在错误，请指出（假设 a 为 int 且 a=1）。

```
a.while (a<5): {printf("A=%d\n",a); a++;
b.while (a<5)  {printf("A=%d\n",a); a--;}
```

2. 用 while 循环写一个程序生成下面的表格（x 是一个 double 类型）：

```
  X        X*X       X+X
 1.0       1.00      2.00
 1.5       2.25      3.00
 2.0       4.00      4.00
 ...
10.0     100.00     20.00
```

3. 用一个 while 循环来显示一个收敛于 2 的数列：

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots$$

4. 用 if 或者其他的 C 语言语句来判断给定的一个整数 N 是否为素数。（提示：用 if 语句判断 N 是否能被 2 或者任何小于等于 $N/2$ 的奇数整除。）

答案

1. a. `while (a<5) {printf("a=%d\n",a); a++;}`

   b. 没错误，但是循环体会永远执行，因为 a 永远不会大于或等于 5。

课程 4.8　while 循环（2）

主题

- 定义 while 循环的检验表达式
- 避免形成无限循环

在下面的源代码中检查第一个 while 循环。注意检验表达式不是一个关系表达式，而只是 i 的值。当 i 为何值时循环将停止呢？

同样检查第一个 while 循环，看你是否能确定 i 的值在循环中是如何变化的。在循环中改变 i 值的是一个带有减号的复合赋值语句。在输出中查看每次语句执行时 i 的值如何变化。你能解释在这个语句中 sum 值发生的变化吗？这需要一些技巧，但你应该记住那些用来计算包含加号和减号的表达式的规则。

查看其他的 while 循环，确定 k 和 sum 的值。在这个 while 循环中你发现问题了吗？如果没有发现问题，请在电脑上运行这段程序，看看输出了什么？运行这个 while 循环时你会遇到问题。如何防止这个问题再次发生？

源代码

```
#include <stdio.h>
void main(void)
{
    int i=4, k=1, sum=0;

    while (i)
    {
       printf("old_i=%2d, ",i);
       sum+=i--;
       printf("new_i=%2d, sum=%2d\n",i, sum);
    }

    printf("\n");

    sum=0;
    while (k)
    {
      printf("old_k=%2d, ",k);
      sum -= k++;
      printf("new_k=%2d, sum=%2d\n",k,sum);
    }
}
```

单独的变量用来充当检验表达式

后减运算符使得复合赋值首先发生，然后减一

后加运算符使得复合赋值首先发生，然后加一

形成了一个无限循环，因为 k 的值永远不会为假

输出

```
old_i= 4, new_i= 3, sum= 4
old_i= 3, new_i= 2, sum= 7
old_i= 2, new_i= 1, sum= 9
old_i= 1, new_i= 0, sum=10

old_k= 1, new_k= 2, sum= -1
old_k= 2, new_k= 3, sum= -3
old_k= 3, new_k= 4, sum= -6
old_k= 4, new_k= 5, sum= -10
old_k= 5, new_k= 6, sum= -15
....
```

解释

1）while(i){语句}是什么含义？当变量i的值不等于0（0为假，非0为真），在括号中的语句将会被执行。变量i被称为标志，因为它是用来控制while循环体中的语句是否执行的一个变量。

本例中第一个while循环开始于：

```
while (i)
```

当i=4、3、2、1时，循环被执行（意味着标记是真值）；当i=0时循环终止（意味着标记是假值）。i的值通过下面的语句求和：

```
sum += i;
```

在每次循环中，i的值被下面的语句减一：

```
i--;
```

2）如何避免形成无限循环？如果执行本课中的程序，你会发现第二个while循环是一个无限循环，这是因为k的值永远不会变成0。解决这个问题的方法是生成一个新的变量做计数器。变量icount在下面的循环中是一个计数变量。它使得循环在运行30次以后终止。

```
icount=0;
while(icount < 30)
{
    printf("old_k=%2d", k);
    sum -= k++;
    printf("new_k=%2d, sum=%2d\n", k, sum);
    icount++;
}
```

3）如果while循环的检验表达式开始是假，while循环会执行吗？不会。例如，printf语句在以下while循环中不会执行，因为100<50是假。

```
while (100 <50) printf ("This will never be displayed\n");
```

概念回顾

1）和while循环关联的检验表达式决定什么时候循环终止。因此写这个表达式时一定要非常小心避免形成无限循环。

2）和while循环关联的检验表达式如果最开始为假，循环不会被执行。

练习

1. 找出下面语句中的错误并修改（假设 a 是 int 并且 a=1）。

```
a.while (5) printf("Good morning\n");
b.while (5<a) {printf("A=%d\n",a); a++;}
c.while (5+1==7) {printf("A=%d\n",a); break;}
```

2. 用一个 while 循环计算 8 的阶乘。

答案

1. a. 没错，但这是一个无限循环，会输出 Good morning 无限次，因为常量 5 代表真。

```
b.while(5<a){printf("a=%d\n",a);a--;}
```

c. 没错，但循环体不会被执行。

课程 4.9 do-while 循环

主题

- do-while 循环的控制结构
- do-while 循环和 while 循环的不同

目前已经讲解了 while 循环，在 C 语言中第二种循环是 do-while 循环，do-while 循环是 while 循环的一种轻微变体。

在本课的程序中使用了两种不同的 do-while 循环。检查第一个 do-while 循环和输出，看一看循环被执行了多少次？

虽然形式上不同，第二个 do-while 循环在运行上与第一个类似。它被执行了多少次呢？这个 do-while 循环的检验表达式曾经为真吗？如果这个 do-while 循环写成 while 循环的格式，它会被执行多少次呢？

源代码

```
#include <stdio.h>
void main(void)
{
    int i=4, j=1;

    do
    {
        printf("old_i=%2d, ",i);
        i--;
        printf("new_i=%2d\n",i);
    } while(i);

    do ++j
        while(j>999);

    printf("j=%2d\n",j);
}
```

检验表达式

语句块被重复运行直到检验表达式为假

检验表达式永远为假

输出

```
old_i= 4, new_i= 3
old_i= 3, new_i= 2
old_i= 2, new_i= 1
old_i= 1, new_i= 0
j= 2
```

解释

1）do-while 循环的结构是什么？如果只有一个语句，do-while 循环可以被写成下面的形式：

```
do 语句 while(表达式);
```

通常，do-while 循环的结构是：

```
do
{
  statement1;
  statement2;
  . . .
}
  while (expression);
```

其中括号中的语句至少被执行一次，不管表达式为真或者为假。然后表达式，通常是关系表达式（包含变量、常量、数学表达式）的值被判断为真或者为假。如果为真，括号中的语句被再次执行。如果循环体中只包含一个语句，那么括号不是必需的！

在本课的第一个 do-while 循环中，当 i=3、2 和 1 时，循环体语句被执行。当 i=0 时，检验表达式为假，循环终止。

在第二个 do-while 循环中，循环体语句被执行一次。其后，j>999 检验表达式为假，因为 j 值为 2，循环终止。

请记住，在 do-while 循环的语句中，我们应该增加或者修改那些出现在检验表达式中的变量，以避免出现无限循环。

2）do-while 循环和 while 循环有什么不同？do-while 循环和 while 循环很相似。你可以用 while 循环代替 do-while 循环，或者反向代替。为此，需要调整循环体中的语句。这两种循环的不同之处在于，在 while 循环中，检验表达式被首先检验，如果结果为假，循环体不会被执行。但是在 do-while 循环中，循环体总是先被执行一次。其后，检验表达式被检验，如果结果为假，循环体不会被执行。

概念回顾

1）在一次迭代中，do-while 循环执行并检验表达式，而 while 循环在执行之前检验表达式。

2）do-while 循环和 while 循环允许我们设计出更加简洁和清晰的循环。

练习

1. 找出下面语句中的错误并修改（假设 a 是 int 并且 a=1）。

```
a. do printf("Good morning\n"); while(5);
b. do {printf("a=%d\n",a); a++;} while (a<5);
c. Do {printf("a=%d\n",a); break;} while (a>5)
```

2. 运用 do-while 循环生成下面的表（x 是 doulbe 类型）。

```
  X       X*X     X+X
 1.0     1.00    2.00
 1.5     2.25    3.00
 2.0     4.00    4.00
 ...
10.0   100.00   20.00
```

3. 利用 do-while 循环计算 8 的阶乘。
4. 利用 do-while 循环和下面的公式计算 π 的值。讨论准确性以及收敛的速度。

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\ldots=\frac{\pi}{4}$$

答案

1. a. 没有错误，但这是一个无限循环（在这个程序执行的时候，Good morning 会被输出无限次）。

```
b.do {printf("a=%d\n",a); a++;} while (a<5);
c.do {printf("a=%d\n",a); break;} while (a>5);
```

课程 4.10 简单 for 循环

主题

- for 循环的控制结构
- 在 for 循环中使用计数器
- while 循环和 for 循环

当你知道一个操作需要被执行多少次时，for 循环是非常适合的。例如，当你知道一个操作需要在一周的每一天执行一次，那么你就知道操作需要被执行 7 次。或者你的程序需要从 10 到 0 的倒计时，那么你知道操作会被执行 11 次。在上面这些情况下，for 循环是非常适合的。

在下面的程序源代码中，使用了两种不同的 for 循环控制结构。在输出中查看 printf 语句是如何被反复执行的。查看紧跟着关键词 for 的括号中的三个语句，这些语句控制着循环的运行。你能看出这些语句和循环运行方式之间的关联吗？（提示：第一个语句指出循环如何开始，第二个语句指出循环如何结束？第三个语句表示循环如何从开始到结束。）在第一个循环中，注意分号的位置。在第二个循环中，注意括号的用法。你能解释为什么在一个循环中使用括号而在另外一个循环中没有使用括号吗？

源代码

```
#include <stdio.h>
void main(void)
{
    int day, hour, minutes;

    for(day=1; day<=3; day++)
        printf("Day=%2d\n", day);

    for (hour=5; hour>2; hour--)
    {
        minutes = 60 * hour;
        printf("Hour = %2d, Minutes=%3d\n",hour, minutes);
    }
}
```

初始化　检验表达式　"增加"表达式　循环体

输出

```
Day= 1
Day= 2
Day= 3
Hour = 5, Minutes=300
Hour = 4, Minutes=240
Hour = 3, Minutes=180
```

解释

1）什么是 for 循环？ for 循环是另外一种迭代控制结构。例如语句：

```
for (day=1; day<=3; day++)
        printf("Day=%2d\n", day);
```

使得 printf() 函数三次显示 day 的值，也就是说，从 day 等于 1 到 day 等于 3。for 循环使用下面的格式：

```
for (loop_expressions)
    single statement for loop body;

或者
for (loop_expressions)
{
    for_loop_body
}
```

其中循环表达式如 `day=1;day<=3;day++` 被分号分隔（不是逗号，这是一个经常犯的错误）。循环表达式被包含在一对括号中，并且在括号的末尾没有分号。循环表达式通常包含以下三个部分：

a）初始化表达式初始化循环变量（或计数器）并且告诉程序开始这个循环的位置。

b）循环重复条件用作验证表达式，当验证表达式为假时，程序循环结束。

c）增加表达式用来增加或减少控制变量，增加可以是正方向（增加）或者是负方向（减少）。

与本例中作用相同的 while 循环如下：

```
day=1;
while (day <= 3)
{
    printf ("Day=%2d\n", day);
    day++;
}
```

图 4-11 概略地描述了 for 循环的结构，从这个图中可以观察到，初始化表达式在第一次循环中运行。在循环运行的其他时候，增加表达式代替初始化表达式被执行。同时，当条件判断为假时，循环体不被执行并且退出循环。

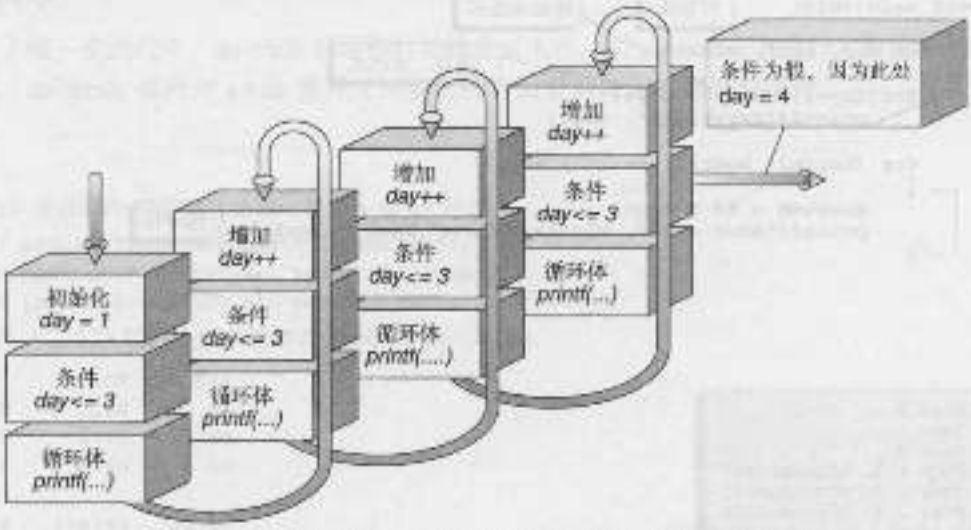


图 4-11 for 循环的执行顺序（以本课中的第一个 for 循环为例）

2）在 for 循环体中，我们能改变计数变量的值吗？可以，但是我们并不推荐这么做。计数变量是用来控制循环的，所以它的值应该在增加表达式中被修改，而不是在循环体中。如果同时在增加表达式和循环体中修改计数变量的值，会非常容易造成错误。

例如，下面的循环貌似是“正确”的。但这个循环是一个永远不会停止的无限循环。每一次循环过程中循环表达式增加了计数变量 k 的值，但是在循环体中把它重新赋值为 1。

```
for (k=1;k<3;k++)   k=1;
```

如果程序中有这样一个循环，你会发现程序永远不会结束运行。你可以尝试一下，当你离开电脑几天后，你会发现程序还在运行。不要在循环体中修改计数变量的值，这样就可以有效避免这个错误。

3）for 循环和 while 循环有什么不同之处？通常，while 循环和 for 循环在执行顺序上是相同的。可以用一个 for 循环来代替一个 while 循环，但是循环体必须是不同的。两者之间的不同如下：

项目	for 循环	while 循环
初始化表达式	是循环表达式的一部分	必须在循环之前给出
检验表达式	是循环表达式的一部分	是循环表达式的一部分
增加表达式	是循环表达式的一部分	必须在循环体内
当迭代次数已知	使用时非常简单和清晰	不是非常简单和清晰
当迭代次数未知	不是非常简单和清晰	比 for 循环更方便

概念回顾

1）任何一个 for 循环总可以用一个 while 循环代替。

2）作为一个通用的准则，当知道迭代的次数时用 for 循环，当不知道迭代的次数但知道何时结束迭代时，用 while 循环。

3）通常 for 循环依赖计数器来计算迭代的次数。

练习

1. 判断真假：

a. 在一个 for 循环中，开始时计数器的值必须小于结束时计数器的值。

b. for 循环中 3 个循环表达式之间必须用逗号分隔。

c. 一些程序员不使用 break 语句从 for 循环中跳出。

d. while 循环并不需要更改它的循环体就能代替 for 循环。

2. 找出下列语句中的错误。

```
a. for day=1,3,1
b. for (day=1,day<3,day++)
c. for (day=10;day<=20;day++);
d. for (day=10;day<5;day++)
e. for (day=100;day<100;day--)
f. for (day=10;day>100;day--)
g. for (i=20;i>10;i--) i=i*3;
```

3. 用一个 for 循环来构建下面的转换表：

inch	feet	metre
1	0.0833	0.0254
2	0.1667	0.0762
3	0.2500	0.0762
...		
100	8.3333	2.5400

答案

1. a. 假　　b. 假　　c. 真　　d. 假

2. a. `for (day=1;day<=3;day++)`

   b. `for (day=1;day<3;day++)`

   c. 没有错误，分号代表一个不做任何事情的空语句，这个空语句被执行了10次。

   d. 没有错误，但是循环体不会被执行。

   e. 没有错误，但是循环体不会被执行。

   f. 没有错误，循环体会被执行一次。

   g. 没有错误，但它是一个无限循环。

课程4.11　嵌套for循环

主题

- 嵌套for循环的控制结构
- 好的编程风格
- 调试循环结构

本课程序中使用的循环是“嵌套”的，“嵌套”代表着一个循环中包含另外一个循环。嵌套循环本质上是自内向外被执行的。查看输出的i和j的值。你能推导出嵌套循环是如何被执行的吗？

源代码

```
#include <stdio.h>
void main(void)
{
    int i, j, k, m=0;

    for (i=1; i<=5; i+=2)
    {
        for (j=1; j<=4; j++)
        {
            k = i+j;
            printf("i=%3d, j=%3d, k=%3d\n", i, j, k);
        }
        m=k+i;
    }
}
```

外层循环

内层循环

输出

```
i= 1, j= 1, k= 2
i= 1, j= 2, k= 3
i= 1, j= 3, k= 4
i= 1, j= 4, k= 5
i= 3, j= 1, k= 4
i= 3, j= 2, k= 5
```

```
i= 3, j= 3, k= 6
i= 3, j= 4, k= 7
i= 5, j= 1, k= 6
i= 5, j= 2, k= 7
i= 5, j= 3, k= 8
i= 5, j= 4, k= 9
```

解释

1）循环表达式 i+=2 的作用是什么？在本课的外层循环中，这个增加表达式在每次循环的过程中把 i 的值增加 2。

你会发现并不是所有的循环中都需要利用加法来写增加表达式。例如，i*=2 也是一个合理的增加表达式。具体使用那种方式完全取决于你要解决的问题。

2）什么是嵌套 for 循环？一个嵌套的 for 循环至少有一个 for 循环在另外一个 for 循环的里面。每个循环就像单独的一层，并且有自己的计数变量、循环表达式和循环体。在嵌套循环中，最外层计数器每一次增加，内层的循环体就被完整地执行一次。这意味着内层的循环要比外层的循环体执行得更频繁。本课的程序中有两个计数器变量 i 和 j，i 为外层循环的计数器变量，j 为内层循环的计数器变量。当 i=1、3 和 5 时，外层循环被执行 3 次。对每一个 i 值，内层循环被执行 4 次。内层循环中，每一次的 j 值分别是 1、2、3 和 4。因此内层循环一共执行了 3×4=12 次。本课中嵌套 for 循环的概念讲解如图 4-12 所示。从这个图中可以看出内层循环的 j 值是如何变化的，并且可以看出，在内层循环的时候 i 的值是不变的。这个图也演示了控制流如何从外层循环到内层循环，并从内层循环退回到外层循环。

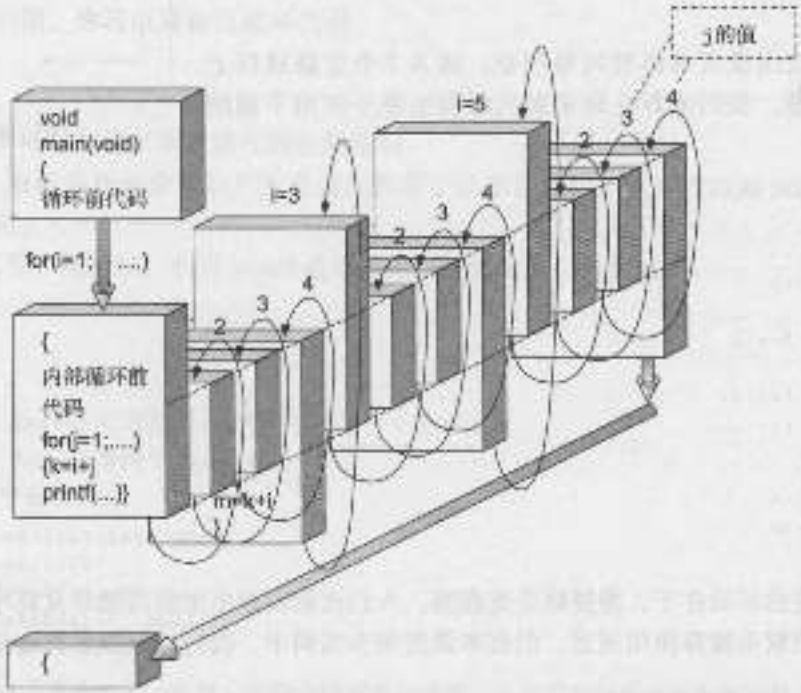


图 4-12 本课中的嵌套 for 循环。没标注的数字是 j 的值。为了简单，没有给出验证表达式和它们正确的位置

通常，一个嵌套的循环语法如下：

```
for(循环表达式1)
{
    循环体_1a              /*外层循环的循环体是可选的*/
    for(循环表达式2)       /*这是内层循环*/
    {                      /*内层循环开始*/
        循环体2            /*内层循环体*/
    }                      /*内层循环结束*/
    循环体1_b              /*外层循环的循环体是可选的*/
}                          /*外层循环结束*/
```

循环体1a和循环体1b是外层循环的循环体。它们是可选的，可以为空。与此相似，循环体2为内层循环的循环体。

3）嵌套for循环中可以有几层嵌套？ANSI C要求一个编译器至少支持15层的迭代控制结构。你的编译器也许支持更多。在通常的情况下，你可能不会用到15层的嵌套循环。

4）针对循环控制结构有哪些允许的嵌套模式？嵌套循环模式没有任何限制，只要循环不交叉（用来界定循环体界限的大括号不交叉）。图4-13简略演示了嵌套循环的一些可能表达方式。嵌套循环的具体使用方式完全取决于你要解决的问题。

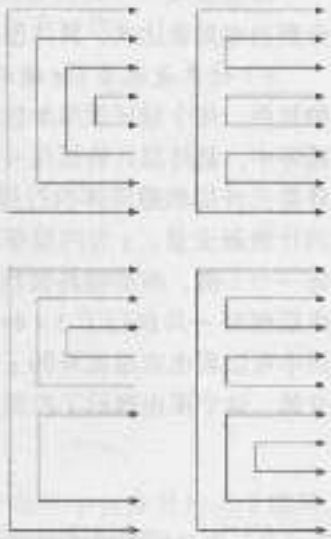


图 4-13

5）在使用嵌套循环的时候，如何让自己的程序更易读？为了让自己的程序更易读，程序员通常使用缩进。缩进对程序的性能没有任何影响，但是它是书写程序时使用的一个标准的方法。

缩进的使用没有具体的风格规范。通常3个空格或以上被认为是缩进。我们推荐在你未来的编程生涯中使用下面的风格：

```
for (...)
{
    ......
    ......

    for (...)
    {
        ......
        ......
    }

    ......
    ......
}
```

使用缩进的好处在于，即使缺乏流程图，人们也能识别出嵌套的循环及程序的流程。虽然出版界的限制不推荐使用缩进，但在本课的很多实例中，我们已经很多次地使用了缩进这种编程风格。

6）如果循环不能正常工作，如何进行调试？对于循环体中的变量以及计数器，可以制作一个表来记录它的每次循环中的具体数值。然后跟踪源代码，核对具体执行时每次的数值。例如，针对本课的程序，表可以如下所示。

j	i	k	m
1	1	2	0
1	2	3	0
1	3	4	0
1	4	5	6
3	1	4	6
3	2	5	6
3	3	6	6
3	4	7	10
5	1	6	10
5	2	7	10
5	3	8	10
5	4	9	14

制作这样的一个表格对于你开始写程序也是有所帮助的，你可以开始写一个嵌套循环，然后填充如图所示的值。如果那些值是你所期望的值，你会对已经正确地写出循环而充满自信。

当调试代码时，你可以在循环里面写上 printf 语句，以输出每一步变量和计数器的值。如果在循环的内部和外部都放上 printf 语句，那么将输出如表格所示的值。

编译器的调试选项（如果有的话）在跟踪循环时是非常有用的。你可以设立某个标志，以便程序能在循环里面的某个位置暂停。暂停的时候你就可以查看计数器和变量的值。如果发现了不期望的值，你可以重新写循环代码。

概念回顾

1）for 循环可以嵌套以形成迭代的控制结构。

2）采用良好的编程风格可以产生易读的和易于追踪的代码，这样可以减少维护和修改错误代码的时间。

3）在调试程序的时候，利用 printf 语句输出中间值帮助进行调试。

练习

1. 判断真假：

a. 在 C 语言中 `day*=5` 可以用在 for 循环中的第三个表达式里

b. 在 C 语言中 `day+=5` 等同于 `day=day+5`

2. 找出下面语句中的错误：

```
a. for (i=1;i<3;i++) i=i+100;
b. for (i=1;i<3;i++)
   for (j=100;j>10;j/=5) k=i+j;
c. for (i=1;i<3;i++)     k=i+10;
   for (j=100;j>10;j/=5) k=i+j;
```

3. 手工计算下面这段程序的值，生成一个包含所有值的表格，然后运行程序检验你的结果。

```
void main(void)
{
  int a,b,c;
  for (a=1;a<3;a++)
  {
```

```
        c=a;
        for (b=1;b<3;b++) c+=a+b;
        c+=10;
    }
}
```

4. 编写一段程序，计算前10个正整数的平均值、平均值方差和标准方差。公式如下所示：

$$m=\sum_{i=1}^{n}\frac{x_i}{n}$$

$$D=\frac{\sum_{i=1}^{n}(x_i-m)^2}{n}$$

$$s=\sqrt{D}$$

5. 运行下面的程序，讨论使用不同的变量类型来作为计数变量时带来的不同的效果。请记住推荐你使用整型数作为计数变量。

```
#include <stdio.h>
void main(void)
{
 int i, isum1=0, isum2=0, N=9999;
 float f, x, sum1=0.0, sum2=0.0;

 x=1.0/N;
 printf("x=%f\n",x);

 for (i=1;i<=N;i++) {sum1 +=x; isum1+=1;}
 printf("sum1=%f, isum1=%d\n",sum1, isum1);

 for (f=x;f<=1.0;f+=x) {sum2+=x; isum2+=1;}
 printf("sum2=%f, isum2=%d\n",sum2, isum2);
}
```

答案

1. a. 真　　b. 假。
2. a. 没有错误。但是不推荐在循环体的内部修改计数变量i。
   b. 没有错误。这是一个嵌套循环。如果检验表达式有一个更小的比较值的话，用一个整数除以另外一个整数会造成小数部分丢失。
   c. 没有错误。这不是一个嵌套循环。

应用程序 4.1　梁交叉——if-else 控制结构

问题描述

两个长横梁要彼此交叉地放到一个桥上。可以用三角学中的截距和斜率来表示它们的放置方法。为了正确地放置连接螺栓，我们需要知道交叉点的坐标。写程序找出一对直线的交叉点。假设这对横梁非常长，如果不平行放置，那么它们一定会相交。输入数据包括截距和斜率。输入数据来自于文件 INTSECT.DAT，文件包含下面的两行。

```
1  m1    b1
2  m2    b2
```

其中，m1= 第一条线的斜率

b1= 第一条线的截距

m2= 第二条线的斜率

b2 = 第二条线的截距

输出就是交叉点的 x 和 y 坐标，并将它们输出到屏幕上。

解决方法

1. 相关公式

我们用 x 和 y 描述一条直线的公式如下：

$$y = mx + b$$

其中，m= 斜率

b= y 的截距

如果 m1、b1、m2、b2 如上面定义，两条直线的公式定义如下：

$$y = m1x + b1 \quad (4.1)$$

$$y = m2x + b2 \quad (4.2)$$

两条线的交叉点可以通过联立两个公式来获得：

$$m1x + b1 = m2x + b2 \quad (4.3)$$

$$x = \frac{b2 - b1}{m1 - m2} \quad (4.4)$$

$$y = m1x + b1 \quad (4.5)$$

2. 算法

在这个特定的例子中，算法可以用以下步骤来书写：

读入斜率和截距的值

计算交叉点的 x 和 y 坐标

输出 x、y 的值

算法并没有考虑两条直线平行的情况（m1=m2）。在这种情况下将没有交叉点。为了应对这种情况，我们修改算法如下：

读入斜率和截距的值

如果两条线不平行

　　计算交叉点的 x 和 y 坐标

　　输出 x、y 的值

3. 源代码

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  double  x, y, m1, m2, b1, b2;
  FILE   *inpt;

  inpt = fopen ("INTSECT.DAT", "r");

  fscanf (inpt, "%lf %lf", &m1, &b1 );
  fscanf (inpt, "%lf %lf", &m2, &b2);

  if (m1 = m2)
  {
```

声明所有变量

声明文件指针

打开文件以便读取

从文件中读入2行

```
    x = (b2 - b1) / (m1 - m2);
    y =  m1 * x +  b1;

    printf ("\n\The slopes and intercepts are:\n\n"
            "m1  =  %lf \n"
            "b1  =  %lf \n"
            "m2  =  %lf \n"
            "b2  =  %lf\n",m1,b1,m2,b2);

    printf ("\nThe intersection point is (%lf, %lf).", x,y);
  }

  fclose(inpt);
}
```

根据公式计算交叉点的坐标

关闭输入文件

输出结果

4. 输入文件 INTSECT.DAT

```
2  -3
5   1
```

5. 输出

```
The slopes and intercepts are
m1  =  2.000000
b1  =  -3.000000
m2  =  5.000000
b2  =  1.000000
The intersection point is (-1.333333, -5.666667).
```

修改练习

1. 修改程序，当发现输入的两条线平行时，输出一条错误信息。
2. 修改程序，当发现输入的两个斜率相等时，提示用户从键盘输入一个新的斜率，以便解决平行的问题。
3. 修改程序以便能处理四对直线。这样就能发现4个交叉点。用一个循环结构来解决这个问题。
4. 修改程序以便能处理若干条直线，输入文件如下所示：

```
n
m1 x1
m2 x2
m3 x3
.
.
.
mn xn
```

应用程序 4.2　面积计算——for 循环

问题描述

计算4个不同的直角三角形的面积。作为演示，for循环可以重复执行C语句。因此它们能够用于第3章执行相同任务的应用程序中。

这个应用程序用一个for循环来执行应用程序3.1中的相同的任务，计算4个不同直角三角形的面积。

查看应用程序3.1中的程序部分，然后在应用程序4.2中观察使用了for循环的程序部分。

仔细比较这两个程序，逐条语句跟踪应用程序4.2的程序流程。用你的计算器计算出相

应的值并填入到下面的表格中：

i	面积	水平边	垂直边

要特别注意水平边和垂直边变量的使用。你可以看到它们在 for 循环之前被初始化。一旦进入 for 循环，面积被首先计算并输出。然后新的水平边和垂直边的数值被计算出来。注意这些水平边和垂直边的新值是如何被保持以及如何在循环中第二次被用来计算面积的。在循环中第二次被计算出的水平边和垂直边的值在循环的第三次计算中被保持并计算面积。循环中的第四次的面积计算是基于第三次循环中计算出的新的水平边和垂直边的值。而在循环第四次中计算的新的水平边和垂直边的值最终并没有被用到。

这个程序比应用程序 3.1 短很多。另外，遵循相同的边计算模式，它也可以很容易被修改成计算 50 个三角形的面积。

1. 源代码

```
#include <stdio.h>
void main(void)
{
   int    i;
   double horizleg, vertleg, area;

   horizleg  = 5.0;
   vertleg   = 7.0;

   for ( i=1; i<=4; i++ )
   {
       area = 0.5 * horizleg * vertleg;
       printf ("Triangle area number %d = %6.2lf \n",i, area);

       horizleg += 1.0;
       vertleg  /= 2.0;
   }
}
```

声明所有变量

用循环来计算 4 个面积

在循环的过程中每次修改边的长度

2. 输出

```
Triangle area number 1 = 17.50
Triangle area number 2 = 10.50
Triangle area number 3 =  6.12
Triangle area number 4 =  3.50
```

注释

这个程序并没有使用 First、Second、Third 和 Fourth 这些词。我们输出的是数字 1、2、3、4。这是因为 printf 语句被包含在 for 循环中。在 for 循环中执行的那些语句中只有变量可以被修改。因此我们输出计数变量 i 来代表正在进行计算三角形。

第 7 章中将学习使用字符变量以及如何把一个词当成一个字符串。

修改练习

1. 修改程序以执行下面的任务：

a. 按照相同的模式计算 30 个三角形的面积。注意值会变得很小。你如何才能输出至少 4 位的有效数字？

b. 在每次循环中垂直边的长度都加倍，按照这种模式计算三角形的面积。

c. 每次循环中水平边的长度递减5%，垂直边的长度递增3%，按照这种模式计算40个三角形的面积。

d. 计算20个梯形的面积，分别读入高、底边长和顶边长。使用你自己的变量名字，注意遵循以下的模式：底边每次循环中递减3%，顶边每次循环中递增8%，高每次递增2%。

应用程序4.3 温度单位转换——for循环

问题描述

写程序生成一个表格，包含摄氏温度以及与之对应的华氏温度。以0摄氏温度开始到100摄氏温度结束，以每20摄氏温度递增。

解决方法

1. 相关公式

相关公式如下：

```
F = C*(9/5) + 32
```

其中F为华氏温度，C为摄氏温度。

2. 算法

因为要以表格的形式输出结果，所以在for循环之前必须输出表格的题头。算法如下：

初始化摄氏温度
循环处理所有的摄氏温度
  计算与之对应的华氏温度
  输出结果以生成表格

源代码按步骤遵循这一算法。通读以下代码看看它是如何实现的。另外仔细遵循for循环以便理解它是如何执行的。变量i的目的是什么？变量degC在初始化的时候值是什么？这个初始化的值如何影响循环计算的顺序？

3. 源代码

```
#include <stdio.h>
void main(void)
{
   int i, degC;
   double degF ;

    printf ("Table of Celsius and Fahrenheit degrees \n\n"
            "Degrees              Degrees     \n"
            "Celsius              Fahrenheit \n");

    for (degC=0; degC<=100; degC+=20)
    {
        degF  = degC * 9/5.0 +32;
        printf ("%5f   %7.2f\n",   degC, degF);
    }
}
```

循环的计数器变量必须被声明

在循环的每次执行过程中递增变量degC的值

4. 输出

```
Table of Celsius and Fahrenheit degrees
Degrees      Degrees
Celsius      Fahrenheit
  0.00        32.00
 20.00        68.00
 40.00       104.00
 60.00       140.00
 80.00       176.00
100.00       212.00
```

修改练习

1. 修改程序以执行下面的任务：

a. 生成一下表格：以摄氏 0 度开始，以摄氏 100 度结束，每次递增 1 度。

b. 生成一个表格将华氏温度放在左侧，华氏温度每次递增 5 度，从 250 度递增到 1300 度。

应用程序 4.4　温度单位转换——循环和 if-else 控制结构

问题描述

写一个程序将输入的摄氏温度和华氏温度互相转换。当输入一个负数时程序终止。

解决方法

1. 相关公式

按应用程序 4.3 中的公式：

```
F = C*(9/5) + 32
```

其中 F 为华氏温度，C 为摄氏温度。另外程序需要将华氏温度转换为摄氏温度，相关公式如下：

```
C = (F-32) *5/9
```

2. 算法

根据问题描述，我们需要一个循环来处理每次输入的值。另外程序需要区别出进行哪种转换。因此算法如下：

```
读入用户输入
循环处理直到输入一个负数
        如果输入的是摄氏温度
        计算与之对应的华氏温度
        否则如果输入是一个华氏温度
        计算与之对应的摄氏温度
```

源代码按步骤遵循这一算法。通读以下代码看看它是如何实现的。另外仔细遵循 for 循环以便理解它是如何执行的。两组不同的 printf 语句和 scanf 语句有什么不同？为什么使用 else if 而不使用 else？如果只使用 else 的话，那么输入需要满足什么样的假设？

3. 源代码

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

```
{
    int input;
    char cORf;
    double convertedDeg;

    printf ("Enter a degree (use C to denote Celsius,"
            "F to denote Fahrenheit), negative value to"
            "stop the process \n");
    scanf ("%d%c", &input, &cORf);

    while (input >= 0)
    {
        if (cORf == 'C')
        {
            convertedDeg = input * 9 / 5.0 + 32;
            printf ("%7.2f F\n", convertedDeg);
        }
        else  if (cORf == 'F')
        {
            convertedDeg = (input - 32) * 5 / 9.0;
            printf ("%7.2f C\n", convertedDeg);
        }
        printf ("Enter a degree (use C to denote Celsius,"
                "F to denote Fahrenheit), negative value to"
                "stop the process \n");
        scanf ("%d%c", &input, &cORf);
    }
}
```

这个语句只执行一次

这个语句可能执行大于一次

4. 输出

```
Enter a degree (use C to denote Celsius, F to denote
Fahrenheit), negative value to stop the process
18F
-7.78 C

Enter a degree (use C to denote Celsius, F to denote
Fahrenheit), negative value to stop the process
123C
253.40 F

Enter a degree (use C to denote Celsius, F to denote
Fahrenheit), negative value to stop the process
-20C
```

注释

第一组的 printf 和 scanf 语句只执行一次，因为它们写在循环的外面。第二组的 printf 和 scanf 语句写在循环的里面，循环运行几次它们就会被执行几次。

注意，这里我们使用 else if 语句是为了确保用户输入一个合法的温度。只有“C”和“F”被认为是合法的。如果我们只使用 else 语句，则假设用户总会输入一个合法的标识符。

修改练习

修改程序以执行下面的任务：

a. 允许用户输入“C”和“c”来代表摄氏温度，“F”和“f”来代表华氏温度。

b. 如果用户输入的数值不合法，也就是说用户输入一个除“C”和“F”以外的字母，输出一个错误信息。

应用程序 4.5 仿真

问题描述

在工程领域，仿真被用来推导出答案而不用直接计算。在下面的例子（如图 4-14 所示）中，我们想找出接触两个钢条的最小圆盘的半径（中心位于原点）。也就是说我们想找出最小的 x^2+y^2 满足约束条件 $xy^2=54$。

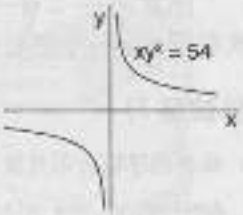


图 4-14 遵循 $xy^2=54$ 分布的两个钢条

解决方法

观察给定的图，我们不需要考虑所有的 x 值，只考虑从 0 到 10 范围内 x 的值，使用不同的 x 的值以发现最小的半径满足给定的约束条件。

1. 算法

我们没有直接使用公式，只考虑到从 0 到 10 之间分布的所有的点，以查找一个距离原点最近的点。算法如下：

包含文件
声明变量
x 从 0.1 开始到 10 以 0.01 为步长递增
  根据约束条件计算对应的 y 值
  计算 (x, y) 和原点的平方距离
  如果这个距离小于当前的最小值，记录这个距离以及对应的 x、y 点坐标
  输出结果

2. 源代码

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

void main ()
{
    double x, y;
    double delta=0.3;
    double distance;
    double min=-1;

    for (x=0.1; x<10; x+=delta)
    {
        y = sqrt (54/x);
        distance = x*x+y*y;
        if ((min == -1) || (min > distance))
            min = distance;
    }
    printf ("radius of the disc: %.1f\n", sqrt(min));
}
```

delta 是工作区间（0,10）之间的 x 值的采样间距

min 初始化为 -1

如果这是第一次迭代（如 min 是 -1），记录下距离值

3. 输出

```
Radius of the disc = 5.2
```

注释

在当前的情况下，通过计算有限的数来发现一个近似的解决方案。注意 delta 的取值会决定循环的次数。更小的 delta 的值会给出更近似标准答案的结果。

注意 min=-1 有一个特殊的目的：min 被赋值为 -1，以便开始第一次循环。循环的第一次赋于 min 一个更有意义的值，即当前的最小距离。

修改练习

1. 修改程序输出圆盘接触到钢条的点的坐标。基于以上信息，输出将会如下所示：

```
Radius of the disc=5.2, touching the strips at
(3.0, 4.2) and (3.0, -4.2)
```

应用程序 4.6 工程经济学——嵌套 for 循环

问题描述

工程经济学的主要部分就是管理资金。写一个程序计算 5 年间每个月底银行账号的总额。这个账号最开始有 5000 美元，并且没有存入和取出。利息按月计算。年利率由键盘输入。输出到一个文件（MONEY.OUT），文件中包含一个表格列出所有年和月的账号余额。

解决方法

1. 相关公式

如果年利率是 i，那么月利率是 $i/12$。每个月的利息将是

$$I = P_o(i/12) \tag{4.6}$$

其中，I = 月利息

P_o = 每月开始时账号余额

i = 年利率

每个月底的余额为：

$$P_f = P_o + I = P_o + P_o\left(\frac{i}{12}\right) = P_o\left(1+\frac{i}{12}\right) \tag{4.7}$$

其中 P_f 为月底时的余额。

2. 算法

基于以上公式，算法如下：

读入年利率 i

打开输出文件

P=5000

从第一年到第五年循环

　　从第 1 个月到第 12 个月循环

　　P=P(1+i/12)

　　输出年、月、P

图 4-15 显示了程序的流程。

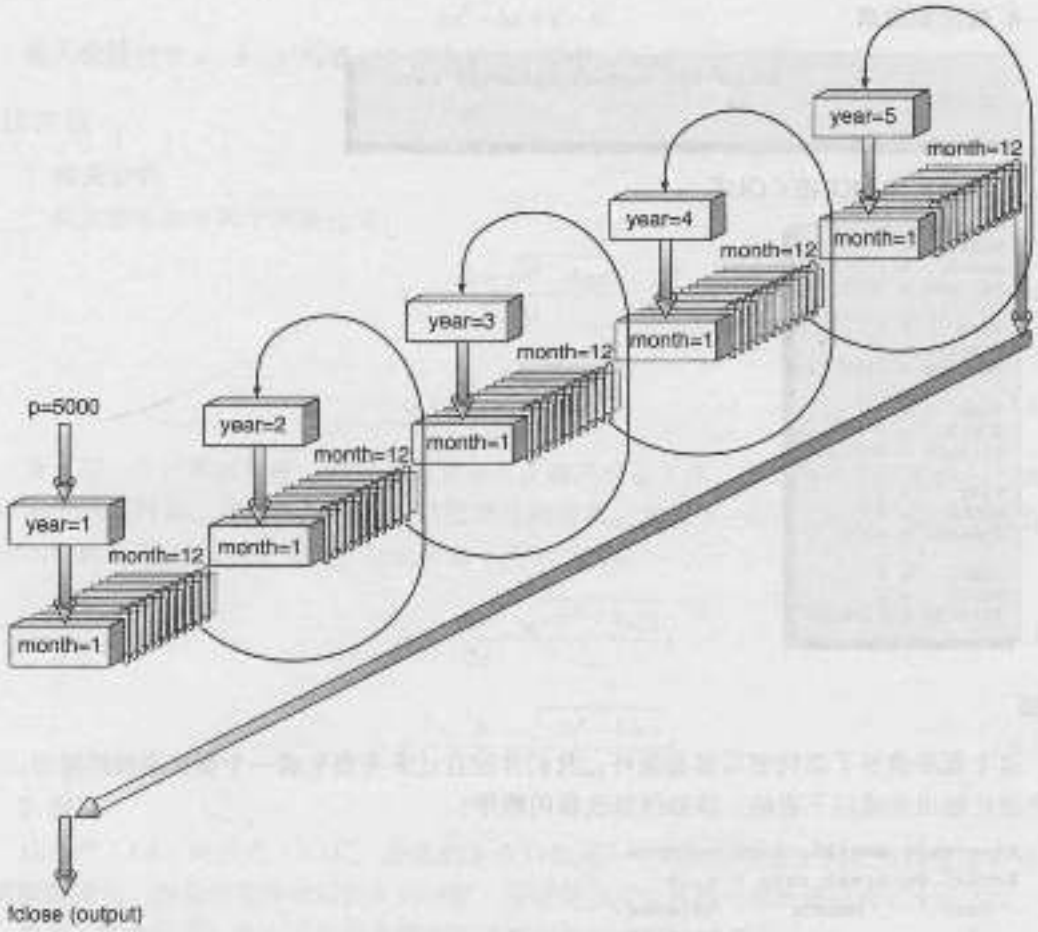


图 4-15 应用程序 4.6 的流程

3. 源代码

```
#include <stdio.h>
void main(void)
{
   int    month, year;
   double i, p;
   FILE *outpt;

   outpt = fopen ("MONEY.OUT", "wt");
   printf ("Enter the annual interest rate\n");

   scanf  ("%lf",&i);

   p = 5000.;

   for ( year = 1;year<=5;year++ )
   {
       for ( month = 1;month<=12;month++ )
       {
           p *= ( 1 + i/12 );
           fprintf (outpt, "\n Year     = %d "
              "\n Month    = %d "
              "\n Balance = %7.2lf \n\n",year,month,p);
       }
   }
   fclose(outpt);
}
```

与算法对应的嵌套循环

4. 输出到屏幕

```
                    Enter the annual interest rate
键盘输入            0.06
```

5. 输出文件 MONEY.OUT

```
Year    = 1
Month   = 1
Balance = 5025.00

Year    = 1
Month   = 2
Balance = 5050.12

Year    = 1
Month   = 3
Balance = 5075.38

Year    = 1
Month   = 4
Balance = 5100.75

Year    = 1
Month   = 5
Balance = 5126.26
```

注释

这个程序演示了如何书写嵌套循环，我们并没有过多考虑生成一个整洁高效的输出。如果你想让输出生成以下表格，该如何修改你的程序？

```
Five-year monthly bank balance
Annual interest rate = 6.0%
  Year        Month        Balance
   1            1          5025.00
                2          5050.13
                3          5075.38
                4          5100.75
                5          5126.26

     and so on, for five years.
```

修改练习

1. 修改程序执行下面的任务：
   a. 不是基于每一个月，而是基于每两个月计算利率。
   b. 基于每星期计算利率。
   c. 把计算期限延长到 10 年。
   d. 初始余额为 10 000，利率为 5%，在 8 年的时间内基于每个星期计算利率，每 6 个月打一次结果。
   e. 输出如注释部分描述的表格。

应用程序 4.7 解二次方程——if-else 控制结构（数值方法例子）

问题描述

写一个程序解决如下的二次方程：

$$ax^2+bx+c=0$$

输入数据包含 a、b、c 的值，从键盘输入。输出 x 的值并打印到屏幕。

解决方法

1. 相关公式

二次方程有如下两个解决公式：

$$x_1=\frac{-b+\sqrt{b^2-4ac}}{2a} \tag{4.8}$$

$$x_2=\frac{-b-\sqrt{b^2-4ac}}{2a} \tag{4.9}$$

为了写一个计算机程序，你应该做完备且正确的准备工作。考虑所有的可能性。二次方程可能没有实数解，你的程序必须能够处理这种情况。如果 b^2-4ac 为正数，那么可以直接用公式计算 x_1 和 x_2 的值。否则，公式将如下所示（其中 $i=\sqrt{-1}$）：

$$x_1=-\frac{b}{2a}+\frac{\sqrt{-(b^2-4ac)}}{2a}i \tag{4.10}$$

$$x_2=-\frac{b}{2a}-\frac{\sqrt{-(b^2-4ac)}}{2a}i \tag{4.11}$$

2. 算法

从公式（4.8）到公式（4.11），公式的左边只出现一个单独的变量，因此可以使用C语言中的赋值语句。当你在程序中写公式的时候，应该使公式变成能用源代码轻松写出的方式。

算法（包含公式）和为了发现负数的平方根的检查方法如下所示：

从键盘输入a，b，c

计算b^2-4ac

若b^2-4ac为正，则

$$x_1=-\frac{b}{2a}+\frac{\sqrt{-(b^2-4ac)}}{2a}$$

且

$$x_2=-\frac{b}{2a}-\frac{\sqrt{-(b^2-4ac)}}{2a}$$

输出x_1和x_2到屏幕

若b^2-4ac为负，则

$$x_1=-\frac{b}{2a}+\frac{\sqrt{-b^2-4ac}}{2a}$$

且

$$x_2=-\frac{b}{2a}-\frac{\sqrt{-b^2-4ac}}{2a}$$

输出x_1和x_2到屏幕

3. 源代码

```
#include <math.h>
#include <stdio.h>
void main(void)
{
    double i, a, b, c, x1, x2, test, real, imag;

    printf("Enter the values of a, b and c (each separated\n"
           "by a space) then press return\n");

    scanf("%lf %lf %lf", &a, &b, &c);

    test=b*b - 4*a*c;

       if (test>=0)
       {
          x1 = (-b + sqrt(test))/(2*a);
          x2 = (-b - sqrt(test))/(2*a);
          printf(" Real result:\n x1=%10.5f\n x2=%10.5f\n\n",x1,x2);
       }

       else

       {
          real = -b/(2*a);
          imag = sqrt (-test) /(2*a);
          printf(" Imaginary result:\n"
               " x1=%10.5f %+10.5f i\n x2=%10.5f %+10.5f i\n",
                 real, imag, real, -imag);
       }
}
```

计算检验的值

如果检验的值大于零，计算两个x的值

如果检验的值小于零，计算实数部分和虚数部分

4. 输出

```
键盘输入        Enter the values of a, b and c (each separated
                by a space) then press return
                15 -2 3
                Imaginary result:
                x1=  0.06667 +  0.44222 i
                x2=  0.06667 -  0.44222 i
```

注释

程序必须能够处理所有可能发生的情况。在本例中你应该能够处理虚数的结果。作为一名程序员，你的责任就是想象出所有可能，并使程序能够处理它们。

注意源代码中使用的变量 test。这个变量的使用只是为了方便和易于查看。我们完全可以不使用这个变量。但是推荐你使用这个变量，因为这会使程序更加易读。

注意下面的包含命令

```
#include <math.h>
```

是必需的，因为这段程序中使用了求平方根的函数 sqrt()。

修改练习

1. 修改程序以便能够处理 5 个不同的输入公式。提示用户输入 5 行，每行包含 3 个参数。
2. 修改程序以便能够处理从文件输入的 5 个不同的输入公式，并把结果输出到一个数据文件中。

应用练习

4.1 抛物线公式如下：

$$y = ax^2 + bx + c$$

使用if语句和for循环，求出4个不同抛物线的最大值或最小值。程序从一个包含a、b、c的文件PARA.DAT中读入参数。

```
1    a1    b1    c1  (coefficients for parabola 1)
2    a2    b2    c2  (coefficients for parabola 2)
3    a3    b3    c3  (coefficients for parabola 3)
4    a4    b4    c4  (coefficients for parabola 4)
```

一个示例的数据文件如下：

```
1.1   -2    1
-1.3  1     15
-0.7  15    18.3
1.5   15.5  14.2
```

作为输出，用以下的格式将结果输出到数据文件PARA.OUT中。

Parabola number	Equation	(x, y) coordinates of minimum or maximum
1	$y = 1.1x^*x - 2x + 1$	(0.909,0.0909) min
2	$y = -1.3x^*x + 1x + 15$	—
3	$y = -0.7x^*x + 15x + 18.3$	—
4	$y = 1.5x^*x + 15.5x + 14.2$	—

4.2 二项式定理如下：

$$(a+b)^n = a^n + na^{n-1}b + \frac{n(n-1)}{2!}a^{n-2}b^2 + \frac{n(n-1)(n-2)}{3!}a^{n-3}b^3 + \cdots$$

写一个程序，当a=1且0<b<1时，利用二项式定理计算$(a+b)^n$到8位有效数字。同时在你的程序中使用pow()函数计算$(a+b)^n$。确保在该程序中0<b<1。（注意：这是一个收敛数列。）

输入数据来源于文件BINO.DAT。第一行包含输入的值的数目m，接下来是m行b和n的值。（b是实型数，m和n是整型数。）

```
1     m
2     b   n
3     b   n
4     b   n
      m lines of b and n


m+1   b   n
```

一个数据文件的例子如下：

```
5
0.5  8
0.2  10
0.33 5
0.08 6
0.45 15
```

输出显示

```
Binomial theorem and pow() output

a = 1        b = -----        n = -----

(a+b)^n                       (a+b)^n
From the                      From the
pow() function                binomial theorem
```

```
---                                    ---
---                                    ---
---                                    ---
---                                    ---
```

4.3 写一个程序计算从1月1号到给定日期的天数。使程序能够计算多余20个不同的日期。输入数据来源于文件DAYS.DAT。文件遵循下面的格式：

```
1    n                (计算三数个数)
2    month  day
3    month  day
n+1  month  day (所有数据是整数)
```

示例数据文件如下

```
5
12   7
8    5
1    27
4    18
7    22
```

以下面的格式输出结果到DAYS.OUT文件：

```
TABLE OF DATES AND DAYS FROM 1 JANUARY
DATE                  DAYS FROM 1 JAN
---------------------------------------
7 December                  ...
5 August                    ...
27 January                  ...
18 April                    ...
22 July                     ...
```

注意必须用词来表示月份，而不是使用数字。

4.4 写程序计算一个整数列表中所有负数的和。

输入数据来源于文件SUMNEG.DAT。文件遵循下面的格式：

```
1  n     列表中整数个数
2  int1
3  int2
4  int3
...
...
...
```

以下面的格式输出结果到屏幕：

```
The sum of the negative integers is:
...
The list of integers is:
...
...
...
...
```

4.5 某课的分数等级如下所示：

90-100 A
80-89 B
70-79 C
60-69 D
< 60 F

写一个程序根据10个不同的数值分数输出对应的分数等级。

输入数据来源于GRADE.DAT文件。文件遵循下面的格式：

```
score1 score2 score3 ... score10
```

以下面的格式输出结果到GRADE.OUT文件：

```
Numerical          Grade
  Score
 ------           -----
   ...              ...
   ...              ...
   ...              ...
   ...              ...
```

4.6 写程序在一个包含20个整数的列表中找两个最大的整数和两个最小的整数。

输入数据来源于TWOMM.DAT文件。数据文件包含：

```
1 int1   int2   int3   .... int10
2 int11  int12  int13  .... int20
```

以下面的格式输出结果到屏幕：

```
The two largest values in the list are:
 ...          ...
The two smallest values in the list are:
 ...          ...
```

4.7 计算资产折旧的常数百分比方法基于如下假设：每年年底的折旧费用是年初的账面价值的规定的百分比。这个假设推导出下面的关系：

$$S=C(1-d)^n$$

其中：

C= 原始的资产价值

D= 每年的折旧率

n= 年数

S=n 年后的账面价值

给定资产的初始价值、折旧率和有效寿命后的账面价值（残值）。写程序计算一个资产的有效寿命的年数。

输入数据来源于键盘，提示用户输入以下数据：

```
Enter the original cost and depreciation rate:
 ...      ...
Enter the book value at the end of the useful life of an asset:
 ...      ...
```

输出结果到屏幕：

```
The useful life of the asset is:   ...years
```

本章回顾

本章中，我们学习了不同的运算符，并使用if语句以及循环结构写决策控制。有不同的if结构（简单if、if块、if-else、if-else-if）来适应不同的决策控制。另外，简单的和嵌套的控制结构可以混合搭配以生成不同的逻辑流程。

函　　数

本章目标

结束本章的学习后，你将可以：

- 描述函数的基本概念和用法。
- 解释函数调用时背后的机制。
- 在函数内部区分关联的变量。
- 函数之间传递数据。
- 编写函数时遵循正确的布局。
- 把大问题分解为小问题，并用函数解决。
- 在应用程序中，构造用户自定义函数。

第 1 章讨论了软件工程，并描述了 C 语言程序中函数是如何充当一个模块组件的。如果我们继续分析建造一个结构和构建 C 程序的相似之处，我们会加深对，C 程序中函数价值的了解。

例如，假设你现在要完成一个任务，在河上建造一个步行桥。一种方法是发现一棵大树，这棵大树足够长可以横跨这条河，并且足够宽能够容纳步行交通（图 5-1）。你可以砍倒这棵树，把它切割成刚好从河的一边能到达另外一边的长度，并加工它以便在上面行走。当这样做时，你会意识到这棵大树又大又沉，很难加工和处理。所以你需要大型设备来完成这个任务。同时由于只有这一棵树，每次只能完成相对有限的任务。任何错误都无法修改。

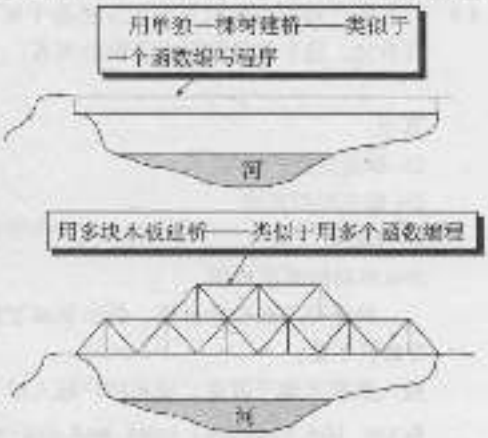


图 5-1　建桥

另外一种建桥的方法是砍下很多小树，并将其加工成木板，然后用螺钉连接木板建成这个桥。因为每一棵小树都比大树小且轻，它们非常易于加工。同时，错误只局限在一块木板上，可以通过替换有问题的木板来修正错误。很多人可以同时并行工作，可以分配一些人砍树，而另外一些人做木板。需要用到的设备也很小，并且很常用。但是与用一棵大树来建桥不同，如果木板建桥，设计变得非常重要。换句话说，你必须画桥的设计草图，以便知道每块木板要多长。同时连接也很关键，你必须清楚板子是如何链接的以及需要多少这样的链接。螺钉的类型长短都必须在订购前决定。分配任务时，必须让每个工人理解任务并了解每一块单独的木板是如何连接的。另外，必须有人负责组织，并确保当每块木板加工后，它们可以组装在一起以正确地建成这个桥。最后的结果就是，当用木板高效率地建桥时，那些需要解决的问题与用一棵大树来建桥是完全不一样的。如果可以正确地完成，那么用木板建桥的方法更好，因为它可以更快、更便宜地建造一座桥。

两种不同建桥方法的区别就像在 C 语言中，你只用一个函数 (main) 来完成程序，或者

用很多不同的函数来完成程序。目前，你只使用一个函数来编写程序（不包括使用的库函数）。本章中会演示如何书写除了 main 以外的函数。为了完成这个任务，在程序中很多新的问题将变得重要。首先，计划将变得更为核心，必须更准确地布局以便将函数连接在一起（哪些信息会被传入并传出这个函数）。当你用不同函数书写程序时，你可能会有点失望，并想回退到只用一个函数来完成这个程序。你要拒绝这种诱惑，争取用多个函数来完成程序，即使有时这并不是必需的。这样你就可以学习并克服那些使用函数带来的问题。

学习表 5-1，加深对只用一个函数来完成程序与用不同函数来完成程序之间的不同的理解。在这个表中，我们将继续用建桥来类比写程序。当学习完这个表后，你会知道为什么要用很多函数来完成一个程序。

表 5-1　建桥和程序

问题	用一棵树建桥	用一个函数（main）编程	用多块木板建桥	用多个函数编程
连接	因为只有一开，所以不存在连接的问题	因为只有一个函数，所以不存在函数间信息传递的问题	木板裁剪和钻好螺栓洞后，连接很重要，它们必须恰好吻合	函数必须彼此连接。函数间传递的信息非常重要
工作人数	砍伐树只需要很少的人	只需要很少的人编程，因为程序中一部分改变必然引起另一部分改变	需要很多人同时砍伐树和准备木板	不同的人分别编写不同的函数，使得可以更快地完成整个程序
规划	这个简单的设计减少了所需的规划	即使只有一个函数的这个程序要求计划，但是由于没有多个函数，函数之间的链接也不需要规划，这也减少了一些规划	桥的构建必须提前全面规划。必须准备好草图和时间表。所有部分必须符合整体，并且这只有精心规划才能完成	这个程序必须被全面规划。时间表必须协调好另一个函数所依赖的函数的完成时间。所有函数都必须符合创建的完整程序
需要的设备	需要大型设备来处理这棵树，这个设备不容易获得或价格昂贵	对于大规模程序，可能必须有一台非常快速的计算机，但是这样的计算机并不容易获得，或者价格昂贵	所需的设备比单棵树设计要小得多。但是所需的设备数量更多。设备容易获取并且价格不贵	即使对于大规模程序，也可以在若干小型计算机上完成，因为每个独立的函数比较小，比较容易处理。小型计算机容易获得，并且价格不贵
监督和组织	监督和组织很少，因为很难有多个任务同时被做	监督和组织很少，因为从事的人很少	必须有相当多的监督和组织来协调多个任务和涉及的不同人的努力	必须有相当多的监督和组织来协调所有程序员的工作
错误	为了修正一个错误，如在一处锯掉得太多，可能要重新锯掉整棵树	为了纠正一个地方的错误，可能要修改程序的许多部分	涉及一块木板的错误，可能只要简单地替换那块木板即可。错误可能会影响整个桥，但是在大多数情况下，可以限制在局部范围内	一个函数中的错误只要求在那个函数中修正这个错误。所犯的错误也可能引起整个程序修改，然而错误的范围可能被减小到最小
部件的可重用性	只有一座桥，除非另外一个地方恰好有相同长度和宽度要求，才可能被重用	当可以复制程序的一部分并在其他程序中使用时，在大多数情况下，必须修改后才能在另一个程序中使用	取一块木板，并且可以在不同结构的其他地方使用，不只是桥	一个函数可以被复制，并且在其他要求类似任务的程序中使用

本文中演示了很多包含函数的程序。为了描述使用和书写函数的细节，我们把这些知识分布成如图5-2所示的结构中。目前本书中已经使用过包括printf，scanf，pow等库函数，本章集中讨论由程序员定义的自定义函数。

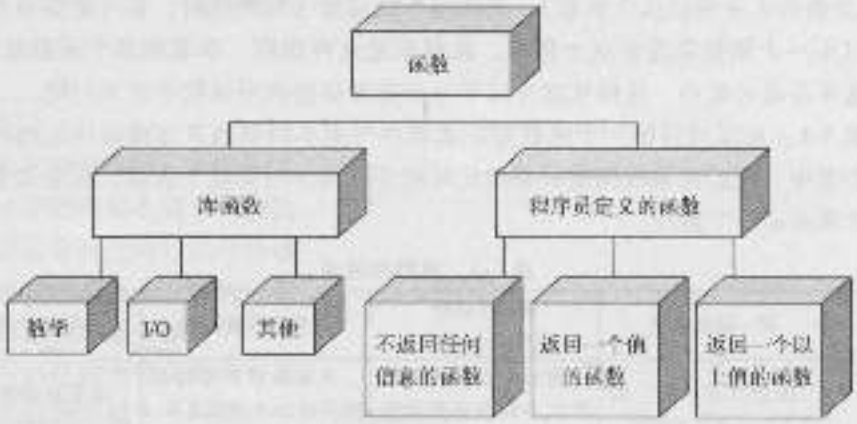


图5-2 函数分类

课程5.1 不返回值的函数

主题

- 定义并调用一个函数
- 用函数控制程序
- 在函数之间传递数值
- 在函数中使用变量
- 定制一个函数并使用函数原型
- 函数的类型

在C语言中函数和变量有很多相同点：

1）函数名和变量名都被认为是标示符，因此它们的命名都要遵循标识符的命名规则。

2）函数（像变量一样）有与它关联的类型（如整型或浮点型）。

3）像变量一样，函数和它们的类型在使用前必须在程序中声明。

本课的程序中有两个函数，function1和function2，它们的类型都是void。在下面的程序中，你能看出在哪行把两个函数声明为void类型？函数声明的位置很重要，它们与main函数的相对位置是什么？

回忆main函数也是一个函数。前面的课程中已经学习了在main函数的内部，我们可以调用其他的函数（如printf、sin和pow库函数）。为了调用这些函数，我们使用函数名后接一个左括号，然后是参数和一个右括号。在main函数中，看那两行调用了function1和function2？

注意调用function1时没有参数。这与括号中包含void的函数声明是一致的。这种声明代表没有参数从main函数传递到function1函数。调用function2时有两个参数，这也符合有两个参数的函数声明。

回想你如何编写 main 函数。函数体被包含在括号 {} 中，函数体中是执行语句。在这个程序中，找到 function1 和 function2 的函数体。注意到它们并没有位于 main 函数的里面，而是在外面。

function2 出现在 3 个不同的行，找到这 3 行。注意在函数调用语句中出现的参数与其他两行中出现的参数是不一样的。但是参数的数量（2）和类型（int 和 double）在三行中都一样。作为变量的参数在函数调用时的顺序和函数声明时要保持一致。对于 function2 中的 n 和 x 变量，main 函数中哪个变量与之对应？查看程序的输出。你能看出 main 函数中变量的值已经传递到 function2 中的变量 x 和 n 中了吗？

查看代码的 printf 语句。你能看出程序的流程吗？变量 m 被同时用在 main 函数和 function2 函数中。在这两个位置它们有相同的值吗（查看输出）？这一点说明了关于在不同函数中命名和声明变量的什么事实？

源代码

```
#include <stdio.h>

void function1(void);
void function2(int n, double x);

void main(void)
{
        int m;
        double y ;

        m=15;
        y=308.24;
        printf ("The value of m in main is m=%d\n\n",m);

        function1( );
        function2(m,y);

        printf ("The value of m in main is still m=%d\n",m);
}

function1 定义

void function1(void)
{
        printf("function1 is a void function that does not receive\n\
                \rvalues from main.\n\n");
}

function2 定义

void function2(int n, double x)
{
        int k,m;
        double z;

        k=2*n+2;
        m=5*n+37;
        z=4.0*x-58.4;

        printf("function2 is a void function that does receive\n\
                \rvalues from main.  The values received from main are:\n\
                \r\t n=%d \n\r\t x=%lf\n\n",n,x);

        printf("function2 creates three new variables, k, m and z\n\
                \rThese variables have the values:\n\
                \r\t k=%d \n\r\t m=%d \n\r\t z=%lf\n\n",k,m,z);
}
```

函数原型标示出了参数的数目和类型以及返回值的类型

调用 function1 和 function2。function1 没有参数，function2 有两个参数

函数声明或函数头，void 代表没有值从这个函数中传入和传出

函数声明或函数头，有两个参数从 main 函数中传入。void 代表没有值从这个函数中传出

赋值语句利用从 main 函数中传入的值生成新的值

输出

```
The value of m in main is m=15

function1 is a void function that does not receive
values from main.

function2 is a void function that does receive
values from main.  The values received from main are:
          n=15
          x=308.240000

function2 creates three new variables, k, m and z
These variables have the values:
          k=32
          m=112
          z=1274.560000

The value of m in main is still m=15
```

解释

1）简要说明如何调用一个函数？写一个函数名，后接一个括号，括号中包含对应的参数。例如本课程序中的代码行

```
function2(m,y);
```

调用 function2。

2）不涉及细节，函数调用时都做了什么？它把程序的控制传给函数。在本课的程序中，当在 main 中执行

```
function1();
```

之后，控制转移到 function1，并且下一行执行的代码为：

```
printf("function1 is a void function that does not receive\n\
          \rvalues from main.\n\n");
```

整个过程如图 5-3 所示。

3）当函数结束运行时发生了什么？如图 5-3 所示，控制返回到函数被调用的地方。本课程中，当 function1 结束运行时，控制返回到 function1 函数在 main 中被调用的位置。

4）用函数写应用程序的主要问题是什么？程序员主要关心的问题是在两个函数中如何正确地传递信息。换句话说，只把函数需要的信息传递给它，而不传递其他的信息。这样做的原因是，如果函数中存在一个错误，它只影响函数能得到的那些信息。这会控制错误影响的范围，有利于写出没有错误的代码。

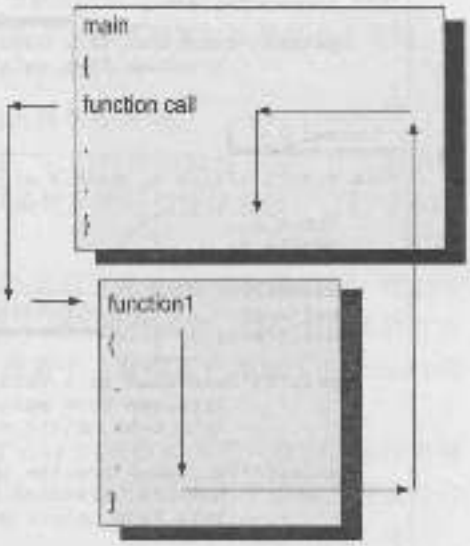


图 5-3 调用 function1 时程序流的方向

初级程序员有时会在需要传递什么给函数及如何正确传递等问题上遇到困难。为了确定函数需要什么样的信息，必须要清楚地定义出函数中包含哪些行为。请参考本章末尾的应用程序，那里我们演

示了如何计划一个实际的程序。本课集中讨论如何在函数中正确地传递信息。

5）在函数内部，目前所学到的知识都还有效吗？是的，在函数内部，你可以使用控制结构、循环、赋值语句以及很多已经学过的编程方法。

6）有哪些函数类型？函数的类型就是它返回给调用方函数的值的类型。基于此，函数类型可以是：

```
int, long int, float, double, long double, char, void
```

以及其他合法的数据类型。你可能注意到了 void，它并不返给调用方函数任何值。因此，本课程序中的 main 函数，并没有给调用方（操作系统）返回任何值。

7）本课程序中，有没有把信息从 main 函数传递给了 function1？没有，注意当调用 function1 时，括号中没有包含任何参数。因此在 main 函数中没有变量值传递给 function1。function1 不需要 main 函数中的任何变量值。它只输出一行代码，因此不需要更多的信息。

8）本课程序中，有没有把信息从 main 函数传递给了 function2？有当调用

```
function2(m,y);
```

有两个变量被包含在括号中。于是我们把 main 函数中的 m 和 y 变量的值传给了 function2。在 main 中，在调用 function2 函数的位置上，m 的值为 15，y 的值为 308.24。因此，function2 接受了值 15 和 308.24。

9）function2 中的值被保存在什么地方？在 function2 中，我们在下面的代码中定义了变量名 n 和 x

```
void function2(int n, double x)
```

函数的类型和名字是 void function2，它后面的括号中给出了这些变量的名字，所以这些变量只属于 function2，并局限于 function2。值 15 和 308.24 被保存在为变量 n 和 x 分配的内存单元里。本课后续会更多地讨论这个问题。但是同时要注意，function2 中那些等待被赋值的变量和那些要赋给值的变量的顺序的一致性。

```
function2 (m, y);

void function2(int n, double x)
```

因为括号中变量的顺序，在 main 函数中 m 的值（15）赋给了 function2 中的变量 n，在 main 中 y 的值（308.24）赋给了 function2 中的变量 x。这样就成功地把信息从 main 函数传到了 function2。在 function2 内部，可以利用变量 n 和 x 以及它们对应的值 15 和 308.24 来做我们感兴趣的事，本课中我们用这些值计算 k 和 n。

10）通常，如何定义和使用自定义函数？我们需要做三件事：

- 一个函数原型
- 一个函数定义
- 一个函数调用

11）什么是函数原型？函数原型就是一个声明，本质上指出了函数的存在并有可能被用在程序中。

函数原型有下面的方式：

r_type f_name (arg_type arg_name, arg_type arg_name, ...);

其中，arg_type = 参数类型（int、double 或者其他）

arg_name = 参数名（合法的标识符）

f_name = 函数名（合法的标识符）

r_type = 函数的返回值（void 代表没有值返回）

…代表被逗号分隔的 arg_type arg_name 对

函数原型的目的在于定义函数需要获取的参数类型以及获取的顺序。换句话说，调用函数的参数的顺序与函数原型中参数列表的顺序需要保持一致。注意函数原型后面有一个分号。例如，function2 的函数原型如下

```
void function2 (int n, double x);
```

它代表 functoin2 不返回值，并且它有两个参数。第一个参数是 integer。第二个参数是 double。本书在函数原型中给出了参数的名字。但是 ANSI C 并不要求给出参数的名字。因此也可以把函数原型写成下面的形式：

```
void function2 (int, double);
```

12）什么是函数定义？函数定义指定函数做什么。它包含一个函数体，并有如下的格式：

```
r_type f_name (arg_type arg_name, arg_type arg_name, ...)
{
    ...
    function body – C declaration and statements
    ...             using the arg_names and other variables
}
```

其中 arg_name 是函数体中用到的参数的名字。程序员需要自己开发函数体。函数体中包含可执行的语句。本课程序中，function1 主要在函数体中调用 printf 函数。function2 执行一些数学运算，并调用了 printf 语句。开发函数体以完成必要的工作是程序员的责任。注意，包含函数名和参数的行的末尾没有分号。这一行叫做函数声明或函数头。除了是否有分号这一区别，这一行和我们给出的函数的原型是一样的。但是与函数原型不同，在函数头中必须包含参数名。

C 把函数当成一个外部定义，这意味着它可以被定义在任何函数的外面。

13）什么是函数调用？函数调用就是包含一个函数名字（标识符）的表达式，并且包含被括号括起来的参数列表，如下所示：

```
f_name (exp, exp, ...)
```

其中，exp 是一个表达式

…代表逗号分隔的表达式（表达式的数目必须与函数原型中参数数目保持一致）

(exp, exp, …) 为参数列表

参数列表代表了从函数调用方向函数传递的值。例如在 main 函数中，调用 function2

```
function2 (m,y);
```

使得 main 中 m，y 的值传递给了 function2。

14）在函数调用时，参数列表中的参数数目、顺序、类型必须和函数定义时一致吗？它们必须一致。例如，对于 function2，我们有

```
原型_____________________void function2 (int n, double x);

函数头_____________________void function2 (int n, double x)

函数调用_____________________function2 (m, y);
```

注意到三者中都有两个参数，第一个是 int，第二个是 double。三者满足数目、顺序、类型一致的要求。

当在 main 函数中通过下面的语句调用 function2 时

```
function2(m,y);
```

我们把 m，y 的值传递给 function2 的 n 和 x 变量。变量的顺序不匹配是初级程序员常犯的错误。为了避免这个错误，你可以给变量起非常有描述性的名字。例如，如果你要写一个用户定义函数 pow，你可以在函数原型和函数定义中使用 base 和 exponent 的变量名。当你写这个函数的调用程序时，可以通过查看函数的原型和定义来了解参数的顺序。

另外一种避免顺序不匹配的方法是使用注释。在你的代码中使用注释，在函数的上部清楚地描述每个参数的意义，这样有助于避免顺序不匹配造成的错误。

记住，即使你输入错误的顺序，程序依旧可以运行。不要因此而高兴。这并不代表你的结果是对的。如果你发现结果不对，可以通过 printf 语句来输出函数调用前以及刚进入函数后那些参数的值，以检查它们的正确性。

这些语句应该输出你希望看到的值，如果没有，你也许会发现顺序的错误。

15）能否进一步讲解 main 函数和 function1 函数之间的关系？因为函数的原型和函数的头指出参数列表为 void，调用 function1 只是简单地把程序控制转交给 function1，并没有传递任何其他信息。当 function1 结束运行时，控制传回给 main。因此 void 类型的函数只是如图 5-4 所示的那样，转移程序的控制。

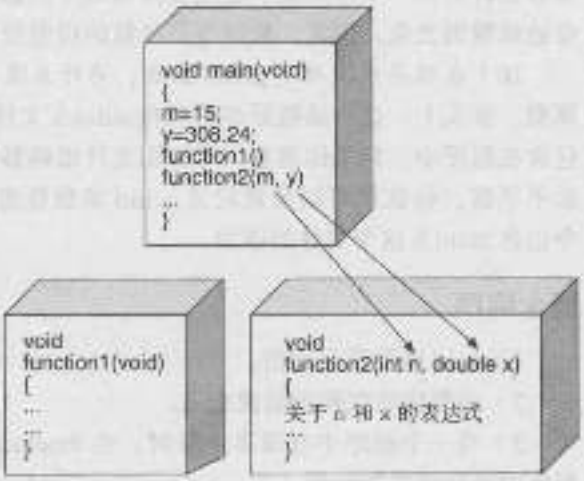


图 5-4 本课程序中的函数。注意 function1 不接受也不返回值。function2 从 main 中接受 m，y，并把它们保存在局部变量 n 和 x 中

16）是否有必要把 main 作为第一个要定义的函数？否。程序列出的所有函数中，第一个并不一定必须是 main 函数。本课中的程序，你可以写成以下的方式：

```
#include . . .
void function1(void);
void function2(int n, double x);
void function2(int n, double x)
{
        function body
}

void function1(void)
{
        function body
}

void main(void)
{
        function body
}
```

一般来说，main是第一个或最后一个定义的函数。

17）函数原型可以写到函数体而不是所有函数的外部吗？可以。但是在函数内部的函数原型只对这个函数体起到一个声明的作用。这意味着，只有这个函数体可以调用这个函数原型对应的函数。

与此相反，放到所有函数外部的函数原型对所有的函数声明了这个函数原型，这意味着程序中所有函数都可以调用这个函数原型对应的函数。注意函数原型必须出现在调用者的前面。这部分内容会在5-3课上进一步讨论。

18）在main函数中的m变量和function2中的m变量有什么关系？没关系。像我们以前讲过的，变量名的声明局限于特定的每个函数。因此，可以在不同的函数中使用相同的变量名，而它们之间没有任何联系。在本课中演示的正是这种情况。

19）当在main中调用function2时，如果传入两个整数或者两个浮点数，而不是一个整数和一个浮点数，那么会发生什么？因为function2需要一个整数和一个浮点数，第二个整型数会转换为一个浮点数。整型数转换成浮点数不会造成错误。但是把浮点数转换成整型数会造成数据丢失。因此，最好保证参数的类型完全匹配。

20）在程序中使用了printf函数，为什么没有看见它的函数原型？表面上这个函数没有原型。事实上，这个函数原型出现在stdio.h文件中，被我们用预处理命令#include<stdio.h>包含在程序中。如果你愿意，可以用文件编辑器来查看stdio.h。虽然这个文件的内容大部分你不了解，你依然可以发现定义printf函数原型的那一行。这就是为什么我们用include命令包含stdio.h这个文件的原因。

概念回顾

1）void函数不返回值。

2）函数应该在调用前被定义。

3）当一个程序中有很多函数时，在#include命令后面写函数的原型可以帮助我们处理程序中函数顺序的问题。

4）通过函数头中定义的函数参数，信息被传入到函数中。

5）传入函数的信息被保存在属于函数的局部变量中。

6）函数的类型就是它返回给调用方的值的类型。

练习

1. 判断真假：

a. 定义一个函数的目的就是避免重复写相同一组C语言语句。

b. 自定义函数可以写在main函数的前面。

c. 自定义函数可以写在main函数的后面。

d. 自定义函数可以写在main函数的里面。

e. 函数体必须包含在一对括号内。

f. 自定义函数必须要调用一次，否则C语言编译器会给出一个警告。

g. 通常，只有在库中没有这个函数时，你才需要写自定义函数。原因在于C库函数是专业的程序员开发的，它们已经被使用和测试很多次了。所以比起自己写的程序，它们更加可靠，更有效率，有更好的移植性。

h. 如果需要，关键词for、double、while等可以当成函数名。

2. 下面的函数原型中是否有错误，如果有，请发现。

```
a. void (function1) void;
b. void function2 (void)
c. void function( n, x, a, b);
d. void function1(int, double, float, long int, char);
e. void function2(int n, double y, float, long int a, char);
f. void function1(int, a, double, b, float,c);
```

3. 给出下面的函数原型和变量声明，在函数调用中找到错误。

```
void function1(void);
void function2(int n, double x);
void function3 (double, int, double, int);
void function4 (int a, int n, int b, int c);
void main(void)
{
        int a, b, c, d, e;
        double r, s, t, u, v;
. . .
}
a. function1 (a, b);
b. function2(a, b);
c. function3(r,a,s,b);
d. function4(a,b,c,d,e);
e. function1( );
f. function2(r, s);
g. function3(r, s, r, s);
h. function4(r, s, t, u);
```

4. 写出这段程序的输出：

```
#include <stdio.h>
void function1(int a, double x);
void main(void)
{
        int a=1, b=2, c=3, d=4;
        double r=3.2, s=4.3, t=5.4, u=6.5;
        function1(a,b);
        function1(r,s);
}
void function1(int a, double x)
{
        printf ("a= %d, x= %lf\n",a,x);
        }
```

答案

1. a. 真　　b. 真　　c. 真　　d. 假　　e. 真　　f. 假　　g. 真　　h. 假

2. a. void function1 (void);
   b. void function2 (void);
   c. void function(int n, double x, int a, int b);
   d. 没有错误，不需要参数名。
   e. 没有错误，但这不是好的形式，参数名全给或全不给。
   f. void function1 (int a, double b, float c);

3. a. function1( );
   b. C 语言不会检查出错误，但是 b 的值会转变为 double 类型。
   c. 没错误
   d. function4(a,b,c,d);

e. 没错误

f. C语言不会检查出错误，但是x的值会转变为int类型。

g. 没有错误，我们可以把相同的值赋给很多变量。

h. C语言不会检查出错误，但是r、s、t和u的值会转变为int类型。

4. m=1, x=2.000000
   m=3, x=4.300000

课程5.2 返回一个值的函数

主题

- 从一个函数返回一个值
- 考虑返回值的类型

我们已经学习了如何使用库函数接收一个值并返回另一个值。例如log函数就是其中之一。如果写

```
y=log(x);
```

log函数接受x的值，并返回给x的自然对数值。然后把这个值赋给y。

本课的程序中，我们写名为fact的函数来计算一个整数的阶乘。查看源代码，找到fact的函数原型，从中确定fact函数的返回值类型。为什么要选择这一类型？（提示：它与函数计算时数字的大小有关）。

回忆n的阶乘是$n\times(n-1)\times(n-2)\times\cdots\times2\times1$。可以使用一个循环来执行这个计算。在源代码中把循环放到fact函数中。

查看fact的函数体。你能看出哪个语句把程序控制返回到了fact的调用点？哪个变量值被返回？返回变量值的类型是什么？回忆我们调用过的那些库函数，定位出fact在main中调用的位置。注意fact出现在一个赋值语句的右边。赋值语句的左边是g。g的类型是什么？从g和fact的类型我们可以得到什么结论？

这个程序实际上计算1/n!，为什么我们用%e作为显示g的格式控制符？输入的n被限定小于或等于12。为什么？

在以前所有的程序中，我们使用void类型的main。对于这个程序，main的类型是什么？回忆main的调用方是操作系统。我们把什么返还给了操作系统？

源代码

函数原型，没有使用void、double、int等类型，本程序使用unsigned long int作为它的返回类型，函数名为fact，它有一个int类型的参数m

```
#include <stdio.h>
unsigned long int fact(int m);

int main(void)
{
```

没有使用void，使用int作为main的返回类型

```
    int n;
    unsigned long int g;
    double one_over_nfactorial;
```

把变量g声明为unsigned long int类型

```
        printf("This program calculates 1/nfactorial.\n\
                \rEnter a positive integer less than or equal to 12:\n ");
        scanf ("%d",&n);
        g=fact(n);
        one_over_nfactorial=1.0/g;

        printf("1/%d! = %e",n,one_over_nfactorial);
        return (0);
}

unsigned long int fact(int m)
{
        int i;
        unsigned long int product;

        product = 1;
        for (i=m; i>=1; i--)
                {
                product*=i;
                }
        return(product);
}
```

这个语句调用 fact 函数，并把它的返回值赋给 g。注意 g 和 fact 的返回值有相同的类型 unsigned long int。n 的值从 main 传递到了 fact

把一个 int 类型的值从 main 传回到操作系统

函数声明，变量 m 包含从 main 函数传过来的 n 值

将变量 product 声明为 unsigned long int 类型，它作为 fact 的返回变量

循环计算 n 的阶乘。

fact 函数的定义

输出

```
                This program calculates 1/nfactorial.
                Enter a positive integer less than or equal to 12:
键盘输入        9
                1/9! = 2.755732e-06
```

解释

1）如何定义一个返回单个值的函数？在它的函数原型和声明中，返回值的函数必须有一个类型反映出返回值的类型。例如：

```
unsigned long int fact(int m);
```

声明函数 fact 返回一个 unsigned long int 类型。

另外，return 语句必须出现在函数体内。在本例的 fact 函数中，变量 product 的值被 `return(product);` 语句返回。

return 语句的格式如下：

```
return expression;
```

C 认为 return 语句是一个跳转语句，因为它把执行无条件地转移到了另外一个地方。以前学过另外一个跳转语句 break，这个语句在第 4 章介绍过。

2）什么信息从 main 传递到了 fact？什么信息从 fact 传递到了 main？ main 中的 n 值由函数调用语句

```
g=fact(n);
```

传入 fact。通过这一语句，main 中 n 的值被传到了 fact 中的 m，如图 5-5 所示。同时这个图也显示了 fact 中 product 的值通过 `return(product);` 语句传回给了 main。

3）product 变量的值在哪里被返回？如图 5-5 所示，product 的值在函数调用的地方被返回。例如，我们要计算 4 的阶乘（24）。product 的值 24 在函数调用的地方被返回。这样，执行完

```
g=fact(n);
```

这一赋值语句后，变量 g 的值为 24。

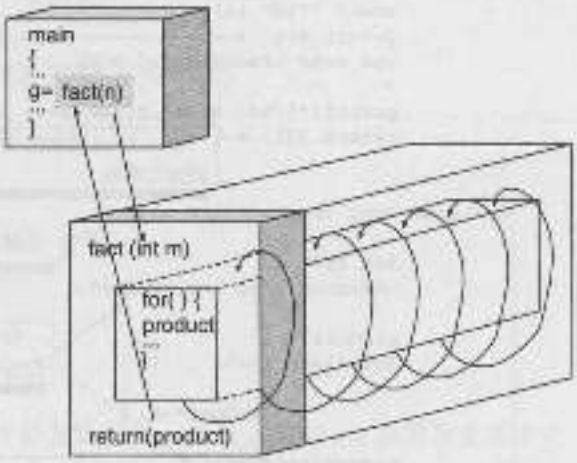


图 5-5 本课程序。注意从 main 到 fact 传递的信息（main 中的 n 对应 fact 中的 m）以及 fact 到 main 传递的信息（返回 product 的值）

4）为什么函数 fact 的类型是 unsigned long int？为什么本课程序中 n 值要小于等于 12？计算阶乘时数字会很快地变大。例如 13! 等于 6 227 020 800。把 n 的类型设计为 unsigned long int（也叫 unsigned long）。在 ANSI C 中规定它至少要占四个字节。这样 unsigned long int 类型不能处理那些超过 4 294 967 265 的值，除非你的编译器扩展了这一标准。因为 13! 大于 ANSI C 中指定的 unsigned long int，所以我们把 n 的值限制成小于等于 12。如果要计算更大的值，需要使用 double 或者 long double。使用 double 或 long double 可以允许计算更大数的阶乘。但是因为实数有精度的限制，所以计算结果可能只是一个近似值。注意，库函数的输入和返回值也有这样的限制。在程序中，你要保证与它们要求的类型一致。

5）return 语句可以用在除了函数体末尾的其他地方吗？可以，return 语句可以出现在函数体的任何地方。另外，函数体可以包含多个 return 语句。通常一个的返回值的结构如下：

```
if (expression)
{
        return (a);
}
else
{
        return (b);
}
```

6）void 类型的函数可以出现在赋值语句的右边吗？不可以。因为 void 函数不返回任何值，所以调用它的语句不可以出现在赋值语句的右面。

7）main 函数的类型为 int，这有什么用？这意味着我们可以把一个值返回给调用方函数。main 函数的调用方是操作系统，我们可以把 0 返还给操作系统，操作系统把这理解成程序正常结束了。现在有很多不同的操作系统。所以你需要检查文档，看看它们是如何定义从 main 返回结束信息的。

概念回顾

1）当一个函数返还给调用方一个值的时候，这个值应该保存在调用方的一个变量中。

2）回忆上一次课中我们提到的，函数的类型就是它返回给调用方的值的类型。因此保存函数返回值的那个变量类型应该与函数的类型一致。

练习

1. 判断真假：

a. 一个正确的 int 类型的函数应该返还给调用方一个 int 类型值。

b. 只有 void 类型的函数才允许有 void 类型的参数。

c. int 类型的函数并不需要遵守参数数目、顺序、类型一致性的限制。

d. 调用 int 或 double 类型的函数的语句一定要出现在赋值语句的右边。

2. 下面函数原型的声明是否有错误，如果有，请指出。

```
a. int (function1) void;
b. double function2 (void);
c. float function1( n, x, a, b);
d. double function1(int, double, float, long int, char);
e. int function2(int n, double y, float, long int a, char);
f. double function1(int, a, double, b, float,c);
```

