普通高等教育“十一五”国家级规划教材

新编计算机类本科规划教材

C 程序设计语言

魏东平　朱连章　于广斌　编著

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书是普通高等教育“十一五”国家级规划教材，从实用性、适应性和先进性出发，以培养读者C语言程序设计的能力为目标，结合大量实例，较全面地介绍C语言的基本概念和程序设计的基本方法。全书共分13章，主要内容包括：C语言基础，顺序、选择和循环程序设计，数组，指针，字符串，函数，自定义数据类型，文件操作，位操作等。本书配套《C 程序设计语言实验与习题指导》，并提供配套电子课件、习题解答和程序源代码。

本书可作为高等学校计算机与信息技术课程的基础教材，也可供相关领域的工程技术人员学习、参考。

图书在版编目（CIP）数据

C程序设计语言／魏东平，朱连章，于广斌编著.—北京：电子工业出版社，2009.2
（新编计算机类本科规划教材）
ISBN 978-7-121-08141-5

Ⅰ. C… Ⅱ. ①魏…②朱…③于… Ⅲ. C程序—程序设计—高等学校—教材 Ⅳ. TP312

中国版本图书馆CIP数据核字（2009）第008421号

责任编辑：王羽佳
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编：100036
开　　本：787×1092　1/16　印张：17.5　字数：448千字
印　　次：2009年2月第1次印刷
印　　数：5000册　　定价：29.90元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

21 世纪，人类社会步入了高速发展的信息时代，掌握信息技术已经成为每一个人最基本的需求。信息技术的核心是计算机技术，计算机应用技能的培养离不开计算机教育。随着科教兴国战略的实施和社会信息化进程的加快，我国高等教育事业的发展驶入了快车道，计算机教育改革也日益受到更加广泛的重视，许多高等学校都把计算机教育“四年不断线”列为教育改革的方向。而计算机技术的核心是程序设计，计算机教育就是围绕程序设计展开的。程序设计的过程贯穿了阅读、判断、分析、思考、抽象、综合、工具、环境等多项技能，对计算机技能的培养至关重要。

理论研究与教学实践都已表明，大学的第一门程序设计课程必须从程序设计领域最基本、最重要的问题出发，也就是要求学生掌握最基本的概念、最基本的思考问题方式和可能使用的技术。

什么是程序设计的基本概念呢？一般来说，这些概念应该包括数据及其表示、变量的类型和值、基本命令（语句）、流程控制结构、子程序（函数与过程）抽象、循环、接口（界面）与实现的分离与相互关联、复杂数据的组织、程序的复杂性及其控制（程序组织）等。程序设计课程应该围绕这些基本概念展开，帮助学生掌握这些概念，并基于这些概念，在使用某种程序设计语言解决实际问题的过程中学习程序设计。

当然，程序设计课程的重点不是单纯地讲授程序设计语言的理论知识，而是以某种程序设计语言为工具，讲授程序设计的基本思想、方法和技术，让学生掌握用编程工具解决实际问题的能力。

对于大多数学生而言，学习程序设计语言就如同学习外语，掌握基本的语言要素（如语法、词法等）就已经比较困难了，还要灵活地使用语言，也就是听、说、读、写，当然更加困难。面对这一问题，许多学者结合教育理论和教学经验提出了多种不同的应对方法，最有影响的是案例教学法，即把程序设计的基本概念由浅入深地融入若干程序“案例”中，通过分析、设计、总结，让学生在不断尝试“编程”的同时，学习程序设计的基本知识，理解程序设计的基本思想，掌握程序设计的基本方法。案例教学突出了实践的重要性，强调让学生在编写程序的实践中逐渐增加成就感、培养学习兴趣，通过形象思维逐步加深理解、巩固知识。

目前，选择 C 语言作为第一门程序设计语言是最普通的做法，这得益于 C 语言所具有的自由的书写格式、良好的表达能力、丰富的数据结构、结构化的程序特征等优势。C 语言具有与汇编语言一样的效率，便于与硬件技术的融合；具有与 C++、Java 等相似的风格，便于用户进一步学习。这些都促成了 C 语言在计算机教学中的特殊地位。

但由于 C 语言涉及的概念较多，语法规则比较繁杂，特别是 C 语言的数据类型、输入和输出等都普遍具有低级语言的特征，与计算机系统的关系密切，对于缺乏计算机基础知识的初学者来说，容易引起混乱。这也是造成 C 语言“难学”的主要原因之一。国内很多学者都对 C 语言的教学进行了研究，并在此基础上编写了许多很有价值的教材和辅导材料，取得了可喜的成绩。

本教材是作者在总结了十几年的教学经验后编写的，融合了许多对于 C 语言教学的认识

和思考。在内容选择和结构组织上，试图体现以培养程序设计能力为核心，以 C 语言基础知识、算法基本概念和程序基本结构为重点的教学理念。教材中引入了大量应用实例，侧重实例分析，避免了过多地罗列 C 语言的语法规则。在实例选取上，既考虑了实例的典型性，又考虑了实用性和趣味性，实例分析的重点也放在了程序设计的思想方法上，力求做到引人入胜，不断增加读者的编程兴趣。当然，作为教材，本书也特别注意了内容选取的科学性和组织的有效性。

本书的前 5 章是基础部分，介绍 C 语言的基础知识和程序的基本结构。第 6～12 章是重点部分，侧重程序设计实践。其中，第 6～8 章的核心是数据的表示和处理，首先介绍了数组和指针的概念和用法，然后在第 8 章集中介绍字符串的处理方法，体现了 C 语言的优势和特点。第 9～11 章以函数为核心，通过程序组织、模块化设计、数据组织等的学习，培养学生应用程序设计解决实际问题的能力。第 12 章介绍文件的操作。出于对部分理工科专业的特殊需要的考虑，第 13 章介绍了 C 语言的位运算。

本书配套《C 程序设计语言实验与习题指导》，并提供配套多媒体电子课件、习题解答和程序源代码，以利于教师备课和学生自学。请登录华信教育资源网（http://www.huaxin.edu.cn 或 http://www.hxedu.com.cn）注册下载。

全书共分为 13 章，第 1～4 章由于广斌老师编写，第 5～8 章由魏东平老师编写，第 9～13 章由朱连章老师编写，全书由魏东平老师统稿。中国石油大学的李宗民教授在百忙之中对全书进行了审阅。在本书编写期间，中国石油大学的葛元康、李克文、孙东海等老师提出了很多宝贵的意见和建议，电子工业出版社的王羽佳编辑在成书过程中也做了大量工作，在此一并表示感谢。

本书中的不足之处，竭诚希望得到广大读者和同行的批评指正。

作　者

目　录

第 1 章　C语言概述

学习目标

通过本章学习，应该掌握：

- C 语言程序的基本特征
- C 语言程序的运行环境

1.1　程序设计与程序设计语言

1.1.1　计算机与程序设计

半个世纪以来，计算机技术无论作为科学学科，还是作为现代产业，都已从一颗幼苗成长为枝繁叶茂的参天大树。回顾其发展历程，计算机也许是人类 20 世纪带给 21 世纪的最有价值的礼物，是人类文明历史上最伟大的发明之一，现在估计它对人类生活将会产生多么大的影响也许还为时尚早。目前，计算机可以在怎样的程度上延长或代替大脑的活动，计算机可以在何种程度上被广泛而深入地应用于各个领域，谁也不能指出一个“到顶”不再发展的时间。不过现在可以指出的是，使计算机具有如此影响力的根本原因是，计算机不是一个一次性的直接服务产品，它为人类服务是有条件的，这个条件就是程序和程序设计。

那么，对计算机而言，程序是什么呢？人们要让计算机解决一个问题时，需要把解决这个问题的步骤通过一条条指令的形式告诉计算机。一般，把人们事先准备好的、用来指挥计算机工作的描述工作步骤的指令序列称为程序，把程序员设计编写程序的过程称为程序设计。用来编写程序的语言称为程序设计语言。没有程序和程序设计，计算机就是一堆废物，也就是说，程序（软件）是计算机的必要组成部分。

计算机首先要求人们不断地在程序设计上付出大量的创造性劳动，然后才能享受到它的服务。计算机好像是唯命是从的仆人，严格地按照程序规定的步骤完成任务。为计算机编写程序是一项非常复杂和具有挑战性的工作。也可以说，自计算机问世的半个世纪以来，人们都是在研究设计各种各样的程序，使计算机完成各种各样的任务。程序设计是一项永无止境、极其困难复杂而又富有魅力和创造乐趣的工作，每年都吸引着数以十万计的优秀人才投入其中，促使计算机产业和计算学科取得了日新月异的发展。

1.1.2　程序设计语言的发展

程序设计的任务就是用程序设计语言编写程序，然后交给计算机去执行。在计算机应用的最初的十几年中，大多数计算机程序都是用机器语言编写的。这种“语言”虽然十分简单，机器可以“看”懂，但对于程序员来说却很不方便。于是，相继出现了汇编语言和高级语言。

1. 机器语言

在 20 世纪 50 年代，计算机专家们直接使用计算机能识别的指令系统（称为机器语言）来编写程序。由于机器语言是由二进制代码组成的，人们编写或阅读程序都十分困难，又容易出错，而且不同机器的指令系统也不同，很难进行交流（在一种计算机上调试通过的程序不能到另外一种计算机上运行）。机器语言程序虽然执行效率很高，但花费在程序设计和调试程序上的时间太多，这样就降低了整个解决问题的效率。

2. 汇编语言

人们为了解决二进制代码编程带来的困难，采用了助忆码和符号地址来代替机器语言中的二进制指令代码和指令地址，然后通过一个叫做汇编程序的翻译程序翻译成机器语言程序，再让计算机执行。这种采用助忆码和符号地址的语言称为汇编语言，汇编程序的翻译方式称为汇编。用汇编语言编写的程序的执行效率与机器语言程序一样高，且其可读性提高了，因此，到了 20 世纪 60 年代，用机器语言编程已经比较少了。但是，汇编语言也是面向机器的语言，同样不利于交流。

3. 高级语言

随着计算机各种技术的发展，程序设计语言也在不断发展，为了脱离机器和便于非计算机专业人员的使用，人们开始用一种比较接近于自然语言（主要指英语）和数学语言的语言来编写程序，这样的语言称为高级语言。高级语言克服了面向机器语言的缺点，使得程序易读、易维护、易交流。高级语言发展很快，如今已达数百种之多，常用的高级语言如下。

① FORTRAN 语言：诞生于 20 世纪 50 年代中期，是第一个算法语言，适合于科学和工程计算。

② BASIC 语言：诞生于 20 世纪 60 年代中后期，简单易学，适合初学者学习。

③ Pascal 语言：诞生于 20 世纪 70 年代初，是一门结构化程序设计语言，适合于教学、科学计算、数据处理和系统软件开发等。

④ C 语言：诞生于 20 世纪 70 年代初，20 世纪 80 年代开始风靡全世界，适合于系统软件、数值计算、数据处理等。

⑤ Java 语言：诞生于 20 世纪 90 年代，是一种新型的跨平台分布式程序设计语言，具有简单、安全、稳定、可移植性强等特点，在网络环境中得到了广泛的应用。

一般地，人们把用高级语言或汇编语言编写的程序称为源程序。像用汇编语言编写的源程序必须经过汇编过程一样，用高级语言编写的源程序同样需要经过一个类似的过程翻译成计算机能识别的二进制代码程序，才能让计算机执行。翻译成二进制代码的方式通常有两种：编译方式和解释方式。目前，大多数常用的高级语言都采用编译方式。

编译方式是将源程序全部翻译成功能等价的机器语言程序（一般称为目标程序），然后经过连接程序将用户的目标程序与系统配置好的一些通用程序连接装配在一起，形成可执行程序，最后让计算机执行。编译方式经编译和连接得到的可执行程序可以重复执行无数次，其效率很高。

解释方式是通过解释程序逐句翻译、执行的，不产生目标代码程序，每次执行都要重新翻译，所以对那些重复执行的程序效率较低。

1.1.3　程序设计方法

程序设计技术的发展，是一个逐步提高的过程，是一个与实际应用需要互相制约和促进的螺旋式的发展进程。程序设计技术的不断进步，导致了计算机应用水平的飞跃；反之，计算机应用领域的扩展、软件规模和复杂度的提高又促进了程序设计技术的升级。在这个过程中，主要出现了结构化程序设计（SP，Structured Programming）方法和面向对象程序设计（OOP，Object-Oriented Programming）方法。

1．结构化程序设计

高级语言的开发和使用，使计算机的应用进入了一个新时期。20 世纪 60 年代出现的“软件危机”和关于“goto 语句应从高级语言中去掉”的讨论，促使人们重视对程序设计方法学的研究。随着一些规模大、复杂度高、使用周期长，以及投入人力、物力较多的大型程序设计任务的提出，程序设计的目标把可靠性、可维护性的要求放在了比高效率更重要的位置上，并促使人们研究程序设计的方法和风格，最终形成了结构化程序设计的基本思想。

结构化程序设计是一种设计程序的技术，主要采用自顶向下、逐步求精的设计方法和单入口、单出口的控制结构。自顶向下、逐步求精的设计方法符合人们解决复杂问题的普遍规则，可以提高软件开发的成功率。由此开发出的程序具有清晰的层次结构，易于阅读理解和修改调试扩充。结构化程序设计只包括顺序、选择和循环 3 种基本控制结构，并建议程序员不用或少用 goto 语句。

2．面向对象程序设计

结构化程序设计对于确定的问题来说，无疑是一种很好的方法，但对于千变万化的世界，特别是当今信息化的世界，一个成为产品的程序很难适应形势的变化，因为在程序的设计阶段问题是确定的，其数据结构、数据模型也可以确定，所以设计出的程序使用、操作没有问题，但当表示问题的数据结构发生变化时，程序就可能会出现问题，而维护起来也相当困难。原因就是当初的设计是面向数据进行自顶向下设计的，要修改就需要全部修改。本来客观世界就是由许许多多、各种各样的对象组成的，每个对象都有各自的内部状态（属性）和运动规律（行为方式），不同对象的相互作用和联系构成了各种各样不同的系统，构成了我们所面对的客观世界。如果按照客观世界对象的特点进行程序设计，就会适应形势的发展，这就是自 20 世纪 80 年代开始流行，现已广泛应用的面向对象的程序设计方法。

面向对象的程序设计方法吸取了结构化程序设计的基本思想和主要优点，将数据与对数据的操作放在一起作为一个相互依存、不可分割的整体来处理，这个整体称为对象。对象的内部状态（属性）用数据表示，运动规律用所谓的方法（也称成员函数）表示。对象是一个整体，所以我们说把数据隐藏在这个整体中了。对象之间的相互作用通过所谓的消息与对象的对外接口（成员函数的调用形式）实现。这个系统通过不断地向对象发送消息而使对象从初始状态到达终止状态，从而实现了问题的求解。

如果问题的数据结构发生了变化，只要描述问题的对象的对外接口不变，所有涉及这些数据结构的地方都不需要修改，而只需修改对象中的方法就行了。

对象是各种各样的，但总有相似性。例如，一匹白马和一匹黑马是两个对象，但它们都

具有马的特性，只是颜色有差异而已。将对象进行抽象、划分便得到了类。类描述了属于该类型的所有对象的性质，包括外部特征和内部实现，实际上是抽象数据类型的具体实现。反之，对象是类的具体实例。

20 世纪 90 年代的程序设计方式，由于有了面向对象程序设计技术的支持，发生了质的变化，未来的软件开发将向着可重用部件的组装方式发展，而可重用部件的存在形式就是类及其派生系统。

1.2 C语言的产生与发展

C 语言是在 20 世纪 70 年代初问世的。1978 年，美国电话电报公司（AT&T）的贝尔实验室正式发表了 C 语言。同时，由 Brian W. Kernighan 和 Dennis M. Ritchie 合著了著名的《The C Programming Language》，通常简称为《K & R》，也称为《K & R》标准。此书介绍的 C 语言成为后来被广泛使用的 C 语言版本的基础，被称为标准 C。但是，在《K & R》中并没有定义一个完整的标准 C 语言，因而出现了许多不同的 C 语言版本。后来，根据这些版本对 C 语言的扩充和发展，美国国家标准协会（ANSI，American National Standards Institute）重新制定了新的标准，并于 1983 年发表，通常称为 ANSI C。1988 年，按 ANSI C 标准又重写了《The C Programming Language》一书。很多 C 语言教材都是以 ANSI C 为基准编写的。目前，广泛流行的各种 C 语言版本的编译系统的基本内容是相同的，只是在个别地方有所不同。

早期的 C 语言主要用于 UNIX 操作系统。由于 C 语言的强大功能和各方面的优点逐渐被人们所认识，到了 20 世纪 80 年代，C 语言开始进入其他操作系统，并很快在各类大、中、小和微型计算机上得到了广泛的使用。特别是微型计算机上的 C 语言的普及，反过来又极大地推动了 C 语言的发展。

C 语言的产生和发展与 UNIX 操作系统是密不可分的，可以说，C 语言就是为编写 UNIX 操作系统而设计并加以实现的。UNIX 操作系统的源代码有 90%以上是用 C 语言编写的，它的流行应归功于 C 语言。

其实，C 语言并不是孤立产生的，它是在 B（BCPL 的第一个字母）语言的基础上发展起来的，而 B 语言又是在 A（ALGOL）语言基础上发展而来的。1960 年出现的 ALGOL 60 与硬件相差甚远，不宜用来编写系统程序。1963 年，英国的剑桥大学推出了比较接近于硬件的 CPL（Combined Programming Language）语言，该语言规模较大，不宜实现。1967 年，剑桥大学的 Martin Richards 对 CPL 进行简化推出了 BCPL（Basic Combined Programming Language）语言。1970 年，美国贝尔实验室又在 BCPL 语言基础上进一步简化推出了 B 语言，但 B 语言的功能太简单。于是，1972 年，贝尔实验室的 D. M. Ritchie 又在 B 语言的基础上推出了 C（BCPL 的第二个字母）语言。

C 语言为 UNIX 系统而设计，又由于 UNIX 系统的日益广泛使用而迅速得到推广，到 20 世纪 80 年代，C 语言已风靡世界。

1.3 C语言的特点

C 语言之所以风靡世界，是因为它有许多不同于其他语言的特点。在这些特点中，大部分是优点（与其他语言比），也有一少部分，对初学者来说，如果运用不当则会适得其反。

① C 语言简洁、紧凑，使用方便、灵活，库函数丰富、实用。ANSI C 一共只有 32 个关键字和 9 种控制语句，压缩了一切不必要的成分，用简单、规整的方法就可以构造出相当复杂的结构。

② C 语言程序书写格式较自由，降低了格式要求，从而降低了程序员的劳动强度。

③ C 语言的数据类型丰富，同时具有现代化程序设计语言普遍配置的多种数据结构。C 语言具有整型、实型、字符型、数组、指针、结构体、共用体和枚举类型等，能用来实现各种复杂的数据结构（如链表、树等）。特别是与地址密切相关的指针，使用起来比其他高级语言更加灵活多样。但指针对于初学者来说也是较难理解的。

④ C 语言的运算符十分丰富，包括算术运算符、关系运算符、逻辑运算符、位运算符、指针运算符、地址运算符等，还把括号、逗号、赋值号、强制类型转换等都作为运算符处理，连同复合赋值运算符共有 44 种运算符。灵活使用这些运算符可以把非常复杂的运算逻辑用最简单的方式表达出来，使得程序更加简洁。

⑤ 具有编写结构良好的程序所需要的各种控制流结构。C 语言具有实现顺序、分支和循环 3 种基本结构的控制语句。而且，C 语言的函数结构非常便于采用自顶向下、逐步求精的结构化程序设计方法将整个系统划分成若干功能相对独立的模块。C 语言是结构化程序设计的理想语言。

⑥ C 语言的目标代码质量高，执行效率高。一般，C 程序只比汇编程序生成的代码效率低 10%～20%，是各类高级语言中最快的。

⑦ 可以直接对硬件进行操作。C 语言允许直接访问物理地址，能实现汇编语言所能实现的大部分功能，如地址处理、二进制数位运算及指定用寄存器存放变量等。在硬件技术领域，C 语言有着非常广泛的应用基础，是最主要的程序设计语言。

由于 C 语言既具有高级语言的特点，又具有低级语言的特点，因此，也有人把它称为中级语言，特别适合作为编写系统程序和各种应用软件的工具。

⑧ 可移植性好。虽然 C 语言具有低级语言的功能，但与汇编语言相比，它不依赖于机器硬件，在硬件结构不同的各种型号的计算机之间不做修改或稍做修改即可实现程序的移植。

⑨ 语法限制不太严格，程序设计的自由度大。例如，对数组下标越界不做检查，由程序员自己保证其正确性；对变量类型的使用比较灵活，整型和字符型数据在一定范围内可以通用；等等。大多数高级语言编译程序对语法检查都比较严格，但 C 语言放宽了语法检查，允许程序员有较大的自由度，这就要求程序员必须具有一定的编程和调试程序的素质。

1.4　C语言程序简介

学习 C 语言的目的就是根据实际问题设计 C 语言程序，那么，C 语言程序是怎样构成的呢？为了了解 C 语言程序的构成，下面先来看几个例子。

【例 1.1】　在屏幕上显示字符串“This is a C program.”。

程序如下：

```
1   #include <stdio.h>      /* 文件包含*/
2   void main()             /* main 是主函数的函数名，表示这是一个主函数*/
3   {                       /* 函数体开始*/
```

```
4       printf("This is a C program.\n");
5   }                          /* 函数体结束*/
```

这是一个只由 main 函数构成的C语言程序，运行后，将在计算机屏幕上显示如下结果：

```
This is a C program.
```

为了更好地对 C 语言程序进行说明，本书将在程序的每行前都加上行号。注意，C 语言程序事实上是没有行号的。

该程序的第 1 行是文件包含的预处理命令，将头文件“stdio.h”的内容包含到该程序中，由于该头文件中包含了输出函数的信息，因此在该程序中就可以调用输出函数了。第 2 行是主函数 main 的说明部分，void 表示函数类型，说明该函数没有返回值，main 是函数名。需要注意的是，在任何一个 C 语言程序中必须有 main 函数，而且只能有一个。第 3～5 行是 main 函数的函数体，函数体以第 3 行的左花括号“{”开始，以第 5 行的右花括号“}”结束。main 函数的函数体只有一条语句，即第 4 行，它是一个函数调用语句，调用了标准库函数 printf，用于在屏幕上输出一个字符串“This is a C program.\n”，其中的“\n”是一个特殊的控制符——换行符，表示在输出最后一个字符“.”后换行，相当于按了回车键。

【例 1.2】 求整数 10、20 的和。

程序如下：

```
1   #include <stdio.h>
2   void main()
3   {
4       int first,second,sum;          /* 定义变量*/
5       first=10;                      /* 给变量赋值*/
6       second=20;
7       sum=first+second;      /*求和，将first和second的值相加后赋给sum*/
8       printf("sum=%d\n",sum);        /* 输出sum的值*/
9   }
```

这还是一个只有 main 函数的 C 语言程序，运行结果如下：

```
sum=30
```

该程序的第 4 行定义了 3 个整型变量，即在内存中开辟了 3 个用于存放整型数据的存储单元。第 5 行和第 6 行都是赋值语句，分别将整数 10 和 20 赋给整型变量 first 和 second，即将 10 和 20 分别存入分配给 first 和 second 的内存单元中。第 7 行也是赋值语句，取整型变量 first 和 second 的值（即从分配给 first 和 second 的内存单元中分别取出 10 和 20）相加后，把和赋给整型变量 sum。第 8 行调用 printf 函数，输出 sum 的值。

【例 1.3】 找出任意两个整数中较大的数。

程序如下：

```
1   #include <stdio.h>
2   int max(int x,int y)               /* 定义max函数 */
3   {   return(x>y?x:y);               /* 求出两数中的较大数并返回 */
```

```
4   }                                   /* max 函数结束 */
5   void main()
6   {   int num1,num2,m;
7       printf("Input the first integer number: ");  /* 提示输入第1个整数 */
8       scanf("%d",&num1);              /* 从键盘上输入第 1 个整数 */
9       printf("Input the second integer number: "); /* 提示输入第2个整数 */
10      scanf("%d",&num2);              /* 输入第 2 个整数 */
11      m=max(num1,num2);               /* 调用 max，计算两个数中的较大数 */
12      printf("max=%d\n",m);           /* 输出结果 */
13  }
```

这是一个由 main 函数和 max 函数构成的C语言程序，运行情况如下：

```
Input the first integer number: 6 ↙①
Input the second integer number: 9 ↙
max=9
```

该程序包括两个函数，第 2～4 行构成 max 函数，第 5～13 行构成 main 函数。第 2 行是 max 函数的说明部分，包括函数类型（int）、函数名（max）、形式参数及其定义（int x, int y）。第 3 行是 max 函数的函数体，是一条 return 语句，用于返回给调用函数一个整型值，该值是条件表达式“x>y ? x : y”的结果，即如果 x>y，则返回 x 的值，否则，返回 y 的值。第 7 行是输入提示，用于提示用户输入程序运行所需的数据，请读者注意体会和使用。第 8 行是函数调用语句，调用了标准库函数 scanf（该函数和 printf 函数是系统提供的标准输入和输出函数），用于从键盘上输入一个整型数据赋给整型变量 num1。第 9 行和第 10 行分别与第 7 行和第 8 行类似。第 11 行是一条赋值语句，赋值号“=”的右边调用了 max 函数，将 max 函数的返回值赋给赋值号“=”左边的整型变量 m。第 12 行是函数调用语句，调用了标准库函数 printf，输出 m 的值。

这 3 个程序中出现了数字、字母及一些符号，它们都属于 C 语言的字符集。程序中的“int”、“return”等是 C 语言中固定的具有特殊意义的词，称为关键字。“first”、“second”、“sum”、“num1”、“num2”等是用户给变量起的名字，称为标识符。“>”、“+”、“=”等是C语言的运算符。“,”、“;”、“"”、“{”、“}”等是 C 语言的分隔符。

1．字符集

字符是组成语言的最基本的元素。在 C 语言程序中，只能使用 C 语言字符集中的字符，不能使用字符集以外的字符。

C 语言的字符集（采用 ANSI 字符集）主要包括英文字母、阿拉伯数字和特殊字符 3 类：

① 大、小写英文字母（52 个）：A、B、C、…、Z、a、b、c、…、z

② 阿拉伯数字（10 个）：0、1、2、…、9

③ 特殊字符（29 个）：= + – * / % # ^ & ~ ( ) _ [ ] { } : ; , ' " ? • < > | ! 空格

① 带下划线的是输入部分，“↙”代表回车。

2. 标识符

标识符是程序员定义的单词，用于给程序中的一些实体命名，以便在程序中引用。如例 1.2 和例 1.3 中的“first”、“second”、“sum”、“num1”、“num2”、“m”等是标识变量的名字的标识符，而“max”是函数的名字的标识符。

一个优秀的程序员为标识符命名时一般会考虑标识符含义、用法等特征，而不采用毫无意义的字母及其组合。

3. 关键字

关键字是 C 语言中规定的一批英文单词，它们被赋予特殊的含义，不能另作他用。C 语言定义了 32 个关键字，如下：

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			

这些关键字将在后面逐步介绍。

4. 运算符（操作符）

运算符用来对运算对象进行规定（系统预定义的）的运算，并得到一个结果值。运算符通常由 1～2 个字符组成，如“+”表示加法运算，“=”表示赋值运算，“==”表示“相等”的判断等。有的运算符中的两个字符是分开的，比如“? :”表示条件运算（参见例 1.3 第 3 行）。

5. 分隔符

分隔符用于分隔各个词法记号或程序正文，分隔符用于表示程序中一个实体的结束和另一个实体的开始。常用的分隔符如下：

（　）{　}　，　：　；　空白

这些分隔符不表示任何操作，仅用于构造程序。

6. 注释

注释是程序员对程序进行的注解和说明，这些内容纯粹是为了阅读程序而加的，编译程序在预处理时就给去掉了，编译时不产生任何代码。但在程序中加入注释是一个好习惯，值得提倡。因为随着程序越来越复杂，程序及程序中的函数、语句、标识符等所要表达的含义会越来越难以理解，或者容易忘记。

C 语言只有一种注释，以“/*”开头，以“*/”结尾，中间可有任意多的字符，也可占任意行。在 C++语言中，可以使用行注释，以“//”开始，至行尾结束。

7．程序的组成

从上面几个例子还可以看出：

（1）C 语言程序是由函数构成的

一个 C 语言程序至少包含一个 main 函数（又称为主函数），也可以包含一个或多个其他函数。函数是 C 语言程序的基本单位。

（2）函数由函数说明和函数体两部分组成

① 函数说明

这部分主要包括函数类型、函数名、形式参数等。例如例 1.3 中的第 1 行：

```
int max(int x,int y)
```

其中，max 是函数名；int 是函数类型，表示函数要返回一个整型值；x、y 是函数的形式参数，都是 int 类型的。

② 函数体

这部分由一对大括号“{ }”界定。函数体内也分为两部分，一部分是变量的声明，另一部分是为了完成该函数规定的功能所需要的一系列可执行语句。

（3）C 语言程序从 main 函数开始执行，也随着 main 函数的结束而结束

其他函数只有被调用时才执行，并不按出现位置的先后顺序执行。

（4）C 语言程序的书写应该遵循一定的规范

主要包括以下 5 条规范。

① C 语言程序中的英文字母是区分大写的，如 Sum 和 sum 是不同的标识符。

② 表示各种名称的标识符一般使用小写字母，也可以使用大写字母，但关键字必须小写。

③ 每条语句都是以“;”结束的。一个程序行内一般只写一条语句，也可以写多条语句；而一条语句一般只写在一行上，也可以写在多行上。

④ 为了增强程序的可读性，在程序的任意位置，都可以用“/*……*/”插入注释内容。

⑤ 为了使程序结构清晰易读，程序的书写一般采用缩进的格式。

1.5　C语言的运行环境

1.5.1　C语言程序的执行步骤

计算机并不能直接识别用 C 语言编写的源程序，因此必须先把 C 语言源程序编译成目标程序，再把该目标程序与系统的函数库和其他的目标程序连接装配成可执行程序，计算机才能执行。C 语言程序的执行过程主要包括编辑、编译、连接和运行 4 个步骤。

1．编辑

这一步主要是使用某种编辑器生成 C 语言源程序文件，扩展名为.c。如果后面的步骤出现了错误，一般都是回到这一步进行修改，再一步一步地进行下面的操作。

2. 编译

这一步主要是将上一步产生的 C 语言源程序文件编译生成目标程序文件，扩展名为.obj。如果编译成功，则进行下一步操作；否则，返回上一步修改源程序，再重新编译，直至编译成功。

3. 连接

这一步主要是将目标程序和库函数及其他目标程序连接起来生成可执行文件，扩展名为.exe。如果连接成功，则进行下一步操作；否则，根据系统的错误提示，进行相应修改，再重新编译、连接，直至连接成功。

4. 运行

通过观察程序的运行结果，验证程序的正确性。如果出现逻辑错误，则必须返回第一步修改源程序，再重新编译、连接和运行，直至程序正确。

1.5.2 C语言程序的集成开发环境

在早期，编辑、编译、连接、运行等分别是通过相应的命令实现的，对用户来说，非常不便。后来，出现了集成开发环境，大大简化了用户的操作。通过 C 语言的集成开发环境能对 C 语言进行编辑、编译、连接和运行等操作，而集成开发环境又依赖于操作系统和计算机硬件，它们共同构成了 C 语言的运行环境。

微机上常用的 C 语言程序集成开发环境主要有 Turbo C 2.0、Borland C++ 3.0、Microsoft Visual C++ 6.0、Borland C++ Builder、Microsoft Visual C++ 2005 等，本书以 Visual C++ 6.0 为基础，多数程序也可在 Turbo C 2.0 下运行。

1. Microsoft Visual C++ 6.0

Visual C++ 6.0 是 Microsoft 公司在 1998 年推出的一款运行在 Windows 上的集成开发环境。使用 Visual C++ 6.0 可以对 C 语言程序进行各种操作，如建立、打开、编辑、保存、编译、连接、运行和调试等。在 Visual C++ 6.0 推出 10 年后，教育部考试中心做出决定，采用 Windows 版本的 C 语言开发环境 Visual C++ 6.0 作为考试环境。

（1）启动 Microsoft Visual C++ 6.0

通过“开始”菜单，或者通过桌面快捷方式，均可进入 Visual C++ 6.0 开发环境窗口，如图 1.1 所示。

（2）创建一个新的 C 语言工程

第一步，在图 1.1 中，选择菜单命令 File→New，显示 New（新建工程）对话框，如图 1.2 所示。

第二步，单击 Projects（工程）选项卡，选择 Win32 Console Application（Win32 控制台应用程序）。在 Location（位置）文本框中指定一个路径，在 Project name（工程名称）文本框中输入一个工程名（如 example），然后单击 OK（确定）按钮。

第三步，在弹出的对话框中选择 An empty project（一个空工程）单选项，然后单击 Finish

（完成）按钮，如图 1.3 所示。

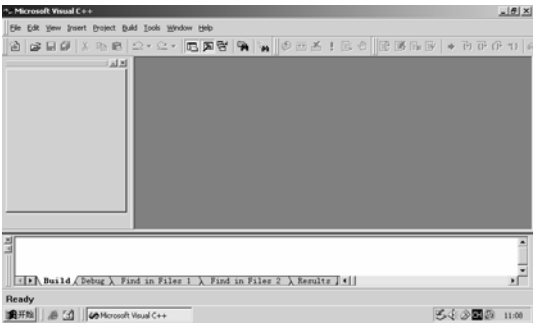


图 1.1　Visual C++ 6.0 开发环境窗口

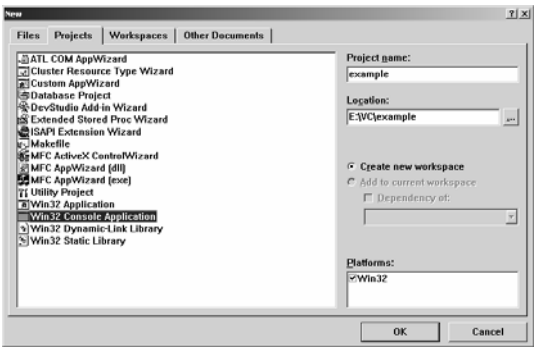


图 1.2　New（新建工程）对话框

第四步，在 New Project Information 对话框中单击 OK（确定）按钮，完成工程的建立。

（3）建立 C 语言源程序文件

第一步，选择菜单命令 Project→Add to Project→New，弹出如图 1.4 所示 New（新建文件）对话框。

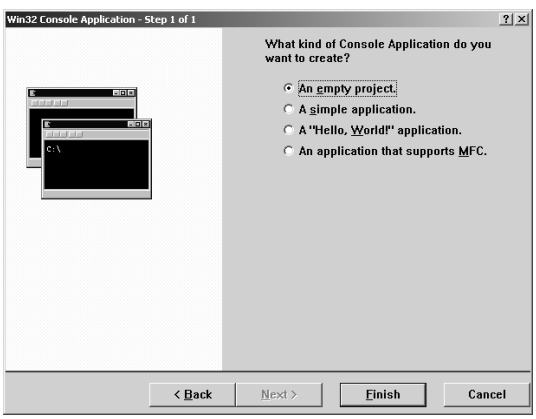


图 1.3　创建空的控制台应用程序

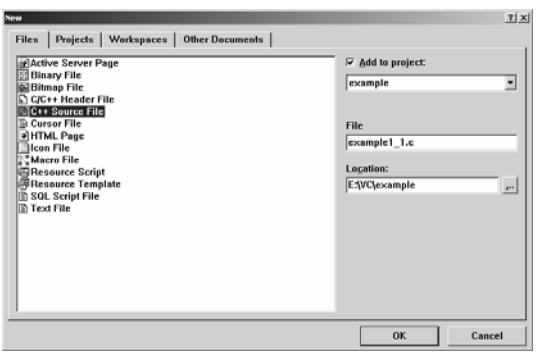


图 1.4　New（新建文件）对话框

第二步，在 Files（文件）选项卡中选择 C++ Source File，并输入文件名称（如 example1_1.c），单击 OK（确定）按钮，完成新建 C 语言源程序文件的工作。

（4）编辑 C 语言源程序文件内容

在文件编辑窗口中可以输入和修改自己编写的程序，如图 1.5 所示。

（5）建立并运行可执行程序

第一步，选择菜单命令 Build→Build example.exe（快捷键为 F7），生成可执行程序。如果正确输入了源程序，此时便成功地生成了可执行程序 example.exe。如果程序有语法错误，则屏幕下方的状态窗口中会显示错误信息，可以根据这些错误信息对程序进行修改后，重新选择菜单命令 Build→Build example.exe，建立可执行程序。

第二步，选择菜单命令 Build→Execute example.exe，运行程序，观察屏幕的显示内容，如图 1.6 所示。

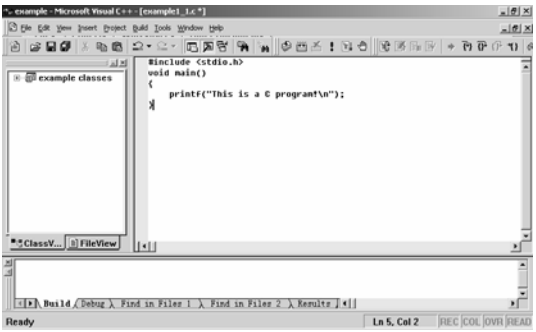


图 1.5　源程序编辑窗口

图 1.6　运行窗口

（6）关闭工作空间

这个程序完成后，选择菜单命令 File→Close workspace，关闭工作空间。重复步骤（3）～（6），可以创建并运行其他的 C 语言程序。

Visual C++ 6.0 的内容很多，初学者只要能掌握上面的步骤，能够编辑和运行 C 语言程序就可以了，一开始不要过多地涉及 Visual C++ 6.0 的相关概念，随着使用次数的增加，对 Visual C++ 6.0 的了解自然就会更多。

2．Turbo C 2.0

Turbo C 是美国 Borland 公司的产品，Borland 公司是一家专门从事软件开发和研制的公司。该公司相继推出了一套 Turbo 系列软件，如 Turbo BASIC、Turbo Pascal、Turbo Prolog 等，这些软件很受用户欢迎。该公司在 1987 年首次推出了 Turbo C 1.0 产品，其中使用了全然一新的集成开发环境，即使用了一系列下拉式菜单，将文本编辑、程序编译、连接及程序运行一体化，大大方便了程序的开发。1988 年，Borland 公司又推出了 Turbo C 1.5 版本，增加了图形库和文本窗口函数库等，而 Turbo C 2.0 则是该公司在 1989 年推出的。Turbo C 2.0 在原来集成开发环境的基础上增加了查错等一系列功能，在 DOS 环境下或 Windows 的命令提示符窗口中均可运行。

（1）启动 Turbo C 2.0

运行时，只要在 Turbo C 2.0 所在的文件夹下输入 TC 并按回车键即可进入其集成开发环境，如图 1.7 所示。在“Windows 的资源管理器”或“我的电脑”中，可以直接双击 Turbo C 2.0 所在文件夹下的 TC.exe。启动 Turbo C 后，其主菜单条横向排列在屏幕顶端，并被激活，其中 File 主项成为当前项。

图 1.7　Turbo C 2.0 的主界面

主菜单的下面，是 Edit（编辑）窗口和 Message（消息）窗口。在两个窗口中，顶端横线为双线显示，表示该窗口是活动窗口。

编辑窗口的顶端为状态行，其中：

① Line 6 Col 1：显示光标所在的行号和列号，即光标位置。

② Insert：表示编辑状态处于“插入”状态。当处于“改写”状态时，此处为空白。

③ C:NONAME.c：显示当前正在编辑的文件名。显示为 NONAME.c 时，表示用户尚未给文件命名。

屏幕底端是 7 个功能键的说明，以及 Num Lock 键的状态（显示“NUM”时，表示处于“数字键”状态；空白，表示处于“控制键”状态）。

（2）命令菜单的使用

① 按下功能键 F10，激活主菜单。如果主菜单已经被激活，则直接转下一步。

② 用左、右方向键移动光带，定位于需要的主项上，然后再按回车键，打开其纵向排列的子菜单。

③ 用上、下方向键移动光带，定位于需要的子项上，按回车键即可。执行完选定的功能后，系统自动关闭菜单。

注意：菜单激活后，又不使用，可再按 F10 或 Esc 键关闭，返回原来状态。

（3）退出 Turbo C 2.0

退出 TC 主要使用以下两种方法：

① 菜单法：File→Quit（先选择 File 主项，再选择并执行 Quit 子项）。

② 快捷键法：Alt+X（先按下 Alt 键，然后按字母键X，然后同时放开）。

（4）编辑并保存一个C语言源程序

激活主菜单，选择并执行 File→Load 项（快捷键为 F3），在 Load File Name 窗口中，输入源程序的文件名。文件名的输入有两种方法：直接输入和选择输入。

① 直接输入

按照文件名的组成，逐个字符地输入。如果是已经存在的文件，系统就在编辑窗口中显示该文件的内容，可以进行编辑和修改。如果是新文件，则给出一个空白编辑窗口，可以输入新的源程序。如果该文件不在当前目录下，则需要使用路径名和（或）盘符。

② 选择文件（仅适用于已经存在的源程序文件）

直接按回车键，打开当前文件夹下后缀为.c 的所有文件的文件名窗口，用上、下、左、右方向键，将光带定位于所需的文件名上，按回车键即可选中相应的文件。

在编辑源程序的过程中，随时都可以按 F2 键（或 File→Save），将当前编辑的文件存盘，然后继续编辑。记住，这是一个良好的习惯！

（5）编译、连接单个源程序文件

选择并执行 Compile→Make EXE File 项（快捷键为 F9），则 Turbo C 将自动完成对当前正在编辑的源程序文件的编译、连接，并生成可执行文件。

如果源程序有语法错误，系统将在屏幕中央的 Compiling（编译）窗口底端提示 Error：Press any key（错误：按任意键）。

此时，按空格键，屏幕下端的 Message（消息）窗口被激活，显示出错（或警告）信息，光带停在第一条消息上。这时 Edit（编辑）窗口中也有一条光带，它总是停在编译有错误在源代码中的相应位置上。

注意：当用上、下键移动消息窗口中的光带时，编辑窗口中的光带也随之移动，始终跟踪源代码中的错误位置!

（6）运行与查看结果

① 运行当前正在编辑的源程序文件

选择并执行 Run→Run 项（快捷键为 Ctrl+F9），程序运行结束后，将返回到编辑窗口。

当认为自己的源程序不会有编译和连接错误时，也可以直接运行（即跳过对源程序的编译、连接步骤）。这时，Turbo C 将一次完成从编译、连接到运行的全过程。

② 查看运行结果

选择并执行 Run→User Screen 项（快捷键为 Alt+F5）。查看完毕后，按任意键返回编辑窗口。

如果发现逻辑错误，则可在返回编辑窗口后，进行修改，然后再重新编译、连接、运行，直至正确为止。

（7）编辑下一个新的源程序

选择并执行 File→New 项，如果屏幕提示如下确认信息：

```
NONAME.C not saved.Save?（Y/N）
```

则说明当前正在编辑的源程序还没有保存。此时，如果需要保存，则键入“Y”，进入下一步操作；否则，键入“N”（不保存），跳转到②。

① 系统提示换名：

```
<d:><path>\NONAME.C
```

直接输入为源程序文件起的名字。

② 系统给出一个空白的编辑窗口，可以开始编辑下一个新的源程序。

（8）使用在线帮助

在任何窗口（或状态）下，按 F1 键激活活动窗口（或状态）的在线帮助。在编辑状态下，按 Ctrl+F1 可激活与光标所在位置相关的在线帮助。

在在线帮助窗口中，可使用 PageDown（下一页）、PageUp（上一页）、Esc（关闭在线帮助）等按键。

习　题　1

1.1　什么是程序？什么是程序设计语言？

1.2　什么是结构化程序设计方法？

1.3　程序设计语言的发展经历了哪几个阶段？

1.4　用高级语言编写的源程序要让计算机执行，需经过哪几步处理？

1.5　在集成开发环境中输入并运行本章的 3 个例题，体会 C 语言程序的运行步骤。

第 2 章　C语言程序设计基础

学习目标

通过本章学习，应该掌握：

- 算法的基本概念，用流程图或 N-S 图描述简单的算法，结构化设计方法的基本思想
- 数据类型的有关概念，数据的取值范围
- 常量的表示方法，变量的定义方法
- 运算符的书写格式，算术表达式的构成及运算规律
- 赋值运算

2.1　算法与程序设计步骤

做任何事情都有一种或多种方法、步骤，解决一个实际问题的方法和步骤就称为算法。实际上，程序就是算法在计算机上的实现。著名计算机科学家沃思（N.Wirth）曾经提出一个公式：程序=数据结构+算法，其中，数据结构是对数据的描述，主要是指数据的类型和组织形式，算法是对操作的描述，主要是指求解问题的具体操作步骤。对初学者来说，根据实际问题确定一个科学的算法是程序设计的第一步，也是比较关键的一步。

2.1.1　算法及其表示

虽然用程序设计语言表示算法是程序设计的最终目的，但由于程序设计语言是一种形式化语言，要求严格，在初学阶段或设计比较复杂的问题时很难直接使用，因此，要了解和掌握几种直接表示算法的方法。

1．自然语言描述

采用自然语言描述算法，通俗易懂，但不太严格，烦琐冗长，不直观，不清晰。

【例 2.1】　有两个存储单元 *a* 和 *b*，要求将它们的值互换。

按存储器的性质，如果将单元 *a* 的值直接送到单元 *b* 中，那么就会覆盖 *b* 原来的内容，因此，需要借助一个临时单元 *c* 来交换。

具体算法如下。

步骤 1：将单元 *a* 的值送给单元 *c*。

步骤 2：将单元 *b* 的值送给单元 *a*。

步骤 3：将单元 *c* 的值送给单元 *b*。

【例 2.2】　求 1+2+3+4+…+10。

假设用存储单元 *S* 存放累加和，具体算法如下。

步骤 1：把 0 存入 *S* 单元中。

步骤 2：把 1 加到 *S* 中（即取 *S* 中的内容 0 加 1 后得到 1，再把 1 送回 *S* 单元中）。

步骤 3：把 2 加到 *S* 中。

步骤 4：把 3 加到 *S* 中。

……

步骤 10：把 9 加到 *S* 中。

步骤 11：把 10 加到 *S* 中。

步骤 12：把 *S* 中的结果输出。

这样的算法虽然正确，但不科学，不实用。可以设一个计数器单元 *n*，每重复一次 *n* 增 1，直到 *n* 大于 10 为止，求和操作可以改为“*n*+*S* 送 *S*”。修改后的算法如下：

步骤 1：将 0 送到 *S* 中。

步骤 2：将 1 送到 *n* 中。

步骤 3：把 *n* 的值加到 *S* 中。

步骤 4：*n* 增 1。

步骤 5：若 *n*≤10 则转回步骤 3，否则执行步骤 6。

步骤 6：输出 *S* 的值。

改进后的算法很科学，是适合编程的算法。

由于自然语言描述容易产生歧义，因此，除了很简单的问题外，一般不用自然语言表示算法。

2．流程图

流程图通常采用一些几何图形来代表各种类型的操作，在图形内标明文字或符号来表示操作的内容，并用箭头来表示操作的顺序。图 2.1 给出了流程图中使用的图形符号及代表的含义。

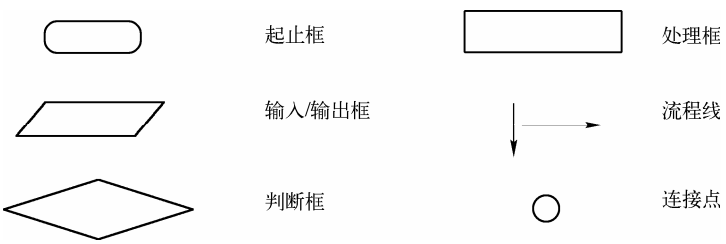


图 2.1　流程图常用符号

图 2.2 所示为例 2.2 的流程图[①]。

用流程图表示算法，直观形象，易于理解。但由于流程图允许使用箭头随意跳转，对表示算法的层次结构非常不利，并且流程图所占的篇幅较大，作图工作量也很大。

3．N-S图

针对流程图存在的缺点，1973 年，美国的计算机科学家 I. Nassi 和 B. Shneiderman 提出了结构化程序设计的流程图，因为他们二人的名字是以 N 和 S 开头的，所以称为 N-S 图。相对于流程图，N-S 图更能体现结构化程序设计的思想，因此，本书推荐使用 N-S 图。

① 在流程图中，用 0=>*S* 表示“将 0 送 *S*”或“*S* 置 0”。

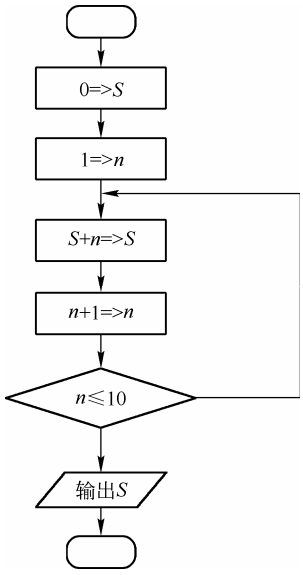


图 2.2　求 $S=\sum_{n=1}^{10} n$ 的流程图

N-S 图去掉了传统流程图中的箭头，全部算法写在一个大的矩形框内，组成 N-S 图的是一系列“方块”，因此 N-S 图又叫盒图。N-S 图规定了一些图形元素，用来表示结构化程序设计的 3 种基本结构——顺序结构、选择结构、循环结构。一个算法可由若干个代表基本结构的图形元素像搭积木一样构造而成。

（1）顺序结构

顺序结构是一系列顺序执行的运算和处理。顺序结构的 N-S 图如图 2.3 所示，其中，A 和 B 分别代表一个基本的操作。

（2）选择结构（判断结构、分支结构）

选择结构通常是根据一个条件 P 是否成立来选择下一步应该执行哪一种处理。图 2.4 所示为选择结构的 N-S 图，其中，T 表示条件成立，F 表示条件不成立。

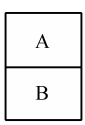


图 2.3　顺序结构

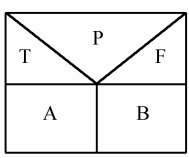


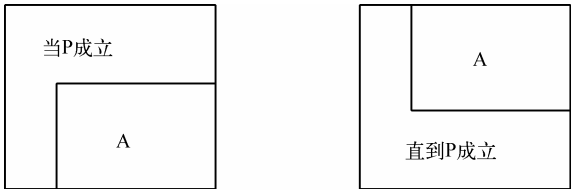
图 2.4　选择结构

（3）循环结构（重复结构）

循环结构根据条件 P 是否成立来判断是否重复执行操作 A。通常有两种结构形式，一种是“先判后做”，另一种是“先做后判”。

“先判后做”称为当型循环：表示当条件 P 成立时就执行 A，然后返回再判断 P 是否成立，若成立则再重复执行 A，……，重复下去，直到条件 P 不成立。其 N-S 图如图 2.5（a）所示。

“先做后判”称为直到型循环：表示先执行 A，再判断条件 P 是否满足，不满足则重复执行 A，……，直到条件成立。其 N-S 图如图 2.5（b）所示。



（a）　　（b）

图 2.5　循环结构

结构化程序设计的三种基本结构具有以下共同的特点：

- 只有一个入口；
- 只有一个出口；
- 结构内的每一部分都有机会被执行到；
- 结构内不存在“死循环”。

图 2.6 所示为求 1～10 的 10 个正整数的和的算法描述（见例 2.2），图 2.7 所示为求 10 个任意整数中的最大数的算法描述。读者可以通过这两个例子仔细体会顺序、选择和循环这三种基本结构的特点。

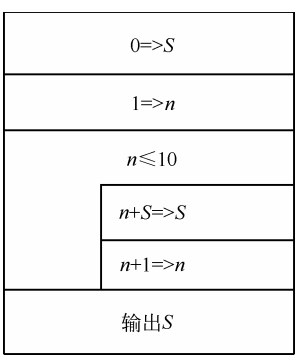


图 2.6　求 $S=\sum_{n=1}^{10} n$ 的 N-S 图

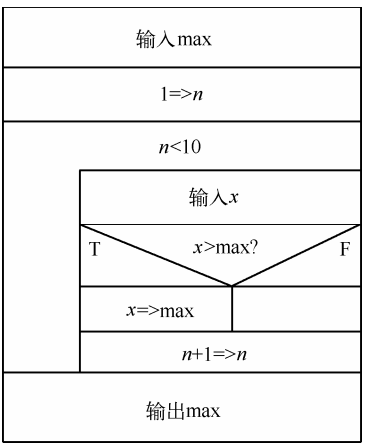


图 2.7　求 10 个数中最大数的 N-S 图

2.1.2　程序设计步骤

根据实际问题设计应用程序大致要经过以下 4 个步骤。

① 分析问题：主要包括两方面的任务，一是确认已知数据和未知结果，并设置好要表示的变量；二是确定问题求解的方法，选择一种简单、易于理解、适合编程的算法，确定解决问题的步骤。利用 C 语言编程解决问题，在这一步通常采用结构化的分析方法，即采用自顶向下、逐步细化的方法。

② 画流程图或 N-S 图：在 C 语言的程序设计过程中一般使用 N-S 图。这一步主要是用 N-S 图描述问题的算法。

③ 编写程序：将上一步画的 N-S 图转化为 C 语言程序。

④ 运行并调试程序：运行编写好的程序，如果程序正确，则得到预期的结果；否则，就要进行程序调试，直至得到正确的结果为止。

【例 2.3】　求 3 个数中的最大数。

对于这个问题，目前只做分析问题和画 N-S 图这两步，后面的两个步骤以后随着学习内容的深入再进行介绍。本题的算法很简单，读者可能一下子就能写出具体的算法，但为了便于理解结构化的分析方法，此处采用自顶向下、逐步细化的方法来设计算法。首先，从整体进行设计，算法应该分为 3 个如图 2.8（a）所示的子问题 S1、S2、S3，然后再对每个子问题进行细化。设 3 个数分别为 *a*、*b*、*c*，并用 max 存放其中的最大数，则可以把 S1、S3 描述为如图 2.8（b）和（c）所示的方法。这两步已成为计算机能实现的基本操作。对 S2 来说，可以分成 S2.1 和 S2.2 两个独立的子任务，如图 2.8（d）所示，再对 S2.1 和 S2.2 进行细化得到图 2.8（e）和（f），将细化得到的图按顺序反插到图 2.8（a）中便得到了该题目的详细算法，如图 2.8（g）所示。

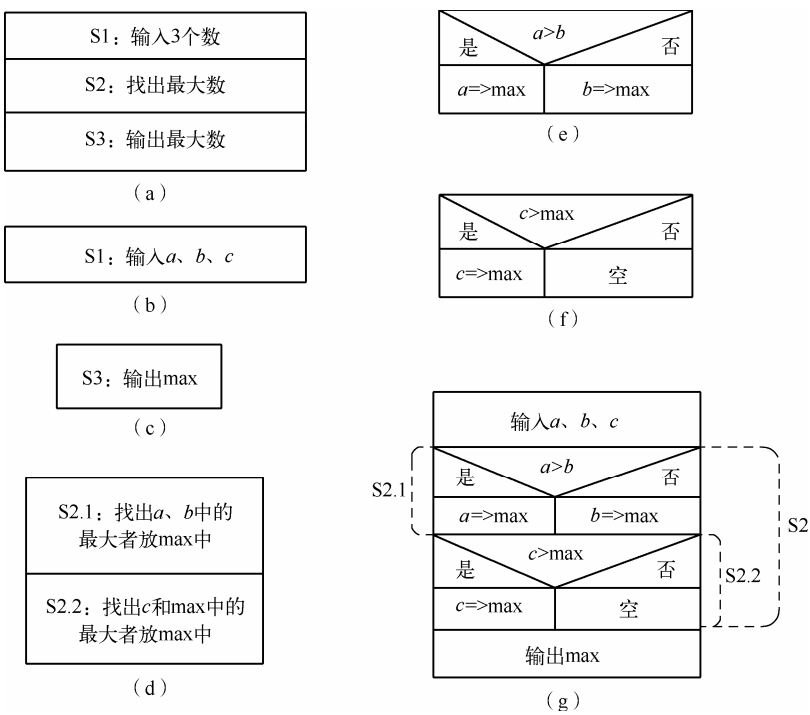


图 2.8　求 3 个数中的最大数

这个例子主要是让读者了解程序设计的步骤（前两个步骤），尤其是结构化的分析方法，读者一定要认真体会。在后面的学习中，读者要逐步理解和掌握这种方法，并能够利用这种方法解决实际问题。对于程序设计的步骤（包括后面的两个步骤），读者在后面的学习中会一直使用，注意在理解的基础上进行掌握。

2.2　数 据 类 型

数据是程序的处理对象，可由常量或变量等表示。程序中使用的数据必须属于某种数据类型，因为数据类型决定了数据对象的存储形式、取值范围及能进行的运算。在 C

语言中，数据类型一般包括基本数据类型、构造数据类型、指针类型和空类型等四大类，如图 2.9 所示。

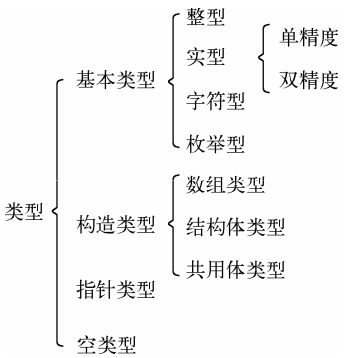


图 2.9　数据类型

基本类型是由系统事先定义好的不可再分隔的类型，可以直接利用这些类型名定义数据。构造类型是由基本类型组成的更为复杂的类型。指针是一种特殊的，同时又是具有重要作用的数据类型，其值用来表示某个量在内存中的地址。空类型主要用于特殊指针变量和无返回值函数的说明。

本章主要介绍除枚举类型之外的基本数据类型，其类型名和名称，以及各类型数据在计算机内所占的二进制位数和取值范围如表 2.1 所示。需要说明的是，数据类型的位数和取值范围与所运行环境有很大关系，这里列举的是 Windows 下的 Visual C++ 6.0 中的情况。

表 2.1　C 语言的基本数据类型

类 型 名	名　称	位　数	取 值 范 围
(signed)char	字符型	8	–128～127
unsigned char	无符号字符型	8	0～255
(signed)short(int)	短整型	16	–32 768～32 767
unsigned short(int)	无符号短整型	16	0～65 535
(signed)int	整型	32	–2 147 483 648～2 147 483 647
unsigned(int)	无符号整型	32	0～4 294 967 295
(signed)long(int)	长整型	32	–2 147 483 648～2 147 483 647
unsigned long(int)	无符号长整型	32	0～4 294 967 295
float	单精度实型	32	$\pm 3.4\times(10^{-38}\sim10^{38})$，6 位精度
double	双精度实型	64	$\pm 1.7\times(10^{-308}\sim10^{308})$，15 位精度

注：

① 关键字 signed 和 unsigned 以及 short、long 称为修饰符。

② 使用 signed 修饰的数据类型称为有符号类型，可以取正数或负数；使用 unsigned 修饰的数据类型称为无符号类型，不能取负数。

③ 修饰符 signed 可以省略，没有修饰符的数据类型都是有符号类型。

④ 当用 short、long、unsigned 修饰 int 时，int 可以省略。

关于数据类型的更详细的内容，将在后面逐步介绍。随着学习的深入，读者将会一步一步地理解和掌握数据类型的相关内容。

C 语言中，最常用的数据类型是 char、int 和 double，分别用于字符型、整型和实型数据的处理。

2.3　常量和变量

对于基本数据类型量，按其取值是否可以改变又分为常量和变量两种。在程序中，常量可以不经说明而直接引用，而变量则必须先说明后使用。

2.3.1　常量

在程序中可以直接使用的数据称为常量，其值在程序运行过程中不可以改变，包括整型常量、实型常量、字符型常量、字符串常量及符号常量等。

1．整型常量

允许使用的整型常量如下。

① 十进制数：以非零数字开始的整数，如–434、0、2345。

② 八进制数：以数字 0 开始的整数，如 0127、030、–0123。

③ 十六进制数：以 0x 开始的整数，如 0x1A、0x4A2、–0xB7。

整型常量可以用下列方法来显式地指明其类型：

① 当任一整型常量后跟一个字母 l（或 L）时，表示是长整型数。例如，3456L、0x2300L、–0146L 分别为十进制、十六进制、八进制长整数。

② 当任一整型常量后跟一个字母 u（或 U）时，表示是无符号整型数。例如，34u。

通常，整型常量是以补码的形式存储在内存中的，最高位（即符号位）为 0 表示正数，为 1 表示负数。但是，也允许使用无符号整数，即将最高位不看做符号位，而用来表示数值。微型计算机中，短整型数在内存中一般用 2 个字节（16bit）存储，整型数和长整型数用 4 个字节（32bit）存储。无符号短整型、整型、长整型比相应的有符号短整型、整型、长整型表示的数的范围在正数的方向上扩大了 1 倍，但不能表示负数。

2．实型常量

实型常量即实数，又称浮点数，由整数部分和小数部分组成，只有十进制数的形式。实数有两种表示方法：小数形式和指数形式。

小数形式：由符号、数字和小数点组成，如 0.5、15.02、–534.、.789、0.0、.0、0.等。注意，小数点必须有，不可省略。

指数形式：即采用可科学计数法表示形式，由尾数、字母 e 或 E 和指数部分组成，例如，1.23E–2、–32e4 分别表示 1.23×10^{-2}、-32×10^{4}。注意，字母 e 或 E 之前必须有数字，之后的指数部分必须为整数。

实数一般分为单精度和双精度两种。单精度实数用 4 个字节（32bit）存储，有效数字位数为 6～7 位；双精度实数用 8 个字节（64bit）存储，有效数字位数为 15～16 位。

3．字符型常量

用一对单引号括起来的单个字符称为字符型常量，如'A'、'b'、'*'等。需要注意的是，单引号只作为定界符，不是字符型常量的内容。字符型常量可以是 ASCII 字符集中任意的可打印字符，包括空格字符。字符型常量具有数值，其值就是该字符的 ASCII 码值（如'A'的值是 65，'a'的值是 97），可以作为整数参与运算。例如，'a'+3 表示将'a'字符的 ASCII 码值 97 与整数 3 相加，结果为 100。

另外，还有一种称为转义字符的特殊形式的字符型常量，用于表示 ASCII 字符集中的控制代码或某些其他功能代码，如表 2.2 所示。转义字符也是由一对单引号括起来的一个字符，以“\”开始后跟一个字符，或一个“\”后跟一个八进制数或十六进制数组成。例如，'\'"表

示双引号，'\r'表示回车，'\0'代表空字符，'\x20'和'\40'都代表空格。

表 2.2　转义字符序列

转义序列	功　能	转义序列	功　能
\n	换行	\a	报警
\t	横向跳格（TAB 键）	\b	退格
\r	回车	\f	走纸换页（用于打印机）
\'	单引号	\v	竖向跳格
\"	双引号	\\	反斜杠字符“\”
\ddd	1 到 3 位八进制数所代表的字符（d 代表 1 位八进制数）		
\xhh	1 到 2 位十六进制数所代表的字符（h 代表 1 位十六进制数）		

通常，以一个字节来存放一个字符。在内存中，字符型数据是以其 ASCII 码值的形式存储的，其存储形式与整数的存储形式类似。如果系统将字符型数据处理成有符号的整数，则有符号字符型数据和整型数据在 0～127 的范围内是可以通用的。如果系统将字符型数据处理成无符号的整数，则无符号字符型数据和整型数据在 0～255 的范围内是可以通用的。不同系统的处理方式是不同的，因此，具体使用时要查看系统的有关资料，或者上机试一下。

4．字符串常量

字符串常量是用一对双引号括起来的若干个字符，例如，"I am a student"、"How are you?"、"b"、"abc567"、""（即空串）。

字符串常量中可以包含空格、转义字符等任意字符，也可以是中文。注意：双引号作为字符串的定界符，不是字符串常量的内容，当计算字符串常量的长度时，双引号不计算在内。

在 C 语言中，字符串常量的长度不受限制，编译程序在处理字符串常量时自动在其最后加一个'\0'（null）作为“字符串结束标志”。因此，字符串的最小存储长度总比字符串中的实际字符个数多 1。例如，"I am a student"的存储长度至少是 15，而不是 14。对于'a'和"a"，前者是字符常量，占 1 个字节，后者是字符串常量，最小存储长度为 2，因为它实际上包含两个字符'a'和'\0'。

5．符号常量

可以将在程序中多次出现的某常量定义成一个标识符，这个标识符称为符号常量。符号常量一般（不是必须）用大写英文字母表示，必须“先定义后使用”，其定义形式如下：

```
#define 符号常量名 所代表的值
```

例如：

```
#define  PI  3.14
```

这样的符号常量在程序中一旦定义之后，在其后面的程序中只要碰到 PI 标识符，系统就认为它的值为 3.14。

注意：在程序中绝对不可以给符号常量赋值。例如，PI 定义为符号常量之后，程序中如果出现如下的类似语句：

```
PI=3.1415926;
```

则是错误的。也就是说，符号常量一旦定义，就只能使用，而不能改变其值。

使用符号常量可以提高程序的可读性以及常量修改的一致性。例如，将上面的 3.14 改为 3.1415926，则随后的程序中所有的 PI 值都变成了 3.1415926。

2.3.2　变量

1．变量的命名和标识符

在程序中可以改变其值的量称为变量。变量名是标识符。

我们刚介绍的符号常量名以及后面要介绍的函数名、数组名、结构体名和共用体名等都是标识符。标识符是由用户定义的，要符合下面的规定。

① 标识符是以字母或下划线开头，由字母、数字和下划线组成的字符序列。例如，area、sum、score、_rec、PI 等都是合法的标识符。

② 标识符是区分大小写的。例如，SUM、sum、Sum 是 3 个不同的标识符。

③ 不能使用 C 语言的关键字定义其他的标识符。例如，if、for、else、while 等都是关键字，不能再作为其他的标识符使用。

④ 定义标识符时，尽量做到“见名知义”，以增加程序的可读性。例如，可以用 area 作为面积的标识符，用 score 作为成绩的标识符，等等。

⑤ 标识符的长度视具体运行环境而定。

2．变量的定义

变量是用于存储数据的，因此每个变量必须属于某种数据类型。而且，在程序中，使用变量之前必须先将其声明为属于某种数据类型。例如：

```
char  c1,c2;                /* 说明 c1、c2 为字符型变量 */
int  sum,score;             /* 说明 sum、score 为整型变量 */
long  L;                    /* 说明 L 为长整型变量 */
float  f1,f2;               /* 说明 f1、f2 为单精度实型变量 */
double  d1,d2;              /* 说明 d1、d2 为双精度实型变量 */
```

由此可见，声明变量的形式为：

数据类型名 变量名列表

变量的类型被声明后，编译程序就为其分配相应类型的存储单元，可以在程序中给该变量赋相应类型允许的值，并决定了该变量所能够执行的操作。例如，在某程序中有了上面的定义之后，编译程序会为变量 c1、c2 各分配 1 个字节的存储单元，为变量 sum、score、f1、f2 各分配 4 个字节的存储单元。在程序中可以为变量 c1、c2 赋字符型的值，如'a'、'A'；为变量 sum、score 赋整型的值，如 100、–90。还决定了 sum、score 可以进行模运算（求余数），而 f1、f2 就不能进行模运算。

注意：C 语言中没有字符串类型，处理该类型时要使用在后面介绍的指针或字符数组。

3. 变量的初始化

变量在声明的同时可以给其赋值，称为变量的初始化。例如：

```
char  c1='*',c2='a';   /* c1、c2 为字符型变量，同时给 c1、c2 分别赋初值'*'、
                          'a'*/
int  sum=340;          /* sum 为整型变量，同时给 sum 赋初值 340 */
float  f1=3.14,f2;     /* f1、f2 为实型变量，同时给 f1 赋初值 3.14，但没有给
                          f2 赋初值，也就是说 f2 没有被初始化 */
```

注意：初始化不是在编译阶段完成的，而是在程序运行时对相应变量赋初值的，相当于有一个赋值语句。例如：

```
int sum=340;
```

相当于：

```
int sum;              /* 说明 sum 为整型变量 */
sum=340;              /* 赋值语句，将 30 赋予 sum */
```

如果任何时候都不希望改变某个变量的值，可以将该变量声明为“固定”变量，方法是在定义并初始化该变量时，为其声明加上保留字 const。例如：

```
const int t=1000;
```

这样，程序中就不能改变 t 的值了，任何时候变量 t 的值都为 1000。

2.4　函　　数

在编写程序时，经常会遇到一些初等函数的计算，如求绝对值、三角函数、指数函数、对数函数、平方根等。如果计算这些函数的程序也要用户自己来编写，那就太麻烦了。因此，为了方便使用，C 语言提供了丰富的内部函数，也称标准函数，供用户编写程序时引用。使用时，只要按照规定的标准写法，编译程序便可自动予以处理。但 C 语言的标准函数并不是系统函数，不是必需的，编程时完全可以不用这些函数，而自己编写。C 语言的标准函数的定义按函数类型存放在不同的“头文件”中，其代码则存放在相应的“库”中，因此这些函数又叫库函数。

下面介绍一些常用的数学函数，见表 2.3。

表 2.3　常用数学函数

函　数　名	功　　能	函　数　名	功　　能
fabs(x)	求 x 的绝对值	sqrt(x)	计算 x 的平方根
sin(x)	计算 sin(x)的值	cos(x)	计算 cos(x)的值
exp(x)	求 e^x 的值	pow(x, y)	计算 x^y 的值
log(x)	求 ln$x$，即 ln x	log10(x)	求 lg$x$

使用数学函数时，应该在源文件中的开始位置使用：

```
#include "math.h"
```

以便把头文件"math.h"包含到源文件中。

引用内部函数时，只需写出相应的函数名，并在后面的括号中给出所要计算的自变量值，即可得到所需要的函数值。例如，只要写出 sqrt(4.0)，就可得到函数值 2.0，只要写出 fabs(−100.0)，就可得到其绝对值 100.0。

使用标准函数时，必须注意以下两点：

① 使用三角函数时，必须注意角度的单位是“弧度”；

② 引用标准函数时，自变量要写在括号里面，自变量可以是常量、变量或表达式。

2.5　运算符和表达式

C 语言中的运算符和表达式数量之多，在高级语言中是少见的。正是丰富的运算符和表达式使 C 语言的功能十分完善，这也是 C 语言的主要特点之一。

C 语言的运算符不仅具有不同的优先级，而且还有一个特点，就是它的结合性。在表达式中，各运算量参与运算的先后顺序不仅要遵守运算符优先级别的规定，还要受运算符结合性的制约，以便确定是自左向右进行运算还是自右向左进行运算。这种结合性是其他高级语言的运算符所没有的，因此也增加了 C 语言的复杂性。

1．运算符的分类

（1）算术运算符

算术运算符用于各类数值运算，包括+、−、*、/、%、++、−−。

（2）关系运算符

用于比较运算，包括>、<、==、>=、<=、!=。

（3）逻辑运算符

用于逻辑运算，包括&&、||、!。

（4）位运算符

参与运算的量，按二进制位进行运算，包括&、|、～、^、<<、>>。

（5）赋值运算符

用于赋值运算，分为三类：简单赋值=、复合算术赋值+=、−=、*=、/=、%=和复合位运算赋值&=、|=、^=、>>=、<<=。

（6）条件运算符

条件运算符（？：）是一个三目运算符，用于条件求值。

（7）逗号运算符

逗号运算符（,）用于把若干表达式组合成一个表达式。

（8）指针运算符

指针运算符包括取内容*和取地址&两种运算符。

（9）求字节数运算符

该运算符（sizeof）用于计算数据类型所占的字节数。

（10）强制类型转换运算符

该运算符用于数据类型的强制转换，形式为：(数据类型)。

（11）其他运算符

其他运算符主要包括括号()、下标[]、成员（→，.）等几种。

2．运算符的优先级和结合性

在 C 语言中，运算符的运算优先级共分为 15 级。1 级最高，15 级最低。在表达式中，优先级较高的先于优先级较低的进行运算。当一个运算量两侧的运算符优先级相同时，则按运算符的结合性所规定的结合方向处理。

C 语言中各运算符的结合性分为两种，即左结合性（自左至右）和右结合性（自右至左）。例如，算术运算符的结合性是自左至右，即先左后右，这种自左至右的结合方向就称为左结合性，而自右至左的结合方向称为右结合性。最典型的右结合性运算符是赋值运算符。C 语言运算符中有不少为右结合性，应注意区别，以避免错误。

改变运算顺序的方法是使用圆括号“( )”。

3．表达式

表达式是常量、变量、数组元素、函数等运算对象和运算符以及括号的有意义组合。C 语言的表达式既可以单独作为语句，也可以在其他语句中作为测试的条件以及调用函数的参数使用。在一个表达式中，运算符起着关键的作用。表达式按其所含运算符和运算对象的不同，可分为算术表达式、赋值表达式、逗号表达式、关系表达式和逻辑表达式等。本章只介绍前三种表达式，后两种将在后面的章节中介绍。

2.6　算术运算符与算术表达式

1．基本的算术运算符

- +：加法运算符或正值运算符，如 4+7、+3。
- −：减法运算符或负值运算符，如 4−7、−3。
- *：乘法运算符，如 4*7。
- /：除法运算符，如 4/7。
- %：模运算符，也称求余运算，要求%两侧均为整型数据，如 7%4 的值为 3。

需要注意的是，当使用除法运算符“/”时，如果除数和被除数都是整型数据，则运算结果也是整型数据，例如 7/3 的结果是 2，1/2 的结果为 0。如果参加运算的两个数据中有一个是实型数据，则运算结果是 double 型数据，因为所有实型数据都按 double 型数据进行运算，所以 1./2 的结果为 0.5。

模运算符“%”的运算结果为两个整型运算对象做除法运算的余数。例如，7%3 的结果为 1。

使用运算符时，需要注意：

① 不允许两个算术运算符紧挨在一起，应该使用括号将其隔开。例如，x/−y 应改为 x/(−y)。

② 不能像数学算式那样省略乘号，也不能用圆点“·”代替乘号。例如，(x−y)(x+y)

或(x−y)·(x+y)是错误的，应改为(x−y)*(x+y)。

2．算术表达式

单个运算对象及用算术运算符和括号将运算对象连接起来的有意义的表达式都称为算术表达式。其中，运算对象包括常量、变量、函数等。例如，y、'c'、10、x1+y*z−a/23+'c'−100 都是合法的算术表达式，其中，x1、y、z、a 是已经定义的变量（已经声明了类型，并且完成了赋值），'c'是字符型常量，10、23、100 是整型常量。

在将数学算式写成 C 语言表达式时，对于一些有具体值的符号，如圆周率π等，要使用该符号所代表的值，或者先定义一个符号常量代替该值，再使用该符号常量；而对于一些没有具体值的符号，如α、β等，则要先定义一个标识符，再使用该标识符。

3．算术运算符的优先级与结合性

每个运算符都有自己的优先级和结合方向。如果一个运算对象的两边有不同的运算符，则先执行优先级高的。例如，乘、除的优先级就高于加、减的优先级。对于算术表达式，在优先级相同的情况下，按自左向右的顺序运算。使用括号可以改变运算顺序，需要注意的是，表达式中只能使用圆括号，如果有必要，可以使用嵌套的圆括号，如 3.5*(2+x/(a+b))。

基本算术运算符的优先级（由高到低）如图 2.10 所示。

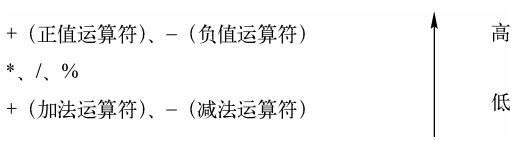


图 2.10　基本算术运算符的优先级

其中，+（正值运算符）和−（负值运算符）只有一个运算对象，称为单目运算符，具有右结合性。*、/、%、+（加法运算符）和−（减法运算符）有两个运算对象，称为双目运算符，具有左结合性。

4．自增、自减运算符

“++”和“−−”是自增、自减运算符，它们的功能是使一个变量的值加 1 或减 1，它们都是单目运算符，而且其运算对象只能是整型变量或指针变量。

“++”和“−−”既可以放在作为运算对象的变量之前，也可以放在变量之后。这样，就形成了 4 种表达式：i++、i−−、++i、−−i。前两个表达式是先使用 i 的值，然后再使 i 的值加、减 1；后两个表达式是先使 i 的值加、减 1，然后再使用变量 i 的值。例如：

```
int  m1,n1,m2,n2;
int  i=1;
int  j=1;
m1=i++;
n1=i;
m2=++j;
n2=j;
```

该程序片段运行后，各变量的值为：m1=1，n1=2，m2=2，n2=2，i=2，j=2。

“++”和“--”的优先级较高，高于所有的算术运算符，并且具有右结合性。注意：“++”和“--”只能用于变量，不能用于常量或表达式。例如：

```
++10;
(x+y)--;
```

都是错误的。

如果把“++”、“--”运算与其他运算符组合在一起构成表达式，就会比较让人费解，例如，表达式“a+++b--”代表什么含义呢？又是如何运算的呢？下面的程序演示了该表达式的运算结果：

```
#include <stdio.h>
void main()
{   int a=3,b=5,c;
    c=a+++b--;
    printf("a=%d,b=%d,c=%d\n",a,b,c);
}
```

因此，不建议读者这么做。

2.7　赋值运算符与赋值表达式

在 C 语言程序中可以将符号“=”作为一个运算符处理，称为赋值运算符，用于构造赋值表达式。

1．赋值运算符

赋值表达式的一般形式为：

变量 = 表达式

赋值运算符的功能就是将“=”右边表达式的值赋给左边的变量。例如，“a=3”是一个赋值表达式，其作用是执行一次赋值操作（或称赋值运算），把常量 3 赋给变量 a。赋值表达式的运算结果是“=”右边的表达式的计算结果，因此，“a=3”的结果为 3。

关于赋值运算符的使用，需要注意：

① 在一个表达式中可以连续使用多个赋值运算符，运算时按自右向左的顺序进行，即赋值运算符具有右结合性。例如，“a=b=c=0”相当于“a=(b=(c=0))”，即最后的结果为 a、b、c 都是 0。

② 赋值运算符的优先级较低（仅高于逗号运算符）。例如，语句“i=j=m*n;”的执行顺序是：首先计算出算术表达式 m*n 的值，然后处理赋值表达式“j=m*n”，将 m*n 的值赋给 j，最后处理赋值表达式“i=j=m*n”，将 j=m*n 的值赋给 i。

③ 当赋值运算符两边的数据类型不一致时，系统自动将结果转换成赋值运算符左边变量的类型后再赋值，具体如下。

- 整型数据赋给实型变量时，数值不变，但以浮点数形式存储到变量中。例如，假设 f

为单精度实型变量，则执行“f=123”后的结果是使 f 的值变为 123.000000。

- 实型数据赋给整型变量时，会自动舍弃实数的小数部分。例如，假设 i 为整型变量，则执行“i=234.856”后的结果是使 i 的值变为 234。
- 将较短（位数较少）的整型数赋给较长的整形变量时，保持数值大小不变直接赋值，例如，对于短整型 s=123，若 l 为长整型变量，则“l=s”的结果还是 123。
- 将较长（位数较多）的整型数赋给较短的整型变量时，会“截断”这些较长的整型数，只保留低位的数字。例如，如果 c 为字符型变量，“c=300”的结果为 44。“截断”是在内存中的二进制位上进行的，可以理解为求余数运算，因此“c=300”等价于“c=300 % 256”（256 是 8 个二进制位所能表达的整数的个数）。
- 将有符号整型数赋值给无符号整型变量或者将无符号整型数赋值给有符号整型变量时，要考虑符号的变化（维持补码不变）。例如，整型变量的位数为 32，如果 i 为有符号整型变量，u 为无符号整型变量，则“i=−1”结果为−1，“u=−1”结果为 4294967295（$2^{32}-1$），而“i=4294967295”结果为−1。

考虑到赋值表达式的复杂性，通常，程序中应显式地使用类型转换，以避免自动类型转换带来的不确定性。

2．复合赋值运算符

在赋值运算符“=”之前加上+、−、*、/、%、<<、>>、%、|、^ 10 种双目运算符，可以构成复合的赋值运算符，其优先级和结合性都与赋值运算符“=”一样。例如：

```
a+=10        等价于  a=a+10
a−=b         等价于  a=a−b
a*=b         等价于  a=a*b
x%=y*8       等价于  x=x%(y*8)
x/=y+4       等价于  x=x/(y+4)
a%=b+2       等价于  a=a%(b+2)
```

2.8　逗号运算符与逗号表达式

在 C 语言中，逗号不仅是一个分隔符，还可以作为运算符出现，用于连接两个表达式，形成逗号表达式。逗号运算符的运算对象是任意表达式，当然也包括逗号表达式。逗号运算符的优先级在所有运算符中是最低的，具有左结合性，也称为“顺序求值运算符”。

逗号表达式的一般形式为：

表达式 1，表达式 2，……，表达式 *n*

在程序执行时，按从左到右的顺序执行组成逗号表达式的各表达式，并将最后一个表达式（即表达式 *n*）的值作为整个逗号表达式的值。例如：

① 5+5, 6+1 的值是 7。

② a=5*6, a*2 的值是 60，而 a 的值是 30。

③ (a=4*5, a*2), a+5 的值是 25，a 的值为 20。

在上面的例子中，逗号表达式“a=5*6,a*2”的计算顺序为：先计算 a=5*6（赋值运算符

比逗号运算符的优先级高），得到 a 的值 30，再计算 a*2，得到整个逗号表达式的值 60。逗号表达式“(a=4*5,a*2),a+5”的计算顺序为：先计算 a=3*5，得到 a 的值 20，再计算 a*2（注意 a 的值并没有改变，还是 20），最后计算 a+5，得到整个逗号表达式的值 25。

其实，逗号表达式的作用无非是把若干个表达式“串联”起来，并不具备运算能力。在许多情况下，使用逗号表达式的目的只是想分别得到各个表达式的值，而并非一定需要得到和使用整个逗号表达式的值。

注意：并不是在所有出现逗号的地方都组成逗号表达式。例如，在变量说明和函数参数表中的逗号只是用做各变量之间的分隔符。

2.9　数值型数据间的混合运算

整型、单精度型、双精度型数据可以进行混合运算。由于字符型数据与整型数据通用，因此，整型、实型（包括单、双精度）、字符型数据之间可以进行混合运算。由于参与混合运算的各数据的类型不同，因此，在运算时需要进行类型转换。

在 C 语言中，共有两种类型转换：第一种是自动类型转换，即在运算时不必由用户指定，系统自动进行类型转换；第二种是强制类型转换，即使用强制类型转换运算符来实现强制类型转换，一般用于自动类型转换不能达到目的的情况。

1．自动类型转换

例如，4/2.1+34%3+'x'−11.1234*'a'是合法的。在进行运算时，不同类型的数据要由系统按照由低到高的次序进行类型转换，其转换规则如图 2.11 所示。

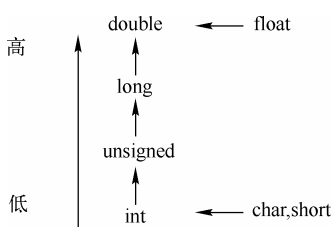


图 2.11　数据类型转换规则

图 2.11 中横向向左的箭头表示必定的转换。例如，只要是字符型或短整型数据参与运算必定先转换成 int 型数据，float 型数据在运算时为提高运算精度一律转换成 double 型数据。

图 2.11 中纵向向上的箭头表示不同类型的数据转换时的方向。例如，一个 int 型数据和一个 double 型数据进行运算时，要先将 int 型数据直接转换成 double 型数据，然后再在两个同类型（double）数据之间进行运算，结果为 double 型数据。注意，在这里，int 型数据是直接转换成 double 型数据的。千万不要理解成 int 型先转换为 unsigned 型，再转换成 long 型，最后才转换为 double 型。又如，一个 int 型数据和一个 long 型数据进行运算时，也是先将 int 型数据直接转换成 long 型数据再进行计算。

在进行数据类型的混合运算时，需要注意：

① 当字符型数据转换成整型时，其 ASCII 码值在 128～255 之间的字符（最高位为 1）会转换成负数。

② 当按照 short→int→long 的顺序进行类型转换时，其符号和数值都能够正确转换。当按照相反的顺序进行类型转换时，如果原数值不超出转换后较低类型的数值范围，一般也能正确转换；如果超出较低类型所能表示的数值范围，则会出现转换错误。

③ 由 float 型到 int 型转换时，会截去小数部分，如果数值超过 int 型的范围，一般会出错。

④ 由整数到实数转换时，如果该整数不能被精确转换（如 long 型转换为 float 型时），可能会丧失一些精度。

⑤ 由 double 型到 float 型转换时，当数值大小不超过 float 型的范围时，可能会丧失一些精度，但转换结果基本正确。

⑥ 如果参与混合运算的数据中有 float 型的，其结果一定是 double 型的，因为在运算时 float 型会先转换为 double 型再进行计算。

例如，假设已经声明 i 为 int 型变量，f 为 float 型变量，d 为 double 型变量，L 为 long 型变量，c 为 char 型变量，则混合运算 i*f−d/L 的运算顺序如下：

① 进行 i*f 的运算，先将 i 和 f 都转换成 double 型再相乘，运算结果为 double 型；

② 进行 d/L 的运算，先将 L 转换成 double 型再相除，运算结果为 double 型；

③ 将第①步的结果与第②步的结果相减，运算结果为 double 型。

又如，假设 c 为 char 型变量，则混合运算 10+'a'−c 的运算顺序如下：

① 进行 10+'a'的运算：先将'a'转换成整数 97，再进行 10+97，得到整型结果 107；

② 将第①步的结果与 c（注意：先将 c 转换成 int 型）相减，运算结果为 int 型。

2．强制类型转换

例如，假设 x 已经声明为 float 型，则“x%3”是不合法的，必须使用如下形式：(int)x%3。其中，(int)x 就是强制类型转换，将 x 的值由 float 型转换成为 int 型。

强制类型转换的一般形式为：

(数据类型名)(表达式)

注意，表达式应该用括号括起来，若是单个变量，则可以省略括号。例如：

- (int)(x*13.45/12)表示将 x*13.45/12 的运算结果转换成 int 型。
- (double)d 表示将 d 的值转换成 double 型，d 本身的类型不变。
- (float)(10%5)表示将 10%5 的运算结果转换成 float 型。
- (int)(x+y)表示将 x+y 的运算结果转换成 int 型。
- (int)x+y 表示将 x 的值转换成 int 型，再与 y 相加，注意与上式的区别。

需要指出的是，在进行强制类型转换时，只是得到了一个所需类型的中间变量，原变量或表达式的类型并没有改变。例如，假设 d 已经声明为 float 型，则“(int)d”进行强制类型运算后得到一个 int 型的中间变量，它的值等于 d 的整数部分，而 d 的类型未变，仍然是 float 型。

习　题　2

2.1　对给定的一个年份，判断是不是闰年。要求用 N-S 图描述该算法。提示：能被 4 整除但不能被 100 整除的年份或能被 400 整除的年份是闰年。

2.2　用 N-S 图描述求 $n!$ 的算法。

2.3　判断一个正整数 n 是否为素数。要求用 N-S 图描述该算法。提示：所谓素数就是除了 1 和本身之外，不能被任何数整除的数。

2.4　用 N-S 图描述求 100 个数中的最小的数的算法。

2.5　用 N-S 图描述将 3 个数按从小到大的顺序排列的算法。

2.6　找出 10～1000 内各位非零数字之积为 12 的所有整数，如 26，62，206，340，430，621，…。要求用 N-S 图描述该算法。

2.7　找出 10～1000 之间所有的对称数。所谓对称数就是正序读和逆序读的值相等，如 33、151、313 等。要求用 N-S 图描述该算法。

2.8　百钱买百鸡。鸡翁一值钱五，鸡母一值钱三，鸡雏三值钱一，百钱买百鸡，问鸡翁、鸡母、鸡雏各多少。要求用自顶向下、逐步细化的方法设计该算法，并用 N-S 图描述。

2.9　根据实际问题设计应用程序大致要经过哪几个步骤?