3. 给定下面的函数原型和变量声明，在函数调用语句中发现错误。

```
double function1(void);
int function2(int n, double x);
double function3 (double, int, double, int);
double function4 (int a, int n, int b, int c);
void main(void)
{
        int a, b, c, d, e;
        double r, s, t, u, v;
        . . .
}
a. a = function1 ( );
b. b = function2(a, b);
c. r = function3(r,s,s,b);
d. s = function4(a,b,c,d,e);
e. u = function1( );
f. c = d + function2(r, s);
g. t = s * function3(r, a, r, s);
h. s = v + function4(r, s, t, u);
```

答案

1. a. 真　　b. 假　　c. 假　　d. 假

2. a. `int function1(void);`

b. 没错误

c. `float function1(int n, double x, int a, int b);`

d. 没错误

e. 没错，但这不是一个好的形式。

f. `double function1(int a, double b, float c);`

3. a. c 不会检查出错误，但是函数返回的 double 类型的数据被保存在一个 int 类型中。

b. c 不会检查出错误，但是 b 以 double 的格式传递给了 function2。

c. 没错误

d. `s = function4(a, b, c, d);`

e. 没错误

f. c 不会检查出错误，但是 s 以 int 的格式传递给了 function2。

g. 没错误

h.c不会检查出错误，但是c、a、b和d以int的格式传递给了function4。同时，赋值语句的右侧有两个double，在赋值语句的左边只有一个int变量。

课程5.3 作用域和传值给函数的机制

主题

- 变量的作用域
- 值调用
- 从函数传入、传出值时的内存管理
- 函数结束运行时的内存管理

我们已经介绍过除了main和库函数以外的其他函数，我们必须对程序中放置声明的位置要多加小心。本课程序并不是一个很好的编程样例，它只是演示了函数体外放置一个声明的效果，以及作用域的概念。

参看源代码以比较m和n声明的异同。注意m只有一个声明，且在所有函数体的外边。但是n有两个声明，一个在main函数中，另一个在funtion1中。注意，无论是m和n，都没有出现在function1的函数列表中。查看输出的前4行。m的值在main function1中初始化为什么值？为什么你认为它们是相等的？（提示：原因在于声明它的位置。）

看一下function1中第一个printf语句后面的行。注意在function1中赋给m一个新值。查看输出。这一行如何影响main中的m？为什么你会认为main中的n会改变？（提示：同样，原因在于声明它的位置。）

n的值在main和function1中是不同的。两个n的值之间有什么关系？

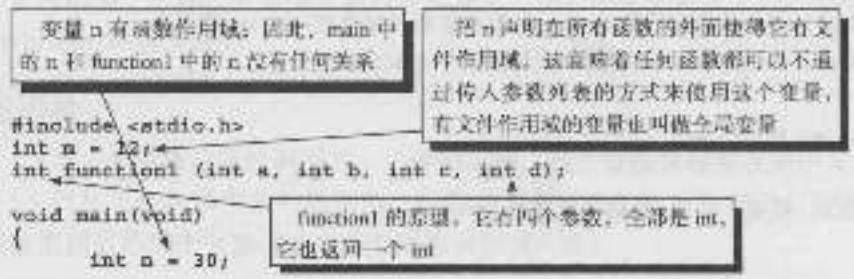
声明的作用域就是使程序有效的范围。有两种作用域，分别是函数作用域和文件作用域。函数作用域的变量只属于它所在的函数中。文件作用域的变量属于这个文件中的任何函数。m的作用域是文件作用域还是函数作用域？同时考虑main中的n。它是文件作用域还是函数作用域？

结构编程作为一个能减少错误的编程方法已经发展了很多年。如果你查看一个大型项目，每个人都写不同的模块。你能想象出把变量声明在函数体外带来的困难吗？

变量a和e有清楚的关联（参看在main中funcion1的调用以及function1的定义）。function1中变量a开始时在main中等于e。在functoin1中，a被修改成1402。当返回给main时，e变成1402了吗？这显示了C语言处理值传递的一些特性，它告诉我们什么？

fuction1有不止一个return语句。跟踪程序流程，看哪个return语句被执行？哪个值赋给了c？

源代码



```
int e,f,g,h,i;
e=1;
f=2;
g=3;
h=4;
printf ("\n\n In main (before the call to function1): \n
        \r m = %d\n\
        \r n = %d\n\
        \r e = %d\n\n",m,n,e );

i=function1(e,f,g,h);

printf ("After returning to main: \n");
printf ("n = %d \n\
        \r m = %d \n\
        \r e = %d \n\
        \r i = %d", n, m,e,i);
}
```

变量 n 被输出。注意在 main 中，它的值并不需要初始化，因为在声明它为全局变量的时候已经初始化它了

调用 function1，e、f、g 和 h 的值被传给了 function1。返回的值保存在 i

变量 m 在 function1 中被修改，因为它是全局变量，所以 m 的值在 main 中也被修改并被输出

function1 的定义

```
int function1 (int a, int b, int c, int d)
{
        int n = 400;

        printf ("In function1:\n\
                \r n = %d\n\
                \r m = %d initially\n\
                \r e = %d initially \n",n,m,e);
        m = 999;

        if (e>=1)
        {
                e+=b+m+n;
                printf ("m = %d after being modified\n\
                        \r e = %d after being modified\n\n",m,e) ;
                return (e);
        }
        else
        {
                c+=d+m+n;
                return (c);
        }
}
```

变量 n 的值与函数作用域，因此 main 中的 n 与 function1 中的 n 没有关系

全局变量 m 的值在这一语句被修改。注意，这个变量并没有在 function1 函数中被声明。因为它被声明在所有函数的外面，所以不声明它是可以的

一个函数可以有很多的 return 语句，但是只有一个被执行

输出

```
In main (before the call to function1):
m = 12
n = 30
e = 1

In function1:
n = 400
m = 12 initially
e = 1 initially

m = 999 after being modified
e = 1402 after being modified

After returning to main:
n =30
m = 999
e = 1
i = 1402
```

解释

1）什么是作用域？作用域就是声明有效的一个区域。在C中有四种作用域：块、函数、文件和原型。一个标识符作用域被这个标识符声明所在的位置所决定。

本课程序变量n有函数作用域，这意味着赋给变量n的值只在定义这个变量的函数内有效。function1声明的变量n和main中的n没有任何关系，因为它的作用域只局限于function1。

另一方面，变量m有文件作用域，因为它被声明在任何一个函数的外边。它的有效范围从声明它的位置开始，一直延续到源文件的末尾。注意m没有声明在任何函数体内，但它被main和function1所使用。这样做的结果是，当m在function1中被修改，main中m的值也被修改。这种变量声明的方法在编程实践中并不被推荐。好的模块化编程风格应该把值通过参数列表传入函数。

标识符function1也是文件作用域。function1的原型在所有函数的外部。它的有效区域从声明点一直延伸到文件的末尾。

回忆块是一段以{开始和以}结束的代码。在这个块中声明的标识符只局限于这个块。以前说过如果用预处理命令#define来定义一个常量，例如，

```
#define P 200
```

这会让预处理器把所有源代码中的p替换成200。这个命令无论放到函数内部还是函数外部，它都有文件作用域。只是文件作用域从这个命令行开始，一直延续到文件的末尾。

函数原型中给出的参数名有函数作用域。这意味着在声明函数原型这一行除外，原型中的所有标识符都被认为没有声明过。这就是为什么函数声明中的变量名与函数定义中的变量名可以不一致。（但是参数类型必须一致。）

语句标签也有作用域。那些用在switch中的语句标签也有函数作用域。这意味着语句标签不能在函数范围内重复。

如果你学过高级的编程技术，就可以把一个标识符的作用域扩展到多个文件。我们在第8章讨论这个方法。

2）C语言如何处理从参数列表传入的参数？当一个函数被调用，C为这个函数的参数列表中的变量及函数体声明的变量分配内存空间。函数调用中的表达式的值被拷贝到为这个函数分配的对应的变量的内存位置中。本课程序中，当main调用function1时，整个调用过程概念如图5-6所示。观察到下面这一点。function1拷贝过来的变量改变时，保存在main函数中变量的值不会改变。

3）当a的值从function1传回main时，内存发生了什么？在main中下面的赋值语句

```
i=function1(m,f,g,h);
```

意味着function1中的返回值a返回给main中的i，如图5-7所示。从图5-6和图5-7可以清楚地看到main和function1中分配的不同区域。本课程序中的信息传递如图5-8所示。

4）什么是局部和全局变量？局部变量和全局变量并不是ANSI C中标准的定义，而是程序员经常使用的一个通用词汇。不严谨地说，全局变量就是有文件作用域的变量，局部变量就是有函数作用域的变量。例如本课程序中，m是全局变量，n是局部变量，如图5-8

所示。

5）本课程序中，当执行完 function1 后，为 function1 变量所分配的内存发生了怎样的变化？针对这个程序，空间被释放。但是当你学习了一些高级编程技巧时，可以通过一些特殊的声明来指定存储变量在函数结束的时候依然被保存。如果你选择第二次调用某个程序，那么内存会被再次分配（依赖于内存中现实的情况，也许是不同位置的内存被分配）。因此在程序运行期间，每次函数被调用，内存就会被分配。如果不是特殊指定，每次函数结束运行时，内存被释放。

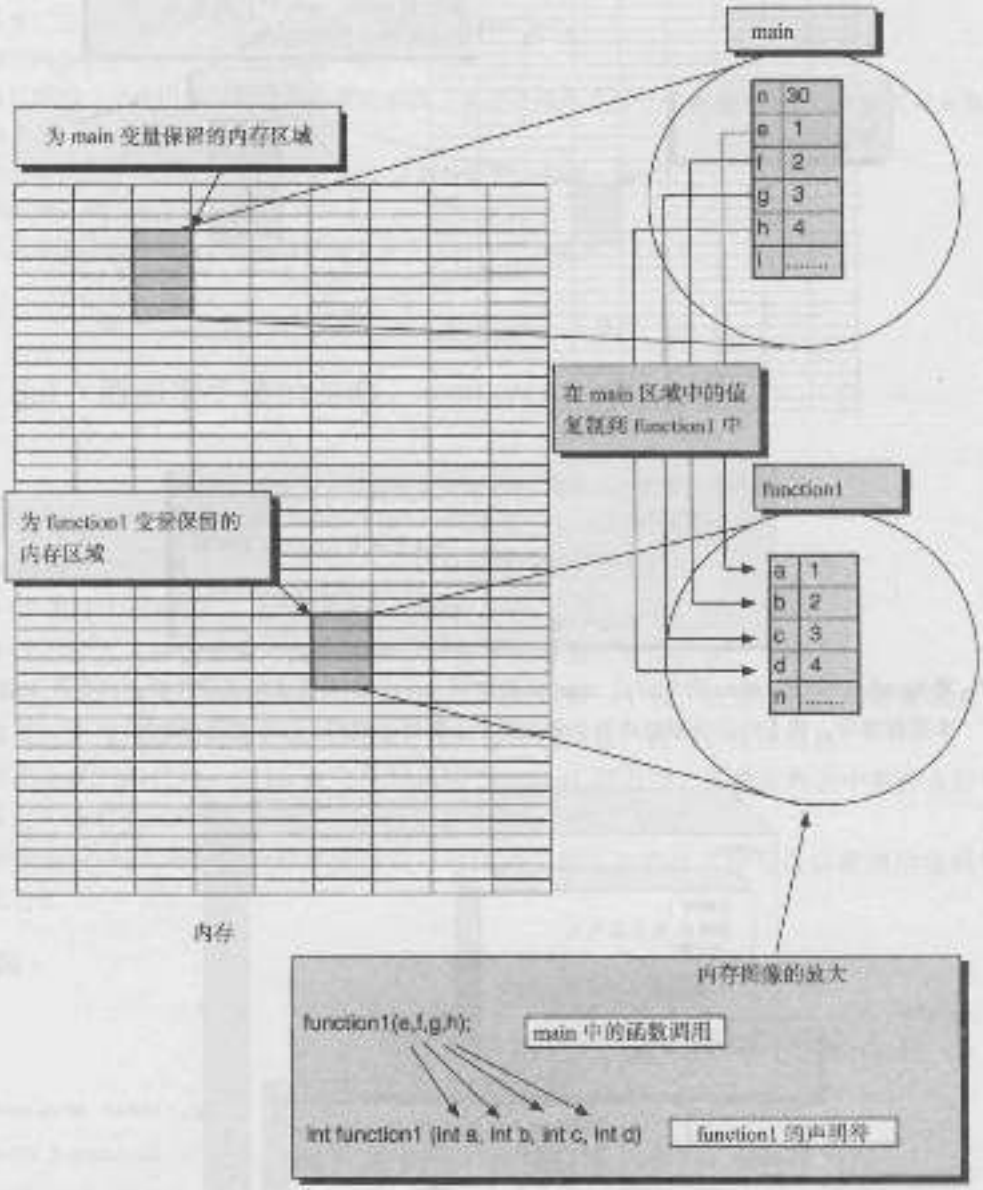


图 5-6　在 main 中调用 function1 时内存的情况。两个内存区域也许并不是如图所示那样分离的，而是彼此相邻的

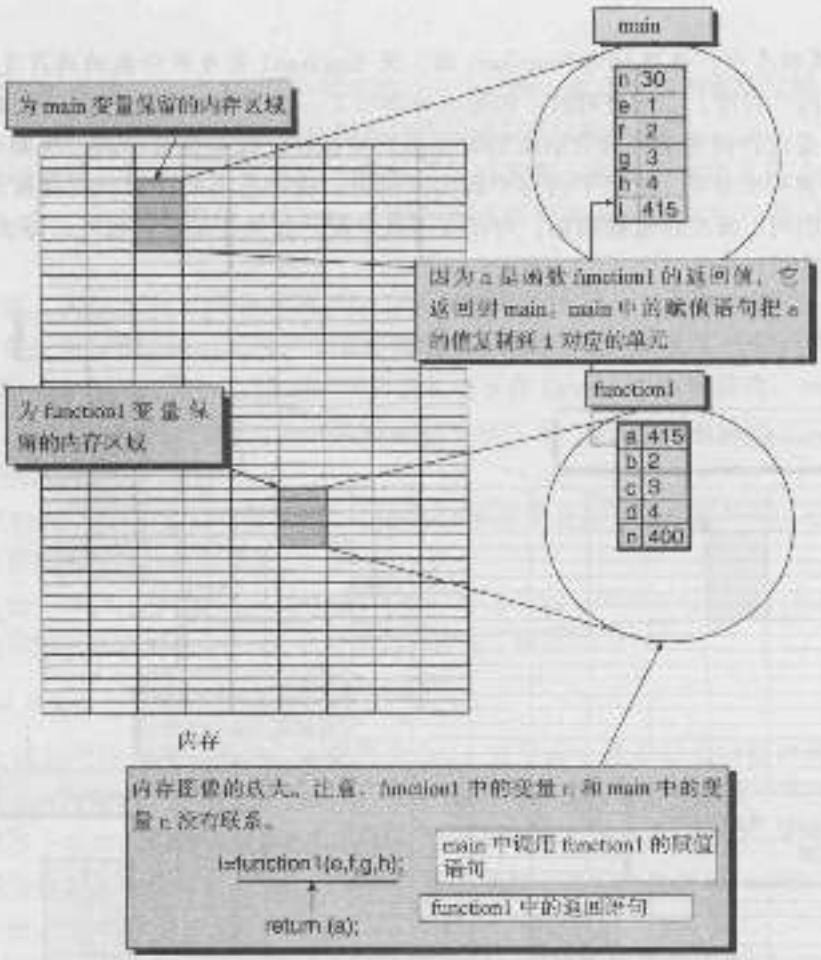


图5-7 当function1执行return语句并且main函数调用function1来完成赋值操作时，内存发生的动作。本课程序中，两个内存区域也许并不是如图所示那样分离的，而是彼此相邻的

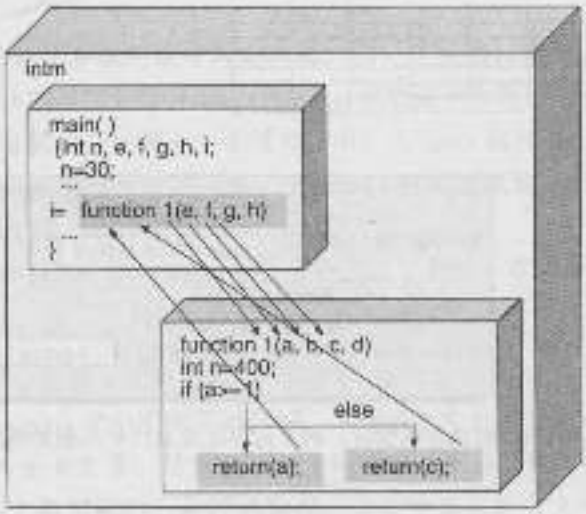


图5-8 本课程序。注意main和function1都可使用n，且其中的n没有关系

概念回顾

1）变量作用域就是变量有效的区域。

2）一个变量在作用域以外是“不可见的”，它的值不能使用也不能修改。

3）全局变量在程序的任何地方可见，而局部变量只在它们的作用域可见。

练习

1. 判断真假：

a. 通常，应该尽量经常使用全局变量（文件作用域变量）。

b. 函数的参数类型必须和函数类型一致。

c. 可以调用一个使用变量 n 作为参数的函数，通过在函数内部改变参数的值，来达到改变 n 值的目的。

d. 为其他函数变量分配的内存与为 main 函数分配的内存是一致的。

e. 函数调用时，内存被分配。

f. 当变量值传入函数后，会在为函数变量分配的内存上生成一个拷贝。

答案

1.a. 假　　b. 假　　c. 假　　d. 假　　e. 真　　f. 真

课程 5.4　返回多个值的函数

主题

- 符号 &
- 符号 *
- 把值传出函数

本课中描述一种返回多个值的方法，下次课来解释背后的机制。

目前，我们演示了如何使用 return 语句让函数返回一个值。假如你想从函数返回多于一个的信息，可使用参数列表，参数列表不仅可以传入信息，也可以传出信息。

检查下面的源代码。在 main 中找到调用 function1 的语句。在参数列表中有什么符号你没有见过？查看 function1 的原型，原型中有什么符号你没见过？

查看 function1 函数体内的赋值语句，赋值语句的右边有什么符号在以前的赋值语句中没有见过？

源代码

函数原型。* 号代表想得到对应参数的值，我们必须在函数体内也使用 * 号

```
#include <stdio.h>

void function1 (int a, int b, double r ,double s, int *c, double *t);

void main (void)
{
        int     i=5,     j=6,     k;
        double  x=10.6,  y=22.3,  z;

        printf (" i = %d \n\r j = %d \n\r x = %lf \n\r y = %lf \n\n",
                                i,j,x,y);
```

```
        function1 (i,j,x,y,&k,&z);
        printf (" k = %d \n\r z = %lf \n\n", k, z);
}
void function1 (int a,int b,double r,double s,int *c,double *t)
{
        *c = a+b;
        *t = r+s +(*c);
        printf (" *c = %d \n\r *t = %lf \n\n", *c, *t );
}
```

调用函数时，必须在最后两个参数上使用&号，就是那些在原型中使用*的参数

函数体内，必须使用*号来得到值

输出

```
 i = 5
 j = 6
 x = 10.600000
 y = 22.300000

*c = 11
*t = 43.900000

 k = 11
 z = 43.900000
```

解释

1）本课中，哪些值从main中传到了function1？ main中i和j的值（分别为5和6）传递到function1中的a和b。另外，main中x和y的值（分别为10.6和22.3）传递到function1中的r和s。如图5-9所示：

2）哪些变量的值从function1"返回"给了main？ *c和*t在function1中的值（分别为11和32.9）被转移到main中的x和z，如图5-9所示。

3）调用函数时，如何指定一个从变量函数接受值？如果想让一个变量从函数中接受一个值，我们可以把符号&放到这个变量的前面。本课程序中的function1，程序调用中的第5和第6个变量k和z前面有&，所以它们从function1中收到值。

4）在函数原型和函数定义中，如何指定一个变量从调用函数返回值？我们用符号*作为变量名的第一个符号。在本课的function1中，在参数列表的第5和第6个变量是*c和*t，这两个变量把值从function1传回main。

5）在函数体内，如何处理那些以*开始的变量？像处理其他变量一样处理这种变量。但是，也许存在一个问题。因为*号也作为乘法运算符。为了避免混淆，我们推荐你在赋值语句的右边使用它们时用括号括起来，即使有的时候括号并不是必需的。

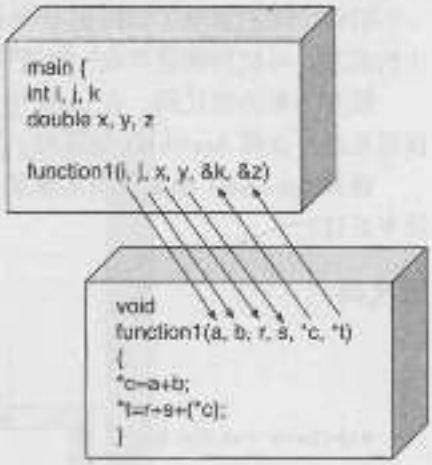


图5-9 本课程序的演示，这个图演示了在main和function1之间传递信息

6）为什么function1是void类型？虽然function1返回了值，但是它并没有用return语句返回任何值。如果你用return返回一个值（与参数列表一起使用），函数类型和return语句

返回的类型必须一致。

7）我们以前提到过 C 只是把调用参数列表中的值拷贝到为函数分配的内存中。如果是这样，从参数列表返回值是怎么一个过程？C 确实把调用参数列表中的值拷贝到为函数分配的内存区域。但是，使用我们以前没有讨论过的运算符，C 能够把值从调用函数返回。也就是说，本课讨论的内容就是从函数返回多个值的具体发生的过程。我们描述的这种方法可以用于写很多实用程序。因此推荐你用本课的内容来写返回多个值的程序。同时，你需要理解这种方法背后的机制，下次课我们详细讨论。

概念回顾

1）在函数调用时，函数参数前的 & 代表变量用来保存从函数传回的对应值。

2）在函数定义时，函数参数前的 * 代表变量关联的值会从函数传回。在函数体内，* 号要一直和那个变量同时使用。

3）函数定义中的 * 和函数调用中的 & 可以认为是一对符号，应该总是配对使用。

练习

1. 给定下面的 C 语言程序，判断下面语句的真假。

```
void plus(short a, long *b);

void main(void)
{
        short x=100;
        long y=9999;

        plus(x+200, &y);
        . . . . . .
}
```

a. plus() 函数为 void 类型，你不能用它在 main 函数返回值。

b. plus() 函数中的短整型变量 a 可以用来把一个值从 plus() 传回 main() 函数。

c. 不用 return 语句，plus() 函数可以把一个值传回到 main 函数。

d. plus 函数原型中的 * 是错的，它应该是 &，而不是 *。

e. y 的值可以被 plus() 函数修改。

f. x 的值可以被 plus() 函数修改。

2. 写一个调用 void 类型的函数，计算三个整型数中的最大值。

3. 不通过函数写一个程序，交换两个 long 类型整数的值。

4. 通过函数写一个程序，交换两个 long 类型整数的值。

答案

1. a. 假　b. 假　c. 真　d. 假　e. 真　f. 假

课程 5.5　从函数返回多个值的机制——地址和指针变量

主题

- 保存地址的变量，指针变量
- 取址运算符 &
- 取值运算符 *
- 把一个地址传入函数

本课的源代码和上一课几乎一致。唯一不同的就是加上两个 printf 语句输出一些值。

C 用一种间接的方法把信息从 main 传入到 function1。它用变量的地址（它们内存中的位置）来传递信息。第 1 章中学习过内存位置上可以保存很多不同的值。内存单元可以保存整型、浮点型、指令和地址。

变量地址和家庭地址有很多相似的地方。如果你知道朋友的地址并想送她一个生日礼物，你要怎么做？当然，你可以把礼物送到朋友所在的地址上。C 语言也一样。它把变量值放到变量的地址上（如果你命令它这么做。）

C 为 main 函数中的变量分配一个内存区域，并为 function1 中的变量分配另一块内存区域。为了执行这个过程，为函数分配的内存区域必须先生成并被保留。为了保留正确大小的内存空间，C 语言必须知道所有的变量和变量类型。因为 main 函数把地址传给了 function1，fucntion1 函数中的一些内存单元必须保存地址。

如何处理地址？地址除以另外一个地址是没有意义的。但是就像你知道某个人的地址并想找到他（你去某个人的地址找到他，然后做后续的事）。C 有一个运算符，它的作用就是"去那个地址，并使用那个地址包含的内容"。

源代码

在函数声明或函数原型中，* 号代表 * 后面的变量保存一个地址。因此，function1 中的变量 c 和 t 不保存整型或浮点型，它们保存地址

```
#include <stdio.h>

void function1 (int a, int b, double r, double s, int *c, double *t);

void main (void)
{
        int    i=5,j=6,k;
        double x=10.6,y=22.3,z;

        printf (" i = %d \n\r j = %d \n\r x = %lf\n\r y = %lf \n\n",
                                    i,j,x,y) ;


        function1 (i,j,x,y,&k,&z);

        printf (" k = %d \n\r z = %lf \n\n", k, z) ;

        printf (" Address of k = %p\n\r Address of z = %p \n", &k, &z);

}

void function1 (int a,int b,double r,double s, int *c,double *t)
{
        *c = a+b;
        *t = r+s+(*c);

        printf (" *c = %d \n\r *t = %lf \n\n", *c, *t );
        printf (" Value contained in c = %p\n\
                \r Value contained in t = %p\n\n", c, t);
}
```

因为 & 是取地址运算符，k 和 z 的地址被放到了为函数 function1 中变量 c 和 t 保留的内存单元中

C 语言语句中的 * 号除了用在函数的声明或原型中，它还有别的用法。作为单目运算符的 *（不是代表着乘法运算符）意味着来到 * 号后接变量中保存的地址上，并取出这个地址上保存的值

i、j、x 和 y 的值被拷贝到为函数 function1 中变量 a、b、r 和 s 保留的内存单元中

输出

```
i = 5
j = 6
x = 10.600000
y = 22.300000

*c = 11
*t = 43.900000

Value contained in c = FFF0
Value contained in t = FFD8

k = 11
z = 43.900000

Address of k = FFF0
Address of z = FFD8
```

解释

1）变量保存方法的概念图是什么？在第2章讨论过，C理论上生成一个函数中所有变量的表。在这个表中保存变量名、变量类型、内存单元地址和变量值。例如，对于main函数，生成下面的表：

变量名	变量类型	变量地址	变量值
i	int	FFF4	5
j	int	FFF2	6
k	int	FFF0	—
x	double	FFE8	10.6
y	double	FFE0	22.3
z	double	FFD8	—

注意C负责生成地址。你不需要指定地址，它们自动地指定。C通过参看main中声明变量的数量和类型来计算在main中需要保留多少个内存空间。

内存位置的值在到初始化时才被填充。在这个表中显示了变量i、j、x和y的值，因为它们在声明的时候就被初始化了。但是变量k和z的值没有列出，因为它们在程序的后面被初始化。但是它们的地址被列出了，因为地址在内存单元没有填充值前就被确定了。这个内存单元只是在等待一个值。

对于function1，生成下面的表：

变量名	变量类型	变量地址	变量值
a	int	FFC0	5
b	int	FFC2	6
c	address to int	FFD4	FFF0
r	double	FFC4	10.6
s	double	FFCC	22.3
t	address to double	FFD6	FFD8

在这个表中显示了所有变量的值。这些值在函数声明中出现。所有这些值从调用function1的参数列表中拷贝。注意a、b、r和s及i、j、x和y之间的对应关系。注意c的值是k的地址、t的值是z的地址。从main到function1的信息传递如图5-10所示。

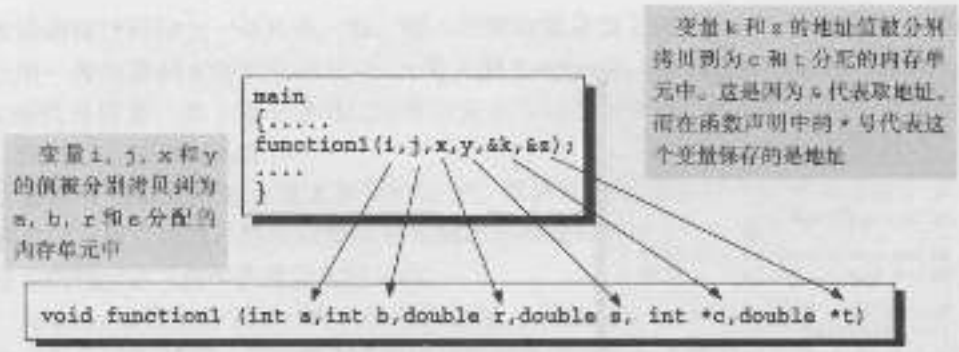


图5-10 本课程序中的信息传递

2）c和t的值从哪里来？为什么它们不是int或double而是地址？运算符用来生成这些值，*号用来指定变量位置中保存的是一个地址类型而不是int和double类型。变量c和t是指针变量，因为它们包含的是一个地址。

符号&是取址运算符。这是一个单目运算符，意味着它被放到单个的标识符的前面。在本课程序中第三个printf语句输出了变量k的地址。调用function1中，运算符&生成"变量k的地址"和"变量z的地址"作为函数的第5和6个参数，这两个参数被拷贝到为function1分配的内存区域的第5和6个参数位置。

在函数的声明中，符号*出现在第5和6个参数位置。在这里，符号*代表后面的变量保存的是地址。声明int *c代表这个c保存的是一个int类型的地址，double *t代表这个t保存的是一个double类型的地址。这样，调用function1（用&）和function1的声明（用*）使得函数function1得到变量k和z的地址。

3）function1怎么使用这些地址？因为function1知道这个地址，它也就知道要把k和z的值放到哪里。这样利用运算符指定地址，就可以把值保存到指定的位置。

4）哪个运算符实现了"指定地址"？function1函数体中的单目运算符*（取值运算符）实现了这种操作。在解释细节之前，只要知道*在本课中一共有三个用途：

a. 双目运算符，例如a = c*t;

b. 声明限定符，代表变量单元里面要保存一个地址，例如：

```
voidfunction1(int a, int b, double r, double s, int *c, double *t)
```

c. 单目运算符代表去某个地址；例如

```
*c = a + b;
*t = c+a+[*c];
```

我们指出这三种不同的*的应用，千万不要把它们搞混淆了。注意当*用在一个声明的时候，代表第二种用法。当*放在赋值语句的左边的时候，代表第三种用法。当*出现在赋值语句的右边的时候，你需要仔细查看表达式以区分出它是单目运算符还是双目运算符，以便确定它是第一种用法还是第三种用法。

最后一个单目运算符*，它实现了"去标识符保存的那个地址所对应的内存单元中。"换句话说，*c实现了"去保存在c中的那个地址所对应的内存单元"。

```
*c = a + b;
```

实现了"去保存在c中的那个地址所对应的内存单元，并存入a+b的值"。使用上个表中最

示的地址，a 和 b 的和被放到了地址为 FFF0 的内存单元中。注意这个地址也是 main 函数中变量 k 的地址。基于此，这个动作使得 main 中变量 k 保存的是 a 和 b 的和。

对于

```
*t = r+s+(*c);
```

r，s 对应的内存单元中的值和 C 内存单元所保存地址对应的值被加在一起，并保存在变量 t 中地址所对应的内存单元中。这样使用表中出现的地址。r、s 和地址 FFF0 中的内容被加到一起，放到了内存地址为 FFD8 的内存单元中。注意，FFD8 也是 main 中变量 z 的地址。所以这个动作使得 main 中变量 z 等于 r、s 和地址 FFF0 中的内容的和。

5）两个地址运算符 & 和 * 的可视图是什么？上述解释比较模糊并难于理解。为了更好地理解这些操作符，运用可视图更有效。在图中依次是变量名、变量类型、地址和值。& 运算符可以当成从变量的值到它的地址的一个箭头。如图 5-11 所示。本图中 & 箭头指定了操作 &k。单目运算符 * 可以看成是从某个内存单元中保存的地址值，到地址值中保存的变量值的一个箭头，本图中 * 运算符指定了操作 *t。

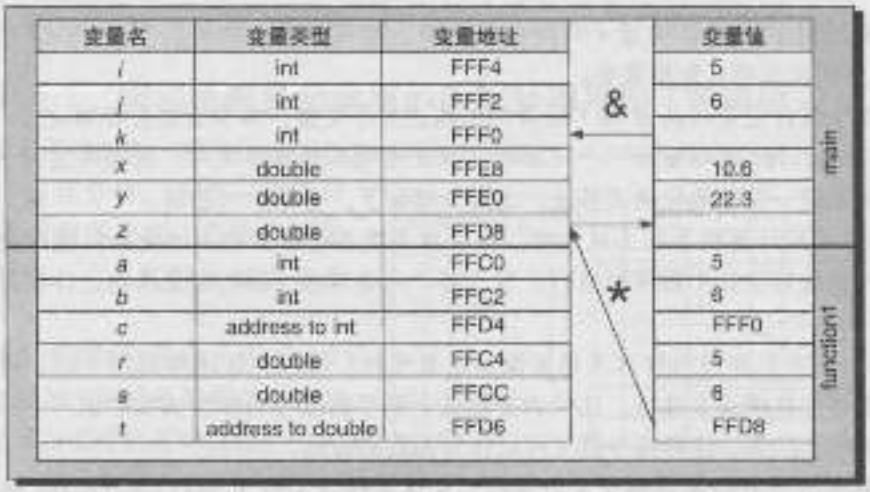


图 5-11　* 和 & 的操作。一个操作是 *t，因为 t 中是地址，*t 代表值 s（因为 s 的地址保存在 t 中）。另外一个操作是 &k。& 是取址运算符，所以 &k 取得 k 的地址 FFF0

图 5-11 演示了 & 运算符操作变量的地址而不是它的值。它为使用单目 * 运算符做了铺垫。* 运算符代表这个地址上保存的变量的值被使用。用语言描述这个过程很困难。我们推荐你用可视图来记忆。这样会有效地使用 & 和 * 运算符。

6）运算符 * 和 & 可以做什么，不可以做什么？上面的可视图准确地描述了如何理解这里的例子。例如 function1 里面有一个语句

```
*t = 83.9;
```

它等同于 s = 83.9

在 main 中，我们不能使用

```
&k = FDD2;
```

因为我们不能确保变量 k 被保存在地址 FDD2 上。我们对地址没有控制权。在写程序的时候，不知道地址为 FDD2 的内存单元是否可用。（这个语句有其他问题，但是现在我们不讨

论。）C语言自己确定地址，所以我们不能把&放到一个赋值语句的左边。

本课中可以在任何变量上使用&运算符（只要不出现在赋值语句的左边）。但是我们不能在任何变量上使用*。单目运算符*必须用在用*声明的指针变量上。因此不能写出

```
*y = 83.9;
```

因为变量y没用*声明（它不保存地址）。也不能写出

```
t = 83.9;
```

因为变量t只保存地址。

同时我们也不能写出

```
t = FFC4;
```

因为事先我们不知道哪个内存单元可用。可以用取址运算符来处理这种情况。例如一个合法的语句是

```
t = &r;
```

这使得r的地址被保存在变量t的内存单元中。注意变量r的类型是double，t的声明是double*。声明类型的匹配很重要。

这里对于指针变量以及&和*运算符的描述并不完整，本书后面会详细论述。

7）当用printf语句输出一个地址时，该使用什么转换限定符？转换限定符%p用来输出与计算机系统一致的地址格式信息。大部分情况下，它是十六进制。

8）在不同的计算机系统上运行时，本课程序是否会输出相同的地址？也许会，也许不会。真实的地址对于这个程序的运行并不重要。只要对应于那些变量的地址存在，程序就会正确地运行。

9）把一个参数即作为输入又作为输出是否可行？可以。在函数调用中以&开头的参数代表着函数可以使用这个地址。这代表着地址上保存的值可以作为函数的输入。函数也可以改变这个地址上的值。这样这个值又可以作为函数的输出。

10）本书中还有哪里用到了&运算符？&运算符也用在了scanf和fscanf函数中。例如，从键盘读入整数k，语句为scanf("%d",&k)。

11）在语句scanf("%d",&k);中，&有什么用？再次，&是取地址运算符。使用这个语句，我们把为变量k保留的内存单元的地址传入scanf函数中去。因为k还没有被初始化，所以在这个地址上还没有任何值。但是这个地址已经分配给了变量k。因为scanf函数知道这个地址，它把从键盘读入的数值保存在这个地址指定的内存单元上。它从转换限定符%d知道这个值需要占多少位。

12）现在有两种方法从函数返回值。一种是用return语句，另外一种用在参数列表中传入地址。如果需要的话，可不可以同时使用两种方法？可以。你可以同时使用return语句和参数列表返回值。为了这样做，不要把函数声明为void类型。

概念回顾

1）*x代表保存在地址x上信息。这样a=*x代表地址x上保存的值赋给变量a。

2）&y代表变量y的地址。这样&y= 12FF代表把y的地址变为12FF。因为我们不能控制内存地址，所以&y=12FF是非法的。

练习

1. 判断真假：

a. C 语言的变量，无论什么类型，都必须有一个地址。

b. 任何类型的变量都可以用来保存地址。

c. 变量的地址用十六进制表示，这样只有整型变量才可以用来保存地址。

d. 任何类型的指针变量可以用来保存一个 double 类型的变量的地址。

e. 指针变量比标量更难用，因为程序员需要手工发现保存在指针变量中的地址。

f. 取值运算符 (*) 只可以出现在赋值语句的左侧。

g. 取值运算符 (*) 可以出现在赋值语句的两侧。

h. 取地址运算符 (&) 只可以出现在赋值语句的左侧。

i. 取地址运算符 (&) 可以出现在赋值语句的两侧。

j. 对一个整型变量 nn，它的地址 &nn 是一个常数，不是变量。

2. 写一个程序，通过调用一个函数来交换两个 long integer 的值。函数原型尽可以包含一个 long integer 的指针。

答案

1. a. 真　b. 假　c. 假　d. 假　e. 假　f. 假　g. 真　h. 假　i. 假　j. 真

应用程序 5.1　使用带有复杂循环的函数处理网格（逻辑例子）

在开发程序的时候，程序员经常要面对网格或网眼。一个网格用来分析一块板的压力或一个固体的温度分布。它把一块感兴趣的区域分成很多小的、易于管理的部分，以便每一个小的部分可以单独计算。

本课的应用程序不是一个标准的工作程序。这里演示的只是帮你学习处理网格及写循环时需要的逻辑技能。

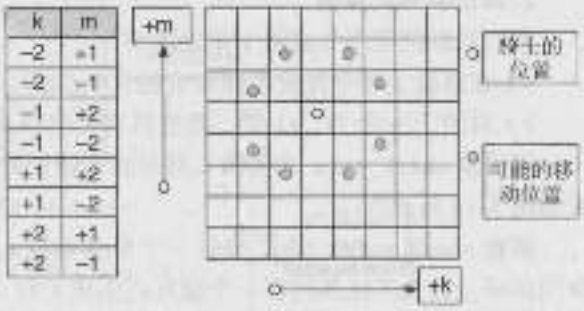


图 5-12　一个骑士（空心圆）在棋盘的中间，以及它可能的移动位置（阴影圈）。注意 k 和 m 的可能的值。

问题描述

一个象棋盘有 8 行 8 列。如果你熟悉象棋，就知道骑士可以从它的位置 (i, j) 向 k 方向或者 m 方向移动。移动的方式或者是向前两格，向右一格，或者向前一格，向右两格。一个棋盘中间的骑士可以如图 5-12 所示向八个方向移动。

但是，一个在角落的骑士因为有棋盘边的限制，所以有较少的移动方式。如图 5-13 所示。

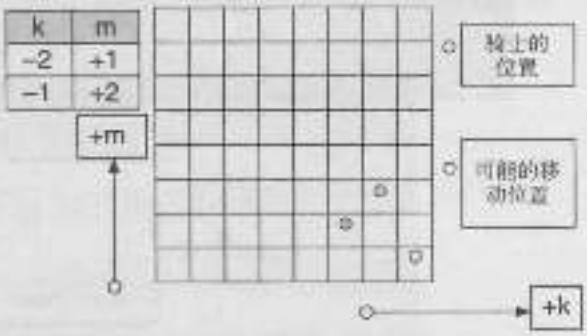


图 5-13　一个骑士（空心圆）在棋盘的角落，以及它可能的移动位置（阴影圈）。注意 k 和 m 的可能的值比图 5-12 要少

针对这个应用，写一个计算在棋盘上任何位置的骑士能移动的数目的程序。没有输入，输出程序以网格模式输出到屏幕，在棋盘每个位置显示可以移

动的步数。

解决方法

处理网格问题时，应该首先标识出网格的位置。以棋盘为例，我们推荐使用数对（i，j）来标识棋盘。图5-14显示了一些网格的位置。i和j的值从左到右，从下到上依次增加。你可以推导出本图没有标出的那些信息。

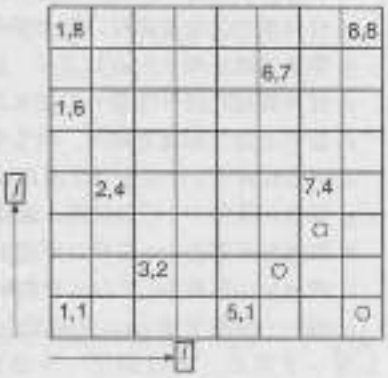


图5-14 网格位置——i和j的值只在部分方格中给出

1. 相关公式

如果一个骑士的初始化位置为（i，j），那么新位置为（$i+k$，$j+m$）。如果满足下面的条件，那么移动是合法的。

$$1 \leqslant (i+k) \leqslant 8$$

且

$$1 \leqslant (j+m) \leqslant 8$$

如图5-14所示。

一个所有可能的k和m方向的移动在图5-12的表中给出。在棋盘上的每个位置（i,j），我们检查所有的k和m的组合，来看它们是否可行。如果一个移动可行，我们进行计数。不可行的不计数。有八种可能的k和m的组合及64个（i,j）的位置，我们需要检查8×64=512种可能的移动。

2. 算法和代码逻辑

我们可以把程序分成两个单独的部分：

1）在棋盘上每个位置生成对应的i和j的值。

2）对应于一个（i，j）值，检查所有可能移动的数量。

用函数check_point来做第二部分的工作，并且在main中做第一部分的工作。整个数据流如图5-15所示。

函数check_point。为了检查一个单独的点，我们必须写出完成特定计算的代码，需要使用循环。首先关注如何在一个循环内生成k和m的组合。

```
for (k = -2; k <= 2; k++)
{
        if (k != 0);
        {...
         ...
        }
}
```

生成一个从-2到2的循环

如果k不等于0，开始计算

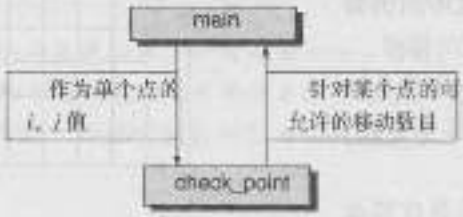


图5-15 骑士移动程序中的数据流向图

注意到在图5-12中的表，k方向移动的可能值从-2到2，不包含0。这暗示了下面这种循环

注意到这个循环结构，生成了四次迭代（$k=-2,-1,1,2$）。从图 5-12 中的表格可以看出对每一个 k 值，需要生成两个 m 值。如果我们能注意到：

$$|k|+|m|=3$$

从 k 值可以计算出 m 值，则其中 $|k|$ 为 k 的绝对值，$|m|$ 为 m 的绝对值。

因此

$$m=(3-|k|)$$

或

$$m=-(3-|k|)$$

在从 -2 到 2 的循环内部，我们生成了一个两次的循环用来计算，循环第一次计算 $m=(3-|k|)$，循环第二次计算 $m=-(3-|k|)$。整个过程如下所示：

```
p = -1
for ( n = 1; n <= 2; n++)
{
        p = -p;
        m = p * (3 - abs (k));
}
```

初始化 p，这是一个虚拟变量，唯一的用途就是把 m 的值从 -1 到 1 来回变化

形成一个执行两次的循环（n 是一个完成这个任务的虚拟变量）

将 p 从 -1 到 1 来回变化

循环第一次计算 $m=(3-|k|)$，循环第二次计算 $m=-(3-|k|)$

把这个循环放到 k 的循环里

```
p = -1
for (k = -2; k <= 2; k++)
{
        if (k != 0)
        {
                for (n = 1; n <= 2; n++)
                {
                        p = -p;
                        m = p * (3 - abs (k));
                }
        }
}
```

初始化 p

外层循环从 -2 到 2

只在 k 不等于 0 时执行内层循环

内层循环执行两次

将 p 从 -1 到 1 来回变化

计算 $m=(3-|k|)$ 或者 $m=-(3-|k|)$

当生成这个复杂的循环后，你应该做一个表格以检查这个循环是否生成了你想得到的值。例如，对于这个循环，有下面的表格。

初始化	k	n	p	m
-1	-2	1	1	1
		2	-1	-1
	-1	1	1	2
		2	-1	-2
	0	结束循环		
	1	1	1	2
		2	-1	-2
	2	1	1	1
		2	-1	-1

对于这个循环结构，我们生成了所期望的 k 和 n 的组合值。这样就可以在程序中使用这个循环结构了。当有一个（i，j）值，可以使用每一个 k，m 的组合值来检查移动是否合法，如果合法，那么计数器加 1。

如果计数器为 icount，那么在移动合法的时候，我们应该在 icount 上加 1。这意味着我们需要 if 和条件语句。以前说过的条件是

$$1 \le (i+k) \le 8$$

且

$$1 \le (j+m) \le 8$$

这个条件如果为真，那么计数器变量加 1。C 源代码如下：

```
if ( 1 <= (i+k) && (i+k)  <=  8            &&
1  <=  (j+m)  &&  (j+m)  <=  8)    icount++;
```

这个语句进入内循环。我们把这些都放入函数 check_point。此函数输入 i 和 j 的值，返回 icount 的值。函数如下：

```
int check point(int i, int j)
{
int icount, p, k, n, m;

icount=0;
p = -1
for (k = -2; k <= 2; k++)
{
    if (k != 0)
    {
        for ( n = 1; n <= 2; n++)
        {
            p = -p;
            m = p * (3 - abs (k));
            if ( 1 <= (i+k) && (i+k)  <=  8 &&
            1  <=  (j+m)  &&  (j+m)  <=  8)    icount++;
        }
    }
}
return(icount);
}
```

函数接受 i 和 j，返回一个整数

外循环

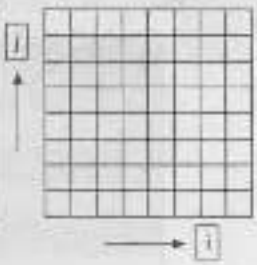
内循环

移动是否合法，如合法就加 1 的条件判断语句

主函数。主函数内需要循环以生成 i，j 的组合值来代表棋盘上的每一个点。这比 check_point 函数简单，循环如下：

```
for ( j = 1; j <= 8; j++)
{
    for ( i = 1; i <= 8; i++)
    {
        ...
        ...
    }
}
```

嵌套的循环生成覆盖所有 64 个点的 i，j 组合值



使用这个嵌套的循环，我们生成对应于棋盘每个方格的 i，j 的值。对于每一个组合 i，j 的值，我们检查所有可能的 k，m 的组合。为了检查 k，m 的组合，调用 check_point 函数。整个代码如下：

```
void main (void)
{
        int n, i, j, icount;
        printf ("Number of possible moves for a knight "
                "on a chess board \n\n");
        for (j = 1; j <= 8; j++)
        {
                for (i = 1; i <= 8; i++)
                {
                        icount=check_point(i,j);
                        printf ("%5d", icount);
                }
                printf ("\n");
        }
}
```

函数 check_point 返回合法的移动的步数

输出合法的移动的步数

为了得到一个正确的表格，需要在完成内部循环时前进一行

结合这两个函数，给出整个代码如下。

3. 源代码

```
#include <stdio.h>
#include <math.h>
int check_point(int i, int j);
void main (void)
{
        int n, i, j, icount;
        printf ("Number of possible moves for a knight "
                "on a chess board \n\n");
        for (j = 1; j <= 8; j++)
        {
                for (i = 1; i <= 8; i++)
                {
                        icount=check_point(i,j);
                        printf ("%5d", icount);
                }
                printf ("\n");
        }
}
```

为每一个点用函数 check_point 计算合法的移动数目

按套嵌环为每个点生成 i，j 值

合法的移动的数量赋给 icount

```
int check_point(int i, int j)
{
        int icount, p, k, n, m;

        icount=0;
        p = -1;
        for (k = -2; k <= 2; k++)
        {
                if (k != 0)
                {
                        for ( n = 1; n <= 2; n++)
                        {
                                p = -p;
                                m = p * (3 - abs (k));
                                if (i <= (i+k) && (i+k) <= 8 &&
                                1 <= (j+m) && (j+m) <= 8) icount++;
                        }
                }
        }
        return(icount) ;
}
```

计算句判断是否是合法的 k 值以及内部的循环

从 k 计算 n

如果 i，j，k 和 n 组合成一个合法的移动，那么合法的步数加 1

对于某点 i，j 返回合法的移动的数目

4. 输出

```
Number of possible moves for a knight on a chessboard:

 2   3   4   4   4   4   3   2
 3   4   6   6   6   6   4   3
```

```
4   6   8   8   8   8   6   4
4   6   8   8   8   8   6   4
4   6   8   8   8   8   6   4
4   6   8   8   8   8   6   4
3   4   6   6   6   6   4   3
2   3   4   4   4   4   3   2
```

注释

check_point 为骑士所写，我们可以为每个棋子写一个这样的程序，以计算任何棋子可以移动的步数。

修改练习

1. 修改程序，以处理 10×10 的棋盘，而不是 8×8 的棋盘。
2. 修改程序，以处理 15×23 的棋盘，而不是 8×8 的棋盘。
3. 写一个新函数，用来计算兵的移动。
4. 写一个新函数，用来计算车的移动。
5. 写一个新函数，用来计算象的移动。

应用程序 5.2 模块化程序设计：平行四边形面积和平行六面体体积（数值方法例子）

我们讲完了函数，下面讲解模块化程序设计。本例中只对展示模块设计的概念感兴趣。实际上，你也通常不会把本课中特定的程序写成和我们演示的完全一致。

问题描述

写一个程序，计算两个向量定义的平行四边形的面积和三个向量定义的平行六面体的体积。（如图 5-16）。从键盘输入向量的三个分量 *i*、*j*、*k*。把结果输出到屏幕上。使用模块设计并给予程序完备的注释。

图 5-16

解决方法

1. 相关公式

数学课上我们学过两个向量 *A*、*B* 定义的平行四边形的面积是两个向量叉乘后的长度。

$$A=a_1i+a_2j+a_3k$$

$$B=b_1i+b_2j+b_3k$$

$$Area=|A\times B|$$

其中 | | 代表向量的长度。

你也学过三个向量 *A*、*B*、*C* 定义的平行六面体的体积如下。

$$A=a_1i+a_2j+a_3k$$

$$B=b_1i+b_2j+b_3k$$

$$C=c_1i+c_2j+c_3k$$

$$Volumn=\mathrm{abs}[A\cdot(B\times C)]$$

2. 算法

分别读入三个向量 *A*，*B*，*C* 对应的分量

计算 *A*、*B* 定义的面积

　　计算 *A*、*B* 的叉乘

　　计算叉乘的模

计算 *A*、*B*、*C* 定义的体积

　　计算 *B*、*C* 的叉乘

　　计算 *A* 和 *B*、*C* 的叉乘的点乘（体积的绝对值）

输出结果

这个算法故意遗漏了很多细节，以帮助我们生成一个模块化的设计。从这个算法中得到了图 5-17 所表示的结构图。注意 main 主要用来调用其他的模块。一个好的模块设计不用 main 来做特殊计算，只是用它来调用那些做特殊计算的模块。这里选择把读入和输出也模块化。当大量数据要读入时，这种编程方法有很多好处。

图 5-17　面积 / 体积程序结构图

你会发现结构图和算法有很多相似之处。注意无论是面积和体积都需要计算两个向量的叉乘。这样，两个模块都用到了叉乘方法。

下一步确定每一个模块传入和传出的信息，表 5-2 列出了模块以及每个函数需要传入和传出的信息。我们把信息放到了图 5-18 所示的数据流程图中。

表 5-2　信息流

模块	传入函数的信息	传出函数的信息
向量模	一个向量的三个分量	向量的长度或模的值（1 个值）
叉乘	计算叉乘的两个向量，每个向量三个分量（6 个值）	叉乘后结果向量的三个分量（3 个值）
点乘	计算点乘的两个向量，每个向量三个分量（6 个值）	两个向量的点乘（1 个值）
平行四边形面积	计算面积的两个向量，每个向量三个分量（6 个值）	平行四边形面积（1 个值）
平行六面体体积	计算体积的三个向量，每个向量三个分量（9 个值）	平行六面体体积（1 个值）
输入	没有	三个向量，每个向量三个分量（9 个值）
输出	三个向量，每个向量三个分量，面积和体积（11 个值）	没有

有了结构图和每个模块的数据流程图，可以为某一个模块开发算法。当你刚开始编程时，你可能不会开发出很好的结构图和数据流程图，不要泄气。尽你所能开发出最好的，然后为每个模块开发算法。这样你会深入到问题的内部。你可以随时返回结构图和数据流程图来做调整。随着经验的丰富，你的结构图和数据流程图会变得越来越好，调整会越来越少。

下面是算法和函数原型：

函数原型：double dot_product()

算法：

接受两个向量的分量：g_1，g_2，g_3，h_1，h_2，h_3

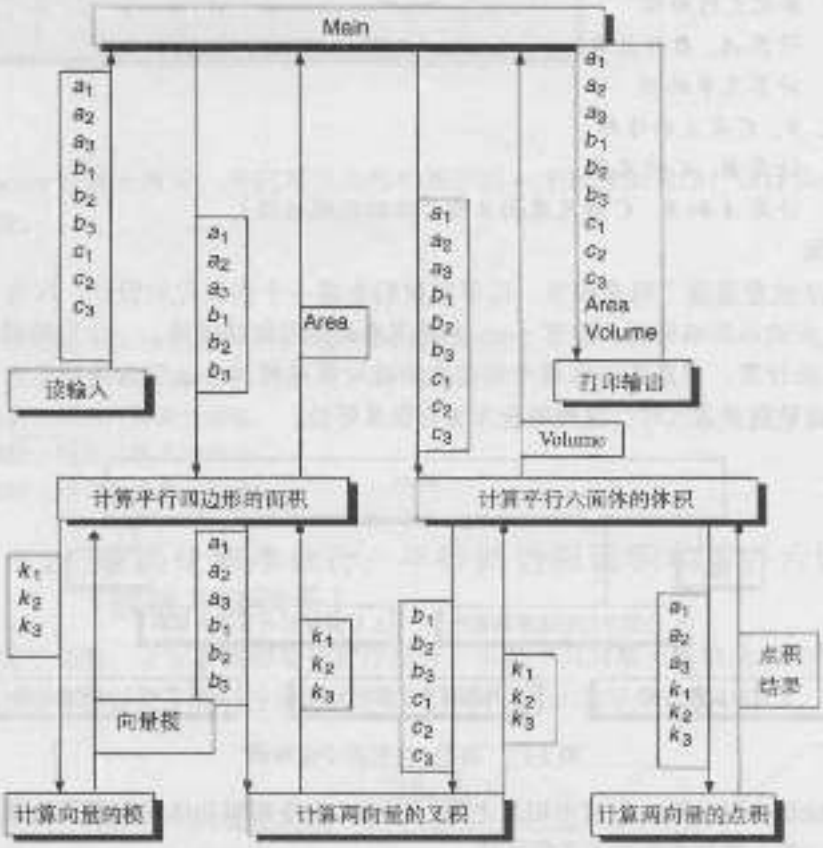


图 5-18

计算点乘：$G \cdot H = g_1h_1 + g_2h_2 + g_3h_3$

返回结果

函数原型：cross_product()

算法：

接受两个向量的分量：g_1，g_2，g_3，h_1，h_2，h_3

计算叉乘

$k_1 = (g_2h_3 - g_3h_2)$

$k_2 = -(g_1h_3 - g_3h_1)$

$k_3 = g_1h_2 - g_2h_1$

返回叉乘的三个分量：k_1，k_2，k_3

函数原型：vector_magnitude()

算法：

接受向量的分量：g_1, g_2, g_3

计算模：$(g_1g_1 + g_2g_2 + g_3g_3)^{0.5}$

返回结果

函数原型：area_parallegogram()

算法：

接受两个向量的分量：$g_1, g_2, g_3, h_1, h_2, h_3$

计算叉乘的模：$|G \times H|$

返回面积

函数原型：volumn_parallelepiped()

算法：

接受三个向量的分量：$g_1, g_2, g_3, h_1, h_2, h_3, k_1, k_2, k_3$

计算叉乘：$H \times H$

计算点乘：$G \cdot (H \times K)$

取绝对值

返回体积

函数原型：read_input()

算法：

不接受输入

输出提示到屏幕

接受三个向量的分量

返回向量的分量

函数原型：printf_output()

算法：

接受三个向量的分量，面积和体积

输出到屏幕

不返回任何值

3. 源代码

源代码作为练习，这里省略。

修改练习

1. 使用这里介绍过的模块化设计方法，计算下面的表达式（A、B、C、D 是用 i、j、k 三个分量表示的四个向量。）

应用练习

5.1　古希腊数学家欧几里得发明了一种计算两个整数 A 和 B 最大公约数的方法，如下：

a. 如果 A/B 的余数为 0，B 为最大公约数

b. 如果不是 0，B 赋给 A，A/B 的余数赋给 B

c. 回到步骤 a，重复此过程。

编写程序使用一个函数来执行这一过程，显示两个整数及其最大公约数。

5.2　成本分析是工程的一个重要部分。实践中，为不同的潜在场景，你需要写一个程序来计算最小的花费。你的程序可以用来作为一个工程的决策工具。

为了修建一个有跑道的机场，我们需要填充一块区域。承包商有两辆翻斗车。一辆车可以运八吨，另外一辆车可以运十二吨。承包商用卡车来向机场运土。八吨和十二吨的卡车每次的运输成本分别为 14.57 美元和 16.26 美元。每辆卡车载重不能超过总运输量的 60%。

写一个程序计算当给定吨数时所需要的最小成本。提示用户输入给定的吨数。输出每辆卡车需要

的运输次数及花费。使用模块化来设计本程序。

5.3 地震的强度可以用震级来表示。1935 年 Charles F. Richter 发明了一种量度，被称为里氏强度，用来确定一个地震的震级。地震中释放出来的能量、地震中断层断裂的长度以及世界范围内的地震的数量都与震级相关。用下面的近似公式来描述：

$$\log_{10} E \approx 11.8 + 1.5M$$
$$\log_{10} L \approx 1.02M - 5.77$$
$$\log_{10} N \approx 7.72 - 0.9M$$

其中 M = 里氏震级（$0 < M < 8.2$）

E = 释放的能量

L = 断层断裂的长度（公里）

N = 100 年内世界范围内地震的数量

公式大约有正负 20% 的变化量。写一个程序输入 E 和 L，计算这个地震的震级。告诉用户输入的数据是否完全不兼容。确定这个震级的地震发生的次数。

5.4 这是一个调试问题。本练习中你要先修改并调试一段几乎可以工作的给定的代码。这段代码见网址：www.mheducation.asia/olc/cprogramming。找到方程的根是几个世纪以来数学家感兴趣的问题。不幸的是，很多方程的根不能通过直接分析来得到。很多情形下，我们需要使用数值分析方法。很多数据分析方法用迭代来发现方程的根。虽然其他方法在一些函数上有效，但本质上就是不断地对解进行试错。

还可以用中值法来发现 $y = f(x)$ 的根。如图 5-19 所示。本练习的代码用来解方程 $y = 2x + 5$，并只用来解那些 x 增加时，y 也增加的函数类型。不能作用于那些 x 增加时，y 减少的函数类型。如 $y=-2x+5$。

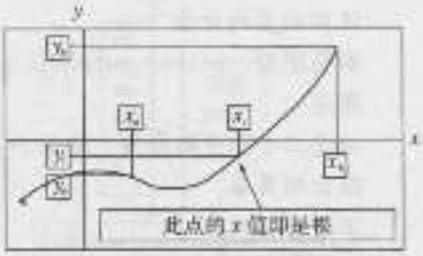


图 5-19

做下面的工作：

A. 修改程序以发现 $y = -2x + 5$ 的根。

B. 修改程序以发现 $y = x^3 - 5$ 的根。

C. 告诉用户在给定范围内的给定函数无根（比如 $y = x^3 + 5$，$-5 \leq x \leq 5$）。

D. 修改程序，寻找并输出一个精确到 5 位小数的根。

5.5 这是一个调试问题。本练习中你要先修改并调试一段几乎可以工作的给定的代码。这段代码见网址：www.mheducation.asia/olc/cprogramming。这段程序用来发现一个整数 N 的最大因子（除了它本身）。每次发现一个因子，如果一个整数 N 能被 2，3，4，…，$N/2$ 整除，那么它就是 N 的一个因子。本程序中有一个 bug，所以它只对一些特定例子才能工作。例如，它能发现 84 的最大因子是 42，但是对于 55 的最大因子，它没有发现。你能修改这段程序以便它能正确发现所有整数的因子吗？

注意如果 A 能被 B 整除，那么 B 和 A/B 都是 A 的因子。这样你就可以一次发现两个因子。另外，你只需要从 2 到 A 的平方根来查找因子（最大的 B 值就是当 $B=A/B$ 时。也就是说 B 是 A 的平方根。）在函数 find_max_divisor 中 if 语句的 false 语句块中使用这种方法，完成程序。

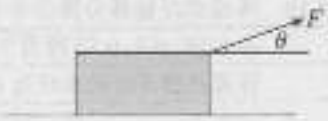
示例输出：

```
Please enter an integer
84
Please enter the method number (1 or 2)
1
Method1----- 2 is a divisor of 84
Method1----- 3 is a divisor of 84
Method1----- 4 is a divisor of 84
```

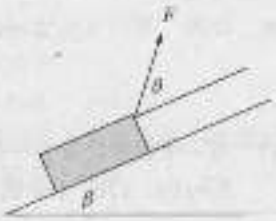
```
Method1----- 6 is a divisor of 84
Method1----- 7 is a divisor of 84
Method1----- 12 is a divisor of 84
Method1----- 14 is a divisor of 84
Method1----- 21 is a divisor of 84
Method1----- 28 is a divisor of 84
Method1----- 42 is a divisor of 84

Using method 1, the maximum divisor of 84 is 42
Please enter an integer
13
Please enter the method number (1 or 2)
1
Using method 1, the maximum divisor of 13 is 6
```

5.6　一个放在水平面的块，受到一个与水平面成 θ 角度的牵引力。块重 30kN，并且有 0.2 的摩擦系数。编写程序用函数计算当角度为 5、10、20、30、40、50、60、70 和 80 时需要多少力来牵引这块物体。

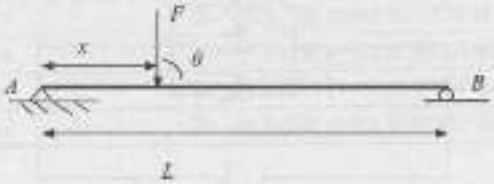


5.7　用 5.6 题中的块，考虑在与水平面成一个角度的平面上。当 β 为 0、10、20、30、40、50、60、70 和 80 时并且 θ 和 β 的和小于 90 时，解决 5.6 中的问题。



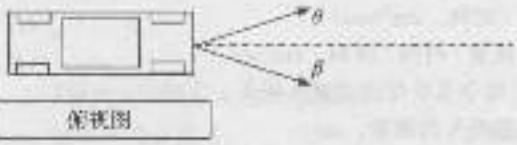
5.8　用 5.7 题中的条件，当摩擦系数分别为 0.1、0.2、0.3、0.4 时解决以上问题。

5.9　写一个程序计算梁支撑 A 和 B 上所受的力。其中 x=0、0.25L、0.5L、0.75L、L。F = 100、200、300、400、500kN。

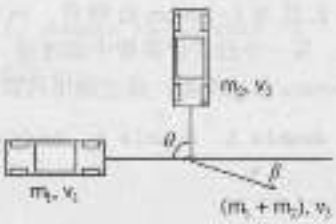


5.10　写一个程序当 θ=0、30、60、90 时解决问题 5.10。

5.11　要拖动一个车，需要在拖动的方向上施加 3kN 的力，同时在与拖动方向垂直的方向上没有力。分别考虑 θ 和 β 为 0、10、20、30、40、50、60、70 和 80 且 θ 和 β 的和小于 140 时，在缆绳上要施加多少力？



5.12　两车相撞后合在一起。写一个程序计算相撞后的方向和速度，使用下面的数据：m_1=100kg，m_2=3000kg，v_1=30m/s，v_2=10、20、30、40、50、60、70 和 80m/s。



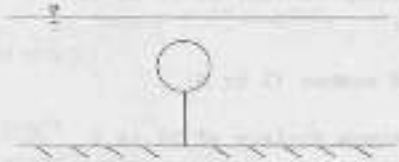
5.13 以下面的变量解决问题5.12。

θ=10、20、30、40、50、60、70、80、90度；m_1=1000、2000、3000kg；m_2=2000、4000、6000kg。

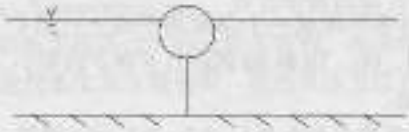
5.14 以下面的变量解决问题5.12。

θ=10、20、30、40、50、60、70、80、90度；v_1=10、20、30m/s；v_2=20、40、60m/s。

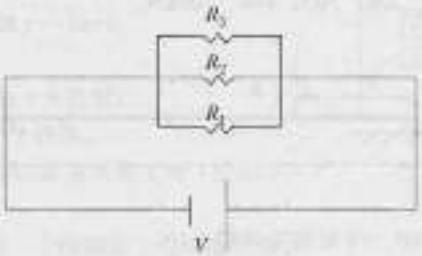
5.15 写一个程序计算拽住漂浮物的绳子所受到的力。漂浮物内为空气，水的密度为9.8kN/m³。计算当漂浮物的半径为1,2,3,4和5m时所受到的力。



5.16 考虑漂浮物部分浸没于水中时，写一个程序计算拽住漂浮物的绳子所受到的力。考虑总体比例的0.25、0.5、0.75浸没于水中。浸没于水的密度为9.8kN/m³和浸没于盐水的密度为10.05kN/m³。计算当漂浮物的半径为1、2、3、4和5m时所受到的力。



5.17 写一个程序计算每一个电阻中流过的电流。电压为80V，R_1=10、20、30欧，R_2=40、60、80欧，R_3=100、120、130欧。



5.18 重复题5.17，从1个电阻到5个电阻。R_1=50、70、90欧，R_3=140、160、180欧。

5.19 流经一个层状多孔介质的水（液体在平行层流动而不混合）遵循Darcy（法国工程师）公式。

$$Q = kiA$$

其中 Q= 流速（流量/时间，cm³/sec）

k = 通过系数（流量/时间/面积，cm/sec）

i= 水力学梯度（每单位长度流动的头损失，无单位，i=H/L）

H= 头损失（摩擦损失的能量，cm）

L = 多孔介质中流动长度

A= 流动发生时交叉区域面积

一个内部直径为D=10cm、长度为L=200cm的钢管，内部充满了沙子。沙子的通过系数k =0.1cm/s。头损失H=50cm。写一个程序计算管中的流速。没有键盘或文件输入。但是程序声明Q、k、i、H、L、A、D为double数据类型。通过调用函数flow_rate()来计算流速，原型如下：

```
void flow_rate(double D, double L, double k, double H);
```

按照下面格式输出到屏幕：

- 所有的 D,L,k 和 H
- i 和 x 的计算值
- 流速 Q

5.20　不用函数 flow_rate 计算 5.19。

5.21　下雨时，给定区域收集到的水流到排水沟或涵洞里。为了计算排水沟或涵洞的尺寸，水利工程师用比例的方法来计算排水的峰值速率。公式如下：

$$Q=CiA$$

其中 Q = 排水的峰值速率 (ft^3/s)[⊖]

C = 加权平均排水系数

i = 平均降水密度 (in/h)[⊖]

A = 附属于兴趣点 (acres) 的分水岭 (降雨区域)

注意 C 的值取决于土地类型，在农村区域，C 的值如下：

土地类型	C	土地类型	C
混凝土	0.9	耕犁土地	0.3
裸露土地	0.6	森林	0.2

如果分水岭区域包含不同类型的土地类型，C 应该用加权平均的方法来计算。例如，如果一个区域 20% 覆盖混凝土，30% 裸露土地，50% 森林，那么加权平均 C

$$C=[(0.9\times20\%)+(0.6\times30\%)+(0.2\times50\%)]=0.46$$

写一个程序计算给定区域的排水的峰值速率。通过调用 read_data ( ) 函数来从一个类似于下表中的输入文件中读入数据。(表中第二列为保留用，不会出现在输入文件中)

6.9		第一个行为降雨密度 i = 6.9in./sec
100	0.9	第一列为类型 C=0.9 的局部区域的面积 (100 acres)
200	0.6	第一列为类型 C=0.6 的局部区域的面积 (200 acres)
300	0.3	第一列为类型 C=0.3 的局部区域的面积 (300 acres)
150	0.2	第一列为类型 C=0.2 的局部区域的面积 (150 acres)

函数的原型为

```
double read_data(double *i, double *A);
```

这个函数完成：

- 计算总共的分水岭面积 A (A = 100+200+300+150 = 750 acres)。
- 计算加权平均系数 C 并返回给 main。
- 在 main 函数中，计算 $Q=CiA$。

把下列数据输出到屏幕：

- i 的原始数据，每一个区域的面积和它的排水系数。
- 整个分水岭的面积 A，加权平均系数 C。
- 排水的峰值速率 Q。

5.22　以下面给出的函数原型解决问题 5.21。

```
void read_data(double *i, double *A, double *C);
```

5.23　下表给出了不同土壤的渗透系数 k

⊖ 1 ft^3=0.0283m^3。

⊖ 1 in=0.0254m。

土壤类型	渗透系数 k 的范围	土壤类型	渗透系数 k 的范围
粉土	1.0E-10 到 1.0E-8	沙子	1.0E-4 到 1.0
黏土	1.0E-8 到 1.0E-4	石头	1.0 到 100.0

给定土壤的渗透系数 k，写一个程序来确定它的类型。输入部分规范：调用 soil_type() 的函数来完成：

- 从键盘读入渗透系数 k。
- 发现渗透系数的范围。
- 确定土壤类型。

函数原型为：

```
void soil_type(void);
```

输出部分规范：把下面内容输出到屏幕

- 用户从键盘读入渗透系数 k。
- 土壤类型。

5.24 加热器消耗的功率可以用下面的公式来计算：

$$p=vi$$

其中 p = 功率（瓦）

v = 电压（伏）

i = 电流（安）

写一个程序来计算加热器的功率。输入规范：从下面的文件读入电压和电流。

电压	电流
110	5.5
220	23.5
90	13.6
370	44.4

输出规范：把下面的内容输出到屏幕。

输入的电压和电流

功率

例如，当输入的文件第一行读入后，应该显示如下：

```
Input voltage = 110.0 volts, current = 5.5 amperes
Power consumed by the heater = 605.0 watts
```

5.25 重做 5.24。这次显示每个加热器的电阻的值 $R(R=v/i)$。当输入的文件第一行读入后，应该显示如下：

```
Input voltage = 110.0 volts, current = 5.5 amperes
Power consumed by the heater = 605.0 watts
Heater resistance = 20.0 ohms
```

本章回顾

本章中学习了如何生成用户定义函数以及如何在程序中使用它们。函数中参数的数目、顺序和类型必须与它的定义一致。使用函数原型可以使程序认识我们的函数。另外，使用地址操作符 & 和取值操作符 * 可以使得我们通过函数参数列表从函数中传入和传出数据。

数值数组

本章目标

完成本章的学习以后，你将可以：

- 对于相同数据类型的一组数据定义一种存储方式。
- 区分出使用数组作为一种数据结构的优点。
- 在应用程序中将数组作为一种数据结构来使用。

一种有用的高级编程语言都内置一些可以帮助简化编程任务的属性。数组就是C语言中这样的一个属性。

数组是C语言中的一个数据结构。它是一组相似类型的数据。简单来说，一个数组可以代表一系列数：例如，某个气象站记录的一年内的每小时的温度记录。这一系列数都代表温度，因而是同一数据类型。

另外，也可以用数组表示一个数据点中的 x 和 y 坐标。如果有 10 000 这样的数据点，那么需要 10 000 个 x 坐标和 10 000 个 y 坐标。为此，我们可以构建两个数组，一个存储 x 坐标，另外一个存储 y 坐标。在C语言中，数组用一个标识符外加一个方括号来表示，同时方括号内部包含一个整型常数或代表一个整型常数的表达式。例如，某个数据点的 x 坐标可以被表示成 $x[129]$，另外某个数据点的 x 坐标可以被表示成 $x[4976]$（在上面的例子中，方括号中可以是从 0 到 9999 的任何整数）。

数组能很方便地表示一组相同类型的数据，因为我们可以很容易写出计算和管理这些数据的表达式。方括号里面的数值通常叫做下标或引用。数组下标的用法和代数表达式中下标的用法非常类似。例如，某条线段经过（x_1，y_1）和（x_2，y_2）两个点，为了计算这条线段的斜率 m_1，我们可以使用下面的代数表达式

$$m_1=(y_2-y_1)/(x_2-x_1)$$

利用C语言的数组，我们可以写成下面的赋值语句

```
m[1] = (y[2] - y[1])/(x[2] - x[1]);
```

请注意，在代数表达式和赋值语句之间的对应性。如果我们想计算不同对点之间的斜率，可以把数组放到一个循环里面，如

```
for (i=0; i<9999; i++)
{
        m[i] = (y[i+1] - y[i])/(x[i+1] - x[i]);
}
```

在所有的数据点内循环

计算连接点 i 和 i+1 之间线段的斜率

目前，你不需要完全理解这个循环的含义，但我们只能说这几行可以用来计算所有链接 10000 个点的线段的斜率。如果没有数组，我们不可能如此简单地完成这个任务。这里，我们可以发现数组是非常有用的。

在本章中，我们将介绍数组的概念以及如何用数组完成对应的工作。本章末尾的应用程序演示了很多数组的实际应用。利用本章所学知识，你可以写出非常有用和复杂的程序。

课程 6.1 一维数组和打印数组元素介绍

主题

- 定义数组
- 数组的特性
- 利用 #define 预处理指令定义数组的尺寸
- 打印数组的元素

与一般的变量类似，数组必须在程序使用之前声明。数组中的元素可以利用赋值语句进行赋值。本课程的程序声明了两个数组，对其中的一些元素进行了初始化并把它们输出。

十一章详细地介绍了一些程序。现在，你应该对阅读源代码很熟悉了。所以，从现在开始，我们只是列出每一个程序中你应该关注的部分，以及在阅读解释部分之前你应该尝试回答的关于程序的一些问题。为了能从本书中获益更多，你需要仔细研究源代码及对应的列表，观察并且回答那些问题。

下面是从本课程的程序中应该观察的事情：

1）第一个声明中，a[ ]被声明为一个包含整型数的数组。

2）第二个声明中，b[ ]被声明为一个包含双精度浮点数的数组。

3）两个声明中都使用了方括号。方括号中是代表数组中元素个数的一个整数，利用这个数字在内存中为数组中的元素预留出空间。

4）对于a[ ]和b[ ]来说，我们只使用了一对方括号，这代表着它们都是一维数组。

5）a[ ]数组被声明为包含两个元素。

6）b[ ]数组被声明为包含十个元素。

7）赋值语句对a[ ]中所有的元素进行了赋值操作。

8）赋值语句只对b[ ]中两个元素进行了赋值操作。

9）在头两个printf语句中，我们打印了所有数组中被赋值的元素。

10）在第三个printf语句中，我们输出了之前并没有被赋值的b[2]的值，由于没有被赋值，输出来的是没有意义的随机值。

11）在第四个printf语句中，即使a[ ]已经被声明为只有两个元素，我们也可输出a[3]的值。这个时候a[3]输出是一个无意义的随机值。

12）上面的10和11对应的是两种程序错误，C语言的编译器并不会对这种错误给出警告或错误提示。

下面是你应该在阅读解释部分前，首先试图回答的一些问题。

1）用一个常量宏来定义数组的尺寸（或者为这个数组的元素预留空间的数目）有什么好处？

2）当我们超过了数组a的边界去打印a[3]的时候，为什么程序并没有提示出错？

源代码

```
    a[0]=11;
    a[1]=22;
    b[3]=777.7;
    b[6]=888.8;

    printf("a[0] = %3d, a[1] = %3d\n", a[0],a[1]);
    printf("b[3] = %8.2lf, b[6] = %8.2lf \n", b[3],b[6]);
    printf("b[2] = %lf\n", b[2]);
    printf("a[3] = %d\n", a[3]);
}
```

这个语句把元素填充到数组a

b数组中一个没有初始化的元素被输出，在执行的时候不报告错误

数组a已经被声明为包含两个元素，但是当我们输出超过两个允许的范围后，在执行的时候不报告错误

上面两种情况，注意输出的结果没有任何意义

输出

```
a[0] =11, a[1] =22
b[3] = 777.70, b[6] = 888.80
b[2] = -336606442844569 64.000000
a[3] = 373
```

解释

1）什么是一维数组，我们如何声明它？

一个一维数组就是一组相同数据类型，在内存中以连续和递增方式保存的一个集合。它由名字、类型、维数及元素个数四个部分来确定。

例如，语句

```
int a[2];
```

声明了数组的名字是a，数组元素的类型为int，维数为一维（它只有一对方括号），数组的元素个数为2（代表内存中有两个元素的空间被保留）。

通常，定义一维数组的语法如下：

```
element_type  array_name [number_of_elements];
```

这里，element_type代表的是数组中元素的类型，例如int、float、double或者任何其他的合法的C语言类型，除了void类型和function类型。array_name是数组的名字；number_of_elements代表能在数组中最多保存多少个元素。这个数字必须是一个正整数。

2）如何命名一个数组？数组名也是标识符。所以数组名必须遵循标识符命名的那些规则。数组名在其声明中给出。例如以下五个是合法的数组名：

```
c[15],    f23_rack[60],
a[ 10], fruit[300 ] and country[100].
```

无效的数组名也是非法的标识符，例如以数字开头或者包含其他非法的字符。如下面三个数组名：

```
12abd[15 ], y4#[30 ] and io*jk[70 ].
```

3）如何区分一维、二维和三维数组？一个一维数组名后面只有一对括号。二维数组名

后面有两对括号，三维数组名后面有三对括号。如下面的声明：

```
int        b[2][4],          c[6][9][5];
           二维数组            三维数组
```

接下来的一些课程中，我们把讨论限制在一维数组中。

4）如何确定一维数组的长度？一维数组的长度（也叫元素的数目或尺寸）由程序员基于特定的问题来决定。一维数组的长度由括号内包含的表达式的值来确定。长度必须是一个大于0的整数。一维数组的长度可以显式定义如下：

```
int a[2], c[20], g[100];
```

另外一个定义数组尺寸的通用方法是用预处理命令定义一个常数，如本课程的程序中使用

```
#define N 10
double b[N];
```

因为允许整数表达式，下列的声明也是合法的（与#define N 10一起使用）

```
int  c[100+N], d[50*N];
```

下列为非法的元素个数：

```
int   c[-25], b[32.5]
```

它们之所以非法是因为它们不是正整数。

注意，你可以在同一行定义单个变量和数组。

```
double b[N], f, g, h;
```

这是合法的。C编译器知道b是一个数组，因为它后面跟着括号，而其他的变量后面没有括号。

5）如何计算为一维数组预留多少内存？通过将在数组声明中给出的元素的个数乘以每个元素所需内存的数量，可以在声明一个数组时知道为它预留多少内存。例如，如果一个int占2 B，那么有两个元素的数组a[]占2×2=4 B。同理，长度为10的double b[]占10×8=80 B（每个double占8 B）。

6）用预处理命令定义的常数来声明数组的长度有什么好处？用常数符号来限定数组的长度可以使你的程序更加灵活。例如你有100个一维数组，它们的长度都是9，也就是说a1[9], a2[9],…, a100[9]。现在你想把长度从9变为25，你需要改变100次。如果使用常数定义，则可以减少工作量和发生错误的机会。如a1[N], a2[N],…, a100[N]。当把N在预处理命令中定义时，唯一需要修改的就是在预处理命令中把9改为25。

7）C语言中数组的第一个索引（也叫下标）的值是什么？在C语言中，默认的第一个索引或下标为0。一些学生甚至有经验的程序员通常会忘掉C以0开始而不是以1开始（有些语言以1开始）。这会导致混淆和错误。例如，本课程的程序声明了

```
int a[2];
```

这意味着数组a[]有两个元素。因为C以0开始，所以本程序用a[0]和a[1]来使用这两个元素。

```
a[0] = 11;
a[1] = 22;
```

观察到在程序中没有使用 a[2]。这是因为声明中使用 a[2] 与程序体中使用 a[2] 代表完全不同的含义。在程序体中 a[2] 代表数组 a[] 的第三个元素。因为 a[] 只有两个元素，所以我们不应该在程序体中使用 a[2]。

8）如果在程序体中使用 a[2]，C 语言是否报错？不！C 语言并不检查数组越界。如果程序使用 a[2]，C 只是去 a[1] 后面的一个位置去取出一个值。程序继续使用这个值运行并给出结果（结果可能是错的。）

写程序的时候，你应该查看结果是否错误并发现源代码中的错误。当程序中包含很多的数组和很复杂的逻辑时，你很难发现那个数组访问造成了数组越界。如果内存某个位置的值没有意义，你可能会得到一个运行时错误，而这更加难以追踪。所以当你编写程序的时候，一定要小心不要超过数组的长度。记住你的数组从 0 开始并终止于数组长度小于 1 的地方。

出于演示的目的，本课程的程序输出了 a[3]。很明显它超出了数组的有效区域。但是无论是编译器还是执行时都不报错。整数 373（在本课程的程序的上下文中它没有意义）保存在 a[3] 代表的内存单元中。我们选择用一个 printf 语句来输出它。我们也可以把它用在赋值语句的右边或者在其中保存一个新值。因为它在声明的有效范围之外。使用 a[3] 会带来整个程序的混乱。因为在 a[3] 中保存的值是不可预知的。（如果在你的计算机上运行这段程序，你可能得到一个不是 373 的数。）如果把一个新值保存到 a[3]，我们可能把一个有用的变量覆盖掉。总之要注意：如果程序中发现了无意义的结果，你应该检查是否有数组越界。

9）如何使用 printf 输出数组元素？我们把一个数组元素当成一个单独的变量。例如语句

```
printf("a[0] = %3d, a[1] = %3d\n", a[0],a[1]);
printf("b[3] = %8.2f, b[6] = %8.2f \n", b[3],b[6]);
```

输出 a[0]、a[1]、b[3] 和 b[6]。

注意，数组的类型决定了输出其元素时使用哪种格式。例如，为了显示 int 类型的数组，使用 %d 格式；为了显示 float 类型的数组，使用 %f、%e 或者 %g 格式。

10）在本程序中，我们给 b[3] 和 b[6] 赋值。但是 b[] 声明为有 10 个元素，b[] 中其他的元素的值为多少？无意义的值。因为没有为 b[] 中其他的元素赋值，所以这些元素没有存储有效值。

```
printf("b[2] = %lf\n", b[2]);
```

输出 b[2] 的值。值 233660644284456964.000000 只是为 b[2] 预留的内存位置中的位的表示。如果运行这段程序，你可能会得到一个不同的值。所以在使用一个数组元素前，必须要初始化它们。注意，这个程序在编译和运行的时候都不报错。但是 b[2] 中的结果没有意义。所以如果你发现程序得到了无意义的结果，应检查你是否初始化了数组中的所有元素。接下来的课程演示了初始化数组元素的几种方法。

概念回顾

1）声明一个一维数组，需要定义数组的名字和其包含的元素的数目。例如 int marks[50] 定义了一个名字叫 marks 的包含 50 个元素的数组。

2）数组的索引总是以 0 开始，这样 marks 的索引从 0 到 49。

3）整个数组保存在一个连续的内存空间上。这意味着元素 i 紧邻元素 i+1。

4）如果程序试图越过数组的边界，编译器不报错，但是程序执行的时候也许会带来严重的后果。

练习

1. 判断真假。

a. 给定数组中的所有元素都有相同的数据类型。

b. 给定数组中的所有元素随机分布在内存中。

c. 给定数组中的所有元素可以用不同域宽的格式显示。

d. 一维数组第一个元素的下标为1。

e. 一维数组有99个元素，它的长度为100。

2. 下面的语句是否有错误，如果有错误，请指出。

```
a. int a, b[2];
b. float a23b[99], lxy[66];
c. void city[36], town[45];
d. double temperature[-100];
e. long phone[200];
```

f. 数组中的第一个和最后一个元素分别为phone[1]和phone[200]

3. 在下面的程序中发现错误。

```
#define (N=2)
void main(void)
float a[N],b;
a[1]=N;
N=99;
a[2]=N;
}
```

答案

1. a. 真　b. 假　c. 真　d. 假　e. 假

2. a. `int a, b[2];`

b. `float a23b[99], xy1[66];`

c. 数组不能是void类型。

d. `double temperature[100];` 下标必须大于0。

e. 没错

f. 第一个和最后一个元素分别为phone[0]和phone[199]。

课程6.2　数组初始化

主题

- 数组初始化和声明

上次课中看到了将一个程序中要使用的数组中元素进行初始化是非常重要的。我们每次给一个元素进行赋值。第一次给数组元素赋值（或给单个变量赋值）叫做初始化元素或初始化变量。

可以在声明中初始化数组。但是对于数组来说，情况有点不一样。因为数组有很多元素，如果想在声明中初始化很多元素，则需要在声明中列出很多值。C语言中用{}来包含用来初始化数组元素的值。

下面是关于本课卷的程序观察到的一些结论：

1）数组a[ ]被声明有3个元素。两个元素在声明中被包含在括号{ }中的值初始化。

2）数组b[ ]没有显式地声明尺寸，但是3个元素在声明中被初始化。

3）第一个printf语句输出了从a[0]到a[2]以及从b[0]到b[2]的值。a[2]的值为零，

虽然它没有被显式地初始化。

4）数组 x[ ] 和 y[ ] 只是被声明，但是没有在声明中被初始化。

5）scanf 语句从键盘读入值，并保存到 x[0] 和 x[1] 中。

6）变量 i 被用在循环里，充当数组 y[ ] 的下标。

7）for 循环初始化 y[ ] 数组的所有 10 个元素的值。

在阅读解释部分之前，尝试回答下面的问题。

1）b[ ] 数组中有多少元素？

2）for 循环执行了多少次？

3）for 循环把数组 y[ ] 初始化成什么值？

源代码

初始化 a[ ] 数组的前两个元素，其他元素被自动设置为零

因为没给出数组大小（方括号内为空），并且在大括号内给出 3 个值，数组被自动声明大小为 3，并用大括号内的值初始化

```
#include <stdio.h>
void main(void)
{
        int a[3]= {11,22}, b[]={44, 55, 66}, i;
        double x[2],y[10];

        printf("a[0]=%2d, a[1]=%2d, a[2]=%2d \n"
               "b[0]=%2d, b[1]=%2d, b[2]=%2d \n\n",
                a[0],a[1],a[2],b[0],b[1],b[2]);

        printf("Please enter two real numbers\n");
        scanf("%lf %lf",&x[0], &x[1]);
        printf("x[0] = %.11f    x[1] = %.11f\n\n", x[0], x[1]);
        for (i=0;i<10;i++)
        {
                y[i]= i*100.0;
                printf("y[%1d]=%.21f\n", i, y[i]);
        }
}
```

读入两个数组元素

用循环填充数组 y[ ] 的所有元素

输出

```
                a[0]=11, a[1]=22, a[2]=0
                b[0]=44, b[1]=55, b[2]=66

                Please enter two real numbers
键盘输入        77.0 88.0
                [0] = 77.0     x[1] = 88.0

                y[0]=0.00
                y[1]=100.00
                y[2]=200.00
                y[3]=300.00
                y[4]=400.00
                y[5]=500.00
                y[6]=600.00
                y[7]=700.00
                y[8]=800.00
                y[9]=900.00
```