2.10　判断下列常量的正误，并指出正确的常量的数据类型：

32.01	'0'	0	"012"	"0"	–0123	0876
–0.1e–2	'123'	ABF	0xef87	'\n'	0x12gf	5–3

2.11　判断下列变量名是否正确：

Aaa	abc	o–3	if	+y	_sss	*x
A#3	f.3	a/b	main	–test	t3	e_2

2.12　选择题

（1）下面不正确的字符常量是（　　）。

A. "c"　　B. '\\'　　C. '\567'　　D. 'K'

（2）下列 4 组选项中，均是不合法的用户标识符的选项是（　　）。

A. W　P_0　do　　B. b–a　goto　int

C. float　la0　_A　　D. –123　abc　TEMP

（3）设 C 语言中，一个 int 型数据在内存中占 2 个字节，则 unsigned int 型数据的取值范围为（　　）。

A. 0～255　　B. 0～32767

C. 0～65535　　D. 0～2147483647

（4）若 x、i、j、k 都是 int 型变量，则计算表达式 x=(i=4,j=16,k=32)后，x 的值为（　　）。

A. 4　　B. 16　　C. 32　　D. 52

（5）设以下变量均为 int 类型，则值不等于 7 的表达式是（　　）。

A. (x=y=6, x+y, x+1)　　B. (x=y=6, x+y, y+1)

C. (x=6, x+1, y=6, x+y)　　D. (y=6, y+1, x=y, x+1)

2.13　填空题

（1）若有定义 int m=5,y=20;，则计算表达式 y+=y–=m*=y 后，y 的值是_____。

（2）若一个 int 型数据在内存中占 2 个字节，则 int 型数据的取值范围为_____。

（3）若 s 是 int 型变量，且 s＝6，则表达式 s%2+(s+1)%2 的值为_____。

（4）若 a 是 int 型变量，则表达式(a=4*5,a*2),a+6 的值为_____。

（5）若 x 和 a 均是 int 型变量，则计算表达式 x=(a=4,6*2)后的 x 值为_____，计算表达式 x=a=4,6*2 后的 x 值为_____。

（6）若 x 和 n 均是 int 型变量，且 x 和 n 的初值均为 5，则计算表达式 x+=n++后 x 的值为_____，n 的值为_____。

（7）若有定义 int x=3,y=2;float a=2.5,b=3.5;，则表达式(x+y)%2+(int)a/(int)b 的值为_____。

2.14　字符型常量和字符串常量有什么区别?

2.15　将下列数学算式转换为 C 语言表达式：

（1）$v_0t+\frac{1}{2}gt^2$　　（2）$\frac{-b+\sqrt{b^2-4ac}}{2a}$

（3）$\frac{2\sin\left(\frac{\pi}{4}\right)}{|x|}$　　（4）$\frac{\lg x+\lg y}{2}$

2.16　已知 a=5，b=3，x=3.5，y=2.6，求下列表达式的值：

（1）x+a%3*(int)(x+y)%2/4

（2）(float)(a+b)/2+(int)x%(int)y

（3）a/b+x/y–(int)x%3*(float)a/b

2.17　设变量 c 是 char 型，i 是 int 型，j 是 long 型，f 和 r 是 float 型，d 是 double 型，请写出下列表达式类型转换的过程：

```
r=(c*i)+(f/d)-(i+f)+(i/j)
```

2.18　写出下列程序的执行结果。

（1）

```
#include <stdio.h>
void main()
{   int m=1,n=2;
    int k=++m;
    printf("k=%d",k);
    k=m+n++;
    printf("m=%d,n=%d,k=%d\n",m,n,k);
    k=--n-m;
    printf("m=%d,n=%d,k=%d\n",m,n,k);
    k=(m>=n);
    printf("k=%d\n",k);
}
```

（2）

```
#include <stdio.h>
void main()
{   int a=4,b=3,c=2,d=1;
    printf("%d\n",(a+b,b+c,c+d));
}
```

（3）

```
#include <stdio.h>
void main()
{   int a=1,b=2,c=0;
```

```
    printf("%d\n",a++-1);
    printf("%d\n",b/++a);
}
```

（4）

```
#include <stdio.h>
void main()
{   int a=-5,b=10;
    float x=1.75,y=2.34e-03;
    printf("a+b=%d\n",a+b);
    a++;b--;
    printf("a*b=%d\n",a*b);
    printf("x+y=%f\n",x+y);
}
```

第 3 章　顺序程序设计

学习目标

通过本章学习，要求掌握：

- 语句的基本概念
- 赋值语句
- 字符输入函数 getchar 与输出函数 putchar
- 格式输出函数 printf 及输出格式控制
- 格式输入函数 scanf 及数据输入方法
- 顺序程序设计

3.1　C语言语句概述

程序是由语句组成的，任何程序都可以分为 3 种基本结构，即顺序结构、选择结构和循环结构。不同结构用不同的控制语句实现。C 语言提供了多种语句来实现这些程序结构。本章首先介绍基本的 C 语言语句，并以此构造最简单的程序，实现基本应用，为后面各章的学习奠定基础。

3.1.1　C语言语句的基本概念

常量、变量、运算符、表达式等都是一些基本语法项目，由这些基本项目，按一定的规则进行组合，并以分号结束，即可构成各种语句。

语句是表达算法命令或编译指示的基本语言单位。在第 1 章中已经看到，C 语言程序是由各种语句组成的，因此，熟练地掌握各种语句，是学好 C 语言的基础。要掌握 C 语言的语句，必须抓住两个基本点：语法和语义。语法，就是语句的书写规则。在 C 语言中，各种语句都有其特定的书写形式。使用时，必须按照规定的格式书写。否则，编译程序将不予编译，或者编译出错误的程序。语义，就是语句的含义和作用，也就是语句的功能。C 语言中的各种语句，都有其特定的功能，必须准确地理解和掌握。

例如，

```
k++;
x=x+1;
x+=1;
b=sqrt(a);
```

都是 C 语言的语句。

需要注意的是，C 语言的语句都是“可执行语句”。在 C 语言中，没有“非执行语句”。

另外，C 语言语句的书写是比较自由的，可以一行写几个语句，也可以将一条语句拆开写在几行上，但通常“一行一句”，即不要把多条语句写在一行上，以免给阅读程序带来不便，也影响程序的调试。

3.1.2　C语言语句的分类

C 语言程序的执行部分是由语句组成的，程序的功能也是由执行语句实现的。C 语言语句的形式很多，一般可分为 5 类：表达式语句、函数调用语句、控制语句、复合语句、空语句。

1．表达式语句

表达式语句由表达式加上分号“;”组成，执行表达式语句就是计算表达式的值，其一般形式如下：

```
表达式;
```

事实上，任何一个表达式后加上“;”就构成了表达式语句，如算术表达式、关系表达式等。最典型的是用赋值表达式组成的赋值语句。

需要注意的是，有的表达式语句有意义，有的表达式语句则没有实际意义。例如：

①“a=1”是一个赋值表达式，而加上分号的“a=1;”则是一条赋值语句；

②“x=y+z”是一个赋值表达式，而加上分号的“x=y+z;”则是一条赋值语句；

③“i++”是一个表达式，而加上分号的“i++;”则是一条自增语句，i 的值增 1；

④“y+z;”是一条加法运算语句，但计算结果不能保留，无实际意义；

⑤“x;”也是一条“语句”，但不执行任何运算，毫无意义。

表达式能构成语句是 C 语言的一个重要特色。其实，下面要介绍的函数调用语句也可以看成是表达式语句，因为函数调用（如 fabs(x)）也是表达式的一种。只是为了便于理解和使用，才把“函数调用语句”和“表达式语句”分成两类来介绍。由于 C 语言程序中的大多数语句是表达式语句（包括函数调用语句），因此，有时也把 C 语言称为“表达式语言”。

2．函数调用语句

函数调用语句由函数名、括号、实际参数加上分号“;”组成，其一般形式为：

```
函数名(实际参数表);
```

执行函数语句就是，调用函数体并把实际参数赋予函数定义中的形式参数，然后执行被调函数体中的语句，求取函数值。

例如：

```
printf("This is a C program.");
```

是一个函数调用语句，将调用库函数 printf，在屏幕上输出字符串“This is a C program.”。

当然，有函数调用不一定就是函数调用语句，例如：

```
y=fabs(x);
```

调用了 fabs 函数，但最后执行的是赋值运算，即把 x 的绝对值赋给变量 y，因此，这是一个赋值语句，属于表达式语句。可见，语句的分类是由表达式中最后一个运算决定的。如果把这个语句改为“fabs(x);”，就变成了函数调用语句，当然由于计算结果被丢弃了，也就没有实际意义了。

3．控制语句

控制语句用于控制程序的流程，以实现程序的各种基本结构。控制语句由特定的语句定义符组成。C 语言共有 9 种控制语句，分成 3 类。

① 选择语句又称为条件语句、判断语句，用于实现选择结构的程序设计，包括 if 语句和 switch 语句。

② 循环语句又称为重复语句，用于实现重复执行的循环结构的程序设计，包括 while、do-while 和 for 三个语句。

③ 转移语句用于实现程序的跳转，包括 break、continue、goto 和 return 四个语句。

控制语句的基本形式和使用方法将在后面的章节中逐步介绍。

4．复合语句

把任意多条语句用大括号“{}”括起来就构成了复合语句。例如：

```
{
    x=y+z;
    a=b+c;
    printf("%d%d",x,a);
}
```

是一条复合语句，由 3 条语句组成。

对于复合语句，需要注意：

① 在程序中应把复合语句看成是单条语句，而不是多条语句；

② 复合语句内的各条语句都必须以分号“;”结尾，最后一条语句的结尾也必须有分号“;”；

③ 复合语句中还可以包含复合语句，即复合语句允许嵌套，但必须保证嵌套是完整的，不允许交叉嵌套。

例如：

```
1   {   x=1;
2       {   y=x*x;
3           x++;
4   }
5           y=x;
6       }
```

从形式上看，似乎有两条复合语句，它们是第 1～4 行组成的一条复合语句和第 2～6 行构成的另一条复合语句。但真实情况是，这里只有一条复合语句，由第 1～6 行组成。该复合语句嵌套了另一个复合语句，由第 2～4 行组成。也就是说，该程序段应该写成：

```
1   {   x=5;
```

```
2       {   y=x*x;
3           x+=10;
4       }
5       y=x;
6   }
```

不难想象，当程序中出现复合语句时，应该注意每一行的缩进与对齐方式，以免造成误解。

5. 空语句

只有一个分号“;”的语句称为空语句。

空语句什么也不做，可用来作为被转向点，或循环语句中的循环体（循环体是空语句，表示循环体什么也不做）。

例如：

```
while(getchar()!='\n') ;
```

该语句的功能是，只要从键盘输入的字符不是回车则重新输入。这里的循环体为空语句。

当程序中出现连续的两个分号“;”时，一般可把后面的分号看做空语句。但要注意的是，随意添加空语句可能造成错误，例如：

```
if(x>y)z=x;;
else z=y;
```

由于 z=x 的后面多使用了一个分号，造成下面的 else 子句出现语法错误，else 与 if 之间多出了一个空语句。

3.2　赋 值 语 句

赋值语句由赋值表达式加上一个分号构成，是用于实现计算和赋值的一类最基本的语句，其一般形式如下：

```
可赋值对象 v=表达式 e;
```

其中，v 是变量和数组元素等可赋值对象的标识符，“=”是赋值运算符，e 是表达式。

有时，也把复合赋值运算符看做赋值运算符，因而把符合赋值运算表达式构成的语句看做赋值语句。例如，可把“x += 10;”看做赋值语句。

赋值语句执行时，系统将先计算出“=”右边表达式 e 的值，然后将该值赋给“=”左边的变量 v。例如，以下赋值语句：

```
x=-x;
```

的执行过程为：

① 求变量 x 的相反数；

② 将求得的 x 的相反数赋给变量 x。

在赋值语句的使用中需要注意以下几点。

① 如果赋值运算符两边的类型不一致，则系统在算出表达式的值之后，先将该值转换为左边变量的类型，然后再赋值给左边的变量。例如，假设变量 x 已经声明为整型，则

```
x=3.45;
```

的执行过程为：先将实型数 3.45 转换为整型数 3，再将整型数 3 赋给整型变量 x。

② 由于在赋值运算符“=”右边的表达式可以又是一个赋值表达式，因此：

```
变量=(变量=表达式);
```

是成立的，从而形成嵌套的情形。其展开之后的一般形式为：

```
变量=变量=…=表达式;
```

例如：

```
a=b=c=50;
```

按照赋值运算符的右结合性，实际上等价于：

```
c=50;
b=c;
a=b;
```

③ 赋值表达式和赋值语句是有区别的。赋值表达式可以出现在任何允许表达式出现的地方，而赋值语句则不能。例如：

```
y=fabs(x=sin(x)/x);
```

是合法的，该语句的功能是计算 sin(x)/x 并赋值给 x，再取 x 的绝对值赋给 y。而

```
y=fabs(x=sin(x)/x;);
```

是非法的，因为“x =sin(x)/x;”是语句，不是表达式，不能作为函数 fabs 的参数。

④ 在变量说明中，不允许连续给多个变量赋初值。例如：

```
int a=b=c=50;
```

是错误的，必须写成：

```
int a=50,b=50,c=50;
```

⑤ 在变量说明中给变量赋初值和在赋值语句中赋值是不同的。给变量赋初值是变量说明的一部分，赋初值后的变量与其后的其他同类变量之间仍必须用逗号间隔，而赋值语句则必须用分号结尾。

3.3　数据的输入与输出

通常，多数程序都包含输入和输出语句。一个实用的程序可以没有输入语句，但不能没有输出语句。没有输入语句的程序通常意味着每次运行程序都处理相同的数据，得到相同的

结果。如果没有输出语句，则程序就没有实际价值。

根据结构化程序设计思想，任何程序都可以简化为由 3 个模块组成：数据输入、数据处理、数据输出。数据输入模块负责接收用户输入的数据并转化为程序能够识别、处理的形式，通常与输入设备相关。数据处理模块负责对输入数据按指定的算法进行加工和处理，计算出结果，它是程序的核心，但并不是程序的全部。数据输出模块负责将内存中的计算结果转化为可以在输出设备上呈现出来的形式，并送往输出设备。

因此，编写任何程序前，都要首先规划数据的输入、输出方法。

3.3.1 输入、输出基本概念

利用计算机解决问题的一般步骤为：先分析问题，再编写程序，最后运行和调试程序直至得到正确的结果。注意，用户要想看到结果，必须将结果显示在屏幕上，或者用打印机打印出来。这是因为，结果是用变量或数组等来存放的，而变量和数组中的内容都保存在内存单元中，而内存单元中的内容，用户却是无法看到的。因此，在计算机中引入了“输出”的概念，其功能就是将保存在内存单元中的变量或数组中的值显示在屏幕上或者用打印机打印出来。

同时，程序在执行时，有可能需要用户提供计算所需的数据，这就引入了另一个概念“输入”，其功能是利用键盘等输入设备为变量或数组等提供数据，并保存到内存单元中，供程序计算时使用。

总之，借助于输入和输出，用户就可以与内存单元进行交互了。输入可以把信息从外设传送到内存，输出则把信息从内存传送到外设。这样，信息从输入设备流向内存，经处理后又从内存流向输出设备。

在计算机系统中，键盘和显示器是最常见的输入设备和输出设备，常称为标准输入/输出（Standard Input/Output）设备。在程序设计中，进行数据的输入、输出操作时，如果不具体指明所使用的外部设备，就隐含地表示使用的标准输入/输出设备。也就是说，如果不特别说明，程序的输入设备是键盘，输出设备则是显示器（屏幕）。

C 语言本身没有提供输入语句和输出语句，输入和输出的功能是靠输入和输出函数实现的。在标准 C 语言函数库中事先存放了一大批标准输入/输出函数，用于在标准输入/输出设备上进行数据的输入/输出。其中，最常用也是最主要的是 printf 函数和 scanf 函数。事实上，由于 C 语言的函数库是可选的，所以编程时也可以不用系统提供的库函数，而是自己编写输入和输出函数，在程序中通过函数调用自己编写的输入和输出函数同样可以实现数据的输入和输出。C 语言的这一特点，为开发系统程序提供了便利。

在使用 C 语言库函数时，要用到预编译命令“#include”，其功能是将有关的头文件包含到源文件中。在使用标准输入、输出库函数时要用到“stdio.h”（stdio 是 Standard Input & Output 的缩写）文件，因此，在源文件开头应有预编译命令“#include <stdio.h>”或“#include "stdio.h"”。printf 函数和 scanf 函数是两个标准输入/输出函数，前者称为格式输出函数，后者称为格式输入函数，它们最后的字母 f 是“格式（format）”的意思。printf 函数的功能是按指定的格式，把指定的数据输出（显示）到显示器的屏幕上。scanf 函数的功能则是将用户按指定格式输入到键盘缓冲区中的数据，存放到用户指定的内存单元中。使用 printf 函数和 scanf 函数可以分别实现整型、实型、字符型、字符串等各种类型数据的输出和输入。C

语言中，考虑到 printf 函数和 scanf 函数的使用比较频繁，允许在仅使用这两个函数时程序中不加“#include <stdio.h>”或“#include "stdio.h"”。

实际上，C 语言中，可实现数据的输入、输出的库函数还有很多。例如，仅在 stdio.h 中就有专门用来输入、输出字符数据的 getchar 函数和 putchar 函数，用来输入、输出字符串数据的 gets 函数和 puts 函数，等等。

本章先介绍 getchar、putchar、scanf、printf 等函数的语法规则和使用方法。

3.3.2　数据的输出函数

1．putchar函数

该函数是字符输出函数，其功能是在标准输出设备（显示器）上输出单个字符。其一般形式为：

putchar(整型表达式)

其中，“整型表达式”的值为要输出的字符的编码。通常，该表达式直接用字符型表达式代替。

例如：

putchar('A');　　输出大写字母 A。

putchar(x);　　输出整型（或字符型）变量 x 的值。

putchar('\n');　　输出换行符，实际就是实现换行操作，并不显示任何字符。

putchar('\101');　　输出大写字母 A。

putchar(65);　　输出大写字母 A。

putchar('a'–32);　　输出大写字母 A。

使用该函数前必须要用文件包含命令：#include<stdio.h>或#include "stdio.h"。

【例 3.1】　putchar 函数的使用方法。

源程序如下：

```
1   #include <stdio.h>
2   void main()
3   {
4       char a='B',b='o',c='k';
5       putchar(a);putchar(b);putchar(b);putchar(c);
6       putchar('\n');
7       putchar(b);putchar(c);
8       putchar('\n');
9   }
```

程序的运行结果为：

```
Book
ok
```

在这个程序中，第 1 行是预编译命令，将头文件“stdio.h”加入进来。第 2～9 行是主函数 main 函数的完整定义，第 2 行的 void 表示函数的类型为空类型。第 4 行定义了 3 个字符型变量 a、b、c，并分别进行了初始化。第 5 行是用 putchar 函数输出相应变量的值，即运行结果的第 1 行。第 6 行输出一个换行符，表示后面的输出将另起一行进行输出。第 7 行同样是用 putchar 函数输出相应变量的值，即运行结果的第 2 行。第 8 行的作用与第 6 行一样。

可见，用 putchar 函数输出一串字符时很不方便，需要写许多 putchar。通常，该函数只用于输出单一字符，若要输出一个字符串，可以使用 printf 函数（见 3.3.3 节）或 puts 函数（见第 8 章）。

2．printf函数

（1）函数的调用方法

printf 函数的一般形式为：

```
printf(格式控制,输出表列);
```

其中，“格式控制”部分是由双引号括起来的字符串，也称“转换控制字符串”，是不可缺少的部分，其功能是控制“输出表列”中的输出项的格式，即在显示器上输出的具体内容。“格式控制”字符串包括两种信息：格式说明和普通字符，如图 3.1 所示。“格式说明”（这里为“%d”）由引导符（百分号“%”）和格式字符（字母“d”）组成，而且总是由“%”开始的，其作用是控制待输出的数据按指定的格式输出。“普通字符”（这里指“a=”）是“格式控制”中除了“格式说明”外的那些字符，它们按原样输出。

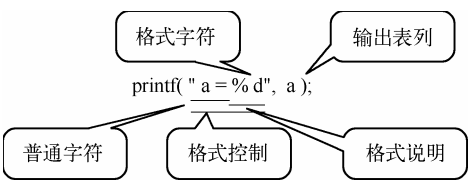


图 3.1　printf 函数的一般形式

“输出表列”由 0 到多个具体参数组成，参数可以是常量、变量、表达式等，参数的个数应该与“格式控制”字符串中“格式说明”的个数一致。

对于图 3.1 中的输出语句，如果假设整型变量 a 的值为 10（应在执行该语句前给变量 a 赋值），则其输出为：

```
a=10
```

这里的“a=”来自于“格式控制”字符串，是“普通字符”，按原样输出。“10”是输出表列中参数 a 的值，其输出格式由格式说明“%d”指定。

由于“格式控制”和“输出表列”实际上都是 printf 函数的参数，因此，printf 函数的一般形式还可以表示为：

```
printf(参数 1,参数 2,参数 3,……,参数 n);
```

其功能是将“参数 2”～“参数 *n*”按“参数 1”指定的格式输出。“参数 2”～“参数 *n*”的输出格式依次由“参数 1”中的“格式说明”决定。

例如：

```
printf("%d%f%c",a,b,c);
```

的执行结果为：a 按%d 的格式输出，b 按%f 的格式输出，c 按%c 的格式输出。

在 printf 函数的参数中，只有第一个参数“参数 1”是必要的，其他参数可以省略。当只有一个参数时，“参数 1”中不能包含“格式说明”字符，只能有“普通字符”。例如，

```
printf("This is a C program.\n");
```

是合法的，但

```
printf("a=%d\n");
```

是错误的（程序运行时将输出错误的结果，但不是语法错误）。

（2）printf 函数的输出格式控制

在“格式说明”中，对于不同类型的数据，要使用不同的格式字符。常见的格式字符如表 3.1 所示。

表 3.1　printf 函数的格式字符

格 式 字 符	说　　明
d	以带符号的十进制形式输出整数（正数不输出符号）
o	以八进制无符号形式输出整数（不输出前导符 0）
X 或 x	以十六进制无符号形式输出整数（不输出前导符 0x）
u	以无符号十进制形式输出整数
c	以字符形式输出，只输出一个字符
s	输出字符串
f	以小数形式输出单、双精度实型数
E 或 e	以标准指数形式输出单、双精度实型数
G 或 g	选用%f 或%e 格式中输出宽度较短的一种格式，不输出无意义的 0

在格式说明中，在引导符“%”和格式字符之间还可以插入附加格式说明字符，以便更精确地控制输出格式。常见的附加格式说明字符见表 3.2。

表 3.2　printf 函数的附加格式说明字符

字　　符	说　　明
字母 l	用于输出长整数，可加在格式字符 d、o、x、u 前面
字母 h	用于输出短整数，可加在格式字符 d、o、x、u 前面
m（*m* 是一个正整数）	*m* 是输出数据的最小宽度（字符个数）
n（*n* 是一个正整数）	对实型数，*n* 为小数位数；对字符串，*n* 为要截取的字符个数
–（减号）	输出的数字和字符向左对齐

格式控制部分中的格式字符的个数应该与输出表列中要输出的数据个数相等，而且位置也要一一对应。

下面简单介绍各个格式字符的用法。

① d、o、x、u 格式符

d、o、x、u 格式符都是用来输出整型数据的，其中 d 控制输出带符号的十进制整数，o 输出无符号的八进制整数，x 输出无符号的十六进制整数，而 u 输出无符号的十进制整数。

下面举例说明这 4 个格式符的用法。

%d、%o、%x、%u：按整型数据的实际长度输出，例如：

```
unsigned int u=4294967295;
printf("%d\n%u\n%o\n%x\n",u,u,u,u);     /* 每个数占一行 */
```

其执行结果为：

```
-1
4294967295
37777777777
ffffffff
```

%*m*d、%*m*o、%*m*x、%*m*u（*m* 是一个正整数）：*m* 为指定的输出宽度。如果实际数据的位数小于 *m*，则左边补足空格；如果大于 *m*，则按实际位数输出，*m* 将不起作用。例如，

```
unsigned int u=4294967295;
printf("%10d\n%10u\n %10o\n%10x\n",u,u,u,u);
```

其执行结果为（“_”代表空格）：

```
       -1
4294967295
37777777777（实际有 11 位数字）
  ffffffff
```

%–*m*d、%–*m*o、%–*m*x、%–*m*u（*m* 是一个正整数）：与%*m*d、%*m*o、%*m*x、%*m*u 基本相同，只是输出的数据向左端对齐，右端补空格，例如：

```
unsigned int u=4294967295;
printf("%-10d<\n%-10u<\n%-10o<\n%-10x<\n",u,u,u,u);  /* 用<表示边界 */
```

其执行结果为：

```
-1      <
4294967295<
37777777777<
ffffffff  <
```

%ld、%lo、%lx、%lu：输出长整型数据。

%hd、%ho、%hx、%hu：输出短整型数据。例如：

```
unsigned short u=65535;
printf("%hd,%hu,%o,%hx\n",u,u,u,u);
```

其执行结果为：

```
-1,65535,177777,ffff
```

通常，有符号整数可以用“%d”、“%hd”等格式符输出，无符号整数可用“%u”、“%o”、“%x”等格式符输出；输出长（短）整型数据时应使用带附加格式说明字符“l”（“h”）的格式说明，如“%ld”。

如果将有符号整数用无符号格式符控制输出，或者反过来，将无符号整数用有符号的格式符控制输出，就需要考虑数据的转化规律了。实际上，不论有符号整数还是无符号整数，在计算机内存中都是用补码表示的。假设在计算机内存中一个整数占 4 个字节（32 位长度），“1111 1111 1111 1111 1111 1111 1111 1111”是内存中的一个二进制数。如果把这个数看做是无符号的整数，它就是十进制数 4294967295（即 $2^{32}-1$）；如果看做是有符号整数，它就是十进制数–1。

② f、e、g 格式符

f、e、g 格式符都是用来输出实型数据的。其中，f 以小数形式输出，e 以指数形式输出，g 则根据输出数据的长度自动选择用小数或指数格式输出。

现举例说明它们的用法。

%f：按带小数点的小数形式输出，整数部分全部输出，并输出 6 位小数。

例如：

```
float f=456.123;
printf("%f\n",f);
```

输出结果为：

```
456.122986
```

注意，由于内存中的实数存在误差，只有 5～6 位有效数字，所以达不到 6 位小数精度。

%e：按标准化指数形式输出（小数部分的小数点前必有且只有 1 位非 0 数字），输出宽度共占 13 列，其中，小数点占 1 位，小数点前的非 0 数字占 1 位，小数点后的小数部分由系统自动指定为 6 位，指数部分占 5 位（字母“e”占 1 位，指数符号占 1 位，指数占 3 位）。

例如：

```
float f=456.123;
printf("%e\n",f);
```

输出结果为：

```
4.561230e+002
```

%g：根据输出数据的长度（位数）自动选择按小数或指数格式输出。

例如：

```
float  f1=123.456789, f2=123456789;
printf("%f,%e,%g\n",f1,f1,f1);
printf("%f,%e,%g\n",f2,f2,f2);
```

输出结果为：

```
123.456787,1.234568e+002,123.457
123456792.000000,1.234568e+008,1.23457e+008
```

%*m.n*f（*m*、*n* 为正整数）：指定输出的数据共占 *m* 列，其中有 *n* 位小数。如果数值长度小于 *m*，则左端补空格；如果超过 *m*，则忽略 *m* 的限制。

例如：

```
float  f=123.456789;
printf("%10.3f,%5.3f\n",f,f);
```

输出结果为：

```
_ _ _123.457,123.457
```

[①]

如果省略了小数位数“.*n*”（同时省略小数点“.”），则输出 6 位小数（即 *n* 的默认值为 6）。

例如：

```
float  f=123.456789;
printf("%10f,%5f\n",f,f);
```

输出结果为：

```
123.456787,123.456787
```

如果只是省略了 *n*，而保留了小数点“.”，如“%5.f”，则不输出小数部分，相当于 *n* 为 0。

%–*m.n*f：与%*m.n*f 基本相同，只是输出的数据向左对齐，在右端补空格。

%*m.n*e：指定输出的数据共占 *m* 列，其中数据的小数部分（又称尾数）有 *n* 位小数。如果数值长度小于 *m*，则左端补空格。

③ c 格式符

c 格式符用来输出一个字符，有两种主要用法：%c 和%*m*c。其中，*m* 指定输出宽度（通常 *m*>1），输出的字符输出在指定的最后一列，即第 *m* 列，前面补 *m*–1 个空格。

例如：

```
char c='c';
printf("%c%3c\n",c,c);
```

输出结果为：

```
c_ _c
```

在字符型数据的取值范围内，整型数据可以与字符型数据通用。因此，一个整数，只要它的值在字符型数据范围内，就可以用字符形式输出。反之，一个字符型数据也可以用整型形式输出。例如：

```
char c='A';
```

① “_”代表空格，下同。

```
int i=65;
printf("%c,%d\n",c,c);
printf("%c,%d\n",i,i);
```

输出结果为：

```
A,65
A,65
```

④ s 格式符

s 格式符用来输出一个字符串，主要有以下几种用法。

%s：直接输出全部字符。例如：

```
printf("%s","china");
```

输出结果为：

```
china
```

%*m*s（*m* 是一个正整数）：输出的字符串占 *m* 列。如果字符串本身的长度大于 *m*，则忽略 *m* 的限制，将字符串全部输出；如果小于 *m*，则左边补空格。

%*m*.*n*s（*m*、*n* 是正整数）：从字符串中截取左边的 *n* 个字符（不足则忽略 *n* 的限制），输出到 *m* 列上。若不足 *m* 列，在左边补空格。

%–*m*.*n*s（*m*、*n* 是正整数）：基本同%*m*.*n*s，只是输出的字符串是左对齐的。

例如：

```
printf("%3s\n%7.2s\n%.4s\n%-5.3s\n","china","china","china","china");
```

输出结果为：

```
china
_ _ _ _ _ch
chin
chi_ _
```

需要注意，第 1 行突破了 3 的限制，第 2 行前面有 5 个空格，第 3 行占 4 列，最后一行占 5 列，右边有 2 个空格。

（3）使用 printf 函数应注意的问题

① 格式字符要区分大小写。

② 不同的运行环境在实现格式输出时，输出结果可能会有一些小的差别。例如，用%e 格式符输出实数时，有些系统输出的指数部分为 4 位（如 e+02）而不是 5 位（如 e+002）；前面的数字部分是 5 位小数而不是 6 位，等等。

③ 可以在“格式控制”字符串中使用“转义字符”，如“\t”、“\n”、“\b”、“\r”、“\f”、“\345”、“\x1A”等，它们将被看做“普通字符”。

④ 如果想输出字符“%”，则应该在“格式控制”字符串中用连续的两个%表示，例如：

```
printf("%f%%",1.0/3);
```

输出结果为：

```
0.333333%
```

（4）printf 函数应用举例

【例 3.2】　用 printf 函数输出数据。

程序如下：

```
1  #include <stdio.h>
2  #define PI 3.141592654
3  void main()
4  {
5     printf("不同精度的圆周率\n");
6     printf("%4s\t%-8s\n","精度","圆周率");  /* 一个汉字占 2 位 */
7     printf("%4d\t%-8.0f\n",1,PI);
8     printf("%4d\t%-8.1f\n",2,PI);
9     printf("%4d\t%-8.2f\n",3,PI);
10    printf("%4d\t%-8.3f\n",4,PI);
11    printf("%4d\t%-8.4f\n",5,PI);
12    printf("%4d\t%-8.5f\n",6,PI);
13    printf("%4d\t%-8.6f\n",7,PI);
14    printf("%4d\t%-8.7f\n",8,PI);
15 }
```

程序的运行结果如下：

```
不同精度的圆周率
精度    圆周率
   1    3
   2    3.1
   3    3.14
   4    3.142
   5    3.1416
   6    3.14159
   7    3.141593
   8    3.1415927
```

3.3.3　数据的输入函数

1．getchar函数

getchar 函数的功能是从键盘上输入一个字符，其一般形式为：

```
getchar();
```

通常把输入的字符赋予一个字符变量，构成赋值语句，例如：

```
char c;
c=getchar();
```

使用该函数前必须要用文件包含命令：#include<stdio.h>或　#include "stdio.h"。

【例 3.3】　getchar 函数的用法。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {
4       char c;
5       printf("input a character\n");
6       c=getchar();
7       putchar(c);
8   }
```

程序的运行结果如下：

```
input a character
a ↙
a
```

程序的第 5 行是输入提示，用来在显示器上显示一条信息，告诉使用程序的人该如何做。建议读者在编程时注意使用这一方法。程序的第 6 行利用 getchar 函数从键盘上输入一个字符，并赋给字符变量 c。程序运行时，显示一个光标，用户输入的字符（这里是字母“a”）将显示在光标位置（参见运行结果的第 2 行），当用户按回车键时，获得输入的字符，结束 getchar 函数的调用，并把结果赋给变量 c。程序的第 7 行利用 putchar 函数将字符变量 c 的值显示在屏幕上。

应该注意的是，getchar 函数只能接受单个字符，输入多于一个字符时，只接收第一个字符。这是因为，getchar 函数是从键盘缓冲区中读取字符数据的，用户输入的字符暂时存在键盘缓冲区中，形成一个序列。每调用一次 getchar 函数，将顺序读取一个字符，其他字符还将保留在缓冲区中。

【例 3.4】　用 getchar 函数读取多个字符。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {
4       char c;
5       printf("input some characters\n");
6       c=getchar();
7       putchar(c);
8       c=getchar();
9       putchar(c);
10      c=getchar();
```

```
11      putchar(c);
12  }
```

程序的运行结果可能是：

```
input some characters
abcd ↙                （这里连续输入 4 个字符）
abc                   （仅显示 3 个字符，且连续输出）
```

程序中（第 6～11 行），getchar 函数和 putchar 函数是交替使用的，但上面的运行结果显示的输入和输出却是分别进行的，为什么？请读者自己体会个中的缘由。读者不妨尝试一下其他不同的输入方法，观察结果有什么不同。

需要注意的是，使用 getchar 函数可以输入任意字符，包括空格、Tab 键、回车等控制字符。

2．scanf函数

（1）scanf 函数的调用方法

scanf 函数的一般形式为：

```
scanf(格式控制,地址表列);
```

其中，“格式控制”的含义同 printf 函数，“地址表列”是由若干个内存地址组成的表列，可以是变量的地址，也可以是字符串的首地址，等等。

对于变量 v，其地址可以用&v 表示，其中“&”是地址运算符（详见第 7 章）。

这里需要读者特别注意，scanf 函数的参数除了第一个参数是“格式控制”字符串外，其他都是变量的地址，不是变量名称，这不同于其他程序设计语言。对于一般变量，出现在 scanf 函数参数中时，通常需要加地址运算符“&”。但有些参数本身就是地址，也就不需要加地址运算符了，比如用指针变量作为参数。

例如：

```
int a,b,c;
scanf("%d%d%d",&a,&b,&c); /* &a、&b、&c 分别代表变量 a、b、c 在内存中的地址*/
printf("%d%d%d\n",a,b,c);
```

运行时输入：

```
1  2  3 ↙
```

则输出结果为：

```
123
```

程序中 scanf 函数的作用是，将用户输入地数据（这里是 1、2、3 三个整数，存放在键盘缓冲区中）按顺序存入变量 a、b、c 在内存中的存储单元（用地址指定）中。格式控制“%d%d%d”表示按十进制整数形式输入 3 个整数。输入这 3 个数据时，可以用一个或多个空格将它们隔开，也可以用回车键和跳格键（即 Tab）作为分隔符。

对于这段程序，下面的输入方式都是合法的：

① 1_2____3 ↙
② 1 ↙
　2 ↙
　3 ↙
③ 1 → 2__3 ↙[①]

再看一个输入实型和字符型数据的例子：

```
float x,y,z;
char c1,c2;
scanf("%f%f%f",&x,&y,&z);
scanf("%c%c",&c1,&c2);
```

可以按如下形式输入数据：

```
1.0  2  .3 ↙          （小数的输入比较自由）
ab ↙
```

其中，“%f”用来输入实型数据，输入多个实数时同样可以使用空格、回车键和 Tab 键作为分隔符。“%c”用来输入一个字符型数据，输入多个字符时不需要任何分隔符，连续输入即可。

（2）scanf 函数的输入格式控制

与 printf 函数不同，scanf 函数的格式控制字符串中，所有项目都是要求用户在运行程序时输入的，包括与格式说明对应的数据和普通字符。

例如，对于如下程序段（假设 x、y 都是整型的）：

```
printf("input data here:\n");
scanf("x=%d",&x);
scanf("y=%d",&y);
printf("x=%d,y=%d\n",x,y);
```