解释

1）哪两种方法可以在声明中初始化一维数组？一维数组中的元素可以在声明中使用以下两种方法进行初始化，如图6-1：

① 声明一个数组，在方括号内指明元素的个数，并紧随在大括号中给出一些元素的值。例如，声明

```
int a[3]={11,22};
```

初始化 a[0] =11, a[1] = 22。注意，虽然以上语句只是显式初始化了数组中前两个元素，第三个元素 a[2] 自动初始化为0。

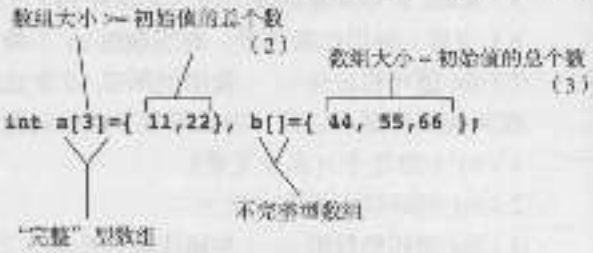


图6-1 在其声明中初始化一个数组

通常使用以下方式初始化一些元素，并把其他设置为零。

type name [*number_of_elements*]={*value_0*, *value_1*, ... *value_n*};

其中 *n* 要小于或等于数组中元素的个数（number_of_elements）。下标从 [*n*+1] 到 [*number_of_elements*-1] 的数组元素自动初始化为0。

② 在数组声明中，方括号内不包括元素的个数，紧随一个大括号，包括数组元素的值。例如，声明

```
int b[]={44, 55, 66};
```

自动把数组 b[] 声明为有3个元素（b[0]、b[1] 和 b[2]），这是因为大括号内有3个元素。本例中如果我们试图存取 b[3]，则会超过 b[] 的范围。通常这种声明有以下的方式：

type name [*number_of_elements*]={*value_0*, *value_1*, ... *value_n*};

这种方式初始化数组中所有元素。数组中有 *n*+1 个元素。

这种方法来声明和初始化数组有两个优点，首先它不需要对数组中元素的个数进行计数，其次当我们修改程序增加更多元素时，也不需要修改指示数组中一共包含了多少元素的值（因为这个值在声明中没有给出）。缺点是对于程序员来说没有显式指明数组中有多少个元素，这可能会引发越界错误。所以，在本文中当我们声明并初始化数组时，主要使用第一种方法。

2）有没有其他的方法初始化数组？可以首先声明数组，然后使用一个输入函数，例如 scanf（）来初始化。

另外的方法就是先声明数组，然后用赋值语句一个一个地进行初始化。

3）本课程的程序中，语句

```
y = 100.;
```

会把数组 y[] 中的所有元素初始化为100吗？不会。在C语言中，我们必须分别初始化每个元素。同样，如果我们想修改整个数组，也必须一个一个修改。

4）声明

```
int b[2]={44, 55, 66};
```

会引发错误吗？是的，因为我们指出 b 有两个元素，但却列出了3个值。虽然C语言不检查越界错误，但是它会检查这种错误。

概念回顾

1）一维数组中的元素可以在声明中或程序体中初始化。

2）当一维数组在声明时没有指明元素的个数，声明会自动根据初始化值的个数来决定数组元素的个数。

练习

1. 判断真假：
   a. 有多于一种方法来初始化一维数组
   b. 一个不完全的数组可以用 scanf 函数来初始化
   c. 给定数组的初始化值的数目必须小于或等于数组的尺寸
2. 下面的语句是否有错误，如果有错误，请指出。

```
a. int a={11,22},b[33];
b. float c[3]={11,22,33,44};
c. double d(4)={11,22,33,44};
d. d[4]={11 22 33 44};
e. int a[3]/11,22,33/;
```

3. 给定某个星期，从周一到周日从桥上通过汽车的数量为 986、818、638、763、992、534 和 683。用这些值初始化一个数组并写一个程序产生以下输出。通过桥的日平均车辆数为 773，周五时通过桥的车的数量最大，为 992。

答案

1. a. 真　b. 假　c. 真
2. a. int a[2]={11,22},b[33];
   b. float c[4]={11,22,33,44};
   or
   float c[ ]={11,22,33,44};
   c. double d[4]={11,22,33,44};
   d. Must show array data type.
   e. int a[3]={11,22,33};

课程 6.3　基本数组输入输出

主题

- 从一个文件中将数据读入数组
- 把数组中的数据写入文件
- EOF 作为文件末尾的标志符
- 在算术表达式中使用数组

在很多实际的问题中，数组通常有很多元素。因此，通常情况下，数组的值不是来自于键盘输入，而是来源于数据文件。同时，将数组值打印到屏幕也是不现实的。因为数组可能包含很多的元素，输出的时候会多于一个屏幕，而且也无法管理输出到屏幕上的值。这样输出结果到一个数据文件也是非常必需的。本课中演示了如何从文件中输入和输出数据。一个带有 fscanf 函数的 while 循环以及一个叫做 EOF 的常数被用来从文件中读入数据。

```
while ( fscanf(, , , ,)  != EOF)
```

fscanf 被反复调用直到它返回 EOF。EOF 以一个宏的方式存在于 stdio.h 中，它代表文件

的末尾。当 fscanf 试图越过文件的末尾去读文件时，fscanf 返回 EOF（在大多数的情况下被操作系统设置为 -1）。有了这个背景知识，读本课的源代码，并在源代码中观察以下事实：

1）3 个数组被声明。

2）数组 x[ ] 和 y[ ] 的第一个元素被第一个 fscanf 语句读入。

3）第一个 fscanf 语句在赋值语句的右侧，这代表 fscanf 返回一个值。fscanf 返回的值是一个整数，并赋给了变量 k。k 的值被输出。

4）本程序中使用的 EOF 没有在变量列表中声明。它的值用第二个 fprintf 语句输出。

5）while 循环中的条件语句将 fscanf 返回的值和 EOF 比较。循环中数组下标依次增加。

6）for 循环遍历了所有的数组元素。在 for 循环中，数组元素被管理和输出。

阅读解释前，尝试回答以下问题。

1）EOF 来自哪？

2）while 循环会执行几次？

3）while 循环后 i 值为多少？

源代码

```
#include <stdio.h>
#include <math.h>
void main(void)
{
        int i, j, k, num_elem;
        double x[20], y[20], z[20];
        FILE *infile, *outfile;

        infile = fopen ("C6_3.IN","r");
        outfile= fopen ("C6_3.OUT","w");

        k = fscanf(infile,"%lf %lf",&x[0],&y[0]);
        fprintf (outfile, "k = %d\n",k);

        fprintf (outfile,"Value of EOF = %d\n", EOF);

        i = 1;

        while ( fscanf(infile,"%lf %lf",&x[i],&y[i])   !=   EOF)  i++;

        num_elem = i;

        fprintf(outfile,"   x[i]     y[i]     z[i]\n");

        for (j=0;j<num_elem;j++)
        {
                z[j]=sqrt(x[j]*x[j]+y[j]*y[j]);
                fprintf(outfile,"%7.1f %7.1f %7.1f\n",x[j],y[j],z[j]);
        }

        fclose(infile);
        fclose(outfile);
}
```

把所有数组的大小声明为 20

fscanf 函数可以用在赋值语句的右侧，因为它会返回一个整数，代表它读入了几个值

EOF 是一个定义在文件 stdio.h 中的常数宏。当 fscanf 越过文件的末尾去读文件时，它会返回 EOF

执行循环直到 fscanf 返回 EOF。循环中 i 的值递增，以便计算数组中元素的个数

和数组中的每个元素配合工作，所以通常我们需要生成一个循环以遍历数组中的所有元素

z[ ] 数组中的每个元素都是从对应的 x 和 y 数组计算得到

输入文件 C6_3.IN

```
3.0   4.0
6.0   8.0
9.0   12.0
```

输出文件 C6_3.OUT

```
k = 2
Value of EOF = -1
   x[i]       y[i]       z[i]
    3.0        4.0        5.0
    6.0        8.0       10.0
    9.0       12.0       15.0
```

解释

1）怎样使用 fscanf 函数从文件中读入一个数组？从文件中利用 fscanf 函数读入一个数组与从文件中读入一个标量类似，如图 6-2。打开输入文件，利用函数 fscanf 将数组元素一个接一个读入。注意在处理每个数组元素之前必须加上 & 符号。例如表达式：

```
fscanf(infile,"%lf%lf",&x[0],&y[0]);
```

从 infile 文件指针指定的文件中读入两个 double 类型数据。数据被分别保存在数组 x 和 y 的第一个元素中。

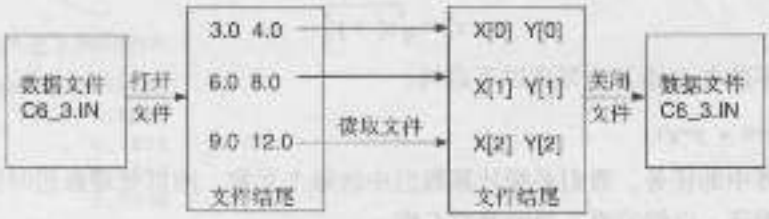


图 6-2 利用数组从文件读入数据

2）fscanf 会返回一个值吗？如果是，能用在赋值语句的右侧吗？是的，函数 fscanf 会返回一个整数，代表调用时成功读入值的数目。本课程序中，

```
k = fscanf (infile, "%lf%lf", &x[0], &y[0]);
```

使得 fscanf 读入两个值到 x[0] 和 y[0]，所以返回整数 2。因此在赋值语句结束后 k=2 并且输出。

3）什么是 EOF？EOF 是一个定义于头 stdio.h 文件中的常量宏。像以前在程序中使用的常量宏那样，EOF 都是大写字母（代表 end of file）。ANSI C 要求 EOF 等于一个负整数。但是在大部分的情形下，它都等于本课演示的 -1。

4）EOF 怎样被用在 C 语言的标准库中？有一些函数使用 EOF，最为显著的就是 fscanf 函数。当一个输入错误发生在它不能成功读取下一个值前，函数 fscanf 返回 EOF。换句话说，如果 fscanf 在读取任何数值之前碰见了文件的末尾，那么调用返回 EOF（大部分情况下等于 -1）。

5）如何在程序中使用 EOF？当从数据文件中读入数据并且不知道这个数据文件中有多少

个值时，EOF 变得非常有用。例如本课程序，数组 x[] 和 y[] 可以包含 20 个元素，这意味着数组最多可以包含 20 个元素，或者也可以更少。输入文件 C6_3.IN 只给出了 3 个数组元素。但是我们事先不知道，所以使用一个循环来读入数据。

```
while ( fscanf(infile,"%lf %lf",&x[i],&y[i]) != EOF)  i++;
```

使得 fscanf 读入整个文件直到末尾。每次它读入两个值（一个给 x[]，一个给 y[]）同时递增 i 的值。在本例中，当 fscanf 试图读取 x[3] 和 y[3] 时，由于遇到文件末尾，它失败了。这样 fscanf 返回一个 EOF 以终止循环。

6）如何在算术表达式中使用数组？必须依次使用数组中的元素，而且我们经常在循环中使用数组。例如：

```
for (j=0; j<num_elem; j++)
{
        z[j]=sqrt(x[j]*x[j]+y[j]*y[j]);
}
```

首先计算

```
z[0]=sqrt(x[0]*x[0]+y[0]*y[0]);
```

循环中第二次计算

```
z[1]=sqrt(x[1]*x[1]+y[1]*y[1]);
```

这个过程直到 z[] 中 num_elem 个元素被计算。这个表达式的数学公式为：

$$z_i = \sqrt{x_i^2 + y_i^2}$$

注意我们不能在 C 语言中写出以下语句。

```
z=sqrt( x*x + y*y);
```

它不会完成循环中的任务。我们必须计算数组中的每个元素。所以处理数组时要花费很多精力来写出一个循环，以便完成正确的管理工作。

7）如何使用 fprintf 语句把数组输出到文件？用 fprintf 语句把数组输出到文件就像用 fprintf 语句把标量输出到文件一样。打开一个文件，用 fprintf 语句将数组元素依次写到文件。我们通常利用循环完成这个任务。例如循环

```
for (j=0; j<num_elem; j++)
{
        fprintf(outfile,"%7.1f%7.1f%7.1f\n",x[i],y[i],z[i]);
}
```

把保存在 x[i],y[i] 和 z[i] 中的 double 类型的三个数据写到文件指针 outfile 指定的文件中。

概念回顾

1）xCoord 是一个独立的变量，x[4] 是数组中的第 4 个元素。x[4] 当成单独的一个变量时与 xCoord 用法一致。

2）EOF 是文件末尾指示符。当从一个文件读入时，库函数 fscanf 读到文件末尾返回 EOF。

练习

1. 基于

```
int A[3]={1,2,3}, B[3]={4,5,6}, C[3]={1,2};
FILE *in;
```

判断下列语句是否正确：

a. `C=A+B;` 会把数组 A 和数组 B 内的元素相加，并把和保存到 C 中。

b. `C[0] = A[1]+B[2];` 是不对的，因为 0、1、2 是不同的下标。

c. `fscanf(in, "%d %d %d",&A);`

可以用来从数据文件中读入数据并保存在数组 A 中。

2. 下面的语句是否有错误，如果有请指出。（i[5] 是一个 int 类型的数组，f[6] 是一个 float 类型的数组，in 是一个文件指针。）

```
a. fscanf(in,"%d %d",i[2],i[4]);
b. fscanf(in,"%d %d",&i[2], &i[4]);
c. fscanf("in,%f %d",&i[2],&f[2]);
d. fscanf(in,"%d %f",&i[5],&f[6]);
e. fprintf(in,"%d %d",i[2],i[4]);
f. fprintf(in,"%d %d",&i[2], &i[4]);
g. fprintf("in,%f %d",&i[2],&f[2]);
h. fprintf(in,"%d %f",&i[5],&f[6]);
```

3. 使用数组读入下面的输入文件。

```
1  30.0
2  45.0
3  60.0
4  90.0
```

然后处理数据产生下面的输出。

```
N    X(degree)    cos(X)
1    30.0         0.8667
2    45.0         0.7071
3    60.0         0.5000
4    90.0         0.0000
```

答案

1. a. 错　b. 错　c. 错

2. a. `fscanf(in,"%d %d",&i[2], &i[4]);`
   b. `fscanf(in,"%d %d",&i[2], &i[4]);`
   c. `fscanf(in,"%d %f",&i[2],&f[2]);`
   d. 元素 i[5] 和 f[6] 不存在
   e. `fprintf(in,"%d %d",i[2],i[4]);`
   f. `fprintf(in,"%d %d",i[2],i[4]);`
   g. `fprintf(in,"%d %f",i[2],f[2]);`
   h. 元素 i[5] 和 f[6] 不存在

课程 6.4　多维数组

主题

- 多维数组的概念
- 声明和初始化多维数组
- 嵌套循环和多维数组
- 从多维数组中输入和输出数据

在代数课上学习过利用矩阵计算。例如你可以把以下代数方程：

$$
\begin{aligned}
3x+4y+8z&=15\\
2x-3y+9z&=8\\
4x-7y-6z&=5
\end{aligned}
$$

表示成：

$$
\begin{bmatrix}3 & 4 & 8\\ 2 & -3 & 9\\ 4 & 7 & -6\end{bmatrix}\begin{Bmatrix}x\\ y\\ z\end{Bmatrix}=\begin{Bmatrix}15\\ 8\\ 5\end{Bmatrix}
$$

这里不用 x、y 和 z，而使用 x_0、x_1、x_2，只进行了简单的修改，下面这种表达式是非常有用的。

$$
\begin{bmatrix}3 & 4 & 8\\ 2 & -3 & 9\\ 4 & 7 & -6\end{bmatrix}\begin{Bmatrix}x_0\\ x_1\\ x_2\end{Bmatrix}=\begin{Bmatrix}15\\ 8\\ 5\end{Bmatrix}
$$

这种表达方式有两个优点：

1）让矩阵变大就可以包含更多的变量，甚至到 x_{100} 或 x_{1000}。

2）通过下标来存取变量，使得我们可以通过数组来保存这些变量的值。

我们写一个计算机程序来解这个特定的方程。但是这个程序不会非常的有用，因为这个特定值的方程不会经常出现。一个更好的方法是写一个程序可以解决这种类型的方程，这类方程把系数值包含在一个矩阵中，并把等号右边的值包含在一个向量中。为了完成这个任务，我们的程序应该把矩阵和向量当成变量，因为在解不同的方程时它们是变化的。传统的命名系数矩阵和向量的方法如下：

$$
\begin{bmatrix}a_{00} & a_{01} & a_{02}\\ a_{10} & a_{11} & a_{12}\\ a_{20} & a_{21} & a_{22}\end{bmatrix}\begin{Bmatrix}x_0\\ x_1\\ x_2\end{Bmatrix}=\begin{Bmatrix}b_0\\ b_1\\ b_2\end{Bmatrix}
$$

如果将第一行命名为 row 0，第一列命名为 column 0，我们看到每一个元素双下标为 $a_{row,\ column}$（注意，有时第一行命名为 row 1，第一列命名为 column 1）。

在本课应用的例子中，当我们解决矩阵的问题时，不会涉及矩阵的过多细节。目前只是让你学会处理一个系数矩阵时双下标的用法，就像使用一位数组时使用单下标，使用二维数组时使用双下标。换句话说，变量 a_{02} 在 C 语言中被表示为 a[0][2]。在 C 语言中的这种表示方法使得我们书写矩阵代数运算变得非常容易。

C 允许使用更高维数的数组。考虑我们要收集某个机场的日降雨量，从 2012 年开始收集，一直持续 10 年（2012 ~ 2022）。当每一天结束时，我们从降雨表中测得降雨量（cm）。第一年，我们对数据进行计算，获得日平均降雨量、月最大降雨量，以及其他感兴趣的信息。

为此，一维数组 rainfall[ ] 并不是特别方便。如果使用三维数组 rainfall[ ][ ][ ] 会更加方便。我们可以用年作为第一维，月作为第二维，天作为第三维。这样，rainfall[6][11][23] 代表的就是 2018 年 11 月 23 号的降雨量，以下循环将计算 2018 年 11 月中的降雨量。

每月的总量初始化为0

```
monthly_rainfall = 0.0;
for (i=1; i<=30; i++)
{
        monthly_rainfall += rainfall[6][11][i];
}
```

循环覆盖某个月的所有天

2018年11月中每一天的降雨量累加

你目前不需要理解这个循环的细节，但是我们希望你能意识到使用多维数组在书写程序时是非常便利的。

建立起多于二维数组的视觉化概念是很困难的。很多时候也不值得去开发出一个对应的物理图像。例如对于数组 rainfall[ ][ ][ ]，没必要去生成一个三维数组的映像，因为按照年/月/日来区分是很正常的。事实上我们可以通过给出 rainfall[year][month][day][hour] 把数据拆分成四维数组，甚至于加上 minute 拆分成五维数组。所有这些都是你不需要建立概念映像的自然的分割。换句话说，你不需要建立一个五维数组的概念映像。当你建立一个多维数组时，你的分割方式是自然的。如果你的分割方式很自然，你会发现可以直接处理。

总结来说，基于工程和科学在聚合信息时所用的方法的相似性，我们发现使用多维数组保存和处理数据是很方便和有用的。

从本课的程序中你可以观察到以下事实：

1）数组 b[ ][ ] 被声明为二维数组，每个元素为一个整型数。

2）b[ ][ ] 数组的元素在声明时初始化。

3）第一个 for 循环是一个嵌套的 for 循环，用来打印 b[ ][ ] 中的所有的元素。

4）在输出中，b[ ][ ] 被显示成矩阵的样式。printf 语句中，下标的顺序是行、列格式。一行中的所有数据被输出，生成一个新行，然后下一行数据被输出。

5）数组 rainfall[ ][ ][ ] 被声明为三维数组。

6）rainfall[ ][ ][ ] 中 month 索引被声明为 13 而不是 12。

7）rainfall[ ][ ][ ] 中 day 索引被声明为 32 而不是 31。

8）31 天（一个月的所有天数）的降雨数据在第二个 for 循环中被读入。

9）在第三个 for 循环中，31 天的降雨量被输出。if 语句使得输出呈现日历的格式，也就是每 7 天生成一个新行。

在阅读解释环节前试图回答以下问题：为什么 rainfall[ ][ ][ ] 的大小对应有 13 个月和 32 天。

源代码

b[][] 和 rainfall[][][] 都是多维数组。b[][] 是两维数组，因为它有两对括号。rainfall[][][] 是三维数组，因为它有三对括号。b[][] 有 2×3=6 个元素。rainfall[][][] 有 10×13×32=4160 个元素

```
#include <stdio.h>
void main(void)
{
        int i, j, year, month, day;
        int b[2][3]={51,52,53,54,55,56};
        int rainfall[10][13][32];
        FILE *infile;
```

```
infile = fopen ("L6_4.DAT","r");
for (i=0;i<2;i++)
{
        for (j=0;j<3;j++)
        {
                printf("b[%ld][%ld]= %5d   ", i,j,b[i][j]);
        }
        printf("\n");
}

fscanf (infile,"%d %d",&year,&month) ;
for (day=1; day<=31; day++)
{
        fscanf(infile, "%d",&rainfall[year][month][day]);
}

printf ("Rainfall for Year = %d, Month = %d\n\n",year+2012, month);
for (day=1; day<=31; day++)
{
        printf("%d ",rainfall[year][month][day]);
        if (day==7 || day==14 || day==21 || day==28) printf("\n");
}
```

循环控制变量的最大值对应于声明的数组的大小，b[2][3]

这个printf语句只是输出一个换行。它使得输出看起来更像一个手写的二维的数组，它被放置在两个嵌套的循环中

处理多维数组时经常使用嵌套循环，这里b[][]的元素被输出

循环读每个月的降雨量，循环做至31天

循环输出一个月的降雨量，循环至31天

数组输出必须看起来比较整洁，这个if语句当天数变量均增加7后生成一个新行，以便使得输出看起来像一个日历

输入文件 L6_4.DAT

```
4 12
0 2 0 29  0  1 2
3 0 7 22 11 12 6
0 3 4  2  8  7 5
7 6 0  4  9  7 8
1 9 8
```

输出

```
b[0][0]= 51          b[0][1]= 52          b[0][2]= 53
b[1][0]= 54          b[1][1]= 55          b[1][2]= 56

Rainfall for Year = 2016, Month = 12
0 2 0 29  0  1 2
3 0 7 22 11 12 6
0 3 4  2  8  7 5
7 6 0  4  9  7 8
1 9 8
```

解释

1）什么是多维数组？像一维数组一样，多维数组是保存在一块连续递增的内存区域的相同数据类型的数据的一个集合。但是为了方便和清晰，并不把多维数组看作一个长列数据，而是看成一个表格或矩阵（如图6-3所示的二维矩阵）或一个更复杂的图像。（把一个三维数组想象成一个块。虽然这些图像在处理多维数组时是很好的视觉工具，C语言实际上把

值保存在一块连续的内存单元内，所以把它想象成一个长列表的图像也是合理的。本文中经常与二维数组打交道，通常用矩阵的图形来描述它。）

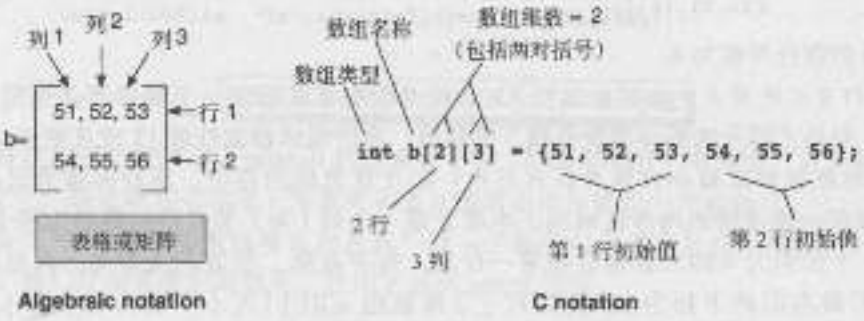


图 6-3 本课程序中两维数组 b[][] 的概念图像

2）如何声明一个多维数组？与声明一维数组类似，例如，

```
int b[2][3];
```

声明了：数组的名字是 b，数组元素的类型是 int，维数是二（它有两对方括号），元素的个数或数组大小是 2×3 = 6（方括号中数字的乘积）。

3）可以在声明多维数组时初始化它吗？可以，在 C 语言中，与一维数组类似，多维数组可以在声明语句中直接初始化。对于一个二维数组，数组元素按行初始化，例如，有 6 个元素的数组 b[][] 在下面的声明语句中初始化

```
int b[2][3]={51,52,53,54,55,56};
```

这样每个元素分别初始化为：

```
b[0][0]= 51   b[0][1]= 52   b[0][2]= 53
b[1][0]= 54   b[1][1]= 55   b[1][2]= 56
```

再次注意数组中下标开始于零，同时在这个列中，最右面的下标最先开始增加。

4）有别的方法可以让我们在声明数组时进行初始化吗？可以用括号来分隔二维数组中的行。本课的程序中并没有演示这种声明，但是可以声明一个二维数组 c[][]，例如，这种声明和初始化把每一行包含在一个括号中。

```
int c[4][3]={{1,  2, 3},
             {4,  5, 6},
             {7,  8, 9},
             {10,11,12}};
```

这种显式的方法使得每个元素的下标变得非常清晰。另外如果在某行我们漏掉一些值，会隐式地把它们初始化为零。例如

```
int c[4][3]={{1,  2},
             {4,  5, 6},
             {7},
             {10,11,12}};
```

把 c[0][2]、c[2][1] 和 c[2][2] 初始化为 0。

我们也可以隐式地声明最左面一维的大小（二维数组中行的数目）。例如声明

```
int c[ ][3]={{1,  2, 3},
             {4,  5, 6},
             {7,  8, 9},
             {10,11,12}};
```

隐式地把行的数目声明为4。

5）我们可以使用多少维的数组？ANSI C并没有显式指定一个编译器必须能支持多少维的数组；但是大部分的编译器能处理7维数组，有的编译器能处理12维或更多。

6）多维数组的数据如何保存在内存中？在计算机的内存中，所有多维数组的数据被连续地保存在一块递增的内存区域内，本质上是一个列（为了节省空间显示为两个列）。在C语言中，下标都为0的元素保存在第一位置。保存在第二位置的元素为，其他所有的下标都为零而最右面的下标为1。例如对于3维数组a[][][]（大小声明为[2][3][4]，一共有2×3×4=24个元素），数组元素保存顺序如下：

可以看到，最右边的下标最先增长，下标增长的顺序为从右到左。

7）如何判断一个多维数组占用了多少内存？多维数组占用内存的数量是由它所有维数的大小的乘积来决定的。例如，一个int数组a[2][3][4]的尺寸是2×3×4 = 24。它包含24个元素，所以占用24×2= 48字节的内存。（每个整型数占用2个字节。）

8）如何使用循环输出多维数组中的所有元素？通常使用嵌套循环来输出多维数组中的所有元素。对于二维数组使用两个嵌套的循环，每个变量控制一个下标。例如，对于本课程序中数组b[2][3]，下面的嵌套循环按矩阵的方式输出所有元素。

```
for (i=0;i<2;i++)
{
        for (j=0;j<3;j++)
        {
                printf("b[%1d][%1d]= %5d   ", i,j,b[i][j]);
        }
        printf("\n");
}
```

注意最外层的循环覆盖行的数目，最内层的循环覆盖列的数目。内层循环（用j作为控制变量）输出一行中的所有列，元素以行的方式显示在输出上（见本课的输出）。

9）在本课程序中，为什么选择rainfall的大小为rainfall[10][13][32]而不是rainfall[10][12][31]？由于只有12个月，并且每个月最多有31天，我们把数组设置得比实际需要的大。这么做是因为C语言中下标从0开始。

10）在本课程序中，如何从文件中读入数据到rainfall？我们开始读文件的第一行，其中包含对应的年和月的数据。

```
fscanf (infile,"%d %d",&year,&month);
```

然后用读入的年和月的值在 for 循环中填充数组的部分值：

```
for (day=1; day<=31; day++)
{
        fscanf(infile, "%d",&rainfall[year][month][day]);
}
```

在某月中循环每一天

用先前读入的年和月的值，读入每一天的降雨量

注意在执行这个循环时，rainfall 的前两个下标（代表年和月）不改变。只是最后一个下标（天）改变。用这种方式填充了为某年和某月预留出来的数组中天的部分。

11）如何把 rainfall 的数组输出到屏幕？以一个好的包含年和月信息的题头开始。为了简单起见，我们并没有使它很漂亮，使用下面的 printf 语句：

```
printf("Rainfall forYear=%d,Month=%d\n\n",year+2012,month);
```

然后使用下面的循环把每天的降雨量输出到屏幕。

在某个月的每一天循环

打印某天的降雨量

```
for (day=1; day<=31; day++)
{
        printf("%d ",rainfall[year][month][day]);
        if (day==7 || day==14 || day==21 || day==28) printf("\n");
}
```

如果是一周的最后一天，输出一个新行

12）在用 fscanf 语句读入数据的 for 循环中，为什么不需要一个 if 语句？fscanf 函数找寻下一个非空白字符时会自动跳转到下一行。这样我们不需要语句来使得 fscanf 读取下一行的内容。

概念回顾

1）多维数组中，每一维对应于一个索引。例如，calendar[12][31] 定义了一个两维数组，calendar 有 12 个月（索引从 0 到 11），每个月有最多 31 项（索引从 0 到 30）。

2）在计算机内存中，数组元素按照索引递增的顺序连续存储。例如 calendar[0][0], calendar[0][1],calendar[0][2],…,calendar[0][30],calendar[1][0],calendar[0][1],…, calendar[1][30], calendar[2][0],…, calendar[11][30]。

3）用嵌套循环存取多维数组中的元素。

练习

1. 基于以下声明：

```
int a[3][1]={1,2,3}, b[3], c[3][2], d[2][3];
```

判断下列语句真假：

a. 数组 c[3][2] 包含 3*2 共五个元素

b. 一维数组 b[12] 可以用来保存二维数组 a[3][4] 中的所有数据

c. 数组 d[2][3] 包含 6 个元素：d[0], d[1],…,d[5]

d. 数组 c[3][2] 包含 6 个元素：c[1][0],c[1][1],c[1][2], c[0][0],c[0][1],c[0][2]

e. 数组 c[3][2] 包含三个一维数组，d[2][3] 包含两个一维数组

2. 判断下面语句是否有错误，如果有，请指出。

```
a.int a[2][0];
b.float a23b[99][77], lxy[66][77];
c.double city[36][34], town(12)(34);
d.int a(2,3)={11,22,33,44};
```

3. 用嵌套 for 循环输出数组 a[3][2] 中的元素，并用嵌套 while 循环输出 b[2][3] 中的元素，以矩阵的方式输出。

```
int a[3][2]={11,22,33,44,55,66},
    b[2][3]={111,222,333,444,555,666};
```

4. 数组 a[3][4][2] 用下面的语句初始化：

```
int a[3][4][2] = {1,2,3,4,5,6,7,8,9,10,11,12,
                  13,14,15,16,17,18,19,20,21,22,23,24};
```

a[1][1][1]、a[2][1][1] 和 a[2][2][1] 的值分别为多少？

5. 数组 x[2][3][4][5] 用下面的语句初始化：

```
int x[2][3][4][5] = {1,2,3, ... through 120};
```

x[1][2][3][4]、x[0][1][3][1] 和 x[1][0][0][4] 的值分别为多少？

答案

1. a. 假　b. 真　c. 假　d. 假　e. 真

2. a. 最小的下标数是 1 不是 0。这个数组代表数组中行的大小。

```
b.float a23b[99][77], xy1[66][77];
b.long city[36][34], town[12][34];
d.int a[2][3]={11,22,33,44};
```

课程 6.5　函数和数组

主题

- 把独立的数组元素传入函数
- 把整个数组传入函数
- 在函数中存取数组

开发模块化设计的程序时，需要写能够传递和接受数组的函数。现在回忆一下，当传递单个的值时，我们有两个选择：可以把一个变量的值的拷贝传递或者把一个变量的地址传递。如果传递一个值的拷贝，我们只能改变传递的拷贝的值，而不能改变元素的变量的值。如果传递一个地址，我们可以改变元素变量的值。本课程序中，我们使用三个函数 function1、function2 和 function3 分别演示了数组元素的地址和值是如何被拷贝的。

从本课程序中你可以观察到以下事实：

1）funciton1 原型中第一个参数是指针变量，在调用 funciton1 时，第一个参数是数组元素的地址。

2）funciton2 原型中第一个参数有方括号，代表一个数组。在调用 funciton2 时，第一个参数是数组名，没有方括号。

3）funciton3 原型中第一个参数在 double　b[ ] 有关键词 const。在调用 funciton3 时，第一个参数是数组名，没有方括号。

4）tenciton1 原型中第二个参数是普通整型变量，在调用 funciton1 时，第二个参数是

数组元素。

5）function2 和 function3 原型中第二个参数是普通整型变量，代表第一个参数代表的数组中包含多少个元素。

6）main 中的 for 循环打印 c[ ] 数组中所有元素。

7）function1 函数体中，变量 *d 修改了传入函数的第一个参数。

8）function2 函数体中，for 循环修改了数组 b[ ]。循环覆盖了数组中的所有元素。同时，这个行为修改了在 main 中的 c[ ]。

9）function3 函数体中，for 循环把数组 b[ ] 所有元素求和。这个行为没有修改在 main 中的 c[ ]。

在阅读解释环节前，尝试回答以下两个问题：

1）函数 function3 原型中的关键词 const 是什么含义？

2）如何把整个数组的存取传入函数？

源代码

指针变量保存一个地址，这代表着调用函数时，第一个参数应该是一个地址

```
#include <stdio.h>
void function1 (int *d, int e);
void function2 (double b[ ],int num_elem);
double function3 (const double b[ ], int num_elem);
```

const 代表，即使 function3 接受了一维数组第一个元素的地址，它也不能通过地址来修改数组中的内容

原型中变量有方括号，这意味着调用时第一个参数应该是一个地址

```
void main(void)
{
        int i, a[10]={0,1,2,3,4,5,6,7,8,9};
        double x, c[5]={2.,4.,6.,8.,10};

        function1(&a[5], a[0]);

        function2(c,5);

        x = function3(c,5);

        printf("\na[5]=%d\n",a[5]);
        printf("c[ ]=");
        for (i=0; i<5 ; i++) printf("%.1lf",c[i]);
        printf("\nx = %.1lf",x);
}
```

这个参数代表a[ ]数组中的6个元素的地址

这些参数中，没有括号的c代表c[ ]数组中第一个元素的地址

在这个函数中，利用从指针符号接受到的地址（代表使用了单目运算符 *）

```
void function1(int *d, int e)
{
        *d = 100+e;
}
```

在这个函数中，使用数组符号（方括号代表）来修改通过地址传递过来的数组中的元素

```
void function2 (double b[ ], int num_elem)
{
        int i;
        for (i=0; i<num_elem; i++) b[i]*=10.;
}
```

这个循环修改通过地址传递过来的数组中的所有元素

对一维数组来说，在函数原型和声明中括号内应该为空

```
double function3 (const double b[ ], int num_elem)
```

```
{
        int i;
        double sum;
        sum=0.0;
        for (i=0; i<num elem; i++) sum += b[i];
        return (sum);
}
```

这个循环将数组中元素累加，并不修改数组中元素

这个函数中，虽使用数组符号，但是，const修饰符意味着不能修改通过地址传递过来的数组中的元素

输出

```
a[5]=108
c[ ]=20.0 40.0 60.0 80.0 100.0
x = 300.0
```

解释

1）如何写一个函数原型和函数调用来传递一个单独的数组元素给函数？像处理单个变量那样处理单个的数组中的元素。如果要改变单个的数组中的元素的值，用“取址”运算符放在函数调用中的数组元素的前面。如果想传递一个不想改变它的值的元素，只需将数组元素放到参数列表中，例如本课程序中调用：

```
function1(&a[5], a[8]);
```

传入了 a[5] 和 a[8]。因为这个调用使得 a[5] 的地址被拷贝到了 function1 的内存区域，所以 a[5] 可以被 function1 修改。相反 a[8] 的值（而不是地址）被拷贝到了 function1 的内存区域，这样 function1 可以利用 a[8] 的值进行计算，它不能改变保存在 a[8] 内存位置上的内容。

对应的函数原型就是传递一个单独变量的地址，另外一个传递一个单独变量的值。当接受一个地址的时候，必须使用一个指针变量（在声明中用 * 代表）。当接受一个值时，我们就用一个简单变量。本课程序中，function1 的原型如下：

```
void function1 (int *d, int e);
```

因此，我们把 a[5] 的地址传给指针变量 d，并且把 a[8] 的值传给变量 e。

在函数体内，我们在赋值语句中使用这两个变量。

```
*d = 100+e;
```

当从函数中返回时，a[5] 的值是 100 加上 a[8] 的值，也就是 108。a[8] 的值保持不变。

2）如何写一个函数原型和函数调用来传递整个的数组变量给函数？可以逐个元素传递，但是在实际上并不这么做。一个简单的替代方法是把数组中第一个元素的地址传递进去。有了第一个元素的地址（和数组类型——例如 int 或者 double，来指明每个元素所占内存的字节长度），C 可以在内部计算任何数组中元素的地址。但是我们不用取值运算符来获得地址。这是因为在 C 语言中，数组中的第一个元素的地址可以用不接方括号的数组的名字来表示。换句话说，对于本课程序中的数组 c[]，下面两个表示等价：

```
&c[0]        c
```

两个表达式都代表数组 c[] 中的第一个元素的地址

所以，在函数调用中，

```
function2 (c,5);
```

把数组中第一个元素的地址传入（还有一个常数 5）function2（如图 6-4）。这使得 function2 可以修改数组 c[]。

函数的原型必须指明它接受一个地址。直到现在，我们在声明中使用 * 来代表要接受一个地址。但是 C 语言允许使用一种对于数值数组更通用的方法，那就是使用方括号。例如对于 function2，下面两个声明是等价的：

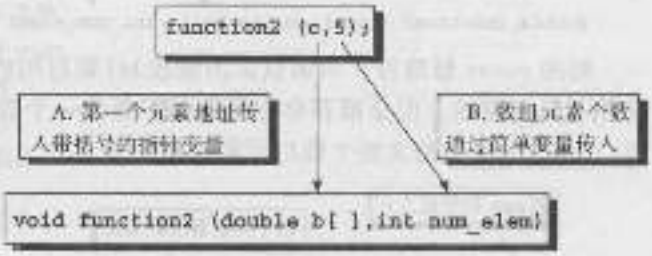


图 6-4 使函数可以访问所有数组元素

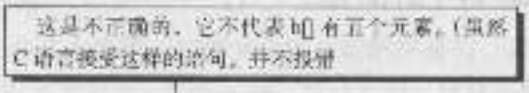
```
void function2 (double *b, int num_elem);
void function2 (double b[ ],int num_elem);
```

观察到两个声明中唯一的不同在于第一个使用 *b，而第二个使用 b[]。在本文中，对于数值型数组使用第二种方法，因为它清晰地表明了我们在使用数组。如果你在原型中用 * 而不用 []，在函数体中你依然可以使用括号来进行数组的管理。换句话说，两种声明的函数体可以如本课程序演示的那样一致。本书的后面会看到使用指针来存取数组中的任意元素。

3）在一个接受了数组地址的函数体内，如何利用数组？我们像在 main 函数中那样处理数组。我们必须知道函数中包含多少个元素，因为 C 语言并不检查数组越界。一个方法是把数组中元素的个数通过参数列表传入函数，这个数字必须在参数列表中分离表示。例如，

```
void function2 (double b[ ],int num_elem)
```

注意，如果把数组放到与数组名相邻的括号内，这是无意义的。换句话说，不能把 function2 接受 5 个元素这种信息写成下面的格式：




```
void function2 (double b[5])
```

本课程序中，在 function2 函数体内，我们分别利用 b[] 数组的元素，就像上面的例子所示。这个循环把每个元素乘以 10，并把元素保存到它自身。

```
for (i=0; i<num_elem; i++) b[i]*=10.;
```

一旦执行这个函数，我们就修改了原始数组。注意在 main 中输出了这个数组。可以从输出中看到数组确实被修改成了原先数值的 10 倍。通过传递数组 c[] 的第一个元素的地址，我们给 function2 修改数组 c[] 的能力。

4）想让函数能读取数组的所有元素，但是不能修改数组中元素的值该如何做？通常这种要求会同时存在，这是因为想要利用数组元素的值进行计算，同时又想保持数组的完整性。我们可以在函数声明中使用 const 修饰符来保证数组不被修改。例如本课程序中的

function3，我们有以下的函数声明和调用。

```
function3(c,5);

double function3 (const double b[ ], int num_elem)
```

利用 const 修饰符，当函数试图修改 b[] 数组的内容时，C 语言会报错。虽然这个修饰符并不是必需的，但是推荐你在读取而不修改一个数组元素的函数中使用它。本课的程序中，function3 只是求整个数组元素的和：

```
sum=0.0;
for (i=0; i<num_elem; i++) sum += b[i];
return (sum);
```

把 sum 初始化为 0

每个元素累加到 sum

把结果传回 main

注意我们使用 return 语句把结果传回 main 函数。这样，在 main 函数中，函数调用语句出现在赋值语句的右边。

5）如何向一个函数传递多维数组？回忆一维数组，在函数原型中用一个标识符后接一对方括号来代表这是一个数组。对于二维数组，函数原型中用一个标识符后接两对方括号来表示。但是与一维数组可以有一个空括号不同，我们需要在二维数组的括号中指定第二个下标的尺寸（最大的列数）。通常除了第一对括号，所有的括号都要有对应的尺寸。例如对于一个二维数组和一个四维数组的声明：

```
#define I 10
#define J 5
#define K 6
#define L 3
int a[I][J];
char b[I][J][K][L];
```

调用它们的函数的原型如下：

```
void function1 (. . . int a[ ][J]. . .);
void function2 (. . . char b[ ][J][K][L]. . .);
```

概念回顾

1）一个没有括号的数组的名字代表数组中第一个元素的地址。

2）当一个数组的名字被当成一个参数传递给函数，函数就有能力存取整个数组。

3）为了指明要把数组传递给函数，我们把函数的参数写成一个数组名后接方括号的格式。

4）当在函数参数上使用 const 修饰符，这个变量的值在函数体内是只读的，不能修改。

练习

1. 判断真假：

a. 在一个函数调用中，当用一个后面不接方括号的数组名作为参数，会把整个数组的内容拷贝到函数的内存区域内。

b. 如果我们想传递一个数组元素，需要在函数调用的参数列表中使用 & 符号。

c. 只给函数读取数组元素的能力，而不给它修改数组元素的能力，这在 C 语言中是不可能的。

d. 一个指针变量被用来保存一个变量的地址。
e. 后面不接方括号的数组的名字，代表数组中第一个元素的地址。
f. 如果想把数组中元素的个数传入函数，应该使用参数列表中一个单独的参数。
g. C 语言允许将数组直接作为函数的形式参数。
h. C 语言允许将数组直接作为函数的实际参数。
i. C 语言允许将数组从一个函数中返回到 main。
j. 指针变量可以保存一个数组元素的地址。
k. 可以将数组的地址作为函数的形式参数。
l. 可以将数组的地址作为函数的实际参数。
m. 可以将指针作为函数的形式参数。

2. 下面的表格显示了一堆钢板的长、宽、厚：

长 (ft)	宽 (m)	厚 (in)
12.0	6.3	2.2
13.0	7.4	3.3
14.0	8.5	4.4
15.0	9.6	5.5

a. 用三个一维数组保存长、宽、厚的信息。然后把单位从 ft[⊖] 或 in 转换成 m。计算钢板的重量，然后把结果保存到一个一维数组中。（假设钢板的密度为 7800kg/m³。）
b. 用一个二维数组保存长、宽、高的信息。然后把单位从 ft 或 in 转换成 m。计算钢板的重量，然后把结果保存到一个一维数组中。（假设钢板的密度为 7800kg/m³。）

答案

1. a. 假　b. 假　c. 假　d. 真　e. 真　f. 真　g. 假
h. 假　i. 假　j. 真　k. 真　l. 假　m. 真

课程 6.6　冒泡排序和最大交换排序

主题

- 冒泡排序
- 在一个数组中交换两个元素
- 最大交换排序

数组排序是一种基本操作，它将元素按一定顺序排列，通常是由小到大。

假设数组 b[] 中元素如下：b[0] = 34，b[1] = 23，b[2] = 64，b[3] = 39，b[4] = 84，b[5] = 91，b[6] = 73。如果我们重新整理数组中元素的值如下：b[0] = 23，b[1] = 34，b[2] = 39，b[4] = 73，b[5] = 84，b[6] = 91。这个数组就是有序的。这是因为在数组中随着下标值的增加，数组元素的值也增加。

为了完成排序工作，可以看到需要很多次交换数组的元素。例如我们在元素 b[0] 和元素 b[1] 间交换 23 和 34。写排序算法的目的就是为了提高交换效率。我们并不介绍所有的排序算法，也不过多描述排序算法中会遇到的所有问题。在本课中只介绍最基本的排序技术：冒泡排序和最大交换排序。在第 8 章演示快速排序。

本课中，你需要首先阅读程序。我们将数组 [33,44,11,22] 用两种方法进行排序。首先阅读解释部分，然后参考源代码以理解它是如何操作的。

⊖ 1ft = 0.3048m。

源代码

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

#define START 0
#define END   4
#define SIZE  10

void main(void)
{
     int i, j, k, b[SIZE], c[SIZE];
     int temp, max, wheremax=END-1, min, wheremin=START;
     int a[END]={33,44,11,22};

/*************************************************
**      INITIALISE ARRAYS b AND c
*************************************************/

     for (i=START; i<END ;i++) b[i]=c[i]=a[i];

/*************************************************
**      BUBBLE SORT
*************************************************/
for (i=START;i <END;i++)
{
     for (j=START; j <END-i-1;j++)
     {
          if (b[j] > b[j+1])
          {
               temp=b[j+1];
               b[j+1]=b[j];
               b[j]=temp;
          }
     }
}

/*************************************************
**    EXCHANGE MAXIMUM SORT
*************************************************/
 for (i=END-1;i>=START;i--)
 {
     max=c[i];
     for (j=i;j>=START;j--)
     {
          if (max <=c[j])
          {
               max=c[j];
               wheremax=j;
          }
     }
     c[wheremax]=c[i];
     c[i]=max;
 }

/*************************************************
**      PRINT RESULTS
*************************************************/

     for (i=START;i <END;i++)
          printf("i=%1d, a[i]=%2d, b[i]=%2d, c[i]=%2d\n",
          i,a[i],b[i],c[i]);
}
```

要排序的数组的起始和结束位置

要排序的数组的最大尺寸

初始化 a[ ] 数组

本例中，所有的数组和 a[ ] 数组相同

每循环一次是冒泡排序的一轮

第一轮，循环在整个范围内执行（从 START 到 END）。在第二轮，1 个元素已经被正确放置了，所以循环的范围减 1。在第三轮，2 个元素已经被正确放置了，所以循环的范围减 2

下面三个语句交换 b[j] 和 b[j+1]

如果小下标的元素比相邻的大下标的元素大，那么就交换

每循环一次是最大交换排序的一轮

在开始的一轮，最大的值被放在数组中最大的下标处位置

第一轮，循环在整个范围内执行（从 START 到 END）。在第二轮，1 个元素已经被正确放置了，所以循环的范围比整个范围小 1 个。在第三轮，2 个元素已经被正确放置了，所以循环的范围比整个范围小 2 个

如果 max 小于或等于我们正在考察的元素，那么我们有一个新的 max，然后保存这个 max 值的下标位置

在一轮的结束，把还没有设置为最高下标的元素和 max 进行交换

输出

```
i=0,  a[i]=33,  b[i]=11,  c[i]=11
i=1,  a[i]=44,  b[i]=22,  c[i]=22
i=2,  a[i]=11,  b[i]=33,  c[i]=33
i=3,  a[i]=22,  b[i]=44,  c[i]=44
```

解释

1）冒泡排序后面的概念是什么？用一个例子来解释这个概念。假设想将一个一维数组 b[4] = {33,44,11,22} 排序成升序（如图 6-5）。首先比较 b[0] 和 b[1]。如果 b[0]>b[1]，我们就交换 b[0] 和 b[1] 的值；否则不重新整理。在 b[1] 和 b[2] 上执行同样的操作。如果 b[1]>b[2]，我们就交换 b[1] 和 b[2] 的值。我们一直持续这个过程，直到到达最后一对 b[2] 和 b[3]。这样就完成了第 1 轮。

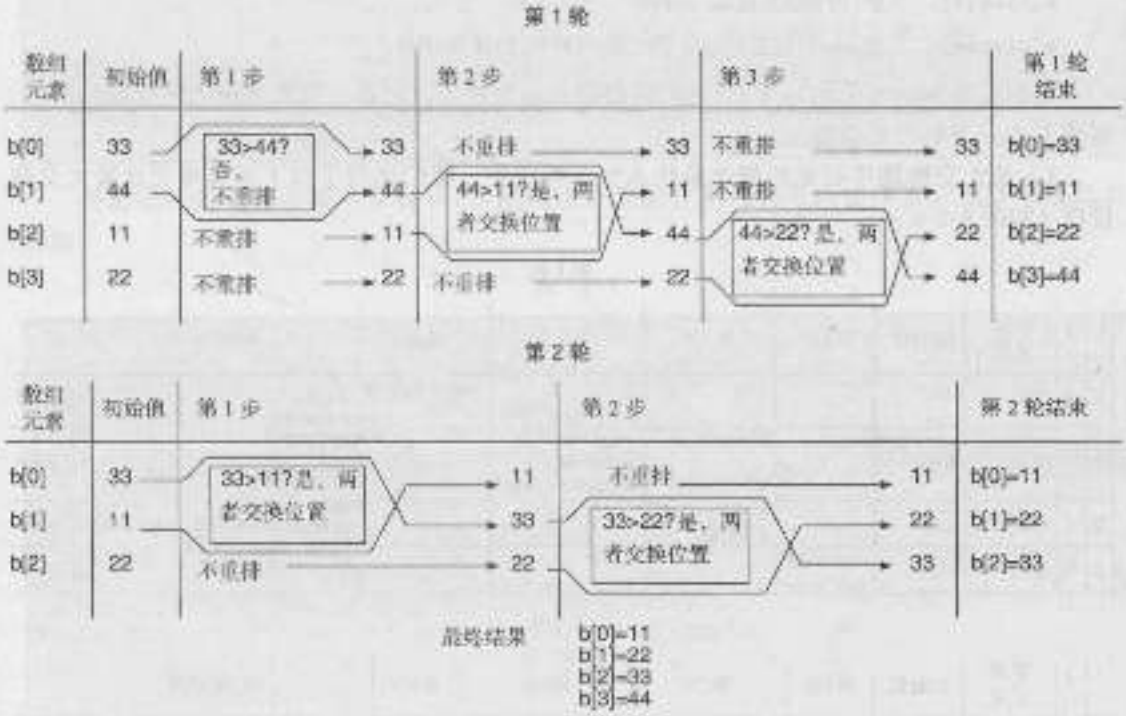


图 6-5 冒泡排序

在第 1 轮中，最大的元素 44，冒泡到最底部（也就是，一步一步地，它移到了数组下标最大的元素位置上）并成为了数组的最后一个元素。然后我们用相同的过程（第二轮）处理头部的剩下的三个元素（因为第四个元素已经有了最大的值）。这使得第二大元素 33，冒泡成数组中的第三个元素。在第 3 轮，第三大元素 22，冒泡成数组的第二个元素。这样我们完成了整个排序过程，因为最后一个元素自动位于数组的第一个位置。通常需要比数组元素个数少一的轮数。

我们可以跟踪循环中计数变量的值，考察源代码中的嵌套循环，通过跟踪循环我们可以得到：

i	j	j+1	顺序	结果
0	0	1	把 b[0] 和 b[1] 中较大的放入 b[1]	
	1	2	把 b[1] 和 b[2] 中较大的放入 b[2]	
	2	3	把 b[2] 和 b[3] 中较大的放入 b[3]	b[3] 是 4 个值中最大的
1	0	1	把 b[0] 和 b[1] 中较大的放入 b[1]	
	1	2	把 b[1] 和 b[2] 中较大的放入 b[2]	b[2] 是 3 个值中最大的
2	0	1	把 b[0] 和 b[1] 中较大的放入 b[1]	b[1] 是 2 个值中最大的
3			不重排	b[0] 自动为最小值

2）在交换两个数组元素的值时做了什么？因为两个操作不能同时发生，必须引入一个临时变量来帮助我们交换两个元素的值。本课程序中，我们使用叫做 temp 的临时变量。例如用以下步骤交换 b[1] 和 b[2] 的值（b[1] = 7, b[2] = 9）。

```
temp=b[1];    把b[1]的值复制temp (temp=7)
b[1]=b[2];    把b[2]的值复制b[1] (b[1]=9)
b[2]=temp;    把temp的值复制b[2]，则交换b[1]和b[2]的值 (b[2]=7)
```

结果就是 b[1] = 9, b[2] = 7。它们的值确实被交换了。记住，在程序中当你想交换值时，需要引入一个临时的变量。

3）最大交换排序后面的概念是什么？本程序中，我们执行了以下步骤来完成最大交换排序（如图 6-6）：

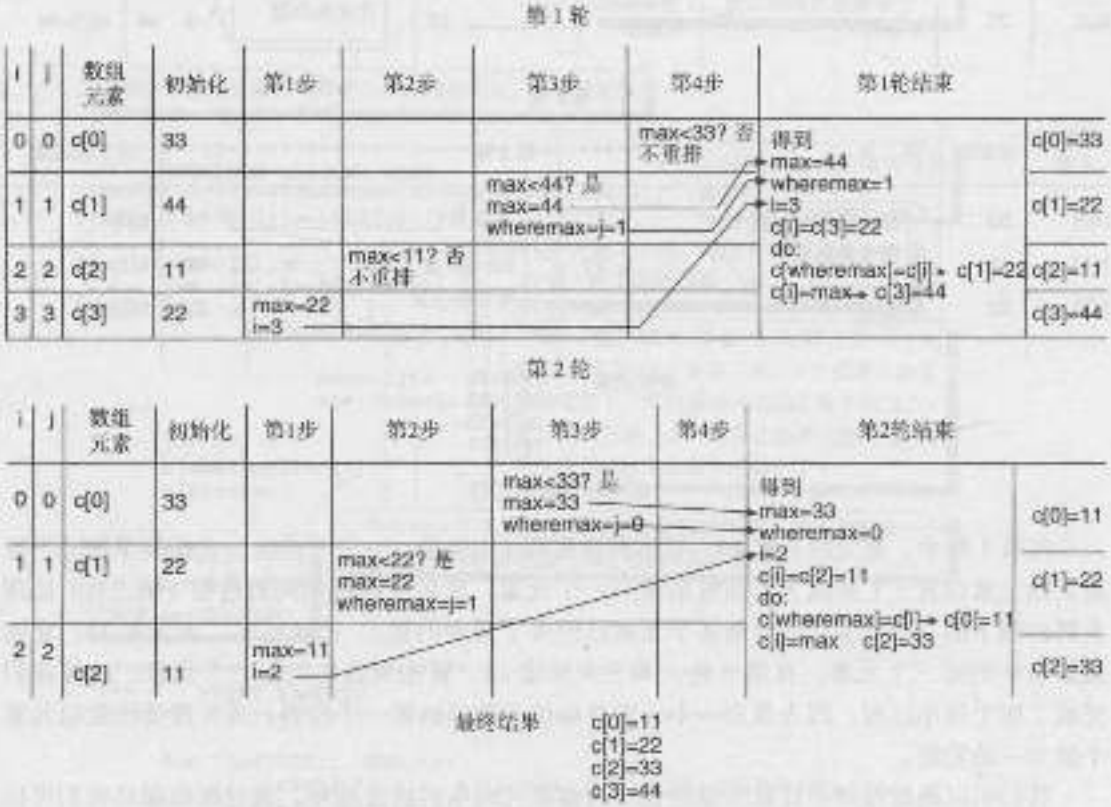


图 6-6 最大交换排序

a. 把最后一个数组元素的值赋给 max。

b. 依次比较 max 和数组中其他元素；如果 max 比元素小，那么用元素的值替换 max，并且用 wheremax 记住元素的具体位置（数组索引）。重复这个过程直到找出最大值。

c. 把发现的最大元素放到元素的最后一个位置，结束第一轮。

d. 因为最大的元素已经被发现并放到最后一个位置，在搜索过程中排除掉最后一个元素。

e. 继续执行 b、c、d 步骤，直到完成升序排列数组。

可以跟踪循环中计数变量的值，考察源代码中的嵌套循环，通过跟踪循环我们可以得到：

i	j	重排	结果
3	3	max = c[3]	对于具体数据，
	2	if c[2] > max, max = c[2], wheremax = 2	max = 44，wheremax = 3，
	1	if c[1] > max, max = c[1], wheremax = 1	44 移入 c[3]
	0	if c[0] > max, max = c[0], wheremax = 0	且 c[3] 的值（22）移入 c[1]
2	2	max = c[2]	对于具体数据，max = 33，
	1	if c[1] > max, max = c[1], wheremax = 1	wheremax = 0，33 移入 c[2]
	0	if c[0] > max, max = c[0], wheremax = 0	且 c[2] 的值（11）移入 c[0]
1	1	if c[1] > max, max = c[1], wheremax = 1	对于具体数据，max = 22，
	0	if c[0] > max, max = c[0], wheremax = 0	wheremax = 1，不需要重排
0		不重排	c[0] 自动为最小值

嵌套循环的具体步骤可见表 6-1。使用这个表配合源代码可以加深你对循环和数组的理解。

表 6-1

初始数组 c[] = {33, 44, 11, 22}																
外循环——交换 c[wheremax] 和 c[i]																
i	c[i]	max (max = c[i])	j	c[j]	max	max < c[j]?	max (max = c[j])	wheremax (wheremax = j)	i	c[i]	wheremax	c[wheremax] = c[i]	max	c[i] = max
3	33	33	3	22	22	是	22	3						
			2	11	22	否	不重排	不重排						
			1	44	22	是	44	1						
			0	33	44	否	不重排	不重排	3	22	1	22 → c[1] = 22	44	44 → c[3] = 44
第 2 轮后的数组 c[] = {33, 22, 11, 44}，（注意，44 和 22 已经交换，44 被正确地放在数组的最后一个元素）																
2	11	11	2	11	11	是	11	2								
			1	22	11	是	22	1								
			0	33	22	是	33	0	2	11	0	11 → c[0] = 11	33	33 → c[2] = 33
第 3 轮后的数组 c[] = {11, 22, 33, 44}，（注意 11 和 33 已经交换；33 被放在正确的位置，为数组的倒数第 2 个元素）																
1	22	22	1	22	22	是	22	1								
			0	11	22	否	不重排	不重排	1	22	1	22 → c[1] = 22	22	22 → c[1] = 22
第 4 轮后的数组 c[] = {11, 22, 33, 44}，（注意，无需交换，因为 22 已处于正确的位置）																
0	11	11	0	11	11	是	11	0	0	11	0	11 → c[0] = 11	11	11 → c[0] = 11
最终数组 c[] = {11, 22, 33, 44}，（注意，无需交换，因为 11 已处于正确的位置）																

概念回顾

1）在冒泡循环的第 n 轮中，第 n 个大的元素会冒泡到正确的排序位置。

2）C 语言中，交换两个元素不可能一步完成。我们需要一个临时变量。

练习

1. 一维数组有以下 10 个元素：

```
4.4  3.3  2.2  5.5  1.1  6.6  7.7  10.0  5.9  8.8
```

用你在本课中学过的三种方法使其成为降序排列。

2. 一个二维数组有 20 个元素：

```
3   33   333   3333
5   55   555   5555
1   11   111   1111
4   44   444   4444
2   22   222   2222
```

用本课学习到的三种方法来完成以下任务：

a. 将数组排序成下面的样子。

```
5   55   555   5555
4   44   444   4444
3   33   333   3333
2   22   222   2222
1   11   111   1111
```

b. 将数组排序成下面的样子。

```
1111   111   11   1
2222   222   22   2
3333   333   33   3
4444   444   44   4
5555   555   55   5
```

应用程序 6.1　将 16 个 1 位加法器组成 1 个 16 位加法器

问题描述

在这个应用程序中，要通过连接 16 个 1 位加法器而生成 1 个 16 位的加法器。通过使用函数和数组，首先生成 1 个 1 位加法器，然后把 16 个这样的 1 位加法器组合起来。我们会要求用户输入两个 16 位二进制数来验证加法器的功能。

解决方法

1. 相关公式

一个完整的一位加法器，用 x,y,c_{in}（进位输入）作为输入，用 sum 和 c_{out}（进位输出）作为输出：

$$sum = x \text{ xor } y \text{ xor } c_{in}$$
$$c_{out} = (x \text{ and } y) \text{ or } (c_{in} \text{ and } (x \text{ xor } y))$$

在 C 语言语法中，“xor”是 ^，“and”是 &，“or”是 |。这样上面的公式可以表示为

```
sum = x ^ y ^ cin
cout = (x & y) | (cin & (x ^ y))
```

2. 算法

为了构建 1 个 16 位加法器，把 16 个 1 位加法器连接起来，把第一个 1 位加法器的输出当成第二个 1 位加法器的输入，把第二个 1 位加法器的输出当成第三个 1 位加法器的输入，

依次类推，算法如下。

主程序：

调用函数从用户读入一个 16 位二进制数 x

调用函数从用户读入一个 16 位二进制数 y

调用 16 位加法器计算 $x+y$

输出结果

函数 readNumber

这个函数从控制台读入一个 16 位二进制数。一个整型变量的大小不足以容纳一个 16 位二进制数。于是我们使用 char 类型的数组来保存输入的二进制数。函数调用和函数声明如下：

```
函数调用 readinput (inputX);
函数声明 void readinput (char in[]);
```

被保存在 in[ ] 的用户输入转换到 inputX[ ]。注意，在函数调用中，因为 inputX[ ] 是一个 main 中的数组，我们只使用数组的名字，而不使用 & 或者跟随名字的 [ ]。

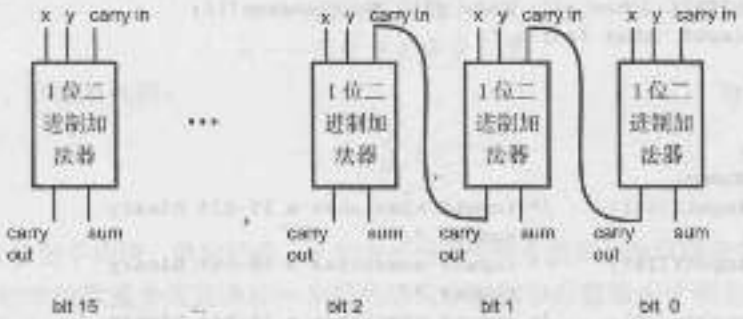
在函数内部，每一个被当成 char 读进的二进制数都被转换成对应的整数值，第一个读进的数是最有意义的位，保存在 in[15]。函数的源代码如下：

```
void readinput (char in[])
{
    int i;
    char bit;

    for (i=15; i>=0; i--)
    {
        scanf ("%c", &bit);
        in[i] = bit - '0';
    }
}
```

函数 adder16bit

这个函数把 16 个 1 位加法器连接起来构建 1 个 16 位加法器。当连接这些加法器时，把第 i 个 1 位加法器的输出当成第 $i+1$ 个 1 位加法器的输入。下图演示了如何进行连接。



函数调用和函数声明如下：

```
函数调用 adder16bit (inputX, inputY, answer);
函数声明 voidadder16bit (charx[], chary[], charanswer[]);
```

inputX 和 inputY 是 main 中的两个数组，保存两个输入的二进制数。两个数的和保存在 main 提供的 answer 数组中。函数体内，cout 代表把进位输出连接到下一位的进位输入。利用上面的设定，函数的源代码如下：

```
void adder16bit (char x[], char y[], char answer[])
{
    int i;
    char cin, cout, sum;

    cin = 0;
    for (i=0; i<16; i++)
    {
        adder1bit (x[i], y[i], cin, &sum, &cout);
        cin = cout;     // routing cout from bit i to cin in bit i+1
        answer[i] = sum;
    }
    answer[16] = cout;     // final carry out stores as the MSB
}
```

函数 adder1bit

这个函数执行一个完整的 1 位加法器。函数调用和函数声明如下：

```
函数调用  adder1bit (x[i], y[i], cin, &sum, &cout);
函数声明  void adder1bit (char x, char y, char carryIn,
          char *sum, char *carryOut);
```

在函数的参数列表中，x[i]、y[i] 和 cin 的值被传入函数进行计算，结果 sum 和 carryOut 传递给调用者。利用以上设定，这个函数的源代码如下：

```
void adder1bit (char x, char y, char carryIn, char *sum,
char *carryOut)
{
    *sum = (x ^ y) ^ carryIn;
    *carryOut = x & y | carryIn & (x ^ y);
}
```

利用以上算法，main 函数的源代码如下：

```
#include <stdio.h>

void adder1bit (char x, char y, char carryIn, char *sum,
                char *carryOut);
void adder16bit (char x[], char y[], char answer[]);
void readinput (char in[]);

void main()
{
    int i;
    char dummy;
    char inputX[16];     /* inputX simulates a 16-bit binary
                            number */
    char inputY[16];     /* inputY simulates a 16-bit binary
                            number */
    char answer[17];     /* inputY simulates a 16-bit binary
                            number plus carry out as the MSB
                            */
    /* MSB = index 15, LSB = index 0 */

    printf ("Enter 2 16-bit binary numbers: \n");
    readinput (inputX);

    scanf ("%c", &dummy);     /* dummy to read the space */
```

```
    readinput (inputY);
    adder16bit(inputX, inputY, answer);
    printf ("The sum is: ");
    for (i=16; i>=0; i--)
            printf ("%d", answer[i]);
    printf ("\n");
}

/* functions here */
```

我们用这个程序来计算 12345+6789 如下：

```
Enter 2 16-bit binary numbers:
0011000000111001
0001101010000101
The sum is: 00100010101011110
```

注释

注意一个空的字符用来在两个二进制数之间跳过一个空格。当我们用字符按位读入一个数时，这是必需的。

应用程序 6.2　浪高的平均值和中位数（数值方法例子）

问题描述

一个研究要评估特定海滩腐蚀特别快的原因，为此测量海浪的高度。为了计算沙子的移动，必须知道海浪的平均高度。有两种平均的方法可以采用：平均值和中位数。写一个程序分别计算海浪高度的平均值和中位数。从一个文件中读入海浪的高度，并把结果输出到屏幕。

解决方法

1. 相关公式

我们定义：

$$x_i = \text{一列数中的第 } i \text{ 个值}$$

$$n = \text{一列数中有多少个值}$$

基于以上定义，计算平均值：

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

其中$\bar{x}$等于 n 个值的平均值。换句话说，平均值是列表中所有值的和除以列表中值的个数。

一个集合的中位数通常描述为在一个列表中有相同数目的数值大于和小于这个数值。例如有 5 个值 10、13、24、9 和 1，中位数就是 10。因为有两个数（13，24）大于它，同时有两个数（1，9）小于它。

这个定义并不是非常准确，因为我们没有考虑具有相等数值的情况。例如，如果有 5 个值 9，10，10，13 和 24，那么中位数为 10。这里，可以看出小于或等于中位数的数值的个数，以及大于或等于中位数的数值的个数都大于所有数的个数的一半。本例中可以看出小于或等于中位数的数值的个数为 3（9，10，10）。大于或等于中位数的数值的个数为 4（10，10，

13，14）。因为无论是3还是4都大于所有数个数的一半（5/2=2.5），所以10是中位数。

我们可以用公式形式写出下面的条件：

$$n_{lower}=\text{小于或等于}\ x_i\ \text{值的个数}$$

$$n_{higher}=\text{大于或等于}\ x_i\ \text{值的个数}$$

如果 n_{lower} 大于 $n/2$ 且 n_{higher} 大于 $n/2$，那么

$$\hat{x}=x_i$$

其中$\hat{x}$为中位数。注意到在本例中，只考虑了列表中有奇数个数值。如果列表中有偶数个数值，中位数的定义就变得不那么清晰。另外一个找出中位数的方法是首先将列表排序，然后取中间那个数值作为中位数。我们在其他的章节中讨论过排序，所以这里不用排序来找中位数。

2. 算法

对于输入来说，执行下列步骤：

1）打开数据文件。

2）从数据文件中读入滚高的数据。

对于平均值：

1）将列表中的所有值加到一起。

2）将和除以列表中值的个数。

对于中位数：

1）将一个值与其他的值比较。

a. 计算有多少个值小于或等于这个值。

b. 计算有多少个值大于或等于这个值。

2）如果1a和1b都大于 $n/2$，那么我们发现了中位数，并停止。

3）如果1a或者1b不大于 $n/2$，我们利用数列中的下一个数重复步骤1。

对于结果，将平均值和中位数打印到屏幕。

3. 源代码

根据算法我们逐步生成求解平均值和中位数的源代码。我们并没有生成读入和输出结果部分的详细源代码，以前已经描述过如何实现它们的技术细节了。这部分代码只是显示在最后的源代码中。

4. 计算平均值

1）把列表中的所有数累加到一起。下面的循环把x[ ]中的所有值加到一起。其中num_pts是x[ ]中值的个数。

```
sum = 0.0;
for (i=0; i<num_pts; i++) sum += x[i];
```

注意到在循环之前使sum=0.0是非常重要的。这使得只有x[ ]中的值被累加到sum中。

2）将sum除以列表中值的个数。下面的代码做这个计算。注意num_pts是一个整型数，而sum和mean是double型。因此要小心混合代数运算。虽然本例中可以使用隐式转换到double的方式（C语言自动完成），但我们选择使用类型转换运算符来将num_pts进行显式的转换，以演示类型转换运算符的用法。当你对这种类型的混合代数运算比较熟悉时，可以忽略它。

```
mean = sum / (double)num_pts;
```

5. 计算中位数

将一个值和其他值比较，并且

a. 计算有多少个值小于或等于这个值。

b. 计算有多少个值大于或等于这个值。

用 x[j] 与其他值比较，下面的循环执行这些操作：

```
count_higher = 0;
count_lower = 0;
for (i=0; i<num_pts; i++)
{
        if (x[j] <= x[i]) count_higher++;

        if (x[j] >= x[i]) count_lower++;

}
```

将计数器初始化为0，必须在循环开始前完成这一步骤

在所有数据点上循环

如果比较的值小于或等于其他值，count_higher 加1

如果比较的值大于或等于其他值，count_lower 加1

我们必须将列表中每一个值作为比较值 x[j]，所以把上面的代码放到一个循环中，并在每个数据点上循环一遍。但是也许我们并不需要循环覆盖所有的数据点，这是因为在前几个数据点的循环中，就已经发现了中位数。

另外一种理解这个问题的方法是意识到循环会一直继续，在 count_higher 或者 count_lower 小于 num_pts/2 的情况下，用一个 do-while 循环给出下面的代码：

```
j=-1;
do
{
        j++;

        PREVIOUS CODE WHICH COUNTS
        THE NUMBER OF LOWER AND
        HIGHER VALUES

}       while (j<num_pts &&
        (count_lower <=((double)(num_pts)/2.) ||
        count_higher <=((double)(num_pts)/2.)));
median = x[j];
```

初始化数组索引变量

在循环中递增数组索引变量

在 count_lower 或 count_higher 小于等于 num_pts/2 时，继续这个循环，否则停止。同时 j 必须小于 num_pts

当 while 循环停止时，比较的值就是中位值

完整的代码（包含上面描述的 do-while 循环）如下：

```
#include <stdio.h>
#define MAX_NUM_PTS 100

void main (void)
{
        int x[MAX_NUM_PTS], num_pts, i, j, count_lower, count_higher, median;
        double sum, mean;
        FILE *infile;

        infile = fopen ("average.dat", "r");
        fscanf (infile, "%d", &num_pts);
        for (i=0; i<num_pts; i++) fscanf (infile, "%d", &x[i]);

/*******************************************************
**  CALCULATION OF THE MEAN
*******************************************************/

        sum = 0.0;
```

```
        for (i=0; i<num_pts; i++) sum += x[i];
        mean = sum/(double)num_pts;
/*********************************************************
**  CALCULATION OF THE MEDIAN
*********************************************************/

        j=-1;
        do
        {
          j++;
          count_lower = 0;
          count_higher = 0;
          for (i=0; i<num_pts; i++)
          {
                  if (x[j] <= x[i]) count_higher++;
                  if (x[j] >= x[i]) count_lower++;
          }
        }         while (j<num_pts && (count_lower <=((double)(num_pts)/2.)
                        ||count_higher <=((double)(num_pts)/2.)));
        median = x[j];

        printf ("The mean of the values is: %.3lf \n"
                "The median value is:      %d  \n", mean, median);
}
```

6. 输入文件 Average.Dat

```
29
67 87 56 34 85 98 56 67 87 90 45 42 31 97 58 78 12 16 22 42 83 95
53 27 49 85 58 79 79
```

7. 输出

```
The mean of the values is: 61.310
The median value is:       58
```

注释

本程序中定义了最大的数据点的个数为100，并且在数据文件的头一行读入了数据点的实际个数。如果我们要分析的数据个数大于100，需要改变这个值。

这里我们要介绍的是如何开发高效的程序。因为要关心如何开发出高效的代码，我们要使得自己的算法更有效率。可以通过衡量在执行算法中发生了多少次的比较来评价一个包含比较操作的算法的效率。确定发生了多少次比较有时候并不是很直接，不同的情况有不同的比较次数。例如，对一个包含 n 个元素的数列求解中位数。如果第一个元素恰好是中位数（巧合），我们只需要 n 次比较（因为只要遍历一遍数列就可以了）。

但是如果最后一个元素恰好是中位数（也是巧合），对于 n 个值，我们分别需要执行 n 次比较，即共需要 n^2 次比较来完成中位数的计算。如果我们有1000个数，那就意味着需要执行 1000^2=100万次比较。可以看到，针对这个特定的问题，比较的次数是非常大的。因此，开发出一个高效的算法是非常有利的。这里不做详细介绍，但是你需要意识到，计算机科学和工程的一部分就是寻找有效的算法。在以后的教育生涯中你会学习更多算法开发方法。

修改练习

1. 用一个带 break 语句的 while 来代替 do-while 循环。
2. 将 x[ ] 作为 double 类型而不是 integer 类型的数组。

3. 修改程序以处理输入文件中的 12 列浪高数据（每一列代表一年中的一个月）。输入数据文件如下：

$n1$
$h_1\ h_2\ h_3 \ldots h_{n1}$
$n2$
$h_1\ h_2\ h_3 \ldots h_{n2}$
.
.
.
$n12$
$h_1\ h_2 \ldots h_{n12}$

4. 去掉类型转换运算符，程序还会正常运行吗？为什么？
5. 为这个程序生成一个模块化设计。构造三个函数：一个用于输入，一个用于计算平均值，一个用于计算中位数。

应用程序 6.3　矩阵－向量乘法（数值方法例子）

问题描述

写一个程序将一个矩阵 a[ ][ ] 和一个向量 x[ ] 相乘，将结果放到向量 b[ ] 中。从一个包含矩阵和向量格式数据的数据文件中读入数据。例如

$$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & x_1 \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & x_2 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & x_3 \\ & & & & & & & x_4 \\ & & & & & & & x_5 \\ & & & & & & & x_6 \\ & & & & & & & x_7 \end{matrix}$$

矩阵中每个元素的下标为 $a_{row\ column}$。输出结果到屏幕。写一个程序来处理行的数比列的数少这种情况。

解决方法

首先，给出相关公式，这里我们简单回顾矩阵和向量的乘法。对于给定的矩阵和向量，结果向量中的各个分量是：

$$\begin{aligned} b_1 &= a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + a_{15}x_5 + a_{16}x_6 + a_{17}x_7 \\ b_2 &= a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + a_{25}x_5 + a_{26}x_6 + a_{27}x_7 \\ b_3 &= a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5 + a_{36}x_6 + a_{37}x_7 \end{aligned}$$

我们选择以和的方式写出这些公式，是因为在第 4 章发现，一旦以这种方式写出，公式可以很方便地转换为一个循环。

通常，如果矩阵有 n 列，那么 b_i 可以表示为：

$$b_i = \sum_{j=1}^{n} a_{ij} x_j$$

如果矩阵有 m 行，那么 b 中会有 m 个分量。这样我们计算 b_i（i 从 1 到 m）。

1. 算法

算法直接描述如下：

1）读入一个矩阵的行和列的个数。

2）读入矩阵和向量。

3）利用公式 6.2 计算 b 向量的一个分量。

4）重复步骤 3 计算 b 向量的所有分量。

5）输出 b 向量到屏幕。

2. 源代码

读入输入数据 如何读入输入数据是值得讨论的，因为所用的循环并不简单。输入数据采用将要相乘的矩阵和向量的形式，其中向量中元素的个数等于矩阵中列的个数。例如下面的示例文件，第一行就是矩阵中行和列的数目，其他行就是矩阵和向量。

```
3 8
2 4 6 4 3 6 8 9 4
9 8 7 6 5 8 9 6 3
8 7 6 4 1 0 2 8 6
                3
                4
                6
                0
                3
```

infile 代表输入的数据文件，num_rows 和 num_cols 代表矩阵中行和列的数目。行

```
fscanf(infile,"%d %d",&num_rows,&num_cols);
```

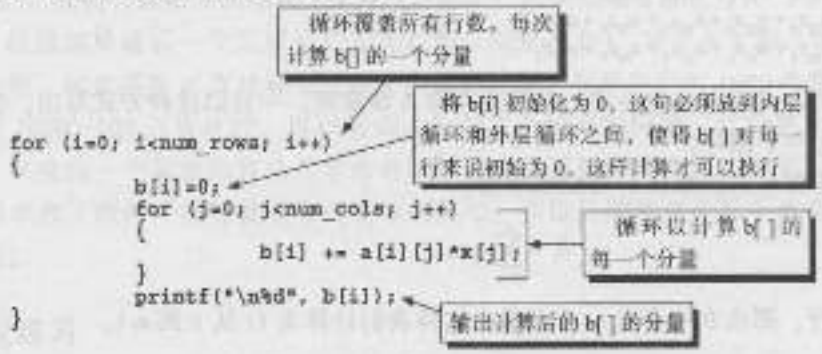
从数据文件中读入第一行。可以用这行读入的 num_rows、num_cols 来建立循环。用下标 i 代表列，用 j 代表行。接下来的嵌套循环从数据文件中读取矩阵和部分向量。（注意，C 语言中下标开始于 0 而不是 1，我们并不严格遵守在应用程序例子开头所显示的计数方法，每项都移动 1。）

利用这段代码，我们将读入数组文件的 2、3、4 行，现在需要读入 5、6、7、8、9 行，这些行只包含一个元素。同理，C 语言中下标开始于 0 而不是 1，我们并不在 num_rows+1 位置开始循环以读入 x[ ]；我们在 num_rows 位置开始。下面的循环读入剩下的 x[ ] 值。

```
for (i=num_rows; i<num_cols; i++) fscanf (infile,"%d",&x[i]);
```

记住 fscanf 在读入数值时会略过空白字符，所以将列变量向右移动很多空格没有意义。

计算 b 向量 我们利用公式 6.2 生成循环计算 b[ ] 中的每个分量，将下面的源代码和公式比较，以理解这个循环是如何生成的。



注意，在循环体内部，我们也把 b[ ] 中的每个分量输出到了屏幕。整个源代码如下：

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

```
#define MAX_NUM_ROWS 20
#define MAX_NUM_COLS 20

void main (void)
{
     int a[MAX_NUM_ROWS][MAX_NUM_COLS], x[MAX_NUM_COLS];
     int b[MAX_NUM_ROWS];
     int i, j, num_rows, num_cols;
     FILE *infile;

     infile=fopen("matvect.dat","r");
     fscanf(infile,"%d %d",&num_rows,&num_cols);

     for (i=0; i<num_rows; i++)
     {
               for (j=0; j<num_cols; j++)
               {
                         fscanf (infile,"%d",&a[i][j]);
               }
               fscanf (infile,"%d",&x[i]);
     }
     for (i=num_rows; i<num_cols; i++) fscanf (infile,"%d",&x[i]);

     printf("\nb vector");

     for (i=0; i<num_rows; i++)
     {
               b[i]=0;
               for (j=0; j<num_cols; j++)
               {
                         b[i] += a[i][j]*x[j];
               }
               printf("\n%d", b[i]);
     }
}
```

3. 输入数据文件

```
4 5
2 4 5 3 6 2
9 8 4 1 4 5
0 9 1 3 9 2
9 8 2 4 1 5
          1
```

4. 输出

```
b vector

55
75
71
83
```

注释

在编程时利用数组，你的大部分精力将用于管理数组分量的下标。而且，采用公式中求和的方式，可以直接写出一个循环以完成这个计算。循环只用于管理求和公式中的下标。这样将公式转换为代码变得非常简单。如果可能，我们推荐你将自己的公式写成求和的方式。

修改练习

1. 将矩阵和向量的数据类型从 double 变为 int。
2. 修改程序处理行数大于列数的情况。为什么现在的程序不能处理这种情况？
3. 为本应用生成模块化设计。生成两个函数：一个函数读入，一个函数计算和输出。

应用程序 6.4 搜索和文件压缩

问题描述

写一个程序压缩文件以生成另外一个新文件。新文件要比原始文件小。检查结果，确保程序可以利用新文件重新生成原始的文件。

解决方法

一个压缩或编码的方法是识别出重复的位，并用重复的位的数值以及重复的位的长度来代替表示。例如行

```
000000000000000111111111111111111110000011111111111111111111111111111111
```

可以被下面的行代替

```
15 20 5 33
```

因为有 15 个 0，20 个 1，5 个 0，33 个 1。我们可以理解它是因为事先知道只有 0 和 1 出现并且 0 首先出现在这一行。新行要比原始的行短很多，需要更少的内存。所以，存储新行要比存储原始行更有效率。为了从新行重新恢复到原始行，需要执行一个相反的操作，例如把编码的行

```
19 23 8 17
```

解码成

```
000000000000000000011111111111111111111111000000001111111111111111
```

1. 算法

下面的算法用来将例子中的原始的行编码成新行：

1）在原始文件的一行中找到与前一个值不同的第一个值。（换句话说，在一些 0 以后找到第一个 1）。

2）计算达到这个值所经历的位数。

3）重新开始计数。

4）重复过程 1 到 3。

这个过程如图 6-7 所示：我们使用的数组为 x[j][i]，其中 j 代表行数，i 代表列数。我们从左边开始直到遇到 1。在此过程中，数组的索引 i 等于最后一个 0 的位置（12），计数变量也等于 12，因为我们经过了 12 个零。

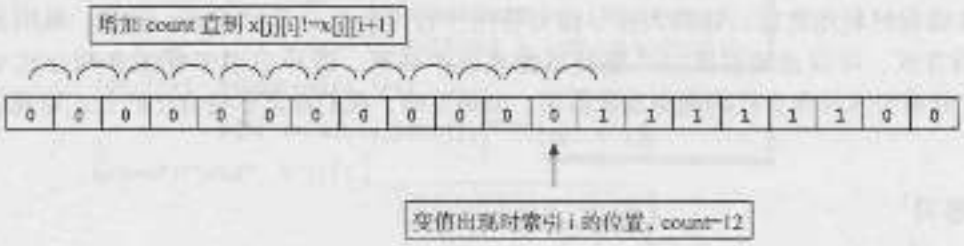


图 6-7 循环搜索一行

为了解码压缩文件，我们遵循以下过程：

1）把第一个 value 设置为 0。

2）读入 count。

3）输出 count 个 value。

4）将 value 从 1 变为 0 或从 0 变为 1。

5）在一行中累加所有 count。

6）重复执行 2 到 5，直到 count = 60（一行中位的个数）。

2. 源代码

```
count = 0;
for (i=0; i<=59; i++)
{
        count++;
        if (x[j][i] != x[j][i+1])
        {
                count = 0;
        }
}
```

初始化计数器

开始循环查找一行中的 60 个值

对每个经过的位计数

如果当前位不等于下一个位

重新开始计数

我们需要加上以下 4 步：

1）搜索每一行。

2）在每一行的末尾计数（i 为 59）。

3）在重置前输出 count 到文件和屏幕。

4）在完成搜索每一行后打印一个空行。

下列代码包含添加的特征：

```
for (j=0; j<=29; j++)
{
        count = 0;
        for (i=0; i<=59; i++)
        {
                count++;
                if (i == 59 || x[j][i] != x[j][i+1])
                {
                        fprintf(outfile,"%d ",count);
                        printf("%d ",count);
                        count = 0;
                }
        }
        fprintf(outfile,"\n");
        printf("\n");
}
```

开始搜索 30 行中的每一行

如果达到行尾，重新开始计数

如果在行尾，或者有 0-1 的改变

把计数输出到文件

把计数输出到屏幕

如果有 0-1 转换，重新开始计数

在文件中开始新的行

在屏幕中开始新的行

我们也需要解码一个压缩文件。如果我们把 value 打印成 a（0 或者是 1），下面的代码执行算法中的步骤。

```
a=0;
sum=0;
do
{
```

把第一个 value 改为 0

初始 count 的和

```
        fscanf(outfile,"%d",&count);
        for (i=1; i<=count; i++) printf ("%1d",a);
        a=!a;
        sum+=count;
}         while (sum<60);
```

读入 count

打印 a 到屏幕 count 次

将 value 从 0 变为 1 或者从 1 变为 0

重复此过程，直到 sum 等于 60

count 求和

我们也加上了一些你已经很熟悉的内容：

1）定义两个宏常量，SIZE1 和 SIZE2 分别为 30 和 60。这些宏在程序中用来代替行长度和行个数。如果我们遇到一个有不同尺寸的 bitmap 类型文件，可以简单地修改这两个常量。

2）在执行任何操作之前，将输入文件打印到屏幕。

3. 修改程序

```
#include<stdio.h>
#define SIZE1 30
#define SIZE2 60
void main (void)
{
    int x[SIZE1][SIZE2];
    int i, j, a, count, sum;
    FILE *infile, *outfile;

    infile = fopen ("input.dat", "r");
    outfile = fopen("output.out", "w");

    /* read input file */
    for (i=0; i<=SIZE1-1; i++)
    {
        for (j=0; j<=SIZE2-1; j++)
        {
            fscanf (infile, "%1d", &x[i][j]);
        }
}
fclose (infile);

/* print input file */
for (i=0; i<=18; i++)
{
    for (j=0; j<=SIZE2-1; j++)
    {
        printf ("%1d", x[i][j]);
    }
    printf ("\n");
}

/* compress file */
count = 0;
printf ("\n\n\n");
for (j=0; j<=SIZE1-1; j++)
{
    for (i=0; i<=SIZE2-1; i++)
    {
        count++;
        if (i==SIZE2-1 || x[j][i]!=x[j][i+1])
        {
```

```
            fprintf (outfile, "%d ", count);
            printf ("%d ", count);
            count = 0;
        }
    }
    fprintf (outfile, "\n");
    printf ("\n");
}
fclose (outfile);

/* expand file */
outfile = fopen ("output.out", "r");
for (j=1; j<=SIZE1; j++)
{
    m = 0;
    sum = 0;
    do
    {
        fscanf (outfile, "%d", &count);
        for (i=1; i<=count; i++)
                printf ("%1d", m);
            m = !m;
            sum += count;
        } while (sum<SIZE2);
        printf ("\n");
    }
    fclose (outfile);
}
```

注释

我们确实生成了一个比原始文件小的压缩文件，同时也能从压缩文件恢复到原始文件。

修改练习

1. 修改程序以便能够处理 20 × 40 大小的位图文件。很容易修改吗？
2. 为本程序生成一个模块化设计。构造四个函数：一个读入文件，一个输出读入的文件，一个压缩文件，一个解压缩文件。

应用练习

6.1 对于一个大城市，持续记录了一个月每天处理的废水（百万加仑）量，数据保存在 EX6_1.DAT 文件中，如下：

```
123, 134, 122, 128, 116, 96, 83, 144, 143, 156, 128, 138
121, 129, 117, 96, 87, 148, 149, 151, 129, 138, 127, 126
115, 94, 83, 142
```

写一个程序，用每天 10 百万加仑的间隔，计算它的频度分布。利用数组 sewage_amt[100] 从 EX6_1.DAT 文件中读入数据。以下面的格式输出到屏幕：

日	百万加仑
1	123
2	134
3	122
...	...

每天废水量	频数
81 ~ 90	3
91 ~ 100	3
101 ~ 111	6
...	...

6.2 修改程序，调用一个叫 get_data() 的函数读入数据。

6.3 修改程序。在 get_data() 函数中调用一个 display () 的函数以显示输出。dsplay() 的原型为：

```
void display (int *mil_gal, int array_size);
```

其中 mil_gal 是一个指针，用来将 sewage_amt[100] 的信息传入函数。array_size 是记录的总数目。

6.4 建筑工程师通常用混凝土来建筑高楼。因为混凝土不能抵御张力负载，所以钢筋，也叫螺纹钢，被加到混凝土中以抵御张力负载。螺纹钢有不同的尺寸。下表展示了 ASTM（American Society for Testing and Materials）标准中的螺纹钢的尺寸、重量和直径。

尺寸	重量	直径 (in)
2	0.167	0.250
3	0.376	0.375
4	0.668	0.500
5	1.043	0.625
6	1.502	0.750
7	2.044	0.875
8	2.670	1.000
9	3.400	1.128
10	4.303	1.270
11	5.313	1.410
14	7.650	1.693
18	13.600	2.257

下表显示了在一个停车场地基中所用到的螺纹钢的长度和型号：

尺寸	长度
4	5000.0
10	2000.0
14	1200.0
18	900.0

写一个程序来计算所用螺纹钢的重量。

输入要求如下：

- 用一个二维数组 bar_data[20][3] 从 ASTM 中读入尺寸、重量和直径。
- 用一个二维数组 bar_used[10][2] 读入地基所用螺纹钢的类型和尺寸。

按以下格式输出到屏幕：

尺寸	直径	长度	重量
4	0.500	5000.0	3340.0
10	1.270	2000.0	…
14	1.693	1200.0	…
18	2.257	900.0	…
总和		…	…

6.5 修改程序，通过在 main 中调用一个 output() 函数产生上面的输出。output() 函数的原型必须包含以下两个形参：

```
void output (double *input_bar_data, double *input_bar_used, ...)
```

6.6 修改程序将输出的单位转换为米制。使用下面的转换公式：

1 in = 2.54 cm
1 ft = 12.0 inch
1 lb = 0.454 kg
1 m = 100 cm

按以下格式输出到屏幕：

大小	直径 (cm)	长度 (cm)	重量 (kg)
4	1.27	1524.0	1516.4
10	…	…	…
14	…	…	…
18	…	…	…
总和		…	…

6.7 写一个程序能够计算三个同样大小的矩阵的和，[A]+[B]+[C]。

输入要求如下：

- 从一个文件中读入输入，文件的第一行是矩阵的行和列的数目。
- 文件中其他的内容为矩阵中的元素。

将结果输出到一个文件。

6.8 写一个程序能将两个 6×6 的矩阵相乘。从文件中按行读入矩阵，将结果以矩阵方式输出到一个文件。

6.9 写一个程序能将 $n\times m$ 的矩阵和 $m\times n$ 的矩阵相乘。输入要求如下：

- 从文件中读入 n 和 m。
- 从文件中读入矩阵。

将结果输出到一个文件。

6.10 写一个程序解 3×3 方程组如下：

$$3x_1+4x_2-5x_3=2$$
$$-2x_1+6x_2-12x_3=-8$$
$$6x_1-3x_2+2x_3=5$$

写出用于工解方程组的算法，输入要求如下：

- 从文件中读入 x_1、x_2、x_3 的系数。
- 从同一个文件读入方程右边的值。

把方程的解输出到屏幕。

6.11 一个一维数组有 10 个元素：

```
4.4  3.3  2.2  5.5  1.1  6.6  7.7  10.0  9.9  8.8
```

用冒泡排序将这个 10 个元素降序排列。

6.12 一个二维数组有 20 个元素：

```
3  33   333   3333
5  55   555   5555
1  11   111   1111
4  44   444   4444
2  22   222   2222
```

使用最大交换排序来完成以下任务：

a. 排序数组使得它看起来如下：

```
5   55   555   5555
4   44   444   4444
```

```
3   33   333   3333
2   22   222   2222
1   11   111   1111
```

b. 排序数组使得它看起来如下：

```
1111   111   11  1
2222   222   22  2
3333   333   33  3
4444   444   44  4
5555   555   55  5
```

6.13 用下面的方法来计算一个 $n \times n$ 二维数组的行列式。假设 $n=3$, array[3][3] 如下：

```
11 12 13
21 22 23
31 32 33
```

行列式 $D=(11\times22\times33)+(13\times21\times32)+(12\times23\times31)-(13\times22\times31)-(11\times23\times32)-(12\times21\times33)=0$。

输入要求如下：

- 读入如 a[3][3] 所示的数组。
- 用一个 for 循环来产生 8 个二维数组 b[n][n]。然后找出它们的行列式，其中 $n=2$、3、4、5、6、7、8。

  保存在数组中的元素 b[i][j] = IJ（语法 IJ 代表将 J 放到 I 的右边。其中 $I=i+1$, $J=j+1$。）

  例如，当 $n=8$ 时，B[0][0] = 11, B[0][7] = 18, B[7][0] = 81 以及 B[7][7] = 88。

  输出要求如下：
- 矩阵 a[3][3] 以及它的行列式。
- 矩阵 b[n][n] 以及它们的行列式。

涉及随机数的练习

rand() 函数是一个标准 C 函数，它返回一个在 0 到 RAND_MAX（C 语言定义的一个常量宏）范围的伪随机数。其中 ANSI C 要求 RAND_MAX 至少为 32767。

因为经常要产生一个范围内的随机数，所以总会在一个函数中使用 mod 运算符。如果我们想模拟一次投掷骰子的点数，那么结果就是 1 到 6 的一个整数，而 $(n\%6)+1$ 就会产生这样的结果。如果 n 是一个随机数，我们就能模拟出一个随机的点数。下面两个语句（roll 代表骰子的点数）会模拟出一个骰子的点数：

```
n=rand();
roll=(n%6)+1;
```

这些语句只能部分模拟出一个骰子的点数，这是因为 rand 并不真实产生一个随机数。产生随机数的理论很复杂，这里不介绍过多的细节。但是 rand 函数只是返回一个伪随机数。为了返回更接近于随机的一个数，我们应该同时使用 rand 和 srand 函数。这两个函数在 C 语言中被设计成配合使用。（这使得它们在 C 库函数中与众不同）。

函数 srand 自动在函数 rand 计算时给它一个“种子”。函数 rand 和 srand 通过程序员不可见的一个全局变量联系在一起。srand 修改这个全局变量而 rand 在生成随机数的时候使用这个全局变量。如果 rand 每次调用的时候这个全局变量相同，那么 rand 返回的值就相同。但是如果 rand 每次调用的时候这个全局变量不同，那么 rand 会返回不相关且近似随机的数。这样，为了使 rand 产生随机的数，在每次调用它之前，应该使这个全局变量不同。

C 有一个时间函数，它会读取计算机的时钟，并返回用秒表示的当前的时间的一个整数。如果你没有精确地在每天的同一时间运行这个程序，那么每次运行这个程序都会产生不同的值。

正因为这个特性，time 函数很适合在程序运行时产生不同的整数，也就可以用在 srand 函数中生成一个全局变量，使得 rand 返回一个近似随机的数。这里不讨论细节。如果我们以空 NULL 调用 time 函数，就会得到想要的结果。于是，

```
srand(time(NULL));
```

在 srand 调用时生成一个新的全局变量。可以用下面的语句调用 rand 函数，来模拟投掷骰子的过程。

```
n=rand();
roll=(n%6)+1;
```

函数 rand 和 srand 要求 stdlib.h 文件，time 函数要求 time.h 文件。

同样，如果我们要模拟一副牌，需要产生从 1 到 13 的随机数。用整型变量 card 的语句

```
srand(time(NULL));
n=rand();
card=(n%13)+1;
```

会产生一个随机数。但是牌还分花色（梅花、黑桃、红桃、方片）。花色可以用随机数 1 到 4 来模拟。为了正确处理一副牌，要求没有一张牌是重复的。在工程界，产生随机数在使用 Monte Carlo 分析方法中非常重要。这里略去不讲。但是也许你会在更高级的课程中遇到它们。有了这个背景知识，你就能够写出以下程序了。

6.14 写一个程序，要求用户投掷两次骰子得到点数 7。如果得到点数 11，用户输。如果和既不是 7 也不是 11，用户不输不赢。

6.15 修改程序使得用户可以得到分数。给用户 100 分并跟踪用户现在有多少分。对于每次决定，允许用户选择他要输赢的分数。选择的分数不能大于他现有的分数。用户可以在任何时间停止。

6.16 写一个程序来模拟单个玩家的 21 点游戏。先给用户两张牌。问用户是否需要另外一张。最多 5 张牌。如果总点数大于 21，用户输。如果点数大于等于 17，用户赢。如果用户有 5 张牌且点数小于等于 21，用户赢。否则无法决定。确保没有两张牌是重复的。

6.17 修改程序 6.16 使得用户像 6.15 那样能够处理点数。

本章回顾

本章中学习了如何用数组在连续的内存区域内保存很多数据项。尺寸为 n 的一维数组的索引从 0 到 $n-1$。初始化数组与初始化变量一样，都可以通过声明或赋值语句完成。当利用数组时，要小心“数组索引越界”不是语法错误，所以 C 会继续使用错误的索引运行而产生一个不可预料的错误。在函数中存取数组元素与存取整个数组不同，为了在函数中存取数组元素，数组元素的值传递给函数。为了在函数中存取整个数组，数组的地址传递给函数。一旦熟悉了一维数组，多维数组与此类似。

第 7 章
C Programming: a Q & A Approach

字符串和指针

本章目标

结束本章的学习后，你将可以：

- 声明并初始化字符串。
- 执行字符串的输入/输出。
- 使用字符串函数简化编程任务。
- 理解数组和指针之间的关系。
- 执行变量的动态分配。

在本章中将继续使用数组，但是与第 5 章不同，数组中将不再包含数值类型的元素，我们将考察元素都是字符类型的数组。当字符类型的数组最后一个元素中保存的是一个空字符(\0)的时候，这个字符数组就叫做字符串。

使用数值型数组时，数组中每一个单独的元素是有意义的。但是处理字符串时，使用字符串的第一个元素的地址对我们来说更方便，所以指针就变得很重要。本章将讨论使用指针传递字符串的首地址，同时使用指针操作字符串中的单个字符。

为了完全理解字符串和指针的操作，需要理解内存是如何分布和管理的。在本章的末尾将讨论程序运行时，内存如何被分配并保持，以便能够满足特定分析问题的需要。课程 3.2 介绍单字符串，如果你略过这个课程，或者忘记了如何处理单个字符，那么现在阅读课程 3.2。

在上一个例子中，为了能在文件函数 fopen 中指定一个文件名，使用了一对双引号标记来定义文件名。在 C 中这叫做一个字符串，它代表适合文本处理的一系列字符。在下面的课程中，将讨论一些处理字符串的方法。但是在讨论这些方法之前，先来简单讨论一下字符串这种数据类型。

基本上，字符串这个词表示的是一些字符组合在一起作为一个单元。一个明显的例子就是以前在 printf 函数中使用过的格式字符串。在这些例子中，你可能已经注意到了字符串只是用来显示一个信息，或者是与转换限定符结合来输出变量的内容。但是由于在日常生活中需要使用大量的英文单词，因此除了保存这些字符串以外，还需要能够操作这些字符串。

现在的问题是，C 语言中的字符串操作不直观而且很复杂；你需要大量的思考才能使它正常工作。下面将通过一些例子来演示其中蕴含的这些挑战，然后用一些课程来讨论这些操作。

注意，课程 7.1 使用的代码在实际的 C 语言程序中并不存在，这里只是为了演示。把它当成字符串操作的简单的演示：让我们声明一些字符串。可能的声明方式如下：

```
string     str_a, str_b, str_c;
```

为了初始化字符串，可以用下面的语句：

```
str_a = "a message we like ";
```

现在想把一些词添加到我们生成的这个字符串的末尾（这个叫字符串连接）。可以用下面的语句：

```
str_b = str_a + "that we want to use as an example";
```

希望生成的字符串是"a message we like that we want to use as an example"。我们也希望比较字符串，这样就能写出下面的语句：

```
if (str_b == str_c)
    printf("two strings are equal!\n");
```

现在需要陈述一个失望的事实：上面所有的语句都是错的。很多都不能通过编译阶段。就算通过编译阶段，在运行阶段也会产生错误的结果。这些语句错误的原因如下：

1）使用错误的数据类型。

```
string     str_a, str_b, str_c;
```

这条是错误的，因为在C语言中没有string这个数据类型。替代的方法是用一个字符数组来代表和存储字符串。

2）字符串的直接赋值使其无效。

当用一个字符数组来代表字符串时，直接的后果就是字符串的名字就是数组的名字。在C语言中，数组的名字只是一个常量地址。因此将一个字符串常量赋给一个地址常量会在编译时产生语法错误。

```
str_a = "a message we like";   //值被给常量是错误的
values to a constant
```

另外，右面的字符串常量的本质也需要进一步说明，我们将在课程7.2中解答这一问题。

3）+运算符不能用来连接字符串。

字符串在C语言中只是一个字符数组，因此用+号代表把一个字符串加到另一个字符串的想法是错误的。（字符串现在是很多现代编程语言的内建数据类型，如Java、Perl、Python等。把它当成内建数据类型的好处是很多字符串的操作如连接等都已经被实现了。从这个角度来说，C语言在易用角度上要略逊一筹。）语句

```
str_b = str_a + "that we want to use as an example";
```

有下面的错误：

a. 在赋值语句的左边是一个地址常量。

b. 右边的加法操作包含一个无意义的地址加法——将两个地址相加不会产生一个包含两个字符串内容的新的字符串。

4）不能用==运算符比较两个字符串。

基于上面同样的原因，语句

```
if (str_b == str_c)
    printf("two strings are equal!\n");
```

不会给出我们想要的结果，因为它只是比较两个地址。这个语句一定会产生一个逻辑假。因为这两个地址在内存中的位置是不一样的。有必要再提起一点，上面的语句在C语言中会通过编译，因为它只是告诉C编译器去比较两个地址常量。这显然并不是我们所希望的，因为我们想要比较两个字符串的内容，而不是地址。

现在，你可能担心在C语言中很多事情都无法用字符串去处理。C语言的架构预料到了这一点，因此提供了很多例程来帮助你完成这些。我们会在7.6节介绍。

接下来会详细介绍C语言中的字符串操作。

课程 7.1 声明、初始化和输出字符串及理解内存布局

主题

- 字符数组
- 初始化单个字符
- 初始化字符串
- 输出字符串
- 内存布局

源代码

```
#include <stdio.h>
#include <string.h>
void main(void)
{
    char aa, bb[4], cc[100], dd[100];
    FILE *outfile;
    outfile=fopen("L7_1.out","w") ;
    printf ("**********  Section 1 - Initialising ************\n");
    aa='g';
    bb[0]='C';
    bb[1]='a';
    bb[2]='t';
    bb[3]='\0';
    strcpy(cc,"This is a string constant, also called a string literal.");
    strcpy(dd,cc);
    printf ("\n**********  Section 2 - Printing **********\n");
    putchar(aa);
    putc(aa, outfile);
    fputc(aa, outfile);
```

函数 strcpy 需要 string.h 头文件

aa 是单个字符变量

bb[],cc[],dd[] 是字符数组，它们可以用来保存字符串

用单引号括起单个的字符，在字符数组 bb[] 的每个元素内保存单个字符

bb[] 数组的最后一个元素是 0，这是一个空元素，也是所有字符串的最后一个元素，bb[] 保存字符串 Cat

一个不后接方括号的数组的名字代表数组第一个内存单元的地址。这样，cc 代表 cc[0] 内存单元的地址，函数 strcpy 只接受地址作为参数

strcpy 把字符串拷贝到指定的内存单元中，我们用 char cc[100] 为 cc 在内存保留 100 个单元，调用 strcpy 使用第二个参数的字符串常量填充了其中 56 个单元。注意字符串常量用双引号包括，我们不能使用这样的赋值语句来完成这个任务：cc = "this is a string constant"；通常处理字符串时不使用赋值语句

一个程序体中使用的字符串常量也保存在内存中，因为它不是变量，所以不保存在靠近变量的内存中。一个程序中的字符串常量代表保存这个常量的地址。这样只接受地址作为参数的 strcpy 函数，可以接受一个字符串常量作为它的第二个参数

这里把保存在用 cc 代表的地址上的字符串拷贝到用 dd 代表的地址上

一个单独的字符可以用 putchar 函数输出到屏幕，或者用 putc 或 fputc 输出到文件

```
puts(bb);
fputs(bb,outfile);
printf("%s\n",bb);
fprintf(outfile,"%s\n",bb);
```

puts 函数可以输出字符串到屏幕

fputs 输出字符串到文件

printf 可以用来把字符串输出到屏幕，fprintf 可以把字符串输出到文件，注意用 %s 作为字符串的转换控制符，注意 bb（是一个地址）对应于 %s

```
puts(cc);
puts(dd);
```

从输出可以看出 cc 和 dd 代表同一个字符串。我们用 strcpy 函数将字符串 cc 拷贝到 dd

```
printf("addresses aa=%p, bb=%p, cc=%p, dd=%p\n",aaa,bb,cc,dd);
}
```

%p 是用做地址的格式限定符

输出到屏幕

```
**********  Section 1 - Initializing ************
**********  Section 2 - Printing **********
gCat
Cat
This is a string constant, also called a string literal.
This is a string constant, also called a string literal.
addresses aa=FFF5, bb=FFF0, cc=FF8C, dd=FF20
```

输出到文件

```
ggCatCat
```

解释

1）什么是字符串？字符串就是字符数组。字符串的末尾用截止符（\0）标记。例如，我们把一个字符串保存在 bb[] 字符串数组中，其中每个字符如下：

```
bb[0]='C';
bb[1]='a';
bb[2]='t';
bb[3]='\0'
```

我们知道数组中的所有元素都保存在一个连续递增的内存区域内。这样一个字符串包含的字符编码（通常是 ASCII）被保存在连续的内存单元内。最后一个内存单元保存一个转义符 \0，它被当成一个单独的字符。字符 \0 叫空字符，不要和后面我们描述的空指针混淆。空字符在内存中以一个字符保存，并且所有的位都是 0。

当书写多于一个字符时，使用双引号来括起那些字符。当你这么做的时候，C 自动在保存它的内存位置末尾加一个空字符。因此，如果你忽略了末尾的空字符而直接查询这个字符串中有多少个字符，这样做是不对的。例如本课程序中，在一个语句中写了以下的字符串：

```
"This is a string constant, also called a string literal."
```

虽然它并没有以 \0 结束。C 认为这是一个字符串，并且在将它保存到内存的时候会在后面自动加一个 \0。这样字符数组必须足够大以包含最后的那个 \0，即使它并没有显示出来。对

于这个字符串，字符数组的尺寸至少应该是57（所有字符的个数加上一个\0）。

注意在变量aa中保存的是一个字符，不是字符串。一个字符串保存在字符数组中，一个字符保存在字符变量中。

2）在程序体中，如果一串字符被包含在双引号中，C语言如何处理它？C会把它当成一个地址。C实际上处理保存那个字符串的第一个内存单元的地址。我们将在后面详细讨论用于字符串常量的地址。现在你只需要记住，当一个字符常量出现在程序体中时，C把它看成地址。如果能利用字符常量作为函数的参数，那么也能使用一个地址作为参数。地址经常用一个没有括号的数组名来表示。

3）如何不通过对每个字符赋值来初始化一个字符串？对每个字符赋值来初始化一个字符串是比较无趣的。幸运的是C语言有一些库函数可以用来方便地初始化一个字符串。例如原型在string.h（必须包含在程序中）中的函数strcpy，将开始于内存中的一个地址的字符串拷贝到另外一个地址。例如语句：

```
strcpy(cc,"This is a string constant, also called a string literal.");
```

把用地址"this is a string constant, also called a string literal."代表的源字符串（第二个参数）拷贝到目的字符串（第一个参数），即用没有括号的数组名所代替的地址cc（代表一个地址）中去。注意cc[]声明的尺寸是100，大于存储这个字符串所需要的57。

同时在这个程序中也使用了语句

```
strcpy(dd,cc);
```

这里很清晰地，两个参数都代表地址。因为它们都是没有方括号的数组名。这个语句使得开始于地址cc的字符串被拷贝到了开始于地址dd的内存区域内。

最后strcpy等同于我们在7.1节之前讨论过的字符串赋值语句。那个时候说过，我们不能用=运算符来给一个字符串赋值，具体的原因在如下解释中。

4）可以使用下面的赋值语句来将字符串保存到数组中吗？

```
cc="This is a string constant, also called a string literal.";
```

不可以，这是一个普遍的错误。因为C把赋值语句右边的字符串常量看成一个地址。这样C语言试图将一个地址保存在cc指定的位置上。但是cc声明保存的是字符，而不是地址，这样赋值语句不会工作。记住当处理一个字符串时，大部分时间你应该使用字符串函数，而不是赋值语句。对于数值类型，赋值语句工作得很好，但是对于字符类型却不太好。

5）使用什么函数来输出一个字符？在第3章中讨论了如何使用printf和putchar来将一个字符输出到屏幕，所以这里不再讨论了。

为了将一个字符输出到文件，C有函数putc和fputc，它们的用法很类似，例如语句：

```
putc(a,outfile);
fputc(a,outfile);
```

使得a代表的一个字符变量输出到outfile指定的文件中去（本课中是L7_1.OUT）。

两个函数的通常定义为：

```
putc (character,file pointer);
fputc (character,file pointer);
```

我们也可以通过%c转换限定符来使用fprintf函数将一个字符输出到文件。因为fprintf和printf用法类似，这里不再详述。

6）什么函数能输出一个字符串到屏幕，它们是如何工作的？我们可以用 puts 和 printf 把一个字符串输出到屏幕，语句如下：

```
puts(bb);
printf("%s\n",bb);
```

使得 bb 指定地址的字符串输出到屏幕。

函数 puts 把 bb 中的元素逐个输出到屏幕，直到遇到空字符（空字符不打印），此时它打印一个换行符（意味着把光标转移到下一行），即使并没有显示要求它怎么做。因为 puts 的这个特性，在字符串的后面有一个空字符就变得非常重要。

puts 的通常定义如下：

```
puts (address);
```

其中 address 是数组中第一个元素的地址。

对于 printf，字符串的转换限定符为 %s。利用这个转换限定符，printf 期待一个用地址或指针来指定的字符串。例如上面给出的语句，%s 和 bb 指定的地址一起配合，输出 bb[] 数组中的字符串内容。从输出来看，整个 bb 的内容都被输出了。这是因为 %s 格式限定符使得 printf 打印整个字符串直到遇到空白字符。但是与 puts 不同，printf 不输出换行符，除非你显式地在语句中包含 \n。

7）如何将一个字符串输出到文件？我们可以使用 fputs 或者 fprintf 函数，例如下面的语句：

```
fputs(bb,outfile);
fprintf(outfile, "%s\n", bb);
```

使得 bb 字符数组中的内容输出到 outfile 指定的文件中去。

fputs 通常定义如下：

```
fputs (address, file_pointer);
```

使得 address 指定地址的字符串输出到 file_pointer 指定的文件中去。与 puts 不同，fputs 不在遇到空白字符时输出一个换行符。从本课输出文件中可以看出 puts 把 Cat 输出到两个不同的行，但是 fputs 把两个 Cat 输出到了同一行。

同样，不详细描述 fprintf 函数了，因为它和 printf 用法类似。

8）如果我们写：

```
char aa, cc[2];
aa='g';
strcpy(cc, "g");
```

那么 aa 中的内存内容和 cc 中的内存内容是一致的吗？不是，这演示了字符串的一个基本特点。下面的图代表了两个内存中的内容。

字符变量 aa 的内存	g

字符数组 cc 的内存	g	\0

很明显，数组的方式中有一个空白符，因为这个区别，我们不能把 aa 当成一个字符串。

9）如何在变量表中描述一个字符数组？可以用一个三维的方式描述一个字符数组，如图 7-1 所示：观察到数组中第一个元素的地址在地址列中给出。第一个元素的内容在值列中给出。其他元素的内容在如图所示的三维部分给出。这代表着它们出现在第一个元素的后面。对于一个短字符串，我们显示所有的字符，如 Cat\0。对于一个长字符串，我们显示头几个字符。其他的字符用虚线来代替。

变量名	类型	地址	值
aa	字符	FFF5	g
bb	字符数组	FFF0	C
cc	字符数组	FF6C	T
dd	字符数组	FF28	T

图 7-1 程序中所用的字符变量和数组变量的内容

10）能总结一下学习过的 C 语言中的字符和字符串的输出函数吗？

可以，下面是一个总结：

- putchar 和 printf 配合 %c——输出字符到屏幕
- putc、fputc 和 fprintf 配合 %c——输出字符到文件
- puts 和 printf 配合 %s——输出字符串到屏幕
- fputs 和 fprintf 配合 %s——输出字符串到文件

扩展解释

1）打印出地址后，能知道本课中的字符串保存在内存的什么位置吗？可以，我们注意到内存的位置和字符串的名字以一个升序的方式出现。

```
FF28   FF6C   FFF0   FFF5
 dd     cc     bb     aa
```

我们可以将这些十六进制数转换为十进制数，然后看看这些地址中间存在多少字节。将十六进制数转换为十进制数得到

```
65320   65420   65520   65525
 dd      cc      bb      aa
```

在 dd 和 cc 之间有 65 420−65 320 = 100 字节。在 cc 和 bb 之间有 65 520−65 420 = 100 字节，在 bb 和 aa 之间有 65 525−65 520 = 5 字节。数组 cc 和 dd 声明为有 100 个元素，而 bb 声明有 4 个元素，aa 只有一个元素。输出的地址说明这些内存区域都是紧密排列的。注意 bb 声明为只有 4 个元素，但是在 bb 和 aa 之间有 5 个字节。精确的地址在不同的 C 编译器上实现是彼此不同的，有的编译器也许在 bb 和 aa 之间就会有 4 个字节。本课不去关心这种特定的细节。对于本文程序的编译和执行，在图 7-2 中显示了内存的占用情况（采用线性方式，而不是表格方式）。

图 7-2 字符数组和字符变量占用内存的顺序

从这个演示中可以看出，如果试图把一个大于 100 的字符串写入 dd，我们会扩展到 cc 的内存区域。事实上，因为 C 不检查是否数组越界，它确实会覆盖掉 cc 的部分内容，而并不给用户报错。因此，程序员有责任确保这种事情不会发生。如果你不小心越过了数组的边界，会得到一个没有预期的结果，而且这种错误很难排查。事实上这是造成程序崩溃的主要原因。

2）什么时候进行内存的布局？当程序被编译时，C 利用声明来为每一个变量和数组分配内存（意味着哪个数组或变量会彼此相邻）以及为所有的数据结构分配相应尺寸的块内存。因为这是在编译阶段，我们必须确保数组的尺寸要大于程序执行时所需要的尺寸。换句话

说，我们不能在执行的时候再去扩展已经声明了尺寸的一个数组。为了容纳多余的数据，需要一些高级的编程方法，后面章节中会介绍。

记住，变量的地址在编译的时候已经被确定（并不是绝对地址而是相对地址，可以理解为相对于其他变量的地址）。不能在运行的时候再去移动这些内存单元。如果试图这么做，会使得程序崩溃。

3）程序执行时发生了什么？程序执行时，程序会按照编译时确定的内存布局来分配出一块内存。在模块化设计程序中，函数中的局部变量在函数调用的时候才被分配出来。在函数调用以后，那些为变量和数组分配出的内存一直有效，直到函数结束运行。这时内存被释放（除非你指定不这么做，这种高级编程方法会在第8章介绍）。这样，下一个函数可以使用这个内存了。

一旦内存被分配，我们就可以改变内存单元中的值。但是不能改变某个变量的地址值，或者把这个内存单元放大或缩小一遍重新分布这块内存。例如对于本课程序，我们不可以把一个新的地址赋给dd数组。换句话说，dd被固定在地址FF28。如果试图改变这个地址，程序会崩溃或者不被编译。提到这点是因为有的时候会一不小心这样做，特别是当利用字符数组（字符串）的时候。你会很容易忘掉没有括号的数组名其实是一个地址，然后把它放到一个赋值语句的左边。例如本课程序中，你不能写

```
dd=cc;
```

或者

```
dd="A sample string";
```

因为dd是一个地址，不能出现在赋值语句的左边（同时注意dd是一个lvalue，l代表left）。这样的赋值语句会让我们改变一个不能改变的地址。记住如果要把cc字符数组中的内容拷贝到dd数组中，必须使用strcpy函数。

我们将会看到，处理字符串时必须记住什么代表地址，以及如何在正确的地方使用它们。学完本章后，你会发现可以以不同的方式表达地址。

4）可以像初始化数值数组那样在声明的时候初始化一个字符型数组吗？可以，但是直到第7.6节才演示这种技术。之所以这么做是为了避免混淆，因为那些非法的赋值语句在声明的时候却是合法的。此时，如果你看到字符数组在声明的时候初始化，虽然使用赋值语句看似是非法的，但是在声明的时候，它们却是允许的。

概念回顾

1）一个字符串就是一个以空字符\0结尾的字符数组。

2）C把程序中双引号括起来的字符串当成一个地址。

3）不能这样使用一个赋值语句：

```
cc="This is a string constant, also called a string literal.";
```

将字符串保存到数组cc中。原因在于赋值语句的右边只是一个地址，而且左边是一个地址常量。

4）C用库函数strcpy来初始化一个字符串。通过在程序中包含头文件string.h，strcpy会把开始于一个地址的字符串拷贝到另外一个地址的开始处。例如

```
strcpy(cc,"This is a string constant, also called a string literal.");
```

会把字符串常量拷贝到字符数组 cc 中。

5）为了将字符串输出到文件，C 有 putc 和 fputc 函数。

```
puto (character, file_pointer);
fputc (character, file_pointer);
```

6）为了将字符串输出到标准输出设备/文件，可以使用 puts/fputs。

```
puts (address);
fputs (address, file_pointer);
```

练习

1. 给定下面的声明

```
char aa, bb[10], cc[15], dd[15];
```

判断下面的语句是否有错误，如果有，请指出：

```
a. strcpy(bb,aa);
b. strcpy(bb,"This is 23");
c. puts(aa);
d. fputs(dd);
e. strcpy(cc,'Many words');
```

2. 找出下面的程序中的错误：

```
#include <stdio.h>
void main(void)
{
        char dd[20], pp[30], rr[5];

        dd[6]="D";
        pp="Panda",
        strcpy("abcd",rr);
        printf("%s, %s,%s\n",dd,pp,rr[1]);
}
```

3. 交替使用 printf 函数和 putchar 函数将下列的内容输出到屏幕。

```
123456789012345678901234567*654321098765432109876543 21
A                          *                        A
BB                         *                       BB
CCC                        *                      CCC
DDDD                       *                     DDDD
...                        *                      ...
XXXXXXXXXXXXXXXXXXXXXXXXXX  *  XXXXXXXXXXXXXXXXXXXXXXXXXX
YYYYYYYYYYYYYYYYYYYYYYYYYY * YYYYYYYYYYYYYYYYYYYYYYYYYY
ZZZZZZZZZZZZZZZZZZZZZZZZZZ*ZZZZZZZZZZZZZZZZZZZZZZZZZZ
123456789012345678901234567*654321098765432109876543 21
```

4. 重做问题 3，只用 puts 函数。

5. 重做问题 3，交替使用 fprintf 函数和 fputc 函数将内容输出到 EX7_1_5.OUT 文件中。

6. 重做问题 3，只用 fputs 函数将内容输出到 EX7_1_6.OUT 文件中。

答案

1. a. strcpy(bb,cc)（我们不能在 strcpy 函数中使用单个字符。）

   b. strcpy(bb,"This is 23");（字符串 "this is 23" 包含有 10 个字符，但是 bb 的尺寸声明为 10。这样就没有空间给 C 语言必须要加上的 \0 字符了。）

   c. putchar(aa);

   d. fputs(dd,outfilw);（其中 outfile 是文件指针。）

   e. strcpy(cc,"Many words");

课程 7.2 确定字符串和字符信息及使用 printf

主题

- %s 转换限定
- 确定字符串信息的字符串函数
- 确定单个字符信息的字符函数

源代码

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
void main(void)
{
        char cc, long_string[50], many_lines[70];
        int occupied, reserved, buffer_size;

        printf ("****** Section 1 - conversion specifications ********\n");
        strcpy(long_string,"This is a complete sentence." );
        strcpy(many_lines,"This sentence \ncovers two lines.");
        printf ("[[%s]]\n",long_string);
        printf ("[[%40s]]\n",long_string);
        printf ("[[%-40s]]\n",long_string);
        printf ("[[%40.10s]]\n\n",long_string);

        printf("%s%s\n%s","This is"," one method for printing","string constants.\n");

        printf("\n****** Section 2 - finding information about strings **\n");
        occupied = (int)strlen(many_lines);
        reserved = sizeof(many_lines)/sizeof(char);

        buffer_size = BUFSIZ;
        printf ("occupied = %d \nreserved = %d\n\nOur buffer size is %d\n",
                        occupied, reserved, buffer_size);

        if(isdigit(many_lines[0])) putchar(many_lines[0]);
        cc=tolower(many_lines[0]);
        putchar(cc);
        putchar('\n');
        puts(many_lines);
}
```

字符串函数要求包含头文件 string.h

用于检查或转换字符的大小写以及字符类型的函数需要头文件 ctype.h

转义序列可以放到字符串中

转换限定符 %s 可以用于将字符串输出

printf 函数中的第一个逗号，把格式字符串和其他字符串常量分割开

函数 strlen 用一个地址作为参数，并且能确定开始于这个地址的字符串中有多少个字符

sizeof 操作符确定参数的字节数

BUFSIZ 是一个定义在 stdio.h 文件中的常量宏

isdigit 和 tolower 用于单个的字符

输出

```
****** Section 1 - conversion specifications ********
[[This is a complete sentence.]]
[[                This is a complete sentence.]]
```

```
[[This is a complete sentence.                    ]]
[[                                   This is a ]]
this is one method for printing
string constants.

****** Section 2 - Finding information about strings **
occupied = 32
reserved = 70

Our buffer size is 512
t
This sentence
covers two lines.
```

解释

1）本课程序中，转换限定符 %s、%40s、%-40s 和 %40.10s 分别代表什么？这些转换限定符与用于整型数的转换限定符 d 的用法类似。格式如下：

```
% flag   field_width . precision  s
```

其中 flag、field_width、小数点和 precision 都是可选的。像前面介绍过的，如果指定的 field_width 比需要的小，C 会将其扩展以容纳要输出的值。我们有下面的限定符及结果：

命令及解释	结果
printf(%s\n", long_string); 因为没有指定宽度，所以宽度被扩展为字符串的长度。	[[This is a complete sentence.]]
printf(%40s\n", long_string); 宽度是40，没有flag，所以右对齐。	[[This is a complete sentence.]]
printf(%-40s\n",long_string); 宽度是40，flag使得左对齐。	[[This is a complete sentence.]]
printf(%40.10s\n", long_string); 宽度是40，精度是10，使得头10个字符被输出，没有flag要求看右对齐。	[[This is a]]

2）strlen 函数有什么用？strlen 函数用来确定一个字符串中真实包含的字符的个数。它的具体实现是通过从数组的第一个元素的地址开始搜索，直到到达空字符，在到达空字符前，它会对字符的数量进行计数。函数格式如下：

```
strlen(address);
```

其中 address 告诉 strlen 从哪里开始搜索。函数返回在遇到空字符之前的字节数或内存单元数，但是它并不包括那个空字符。出于实际目的它返回一个整数值。如果要把这个值保存到一个整型变量中，一个稳妥的方法是在返回值上使用 int 转换操作符。例如语句

```
occupied = (int)strlen(many_lines);
```

将字符串 many_lines 中字符的数量赋给变量 occupied（不包含空白字符）。

3）strlen 有什么用？它使我们知道一个字符串需要多少内存，进而优化内存使用。另外，也能帮助我们避免内存越界造成的覆盖错误。

4）sizeof 运算符有什么用？sizeof 运算符计算指定的变量或变量类型所占用的内存的字节数。它是一个非常重要的运算符而不是一个函数。注意我们并不总是需要将参数放到括号内。本文中一直用括号以避免不用括号带来的潜在问题。

sizeof 的格式如下：

```
sizeof(name)
```

其中 name 可以是变量类型或者变量名。例如两个合法的 sizeof 的用法如下：

```
sizeof(int);
sizeof(many_lines);
```

在第一个应用中，sizeof 返回系统中 int 类型所占用的字节数。第二个用法中，返回为字符串 many_lines 分配的内存字节数。注意本例中，sizeof 的返回值和 strlen 的返回值是不一样的。这是因为 sizeof 返回的是为变量 many_lines 分配的内存的字节数，它是 100，而 strlen 返回的是其中实际存储的字符数，它是 34。

5）本课的程序中将 sizeof(many_lines) 除以 sizeof(char)，为什么这么做？这允许我们确定数组中元素的个数。如果想知道数组中元素的个数，只知道数组占用了多少字节是不够的，我们必须将总的字节数除以每个数组元素所占用的字节数，这是因为除了 char，ANSI C 并没有规定每种数据类型需要占用多少个字节。例如，有的 C 编译器用 2 个字节保存一个 integer，而有的用 4 个字节。为了保证移植性，我们将数组的尺寸除以每一个元素所占的字节数。C 语言给 char 一个字节，sizeof(char) 总等于 1，所以在我们的程序中并不需要除以 sizeof(char)。但是我们这么做是为了强调当你想确定数组中元素的个数时，应该将数组总的字节数除以每个数组元素所占用的字节数。

6）在 sizeof(many_lines) 这个表达式中，数组名 many_lines 没有后接括号。这是否意味着 sizeof 只能用于一个地址？不是，这是在 C 语言中不把没有括号的数组名看作地址的一个例子（我们在本书中不会看到另外一次了）。所以记住这个表达式，当使用 sizeof 运算符时，一个没有括号的数组名不代表地址，它只是一个标识符。sizeof 发现和这个标识符关联的存储。

7）函数 isdigit 有什么用？这个函数作用于一个字符，不是字符串。这个函数确定参数传入的字符是否是一个数字（0，1，2，3，4，5，6，7，8，9）。如果不是，返回 0。如果是，返回非 0 整数。例如本课程序中，isdigit 用于下面的行

```
if (isdigit (many_lines[0]))  puts(many_lines);
```

参数，many_lines[0] 是一个字符 T。所以函数返回 0，这样 if 语句中的逻辑值为假，所以 puts 函数没有执行。

8）在我们的程序中使用函数 isdigit 有什么用处？我们发现使用 isdigit 在评估字符串的内容的时候是非常有用的。例如想在字符串中查找一个数字，例如量度或价格。有了 isdigit 函数，我们可以排除那些不是数字的字符，而把精力放到那些是数字的字符串内容上。

它也可以帮助我们检查输入。例如，如果提示用户从键盘上输入数字，我们使用 isdigit 函数来检查用户是否输入了一个数字。如果没有输入数字，可以显示一个错误信息，提示用户重新进行输入。

9）tolower 函数用来做什么？这个函数用于单个字符，而不是字符串上。如果 tolower 函数的参数是一个大写的字符，那么函数返回这个字符的小写版本。它并不修改参数的内容。例如，本课的程序中，我们使用下面的语句：

```
cc = tolower(many_lines[0]);
putchar(cc);
puts(many_lines);
```

因为 many_lines[0] 是一个大写字符 T，tolower(many_lines[0]) 返回小写字符 t。赋值语句把 t 赋给变量 cc。我们可以通过 putchar(cc) 函数的输出来验证。但是，因为 tolower 并不修改参数 many_lines[0]，所以 many_lines 的第一个字符还是大写的字符 T。我们可以通过 puts(many_lines) 函数的输出来验证。

扩展解释

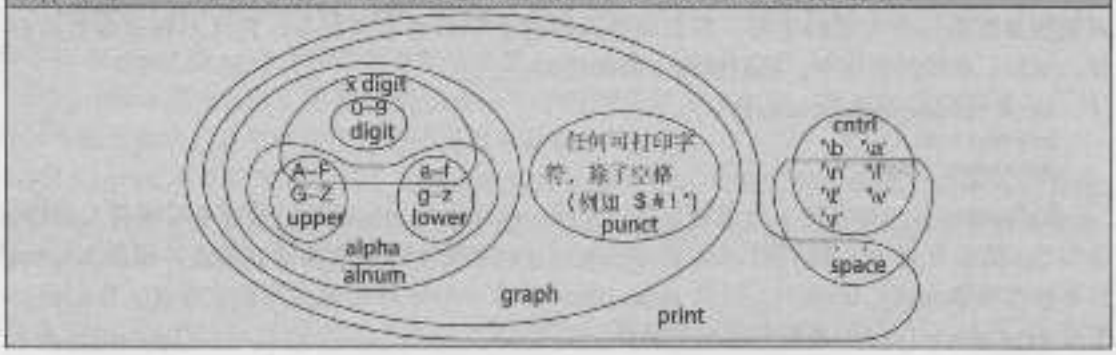
1）BUFSIZ 是什么？ BUFSIZ 是一个常量宏，代表输入缓冲区的尺寸。它在 stdio.h 头文件中定义，代表 C 编译器实现的尺寸。ANSI C 要求它至少是 256 个字节。通常是 512 字节。它给出了通过标准输入流 stdin 一次能传输的最大字符数目。以这种方式传输的字符串不能超过这个尺寸，知道这一点对我们来说很有用。这样就能使用宏 BUFSIZ 来定义从标准输入（大部分情况是键盘）读入的字符数组的尺寸了。后续章节可以看到，对于 C 实现来说不能一次读入超过 512 个字符。

2）strcpy 和 strlen 都使用地址作为输入参数，是不是大部分字符函数都是用地址而不是用变量的名字作为参数呢？是的，例如，与很多数学函数使用变量名字作为参数不同，字符串函数通常使用字符数组的第一个元素的地址来作为它的输入参数。这样在使用字符串函数时，必须知道如何在 C 语言中表达一个地址，对于这个地址有多少个内存单元与之关联，以及我们的程序是如何管理内存的。

3）C 库函数中有没有其他的字符函数？有，表 7-1 列出了原型在 ctype.h 头文件中定义的 C 库中的字符函数。这里并不演示具体的用法，但是我们相信表中的描述会给你足够的信息以便能在自己的程序中使用它们。注意每一个函数都接受单个字符作为参数，用 int 类型代表，因为 C 把字符当成整数。表后面的草图演示了 C 语言中的字符函数类别。

表 7-1 C 库中的字符函数

函数	操作
isalnum(int)	如果参数是任何一个大小写字母或数字（0～9）返回非零，否则返回零。
isalpha(int)	如果参数是任何一个大小写字母返回非零，否则返回零。
iscntrl(int)	如果参数是任何一个控制字符（newline('\n')、horizontal tab('\t')、carriage return('\r')、form feed('\f')、vertical tab('\v')、backspace('\b') 或 alert('\a')）返回非零，否则返回零。
isdigit(int)	如果参数是任何一个数字（0～9）返回非零，否则返回零。
isgraph(int)	如果参数是任何一个可打印字符（除了空格）返回非零，否则返回零。
islower(int)	如果参数是任何一个小写字母返回非零，否则返回零。
isprint(int)	如果参数是任何一个可打印字符（包括空格）返回非零，否则返回零。
ispunct(int)	如果参数是任何一个可打印字符，除了空格或大小写字母或数字（0～9），返回非零，否则返回零。
isspace(int)	如果参数是任何一个空格、newline('\n')、horizontal tab('\t')、carriage return('\r')、form feed('\f')、vertical tab('\v'), 返回非零，否则返回零。
isupper(int)	如果参数是任何一个大写字母返回非零，否则返回零。
isxdigit(int)	如果参数是任何一个数字（0～9）或小写 a～f 或大写 A～F，即十六进制数时返回非零，否则返回零。
tolower(int)	如果参数是任何一个大写字母返回对应的小写字母，否则返回没有变化的参数。
toupper(int)	如果参数是任何一个小写字母返回对应的大写字母，否则返回没有变化的参数。



概念回顾

1）字符串可以按照下面的格式控制输出。

```
%[flag][field_width][.precision]s
```

其中 flag、field_width、小数点和 precsion 是可选的。

2）计算字符串的长度可以用函数

```
strlen(address);
```

函数返回内存单元数或字节数，但是并不包括空白字符。

3）sizeof 运算符返回为某个特定数据类型或变量分配的内存字节数。格式如下：

```
sizeof(name);
```

4）字符函数 isdigit 确定输入的参数是否是 0 到 9 的一个数字。如果不是，函数返回 0，如果是，函数返回非 0，函数定义如下：

```
isdigit( character );
```

5）tolower 函数将输入的字母转化成小写。如果参数是任何一个大写字母，函数返回对应的小写字母。它不修改参数本身，函数定义如下：

```
tolower( character );
```

练习

1. 判断下面的语句是否有错：

```
a. int b;
   b=strlen(double);
b. int d;
   d=strlen("1234567890");
c. long f;
   f=strlen('q');
d. char g[20];
   int h;
   strcpy(gg,"1234567890");
   h=strlen(g[ ]);
e. char aa[30];
   int bb;
   strcpy(aa, "APPLE");
   bb=sizeof(aa[]);
```

2. 在下面的程序中指出错误。

```
#include <stdio.h>
void main(void)
{
char aa[10], bb[50];
strcpy(aa,'Dragon');
strcpy(bb, "Apple, pear, peach, plum");
strcpy(aa,bb);
}
```

3. 写一个程序，程序应该定义

```
char date[30], sender[50], status[40], page[10];
```

并把它输出到屏幕：

```
12345678901234567890123456789012345678901234567890123456789012345678901234
Fax Transaction Report                P.01
-------------------------------------------------------------------------

Date          Sender          Status
12/23/1997    Dell Kevin      OK
12/24/1997    Halton Bors     Out of paper
```

答案

1. a. b = sizeof (double);
   b. No error.
   c. f=strlen("q");
   d. h=strlen(g);
   e. bb=sizeof(aa);

课程 7.3 二维字符数组

主题

- 初始化在二维数组中的字符串
- 输出在二维数组中的字符串

我们已经看到了一维数值型数组的应用性，本课学习二维字符数组。

源代码

```
#include <stdio.h>
#include <string.h>
#define NUM_ROWS 3
#define NUM_COLS 15

void main(void)
{
        char aa[2][50], bb[NUM_ROWS][NUM_COLS];
        int i, occupied, reserved;
        FILE *outfile;
        outfile=fopen("L7_3.OUT","w");

        printf("********** Section 1 - Initialising ************\n");
        strcpy(aa[0],"The aa[ ][ ] array ");
        strcpy(aa[1],"has 2 strings.\n");
        strcpy(bb[0],"The bb array ");
        strcpy(bb[1],"has ");
        strcpy(bb[2],"3 strings.");

        printf("********** Section 2 - Printing ************\n");
        for(i=0; i<2; i++) printf("%s",aa[i]);
        for(i=0; i<NUM_ROWS; i++) puts(bb[i]);
        for(i=0; i<NUM_ROWS; i++) fputs(bb[i],outfile);
```

二维字符数组可以用常数宏来声明，宏数组允许程序员轻松地修改字符数组的尺寸

每个 strcpy 函数的第一个参数是用数组名后接一对括号代表的地址

不同的字符串被放到了 aa 和 bb 中的每一行

因为地址必须用作函数参数，这里使用有一个括号的数组名

循环覆盖数组中的所有行，可以输出二维数组中的所有字符串

```
    reserved = sizeof(bb[0]);
    occupied = strlen(bb[0]);
    printf("Reserved bytes for bb[0]=%d \nOccupied bytes for bb[0]=%d\n",
                    reserved, occupied);
}
```

用在二维数组每行开始位置的 sizeof 操作符返回这个二维数组的列的数目

strlen 返回每行中实际保存的字符数

屏幕输出

```
********* Section 1 - Initialising ***********
********** Section 2 - Printing ************
The aa[ ][ ] array has 2 strings.
The bb array
has
3 strings.
Reserved bytes for bb[0]=15
Occupied bytes for bb[0]=13
```

输出文件 L7_3.OUT

```
The bb array has 3 strings.
```

解释

1）如何定义一个二维字符数组？我们用关键字 char 后接一个标识符以及两对方括号来定义一个二维字符数组。格式如下：

```
char  array_name [number_of_rows][number_of_columns];
```

其中，array_name 可以是任何一个合法的标识符，number_of_rows 和 number_of_columns 必须是正的整常数。例如：

```
char aa[2][90];
```

声明了 aa[][] 是一个占用了 180 内存单元的字符数组。这些单元可以被想象成 2 行 90 列。

2）如何初始化一个二维字符数组？我们用本课程序的例子来描述这一过程，例如，

```
#define NUM_ROWS 3
#define NUM_COLS 15
char bb[NUM_ROWS][NUM_COLS];
strcpy(bb[0],"The bb array ");
strcpy(bb[1],"has ");
strcpy(bb[2],"3 strings.");
```

声明了 bb[][] 是一个 3 行 15 列的数组，行被初始化如下：

T	h	e		b	b		a	r	r	a	y		\0	
h	a	s		\0										
3		s	t	r	i	n	g	s	.	\0				

每个被双引号括起来的字符串被保存到行中。注意每一个字符串的长度必须小于第二维的长度。本例中，第二维的长度为 15，比最长的 14 个字符（包含空字符）的字符串（"The bb array \0"）还长。

当你用这种方式初始化字符串时，千万不要忘了空字符必须要放到每个串的结尾。第二维的尺寸中必须要包括这一字符。例如，bb 最小的可接受的容量是 bb[3][14]，因为第一个字符串中有 14 个字符。

3）二维字符数组中如何使用 sizeof 运算符？ sizeof 可以用来确定整个数组的声明尺寸，或者是一行的，或者是单个元素的。例如，对于本课中的二维数组 bb[][]，语句

```
reserved = sizeof(bb[0]);
```

会评估 bb[][] 中第一行的的声明尺寸（列数），因为 bb[0] 有一对方括号，这代表着它是第一行的地址。从输出中我们可以看出值是 15。这也是我们声明的尺寸。

如果不使用任何方括号，我们可以知道整个数组的尺寸。例如，如果我们使用

```
reserved = sizeof(bb);
```

将返回 45，这是为整个数组 bb 分配的字节的数目。如果我们想知道 bb 中有多少行，可以使用

```
reserved = sizeof(bb)/sizeof(bb[0]);
```

这个语句计算 45/15 = 3，代表其中包括 3 行。

4）如何使用 puts 和 fputs 来输出二维数组？ puts 和 fputs 都使用地址作为它们的参数，然后一直输出直到遇到一个空字符。所以对于一个参数，可以使用数组名后接一对方括号来输出整个一行。例如，

```
puts(bb[i]);
fputs(bb[i],outfile)
```

使得数组 bb 的第 i 行输出。函数 puts 输出到屏幕，函数 fputs 输出到 outfile 文件指针指定的文件中。为了输出 bb 中所有的行，可以把它们放到一个循环中去，如下所示：

```
for (i=0; i<NUM_ROWS; i++) puts(bb[i]);
for (i=0; i<NUM_ROWS; i++) fputs(bb[i],outfile);
```

5）在上面的循环中，puts 和 fputs 会输出相同的内容吗？不会，puts 产生的屏幕输出是：

```
The bb array
has
3 strings.
```

而 fputs 输出到文件中的是

```
The bb array has 3 strings.
```

从这里我们可以清晰地看出 puts 会在每个字符串的后面输出一个换行符而 fputs 不会。使用这些函数的时候，要注意它们的这些差别。

扩展解释

1）如何将一个二维字符数组拷贝到另外的一个二维数组中去？本课程序并没有演示怎么做，但是我们可以使用 strcpy 函数。例如，如果我们声明了两个数组：

```
char cc[4][60], dd[4][40];
```

可以使用下面的循环

```
for (i=0; i<4; i++) strcpy(cc[i], dd[i]);
```

来完成这一任务。这个循环每次把 dd[][] 的一行的内容拷贝到 cc[][]。为了成功拷贝字符

串，cc[][] 数组要有足够多的内存以容纳 dd[][] 中的内容。

2）二维字符数组如何使用 strlen？为了使用 strlen，我们需要把每一个字符串的开始地址，也就是每一行的开始地址作为参数传给 strlen，这样，语句

```
occupied = strlen(bb[0]);
```

把 bb[][] 中第一行的地址传给 strlen，并返回这一行包含的字符数（不包括空白符）。返回的整数值赋给变量 occupied。

概念回顾

1）一个二维字符数组声明如下：

```
char  array_name [number_of_rows][number_of_columns];
```

2）array[row] 可以用作一个地址（注意只有一对括号），这代表第一行的开始位置。这个地址在 C 语言中用来管理二维数组的内容。

3）sizeof 可以用来确定一个二维数组的尺寸以及二维数组中的行数。例如，以 bb 这个二维数组为例：

```
reserved = sizeof(bb[0]);
reserved = sizeof(bb)/sizeof(bb[0]);
```

练习

1. 找出下面语句中的错误：

```
a. char aa[2][10]
   strcpy(aa[0],"aaa);
   strcpy(aa[1],"bbb");
   strcpy(aa[2],"ccc");
b. char bb[2][3]
   strcpy(bb[0],"aaa");
   strcpy(bb[1],"bbb");
c. char cc[ ][25]
   strcpy(cc[0],"Good");
   strcpy(cc[1] "morning");
```

2. 找出下面程序中的错误。

```
void main void()
{
        char a[][12] ={'aaa', 'bbb', 'ccc'};
        char b[2][2];
        strcpy(a[0],a['aaa']);
        strcpy(a[1],'bbb');
        strcpy(b[0],a[0]);
        a[0][0]=strlen(a);
        strcpy (b[0][1], a[0][1]);
        b[1][0]=a[1][0];
}
```

3. 写一个程序，包含一个 student[5][100] 二维字符数组。数组用来保存以下信息：

Name	Age	Math grade
John Kelly	21	3.3
Brian Jason	23	1.8
Mary Fox	19	4.0

第1列是学生的名字，第2列是年龄，第3列是分数，在屏幕上显示表格。

答案

```
1. a. char aa[2][10];
      strcpy(aa[0],"aaa");
      strcpy(aa[1],"bbb");
      aa[][]中两个字符串的最大
   b. char bb[2][4];
      strcpy(bb[0],"aaa");
      strcpy(bb[1],"bbb");
   c. char cc[2][25];
      strcpy(cc[0],"Good");
      strcpy(cc[1], "morning");
```

课程 7.4 从键盘和文件读入字符串

主题

- 从键盘读入字符串
- 从文件读入字符串

到目前为止，我们学习了如何输出一个字符串，但是还没有学习如何从键盘和文件输入一个字符串。本课将描述如何从键盘和文件读入一个一维或二维的字符数组。

源代码

```
#include <stdio.h>
void main(void)
{
        char a[40],b[3][60],c[4][100],d[50],e[30];
        int i;
        FILE *infile;
        FILE *outfile;
        infile=fopen("L7_4.TXT","r");
        outfile=fopen("L7_4.OUT","w");

        printf("************* Section 1 for Array a[ ] *************\n");
        printf("Enter a line of text for a[ ]\n");
        gets(a);
        puts(a);

        printf("\n\n************ Section 2 for Array b[ ][ ] *************\n");
        printf("Type 3 lines for b[ ] and press return at the end of each one\n");
        for (i=0; i<3; i++)
        {
                gets(b[i]);
                puts(b[i]);
        }

        printf("\n\n************ Section 3 for Array c[ ][ ] *************\n");
        printf("We will read c[ ] from a file\n");
        for (i=0; i<4; i++)
        {
                fgets(c[i],100 ,infile);
                puts(c[i]);
                fputs(c[i] ,outfile);
        }
```

数组 a[]、d[] 和 e[] 是一维数组，数组 b[][] 和 c[][] 是二维数组

函数 gets 读入从键盘键入的字符直到遇到回车符。它将字符保存到一个用参数指定的地址上，本例中是 a[] 数组的开始

利用 gets 和二维数组，一个循环可以从键盘读入很多行

保存字符串的位置的地址

保存字符串的文件的指针

函数 fgets 从文件中读入字符串

最大的读入字符串数目

```
    printf("************* Section 4 for Arrays d[ ] and e[ ] *************\n");
    printf("Enter a line of text for d[ ] and press return\n");
    scanf ("%s",d);
    puts(d);
    gets(e);
    printf("This is the rest of the text entered\n");
    puts(e);
}
```

带有%s的函数scanf读入直到遇到一个空白符，所以它不会读入整个字符串

gets读入直到遇到一个换行符，它把除换行符以外的字符读入

输出

```
                    ************ Section 1 for Array a[ ] *******
                    Enter a line of text for a[ ]
键盘输入            This is a short string.
                    This is a short string.

                    ************ Section 2 for Array b[ ][ ] ****
                    Type 3 lines for b[ ] and press return at the end
                    of each one.
键盘输入            This is the first string.
                    This is the first string.
键盘输入            This is the second string.
                    This is the second string.
键盘输入            This is the third string.
                    This is the third string.

                    ************ Section 3 for Array c[ ][ ] ****
                    We will read c[ ] from a file
                    We read

                    four lines of

                    text from our

                    input file.

                    ************ Section 4 for Arrays d[ ] and e[ ] ******
                    Enter a line of text for d[ ] and press return
                    At first, not all of this string is printed.
                    At
键盘输入            This is the rest of the text entered
                    first, not all of this string is printed.
```

输入文件 L7_4.TXT

```
We read
four lines of
text from our
input file.
```

输出文件 L7_4.OUT

```
We read
four lines of
text from our
input file.
```

解释

1）如何用 gets 从键盘读入一个字符串？ gets 函数的形式如下：

```
gets(address);
```

其中 address 是字符串将要保存到的内存中第一个内存单元的地址。大部分情况下，address 是数组的第一个元素的地址，或者是数组中某一行的第一个元素的地址。例如本课的代码中声明并调用了函数

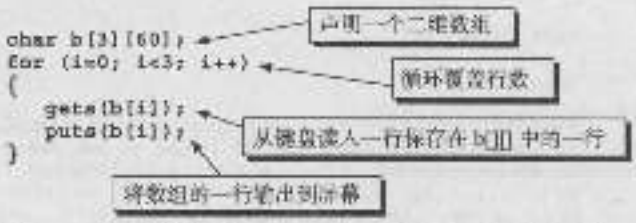
```
char a[40];
gets(a);
```

使得 gets 从键盘读入字符串的输入并将其拷贝到以 a 的第一个元素开始的为 a 分配的内存区域内。函数从键盘读入字符直到遇到一个换行符（通过按回车）。因此，gets 会读入所有键盘输入的字符，但是换行符并不会被保存在内存中。gets 会丢弃掉换行符并在这个位置插入一个空字符。空字符代表内存中字符串的截止。这样，下面的行

```
puts(a);
```

会把 a 输出到屏幕。函数 puts 自动在字符串的末尾加上一个换行符。puts 输出时自动加上换行符的好处在于我们不需要在 a[] 的内存中保存这个换行符了。

2）如何使用 gets 函数从键盘输入来填充二维数组的每一行？可以通过把 gets 函数放到循环里，以使它能依次读入每一行。像我们演示的那样，gets 读入字符直到它遇到换行符。每次你在键盘上敲击回车，我们就执行了一次调用以读入键盘上输入的字符。为了填充一个二维数组，把 gest 放到循环里，循环覆盖所有的行。gets 的参数应该是数组中每一行第一个元素的地址。本课的程序中，声明和循环如下：



循环的每一次使得 gets 依次读入输入的三行，并将每一行保存到 c[i] 中的一行中。注意 gets 的参数是 b[i] 每一行的第一个元素的地址。另外 gets 会丢弃换行符并立即在每一行的最后一个正常的字符后面插入一个空字符。

从输出可以看出，即使 gets 丢弃了换行符，函数 puts 也会在输出时加上换行符。这是因为，gets 丢弃了换行符，puts 加上了换行符，使得输出就像输入的回显一样。

3）如何从文件输入到二维数组？C 语言提供了函数 fgets 从文件中读入。函数 fgets 的格式如下：

```
fgets (address, number_of_char, file_pointer);
```

其中 address 是字符将要保存的第一个单元的地址，number_of_char 是一个比最多读入的字符数还大的数，file_pointer 代表要读入内容的文件。例如，本课语句：

```
fgets(c[i],100,infile);
```

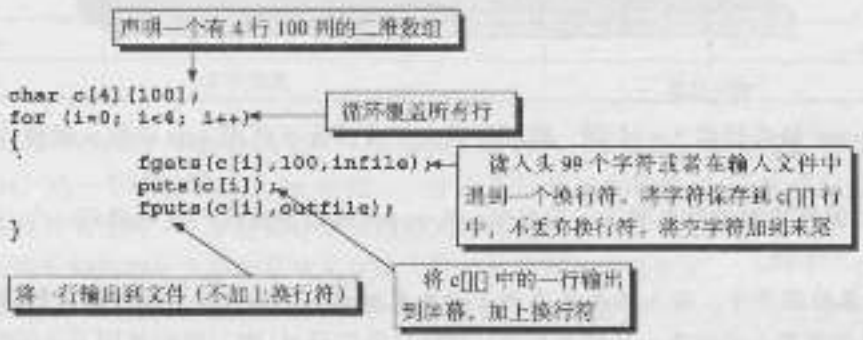
使得头 99 个字符（或者直到遇到换行符）从 infile 指定的文件中读入，并保存到 c[i] 数组

的第 i 行中。如果一个换行符被读入，fgest 并不丢弃它。一个空白字符被加在最后读入的字符后面，不管最后一个字符是否是换行符。这样，如果 99 个字符被读入，那么空白字符被放到第 100 个内存单元内。

下面的声明和循环使得 c[][] 的每一行被输入的字符部分填充：

```
char c[4][100];
for (i=0; i<4; i++)
{
    fgets(c[i],100,infile);
}
```

4）本课中用于 c[][] 的循环将输入内容以同样的方式输出到文件，但是在输出到屏幕时，却在每一行之间加上一个空行，为什么？这是因为函数 puts 在向屏幕输出字符串的时候在每一行的末尾加上一个换行符。但是 fputs 将每一行输出到文件的时候，不加换行符。声明以及循环的注释如下：



输入文件显示如下：

```
We read\n
four lines of\n
text from our\n
input file.\n
```

读入这个输入文件的时候，fgets 在末尾加上一个空白字符，但是并不丢弃任何换行符。这样 fgets 把以下内容放到 c[][] 数组中。

W	e		r	e	a	d	\n	\0						
f	o	u	r		l	i	n	e	s		o	f	\n	\0
t	e	x	t		f	r	o	m		o	u	r	\n	\0
i	n	p	u	t		f	i	l	e	.	\n	\0		

函数 puts 在每一行的末尾加上换行符并且不输出空白符，所以 puts 输出如下：

W	e		r	e	a	d	\n	\n						
f	o	u	r		l	i	n	e	s		o	f	\n	\n
t	e	x	t		f	r	o	m		o	u	r	\n	\n
i	n	p	u	t		f	i	l	e	.	\n	\n		

我们产生了隔行输出，这是因为在每一行有两个换行符。结论就是 C 语言的字符串函数对于换行符和空字符的处理是不一样的。当你读入或者输出时，要小心输入和输出时的这

种差异性。

5）我们可以用 scanf 从键盘读入单独的一行吗？可以，但是整个过程并不是很直观。原因在于 scanf 中的 %s 转换限定符使得 scanf 在遇到一个空白符（不是空字符或者是换行符）时就停止了。所以带有 %s 的 scanf 每次只是读入一个词，不是整个行。例如，声明和语句

```
char d[50];
scanf("%s",d);
```

如果键盘输入是

```
At first, not all of this string is printed.
```

使得 scanf 只是读入词 At，并且保存到数组 d[] 中。句子中剩下的部分还保留在输入缓冲区内，缓冲区内的位置在 At 的 t 和接下来的空格之间。

因为 scanf 每次只读入一个词，我们发现大部分情况下使用 gets 函数从键盘读入整个行更方便。

虽然本课的程序中没有演示，带有 %s 的 fscanf 函数也是读入直到遇到一个空白符，所以也是读入一个词。

6）本课的程序中，输入缓冲区内剩下的内容被读入了吗？是的，gets 函数读入了余下的内容（从位置指示的地方一直到换行符）并将它保存到 e[] 中，我们使用了下面的语句：

```
gets(e);
```

7）我们已经讲解了字符和字符串的输入和输出函数，先总结一下它们吗？图 7-3 演示了字符和字符串的输入输出函数。

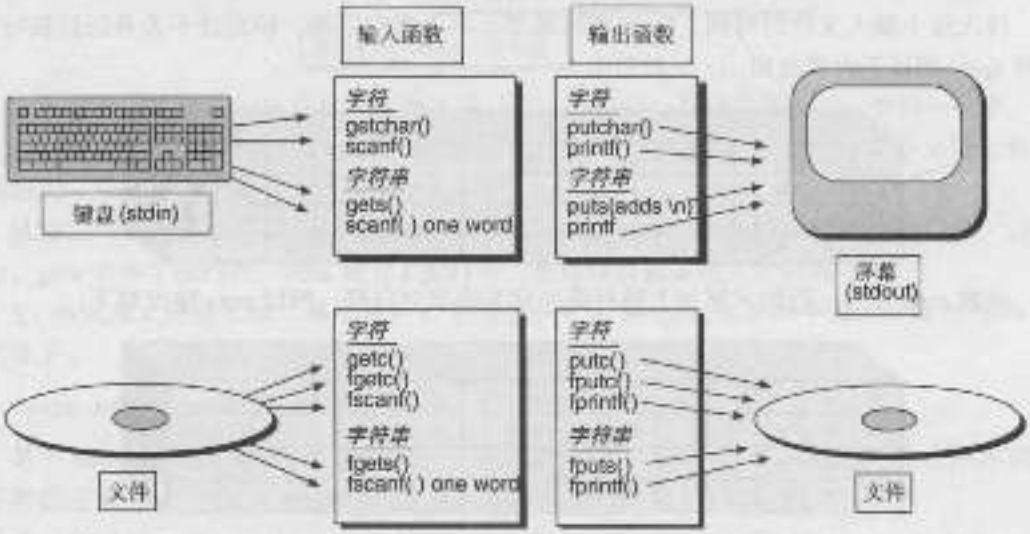


图 7-3 字符和字符串的输入和输出函数

8）以前讲过当 scanf 和 fscanf 遇到读入错误的时候会返回 EOF，而 printf 和 fprintf 函数在输出过程中遇到错误会返回一个负数，那么字符和字符串函数在输入/输出发生错误时会返回特别的值吗？是的，表 7-2 总结了当各种错误发生时，这些函数返回的值。

表 7-2 错误警告

函数	如无误则返回	遇到错误返回
getchar	输入的下一个字符	EOF
gets	字符串第一个元素的地址	空指针
getc	输入的下一个字符	EOF
fgetc	输入的下一个字符	EOF
fgets	字符串第一个元素的地址	空指针
putchar	已写字符	EOF
puts	非负整数	EOF
putc	已写字符	EOF
fputc	已写字符	EOF
fputs	非负整数	EOF

9）什么是空指针，如何使用它？一个指针，当然是一个地址。但是我们可以通过类型转换符（int）把一个地址转换成 int 类型。一个空指针是当你转换成 int 类型后值为 0 的地址。一个空指针与任何一个非空指针比较都是不相等的。

使用空指针特性的一个用法是确定在输入的时候是否有错误发生。

fgets 和 gets 在错误的时候都会返回一个空指针。但是需要一个方法去比较它们。定义在 stdlib.h 中的宏 NULL 就是这样一个常量。我们可以用 NULL 来和函数的返回值做对比以便确定是否有错误发生。

类似的检查输入错误的流程是读入输入的一个改进方法。本课程序中并没有使用是因为我们要降低程序的复杂程度。但是应用程序中会使用错误检查方法。本文中并没有使用很多错误检查方法，不是因为不推荐使用（事实上我们推荐使用），只是因为我们想首先关注其他的一些编程特性。

概念回顾

1）函数 gets 会读入在键盘输入的一个字符串，格式如下：

```
gets(address);
```

其中，address 是字符串将要保存的内存第一个单元的地址。gets 当读入换行符（键盘输入 enter）时停止。

2）fgets 与 gets 函数类似，只是从文件读入，格式如下：

```
fgets (address, number_of_char, file_pointer);
```

3）scanf 处理字符串输入的时候有一个问题。它只读到一个空白字符（不是空字符或换行符）。所以，带有 %s 的 scanf 只是读入一个词，而不是一行。

4）一个空指针就是转换成 int 后是 0 的一个地址。它被一些操作地址的函数如 gets 等当成函数的返回值以代表错误的状态。

练习

1. 下列语句是否有错误，如果有请找出。

```
a. char a[10];
   gets( );
b. char a[1],b[2];
   gets(a, b);
c. puts("AAA BBB");
d. char a[3];
   scanf("%f", a);
e. char a[30];
   gets(a);
```

2. 在下面的程序中找出错误。

```
void main( void)
{ char name[2][30], number[2][10];
printf("Please type your first name, a blank, and last
name);
gets(name[0]);
scanf("   %s     ",name[1]);
printf("name=%s, %s\n", name[0], name[1]);
printf("Please type a number, press the return key, and
another number);
scanf("  %s   ", number[0]);
gets(   number[1]);
printf("number =%s, %s\n",number[0], number[1]);
}
```

3. 利用你学习过的各种字符输入/输出函数从键盘读入你的姓名和邮寄地址，然后
   a. 将它们输出到屏幕和文件中。
   b. 读入刚刚获得的文件，修改它使之成为你的名片的一个硬拷贝。
4. 给定课程7.4的源代码，文件L7_4.C，写一个程序完成以下任务：
   a. 使用循环语句以及fscanf函数读入文件，使用%c每次读入一个字符，然后使用printf和fprintf函数将内容输出到屏幕和文件L7_4.CA中。检查L7_4.C和L7_4.CA是否一致。
   b. 使用循环语句以及fgets函数读入文件，每次读入一个字符，然后使用putc和fputc函数将内容输出到屏幕和文件L7_4.CB中。检查L7_4.C和L7_4.CB是否一致。
5. 下面的程序显示了ANSI C中定义的一些字符输入/输出函数。拷贝并运行这个程序，理解它们是如何正确使用的。

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

void main(void)
{char aa,bb,cc;
  FILE *inptr, *outptr;
aa='A';
bb='B';
cc='C';

  printf("\n\nWrite output file--------------------\n\n");
  outptr= fopen("7_4.OUT","w");

  fputc(aa,outptr);
  putc(bb,outptr);
  fprintf(outptr,"\nThis is output file 7_4.OUT, aa=%c,
```

```
    bb=%c\n",aa,bb);
    fclose(outptr);

    inptr= fopen("7_4.OUT","r");

    aa=getc(inptr);
    printf("\n   d. Use getc() to read character aa=%c from
    a file\n",aa);

    bb=fgetc(inptr);
    printf("\n   d. Use fgetc() to read character bb=%c from
    a file\n",bb);

    printf("\n   f. Use fscanf() to read the rest of file 7_4.
    OUT\n");
    while( (fscanf(inptr,"%c",&cc) !=EOF) ) printf("%c",cc);
    fclose(inptr);
  }
```

答案

1. a. `gets(a);`
   b. `char a[2],b[2];`
      `gets(a);`
      `gets(b);`
   c. No error.
   d. `char a[3];`
      `scanf("%s", a);`
   e. No error.

课程 7.5 指针变量与数组变量

主题

- 指针变量和数组变量的异同
- 初始化及输出数组和指针

我们已经看到，字符串中第一个字符的地址被经常使用。如果有很多的字符串，那么只是保存数组中第一个元素的地址是很方便的。

例如，我们有 10 000 个句子，每一个都是一个单独的字符串。如果将每一个的首地址保存到一个数组中 `a[10000]`，那么可以将 `puts(a[i])` 这个语句放到 `i =0` 到 `i = 9999` 的循环中以输出所有的字符串。但是这并不是经常采取的方法，我们发现，生成一个指针数组以指向所有字符串的开头，这种方法更方便。

回忆一下，我们用 * 声明一个指针变量。例如

```
char *aa;
```

会分配内存给一个叫做 aa 的指针变量。因为使用了关键字 char，aa 被设计成保存一个字符的地址。如下声明

```
char *cc[5]
```

在内存中预留 5 个内存单元，每个单元保存一个字符类型的地址。这种方式刚开始看有点迷惑，只要简单地记住，C 语言把这个声明当成一个指针数组。

以前，我们使用一维和二维字符数组来处理字符串。本课中演示如何使用指针和指针数组来管理字符串以及字符串中的字符。我们会看到两种方法的异同。本程序有三个部分：初始化字符串，使用 puts 输出字符串，使用 putchar 输出字符串。

源代码

```
#include <stdio.h>
#include <string.h>
void main (void)
{
        char *aa;
        char bb[25];

        char *cc[5];
        char dd[5][30];
        char ee[25], ff[30];
        int i, j;

        strcpy(ee,"This is a sample string.");
        strcpy(ff,"Another string.");

        printf("************ Section 1 - Initialising **************\n");

        aa=ee;
        cc[0]=ff;
        cc[1]=ee;

        strcpy(bb,ee);
        strcpy(dd[0],ff);
        strcpy(dd[1],ee);

        printf("************ Section 2 - Printing using puts **************\n");

        puts(aa);
        puts(bb);
        puts(cc[0]);
        puts(dd[0]);

        printf("************ Section 3 - Printing using putchar **************\n");

        for(i=0; i<10; i++) putchar(aa[i]) ;
        putchar('\n');
        for(i=0; i<10; i++) putchar(bb[i]);
        putchar('\n')  ;
```

指针变量

一维字符数组

指针数组

二维字符数组

我们首先声明并初始化了 ee 和 ff 字符串，然后利用它们和 aa、bb、cc、dd 一起工作

利用指针变量 aa 和指针数组 cc[]，可以使用赋值语句，这些赋值语句把 ee 和 ff 的第一个元素的地址赋给指针变量。

strcpy(aa,ee);
strcpy(cc[0],ff);
strcpy(cc[1],ee);

我们不能像使用 bb[] 和 dd[][] 那样使用下面的语句

利用数组 bb[] 和数组 dd[][]，可以使用 strcpy。它们把 ee 和 ff 的实际元素拷贝到了为 bb[] 和 dd[][] 数组分配的内存中。

bb=ee;
dd[0]=ff;
dd[1]=ee;

我们不能像处理指针变量 aa 和指针数组 cc[] 那样去使用下面的语句

但是当利用 aa 输出时，与输出 bb 时类似。显然 aa 是一个指针，而 bb[] 是一个数组

同理，当利用 cc 输出时，与输出 dd 时类似。虽然 cc 是一个指针数组，而 dd[][] 是一个二维数组

我们能使用 aa 来存取 ee 字符串中的任何一个元素，我们能使用数组的符号，即使 aa 是一个指针。注意我们把 ee 的地址赋给了 aa

在 putchar 函数中按相同的方式使用 bb 和 aa，即使 aa 是一个指针，而 bb[] 是一个一维数组

嵌套循环使用 putchar 来输出二维数组

我们在 putchar 函数中按相同的方式使用 dd[][] 和 cc[]，即使 cc 是一个指针数组，而 dd[][] 是一个二维数组

```
    for(i=0; i<2; i++)
    {
            for(j=0; j<10; j++) putchar(cc[i][j]);
            putchar('\n');
    }

    for(i=0; i<2; i++)
    {
            for(j=0; j<10; j++) putchar(dd[i][j]) ;
            putchar('\n');
    }
}
```

我们能通过使用 cc 的二维数组符号来存取 ee 和 ff 中的每一个元素的值，即使 cc 是一个指针数组。注意 cc[0] 被赋予了 ff 的地址，而 cc[1] 被赋予了 ee 的地址

输出

```
*********** Section 1 - Initialising **************
*********** Section 2 - Printing using puts **************
This is a sample string.
This is a sample string.
Another string.
Another string.
*********** Section 3 - Printing using putchar **************
This is a
This is a
Another st
This is a
Another st
This is a
```

解释

1）解释下面两个声明的不同之处：

```
char *aa;
char bb[25];
```

第一个声明为指针变量分配了一个单独的内存位置用来保存指针变量 aa。第二个声明分配了 25 个位置保存字符值。但是所有的内存单元都是空的，直到在程序体中对其进行填充。

2）解释下面两个声明的不同：

```
char *cc[5];
char dd[5][30];
```

第一个声明了 5 个内存位置用来保存指针变量。第二个声明了 5 个内存区域，每个区域包含 30 个字符，所有的内存区域都是空的，直到在程序体中填充它们。

3）为什么语句 aa =ee, cc[0] = ff 和 cc[1] = ee 是正确的？这些语句都是赋值语句，它们把赋值语句右边的值放到赋值语句左边所代表的内存位置上。因为 aa 是一个指针变量，cc[0] 和 cc[1] 是一个指针数组的元素，所以我们能在这些位置上保存一个地址。因为 ee 和 ff 代表一个地址，赋值语句可以被容易地执行。这样我们可以把地址保存在内存单元中，以便以后使用这些地址。

4）为什么语句 bb =ee, dd[0] = ff 和 dd[1] = ee 是不正确的？因为 bb 不是一个指针变量。我们不能在 bb 指定的内存位置上保存一个地址。这个语句明显没有任何意义，因为一个没有括号的 bb 代表数组 bb[] 第一个元素的地址。这个地址在编译的时候被设定，所以不能试图在程序体中改变它。同样，dd[0] 和 dd[1] 也不是指针数组的元素，我们不能在这些单元保存地址。

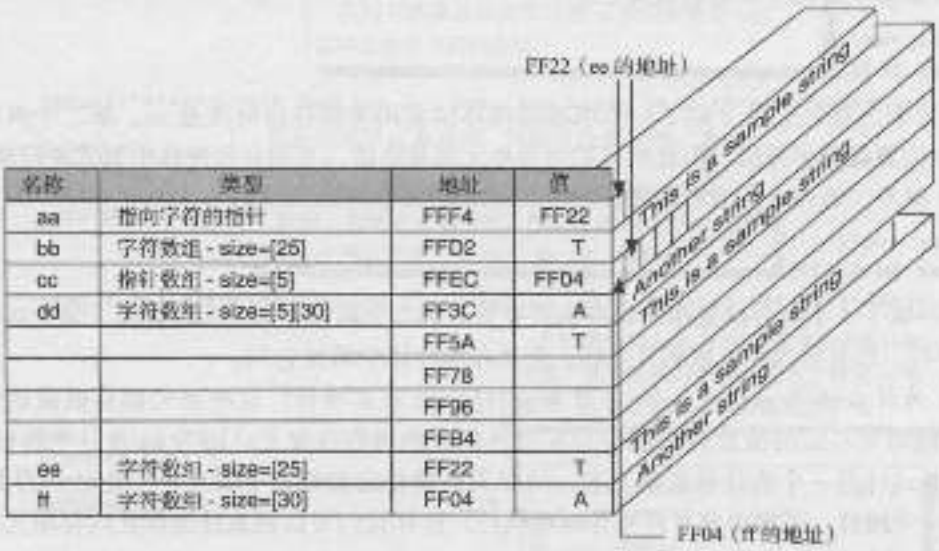
5）为什么语句 strcpy(bb,ee), strcpy(dd[0],ff) 和 strcpy(dd[1],ee) 是正确的？这些语句使得 ee 和 ff 代表的字符串的实际的字符被拷贝到了以 bb、dd[0] 和 dd[1] 所代表的地址作为开头的一些内存单元中去。这是正确的，因为我们为 bb 分配了 25 个内存单元，而 dd[0] 和 dd[1] 分别分配了 30 个内存单元。这意味着 ee[] 和 ff[] 中的所有字符被保存在 bb[] 和 dd[][] 中。

6）为什么语句 strcpy(aa,ee), strcpy(cc[0],ff) 和 strcpy(cc[1],ee) 是不正确的？我们首先考虑 `strcpy(aa,ee)`。因为我们以前并没有把后面一共有 25 个可用的内存单元的地址赋给 aa，所以不能使用这一语句。如果我们把一个分配了内存的地址赋给 aa，也许可以使用 `strcpy(aa,ee)`。

注意，如果分配给 aa 的内存单元数比要求的少（本例是 25），那么使用 `strcpy(aa,ee)` 就会使其他的内存单元被覆盖。这种错误会造成程序的重大问题。如果这样做，注意 C 语言在编译阶段也不报错。本章的后面，我们会演示如何在程序执行的时候为 aa 安全地分配内存以便能使用这个语句。

一个相似的问题存在于 `strcpy(cc[0],ff)` 和 `strcpy(cc[1],ee)` 语句中。因为 cc[] 只是一个数组指针，我们只有保存 5 个地址的空间。这个声明并没有为保存实际的字符分配任何空间。为了安全地对一个指针变量使用 strcpy 函数，我们必须确保第一个参数是带有分配内存的一个地址。

7）当执行完本程序的 Section 1 部分以后，内存中的映像是什么？当执行完本程序的 Section 1 部分以后，有如图 7-4 所表述的内存映像。注意 aa 变量包含 ee[] 的地址，而 bb 中包含 ee[] 的一个拷贝。同理 cc[0] 中包含 ff[] 的地址，而 dd[0] 中包含 ff[] 的一个拷贝。另外，cc[1] 包含 ee 的地址，dd[1] 中包含 ee 的一个拷贝。这演示了指针变量和数组的不同之处。



名称	类型	地址	值
aa	指向字符的指针	FFF4	FF22
bb	字符数组 - size=[25]	FFD2	T
cc	指针数组 - size=[5]	FFEC	FF04
dd	字符数组 - size=[5][30]	FF3C	A
		FF5A	T
		FF78	
		FF96	
		FFB4	
ee	字符数组 - size=[25]	FF22	T
ff	字符数组 - size=[30]	FF04	A

图 7-4 执行完本程序的 Section 1 后的内存配置情况

8）如何使用 aa 指针变量和 puts 函数将保存在 ee 的字符串输出？因为已经利用语句在 aa 中保存了 ee 中第一个字符的地址：

```
aa=ee;
```

我们可以使用带有 aa 的 puts(要求使用一个地址作为参数) 语句来输出保存在 ee 的字符串。

```
puts(aa);
```

9）如何使用 cc[] 指针数组和 puts 函数输出保存在 ee 和 ff 中的字符串？因为我们已经分别利用语句在 cc[0] 和 cc[1] 中保存了 ee 和 ff 中第一个字符的地址，执行以下语句

```
cc[0]=ff;
cc[1]=ee;
```

我们能使用带有 cc[0] 和 cc[1] 的 puts 输出 ee 和 ff 字符串。

```
puts (cc[0]);
puts (cc[1]);
```

10）如何使用 aa 指针变量和 putchar 函数输出保存在 ee 中的字符串？因为 putchar 要求单个的字符元素作为参数，我们必须通过 aa 存取 ee[] 数组中的每一个元素（通过 aa=ee 将 ee 的地址赋给了 aa）。这可能看起来有点奇怪，但是 C 语言允许我们在一个指针上使用数组的符号来存取数组中的单个元素。这意味着 aa[0] 存取 ee[0]，aa[1] 存取 ee[1]，其他类似。这样语句和循环：

```
aa=ee;
for(i=0; i<10; i++) putchar(aa[i]) ;
```

会把字符串 ee 中的前 10 个字符打印输出（即 "This is a"）。

11）如何使用 cc[] 指针数组和 putchar 函数输出保存在 ee 和 ff 中的字符串？同理，我们必须利用保存在指针数组中的指针存取 ee[] 和 ff[] 数组中的每一个元素。C 也允许我们使用数组的符号来完成这个任务。换句话说，当利用下面的语句把 ff 的地址保存在 cc[0] 中的时候

```
cc[0]=ff;
```

cc[0][0] 代表 ff[] 的第一个元素，cc[0][1] 代表 ff[] 的第二个元素，以此类推。因此循环

```
for(j=0; j<10; j++) putchar(cc[0][j]);
```

会把 ff[] 中头 10 个字符输出出来。为了打印 ee 和 ff 的头 10 个字符，我们使用下面的嵌套循环：

```
cc[0]=ff;
cc[1]=ee;
for(i=0; i<2; i++)
{
   for(j=0; j<10; j++) putchar(cc[i][j]);
   putchar('\n');
}
```

12）为什么 C 语言中允许对指针变量使用数组的符号？C 允许这样做是因为当程序编译的时候，它把数组符号的表示转换为指针符号的表示。回忆一元运算符 * 用做指针符号，而方括号用作数组符号。利用这两种表示方法的任何一种，我们都能在特定地址或者特定地址后的几个元素存取一个元素。例如，本课程序中的 aa[5] 存取保存在 aa 中的地址向后数第 5 个元素。这些内容留在后面的课程中讨论。

13）我们能在单个的字符上使用赋值语句吗？可以，虽然我们在本课中没有演示，但是下面的赋值语句：

```
bb[5]='e';
bb[6]=dd[0][3]
aa[2]='p';
cc[1][6]=bb[2];
```

这些语句修改 bb[]

这些语句修改 cc[]

是完全可以接受的。只是对于字符串，我们才需要使用 strcpy 来修改和初始化。如果我们在本课程序中执行初始化以后再执行上面列出的赋值语句，那么字符串将会是：

数组	字符串
bb[]	This is a sample string
cc[]	Thps is a sample string

记住，利用一个指针变量（本例中是 aa）C 允许使用一维数组的符号来存取单个元素。利用指针数组（以 cc 为例）C 允许使用二维数组的符号来存取单个元素。

14）本课的要点是什么？你可以这么理解：

a. 为了初始化字符数组，我们使用 strcpy。为了初始化指针变量和指针数组，我们使用赋值语句。

b. 即使使用不同的方法来初始化字符数组和指针（如我们在 a 中表述的那样），我们用相同的方法来存取字符串和单个的字符。换句话说，可以使用数组符号（方括号）在指针或数组上存取字符串和单个字符。

概念回顾

1）声明 `char* aa;` 只保留了单个内存位置给指针变量 aa。但是声明 `char bb[25];` 为字符分配了 25 个位置。

2）数组名就是地址常量。这样在程序的执行过程中它不能被修改。但是指针变量在程序的执行过程中可以被自由修改。

3）指针数组

```
char *cc[5];
```

声明了一个含有 5 个字符指针的数组。

4）当使用如 strcpy 那样的字符函数时，确保目的参数的地址是一个分配了空间的地址。也就是说，是一个有足够空间容纳要传递过来的内容的一个数组。

5）C 允许你把数组符号用在指针上，因为当程序被编译的时候，它把数组符号转换成指针符号。因此可以在程序中混合使用这两种符号表示。

练习

1. 基于以下声明和初始化：

```
char      aa[10], *bb, *cc[2], dd[2][10], ee[10], ff[10];

strcpy(aa,"Apple");
strcpy(ee,"Cat");
strcpy(ff,"Cow");
strcpy(dd[0],"Dog");
strcpy(dd[1],"Doll");
```

指出下面语句中的错误。

```
a. bb[0]=dd[0];
b. cc[2]=dd[0];
c. strcpy(aa,dd[1]);
d. aa[1]=dd[1][1];
e. strcpy(aa[1], cc[1]);
f. strcpy(dd[1][1],aa);
```

2. 在下面的程序中找出错误。

```
main()
{
    char a[10], *b,  *c[2], d[2][10];

    strcpy(aa,"Apple");
    strcpy(d[1],"Dog");
    d[0][0]=strlen(b);
    d[0][1]='\0';
    b=D[0];
    c[0]=a;
    strcpy(c[1],b);
}
```

3. 一个文件目录包含以下信息：

File name	File size	Date
AAA.C	1234	08-12-2004
XXX.TXT	5678	12-11-2006
DDD.C	9876	01-12-2007
BBB.TXT	4455	06-08-2003

写一个程序使用二维数组来读入这个目录的内容，然后基于以下内容排列目录：

a. 文件尺寸

b. 文件时间

c. 文件类型和名字（例如，文件应该如下排序 AAA.C, DDD.C BBB.TXT 和 XXX.TXT）

将结果输出到屏幕，你不可使用 strcpy 函数。

答案