运行时的输入、输出是：

```
input data here:
x=123 ↙
y=456 ↙
x=123,y=456
```

如果不输入“x=”、“y=”，程序就会出错。因此，通常不在 scanf 函数的“格式控制”中使用“普通字符”，以避免输入困难。

而与 printf 函数中的格式说明类似，scanf 函数的格式说明也以“%”开始，以一个格式字符结束，中间可以插入附加格式说明字符。表 3.3 列出了 scanf 函数所用到的格式字符，表 3.4 列出了 scanf 函数可以使用的附加格式说明字符。

① “→”代表 Tab 键，下同。

表 3.3　scanf 函数的格式字符

格式字符	说　明	格式字符	说　明
d	输入的是十进制整数	c	输入单个字符
o	输入的是八进制整数	f 或 e	输入的是一个实数；可以用小数形式，也可以用指数形式；e 与 f 可互换使用
x	输入的是十六进制整数		
s	将输入字符串送到一个字符数组中。输入时，以非空白字符开始，以第一个空白字符（空格、Tab 键、回车）结束		

表 3.4　scanf 函数的附加格式说明字符

字　符	说　明
字母 l	用于输入长整数（对应%ld、%lo、%lx 等），以及双精度实数（对应%lf 或%le）
字母 h	用于输入短整数（对应%hd、%ho、%hx 等）
m（*m* 为一正整数）	指定输入数据所占宽度（列数），常省略
*	表示本输入项在读入后不赋给相应的变量，也就是不与“地址列表”中的某输入项对应

说明：

① C 语言中，要连续输入 2 个以上的数据，一般使用空白（即空格、Tab 键或回车键）作为分隔符。但有的人习惯使用逗号“,”分隔数据，为此，可以在 scanf 函数的“格式控制”字符串中，用逗号分隔对应的格式说明，例如：

```
scanf("%d,%d",&a,&b);
```

输入时，应采用如下形式：

```
1,2 ↙
```

注意，1 后面是逗号，它与 scanf 函数中格式控制中的逗号相对应。如果输入时不用逗号而用空格或其他字符，将不能输入正确的数据。

反过来，如果“格式控制”字符串中没有出现逗号分隔符，输入时用逗号分隔也是错误的。因此，一个好的习惯是，在程序中用提示信息提醒用户该如何输入数据。例如：

```
printff("输入用逗号分开的两个整数\n");
scanf("%d,%d",&a,&b);
```

② 用“%c”输入字符时，空格、Tab、回车及其他控制字符都可以作为有效字符输入。因此，对于：

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

如果输入：

```
a  b  c ↙
```

（3 个字母间各有一个空格）

则字符'a'送给变量 c1，' '（空格）送给变量 c2，'b'送给变量 c3。

由于 c 格式符的特殊性（其他格式符都把空白看做数据的分隔符，而不会读取空白），使用 scanf 时，最好不要将 c 格式符与其他格式符混合使用，以免造成输入错误。

③ 在输入数据时，遇到以下情况则认为该数据结束。

- 遇到空格、回车、Tab 键。
- 以满足宽度限制，比如对于“%5d”，只取 5 位整数。
- 遇到非法输入，比如“a”不是合法的十进制数，但是合法的十六进制数。

例如，对语句：

```
scanf("%d%c%f",&x,&y,&z);
```

如果输入的是：

```
56t8o.34↙          （结果 x 为 56，y 为't'，z 为 8）
```

则对应第一个格式%d 输入 56 之后遇到字母 t，便认为第一个数据到此结束，把 56 送给变量 x。字符 't' 送给变量 y，由于%c 只要求输入一个字符，因此“t”后面不需要空格，后面的数值应送给变量 z。如果由于疏忽把本来应为 80.34 的数错输入成 8o.34，则由于 8 后面出现字符 o，就认为此数值结束，将 8 送给 z，多余部分自动舍弃。

④ 标准 C 在 scanf 函数中不使用%u 说明符，对 unsigned 型数据，以%d 或%o、%x 格式输入。

⑤ 输入数据时不能规定精度，形如"%8.2f"的格式说明是错误的。

⑥ 可以指定输入数据所占的列数，系统自动按此列数截取所需数据。例如：

```
scanf("%3d%3d",&a,&b);
```

如果输入：

```
12345↙
```

则系统自动将 123 赋给 a，45 赋给 b。

⑦“%”后的“*”附加说明符，用来表示跳过相应的数据。在利用现成的一批数据时，如果不需要其中的某些数据，则可以利用此种方法。例如：

```
scanf("%2d%*3d%2d",&a,&b);
```

如果输入以下信息：

```
12 34567↙
```

则将 12 赋给变量 a，67 赋给变量 b。第二个数据 345 被跳过不赋给任何变量。

【例 3.5】　求任意两个整数的和。

分析：如果问题中出现“任意”等字眼，通常意味着，程序所处理的数据是在运行程序时从键盘输入的。输入整数的方法是调用 scanf 函数。

程序的算法过程可用图 3.2 表示。

图 3.2　求任意两个整数的和

源程序如下：

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

```
3   {   int a,b,sum;
4       printf("输入用逗号分隔的两个整数：");          /* 输入提示 */
5       scanf("%d,%d",&a,&b);                          /* 格式输入语句 */
6       sum=a+b;                                       /* 赋值语句 */
7       printf("%d 和%d 的和是%d",a,b,sum);            /* 格式输出语句 */
8   }
```

程序的运行结果如下：

```
输入用逗号分隔的两个整数：3, 5↙
3 和 5 的和是 8
```

3.4　顺序程序设计

到目前为止，已经学习了数据的输入、处理和输出的一般方法，也就具备了编写简单程序的基本条件。下面结合实例，学习简单的程序设计。

所谓简单的程序设计，就是在进行程序设计时，只使用表达式语句和函数调用语句，不涉及复合语句和控制语句等。这样的程序称为“顺序程序”，这是结构最为简单、也是最基本的一类程序，程序中的语句按照排列的先后顺序依次执行。

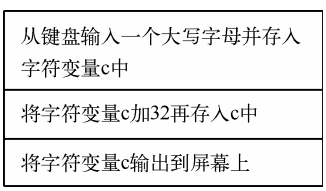


图 3.3　将大写字母转化为小写字母

【例 3.6】　输入一个大写字母，转化为小写字母输出。

分析：单字母的输入、输出可以用 getchar、putchar 函数，也可以用 scanf、printf 函数。

在字符编码集中，大写字母和小写字母的编码相差 32，因此，小写字母=大写字母+32。

算法用 N-S 图表示，如图 3.3 所示。

程序如下：

```
1   #include <stdio.h>
2   void main()
3   {   char c;
4       printf("输入一个大写字母：");          /* 输入提示 */
5       c=getchar();
6       c=c+32;                                /* 大写字母转化为小写字母 */
7       printf("对应的小写字母为：");
8       putchar(c);
9   }
```

试试看，如果输入的不是大写字母，结果是不是很可笑呢？能不能对输入数据进行限制：如果输入的不是大写字母，就不执行转换和输出呢？答案当然是肯定的，不过需要引入选择结构，详见第 4 章。

【例 3.7】　输入 a、b、c 的值，设 $b^2-4ac>0$ 且 $a\neq 0$，输出一元二次方程 $ax^2+bx+c=0$ 的实根。

分析：对于一元二次方程 $ax^2+bx+c=0$，如果 $b^2-4ac>0$ 且 $a\neq 0$，则方程有两个实根：

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

因此，只要输入的系数满足要求，就可以求得该一元二次方程的两个不同的实根。当然，在求根公式中，需要进行求平方根的运算，可以使用数学函数库中的平方根函数 sqrt 来完成。

N-S 图如图 3.4 所示。

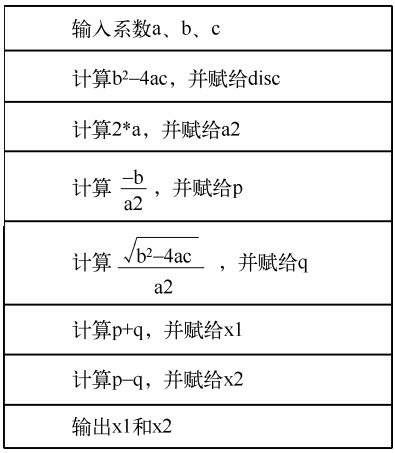


图 3.4　求一元二次方程的根

程序如下：

```
1   #include "stdio.h"
2   #include "math.h"
3   void main()
4   {   float a,b,c,disc,x1,x2,p,q,a2;
5       printf("Please Input a、b、c: ");            /* 输入提示 */
6       scanf("%f%f%f",&a,&b,&c);                    /*输入系数a、b、c*/
7       disc=b*b-4*a*c;
8       a2=2*a;
9       p=-b/a2;
10      q=sqrt(disc)/a2;
11      x1=p+q;
12      x2=p-q;
13      printf("Roots of equation%.1f*x^2+%.1f*x+%.1f are: \n",a,b,c);
14      printf("x1=%.3f\nx2=%.3f\n",x1,x2);
15  }
```

运行结果为：

```
Please Input a、b、c: 1  5  4↙
Roots of equation 1.0*x^2+ 5.0*x+4.0 are:
x1=-1.000
x2=-4.000
```

【例 3.8】　已知三角形的两条边及其夹角，求其第三边长度。

分析：根据题目给定的条件，数学上利用余弦定理求第三边的长度。方法是：若给定的两条边是 a 和 b，夹角为α，则第三边的长度 c 为：

$$c = \sqrt{a^2 + b^2 - 2ab\cos(\alpha)}$$

程序中，α用标识符 alpha 代表。

N-S 图如图 3.5 所示。

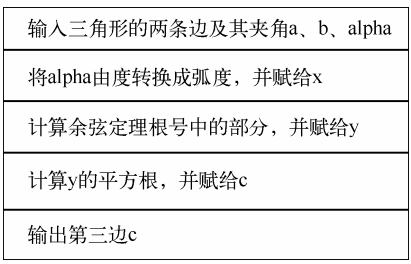


图 3.5　利用三角形的两条边及其夹角求其第三边长度

程序如下：

```
1   #define PI 3.14                          /* 定义符号常量 PI */
2   #include "stdio.h"
3   #include "math.h"
4   void main()
5   {   float a,b,c,x,y;
6       float alpha;
7       printf("Input a、b、alpha: ");        /* 输入提示 */
8       scanf("%f,%f,%f",&a,&b,&alpha);       /* 输入三角形的两条边及夹角 */
9       x=alpha*PI/180;                       /* 将 alpha 由度转换成弧度 */
10      y=a*a+b*b-2*a*b*cos(x);               /* 利用余弦定理求第三边的长度 */
11      c=sqrt(y);
12      printf("c=%7.3f\n",c);
13  }
```

运行情况为：

```
Input a、b、alpha: 3,4,90↙        （注：按程序要求，采用逗号分隔数据）
c=4.998
```

习　题　3

3.1　C 语言的语句有哪几类?

3.2　选择题

（1）putchar 函数可以向屏幕输出一个（　　）。

A. 整型变量表达式　　B. 实型变量值

C. 字符串　　D. 字符或字符型变量值

（2）printf 函数中用到格式符“%5s”，其中数字 5 表示输出的字符串占用 5 列。如果字符串长度大于 5，则输出方式为（　　）；如果字符串长度小于 5，则输出方式为（　　）。

A. 从左起输出该字符串，右补空格　　B. 按原字符长从左向右全部输出

C. 右对齐输出该字符串，左补空格　　D. 输出错误信息

（3）阅读以下程序，当输入数据的形式为：25, 13, 10<CR>（注：<CR>表示回车），则正确的输出结果为（　　）。

```
void main()
{   int x,y,z;
    scanf("%d%d%d",&x,&y,&z);
    printf("x+y+z=%d\n",x+y+z);
}
```

A. x+y+z=48　　B. x+y+z=35　　C. x+z=35　　D. 不确定值

（4）根据下面的程序及数据的输入和输出形式，程序中输入语句的正确形式应该为（　　）。

```
void main()
{   char ch1,ch2,ch3;
    /* 此处加入输入语句 */
    printf("%c%c%c",ch1,ch2,ch3);          /* “%c”间都没有空格 */
}
```

输入形式：A _ B_ C ↙

输出形式：A _ B

A. scanf("%c%c%c", &ch1, &ch2, &ch3);　　（“%c”间都没有空格）

B. scanf("%c,%c,%c", &ch1, &ch2, &ch3);

C. scanf("%c %c %c", &ch1, &ch2, &ch3);　　（“%c”间都有一个空格）

D. scanf("%c%c", &ch1, &ch2, &ch3);　　（两个“%c”间没有空格）

（5）已知 ch 是字符型变量，下面不正确的赋值语句是（　　）。

A. ch='a+b';　　B. ch='\0';　　C. ch='0'+'9';　　D. ch=40+9;

3.3　填空题

（1）以下程序的输出结果是_____。

```
void main()
{   short  i;
    i=-4;
    printf("i:dec=%hd,oct=%ho,hex=%hx,unsigned=%hu\n",i,i,i,i);
}
```

（2）以下程序的输出结果是_____。

```
void main()
{   char c='x';
```

```
    printf("c:dec=%d,oct=%o,hex=%x,ASCII=%c\n",c,c,c,c);
}
```

（3）以下程序的输出结果是_____。

```
void main()
{   int x=1,y=2;
    printf("x=%d  y=%d  sum=%d\n",x,y,x+y);
    printf("10 Squared is:%d\n",10*10);
}
```

（4）假设变量 a 和 b 均为整型，以下语句可以不借助任何变量把 a、b 中的值进行交换。请填空。

```
a=（    ）; b=（    ）; a=（    ）;
```

（5）若 x 为 int 型变量，则执行以下语句后的 x 值为_____。

```
x=7; x+=x-=x+x;
```

（6）有一输入语句 scanf("%d", k);，则不能使 float 类型变量 k 得到正确数值的原因是_____和_____。

3.4　怎样区分赋值表达式和赋值语句？什么时候使用赋值表达式？什么时候使用赋值语句？

3.5　已知矩形的长和宽分别是 300 和 20，请编写计算其周长和面积的程序。

3.6　输入一个圆的半径，分别求圆的周长和面积。

3.7　输入三角形的三条边的边长，请编写求其面积的程序。

3.8　鸡兔同笼问题：已知笼中有头 *h* 个，有脚 *f* 条，问笼中鸡、兔各有多少只？试编程。

3.9　输入一个华氏温度，要求计算出摄氏温度。计算公式为：

$$C=\frac{5}{9}(F-32)$$

要求：输入要有提示，输出要有文字说明，并取两位小数。

3.10　从键盘输入任意一个 4 位数，编程分隔出该 4 位数的各位数字，计算它们的和并输出到显示器上。

3.11　输入任意一个字符，输出其对应的 ASCII 码。

3.12　请写出下面程序的输出结果：

```
void main()
{   int a=5,b=7,c=-1;
    float x=67.6584,y=-879.123;
    long n=123456789;
    unsigned u=-1;
    printf("%d%d\n",a,b);
    printf("%3d%3d\n",a,b);
    printf("%f,%f\n",x,y);
    printf("%-10f,%-10f\n",x,y);
```

```
    printf("%8.2f,%8.2f,%.4f,%.4f,%3f,%3f\n",x,y,x,y,x,y);
    printf("%e,%10.2e\n",x,y);
    printf("%c,%d,%o,%x\n",c,c,c,c);
    printf("%ld,%lo,%lx\n",n,n,n);
    printf("%u,%o,%x,%d\n",u,u,u,u);
    printf("%s,%5.3s\n","COMPUTER","COMPUTER");
}
```

3.13　已知三角形的三个顶点坐标为（1.5, 2）、（3, 1）、（2.1, 4），求该三角形的重心坐标和各边长度。提示：如图 3.6 所示，已知三角形的三个顶点，则该三角形的重心 G 点的坐标为 $x_G=(x_1+x_2+x_3)/3$，$y_G=(y_1+y_2+y_3)/3$。

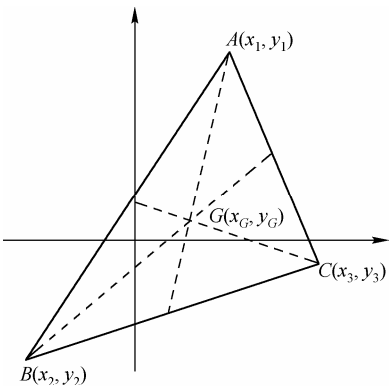


图 3.6　三角形的重心

3.14　已知圆半径和圆柱高，求圆球表面积、圆球体积和圆柱体积。要求：输入要有提示，输出要有文字说明，并取两位小数。

第 4 章　选择程序设计

学习目标

通过本章学习，应该掌握：

- 关系运算与逻辑运算的规律
- if 语句的语法规则、执行过程和使用方法
- if 语句的嵌套
- switch 语句的基本概念和用法
- 选择结构的程序设计

在第 3 章中我们学习了顺序结构的程序设计。顺序结构是一种由上而下逐条语句依次执行的结构，其特点是程序的执行流程固定不变。然而，在进行实际程序设计时，经常需要根据不同的条件改变程序执行的流程。这就需要另一种程序控制结构——选择结构（也称分支结构、判断结构），其特点是可以根据所给定的条件是否成立来决定程序的执行流程。

设计选择结构程序，首先需要考虑两个方面的问题：一是如何来表示条件，也就是如何作判断；二是用什么语句实现选择结构。

在 C 语言中，一般用关系表达式或逻辑表达式表示条件，用 if 语句或 switch 语句来实现选择结构。

4.1　关系运算符和关系表达式

关系运算就是比较运算，即将两个数据进行大小比较，以判定两个数据是否符合给定的关系。这不同于数学上的不等式：例如，数学上“a>b”表示 a 的值比 b 大，而在 C 语言中，“a>b”表示比较 a 和 b 的大小，若 a 的值比 b 大，则结果为“真”（即不等式成立），反之，若 a 不比 b 大，则结果为“假”（即不等式不成立）。也就是说，不论 a 是否比 b 大，都可以进行关系运算。

关系运算的结果为逻辑值“真”和“假”，也可以说成是“对”与“错”、“是”与“否”等。C 语言中，逻辑值“假”用数值 0 表示，而逻辑值“真”用 1 表示。C 语言中并没有专门表示逻辑值的数据类型，实际上，任何类型的数值都可以理解为逻辑值，比如，x 位整型变量，若 x 的值为 0，则可以把 x 的值看做逻辑“假”，反之，若 x 的值不为 0，则可以把 x 的值看做逻辑“真”。

4.1.1　关系运算符

关系运算符用于比较两个数据的大小，共有如下 6 种运算符。

① <　　：小于。

② <=　：小于等于，对应的数学符号是“≤”。

③ >　：大于。

④ >=　：大于等于，对应的数学符号是“≥”。

⑤ ==　：等于，由连续两个等号组成。

⑥ !=　：不等于，对应的数学符号是“≠”。

要注意它们的优先级。前 4 种关系运算符的优先级相同，后两种也相同，前 4 种高于后两种。所有的关系运算符的优先级都低于算术运算符，而高于赋值运算符和逗号运算符。

例如：

① c>a+b　　等价于　c>(a+b)

② a==b<c　　等价于　a==(b<c)

③ a=b<=c　　等价于　a=(b<=c)

使用关系运算符，需要注意以下几点。

① 关系运算符是双目运算符，其两边的操作数可以是整型、实型、字符型、指针型及枚举型等。如果参与运算的两个数类型不一致，则先要进行类型转换，再进行关系运算。

例如，'x'>'c' 是两个字符型常量进行比较，实际上比较的是这两个字符型常量的 ASCII 码值。又如，'a'>70 的运算顺序是先将字符 'a' 转换成其 ASCII 码值 97，再与整型常量 70 进行比较。

② 关系运算符具有左结合性，可以在一个关系表达式中连续使用。例如，关系表达式–1<=x<=1 的运算顺序是先判断–1<=x 是否成立，结果为 0 或 1，然后再用 0 或 1 去判是否小于等于 1，结果永远是 1。但是，我们知道关系运算的结果是个逻辑值，或为“真”或为“假”，让它去跟一个整型、实型数据去比较大小，并没有实际意义，所以一般不要连续使用关系运算符组成表达式，有的系统甚至把连续出现的关系运算符看做“错误”。

③ 必须注意区别“=”和“==”。前者是赋值运算符，后者是关系运算符中的等于运算符。例如，对于下面的程序：

```
1   #include<stdio.h>
2   void  main()
3   {   int a=5,b=3,c;
4       printf("c=%d\n",c==a);
5       printf("a=b?%d\n",a=b);
6   }
```

程序的运行结果为：

```
c=0
a=b?3
```

（1 表示二者相等，0 表示不等）

结果显然是错误的。这是因为，第 4 行中，错把“c=a”写成了“c==a”，多写了一个“=”，赋值运算变成了关系运算（程序中变量 c 未做初始化，编译时显示有“警告”错误）；而在第 5 行中，错把“a==b”写成了“a=b”，遗漏了一个“=”，关系运算变成了赋值运算。

4.1.2　关系表达式

用关系运算符将两个表达式（可以是任意表达式，通常为数值表达式）连接起来的式子，称为关系表达式。例如，下面的关系表达式都是合法的：

```
a>b,a+b>c-d,(a=3)<=(b=5),'a'>='b',(a>b)==(b>c)
```

关系表达式的运算结果是逻辑值“真”或“假”。当关系表达式成立时为“真”，不成立时为“假”。

对于下面的表达式，若假设 a=1，b=3，c=4，根据运算符的运算规律不难计算出结果：

① a > b　　结果为“假”。

② a+b = = c　结果为“真”。

③ x = a > c　结果为 0（这是一个赋值表达式）。

④ (x=a+b)>c 结果为“假”。

⑤ a++ > b–c 结果为“真”。

⑥ b < c > a　结果为“假”。

4.2　逻辑运算符和逻辑表达式

关系表达式只能描述单一条件，例如“x>=0”。如果需要描述“x>=0”同时“x<10”，就必须得借助于逻辑表达式了。这是因为表达式“0 <= x <10”并不能表示上述含义。假设 x 为 4，表达式“0 <= x<10”的计算结果为“真”，似乎是正确的，但是，任取一个不在此范围内的数试一试，就会发现该表达式的值总为“真”。

4.2.1　逻辑运算符

逻辑运算符用于对操作数进行逻辑操作，参与逻辑运算的值是逻辑值。C 语言共有如下 3 个逻辑运算符。

① &&　：逻辑“与”。

② ‖　：逻辑“或”。

③ !　：逻辑“非”。

其中，“&&”和“‖”是双目运算符，要求有两个运算量（操作数）；“!”是单目运算符，只要求有一个运算量，向右结合。

逻辑运算符的功能如下。

① a&&b　：只有当 a、b 都为真时，其结果才为真；否则，为假。

② a‖b　：只有当 a、b 都为假时，其结果才为假；否则，为真。

③ !a　：a 为真（非 0），则!a 为假；a 为假，则!a 为真。

逻辑运算的“真值表”如表 4.1 所示。

表 4.1　逻辑运算真值表

a	b	!a	!b	a&&b	a‖b
真	真	假	假	真	真
真	假	假	真	假	真
假	真	真	假	假	真
假	假	真	真	假	假

在逻辑运算符中，逻辑非“!”的优先级最高，逻辑与“&&”次之，逻辑或“||”最低。而且，逻辑非“!”的优先级高于算术运算符，逻辑与“&&”、逻辑或“||”低于关系运算符但高于赋值运算符。逻辑运算符的优先级如表 4.2 所示。

表 4.2　运算符的优先级

运　算　符	优　先　级
!（非）	高
算术运算符	
关系运算符	
&&（与）	
‖（或）	
赋值运算符	低

在逻辑运算符中，“!”（非）具有右结合性，“&&”（与）和“||”（或）具有左结合性。例如：

① a>b&&a–b　　等价于　(a>b)&&(a–b)

② c=0||a!=b　　等价于　c=(0||(a!=b))

③ !a&&a–b　　等价于　(!a)&&(a–b)

④ !a&&a–b&&a>b　　等价于　((!a)&&(a–b))&&(a>b)

4.2.2　逻辑表达式

用逻辑运算符将表达式连接起来的式子叫逻辑表达式，其中的表达式可以是逻辑表达式、关系表达式、算术表达式、赋值表达式等。

逻辑表达式的结果只能是“真”或“假”。逻辑表达式成立，即为“真”；逻辑表达式不成立，即为“假”。需要注意，C 语言规定在判断一个运算量是“真”或“假”时，以“0”代表假，以“非 0”代表真。也就是说，只要该运算量的值不为 0 就可以看做“真”。

例如，若 a=10，则!a 的值为“假”（用 0 表示）；若 a=10，b=15，则 a&&b 的值为“真”（用 1 表示），a||b 的值为“真”，!a&&b 的值为“假”。

进行逻辑运算时，应注意：

① 逻辑运算符的操作数可以是任意类型，只要能表示为 0 或非 0 值即可。

例如：

'a'&&'\0'　　结果为“假”

10||'a'+'b'　　结果为“真”

!10　　结果为“假”

② 逻辑运算符&&和||运算时强调由左到右按序进行，但求解时不一定对逻辑运算符两边的表达式都求值，如果前面的运算已经可以确定整个表达式的计算结果，则后面的计算就不是必要的了，称为运算截断。

例如，对于&&运算：

```
表达式 1 && 表达式 2
```

当表达式 1 的值为“假”时，该逻辑表达式的结果肯定为“假”，表达式 2 的计算就被忽略了。例如，假设 n1、n2、n3、n4、x、y 的值分别为 1、2、3、4、1、1，则求解表达式(x=n1>n2)&&(y=n3>n4)后，x 的值变为 0，而 y 的值不变，仍等于 1。

同理，对于||运算：

```
表达式 1 || 表达式 2
```

当表达式 1 的值为“真”时，该逻辑表达式的结果肯定为“真”，因此表达式 2 的计算也不再需要了。

③ 在一个逻辑表达式中，如果有多个运算符，则要按照它们的优先级进行运算。

例如：

```
10>3&&10||0
```

首先处理“10>3”，结果为“真”，用 1 表示；再处理“1&&10”，结果为“真”，用 1 表示；接着再处理“1||0”，结果为“真”。

4.3　if语句

用 if 语句可以构成选择结构。If 语句根据给定的条件进行判断，以决定是否执行或不执行某个分支程序段。C 语言的 if 语句有三种基本形式：简单 if、if-else、if-else if。

4.3.1　if语句的基本形式

1．简单if语句

简单 if 语句也叫条件执行语句，一般格式为：

```
if(表达式)语句;
```

其中，“表达式”表示条件，一般为逻辑表达式，也可以是整型表达式或者字符型表达式等，只要其值为非 0 则表示“真”，为 0 表示“假”。“语句”部分可以是一条简单语句，也可以是一条复合语句。

例如：

```
① if(x>max)max=x;
② if(x<a && x>b)
  {   x=(a+b)/2;
      y=a*x+b;
  }
```

简单 if 语句的执行过程为：先计算条件“表达式”的值，若结果为“真”则执行“语句”，然后执行 if 语句后面的语句。如果结果为假，则不执行“语句”，而直接执行 if 语句之后的语句。图 4.1 所示为其执行过程的 N-S 图。

【例 4.1】　输入任意三个整数 num1、num2、num3，求三个数中的最大数。

分析：首先取一个数预置为 max（最大数），然后再用 max 依次与其余的数逐个比较，如果发现有比 max 大的，就用它给 max 重新赋值，比较完所有的数后，max 中的数就是最大数。

N-S 图如图 4.2 所示。

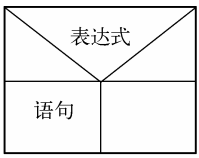


图 4.1　简单 if 语句的执行过程

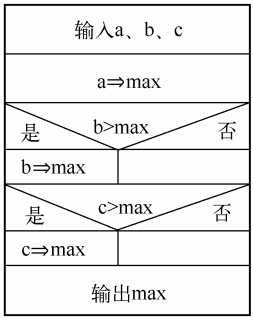


图 4.2　求三个数中的最大数

程序如下：

```
1   #include<stdio.h>
2   void  main()
3   {   int a,b,c,max;
4       printf("Please input three numbers:");
5       scanf("%d,%d,%d",&a,&b,&c);     /* 输入 3 个整数，用逗号分隔 */
6       max=a;                          /* 假设 a 最大 */
7       if(b>max)max=b;              /* 如果 b 较大则最大数为 b */
8       if(c>max)max=c;              /* 如果 c 较大则最大数为 c */
9       printf("The three numbers are:%d,%d,%d\n",a,b,c);
10      printf("max=%d\n",max);
11  }
```

2．基本if-else语句

带有 else 分支的 if 语句是 if 语句的基本形式，一般格式为：

```
if(表达式)语句 1;
else 语句 2;
```

其中，“语句 1”、“语句 2”都可以是一条简单语句或复合语句。一般称“语句 1”为 then 子句，称“语句 2”为 else 子句，把 if-else 语句理解为英语中的“if … then …, else …”结构。

下面是 if-else 结构的几种书写形式：

① `if(x<0)y=-1;else y=1;`

②
```
if(sum==m)
    printf("right!\n");
else
    printf("error!\n");
```

③
```
if(x>1 || x<-1)y=1;
else  y=sqrt(100-x*x);
```

if-else 语句的执行过程为：先计算条件“表达式”的值，如果结果为真则执行“语句 1”，否则执行“语句 2”，然后跳到整个 if 语句之外继续执行程序。图 4.3 所示为其执行过程的

N-S 图。

【例 4.2】 铁路部门规定，行李托运时重量不超过 50 千克，按 0.95 元/千克计价，超过 50 千克，超过部分按 1.5 元/千克计价。编程从键盘输入行李重量，计算并输出运费。

分析：设重量为 w，运费为 p，则 p 与 w 的关系为：

$$p=\begin{cases}0.95\times w, & w\leqslant 50\\ 0.95\times 50+(w-50)\times 1.5, & w>50\end{cases}$$

算法的 N-S 图如图 4.4 所示。

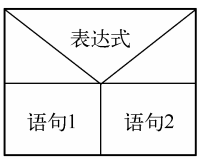


图 4.3　if-else 语句的执行过程

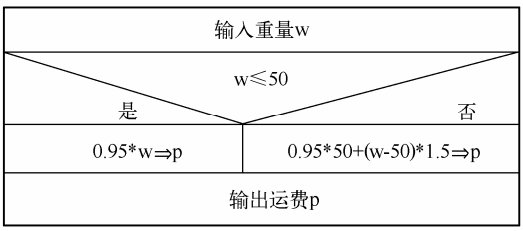


图 4.4　例 4.2 的算法 N-S 图

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   float w,p ;
4       printf("Please input weight:");
5       scanf("%f",&w);
6       if(w<=50)
7           p=0.95*w;
8       else
9           p=0.95*50+(w-50)*1.5;
10      printf("price is %.2f\n",p);
11  }
```

【例 4.3】 输入任意一个正整数 n，如果 n 是 7 的倍数，则输出该倍数；反之，输出 n 除以 7 的商和余数。例如，若 n 为 17，则输出商 2 和余数 3。

分析：要判断整数 n 是不是整数 m 的倍数，可以计算 n 除以 m 的余数 r，如果 r 为 0，则 n 是 m 的倍数，即 n 能够被 7 整除。

算法的 N-S 图如图 4.5 所示。

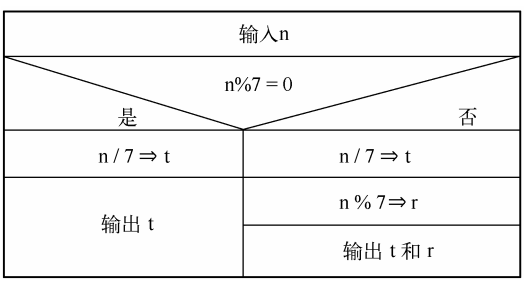


图 4.5　例 4.3 的算法 N-S 图

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int n,r,t;
4       printf("enter a number:");
5       scanf("%d",&n);
6       if(n%7==0)
7       {   t=n/7;
8           printf("quotient is %d\n",t);
9       }
10      else
11      {   t=n/7;
12          r=n%7;
13          printf("quotient is %d,remainder is %d\n",t,r);
14      }
15  }
```

注意，程序中第 6～14 行构成一个完整的 if 语句，该 if 语句的 then 子句和 else 子句都是复合语句。编程时，如果错用复合语句的“{}”会导致程序出错。

3．多分支if-else if语句

基本 if 语句只有两个分支。要表示多分支选择，可以采用 if-else if 语句。多分支 if 语句的一般格式为：

```
if(表达式 1)
    语句 1;
else if(表达式 2)
    语句 2;
……
else if(表达式 n)
    语句 n;
else
    语句 n+1;
```

执行过程为：先计算“表达式 1”，如果为“真”，则执行“语句 1”，然后跳出 if 语句，转到 if 语句后的语句（位于“语句 n+1”之后）继续执行程序；若“表达式 1”为“假”，则跳过“语句 1”，计算“表达式 2”；若“表达式 2”为“真”，则执行“语句 2”，然后跳出 if 语句，否则再计算“表达式 3”，……，以此类推。当“表达式 n”为“假”（意味着，所有表达式的值都为“假”）时，执行“语句 n+1”。图 4.6 所示为执行过程的 N-S 图。

【例 4.4】　输入一个年份，判断它是否闰年。

分析：

① 闰年的条件是年份能被 4 整除，但不能被 100 整除；或者能被 400 整除。

② 如果 X 能被 Y 整除，则余数为 0，即如果 x%y 的值等于 0，则表示 X 能被 Y 整除。

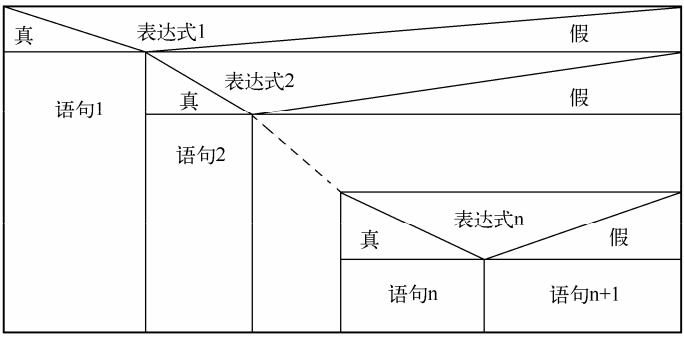


图 4.6　多分支 if-else if 语句的执行流程

③ 用标识符 leap 表示是否闰年，leap 为 1 表示是闰年，为 0 则表示非闰年。

判断某一年份是否闰年的方法是：先输入年份 year；再根据①进行判断，如果是闰年则将 leap 置为 1，否则将 leap 置为 0；最后根据 leap 的值输出判断结果，如图 4.7 所示。这种处理两种状态值的方法，对优化算法和提高程序可读性非常有效，请读者仔细体会，并注意在程序中使用。

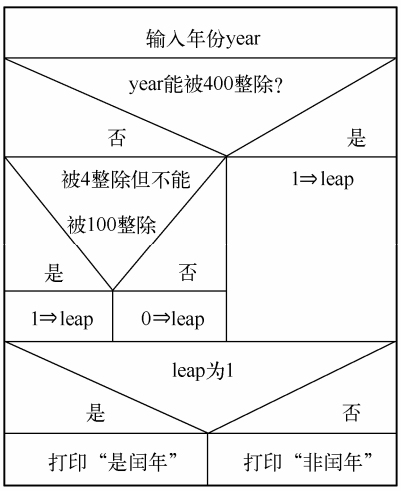


图 4.7　判断是否闰年

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {
4       int year,leap;
5       printf("Please enter year:");
6       scanf("%d",&year);
7       if(year%400==0)leap=1;
8       else if(year%4==0 && year%100 !=0)leap=1;
9       else leap=0;
10      if(leap)printf("%d is a leap year.\n",year);
```

```
11      else printf("%d is not a leap year.\n",year);
12  }
```

第 7～9 行是一个多分支 if 语句，有 3 个分支。第 10～11 行是 if-else 语句。在第 10 行中，条件表达式“leap”是对“leap!=0”的简化，这也是 C 语言程序设计中常见的一种简化策略。一般地，可以把“x!=0”简化为“x”，而把“x==0”简化为“!x”。

【例 4.5】　根据输入的一个学生的成绩，输出成绩的等级（优秀、良好、中等、及格、不及格）。成绩等级划分办法如下。

优秀：90～100

良好：80～89

中等：70～79

及格：60～69

不及格：0～59

算法的 N-S 图如图 4.8 所示。

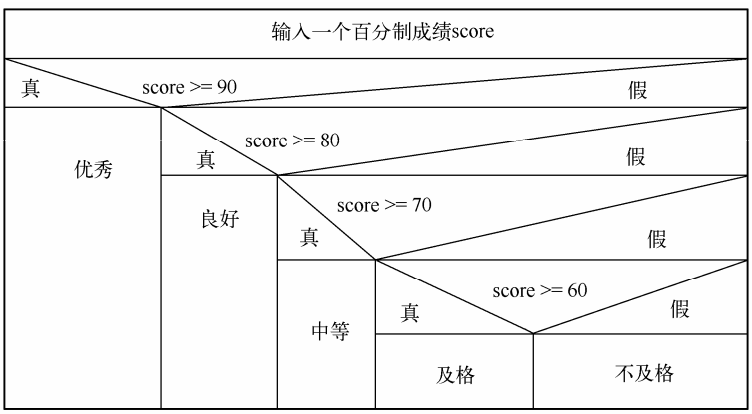


图 4.8　例 4.5 的算法 N-S 图

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int score;
4       printf("输入分数：");
5       scanf("%d",&score);
6       if(score)=90)printf("优秀\n");
7       else if(score)=80)printf("良好\n");
8       else if(score)=70)printf("中等\n");
9       else if(score)=60)printf("及格\n");
10      else printf("不及格\n");
11  }
```

4．使用if语句应该注意的问题

① 在三种形式的 if 语句中，在 if 关键字之后的括号中均为表达式。该表达式通常是逻

辑表达式或关系表达式，但也可以是其他表达式，如赋值表达式等，甚至也可以是一个变量或常量。例如，“if(a=5)……;”和“if(b)……;”都是允许的。只要表达式的值为非 0，即为“真”。因此，对于“if(a=5)……;”，表达式的值恒为 5，所以其后的语句总是要执行的。当然这种情况在程序中不一定会出现，但在语法上是合法的。

又如：

```
if(a=b)
    printf("%d",a);
else
    printf("a is 0");
```

本语句的作用是，把 b 值赋予 a，如为非 0 则输出该值，否则输出“a is 0”字符串。但如果要表达的真实意图并非如此，而是“如果 a 和 b 相等，则……”，那么该语句就是“错误”程序，错在把“= =”写成了“=”。希望读者注意这种常犯的错误。

② 在 if 语句中，条件判断表达式必须用括号括起来，在语句之后必须加分号。

③ 在 if 语句的三种形式中，所有的语句都应为单条语句，如果要想在满足条件时执行一组（即多条）语句，则必须把这一组语句用“{}”括起来组成一个复合语句。

④ 由于 if 语句与其子句（then 子句或 else 子句）之间存在归属关系，因此编程时把子句缩进一格，以体现不同层次。

⑤ else 子句部分不能独立存在，必须紧跟在 then 子句之后。所以，对于以下语句：

```
if(x>y)z=x;;
else z=y;
```

由于第一行多了一个分号“;”（形成一个空语句），else 部分与前面的 if 部分分开了，将导致出错。

因此，当 then 子句有可能是多条语句时，一个较好的习惯是把它们放在“{}”中，形成复合语句。下面的书写格式可以避免类似错误，希望读者认真体会。

```
if(x>y)
{   z=x;
}
else
{   z=y;
}
```

4.3.2　if语句的嵌套

当 if 语句子句又是 if 语句时，就构成了 if 语句的嵌套。if 语句既可以嵌套在 then 子句中，也可以嵌套在 else 子句中。可以有如下形式的嵌套：

① **if(表达式)**
　　if 语句;

② **if(表达式)**
　　if 语句;

```
else
    if 语句;
```

嵌套的 if 语句可能又是 if-else 型的，这将会出现多个 if 和多个 else 重叠的情况，这时要特别注意 if 和 else 的配对问题。

if 语句嵌套时，else 子句与 if 的匹配原则是与在它上面、距它最近且尚未匹配的 if 配对。为明确匹配关系，避免匹配错误，强烈建议：将内嵌的 if 语句，一律用大括号括起来。

另外，还要注意 if 语句的嵌套的层数不宜太多。在实际编程时，应适当控制嵌套层数（一般为 2～3 层）。

【例 4.6】　用嵌套的 if 语句改写例 4.4 的程序：判断一个年份是否闰年。

这里只给出源程序，请读者对照例 4.4 的程序进行分析。程序如下：

```
#include<stdio.h>
void main()
{
    int year,leap=0;                    /* leap 预置为 0，非闰年 */
    printf("Please input the year:");
    scanf("%d",&year);
    if(year%100!=0)
    {   if(year%4==0)leap=1;
    }
    else
    {   if(year%400==0)leap=1;
    }
    if(leap)printf("%d is a leap year.\n",year);
    else printf("%d is not a leap year.\n",year);
    }
```

请大家试一试，如果去掉第 8～9 行的大括号“{}”，会是什么结果？去掉第 11～12 行的“{}”又会怎样呢？

【例 4.7】　根据平面上一点 $M(x, y)$的坐标判断点 M 所在的象限。

分析：将 x 和 y 的值分别与 0 进行比较，即可判断出点 $M(x, y)$所处的象限。在这里，不考虑坐标轴上的点，认为它们可以处于任意象限。

程序如下：

```
#include<stdio.h>
void main()
{
    float x,y;
    printf("input x,y:");
    scanf("%f,%f",&x,&y);
    if(x)0)
        if(y)0)printf("在第一象限\n");
        else printf("在第四象限\n");
```

```
10      else
11          if(y)0)printf("在第二象限\n");
12          else printf("在第三象限\n");
13  }
```

4.4　条件运算符和条件运算表达式

条件运算符“?:”要求有 3 个运算对象，称为三目运算符，用于构造条件运算表达式，其一般形式为：

表达式 e1?表达式 e2:表达式 e3

其中，表达式 e1 是一个关系表达式或逻辑表达式，结果是逻辑值“真”或“假”。表达式 e2、e3 可以是任意表达式。

条件运算表达式的执行过程是：先求解表达式 e1；若 e1 为真（非 0），则计算表达式 e2，并将 e2 的结果作为整个条件表达式的结果；若 e1 为假（0），则计算表达式 e3，并将 e3 的结果作为整个条件表达式的结果。

例如，假设 x 已声明为整型变量，并初始化为 1，则表达式：

```
x>0?10:-10
```

的结果为 10。如果 x 初始化为–1，则结果是–10。

可以把条件运算理解为对 if-else 语句的简化，例如，语句：

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

与下面的 if 语句等价：

```
if(a>b)y=a;
else y=b;
```

关于条件运算表达式，需要注意：

① 表达式 e1、e2、e3 的类型可以不同。如果 e2 和 e3 的类型不同，则整个表达式的结果的类型为二者中级别较高的类型。

例如，表达式：

```
a>b?10.0:-10
```

的类型是实型。也就是说，如果 a>b 成立，则整个表达式的值是 10.0；如果 a>b 不成立，则整个表达式的值是实数–10.0，而不是整数–10。

② 条件运算符的优先级高于赋值运算符而低于算术运算符和关系运算符。

例如，表达式：

```
x=(a<b)?b:(a+1)
```

可以写成

```
x=a<b?b:a+1
```

③ 条件运算符具有右结合性，例如，下面的两个表达式是等价的：

```
a>b?c:d>e?f:g   等价于  a>b?c:(d>e?f:g)
```

如果 a=1、b=2、c=3、d=4、e=5、f=6、g=7，则整个表达式的值为 7。

【例 4.8】 从键盘上输入任意一个字符，如果它是大写字母，则把它转换成小写字母输出，否则不做转化，直接输出。

分析：输入的字符只要在'A'和'Z'之间，就是大写字母，再加上 32 即可转换成小写字母。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   char ch;
4       printf("Input a character:");
5       scanf("%c",&ch);
6       ch=(ch>='A' && ch<='Z')?(ch+32):ch;
7       printf("=> %c\n",ch);
8   }
```

4.5　switch语句

当分支较多时，利用 if 语句设计出来的程序会变得复杂冗长，并且很容易产生 if 与 else 不匹配的问题。C 语言提供了 switch 语句专门处理多路分支的情形，以使程序变得更简洁。但 switch 语句比较复杂，使用时要多加注意。

switch 语句的一般形式为：

```
switch(表达式)
{
    case 常量表达式1:语句1;
    case 常量表达式2:语句2;
    ……
    case 常量表达式n:语句n;
    default:语句n+1;
}
```

其中，switch、case、default 是 C 语言的保留字。

switch 语句的执行过程为：先计算“表达式”的值，并逐个与其后的“常量表达式 *i*”（即常量表达式 1、常量表达式 2、……、常量表达式 *n*）的值相比较，当“表达式”的值与“常量表达式 *i*”的值相等时，即开始执行该“常量表达式 *i*”后的“语句 *i*”，如果没有遇到 break 语句，则继续执行后面所有的 case 后（跳过 case 部分）的语句，直到遇到 break 语句或者 switch 语句最后的“}”。如果“表达式”的值与所有的“常量表达式”均不相同，则执行 default 后的语句。

在 switch 语句中，常使用 break 语句来结束 switch 语句。break 语句的格式为：

```
break;
```

是 C 语言最简单的语句之一。

【例 4.9】 某幼儿园只收 2～6 岁的儿童。2～3 岁入小班，4 岁入中班，5～6 岁入大班。根据输入的年龄，求应入的班级。

分析：设年龄为 age，当 age 为 2 和 3 时，应输出同样的内容“入小班”，age 为 5 和 6 时输出同样的内容“入大班”。该题算法比较简单，直接给出程序。

程序如下：

```
#include<stdio.h>
void main()
{   int age;
    printf("Please enter age:");
    scanf("%d",&age);
    switch(age)
    {   case 2:
        case 3: printf("入小班\n");  break;
        case 4: printf("入中班\n");  break;
        case 5:
        case 6: printf("入大班\n");  break;
        default:    printf("不能入园\n");
    }
}
```

【例 4.10】 用 switch 语句重写例 4.5，将百分制成绩划分为 5 段输出。

分析：除不及格档次外，其他各等级中每个数的十位数字均相同（100 除外）。可以用成绩的十位数字作为条件来写程序。设成绩为 score，除以 10 取整后的结果为 k，则本题的程序如下：

```
#include<stdio.h>
void main()
{   int score,k;
    printf("input a score:");
    scanf("%d",&score);
    k=score/10;
    switch(k)
    {
        case 10:
        case 9: printf("优秀\n");    break;
        case 8: printf("良好\n");    break;
        case 7: printf("中等\n");    break;
        case 6: printf("及格\n");    break;
        default:    printf("不及格\n");
```

```
15      }
16  }
```

在使用 switch 语句时，应注意以下几点：

① switch 后面的“表达式”，可以是 int、char 和枚举型中的一种；

② 在 case 后的各常量表达式的值不能相同，否则会出现错误；

③ case 后面的常量表达式仅起语句标号的作用，表示程序的入口，但不表示结束，也就是说，执行完某个 case 后的语句后，如果后面有其他 case 的语句，则继续执行这些语句，为此，需要使用 break 语句，以便跳出 switch 语句；

④ 在每个 case 后，允许有多个语句，不必用“{}”括起来；

⑤ 各 case 和 default 子句的先后顺序可以任意交换，而不会影响程序的执行结果，但要注意合理地添加 break 语句；

⑥ default 部分可以省略；

⑦ 用 switch 语句实现的多分支程序，完全可以用 if 语句来实现。

4.6　选择程序设计

【例 4.11】　将任意 3 个整数 a、b、c 按从大到小的顺序输出。

分析：

① 如果只有两个整数 a、b，可以用一个 if 语句直接输出结果，例如：

```
if(a>b)printf("%d,%d ",a,b);
else printf("%d,%d ",b,a);
```

② 对于 3 个整数，有 6 种可能：

a > b > c　　a > c > b　　b > a > c

b > c > a　　c > a > b　　c > b > a

若直接输出，写出的程序会很“累赘”，例如，

```
if(a>b&&b>c)printf("%d,%d,%d",a,b,c);
if(a>c&&c>b)printf("%d,%d,%d",a,c,b);
……
if(c>b&&b>a)printf("%d,%d,%d",c,b,a);
```

这样写程序不仅“难看”，而且“难懂”，还容易犯错。

③ 可采用交换法先排序，然后再输出结果，方法如下：

- 若 a < b，则交换 a 和 b；
- 若 a < c，则交换 a 和 c，结果 a 最大；
- 若 b < c，则交换 b 和 c，结果 a > b > c；
- 输出 a、b、c。

其 N-S 图如图 4.9 所示。

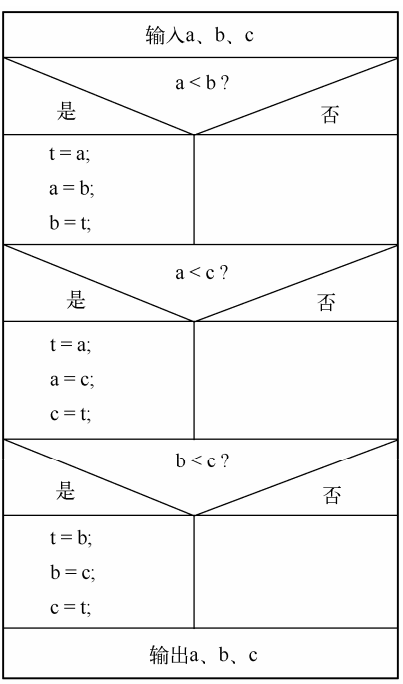


图 4.9　3 个数的排序算法

程序如下：

```
1   #include<stdio.h>
2   void main()
3    {
4       int a,b,c,t;
5       printf("input 3 numbers:");
6       scanf("%d,%d,%d",&a,&b,&c);
7       if(a<b)
8       {   t=a;a=b;b=t;        }
9       if(a<c)
10      {   t=a;a=c;c=t;        }
11      if(b<c)
12      {   t=b;b=c;c=t;        }
13      printf("sorted result: %d,%d,%d\n",a,b,c);
14  }
```

程序第 5～6 行是数据输入部分；第 7～12 行是数据处理部分，完成排序；第 13 行是数据输出部分。这种结构符合结构化程序设计的要求。

【例 4.12】 在射击比赛模拟系统中，靶面被分隔为若干同心圆，如图 4.10 所示。如果与 10、9、8、7、6 环对应的圆的半径分别为 1、2、3、4、5，脱靶计为 0 环。若以靶心为坐标原点，那么根据中靶点 $M(x, y)$ 的坐标，就可以计算出每次击发的成绩。试编程模拟这一计算过程。

分析：要计算 M 点对应的环数，就要先计算 M 到靶心的距离 $r=\sqrt{x^2+y^2}$ 。若 $r \leqslant 1$ 则为 10 环；若 $1<r \leqslant 2$，则为 9 环；……；以此类推。若 $r>5$，则为 0 环。

算法 N-S 图如图 4.11 所示。

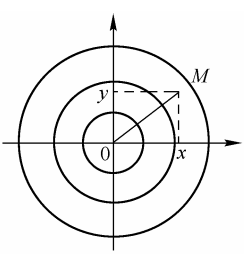


图 4.10　例 4.12 的示意图

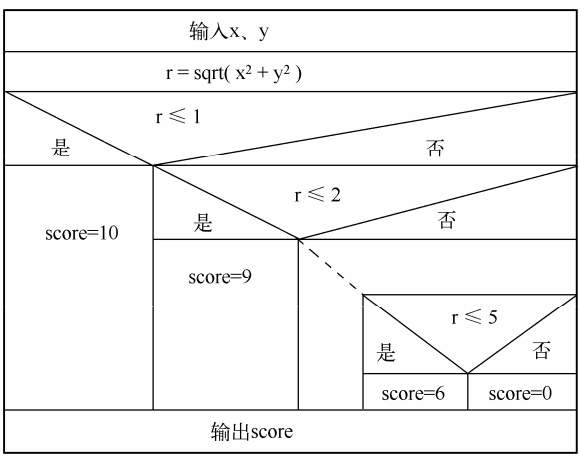


图 4.11　例 4.12 的算法

程序如下：

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

```
2   #include<math.h>              /* 用到了数学库函数 sqrt */
3   void main()
4   {
5       float x,y,r;
6       int score;
7       printf("input x,y:");
8       scanf("%f,%f",&x,&y);
9       r=sqrt(x*x+y*y);
10      if(r<=1)score=10;
11      else if(r<=2)score=9;
12      else if(r<=3)score=8;
13      else if(r<=4)score=7;
14      else if(r<=5)score=6;
15      else score=0;
16      printf("score:%d\n",score);
17  }
```

【例 4.13】　求一元二次方程 $ax^2+bx+c=0$ 的实根。如果没有实根，则输出提示信息。

分析：根据数学知识，对于一元二次方程，求解的关键是判断 disc=b^2-4ac 的值。disc>0 时，方程有两个不相等的实根；disc=0 时，方程有两个相等的实根；disc<0 时，方程没有实根。由此不难画出算法的 N-S 图，如图 4.12 所示。

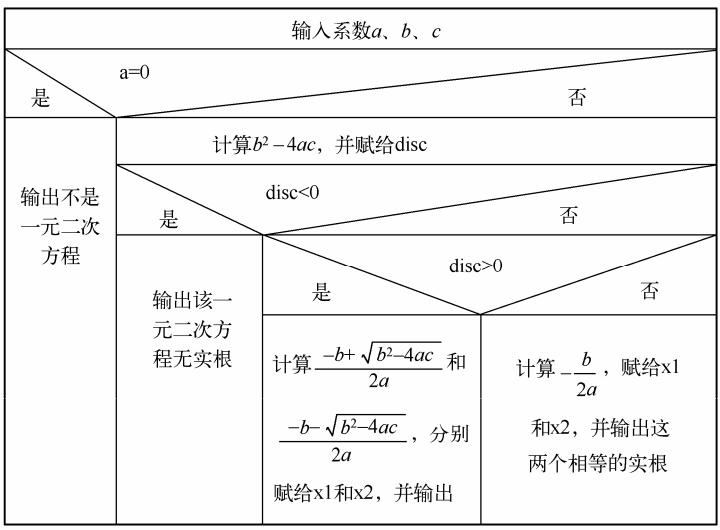


图 4.12　一元二次方程的求根算法

程序如下：

```
1   #include "stdio.h"
2   #include "math.h"
3   void main()
4   {
5       float a,b,c,disc,x1,x2;
6       printf("Please input a、b、c: ");
```

```
    scanf("%f%f%f",&a,&b,&c);   /* 输入数据用空白分开 */
    if(a==0)
        printf("The equation is not a quadratic.\n");
    else
    {
        disc=b*b-4*a*c;
        if(disc<0)
            printf("The equation has not real roots.\n");
        else if(disc>0)
        {
            x1=(-b+sqrt(disc))/(2*a);
            x2=(-b-sqrt(disc))/(2*a);
            printf("The equation has distinct real roots:%7.3f and
            %7.3f\n",x1,x2);
        }
        else
        {
            x1=-b/2/a;
            printf("The equation has two equal real roots:%7.3f\n",x1);
        }
    }
}
```

以下是分别输入不同参数运行了 4 次的结果：

① Please input a、b、c: 1 5 4 ↙
The equation has distinct real roots: -1.000 and -4.000

② Please input a、b、c: 1 2 1 ↙
The equation has two equal real roots: -1.000

③ Please input a、b、c: 2 3 5 ↙
The equation has not real roots.

④ Please input a、b、c: 0 2 5 ↙
The equation is not a quadratic.

通过以上例子，可以看到选择程序设计的特点：从整体上看，程序的执行流程还是自顶向下的；从程序的内部具体来看，程序的执行流程是可以根据所给定的条件是否成立来决定的。

对于选择程序，程序的一次运行只能执行选择结构的某个分支，其他分支则不被执行。因此，在调试程序的时候，应该选择不同的输入数据多次试运行程序，确保每个分支都被调试过，以免遗漏错误。以例 4.13 为例，程序有 4 个分支，因此至少选择 4 组不同类型的数据进行测试，程序最少也要试运行 4 次。

习　题　4

4.1　已知 a=10，b=5，请写出下列表达式的运行结果。

（1）!(a+b)　　（2）a%b||1　　（3）a&&b

（4）a>=5&&b<5　　（5）b||a&&0a%3–1　　（6）a&&(a++)/(++b)

4.2　写出下列表达式的值

（1）1<100&&4<7||60>=5　　（2）!(12<3==3)

（3）!(100<=300)||(1!=2)　　（4）!(5>3)&&(30<=90)

4.3　选择题

（1）逻辑运算符两侧的运算对象的数据类型为（　　）。

A. 只能是 0 和 1　　B. 只能是 0 或非 0 正数

C. 只能是整型或字符型数据　　D. 可以是任何类型的数据

（2）判断 char 型变量 ch 是否为大写字母的正确表达式是（　　）。

A. 'A'<=ch<='Z'　　B. (ch>='A')||(ch<='Z')

C. (ch>='A')&&(ch<='Z')　　D. ('A'<= ch)AND('Z'>= ch)

（3）若希望当 A 的值为奇数时，表达式的值为“真”，A 的值为偶数时，表达式的值为“假”，则以下不能满足要求的表达式是（　　）。

A. A%2= =1　　B. !(A%2= =0)　　C. !(A%2)　　D. A%2

（4）设有 int a=1, b=2, c=3, d=4, m=2, n=2，执行(m=a>b)&&(n=c>d)后 n 的值为（　　）。

A. 1　　B. 2　　C. 3　　D. 4

（5）以下程序的运行结果是（　　）。

```
void main()
{   int a,b,d=241;
    a=d/100%9;
    b=(-1)&&(-1);
    printf("%d,%d",a,b);
}
```

A. 6,1　　B. 2,1　　C. 6,0　　D. 2,0

（6）已知 int x=10, y=20, z=30，以下语句执行后 x, y, z 的值是（　　）。

```
if(x>y)z=x;x=y;y=z;
```

A. x=10, y=20, z=30　　B. x=20, y=30, z=30

C. x=20, y=30, z=10　　D. x=20, y=30, z=20

（7）以下程序的运行结果是（　　）。

```
void main()
{   int m=5;
    if(m++>5)
```

```
        printf("%d\n",m);
    else
        printf("%d\n",m--);
}
```

A. 4　　B. 5　　C. 6　　D. 7

（8）若运行时给变量 x 输入 12，则以下程序的运行结果是（　　）。

```
void main()
{   int x,y;
    scanf("%d",&x);
    y=x>12?x+10:x-12;
    printf("%d\n",y);
}
```

A. 12　　B. 22　　C. 1　　D. 0

4.4　编程计算下面的函数，其中，x 由键盘输入。

$$y=\begin{cases}(x-2)^2, & x \leqslant -2\\ \mathrm{e}^x, & -2<x\leqslant 2\\ (x+2)^2, & x>2\end{cases}$$

4.5　编程实现：输入整数 a 和 b，若 a^2+b^2 大于 100，则输出 a^2+b^2 百位以上的数字，否则输出 a、b 之和。

4.6　编程实现以下功能，读入两个数（d1,d2）和一个运算符（op），计算 d1 op d2 的值。例如，若输入 15、5 和“/”，则计算“15 / 5”，结果为 3。

4.7　从键盘输入两个整数，输出较大数（用条件表达式求解）。

4.8　输入一个正整数，编程判断其是否既是 5 又是 7 的倍数。若是，则输出“yes”；否则输出“no”。

4.9　输入一个正整数，判断其是否能被 3、5、7 整除，并输出能同时被几个数整除。

4.10　从键盘输入 4 个整数 a、b、c、d，找出其中的最大值。

4.11　从键盘输入一个字符，如果是大写字母或者是小写字母，就输出其对应的 ASCII 码。

4.12　对于给定的一个百分制成绩，输出相应的五分制成绩。提示：90 分以上为'A'，80～89 分为'B'，70～79 分为'C'，60～69 分为'D'，60 分以下为'E'。要求，必须使用 switch 语句，并且只用一个输出语句输出计算结果。

4.13　企业发放奖金是根据利润提成的，提成的办法如下。

（1）利润（用 I 表示）≤10 万元时，可提成 10%；

（2）超过 10 万元时：

- 10 万＜I≤20 万的部分，可提成 7.5%，低的部分同上（下同）；
- 20 万＜I≤40 万的部分，可提成 5%；
- 40 万＜I≤60 万的部分，可提成 3%；
- 60 万＜I≤100 万的部分，可提成 1.5%；
- 超过 100 万的部分，可提成 1%。

从键盘输入当月利润 I，输出可发放的奖金总额。

4.14　给出一个不超过 5 位的正整数，编程完成下面的功能：

（1）判断它是几位数；

（2）计算其各位数字的和；

（3）将其各位数字按逆序排列组成一个新数，并比较其与原数的大小，然后按从大到小的顺序输出。

第 5 章　循环程序设计

学习目标

通过本章学习，应该掌握：

- 循环的基本概念
- while、do-while 和 for 语句的使用方法
- 循环结构程序设计
- 循环控制条件及其应用

5.1　概　　述

计算机程序由若干语句顺序组成。顺序结构的程序只能按照语句的先后顺序从前向后依次执行，分支结构程序虽然允许在执行了某个判断后跳过某些语句执行后面的语句，但也只能按语句的前后次序执行。有时需要在程序中重复执行某个语句或语句块，就需要循环结构。

【例 5.1】　计算累加和 $\sum_{i=1}^{100} i = 1+2+3+\mathrm{L}\ +100$。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int i=1,sum=0;
4       while(i<=100)
5       {   sum=sum+i;
6           i++;
7       }
8       printf("Sum=%d\n",sum);
9   }
```

程序的执行过程可以用图 5.1 表示，不难看出，第 5 行的赋值语句执行了 100 次。如果程序中的某条语句需要执行 1 次以上，就需要使用循环结构了。

现实生活中也不乏循环的例子，比如击鼓传花游戏，大家坐成一个圈，鼓声响起的时候将花束顺序交到下一个人的手里，依次向下传递，当鼓声突然中断时停止传花，花束落在谁的手里便成为输家。再比如，在 4×100 米接力赛跑中，第 1 个人跑完 100 米后将接力棒传给第 2 个人，第 2 个人再跑 100 米，然后是第 3 个人，直到第 4 个人跑完最后一个 100 米。不难看出，这些实例有以下两个共同的特点。

① 都要完成相同的任务。击鼓传花中每个人都从前面的人手里接过花束再传给下一个人，接力赛跑中每个人都要跑 100 米。

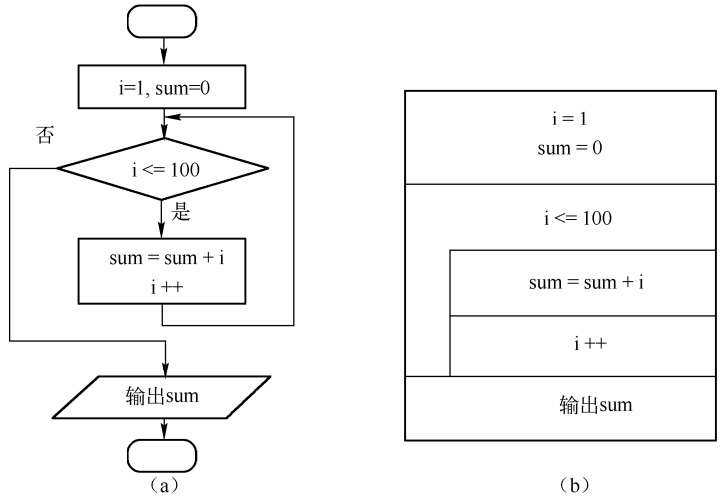


图 5.1　例 5.1 的程序执行过程

② 都有结束条件。击鼓传花以鼓声为号，鼓停游戏止；接力赛跑则是所有人都要跑完 100 米。

这正是循环结构的计算机程序的特点。在例 5.1 中，第 4～7 行构成循环结构，第 5 行的赋值语句是每次都要执行的，与第 6 行一起构成循环的**循环体**。第 4 行的 i≤100 作为循环结束的判断依据，当 i 大于 100 时循环结束，程序将执行后面的输出语句（第 8 行），因此 i≤100 是该循环的**循环控制条件**。

容易理解，计算机程序中的语句不论是否重复执行，都只能执行有限次，也就是说循环体的执行次数是有限的，否则就是“死循环”。存在“死循环”的程序运行时，一旦开始循环就会无休止地循环下去，除非强行中断程序的运行。通常，在集成开发环境（IDE，如 Turbo C、Visual C++ 6.0 等）中可以通过按 Ctrl+Break 组合键强行中断程序的运行。如果程序运行脱离了集成开发环境，在 DOS 下，就需要重新启动计算机，而在 Windows 下，可以使用“任务管理器”强行中止相应的进程。

循环控制条件必须保证循环体重复执行有限次数后结束，因此循环体中必须有能够修改循环控制条件的语句。例 5.1 中，第 6 行的 i++运算就起到了这样的作用，每次循环 i 的值增加 1，可以保证 i 从 1 顺序地增加到 100。当 i 等于 101 时循环控制条件的值为“假”，循环结束。可以想象，如果没有 i++这一运算，i≤100 就总是成立的，该循环将成为“死循环”。在该循环结构中变量 i 的作用是特别的，不仅参与 sum=sum+i 运算，是运算变量，而且控制循环次数，是**循环控制变量**。

读者可能已经发觉，第 3 行的 i=1 也与循环有关，与 i++运算一起决定了循环的次数。i=1 是对循环控制变量的初始化，也是**循环的初始化**。

【例 5.2】　计算 100 以内的偶数的和 sum = 2+4+6+…+100。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int i=2,sum=0;
4       while(i<=100)
```

```
5       {   sum=sum+i;
6           i+=2;                /* 相邻的两个偶数之差为 2 */
7       }
8       printf("Sum=%d\n",sum);
9   }
```

如果题目改成了求 50～100 之间的偶数的和，只需修改第 3 行的 i=2 为 i=50 即可。

5.2　while语句和do-while语句

C 语言中，可以用 while 语句实现当型循环，用 do-while 语句实现直到型循环。

5.2.1　用法

while 语句的一般格式为：

while(表达式)语句

do-while 语句的一般格式为：

do 语句 while(表达式);

说明：

① 括号中的“表达式”是循环控制条件，但可以是任何表达式，而不仅限于关系表达式和逻辑表达式。如果“表达式”的值不是逻辑值，则将非 0 值看做逻辑“真”，而把 0 看做逻辑“假”。

②“语句”可以是单一语句，也可以是复合语句。如果是单一语句，语句后面应该有结束符“;”。如果是复合语句，必须放在花括号中。循环体为单一语句时，可以为空语句，即“语句”部分只有“;”。

③ 如果 while 语句的括号后面只有“;”，则循环体为空。如果 do-while 语句的括号后面（分号前面）还有其他符号，则出现语法错误。

5.2.2　执行过程

while 语句的执行过程如图 5.2 所示，do-while 语句的执行过程如图 5.3 所示。

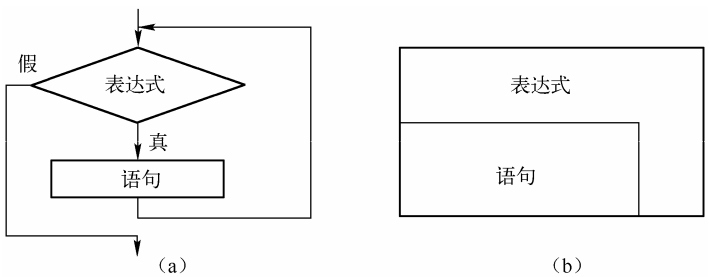


图 5.2　while 语句的执行过程

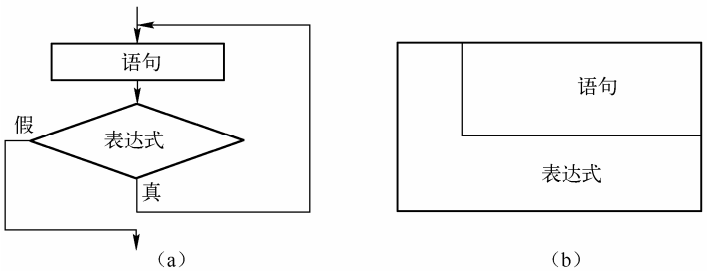


图 5.3　do-while 语句的执行过程

while 语句的执行过程如下：

① 计算“表达式”的值；

② 若“表达式”的值不是 0，则为“真”，继续执行③，否则，结束循环，转到 while 语句后面的语句继续执行程序；

③ 执行“语句”部分，即执行循环体；

④ 返回①继续执行。

do-while 语句的执行过程与此类似，只是顺序不同，读者可以自己体会。

while 循环是“先判断后循环”，如果开始时循环条件不满足，循环体就一次也不执行。do-while 循环是“先循环后判断”，即使第一次判断的结果为“假”，循环体也已经执行了一次。但不论哪种循环，循环体中必须包含能够使循环条件变为“假”的运算。

【例 5.3】　重写例 5.1 计算累加和的程序。

方法一：循环体为单一语句。

```
1   #include<stdio.h>
2   void main()
3   {   int i=1,sum=0;
4       while(i<=100)
5           sum=sum+i++;    /* 该语句有两个功能，较难理解，因此不建议使用 */
6       printf("Sum=%d\n",sum);
7   }
```

方法二：采用 do-while 语句。

```
1   #include<stdio.h>
2   void main()
3   {   int i=1,sum=0;
4       do
5       {   sum=sum+i;
6           i++;
7       } while(i<=100);
8       printf("Sum=%d\n",sum);
9   }
```

可见，while 语句和 do-while 语句是可以通用的，使用 while 语句编写的程序可以改用 do-while 语句编写，反之亦然。

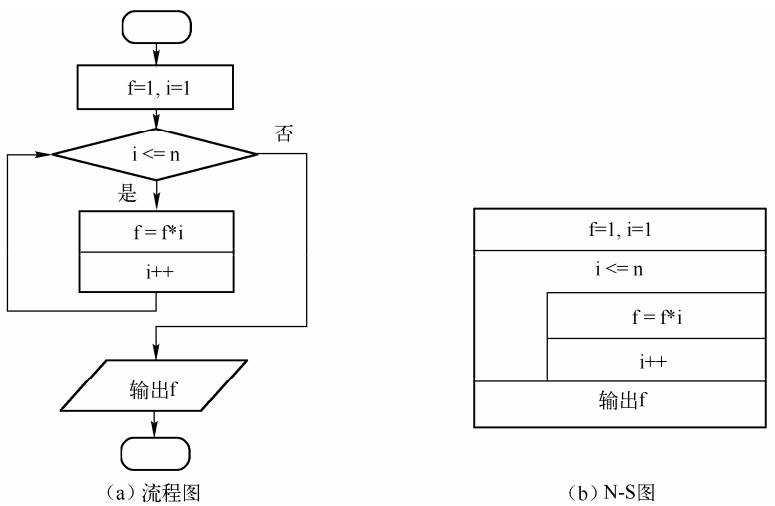
值得注意的是，在 do-while 语句中，do 和 while 是同一语句的两个部分，不能分开。在 do 和 while 之间只有一个语句（可以是复合语句）。不妨体会一下，如果缺少了 do，会出现什么结果?

5.2.3　循环的嵌套

如果在一个循环的循环体内包含另一个完整的循环则称为**循环的嵌套**，其中被嵌套的循环称为**内循环**，而嵌套了内循环的循环是**外循环**。内循环还可以嵌套循环，形成多级（层）嵌套。

【例 5.4】　顺序打印 1～10 的阶乘，即 1!，2!，…，10!

分析：数学上，$n! = \prod_{i=1}^{n} i$，与求 $\sum_{i=1}^{n} i$ 的方法类似。计算 $n!$ 的基本过程可用图 5.4 表示。



（a）流程图　　（b）N-S图

图 5.4　计算 $n!$

方法一：用内循环计算 $n!$，外循环输出结果。

```
1   #include<stdio.h>
2   void main()
3   {   int i=1,j;
4       float f;
5       while(i<=10)
6       {   j=1;
7           f=1;                /* 注意：为什么不能放在第 4 行中？ */
8           while(j<=i)         /* 内循环计算 i! */
9           {   f=f*j;
10              j++;
11          }
12          printf("%2d!=%.0f\n",i,f);
13          i++;
14      }
```

```
15  }
```

方法二：采用单循环结构。

```
1   #include<stdio.h>
2   void main()
3   {   int i=1;
4       float f=1;
5       while(i<=10)
6       {   f=f*i;
7           printf("%2d!=%.0f\n",i,f);
8           i++;
9       }
10  }
```

有的读者可能认为方法一程序较长，比较烦琐，不如方法二简单。但试想一想，如果不是从 1 的阶乘开始计算，问题改为“顺序打印 8～10 的阶乘”结果如何？方法一把计算阶乘的过程与输出计算结果分开，可以满足自由组合程序的需要，更具灵活性和可用性，是我们推荐的方法。

循环的嵌套不仅可以发生在相同语句之间，如 while 嵌套 while、do-while 嵌套 do-while，while 语句和 do-while 语句也可以互相嵌套，因此，以下嵌套结构都是合法的：

（1）
```
while(…)
{   …
    while(…)
    {…}
    …
}
```

（2）
```
while(…)
    {   …
        do
        {   …
        }while(…);
        …
    }
```

（3）
```
do
{   …
    while(…)
    {…}
    …
} while(…);
```

（4）
```
    do
    {   …
        do
        {   …
        } while(…);
        …
} while(…);
```

但要注意内循环必须是完整的循环，不允许内外循环交叉，比如下面的程序块是非法的，存在语法错误：

```
f=1;
while(i<10)
{   i=1;
    do
```

```
    {   f=f*i;
        i++;
}
    } while(i<10);
```

另外，内外循环的循环控制条件通常是分开的，相对独立的，使用不同的控制变量，以避免出现混乱。例如，在下面的程序段中，由于内外循环使用了同一个循环控制变量，导致该程序所要表达的语义让人费解：

```
i=1;s=0;
while(i<3)
{   s=s+f;
    i=1;f=1;
    while(i<5)
    {   i+=2;
        f=f*i;
    }
    i++;
}
```

5.2.4　应用举例

【例 5.5】　输入一个 5 位数，如 12345，计算并输出各位的和，即 15。

分析：有两个方法获取输入的数字：可以将输入的 5 位数看做是 5 个独立的数字字符；也可以把它看做是某个整数，先读取该整数，再分离出各位数字。

方法一：依次读取每一个数字对应的字符，将其转换为对应的数字后再求和。

算法如图 5.5 所示。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   char ch;
4       int i=0,sum=0;
5       printf("Enter a number with 5 digits:");
6       while(i<5)
7       {   ch=getchar();       /* 或者用 scanf("%c",&ch);*/
8           sum+=ch-'0';        /* 将数字字符 ch 转换为对应的整数 */
9           i++;
10      }
11      printf("Sum of these 5 digits is%d\n",sum);
12  }
```

运行结果如下：

```
Enter a number with 5 digits:12345↙
```

```
Sum of these 5 digits is 15
```

程序中，变量 i 的作用是控制循环次数，也就是说，i 是由第 7～10 行组成的循环体的执行次数。ch 是从键盘缓冲区中读取到的字符，代表该 5 位数的某一位。第 8 行的运算过程是，将 ch 中的字符转换为数字，再累加到 sum 中。

对于字符型数据，有几种运算是在 C 语言编程中经常遇到的，希望读者能够理解和掌握：

① 如果 ch 为数字字符，则表达式 ch - '0'的值为该字符对应的整数；

② 如果 ch 为大写字母，则表达式 ch-'A'+'a' 将该字符转换为小写字母；

③ 如果 ch 为小写字母，则表达式 ch-'a'+'A' 将该字符转换为大写字母。

方法二：设输入的 5 位数是 x，令 x1=x，则 r = x % 10 为 x1 的个位数；再令 x = x/10，则 r = x % 10 为 x1 的十位数；以此类推，可按逆序求出 x1 的各位数。算法如图 5.6 所示。

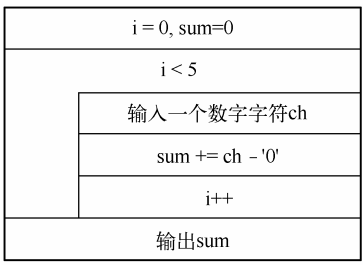


图 5.5　求输入的数字字符的累加和

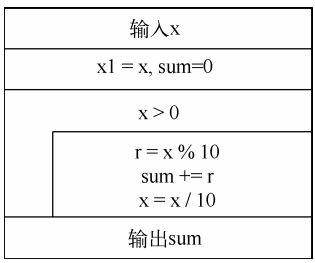


图 5.6　求自然数各位数的和

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int x,x1,r,sum=0;
4       printf("Enter a number:");
5       scanf("%d",&x);
6       x1=x;
7       while(x>0)
8       {   r=x%10;
9           sum+=r;
10          x/=10;
11      }
12      printf("Sum of the digits in%d is%d\n",x1,sum);
13  }
```

运行结果为：

```
Enter a number:12345↙
Sum of the digits in 12345 is 15
```

【例 5.6】　任意输入一行字符，统计字母 a 和 A 的个数。

分析：由于事先不知道输入的字符有多少，因此循环次数不固定，本程序宜采用直到型循环。结束循环的条件是读取到的字符为回车符'\n'。

算法如图 5.7 所示。

程序如下：

```
#include<stdio.h>
void main()
{   char ch;
    int count=0;
    printf("Enter characters:");
    do
    {   ch=getchar();
        if(ch=='A' || ch=='a')count++;  /* 判断是否为字母 A 或 a */
    } while(ch !='\n');              /* 判断是否为回车 */
    printf("Counter of 'a' or 'A':%d\n" ,count);
}
```

通常，do-while 循环和 while 循环是可以互相转换的，读者可以尝试用 while 语句改写这个程序。

【例 5.7】　利用公式 $\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\mathrm{L}$ 求 π 的近似值。

分析：

① 由于受到计算机中数字的精度的限制，常规的计算方法不可能达到很高的精度。这里取最后一项的绝对值小于 10^{-5} 作为控制条件。

② 经过分析，公式右边的第 n 项为 $\frac{(-1)^{n+1}}{2n-1}$，因此可将公式变形为 $\frac{\pi}{4}=\sum_{n=1}\frac{(-1)^{n+1}}{2n-1}$。如何计算 $(-1)^{n+1}$ 呢？读者可能首先想到的是用指数运算，其实大可不必，可以取 f 的初值为 −1，那么第 1 项为 $f=-f$，第 2 项也是 $f=-f$，……

算法如图 5.8 所示。

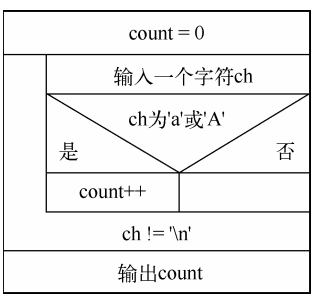


图 5.7　统计字母 a 或 A 的个数

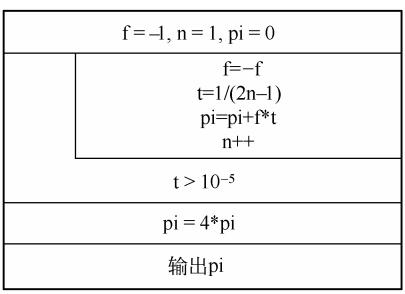


图 5.8　计算圆周率

程序如下：

```
#include<stdio.h>
void main()
{   int  f=-1,n=1;
```

```
4       float  t,pi=0;          /* t 为最后一项的值 */
5       do
6       {   f=-f;               /* 第 n 项的符号 */
7           t=1.0/(2*n-1);      /* 用 1.0 是为了避免整除 */
8           pi+=f*t;
9           n++;
9       } while(t>1e-5);
10      pi=4*pi;
11      printf("PI=%.4f\n",pi);
12  }
```

在程序的第 7 行，由于 n 是整型的，如果右边的表达式写成 1/(2*n–1)，则会由于整除导致计算结果为 0。

程序的运行结果如下：

```
PI=3.1414
```

为什么计算精度并没有达到小数点后第 5 位呢？是否可以通过改用双精度实数等方法提高精度呢？类似的问题，读者自己也不难回答。问题是有没有更好的办法计算高精度的π呢？

π的计算是一个古老的数学问题，也是对现代高性能计算机、计算方法的挑战。公元 5 世纪的祖冲之将圆周率精确到小数点后第 7 位，奠定了他在数学史上的至高地位。现代数学家，借助超级计算机已经可以计算到小数点后的 1.2 兆亿位。有兴趣的读者可以在搜索引擎（如 Google）中搜索“圆周率的计算”了解这方面的情况。

5.3　for语句

for 语句是最常用的循环控制语句。for 语句的功能强大，使用灵活，变化多样。

5.3.1　用法

for 语句的格式为：

for(表达式 1;表达式 2;表达式 3)语句

其中，“表达式 1”用于循环的初始化；“表达式 2”是 for 语句的循环控制条件，可以是任何形式的表达式；“表达式 3”用于修改循环控制变量的值，目的是使“表达式 2”的值变为 0，以结束循环；“语句”部分是 for 语句的循环体，可以是单一语句，也可以是复合语句；循环体为空时，“语句”部分只有一个“;”。

for 语句中的 3 个表达式都可以省略，如果省略了“表达式 2”，则默认循环控制条件恒为“真”。

如果同时省略括号内的 3 个表达式，则 for 语句变为：

```
for(;;)语句
```

此时，必须保证“语句”部分包含结束 for 语句的操作，以避免出现“死循环”，参见 5.4 节的 break 语句。

需要特别说明的是，括号内的两个分号用于分隔 3 个有不同作用的表达式，是不能省略的！

5.3.2　执行过程

for 语句的执行过程如下（见图 5.9）：

① 计算“表达式 1”；

② 计算“表达式 2”，如果“表达式 2”的值为 0（表示“假”），则转到⑤，否则转到③；

③ 执行“语句”部分；

④ 计算“表达式 3”后，转到② ；

⑤ for 语句结束。

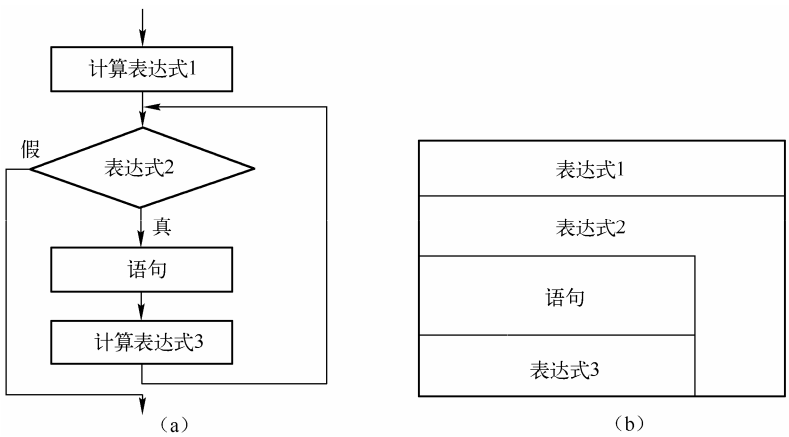


图 5.9　for 语句的执行过程

从形式上看，for 语句是对 while 语句的扩充，起到了相同的作用。但对比 while 语句，for 语句所要表达的语义更完整。如果采用 while 语句构建循环，循环的初始化条件必然放在 while 语句的前面，是相对独立而又不可或缺的，有时由于疏忽会漏写初始化语句，造成程序出错。而采用 for 语句时，所有与循环相关的内容都可以包含在 for 语句之中，形成一个完整的整体，不容易失误。

【例 5.8】　用 for 语句重写例 5.1 计算累加和的程序。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int i,sum=0;
4       for(i=1;i<=100;i++)
5           sum+=i;
6       printf("Sum=%d\n",sum);
7   }
```