```
1. a. bb=dd[0];
   b. cc[1]=dd[0];
   c. 无误
   d. 无误
   e. cc[1]=aa;
   f. strcpy(dd[1],aa);
```

课程 7.6　在声明中初始化

主题

- 在声明的时候初始化一个字符串
- 在声明的时候初始化和在程序体中初始化的不同

本程序中使用 4 种不同的声明来保存字符串并在声明的时候初始化这些字符串。我们故意延迟讨论这些貌似简单的操作，其实是为了避免混淆。在解释之前，我们希望你能正确地理解如何利用数组、指针和指针数组，这就使得本部分必须延后。有了这些背景知识，你会很容易本课中要讲述的概念。

本课中你会学到，在声明中，被包含在双括号中的字符有时并不代表地址。当我们初始

化字符数组的时候就是这种情况。当我们初始化指针或指针数组时，它代表一个地址。这种使用等号对一个普通的字符数组进行声明和赋值的格式，与以前使用过的赋值语句类似。这就使得学生在学习C语言编程的时候很困惑。因为以前讲过，我们不能在程序体中使用赋值语句对一个字符串初始化。

本课中，我们讲解了字符串常量被保存在内存的位置，并且与数组的保存位置做了对比。我们指出，当存取一个字符常量时，必须使用地址，因为没有名字和它关联。阅读程序并查看输出，理解字符串如何在声明中被初始化。

源代码

一个指针变量可以指向一个字符串常量的位置，这个声明使得变量aa保存字符串位置的地址

当用数组声明和初始化的时候，字符串被保存到数组中。注意C不允许用bb数组在程序体中使用赋值语句来进行初始化

```
#include <stdio.h>

void main(void)
{
        char *aa= "We can use a pointer to a string constant." ;
        char bb[ ]="We can also use an array.";
```

初始化一个数组指针，这些指针指向下面的字符串常量的位置。数组的尺寸并不需要显式给出

这些字符串并不代表地址（像它们在程序的其他地方那样），因为它们被用在数组的声明中。这些被包含在双引号中的字符串被直接放置到为数组分配的内存中去

```
        char *cc[ ]= {"We can","use an array","of pointers to a constant."};
        char dd[ ][11]={"Or we can","use a 2-D","array."};
        int i ;
```

一个二维数组的声明使得字符串被保存到为数组分配的内存单元中。我们并不需要显式指明第一维的尺寸。注意C不允许我们在程序体中使用这样的赋值语句

这些串代表地址

```
        printf("Addresses: \n\taa=%p \n\tcc[0]=%p \n\tcc[1]=%p \n\tcc[2]=%p"
                "\n\tbb=%p \n\tdd[0]=%p \n\tdd[1]=%p \n\tdd[2]=%p \n",
                        aa, cc[0], cc[1], cc[2], bb, dd[0], dd[1], dd[2]) ;
```

输出所有地址

```
        puts(aa);
        puts(bb);
```

aa和bb代表地址

```
        for (i=0; i<3; i++) puts(cc[i]);
        for (i=0; i<3; i++) puts(dd[i]);
```

cc[i]和dd[i]代表地址

```
}
```

输出

```
aa=00E9
cc[0]=0114
cc[1]=0119
cc[2]=0128
bb=FFD6
dd[0]=FFB4
dd[1]=FFBF
dd[2]=FFCA
```

指针变量中的地址全以0开始

数组地址全以F开始

```
We can use a pointer to a string constant.
We can also use an array.
We can
use an array
of pointers to a constant.
Or we can
use a 2-D
array.
```

解释

1）如何为一个指针变量分配内存，并在声明的时候使其指向一个字符串（即初始化）？可以声明并初始化一个指针，使其指向一个字符串常量。例如，

```
char *aa="We can use a pointer to a string constant.";
```

声明了一个字符类型的指针变量aa来保存字符串"We can use a pointer to a string constant."第一个字符的地址。

2）aa指向的这个字符串被保存在内存的什么位置？我们输出了这个地址：00E9。注意与我们以前查看过的地址不同，这个地址并不是以F开头的。这代表着这个字符串常量被保存在与变量分离的一块内存区域内。

3）以字符串被存储的角度来说，下面这两个声明有什么不同之处？

```
char *aa="We can use a pointer to a string constant.";
char bb[ ]={"We can also use an array."};
```

第一个声明，使用指针变量aa使得字符常量"We can use a pointer to a string constant."的地址保存到指针变量aa的内存单元内。第二个声明使得"We can also use an array."字符串被保存到为bb[]分配的内存单元内。整个内存的映像如图7-5所示。我们可以看到，虽然两个声明都使得字符串被保存到内存中，但保存的方法是不一样的。注意字符串常量"We can use a pointer to a string constant."没有名字和它关联，唯一访问它的方法就是用它的地址。

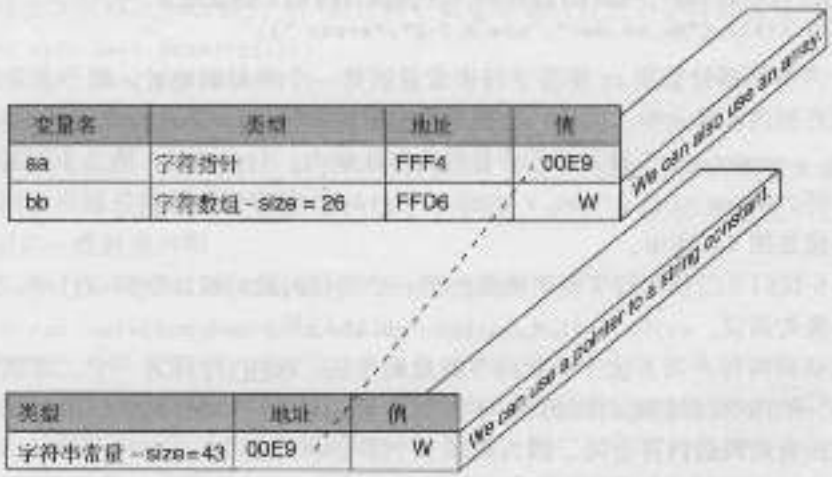


图7-5 aa和bb在初始化后的内存配置情况

4）给定这些存储方法，如何输出保存的字符串？我们可以使用puts函数。回忆puts使

用输出的字符串第一个字符的地址作为参数。本程序中，下面两个语句：

```
puts(aa);
puts(bb);
```

将字符串输出到屏幕。注意两个语句的相似之处。它们都可以工作，因为 aa 和 bb 都代表一个地址。因为 aa 是一个指针变量，表达式 aa 代表变量 aa 的值，是一个地址，也就是字符串常量开始的地址。因为 bb 是一个没有括号的数组名，它也代表一个地址，第一个内存单元的地址。因此，语句和它们的输出结果看起来就很相似。

5）如何访问指针 aa 指向的字符串常量中的单个字符？我们可以使用上面介绍过的数组符号。例如，aa[3] 代表字符串中第 4 个字符 "c"。

6）如何声明并初始化一个指针数组以使指向很多的字符串？我们可以声明并初始化一个指针数组使得每一个元素指向不同的字符串常量，例如声明

char *cc[] = {"We can", "use an array", "of pointers to a constant."}; 生成一个数组以包含三个指针变量。每一个值都代表三个给定字符串的第一个字符的地址。第一个指针变量的值是字符串 "We can" 第一个字符的地址。第二个指针变量的值是字符串 "use an array" 第一个字符的地址。第三个指针变量的值是字符串 "of pointers to a constant." 第一个字符的地址。另外，即使这些字符串不在程序体内，它们也代表地址。

学习并牢记以下给定的符号，格式如下：

```
char *array_name[ ]={"string_1", "string_2",  "string_3"};...
```

观察到一对括号代表我们正声明一个一维数组，* 号代表我们在其中保存地址，这些可以帮助你记忆这种表达方式。

本课程序中，我们并没有指定数组尺寸。C 可以根据列出的字符串数目自动指定尺寸。本例中，C 指定为 *cc[3]。

7）指针 cc 指向的字符串在内存的什么位置？同理，这些字符串常量并不保存到与变量相同的内存区域，而是与常量保存在一起。

8）以字符如何被保存的角度来说，下面两个声明有什么不同之处？

```
char *cc[ ]={"We can","use an array","of pointers to a constant."};
char dd[ ][11]={"Or we can","use a 2-D","array."};
```

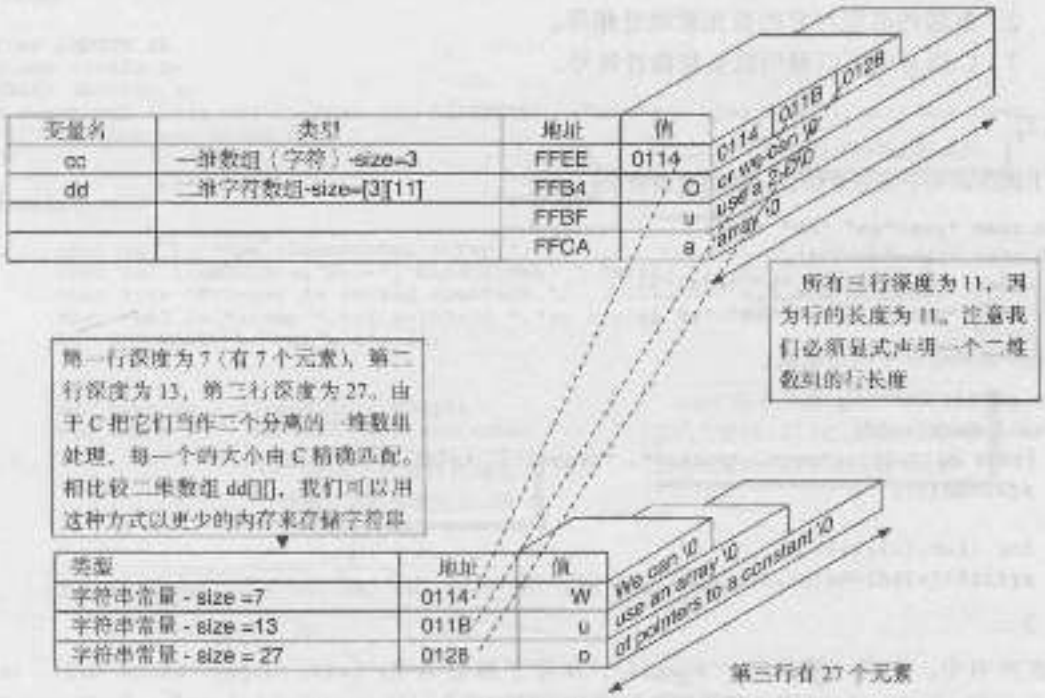
第一个声明用指针数组 cc 使得字符串常量的第一个字母的地址，而不是常量中的字符本身，被保存到内存单元中。因此，字符串中实际的字符，"We can"，"use an array"，"of pointers to a constant." 被保存在常量的内存区域内。与此相反，第二个声明使得字符串中实际的字符 "Or we can"，"use a 2-D"，"array." 保存到为数组分配的内存单元中。整个的内存映像如图 7-6 所示。

从图 7-6 我们可以观察到字符串常量的第一个字母的地址被保存到 cc[] 中。虚线代表它们的关联。换句话说，cc[0] = 0114，cc[1] = 011b 以及 cc[2] = 0128。

同时观察到两种声明方法中分配内存数量的差别，dd[][11] 作为一个二维数组，使得一个"长方形"的内存区域被分配处理，本例中是 3 × 11。作为这样做的结果，最后一行的末尾处有一些没有用到的内存空间，因为最后一个字符串只占用了 7 个内存单元。我们没有办法使用这种声明的同时避免最后一行的内存浪费。

但是，声明 cc[] 的方法产生了三个分离的一维数组，使得 C 语言可以对每一个正确地确定尺寸以避免内存的浪费。虽然在 d[][] 数组中被浪费的内存数量没有什么大的影响，我

们还是需要指出这两种方法的不同，因为有的时候使用二维数组的方法造成的内存浪费会非常的可观。以内存使用的角度来说，使用数组指针的方法更好。



变量名	类型	地址	值
cc	一维数组（字符）-size=3	FFEE	0114
dd	二维字符数组-size=[3][11]	FFB4	O
		FFBF	u
		FFCA	a

类型	地址	值
字符串常量 - size =7	0114	W
字符串常量 - size =13	011B	u
字符串常量 - size = 27	0128	p

图 7-6 指针数组初始化后的内存配置

9）给定这些保存方法，如何将保存的字符串输出？我们可以使循环覆盖要输出的字符个数并使用 puts 函数；本例中，有三个字符串。回忆 puts 使用输出字符串第一个字符的地址作为参数。因此，作为 puts 的参数，cc[0]、cc[1] 和 cc[2] 使得 cc[] 指定的三个字符串被输出。同理，回忆对于一个二维数组，数组中每一行的地址可以用数组名加一对括号来指定。这样 dd[0]、dd[1] 和 dd[2] 代表 dd[][] 每一行的地址，并可以用作 puts 的参数。下面两个循环使得这些参数用在 puts 函数中，并且两个数组中的所有三个字符串被输出。

```
for(i=0; i<3; i++) puts(cc[i]);
for(i=0; i<3; i++) puts(dd[i]);
```

注意这两个循环的相似性，即使 cc 和 dd 声明是如此的不同。

10）如何访问用 cc 指定的字符串中的单个字符？我们可以使用以前课程介绍过的数组符号。例如 cc[1][4] 代表第 2 个字符串的第 5 个字符“a”。注意 cc 可以使用二维数组符号，即使它使用一维数组声明。

11）在本课的程序体中，我们能使用下面的赋值语句吗？

```
aa = "We can use a pointer to a string constant.";
bb = "We can also use an array";
```

第一个语句可以，但是第二个不行。第一个语句可行是因为被包含在双引号中的字符串代表一个地址，我们可以把一个地址赋给指针变量 aa。第二个不可行是因为 bb 在编译的时候给定地址（因为这是一个数组）。一个新的地址不能在程序运行的时候赋给 bb。C 语言有的时候会令人迷惑，因为我们还允许 `char bb[] = "We can also use an array"` 这样的声明，但是不允许 `bb = "We can also use an array"` 这样的语句。

概念回顾

1）指针变量的声明不会使得为数组分配内存。

2）数组的名字与它的首元素地址相等。

3）C语言中可以混用数组和指针符号。

练习

1. 下面的语句中是否有错误，如果有请指出：

```
a. char *paa="aa" "bb" "cc";
b. char *pbb="abc[3]";
c. char *pcc[3]={"a","b","c[3]"};
d. char *pdd[2]={"aa" "bb" "cc"};
```

2. 在下面的程序中找出错误。

```
#include <string.h>
void main(void)
 {char dd[3][8]={"Dog", "Donkey", "Dragon"}, *x[3]={"aa",'bb'};
 x[2]=dd[3];

 for (i=0;i<3;i++)
 printf("x[%d]=%s\n",i,x[i]);

 }
```

3. 在声明中，使用一维数组 * name[4]，保存下面的名字：peter dodge、keith hill、eric randy、lisa fredo。写一个程序把名字转换为：

```
a. Peter Dodge, Keith Hill, Eric Randy, Lisa Fredo
b. PETER DODGE, KEITH HILL, ERIC RANDY, LISA FREDO
```

答案

1. a. 无误，paa 初始化为字符串常量地址 "aabbcc"。

b. 无误，pbb 初始化为字符串常量地址 "abc[3]"。

c. 无误，pcc[2] 初始化为字符串常量地址 "c[3]"。

d. 无误，但只有 pdd[0] 初始化为字符串常量 "aabbcc"。

课程 7.7 将字符串传入用户自定义函数

主题

- 计算在一个数组中行和列的数目
- 将数组和指针传递给函数

本课中演示如何写一个程序，以及程序如何接受和传递一个字符串，目前为止，我们已经用过4种不同的方法来声明一个字符串：

1）一维字符数组。

2）二维字符数组。

3）指向字符的指针。

4）指向字符的指针数组。

它们并不只是所有可用的候选，只代表着我们可以学习的一些方法。本课的源代码中，每一

种方式都被声明并传递给函数。

源代码

```
#define LENGTH 20
#include <stdio.h>
#include <string.h>
void function1 (char ee[ ], char ff[ ][LENGTH], char *gg, char *hh[ ], int num_rows_
        ff, int num_elems_hh);

void main(void)
{
        char aa[ ]= "One-dimensional array.";
        char bb[ ][LENGTH]={"Two-","dimensional ","array."};
        char *cc= "Pointer to string constant.";
        char *dd[ ]={"Array ","of pointers ","to string ","constants."};
        int num_rows_bb, num_elems_dd;

        num_rows_bb = sizeof(bb)/LENGTH;
        num_elems_dd = sizeof(dd)/sizeof(char *);

        function1(aa, bb, cc, dd, num_rows_bb, num_elems_dd);
}

void function1 (char ee[ ], char ff[ ][LENGTH], char *gg, char *hh[ ], int num
        rows_ff, int num_elems_hh)
{
        int i;

        puts(ee);
        for (i=0; i<num_rows_ff ; i++) puts(ff[i]);
        puts(gg);
        for (i=0; i<num_elems_hh; i++) puts(hh[i]);
}
```


输出

```
One-dimensional array.
Two-
dimensional
array.
Pointer to string constant.
Array
of pointers
to string
constants.
```

解释

1）如何把这四种声明传递给函数？使用下面的函数调用

```
function1(aa, bb, cc, dd, num_rows_bb, num_elems_dd);
```

我们把下列传递给函数：

- aa[] 的第一个元素的地址。
- bb[][] 的第一个元素的地址。
- cc 的值，是一个地址，因为它是一个指针变量。
- dd[] 的第一个元素的地址。

注意，在把这些传递给函数时，我们只是给出了每一个的名字。所以，即使它们被声明的如此不同，我们也可以用很简单的方式把它们传递给函数。

2）如何定义 function1 的函数原型？在程序中使用函数时，书写正确的函数原型是非常重要的。本例中因为函数直接使用了 main 中相应参数的声明，所以显得非常直接。下面的表格中我们给出了函数原型中的声明和 main 函数中的声明：

main 的声明	function1 的声明
char aa[]	char aa[]
char bb[][LENGTH]	char bb[][LENGTH]
char *cc	char *aa
char *dd[]	char *bb[]

注意两个声明的相似性。

3）调用 function1 的时候，我们指出传递了 4 个地址。每一个和其他的很类似，都只是一个标识符的名字。为什么不需要其他的参数格式？为了使得函数原型中的每一个参数不同，我们只是对每一项指定不同的指针代数运算。本课的后面以及课程 7.9 中我们会介绍更多的指针代数运算。现在仅仅提到，函数需要有足够的信息来正确地执行指针代数运算以利用正确的内存单元。

在很多情况下，如果函数在 main 中被调用时使用的参数和在函数原型中定义的参数不匹配，程序不会编译。因此，如果编译器指出了函数调用和函数原型中的类型不匹配，你就要检查并使得它们匹配。

4）为什么以前不用担心函数调用和函数原型中的类型不匹配的问题？以前没有对这个问题过分关注是因为我们仅仅利用了一维或多维数值类型的数组，因为它们在函数的声明中使用括号来代表接受一个地址，使用非常直接。现在必须使用数组指针，情况就有点复杂了。

5）为了正确写出函数调用、函数定义和函数原型，我们应该记住哪些内容？记住，通过每一个参数，我们或者传入一个地址，或者传入一个值。我们不能把整个数组的内容通过一个参数传递给函数。如果传递地址，那么在函数调用时就用一个地址。但是为了接受地址，在函数的原型中，我们不能仅仅只是接受一个地址这么简单，为了函数能够使用这个地址进行指针的算术运算，必须要传递另外的足够的信息。例如，必须用指明行的长度的方式来指出这个地址是一个二维数组的地址。

在函数原型中，我们使用方括号或者 * 号来代表接受一个地址。这种使用方括号或者 * 号的方法指明了 C 语言如何在函数体中进行指针的代数运算。例如，两个括号代表这是一个二维数组地址，并且在第二个括号中的数字代表行的长度。这代表了当我们执行指针算术运算的时候必须遵循的原则。在函数体中使用的符号使得这种算术运算得以执行。

例如，在图 7-7 所表述的函数原型定义中，我们把四个地址和两个值传递给了函数。这些地址和值都被拷贝到了 function1 中分配的内存区域内。

图 7-7 function1 的函数原型

6）假如把地址传给 function1，main 和 function1 中的内存区域是什么样的？如图 7-8 所示，我们显示了内存的分布情况。注意从这个图中，function1 中内存区域包含（作为值）四个地址。在这个内存区域内也没有数组。但是 function1 函数能利用所有的数组，因为在函数原型中代表的类型已经给出了足够的信息以便它使用数组。

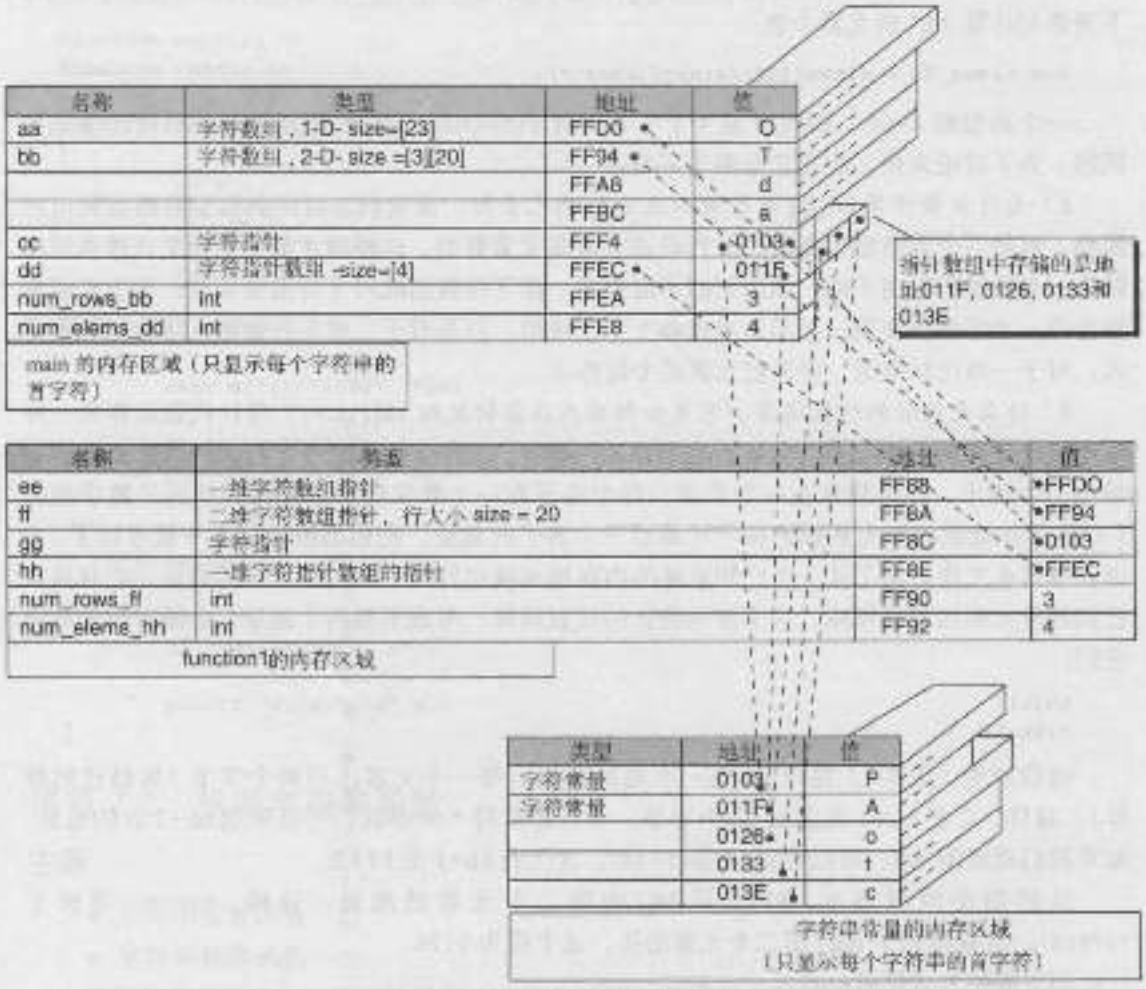


图 7-8 行程序时内存的分配情况

同时，注意main和function1内存区域的关联性。虚线代表main中aa、bb和dd的地址被以cc、ff和hh保存在function1的内存单元中。同时cc的值，也是一个地址，被保存在function1的gg内存单元中。注意function1有了全部的地址，它能修改这些数组。在你自己的函数中，当传递了一个地址，就可以使用类似于puts、gets和以前我们在本章中描述过的函数来使用这些地址。我们已经描述过字符串常量和指针数组的关联，这里不再详细讨论了。

7）如何确定bb[][]的行数以及dd[]中的元素数？我们可以使用sizeof运算符来获得声明它们时的尺寸。例如，sizeof(bb)和sizeof(dd)分别计算为bb和dd分配内存的字节总数。对于bb[]，我们知道每个字符占用一个字节，因此通过将总数除以每一行的长度就知道了有多少行数。以下语句计算bb[][]中有多少行数：

```
num_rows_bb = sizeof(bb)/LENGTH;
```

因为dd[]是一个一维数组，我们把总数除以每个元素占用的字节数。意识到每个元素是一个char类型的地址，而不是如int或者double的数值类型。因此我们除以sizeof(char*)。虽然看起来有点奇怪，sizeof(char*)给出了为保存一个char类型的地址需要多少字节数。下面语句计算dd[]的元素个数：

```
num_elems_dd = sizeof(dd)/sizeof(char *);
```

一个典型的char *的尺寸是4个字节。对于int*、double*及其他类型的地址也是相同的。为了讨论方便，假定它是两个字节。

8）为什么要计算bb[][]的行数以及dd[]的元素数？像我们在讨论数值型数组提到过的那样，通过一个参数把数组的尺寸传递过去是非常重要的。这样做才能让函数更高效地使用数组。与数值型数组不同，然而我们不需要把一维字符数组的尺寸传递给函数，因为它经常被当成一个字符串使用，而不是当成单个字符使用。但是对于二维字符数组，应该把行数传入。对于一维指针数组，应该把元素的个数传入。

9）什么是指针的代数运算，它是如何应用在指针数组hh[]上的？指针代数运算是一种在地址上进行的代数运算（就像加法和减法一样）。以前没有描述过它们，所以这里我们要给出一个例子。假设你住在一个街道，每个房子有一个数字地址并且相邻的房子数字相差1；很简单地就能算出从你的房子开始后三个房子的地址，把你的地址加上3就可以了。C语言也这么工作。为了从一个已知地址的内存单元确定另外一个内存单元的地址，它只是从已知的单元相加或者相减。为了演示指针的代数运算，考虑下面两个地址，C语言同等对待它们：

```
hh[1]
*(hh+1)
```

函数原型（图7-7）指出hh是一个地址数组。每一个元素占用两个字节（如描述的那样）。这样，C把hh+1当成hh+2个字节。单目运算符*使得我们可以得到hh+2后的地址。如果我们观察图7-8，可以看到hh是FFEC，这代表hh+1是FFEE。

虽然图中没有显示，FFEE是dd[]中第二个元素的地址。这样，*(hh+1)等价于*(FFEE)，这就给出了dd[]第二个元素的值，这个值为0126。

用下面的方式使用puts

```
puts (*(hh+1));
puts(hh[1]);
```

与 puts(0126) 是一样的。都会打印出 "of pointers"。

我们会在 7.9 课中更详细的讨论指针的算术运算。

概念回顾

1）将一个数组作为函数的参数，必须确保 main 中的函数定义和函数原型之间是匹配的。在调用函数的语句中，只有数组的名字（开始地址）需要指定；在函数的定义中，必须使用一对括号以代表它是一个数组。在函数的定义中，除了第一维的尺寸以外，其他的尺寸必须被指定。

2）指针的代数运算就是基于地址的代数运算（如加法和减法）。

3）当一个指针递增 k，C 语言首先需要检查指针指向的类型所占用的内存的字节数。例如，一般来说整数占用 4 个字节，这样它会在指针上加上 $4 \times k$ 的数值来计算它所引用的那个内存单元的地址。

练习

1. 在下面的程序中手工计算 x 的值，然后运行这个程序检查你的计算。

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

void f1 (char a, char b[], char *c);
{
        a='a';
        strcpy(b,"bcde");
        b[0]=a;
        c[2]=*b+5;
}
void main(void)
{
        char x[25]="9876", *px;

        f1('1',x,&x[0]);
        printf("1. x=%s\n",x);

        f1('2',x,&x[1]);
        printf("2. x=%s\n",x);

        px=x;
        f1(*(px+1), x+2, px+2);

        printf("3. x=%s\n",x);
}
```

课程 7.8　标准字符串函数

主题

- 字符串管理函数
- 字符串转换函数

像我们在课程 7.1 中提到的那样，字符串管理与数值数据管理之间并不一样。很多的运算符并不能直接用在字符串上。为了帮助程序员高效处理字符数据，C 提供了很多库函数。

C 库函数执行下列任务：

1）把一个字符串连接到另外一个字符串的末尾。

2）搜索一个字符串以找到特定的字符。

3）搜索一个字符串以找到特定的子字符串。

4）将一个字符串的数字部分转换成 int 或者 double。

5）将一个字符串的内容或者一部分内容拷贝到其他的内存单元里（我们已经看过如何使用 strcpy）。

6）按照字母的顺序比较两个字符串。

7）把一个大字符串分解成一系列子字符串。

本课的程序使用 C 的大部分字符串管理函数。本课中使用的函数就是你会经常使用的函数。现在你不需要仔细阅读这个程序，这主要是为你提供参考。但是请仔细阅读本课的解释部分以便了解这些函数都执行了哪些操作。

另外，学习表 7-3 中给出的本课源代码中的例子，为了更好地理解这些例子，请同时查看表格和源代码。

我们用来描述这些函数的格式并不符合 ANSI C 所规定的标准方法。之所以选择这种格式是因为比起标准的格式语言来说，这种方法更易于理解。如果你想知道更多的细节或者更标准的描述，请参考 ANSI C 标准。

另外，现在你不需要读源代码，可以直接跳到本课的解释部分。阅读解释部分并参考源代码。当你在自己的程序中使用这些函数时，利用这些源代码作为指导。

源代码

这个源代码只是一个参考，你不需要学习这个源代码，除非你想使用源代码中给出的那些字符串函数

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

void main(void)
{
  int pos, len, ia, ib;
  char hello[50]="Good", token separator[]=",\n \t...";
  char *pa, *pb, *pc;
  double da;
  long la;
  unsigned long ula;
  FILE *infile;

  printf("/****** A - function atoi ********************************/\n");
  ia=atoi("-123.45xyz");
  printf("A--- atoi() converts -123.45xyz to ia=%5d\n\n\n",ia);
  printf("/****** B - function atof ********************************/\n");
  da=atof("-987.65E+01pqr");
  printf("B--- atof() converts -987.65E+01pqr to da=%8.2lf\n\n\n",da);

  printf("/****** C - function atol ********************************/\n");
  la=atol("-456.89abc");
  printf("C--- atol converts -456.89abc to la=%5ld\n\n\n",la);

  printf("/****** D - function strcat ******************************/\n");
  pa=strcat(hello," xorning!");
  printf("D--- hello=%s ### String at pa=%s\n\n\n",hello, pa);

  printf("/****** E - function strchr ******************************/\n");
  pb=strchr(hello,'n');
  pos=pb-hello+1;
  printf("E--- Character 'n' is the %ldth character of string %s, string at  "
         "pb=%s\n\n\n",pos,hello,pb);
  printf("/****** F - function strcmp ******************************/\n");
  ia=strcmp(hello,"Good xyz");
  if (ia<0)  printf("F--- ia=%2d, %s is less than    Good xyz!\n\n\n",ia, hello);
```

```
if (ia==0) printf("F--- ia=%2d, %s is identical to Good xyz!\n\n",ia,hello);
if (ia>0)  printf("F--- ia=%2d, %s is greater than Good xyz!\n\n\n",ia,hello);

printf("/****** H - function strcpn ********************************/\n");
pos = strcspn(hello, "dog");
printf("H--- The first occurrence of any character in substring, dog,  \n"
     "     in string %s is the %ldnd character, o\n\n\n",hello,pos+1);

printf("/****** I - function strerror ********************************/\n");
errno=0;
infile=fopen("abcdefgh.ijk","r");
if (errno)
{
pa=strerror(errno);
printf("I--- Does file  abcdefgh.ijk exist? strerror() says %s\n",pa);
}

printf("/****** J - function strlen ********************************/\n");
len=strlen(hello);
printf("J--- Not including the null character, %s has %2d characters\n\n\n",
      hello,len);

printf("/****** K - function strncat ********************************/\n");
pb=strncat(hello," John. How are you!",5);
printf("K--- hello=%s $$$ String at pb=%s\n\n\n",hello, pb);


printf("/****** L - function strncmp ********************************/\n");
ib=strncmp(hello,"Good car",15);
if (ib>0) printf("L--- ib=%ld, %s is greater than Good car\n\n\n",ib, hello);


printf("/****** M - function strncpy ********************************/\n");
pa=strncpy(hello+14,"Linda. How are you!", 6);
printf("M--- hello=%s $$$ String at pa=%s\n\n\n",hello,pa);

printf("/****** N - function strpbrk ********************************/\n");
pb=strpbrk(hello,"dear");
printf("N--- hello=%s $$$ String at pb=%s\n\n\n",hello,pb);

printf("/****** O - function strrchr ********************************/\n");
pa=strrchr(hello,'m');
printf("O--- hello=%s, String at pa=%s\n\n\n\n",hello,pa);


printf("/****** P - function strspn ********************************/\n");
ia=strspn("Good year",hello);
printf("P--- The %dth character 'y' is the first character in oGdo year\n"
     "     that is not present in %s\n\n\n",ia+1, hello);
printf("/****** Q - function strstr ********************************/\n");
pa=strstr(hello,"Linda");
ia=pa-hello+1;
printf("Q--- Linda was found at position %d of %s $$$ String at pa=%s\n\n\n",
      ia,hello,pa);

printf("/****** R - function strtod ********************************/\n");
da=strtod("123.45abc",&pb);
printf("R--- Find double number %6.2lf in 123.45abc $$$ String at  "
          "pb=%s\n\n\n",da,pb);

printf("/****** S - function strtol ********************************/\n");
la=strtol("98765xyz",&pa, 10);
printf("S--- Find long number %6ld in 98765xyz $$$ String at"
          "pa=%s\n\n\n",la,pa);

printf("/****** T - function strtoul ********************************/\n");
ula=strtoul("45678pqr",&pc,10);
printf("T--- Find unsigned long %6ld in 45678pqr $$$ String at  "
          "pc=%s\n\n\n",ula,pc);

printf("/****** U - function strtok ********************************/\n");
printf("hello=%s,     token_separator=%s\n",hello,token_separator);
pa=strtok(hello,token_separator);
while ( pa!=NULL)
     {
     printf("U--- String at pa=%10s pa=%3c\n",pa,pa);
     pa=strtok(NULL,token_separator);
     }
}
```

输出

```
****** A - function atoi ********************************
A--- atoi() converts -123.45xyz to ia= -123

****** B - function atof ********************************
B--- atof() converts -987.65E-01pqr to da=-9876.50

****** C - function atol ********************************
C--- atol converts -456.89abc to la= -456

****** D - function strcat ******************************
D--- hello=Good morning! $$$ String at pa=Good morning!

****** E - function strchr ******************************
E--- Character 'm' is the 6th character of string Good morning!,
      String at pb=morning!

****** F - function strcmp ******************************
F--- ia=-11, Good morning! is less than    Good xyz!

****** H - function strcspn *****************************
H--- The first occurrence of any character in substring, dog,
     in string Good morning! is the 2nd character, o

****** I - function strerror ****************************
I--- Does file abcdefgh.ijk exist? strerror() says No such file or
     directory

****** J - function strlen ******************************
J--- Not including the null character, Good morning! has 13 characters

****** K - function strncat *****************************
K--- hello=Good morning! John $$$ String at pb=Good morning! John

****** L - function strncmp *****************************
L--- ib=10, Good morning! John is greater than Good car

****** M - function strcpy ******************************
M--- hello=Good morning! Linda. $$$ String at pa=Linda.

****** N - function strpbrk *****************************
N--- hello=Good morning! Linda. $$$ String at pb=d morning! Linda.

****** O - function strrchr *****************************
O--- hello=Good morning! Linda., String at pa=morning! Linda.

****** P - function strspn ******************************
P--- The 6th character 'y' is the first character in oddo year
     that is not present in Good morning! Linda.

****** Q - function strstr ******************************
Q--- Linda was found at position 15 of Good morning! Linda. ###
      String at pa=Linda.

****** R - function strtod ******************************
R--- Find double number 123.45 in 123.45abc $$$ String at pb=abc

****** S - function strtol ******************************
S--- Find long number  98765 in 98765xyz $$$ String at pa=xyz

****** T - function strtoul *****************************
T--- Find unsigned long  45678 in 45678pqr $$$ String at pc=pqr
```

```
****** U - function strtok ***********************************
hello=Good morning! Linda..     token_separator=!,
    ...
U--- String at pa=      Good pa=65416
U--- String at pa=   morning pa=65421
U--- String at pa=     Linda pa=65430
```

解释

1）如何使用本课程序中的字符串函数？表 7-3 中列出了本课所使用的与 ANSI C 标准兼容的字符串函数。同时在本课的源代码中的简单解释给出了如何使用。如果你的编译器并不是 ANSI C 兼容的，它可能不支持所有的这些函数。查看你的手册以获取相应的细节。

本课程序主要处理字符串 hello[]。注意当使用字符串函数的时候，表中 hello[] 的内容变化的过程。

注意本程序所用的函数在表格中按照字母排序，除了最后一个函数 strtok。

表 7-3 标准字符串函数

函数名、例子和要求的头文件	解释
A atoi ia=atoi("-123.45xyz"); #include<stdlib.h>	将下面格式的字符串转换成一个整数值： "whitespace sign digits" 如果输入不能被转换，返回 0。如果溢出了，那么返回值未定义。例子中只是把字符 -123 转换成了 int。因此，ia 等于 -123。注意：如果在符号位之前有任何的非空白符，那么不发生转换
B atof da=atof("-987.65E+01pqr"); #include<stdlib.h>	将下面格式的字符串转换成一个 double： "whitespace sign digits.digits d|D|d|E digits" 如果输入不能被转换，返回 0.0。如果溢出了，那么返回值未定义。例子中除了 pqr 以外把所有字符转换成了 double。于是，da 等于 -987.65E+01。注意：如果在符号位之前有任何的非空白符，那么不发生转换
C atol la=atol("-456.89abc"); #include<stdlib.h>	将下面格式的字符串转换成一个 long int： "whitespace sign digits" 如果输入不能被转换，返回 0。如果溢出了，那么返回值未定义。例子中把 -456 转换成了 long。于是，la 等于 -456。注意：如果在符号位之前有任何的非空白符，那么不发生转换
D strcat pa=strcat(hello, "morning!"); #include<string.h>	将第二个字符串的拷贝"morning!"追加到用 hello 地址描述的第一个字符串后面，并且返回指向已经被连接（或追加）了内容的第一个字符的指针。"morning!"中的第一个字符覆盖了 hello 后面的空字符。数组 hello[] 必须声明为足够大以便能容纳追加过来的"morning!"。注意：执行这段代码之前，hello[] 里的内容是"Good"，执行完这段代码后，hello[] 里面的内容是"Good morning!"
E strchr pb=strchr(hello,'m'); The string hello[] is, "Good morning!" #include<string.h>	在用指针 hello 指向的字符串中查找给定的字符'm'。返回一个指针赋给 pb，指向字符串中第一个'm'的位置。如果没有找到'm'，那么返回一个空指针。'm'在字符串中的位置可用 pos = pb-hello+1 语句计算得到。其中 pos 可以转换为 int，代表'm'在字符串中的位置

（续）

函数名、例子和要求的头文件	解释
F strcpy ia=strcmp(hello, "Good xyz"); The string hello[ ] is, "Good morning!" #include<string.h>	将第一个 hello[] 字符串和第二个字符串“Good xyz”按照词典模式比较。返回一个 int 值，赋给 ia，如下： ia<0，第一个字符串小于第二个字符串 ia=0，第一个字符串等于第二个字符串 ia>0，第一个字符串大于第二个字符串 在本例中，两个字符串的前 5 个字符是相等的，但是第 6 个字符是不同的，字符分别为‘m’和‘x’，它们的 ASCII 值分别为 109 和 120。返回值用以下方式计算： ia=‘m’－‘x’＝109－120＝－11 代表字符串 hello[]（其中是“Good morning!”）比字符串“Good xyz”小。按照字母排序的话，“Good morning!”出现在“Good xyz”前面
G strcpy pa=strcpy(hello, "Good morning!"); After execution, the string hello[ ] is,"Good morning!" #include<string.h>	将第二个字符串拷贝到为第一个字符串 hello[] 分配的内存单元中，然后返回一个指针 pa，指向第一个字符串。注意，本课程序没有使用这个函数，用法见课程 7.3
H strcspn pos = strcspn(hello, "dog"); The string hello[ ] is, "Good morning!" #include<string.h>	找出在第一个字符串 hello[]（“Good morning!”）中发现第一次出现的第二个字符串包含的任何字符。返回第一个字符串中的一个位置（与第一个字符的相对位置），这个位置是第二个字符串包含的一个字符，且这个字符在第一个字符串中又第一次出现。例如第二个字符串中的两个字符‘d’和‘o’，在第一个字符串中出现了。因为字符‘o’的位置出现在字符‘d’的前面，那么返回的值 pos 就是字符‘o’在第一个字符串中第一次出现的位置，等于 2。注意，因为空格也是一个字符，两个有空格的字符串可以以空格匹配，而不以其他字符匹配

I strerror
errno=0;
infile=fopen
("abodefgh.ijk","r");
if (errno)
{
pa=strerror(errno);
printf("%s=n",pa);
}
 	
#include<string.h> for
strerror
#include<errno.h> for full
use of errno
	这个函数经常和 errno（在很多的 C 语言编译器中用一个全局变量来实现，这意味着它可以不通过函数的参数列表而被函数使用）配合使用。errno.h 文件中包含了它的相关信息。全局变量 errno 是为数不多的用在 C 库函数中的全局变量。不要把你的变量命名为 errno！当函数检测到错误的时候，C 库函数会把一个非 0 的整数值赋给 errno（通常由传入不恰当的参数造成）。当把这个整数值传入函数 strerror 的时候，它会确定一个字符串的地址，这个字符串描述由函数设定的 errno 的具体的内容。所以 strerror 和 errno 通常配合使用，虽然并不要求怎么做。为了使用 errno，首先设置 errno 为 0（代表当前没有错误），然后调用一个 C 库函数，如果 C 库函数检测到了错误，它会把一个非零的值赋给 errno。如果 errno 的值非零，那么可以用 errno 的值去调用 strerror，函数 strerror 返回一个字符串的地址，这个字符串是 errno 的文字描述。本列中 errno 被设为 0，然后 fopen 函数被调用以打开一个不存在的文件以便读取。因为这种错误，fopen 把 errno 设为一个非零的值，因为 errno 是全局变量，它可以被程序存取。如果 if 语句检测到 errno 的非零，那么程序用 errno 的值调用 strerror。函数 strerror 函数返回一个字符串的地址，这个字符串是 errno 的文字描述，然后就可以输出这个文字描述了。并非每个编译器可能实现的细节不一样，这代表着对应于 errno=2 的文字描述，编译器之间的定义可能并不相同。另外 errno 也可以不用全局变量实现，它可以是一个宏。但是，如果你使用这里描述的 errno 和 strerror，你的程序在编译器之间是可移植的

（续）

函数名、例子和要求的头文件	解释

J strlen
`len=strlen(hello);`
The string hello[ ] is,
"Good morning!"
`#include<string.h>`
	返回字符串 hello[ ]（内容是"Good morning!"）的字节数。长度并不包含字符串结束符，本例中 len = 13

K strncat
`pb=strncat(hello," John. How are you!",5);`
`#include<string.h>`
	把第二个字符串的前 5 个字符连接到第一个字符串 hello[ ] 上。第二个字符串的第一个字符覆盖掉第一个字符串末尾的空白字符。第二个字符串中的空字符不会被拷贝，但是一个空字符会被加到已经发生连接的字符串的末尾。这意味着连接以后的字符串不会在末尾有两个空字符。第一个字符串的声明尺寸必须要能够容纳第二个字符串要连接过来的内容。返回一个指针 pb，指向新的连接后的字符串 hello[ ]。注意在执行这段代码前，hello[ ] 是"Good morning!"执行完以后这段代码以后，字符串 hello[ ] 是"Good morning! John."。

L strncmp
`ib=strncmp(hello,"Good car",15);`
The string hello[ ] is,
"Good morning! John."
`#include<string.h>`
	将第一个字符串 hello[ ] 的前 15 个字符和第二个字符串"Good car"按照词典模式比较。返回一个 int 值，赋给 ib，如下：
	ib<0，第一个字符串小于第二个字符串
	ib = 0，第一个字符串等于第二个字符串
	ib>0，第一个字符串大于第二个字符串
	如图 7-9 所示，在本例中，两个字符串均前 5 个（小于 15）字符是相等的，但是第 6 个字符是不同的，字符分别为'm'和'c'，它们的 ASCII 值分别为 109 和 99。返回值用以下方式计算：
	ia = 'm' − 'a' = 109−99 = 10
	这代表字符串 hello[ ]（"Good morning!"）比字符串"Good car"要大

M strncpy
`pa=strncpy(hello+14, "Linda. How are you!", 6);`
`#include<string.h>`
	从第二个字符串拷贝前 6 个字符"Linda."到第一个字符串用 hello+14 来代表的地址上。表达式 hello+14 代表从第一个字符开始后的第 14 个字符。这种加法（利用指针）在本节后面详细讨论。这个函数返回一个指针 pa，它指向第一个字符串。在本例中，第一个字符串中从第 14 个位置开始，被字符串"Linda"代替，替换完毕后，在新的字符串 hello[ ] 的末尾加上一个空字符。注意，在执行这段代码前，hello[ ] 是"Good morning! John."。执行完以后这段代码以后，字符串 hello[ ] 是"Good morning! Linda."

N strpbrk
`pb=strpbrk(hello,"dear");`
The string hello[ ] is,
"Good morning! Linda."
`#include<string.h>`
	扫描第一个字符串 hello[ ]，并确定它是否包含第二个字符串的任何字符。如果发现匹配，返回一个指针（赋值给 pb），指向本第一个字符串中第一个匹配的字符的位置。例如，第二个字符串为"dear"，包含字符'd'，它也是第一个字符串"Good morning! Linda."的一个部分。指针 pb 被赋予了第一个字符串中"Good"那个 d 的地址。如果没有匹配，那么返回一个空指针

O strrchr
`pa=strrchr(hello,'o');`
The string hello[ ] is,
"Good morning! Linda."
`#include<string.h>`
	扫描第一个字符串 hello[ ]（"Good morning! Linda."），确定特定字符'o'的最后一个出现的位置。如果发现匹配，返回一个指针（赋值给 pa），指向在第一个字符串中最后一个匹配的字符的位置。例如，第一个字符串包含三个'o'。指针指向最后一个'o'（在"morning"中）。如果没有匹配，那么返回一个空指针

P strspn
`ia=strspn("oQdo year",hello);`
The string hello[ ] is,
"Good morning! Linda."
`#include<string.h>`
	返回第一个字符串的一部分的长度，这一部分只包含第二个字符串中给定的字符。例如，第一个字符串 hello[ ]（"Good morning! Linda."）中的前 5 个字符出现在第二个字符串中；但是出现在第一个字符串中的字符'y'并没有出现在第二个字符串中，所以，ia= 5

（续）

函数名、例子和要求的头文件	解释

Q strstr
`pa=strstr(hello,"Linda");`
The string hello[ ] is.
"Good morning! Linda."
`#include<string.h>`
	在第一个字符串 hello[]（"Good morning! Linda."）中找到第二个字符串（不包含空字符），返回一个指针，赋值给 pa，指向第二个字符串在第一个字符串中的出现位置。本例中两个字符串都包含"Linda."，这样，pa 指向第一个字符串中'L'的位置。如果字符串没有发现，返回一个空指针

R strtod
`da=strtod("123.45abc",&pb);`
`#include<stdlib.h>`
	将一个字符串（必须以一个非空白字符开始，第一个字符是正负号、数字或者小数点）转换为一个 double 值。函数 strtod 和 atof 的不同之处在于 strtod 的参数包含一个指针变量 &pb，pb 的值被函数 strtod 所设定，指向它在字符串中停止扫描的位置，如果没有执行转换函数返回 0。例如，函数不会扫描'abc'，因为它们在字符串的数字部分的后面。这样，pb 等于'abc'中'a'的地址，da 的值为 123.45

S strtol
`la=strtol("98765xyz",&pa,10);`
`#include<stdlib.h>`
	将一个字符串（必须以一个非空白字符开始，第一个字符是正负号、数字或者小数点）转换为一个 long 值。函数 strtol 和 atol 的不同之处在于 strtol 提供一个转换字符串所用的基数（本例中为 10）。另外，参数包含一个指针变量 &pa，pa 的值被函数 strtol 所设定，指向它在字符串中停止扫描的位置。如果没有执行转换函数返回 0。例如，函数不会扫描'xyz'，因为它们在字符串的数字部分的后面。这样，pb 等于'xyz'中'x'的地址，la 的值为 98765

T strtoul
`ula=strtoul("45678pqr",&pc, 10);`
`#include<stdlib.h>`
	将一个字符串（必须以一个非空白字符开始，第一个字符是正负号、数字或者小数点）转换为一个无符号整型值。函数提供一个转换字符串所用的基数（本例中为 10）。另外，参数包含一个指针变量 &pc，pc 的值被函数所设定，指向它在字符串中停止扫描的位置。如果没有执行转换函数返回 0。例如，函数不会扫描'pqr'，因为它们在字符串的数字部分的后面。这样 pc 等于'pqr'中'p'的地址。注意 pc 的用法和 strtod 中的 pa 和 strtol 中的 pb 的用法有些不同，不过它们完成相同的功能

U strtok
`pa=strtok(hello, token_separator);`
`while (pa!=NULL)`
`{`
`printf("%10s\n",pa);`
`pa=strtok(NULL, token_separator);`
`}`
The string hello[ ] is."Good morning! Linda."
Note:The declaration for token_separator is `token_separator[]=",\n \t...";`
Therefore, strtok examines the string hello for any character in the preceding string.
#include<stdlib.h>for NULL
#include<string.h> for strtok
	将一个字符串分解为很多小字符串片以空字符为分界。方法如下：它在第一个字符串中查找第一个没有出现在 token_separator[ ]字符串中的字符，并给这个字符定义了一个记号。这是第一个记号。然后它开始查找第一个出现在 token_separator[ ]字符串中的字符（叫做分界符）。将这个分界符用空字符代替。它返回一个指针（赋值给 pa）指向第一个记号（在第一个分界符发现之前的字符串的开始位置）。函数在内部保存指向分界符后面的字符的指针。为了持续将分界符代替为空白符，我们必须后续调用 strtok。但是不是以感兴趣的字符串（本例中的 hello[ ]）而是以 NULL（空指针）作为参数。strtok 会使用内部保存的指针作为搜索下一个分界符开始的位置。当执行完所示的语句后 hello[ ]变成了：
	`Good\0morning\0Linda\0`
	如果 strtok 函数到达了 hello 字符串的末尾，那么它返回一个空指针。这就是我们使用校验表达式（pa!=NULL）的原因

你不需要记住这个表格，但是要阅读整个表格以便对函数的功能有一个大概的了解。对

于描述的例子要重点关注。这些描述会让你知道编程的时候，哪些功能是可用的。

当你想用这些函数时，可以参考这个表格以及本课程序中的这些函数的使用。有了这些信息，你可以在自己的程序中使用这些函数了。

我们在应用练习部分会使用这些函数，帮助你进一步地认识它们。

图 7-9 用 strncmp 比较字符串

2）可以按照用法给这些函数分类吗？可以，表 7-4 将这些函数基于用法分类。

表 7-4 操作及其函数

操作的类型	函数名	注释
将字符串转换为数值	atoi	返回一个 int
	atof	返回一个 float
	atoll	返回一个 long
	strtod	返回一个 double，传递结束数字部分的地址
	strtol	返回一个 long，传递结束数字部分的地址
	strtoul	返回一个 unsigned long，传递结束数字部分的地址
把一个字符串或字符串的一部分拷贝到另外一个字符分配的内存单元中	strcat	将第二个字符串拷贝到第一个字符串的末端，返回第一个字符串的地址
	strcpy	将第二个字符串拷贝到第一个字符串的开头，返回第一个字符串的地址
	strncpy	将第二个字符串中特定数目的字符拷贝到第一个字符串的开头，返回第一个字符串的地址
	strtok	在第一个字符串中将出在第二个字符串中出现的字符变为空字符，然后返回空字符位置的地址
在一给定的字符串中，找一特定字符，字符串或字符串的一部分的地址或者位置	strchr（地址）	在一给定的字符串中发现一特定字符，返回这一字符第一次出现的地址
	strcspn（位置）	在第一个字符串中发现在第二个字符串中的任何字符第一次出现的位置，返回这一字符在第一个字符串中第一次出现的位置
	strpbrk（地址）	在第一个字符串中发现在第二个字符串中的任何字符第一次出现的位置，返回这一字符在第一个字符串中第一次出现的地址
	strrchr（地址）	在第一个字符串中发现在第二个字符串中的任何字符最后一次出现的位置，返回这一字符在第一个字符串中出现的地址
	strspn（位置）	在第一个字符串中发现没有在第二个字符串出现的第一个字符，返回这一字符在第一个字符串中的位置
	strstr（地址）	发现第二个字符串在第一个字符串中出现的位置，返回在第一个字符串中出现的第二个字符串的开始位置

（续）

操作的类型	函数名	注释
比较两个字符串	strcmp	按词典顺序比较两个字符串，用 int 来代表哪个字符串更大
	strncmp	按词典顺序比较两个字符串中的特定数目字符，用 int 来代表哪个字符串更大
计算字符串长度	strlen	返回字符串中字符的数目（不包含空字符）
给出描述错误的字符串的地址	strerror	返回描述错误的字符串的地址

3）如何翻译标准 C 库函数的原型？我们现在已经有了足够的知识，可以深入查看 C 库标准函数中每一个函数的原型并能正确的翻译它们了。我们将为你演示这些内容，因为深入学习 C 语言时你会更将频繁地遇到它们。当你第一次看到这些内容时，可能会感到有点奇怪。所以在表 7-5 中讲解了一些例子。一旦你理解这些例子，就可以翻译更多的函数原型。在表格的左边查看函数的原型，然后阅读表格右边的解释，必要的时候可以参考表格的左边。

表 7-5 函数原型

函数原型（ANSI C 标准）	解释
`int puts(const char *s)`	括号中给定的一个参数是“s”，它代表一个地址，因为前面带有 * 号，这意味着当我们调用 puts 的时候，必须传递一个地址。char 代表必须是一个 char 类型的地址。const 是一个修饰符，代表函数不能修改该地址“s”指出的字符串的内容。函数前面的 int 代表 puts 返回一个 int 类型的值（回忆 puts 返回一个非负值代表没有错误，否则返回 EOF）。函数原型中的“s”本身没有意义，它只是 ANSI C 中的一个典型的标识符，用来代表一个字符串，因此，这个函数原型告诉我们，可以按以下方式使用这个函数： `integer_variable = puts(address_of_char);`
`double sin (double x)`	因为参数 x 的周围没有 * 号也没有括号，必须把一个值传递给 sin 函数。括号中的 double 代表必须把一个 double 类型的值传递给 sin。在 sin 函数前面的 double 代表 sin 返回一个 double 类型的值。函数原型中的“x”本身没有意义，它只是 ANSI C 中的一个典型的标识符，用来代表一个 double 类型值。从这个原型中，我们可以用以下方式使用 sin 函数： `double_variable = sin(double_value);`
`char *strcpy(char *s1, const char *s2)`	这个函数原型代表两个参数 s1 和 s2。这两个都是地址，因为它们前面都有 * 号。它们都是 char 类型的地址，因为它们前面都有 char。第二个参数前面有 const 修饰符，代表着函数不能修改参数 s2 传递进来的字符串。在函数前面的 * 代表这个函数返回一个地址，char 代表这是个 char 类型的地址。s1 和 s2 本身没有意义，只是 ANSI C 中的典型的标识符。这个函数原型代表我们可以用以下方式使用这个函数： `char_pointer_variable = strcpy(address_of_char, address_of_char);`
`double atof(const char *nptr)`	参数是 nptr，这是一个用 char * 代表的 char 类型的地址。const 代表函数 atof 不能修改 nptr 传递进来的字符串的内容。nptr 本身没有意义，它只是 ANSI C 中的一个典型的标识符，有时用来代表一个指针变量。double 代表函数返回一个 double 值。我们可以用下面的方式使用 atof： `double_variable = atof(address_of_char);`

4）两种不同版本的相同功能的字符串函数，例如 strcpy 和 strncpy。它们的主要区别是什么？带有 n 的版本对应一个安全的版本。以 strcpy 为例。strcpy 会从原字符串中拷贝所有的字符到目标字符串，直到发现一个 NULL 字符。但是在拷贝之前，strcpy 并不检查在目标字符串中是否有足够的空间，这样如果原字符串非常长，那么 strcpy 的执行就会覆盖程序的某些区域，也许是其他的变量。事实上，这会造成缓存溢出，这种现象有的时候会被一些病

毒程序利用来威胁你的程序的安全性。为了防止以上的问题发生，我们提供了一个更安全的版本，它提供了一个参数来指定每次传递字符的最大数目。你应该使用这个安全的版本以防止内存安全的漏洞。

注意，函数原型本身并没有给出有效利用函数的全部信息。在本书前面描述了使用过的所有函数，所以现在不详细介绍它们了。为了使用其他的 ANSI C 标准函数，你应该去参考它们的原型和具体描述。

概念回顾

1）表 7-3 显示了大部分常用的字符串函数。

2）带有“n”的字符串函数代表着安全版本。不安全的版本在拷贝前并不检查目标地址是否有足够空间，但是安全版本设置一个上限以限制传递的字符 / 元素的个数。

练习

1. 写一个程序读入本课的源代码（文件 L7_8.C）然后完成以下：

   a. 查找在代码中用了多少标记。标记的分隔符如下面的字符所示：

   `"" [] () {}*,;=/...#\n\t\` and space

   b. 查找在代码中出现了多少次字符串常量“Good”。

   c. 将含有数字的字符串转换为 double。

2. 对于表 7-4 中的每一个函数，开发一个函数包含同样的参数，返回同样的值。例如函数

```
pb=strncat(hello," John. How are you!",5);
```

   有三个参数。函数将第二个字符串中前 5 个字符拷贝到第一个字符串 hello[]。函数返回一个指针，赋值给 pb，指向新的连接后的字符串 hello[]。针对这个函数，开发一个叫做 M_strncat( ) 的函数，其中 M 代表 mine。用同样的参数实现这两个函数：

```
pc=M_strncat(hello," John. How are you!",5);
```

   指针 pb 和 pc 指向相同的对象。

3. 给定课程 7.8 的源代码（文件 L7_8.C），写一个程序显示程序中使用的任何字符以及这个字符的频度。按以下格式输出：

```
Input file ----- L7_8.C
--------------------------------
Character                    number of occurrences
a                                   ??
b                                   ??
...
```

4. 给定课程 7.8 的源代码（文件 L7_8.C），使用 ANSI C 字符类型函数写一个程序来确定每一个字符的类型以及它们的频度，按以下格式输出：

```
Input file ----- L7_8.C
-----------------------------------
Character type       number of occurrences
Alphanumeric                        ??
Alphabetic                          ??
...
```

5. 写一个程序将下面格式的字符串转换 int 类型的值。然后使用 atoi 函数检查转换是否正确。

```
whitespace  sign  digits
```

6. 写一个你自己版本的 strchr() 函数。但是函数返回发现字符的位置。如果没有字符被发现，函数返回 -999。使用 strchr 函数检查你的输出。

课程 7.9 指针符号与数组符号

主题

- 使用指针符号来存取数组元素和字符串

我们已经暗示但没有演示以下事实：C 允许数组元素被数组符号或者指针符号存储。本课将演示它们。

源代码

```
#include <stdio.h>
void main(void)
{
        char aa[35]={"This is a one-dimensional array"};
        char bb[5][40]={"We can","use both","array and pointer",
                        "notation to access","one and two-dimensional arrays"};
        char cc[2][3][20]={"A three ","dimensional ","array ","is ","shown ","also"};
        char *dd;

        printf("***************** Section 1  1-D array ****************\n");
        putchar(aa[0]);
        putchar(*aa);
        putchar('\n');

        putchar(aa[16]);
        putchar(*(aa+16));
        putchar ('\n');

        printf("***************** Section 2  2-D array ****************\n");

        putchar(bb[3][5]);
        putchar(*(*(bb+3)+5));
        putchar('\n');

        puts(*(bb+2));

        dd=&bb[0][0];
        putchar(*(dd+125));
        putchar('\n');

        printf("***************** Section 3  3-D array ****************\n");

        putchar(cc[1][2][3]);
        putchar(*(*(*(cc+1)+2)+3));
        putchar('\n');

        puts(*(*(cc+1)+2));

        dd=&cc[0][0][0];
        putchar(*(dd+80));
        putchar('\n');
        puts(dd+80);
```

一维数组

二维数组

二维数组

存取一维数组中单个字符的数组符号

存取一维数组中单个字符的指针符号

存取一维数组中单个字符的数组符号

存取一维数组中单个字符的指针符号

存取二维数组中单个字符的数组符号

存取二维数组中单个字符的指针符号

存取二维数组中单个字符的指针符号

如果把数组的开始地址赋给一个单独的指针变量，可以用带有 * 号的指针符号来存取二维数组中的单个元素。可以这么做是因为使用单个指针变量，我们可以用不同的指针代数运算。注意 dd=bb 不会工作

存取三维数组中单个字符的数组符号

存取三维数组中单个字符的指针符号

对于三维数组，带有两个 * 的指针符号代表着字符串的开始地址

如果把数组的开始地址赋给一个单独的指针变量，我们可以用带有 * 号的指针符号来存取一维数组中的单个元素。注意 dd=cc 不会工作

利用一个指针变量的代数运算，可以不用 * 号来存取三维数组中的字符串

```
    printf("Address of cc[0][0][0]=%p, cc+1=%p, *cc+1=%p, **cc+1=%p\n",
    &cc[0][0][0],cc+1,*cc+1,**cc+1);
}
```

&cc[0][0][0]、cc、*cc 和 **cc 都代表 cc[][][] 中第一个元素的地址，但是今 cc、*cc 和 **cc 上加 1 进行的指针代数运算会产生不同的结果

输出

```
*************** Section 1  1-D array ****************
tt
nn
*************** Section 2  2-D array ****************
ii
array and pointer
i
*************** Section 3  3-D array ****************
oo
also
o
shown
Address of cc[0][0][0]=FE32, cc+1=FECH, *cc+1=FEA6, **cc+1=FE93
```

解释

1）如何使用指针符号存取一维数组中的元素？对于一维字符数组 aa[ ]，我们可以使用数组符号或者指针符号来存取一个元素。为了存取数组的第一个元素 (aa[0])，可以使用 *aa。为了存取 aa[16] 的元素，可以使用 *(aa+16)。因此，本课中的两个表达式：

```
putchar(aa[16]);
putchar(*(aa+16));
```

是等价的。

2）指针符号后面的逻辑是什么？这个符号涉及指针的代数运算，即将一个整数加上一个地址。C 语言在执行这个操作时首先注意到这是一个地址类型，例如表达式

```
aa+16
```

aa 是一维字符数组的首地址。因为字符占据一个字节的内存，并且我们现在使用的是一维数组，C 把 16 个字节加到了 aa 代表的地址上。这样，这个表达式给出了 aa[16]，也就是第 17 个字符的地址。利用单目运算符 *，表达式 *（aa+16）给出了第 17 个字符的值。这个表达式可以用作 putchar 函数的参数。

3）如何存取二维数组中的元素？本课的程序中，我们使用二维字符数组 bb[5][40]。我们可以使用数组符号或者指针符号存取它的元素，如下：

```
putchar(bb[3][5]);
putchar(*(*(bb+3)+5));
```

将输出 bb 中的同一个字符。

第二个表达式再一次使用了指针代数运算。我们首先从最内层的括号开始描述。为了完成这个加法，C 语言需要确定地址的类型。在处理多维数组时，C 语言使用了一些技巧，它把二维数组当成了一个数组类型的数组。换句话说，本课程序中的二维数组 bb[][] 被认为

是5个尺寸为40的一维数组（因为bb声明为bb[5][40]）。这样bb代表的地址就是首行的地址。因此，一个单独元素的地址尺寸就是40而不是1。这样，当我们将bb加上3以后，我们其实是加了3×40=120个字节在bb代表的地址上。

在二维数组上，单目运算符*也工作得不太一样。表达式：

```
*(bb+3)
```

代表的是第4行的开始地址（不是值，像我们利用单个指针变量在一维数组上时）。所以表达式

```
*(bb+3)+5
```

是将*(bb+3)加上5。同理，C必须在执行加法操作之前考虑地址的类型（必须在声明的时候给定）。因为它是二维数组并且我们用*运算符给出了行的开始地址，整数使得这个地址又加上了5，这时我们得到了在bb[][]数组中单个字符'i'的地址。*在这个表达式上使用单目操作符给出

```
*(*(bb+3)+5)
```

得到字符的值。因此

```
putchar(*(*(bb+3)+5));
```

输出字符i。注意，为了在二维数组中使用指针符号来存取单个字符，我们需要使用两个*号。为了在三维数组中使用指针符号来存取单个字符，需要使用三个*号。

4）对于二维数组bb[][]，*(bb+2)代表什么？它等同于bb[2]，代表bb[][]中第三行的首地址。它可以当成puts函数的一个可接受的参数。并且语句

```
puts(*(bb+2));
```

输出字符串"array and pointer"。

5）对于声明char *dd，dd+125代表什么？变量dd代表一个地址，又因为dd是一个简单的char类型的指针变量，"125"使得125字节加到了dd代表的地址上。另外，因为dd是一个简单的指针变量，所以只需要一个单目运算符*来存取值。因此，

```
dd=&bb[0][0];
putchar(*(dd+125));
```

输出字符i，也就是bb[][]数组的第126个字符。

记住，为了确定第126个字符，我们必须使用声明时的尺寸bb[5][40]，而不是声明时所使用的字符串。换句话说，第126个字符是第4行的第6个字符，也就是i。

6）本课程序中，我们能使用dd=bb语句来代替dd=&bb[0][0]语句吗？不可以，因为C会考虑类型是否匹配。即使bb是一个地址，而dd是一个指针，bb是一个二维数组的地址。因为dd被声明为简单的字符类型的指针，它不能接受一个数组类型的地址。我们将&取值操作符，用在单个的bb[0][0]元素上，这样就得到了一个dd可以接受的地址类型。

7）putchar(*(bb+125));会输出bb[][]中的第126个字符吗？不会，语句不会编译通过，因为如果使用bb那么就意味着我们必须使用两个单目运算符*来获得单个的字符。另外，指针代数运算也不会让我们前进125个字节，而是125×40=5000个字节！所以，即使语句被编译通过，它也指向了一个错误的位置。

8）如何存取三维数组中的元素？本课程序中，我们使用了三维字符数组cc[2][3][20]。

我们可以使用数组符号或者指针符号来存取一个元素。例如，

```
putchar(cc[1][2][3]);
putchar(*(*(*(cc+1)+2)+3));
```

输出 cc[][][] 中的同一个字符。

第二个表达式利用指针代数运算，从最内层的一对括号开始解释。同理，为了执行这个运算，C 确定地址的类型。因为 C 认为三维数组是数组的数组的数组，cc[][][]（声明为 cc[2][3][20]）是 2 个尺寸为 [3][20] 的两维数组。cc 被当作一个两维数组的首地址。于是，这个地址的单个元素就有 3×20=60 的尺寸，而不是 1。当将 cc 加上 1 的时候，我们就加上了 1×3×20=60 字节到 cc 所代表的地址上。

同理，因为我们利用的是多维数组，一个单独的 * 运算符不代表一个值，而是代表一个地址，这样，表达式

```
*(cc+1)
```

代表的是第二个二维数组的首地址。表达式

```
*(cc+1)+2
```

是另外一个地址。同理，C 必须在加法之前考虑地址的类型（在声明的时候指定）。因为它是一个三维字符数组并且地址是通过使用 * 符号得到下一个二维数组的首地址，偏数 2 使得再加上 2×20=40 个字节。表达式

```
*(*(cc+1)+2)
```

代表的是第二个二维数组（因为我们有 +1）中第三行的首地址（因为我们有 +2）。我们用表达式

```
*(*(cc+1)+2)+3
```

将这个地址加 3。因为它是一个三维字符数组并且我们使用了两次 * 符号，3 代表着加上 3 个字节到地址 *(*(cc+1)+2) 上。使用另外一个 * 符号，我们得到

```
*(*(*(cc+1)+2)+3)
```

因为这是一个二维数组，并且我们使用了三个 * 符号。表达式代表一个单个的字符，并且可以用在 putchar 函数中，

```
putchar(*(*(*(cc+1)+2)+3));
```

并输出一个单个的字符。

9）对于三维数组 cc[][][]，*(*(cc+1)+2) 代表什么？我们已经说过，它代表的是第二个（因为 +1）二维数组的第三行（因为 +2）。我们可以用 puts 来输出这行代表的地址。语句

```
puts(*(*(cc+1)+2));
```

打印这一行。

10）如何使用 putchar 函数以及一个单目运算符 * 来打印 cc[][][] 数组中的一个字符？把 cc[][][] 第一个字符的地址赋给一个指针变量 dd，就能使用一个单目运算符 * 来存取 cc[][][] 数组中的单个值了。例如，语句

```
dd=&cc[0][0][0];
putchar(*(dd+80));
```

使得第 81 个字符被输出。使用 cc[2][3][20] 声明的尺寸，第 81 个字符可以被认为是第二个 3×20 的二维数组的第 21 个字符。因为一行有 20 个字符，第 21 个字符是第二行的第一个字符，是 's'。

11）如何使用 dd 和 puts 输出 cc[][][] 数组中单独的行？因为 dd 代表一个字符的地址，为了输出单独的行，我们必须将 dd 加上正确 数量的字节数以便得到我们感兴趣的行。例如，每一行的长度是 20，下列语句

```
puts(dd+80);
```

会跳过前 4 行，把第 5 行的字符串 "shown" 输出。

12）cc、*cc、**cc 分别代表什么？这三个都代表 cc[][][] 第一个元素的地址。但是请记住，当你在每个地址加上一个整数的时候，我们会得到不同的结果：

```
  cc+1 = 首字符地址 +60 字节
 *cc+1 = 首字符地址 +20 字节
**cc+1 = 首字符地址 +1 字节
```

本课的程序中，这得到了

```
cc[0][0][0]=FF92, cc+1=FECE, *cc+1=FEA6, **cc+1=FF93
```

如果你运行这个程序，可能会得到不同的结果，但是各个地址间的关系是一致的。

13）本课的程序中，用什么样的可视图来代表用在 bb[][] 和 cc[][][] 数组上的指针符号？图 7-10 演示了指针符号。在这个图中，我们仅仅显示了每一行的第一个字符。就像其他的这类图一样，其他的字符被放到每一行首字符的后面。

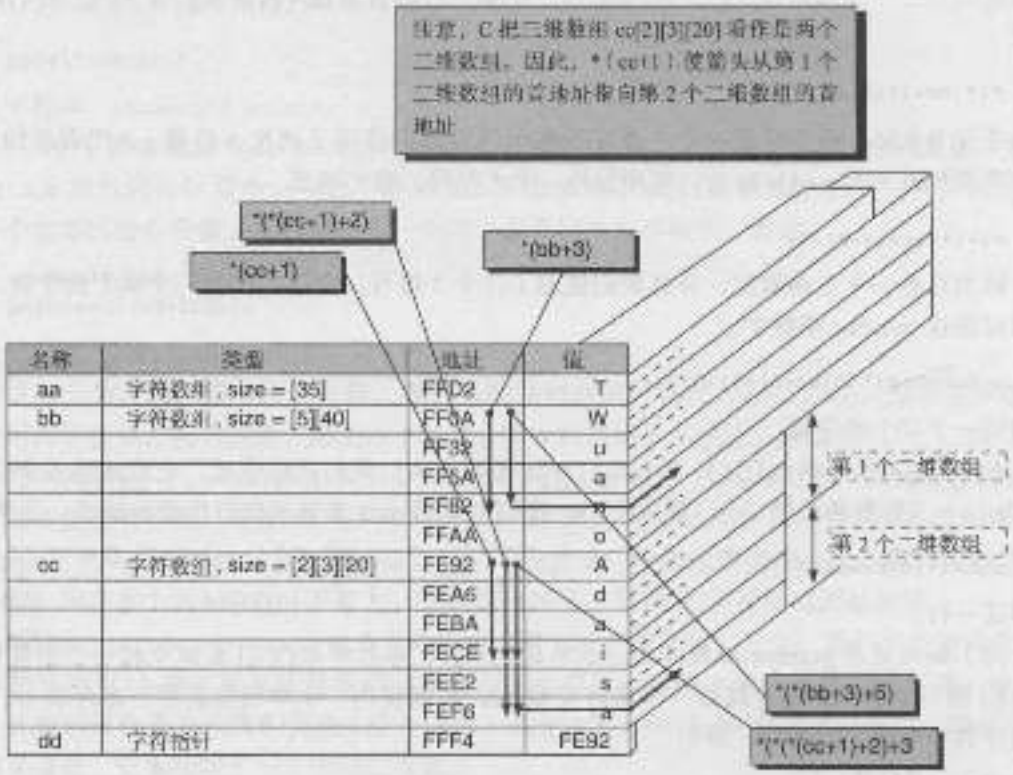


图 7-10 本课程序中的指针符号

扩展解释

1）我们能使用指针符号的其他方式来存取数组元素吗？是的，你可以混合使用数组符号和指针符号。例如，*(*(bb+3)+5) 可以写成 *(*(bb[3]+5)。这是有效的，因为 *(bb+3) 和 bb[3] 在意义上是等价的。本课不会使用这种特定的符号，但是你也许会在别的书中看到它们。

2）为什么我们使用这么多不同的符号？很多 C 编译器会把数组符号在编译的时候转换成指针符号。这样，即使你已经写了数组符号，在目标代码中也会变为指针符号。大多数情况下，使用数组符号或者指针符号对程序的运行速度没有什么影响。

3）如果我们不使用 char，那么指针的代数运算是如何进行的？我们必须意识到，其他类型的单个元素占据多余 1 个字节的内存。回忆以前讲过的常见的数据类型占据的内存空间。ANSI C 将 char 定义为 1 个字节，其他的类型并没有规定。但是，int 通常是 4 个字节，float 是 4 个字节，double 是 8 个字节。如果有下面的一维数组

```
int aa[50];
float bb[100];
double cc[70];
```

指针运算如下：

声明	每个元素的字节数	操作	行为
int aa[50]	4	aa + 20	加 (4 B × 20) = 80 B
float bb[100]	4	bb + 12	加 (4 B × 12) = 48 B
double cc[70]	8	cc + 9	加 (8 B × 9) = 72 B

因为 C 在执行指针代数运算的时候使用数组类型，所以指针符号和数组符号之间有一个对应关系，即 int、float、double 和其他数据类型。例如，下面是等价的。

指针记号	数组记号
*(aa + 20)	aa[20]
*(bb + 12)	bb[12]
*(cc + 9)	cc[9]

如果使用 int、float、double 类型的多维数组，我们描述过的指针符号原则也同样适用。因为 C 语言把多维数组当成数组的数组。

记住，为了确定指针符号的意义，你必须要查看它的声明以理解 C 编译器如何利用这个声明去翻译指针符号的意义。如果不查看声明，那么你不会理解符号代表的是什么。

这在相反的条件下也适用。因为 C 能够执行正确的指针代数运算，那么它一定有正确的声明。这一点在处理用户定义函数的时候非常重要，因为函数的定义必须是正确的以便 C 能够在函数体内执行指针的代数运算。幸运的是，C 在函数的定义不正确的时候会指出错误。但是通过理解指针代数运算，我们理解 C 如何使用函数中的命令。下一课将讨论用户定义函数的话题。

4）如果 *(dd+2) 用在了本课程序的末尾，它是否代表字符 'c'（在 dd[][] 数组中的第三个字符）？ *(dd+2) 的含义是什么？如果在计算 dd 的时候没有括号，dd 代表字符 'A'，因为 C 语言把字符当成一个整数，在 'A' 上加 2 得到 ASCII 字符集中的 'C'。从这里我们可以看出括号在指针符号中的重要性。

概念回顾

1）C允许你混合使用数组符号和指针符号。例如，在一维数组 char aa[20] 中：

```
aa[2]
*(aa+2)
```

是等价的。第二个符号中，指针代数运算将一个整数加到了一个地址上。C执行这个操作的时候首先确定地址的类型，然后根据这个计算地址的偏移量。

2）我们可以通过数组符号和指针符号存取二维数组 bb[5][10] 的元素。例如，

```
putchar(bb[3][5]);
putchar(*(*(bb+3)+5));
```

第二个语句首先通过 (bb+3) 得到第4行的开始地址，然后在这个中间地址上加上6个偏移量以得到 bb[3][5] 的地址。

练习

1. 基于下面的声明和语句，

```
char y[4]={"321"}, z[2][4]={"CAT","DOG"}, *py, *pz;
py=&y[0];
pz=&z[1][0];
```

确定下面每一个语句的真假：

a. &y[0]和&y等代表y[0]的地址
b. &y[0]和y都代表y[0]的地址
c. &y[2]和y+2都代表y[2]的地址
d. y[0]等价于y
e. y[0]等价于py
f. y[0]等价于*py
g. *pz等价于"D"
h. *(pz+1)等价于"O"
i. *pz+1等价于"C"
j. *y+2等价于"5"
k. *(y+2)等价于"1"
l. *py+2等价于"3"
m. *(py+2)等价于"1"

2. 基于下面的声明

```
char y[4]={"321"}, z[2][4]={"CAT","DOG"}, *py, *pz;
```

指出下面语句中的错误：

```
a.py= y[1];
b.py = &y[4];
c.pz = &z[1][2];
d.*(*z+2) = *y+2;
```

3. 手工计算a、b和c的值并运行程序检验你的结果：

```
#include <stdio.h>
void main(void)
{
        char x[5]="ABCD", y[2][6]={"EFGH", "JKLM"}, *px, *py;
        char a, b, c;

        px=x;
```

```
py=&y[1][0];
a = *px+2;
b = *x + 10 ;
c = 10+ *(*(y+1));
```

4. 手工计算下列程序中 x 的值的并运行程序检查你的结果：

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

void f1 (char a[][10], char *b[], char *pa)
{

        pa=&b[1][2];
        strcpy(a[0],b[0]);
        *a[1]=*pa;
        *(*(a+1)+1)= *(pa+1);
        a[1][2]='\0';
}

void main(void)
{
        char x[10][10],*y[10]={"abcde","wxyz"}, *px;

        px=y[1];
        f1(x,y, px);
        printf("x[0]=%s\nx[1]=%s\n",x[0],x[1]);
}
```

答案

1. a. 真　b. 真　c. 真　d. 假　e. 假　f. 真　g. 真　h. 真
   i. 真，因为 *pz 是 "D" 且 C 把字符当作整数，加 1 到 "D" 给出 "E"
   j. 真，因为 *y 是 "3" 加 2 到 *y 给出 "5"
   k. 真　l. 真　m. 真
2. a. 错，因为 py 是指针，y[1] 是字符 "2"
   b. y<=3
   c. 无误
   d. 无误，语句用字符 "5" 代替 z[0][2]

课程 7.10　动态内存分配

主题

- 运行时分配内存
- 使用 calloc、malloc 和 realloc

在本章的开始提到过，如果你试图使用一个很大的固定尺寸的数组去解决一些极大数目的问题可能会有些困难。原因在于你的程序也许在一些有足够内存的系统上可以工作，但是在一些缺乏内存的系统上就不能工作了，哪怕你的程序只是处理了一小部分数据。

为了使你的程序在很多内存或很少内存的系统上都能工作，我们不应该使用固定尺寸的数组，你应该根据运行时问题的规模来分配内存。换句话说，如果你使用 char 数组存储一个或大或小的工程报告，你可以在报告很大的时候分配一个大内存，在报告很小的时候分配一个小内存。这样做，你的程序在很多内存或很少内存的系统上都能工作。但是程序在一个很少内存的系统上要求很大的内存时会失败。

本课中演示了C语言中标准函数calloc、malloc、realloc和free，用来执行动态的内存分配。另外，应用程序7.2演示了如何在特定问题中使用calloc函数。

C语言中动态内存分配使用calloc和malloc。调用时的参数指定了分配多少内存。这使得我们在调用calloc和malloc时只分配需要的内存。这些函数返回分配的内存的首地址。利用这个地址，我们可以存取内存并保存相应的信息。观察下面的程序并阅读解释环节理解动态存储的方法。

源代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main(void)
{
        char aa[19];
        char *bb[22];
        char cc[22][19]={"Example", " String 1 ", "Words"};
        int xx, yy;
        printf("*********** Section 1 - Using calloc *************\n");

        strcpy(aa,cc[0]);
        xx=strlen(aa);
        bb[0]=(char *)calloc(xx,sizeof(char));

        strcpy(bb[0],aa);
        puts(bb[0]);

        printf("*********** Section 2 - Using malloc *************\n");

        strcpy(aa,cc[1]);
        xx=strlen(aa);
        yy=xx*sizeof(char);
        bb[1]=(char *)malloc(yy);

        strcpy(bb[1],aa);
        puts(bb[1]);

        printf("*********** Section 3 - Using realloc *************\n");
        strcpy(aa,cc[2]);
        xx=strlen(aa);
        yy=xx*sizeof(char);

        bb[1]=(char *)realloc(bb[1],yy);
        strcpy(bb[1],aa);
        puts(bb[1]);
        free(bb[1]);
}
```

声明了22个字符串并初始化了3个

将cc[][]中的第一个字符串暂时保存到aa[]中

确定cc[][]第一个字符串的长度

根据cc[][]中的第一个字符串的长度分配内存，并将内存地址保存到bb[0]中，目前这个地址只是被分配了，但是内容为空

将cc[][]中的第一个字符串拷贝到bb[0]指定的地址中

将cc[][]中的第二个字符串暂时保存到aa[]中

确定cc[][]第二个字符串的长度

计算为了保存cc[][]第二个字符串需要多少字节数

为cc[][]第二个字符串分配内存，并将地址保存在bb[1]中。目前这个地址只是被分配了，但是内容为空

将cc[][]中的第二个字符串拷贝到bb[1]指定的地址中

同样地执行第二部分的前三步，但是用第三个字符串

在以前用来保存第二个字符串的地址上，根据cc[][]中的第三个字符串的长度重新分配内存（如果可能）。如果不可能，内存在其他的地方被realloc分配。分配的内存地址被保存在bb[1]中。如果生成了一个新地址，以前的内容被拷贝到新地址上

将cc[][]中的第二个字符串拷贝到bb[1]指定的地址中

释放bb[1]指定的内存

输出

```
*********** Section 1 - Using calloc *************
Example
*********** Section 2 - Using malloc *************
String 1
*********** Section 3 - Using realloc *************
Words
```

解释

1）calloc 函数做什么？函数 calloc 在程序运行的时候分配内存。分配内存的数量被传入的参数确定。调用 calloc 的格式如下：

```
calloc (number_of_elements, bytes_per_element)
```

分配内存的数量等于整数 number_of_elements 和 bytes_per_element 的乘积。例如本课程序

```
calloc(xx,sizeof(char));
```

使得 xx 个尺寸是 sizeof(char)(1 字节）的元素（在本课程序中 xx 为 8）被分配出来（它也把每一位都初始化为 0）。换句话说，它调用 calloc 分配了 8 字节的内存。

函数 calloc 返回分配内存的第一个元素的地址，例如，

```
bb[0]=(char *)calloc(xx,sizeof(char));
```

使得分配出来的内存的首地址保存在指针数组 bb[] 中的第一个元素内。转换符 (char *) 使得 calloc 返回的指针是一个指向字符的指针，与 bb[] 数组中的元素类型一致。但是当我们处理数值数组的时候，应该对 calloc 使用 (int *) 或者 (double *) 转换符。

2）函数 malloc 做什么？函数 malloc 也在程序执行的时候分配内存。分配的内存数量被传入 malloc 的参数所指定。格式如下：

```
malloc (number_of_bytes)
```

分配的内存数量等于 number_of_bytes。为了分配正确数量的内存，程序员有必要先计算 number_of_bytes 的值。例如本课程序，

```
yy=xx*sizeof(char);
```

将 xx（值等于将要保存的字符串的长度）乘以保存单个字符所需要的字节数（1 字节）。这样 yy 代表保存这个字符串所需要的字节数。然后，

```
malloc(yy);
```

使得 yy 个字节的内存被分配。函数 malloc 返回分配的内存的第一个元素的地址。例如

```
bb[1]=(char *)malloc(yy);
```

使得分配出来的内存的首地址保存在指针数组 bb[] 中的第二个元素内。转换符 (char *) 使得 malloc 返回的指针是一个指向字符的指针，与 bb[] 数组中的元素类型一致。但是当处理数值数组的时候，我们应该对 malloc 使用 (int *) 或者 (double *) 转换符。

3）函数 realloc 做什么？函数 realloc 修改以前用 malloc 或者 calloc 申请的内存的数量。

格式如下：

```
realloc (pointer, number_of_bytes);
```

分配的内存的数量等于 number_of_bytes，这个数值必须在调用 realloc 函数之前被计算。分配出来的内存的地址用 pointer 指定。参数 pointer 必须是以前通过调用 calloc 或者 malloc 函数而返回的一个值。例如，

```
realloc(bb[1],yy);
```

使得 yy 个字节的内存在 dd[1] 指定的位置被分配出来。bb[1] 中保存的地址是 malloc 函数返回的，所以我们可以把它用作 realloc 的参数。

如果 number_of_bytes 比以前通过调用 calloc 或者 malloc 函数所分配的内存要少，那么 realloc 可以成功保留 pointer 指向的那一片内存地址。本例中，内存的内容在调用 realloc 后保持不变。但是如果 number_of_bytes 比以前通过调用 calloc 或者 malloc 函数所分配的内存要大，那么 realloc 不会保留同一块内存。这种情况下，realloc 会在一个新位置分配内存，函数返回一个指向新位置的指针。因此

```
bb[1]=(char *) realloc(bb[1],yy);
```

使得内存被重新定位并且内存块的首地址被保存在 bb[1] 中。如果 bb[1] 的值在执行这个语句的前后是不一样的，realloc 会将旧地址的内容拷贝到新地址中去。

与 calloc 和 malloc 函数类似，转换符 (char *) 使得 realloc 返回的指针转向一个与 bb[] 元素类型一致的字符。但是当处理数值数组的时候，我们应该对 realloc 使用 (int *) 或者 (double *) 转换符。

4）free 函数做什么？ free 函数将以前用 calloc、malloc 或者 realloc 函数分配的内存取消。格式如下：