【例 5.9】　任意输入 10 个数，求平均值。

分析：平均值的定义为$\frac{1}{n}\sum_{i=1}^{n}x_i$，因此要先求索输入数据的累加和，方法与$\sum_{i=1}^{n}i$相同。

程序如下：

```
#include<stdio.h>
void main()
{   int  i;
    float  x,sum=0;
    printf("Enter 10 numbers one by one:\n");
    for(i=1;i<=10;i++)
    {   scanf("%f",&x);
        sum+=x;
    }
    printf("Average is%f\n",sum/10);
}
```

运行程序时，10 个数可以分 10 次输入，也可以在一行内完成。这是由于用 scanf 函数读取数值时，每次读取键盘缓冲区中的 1 个数，其余的数仍在缓冲区中等待下次读取。但要注意，这 10 个数之间应该用空白（空格、制表符或回车符）分开。下面是某次运行的结果：

```
Enter 10 numbers one by one:
1  3  5  7  2  4  6  8  9  10 ↙
Average is 5.500000
```

5.3.3　循环的嵌套

for 语句可以嵌套 for 语句构成循环的嵌套，也可以与 while 语句、do-while 语句互相嵌套。

【例 5.10】　用 for 语句重写例 5.4 的程序。

程序如下：

```
#include<stdio.h>
void main()
{   int  i,j;
    float  f;
    for(i=1;i<=10;i++)
    {   f=1;
        for(j=1;j<=i;j++)   f=f*j;   /* 计算 i! */
        printf("%2d!=%.0f\n",i,f);
    }
}
```

```
******
 ******
  ******
   ******
```

图 5.10　几何图形

【例 5.11】　打印如图 5.10 所示的几何图形。

分析：对于这类问题，每行中星号的个数和总行数等都应该用循环结构进行控制，而不是直接输出若干行字符串。方法是，对于每一行，先打印出左边的空格，然后再打印星号。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int i,j;
5       for(i=1;i<=4;i++)
6       {   for(j=0;j<i;j++)putchar(' ');   /* 打印每行开头的空格 */
7           for(j=1;j<=6;j++)putchar('*');  /* 打印每行的6个星号 */
8           putchar('\n');                  /* 换行 */
9       }
10  }
```

程序中，外循环开始于第 5 行，至第 9 行结束，循环体由 3 行组成：第 6 行为第一个内循环，输出每一行前面的空白（若干空格）；第 7 行为第二个内循环，在当前行上打印 6 个星号；第 8 行输出一个换行符，使得下一次的输出发生在新的一行上。

【例 5.12】　甲、乙两个会计进行点钞票比赛，甲的速度为 5 张/秒，乙为 8 张/秒。乙在甲已经点了 100 张钞票后才开始，问：只要几秒时间乙就可以超过甲?

分析：经过 t 秒，甲点过的钞票数有 $x=100+5t$ 张，乙点过的钞票数为 $y=8t$ 张。问题就是求 $y \geqslant x$ 时的 t 值。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int  t;
4       int  x=100,y=0;
5       for(t=0;x>y;t++)
6       {   x=100+5*t;
7           y=8*t;
8       }
9       printf(" Time is%d seconds\n",t);
10  }
```

本程序的特殊性在于控制循环控制的表达式并不直接由控制变量构成，循环控制变量的作用是间接的。

5.3.4　for语句的变化形式

1. 可以省略for语句的“表达式 1”或“表达式 3”，也可以都省略

省略“表达式 1”时，须保证在 for 语句前面进行循环的初始化。若省略了“表达式 3”，

必须在循环体中修改控制变量的值，以避免“表达式 2”的值总为非 0，出现“死循环”。例如下面的程序段是合法的：

```
1       i=10;
2       for(;i<=20;)
3       {   sum+=i;
4           i++;
5       }
```

它是语句“for(i=10; i<=20 ; i++) sum+=i;”的展开形式。读者很容易就可以把这段程序改为 while 语句结构。

【例 5.13】　将用键盘输入的若干字符顺序输出到屏幕上。

分析：用 getchar 函数获取输入的字符，然后用 putchar 函数输出这个字符。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   char ch;
4       for(;(ch=getchar())!='\n';)putchar(ch); /* 当 ch 为回车符时循
                                                   环结束 */
5   }
```

运行程序时，结果如下：

```
Apple ↙
Apple
```

为什么不是 AApppplle 呢？因为 getchar 函数是从键盘缓冲区中读取字符的，在按下回车（Enter）键后，输入的字符才进入键盘缓冲区。

该程序可以用 while 语句改写为：

```
1   #include<stdio.h>
2   void main()
3   {   char ch;
4       while((ch=getchar())!='\n')putchar(ch);
5   }
```

二者是完全等价的。可见此时使用 for 语句并没有多少优势。

2．如果省略“表达式 2”，则循环控制条件总为“真”

下面的语句是“合法”的，但是构成了“死循环”：

```
for(i=1;;i++)sum+=i;
```

下面的程序在运行时会因为“除以零”而出错：

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

```
2   void main()
3   {   char  i;
4       int sum=0;
5       for(i=1;;i++)   sum+=1000/i;
6       printf("sum=%d\n",sum);
7   }
```

这是因为计算机中的字符型整数是有限的，取值范围为–128～127。i 从 1 开始做 i++运算，当 i 等于 127 时，下一个可取得的数是–128，然后是–127，从而形成了一个循环，当 i 的取值为 0 时，程序出错。

为避免类似情况的出现，需要在 for 语句的循环体中引入 break 语句（见 5.4.2 节）以结束循环，上面的程序可改为：

```
1   #include<stdio.h>
2   void main()
3   {   char i;
4       int sum=0;
5       for(i=1;;i++)
6       {   if(i<=0)break;      /* 当 i 变为负数时，结束循环 */
7           sum+=1000/i;
8       }
9       printf("sum=%d\n",sum);
10  }
```

5.4　循环的控制

5.4.1　复杂的循环控制条件

对循环的控制问题，不仅仅是防止“死循环”，还包括选择恰当的时机，正确地结束循环。有的时候，循环控制条件是复杂的，还可能是多种条件的综合。

【例 5.14】　编程实现求某班同学的平均成绩。假设每个班最多有 30 个人，成绩取 0～100 之间的数。成绩是一个一个依次输入的，如果输入的不是 0～100 之间的数，或者已经输入了 30 个数，则结束输入，输出计算结果。

分析：该程序的算法如图 5.11 所示。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int i=0,flag=1;
4       float score,ave=0;
5       printf("Enter scores one by one:\n");
6       while(i<30 && flag==1)  /* 循环条件 */
```

```
7       {   scanf("%f",&score);
8           if(score<0||score>100)
9               flag=0;          /* 如果输入的 score 超出范围则结束循环 */
10          else
11          {   ave+=score;
12              i++;
13          }
14      }
15      if(i>0)ave=ave/i;        /* 如果第一次输入的 score 非法，则不执行 */
16      printf("Average:%.2f\n",ave);
17  }
```

本程序的循环控制条件不仅与班级人数有关（用变量 i 控制），而且与输入的数字 score 相关。程序中用标记变量 flag 是否为 1 表示输入的 score 是否合法，如果输入的 score 超出范围，不仅不计入合计，而且将 flag 改为 0，以迫使 while 语句提前结束。while 语句的循环次数是不固定的，与程序的执行过程相关，取决于用户的输入。

这种情况是会经常遇到的，例如，下面的程序块常用来对输入的数据进行限制：

```
do  scanf("%d",&iInput);
while(iInput<0 || iInput>MAXIMUM);
```

类似地，经常用下面的程序块要求用户选择 Yes 或 No，忽略其他输入：

```
printf("Enter Yes/No?");
do ch=getchar();
while(ch!='y'&& ch!='Y'&& ch!='n'&& ch!='N');
```

【例 5.15】 利用随机数产生一个乘法算式，用户输入算式的运算结果，由程序判断对错。

分析：在 C 语言中，可以使用 rand 函数产生伪随机数。请读者查阅有关书籍了解这方面的知识。

算法如图 5.12 所示。

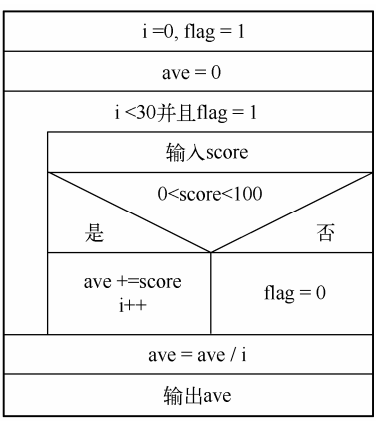


图 5.11　输入数据的校验

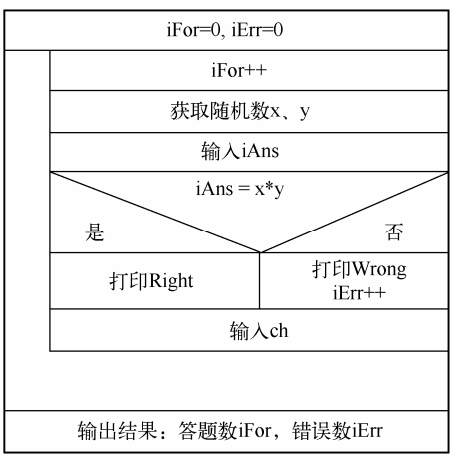


图 5.12　对乘法的估算

程序如下：

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   void main()
4   {   int iFor=0,iErr=0;
5       int x,y,iAns;
6       char ch;
7       do
8       {   iFor++;
9           x=rand()% 100;      /* 产生一个 0～100 间的随机数 */
10          y=rand()% 100;
11          printf("%d x%d=? ",x,y);
12          scanf("%d",&iAns);
13          if(iAns==x*y)   printf("Right!\n");
14          else
15          {   iErr++;
16              printf("You are wrong.\n");
17          }
18          printf("Are you want to continue?(Y/N)");
19          do ch=getchar();        /* 选择是否继续 */
20          while(ch!='y' && ch!='Y' && ch!='n' && ch!='N');
21      } while(ch=='y' || ch=='Y');        /* 回答 Yes，则继续 */
22      printf("In%d questions,",iFor);
23      printf("you've made%d right,%d wrong.\n",iFor-iErr,iErr);
24  }
```

程序中，iFor 表示题目的个数，iErr 表示答错的题目个数。iAns 代表用户输入的计算结果。第 9、10 行与随机数有关，读者可自行学习。第 18～20 行用于确定是否要继续做练习，第 21～23 行是对用户所做出的选择的响应——是否继续出题，如果选择停止做练习，则显示所做的题目总数、答对的题目数和答错的题目数。

5.4.2　break语句和continue语句

1. 格式

break 语句和 continue 语句的格式都比较简单。

break 语句的格式为：

```
break;
```

continue 语句的格式为：

```
continue;
```

2．功能

在 switch 语句中，break 语句用于跳过后面的语句，结束 switch 语句。在循环结构中（包括 while 语句、do-while 语句和 for 语句），break 语句的作用是跳出循环结构，执行循环后面的语句。

在下面的程序块中，由于 break 语句的作用，循环只进行了 1 次，printf 语句一次也没有执行，并且永远也不会被执行：

```
for(i=1;i<100;i++)
{   sum+=i;
    break;
    printf("%d",sum);
}
```

continue 语句只能用在循环结构中，用于结束本次循环，即跳过循环体中后面的语句，开始下一次循环。下面的语句块中，循环次数是 100 次，但 printf 语句一次也没有执行：

```
for(i=1;i<100;i++)
{   sum+=i;
    continue;
    printf("%d",sum);
}
```

显然，break 语句或 continue 语句在使用时最好放在 if 语句之后，其实，这也正是它们通常的用法。

值得注意的是，break 语句和 continue 语句的有效范围都仅限于所在的循环语句之内。如果想从嵌套的循环结构的内循环中直接跳出到外循环之外，使用一个 break 语句是不行的。可行的方法是：在程序中引入一个标识变量，在各循环的结束位置检测这一标识，用多个 break 语句做连续的跳转。例如：

```
flag=1                      /* 标识的初始化 */
while(i<10)
{   for(j=i;j<10;j++)
        if(j==2*i)
        {   flag=0;         /* 修改标识 */
            break;          /* 结束内循环 */
        }
    if(flag !=1)break;      /* 结束外循环 */
    printf("%d",i);
}
```

3．break语句和continue语句的应用

【例 5.16】　顺序打印 100～1000 之间所有 9 的倍数，如果一个数同时也是 7 的倍数则停止打印。

算法过程如图 5.13 所示。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int i;
4       for(i=100;i<1000;i++)
5       {   if(i%9 !=0)continue;
6           printf("%5d",i);
7           if(i%7==0)break;
8       }
9   }
```

第 5 行的作用是判断 i 是否是 9 的倍数，如果不是 9 的倍数即跳过第 6、7 两行。第 7 行判断是否是 7 的倍数，如果是 7 的倍数则结束循环，程序也就跟着结束了。

程序的运行结果如下：

```
108  117  126
```

【例 5.17】　求任意正整数的除了自身以外的最大因数，比如 24 的最大因数是 12。

分析：整数 n 的因数 r，是能整除 n 的整数，即 n % r 为 0。要求 n 的最大因数 r，可以从 r=n–1 开始顺序尝试所有小于 n 的正整数，求得的第 1 个因数就是最大因数。

算法如图 5.14 所示。

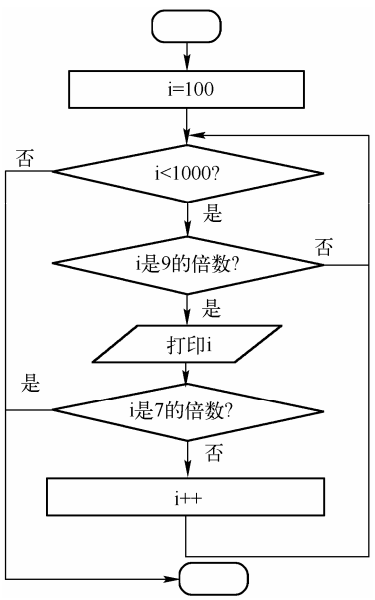


图 5.13　例 5.16 的程序执行过程

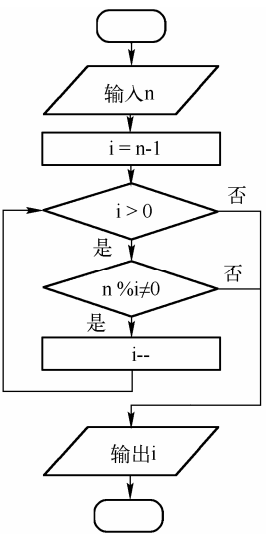


图 5.14　求自然数的最大因数

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int i,n;
```

```
4       printf("Enter a number:");
5       scanf("%d",&n);
6       for(i=n-1;i>0;i--)
7           if(n%i==0)break;
8       printf("The biggest factor of%d is%d\n",n,i);
9   }
```

*5.4.3　goto语句

goto 语句是从早期的程序设计语言遗留下来的一个语句，称为无条件跳转语句，也就是说可以任意跳转到程序中的其他位置。虽然 goto 语句使用灵活，但由于破坏了程序的结构化特征，已经逐渐被淘汰了。

goto 语句的使用格式是：

goto 语句标号;

“语句标号”是一个特殊的标识符，由字母、数字或下划线组成（第一个字符不能是数字），后面跟冒号“:”，放在语句行的最前头。例如：

```
Loop:   i=1;
        while(i<10)
Calc:   sum+=i;
```

其中，Loop 和 Calc 都是语句标号。

【例 5.18】　打印所有的 100～1000 之间的 9 的倍数。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int i=100;
4       while(i<1000)
5       {   if(i%9 !=0)goto Next;
6           printf("%5d",i);
7   Next:   i++;
8       }
9   }
```

【例 5.19】　用 goto 语句构建循环结构。

```
1   #include<stdio.h>
2   void main()
3   {   int i=1,sum=0;
4   loop:
5       sum+=i;
6       i++;
7       if(i<=10)goto loop;
```

```
8       printf("Sum=%d\n",sum);
9   }
```

5.5 应 用 举 例

利用现有的知识，我们已经可以解决一些简单的编程问题了。

【例 5.20】 阿米巴用简单分裂方式繁殖，每分裂一次要用 3 分钟。将若干个阿米巴放在一个盛满营养液的容器内，45 分钟后容器内充满了阿米巴。已知容器最多可以装阿米巴 2^{20} 个，试问，开始的时候往容器内放了多少个阿米巴?

分析：根据题意，阿米巴每 3 分钟分裂一次，那么从开始将阿米巴放入容器里面，到 45 分钟后充满容器，需要分裂 15 次。而“容器最多可以装阿米巴 2^{20} 个”，即阿米巴分裂 15 次以后得到的个数是 2^{20}。题目要求计算分裂之前的阿米巴数，不妨使用倒推的方法，从第 15 次分裂之后的 2^{20} 个，倒推出第 15 次分裂之前（即第 14 次分裂之后）的个数，再进一步倒推出第 14 次分裂之前、第 13 次分裂之前直至第 1 次分裂之前的个数。

设开始时阿米巴的个数为 x_0，第 1 次、第 2 次、第 3 次……第 15 次分裂之后的个数分别为 x_1，x_2，x_3，…，x_{15}，则有以下关系：

$x_0=x_1/2$，$x_1=x_2/2$，$x_2=x_3/2$，…，$x_{14}=x_{15}/2$

已知 $x_{15}=2^{20}=1048576$，可以利用上面的关系倒推出 x_0。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int  i;
4       long x=1048576;     /* 用长整数表示 2M */
5       for(i=15;i>0;i--)x=x/2;
6       printf("There are%d amebas.\n" ,x);
7   }
```

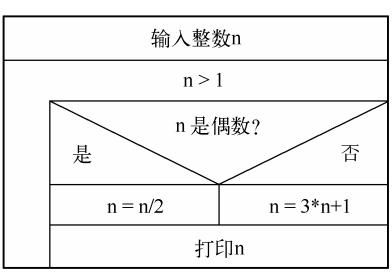


图 5.15　谷角猜想

【例 5.21】 日本数学家谷角静夫在研究自然数时发现了一个奇怪的现象：对于任意一个自然数 n，若 n 为偶数，则将其除以 2；若 n 为奇数，则将其乘以 3，然后再加 1。如此反复，经过有限次运算后，总可以得到自然数 1。人们把这一发现叫做“谷角猜想”，试编程验证之。

算法如图 5.15 所示。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int n;
4       printf("Enter a number:");
5       scanf("%d",&n);
6       printf("%5d",n);
```

```
7       while(n>1)
8       {   if(n%2==0)n/=2;
9           else n=3*n+1;
10          printf("%5d",n);
11      }
12  }
```

【例 5.22】　打印九九乘法口诀。

分析：九九乘法口诀表呈三角形，共有 9 行。第 1 行有 1 列，第 2 行有 2 列，……，第 9 行有 9 列。打印方法是，用外循环控制行数，包括打印一行中的所有列和换行，在内循环中，打印某一行上的所有列。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int i,j;
4       for(i=1;i<=9;i++)
5       {   for(j=1;j<=i;j++)            /* 列数与行数相关 */
6               printf("%1dx%1d=%-4d",j,i,j*i);/* 每个算式占 8 位，最
                                                  后一行占 72 位 */
7           printf("\n");
8       }
9   }
```

在本程序的设计中，输出格式控制是值得思考的。在微机上，用 printf 函数输出时，屏幕上可显示 25 行 80 列字符，也就是说每行最多有 80 个字符。在程序的第 6 行中，通过格式控制可以将每个算式的输出控制在 8 个字符内，避免了不必要的换行。为了使得输出的算式更紧凑，程序中数据的输出使用了左对齐。

程序的运行结果如下：

```
1x1=1
1x2=2   2x2=4
1x3=3   2x3=6   3x3=9
1x4=4   2x4=8   3x4=12  4x4=16
1x5=5   2x5=10  3x5=15  4x5=20  5x5=25
1x6=6   2x6=12  3x6=18  4x6=24  5x6=30  6x6=36
1x7=7   2x7=14  3x7=21  4x7=28  5x7=35  6x7=42  7x7=49
1x8=8   2x8=16  3x8=24  4x8=32  5x8=40  6x8=48  7x8=56  8x8=64
1x9=9   2x9=18  3x9=27  4x9=36  5x9=45  6x9=54  7x9=63  8x9=72  9x9=81
```

【例 5.23】　利用迭代公式 $x_{n+1}=\frac{1}{2}\left(x_n+\frac{a}{x_n}\right)$ 求 $x=\sqrt{a}$ 的近似值，结果保留 5 位有效数字。

分析：用该迭代公式求 a 的平方根 x，实际是求近似值 x_{n+1}，即 $x\approx x_{n+1}$，“保留 5 位有效数字”可理解为$|x_{n+1}-x_n|/\ x_{n+1}<10^{-5}$。C 语言中，求绝对值可用 fabs 函数。fabs 是一个数学函

数，其他数学函数还包括 log 函数、exp 函数、sin 函数等，在程序中若出现数学函数，则需要在程序开始加入独立的 1 行：#include <math.h>。

程序如下：

```
1   #include<math.h>
2   #include<stdio.h>
3   void main()
4   {   float x,y,y0;
5       printf("Enter a positive:");
6       do scanf("%f",&x);          /* 要求输入的是正数 */
7       while(x<0);
8       y=1;                        /* 初始值可以任意选择一个正数 */
9       do
10      {   y0=y;
11          y=0.5*(y+x/y);
12      } while(fabs(y-y0)/y>1e-5); /* fabs 函数的定义在 math.h 中*/
13      printf("Square root of%f is%f\n",x,y);
14  }
```

【例 5.24】　利用牛顿迭代法求方程的 $5x^3-3x^2-22=0$ 在 2 附近的一个实根。

分析：牛顿法是把非线性方程线性化的一种近似方法，如图 5.16 所示。

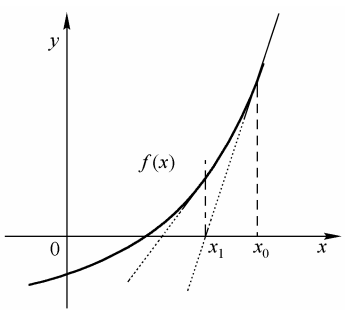


图 5.16　牛顿迭代法求方程的根

对于非线性方程 $f(x)=0$，把 $f(x)$在 x_0 点附近展开成泰勒级数：

$$f(x) = f(x_0) + (x-x_0)f'(x_0) + (x-x_0)^2 f''(x_0) + \cdots$$

取其线性部分，作为非线性方程 $f(x) = 0$ 的近似方程，则有：

$$f(x_0) + f'(x_0)(x-x_0) = 0$$

设 $f'(x_0) \neq 0$，则方程的近似解为：

$$x_1=x_0-f(x_0)/f'(x_0)$$

再把 $f(x)$在 x_1 附近展开成泰勒级数，也取其线性部分作为 $f(x)=0$ 的近似方程。若 $f(x_1)\neq 0$，则得到方程的另一个近似解：

$$x_2=x_1-f(x_1)/f'(x_1)$$

这样，即可得到牛顿法的一个迭代序列：

$$x_{n+1}=x_n-f(x_n)/f'(x_n)$$

当 $f(x_{n+1})\approx 0$ 时即可认为 x_{n+1} 就是方程 $f(x)=0$ 的近似解。

对于本题：

$f(x) = 5x^3-3x^2-22$

$f'(x) = 15x^2-6x$

$x_0 = 2$

程序如下：

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

```
3   {   float x,y,y1;
4       x=2;
5       do
6       {   y=x*x*(5*x-3)-22;       /* 计算 f(x)*/
7           y1=(15*x-6)*x;          /* 计算 f '(x)*/
8           x=x-y/y1;
9       } while(fabs(y)>1e-5)       /* 计算精度为 0.00001 */
10      printf("The root is%f\n",x);
11  }
```

5.6　程 序 调 试

对于程序设计而言，即使再优秀的程序员也不能保证不会犯错误，而他比别人高明的地方在于少犯错误。一个优秀的程序，不在于使用了先进的算法，而在于仅仅包含了少量的错误。

初学程序设计的人有必要从一开始就养成良好的习惯，培养严谨的工作作风，并逐步掌握一些编程技巧，尽量减少犯错误的机会。

程序中的错误是在所难免的，关键是发现并纠正错误。对于复杂的程序，查错和纠错都是非常困难的，这是程序设计中的一个难题。下面介绍一些基本的应对策略，以及在 Visual C++ 6.0 中查错、纠错的一些技巧。

5.6.1　程序调试的一般策略

所谓调试程序就是查找程序中的错误并纠正这些错误。

程序中的错误一般可分为三类：语法错误、逻辑错误和设计错误。语法错误比较容易发现。通常，有语法错误的程序不能通过编译和连接，也就不能生成可执行的程序。逻辑错误又叫语义错误，也就是不能正确地表达所需要的功能，是较常见的错误之一。其外部表现为，程序可以运行，但有时出错，有时又不出错。逻辑错误通常比较难以被发现，逻辑错误的查错和纠错对任何程序员来说都是挑战。设计错误比较少见，通常是由于对问题的分析不彻底造成的，纠正这类错误需要重新设计程序。

调试程序的一般步骤如下。

（1）静态检查

静态检查也就是人工检查，是在完成程序设计后，在上机调试前，仔细地对程序代码进行全面的检查。虽然利用计算机可以提高工作效率，但作为优秀的程序员，不能养成过分依赖于计算机的坏习惯。一个对工作高度负责的程序员，不会把问题留给后面的工作。

通过静态检查，不仅可以发现程序中的语法错误，也可以发现逻辑错误，甚至发现设计上的缺陷。随着编程经验的积累，程序员所掌握的技巧会越来越多，解决问题的能力也会不断提高，查错和纠错能力也会相应地增强。

为了便于查错，编写程序时应该力求做到程序编码的标准化、文档化，以增强编码的可读性、可理解性、可维护性。要做到这一点，应该力求做到：采用结构化方法，划分功能模

块和程序段，采用必要的缩进和对齐，简化表达式，每行只有一个语句，尽量使用注释，使用有意义的标识符，等等。

（2）动态检查

动态检查是指通过上机调试发现错误的过程。完成编码后，可以借助于编译程序检查隐藏的语法错误，如错误的标识符、非法的表达式、错误的函数调用等；利用连接程序，可以检查是否存在连接错误，如调用了未定义的函数、缺乏必要的函数定义等；通过试运行程序，可以发现一些逻辑错误，如错误的计算、有问题的输入和输出等。

初学者总是发现自己的程序中存在较多的错误，而且随着程序越来越长，出错的概率也越来越大。但似乎没必要因此而灰心丧气，只要逐一纠正这些错误就可以了，当然在纠错的时候要注意总结，通过不断总结来提高编程能力，以免再犯同样的错误。

在上机调试时，程序中的错误通常有两种形式，即“错误”（Error）和“警告”（Warning）。“错误”是必须要纠正的，任何一个“错误”都会导致编译或连接失败。“警告”有时是可以忽略的，一般不会影响生成可执行的“程序”，但有“警告”错误的程序是有缺陷的，多数时候会导致运行错误。程序员不仅要纠正致命“错误”，也要纠正“警告”错误。如果认为一个“警告”不会造成任何实质的损害，也可以不去理会。

当程序中的错误较多时，应该从最前面的错误开始逐一改正。如果前面的错误还没有修正，先不要试图纠正后面的错误。有些错误实际出自于同样的原因，因而只要纠正了其中的一个就会消除所有这些错误。

例如，对于下面的程序：

```
void main()
{   int a=5,b;
    b=a++;
    printf(%d\n",a,b);
}
```

在编译时会显示多达 11 条错误：

```
error C2143:syntax error:missing ')' before '%'
error C2198:'printf':too few actual parameters
error C2017:illegal escape sequence
error C2065:'d':undeclared identifier
error C2146:syntax error:missing ';' before identifier 'n'
warning C4552:'%':operator has no effect;expected operator with
side-effect
error C2001:newline in constant
error C2065:'n':undeclared identifier
error C2143:syntax error:missing ';' before 'string'
error C2143:syntax error:missing ';' before '}'
Error executing cl.exe.
```

根据第一条错误信息（“%”前缺少“)”），可以基本确定问题出现在程序的第 4 行，在“%”的前边。这时不难发现，只是遗漏了一个引号“"”。修正了这一错误，再编译时发现所有错误都已经不见了。

可见，编译系统给出的错误信息并不确切，只能作为查错时的参考，如果教条地去使用这些信息，非要找到“指定”的错误，则很可能是徒劳的。

5.6.2　程序的跟踪与调试

程序通过了编译和连接，并不意味着就没有错误了。程序中隐藏了逻辑错误或设计错误，一般不影响编译和连接，但却会给程序的运行带来伤害。例如，对于以下程序：

```
1   void  main()
2   {   int a,b,max;
3       scanf("%d%d",&a,&b);
4       if(a>b)max=a;
5       else max=a;
6       printf("max=%d\n",max);
7   }
```

程序的功能是求两个数中的较大数，编译连接都正常，没有发现任何错误。运行时输入 5 和 3，结果也“正常”。是否就可以了呢？如果输入的还是这两个数，但顺序颠倒过来，即先输入 3 后输入 5，结果就出错了！程序并不复杂，语句不多，算法也很简单，很容易就可以找到问题的所在。第 5 行的“max=a”应改为“max=b”。

有没有办法帮助我们去发现类似的错误呢？下面介绍在 Visual C++ 6.0 中调试程序的方法。

1．程序的“调试”运行方式

打开 Visual C++ 6.0 的主菜单（如图 5.17 所示），会发现运行程序的两种方法：一种是 Execute…（快捷键是 Ctrl+F5，工具栏按钮为 ），另一种是 Go（快捷键是 F5，工具栏按钮是 ）。Execute…方法是直接运行程序的最简单的方式，对于已经确认没有错误的程序，推荐使用这一方法。但在调试程序时，需要使用“Go”方法，也就是说使程序运行在“调试”（Debug）状态。

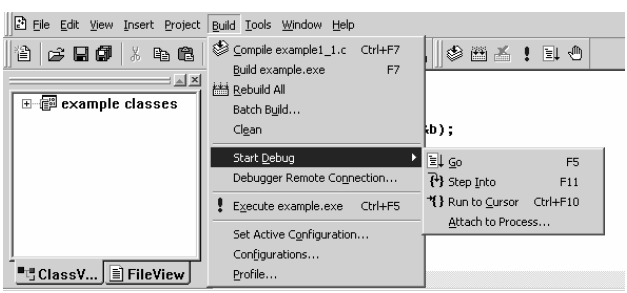


图 5.17　Visual C++ 6.0 的“Build”菜单

程序在“调试”方式下运行，其外部表现并没有不同，不同的地方发生在系统内部。此时，可以暂时中断程序运行，以观察程序的运行状态，比如运行到了什么地方，变量的值现在是多少，等等，甚至可以临时修改数据以干预程序的执行过程。暂停的程序，可以恢复运行，直到程序结束。

2. 使程序暂停的方法

要调试程序，就需要暂时中断（暂停）程序的执行，有以下 3 种方法。

（1）执行到光标处

在源码编辑状态，将光标放在一个语句行上，单击 Debug 工具栏（如图 5.18 所示）中的 Run to Cursor 按钮，程序将进入调试运行状态，当程序运行到光标处时，进入暂停状态。

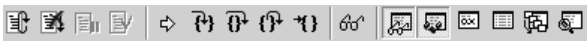


图 5.18 “Debug”工具栏

（2）设置断点

将光标放在一个语句行上，然后单击工具栏上的 Add/Remove Breakpoint 按钮，即可在此设置（或取消）一个断点（在该行的右端将显示一个暗红色的实心圆）。然后，单击 Go 按钮启动程序，当程序运行到某断点时，被自动暂停，此时的屏幕显示如图 5.19 所示。此时，工作区分为 4 个部分，最上面是源代码区，中间靠左是变量（Variables）区，中间靠右是观察（Watch）区，下面是输出（Output）区。在代码区的断点处，左边显示一个黄色的右箭头，表示程序恢复执行时将从该语句处继续。

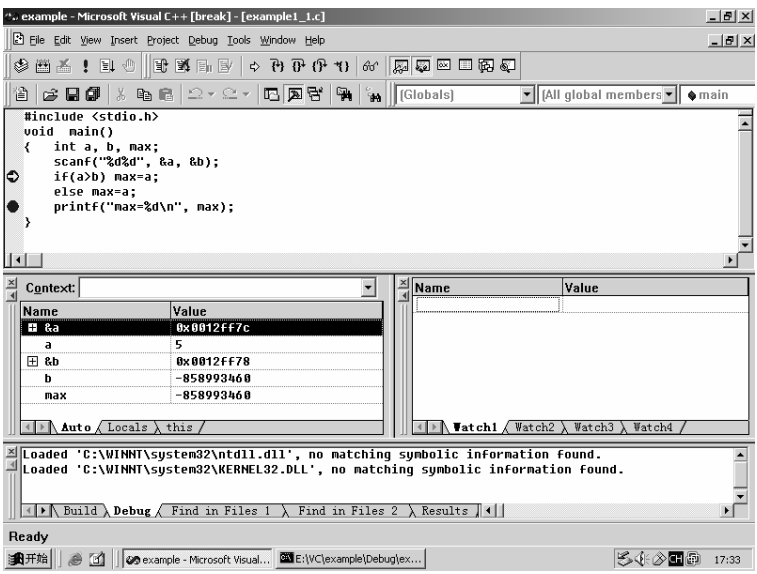


图 5.19　程序暂停运行时，Visual C++ 6.0 的主窗口

（3）中断（Break）方式

当程序运行在调试方式下时，单击 Visual C++ 6.0 工具栏上的 Break 按钮（位于 Debug 工具栏中），即可立即“中断”程序的运行。但与前两种方法不同的是，这种方法一般显示程序的汇编代码，而不是源程序。因此，并不建议初学者使用。

要恢复暂停程序的执行，只要单击 Go 按钮即可。

3. 在暂停状态下常见的操作

程序被暂停后，也就是当出现类似于图 5.19 所示的屏幕显示时，就可以使用 Visual C++ 6.0 的调试程序工具了。

① 单步运行程序。单击 Debug 工具栏的 Step Over 按钮或 Step Into 按钮即可一行一行地执行程序，每执行完一行都自动进入暂停状态。

② 观察变量的值。使用 Variables 窗口可观察程序中的各种变量的当前值，单击其中的“Locals”选项卡可以显示所定义的变量的值。在 Watch 窗口中，也可以添加或删除要观察的变量的值。当 Watch 有效时，单击选中源代码窗口，把鼠标指向任何变量，稍候片刻，就会弹出该变量的值。

③ 修改变量的值。要临时修改变量的值，可以双击改变量的 Value 域（在 Variables 窗口或 Watch 窗口都有效）进行编辑。

④ 模拟计算。单击 Debug 工具栏的 QuickWatch 按钮，弹出 QuickWatch 对话框，可以用当前程序中的变量组成算式进行简单的模拟运算，如图 5.20 所示。

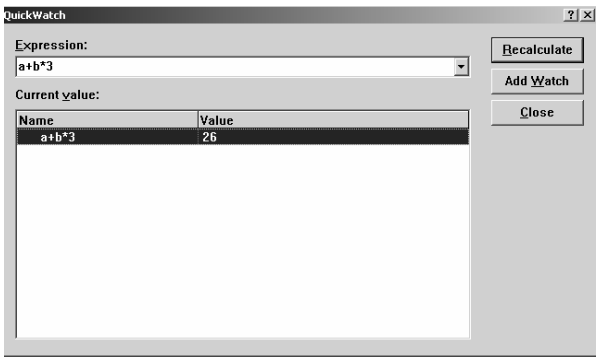


图 5.20　Visual C++ 6.0 的“QuickWatch”对话框

4．查错的技巧

程序中隐藏的错误通常难以发现，可以采取一些技巧。

① 在程序中加入一些“调试代码”，输出变量的中间值以帮助判断。

例如：

```
scanf("%d,%d",&a,&b);
printf("%d,%d ",a,b);
```

这里的 printf 语句可能不是程序所必需的，但可以用来帮助判断输入数据是否有错，比如输入的数据是用空白分隔的，b 的值将出现异常（运行程序时，没有提示，用户只能凭感觉操作）。

但这种办法并不理想，一来使用起来不便，二来破坏了程序的原有结构，三是在不需要的时候还要一一删除，不注意会遗漏。

使用条件编译会更好些，参见有关书籍。

② 通过设置断点、跟踪运行等方式调试程序，这是比较高效的方法。

一旦怀疑某段程序有错，可以在程序段的开始位置设置断点，当程序运行到断点而暂停时，通过一步一步地执行程序，并观察变量的变化，判断错误是否存在。

但这种方法也有不利的地方。比如，如果程序段所表达的逻辑比较复杂，或者循环次数很多，都会导致调试困难。

③ 程序的单元测试法。

对程序段的执行逻辑进行仔细分析，划分程序执行路径的种类，针对每种路径设计若干组不同的输入数据，依次使用这些数据来测试程序，观察程序的运行状态和结果，以发现可能隐藏着的错误。

【例 5.25】 对于任意的 *m*（*m*>0）个实数，求介于 *a* 和 *b*（*a*<*b*）之间所有实数的个数 *n* 和平均数 ave。注意 *n* 为 0 的情况。

程序的 N-S 图如图 5.21 所示。

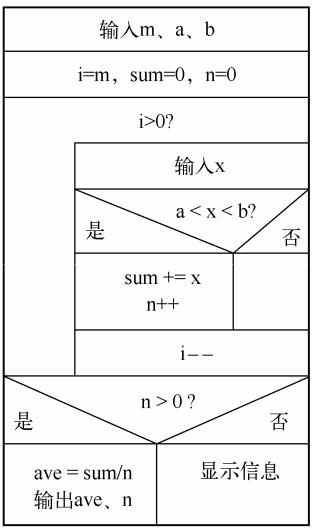


图 5.21　例 5.25 的算法 N-S 图

程序如下：

```
#include<stdio.h>
#include<stdlib.h>
void main()
{   float a,b,x;
    float ave=0;
    int m,n=0;

    printf("请输入实数个数？");
    scanf("%d",&m);
    if(m<=0)
    {   printf("个数无效！\n");
        exit(0);
    }

    printf("请输入有效范围的上下界（格式为:a,b）？");
    scanf("%f,%f",&a,&b);
    if(a>=b)
    {   printf("范围无效！\n");
```

```
19          exit(0);
20      }
21
22      printf("依次输入%d 个实数（用空白分开）: \n",m);
23      while(m>0)
24      {   scanf("%f",&x);
25          if(x>a && x<b)
26          {   ave+=x;
27              n++;
28          }
29          m--;
30      }
31
32      if(!n)printf("没有有效的数字。\n");
33      else printf("共有%d 个有效数字，平均值为%f\n",n,ave/n);
34  }
```

程序的输入数据包括 m、a、b、x（m 个），可能的分组包括：

① 把 m 的取值分为 m＞0 和 m≯0 两组；

② 把 a、b 的取值也分为两组 a＜b 和 a≮b；

③ 把 x 的值可分为三种情况全部 x 都有效、全部 x 都无效、部分 x 有效。

经过以上分析，不难设计出几种测试数据。请大家自己尝试。

习　题　5

5.1　找出程序中的所有错误并改正。

（1）求若干个正数的平均值，若输入的是负数则结束输入。

```
1   #include<stdio.h>
2   void main()
3   {   int n;
4       float x,ave;
5       do
6       {   scanf("%f",&x);
7           ave+=x;
8           n++;
9       } while(x<0);
10      printf("Average:%f\n",ave);
11  }
```

（2）对于 100～199 之间的每个自然数，打印它们的所有因数，比如 124 的因数为 1、2、4、31、62、124。

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

```
3   {   int i,j;
4       for(i=100;i<199;i++);
5       {   printf("%d:");
6           for(j=100;j<199;j++)
7               if(i%j=0)printf("%d ",j);
8           printf("\n");
9       }
10  }
```

5.2　在空白处填上适当内容使程序变得完整。

（1）计算 n!。