```
free (pointer)
```

其中 pointer 是一个将要释放的内存区域的地址。例如，

```
free(bb[1]);
```

使得以前用 realloc 分配的用 bb[1] 指定地址的块内存释放掉，以便被其他的 calloc 和 malloc 函数再次使用。保存在 bb[1] 中的地址在调用 free 后保持不变。但是不能再用这个地址存储信息了，因为这个地址的内存没有被预留。如果我们以后还想用 bb[1]，则需要调用 calloc 或者 malloc（不是 realloc）函数并把返回值保存到 bb[1]。

将不再需要的内存释放掉是一个好的编程习惯。不要依赖操作系统在程序执行完后去释放内存。

5）如果 calloc、malloc 或者 realloc 不能按照要求那样去分配一个内存时，会发生什么？它们返回一个空指针。检查这些函数的返回值以确保内存被成功分配是非常必要的。我们在本课程序中没有这么做只是为了简单。

6）从概念上说，在内存的什么地方分配空间？ ANSI C 并没有指定使用动态内存分配的时候应该在哪里分配空间。所以不同的编译器可能处理起来不一样。这里描述一般的内存映像，虽然不是很精确，但是也将一些内存的问题和实现可视化了。

内存管理函数可以被认为在一个叫堆（heap）的内存区域分配空间。概念上说，C 把内存分为 4 个部分：

a. 一个部分叫栈。

b. 一个部分叫堆。

c. 一个部分保存全局变量。

d. 一个部分保存程序指令。

最后三个部分在内存的低地址端，而第一个在内存的高地址端。c 区域和 d 区域在程序执行的时候并不扩大或收缩，但是 a 区域和 b 区域却有变化。图 7-11 给出了可视的映像。

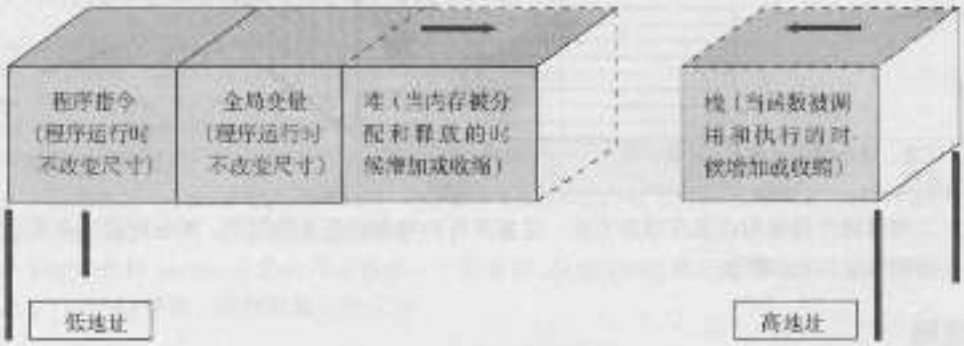


图 7-11　不同区域的内存分布

在图 7-11 中，程序指令和全局变量被保存在左侧，内存的低地址端。它们被包含在实线中，代表在执行的时候是固定的尺寸。堆紧挨着这个区域而栈在内存的最右端的高地址端。当内存在堆被内存管理函数分配的时候，堆的尺寸向右边增长。当内存被释放，它会收缩。

当程序被调用时内存被分配，用来保存和这个函数相关的变量和数据结构，这个时候栈向左边增长。当执行完这个函数后，如果不特殊指定，那么这个内存被释放并且栈收缩。

如果堆或者栈中用了太多的内存，那么这两个区域会相遇或者重叠，这会造成异常的发生并且程序被终止。这种类型的系统，可以使得每一块内存独立地增长，以便使用掉所有可用的内存。

7）使用内存管理函数生成的存储和使用固定尺寸的数组生成的存储之间有什么不同？固定尺寸数组的一个特点是，显然一些数量的内存可以被预留，但是并不是所有的空间都会被程序占用。原因在于内存块的尺寸是程序编译阶段的时候确定的，这个尺寸应该足够大以便能够应付在实际应用时可能出现的最大的尺寸要求。对于这个程序的某个特定的执行，数组可能在全部时间都是空的。这样，一部分的内存就没有被使用。如果运行程序的系统足够大，那么使用固定尺寸的数组是可以接受的。

但是，如果不是这种情况，图 7-12 显示的动态分配内存的方法更好，因为它使用了几乎全部分配的内存。图中显示了只有需要的内存在堆中被分配，而堆中内存的地址被保存在 stack 中的一个指针数组中。同时也显示了栈中一个标准的数组（例如，char 类型）。这个数组用来临时保存一行的信息，以便将来保存到堆中。

固定尺寸的数组有相当的一部分空间是空闲的，而图 7-12 描述的动态分配的系统用了更少的内存。

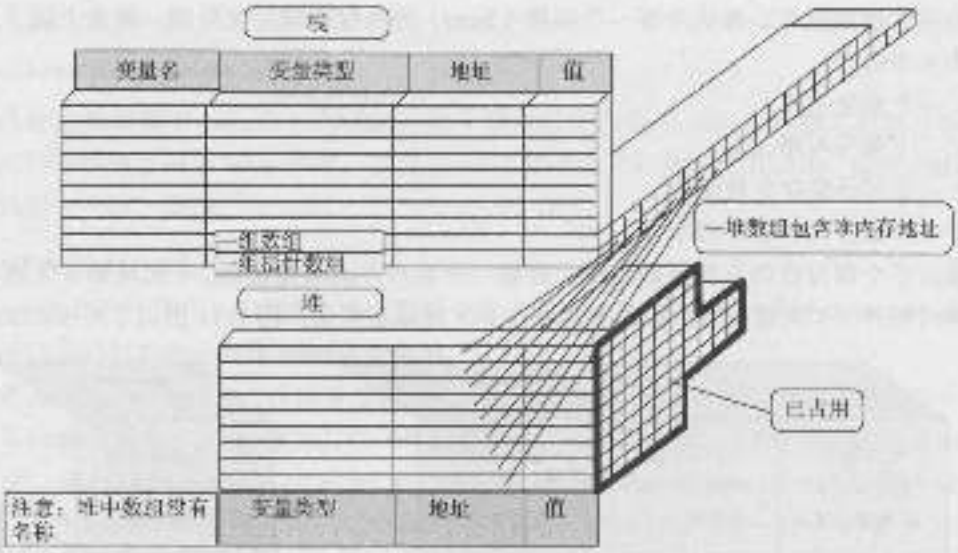


图 7-12 二维数组存储使用动态存储的方法。注意所有的堆存储都被使用了。所分配的内存要远远小于使用固定尺寸的数组

概念回顾

1）我们可以使用动态分配内存的方法在程序执行的过程中生成变量。分配函数是 calloc、malloc、realloc 和 free。在使用这些函数之前，必须使用指令

```
#include <malloc.h>
```

2）函数 calloc 分配了一定数量的内存，尺寸是用传递给 calloc 的参数指定的。调用格式如下：

```
calloc (number_of_elements, bytes_per_element)
```

返回分配的内存的开始地址。如果要求的条件没有满足，返回一个空指针。

3）函数 malloc 与 calloc 做相同的工作，除了分配的尺寸是由单个参数确定的。调用格式如下：

```
malloc (number_of_bytes)
```

返回分配的内存的开始地址。如果要求的条件没有满足，返回一个空指针。

4）函数 realloc 修改以前用 malloc 或者 calloc 申请的内存的数量。格式如下：

```
realloc (pointer, number_of_bytes);
```

参数 pointer 必须是以前通过调用 calloc 或者 malloc 函数而返回的一个值。原始分配的内存中的内容会在重新分配以后保持。

5）free 函数将以前用 calloc、malloc 或者 realloc 函数分配的内存释放。格式如下：

```
free (pointer)
```

在程序终止前将不再需要的内存释放掉是一个好的编程习惯。

练习

1. 判断真假：

a. C 的内存管理函数在程序的执行过程中分配内存。
b. 在编译的时候，安排固定尺寸的数组的内存。
c. C 的内存管理函数使得栈的尺寸在程序运行的过程中增加或收缩。
d. 函数 realloc 可以是在程序中第一个被调用的内存管理函数。
e. 函数 malloc 将分配的内存的所有位初始化为 0。
f. 函数 calloc 将分配的内存的所有位初始化为 0。

2. 给定以下声明

```
char aa[10], *bb, cc[5][50], *dd[8];
int xx, yy, *zz;
```

在下面的语句中发现错误：

```
a. bb = calloc (xx, sizeof(char));
b. bb = (char *)malloc(xx, sizeof(char));
c. aa[0] = (char *) calloc(xx, sizeof(char));
d. zz = (int *)calloc(xx, sizeof(int));
e. cc[0] = (char *) malloc(xx);
```

3. 写一个程序使用 calloc 分配内存来保存一个等价的 a[400][800] 字符数组的内容。a[400][800] 中只被填充了 [10][30] 字符，将字符输出到屏幕。
4. 写一个程序使用 malloc 分配内存来保存一个等价的 a[400][800] 整型数组内容。a[400][800] 中只被填充了 [10][30] 字符，将整数输出到文件。

答案

1. a. 真　b. 真　c. 假　d. 假　e. 假　f. 真
2. a. `bb=(char *)calloc (xx, sizeof(char));` 需要 (char *) 转换指针类型。
   b. `yy=xx*sizeof(char);`
   `bb = (char *)malloc(yy);`
   c. `dd[0] = (char *) calloc(xx, sizeof(char));` 左边必须用指针变量。
   d. 无误。
   e. `dd[0] = (char *) malloc(xx);`

程序开发方法

到目前为止，我们已经使用了四步法来开发程序。但是因为介绍了更加复杂的数据结构，如指针和数组，我们将加上一步来确定程序的数据存储。当程序变得更加复杂时，会有几种不同的数据存储方法，我们可以选择能简化开发流程并减少错误的那一种方法。另外，也可以降低内存的要求以及增加执行的速度，我们开发程序的步骤如下：

1）写出相关公式和背景知识。
2）解决一个特定的例子。
3）决定要使用的主要数据结构。
4）开发算法、结构图和数据流程图。
5）写源代码。

应用程序 7.1　管流速、检查输入数据及模块化设计

问题描述

写一个程序来全面检查键盘的输入数据。当发现数据可接受的时候，利用程序计算管子的一部分的流速 V，键盘输入的数据是另外一部分的流速 v，以及两部分管子的直径 d 和 D。

程序允许下面的输入模式，例如

$d = 12.3$

$v = 23.4$

$D = 34.5$

程序应该允许任何顺序的输入，换句话说，v 可以在 d 的前面输入。如果不是这三个字符中的任何一个被输入，程序显示一个错误信息并提示用户重新输入。如果一个负值或者非数字值在等号后面被输入，程序显示一个错误信息并提示用户重新输入。程序允许用户输入5次错误，不允许输入空格，使用模块化程序设计。

解决方法

1. 相关公式

管中流动的液体用以下的公式描述。在管中的所有部分，流速乘以截面积是一个常数：

$$vA = 常数 \tag{7.1}$$

其中

v= 流速

A= 截面积

因为圆形管子的截面积是 $\pi d^2/4$，可知下面的公式也是正确的：

$$vd^2 = 常数 \tag{7.2}$$

可以利用这个公式来计算管子中不同位置的流速。计算非常直观。如果给定流速 v，管子直径 d，在管子直径为 D 的地方计算流速 V。则：

$$vd^2 = VD^2$$

为了得到 V，我们写出

$$V = vd^2 / D^2 \tag{7.3}$$

2. 特定例子

对于如图7-13所示的管子，

$d = 20cm$

$D = 3cm$

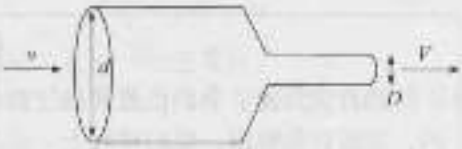


图7-13 管子的示意图

直径为20cm的管子中的流速为50cm/s，那么可以看到，管子中较窄的部分流速较大。这就是当你把喷嘴接上软管的末端时产生的效果。

$$V = (50cm/s)\ (20cm)^2 / (3cm)^2$$

$$V = 2222.22cm/s$$

3. 数据结构

对于这种类型的问题，所用的数据结构取决于如何检查输入数据的合法性。一个通常的方法就是将整行数据读入一个字符数组中，因为字符数组可以接受任何类型的字符。然后去检查字符数组中的每一个元素，看看是否满足我们的要求。如果字符是不可接受的，那么打印一个错误信息，要求用户重新输入一个数据。

对于这个问题，我们选择使用尺寸是30的一维字符数组（`data_line[]`）来代表我们从键盘输入的字符串：如 $v = 23.4$。因为正确输入的字符串只有一个字符（d、v 或者 D）和一个等号，长度为30的数组允许我们有28个字符代表输入数据的数字部分。我们认为这是足够的，所以把字符数组声明为30。

4. 算法

流速的计算公式不是很难。最难的部分在于使用模块化的设计来检查输入的正确性；因此我们开发的算法集中解决这一问题。

本程序中，我们列举了以下的任务：

1）提示、读取和保存输入字符串。

2）分析字符串的开始部分（字母部分）。如果不合法，打印错误信息然后提示用户重新输入。如果合法，将它发给第二部分（数值部分）的分析单元。

3）分析字符串的第二部分。如果不合法，打印错误信息然后提示用户重新输入。如果合法，保存这个数值。

4）当接受了完整的合法条目后，计算流速并打印结果。

这引出了如图 7-14 所表示的结构图。各个函数间的数据流程图如图 7-15 所示：

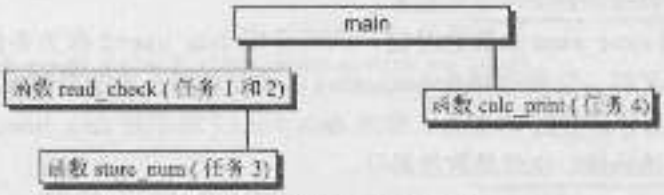


图 7-14 分解的函数结构图

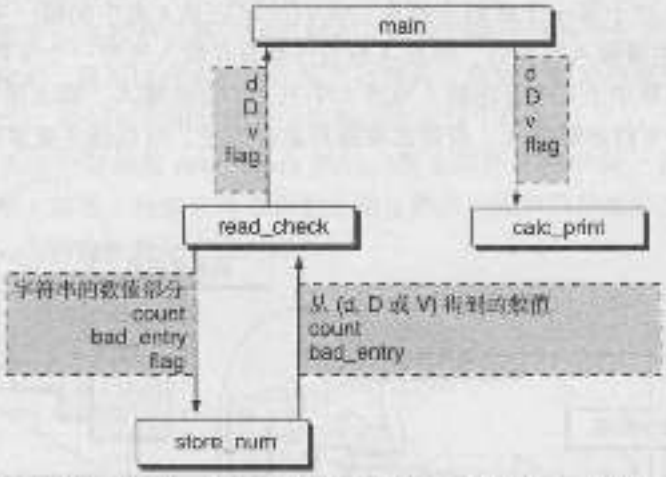


图 7-15 计算管子流速的数据流程图。注意函数间传递的标记和计数器。本例中我们使用 flag、count、bad_entry 变量作为标记和计数器

下面描述每一个函数。

函数 read_check 我们选择将输入的字符串分解为两个部分，"letter-" 部分和数值部分。开发算法的自然入口就是读入并检查字符串第一部分的合法性，这一部分准备放到函数 read_check 中。你会发现开发数据检查程序时，它们包含了一个循环和判断的组合。一些组合也许会变得很复杂。为了使程序流程可视化，画一个如第 4 章演示的三维的流程图会有很大的帮助。这里我们也将使用它们。

输入的字符串的开始部分有 4 类。它们可以被分解为三个可接受的部分以及一个不可接受的部分。这些部分以及接受这一部分后要采取的行为显示如下：

输入	是否接受及采取的行为
d=	可接受，传入计算数值
V=	可接受，传入计算数值
D=	可接受，传入计算数值
其他非法字符串	不可接受，打印错误信息并提示重新输入

因为要从以上的4种操作中选择一个，我们使用if-else-if控制结构。在这个结构中，如果输入是可接受的，调用函数store_num处理data_line[]中的数字部分。如果它是不可接受的，我们将flag设置为0（flag=1代表合法输入），增加对坏条目的计数数量bad_entry++，然后输出错误信息。代码显示如下（使用数组，data_line[]作为输入的字符串数组，count作为好的输入的计数器）。

目前，不要在代码中关注&和*的使用。当你查看整个源代码并学习如何在函数中传递信息的时候，它们的用法会变得清晰起来。

注意，当调用stroc_num函数的时候，我们使用data_line+2作为参数。就像我们学习过的指针代数运算那样，它是存储在data_line[]数组中第三个字符的地址。或者也可以使用&data_line[2]来传递相同的信息。传递data_line+2而不是data_line，我们只是把等号后面的字符串传递给函数，也就是数值部分。

我们必须将if-else-if结构放到一个循环结构里面以读取一个新字符串，确定它的合法性以及得到它的数值部分。结构如图7-16所示。从这个图中，观察到通过每一次的循环读入一个新的字符串。对于每一个新的字符串，我们决定它是4类中的哪一类（##只在本图中代表非法输入）。当将输入归类后，检查其他条件看看是读入另外一个字符串还是退出循环。重复条件说明，如果少于三个合法输入或者少于五个不合法输入，那么循环继续执行。这个图演示了只有三个字符串被读入，但是如果循环条件满足，可以读入更多的字符串。

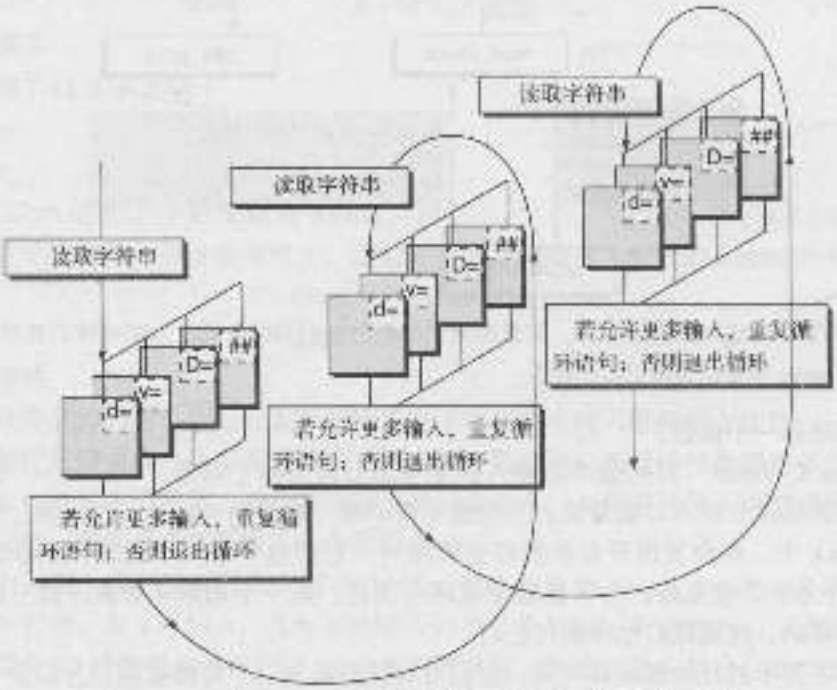


图7-16 函数read_check的流程演示，注意##代表非法输入

循环的代码如下：

```
do
{
        *flag=1;
        printf("Enter data for d, v or D in the form:    d=35.6\n"
                        "No spaces are allowed.\n");
        gets(data_line);

        PREVIOUS IF_ELSE_IF CONTROL STRUCTURE

}       while (count<3  && bad_entry<5);

if  (bad_entry==5) printf ("Too many errors on input."
                        "Execution terminated.\n");
```

开始循环

初始化标志

打印提示符

读一行数据

用来保存合法输入次数和非法输入次数的结构

如果少于三个合法输入或者少于五个不合法输入，那么循环继续执行

如果有太多的不合法输入，退出循环并输出消息

函数 store_num 这个函数检验输入的数值部分的合法性，就像检验输入第一部分的合法性一样。将数据当成字符串而不是 double，字符串中的每一个元素都可以被检验。例如，接受小数点和任何数字但是拒绝负数、字母和符号。我们演示这种方法是因为它非常通用。一旦你学会了这种技术，就可以将它应用于很多的情况。例如，如果想接受流速的负值（代表向相反的方向流动），我们可以用这种方法简单地完成这个任务。

这个函数的主要部分与函数 read_check 类似，都是循环结构中有一个控制结构，在每一个单独的字符元素上循环，检查它是否是数字和小数点。因为以前描述过这种结构，我们这里不介绍细节了。代码的注释见下面内容。

```
do
  {
  if(isdigit(num_string[i]) || num_string[i]=='.'
          || num_string[i]=='\0')  i++;
  else
        {
        (*bad_entry)++;
        *flag=0;
        printf("Error on input.  Try typing your data"
                " again\n");
        }

  } while (*flag==1 && num_string[i-1]!='\0');
```

开始循环

如果数据当元素是一个数字、小数点或字符数组结尾符，增加数组的索引

否则

增加非法输入的计数

将标志设为 0

输出错误信息

当读入的字符是合法的（标志为 1）并且没有到字符串的末尾，那么继续循环

从代码中观察到，我们使用了 isdigit 函数检查每一个字符是否是数字。如果是数字（代表真）返回一个非零整数，否则返回 0（代表假）。在一个控制结构的开头

```
if   ( isdigit(num_string[i] )
```

我们得到了真或假的结果。在上一个循环中，如果考虑的字符是一个数字，就只是简单地增加数组的索引来移到下一个字符。如果不是一个数字（或者小数点），我们将标志设为0以终止循环。当遇到字符串末尾的时候也终止循环。

这个函数的最后一部分就是从一个合法的字符串翻译出double的值。我们使用函数atof，在课程7.8中介绍过，它可以将一个字符串转换为double。下面的代码执行这个操作：

```
if (*flag==1)   ← 如果输入合法
        {
        *value=atof(num_string);   ← 将字符串转换为double
        (*count)++;
        }
```

增加合法输入的计数器

我们使用一个指针变量value来保存输入的数字值，因为想把这个值返回给函数read_check。同理，注意我们使用(*count)++而不是*count++。前面讲过必须使用括号，如果不使用括号，那么增加的就是地址的值，然后在这个地址上取值。事实上，我们需要*count代表的值并增加这个值，所以必须写成(*count)++。

函数calc_print 这个函数利用公式7.1计算流速。只有所有的输入都是合法的，才开始这个计算即当最后一个字符串的翻译标记为1的时候。代码如下：

```
if (flag==1)   ← 如果输入合法
        {
        V=v*(d*d)/(D*D);   ← 计算流速
        printf("The exit velocity for the fluid is V=%lf\n",V);
        }
```

输出结构

函数调用和声明 模块设计的一个重要的部分是在模块中传递信息。这可以通过函数调用和函数定义来完成（应该和函数的声明一致）。对于本程序，我们像图7-16那样显示了每一个函数的声明和调用（图7-17）。从这个图中，注意每一个变量的地址都和函数原型的指针类型相匹配。另外，函数store_num使用const来修饰num_string[]，因为这个函数并不修改字符串而只是存储它。记住，在函数调用的时候每一个参数只能传递单个的地址或者单个的值。如果地址传递给函数，函数内部可以决定是使用这个地址还是使用这个值。另一方面，如果一个值传递给了函数，函数体内我们只能利用这个值。这个规则确定了在每一个函数中应该使用&还是*。

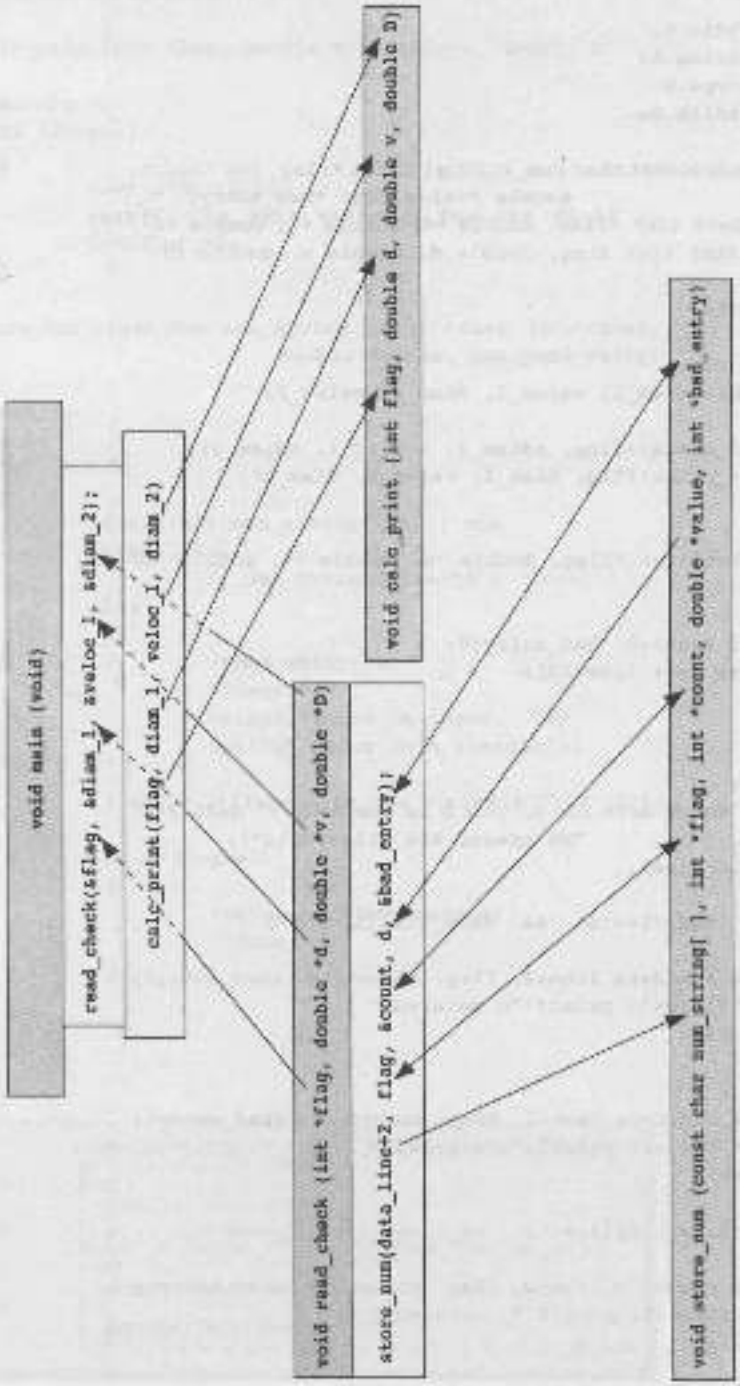


图 7-17 函数调用和声明

源代码

刚刚描述的代码用于以下程序。

这个程序中没有注释，因为我们已经全面分析过它。

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

void store_num (const char num_string[], int *flag, int *count,
                        double *value, int *bad_entry);
void read_check (int *flag, double *d, double *v, double *D);
void calc_print (int flag, double d, double v, double D);

void main(void)
{
        int flag;
        double diam_1, veloc_1, diam_2, veloc_2;

        read_check(&flag, &diam_1, &veloc_1, &diam_2);
        calc_print(flag, diam_1, veloc_1, diam_2);
}

void read_check(int *flag, double *d, double *v, double *D)
{

         int count=0, bad_entry=0;
         char data_line[30];

 do
 {
     *flag=1;
     printf("Enter data for d, v or D in the form:   d=35.6\n"
                        "No spaces are allowed.\n");
     gets(data_line);

     if(data_line[0]=='d'  &&  data_line[1]=='=')
        {
        store_num(data_line+2, flag, &count, d, &bad_entry);
        if (*flag==1) printf("d entered,"
"d=%lf\n",*d);
         }
     {
        store_num(data_line+2, flag, &count, v, &bad_entry);
        if (*flag==1) printf("v entered,"
"v=%lf\n",*v);
     }
     else if(data_line[0]=='D'  &&  data_line[1]=='=')
     {
        store_num(data_line+2, flag, &count, D, &bad_entry);
        if (*flag==1) printf("D entered,"
"D=%lf\n",*D);
     }
     else
     {
                *flag=0;
                bad_entry++;
                printf("Error on input.  Try typing your data
```

```
            again\n");
        }
        } while (count<3  && bad_entry<5);
        if (bad_entry==5) printf ("Too many errors on input.
        Execution terminated.\n");
}
void calc_print(int flag, double d, double v, double D)
{
        double V;
        if (flag==1)
        {
                V=v*(d*d)/(D*D);
                printf("The exit velocity for the fluid is
                V=%lf\n",V);
        }
}
void store_num(const char num_string[], int *flag, int *count,
                                double *value, int *bad_entry)
{
        int i;
        i=0;
        do
        {
                if(isdigit(num_string[i]) || num_
                string[i]=='.'
                        || num_string[i]=='\0')  i++;
                else
                  {
                        (*bad_entry)++;
                        *flag=0;
                        printf("Error on input.  Try
                        typing" "your data again\n");
                  }
                } while (*flag==1 && num_string[i-1]!='\0');

                if (*flag==1)
                {
                        *value=atof(num_string);
                        (*count)++;
                }
}
```

输出

```
                Enter data for d, v or D in the form: d=35.6
                No spaces are allowed
键盘输入        d=20
                d entered, d=20.000000
键盘输入        f=50
                Error on input. Try typing your data again.
键盘输入        v=50
                v entered, v=50.000000
键盘输入        D=3
                D entered, D=3.000000
                The exit velocity for the fluid is V=2222.222222
```

注释

为了保证这个程序足够简单，我们并没有使这个程序可以完全无错运行。例如，如果三

次没有错误地输入 =，程序也会假定已经有了全部的合法输入，然后开始执行运算，最后程序崩溃。对于这个例子，我们接受这个缺点，但是对于一个商用程序来说，这是不可接受的。

同时，写这样的程序，定义合法的输出和非法的输出也是非常重要的。例如，对于这个程序，决定不接受空格、负号和科学计数法，尽管它们也是可以接受的。我们只是想通过这个例子展示一些编程的原则，所以这里不包含这些可能性。在你的软件中，你需要考虑更大范围的输入可能，这样会使得程序更加复杂。

修改练习

1. 修改程序，完成以下任务：
   a. 接受一个负的流速（代表相反方向流动）。
   b. 如果相同的变量（d、v 或者 D）输入多于一次，识别出这种错误。
   c. 接受空格作为输入。
   d. 接受科学计数法作为输入（使用 e 或者 E）。
   e. 接受 V、v 和 d 的值，利用程序计算 D。

应用程序 7.2 地震轶事报告分析、字符串操作和动态内存分配

问题描述

写程序帮助我们从轶事报告中生成一个中级地震的修改麦氏强度地图，轶事报告中有人们对这个地震强度的描述。

你可以利用的是一些描述性的句子，这些描述来源于大量的个人在地震的时候对震动的描述。所有的描述都在一个文件中。每一个描述都开始于个人所在的城市，然后跟着一个对晃动的描述。描述以 # 截止。

程序的输出应该包含一个城市以及强度的值的表格。在每一个城市感受到的强度的数值也被列在表格中。

解决方法

1. 相关公式和背景知识

地震的麦氏强度可以类比于洪水中水的深度。对于一个给定的洪水，有些地方的水要比其他地方的水深。当一个大洪水发生后，可以通过地图描述一个洪水区域内某一个地方的水深。这个地图可以用来预测以后洪水的某个位置的水深。这样可以帮助工程师和城市规划者减少未来洪水的影响。

类似，当一个大地震结束后，在不同位置的晃动的信息被收集起来。对于一个给定的地震，有些地方的晃动要比其他地方大。可以生成一个地图以指示每个地方的晃动的程度。这个图可以用来预测以后这个地区地震的晃动的程度。这样可以帮助工程师和城市规划者减少未来地震的影响。

因为在一个区域内安装的地震测量的地点已较少，所以工程师也依赖于轶事报告来帮助预测某个没有设备的地区的晃动程度。这样会产生大量的信息。一个计算机辅助的笔记分析系统会提高处理信息的效率。

对于本例，我们给出了麦氏强度的一个简化版本。它以下面的方式工作。如果某个人用 strong 来描述晃动，那么这个人感受到修正的麦氏强度（MMI）为 8。如果某个人用 weak 来

描述晃动，那么这个人感受到修正的麦氏强度（MMI）为 4。描述的词和对应的强度如表：

MMI	描述词
4	mild, weak, slow
6	moderate, medium, tempered
8	strong, powerful, sharp
10	violent, destructive, extreme

我们从 5 个城市获得了最近一场中级地震的轶事报告：San Francisco、Berkeley、Palo Alto、Santa Cruz 和 San Jose。意识到每个城市也许并不是只有一个 MMI，我们只是简单地把所有的 MMI 都记录下来。

2. 特定例子

假设有下面的一个地震的轶事记录。注意城市的名字被首先给出，后面接一或者两个描述，每个描述以 # 符号终止。

```
San Francisco.
I felt a strong shock followed by rolling waves.
It lasted a long time.#
Berkeley.
It was mild shaking, rattling windows.
It frightened my dog.#
Palo Alto.
The shaking was very violent.#
Santa Cruz.
The earthquake was very destructive.
It knocked down the chimney on my house.#
Palo Alto.
The extreme shaking made me feel like I was on a
boat in rough sea.#
San Francisco.
I slept right through it. It was much weaker than our
last earthquake.#
```

我们可以使用表格给每个描述指定一个 MMI，得到

城市	使用的描述词	MMI
San Francisco	strong	8
Berkeley	mild	4
Palo Alto	violent	10
Santa Cruz	destructive	10
Palo Alto	extreme	10
San Francisco	weak	4

我们也可以根据 MMI 的值针对每一个城市进行赋值。

每一个 MMI 对应的数量					
MMI	San Francisco	Berkeley	Palo Alto	Santa Cruz	San Jose
4	1	1	0	0	0
6	0	0	0	0	0
8	1	0	0	0	0
10	0	0	2	1	0

这个表可以帮助我们度量每个程序的晃动的强度。这是最终产品的一个简单的描述性的例子。

3. 数据结构

动态数据结构。这个程序的基本存储问题在于如何处理报告。因为需要给定数组的尺寸，我们需要决定将有多少个报告，以及每一个报告里面最多有多少个字符。可以将这些尺寸放到常量宏里面，以后如果需要，我们可以方便地更改它们。现在我们先不用这个特性，只是定义一些数据常量。

我们决定有1000个报告，并且每一个报告最多有800个字。每个字符一个字节，因此我们需要1000×800=800kB。以目前计算机的内存容量看，这完全不是问题。但是这是我们需要的最大容量，而不是实际使用的容量。这是因为，当发生某个地震后，只会有少于1000个人给出报告，而且每一个报告的字数也不会超过800个。这里我们不申请最大容量，而是采用在每次需要内存的时候，动态地进行内存分配。也就是说，根据每个报告的大小，我们只分配对应尺寸的内存。为了达到这个目的，我们预先保留1000个指针，每一个指针对应一个800字的报告。总体来说，程序使用内存的数量还是有一定限制的。当你写程序的时候，可以设置这些限制。

我们演示如何使用动态分配的内存，与使用固定大小的数组做对比。例如，如果在固定大小的数组中保存报告，需要声明一个二维数组，来处理1000个报告，每一个报告的长度为800字符，如下：

```
char all_reports[1000][800];
```

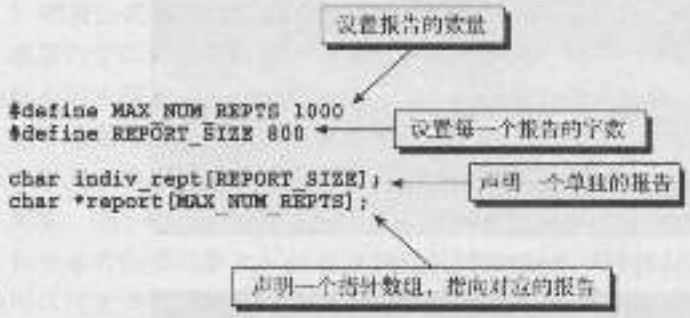
但是，因为使用动态的数据结构，我们定义两个不同的一维数组，其中一个能处理800个字符长的一个报告，

```
char indiv_rept[800];
```

另外一个是指针数组，指向最多的1000个报告。

```
char *report[1000];
```

我们使用常量宏来定义这些数组的尺寸，所以在这个程序中，我们有下面的预处理指令以及数组的声明。



在程序中使用下面描述的数据结构：

1）从报告文件中读入第一个报告（文件中的字符，直到遇到第一个#符号）并将它们暂时保存到indiv_rept数组中。注意这个数组存储单个报告的最大尺寸是800个字符。

2）用strlen函数来计算这个数组中有多少个字符。现在我们知道保存这个报告需要多少内存。

3）用calloc或者malloc函数去分配对应的内存来保存这个报告。在这个过程中，我们

从 calloc 和 malloc 返回了分配的内存的首地址。

4）我们将第一个元素的首地址保存到字符指针数组 report[] 中。那么这块内存以后就可用通过标识符 report[0] 来存取了。

5）使用 strcpy 函数，把数组 indiv_rept[] 中的内容保存到 calloc 返回的地址指定的内存中。这个地址被保存在 report[0] 中。第一个报告就被保存到了内存中，并且可以用地址来存取它。

6）我们从报告文件读入第二个报告到 indiv_rept[] 数组中。这会将第一个报告覆盖掉。但是，这是可接受的，因为已经把第一个报告拷贝到用 report[0] 指示的内存中了。现在执行下面的任务。

- 用 strlen 计算报告中的字符数。
- 用 calloc 或者 malloc 申请这个长度的内存。
- 将申请的内存的地址保存到 report[1] 中。
- 将报告从 indiv_rept[] 拷贝到 report[1] 指定的内存中。

7）对文件中所有的其他报告（当我们读到文件截止符时结束读入报告）都执行步骤 6。观察到我们只利用了文件中的报告，所以并不需要像使用数组声明那样，为 1000 个报告预留空间了。我们只为真实存在的报告及报告中真实包含的字符数分配空间。

很多情况下，为了生成动态的数据结构，你需要

1）确定一个普通的二维数组的尺寸，以理解你最大的内存需求。

2）不是生成一个二维数组，而是生成两个一维数组：

- 普通的一位数组尺寸是步骤 1 中的二维数组的第二维下标的尺寸。
- 一维指针数组等于步骤 1 中的二维数组的第一维下标的尺寸。

3）将部分的内容暂时保存在步骤 2 生成的一维数组中。

4）发现这个数组中实际的填充的数量（例如使用 strlen）。

5）使用 calloc 或者 malloc 分配只需要存储这个数组的内存。

6）保存 calloc 或者 malloc 返回的地址到第 2 步生成的指针数组中。

7）将一维数组中的内容拷贝第 6 步指针数组元素指定的地址中。

8）重复 3 到 7 步，直到保存所有的信息。

9）在后续的程序中使用步骤 6 中的数组元素来存取信息。

我们已经在课程 7.10 中描述过步骤 3 到 6，并且在图 7-12 中演示了它们。

4. 本课中其他的数据结构

我们选择

```
char city[NUM_CITIES][15]={"San Francisco", "Berkeley","Palo
Alto", "Santa Cruz", "San Jose"};
char descriptor[NUM_DESCRIPTORS][15]={"mild","weak","slow",
"moderate", "medium", "tempered", "strong", "powerful",
"sharp", "violent", "destructive", "extreme"};
```

来代表城市和定性的描述。

我们已经确定将定性的描述的顺序和它们代表的强度关联起来。例如，头三个词“mild”，“weak”和“slow”代表 MMI=4。接下来三个词代表 MMI=6。接下来三个词代表 MMI=8。最后三个词代表 MMI=10。这样，这个数组的下标就和它对应的强度关联了。我们将在程序中使用这种关联性。可以使用一个三维数组（其中，一维尺寸是 4，代表我们有

4个MMI)。但是我们发现二维数组就够用了。

我们的结果是一个4行(每一行对应一个MMI)5列(每一列对应一个城市)的表格。如同在其他章节描述的那样，我们使用一个叫做tally[][]的二维数组。声明如下：

```
#define NUM_CITIES 5
int tally [4] [NUM_CITIES];
```

5. 算法

因为选择了一个不同的程序作为本章的模块化设计的例子，并且我们想关注动态内存数据存储，所以这里我们不用模块设计了。

通用的算法可以用一个特定的例子推导得到：

① 从输入文件读入报告。

② 利用数据结构中描述的动态内存分配方法来保存报告的信息。

③ 重复步骤1和2直到所有的报告都成功读入(直到文件末尾被返回)。

④ 在报告中查找城市名。

⑤ 在报告中查找那些对应MMI的描述。

⑥ 对于给定的MMI和城市，将tally加1。

⑦ 重复步骤4到7。

⑧ 将结果按照表格方式输出。

我们将描述每一步的源代码开发，除了第8步。第8步我们在前面的课程中已经描述很多次了。

1)从输入文件读入报告。每一个报告都有一些词并且覆盖好几行。文件中空格用来分隔词，newline用来分隔行。并且#符号用来分隔报告，我们使用#作为哨兵值(意味着搜索它以便分割报告)。

我们可以使用在一个循环中的fscanf(用%s转换限定符)每次将一个词读入二维数组中。但是我们需要搜索#符号作为报告的结尾。这样，我们选择用getc将输入的字符串每次读入一个字符到一维数组(indiv_rept[])并且检查读入的字符是否是#，或者getc返回的是否是文件的结束符。如果任何一个值从getc返回，则跳出循环。因为每一个报告最多容纳800个字符，我们在读入799个字符的时候停止，以便能在字符的末尾加上\0。下面的循环执行了这些操作：

在本例的注释部分，演示了另外一种不使用break语句以跳出循环的方法。

2)利用数据结构中描述的动态内存分配方法来保存报告的信息。在步骤1中计算了报告的长度(rept_len)，我们用以下的语句申请一个足够大的内存。

```
report[j]=(char *)calloc(rept_len, sizeof(char));
```

当这个语句执行完毕后，在用 report[j] 指定的开始地址的内存上还没有保存什么信息。可以将步骤 1 读入的保存在 indiv_rept 地址的单独的报告内容拷贝到这块内存上。下面这个语句完成这个任务：

```
strcpy (report[j], indiv_rept);
```

如果我们希望以后存取这个单独的报告，使用 report[j]。

3）重复步骤 1 和 2 直到所有的报告都成功读入（直到文件末尾或返回）。步骤 2 中的两个语句（和数据结构）是本课使用的动态数据存储的核心内容。但是还需要做另外两件事：

- 循环以便保存所有的报告。
- 检查 calloc 是否成功分配了内存。

下面的代码执行这些附加的任务：

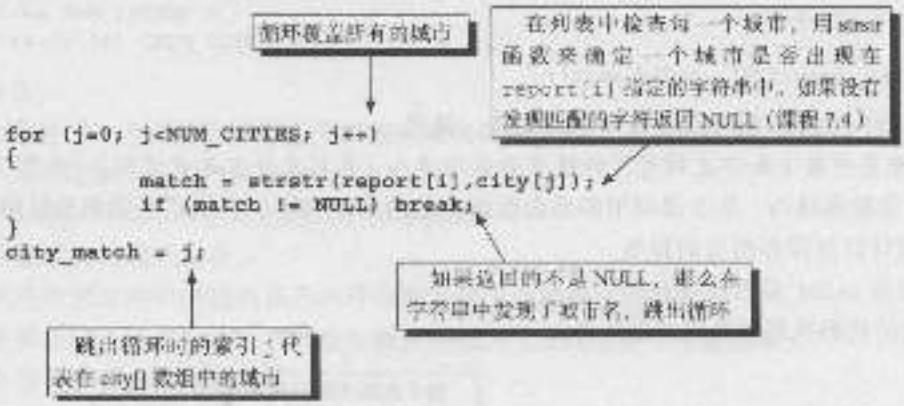
注意在每次循环中，我们增加数组的索引 j。换句话说，每次循环的过程中，重新分配一个内存用来保存新的报告。然后将报告保存到那个地址。

虽然没有要求，但是我们选择将报告输出到屏幕以便验证被保存到地址 report[j] 的每个报告的内容。因为函数 puts 打印一个字符串直到遇到 \0，它很适合这里的任务。回忆在步骤 1 中每个报告的末尾都加上一个 \0。这里我们用 puts，因为它比 printf 或者循环调用 putchar 更好。

4）在报告中查找城市名。现在将所有的报告读入到了内存中。并且建立了所有的指针以指向每一个报告的开始地址，这样就能简单地分别存取每一个报告了。例如，可以设置一个数组索引 i 在 report[i] 指定的字符串中搜索我们感兴趣的信息。首先确定报告中提到的城市。要研究的城市在下面的数组中给出。

```
char city[NUM CITIES][15]={"San Francisco",
"Berkeley","Palo Alto","Santa Cruz", "San Jose"};
```

我们可以用 strstr 函数检查这些城市是否出现在报告中，如下所示：



注意 match 在程序中被声明为一个指针，因为 strstr 返回一个指针。不能将 match 声明为一般的变量。

5）在报告中查找那些对应 MMI 的描述。我们做相似的搜索以确定在报告中出现了那个描述性的词。在报告中出现的描述性的词指明了观察者感到的 MMI 的强度。以下是保存描述词汇的数组：

```
char descriptor[NUM DESCRIPTORS][15]={"mild","weak","slow",
  "moderate", "medium", "tempered", "strong", "powerful",
      "sharp","violent", "destructive", "extreme"};
```

下一个循环在这些词和报告字符串中查找匹配：

在这个循环的末尾，我们保存索引 k。这个索引确定 MMI 以便我们将匹配城市和 MMI 的 tally 加 1。下一个步骤演示如何发现 MMI。

6）对于给定的 MMI 和城市，将 tally 加 1。我们不涉及这个特定步骤的过多细节，因为它并没有使用任何新的字符串操作。简单说会生成下面的表：

每个MMI反应的数目					
MMI	San Francisco	Berkeley	Palo Alto	Santa Cruz	San Jose
4	1	1	0	0	0

6	0	0	0	0	0
8	1	0	0	0	0
10	0	0	2	1	0

这使得以 tally[row][column] 的顺序生成了二维数组 tally[][]。例如，tally[0][0] 代表 MMI = 4 且 city = San Francisco，tally[3][2] 代表 MMI = 10 且 city = Palo Alto。跟踪下面的代码理解我们如何增加那些配对的城市和描述词汇的 tally 的值。

```
if (0 <= descriptor_match && descriptor_match <= 2)
{
                    tally[0][city_match]++;
}
else if (3 <= descriptor_match && descriptor_match <= 5)
{
                    tally[1][city_match]++;
}
else if (6 <= descriptor_match && descriptor_match <= 8)
{
                    tally[2][city_match]++;
}
else if  (9 <= descriptor_match && descriptor_match <= 11)
{
                    tally[3][city_match]++;
}
```

如果索引在 0 和 2 之间，MMI = 4

将 MMI=4 及匹配的城市的记录加 1

如果索引在 3 和 5 之间，MMI = 6

将 MMI=6 及匹配的城市的记录加 1

如果索引在 6 和 8 之间，MMI = 8

将 MMI=8 及匹配的城市的记录加 1

如果索引在 9 和 11 之间，MMI = 10

将 MMI=10 及匹配的城市的记录加 1

6. 源代码

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX NUM REPTS 1000
#define NUM_CITIES  5
#define NUM_DESCRIPTORS 12
#define REPORT_SIZE 800

void main (void)
{
        int i, j, k, rept_len, tally[4][NUM_CITIES]={0}, city_match,
                        descriptor_match, mm, undefined, num_repts;
        char *report[MAX_NUM_REPTS], *match;
        char indiv_rept[REPORT_SIZE]={0};
        char city[NUM_CITIES][15]={"San Francisco", "Berkeley",
                "Palo Alto", "Santa Cruz", "San Jose"};
        char descriptor[NUM_DESCRIPTORS][15]={"mild","weak","slow",
                "moderate", "medium", "tempered",
                "strong", "powerful", "sharp",
                "violent", "destructive", "extreme"};
        FILE *infile;

        infile = fopen ("EQREPT.TXT","r");
        rept_len=1;
        j=0;
        i=0;
        undefined=0;

/*********************************************************************
**This loop reads all of the reports in the file. It stores them in memory
**using the dynamic memory allocation system using calloc.
*********************************************************************/
         while(indiv_rept[i] != EOF  && j<MAX_NUM_REPTS)
         {
```

我们用宏来指定数组的尺寸

match 是一个指针变量

初始化标志和计数器

```
for (i=0; i<REPORT_SIZE-1; i++)
    {
    indiv_rept[i]=getc(infile);
    if (indiv_rept[i]=='#' || indiv_rept[i]==EOF) break;
    }
indiv_rept[i+1]='\0';
rept_len=i+2;
report[j]=(char *)calloc(rept_len, sizeof(char));
(report[j]==NULL)
{
    printf("Memory not allocated for report number %d",
     j+1);
}
else
{
    strcpy (report[j], indiv_rept);
}
puts(report[j]);
j++;
}

num_repts = j;

for(i=0; i<num_repts; i++)
{
        for (j=0; j<NUM_CITIES; j++)
        {
                match = strstr(report[i],city[j]);
                if (match != NULL) break;
        }
        for (k=0; k<NUM_DESCRIPTORS; k++)
        {
                match = strstr(report[i],descriptor[k]);
                if (match != NULL) break;
        }
        city_match = j;
        descriptor_match = k;
        if (0 <= descriptor_match && descriptor_match <= 2)
                {
                        tally[0][j]++;
                }
                else if (3 <= descriptor_match &&
                        descriptor_match <= 5)
                {
                        tally[1][j]++;
                }
                else if (6 <= descriptor_match &&
                        descriptor_match <= 8)
                {
                        tally[2][j]++;
                }
                else if (9 <= descriptor_match &&
                        descriptor_match <= 11)
                {
                        tally[3][j]++;
                }
        }
printf ("Final tally of Modified Mercalli intensities:\n\n");
printf ("Modified\nMercalli\nIntensity ");
for (i=0; i<NUM_CITIES; i++) printf("%12s", city[i]);
mm = 2;
for (i=0; i<\#60>4; i++)
{
        mm+=2;
        printf("\n\n%6d",mm);
        for (j=0; j<NUM_CITIES; j++)
        {
                printf("%12d",tally[i][j]);
        }
}
}
```

利用getc按字符读入输入文件，在#或者EOF处停止。这个循环结束后，indiv_rept[]数组里面保存的是单个的报告

分配内存以保存单个的报告

如果内存不能申请（calloc返回NULL），输出错误信息

否则将报告保存到calloc分配的内存中去

将报告输出到屏幕

目前，所有的报告都使用report[]数组访问

循环搜索所有报告

在报告中确定城市名

在报告中确定描述词

将城市和描述词的记录增加

使用嵌套循环将记录按表格形式输出

输入文件 EQREPT.TXT

```
San Francisco.
I felt a strong shock followed by rolling waves.
It lasted a long time.#
Berkeley.
It was mild shaking, rattling windows.
It frightened my dog.#
Palo Alto.
The shaking was very violent.#
Santa Cruz.
The earthquake was very destructive.
It knocked down the chimney on my house.#
Palo Alto.
The extreme shaking made me feel like I was on a
boat in rough sea.#
San Francisco.
I slept right through it. It was much weaker than our
last earthquake.#
```

输出

```
Final tally of Modified Mercalli intensities:

Modified
Mercalli
Intensity   San Francisco   Berkeley   Palo Alto   Santa Cruz   San Jose

4               1              1          0           0           0
6               0              0          0           0           0
8               1              0          0           0           0
10              0              0          2           1           0
```

注释

对于这种类型的分析，这个程序不是完美的。例如，如果一个报告有“the motions were not strong”，我们会把注意力集中在 strong 而忽略了 not，所以会把报告错误地分类。这样，这个程序在实际中是不能工作的。但是这里借助它演示了动态数据结构和字符串处理操作。这个程序如果利用模块化设计会更好，如果我们使用数据结构 descriptor[][]，那也会更能平行地表达 MMI。

为了简单，我们再一次忽略了数据检查。如果一个报告中没有城市名或者描述词，那么至少你应该打印一个错误信息或设置一个标志。

用一个不带 break 的 for 循环把报告按字符读入也是可以的，像这里做的这样。一个没有 break 的循环被认为更加的结构化。我们在图 7-18 中显示了多少有点怪异的 for 循环。你要理解它的含义，因为它会增加你对 for 循环操作内部的认识。

图 7-18 for 看起来怪异的 for 循环

这个 for 循环能工作是因为循环的执行过程中有初始化、条件以及增加语句。控制语句及循环体的执行的循序见图 7-19。对于图 7-18 中的 for 循环，我们采用以下步骤：

1）循环的初始化使得一个字符被读入。

2）然后检查条件，如果是真，循环体被执行。条件检查读入的字符是否是 # 或者 EOF，并且数组的索引是否小于报告的允许的字符。

3）在本例中，循环体使得读入的字符被保存在 indiv_rept[] 数组中。当字符被保存到数组中后，数组的索引加 1（记住这是后自增符的用法）。

4）所谓的循环控制语句的增加表达式被执行。在这个循环中，“增加”表达式根本不代表一个增加。它只是使得另外一个字符被读入。

5）条件、循环体和增加表达式不断地重复执行直到条件变为假。如图 7-19 所示。

因为“增加”表达式被重复执行，文件逐个字符地被读入。循环体使得字符被保存到字符数组 indiv_rept[] 中。这两部分一直持续发生直到条件变为假。这样，这个循环完成了以前程序完成的工作，但是并没有使用 break。虽然这个循环比起应用程序里使用的循环看起来有点不正统，这个循环被认为更加的结构化，因为它没有使用 break 语句。

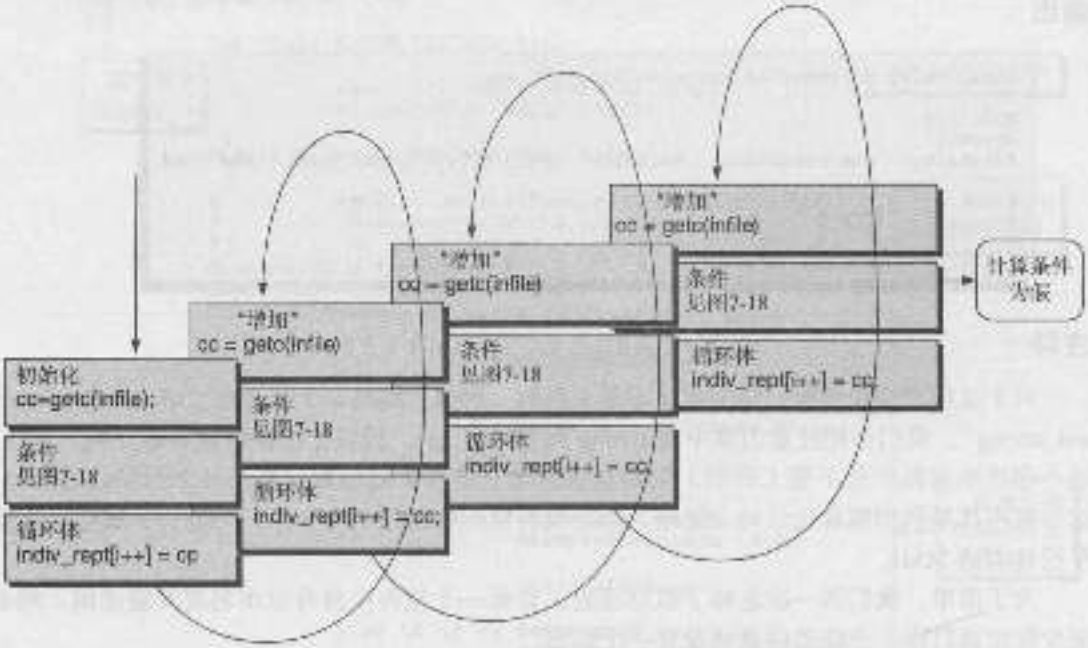


图 7-19 图 7-18 中所示的 for 循环的操作顺序。“初始化”和“增加”语句是一样的，都将单个的字符从文件读入。循环体把这个字符放到报告数组中

修改练习

1）改变 descriptor[][] 数组的结构使得它成为三维数组。作以下声明：

```
#define NUM_DESCRIPTORS 3
char descriptor[4][NUM_DESCRIPTORS][15]={"mild","weak","slow",
        "moderate", "medium", "tempered","strong", "powerful",
        "sharp", "violent", "destructive", "extreme"};
```

其中，NUM_DESCRIPTORS 等于每个变长强度对应的描述词的个数。本例中每个强度有三

个描述词。在程序中修改 descriptor[][] 数组以便使用新的声明。

2）做完以上改变后，加上下列的描述词：

```
MMI     Descriptor
4       feeble
6       firm
8       jolting
10      devastating
```

3）如果在报告中没有正确的城市名或描述词修改程序输出错误信息（但是继续执行）。

4）使得程序能在一个报告中检查多个可接受的描述词。如果报告词彼此冲突，输出错误信息。

5）不需要在程序中增加特性，修改这个程序使得它有模块化设计。画出结构图和数据流程图来适应你的设计。除了 main 函数，你可能还需要 3 个函数。

应用练习

7.1 对于一个工程咨询公司，你有下列的客户：

客户	商业类型
Acme Construction	Machinery design
Johnson Electrical	Switch manufacturing
Brown Heating and Cooling	Boiler design
Smith Switches	Switch manufacturing
Jones Computers	Computer sales
Williams Cleaning	Machinery sales

为了按顺序跟踪客户，你需要一个程序按客户的名字按字母顺序排列，或者按商业类型排序。用冒泡排序写这样一个程序。输入的规范如下：

- 从一个文件中读入客户及商业类型。
- 从键盘读入要求，来决定是根据名字的字母顺序排序还是按商业类型排序。

7.2 在工程界，你也许需要向第三方演示你的工作以证明能力。但是为了保护以前客户的隐私，你需要删掉一些细节。写一个程序将文字报告中的花费数字（以 $ 开头）用 × × × × 来代替，并且将客户的名字用 Client X 来代替。输入的规范如下：

- 从一个文件中读入文字报告。
- 从一个键盘中读入客户的名字。

输出规范为将一个审查完毕后的报告输出到一个文件中。

7.3 写一个程序将两个公式相加，例如以下两个公式：

$$3a + 9b + 10c - 8d = 6$$
$$9a - 2b + 3c - 4d = 4$$

相加得 $12a + 7b + 13c - 12d = 10$

输入规范是将两个公式从键盘读入。输出规范是将结果输出到屏幕。

7.4 电路中电流、电压和电阻的公式是 $I = V/R$，其中 I = 电流，V = 电压，R = 电阻。写一个程序从键盘接受三个分量中的两个，然后根据公式计算第三个并将结果输出到屏幕。以 $I = 2.3$ 的格式输入数据，输入顺序不限。

7.5 速度为 v，质量为 m 的物体的动能 $E = 0.5mv^2$。写一个程序可以从键盘读入 m 和 v 的值以及单位，格式如下：$m = 5$kg。m 可接受的单位是 kg、gm 和 mg。v 可接受的单位为：m/s，km/h，cm/s 和 mm/s。将能量以 joule 的形式输出，1 joule = 1 kg m^2/s^2。

7.6 我们想确定在不使电缆过度延伸的条件下，最多可以给它施加多少力？长度的变化 D 可以通过以下公式获得：$D=(PL)/(AE)$，其中 P=电缆的张力，L=初始的长度，A=横截面积，E=制作电缆的材料的弹性系数。下表给出了一些 E 的值（其中 1MN = 225 000lb[9]）。

材料（Material）	E(MN/m²)
钢（Steel）	200 000
铁（Iron）	100 000
铝（Aluminium）	190 000
黄铜（Brass）	100 000
青铜（Bronze）	80 000
红铜（Copper）	120 000

写一个程序在给定下面的输入数据后，计算电缆的长度变化。

- 材料类型
- 电缆长度
- 截面积
- 电缆张力

输入规范是从键盘读入，用户应该提示输入每一个值。输出规范是将结果输出到屏幕。

7.7 重写问题 7.6，但是将所有的数据以“Material=Brass”格式以任何顺序输入。

7.8 一个二维数组有下面的字符单元素：

```
cc  c   ccc   cccc
ee  e   eee   eeee
aa  a   aaa   aaaa
dd  d   ddd   dddd
bb  b   bbb   bbbb
```

写一个程序完成以下任务：

a. 使用最大交换排序将数组排序，使得它看起来如下：

```
e   ee   eee   eeee
d   dd   ddd   dddd
c   cc   ccc   cccc
b   bb   bbb   bbbb
a   aa   aaa   aaaa
```

b. 使用冒泡排序将数组排序，使得它看起来如下：

```
aaaa   aaa   aa  a
bbbb   bbb   bb  b
cccc   ccc   cc  c
dddd   ddd   dd  d
eeee   eee   ee  e
```

7.9 写文本格式程序读入一个给定的文本文件然后生成另外一个文件，文件中的每一行的长度由用户指定。键盘输入规范如下：

```
• The name of the original text file.
• The maximum length, L, allowed in each line.
• The name of the output file.
```

程序规范是程序应该至少包含一个一维数组和一个字符指针。输出规范是打印以下：

- 原始的文本文件。
- 一个横线以分割原始的文本文件以及新格式的文本文件。线中包含共 L 个数字字符（0 到 9）。

[9] 1lb=0.5436kg。

从 1 开始。例如，如果 L 是 25，那么分割线就应该是：

```
1234567890123456789012345.
```

- 新的格式化的文本文件。

程序不应该拆分任何词，并且不能把两段合为一段。

7.10 修改程序 7.9 使得每段左缩进对齐。

7.11 修改程序 7.9 使得每段右缩进对齐，你可以在每个词之间加空格。

7.12 修改程序 7.9 使得每段中间对齐，你可以在每个词之间加空格。

7.13 修改程序 7.9 使得每段两边对齐，你可以在每个词之间加空格。

7.14 写一个程序在一个文件中确定是否包含特定词。键盘输入规范是

- 原始的文本文件名字。
- 要查找的词。

屏幕的输出规范是显示包含那个词的特定的行。如果文件包含多于一个的特定词，所有的行都应该被显示。

7.15 写一个程序用正确的词来替换一个拼写错误的词。键盘的输入规范如下：

- 原始的文本文件的名字。
- 要替换的拼写错误的词。
- 替换错误的正确的词。
- 新的文本文件的名字。

输出规范是：

- 显示包含错误词的行。
- 显示替换以后的行。
- 将替换以后的行保存成文件。

7.16 写一个程序能对一个文件执行简单的剪掉和粘贴操作，键盘的输入规范如下：

- 原始的文本文件的名字。
- 剪掉部分的边界，用开始词和结束词来代表。注意用于剪掉和粘贴操作的词的边界必须是唯一的。如果词不是唯一的，你可能需要多于一个词来定义边界。
- 开始粘贴的位置，用特定的词来代表。
- 新的文本文件的名字。

输出规范是：

- 调用函数来显示包含剪掉开始词的那一行。函数必须能接受那一行的内容，并将接受的那一行保存到 main 函数中的一维数组中。
- 调用函数来显示包含剪掉结束词的那一行。函数必须能接受那一行的内容，并将接受的那一行保存到 main 函数的字符指针中。
- 显示粘贴操作要粘贴内容的那一行。
- 将剪掉和粘贴后的内容保存到文件。

7.17 写一个程序将一个文本文件分割成等长度的几个文件，键盘的输入规范如下：

- 原始的文本文件的名字。
- 被分割的文件的个数。
- 每一个分割文件的名字。

输出规范是：

- 显示原始的文本文件的名字。
- 显示被分割的文件的个数。
- 显示每一个分割文件的名字。

- 保存每个分割文件。

7.18 写一个程序能够正确地将一个文件中包含的以大小写字母开头的词进行排序。例如文件中包含所有本页出现的词。用最大交换排序来将词排序，不考虑第一个字符的大小写。例如，词 For 应该排在 influence 的后面。为了完成这个任务，你需要暂时将第一个字母转换为小写，以便和其他的词进行比较。然后将它回复到大写的状态。

本章回顾

本章，我们学习了C语言编程中很多重要的概念。我们探讨了C语言中字符串处理的概念。首先，讨论了一些看似自然但是非法的字符串处理的语句，讨论了C语言的初学者经常犯的错误。也将字符串处理函数进行了分类。我们也讨论了字符串的声明以及处理时的一些注意事项。介绍通用字符串处理的C字符串库函数，将两个字符串进行连接以及C语言中的动态内存分配。现在已经讲解了C语言中最重要的编程概念，读者们应该有能力处理很多复杂的编程任务了，恭喜你们！

结构和大型程序设计

本章目标

结束本章的学习后，你将可以：

- 使用结构来处理不同的数据描述一个实体的应用程序。
- 写一个递归函数。
- 使用静态、外部和寄存器存储类别来满足应用的需要。
- 使用位操作符来管理整数 / 字节中的某些特定位。

数据结构

编程时使用数据结构可以使得数据针对某个要解决的问题，以一种相对简单的方式来存取和管理。回忆我们讨论的第一个数据结构是数组，它将一个类似的数据的集合保存在一个连续递增的内存位置上。数组可以用下标符号（数组索引）或者指针符号来引用。数组对于编程来说是很有价值的，因为它把相关的数据集合在一起，可以方面地存取和管理。

当深入讨论编程的时候，我们发现有其他的方法来组合或存取数据。例如，一个记录是一个包含很多不同数据类型的信息。

本章中，我们会讨论一个叫做 C 语言内建的 struct 的数据结构。本章也考察了开发大型程序时需要注意的一些问题。因为你如果进入了商业程序的开发，会遇到比现在遇到的程序大很多的程序。大型程序开发需要特定的技巧，如使用多个源代码文件以及头文件。

我们以 C 语言的结构开始。C 语言版本中的结构通常也叫做一个记录。从现在开始我们只用结构这个词。

C 中的结构

回忆现在用过的数据类型有 int、float 和 double 等。C 语言中的一个结构就是一组数据类型。程序员为了程序的需要定义结构，所以结构被叫做派生数据类型。在一个程序中你可以定义很多的结构。一旦结构被定义，我们像对待 C 语言的 int 或者 double 数据类型那样对待它。

结构对于存储和管理信息来说是非常有用和方便的。例如，在工程界，我们发现经常将不相似的信息放到一起。假如要理解全世界的用水量，我们要把收集到的信息按下面的方式管理和组织：

- 城市名
- 年
- 用水量

我们从尽可能多的城市收集数据。然后可以将这些信息分开进行处理，但是如果把它们组合在一起无疑是更简单和方便的。

例如，就像生成一个 int 或者 double 的数组那样，我们能生成一个结构的数组。其中每一个元素都包含三个分量：城市名、年份、用水量。

每一个分量被当成一个域（field），如图 8-1a 所示。

如果想按照城市名字的顺序将这一信息排序，我们可以用以前介绍过的排序算法。如果把信息保存到一个结构中，我们会发现当按照字母顺序重新排列后，年份和用水量还是与正确的城市联系在一起。如图 8-1b 所示。这样不需要另外的处理以使得年份和用水量与城市名匹配。执行这个管理工作中，我们用城市名当成关键词（key）。这个词汇用来代表在管理工作的时候基于哪个域。

另外，如果想按照用水量递增的顺序排列信息（usage 是 key），排序以后城市名和年份也会正确地被联系在一起，如图 8-1c 所示。数据的关联性（使用 struct 存储数据造成的）确保即使经过重新布局，一个结构中的三个数据也是彼此关联的。

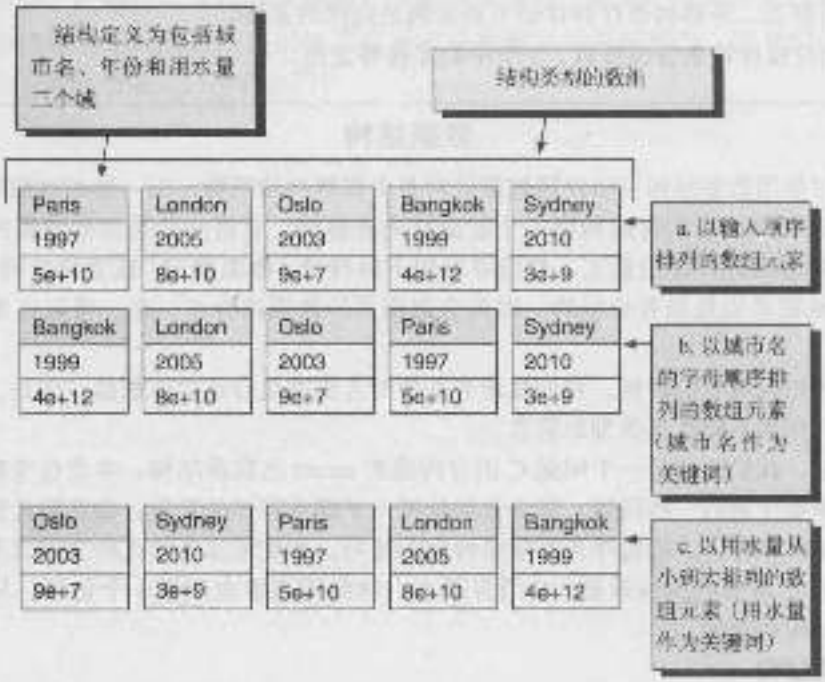


图 8-1 一个结构数组的 5 个元素，其中每个结构包括城市名、年份和用水量

我们并没有用结构数组开始讨论，只是讨论结构。当你理解了结构以及如何存取一个成员（域），我们会开始描述结构数组以及其他和结构相关的知识。

课程 8.1 结构

主题

- 结构声明和初始化
- 结构管理

C 语言像处理 char、int、float 或 double 数据类型那样处理结构。例如，如果一个内存位置被用来保存一个 int，C 知道 4 个字节（目前平台的一个通常标准）内存需要关联并当成一个单元。同理对于 double，C 知道 8 个字节（目前平台的一个通常标准）内存需要关联并

当成一个单元。对于结构，我们可以生成一个有 50 或者 100 个字节的新的数据类型。C 语言允许我们定义这种数据类型，并且确定把多少个字节分配给它。

通过在一个结构中包含一些标准的数据类型和数组来定义一个结构的尺寸。例如，一个结构可以包含一个尺寸是 [20] 的 char 数组，一个 int 和 double（占用 20+4+8=32 字节），或者一个尺寸是 [20] 的 char 数组，三个 double 和另一个尺寸是 [30] 的 char 数组（占用 20+3×8+30=74 字节）。

我们在本课的程序中同时生成了这两个结构。这个程序中定义了两个结构，声明了结构类型的一些变量，并输出了结构成员的值。点（.）操作符在使用结构中是非常重要的。它使得我们可以存取结构成员的内存单元。我们将它用在变量名和成员名之间，你可以在程序中观察它的用法。

源代码

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

struct Consumption
{
        char city[20];
        int year ;
        double usage;
};

struct Resource
{
        char material[30] ;
        double longitude;
        double latitude;
        double quantity;
        char units[20];
};

void main (void )
{
        struct Resource metal, fuel;
        struct Resource wood = {"Oak", 32.5, 13.2, 5e+8, "hectares"};
        struct Consumption water, power;

        metal.longitude = 57.3 ;
        metal.latitude = 32.1 ;
        metal.quantity = 3e-11;
        strcpy (metal.material, "Iron");
        strcpy (metal.units, "cubic metres");

        printf ("The metal information is:\n%s\n%4.1lf degrees longitude\n"
                        "%4.1lf degrees latitude\n%4.0e %s\n\n",
                        metal.material, metal.longitude, metal.latitude,
                        metal.quantity, metal.units);


        printf("Enter water and power:\ncity, year and usage\n");
        scanf ("%s%d%lf%s%d%lf", water.city, &water.year, &water.usage,
                        power.city, &power.year, &power.usage);


        printf ("\n\nThe water and power are:\n%s\n%d\n%4.0lf million litres\n\n"
                        "%s\n%d\n%4.0lf mega watts\n",
                        water.city, water.year, water.usage,
                        power.city, power.year, power.usage);
}
```

结构标签

结构成员

结构定义

结构成员

结构定义

初始化结构变量 wood 的所有成员

声明一个结构类型的变量

初始化结构变量的成员

读入并输出结构的成员

输出

```
The metal information is:
Iron
57.3 degrees longitude
32.1 degrees latitude
3.00000 e + 10 cubic metres

Enter water and power:
city, year and usage
Paris 2003 120 Chicago 2010 50,000

The water and power are:
Paris
2003
120 million litres

Chicago
2010
50,000 mega watts
```

键盘输入

解释

1）C 语言中如何定义一个结构？我们用一个名字（叫结构标签或标签）定义一个结构以及其中一系列成员的名字和类型。例如，

```
struct Consumption
{
    char city[20];
    int year;
    double usage;
};
```

定义了一个 Consumption 标签，包含一个字符数组（city[20]），一个 integer（year）和一个 doube（usage）。同时，

```
struct Resource
{
    char material[30];
    double longitude;
    double latitude;
    double quantity;
    char units[20];
};
```

定义了一个 Resource 标签，包含两个字符数组（material[30] 和 units[20]）和三个 doube（longitude、latitude 和 quantity）。

通常，格式如下：

```
struct Tag
{
    type member_1;
    type member_2;
    .
    .
    .
};
```

其中标签可以是任何合法的标识符。类型可以是任何合法的数据类型，包括 int、float、double 等。另外，member_1 和 member_2 也是合法的标识符并且可以包含数组、指针甚至另外一个结构。注意每一行的末尾需要一个分号，并且结构的末尾需要一个右括号。

2）是否有必要将结构的标签的首字母大写？不需要，但是我们遵循标准的 C 语言命名规范将标签的第一个字母大写。这并不是 C 语言要求的，也不是所有程序员都遵循的方法。但是在某些圈子里这么使用。你要在公司或大学里查阅雇员或者领导的具体的要求。

3）如何将一个变量声明为一个特定的结构类型？就像声明一个 int 或者 double 类型的变量一样。在函数开始的地方（如果声明在函数域）我们写上类型后接一个变量名。如

```
struct Resource metal, fuel;
struct Consumption water, power;
```

声明了 metal 和 fuel 两个 Resource 结构类型的变量，以及 water 和 power 两个 Consumption 结构类型的变量。通用格式如下：

```
struct Tag  variable_1, variable_2, variable_3;
```

其中 Tag 是以前定义的结构标签，variable_1、variable_2 和 variable_3 是程序员生成的变量的标识符。

4）如何在声明结构时初始化？我们能够在声明变量的时候在它的后面使用等号和括号来初始化每一个成员变量。例如 `Struct Resource wood = {"Oak", 32.5, 13.2, 5e+8, "hectares"};` 将变量初始化为如下：

结构变量 wood 成员	值
material[30]	Oak
longitude	32.5
latitude	13.2
quantity	5e+8
units[20]	hectares

注意初始化列表和结构中变量的顺序有对应的关系。

5）如何存取一个特定结构变量的成员？使用点（.）运算符，也叫结构成员运算符。点运算符放在结构变量和成员变量之间。例如，

```
metal.longitude
```

代表结构变量 metal 中的 longitude 成员。变量 metal 已经用类型 `struct Resource` 声明，结构中有 longitude 成员。类似，`metal.latitude` 和 `metal.quantity` 也存取变量 metal 中的其他成员。

6）如何得到包含在一个结构中的字符数组的首地址？使用点操作符，我们能得到字符串首地址，例如对于声明为 `char material[30]` 的 material 成员，`metal.material` 代表的是变量 metal 中成员数组 `material[]` 的首地址。这样我们可以用 `strcpy(metal.material,"Iron");` 将字符串"Iron"拷贝到为变量 metal 中成员数组 material 分配的内存空间中。

7）如何以表格的方式表示一个结构变量？点操作符是如何工作的？我们将每一个结构变量的名字显示在表格的左列，结构的成员显示在 type 列，value 列中显示的是每个单独的成员变量的值。本课程序中的结构如图 8-2 所示：

点操作符metal.latitude如下

名称	类型		地址	值
metal	struct Resource		FFAC	
	material	char[30]	FFAC	I
	longitude	double	FFD2	57.3
	latitude	double	FFCA	32.1
	quantity	double	FFDA	3e+10
	units	char[20]	FFE2	c
fuel	struct Resource		FF62	
	material	char[30]	FF62	未初始化
	longitude	double	FF88	未初始化
	latitude	double	FF80	未初始化
	quantity	double	FF90	未初始化
	units	char[20]	FF98	未初始化
wood	struct Resource		FF18	
	material	char[30]	FF18	O
	longitude	double	FF3E	32.5
	latitude	double	FF38	13.2
	quantity	double	FF46	5e+8
	units	char[20]	FF4E	h
water	struct Consumption		FEFA	
	city	char[20]	FEFA	P
	year	int	FF0E	2003
	usage	double	FF10	120
power	struct Consumption		FEDC	
	city	char[20]	FEDC	C
	year	int	FEF0	2010
	usage	double	FEF2	50000

图 8-2 结构变量表格以及点操作符的演示

8）如何得到结构中数值型成员的地址？我们使用 & 和点操作符。例如，&water.year 得到如图 8-2 所示的地址 FF0E。它可以用在下面的 scanf 语句中：

```
scanf ("%s%d%lf%s%d%lf", water.city, &water.year, &water.
usage, power.city, &power.year, &power.usage);
```

9）是否有其他的方法来定义和声明 C 语言的一个结构？有，但是这里我们描述的是推荐的方法。C 也允许你将变量的名字包含在结构的定义中，例如本课的程序，你也可以用：

```
struct Consumption
{
    char city[20];
    int year;
    double usage;
} water, power;
```

来定义两个 Consumption 结构的变量 water 和 power。如果使用这种方式，C 语言允许你省略结构的标签；于是

```
struct
{
    char city[20];
```

```
    int year;
    double usage;
} water, power;
```

也是一个合法的结构定义和声明。但是不使用标签会阻止以后我们在程序中定义这种结构类型的变量。

当使用这种定义和声明的方法时，必须注意作用域原则。如果它们写在一个函数的外面，那么就把 water 和 power 定义成了全局变量，这降低了整个程序的模块化。如果这种定义写在了函数的里面，那么结构变量只能应用于函数。因为这些缺点，我们不用这种类型的结构定义方法。

10）C 语言在生成一个结构的时候，成员之间是否有空闲的内存？ANSI C 并没有指定是否所有的成员应该在内存中连续，所以在成员之间允许有空闲的内存。这样就不能严格地把各个成员的内存需要累加后计算整个结构需要多少内存。本课的介绍部分，当描述一个 32 字节或者 74 字节的结构时，其内存可能需要大于 32 或者 74 字节。实际的数量取决于具体的 C 语言编译器。

概念回顾

1）结构使用下面方式声明：

```
struct Tag
{
    type member_1;
    type member_2;
    .
    .
    .
};
```

其中 Tag 可以是任何一个合法的标识符。一个模板被建立，但是并不分配内存。

2）变量能用以下方式声明

```
struct Tag  variable_1, variable_2, variable_3;
```

其中 Tag 是上面定义的结构的标签。

3）结构可以用点操作符来管理，也叫做结构成员操作符。点操作符被放到变量名和成员名之间。

```
struct_variable.member.
```

4）一旦利用成员操作符得到了成员，可以像操作一般变量那样操作它。例如可以用 &struct_var.member 得到一个成员的地址。注意这里得到的是一个成员的地址，而不是结构的地址。

练习

1. 判断真假：
   a. 结构在 C 语言中也叫记录
   b. 我们可用点操作符来获得结构成员
   c. ANSI C 要求结构成员保存在一块连续的内存空间中
   d. 结构定义被要求有标签
   e. 结构标签的第一个字符通常大写
2. 定义和声明下面的结构：

```
struct Force
{
    char name[40]
    int point_number;
    double xforce;
    double yforce;
}

struct Force wing, fuselage;
```

指出下面的语句中的错误：

```
a. wing.xforce=15.3;
b. wing.xforce.yforce=28.5;
c. fuselage.name = 'v';
d. strcpy (wing.name, "DC 10");
e. fuselage.point_number = 85;
```

3. 写程序从一个数据文件中读入结构并将它们输出到屏幕。

答案

1. a. 真　　b. 真　　c. 假　　d. 假　　e. 真

2. a. 无误

   b. `wing.yforce = 28.5;`

   c. `strcpy (fuselage.name, "v");`

   d. 无误

   e. 无误

课程 8.2　结构成员

主题

- 指针作为结构成员
- 结构作为结构成员
- 结构的赋值

本课的程序中进一步考察结构的成员。在源代码中，我们定义两个结构，声明了一些结构变量，初始化这些变量并输出它们。

源代码

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

struct Xxx
{
        char aa[30];
        int bb[2];
        double cc;
};
struct Yyy
{
        char dd[50];
        struct Xxx ee;
        char *ee;
        char *ff[3];
};

void main (void)
{
        struct Xxx gg={{"Sample"},{0,1},{5.4}};
        struct Yyy hh={{"String constant"},{{"Text"},{7,8},{12.1}},{"Address"},{"a","b","c"}};
        struct Yyy pp, rr ;
```

结构作为另外一个结构的成员

指针变量作为一个结构的成员

指针数组作为一个结构的成员

结构定义

括号用来分隔结构成员的初始化

```
    strcpy (pp.dd,"Structure ");
    strcpy (pp.mm.aa,"in structure");
    pp.mm.bb[0]=10;
    pp.mm.bb[1]=12;
    pp.mm.cc=57.8;
    pp.ee="Pointer and ";
    pp.ff[0]="array ";
    pp.ff[1]="of ";
    pp.ff[2]="pointers. ";

    rr=pp;

    printf("%s%s %d %d %lf\n%s%s%s%s\n",rr.dd,rr.mm.aa,rr.mm.bb[0],
                rr.mm.bb[1],rr.mm.cc,rr.ee,rr.ff[0],rr.ff[1],rr.ff[2]);
}
```

在结构内部存取结构的成员

存取一个结构的指针成员

将一个结构中的所有的成员的值拷贝到另外一个结构变量的成员内存中

输出

```
Structure in structure 10 12 57.800000
Pointer and array of pointers.
```

解释

1）一个结构的成员可以是另外一个结构吗？可以，本课程序中，结构 Yyy 有图 8-3 所示的这样的成员。

2）如何存取一个结构的成员，当这个结构本身是另一个结构的成员？我们使用两次点操作符。例如，对于结构 Yyy 变量 pp，我们用以下方式存取它的所有成员：

```
pp.dd
pp.mm.aa
pp.mm.bb
pp.mm.cc
pp.ee
pp.ff
```

对于一个结构的结构成员使用两次点操作符

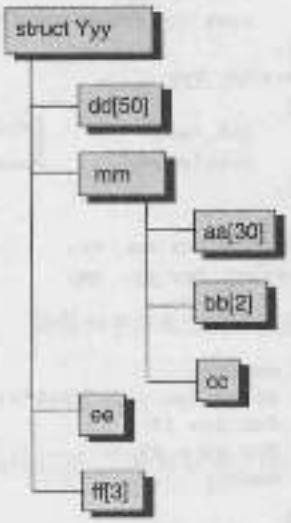


图 8-3 结构 Yyy 和它的成员构成

3）在初始化结构成员的时候如何使用括号？括号不是必需的，但是如果不用，有的编译器会给出警告。如果使用它们，那么能把单独的成员彼此分离开来。当结构成员是数组或者另外一个结构时，用括号会很有帮助。

4）如何把一个结构中的所有成员拷贝到为另外一个结构变量分配的内存中去？我们可以使用单独的赋值运算符。没有必要引用结构中单独的成员或者数组中的元素。例如，

```
rr=pp;
```

把变量 pp 中的成员的值拷贝到结构变量 rr 的内存中去。拷贝一个结构非常简单。注意，我们不能用此方式执行其他类型的操作。换句话说，

```
rr = 3*pp;
```

不是一个合法的 C 语句。如果不引用结构中的成员，我们不能在结构变量本身上使用乘法。

概念回顾

1）C 允许声明一个结构作为另外一个结构的成员，即一个嵌套结构。我们需要另外一

层的成员访问来存取嵌套的结构的成员，例如

```
pp.mm.aa
```

2）允许整个结构的赋值操作，也就是拷贝所有的成员。

```
struc_var2 = struct_var1;
```

练习

1. 判断真假：

a. C不允许声明一个结构作为另外一个结构的成员

b. 为了将一个结构的所有成员拷贝到另外一个相同类型的结构变量中，我们必须逐个成员拷贝

c. 为了存取在另外一个结构中的一个结构的成员，我们必须使用点操作符2次

d. 声明时初始化结构，括号是不需要的

e. 结构中不能有指针成员

2. 给定下面的结构定义和声明

```
struct Xxx
{
    int aa;
    double *bb;
    char *cc[30]
};
struct Yyy
{
    int *dd;
    double ee;
};

struct Xxx mm, nn;
struct Yyy pp, qq;
```

指出下面的语句中的错误

```
a. mm=pp;
b. strcpy(nn.cc, "Test");
c. Xxx.aa= 15;
d. Yyy.ee = 43.8;
e. mm=nn;
```

答案

1. a. 假　b. 假　c. 真　d. 假　e. 假

2. a. 不能指定不同类型的结构。

```
b. nn.cc[0]= "Test";
c. mm.aa = 15;
d. qq.ee =43.8;
e. 无误
```

课程8.3　指向结构的指针

主题

- 结构中的指针操作

我们可以通过使用指针来获取结构中的成员变量。为了完成这个任务，需要生成一个指针变量以保存一个结构的地址。本课的程序定义了一个结构，声明了一个普通的结构变量和

一个指向结构的指针。我们初始化这个普通的变量，然后使用指针变量改变了成员变量的值。

源代码

```
#include <stdio.h>

struct Xxx
{
        int aa;
        double bb;
};

void main (void)
{
        struct Xxx mm;
        struct Xxx *pp;

        mm.aa=0;
        mm.bb=23.2;

        pp=&mm;

        (*pp).aa = 12 ;
        pp->bb   = 97.2;

        printf('%d %lf\n', mm.aa, mm.bb);
}
```

结构定义

声明一个指向结构的指针（一个变量能保存Xxx结构的首地址）

将地址赋给指针

使用指针获取结构变量的成员，显示了两种方法，完成同样的目的

输出mm成员，输出显示使用指针操作的赋值语句确实修改了成员mm

输出

```
12 97.200000
```

解释

1）如何申明一个指向结构的指针变量？使用关键词struct、结构标签、* 和变量名。例如，

```
struct Xxx *pp;
```

声明了一个指针变量pp能够存储struct Xxx类型的地址。

2）如何将一个地址赋给一个结构指针变量？我们使用取址运算符（&）。例如，

```
pp = &mm;
```

将mm结构的开始地址赋给指针变量pp。

3）什么是一个结构的开始地址？结构中第一个成员变量的地址。例如，本课的程序中&mm等于&mm.aa（但是含义有点不一样，前者是一个结构的地址，而后者是一个整数的地址）。

4）如何用指针获取结构的成员？使用单目运算符（*）和点（.）运算符。例如，本课的程序

```
pp = &mm;
(*pp).aa = 12 ;
```

使得结构变量mm的成员aa= 12。这里括号是必需的，因为点操作符比 * 操作符有更高的优先级。于是，*pp.aa = 12 将不会使mm.aa等于12。

输入另外的一个括号是很不方便的。我们使用结构指针运算符（->），也叫做箭头运算符替代它。例如

```
pp = &mm;
pp->bb  = 97.2;
```

使得结构变量 mm 的成员 bb 等于 97.2。使用这种操作符的格式如下：

```
pointer_to_structure -> member_name
```

其中，在 - 和 > 之间没有空格。箭头运算符比 *. 组合更通用。

概念回顾

1）一个指向结构的指针与指向其他基本的数据的指针定义一致。

```
struct tag *pointer_var;
```

2）一旦指向结构的指针被定义，我们可以使用操作符 & 和 *。例如，

```
pointer_var = &struct_var;
(*pointer_var).member = xxx;
```

注意必须使用一对括号。

3）指针运算符用来存取结构中的成员

```
pointer_var->member
```

练习

1. 判断真假：

a. 指针变量用箭头运算符存取结构的成员

b. 对于指针变量我们可以使用 * 和点运算符来存取结构的成员

c. 单目 * 运算符和点运算符比箭头运算符在存取结构成员时更常用

d. 当使用单目 * 运算符和点运算符存取结构成员时不需要使用括号

2. 给定下面的结构定义和声明

```
struct Xxx
{
    int aa;
    double bb;
    char cc[12];
};
struct Xxx mm, nn, *pp, *qq;
```

指出下面的语句中的错误

```
a.pp = mm;
b.qq = &nn;
c.mm->bb = 54.2;
d.*nn.aa =5;
e.*qq.cc = "Sample";
```

答案

1. a. 真　　b. 真　　c. 假　　d. 假

2. a. pp = &mm;
   b. No error.
   c. pp->bb=54.2;

```
d. nn.aa=5;
e. qq=&nn;
   strcpy(qq->cc, "Sample");
```

课程 8.4 结构和函数

主题

- 结构作为函数参数
- 将结构地址传递给函数

前面的课程中学习了指向结构的数组。本课中，我们使用这个知识来将一个结构传入函数并在函数内部使用传入的结构。

源代码

```
#include <stdio.h>

struct Xxx
{
        int aa;
        double bb[2];
        char *cc;
        char *dd[3];
};

void function1 (struct Xxx *qq, struct Xxx *rr);
void main (void)
{


        struct Xxx mm, nn;

        function1 (&mm, &nn);

        printf ("%d %lf %lf %s%s%s\n", mm.aa, mm.bb[0], mm.bb[1],
                mm.cc, mm.dd[0], mm.dd[1], mm.dd[2]);
        printf ("%d %lf %lf %s%s%s\n", nn.aa, nn.bb[0], nn.bb[1],
                nn.cc, nn.dd[0], nn.dd[1], nn.dd[2]);


}

void function1 (struct Xxx *qq, struct Xxx *rr )
{
        (*qq).aa = 12;
        (*qq).bb[0] = 23.4;
        (*qq).bb[1] = 34.5;
        (*qq).cc = "Structure ";
        (*qq).dd[0] = "passed ";
        (*qq).dd[1] = "to ";
        (*qq).dd[2] = "function.";

        rr->aa = 15 ;
        rr->bb[0] = 45.6  ;
        rr->bb[1] = 67.8  ;
        rr->cc = "Pointer ";
        rr->dd[0] = "operators ";
        rr->dd[1] = "can ";
        rr->dd[2] = "be used.";
}
```

将结构变量的第一个成员的地址传递给函数

函数定义有 struct Xxx 指针变量，当调用这个函数的时候，它们包含结构 mm 和 nn 的地址

因为 qq 包含 mm 的地址，这些赋值语句使用 * 和点操作符来填充 mm 结构

填充指针数组 mm.dd[ ]

因为 rr 包含 nn 的地址，这些赋值语句使用 -> 操作符来填充 nn 结构

填充指针数组 nn.dd[ ]

输出

```
12 23.400000 34.500000 Structure passed to function.
15 45.600000 67.800000 Pointer operators can be used.
```

解释

1）如何将一个结构传递给函数，以使函数能修改结构的成员？我们可以将结构的地址（也是结构中第一个成员的地址）传递给函数，这样函数就可以存取为结构成员变量分配的内存单元。例如，本课中调用函数 function1

```
function1 (&mm, &nn);
```

传递 struct xxx 变量 mm 和 nn 的地址。

2）在函数内部，如何存取结构的成员？我们可以使用取值运算符（*）及点运算符，或者使用结构指针运算符 ->。例如，对于本课的程序，

```
(*qq).aa = 12;
rr->aa = 15 ;
```

都会存取成员 aa。因为函数调用和函数定义的关联性，(*qq).aa 存取 mm.aa，rr->aa 存取 nn.aa。

3）在上面的方法中，哪种方法更好一些？大部分程序员都使用结构指针运算符 ->，而不是用 * 和点运算符。从它和数组的相似之处你能获得更直觉的认识。我们在生成链表数据结构的时候也使用这种方法。

概念回顾

为了将一个结构传递给另外一个函数以便程序能修改它，需要传递一个结构地址的引用（这个过程叫做引用传递）。例如，

```
function1(&struct_var1, &struct_var2);
```

指针操作符能够用于修改结构。

练习

判断真假：

a. 结构不能用于函数

b. 将结构变量的地址传入，我们能利用函数修改结构的内容

c. 我们可以通过值（一次一个成员）或者引用（使用结构地址）将结构传递给函数

答案

a. 假　　b. 真　　c. 真

课程 8.5　结构数组

主题

- 声明一个结构数组

像我们在本章介绍部分提到的，C 语言中使用结构类型的一个好处就是我们可以生成一个结构数组。利用结构数组能更快更方便地解决一些问题。

例如，假设我们有一个文件以随机顺序包含 9 个时间 / 电压测量值（当成 x-y 坐标），如：

```
12 87
7 43
10 22
5 56
29 85
```

```
3 34
0 10
14 3
8 65
```

每行的第一个数是以秒代表的时间，第二个数是在此时间测量的以毫伏代表的电压。另外，假设我们的目标是重新以时间的升序排序这些数据。

一个有效的方法就是生成一个结构数组。其中结构可以定义如下：

```
struct coord
{
    double x;
    double y;
}
```

然后声明一个数组结构，例如，

```
struct coord volt_time[1000];
```

于是我们生成了一个关联成员的数组结构，如下所示：

时间（秒）	电压（毫伏）
12	87
7	43
10	22
5	56
29	89
3	34
0	10
14	3
8	65

当重新按照时间的升序排列后，数组变成如下：

时间（秒）	电压（毫伏）
0	10
3	34
5	56
7	43
8	65
10	22
12	87
14	3
29	89

注意，当重新布局数组的第一列时，第二列也相应地被重新布局。并且保证了时间和电压之间的关联性。如果不使用数组，则不能有这个效果。

例如，如果我们把时间放到一个数组 x[9] 中，把电压放到另外一个数组 y[9] 中。当我们重新布局 x 的时候，我们并没有自动地去布局 y 数组。这会丢失掉时间－电压之间的关联性。当然我们可以写代码重新布局 y 数组，但是使用结构数组是更有效率的，因为它自动完成了重新布局。

概念回顾

结构数组和一般数据类型的数组定义一样。

练习

1. 判断真假：
   a. 结构数组和结构中包含数组是一样的
   b. 工程师很少使用结构数组
   c. 我们可以对结构数组的元素排序
   d. 排序一个结构数组，结构中一个成员必须当成 key
2. 写程序用结构数组读一个像下面描述的文件。然后将文件整洁地输出到屏幕。

```
Name          Height(ft)    Age    SSN
Jean Garcia   5.61          21     123-45-6789
Tony Lutz     6.12          36     987-65-4321
Roger Ron     5.87          87     111-22-3333
Jim McKay     3.14           4     444-55-6666
```

答案

1. a. 假　　b. 假　　c. 真　　d. 真

以下课程属于高级编程技术。

课程 8.6　带一个递归调用的函数

主题

- 递归的概念
- 跟踪递归调用函数的流程
- 生成一个递归调用函数

本课中描述了最简单的递归——一个调用的递归。我们在应用程序部分演示两个调用的递归。为了理解递归，我们必须描述递归函数的行为。这里以 log 函数为例。

假设你想计算 5239.7 的自然对数的自然对数的自然对数的自然对数。你可以写出简单的一行程序“`x=log(log(log(log(5239.7))))`”。由于表达式都是从左向右进行计算的，于是在计算赋值语句的右边语句时，最左边的 log 操作被最先执行。这个操作是调用数学函数 log。但是当数学函数 log 看起来像一个参数，它调用下一个函数 log。所以它再一次调用了 log 函数。此时第三个 log 被发现，并且 log 被第三次调用。然后是 log 被第四次发现并调用。目前 log 已经被调用了 4 次，但是还没有计算对数。当这个函数执行了 4 次调用后，函数意识到参数不再是一个调用而是一个真实的值 5239.7 了。

现在，整个过程反过来了。首先 log（5239.7）等于 8.564 02。这个值被返回给第三个 log 调用并且对这个值执行 log 运算后得到 2.147 57。这个值被返回给第二个 log 调用并且对这个值执行 log 运算后得到 0.764 34。这个值被返回给第一个 log 调用并且得到最后的值 −0.268 75。然后将这个值通过赋值语句赋给 x。

对于一个简单的运算解释有点太长了，但是它介绍了递归概念的一个重要方面：在递归调用操作中有三个阶段。第一个是调用阶段，就是重复的调用自己。这一阶段，也许有也许没有（取决于你的设计）很多的计算和操作。当遇到一个最简单的例子并完成了整个函数的操作后，这一阶段结束了。然后回退阶段开始。返回的值使得以前的函数调用能够完成它们的操作并把返回值返回给最开始。当第一个函数调用完成它的操作后，过程结束。

Log 函数在 C 语言中不认为是一个递归函数，因为它并不递归调用自己。在上述简短

的例子中，我们使用 log 函数递归调用自己只是演示在递归调用中发生了什么。本课中，我们演示了一个自动调用自己的递归函数。如果在一个函数体内有一行代码调用自己名字的函数，那么在 C 语言中你就可以根据这一点识别出递归调用函数。例如，如果一个函数的名字是 average，它的原型如下：

```
double average ( double a, double b);
```

那么，在函数体内部，也调用了 average。例如，

```
median += average(c,d);
```

在本课的源代码中有函数 function1。在函数体内部的那一行调用以便你能确认这是一个递归函数。程序的目的就是将 i 和 j 的值加若干次。我们引导你观察每次调用、回退及从调用函数返回时，语句是如何被执行的。这是一个简单的程序，目的就是演示只有一个简单递归调用函数（意味着程序中只有一个语句调用自身）的程序流程。

当递归调用 log 函数的时候，我们提前确定要调用多少次。我们决定调用 4 次。递归函数的问题在于，因为它们自动调用自身，它们可能永远地持续调用自己。换句话说，很有必要在递归函数中包含一部分代码不去调用自身，这一部分叫做回退部分。通常这种函数包含一个 if 控制结构。

源代码

```
#include <stdio.h>

int function1 (int i, int j, int k);
void main (void)
{
        int a=10, b=15, n=5, sum ;
        sum = function1 (a,b,n);
        printf ("\n\n The end result is sum = %d \n", sum);
}

int function1 (int i, int j, int k)
{
        int tot;
        k--;
        if (k != 0)
        {
                printf ("Values in phase 1 - calling phase\n"
                                " i = %d  j = %d  k = %d  tot = %d \n",
                                i,j,k,tot);
                tot = (i+j) + function1 (i,j,k) ;
                printf ("Values in phase 2 - returning phase\n"
                                " i = %d  j = %d  k = %d  tot = %d \n",
                                i,j,k,tot);
                return (tot);
        }
        else
        {
                tot = i+j;
                printf ("Values at reverse1\n"
                                " i = %d  j = %d  k = %d  tot = %d \n"
                                i,j,k,tot);
                return (tot);
        }
}
```

function1 的原型

调用 function1

调用 function1 前的语句，这些语句在函数调用过程中反复执行

真逻辑块

变量 k 在每次调用的时候改变，因为它是一个用在 if-else 结构中的条件表达式中的变量

在 function1 中调用 function1，这是一个递归调用，使得 function1 是一个递归函数

真逻辑块和假逻辑块都有返回语句

假逻辑块，这些语句只在回退的时候执行一次

假逻辑块，注意在假逻辑块中没有递归调用，如果控制进入假逻辑块，那么递归调用停止

function1 中的真逻辑块

调用 function1 后的那些语句，这些语句在返回的过程中反复调用

输出

```
Values in phase 1 - calling phase
i = 10  j = 15  k = 4  tot = 0
Values in phase 1 - calling phase
i = 10  j = 15  k = 3  tot = 0
Values in phase 1 - calling phase
i = 10  j = 15  k = 2  tot = 0
Values in phase 1 - calling phase
i = 10  j = 15  k = 1  tot = 0
Values at reversal
i = 10  j = 15  k = 0  tot = 25
Values in phase 2 - returning phase
i = 10  j = 15  k = 1  tot = 50
Values in phase 2 - returning phase
i = 10  j = 15  k = 2  tot = 75
Values in phase 2 - returning phase
i = 10  j = 15  k = 3  tot = 100
Values in phase 2 - returning phase
i = 10  j = 15  k = 4  tot = 125

The end result is sum = 125
```

解释

1）如何判断一个函数是否是递归函数？在 funcitonl 中的函数体中有语句

```
tot = (i+j) + functionl (i,j,k);
```

这个函数体内的语句调用函数自身，使得 functionl 成为一个递归函数。

2）哪一块代码使得 functionl 不会无限地调用自身？if 控制结构中的假逻辑块（下面所示），使得程序流程不去再次调用 funcitonl。这个块使得 tot 的值被返回。

```
if (k <= 0)
{
        ..
}
else
{
        tot = i+j;
        printf ("Values at reversal\n"
                " i = %d  j = %d  k = %d  tot = %d \n",
                i,j,k,tot);

        return (tot);
}
```

假逻辑块

这块代码以相反的顺序执行。它代表基本或者最简单的例子。假逻辑块代码只执行一次，而真逻辑块被执行了 4 次。

初级程序员经常在一个递归函数中写一个检验条件，但是这个检验条件从来不会从 true 变成 false（或者从 false 变成 true）。当你设计递归程序时，要对此特别小心。要确保一个程序控制能够进入一个没有递归调用的程序块中。

3）这个程序的流程是什么？一个递归调用 functionl 的概念描述如图 8-4 所示。从这里你能看到前 4 次调用 functionl，回退以及从函数调用的返回值。使用这个图以及源代码你就可以跟踪下面要讨论的程序的流程。

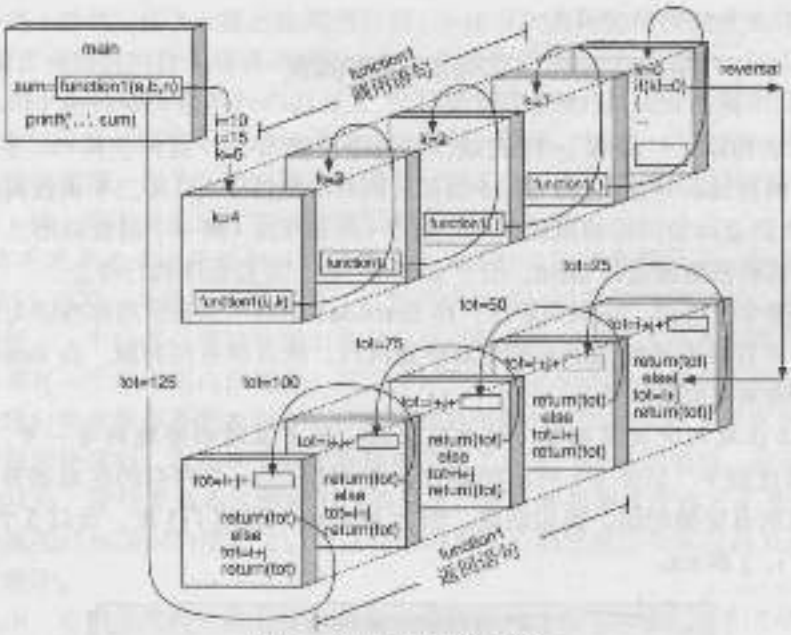


图 8-4 递归调用时的控制流

按步骤考虑下面的过程。我们从 main 调用 function1 开始。在第一个 function1 的调用中，在 if 控制语句前面的语句被执行，k 的值被降低到 4（从 5）。因为 k 不等于 0，控制进入 if 控制的真逻辑块。

当包含函数调用的赋值语句进入真逻辑块时（就像所有的赋值语句一样），赋值语句的右边被首先运算。因为这个表达式包含一个 function1 的调用，在整个表达式计算前，调用 function1 发生。于是，function1 在这一点上被调用（在赋值给 tot 前。）

在这个函数调用中，控制进入 function1 的开始处。k 的值被减为 3。进入真逻辑然后再次调用 function1。

注意从概念上来说，假设一个新的 function1 函数被生成了。因为这是一个新的函数，新的一组参数也被生成。控制再一次转换到 function1 函数的开始处。k 的值被减为 2。进入真逻辑然后再次调用 function1。控制再一次转换到 function1 函数的开始处。k 的值被减为 1。进入真逻辑然后再次调用 function1。控制再一次转换到 function1 函数的开始处。k 的值被减为 0。所有这些过程代表调用阶段。

现在，由于当前的 k 值为 0，进入假逻辑值模块。这是回退阶段。赋值语句 `tot=i+j;` 被执行并且 `tot` 变为 25。`tot` 的值返回到返回阶段的开始处。它返回到哪里呢？

返回的地点就是真逻辑块中的代码：

```
tot = (i+j) + function1 (i,j,k);
```

返回位置。在这一点上，有值 25。语句等同于 `tot=(i+j)+25`

返回地点就是真逻辑块中上面的那个函数调用（其中 k=1，第 4 次函数调用）。现在赋值语句 `tot=i+j+function1(i,j,k);` 可以被执行了。返回值是 25，i 是 10 且 j 是 15；这使得 `tot` 等于 50。现在真逻辑块中的其他语句执行（即函数调用以后的那些语句）。下一个语句是返回语句。这个值返回到哪里呢？返回到这个调用（第三次函数调用）：

```
tot = (i+j) + function1 (i,j,k);
```

返回位置，在这一点上，有值50。语句等同于tot=(i+j)+50

这个赋值语句以 i = 10 和 j=15 完成，并且返回值为 50，这样使得 tot 等于 75。然后在真逻辑块中的其他语句被执行。将 75 返回给同样的赋值语句（第二个函数调用）。这给予 tot 值 100，然后返回给同样的赋值语句，给予 tot 值 125（第一个函数调用）。这是最后一个返回语句，然后控制流返回 main，给予 sum 值 125，并且程序执行停止。

本质上，整个过程是，在调用阶段，在 function1 中调用函数前的那些语句被反复执行。回退被触发，并且没有函数调用的假逻辑块被执行。然后在返回阶段，在 function1 中调用函数后的那些语句被反复执行。

4）为什么在返回阶段没有将 k 增加的语句，但是在返回阶段的每一步，k 的值都在增加？在调用过程中（如图 8-5 所示的每次函数调用），一个内存的区域被开辟出来保存 function1 中的所有变量的值。换句话说，因为 function1 调用了 5 次，所以 5 个不同的区域用来保存变量 i、j 和 tot。

图 8-5 本课程序中调用阶段和返回阶段内存区域及变量的值。注意因为 function1 被调用了 5 次，5 个内存区域被保留出来

这一点也在图 8-5 中演示。在头 5 个箱子中的变量值代表调用阶段的变量值。在返回阶段，这些内存区域被返回。在返回阶段被修改的值都是因为赋值语句被执行，进而值被修改了。但是那些没有改变的值被记住了，因为在调用阶段，内存区域没有被修改。

例如，考虑 k 的值。图 8-5 显示在调用阶段，k 在每次调用的时候被改变（由于 k--; 语句）。这样，每次调用的内存区域有不同的 k 值。但是在返回阶段（在 function1 的 if 控制结构中真逻辑块的下面的部分被执行的时候），没有语句改变 k，但是 k 在调用阶段的值被记住。在图 8-5 中对应的上面和下面的盒子中 k 的值是一致的。变量 tot 是唯一的在上下两个盒子中有不同值的变量，因为 tot 出现在赋值语句的左侧，它在返回阶段执行。看似我们在返回阶段改变 k 值，其实只是记住在函数调用阶段 k 的值。

5）一定有更简单的方法得到相同的结果，请问是什么？很多情况下，你会发现写一个迭代结构（循环）比写一个递归程序简单。另外迭代结构比递归结构效率高，这是因为每次函数调用的时候，一个内存必须被分配出来并保留直到函数关闭及其他外围操作结束。由于每次函数调用都有一个潜在的内存需求，在内存用光之前达到回退条件就变得很重要。一个设计不良好的递归程序很容易吃光内存。循环没有这个额外的负担所以它更有效率。

你可以看到利用递归，在基本实例或最简单条件之前有一些函数调用。这些函数活动其实是在中间的状态，等待着基本实例或回退的发生。在计算机系统中这并不是最有效率的。另外，理论上证明任何递归程序都可以用迭代结构重写。回忆有三个部分的 for 结构看起来就像一个迭代程序。

也必须提出，有时迭代程序是一种对重复过程编码的清晰的方法。它们不需要写大量的 if 控制结构，所以看起来很优雅。在那些场景，递归是一个有价值的编程方法。

概念回顾

1）在内部代码中调用自身的一个函数叫做递归函数。

2）写一个递归程序通常包括两个实例：

a. 通常实例中问题的尺寸被减少。当递归调用发生时总是这样。

b. 结束实例中递归调用结束。通常一个结果会在结束实例中返回。

3）递归程序是另外一种解决迭代问题的方法，不同在于递归方案需要更少的代码。

练习

1. 判断真假：

a. 对递归函数的调用必须出现在它自己的函数体中至少一次

b. 一个递归函数必须有控制语句来防止程序无限运行

c. 在递归程序中没有 void 类型函数

d. 一个递归程序必须返回一个值给它的调用函数；否则不能继续递归过程

2. 写程序调用一个函数读入 1 到 6 位八进制数，并把它转换为十进制数。

3. 用递归程序重写问题 2。

4. 在你的数学书中看看计算 pi 的公式，用一个递归程序重新实现这个公式。

答案

1. a. 真　　b. 真　　c. 假　　d. 假

课程 8.7　生成头文件

主题

- 何时生成新的头文件
- 头文件包括什么

当开发一个大型程序时，你会发现正确地管理不同的代码是高效工作的关键。不同的代

码部分放到不同的文件中。C 允许我们生成头文件，在源代码文件的开头候选生成一个单独包含代码的文件。下面就是一个如何生成合适头文件的例子。

源代码

File 1

```
#include "header_1.h"
void main(void)
{
        int ii;
        double xx;

        ii=3;
        xx=44.7;

        function1(ii,xx,MAX);
}

void function1(int kk, double yy, int nn)
{
        double pp;

        pp=kk+log(yy)+nn;
        printf("pp=%lf\n",pp);
}
```

我们生成的头文件。用 .h 作为文件的扩展名，并将名字包含在双括号中而不是 < > 中

注意，在这个文件中，没有 function1 的原型及 MAX 的定义

File 2 Header_1.H

```
#include <stdio.h>
#include <math.h>
#define MAX 10
void function1 (int, double, int);
```

在我们的头文件中，有预处理指令来包含标准的头文件。头文件中也定义了宏并列出了函数的原型

输出

```
pp= 16.799974
```

解释

1）如何生成自己的头文件？我们生成一个扩展名为 .h 的头文件，并把它放到编译器查找头文件的位置。你需要检查编译器的文档来确定编译器在磁盘的那里（例如哪个目录），寻找头文件并把它放到那个位置上去。

2）我们应该把什么放到头文件中去？像宏、函数原型、注释及结构定义都可以放到程序员生成的头文件中去。目前你需要理解标准头文件，如 stdi.h 中的语法和代码的含义。你可以用任何一个编辑器来查看这些头文件。我们推荐你查看这些头文件以便理解放到头文件中的语句的类型。

3）如何指出程序中包含的头文件是我们自己生成的头文件？将头文件包含在 "" 中而不是 <> 中并没有显示地告诉编译器我们生成了新的头文件，它只是告诉编译器这个头文件可能不在标准库函数定义的头文件目录中。于是编译器在磁盘的其他地方搜索这个头文件。如果你把头文件放到库的头文件目录中，可以使用 <> 来包含头文件名字。但是我们推荐使

用 "" 来将你的头文件放到与其他的源代码相同的目录中。

概念回顾

1）通过拷贝一个文件中用到的函数原型，你可以生成自己的头文件并保存成 .h 文件。

2）生成的头文件可以通过在源代码中包含下面的行来使用：

```
#include "ourheaderFile.h"
```

函数原型被包含在文件的头部，以便能被所有的文件语句引用。使用双引号 "" 而不是 <> 代表文件在我们自己的路径中。

练习

1. 判断真假：

a. 我们经常使用 "" 来代替 <> 将我们自己生成的文件括起来

b. 自己生成的头文件不能超过 20 行的长度

c. 我们经常在头文件的定义中包含常量宏

答案

1. a. 真　　b. 假　　c. 真

课程 8.8　使用多个源文件及存储类别

主题

- 使用多个源代码文件
- 全局变量
- 理解 extern（外部）、register（寄存器）、static（静态）和 auto（自动）四种存储类别。
- 生成个人库

在开发大型程序的时候，我们不把所有的代码放到单独的一个文件中。并且利用模块化设计，一次不使用很多的函数。这样生成很多的源文件，每一个包含相关的函数，这种方法更有利于管理。

当你在一个文件中开发所有函数的工作版本时，文件中的代码可以被执行，从中可以生成目标代码并保存到自己的库。当你在其他文件中要使用这些函数的时候，可以使用链接器将它们链接进你的编译器中。这个过程的一个好处在于它不需要重复编译这些代码。这可以减少开发的时间以及错误的发生。

源代码

File 1

这里我们有函数 function1 的原型，但是函数 function1 的定义在 file2

```
void function1(int xx);
int aa;
```

aa 是一个全局变量，因为它被声明在所有函数的外面

我们可以通过使用 register 关键字选择将一个变量值保存在 register 里面

```
void main(void)
{
        register int bb;
```

```
        auto int cc;
        aa=5;
        cc=10;
        bb=aa+cc;
        function1(bb);
}
```

对于函数域的变量默认的存储类别是auto，这个类型的存储类别在控制返回给调用单元后，其内存被释放

我们依然可以调用function1，即使函数function1的定义不在这个文件中

File 2

这个语句并不为变量aa分配内存。关键字extern代表aa已经在另外一个文件中声明了。这个语句对于任何一个想在不是声明aa的文件中使用aa变量都是必需的

```
#include <stdio.h>
extern int aa;

void function1(int xx)
{
        static int yy;

        yy=2*xx+aa;
        printf("yy=%d\n",yy);
}
```

用关键字static声明的变量在函数结束并且控制返回到调用方函数的时候，值也会被保存

全局变量aa，能被任何一个函数使用，它们不需要从参数列表中传送。因为它们降低了程序的结构化，所以只在必需的地方使用它们

输出

```
yy=35
```

解释

1）C语言的存储类别是什么，使用它们的通用格式是什么？C语言有5个存储类别：extern、register、auto、static和typedef。使用它们的格式如下：

```
storage_class_specifier  type  identifier_1,  identifier_2,
identifier_n;
```

其中stroage_class_specifier是任何一个列出的限定符，type是任何合法的C语言的数据类型，identifier_1, identifier_2和identifier_n是合法的标识符以代表变量。允许多于一个标识符。

2）每一个存储类别限定符的意义和用法是什么？在本书的补充材料中讲解typedef（你的指导者有这本书）。注意它和其他的限定符不同。其他类型的存储类别限定符如下。

限定符extern用于声明一个在其他文件中声明的全局变量。例如在文件file2中用extern声明了变量aa，因为最开始它是在file1中声明的。声明extern int aa;并不为aa申请内存，因为在file1中的声明int aa;已经分配了内存。使用extern只是让编译器知道全局变量aa在另外一个文件中声明了。如果不使用extern去声明一个在其他文件中声明的全局变量，大部分情况下，你的编译器会指出一个相同变量名多次声明的错误。当编译器所有文件被放到一个地方，也就是编译器将所有的文件组合成一个应用程序的时候，它会发现有多个变量有同样的名字。

虽然ANSI C并没有要求（它只是建议，因为寄存器变量比其他的变量存取更快），在声明的时候使用限定符register会使得单个的变量被保存在中央处理器的寄存器部分。大型的数组不应该保存在寄存器中，因为没有足够的地方。不管在实现的时候变量或者数组如何保

存在寄存器，ANSI C 并不允许用 & 来获得它的地址。因为对寄存器变量存取速度的增加，寄存器限定符用在程序中经常存取的变量上会有更高的效率，就像循环中反复使用的变量。寄存器限定符只能用在局部变量。

auto 限定符是局部变量的默认限定符，它比其他的存储类别限定符用得更少。使用 auto 的变量的内存在函数结束运行以后被释放。我们本文中已经详细描述过了。

static 限定符使得局部变量在控制返回到调用函数的时候依然被保留。当你再次调用的时候，static 变量保持上次调用结束时的相同的值。static 有的时候可以代替全局变量。因此在使用全局变量之前，考虑使用局部的 static 变量。使用 static 局部变量比全局变量更可取，因为它保证了程序的模块化。

一个 static 全局变量是一个只能被固定文件中的函数存取的全局变量。声明一个 static 的全局变量可以减少在一个大型程序其他文件中的函数不小心改变全局变量的可能性。在你声明一个普通的全局变量之前，考虑声明一个 static 的全局变量。如果能将全局变量声明为 static，你应该这么做。

3）对于一个函数来说，默认的存储类别是什么？默认的规范是 extern。这样，没有必要将本课中函数的原型声明为

```
extern void function1(int xx);
```

在每一个函数被调用的文件中，必须给出函数原型。

概念回顾

1）C 语言有 5 个存储类别：extern、register、auto、static 和 typedef。使用它们的格式如下：

```
storage class specifier  type  identifier_1,  identifier_2,
identifier_n;
```

2）限定符 extern 用于声明一个在其他文件中声明的全局变量。使用 extern 声明一个变量意味着全局变量已经在另外一个文件中声明了。

3）在声明的时候使用限定符 register 会使得单个的变量被保存在中央处理器的寄存器部分，这样会加快程序的运行。不能在这一存储类别上使用地址运算符。另外也不能保证编译器会把变量放到寄存器，如果是那样，变量会被当成 auto 类别。

4）auto 限定符是局部变量的默认限定符。使用 auto 的变量的内存在函数结束运行以后被释放。

5）static 限定符使得局部变量在控制返回到调用函数的时候依然被保留。一个 static 全局变量是一个只能被固定文件中的函数存取的全局变量。

6）typedef 在本书的补充材料中介绍。

练习

判断真假：

a. 我们经常显式使用 auto 去声明变量的存储类别

b. 当有很多源代码文件时使用存储类别限定符 extern

c. 函数的默认存储限定符是 auto

d. typedef 关键字被 ANSI C 归类为一个存储分类限定符

答案

a. 假　b. 真　c. 假　d. 真

课程 8.9 位操作

主题

- 位运算符
- 使用C语言中的十六进制表示
- 将整数的单个位输出
- 位域

C是一门能够执行很多低层次操作的语言，如在内存中管理单个的位。C语言提供了位操作运算符来执行这些操作。

为了在程序中使用位操作符，需要考虑内存单元中单个位的状态（每一个位或者是1或者是0）。通常十六进制和八进制符号在表示位模式的时候很方便。本课中在源代码及输出中使用十六进制符号，所以我们能够参看并理解位操作符的含义。方便起见，我们再一次在表8-1中给出了十六进制符号以及对应的位模式。

表8-1 十六进制符号及位模式

十进制	十六进制	位模式	十进制	十六进制	位模式
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

如果一个位的值为1我们说这个位被设置了，如果这个位的值为0，说明设置被清除了。这些词有的时候可以当成一个动词，所以我们可以说设置某个位（使这个位的值为1）或者清除某个位（使这个位的值为0）。

C语言允许使用6个操作符来管理一个内存单元中单独的位：&（位与）、|（位或）、^（位异或，也叫XOR）、~（取反）、>>（右移）和<<（左移）。符号~（取反）是单目运算符（意味着它只有一个操作数），其他的都是双目运算符（意味着它需要两个操作数）。

前两个运算符，&（位与）和 （位或）与它们在逻辑运算中对应的&&和||使用方法上一致。回忆在逻辑表达式中，真值用1或者非0代表，假值用0代表。在第4章，我们发现了下面的&&和||规则：

1&&1 = 1（其他的情形下得0；也就是说，1&&0 = 0，0&&0 = 0）

0||0 = 0（其他的情形下得1；也就是说，1||0 = 1，1||1 = 1）

在单独位上使用位的AND和OR产生同样的结果：

1&1 = 1（其他的情形下得0；也就是说，1&0 = 0，0&0 = 0）

0|0 = 0（其他的情形下得1；也就是说，1|0 = 1，1|1 = 1）

使用这些操作符，我们得到下面这4种位操作的例子：

位与 AND				
(hex A)	1	0	1	0
&				
(hex C)	1	1	0	0
(hex 8)	1	0	0	0

位或 OR				
(hex A)	1	0	1	0
|				
(hex C)	1	1	0	0
(hex E)	1	1	1	0

注意 & 和 | 操作符是可交换的，这代表 1&0= 0&1 并且 1|0=0|1。

取反运算符（~）将它操作的位取反。这样 ~（1010）= 0101。在源代码中，我们使用这三个操作符。

位异或（^）给出下面的结果：

0^1 = 1（其他的情形下得 0；也就是说，0^0 = 0，1^1 = 0）

位异或 XOR				
(hex A)	1	0	1	0
^				
(hex C)	1	1	0	0
(hex 6)	0	1	1	0

另外，注意这个操作符是可交换的，这代表 0^1=1^0。

左移和右移操作符（<< 和 >>）将一个单元内的所有的位或者向左移动，或者向右移动。在移动的过程中填充 0 位。例如，如果我们把 1011 向右移动 1 位得到 0101，如图 8-6 中演示。在这个操作中，最左边的位 1 丢弃，并且 0 被添加到了最右边位的位置。类似，向左移动 1 位得到 0110，如图 8-7 所示。

图 8-6 向右移动 1 位

图 8-7 向左移动 1 位

在移动操作中可以移动多于 1 位。如果我们移动 2 位那么就添加两个 0，两位丢弃并且其他的位移动两个位置。

另外，你可能想将一个整型数以位的方式输出到屏幕，可以通过生成一个 mask 来完成，使用这个 mask 和相关的位操作运算符来隔离单独的位，当一个位被隔离出来以后，它可以被放到一个全尺寸的整数单元中，然后使用标准的 printf 语句输出。在本书的解释单

元。我们描述如何用 for 循环来执行这一操作。

在本课的程序中，我们演示了所有位操作符的用法。另外还演示了输出一个给定变量的位。

注意目前在大部分的平台上，integer 变量占据 4 个字节。这意味着当我们考虑它的值的时候有 32 位。在下面的讨论中，为了简洁性，使用每个整数两字节（16 位）来演示这个概念。

源代码

```
#include <stdio.h>

void main(void)
{
        unsigned aa, bb, cc, dd, ee, ff, gg, hh, ii, jj, kk, mm=0x0000, nn;
        int i;

        aa=0xDFFF;
        bb=0x2840;
        cc=0xF7F;
        dd=0x0004;
        ee=0xA3C5;

        ff = aa & cc;
        gg = bb | dd;
        nn  =  aa&(~dd);
        hh = aa ^ bb;
        ii = cc >> 1;
        jj = dd << 3;

        printf ("ff=%p,  gg=%p,  hh=%p,  ii=%p,  jj=%p,  nn=%p\n\n",ff,gg,hh,ii,jj,nn);

        printf ("The bits for ee (hex A3C5) are:\n");

        mm = 1 << 15;
        for (i=16; i>=1; i--)
        {
                kk = ee & mm;
                kk >>= (i-1);
                printf ("%u ",kk);
                mm >>= 1;
        }
        printf ("\n");
}
```

无符号数据类型通常使用2字节或16位

十六进制符号以0x开头，0x后面的4个十六进制符号代表16位

使用位与、位或、异或和取反

向右移动1位

向右移动3位

将最左边的1位置1以便制作一个mask（为了初始进入最左边的位）

这里使用&和只有一个置1的位mm使得ee的位（将mm中置1的那位以外的位清零）赋值给kk

将kk的位右移i-1个位置。这使得位被移动到了最右端（最低）的位置

将在mask mm中的位左移1位

将ee的单独的位输出

输出

```
ff=DF7F,   gg=2844,   hh=F7BF,   ii=7FBF,   jj=0020, nn=DFFB

The bits for ee (hex A3C5) are:
1 0 1 0 0 0 1 1 1 1 0 0 0 1 0 1
```

解释

1）在 C 语言中，如何将一个整数按十六进制来表示？整数可以用一个以 0x 或者 0X 开头的十六进制符号来表示。例如，0xDFFF 代表十六进制 DFFF，0xA3C5 代表十六进制 A3C5。

2）在C语言中，如何将一个整数按八进制来表示？虽然本课的程序中并没有使用八进制符号，我们可以用一个以0开始的八进制符号来表示一个整数。例如，0364代表八进制364，0751代表八进制751。

3）如何运用位操作符？表8-2列举了位操作符及其含义，表8-3给出了结果。

表8-2 位操作

操作符	名字	类型	结合性	例子	解释
&	位与	双目	左到右	aa&cc	当两个操作数中全是1时，把这个位置1。其他情况置0
|	位或	双目	左到右	bb|dd	当两个操作数中全是0时，把这个位置0。其他情况置1
^	位异或	双目	左到右	aa^bb	当两个操作数不一致时，把这个位置1。一致时置0
~	位取反	单目	左到右	~dd	将位从0变成1或者从1变为0
<<	左移	双目	左到右	dd<<3	将位向左移动右面操作数指定的位数
>>	右移	双目	左到右	cc>>1	将位向右移动右面操作数指定的位数

表8-3 位操作运算

表达式	结果	表达式	结果	表达式	结果	表达式	结果
1&1	1	1|1	1	1^1	0	~1	0
1&0	0	1|0	1	1^0	1	~0	1
0&1	0	0|1	1	0^1	1		
0&0	0	0|0	0	0^0	0		

4）我们可以在double或者float类型的数据上使用位操作符吗？不可以。位操作符只能用在整型数据类型上（char、int以及这些类型的修改）。

5）使用位操作的程序是否在所有的计算机系统上都有相同的结果？不是，因为所有的系统并不用相同的位表示，一个给定的程序也许不会在所有的系统上得到相同的结果。这一点非常重要，当你执行位运算的时候必须对此小心。

为了有效地在一个程序中处理单个的位，有必要认知你要使用的具体的实现。换句话说，你需要知道对于某个数据类型（char、int、double或者其他）使用了多少位。同时，第1章中我们没有太多涉及细节，但是你应该知道一个负数是如何表示的。在我们的程序中只使用了无符号整型数，所以不需要关心负数。在我们的实现中，正数被第1章介绍过的二进制方式描述。我们也使用一种实现，其中无符号整数占2个字节或16位。这里的描述正是基于这种实现。

表8-4 本课程序中变量的位表示

变量名	十六进制表示	位表示
aa	DFFF	1101 1111 1111 1111
bb	2840	0010 1000 0100 0000
cc	FF7F	1111 1111 0111 1111
dd	0004	0000 0000 0000 0100
ee	A3C5	1010 0011 1100 0101

6）本课第一部分中位操作的结果是什么？第一部分用的变量和它们的位表示如表8-4所示。操作的结果如表8-5所示。十六进制表示的结果在表8-6中给出。它们在输出中显示。

表8-5 本课程序中位运算的结果

运算	结果	注释
ff = aa & cc;	1101 1111 1111 1111 = aa 1111 1111 0111 1111 = cc 1101 1111 0111 1111 = ff	我们使用cc作为mask来清除aa的第8位，将结果保存到ff
gg = bb | dd;	0010 1000 0100 0000 = bb 0000 0000 0000 0100 = dd 0010 1000 0100 0100 = gg	我们使用dd作为mask来设置bb的第3位，将结果保存到gg

（续）

运算	结果	注释
nn = aa&(~dd);	1101 1111 1111 1111 = aa 1111 1111 1111 1011 = ~dd 1101 1111 1111 1011 = nn	我们使用(~dd)作为mask来清除aa的第3位，将结果保存到nn
hh = aa ^ bb;	1101 1111 1111 1111 = aa 0010 1000 0100 0000 = bb 1111 0111 1011 1111 = hh	我们使用bb反转aa的第7-12位及第14位，将结果保存到hh
ii = cc >> 1;	1111 1111 0111 1111 = cc 0111 1111 1011 1111 = ii	我们将cc向右移动1位，将结果保存到ii
jj = dd << 3;	0000 0000 0000 0100 = dd 0000 0000 0010 0000 = jj	我们将dd向左移动3位，将结果保存到jj

表 8-6 在表 8-5 中的结果的十六进制表示

变量名	位表示	十六进制表示	变量名	位表示	十六进制表示
ff	1101 1111 0111 1111	DF7F	hh	1111 0111 1011 1111	F7BF
gg	0010 1000 0100 0100	2844	ii	0111 1111 1011 1111	7FBF
nn	1101 1111 1111 1011	DFFB	jj	0000 0000 0010 0000	0020

7）什么是 mask？就像我们这里使用的，mask 是一个位模式配合位操作符用于修改另外一个位模式。例如在本课程序中的前两个赋值语句中，我们使用 cc 和 dd 作为用在变量 aa 和 bb 上的 mask 以分别生成新的位模式 ff 和 gg。

通常，我们用 mask 在一个给定的模式中清除或者设置某个单独的位。这样必须要了解用于清除和设置位的方法。

8）如何在一个位模式中描述不同的位？最右边的位是第 1 位（也叫作最不关键位），从右边起第二个叫第 2 位，以此类推，从右向左。例如位模式

```
0001  0001  0010  1100
```

第 3、4、6、9 和 13 个位是 1。其他是 0。

9）如何设置某一位？我们可以生成一个 mask，在 mask 上想设置的那一位的位置上放 1，其他的位放 0。然后将 mask 与要修改的位模式进行位或（OR）操作。

例如，dd 的位模式是（0000 0000 0000 0100），当它用作 mask 的时候，使得第 3 位被设置为 1（见表 8-5）。当在 bb（0010 1000 0100 0000）上以或操作（|）使用这个 mask 后，如同本课程序中使用的那样，gg=bb | dd，我们生成了一个新的位模式 gg（0010 1000 0100 0100）。注意 gg 的位模式与 bb 的位模式是一致的，只是第 3 位被设置为 1。这样就成功地使用 mask 将某位设置为 1 了。

10）如何清除某位？我们可以使用下面两种方法来清除某位：

a. 我们可以生成一个 mask，在 mask 上想清除的那一位的位置上放 0，其他的位放 1。然后将 mask 与要修改的位模式进行位与（AND）操作。例如，cc 的位模式是（1111 1111 0111 1111），当它用作 mask 的时候，使得第 8 位被设置为 0（见表 8-6）。当在 aa（1101 1111 1111 1111）上以与操作（&）使用这个 mask 后，如同本课程序中使用的那样，ff=aa&cc，我们生成了一个新的位模式 ff（1101 1111 0111 1111）。注意 ff 的位模式与 aa 的位模式是一致的，只是第 8 位被清为 0。这样就成功地使用 mask 将某位清为 0 了。

b. 我们可以生成一个 mask，在 mask 上想清除的那一位的位置上放 1，其他的位放 0。

然后采用取反操作将 mask 的所有位取反。完成这个操作后，将 mask 与要修改的位模式进行位与（AND）操作。例如，dd 的位模式是（0000 0000 0000 0100），当它用作 mask 的时候，使得第 3 位被设置为 0（见表 8-6）。第一步用取反操作符将 mask 取反后得到位模式（1111 1111 1111 1011）。当在 aa（1101 1111 1111 1111）上以与操作（&）使用这个 mask 后，如同本课程序中使用的那样，nn=aa&(~dd)，我们生成了一个新的位模式 nn（1101 1111 1111 1011）。注意 nn 的位模式与 aa 的位模式是一致的，只是第 3 位被清为 0。这样就成功地使用 mask 将某位清为 0 了。

11）上面描述的用于将某位清 0 的两种方法，哪种更方便些？很多情况下，第二种方法比第一种更方便些。这是因为它能够比较简单地生成某个位置为 1 的特定的位模式。

12）那么如何能够比较简单地生成某个位置为 1 而其他位置为 0 的特定的位模式？我们可以使用左移动符，例如，语句

```
mm = 1 << 15;
```

使得整数 1 的位表示（0000 0000 0000 0001）向左移动 15 个位置从而得到位模式（1000 0000 0000 0000）。注意因为 1 在最右边的第 16 个位置上，所以我们移动 15 个位置。

同样，如果我们使用

```
mm = 1 << 7;
```

我们可以生成一个位模式 0000 0000 1000 0000。这样，第 8 位上就是 1。

因为某位是 1 的位模式生成比较简单，我们经常用第二种方法来对某位清零。

13）我们可以在位上使用组合赋值运算符吗？可以，所有的都可以除了 ~ 运算符；换句话说，&=、|=、^=、>>=、<<= 都是合法的运算符，而 ~= 不是。

14）在问题 13 中提到的这些运算符是什么含义？它们就像我们使用过的其他的组合赋值运算符一样。例如

```
kk<<=7;
```

等同于

```
kk = kk<<7;
```

并且

```
kk &= aa;
```

等于

```
kk = kk & aa;
```

15）为什么不能使用 ~= 运算符？我们不在组合赋值语句中使用 ~ 是因为它是一个单目运算符（意味着它只有一个操作数）。只有双目运算符才能使用组合赋值语句。

16）为什么使用位操作符？位操作符有大量的实际用处。它们可以用在控制外用设备如打印机、监视器、磁盘驱动器及调制解调器的程序中，因为与这些程序通信的时候，经常要将某位清零或者置 1。

另外不用整数作为表示真假的标志，一个替代的方法是我们用一个单独的位来作为标志。如果我们这么做，可以在一个整数中保存 16 或者 32 个标志。这可以节省内存并允许更快的通信。另外，文件的加密也需要用到位操作符。

任何数组，其中的成员如果只有两种状态，我们就能使用位来表示。例如，如果我们跟踪一个班级中32个学生每天的出勤情况，可以将第一个位代表按字母排序的第一个学生，然后其他的位分别代表按字母排序的其他的学生。1代表这个学生在某天出勤，而0代表他缺勤。这样我们只需要32位的内存来维护一个出勤的记录。对于需要两个状态的某些场景，你可以使用相同的表示方法。

扩展解释

1）我们知道可以使用～将一个位模式中的所有位都取反，那么如何将一个位模式中特定的位取反？我们可以生成一个mask，将想取反的位置上放1然后将其他的位置0。然后在mask和想修改的位模式上使用异或^操作符。

例如，bb的位模式（0010 1000 0100 0000），当以这种方式用作一个mask的时候，使得第7、12、14位取反。当这个mask和变量aa（1101 1111 1111 1111）进行异或运算的时候，如本课的程序所示：bb= aa^cc，我们生成了新的位模式bb（1111 0111 1011 1111）。注意bb的位模式与aa的位模式是一致的，只是第7、12、14位取反。这样我们就成功地使用mask将某位取反了。

2）如何检查一个整型变量的最左边的位的状态（即如何知道某个位是置1还是清0）？我们可以执行下列的步骤：

a. 生成一个最左边是1，其他位都是0的mask。

b. 将mask和整数执行位与操作，然后将结果保存到另外一个整数中。

c. 在整数变量上使用右移操作将最左边的一位移动到最右边的位置。

d. 确定这个整数的值是0还是1。

下面的代码用mask（mm）执行了确定整型变量ee最左边位的状态的操作：

生成一个mask，mm = 1000 0000 0000 0000

使用mask和&生成kk，kk中最左边的一位的值和ee的一致，但是kk上其他的位都是0

```
mm = 1 << 15;
kk = ee & mm ;
kk=kk >>15;
if (kk==1) printf ("The bit is set");
if (kk==0) printf ("The bit is clear");
```

将kk右移15位。这使得最左边的一位移动到了最右边位的位置上。其他位都是0

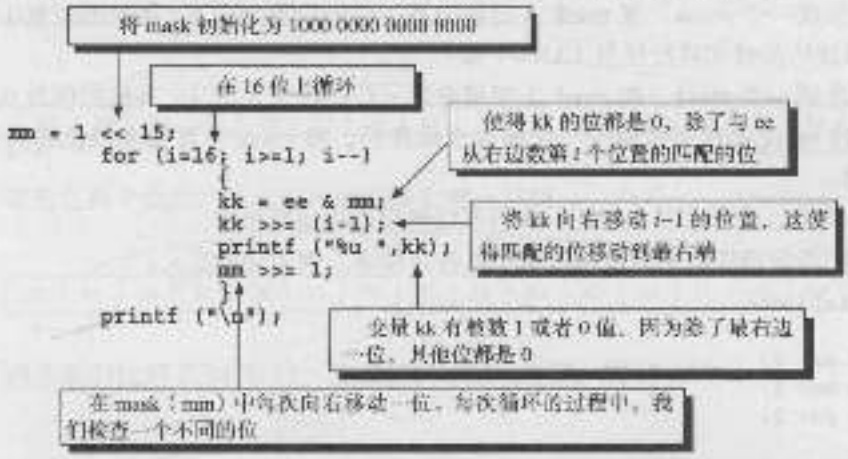
其他位都是0，只有最右边的一位或者是1或者是0。这样kk就有1值或者0值

3）如何确认并输出在一个位模式中的每一位的状态？我们可以使用刚刚描述的确定最左边位的状态的方法，但是需要做一点小小的修改：每次循环地使用一个修改过的mask。mask需要每次向右移动1位，如下所示：

```
1000 0000 0000 0000
0100 0000 0000 0000
0010 0000 0000 0000
...
...
0000 0000 0000 0010
0000 0000 0000 0001
```

然后使用&和mask生成变量kk，在其他的位置上都是0，除了mask上值为1的某个位置。

在这个位置上，这个位对应于 ee 中的 1。我们只需要将 kk 中这个匹配的位移动到最右边，然后用正常输出整数的方法输出就可以了。下面的代码执行这个任务：



4）我们能使用位操作执行代数运算吗？可以，如果想将某个整数乘以 2 或者除以 2，我们可以使用位移动操作符。例如，位模式 0000 0000 0010 1000 代表整数 40。如果向左移动 1 位，位模式变成了 0000 0000 0101 0000。这代表整数 80，即乘以 2。如果我们将原始的位模式向左移动 3 位，得到 0000 0001 0100 0000，它的值为 320，也就是 2^3 或者 8 倍以前的整数。向左移动 n 位相当于乘以 2^n。如果我们不把位移动到最左边以外的位置，就能得到这个结论。

如果我们将原始的位向右移动一个位置，得到 0000 0000 0001 0100。这样我们得到整数 20，即原数除以 2。如果整数是一个奇数，结果就是整数减去 1 后除以 2。例如 0000 0000 0010 1001 是 41，向右移动一位使得最右边的一位丢失，并且结果是 0000 0000 0001 0100，就是 20。为了除以 8（2^3），我们向右边移动 3 位。另外，如果我们移动到最右边位置以外，也不会得到正确的答案。

5）什么是位域？本课的程序中我们没有使用它们。但是 C 允许我们指定特定数目的位用作保存结构的成员。例如结构定义：

```
struct Bitfield_str
{
    unsigned aa: 3;
    unsigned bb: 4;
    unsigned cc:  2;
};
```

使成员 aa 占据 3 位，bb 占据 4 位，cc 占据 2 位。如果在程序中我们声明

```
struct Bitfield_str mm;
```

然后就能够使用 mm.aa、mm.bb 和 mm.cc 存取结构中单独的成员。这里不讨论细节了。使用尺寸为一位的位域是控制单个位变量的另外一种方法。

概念回顾

1）C 语言允许使用六个操作符来管理一个内存单元中单独的位：&（位与）、|（位或）、^（位异或，也叫 XOR）、~（取反）、>>（右移）和 <<（左移）。表 8-5 演示了它们的用法。

2）如何设置某一位？我们可以生成一个 mask，在 mask 上想设置那一位的位置上放 1，

其他的位放0。然后将mask与要修改的位模式进行位或（OR）操作。

3）我们可以使用下面两种方法来清除某位：

a）我们可以生成一个mask，在mask上想清除那一位的位置上放0，其他的位放1。然后将mask与要修改的位模式进行位与（AND）操作。

b）我们可以生成一个mask，在mask上想清除那一位的位置上放1，其他的位放0。然后采用取反操作将mask的所有位取反。完成这个操作后，将mask与要修改的位模式进行位与（AND）操作。

4）将一个整数乘以或者除以2的幂，我们可以使用位移动操作符。

5）C允许我们指定特定数目的位用作保存结构的成员。例如结构定义：

```
struct Bitfield str
{
    unsigned aa: 3;
    unsigned bb: 4;
    unsigned cc: 2;
};
```

使成员aa占据3位，bb占据4位，cc占据2位。

练习

对于aa = 0xAD3F，bb = 0XCC43，cc = 0xAC23，dd = 0xFFFB和ee = 0x23F2，计算

a. aa & bb	b. cc | dd	c. dd ^ ee
d. ~cc	e. aa >> 5	f. dd << 4

根据表8-1将结果以十六进制表示。

应用程序8.1 排序——快速排序算法

问题描述

写一个程序使用快速排序方法将一列整数排序。

解决方法

快速排序算法要比冒泡排序算法好，并且在很多不同的排序问题中被认为是一种很有效率的排序算法。它被用在商用的程序中。

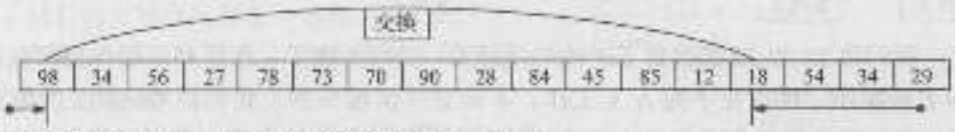
快速排序使用一种分治的方法来排序。它把一个列表分成两个部分（左边和右边），其中左部分只包含比某个值小的数，右部分只包含比某个值大的数。这个值被插在两部分的中间。当快速排序完成分割以后，那个值已经在一个正确的排序位置上（所以以后不需要处理它），同时也生成了两个列表，我们可以分别对它们进行排序而不用考虑另外的一个，这样就会形成一个排序的列表了。快速排序在两个分离的列表上工作，重复这个过程直到所有的值都在正确的位置上。

1. 特定例子

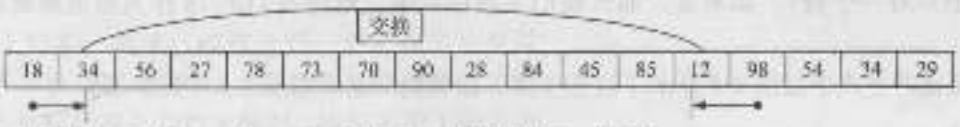
我们以一个没有排序的数字列表开始。这些数字显示在下面的盒子内。首先，这个列表中的一个值被选择，这个值叫做枢值（pivot value）。我们的算法使用最右边的值（29）作为枢值。利用这个值分别从左和从右开始扫描整个数组。这次处理的目的在于把数值29放到一个正确的位置上。从右边我们查找比29小的值（因为这一部分任何比29小的值都应该移

走）然后从左边开始查找比 29 大的值（因为这一部分任何比 29 大的值都应该移走），当在两边都发现了应该移走的值后我们交换它们。

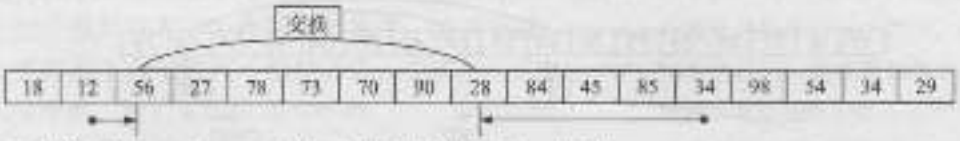
从左边我们找到了 98，从右边找到了 18。



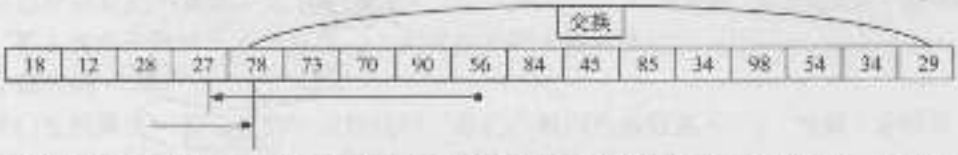
交换这两个值后，从它们开始继续处理，得到



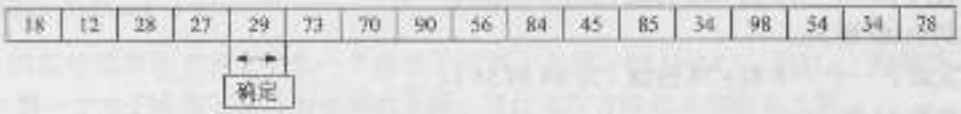
现在我们找到了 34 和 12，交换它们并继续处理，得到



现在找到了 56 和 28，交换它们并继续处理，得到

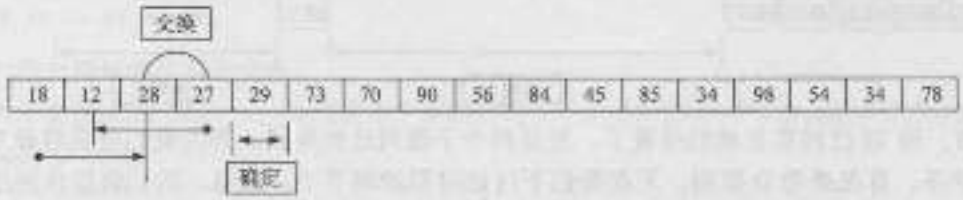


现在得到了 78 和 27。目前，我们左边的位置已经超过了右边的位置，使得箭头重叠。这代表应该停下来，并把我们的枢值与最左边位置的值 78 交换，得到

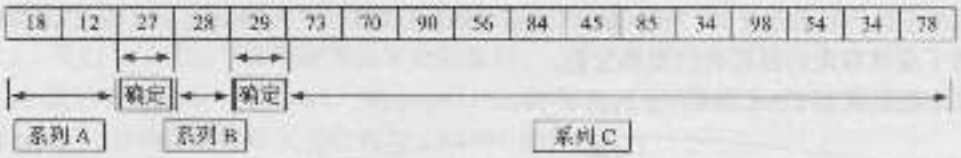


现在，我们的枢值已经在正确的位置上了，它的所有左边的值都小于它，它的所有右边的值都大于它。结果就是我们不再需要处理 29 所在的这个位置了。另外，我们也生成了两个新列：29 左边的列和 29 右边的列。如果将这两个列分别排序，我们就得到了一个完整的排序的数列。

在我们的算法中，当给定一个选择，会首先处理左边的列；于是我们在左边的列执行操作。我们把 27 当成枢值（这个子列中最右边的值），并且分别从左边和从右边查找比 27 小和比 27 大的那些值。从左边的箭头到达了 28，从右边的箭头到达了 12。



因为两个箭头已经重叠了，我们停止并将枢值（27）与左边箭头标示位置的值 28 交换，得到



现在，我们将27和29都放到了正确的位置了，并且生成了三个子列。因为我们总是将最左边的子列排序，现在在子列A上工作。不需要讲解细节了，我们将18和12的值交换到正确的位置。然后我们在子列B上工作，检查左侧和右侧开始位置是否是一致的（代表这个子列中只有一个值）。如果是，那么我们不做任何事。然后我们用78作为枢值来处理子列C。

在简化的方式中以图8-8显示了这些步骤，让我们一起跟踪图8-8中的步骤：

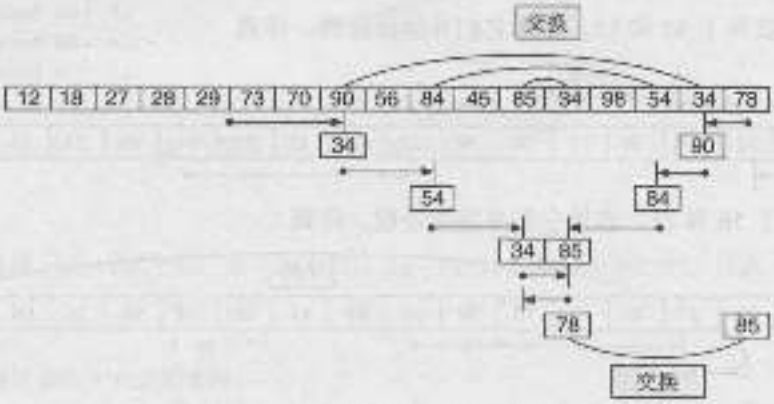


图8-8 快速排序中的步骤，右列表

1）发现下一个>78和<78的值（为90和34）。

2）交换90和34。

3）发现下一个>78和<78的值（为84和54）。

4）交换84和54。

5）发现下一个>78和<78的值（为85和34）。

6）交换85和34。

7）发现下一个>78和<78的值（同样为85和34）。

8）因为箭头已经交叉，停止并交换85和最右边的值78。

9）这一遍处理的结果，如图8-8中每列最下面的数值所示：

目前，值78已经在正确的位置了，并且两个子数列已经生成。因为我们总是将最左边的数列排序，首先处理D数列。下面我们不讨论过程的细节了。但是，我们鼓励你利用手工的方法执行这些步骤并完成这个排序。一个简要的数列操作演示如图8-9所示：

2. 算法

从手工解决的例子和图8-9中，可以看到整个的过程如下：

1）建立左箭头和右箭头的开始位置。

2）将最右边的开始点的值设置为枢值。

3）从左边开始找比枢值大的值。

4）从右边开始找比枢值小的值。

5）如果指针没有重叠，交换步骤3和步骤4发现的值，并重复执行步骤3到步骤5。

6）如果指针重叠，停止并将左箭头指示的值和枢值交换。

7）目前，枢值已经被放到一个正确的位置并且两个子数列已经建立。在生成的最左边的子序列上生成新的两个左右子数列（执行步骤1到步骤6）。然后处理最左边的子序列。持续这个过程直到最左边的子数列只有三个数值需要排序。然后处理最后生成的那个右子数列。换句话说，在最后一个左序列排序前，我们不去处理右序列。

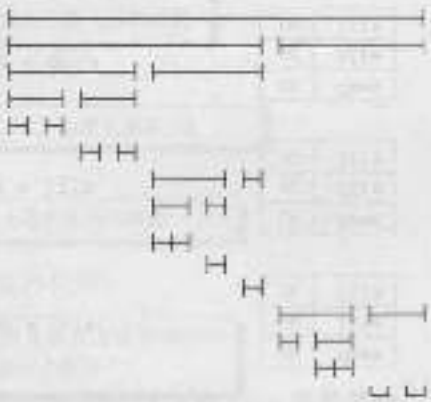


图8-9 一个数列执行快速排序的操作模式。从顶行开始阅读这个图示，然后遵循下面的行。每次数列被分解成子数列，最左边的子数列被首先处理。这个过程一直持续到左边的数列全部排序。然后右边的子数列被处理

在这些步骤之前，我们需要一些其他的步骤：检查右边开始点是不是在左边开始点的右边。如果不是，那么这个子数列已经排好序，上面这些步骤不需要执行。

3. 源代码

我们按照算法一步一步去生成源代码。我们把数列叫做数组a[ ]。变量i是数组a的索引并且从左边开始移动，变量j是数组a的索引并且从右边开始移动。

我们可以使用一个while循环从左边开始移动到比枢值大的位置，例如循环

```
while (a[++i] < pivot);
```

使得i的值持续增加直到a中的一个值大于或等于枢值。循环执行结束后，i的值等于从左开始的第一个大于或等于枢值的元素的下标。这代表左方向箭头的箭头头部。

同样，循环

```
while (a[--j] > pivot);
```

使得j的值持续减少直到a中的一个值小于或等于枢值。循环执行结束后，j的值等于从右开始的第一个小于或等于枢值的元素的下标。这代表右方向箭头的箭头头部。

注意，这一步结束后，如果i的值大于或者等于j的值，那么箭头是重叠的，我们不交换元素。语句

```
if (i >= j) break;
```

把我们带出循环并且不做交换。

为了交换a[i]和a[j]，我们使用下面的代码（一个临时的存储变量保存要交换的值）：

```
swap = a[i];
a[i] = a[j];
a[j] = swap;
```

我们没有演示过这个特殊的指令序列；但是交换值在编程中使用非常普遍。因为两个动作不能同时进行，我们必须使用一个中间的存储位置（这里用变量swap来代表）。这三个语

句（在这三个语句执行前，a[i] = 90 而且 a[j] = 34）的行为是：

a[i]	90
a[j]	34
swap	90

```
swap = a[i];
```

a[i]	34
a[j]	34
swap	90

```
a[i] = a[j];
```

a[i]	34
a[j]	90
swap	90

```
a[j] = swap;
```

结果就是 a[i] 和 a[j] 的值被交换。我们能把这些语句放到一个循环里去形成处理子数列的一个完成的流程。

```
while (1)
{
        while (a[++i] < pivot);
        while (a[--j] > pivot);
        if (i >= j) break;
        swap = a[i];
        a[i] = a[j];
        a[j] = swap;
}
```

生成一个需要 break 来结束的循环

增加 i（从左边开始）直到值大于等于枢值

减少 j（从右边开始）直到值小于等于枢值

如果箭头头部重叠，退出循环

将左箭头和右箭头指示的值交换

行 while(1) 生成的循环是一个无限循环。因为 1 永远都是真。但是语句 if(i>=j) break; 使得循环终止。当写这类循环的时候你要非常小心。如果 i 不大于或等于 j 的话，那么循环不会终止而且程序（如果它运行在 PC 上）会看起来被锁住了一样。

如果初始的左箭头和右箭头的尾部的下标是变量 left_position 和 right_position，我们可以用下面的语句交换枢值（a[right_position]）和左箭头头部的值（a[i]）。

```
swap = a[i];
a[i] = a[right_position];
a[right_position] = swap;
```

把这个语句加入到上面的代码得到下面的代码：

```
while (1)
{
        while (a[++i] < pivot);
        while (a[--j] > pivot);
        if (i >= j) break;
        swap = a[i];
        a[i] = a[j];
        a[j] = swap;
}
swap = a[i];
a[i] = a[right_position];
a[right_position] = swap;
```

循环执行把箭头的头部从左及从右移动，然后将箭头头部指向的值交换。如果箭头头部重叠，退出循环

退出循环后，将左箭头头部指向的值和最右面的值交换

在我们执行这个过程前，必须检查左位置不在右位置的右面，并且我们要初始化 i、j 和枢值。语句如下：

如果右箭头的尾部在左箭头尾部的右边，那么把枢值设定为右箭头尾部的值

```
if (right_position > left_position)
{
    pivot = a[right_position];
    i = left_position - 1;
    j = right_position;
}
```

初始化 i 等于左箭头尾部 - 1

初始化 j 等于右箭头尾部

我们可以把上面的语句加到条件语句中，得到下面的代码：

如果右箭头的尾部在左箭头尾部的左边，没必要排序子数列

初始化箭头头部和枢值

交换箭头头部指定的值

```
if (right_position > left_position)
{
        pivot = a[right_position];
        i = left_position - 1;
        j = right_position;
                while (1)
                {
                        while (a[++i] < pivot);
                        while (a[--j] > pivot);
                        if (i >= j) break;
                        swap = a[i];
                        a[i] = a[j];
                        a[j] = swap;
                }
                swap = a[i];
                a[i] = a[right_position];
                a[right_position] = swap;
}
```

将枢值移动到正确的位置

这些语句完成了一个完整的流程。在这个流程的结尾，我们已经把枢值放到了正确的位置，并且生成了左子数列和右子数列。可以通过不同的方法完成，这里使用递归。因为我们在左右两个子数列上执行这一过程，所以需要递归调用两次，而不是我们前面例子中描述的一次。

像以前的一次递归调用，我们必须考虑调用过程和返回过程。但是，利用两个递归调用，情况并不是很直接。图 8-10 演示了两个递归调用（显示了在几个调用以后每一个调用的回退）的函数的流程。这是一个重要的图示。跟随其中的数字，并理解为什么有两个递归调用的函数会这样工作。注意每一个函数递归调用自己两次（除非它开始回退，也就是说，一个 if 控制语句使得它跳过两次调用。）

对于我们的函数，首先放入递归调用然后跟随程序的流程。调用 qksort 的源代码。注意这个函数中的两个递归调用。第一个调用处理左子列，第二个调用处理右子列。

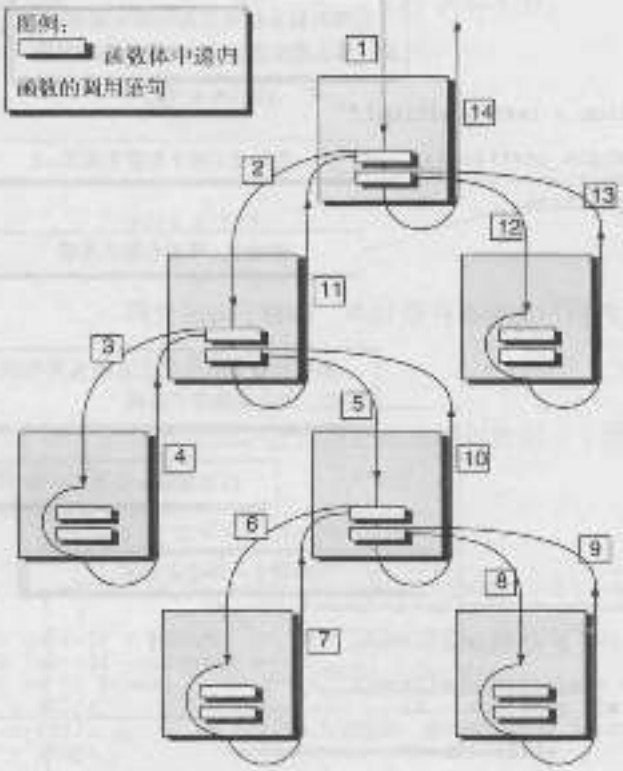


图 8-10 函数回调用程

```
int qksort(int a[ ], int left_position, int right_position)
{
        int pivot, i, j, swap;

        PREVIOUS CODE

            qksort(a, left_position, i-1);
            qksort(a, i+1, right_position);
            }
}
```

参数是数组以及左右指头尾部的位置

调用 qksort 处理左子列

调用 qksort 处理右子列

程序的流程如图 8-11 所示，一共有 39 次递归调用。图中每一个方块代表函数。函数中小的方块代表两个递归调用。在递归调用前的代码在一个子数列上执行一个完成的流程。这样每次调用都把一个值放到了正确的位置并生成了两个子数列（除了回退阶段）。因为第一个递归调用处理左数列，而第二个递归调用处理右数列，图中的方块从左向右布置以符合它们的调用顺序。

整个流程很复杂，在方块 6、7、8 和 9 中，处理数列左部分的函数被重复执行（从 6 到 7 到 8 到 9）。然后从 9 开始，数列左部分完成（因为 right_position>left_position）。回到 8，函数调用处理右子数列并返回。如果你将它与我们手工计算的过程相比较，会发现这个图和图 8-10 很近似。

完整的源代码，包括 main 函数如下：它把向量 list[] 中的数列进行排序。

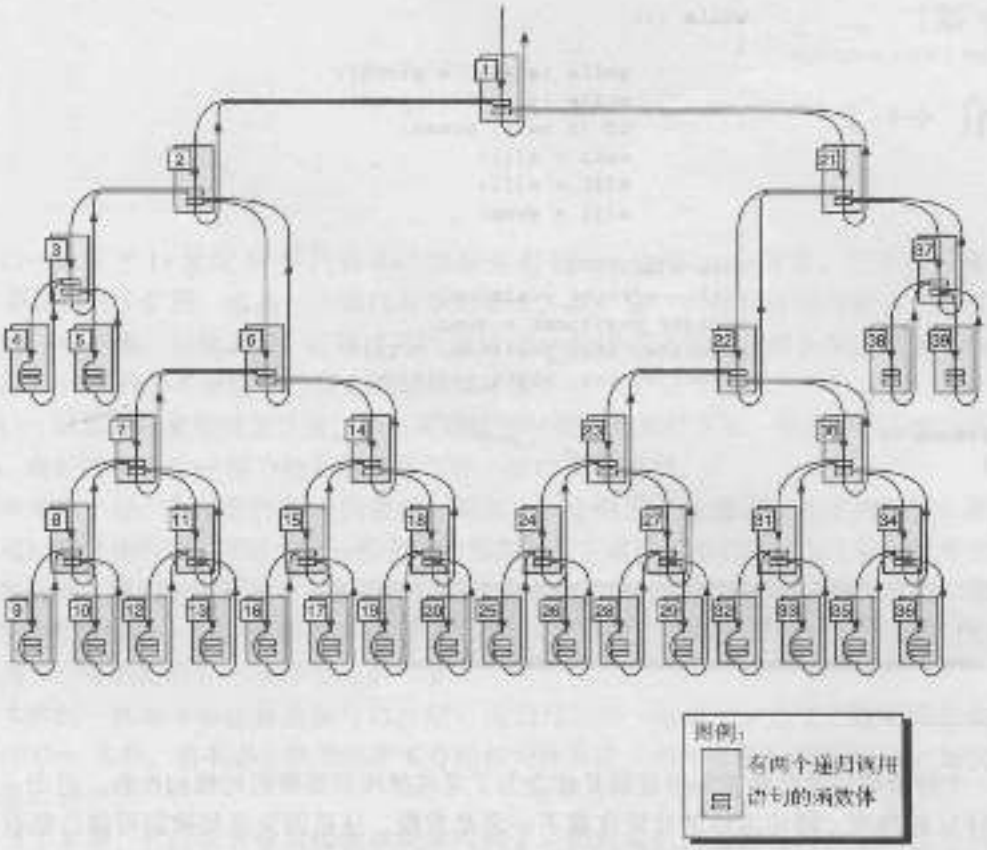


图 8-11 递归调用 qksort 的程序流程

源代码

```
#include <stdio.h>
#define SIZE 17
int qksort(int a[ ], int left_position, int right_position);

void main (void)
{
        int list[]={98,34,56,27,78,73,70,90,28,84,45,85,12,
10,54,34,29};
        int i;
        qksort(list, 0, SIZE-1);
        for (i=0; i<SIZE; i++)
                printf("%d ", list[i]);
        printf("\n");
}

int qksort(int a[ ], int left_position, int right position)
{
        int pivot, i, j, swap;
        if (right_position > left_position)
        {
                pivot = a[right_position];
                i = left_position-1;
                j = right_position;
```

```
                while (1)
                {
                        while (a[++i] < pivot);
                        while (a[--j] > pivot);
                        if (i >= j) break;
                        swap = a[i];
                        a[i] = a[j];
                        a[j] = swap;
                }
                swap = a[i];
                a[i] = a[right_position];
                a[right_position] = swap;
                qksort(a, left_position, i-1);
                qksort(a, i+1, right_position);
        }
return 0;
}
```

输出

```
12 18 27 28 29 34 34 45 54 56 70 73 78 84 85 90 98
```

注释

一个评价排序算法性能的方法就是检查为了完成排序而需要的比较的次数。对于一个给定数目 N 的数列，特定方法的比较次数不一定是常数，这是因为有些数列可能已经有很好的顺序性了。例如对于一个快速排序来说，一个最理想的布局就是左右两个子数列有相同的尺寸。分析的方法超出了本书的范围。但是，我们可以说对于快速排序来说，平均需要的比较次数是 $N\log_2N$。对于冒泡排序来说，它是 N^2。所以，如果在一个数列中有 10 000 个值，快速排序平均需要 $10\ 000\log_2 10\ 000 \approx 132\ 900$ 次比较，而冒泡排序需要 $10\ 000^2=1$ 亿次比较。显然快速排序比冒泡排序要好。

对于提高快速排序算法的建议已经有很多了。一个比较流行的方法是不用最右边的值作为枢值。如果使用左、中、右三个值的中位数，可以增大平均分割子数列的可能性。这就提高了快速排序算法的效率。

本章回顾

这章涉及一些 C 语言高级编程技巧。我们学习了如何声明和操作一个结构。结构是 C 语言的一个特性，它使得我们管理不同的数据类型。我们学习了如何写递归程序，即一个函数调用自身，也是一种迭代的方法。然后检查了四个不同的存储类：外部、静态、自动和寄存器；讲述了多个源文件如何帮助我们开发一个大型的系统。

作为一个高级编程方法的补充，我们学习了位操作以及如何将整数数据中某个特定位置 1 或清 0。它们是在底层的电子电路编程中非常重要的一些操作，例如操作系统的各种设备驱动程序的开发。

以上这些技术对于你将来成为一个熟练的 C 程序员都是非常有帮助的。

C++ 介绍

C++ 语言于 19 世纪 80 年代在 Bell 实验室由 Bjarne Stroustrup 开发。它不仅仅是强大的 C 语言的一个扩展，也是一个面向对象的语言，而 C 是一个面向过程的语言。同时 C 是 C++ 的一个子集。也就是说，C 程序可以使用 C++ 编译器来编译，但是反过来就不行。这意味着 C++ 的程序不能使用 C 语言的编译器来编译。

C++ 以面向对象的理念开发，所以可以使用封装、继承和多态。现在我们不想过多涉及细节。最好的理解 C++ 能力的办法就是考察一些 C++ 的程序。

本章并不是一个完整的 C++ 的描述。相反，只介绍了 C++ 语言的主要的特性，给你一个大概的感觉和印象，同时给你一些例子演示如何用它解决实际问题。为了在一章中完成所有这些任务，我们有意回避了一些细节，而且也并没有全部遵循标准的方法。目前，还没有像 ANSI C 那样的一个 C++ 的标准存在。尽管有这些缺点，当阅读完本章后，你也已经迈出了成为一个有成就的 C++ 程序员的第一步。

本章的一些练习会让你重新写以前用 C 语言写过的一些程序。这种方法可以让你更多地关注 C++ 本身，而不必去熟悉程序本身的结构和算法，另外也帮助你理解 C++ 如何增强和改进 C 语言。

到了本章，你应该有能力阅读和理解代码了。因此我们将限制每课中介绍性注释的数量，希望你能通读所有的代码及代码的各种注释。在阅读解释的时候参考代码，并完成练习。

课程 9.1　C++ 注释和基本输入输出流

主题

- 写 C++ 注释
- 使用标准输入输出流

让我们看第一个 C++ 程序。本课程序显示了如何写一个 C++ 的注释，从键盘接受输入以及在屏幕上显示输出。读源代码和每一个注解看看 C++ 如何完成注释和输入 / 输出（I/O）语句。

源代码

C++ 注释以双斜杠符号开头

```
// This is a C++ comment
/* C++ supports C comments */
```

因为 C 是 C++ 的一个子集，我们也可以在 C++ 中使用 C 的注释

```
/* This is a C++ comment enclosed in // a C comments, it is acceptable but
   we recommend that you not mix C++ and C comments */

#include <stdio.h>
#include <iostream.h>                    // for cout, cin

void main(void)
{
    int     age=21;
    float   units = 16.0;
    char*   name="Greg";

    printf("1. This is C++!\n");

    cout << "2. This is C++!\n\n";

    // Display Greg's age and units

    cout << name << " is " << age << " years old and his units are "<< units;

    // Get data from the input stream

    cout <<"\n\nPlease type your name and the number of units you have:";
    cin  >> name >> units;
    cout<<"\nYour name is " << name <<" and your units are " << units;

}
```

C++中输入/输出需要包含iostream.h

我们可以将C++的注释包含在C风格的注释中（/*和*/），这允许我们在调试的时候容易地将一部分代码注释掉而不用担心嵌套注释的问题

C++在行的末尾加注释是很方便的

C语言使用printf函数输出

C++使用cout和<<运算符方法输出

你可以在一个语句中使用很多<<

在<<之间只允许一个变量或字符串，不需要通过使用%d或%f来指定输出变量的类型

不使用scanf函数，C++使用>>运输符读取键盘上的输入

输出

```
1. This is C++!
2. This is C++!