```
1   #include<stdio.h>
2   void main()
3   {   int i,n;
4       ____①____
5       printf("Enter a number:");
6       scanf("%d",&n);
7       if(n<0)printf("Error number.\n");
8       else
9       {   if(____②____)f=1;
10          else
11              for(____③____)f=f*i;
12          printf("%d!=%.0f\n",n,f);
13      }
14  }
```

（2）计算 $\ln(1+x)=x-\frac{x^2}{2}+\frac{x^3}{3}-\frac{x^4}{4}+\mathrm{L}\ +(-1)^{n-1}\frac{x^n}{n}+\mathrm{L}$　　$(-1<x\leqslant 1)$，精确到 n=10。

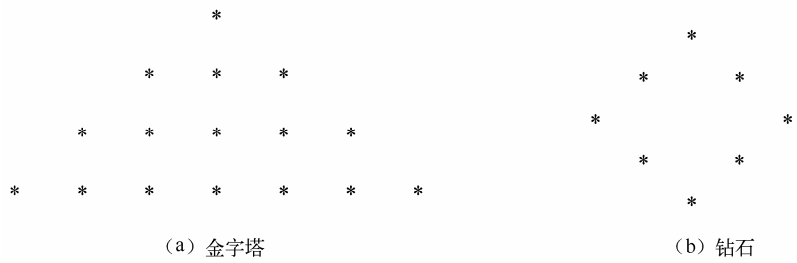
```
1   #include<stdio.h>
2   #define N 10
3   void main()
4   {   ____①____
5       float  x,sum=0,t=1;
6       printf("Enter a digit(-1～1):");
7       do  scanf("%f",&x);
8       while(____②____);
9       do
10      {   ____③____;
11          ____④____;
12          sum+=f*t/i;
13          i++;
14      } while(____⑤____);
15      printf("ln(%f)=%f\n",____⑥____,sum);
16  }
```

5.3　求 10 个数的最大数和最小数。

5.4　某养兔场引进一只刚出生的新品种兔子，这种兔子从出生的下一个月开始，每月新生一只兔子，新生的兔子也如此繁殖。如果所有的兔子都不死去，问到第 12 个月时，该养兔场共有这种兔子多少只?

5.5　相传，古印度的舍罕王打算重赏国际象棋的发明者——宰相西萨·班·达依尔。于是，这位宰相跪在国王面前说：“陛下，请您在这张棋盘的第一个小格内，赏给我一粒麦子；在第二个小格内给两粒，第三格内给 4 粒，照这样下去，每一小格都比前一小格加一倍。陛下啊，把这样摆满棋盘上所有 64 格的麦粒，都赏给您的仆人罢!”国王慷慨地答应了宰相的要求，却发现无法兑现他许下的诺言。你知道，聪明的宰相到底要求的是多少麦粒呢? 请编程求解。

5.6　打印如图 5.22 所示的几何图形：



（a）金字塔　　（b）钻石

图 5.22　几何图形

5.7　《张邱建算经》(公元 5 世纪) 内的“百鸡问题”：“今有鸡翁一，直钱五；鸡母一，直钱三；鸡雏三，直钱一。凡百钱，买鸡百只，问鸡翁、母、雏各几何? ”

5.8　计算 $e=1+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\ldots$，要求精确到小数点后 5 位。请编程求解。

5.9　用辗转相除法求任意两个正整数 m 和 n 的最大公约数。请编程求解。

提示：辗转相除法又叫做欧几里得算法，可以较快地求出两个自然数的最大公约数 gcd，在数值很大时尤其有用。其基本过程是：如果 q 和 r 分别是 m 除以 n 的商及余数，即 $m=n\cdot q+r$，则 gcd(m, n)=gcd(n, r)。

5.10　打印 100～1000 之间的所有素数。

提示：素数 n 是只能被 1 和自身 n 整除的自然数，例如 17 是素数，而 21 不是素数。

5.11　打印所有的“水仙花数”。所谓“水仙花数”是指一个 3 位数，它的每位数字的立方和等于该数本身，例如 153 是一个“水仙花数”，因为 $153=1^3+5^3+3^3$。

5.12　用二分法求方程 $\ln x+2x-3=0$ 的近似解。

提示：

二分法的求解过程如下：

① 选择一个求解区间(a, b)，使得 $f(a)\cdot f(b)<0$；

② 取 $x=(a+b)/2$，若 $f(x)\approx 0$ 则 x 为近似解，输出 x，程序结束；

③ 若 $f(x)\cdot f(a)<0$ 则令 $b=x$，否则令 $a=x$，转到②继续求解。

第 6 章　数　　组

学习目标

通过本章学习，应该掌握：

- 数组的基本概念
- 一维数组的概念和应用
- 多维数组的有关概念、存储方式
- 查找、排序等常见算法的编程

6.1　数组的概念

6.1.1　为什么要使用数组

先来看一个例子，某班有 3 个学生选修了某门课程，要计算他们的平均成绩并打印出所有低于 80 分的成绩，源程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int score1,score2,score3;
4       float ave;
5       scanf("%d%d%d",&score1,&score2,&score3);/*顺序输入 3 个成绩*/
6       ave=(score1+score2+score3)/3.0;          /*计算平均成绩*/
7       printf("Average is %.1f\n",ave);         /*打印平均成绩*/
8       printf("Less than 80:");                 /*打印低于 80 分的成绩*/
9       if(score1<80)printf("%d",score1);        /*判断每个成绩是否低
                                                 于 80 分*/
10      if(score2<80)printf("%d",score2);
11      if(score3<80)printf("%d",score3);
12  }
```

可以设想，如果学生数不是 3，而是 30，甚至更多，按这个方法编写的程序是“拙劣”的。也许，有的读者会将这个程序分为两部分，一部分计算平均成绩，另一部分判断并输出低于 80 分的成绩。

程序一：求平均成绩。

```
1   #include<stdio.h>
2   void main()
3   {   int  i,n,score;
```

```
4       float  ave=0;
5       scanf("%d",&n);              /*输入学生人数*/
6       for(i=0;i<n;i++)             /*依次输入每个成绩，并求和*/
7       {   scanf("%d",&score);
8           ave+=score;
9       }
10      ave=ave/n;                   /*计算平均成绩*/
11      printf("Average is %.1f\n",ave);
12  }
```

程序二：打印低于 80 分的成绩。

```
1   #include<stdio.h>
2   void main()
3   {   int  i,n,score;
4       scanf("%d",&n);                       /*输入学生人数*/
5       for(i=0;i<n;i++)
6       {   scanf("%d",&score);               /*依次输入所有的成绩*/
7           if(score<80)printf("%d",score); /*判断每个成绩是否低于 80 分*/
8       }
9   }
```

对于程序二的输入，读者可能已经发现，必须一次完成所有成绩的输入（按回车键前输入所有数据），否则将导致输出结果混乱。但这还不是关键问题，根据题目要求，需要在一个程序中完成两个任务，也就是说，必须设法将两个程序合并为一个程序。

也许，有人会这样编写程序：

```
1   #include<stdio.h>
2   void main()
3   {   int  i,n,score;
4       float  ave=0;
5       scanf("%d",&n);              /*输入学生人数*/
6       for(i=0;i<n;i++)             /*依次输入每个人的成绩，并求和*/
7       {   scanf("%d",&score);
8           ave+=score;
9       }
10      ave=ave/n;                   /*计算平均成绩*/
11      printf("Average is %.1f\n",ave);
12
13      printf("Less than 80:");
14      for(i=0;i<n;i++)
15      {   scanf("%d",&score);      /*依次输入所有人的成绩*/
16          if(score<80)printf("%d",score); /*判断每个成绩是否低于 80 分*/
```

```
17      }
18  }
```

问题是不是解决了？没有！因为运行程序时，不得不两次输入相同的数据。如果学生人数众多，对程序的使用者而言将是灾难性的。当然，可以将两部分程序的次序颠倒过来，从而避免第二次输入。但是，这样也不能彻底解决问题。试想，假设不是打印低于 80 分的成绩，而是打印低于平均分的成绩，那么这两部分程序还能颠倒过来吗？

有的读者可能已经想到了解决的办法，参考本节开始时的第一个源程序，如果可以将输入的所有数据都存放在内存中，程序中就可以反复使用而不必再输入一次了。问题是如何存放和处理这么多数据呢？问题的答案是利用数组！

6.1.2　什么是数组

数组是程序设计语言的一个重要概念，是程序中最常见的数据结构之一。简单地讲，数组是一组数据的集合。比如，所有同学的考试成绩、为测量某钢球的直径用卡尺测得的一组数据、全国各主要城市的最高温度等，都是数组数据。当然，数组数据不仅仅限于数值型数据，也可以是其他类型的：全国各省会城市的名称可以组成字符串型的数组，全班同学的生日也可以组成日期型的数组。如果把每个人的信息（包括姓名、年龄、性别等）看做一条数据记录，那么全班每个同学的个人信息可以组成一个记录型的数组。

数组是一组相同类型的数据组成的有限集合。数组中的数据称为**数组元素**。数组元素的个数称为**数组长度**，因此具有 n 个元素的数组的长度为 n。数组中的元素具有先后次序，通常把最前边的元素称为第 1 个元素，然后依次是第 2、第 3、……第 n 个元素。如果用某个标识符 α 表示数组的名称，则该数组的第 i（$1 \leqslant i \leqslant n$）个元素可用数组名 α 和下标 i 表示为 α_i。如果数组元素的下标只有 1 个，则这样的数组是一维数组，如果有 2 个下标则是二维数组，依次类推。数组元素的下标的个数称为数组的**维数**。习惯上将维数大于 2 的数组统称为多维数组。

与其他程序设计语言一样，C 语言中，数组名是数组的唯一标识符。数组元素用数组名和元素下标表示。数组中的所有元素具有相同的数据类型，元素的数据类型常被称为**数组类型**。在计算机内存中，同一数组的所有元素按下标顺序依次存放在相邻的存储单元中，因此数组占据的内存空间大小可由数组长度和数组类型计算出来。

将数组和循环结构结合起来，可以大幅度地提高处理大规模数据的效率，可以说数组离不开循环控制。

6.2　一 维 数 组

一维数组与数学中的向量对应，是 C 语言程序中最简单的数组，但也是最常见的数组形式之一。

6.2.1　一维数组的定义和引用

1．数组的定义

定义一维数组的一般方法是：

类型说明　数组名称[数组长度]

其中，“数组名称”是一个标准的 C 语言标识符，“数组长度”是一个整型常量（常数）或常量表达式。“类型说明”给出了数组类型，即数组元素的数据类型，可以是任意合法的 C 语言类型，如 int、float、char 等，也可以是用户自定义的数据类型。

C 语言中，定义数组时必须给出明确的数组类型说明和数组长度。

例如：

```
int  a[5],b[10];
```

定义了两个整型数组 a 和 b，它们的长度分别为 5 和 10，即分别具有 5 个和 10 个数组元素：

再如：

```
#define  N 10
int  student[3*N];
```

定义了一个包含 30 个元素的整型数组，这里 N 为符号常量。

一个初学者容易犯的错误是试图定义可变长度的数组，例如：

```
1       int  n;
2       scanf("%d",&n);
3       int  data[n];
```

其想法是，在运行程序时再输入元素个数，但却发现这段程序不能通过编译。为什么呢？除了这里所讲的 C 语言不允许定义可变长度数组（数组长度为变量或含变量的表达式）的语法规则外，程序中还有另一个语法错误：第 2 行是一个可执行语句，而第 3 行的数组定义属于程序说明。按照 C 语言的规定，程序说明必须位于可执行语句的前面！当然，交换第 2 行和第 3 行的顺序也不能解决问题。

不难理解，若定义

```
int n=10,data[n];
```

也是错误的，因为 n 是变量，变量的初始化可以认为是语句执行过程。

同样，下面的程序段仍然是错误的：

```
const int n=10;
int  data[n];
```

有关 const 的知识请读者自己查阅有关书籍。

2. 数组的引用

首先要说明的是，C 语言中，不允许直接引用数组本身进行某种运算，只能通过引用数组的元素间接引用数组。

数组元素的表示方法如下：

数组名称[下标]

“下标”是一个整数，代表元素的位置。数组元素的“下标”可以是常量，也可以是变量或表达式。如果是表达式，则先计算出表达式的值，再转换为整型值。

值得注意的是，C 语言的数组元素的下标是从 0 开始的，a[0]是数组 a 的第 1 个元素，a[1]是第 2 个元素，……，a[i−1]是第 i 个元素。如果数组 a 的长度为 n，那么最后一个元素为 a[n−1]。也就是说，数组元素的下标的取值范围是 0～n−1。

在计算机内存中，一维数组的诸元素按照先后次序依次存放在相邻的存储单元中，占据连续的存储空间。若数组 a 定义为 int a[5]，则 a 的元素在内存中的位置关系如图 6.1 所示。

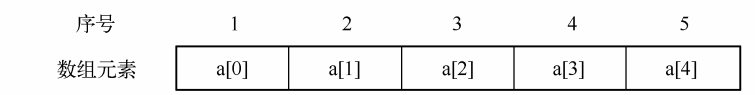


图 6.1　内存中的数组元素关系

与变量相同，可以将数组元素放在表达式中参与运算，也可用赋值运算符给数组元素赋值，例如：

```
a[1]+a[2]*b[i+1]-3
```

这里 i 是一个变量，a[1]、a[2]、b[i+1]是数组元素。

【例 6.1】　输入任意 5 个整数，输出它们的和并打印出这些数。

分析：该程序的算法如图 6.2 所示。

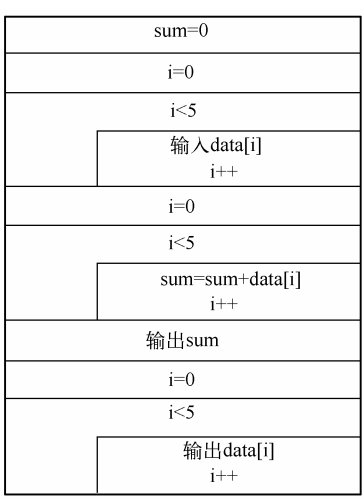


图 6.2　求 5 个数的和

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int  i,sum=0;
4       int  data[5];                  /*定义数组*/
6       for(i=0;i<5;i++)               /*依次输入 5 个数*/
7           scanf("%d",&data[i]);      /*输入数组元素的值*/
8       for(i=0;i<5;i++)               /*求和*/
9           sum=sum+data[i];           /*引用数组元素*/
10      printf("Sum=%d\n",sum);
11      printf("List is:");
12      for(i=0;i<5;i++)               /*打印出 5 个数*/
13          printf("%d",data[i]);      /*引用数组元素*/
14      printf("\n");
15  }
```

程序分为三部分，第 6、7 两行为第一部分，用来输入数据；第 8、9 两行为第二部分，用来计算 5 个数的和；第 10～14 行为第三部分，用来输出程序运行结果。这种将程序按功能分块的方法正是结构化程序设计所倡导的，对于设计较复杂的程序是有益的。有的读者可能会将第 6～9 行的两个循环合并起来，形成以下程序段：

```
for(i=0;i<5;i++)
{   scanf("%d",&data[i]);
    sum=sum+data[i];
}
```

即边输入边做累加，从而减少了程序行数。但由于将程序块的功能复杂化了，降低了程序的可读性。当需要编写较复杂的程序时，这种不良习惯将会造成危害。

【例 6.2】　假设数组 a 中已有 5 个数，要在第 1 个数的前面插入一个数 x，并保持这 5 个数的前后关系不变，试编程实现。

分析：题目中，虽然没明确说明从键盘输入的数组 a 中已有的 5 个数，但显然这一过程是必要的。题目的关键是，如何实现数组中数据的插入。根据定义，数组中的数据是从第 1 个元素开始依次按顺序存放的，也就是说，如果数组中已有 5 个数，那么这 5 个数之间，也包括它们的前面，是没有空的位置的。要插入一个数，就必须把其他数移走。只把第 1 个数移走是不够的，因为移走第 1 个数后，虽然可以把 x 放下了，但肯定改变了第 1 个元素与第 2 个元素的位置关系。因此，必须移动所有这 5 个数才可以维持原来的顺序。图 6.3 描述了将数组中的数据依次向后移动一位的过程。

程序如下：

```
1   #include<stdio.h>
2   #define N 5
3   void main()
4   {   int  i,x,a[N+1];                  /*数组长度应为 6*/
```

```
5       printf("Enter %d numbers:",N);
6       for(i=0;i<N;i++)scanf("%d",&a[i]);  /*只输入 5 个数，保留
                                              最后一个为空*/
7       printf("Before insert:");
8       for(i=0;i<N;i++)printf("%d",a[i]);  /*打印 5 个数*/
9       printf("\n");
10      printf("Enter a number?");
11      scanf("%d",&x);                     /*输入 x*/
12      for(i=N;i>0;i--)a[i]=a[i-1];        /*将各数依次后移一位，第 1 位将空
                                              出来*/
13      a[i]=x;                             /*循环结束后把 x 存入元素 a[0]*/
14      printf("After insert:");
15      for(i=0;i<N+1;i++)printf("%d",a[i]);    /*打印数组中的 6 个数*/
16      printf("\n");
17  }
```

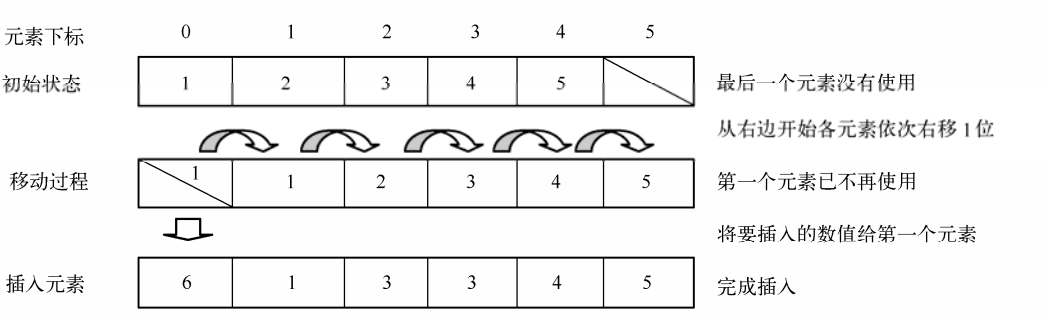


图 6.3　在数组中插入元素的过程

该程序的算法如图 6.4 所示。

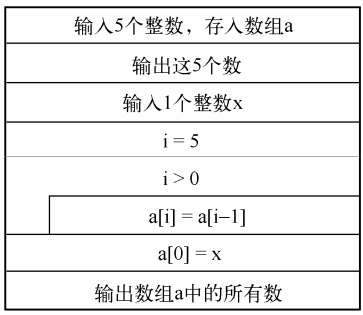


图 6.4　在数组中插入一个数

需要特别强调的是，C 语言对数组元素的下标不进行越界检查，也就是不检查数组元素的有效性，因此尽管定义数组 a 只有 5 个元素（意味着只有 a[0]，a[1]，…，a[4]这 5 个元素是有效的），但程序中出现 a[−1]、a[5]等“元素”仍然是“允许”的。读者不妨把源程序中的第 4 行的 N+1 改为 N，试试看会有什么结果（在 Visual C++ 6.0 中会产生程序运行错误，而在其他系统中可能不出错）。

另外，C 语言对数组的大小有限制吗？比如，是否可以定义长度达 100 万的大型数组？

答案是不确定的，因为与计算机系统相关。对于 DOS 等采用分段内存管理的操作系统，使用超过 64KB 的数组需要特殊的技巧。而对于目前更常见的操作系统，如 Windows、Linux 等，内存管理采用平板式结构，使用多大的数组仅受内存大小的限制。

3．数组的初始化

程序中，引用数组元素的数值以前必须先给数组元素赋值。因此，下面的程序段是错误的：

```
int  i,sum=0;
int  a[10];
for(i=0;i<10;i++)sum+=a[i];      /*数组 a 中各元素的值是随机的“脏数据”*/
```

C 语言规定，局部数组如果没有初始化，数组元素的取值是不确定的。直接引用没有赋值的数组元素是错误的。给数组元素赋值，可以使用赋值语句，也可以通过调用 scanf 等函数从键盘输入（参见例 6.1 和例 6.2）。

在定义数组时，可以用数值集合对数组进行初始化，即给数组元素赋初值，方法是：

类型说明 数组名称[数组长度]={数值列表}

花括号中的数值列表是用逗号分隔的若干个数值的集合，这些数值的数据类型与数组类型保持一致，数值的个数不能超过数组长度！例如：

```
int  a[5]={1,2,3,4,5};
```

如果只是给某些元素赋初值，其他元素的值（可能是 0）也必须给出。例如：

```
int  b[5]={1,1,0,0,1};
```

当然，如果要赋初值的元素位于数组的前列，初始化时可以采用省略格式。例如：

```
int  c[5]={1,2,3};
```

不同于未初始化的数组，这里的数组元素 a[3]、a[4]的初值是确定的，为 0。

一个较极端的情况是，右边的括号中只有一个 0 值，即 int c[5]={0}，此时各元素也自动被初始化了，初始值都为 0，与“int c[5];”不同。当然，所有元素都为 0 时，也不能写为“int c[5]={};”，右边的花括号中没有任何数值是不允许的。而“int c[5]={5*1};”试图同时给 5 个元素赋初值也是不合法的。

如果在“数值列表”中给出了所有元素的初值，“数组长度”说明是可以省略的，但不要省略方括号。例如：

```
int  d[]={1,2,3,4,5};
```

注意，如果缺少等号和右面的数值列表，只书写“int d[];”是错误的。

【例 6.3】　求数组中 0 元素的个数。

```
1   #include<stdio.h>
2   #define N 10
```

```
3   void main()
4   {   int i,count=0;
5       int  data[N]={1,0,3,5};
6       for(i=0;i<N;i++)
7           if(data[i]==0)count++;
8       printf("Count of non zero is %d\n",count);
9   }
```

程序中，虽然只显式地给数组 data 的前 4 个元素赋了初值，分别是 1、0、3、5，其他元素实际上也做了初始化，初值都为 0。如果省略第 5 行中的“={1,0,3,5}”，则编译该程序时会报错。

6.2.2　一维数组的应用

【例 6.4】　某班有 30 个同学参加了某门课的考试，试计算他们的平均成绩，并打印出低于平均成绩的成绩列表。

分析：这是 6.1.1 节的例题的一个变例，算法如图 6.5 所示。

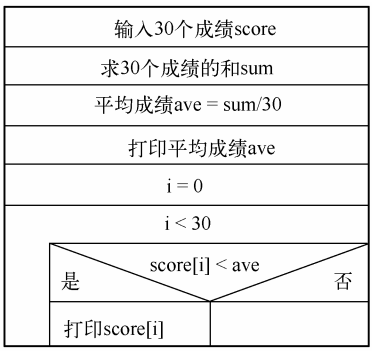


图 6.5　求平均成绩

程序如下：

```
1   #include<stdio.h>
2   #define N 30
3   void main()
4   {   int i,score[N];
5       float ave=0;
6       printf("Enter scores one by one?");
7       for(i=0;i<N;i++)scanf("%d",&score[i]);
8       for(i=0;i<N;i++)ave+=score[i];
9       ave=ave/N;                                  /*计算平均成绩*/
10      printf("Average is %.1f\n",ave);
11      printf("Less than average:");
14      for(i=0;i<N;i++)
15          if(score[i]<ave)printf("%d",score[i]);/*打印低于平均值的成绩*/
```

```
16      printf("\n");
17  }
```

【例 6.5】　在某体育项目中，同时有 7 位裁判为运动员打分，运动员的最终成绩是去掉一个最高分和一个最低分后的 5 个得分的平均值，试编程实现计算运动员成绩的过程。

分析：本程序包含两个过程，一是求最高得分和最低得分，二是求平均值。求平均值的方法是大家已经熟悉了，这里只分析第一个过程。得容易理解，求最高得分和求最低得分是类似的，实际就是数学上的求最大值、最小值问题。

以求最大值为例，可以先假设第 1 个数最大，再与第 2 个数比较，如果小于第 2 个数，则最大数用第 2 个数代替，否则不变，然后与第 3 个数比较，以此类推，直到比较完所有的数值。可用图 6.6 表示这一算法过程。

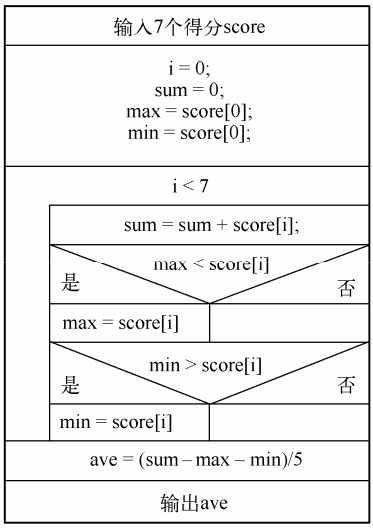


图 6.6　计算运动员的成绩

程序如下：

```
1   #include<stdio.h>
2   #define N 7
3   void main()
4   {   int i;
5       float score[N];
6       float max,min,ave=0;
7       printf("逐一输入裁判给分?");
8       for(i=0;i<N;i++)scanf("%d",&score[i]);
9       max=score[0];
10      min=score[0];
11      for(i=0;i<N;i++)
12      {   ave+=score[i];                          /*求和*/
13          if(max<score[i])max=score[i];           /*求最高得分*/
```

```
14          if(min>score[i])min=score[i];        /*求最低得分*/
15      }
16      ave=(ave-max-min)/(N-2);                 /*计算平均值*/
17      printf("运动员得分是%.1f\n",ave);
18  }
```

程序的运行结果如下：

```
逐一输入裁判给分? 5.6   5.7   5.8   5.7   5   5.8   6 ↙
运动员得分是 5.72
```

【例 6.6】 假设数组 a 为{1，2，3，4，5，6，7，8，9，10}，试编程将其倒置，结果为{10，9，8，7，6，5，4，3，2，1}。要满足两个条件：①程序可适应任何数组长度；②程序中只能使用一个数组，不能引入第二个数组。

分析：对于第一个要求，虽然 C 语言程序中不能定义“任意”长度的数组，但可以将数组长度定义为较大的 N，而实际大小为 n，n≤N，即程序只处理长度为 N 的数组的前 n 个元素。n 的值可在运行程序时输入，从而实现了“任意”长度的数组。

如果没有第二个要求，可以再定义一个数组 b，数组 b 的类型、长度与数组 a 完全相同，然后令 b[0]=a[n−1]、b[1]=a[n−2]、……、b[n−1]=a[0]即可。这样做的缺点是要多占用计算机内存。而正如题目要求的，如果不能定义第 2 个数组，该如何做呢?

在前面的章节中，我们已经学会了一个交换两个数的办法，要交换 x 和 y，可以令 t=x，x=y，y=t。这一方法同样可以用于数组元素的交换，比如要交换 a[i]和 a[j]，可以令 t=a[i]，a[i]=a[j]，a[j]=t。

算法如图 6.7 所示。

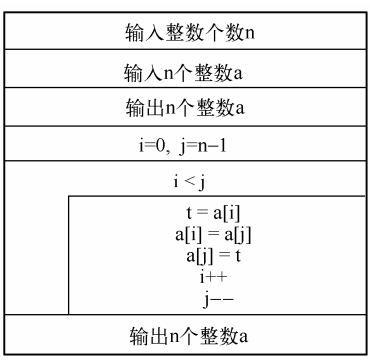


图 6.7　将数组倒置

程序如下：

```
1   #include<stdio.h>
2   #define N 100
3
4   void main()
5   {   int i,j,n,t,a[N];
6       printf("请输入数据个数?");
```

```
7       scanf("%d",&n);                 /*输入数组的实际长度*/
8       while(n<1||n>N)                 /*本循环用来校验输入的有效性*/
9       {   printf("错误，数据个数应介于 1～%d 之间!\n 请重新输入?",N);
10          scanf("%d",&n);
11      }
12      printf("请输入%d 个整数:\n",n);
13      for(i=0;i<n;i++)scanf("%d",&a[i]);  /*输入数组的 n 个值*/
14      printf("\n 输入的整数是:");
15      for(i=0;i<n;i++)printf("%d",a[i]);  /*输出原始数组*/
16      i=0;
17      j=n-1;
18      while(i<j)                          /*求逆*/
19      {   t=a[i];
20          a[i]=a[j];
21          a[j]=t;
22          i++;
23          j--;
24      }
25      printf("\n 倒置后的结果为:");
26      for(i=0;i<n;i++)printf("%d",a[i]);  /*输出结果*/
27      printf("\n");
28  }
```

6.3　多 维 数 组

在程序设计语言中，数学上的矩阵、方阵常用二维数组表示。矩阵运算是代数问题的核心，在科学工程运算中占有重要位置。

6.3.1　多维数组的定义

1．二维数组

定义二维数组的格式为：

类型说明 数组名称[行数][列数]

“类型说明”、“数组名称”的定义与一维数组相同。“行数”、“列数”都是整型常量或常量表达式。但要注意，C 语言中，定义二维数组时“行数”、“列数”必须分别放在独立的方括号中。

例如：

```
float  a[3][4];
```

定义了一个有 3 行、4 列共 12 个单精度实数的二维数组 a。

二维数组的数组元素的格式为：

数组名称[行下标][列下标]

对于有 *m* 行 *n* 列的二维数组 a，其行下标的取值范围是 0～*m*−1，列下标的取值范围是 0～*n*−1。第 1 行第 1 列的元素为 a[0][0]，这也是数组的第一个元素；第 *m* 行第 *n* 列的元素为 a[*m*−1][*n*−1]，它是最后一个元素。那么，第二个元素是 a[1][0]还是 a[0][1]呢？

计算机内存地址是线性编址的，程序中的一维数组的元素是线性排列的，与内存的线性特征是吻合的。但与矩阵对应的二维数组是平面结构的，一个元素可以有上、下、左、右 4 个相邻元素，与内存的线性特征不一致，因而需要进行转换。

C 语言中，二维数组是按行存储的。可以这样理解，可以把二维数组看做是用一维数组定义的一维数组，即该一维数组的每个元素又是一个一维数组。例如，对于 int a[3][4]，可以理解为是由 3 个元素组成的一维数组，这 3 个元素是 a[0]、a[1]、a[2]，分别对应数组第 1、2、3 行，这些元素按 a[0]、a[1]、a[2]顺序存放。常把与行对应的一维数组称为行数组，所以数组 a 由 3 个行数组 a[0]、a[1]、a[2]组成。这 3 个行数组又分别是一个一维数组，以 a[0]为例，其元素为 a[0][0]、a[0][1]、a[0][2]、a[0][3]，这 4 个元素也是顺序存放的。因此，二维数组 a 的各元素的存放顺序为 a[0][0]、a[0][1]、a[0][2]、a[0][3]、a[1][0]、a[1][1]、a[1][2]、a[1][3]、a[2][0]、a[2][1]、a[2][2]、a[2][3]，可以用图 6.8 表示这种存储格式。

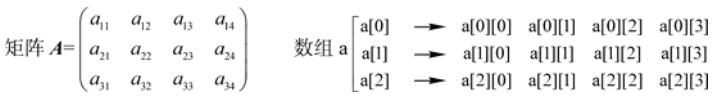


图 6.8　二维数组的存储格式

2．多维数组

可以将用一维数组定义二维数组的方法延伸到多维数组的定义。

三维数组的定义格式为：

类型说明 数组名称[行数][列数][层数]

与二维数组类似，可以把三维数组看做是由二维数组组成的一维数组，即该一维数组的元素是一个二维数组。在内存中，三维数组中的各元素的存放顺序是先按行存储，再按列存储，最后分层。

也可以这样理解，C 语言中多维数组的元素在内存中的存储顺序是：最右边的下标变化最快。分析二维数组 a[3][4]的存储格式就不难理解这一点。

6.3.2　多维数组的初始化

多维数组的初始化方法可以参照一维数组。以二维数组为例，初始化方法有以下两种。

（1）按元素存储顺序赋初值

例如：

```
int a[2][3]={1,2,3,4,5,6};
```

初值的顺序与数组各元素的内存顺序对应，因此，数组 a 的第 1 行为 1、2、3，第 2 行为 4、5、6。

容易理解：

```
int a[2][3]={1,2,3};
```

只给部分元素（这里是 a[0][0]、a[0][1]、a[0][2]）赋初值，其他元素的值被初始化为 0。

（2）按行赋初值

初始值放在嵌套的花括号中，最外面的（第 1 层）花括号与第 1 维（行）对应，第 2 层花括号对应第 2 维（列），以此类推。

例如：

```
int a[2][3]={{1,2,3},{4,5,6}};
```

{1,2,3}对应第 1 行元素，{4,5,6}对应第 2 行元素。

再如：

```
int a[2][3]={{1},{4,5}};
```

初始化结果是，a[0][0]的值为 1，a[1][0]为 4，a[1][1]为 5，其他元素都为 0。

对于三维数组，例如：

```
int a[2][3][4]={{{1},{2,3},{4,5,6}},{{7,8,9,10}}};
```

请读者自己分析初始化后的结果。

定义二维数组时，如果根据初始值的情况可以确定数组的行数，则可以省略数组定义中的行数声明。例如：

① int a[][4]={{1,2,3,4},{5,6,7,8}};省略的是 2。

② int b[][3]={1,2,3,4,5,6,7,8,9};省略的是 3。

③ int c[][3]={{1,2},{3}};省略的是 2。

是否可以省略列数说明呢？例如：

① int a[2][]={{1,2,3,4},{5,6,7,8}};

② int b[3][]={1,2,3,4,5,6,7,8,9};

答案是否定的，为什么？由于给出的数据个数可能少于数组元素的个数，所以 C 语言编译器无法确定数组的列数到底是多少，也就无法为数组分配内存。

这一规律可以推广到任意维数的数组，也就是说多维数组初始化时只可以省略第 1 维的长度。

【例 6.7】　用 1～12 初始化数组 a[3][4]，然后用从键盘输入的任意两个数替换 12 个数的中间两个，输出替换前后的二维数组。

分析：对于二维数组 a[3][4]，中间两个元素是 a[1][1]和 a[1][2]（见图 6.8）。

程序如下：

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

```
3   {   int i,j,x,y;
4       int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};   /*按元素顺序赋初值*/
5       printf("替换前的数组是:\n");
6       for(i=0;i<3;i++)
7       {   for(j=0;j<4;j++)
8               printf("%5d",a[i][j]);
9           printf("\n");                           /*分行*/
10      }
11      printf("请输入 2 个整数:");
12      scanf("%d%d",&x,&y);
13      a[1][1]=x;
14      a[1][2]=y;
15      printf("替换后的数组是: \n");
16      for(i=0;i<3;i++)
17      {   for(j=0;j<4;j++)
18              printf("%5d",a[i][j]);
19          printf("\n");                           /*分行*/
20      }
21  }
```

程序中第 6～0 行、第 16～20 行实现了二维数组的输出，分行的目的是为了更直观。习惯上，一维数组打印在一行内，而二维数组打印成二维的矩阵形式。

6.3.3　多维数组的应用

【例 6.8】　对于矩阵 $\boldsymbol{A}=\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{32} & a_{32} & a_{33} \end{pmatrix}$，与数值 t 的乘积定义为 $t\cdot\boldsymbol{A}=\begin{pmatrix} ta_{11} & ta_{12} & ta_{13} \\ ta_{21} & ta_{22} & ta_{23} \\ ta_{32} & ta_{32} & ta_{33} \end{pmatrix}$。试编程实现这一运算。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int  i,j,t;
4       int  a[3][3];
5       printf("请输入一个 3×3 数组: \n");
6       for(i=0;i<3;i++)                /*输入数组元素的值*/
7           for(j=0;j<3;j++)
8               scanf("%d",&a[i][j]);
9       printf("该数组是: \n");
10      for(i=0;i<3;i++)                /*打印原数组*/
11      {   for(j=0;j<3;j++)
12              printf("%5d",a[i][j]);
13          printf("\n");               /*分行*/
```

```
14      }
15      printf("请输入一个整数: ");
16      scanf("%d",&t);
17      for(i=0;i<3;i++)                    /*计算乘积*/
19          for(j=0;j<3;j++)
20              a[i][j]=t*a[i][j];
21      printf("乘积是: \n");
22      for(i=0;i<3;i++)                    /*打印结果*/
23      {   for(j=0;j<3;j++)
24              printf("%5d",a[i][j]);
25          printf("\n");                   /*分行*/
26      }
27  }
```

程序的第 6～8 行用于控制输入数值的顺序，不难看出该顺序与数组元素的内存顺序是一致的。因此，输入的数值可以分为 3 行，每行从左到右有 3 个数，也可以在 1 行内输入所有数值或其他格式。

下面是程序某次运行的结果：

```
请输入一个 3×3 数组:
   1  2  3  4 ↙
   5  6  7 ↙
   8  9 ↙
该数组是:
   1    2    3
   4    5    6
   7    8    9
请输入一个整数: 10↙
乘积是:
  10   20   30
  40   50   60
  70   80   90
```

【例 6.9】　对于给定的二维数组 $\begin{pmatrix}1&4&7&2\\1&2&5&0\\8&1&3&1\end{pmatrix}$，求最大值所在的行和列。

分析：该程序不仅要求出最大值，而且要记录最大值的下标，其算法如图 6.9 所示。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int i,j,max,x,y;
4       int a[3][4]={{1,4,7,2},{1,2,5,0},{8,1,3,1}}; /*数组的初始化*/
5       max=a[0][0];                        /*假设第 1 个元素最大*/
```

```
6       x=y=0;
7       for(i=0;i<3;i++)
8           for(j=0;j<4;j++)
9               if(max<a[i][j])
10              {   max=a[i][j];                    /*max 为当前最大元素*/
11                  x=i;                            /*x 为行下标*/
12                  y=j;                            /*y 为列下标*/
13              }
14      printf("最大值在第%d 行第%d 列",x+1,y+1);
15  }
```

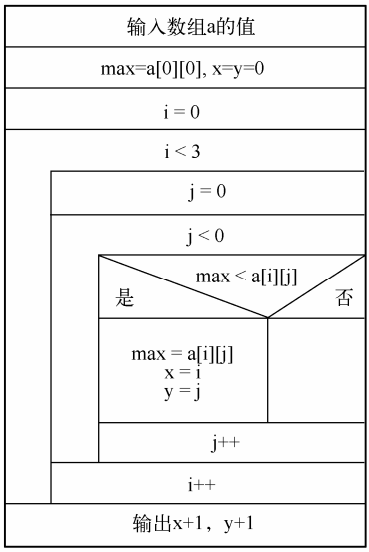


图 6.9　求二维数组中最大元素的下标

其实，max 变量并不是必需的，可以直接用 a[x][y]代替，修改后的程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int i,j,x,y;
4       int a[3][4]={{1,4,7,2},{1,2,5,0},{8,1,3,1}};
5       x=y=0;
6       for(i=0;i<3;i++)
7           for(j=0;j<4;j++)
8               if(a[x][y]<a[i][j])          /*a[x][y]是当前的最大值*/
9               {   x=i;                     /*x 为行下标*/
10                  y=j;                     /*y 为列下标*/
11              }
12      printf("最大值在第%d 行第%d 列",x+1,y+1);
13  }
```

6.4　应 用 举 例

【例 6.10】　求向量的内积。设 $\boldsymbol{A}=(a_1, a_2, \cdots, a_n)$、$\boldsymbol{B}=(b_1, b_2, \cdots, b_n)$为 n 维向量，向量 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的内积为 $a_1*b_1+a_2*b_2+\cdots+a_n*b_n$。

分析：

① 题目中并未限制向量的长度，设计程序时应把 n 看做“任意”整数；

② 程序中，向量用一维数组表示，其数据类型为实型。

算法如图 6.10 所示。

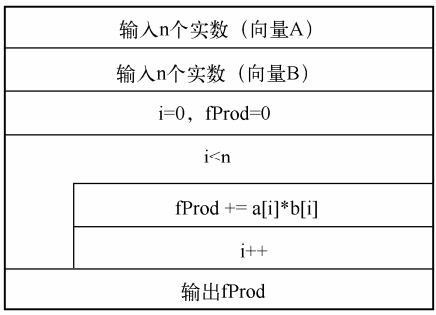


图 6.10　求向量的内积

程序如下：

```
1   #include<stdio.h>
2   #define N 100
3
4   void main()
5   {
6       int  n,i;                /*n 为向量的实际长度*/
7       float  fa[N],fb[N];      /*向量 A 和 B，N 为存储单元个数*/
8       float  fProd;            /*内积*/
9
10      printf("向量长度为（1～%d）？",N);
11      do
12          scanf("%d",&n);
13      while(n<1||n>N);         /*如果输入的 n 不在有效范围内则重新输入*/
14      printf("向量 A 为？");
15      for(i=0;i<n;i++)scanf("%f",&fa[i]);
16      printf("向量 B 为？");
17      for(i=0;i<n;i++)scanf("%f",&fb[i]);
18
19      fProd=0;
20      for(i=0;i<n;i++)
21          fProd+=fa[i]*fb[i];
```

```
22
23      printf("向量 A 和 B 的内积为：%f",fProd);
24  }
```

程序的第 2 行定义了一个较大的符号常量 N，在第 7 行用 N 代表数组长度，数组的实际“长度”为 n，在执行第 10～13 行时输入。

【例 6.11】 系统工作时，工程师通过分析一段时间内某传感器记录的数据发现系统的异常。假设已经记录了 10000 个数据，试统计超过某个阈值的数据的个数。

分析：要输入的数据较多，可以用自动获取的随机数来仿真。算法如图 6.11 所示。

图 6.11　数据统计的仿真

程序如下：

```
1   #include<stdio.h>
2   #include<stdlib.h>
3
4   #define N 10000
5   #define MAX_VALUE 1000
6
7   void main()
8   {   int i,iData[N];
9       int iThreshold;
10      int iCount;
11
12      printf("输入阈值（应低于最大值%d）?",MAX_VALUE);
13      scanf("%d",&iThreshold);
14
15      for(i=0;i<N;i++)
16          iData[i]=rand()%MAX_VALUE;       /*用随机数仿真记录的数据*/
17
18      iCount=0;
19      for(i=0;i<N;i++)
20          if(iData[i]>iThreshold)iCount++;
21      printf("共发现了%d 次异常! \n",iCount);
22  }
```

读者可能已经发现，每次运行程序得到的结果都相同，这是因为用 rand 函数产生的随机数序列相同。解决方法是用系统的时间作为种子初始化随机数序列，将程序的第 3 行改为：

```
3   #include<time.h>
```

将第 14 行改为：

```
14      srand((unsigned)time(NULL));
```

这里调用 time 函数获取系统时间并转换为一个整数，用做 seed 函数的参数——随机数序列的种子，以获取不同的随机数。

【例 6.12】　假设某数组中存放了若干整数，试用顺序查找方法查找某个整数。

分析：查找（也叫搜索）是计算机数据处理的常用算法之一，也是日常生活最常见的问题。如果待查序列中数据的个数不是非常多，简单的顺序查找算法就可以解决问题。当数据个数非常大时，如何提高查找效率就成了算法的关键。

查找的结果只能是找到还是没找到，如果找到了，则该查找是成功的，否则就是失败的。对于成功的查找，只要找到一个就可以了，不必比较所有的数据。但对于失败的查找，需要确认每个数据都不是要找的。是否需要比较所有数据呢？用顺序查找方法则需要，而用折半法查找则不需要，这也是折半法查找效率高的关键所在。

顺序查找时，从数组的第 1 个元素开始逐一与要查找的数据进行比较，如果有一个元素与之相同，就是找到了，查找过程即可结束。如果所有元素都不同于要查找的数据，则查找失败。因此，对于顺序查找，成功的查找只要保证能找到一个相同的元素即可，无须比较所有元素，而失败的查找则要比较所有元素。顺序查找过程如图 6.12 所示。