Greg is 21 years old and his units are 16.0

Please type your name and the number of units you have:

Linda 15.0
Your name is Linda and your units are 15.0
```

键盘输入

解释

1）如何写C++的注释？我们以双斜杠符号开始的一个字符串作为C++的注释语句。任何符号后面的内容（除非符号在一个字符串的内部）一直到这行的末尾都当成是注释。例如，

```
// This is a C++ comment
```

就是一个C++的注释（见图9-1）。

C	C++
/*This is a C comment*/ #include <stdio.h> ..... printf("Please type your name"); scanf("%s",name);	//This is a C++ comment #include <iostream.h> ..... cout << "Please type your name"; cin >> name; .....

图 9-1 C++ 的第一印象

注意 C++ 也支持 C 语言模式的注释，你可以将 C++ 的注释包含在 C 的注释里面。但是，推荐你对于 C++ 程序写 C++ 风格的注释，对于 C 程序写 C 风格的注释。另外注意，如果一个注释延伸了很多行，那么每一行的开头都要以双 // 开始。

2）什么是 iostream.h？什么是流？在 C 语言中，标准的输入/输出方法在头文件 stdio.h 中定义。在 C++ 语言中，等价的头文件是 iostream.h。流被定义在这个头文件中。简单来说，流可以被当成是与外部设备如键盘、屏幕和磁盘驱动等通过 C++I/O 系统连接起来的内存单元。我们可以通过填充或读取这些内存单元与这些外部设备通信。在 C++ 中，I/O 的概念使得我们可以读取或者填充一个流而不去关心和它连接的设备。这样不管是和键盘还是和磁盘驱动通信，从程序员的角度来说，都是一样的。

cout 和 cin 都是标识符，它们在 iostream.h 中定义。默认分别指向标准输出设备（屏幕）和标准输入设备（键盘）。这些流在 C++ 程序执行的时候是自动打开的，它们对我们所写的任何程序来说都是可用的。

3）C++ 中如何输出到屏幕？在 C++ 中，依然可以使用 C 函数 printf() 来将显示输出到屏幕。但是也能使用 cout 和 << 运算符来直接完成这个任务（见图 9-2）。运算符 << 定义在 C++ 中，没有定义在 C 语言中。它叫插入运算符。这个运算符把右边操作数输入的数据送到它应该去的地方（cout 流）。例如语句

```
cout << "2. This is C++!\n\n";
```

将字符串 "2. This is C++!\n\n" 送到 cout 流，然后自动显示到屏幕。

注意你只能将内建的数据通过 << 发送到 cout，如 char，short，int，long，char*（string），float，double，long double 或者 void*。我们不需要使用格式字符串。I/O 系统非常智能，它能自动识别出不同的数据类型并正确地显示它们。

你可以使用多于一个 << 运算符来将不同类型的数据输出到 cout。在两个邻接的 << 运算符之间，你只能插入一个数据项（表达式或字符串）。运算符的结合性是从左到右。例如，语句

```
cout << name << " is " << age << " years old and his units
                    are " << units;
```

显示字符串（name）、一个整数（int）、一个浮点数（units）和一些字符串常量。输出流显示 "Greg is 21 years old and his units are 16.3" 到屏幕。

4）如何从键盘读入输入？在 C++ 中，你可以使用 C 语言中的 scanf 函数来读入键盘的输入。但是你也可以使用 cin 和 >> 运算符来完成这个任务（见图 9-2）。>> 符号定义在 C++ 中，没有定义在 C 语言中。它也叫抽取符。例如语句

```
cin >> name >> units;
```

把键盘输入的值（一个字符串和一个 double 数据项）自动放到 cin 流中，然后传递给变量 name 和 units。

注意你只能将内建的数据通过 >> 发送到 cin，如 char，short，int，long，char*（string），float，double，long double 或者 void*。我们不需要使用格式字符串。I/O 系统非常智能，它能自动识别出不同的数据类型并正确地读入它们。注意你可以在 cin 后面使用多于一个 >> 运算符来输入不同类型的数据。

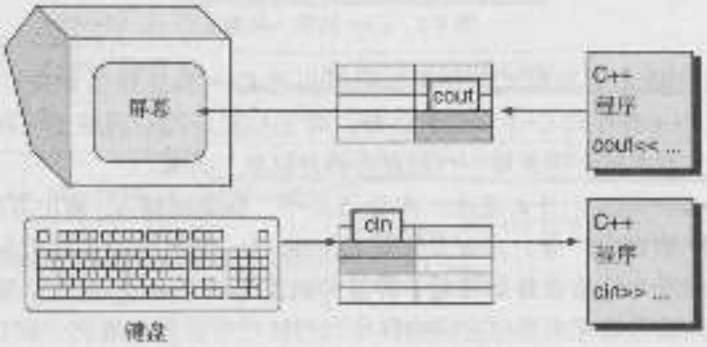


图 9-2 使用 cout、cin << 和 >> 输入输出

概念回顾

1）C++ 的注释以两个斜杠符号 // 开始。符号后的任何内容直到行末尾都是注释。

2）C++ 中与 stdio.h 等价的文件是 iostream.h。

3）C++ 使用 cout 和 << 运算符输出，例如，

```
cout << "Things to be out\n";
```

4）从键盘的输入采用和 cout 相同的方式处理。

```
cin >> var1 >> var2;
```

练习

1. 将课程 2.2 的程序中的 C 语言注释用 C++ 注释重写。
2. 使用 C++ I/O 流从键盘读入下面的数据并输出到屏幕。

```
char      'A'
int       123
long      987654
double    3.141592
char[20]  Welcome to C++!
```

课程 9.2 格式操纵符及格式化输出

主题

- 格式操纵符
- 基本的 iostream 类

在上面的课程中使用了 C++ I/O 流 cin 和 cout 以及 << 和 >> 执行基本的标准输入和输

由。它们非常易于使用，但是默认的格式也许并不是你想要的。例如，假设你想将 pi 显示成有三个小数位。我们怎么做呢？检查下面的代码，你会发现，只要加上一些语句就可以以任何格式显示数据。我们引入了两个新词：格式操纵符和类。

源代码

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
        int    ninety = 90;
        double pi = 3.141592654;

        cout << "Using manipulators to control format ----------------------\n\n";
        cout <<"Ninety in decimal  (default) is " <<        ninety << "\n";
        cout <<"Ninety in octal is        " << oct << ninety << "\n";
        cout <<"Ninety in hexadecimal is     " << hex << ninety <<"\n\n";

        cout << "Using parameterised manipulators to control format -------\n\n";
        cout << "1. PI="                                << pi << endl;
        cout << "2. PI=" << setw(15)                    << pi << endl;
        cout << "3. PI=" << setprecision(3)             << pi << endl;
        cout << "4. PI=" << setw(20) << setfill ('*')   << pi << endl;

        cout << "5. PI=" <<setiosflags(ios::left) << setw(20)        << pi << endl;
        cout << "6. PI=" <<setiosflags(ios::scientific | ios::showpos)<< pi << endl;
        cout << "7. PI=" <<setiosflags(ios::fixed) << setprecision(5) << pi << endl;

}
```

为了格式化输出，需要 iomanip.h

不用格式操纵符显示 ninety

将格式操纵符插入到流中以改变输出

endl 生成一个换行，它类似于 "\n"

使用带有标志的参数化格式操纵符

标志类

标志

可以通过使用 | 运算符来使用多于一个标志

域解析符 (::) 在类和标志之间使用

输出

```
Using manipulators to control format --------------------

Ninety in decimal (default) is 90
Ninety in octal is              132
Ninety in hexadecimal is        5a

Using parameterised manipulators to control format -------

1. PI=3.141593
2. PI=        3.141593
3. PI=3.142
4. PI=***************3.142
5. PI=3.142***************
6. PI=+3.142e+00
7. PI=+3.14159
```

解释

1）什么是流的格式操纵符？一个流的格式操纵符是一个特殊的函数/运算符，它们只能用在cout、cin以及<<和>>运输符上。流的格式操纵符不一定有参数。没有参数的格式操纵符不需要括号。

2）如何使用流的格式操纵符？我们把它们放到一个cin和cout语句中，紧邻着<<或者>>运算符。例如，为了将整数90显示为八进制，我们将流的格式操纵符如下插入到语句中。

```
cout <<"Ninety in octal is " << oct << ninety << "\n";
```

这个语句使用格式操纵符oct来把整数参数从默认的十进制更改为八进制。

其他有用的不需要参数来指定它们行为的流格式操纵符如下：

- dec，将转换格式设置为十进制
- hex，将转换格式设置为十六进制
- endl，插入一个换行符并冲洗流

关于不需要参数来指定行为的流格式操纵符的完整列表，请参考你的C++编译器的手册。

3）如何改变cout中默认的域宽？在输出流中插入一个参数的格式操纵符setw()，来改变默认的域宽。例如语句

```
cout << "2. PI=" << setw(15)<< pi << endl;
```

使用格式操纵符setw(15)将默认域宽改变为15。setw的参数是一个整型数。参数化的格式操纵符在头文件iomanip.h中声明。如果你想在程序中使用参数化的格式操纵符，要包含这个头文件。

其他有用的参数格式操纵符如下：

格式操纵符	动作	例子
setfill (int f)	将填充字符设置为f	setfill ("*")
setprecision(int p)	将浮点数精度设置为p	setprecision(3)
setw(int w)	将域宽设置为w	setw(20)
setiosflags(long f)	设置格式标志f	setiosflags(ios::left)

在下面的例子中，left被认为是一个标志，本课用到的标志如下：

标志类型	用法
left	左对齐输出
fixed	对浮点数使用规定小数点符号
scientific	对浮点数使用科学计数符号
showpos	对于正数显示+号

为了在一个参数化的格式操纵符中使用多个标志，你可以每次使用带有一个标志的格式操纵符。你也可以通过|运算符来组合这些标志。例如，对于格式操纵符setiosflags()，

```
cout << "6. PI=" <<setiosflags(ios::scientific |
        ios::showpos)<< pi << endl;
```

我们使用|运算符来组合科学计数法标志和显示符号标志。

对于需要参数来指定行为的流格式操纵符的完整列表，请参考你的C++编译器的手册。

4）ios::scientific符号是什么？C++流定义在流库中，由它们的类以及定制的插入和抽

取运算符所确定。在后面会详细讨论 C++ 的类，现在记住 C++ 类包含数据以及管理这些数据的函数。一个 C++ 的类有点像 C 语言中的结构，只是类不仅包含数据也包含函数。注解中包含 C++ I/O 的一个类，ios。格式标志如 scientific、left 或 showpos，是一个定义在 ios 类中的枚举（意味着它代表一个 int 类型值）。为了使用这些标志，我们首先写一个类名 ios，后接域解析符（::），然后是标志名字。域解析符代表这个标志是它前面的类中的一个成员。

概念回顾

1）一个流的格式操纵符是用在输入输出操作的一个特殊的函数/运算符。例如 oct、dec、endl 等。它们通常在数据流上执行一些附加的操作，例如数字基数的转换。

2）参数化的格式操纵符能执行更复杂的功能，如设置打印及输出的域宽。

3）域解析符用来引出 C++ 类中一个特定的值或成员。这个概念类似于结构的成员运算符。C++ 类的细节后面介绍。

练习

1. 使用 C++ 无参数格式操纵符重写课程 3.1 程序。
2. 使用 C++ 参数格式操纵符重写课程 3.2 程序。
3. 对于任何数据类型，使用 C++ I/O 流从键盘读入下面的数据（注意你需要使用 resetiosflags(long f) 来清除掉以前用 f 设置的标志，本文并没有介绍如何使用 resetiosflags(long f)，所以你必须做一些实验来使用它）。

```
char            'A'
int             123
long            987654
double          3.141592
char[20]        Welcome to C++!
```

在屏幕上按如下格式显示数据：

```
12345678901234567890123456789012345678901234567890
Char        A                 A*********
Int         123               +123******
Long        987654            987654****
Double      3.141592          3.14e+00**
char[20]    Welcome to C++!   **********Welcome to C++!*****
```

课程 9.3　函数重载

主题

- 重载的函数

假设你想写一个函数来将两个数字相加，另外一个函数将一个字符串连接到另外一个字符串。在 C 语言中，你必须给它们两个不同的名字，例如 add_num 和 add_string，即使它们都是将两个值合并成一个。但是在 C++ 中，你可以给它们相同的名字，如 add。这意味着你可以重载两个不同的函数并给予它一个通用的名字。通过这么做，你可以通用或标准化执行相同任务的函数。这使得你的程序更加易于管理和理解。你可以重载任何多的函数。函数重载是 C++ 中一个有用的特性。在下面的程序中，我们看函数是如何被重载的。

源代码

这三个函数有相同的名字，它们返回 int、double 和 char

两个参数

三个参数

两个参数

用两个参数调用 add

用两个 char* 参数调用 add

用三个参数调用 add

add 有三个定义，所以 add 是一个重载函数

```
#include <iostream.h>
int     add   (int  a,   int b );
double  add   (int  a,   float b,   double c);
char*    add  (char *a,  char *b  );

void main()
{
        cout << "1a. add(44 ,   55) =" << add(44 ,   55) << endl;
        cout << "1b. add(1.2, 3.4) =" << add(1.2, 3.4) << endl <<endl;

        cout << "2. add(1, 2.3, 3 .4E+1) = " << add(1, 2.3, 3.4E+1) << endl <<endl;

        cout << "3. add(""Good "","Day!"") =" << add("Good ", "Day!");
}

int add(int a, int b)
{
        return (a+b);
}

double add(int a, float b, double c)
{
        return (double)(a+b+c);
}

#include <string.h>
char  *add(char *a, char *b)
{
        char  ab[200];
        strcpy(ab,a);
        strcat(ab,b);
        return(a,b);
}
```

输出

```
1a. add(44,   55) =99
1b. add(1.2, 3.4) =4

2. add(1, 2.3, 3.4E+1) = 37.3

3. add("Good", "Day!") =Good Day!
```

解释

1）什么是函数重载？函数重载是 C++ 的一个编程技术，它可以对一个给定的函数名字提供多于一种定义。这个 C++ 特性允许你使用相同的名字去开发执行相似任务的函数。C++ 编译器决定哪个函数应该调用。例如在程序中有三个函数，第一个将两个 int 相加，第二个将 int、float 和 double 相加，第三个连接两个字符串。所有这三个函数都有相同的名字 add。因此 add 函数是重载函数（见图 9-3）。

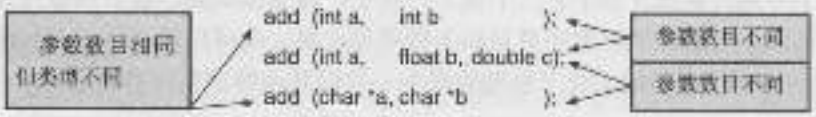


图 9-3 一个重载函数

2）如何调用重载函数？为了调用重载函数，我们首先确定需要哪个重载函数，然后把正确的参数类型和参数数目放到函数调用中。例如，如果将两个数相加，我们放两个数字参数，如

```
add(44,   55)
```

或者

```
add(1.2, 3.4)
```

来调用函数。如果我们想连接两个字符串，我们放两个字符串参数，如

```
add("Good ", "Day!")
```

来调用函数。参数的类型和数目要匹配至少一个函数的定义。如果没有匹配或者有多于一个匹配，都会从编译器接受到一个错误的信息。

3）写一个重载函数的主要限制是什么？不是所有的函数都可以重载。在开发重载函数的时候有一些规则和限制。主要的限制在于重载函数必须有不同的参数列表（参数的数目或者类型必须是不同的）。没有了这个限制，C++ 编译器就不能区分出哪个函数应该被调用来执行运算。例如本课重载函数的原型是：

```
int       add (int   a,    int b                 );
double    add (int   a,    float b,    double c);
char     *add (char *a,    char *b               );
```

显然，在这个集合中任何两个函数的参数都是没有歧义的。但是，如果我们把第二个函数原型改成了

```
double add(float b, double c);
```

那么在第一个和第二个函数的参数中就存在歧义了，即使它们有不同的返回类型和参数类别。例如，如果我们使用语句

```
cout<<add(1.2,3.4);
```

那么 C++ 编译器也许会把浮点数变为 int 然后调用函数

```
cout<<add(1 ,3);
```

此时，C++ 编译器无法选择使用哪个函数，所以它会产生一个错误。通常当你开发一个重载函数的集合时，应确保不同函数的参数之间没有歧义性。

概念回顾

函数重载允许你使用相同的名字去开发执行相似任务的函数。因此函数调用时参数的名字和类型必须与你要调用的函数完全一致。

练习

1. 在程序中开发重载函数以便发现：
   a. 三个数中的最大值。
   b. 一个字符串中按字母排序的最后且最长的词。
   使用下列数据作为输入：

```
1 23 4.56
Have a nice day !
100 3.14 999.9
```

程序显示下列输出：

```
The maximum of 1, 23 and 4.56 is 23
The longest and last alphabetical word in "Have a nice day
!" is "nice"
The maximum of 100, 3.14 and 999.9 is 999.9
```

2. 在程序中开发重载函数进行排序。

a. 有四个元素的 int 数组升序排列

b. 有五个元素的浮点数组降序排列

使用下列数据作为输入：

```
33 22 11 55
8.8 6.6 9.9 7.7 5.5
```

程序显示下列输出：

```
11 22 33 55
9.9 8.8 7.7 6.6 5.5
```

课程 9.4 默认函数参数

主题

- 有默认参数的函数

有时，你需要在程序中以同样的参数反复地调用一个函数。这么做不小心就会输入一个错误的参数。为了避免这种错误，C++ 提供了一种解决方案，允许你不使用任何参数去调用函数。本课程序中函数原型有三个参数，但是你可以用三个、两个、一个或者不用参数去调用它。检查这个程序，找出如何不用参数去调用函数。

源代码

add 函数有三个默认参数

```
#include <iostream.h>
double  add (int aa=10,   double bb=20.0,   char *cc=" aa plus bb");
int     subtract (int xx,   int yy=11,   int zz=22);
// Wrong --- int subtract(int xx=11, int yy, int zz);

void main(void)
{
```

subtract 函数有两个默认参数

用少于函数原型定义的参数数目去调用这个函数的时候，使得最左边的参数被传递，默认的值用在其他的参数上

```
        cout << "1. add(  ) = "          << add(  );
        cout << "2. add(30  ) = "        << add(30 );
        cout << "3. add(40, 50  ) = "    << add(40, 50 );
        cout << "4. add(60, 70.0," "Result""(= " << add(60, 70.0,"Result")<<endl;

        cout << "5. subtract(77    ) = " << subtract(77 );
        cout << "6. subtract(99, 44) = " << subtract(99, 44) ;
}
double add(int a, double b, char *c)
{
        cout << endl << endl;
        cout << "aa = " << a << ",          ";
        cout << "bb = " << b << ",          ";
        cout << "cc = " << c << endl;

        return (a+b);
```

add 函数返回前两个参数的和

```
}
int subtract(int x, int y, int z)
{
        cout << "\n\n";
        cout << "xx = " << x << ",          ";
        cout << "yy = " << y << ",          ";
        cout << "zz = " << z << endl;

        return (x-y-z);
}
```

subtract 函数对三个整数做减法

输出

```
aa = 10,      bb = 20,      cc =  aa plus bb
1. add(   ) = 30

aa = 30,      bb = 20,      cc =  aa plus bb
2. add(30 ) = 50

aa = 40,      bb = 50,      cc =  aa plus bb
3. add(40, 50  ) = 90

aa = 60,      bb = 70,      cc = Result
4. add(60, 70.0,Result)= 130

xx = 77,      yy = 11,      zz = 22
5. subtract(77 ) = 44

xx = 99,      yy = 44,      zz = 22
6. subtract(99, 44) = 33
```

解释

1）如何不使用要求的参数去调用一个函数？为了不用要求的参数数目去调用一个函数，需要写一个有默认参数的函数。默认的参数在原型中用默认的值来初始化。例如函数原型

```
double  add (int aa=10,  double bb=20.0, char *cc=" aa
                        plus bb");
```

有三个形参：aa、bb 和 cc。每一个参数都用一个默认的值初始化。在函数调用时如果不提供实参的话，C++ 编译器会使用默认的值。但是如果提供实参的话，那么默认的值被覆盖。

上面的参数有三个默认参数，因此可以用一个参数、两个参数或者三个参数去调用它。注意，如果函数有多于一个默认参数，当我们调用这个函数的时候，不能任意放置实参然后假设编译器会在我们不放置实参的地方使用默认值。真实的规则是，如果想将某个默认值用用户定义值来代替的话，需要把它的左边所有的参数用用户定义值来代替默认值。例如在函数 add 中，如果我们想覆盖第二个参数（参数 bb），必须在第一个和第二个参数上使用用户定义的值，然后在第三个参数上使用默认值。下面的表格显示了当以不同的参数调用 add 函数时默认值是如何被使用的。

函数调用	默认参数
add()	aa, bb, cc
add(30)	bb, cc
add(30, 50)	cc
add(60, 70.0,"Result")	None

2）我们需要对所有的形参初始化默认值吗？不，我们只选择一些参数作为默认参数，

而其他的当成正常参数。但是不能随机选择。规则是如果我们想把第 n 个参数当成默认参数，那么第 n 个参数后面的所有的参数都是默认参数。例如在函数 subtract 中，选择了第二个参数 yy 作为默认参数，那么第三个参数也需要是一个默认参数（图 9-4）。如果我们定义了 subtract 如下：

```
int subtract(int xx=11, int yy, int zz);
```

那么 C++ 编译器在编译的时候会产生一个错误信息，因为第一个参数是默认的参数，但是第二个和第三个参数不是默认的参数。

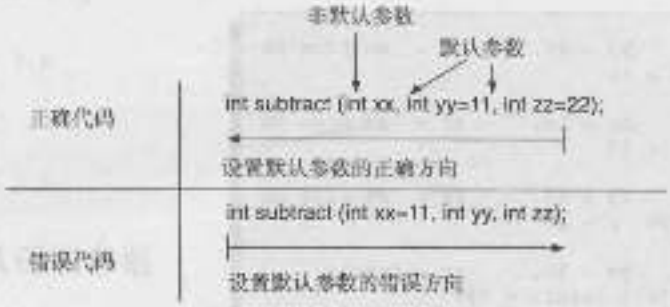


图 9-4 默认函数参数

概念回顾

C++ 函数中，一个默认的参数使用一个默认的值在原型中来初始化。例如，

```
double  add (int aa=10,  double bb=20.0, char *cc=" aa
                    plus bb");
```

练习

1. 使用默认参数开发程序：

a. 生成以下的默认输出：

```
ABCD CORPORATION
Project _________   Contract No. 3815-A   FileNo.
Designed: JKR       Checked ___________   Date _____/2011
```

b. 从用户输入 Project、File No.、Checked 和 Date 数据。

c. 生成非默认输出：

```
ABCD CORPORATION
Project USA-OIL-1  Contract No. 3815-A  File No. OIL-A12345
Designed: JKR      Checked John & Ken  Date 12/13/2011
```

2. 使用默认参数开发程序：

a. 从用户得到输入数据的文件名。如果用户只是按回车键，默认的文件名为"INPUT.DAT"。

b. 从文件中读入数据。文件有 4 列代表点数以及 X、Y 和 Z 的坐标。列数可能有变化，输入的数据如下：

```
No.     X        Y        Z
 1    755.0    221.9    696.4
 2    744.4    204.3    698.6
 3    743.1    206.8    689.9
 4    734.0    225.4    701.3
```

c. 要求用户输入两个点的数字，例如，如果用户输入34，那么计算点3和点4的距离。但是如果用户只是按回车，那么计算1和2、2和3、3和4、4和1之间的距离。

d. 产生整洁的屏幕输出。

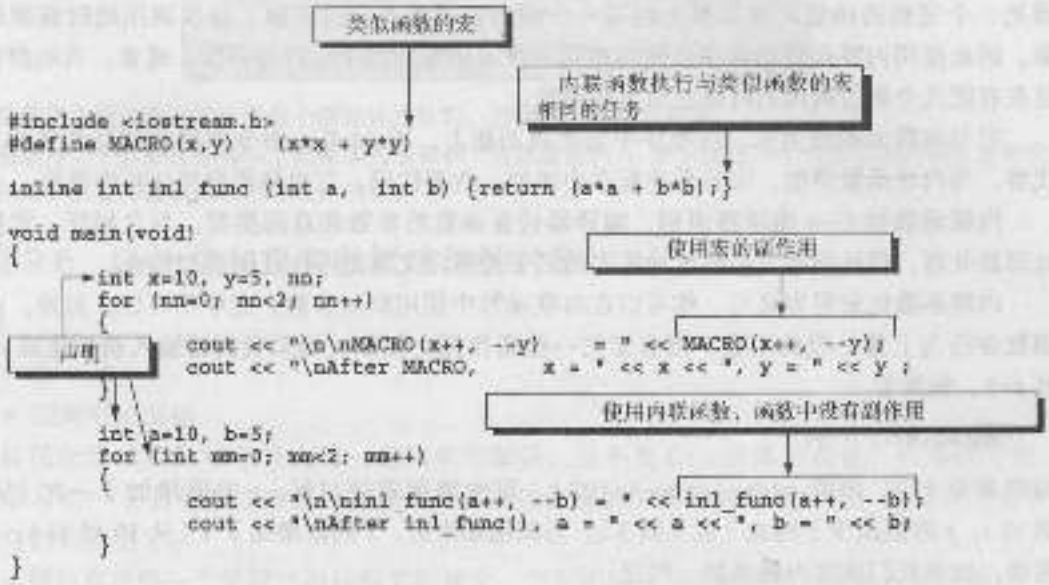
课程 9.5 内联函数和变量声明的位置

主题

- 内联函数
- 变量声明的位置

本课检查了C++对C语言的两个方面的加强。一个是内联函数和宏的比较，另外一个是放置变量声明的灵活性。查看下面程序中的内联函数和宏。你能看出内联函数对于类似函数的宏有哪些优点和缺点吗？你能发现程序中变量不是在代码开头声明的吗？

源代码




```
#include <iostream.h>
#define MACRO(x,y)    (x*x + y*y)

inline int inl func (int a,  int b) {return (a*a + b*b);}

void main(void)
{
        int x=10, y=5, nn;
        for (nn=0; nn<2; nn++)
        {
                cout << "\n\nMACRO(x++, --y)     = " << MACRO(x++, --y);
                cout << "\nAfter MACRO,      x = " << x << ", y = " << y ;
        }
        int a=10, b=5;
        for (int mm=0; mm<2; mm++)
        {
                cout << "\n\ninl func(a++, --b) = " << inl_func(a++, --b);
                cout << "\nAfter inl_func(), a = " << a << ", b = " << b;
        }
}
```

输出

```
MACRO(x++, --y)     = 122
After MACRO,      x = 12, y = 3

MACRO(x++, --y)     = 158
After MACRO,      x = 14, y = 1

inl_func(a++, --b) = 116
After inl_func(), a = 11, b = 4

inl_func(a++, --b) = 130
After inl_func(), a = 12, b = 3
```

解释

1）什么是内联函数？一个内联的函数就是一个正规的函数，除了它用关键字 inline

来做它的修饰符。这意味着函数原型中出现的第一个词是 inline，后跟整个函数体（如图 9-5）。例如语句

```
inline int inl_func (int a, int b) {return (a*a + b*b);}
```

声明 inl_func 为一个内联的函数。它是 int 类型，有两个参数 a 和 b，返回 a*a+b*b 给调用它的函数。

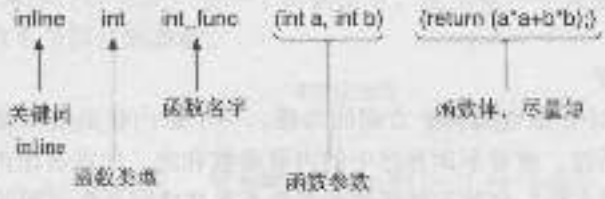


图 9-5 一个内联函数

2）内联函数和宏在你的程序中有什么用？对于内联函数，inline 关键字使得 C++ 编译器把一个完整的函数的拷贝插入到每一个调用它的地方。这省掉了每次调用的时候装载参数。因此使用内联函数的程序比使用相同的普通函数的程序运行得要快。通常，当函数很短且在有限几个地方调用的时候使用内联函数。

宏与内联函数很类似。在程序中宏出现的地方，用 #define 命令声明的符号名字被文本代替。与内联函数类似，宏只是在程序中增加一些源代码，它也使得程序运行得更快。

内联函数被 C++ 编译器识别，编译器检查函数的参数和返回类型。与之相反，宏被预处理器处理。预处理器只是简单地将宏的名字替换为文本而不作任何类型检查。

内联函数比宏更容易写。你可以在内联函数中使用默认参数，宏中不可以。另外，内联函数在行为上像一般的函数，没有宏的一些副作用。例如，如果我们把输入初始化成 $x=10$ 且 $y=5$，触发宏

```
MACRO(x++, --y)
```

的结果是 122，即值（$10\times11+4\times3=122$）。每次调用宏的时候，x 的值增加了一次（从 10 到 11），y 的值减少了两次（从 5 到 3）。当调用结束后，x 的值增加了 1（从 10 到 11）。但是，如果我们使用内联函数，调用

```
inl_func(a++, --b)
```

结果是 116，即值（$10\times10+4\times4=116$）。每次 inl_func() 被调用，a 的值没有改变（10），b 的值减 1（从 5 到 4）。当调用结束后，a 的值增加了 1（从 10 到 11）。对于 MACRO 和 inl_func() 中的乘法操作，我们可以看到副作用带来的差别是很大的。

3）我们可以在程序的任何地方声明变量吗？可以，只要位置是有意义的，并且把声明放到使用这个变量的前面。可以把声明放到代码的开头，如 x 和 y 的声明：

```
int x=10, y=5, nn;
```

或者可以把它们放到代码的某个地方中，如变量 a 和 b

```
int a=10, b=5;
```

另外一个方法就是把它们放到一个代码块中，如在一个 for 循环的内部声明 mm。

```
for (int mm=0; mm<2; mm++)
```

将变量的声明放到使用变量之前使得程序更加易读。但是如果我们随机在代码的任意地方放置声明，从 C++ 中获得的这种灵活性会给阅读程序的程序员增加麻烦。

概念回顾

1）对于内联函数，一个完整的函数体被插入到它调用的地方。因此函数运行的比普通函数快。

2）C++ 允许变量声明在使用它们之前的任何地方。

练习

1. 写程序读下面的数据：

圆锥体底部半径	圆锥体的高
10.1	66.6
20.2	55.5
30.3	44.4

然后使用内联函数和宏计算每个圆锥体的体积，并将它们输出到屏幕。

2. 使用练习 1 中数据计算每一个圆锥体的表面积（包括底面积）。要求使用一个主内联函数和主宏来完成计算，主宏可能调用任何你需要的宏。

课程 9.6 C++ 类和只有数据成员的对象

主题

- C++ 类
- 面向对象编程

目前介绍的都是对 C 语言的一些简单的加强，并不是 C++ 的本质内容。从本课开始，我们学习 C++ 语言的核心内容：类和对象。一旦你理解了类和对象是什么，你就会开始意识到它们的价值了。

本课的程序将一个值赋给结构和类的成员，然后把结构和类的成员的值输出到屏幕。

源代码

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

struct Bus
{
        char  colour[10];
        float price;
};

class Car
{
        public:
        char  colour[10];
        private:
        float price;
};
```

定义 Bus 结构

可以将 class 中成员设计成 public 或者 private，private 的成员不能被 class 外面的函数访问

class car 的定义

```
void main(void)
{
        Bus newbus, oldbus;

        strcpy(newbus.colour, "Red");
        cout << "newbus.colour = " << newbus.colour << endl;
        newbus.price = 1234.5;
        cout << "newbus.price = " << newbus.price << "\n\n";

        Car newcar, oldcar;

        strcpy(newcar.colour, "Blue");
        cout << "newcar.colour = " << newcar.colour << endl;

        //newcar.price = 1234.5;
        //cout << "newcar.price = " << newcar.price << "\n\n";
}
```

初始化并输出 Bus 结构变量

声明类 Car 的两个对象 newcar 和 oldcar

不能访问 newcar.price，因为它是一个 private 成员

输出

```
newbus.colour = Red
newbus.price = 1234.5

newcar.colour = Blue
```

解释

1）什么是类和对象？C++ 中的 class 是用户定义的数据类型，就像 C 中的用户定义数据类型 struct 一样。但是 class 比 struct 更强大。C 中的结构只能包含数据而 C++ 中的类可以包含数据和成员函数。成员函数管理数据成员。本课中讨论只包含数据成员的类。

一个类的实例就是一个对象，一个对象是一个包含数据和函数的实体。当使用类和对象的时候，我们的程序是面向对象的；我们正在写面向对象的程序。

2）如何定义一个类，如何在程序中声明一个对象？在 C++ 中定义类就像在 C 语言中定义结构。比较下面的两个：

```
struct Bus
{
   char  colour[10];
   float price;
};

class Car
{
   public:
   char  colour[10];
   private:
   float price;
};
```

它们之间的主要区别在于关键字 public 和 private。它们都包含数据成员 char 类型的 colour[10] 和 float 类型的 price。一旦你定义了 Bus 结构，就能够声明属于这个结构的变量。同样，一旦我们定义了 Car 类，也能声明属于这个类的对象。例如语句

```
Bus newbus, oldbus;
Car newcar, oldcar;
```

声明了属于这个 Bus 结构的变量 newbus、oldbus，以及属于 Car 类的对象 newcar、oldcar。

Bus 结构有两个数据成员。因此任何属于这个类型的变量也有两个数据成员。类似，Car 类也有两个数据成员。因此任何属于这个类的对象也有两个数据成员。如果正确处理，任何结构或类中的数据成员都可以像类似的 C++ 数据类型那样处理。

一个类中的数据成员被保存成两组（如图 9-6）：公有和私有（还有一组，保护组，我们这里不讨论）。类定义的公有和私有标签指定了类中数据成员的可用性。任何出现在指定标签后面的成员属于那个组。你可以将公有标签和私有标签按照需要以任何顺序放置任何多次，但是我们推荐你将它们分成两类，一般将数据成员设置为私有，把函数成员设置为公有（本程序中没有演示）。公有数据成员比私有数据成员有更高的可用性，这意味着在存取公有数据成员的时候比私有成员有更少的限制（见下面的解释）。

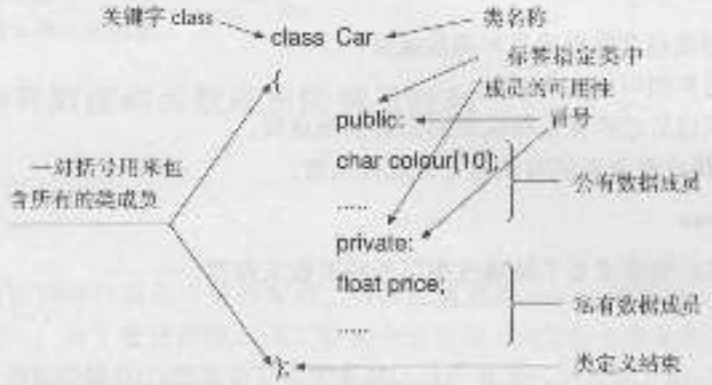


图 9-6 C++ 类定义

在 C++ 中，你也可以通过使用关键字 struct 和 union 来定义一个类。这意味着 C++ 中的 struct 和 union 比起 C 语言中的定义要更灵活一些，这是因为它们都可以既包含数据成员又包含函数成员。与类的主要的区别在于它们对成员的存取能力。

类的类型	默认存取类别	存取类别的修改
class	私有	用户可以更改
struct	公有	用户可以更改
union	公有	用户不可更改

注意，一个类能被定义在函数里面或者函数外面。如果定义在函数的里面，那么它是局部的；否则它是全局的。本课程序中的 Bus 和 Car 都是全局的。

类的默认的存取类别是私有的。如果没有存取类别指定，C++ 认为是私有。

3）如何存取一个类对象的公有数据成员？存取一个类对象的公有数据成员就像存取一个结构的数据成员一样。例如，为了存取结构变量 newbus 的数据成员 color 和 price，我们首先使用语句

```
strcpy(newbus.colour, "Red");
newbus.price = 1234.5;
```

初始化它们。然后数据成员可以像任何正规数据那样被管理。同样的方法可以用来存取公有数据成员。例如可以使用语句

```
strcpy(newcar.colour, "Blue");
cout << "newcar.colour = " << newcar.colour << endl;
```

初始化并且显示公有数据成员，并将对象 newcar 中的 colour 显示到屏幕。这个例子显示了公有数据成员可以被任何程序中任何函数访问（用户定义函数或者库函数）。

4）如何存取一个类对象的私有数据成员？在类 Car 中数据成员 price 是私有的。一个私有的数据成员不能被类外面的函数所访问。例如语句

```
//newcar.price = 1234.5;
//cout << "newcar.price = " << newcar.price << "\n\n";
```

是非法的，因为 price 是一个私有成员。它不能被赋值语句初始化，也不能用 main 中的 cout 语句输出到屏幕，因为 main 函数不是 Car 的成员。私有成员只能被属于相同类中的成员函数所访问。在下次课讨论成员函数。

概念回顾

1）C++ 中的类包含数据成员和函数成员。

2）一个类的实例叫做一个对象。

3）我们使用成员运算符来存取类的公有数据成员。

我们使用点类作符来连接对象和它的成员函数：

```
Object.member
```

4）一个私有的数据成员不能通过类以外的函数来存取。

练习

1. 在一个实验课上要求测量氧气的消耗，湿的豌豆和干的豌豆在给定湿度的呼吸计中呼吸时，测量气体容量的变化。下表显示了测量的结果：

温度(℃)	时间(分min)	干豌豆(x时间气体容量)	湿豌豆(x时间气体容量)
23	初始值，0	0.89	0.65
23	0～5	0.86	0.39
23	0～10	0.85	0.18
23	0～15	0.84	0.02
23	0～20	0.83	0.00

写一个程序显示这些测量的结果。程序应该

a. 从原始的测量数据文件中读入数据。

温度	时间	干豌豆	湿豌豆
23	0	0.89	0.65
23	5	0.86	0.39
23	10	0.85	0.18
23	15	0.84	0.02
23	20	0.83	0.00

使用包含下面数据成员的结构：

```
int temp;
int time[10];
double dry[10];
double soaked[10];
```

b. 使用一个类代替结构执行上面的过程。

2. 光可以被用来测量一个湖中污染物的变化。例如，在一个泥浆湖，光只能深入几英寸，但是一个清澈的湖，光可以深入更多。下面的表格给出了不同的湖，光深入的百分比：

入射光 (%)	湖1入射光深度 (米)	湖2入射光深度 (米)
100	0.0	0.0
65	1.5	2.4
25	2.3	12.0
10	7.4	25.3
2	12.0	30.2

写一个程序

a. 使用类读入数据。

b. 将深度的尺寸单位从英寸变为米。

c. 确定哪个湖更清澈。

d. 在屏幕上显示 b 和 c 的信息。

课程 9.7 带有数据和函数成员的类及封装

主题

- 成员函数
- 封装

前面课程讨论的类仅仅包含数据成员。当数据成员是 public 的时候这种类型的类和 C 语言中的结构类似。为了管理传统 C 语言结构中的数据（例如显示数据到屏幕），我们需要使用函数。数据和管理数据的函数是不同的实体，也没有显式的关联。换句话说，可以使用任何函数处理数据，或者使用一个函数处理任何数据。

但是在 C++ 中，数据成员和成员函数被链接在一起并被放到了一个类对象中。把数据和管理数据的成员链接到一个单独的对象中叫做封装。因为类链接了数据和函数，我们使用类的方法和使用结构的方法是不一样的。本课演示了如何通过类或对象调用一个和数据链接到一起的函数。它生成了三个对象：newcar、oldcar 和 mycar。这些对象的类是 Car。我们将数据成员赋值并输出。调用成员函数将某些值赋给数据成员。

源代码

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

class Car
{
        char  owner[11];

        public:
        char  colour[10];
        int   year made;
        void  get_info(char *who, int year, double cost);
        void  display(void);

        private:
        double price;
        double   sellcar (double sell_price);
```

因为这既不是公有函数也不是私有函数，它被给予默认的访问权限（私有）

类 Car 的公有成员

公有函数可以从任何函数中调用

Car 类的私有成员。这些成员只能在成员函数 get_info 和 display 中被存取

```
};

void main(void)
{
        Car newcar, oldcar, mycar;

        strcpy (newcar.colour, "Blue");
        cout << "newcar.colour      = " << newcar.colour << endl;
        newcar.get_info("Mary", 1998, 6543.2);
        newcar.display();

        strcpy (oldcar.colour, "White");
        cout << "oldcar.colour      = " << oldcar.colour << endl;
        oldcar.get_info("John", 1921, 1234.5);
        oldcar.display();

        oldcar.year_made=1934;
        mycar=oldcar;
        cout << "mycar.colour       = " << mycar.colour << endl;
        mycar.display();
}

void Car::get_info(char *who, int year, double cost)
{
        strcpy(owner, who);
        year_made = year;
        price = cost;
}

double Car::sellcar(double sell_price)
{
        if (sell_price < 5000) return (sell_price + 265.5 );
        else
        return (sell_price + 456.8 );
}
void Car::display(void)
{
        cout << "owner              = " << owner      << endl;
        cout << "year_made          = " << year_made  << endl ;
        cout << "price              = " << price      << endl;

        cout << "sellcar(price)     = " << sellcar(price) << "\n\n";
}
```

声明一个类 Car 的 newcar、oldcar 和 mycar 对象

在 main 函数中，能存取公有数据成员 colour 和 year_made

我们能调用公有函数 get_info 和 display。为了调用，我们必须通过一个声明的对象。本例中对象是 oldcar

通过一个简单的赋值语句可把一个对象拷贝到另外一个对象

在函数头中，必须指定 get_info、sellcar 和 display 是类 Car 的成员函数。原因在于在 C++ 中允许在不同的类中有相同名字的函数

因为 sellcar 是一个私有成员函数，我们只可以从其他的成员函数中调用它。这里从 Car::display 中调用它

在成员函数内部，我们不需要将函数调用与特定的对象关联。这是因为在函数的调用方已经指定了一个对象。注意在 main 中调用 oldcar.display，在调用方函数已经指定了对象 oldcar

输出

```
newcar.colour      = Blue
owner              = Mary
year_made          = 1998
price              = 6543.2
sellcar(price)     = 7000

oldcar.colour      = White
owner              = John
year_made          = 1921
price              = 1234.5
sellcar(price)     = 1500

mycar.colour       = White
owner              = John
year_made          = 1934
price              = 1234.5
sellcar(price)     = 1500
```

解释

1）什么是类的成员函数？类的成员函数是C++类中的一个重要的组成部分。记住在类定义中定义的函数属于这个类。例如，类定义

```
class Car
{
    char  owner[++];

    public:
    char  colour[10];
    int   year_made;
    void  get_info(char *who, int year, double cost);
    void  display(void);

    private:
    double price;
    double   sellcar (double sell_price);
};
```

定义了三个成员函数get_info()、display和sellcar()。类成员函数的原型与任何C或者C++一般函数的原型一致。一个成员函数可以有任何数量的任何合法数据类型的形参，并且可以返回任何类型的数据给调用方函数。例如，get_info()成员函数是一个void类型的函数并有三个形参：who、year和cost。sellcar()函数是一个double类型并且只包含一个形参，sell_price。display函数是void类型并且不包含形参。

与类的数据成员类似，类的成员函数被分类为公有或者私有。例如，get_info()和display()函数是公有，但是sell_car()函数是私有。

2）如何调用公有函数和私有函数？像任何其他一般的函数，一个公有函数可以从程序的任何地方调用。为了调用公有函数，不仅需要提供一个函数名，也需要提供一个包含这个函数的对象。例如语句

```
newcar.get_info("Mary", 1998, 6543.2);
oldcar.get_info("John", 1921, 1234.5);
```

使用三个实参调用成员函数get_info。第一个调用针对newcar，而第二个调用针对oldcar。两个对象都属于同一个类Car。注意使用点运算符连接对象名和它的成员函数名（如果我们通过一个指向对象的指针来调用函数，我们使用->代替点运算符）。调用结束后，实际的参数传入到对象。例如当完成第一个调用，信息“Mary”，1998，6543.2被分别赋予了数据成员newcar.owner、newcar.year_made和newcar.price。我们完成了赋值是因为在get_info函数体中使用了变量owner[]、year_made和price。注意在函数get_info中并没有声明变量owner[]、year_made和price。这是因为它们是Car类的数据成员。当newcar.display和oldcar.display被调用后，display函数在屏幕上显示对象的信息。

与公有的成员函数不同，一个私有的成员函数只能被类中的其他的成员函数所调用。例如sellcar()成员函数是私有的。我们不能从main中调用它。但是可以从display成员函数中调用它，因为display函数也属于Car类。因为display在调用的时候已经和一个对象关联，所以在display函数内部不需要通过一个对象调用sellcar()，如oldcar.sellcar()。

3）如何存取私有的数据成员？Car类中的数据成员price是私有的。任何私有的数据成员只能被属于同一个类中的成员函数存取。例如，类Car的成员函数get_info()使用语句

```
price = cost;
```

把 cost 的值赋给私有成员变量 price。另外一个成员函数使用语句

```
cout << "price = " << price << endl;
```

将 price 输出到屏幕。

注意当调用 get_info() 成员函数的时候，一个对象已经和成员关联了。因此不应该在数据成员上使用对象的名字了，如在函数中使用 oldcar.price。

4）如何开发一个成员函数？开发一个成员函数与开发其他函数一样。我们必须声明（写函数原型）和定义（写函数体）一个函数。例如语句

```
prototype --- void      get_info(char *who, int year, double cost);
definition --- void Car::get_info(char *who, int year, double cost)
{                       ...
                        ...function body
}
```

显示了 get_info 函数是如何声明和定义的。注意，在函数的定义中使用了域解析符 ::，这个符号代表后面的函数是属于符号前面的类的。这意味着可以在程序中声明另外一个 get_info() 函数，只要 get_info() 函数不属于类 Car。

5）什么是封装？将数据和管理数据的函数链接到一个单独的对象的过程叫做封装。封装提供了一个高效的方法来控制我们对数据的访问。这是 C 语言没有的特性。在 C 语言中，数据和函数没有被封装，它们是分离的实体。

当函数 display 通过 oldar.display 调用的时候，所有的 oldcar 的数据成员对于 display 来说都是可存取的，即使没有数据通过参数列表来显式传递（拷贝）。你可以看到这是真的，因为函数确实没有参数。注意，oldcar.owne、oldcar.year_made 和 oldcar.price 的值已经被函数访问了，这是由于封装性使得它成为可能（见图 9-7）。

6）封装对于函数访问数据有哪些影响？在 C 语言中，我们看到了向一个函数传递信息的唯一方法是通过参数列表或使用全局变量（我们不推荐全局变量除非它非常必要，因为它降低了程序的结构性）。

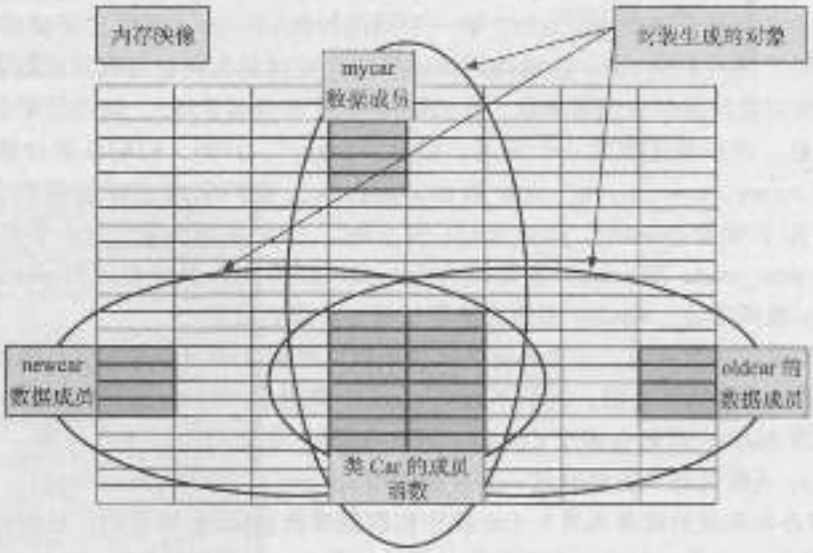


图 9-7 一个内存的映像演示了对象和封装的概念。每个对象的数据成员是分离且唯一的。但是成员函数（内存中保存的指令）是对象间共享的。数据成员和函数被链接到一个对象中

因为C++使用封装，通过一个对象调用成员函数可以自动存取一个对象的数据成员。我们没必要再通过参数列表来输入数据成员了。但是因为数据成员不是全局变量，我们也维护程序设计的模块化。这是C语言所缺少的面向对象编程的概念，如图9-8所示。

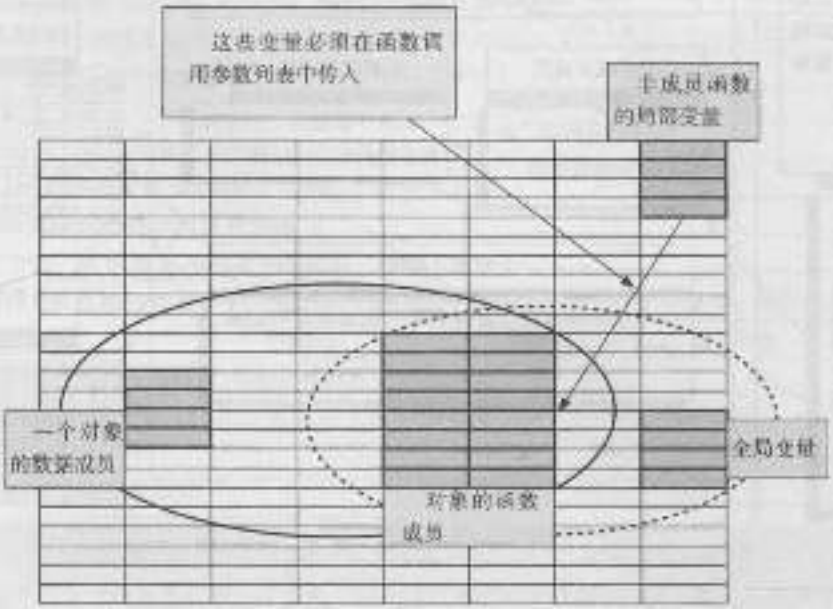


图9-8 一个对象的成员函数对数据成员有直接的访问能力，就像全局变量一样（用虚线表示）。但是非成员函数中的局部变量必须通过参数列表来传递

7）通常，如何从一个非成员函数中同时存取公有和私有数据成员？在一个非成员函数中我们必须使用一个对象名来访问成员，如图9-9所示。可以直接使用对象的名字访问公有数据成员，就像我们在本课程序main（也是一个非成员函数）中用oldcar.colour和oldcar.year_made那样。也能像在main中那样，通过对象的名字调用公有的成员函数oldcar.get_info()和oldcar.display()。

但是我们不能在一个非成员函数里直接访问任何私有成员（既不能访问数据也不能访问函数）。一个替代方法是必须调用一个公有的成员函数，然后让它去调用一个私有的成员函数或者直接访问私有的数据成员。本课的程序中，在main函数中通过oldcar.display调用了公有的函数display，然后它又调用私有的成员函数sellcar（见图9-10）。注意在display中，我们没有使用oldcar.sellcar，因为对象oldcar已经在调用display的时候确定了。另外，从main可以通过oldcar.get_info调用公有的成员函数get_info，它又使用了私有的数据成员price。没有使用oldcar.price，因为对象oldcar已经在调用get_info的时候确定了。

8）我们为什么把Car的定义放到main函数的外边？类Car包含不是内联的成员函数。C++编译器只允许有内联成员函数（有函数）的类是局部的，否则它们必须是全局的。这样，类Car必须是全局的并且放到main的外面。它的影响是通过首先声明类的一个对象，所有的非成员函数能调用公有的成员函数。

9）本课的程序是否演示了一个典型的类定义？不是，一个典型的类的定义是将数据成员声明为私有并且成员函数声明为公有。这使得任何函数可以调用一个成员函数，但是只允许成员函数去访问数据成员。这种安排可以保护数据成员避免被一些没有授权的函数所修

改，同时对于成员函数也给予了访问的权利。在本课中并没有使用这种典型的模式，因为我们要演示不同的类和对象的特性。

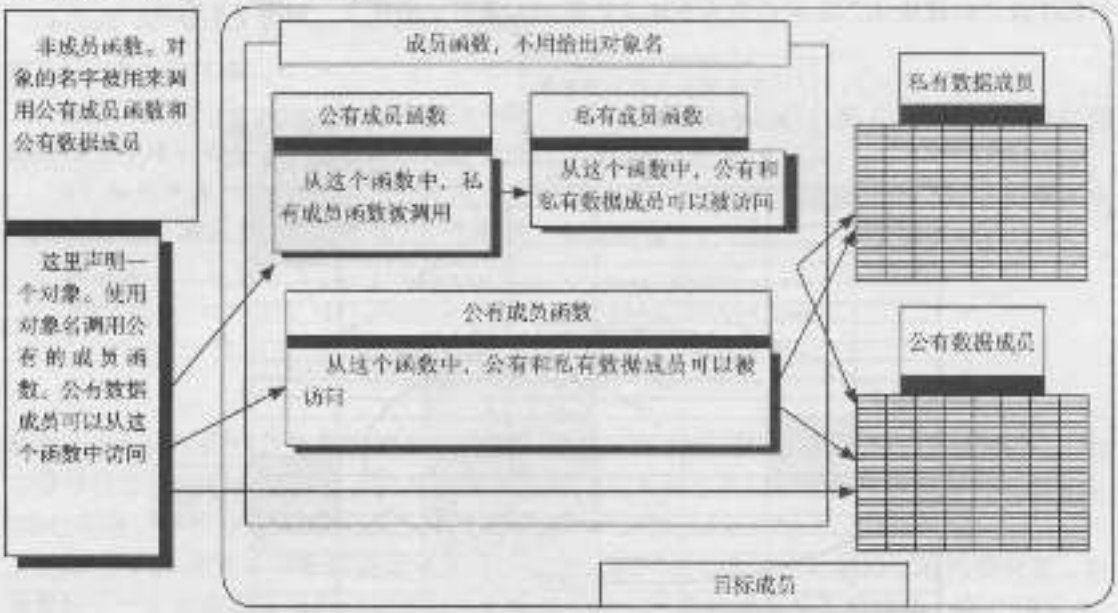


图 9-9 从一个非成员函数中同时存取私有和公有数据成员。注意从一个非成员函数中调用成员函数来访问私有的数据成员。公有的数据成员可以直接被非成员函数访问。在所有情况下，对象的名字必须在非成员函数中使用以便能够访问公有的数据成员以及公有的函数。与图 9-10 比较

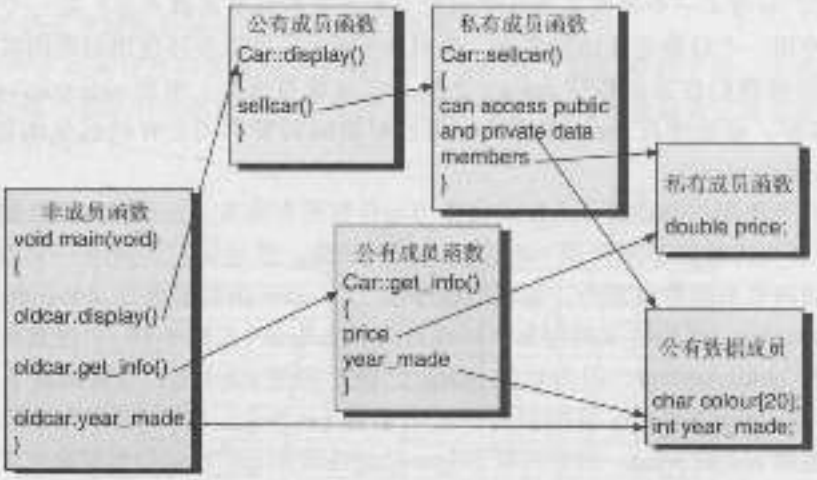


图 9-10 本课的程序在一个非成员函数中存取公有和私有数据成员。与图 9-9 比较

概念回顾

1）成员函数被定义在包含它的类的定义中。

2）为了调用一个公有成员函数，需要提供一个函数以及包含这个函数的对象名。

3）一个私有的成员函数只能被属于同一个类中的其他成员函数访问。

4）把数据和管理数据的函数链接到一个单独的对象的过程叫做封装。通过封装，一个

通过对象名字调用的成员函数可以直接访问这个对象的数据成员。

练习

1. 一个人的血压依赖与这个人的年龄、性别、健康及其他的环境因素。表 9-1 显示了年龄从 20 岁到 49 岁的成年人的标准血压的平均值。显示在表 9-2 中的是 4 个人的相关信息。

   基于这些信息，写一个程序

   a. 开发一个类，名字是 Normal_pressure，将表 9-1 中的数据读入（所有的数据成员是公有）。

   b. 开发一个类，名字是 Indivual_pressure，将表 9-2 中的数据读入（所有的数据成员是公有，除了病人的名字是私有）。

   c. 将个人的血压和表 9-1 给出的他的年龄组里的标准血压做比较，标识出某个人的血压值是高于或低于标准值的百分比。

   d. 将步骤 c 中的信息显示如下。

表 9-1 成年人的平均标准血压

年龄（单位：岁）	血压（单位：mm/Hg）
20 — 24	119
25 — 29	121
30 — 34	123
35 — 39	125
40 — 44	128
45 — 49	130

表 9-2 血压信息

病人名字	年龄	血压
Jerry	42	132
Linda	36	124
Mary	22	118
Ken	46	144

病人名字	年龄	血压	结果
Jerry	42	132	高于正常值
Linda	36	124	低于平均值
Mary	22	118	……
Ken	46	144	……

2. 作为一个程序员，对于一个专门从事娱乐和休闲产业的旅行社的网页，要求开发一个主题索引。部分主题索引如下：

主题	索引	电话
Amusement places	321	418-221-3898, 800-761-2001
Boat renting	456	798-652-1980
Campgrounds and parks	987	238-886-1899
Night clubs	765	457-734-1934, 888-856-3467

为了使用这个程序，用户必须输入一个密码（可以是下列中的任何一个：A123、X987 和 K456），然后是索引或者主题的前 4 个字符。如果输入是正确的，程序会在屏幕上显示主题及相关的电话号码。要求：

a. 声明一个类并且使用一个成员函数读入数据（除了电话号码和密码，所有的数据成员都是公开的）。

b. 调用另外一个成员函数检查密码和输入是否正确。

c. 如果输入是正确的，调用另外一个成员函数将显示输出到屏幕。

课程 9.8 构造函数和析构函数

主题

- 构造函数
- 析构函数

在上一次课，我们学习了 C++ 将数据和函数封装从而提供了一个有效的方法来管理数

据。像我们在程序中多次看到的那样，初始化数据是很必要的。C++提供了特殊的函数，叫做构造函数，它们可以在一个对象被声明的时候自动调用。这些函数可以用来初始化数据成员的值并执行必要的初始化操作。

同时，C++有析构函数，在一个超出它的域（就像C语言的变量超出了它的域，如发生在一个局部变量在函数结束的时候）时自动调用。它们通常的目的就是将动态分配的内存释放掉。本课中，我们没有在析构函数中执行什么特定的操作，只是简单地演示它们的用法。

这个程序输出了三个人的电话号码，通过生成一个有数据成员的类来保存电话号码以及用来初始化和输出号码的成员函数。它将号码输出到屏幕并且指出一个对象已经被析构了。

源代码

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

class Phone
{
        public:
        void      print_number (char *who);
        long      get_phone_no (char *who);
        Phone (char *city);
        ~Phone();

        private:
        long      phone_no;
        int       area_code;
};

Phone::Phone(char *city)
{
        if        (strcmp(city, "Denver")==0) area_code = 303;
        else if   (strcmp(city, "Boston")==0) area_code = 617;
        else                                  area_code = 800;
}
Phone::~Phone()
{
        cout<<"Object destroyed"<<endl;
}

void main(void)
{
        Phone caller1("Denver"), caller2("Boston"), caller3("USA");

        caller1.print_number    ("John");
        caller2.print_number    ("Mary");
        caller3.print_number    ("Tom");
}

void Phone::print_number(char *who)
{
        cout << "who          = " << who                  << endl;
        cout << "Area_code    = " << area_code            << endl;
        cout << "Phone no     = " << get_phone_no(who) << "\n\n";

}

long Phone::get_phone_no(char *who)
{
        if        (strcmp(who,"John")==0)       phone_no=1112233;
        else if   (strcmp(who,"Mary")==0)       phone_no=4445566;
        else                                    phone_no=7778899;

        return(phone_no);
}
```

构造函数必须和类与一样的名字。它不能有返回类型，它可以接受参数，并在一个对象被声明的时候自动地调用

析构函数必须有和类一样的名字，但是有一符号在函数名字的前面。它不能有返回值。当一个对象超出域以后，它被自动调用

构造函数定义

析构函数定义

调用构造函数时使用的参数

声明一个对象

通过对象caller1，caller2和caller3调用公有函数print_number输出电话号码

输出

```
who          = John
Area_code    = 303
Phone_no     = 1112233

who          = Mary
Area_code    = 617
Phone_no     = 4445566

who          = Tom
Area_code    = 800
Phone_no     = 7778899

Object destroyed
Object destroyed
Object destroyed
```

解释

1）什么是构造函数？一个构造函数是一个特殊的函数，用于初始化对象或者为了对象分配内存。每次一个特定类的对象被声明，构造函数都被自动地调用。

构造函数的名字和类的名字永远一致。函数可以包含任何参数或者不包含参数。例如语句

```
Phone (char *city);
```

如果在一个类 Phone 中定义，那么它是一个构造函数，因为函数的名字和类的名字完全一致。函数包含一个形参 city（见图 9-11）。

图 9-11 构造函数和析构函数

构造函数是可选函数。当你定义一个类的时候，可以不定义构造函数。这样定义的类就会使用一个空的默认构造函数。默认构造函数不做任何事，不初始化和检查类中的数据成员。你也可以在某个给定类中定义多于一个构造函数，每一个有不同的参数列表。这些构造函数就像是重载函数一样。对于更多的细节，你可以查看 C++ 编译器的手册。

2）如何定义一个构造函数？一个构造函数的定义多少有点像一个普通函数的定义。通常，定义开始于类名，后接一个域解析符、构造函数的名字、参数列表以及函数体。例如语句

```
Phone::Phone(char *city)
{
   if (strcmp(city, "Denver")==0) area_code = 303;
   else if  (strcmp(city, "Boston")==0) area_code = 617;
   else area_code = 800;
}
```

定义了构造函数 Phone。这个函数包含一个参数，参数被用来选择正确的地区码。你可以在构造函数中加入数据的初始化和检测部分。函数体一般包含对成员数据的赋值操作，分配内存和对输入数据的检查。

注意，构造函数没有类型（也不是 void 类型），且不返回值。

3）如何用一个包含构造函数的类来声明一个对象？如果一个构造函数包含参数，那么我们就用实际的参数去声明一个对象，否则不用参数去声明对象。例如语句

```
Phone caller1("Denver"), caller2("Boston"), caller3("USA");
```

声明了三个对象，caller1，caller2 和 caller3，每一个分别有不同的实参。当声明完毕后，构造函数被自动调用，然后三个对象分别给予了地区码 303、671 及 800。

4）什么是析构函数？一个析构函数是一个可选的成员函数，当一个对象离开域以后自动调用。析构函数的名字以~开始，后接类的名字。析构函数没有参数和返回值，对于简单的类，我们通常不用写析构函数。对于更加复杂的类，析构函数当一个对象被清除时做必要的清理工作。本程序包含析构函数

```
~Phone();
```

当一个对象被清除时它输出一个消息。在这个程序中，三个对象——caller1、caller2 和 caller3 当程序结束时离开域。析构函数通常在复杂的类中做清理工作。

概念回顾

1）构造函数用来为一个对象初始化或申请内存。

2）析构函数是一个可选的成员函数，当一个类离开自己的域后被自动调用。析构函数通常在复杂的类中做清理工作。

练习

1. 写程序使用一个特定的字符值在屏幕上画一个长方形。程序应该包含一个构造函数，使用下面的形参：

```
char border  - any printable character to be used to draw
               the border of a rectangle.
double left  - left coordinate of a rectangle on the
               screen, 0<=left<=80
double right - right coordinate of a rectangle on the
               screen, right>=left, 0<=right<=80
double top   - top coordinate of a rectangle on the
               screen, 0<=top<=25
double bot   - bottom coordinate of a rectangle on the
               screen, bot>=top, 0<=bot<=25
```

例如，如果用户输入

```
* 10 60 5 20
```

程序应该用字符 * 从 $x=10$ 到 $x=60$，$y=5$ 到 $y=20$ 画一个长方形。如果用户输入一个不正确的数据，如 left = 99，那么程序使用默认的 left = 0 来画这个长方形。

2. 写程序来计算从 1 月 1 日到特定日期（年，月，日）的天数。程序包含一个类

a. 有成员函数用来读入输入数据，输入格式是日 / 月 / 年。

b. 有构造函数初始化给定年的任何月里包含的天数。例如，对于闰年，二月有 29 天。

c. 有函数显示输出。

例如，如果用户输入

```
3/5/2012
```

程序显示

```
There are 65 days between 1/1/2012 and 3/5/2012.
```

课程 9.9 继承

主题

- 继承
- 基类和派生类
- 可重用代码

在日常生活中，我们把一些事物根据一般的特征分成组。那些有最通用特点的形成根或基，基被用来派生出有更特性化特征的事物。C++ 允许我们使用基类和派生类来模块化系统。

例如，在 C++ 中，我们可以选择 Motor_vehicle，就是在轮子上移动的车来作为基类。给基类特征 num_headlights=2。从 Motor_vehicle 我们可以派生出一个类 Bus，以及其他的类 Truck。Bus 类包含 num_headlights=2，也许还有 seat_number = 50。Truck 有 num_headlights=2，也许还有 load_capacity= 10 tons。通过使用 C++ 的基类和派生类，我们不需要在 Bus 和 Truck 类中包含 num_headlight=2，因为这个特征已经自动由基类 Motor_vehicle 获得了。

从一个简单的或者通用的类型获得特征的机制叫做继承。继承是区分 C++ 和 C 语言的最重要的一个特征。继承允许我们重新书写或扩展一个存在的类而不用重新书写它的原始代码。在本课的程序中，我们开发了一个名字为 Parent 的基类和一个名为 Child 的派生类。Child 类的对象，son 继承了基类中的成员。程序将 Parent 和 Child 的名字和其他信息输出。

源代码

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

class Parent
{
     public:
     void display(void);
     Parent();

     private:
     char      last_name [20];
     char      first_name[20];
     double    income;
};


class Child :  public Parent
{
     public:
     void info(char *first, int age);
     void print_info(void);
     Child();


     private:
     char first_name[20];
     int  age;
};
```

基类是 Parent

Parent 类和 Child 类都有 first_name 数组，但是只有 Parent 类有 last_name 数组

代表 Child 继承于 Parent

Child 是一个派生类

```
Parent::Parent()
{
     strcpy(last_name,"Smith");
     strcpy(first_name,"John");
     income=1234.56;
}

Child::Child():Parent()
{
}

void main(void)
{
     Child    son;

     cout << "About son     information ---------------------------\n";
     son.info("Ali", 23);
     son.print_info();

     cout << "\n\nChild uses Parent's member function in main-----\n";
     son.display();

}

void Child::print_info(void)
{
     cout << "\nChild uses Parent's member function in print_info --";
     cout << "\nChild's first_name      = "<<first_name<<endl;
     display();
}

void  Parent::display(void)
{
     cout << "Parent firstname = " << first_name <<endl;
     cout << "Parent last_name  = " << last_name  <<endl;
     cout << "Parent income     = " << income     <<endl<<endl;
}

void Child::info(char *first, int years)
{
     strcpy(first_name, first);
     age=years  ;
     cout << "Child firstname = " << first_name        << endl;
     cout << "Child age       = " << age      << endl;
}
```

Parent 类的构造函数。这个函数给 Parent 类 first_name、last_name 和 income

Child 类的构造函数，传送 Parent 的函数被给出

声明 son 是 Child 类的一个对象。声明对 Parent 和 Child 的构造函数被调用

通过 son 调用 Child 的成员函数 info，这个函数给 son 名字和年龄

从 main 中调用 display()。我们用一个对象（son）关联这个函数

在 print_info 中调用 display()。我们不用关联一个对象，因为在调用 printf_info 的时候，已经关联了一个对象（son）

这里，first_name 代表 Parent 类中的 first_name

这里，first_name 代表 Child 类中的 first_name

输出

```
About son    information ------------------------------
Child firstname = Ali
Child age       = 23

Child uses Parent's member function in print_info ------
Child's firstname = Ali
Parent firstname  = John
Parent last_name  = Smith
Parent income     = 1234.56

Child uses Parent's member function in main ------------
Parent firstname  = John
Parent last_name  = Smith
Parent income     = 1234.56
```

解释

1）什么是基类和派生类？新类从它派生的类叫做基类或父类。一个从基类派生的类叫做子类或派生类。一个子类也可以作为基类来派生出下一代的类。因此我们可以生成一个类的层次结构，其中任何类都可以作为新类的父类或者是基类。

2）如何定义 C++ 中派生类？一个 C++ 派生类从基类中产生。因此当我们定义一个派生类之前，我们必须定义一个基类。任何正规的 C++ 类，不管有没有构造函数，都可以作为一个基类。但是如果想从基类中继承一些特性，我们需要包括一个基类的构造函数来初始化这些特性。例如，在程序中我们使用 Parent 基类和它的构造函数

```
Parent::Parent()
{
   strcpy(last_name,"Smith");
   strcpy(first_name,"John");
   income=1234.56;
}
```

来初始化数据成员 Parent 类的成员 last_name、first_name 和 income。任何从这个类派生出来的类将会自动继承这些数据。继承通过现有的类和从它们派生新的类达到。一个继承的类可以选择性地继承一些成员，拒绝一些成员，修改其他的基类成员或者加上自己的新的成员。如果我们不在基类中包括一个构造函数的话，任何派生类定义的对象就包含基类的数据成员和成员函数，只不过这些成员没有初始值。

一旦你定义了一个基类，可以定义一个派生类。一个基类可以派生出任何数量的子类。每一个子类可以有与其他子类不同的数据成员或成员函数。在本课中只定义了一个派生类，名字叫做 Child。派生类的语法如下：

```
class derived_class_name : access_modifier  base_class_name
{
  derived class data and function members
  ...
}
```

access_modifier 用来指定派生类的存取性，它必须是关键词 private、protected 或者 public 中的一个。通过选择正确的 access_modifier，我们控制了从派生类中可以访问的基类中的数据成员和成员函数。通常对于一个给定的派生类，可以把对基类数据的访问控制得更为严格，但不能更放松控制。这意味着低层的类可能不被允许访问上层的类。像在 C++ 一样，在现实生活中也是如此。没有了这个控制，任何人从派生类中都可以修改或者破坏基类的数据。下面的语句

```
class Child :  public Parent
 {
        public:
        void info(char *first, int age);
        void print_info(void);
        Child();

        private:
        char first_name[20];
        int  age;
 };
```

定义了 Child 从 Parent 类派生，access_modifier 是 public。public 允许 Child 的对象可以自由存取一个 Parent 类的成员，就像一个 Parent 类的对象一样。如果 access_modifier 是 private 或者 protected 而不是 public，Child 的对象可能不能自由存取 Parent 类的某些成员。对于更多细节，请参考 C++ 编译器手册。这个定义告诉我们派生类 Child 包含两个数据成员，（first_name 和 age），成员函数 info( ) 和 print_info( )，以及一个构造函数 Child( )。对于一个通用的派生子类，见图 9-12。

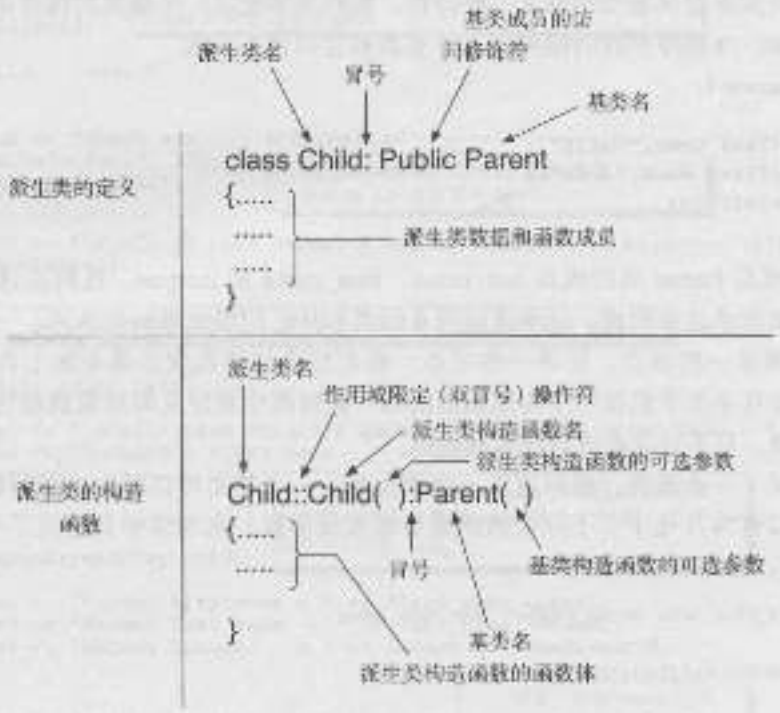


图9-12 派生类的通用格式

3）如何定义一个派生类的构造函数？定义一个派生类的构造函数语法如下：

```
dc_name::dc_name(dc_list):Bc_name(Bc_list)
{
 derived class constructor function body
...
}
```

其中 dc_name 是派生的类名，dc_list 是派生的参数列表，Bc_name 是基类名，Bc_list 是基类的参数列表。

例如，语句

```
Child::Child():Parent()
{
}
```

定义了一个派生类 Child 的构造函数。在本课的程序中，基类的构造函数和派生类的构造函数都是空参数列表。本文并不包括一个参数列表非空的基类构造函数。

4）如何声明一个子类的对象？任何子类的对象都可以直接声明。我们不要求在声明子

类对象的时候先声明一个基类对象。例如语句

```
Child    son;
```

声明 son 为一个 Child 对象，我们并没有声明任何 Parent 的对象。当我们声明了 Child 类的对象后，基类 Parent 的构造函数和子类 Child 的构造函数都被自动运行了。

5）当声明一个子类的对象时，它占用多少内存？父类和子类中的数据成员都要求新的内存分配。

6）对于基类和派生类，当它们的数据成员是私有并且成员函数是公有的时候，如何从一个非成员函数存取派生类的数据成员？首先，我们在函数中声明一个派生类的对象。然后调用一个 public 的派生类或者基类的成员函数来存取私有的数据成员，如图 9-13 所示。学习这个图并理解为什么我们不能从一个非成员函数中直接去访问私有数据。

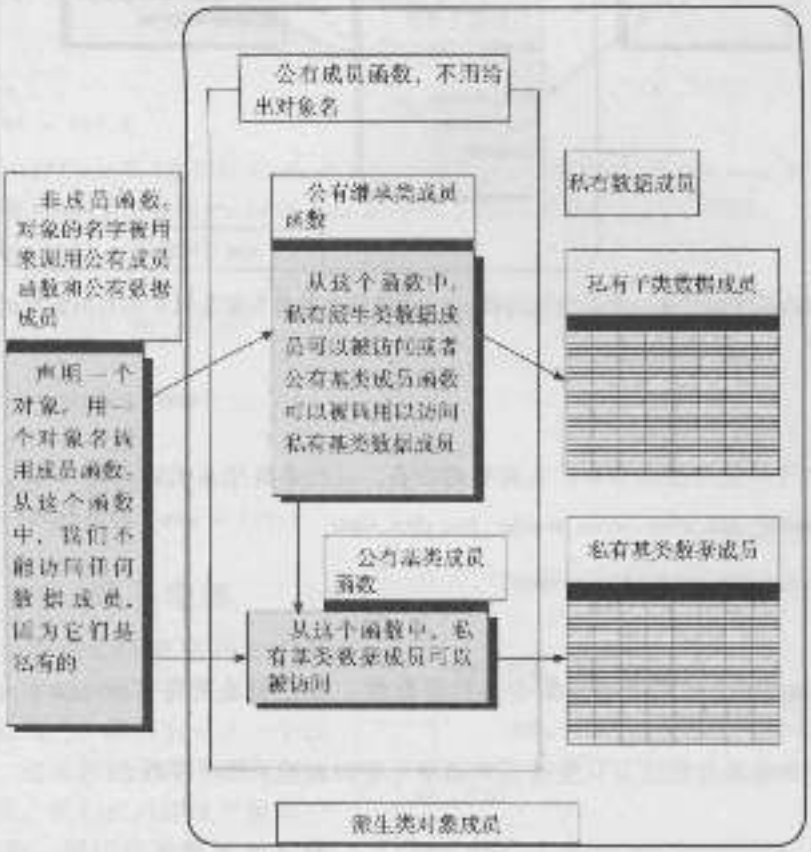


图 9-13 在这个示意图中我们假设所有的数据成员都是私有的，所有的成员函数都是公有的。为了从一个非成员函数中存取一个派生类中的私有的数据成员，必须调用一个公有的派生类函数。为了在一个非成员函数中存取一个基类的私有数据成员，可以直接调用一个公有的基类成员函数或者一个公有的派生类成员函数，然后去调用公有的基类成员函数。与图 9-14 比较

7）我们如何存取派生类和基类的私有数据成员？如图 9-14 演示，图中显示了在一个非成员函数中应该使用一个对象名，并且可以通过基类的公有成员函数存取私有数据成员。与图 9-14 比较。注意图 9-13 和图 9-14 的类比性。

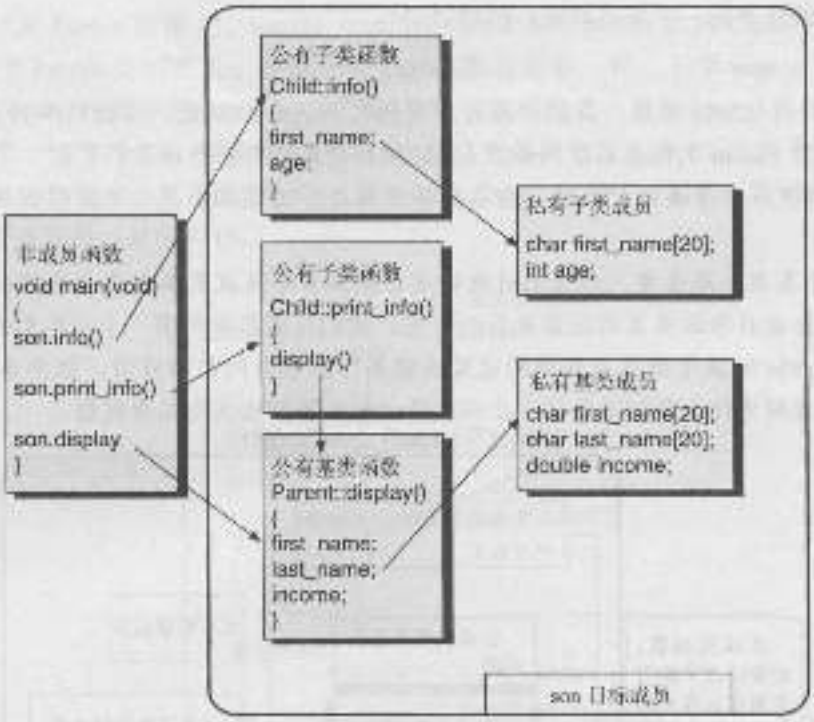


图 9-14 本课的程序如何从一个非成员函数 main 中访问在基类和派生类中私有的数据成员。与图 9-13 比较

概念回顾

1）我们可以使用继承类来扩展基类的功能，以此来重用基类的代码

```
class derived_class_name : access_modifier  base_class_name
{
   derived class data and function members
   ...
}
```

access_modifier 被用来指定派生类的存取性。它必须是关键字 private、protected 或者 public 中的一个。

2）基类的私有数据成员只能被基类和派生类的成员函数所存取。

练习

1. 写一个基类 Car 和它的两个派生子类 Bus 和 Truck。基类必须包含下面的数据成员：

```
Car  : wheel, colour
Bus  : seat_num
Truck: load_capacity
```

输出如下：

```
Base class CAR information--------------
Wheel = 4
Colour = White

Derived class Bus information------------
```

```
Wheel = 4
colour = Yellow
Seat num = 50

Derived class truck information----------
Wheel = 4
colour = Blue
Load capacity = 8 tons
```

2. 给定下面的 Circle 类定义，从中派生出新类 Cone。

```
class Circle
{
     double x0, y0, radius;
}
```

类包含一个成员函数可以计算并显示 Cone 的容积。使用以下参数检测你的程序。

```
x0=100.0
y0=200.0
radius=300.0
cone height = 400.0
```

3. 基于练习 2 中的 Cone 类（我们从 Circle 类派生），派生出一个新类 Truncated_cone 类。这个类包含一个成员函数可以计算并显示 truncated cone 的容积。用以下的参数实验你的程序。

```
Top of truncated cone
x0=100.0
y0=200.0
radius=300.0

Bottom of truncated cone
x0=100.0
y0=200.0
radius=500.0
height of truncated cone = 400.0
```

应用程序 9.1　电子电路

设计一个类在 C++ 程序开发中是非常重要的。本文中没有空间去讨论这个设计。取而代之，我们演示了一个设计类的例子。这里的例子并不是完整的类开发的代表。我们的兴趣在于给你一些 C++ 的了解，所以注重简单而不是通用和效率。

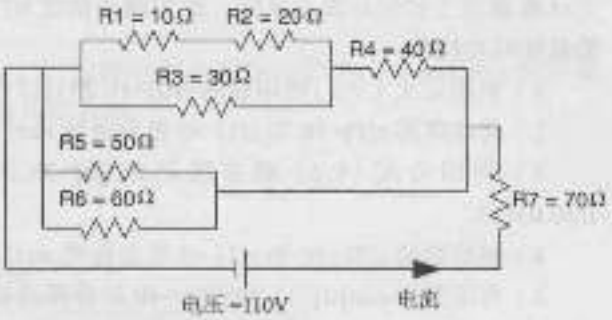


图 9-15　一个电路的例子

问题描述

如图 9-15 所示的电子电路包含 7 个电阻和一个 110 伏的直流电源。写程序找出经过电源的电流强度。

对于这个电路，输入如下：

```
a 10 20
a 30 0
p
```

```
s 40 0
s
p 50 60
p
s 70 0
s
```

其中 *s* 代表两个电阻串联，*p* 代表它们是并联的。

两个电阻可以在一行输入，第一行代表电路中的 R1 和 R2 是串联的。第二行用一个零值 0 电阻与 R3 串联。第三行代表前两行的电阻（行 1 和行 2）是并联的。第四行代表 R4 和一个零值 0 电阻串联。第五行代表 R4 和所有以前计算过的电阻串联。第六行代表 R5 和 R6 并联。第七行代表 R5 和 R6 形成的子电路与前面计算过的电阻并联。第八行代表 R7 和一个零电阻串联。最后一行代表 R7 与前面计算过的电阻串联。我们使用这个输入来计算电路的总电阻及电源中的电流。

从这里可以看出需要一些技巧来准备特定电路的输入数据。对于一些高级的工程问题确实应该这样。你会发现实际上，有必要成为一个有才华的程序员以处理一些工程问题。

解决方法

1. 相关公式和背景知识

为了找出电流的强度，我们用欧姆定律：

$$I=V/R \tag{9.1}$$

其中 V 是电压，R 是电路中的电阻。本例中电压给定（110 伏特），但是电阻必须计算。

当两个电阻并联，电阻的和 R12 等于

$$R12 = 1/(1/R1+1/R2) \tag{9.2}$$

当两个电阻串联，电阻的和

$$R12 = R1+R2 \tag{9.3}$$

2. 特定的例子

在这个例子中，电路包含 7 个电阻以串并联的方式连接。我们通过几步来计算总的电阻。对于每一步，把两个电阻组合成一个。这么做几次后，可以计算出总的电阻。下面的手工计算演示了如何计算总电阻。使用两个数组 r[] 和 rtot[] 来辅助计算。参考图 9-15 查看电阻是如何连接的：

1）利用公式（9.3）将串联的 r[0]=10 和 r[1] =20 组合得到 rtot[0] = 10+20=30。

2）将串联的 r[0]=30 和 r[1] =0 组合得到 rtot[1] = 30+0=30。

3）利用公式（9.2）将并联的 rtot[0]=30 和 rtot[1] =30 组合得到 rtot[0] = 1/(1/30.0+1/30.0)=15。

4）将串联的 r[0]=40 和 r[1] =0 组合得到 rtot[1] = 40+0=40。

5）将串联的 rtot[0]=15 和 r[1] =40 组合得到 rtot[0] = 15+40=55。

6）将并联的 rtot[0]=50 和 r[1] =60 组合得到 rtot[1] = 1/(1/50.0+1/60.0)=27.27。

7）将并联的 rtot[0]=55 和 r[1] =27.27 组合得到 rtot[0] = 1/(1/55.0+1/27.27)=18.23。

8）将串联的 r[0]=70 和 r[1] =0 组合得到 rtot[1] = 70+0=70。

9）将串联的 rtot[0]=18.23 和 rtot[1] =70 组合得到 rtot[0] = 18.23+70=88.23。这是电路中的总的电阻值。

电流就可以通过公式 9.1 得到

$$I=110/88.23=1.30 \text{ amp}$$

注意这个解法中，我们交替使用 rtot[0] 和 rtot[1] 来保存总的电阻。

3. 数据结构和类

为了将第一个问题尽量简化，我们只使用一个类和一个对象。类叫做 Circuit 并且它的公有成员是函数，私有成员是数据。定义如下：

```
class Circuit
{
        public:
        double series(double r[ ]);
        double parallel(double r[ ]);
        void find_resistance(void);
        void find_current(void);
        Circuit (double dummy);

        private:
        double r[2], rtot[2];
        double voltage, current;
        char flag;
};
```

成员的含义可以通过源代码中的符号得到。一个对象 res_circuit 声明如下：

```
Circuit     res_circuit1(110.);
```

声明中的参数在构造函数中将电压初始化为 110 伏。

4. 算法

算法可以从特定的例子中推导如下：

1）对于第一个子电路读入输入数据并计算总电阻。

2）对于第二个子电路读入输入数据并计算总电阻。

3）读入两个子电路的关系（串并联）并计算总电阻。

4）对于下一个子电路读入输入数据并计算总电阻。

5）读入前两个子电路的关系（串并联）并计算总电阻。

6）重复 4、5 步骤直到得到电路的电阻值。

7）根据公式计算电流。

因为在本章并不关注问题的具体步骤，我们留给你去验证函数 find_resistance 中的循环，这些循环执行了在例子中给出的具体的计算步骤。

源代码

```
#include <iostream.h>

class Circuit
{
        public:
        double series(double r[ ]);
        double parallel(double r[ ]);
        void find_resistance(void);
        void find_current(void) ;
        Circuit (double dummy);
```

所有的成员函数都是公有的

计算串联子电路的函数

计算并联子电路的函数

计算整个电路的电阻的函数

计算电路中电流的函数

所有的数据成员都是私有的

```
private:
double r[2], rtot[2];
double voltage, current;
char flag;
};
```

子电路和整个电路的电阻

代表并联的标志"p"和代表串联的标志"s"

```
Circuit::Circuit(double dummy)
{
        r[0]=0.0;
        r[1]=0.0;
        rtot[0]=0.0;
        rtot[1]=0.0;
        current=0.0;
        flag='s';
        voltage=dummy;
}
```

构造函数初始化数据

构造函数声明和执行的时候把电压初始化为110伏

```
void main(void)
{
        Circuit res_circuit1(110.);

        res_circuit1.find_resistance();
        res_circuit1.find_current();
}
```

发现res_circuit1的总电阻

发现res_circuit1中电源的电流

输入并分析第一个子电路的电阻

```
void Circuit::find_resistance(void)
{

        cout<<"Enter flag, r0, r1"<<endl;
        cin>>flag>>r[0]>>r[1];

        if(flag=='s') rtot[0]=series(r);
        if(flag=='p') rtot[0]=parallel(r);
        cout<<"Subtotal resistance="<<rtot[0]<<endl;
```

输入并分析最后两个子电路的电阻

```
        for (int i=1; i<=4; i++)
        {
                cout<<"Enter flag, r0, r1"<<endl;
                cin>>flag>>r[0]>>r[1];
                if(flag=='s') rtot[1]=series(r);
                if(flag=='p') rtot[1]=parallel(r);
                cout<<"Subtotal resistance="<<rtot[1]<<endl;
                cout<<"Enter flag"<<endl;
                cin>>flag;
                if(flag=='s') rtot[0]=series(rtot);
                if(flag=='p') rtot[0]=parallel(rtot);
                cout<<"Subtotal resistance="<<rtot[0]<<endl;
        }

        cout<<"Total resistance="<<rtot[0]<<endl;
```

输出电路的总电阻

```
}
void Circuit::find_current(void)
{
        current = voltage/rtot[0];
        cout<<"Current at the power supply = "<<current<<endl;
}
```

利用公式(9.1)计算电流

```
double Circuit::series(double r[ ])
{
        double rtot;
        rtot=r[0]+r[1];
        return rtot;
}
```

利用公式(9.3)计算电阻

```
double Circuit::parallel(double r[ ])
{
        double rtot;
        rtot=1./(1./r[0]+1./r[1]);
        return rtot;
}
```

利用公式(9.2)计算电阻

输出

```
键盘输入    Enter flag, r0, r1
            s 10 20
            Subtotal resistance=30
键盘输入    Enter flag, r0, r1
            s 30 0
            Subtotal resistance=30
键盘输入    Enter flag
            p
            Subtotal resistance=15
键盘输入    Enter flag, r0, r1
            s 40 0
            Subtotal resistance=40
键盘输入    Enter flag
            s
            Subtotal resistance=55
键盘输入    Enter flag, r0, r1
            p 50 60
            Subtotal resistance=27.27
键盘输入    Enter flag
            p
            Subtotal resistance=18.23
键盘输入    Enter flag, r0, r1
            s 70 0
            Subtotal resistance=70
键盘输入    Enter flag
            s
            Subtotal resistance=88.23
            Total resistance=88.23
            Current at the power supply=1.30
```

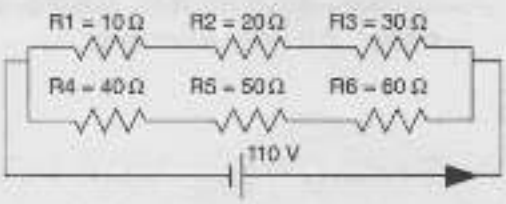
注释

为了使程序尽量简单，我们省略了数据检查，并且避免了当数据输入错误的时候产生的被 0 除的错误。但是在商用程序中应该考虑这些。

修改练习

1. 修改程序使得在函数 parallel 中不会出现被 0 除的情况。
2. 修改程序处理两个相似的电路。第一个电源是 300 伏特，第二个电源是 450 伏特。
3. 修改程序使得它每次可以处理包含三个电阻的子电路，而不是目前的两个。

应用练习



9.1 写一个程序，对于任何值的电阻，能计算下面电路中的电流。假设电路中可以包含很多这种类型的子电路。

9.2 一个城市某天的污染程度是一个时间的函数（小

时)。作为一个环境专家，你已经在不同的时间收集了污染程度的数据如下：

时间	污染程度
0:00	58
2:00	51
4:00	47
6:00	51
8:00	55
11:00	67
14:00	78
16:00	86
18:00	82
20:00	86
23:00	65

写一个程序来画出污染的曲线。程序应该包含两个类。第一个类有一个成员函数能从文件读入表格所示的数据，另外一个成员函数可以发现数据的范围以便于画图。第二个类应该处理画图问题。

9.3 作为一个软件工程师，你被要求写一个应用程序的用户友好的界面。这一部分试图基于用户输入的两个字符串来相乘两个数，数字或者是实型或者是复型。用下面的输入字符串检验你的程序：

```
3 × 4
5 × (6 - 7i)
(-8 + 9i) × 10
(1 + 2i) × (-3 - 4i)
```

程序应该产生以下的输出：

```
3 × 4 = 12
5 × (6 - 7i) = 30 - 35i
(-8 + 9i) × 10 = -80 + 90i
(1 + 2i) × (-3 - 4i) = 5 - 10i
```

程序应该包含两个类。属于第一个类的对象应该可以分解输入的字符串得到正确的操作符和操作数。第二个类的对象应该执行计算和输出到屏幕的任务。

9.4 给定三角形的三个边 a、b 和 c，它的面积可以用下面的公式计算。

海伦公式

面积 $= \sqrt{s'(s'-a)(s'-b)(s'-c)}$

s' = 半周长

⇒ 公式应为

$$面积 = \frac{1}{4}\sqrt{s(s-2a)(s-2b)(s-2c)}$$

其中 $s' = \dfrac{a+b+c}{2}$ 且 $s = a + b + c$

其中 s 是三角形的周长，写程序包含一个类 Tri_area。类包含数据成员接受 a、b 和 c 的值作为参数，并且计算三角形的面积。

ASCII 码

字符	ASCII 值	字符	ASCII 值	字符	ASCII 值	字符	ASCII 值
NUL	0	space	32	@	64	96	p
SOH	1	!	33	A	65	a	97
STX	2	"	34	B	66	b	98
ETX	3	#	35	C	67	c	99
EOT	4	$	36	D	68	d	100
ENQ	5	%	37	E	69	e	101
ACK	6	&	38	F	70	f	102
BEL	7	,	39	G	71	g	103
BS	8	(	40	H	72	h	104
HT	9	)	41	I	73	i	105
LF	10	*	42	J	74	j	106
VT	11	+	43	K	75	k	107
FF	12	,	44	L	76	l	108
CR	13	-	45	M	77	m	109
SO	14	.	46	N	78	n	110
SI	15	/	47	O	79	o	111
DLE	16	0	48	P80	'	112	
DC1	17	1	49	Q	81	q	113
DC2	18	2	50	R	82	r	114
DC3	19	3	51	S	83	s	115
DC4	20	4	52	T	84	t	116
NAK	21	5	53	U	85	u	117
SYN	22	6	54	V	86	v	118
ETB	23	7	55	W	87	w	119
CAN	24	8	56	X	88	x	120
EM	25	9	57	Y	89	y	121
SUB	26	:	58	Z	90	z	122
ESC	27	;	59	[	91	{	123
FS	28	,	60	\	92	|	124
GS	29	=	61	]	93	}	125
RS	30	.	62	^	94	~	126
US	31	?	63		95	DEL	127

ASCII 码描述

NUL	空
SOH	头标开始
STX	正文开始
ETX	正文结束
EOT	传输结束
ENQ	查询
ACK	确认
BEL	铃响
BS	退格
HT	水平制表符
LF	换行
VT	垂直制表符
FF	换页
CR	回车
SO	移出
SI	移入

DLE	数据链路转义
DC1-DC4	设备控制
NAK	反确认
SYN	同步空闲
ETB	传输块结束
CAN	取消
EM	介质中断
SUB	替换
ESC	换码
FS	文件分割符
GS	组分割符
RS	记录分割符
US	单元分割符
DEL	删除

以上的 ASCII 码只使用了 7 位，但是现在的计算机系统用 8 位单元保存数据，因此所有的系统把最高位（第 7 位，最有意义位）设置为 0。

另外，一些代码符号被用于过去终端模式下控制数据传输，举例来说，那些多于两个字符的符号，它们现在不再使用了。这些符号没有在现代代码集合，也就是 Unicode 中重用。Unicode 有大于或等于 65536 个可用字符，所以 Unicode 是 ASCII 的超集。

C语言程序设计 问题解答和实例解析方法

C Programming a Q & A Approach

本书是一本优秀的C语言程序设计教材。作者通过问题-解答方式来介绍C语言，内容包括编程基础、变量、算术表示和输入/输出、C语言基础——数学函数和字符文件输入/输出、选择结构和循环、函数、数组、字符串和指针、结构和大型程序设计。书中既详细介绍了C语言程序设计的基础知识，又结合实际应用，给出了应用程序。应用程序包含问题描述、算法、源代码、注释和修改练习等。

本书特色

- 每一课都从一个示例程序开始，通过了解代码的细节，加深读者对C语言的理解。
- 以问题-解答方式清晰阐释示例程序，便于读者轻松掌握C语言的要点。
- 使用大量图片形象而生动地解释编程概念，有助于读者快速理解和掌握C语言编程。
- 通过应用程序来说明C语言在求解工程问题和计算机科学问题方面的用处。
- 大部分章后都提供应用练习，便于读者进一步实践和巩固所学知识。

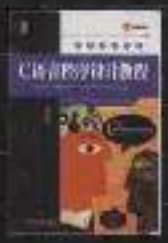
作者简介

H. H. 塔思（H. H. Tan）目前供职于Morrison Knudsen公司。

T. B. 多拉齐奥（T. B. D'Orazio）目前是旧金山州立大学土木工程专业负责人。

柯兆恒（S. H. Or）香港中文大学计算机科学与工程系电脑游戏技术中心创始人和项目主管。他于1988年获得香港大学博士学位，讲授程序设计课程20余年。他的研究兴趣是计算机图形学、计算机视觉、多媒体和游戏开发。他是IEEE和ACM会员。

玛丽安 M. Y. 周（Marian M. Y. Choy）香港大学工程学院教学顾问。她热爱教学并且积极参与教学活动，拥有新南威尔士大学博士学位。研究兴趣是计算机教育、自适应技术和图像处理。



本书影印版
书号：978-7-111-10432-3
定价：49.00元

Mc Graw Hill Education
www.mhedu.com.cn

图书热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书咨询：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字服务：www.hzmedia.com.cn

ISBN 978-7-111-24334-3

9 787111 243343 >

定价：79.00元