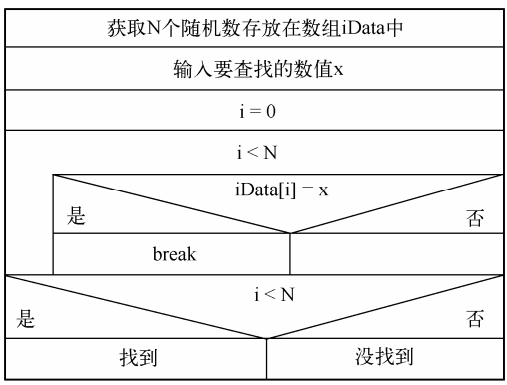


图 6.12　顺序查找过程

程序如下：

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include<time.h>
4   #define N 100
5
6   void main()
7   {   int  i,iData[N];                    /*存放待查序列*/
8       int  x;                             /*要查找的数*/
9       char ch;
```

```
10
11      srand((unsigned)time(NULL));                    /*初始化随机数序列*/
12      for(i=0;i<N;i++)iData[i]=rand()%100;            /*获取若干随机数组成
                                                        待查序列*/
13
14      do
15      {   printf("要查找的整数是? ");
16          scanf("%d",&x);
17                                                      /*顺序查找*/
18          for(i=0;i<N;i++)
19              if(x==iData[i])break;    /*如果有一个元素相同则结束查找过程*/
20                                       /*查找结果*/
21          if(i<N)printf("找到了，%d 是数组的第%d 个元素。\n",x,i+1);
22          else printf("没有找到%d。\n",x);
23
24          printf("要查找其他整数吗(y/n)?");
25          while((ch=getchar())<97||ch>123);
26      } while(ch=='y');
27      printf("\n");
28  }
```

以下是程序的运行结果：

```
要查找的整数是? 0↙
找到了，0 是数组的第 58 个元素。
要查找其他整数吗(y/n)? y↙
要查找的整数是? 3↙
找到了，3 是数组的第 11 个元素。
要查找其他整数吗(y/n)? y↙
要查找的整数是? 4↙
没有找到 4。
要查找其他整数吗(y/n)? n↙
```

【例 6.13】　用起泡法对数组元素进行排序，排序后元素按数值从小到大顺序排列。

分析：与查找一样，排序也是最基本的算法之一，但由于算法的复杂性更高，效率问题更为突出。对于少量数据的排序，可以采用起泡法、选择法、插入法等简单排序方法，而对于大规模数据的排序，则需要更高速的排序方法，如快速排序、希尔排序等。

起泡法又叫冒泡法，属于交换排序，是一种较简单的算法。假设待排序数组 a 的长度为 n，排序过程如下（参见图 6.13）：

① 令 i=0;

② 令 j=n−1（从最后一个元素开始）;

③ 比较 a[j−1]与 a[j]，若 a[j−1]>a[j]，则把它们交换过来；

④ 若 j>i，则令 j−−，转到③，否则转到⑤；

⑤ 若 i<n−1，则令 i++，转到②，否则转到⑥；

⑥ 完成排序。

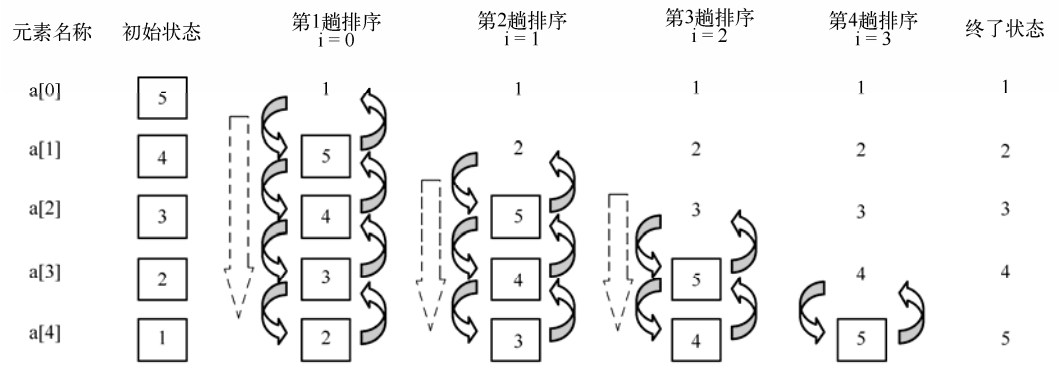


图 6.13　起泡法排序

注：实线箭头表示交换过程，虚线箭头表示推进方向

可以把某一次外循环过程称做某趟排序。可以看出，对于有 n 个元素的数组，起泡法需要执行 $n-1$ 趟排序，第 1 趟排序需要比较 $n-1$ 次，第 2 趟排序则需要比较 $n-2$ 次，……，第 $n-1$ 趟排序仅需要比较 1 次。因此，总的比较次数是$(n-1)+(n-2)+\cdots+2+1=n(n-1)/2$。当元素数 n 较大时，起泡法排序需要的比较次数会非常大，则影响了排序的效率。

程序如下：

```
#include<stdio.h>
#define N 100

void main()
{   int  i,j,n;
    int  iTemp,iData[N];

    do                                          /*限制数据个数，避免数组越界*/
    {   printf("请输入数据个数（1～%d）?",N);
        scanf("%d",&n);
    } while(n<1||n>N);

    printf("顺序输入%d 个整数:",n);
    for(i=0;i<n;i++)scanf("%d",&iData[i]);  /*输入带排序序列*/
    printf("原序列是:");
    for(i=0;i<n;i++)printf("%5d",iData[i]); /*打印输入结果*/
    for(i=0;i<n-1;i++)                      /*起泡法排序*/
        for(j=n-1;j>i;j--)
            if(iData[j-1]>iData[j])
            {   iTemp=iData[j-1];
                iData[j-1]=iData[j];
                iData[j]=iTemp;
            }
```

```
24      printf("\n 排序后是:");
25      for(i=0;i<n;i++)printf("%5d",iData[i]);      /*打印排序结果*/
26      printf("\n");
27  }
```

程序中，第 17～23 行是一个复杂的 for 循环语句，为外循环；第 18～23 行是其循环体，也是一个 for 语句，为内循环；第 19～23 行是内循环的循环体，是一个条件执行语句，该条件执行语句的执行语句部分是一个复合语句，由 20～23 行的 3 条语句组成。这一程序结构可用于其他排序算法，基本是固定的，值得初学者仔细地分析和理解。

【例 6.14】　判断 n 阶方阵 $\boldsymbol{A}=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$ 是否为对称方阵。

分析：判断方阵 $\boldsymbol{A}$ 为对称方阵的条件是对于每一个元素 a_{ij}（$1 \leqslant i$、$j \leqslant n$）都有 $a_{ij}=a_{ji}$。只要有一个元素 $a_{ij} \neq a_{ji}$，则 A 就不是对称的。容易理解，可以将 j 的取值范围缩小到 $i<j \leqslant n$ 或 $1 \leqslant j<i$，即只处理对角线下面或上面的元素。

程序如下：

```
1   #include<stdio.h>
2   #define N 100
3
4   void main()
5   {
6       int  n,i,j;              /*n 为方阵阶数*/
7       float  fMat[N][N];       /*可存放 N 阶方阵*/
8       float  iFlag;            /*对称标识：0—不对称，1—对称*/
9
10      printf("方阵的阶数为（1～%d）？ ",N);
11      scanf("%d",&n);
12      printf("按行输入方阵的各元素: ");
13      for(i=0;i<n;i++)
14          for(j=0;j<n;j++)
15              scanf("%f",&fMat[i][j]);
16
17      iFlag=1;
18      for(i=0;i<n && iFlag;i++)        /*只要有 iFlag 为 0，就结束循环*/
19          for(j=i+1;j<n && iFlag;j++) /*只处理对角线下的元素*/
20              if(a[i][j]!=a[j][i])     /*只要有一个元素不等，就不是对称
                                           方阵*/
21                  iFlag=0;
22
23      if(iFlag)printf("该方阵是对称方阵。");
24      else printf("该方阵不是对称方阵。");
24  }
```

习　题　6

6.1　程序改错

（1）统计数组中正数和负数的个数，算法如图 6.14 所示。

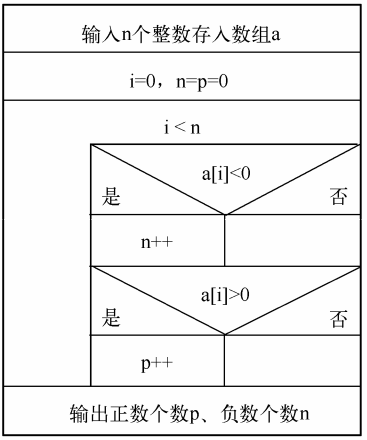


图 6.14　统计正数和负数的个数

程序如下：

```
#include<stdio.h>
void main()
{   int a[10];
    int p=0,n=0,i;
    for(i=0;i<10;i++)scanf("%d",a[i]);
    for(i=0;i<10;i++);
    if(a[i]<0)n++;
    if(a[i]>0)p++;
    printf("有 %d 个正数，%d 个负数\n",p,n);
}
```

（2）n 阶方阵 $\boldsymbol{A}=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$ 的转置方阵为 $\boldsymbol{A}^{\mathrm{T}}=\begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ a_{1m} & a_{2m} & \cdots & a_{nm} \end{pmatrix}$，也就是，把方阵 $\boldsymbol{A}$ 的行和列互换可得到 $\boldsymbol{A}$ 的转置矩阵 $\boldsymbol{A}^{\mathrm{T}}$。以下程序实现任意 4 阶方阵的转置。

提示：为避免重复，应只对主对角线下（或上）的元素进行处理。

```
#include<stdio.h>
#define N 4
void main()
{   int a[N][N]={{1},{2,3},{4,5,6},{7,8,9,10}};
    int i,j;

```

```
7       printf("原矩阵为：\n");
8       for(i=0;i<N;i++)
9       for(j=0;j<N;j++)
10      printf("%6d",a[i][j]);
11      printf("\n");
12      for(i=0;i<N;i++)
13          for(j=0;j<N;j++)
14              t=a[i][j];
15              a[i][j]=a[j][i];
16              a[j][i]=t;
17      printf("转置矩阵为：\n");
18      for(i=0;i<N;i++)
19      for(j=0;j<N;j++)
20      printf("%6d",a[i][j]);
21      printf("\n");
22  }
```

6.2　程序填空

（1）判断两个数组相等。

提示：若数组 A={a_i | 1≤i≤n}和数组 B={b_i | 1≤i≤n}相等，则有 a_i=b_i（1≤i≤n）。

```
1   #include<stdio.h>
2           ①
3   void main()
4   {   int a[NUM],b[NUM];
5       int n;
6       int i,iFlag;
7       printf("数组长度（1～%d）为？",NUM);
8       scanf("%d",      ②      );
9       printf("输入 A 数组：");
10      for(i=0;i<n;i++)      ③
11      printf("输入 B 数组：");
12      for(i=0;i<n;i++)      ④
13          ⑤
14      for(i=0;i<n      ⑥      ;i++)
15          if(      ⑦      )iFlag=0;
16
17      if(iFlag)printf("二者相等！");
18      else printf("二者不等！");
19  }
```

（2）有一种集体舞，所有人手拉手站成一个环，随音乐节拍按顺时针或逆时针方向边歌边跳。若把有 *n* 个元素的数组 a 的首尾相接，就可以看做一个环，如图 6.15（a）所示。将所有元素向前推进 *m* 位，可得到图 6.15（b）。

提示：可把问题分解为 *m* 次转动，每次只推进 1 位。

```
#include<stdio.h>
#define N 50
void main()
{   int  a[N],iTemp;
    int  i,j,m,n;
    printf("共有几个数(不大于%d)？",N);
    do scanf("%d",&n);       /*该循环用来限制数值的个数*/
    while(n<0          ①          );
    printf("顺序输入%d 个整数：",N);
    for(i=0;i<n;i++)
        scanf("%d",          ②          );
    printf("前进几位？");
    scanf("%d",&m);
    for(i=0;          ③          ;i++)
    {          ④
        for(          ⑤          )
            a[j]=a[j-1];
            ⑥          ;
    }
    printf("结果为：");
    for(i=0;i<n;i++)
        printf("%d",          ⑦          );
}
```

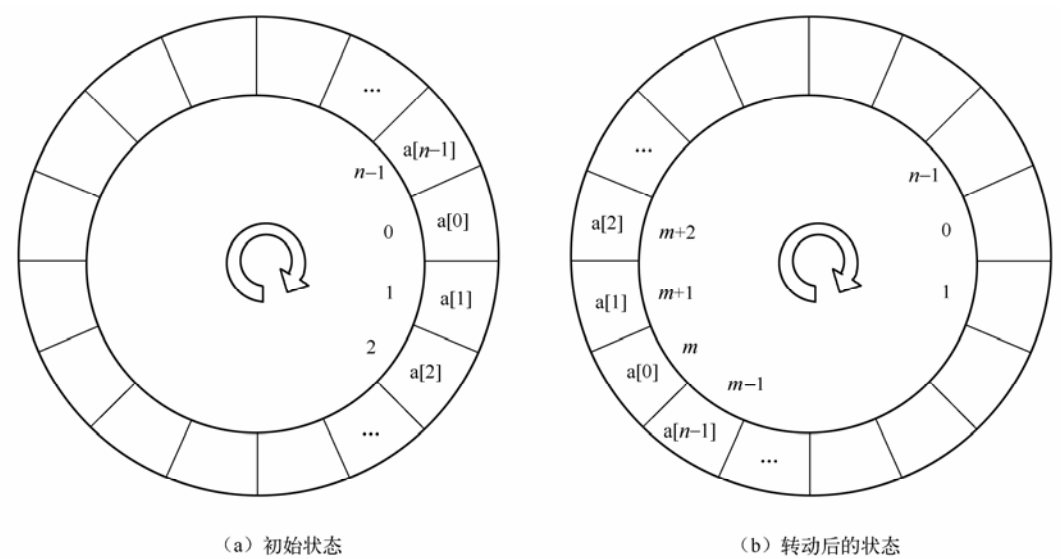


图 6.15　数字转轮

6.3　数组 a 中存放了 n 个整数，试求出数组中的最大元素和第二大元素的下标，并输出相应的元素的值。

6.4　有 10 个整数组成一个数组 a，其中整数 x 出现了若干次，试编程删除多余的 x，只

保留一个 x，显示删除了的 x 的个数和最后的数组 a。

6.5　求大于正整数 m 但最靠近 m 的 k 个素数，并存放在一个数组中，然后输出这些素数。

6.6　求矩阵 $\boldsymbol{A}=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$、$\boldsymbol{B}=\begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1r} \\ b_{21} & b_{22} & \cdots & b_{2r} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nr} \end{pmatrix}$ 的乘积 $\boldsymbol{C}=\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1r} \\ c_{21} & c_{22} & \cdots & c_{2r} \\ \cdots & \cdots & \cdots & \cdots \\ c_{m1} & c_{m2} & \cdots & c_{mr} \end{pmatrix}$，

其中 $c_{ij}=\sum_{k=1}^{n} a_{ik}\times b_{kj}$，$1\leqslant i\leqslant m$，$1\leqslant j\leqslant r$。

6.7　用卡尺测量某钢珠的直径，共测得 15 组数据，试根据拉伊达判据（即 3σ 法）剔除错误数据，然后再计算平均值。

提示：对于某真值 x 的 n 次测量，其平均值为 $\bar{x}=\frac{1}{n}\sum_{i=1}^{n} x_i$，标准偏差为 $\sigma=\sqrt{\frac{1}{n(n-1)}\sum_{i=1}^{n}(x_i-\bar{x})^2}$。若某次测量的随机偏差 $\delta_i=\left|x_i-\bar{x}\right|$ 超过 3σ 即为错误数据。

6.8　选择法排序的思路是从 n 个数中找出最小的数，与第 1 个数交换；再从剩余的 $n-1$ 个数中找出一个次最小的数，与第 2 个数交换；以此类推。试编程实现对任意多个整数的排序。

6.9　若数组 a 是有序的（从小到大排列），编程实现用折半法查找数值 x 的过程，若找到了则显示 x 的位置。

提示：在[a_i，a_j]($i<j$)上用折半法查找 x 的步骤如下：

① 若 $i>j$ 则转到④，否则计算 $m=(i+j)/2$；

② 若 $x=a_m$，则转到④；

③ 若 $x>a_m$ 则令 $i=m+1$，否则令 $j=m-1$，转到①；

④ 若 $i>j$ 则没找到；否则就是找到了，x 的位置为 m。

6.10　已知数组 A 是有序的（从小到大排列），数组 B 是无序的。要将数组 B 的元素逐一插入到数组 A 中，并保持数组 A 仍然是有序的，试编程。例如 A 为（1，3，5，7），B 为（8，2，4），结果为（1，2，3，4，5，7，8）。

提示：在有序序列 A 中插入一个新元素 x 的方法是，将序列 A 中大于 x 的所有元素从最后一个元素开始依次向后移动 1 个位置，然后把 x 放在空出的位置上。

第 7 章　指　　针

学习目标

通过本章学习，应该掌握：

- 计算机的内存地址、指针的基本概念
- 变量的地址、一维数组与指针的关系
- 指针的有关运算
- 指针的应用

7.1　指针的概念

指针是 C 语言中最重要的基本概念之一，也是一个难点。使用指针，可以构造复杂数据结构（如链表、栈、队列、树等），可以实现动态内存管理，还可以访问受保护的内存区域……

1．计算机的内存地址

什么是地址呢？先来看一个生活中的例子。有 3 个人临时相约在某饭店小聚，A 先到饭店，在服务台登记了房间 1088，然后电话通知了 B，但未告知 C。显然，B 和 C 的到达方法会有所不同，B 可以直奔 1088 房间，而 C 呢？可以到服务台查到 A 所登记的房号。在这里，房号 1088 是 A 的地址，见图 7.1。

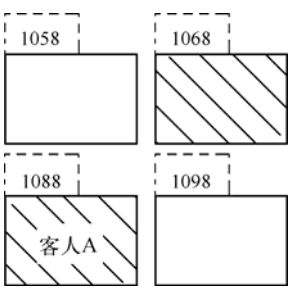


图 7.1　饭店客房管理

在计算机中，内存中的程序和数据都是以二进制形式存放的，存储单位为字节（Byte），1 个字节占 8 个二进制位（bit），即 1B=8bit。计算机的内存大小用这台计算机所拥有的存储单元的个数来衡量，通常，每个存储单元可存放 1 个字节的数据。因此，内存大小可表示为若干字节，如 640KB、256MB、1GB 等。为了便于管理，对计算机的所有内存单元进行统一编号，每个存储单元都有一个唯一的编号，称为该存储单元的地址编码，简称地址。这里，存储单元相当于饭店的房间，地址编码相当于房号，存放在某存储单元中的数据就是住在某房间的房客。

不同于饭店的房间编号，计算机的内存地址编码是线性的。

2．访问内存中的数据

程序运行时所有程序指令和数据都要存放在内存中，也就是要占用存储单元。存放在某存储单元中的指令或数据可称为该单元的内容。不同类型的内容所占据的存储单元的个数不同，比如，一个 int 型整数占 2 个或 4 个存储单元，也可以说它的内存长度为 2 字节或 4 字节。通常，把一个数据所占据的若干存储单元中第一个单元的地址称为该数据的地址。

计算机中，有关内存的操作有两种，即“写”和“读”。所谓“写”是把内容按地址存入对应的存储单元的过程，所谓“读”是获取（但不消除）某地址中的内容的过程。容易理解，为保障程序的运行，存储单元的使用是排他性的，即一旦被占用，则不允许再存入其他内容。程序运行时所占用的存储单元，在程序运行结束后会被操作系统回收。

要访问存放在某存储单元中的数据，可以直接访问，也可以间接访问。所谓直接访问是指可以直接获取内存中的数据。比如，C 语言中，可以用变量的名称直接访问变量的值，在图 7.2 中，变量 a 的值为 3.5。所谓间接访问是指如果某数据的地址是已知的，则可以通过地址间接访问该数据。正如前面的生活实例，只要 C 能够查到 A 所登记的房号，就可以找到他了。同样地，如果出于某种原因不能直接获取变量 a 的值，但可以获得变量 a 的地址 1AD2（见图 7.2），则可以通过该地址间接获取变量 a 的值。

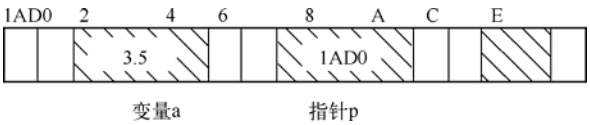


图 7.2　计算机内存中的数据

3. 指针

在程序设计语言中如何存取存储单元的地址呢？C 语言中，用指针变量存放地址值。在图 7.2 中，p 为指针变量，它的值为内存地址 1AD2，据此可以获得变量 a 的值 3.5。

C 语言中，使用指针不仅可以访问变量，也可以访问数组或其他指针，甚至可以访问函数（即程序指令）等。

7.2　变量与指针

7.2.1　指针变量的定义

“定义变量”也就是为变量分配内存空间。数据类型不同，所需要的内存空间大小也不同，因此定义变量时必须指明变量的类型。指针变量的值是一个地址，指针变量的类型不是该地址的类型，而是存放在该地址存储单元中的数据的类型，也就是该指针所指向的数据的类型。

定义指针变量的方法如下：

类型说明*指针变量名

其中，“类型说明”可以是 C 语言的基本数据类型，如 char、int、float、double 等，也可以是用户定义的复杂数据类型，如结构体等。星号“*”称为指针说明符，表示后边的标识符为指针变量名。“指针变量名”的命名规则与一般变量名相同。

如果同时定义多个指针变量，各变量名用逗号“,”分开，同时要在每个变量名的前边加星号“*”，格式如下：

类型说明　*指针变量名 1,*指针变量名 2,……

例如：

```
char  *p1;
int  *p2,*p3;
```

分别定义了一个可指向字符型数据的指针 p1 和两个可指向整型数据的指针 p2、p3。

指针变量可以与其他变量一起定义，例如：

```
int  i,a[5],*p;
```

同时定义了一个整型变量 i、一个整型数组 a 和一个整型指针 p。

特别强调的是，指针变量的类型不是它的值的类型，这与一般变量不同。当然，指针变量也要占据内存空间，因而也有自己的地址，该地址也可以存放在另一个指针变量中（即指针的指针，见 7.4 节）。

7.2.2　指针变量的值

1. 变量的地址

C 语言程序中的变量都必须“先定义，后使用”。变量的定义有两层含义，首先是为变量分配内存，其次是为变量赋值，例如：

```
int x,y;
x=3;
y=x;
```

首先为变量 x 和 y 分配内存（见图 7.3（a）），然后用整型常量为变量 x 赋值（见图 7.3（b）），最后通过直接引用变量 x 的值为变量 y 赋值（见图 7.3（c））。

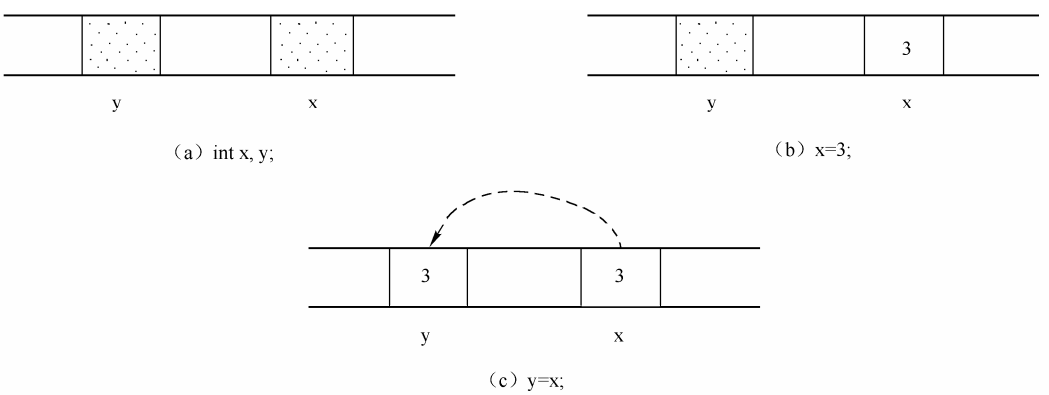


图 7.3　变量的定义

变量 x 的地址可以用 &x 表示，其中，“&”为取地址运算符，可作用于任何类型的变量，一般格式为：

&变量名

取地址运算表达式的返回值是一个地址（可理解为整数，常用十六进制数表示）。

2. 指针变量的赋值

指针变量也是变量，有关普通变量的语法规则同样适用于指针。指针变量的值是存储单元的地址，是一个特殊值。虽然可以理解为整数，但不能直接用整数为指针变量赋值，因此下面的程序段是错误的：

```
int  *p;
p=0x1AB0;        /*用整数为指针变量赋值是错误的*/
```

要为指针变量赋值，可以用已定义的相同类型的普通变量的地址，也可以用已经赋值的另一个相同类型的指针变量的值。因此，下面的程序段是合法的：

```
int  x;
int  *p,*q;
p=&x;            /*可以用已定义（分配内存）相同类型的变量的地址*/
q=p;             /*可以用已定义的相同类型的指针变量的值*/
```

为了便于表达，有时把为指针变量赋值称为“指向”，所以，把 p=&x 称为 p 指向 x，而把 q=p 称为让 q 和 p 指向同一地址。

若所指向的数据的类型与指针不同，则将产生“不兼容”警告，这一错误通常是需要改正的。

需要特别注意的是，程序中永远不要让指针包含一个未知的地址！如果不经意使用了未赋值的指针，其后果可能是灾难性的，可能会中断程序运行，也可能导致系统崩溃，甚至出现硬件问题。与引用未赋值的普通变量一样，程序中引用了未初始化的指针，也仅仅提示有警告性错误，因此，消除这类错误是程序员的责任。

如果程序中暂时还不能确定指针的指向，或者不需要指向某个特定的存储地址，则可以用空指针 NULL（或 0）为指针赋值，方法如下：

```
int  *p,*q;
p=NULL;              /*用空指针赋值*/
q=0;                 /*同样是空指针*/
```

在定义指针变量的同时，也可以初始化。

```
int  x=3;            /*变量 x 的初始化*/
int  *p=&x;          /*指针 p 用 x 的地址初始化*/
int  *q=p;           /*指针 q 用指针 p 初始化*/
int  *r=NULL;        /*指针 r 用空指针初始化*/
```

可以用空指针 NULL 为任何类型的指针赋值，甚至为空类型（用 void 声明）的指针赋值。void 类型的指针可指向任何类型的变量，常作为通用的指针，例如：

```
void  *p;            /*p 为 void 型的指针*/
int  x;
```

```
float  y;
p=&x;            /*p 指向整型变量 x*/
p=&y;            /*p 指向实型变量 y*/
```

3. 指针变量的引用

指针运算符“*”的运算规则是：

***指针变量名**

其运算结果是该指针变量所指向的变量的值。

当指针指向某个确定的地址时，可以通过指针运算符“*”引用所指向的数据，例如：

```
int  x=3,y;
int  *p=&x;
y=*p;
*p=5;
```

该程序段的相关操作可用图 7.4 说明：首先指针 p 指向 x，然后引用指针 p 所指向的变量 x 的值并赋值给变量 y，相当于 y=x，最后将指针 p 所指向的变量的值改为 5，也就是 x=5。

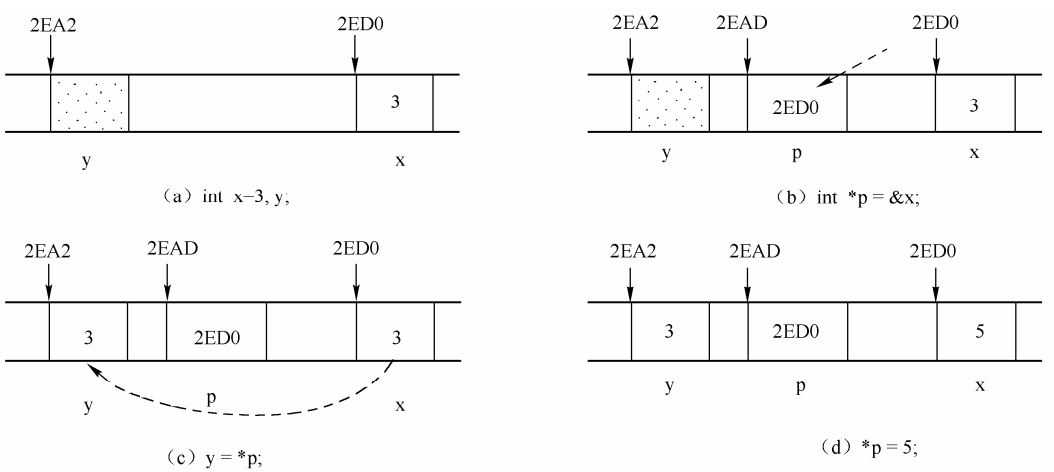


图 7.4　指针的赋值和引用

指针运算符“*”与取地址运算符“&”都是单目运算符，它们的优先级和其他单目运算符如“!”、“++”、“--”相同，高于算术运算符。这些单目运算符都是右结合的，也就是从右向左运算。因此，若有“int y, x=3, *p=&x;”，则 y=10**p 是合法的，也可以写做 y=*p*10，但 y=x**10 是非法的。

不难看出，&*p 等价于 p，而*&a 等价于 a，但 p 必须是指针变量，a 可以是任何类型的变量。

一般不能将指针运算符“*”直接应用于 void 类型的指针，比如对于“int x; void *p=&x;”直接写*p=3 是非法的，因为无法确定*p 的数据类型。为此必须使用强制类型转换，即在引用 p 所指向的变量前，先强行变更为该变量的类型，可将*p=3 改为*((int *)p)=3，其中“(int *)”为强制类型转换。

7.2.3　应用举例

【例 7.1】　用指针实现求两个数的和。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int a=5,b=3,sum;
4       int*pa=&a;
5       int*pb=&b;
6       sum=*pa+*pb;
7       printf("%d+%d=%d",a,b,sum);
8   }
```

【例 7.2】　用指针实现求累加和。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int i,x,sum;
4       int *p1=&x;              /*p1 指向 x*/
5       int *p2=&sum;            /*p2 指向 sum*/
6
7       *p2=0;                   /*即 sum=0*/
8       printf("输入 10 个整数: ");
9       for(i=0; i<10;i++)
10      {   scanf("%d",p1);      /*输入一个 x。注意不能用&p1*/
11          *p2=*p2+*p1;         /*即 sum=sum+x*/
12      }
13      printf("累加和为%d",*p2); /*输出 sum。注意不能用 p2*/
14  }
```

在以前的程序中，第 10 行为“scanf("%d",&x);”，第 2 个参数为变量 x 的地址。本程序中，由于 p1 指向 x，因此用 p1 直接代替&x。同样地，第 13 行的表达式*p2 代替的是变量 sum。

【例 7.3】　求 1～100 的累积和，即 $\text{sum}=\sum_{i=1}^{100} i$。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int  i,x=1,sum=0;
4       int  *p1=&x;
5       int  *p2=&sum;
6
```

```
7       for( i=0;i<100;i++)
8           *p2+=(*p1)++;        /*即 sum+=x++*/
9       printf("1+2+...+100=%d\n",sum);
10  }
```

考虑到运算符的优先级和结合律，第 8 行中的括号不能省略。

请读者思考，如果把第 4 行改为“int *p1=&i;”，怎样修改程序?

7.3　一维数组与指针

数组的诸元素占据相邻的存储空间，每个元素都可以看做具有某种类型的一个变量。因此，可以定义指针指向数组的某个元素，从而通过该指针间接地访问该数组元素。

7.3.1　一维数组的地址

对于数组定义“int　a[5]={3,5,2,1,4};”，假设数组的首地址为 508A，每个数组元素占 4 个字节，则如图 7.5 所示，数组 a 的各个元素 a[0]、a[1]、a[2]、a[3]、a[4]在内存中的地址依次为 508A、508E、5092、5096、509A。

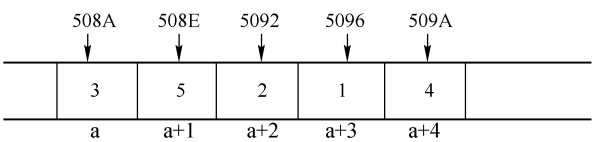


图 7.5　数组的地址

取地址运算同样适用于数组元素，因此数组元素 a[i]的地址是&a[i]。

第一个数组元素的地址称为数组的首地址,用数组名表示,所以数组名 a 被定义为&a[0]。

数组元素 a[i]的地址可表示为 a+i，也就是说对于数组 a，a+i 就是&a[i]。那么容易理解，数组元素 a[i]可用*(a+i)表示。

【例 7.4】　打印数组。

程序如下:

```
1   #include<stdio.h>
2   void main()
3   {   int  i,a[5]={5,3,1,2,4};
4       for(i=0;i<5;i++)
5           printf("%d",*(a+i));
6   }
```

7.3.2　指向数组元素的指针

1. 指向数组元素的指针变量

若有“int a[5],*p;”，可以令指针 p 指向任何一个数组元素 a[i]，方法是令 p=&a[i]，也

可以写为 p=a+i。

若指针 p 指向数组 a 的第一个元素，即 p=a=&a[0]，则 p+i 指向 a[i]，*(p+i)就是 a[i]。因此，若指针 p 和数组 a 的类型相同，且 p=a，则数组 a 的第 i 个数组元素的地址可写为 a+i、p+i、&a[i] 或 &p[i]，该元素的值可表示为 a[i]、*(a+i)、*(p+i) 或 p[i]。值得注意的是，这里 p 是变量名，取值（指向）是可变的，而 a 是数组名，是数组的首地址，是不可改变的。

2. 指针的运算

C 语言中，与指针相关的运算除了赋值运算“=”、取地址运算“&”和指针运算“*”，还有以下 4 种。

（1）指针与整数的加、减法

语法：**指针±整数**

若指针 p 指向数组 a 第 i 个元素 a[i]，则 p+1 指向下一个元素，即 a[i+1]，而 p−1 指向前一个元素，即 a[i−1]。p+j 指向左边（j<0）或右边（j>0）的第 j 个元素。

（2）左移指针“−−”和右移指针“++”

语法：**指针++或 ++指针**

指针−−或 −−指针

若指针 p 指向数组 a 的第 i 个元素 a[i]，则 p++使指针右移一位，指向 a[i+1]，而 p−−则使指针左移一位，指向 a[i−1]。

容易理解，“++”和“−−”运算都只能用在指向某个数组的指针变量上，而不能作用在数组名称上。

【例 7.5】 打印数组。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   int  i,a[5]={5,3,1,2,4};
4       int  *p=a;
5       for(i=0;i<5;i++)
6           printf("%d",*p++);       /*注意运算符的优先级和结合律*/
7   }
```

在第 6 行中，*p++的运算顺序是先“++”后“*”，但由于“++”在运算对象的右边，因此整个表达式的计算过程是返回*p，然后 p++。

请读者思考，*++p 和++*p 两个表达式是等价的吗？为什么?

（3）指针与指针的减法

语法：**指针−指针**

若指针 p 和 q 分别指向数组 a 的两个元素 a[i]和 a[j]，则 p−q 的值为 i−j。由此可知，p−a 为指针 p 所指向的元素的下标。

有时把 p−q 称为指针 p 和 q 之间的距离或间距。

（4）指针的关系运算

语法：**指针 关系运算符 指针**

若指针 p 和 q 分别指向数组 a 的两个元素 a[i]和 a[j]，则 p 和 q 之间的关系为：若 i<j，则 p<q；若 i=j，则 p=q；若 i>j，则 p>q。

7.3.3　内存的动态分配

C 语言中，数组长度是固定的，也就是说，不能定义可变长度的数组。而在现实生活中，要处理的数据的规模有时是难于预期的。比如要处理的是某次考试的成绩，参加考试的考生有多少呢？可能是几十人的一个班，也可能是几千人的一个考点，甚至是几百万人的全国考生。如果使用数组，需要按最大规模设计程序，这在多数时候是不必要的。

数组所占据的内存空间是相对固定的，那么能否根据需要动态地使用内存空间呢？可以，这就要使用动态内存分配技术了。

C 语言所提供的有关函数有 3 个，即 malloc 函数、calloc 函数和 free 函数。

1．malloc函数

malloc 函数的使用方法是：

指针=(类型说明*)malloc(数据长度)

其中，“(类型说明 *)”是强制类型转换，以保证与指针的类型一致。“数据长度”是一个正整数，表示数据所占内存单元个数——字节数，常用 sizeof()运算符来获取。malloc 函数的函数值是成功分配的内存单元的地址，若分配内存失败则返回空指针 NULL。

例如：

```
1       int *p;
2       p=(int*)malloc(sizeof(int));
3       *p=3;
```

第 2 行调用 malloc 函数动态分配一个可存放一个整数（int 类型）的内存空间，并将其地址赋给指针 p。第 3 行的作用可以理解为变量赋值。

sizeof()运算符的用法是：

sizeof(运算对象)

表达式的值是“运算对象”所占内存的存储单元个数。sizeof()的运算对象可以是常量、变量，也可以是类型说明符，例如：

① sizeof(123)返回存放一个整数的内存长度，2 或 4 个字节；

② 对于 float x=123.45，sizeof(x)返回变量 x 的内存长度；

③ 对于 int a[5]，sizeof(a)返回数组 a 的内存长度，即可以存放 5 个整数的存储空间的字节数，不是数组长度；

④ sizeof("apple")返回字符串的内存长度，这里为 6，包含字符串的结束标记'\0'；

⑤ sizeof(int)返回存放 int 型整数的内存长度，与 sizeof(123)等价。

使用 sizeof 运算符可以提高 C 语言程序的可移植性，以适应不同的系统平台。

2. calloc函数

calloc 函数的使用方法是：

指针=(类型说明*)calloc(数据个数,数据长度)

与 malloc 函数不同的是，calloc 函数可分配若干个数据所占据的连续空间，因此可以将 calloc 函数分配的空间与等长度的数组对应，而 malloc 函数只能分配一个变量地址。

例如：

```
1       int *p,n=5;
2       p=(int *)calloc(n,sizeof(int));
3       *p=1;
4       p[3]=4;
```

第 2 行分配一个可存放 n 个整数的空间，p 指向其首地址。第 3～4 行把 p 看做数组名，在第 1、4 个单元里存入数据。

3. free函数

使用 malloc 函数或 calloc 函数成功分配内存空间后，这些存储单元将被占用，如果不释放，就不能被其他变量使用。为了有效地使用内存资源，程序中动态分配的内存空间必须在使用结束后立即释放，方法是调用 free 函数。

free 函数的使用方法是：

free(指针);

作用是释放“指针”所指向的动态分配的内存。

【例 7.6】 求若干整数中的最大值。

分析：在程序运行时输入要处理的数值的个数。

程序如下：

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   void main()
4   {   int n;
5       int*pData,*p,*pMax;
6
7       printf("数据个数为?");
8       scanf("%d",&n);
9       if((pData=(int*)calloc(n,sizeof(int)))==NULL)
10      {   printf("内存分配失败\n");
11          exit(0);
12      }
13      printf("它们是? ");
14      for(p=pData;p<pData+n;p++)
```

```
15          scanf("%d",p);
16
17      pMax=pData;
18      for(p=pData+1;p<pData+n;p++)
19          if(*pMax<*p)pMax=p;
20
21      printf("第%d 个数最大，为%d\n",pMax-pData+1,*pMax);
22
23      free(pData);
24  }
```

程序第 9 行调用 calloc 函数动态分配 n 个整数空间；第 11 行调用 exit 函数强行结束程序的运行，也就是说，如果分配内存失败则终止程序的运行；第 14～15 行是数据的输入过程；第 17～19 行将 pMax 指针指向最大数；第 23 行调用 free 函数释放动态分配的内存空间。

从例 7.6 可以看出，使用动态内存分配极大地改善了程序的适用性。

7.3.4　应用举例

【例 7.7】　用指针实现将数组倒置。

程序如下：

```
1   #include<stdio.h>
2   #define  N  5
3   void main()
4   {   int  a[N]={5,4,3,2,1};
5       int  *p=a;              /*p 指向第一个元素*/
6       int  *q=a+N-1;          /*q 指向最后一个元素*/
7       int  i,t;
8
9       while(p<q)              /*比较指针的大小*/
10      {   t=*p;*p=*q;*q=t;    /*交换 p、q 所指向的元素*/
11          p++;                /*p 右移 1 位*/
12          q--;                /*同时，q 左移 1 位*/
13      }
14      for(i=0;i<N;i++)
15          printf("%5d",a[i]);
16  }
```

【例 7.8】　在数组中任意位置插入一个新元素。

程序如下：

```
1   #include<stdio.h>
2   #define N 100
3   void main()
4   {   int iList[N];          /*存放要处理的数列*/
```

```
5       int *p;
6       int i,l,x,n;            /*l 为插入位置，x 为要插入的数，n 为数值个数*/
7
8       printf("插入前数字个数(1～%d)为? ",N-1);
9       do scanf("%d",&n);
10      while(n<1||n>N-1);      /*限制 n 的取值*/
11
12      printf("顺序输入%d 个数: ",n);
13      p=iList;                /*p 指向第 1 位*/
14      for(i=0;i<n;i++)
15          scanf("%d",p++);    /*输入一个数，并右移指针*/
16                              /*循环结束时 p 指向第 n 位，对应 iList[n]（没
                                  有输入）*/
17      printf("要插入的数是? ");
18      scanf("%d",&x);
19      printf("插入到第几个数的后面? ");
20      scanf("%d",&l);
21
22      for(i=n;i>l;i--)        /*将第 l～n 位数值后移一位*/
23      {   *p=*(p-1);          /*右移数值*/
24          p--;                /*左移指针*/
25      }
26      *p=x;                   /*循环结束时，p 指向第 l 位*/
27      printf("插入后的结果为: ");
28      p=iList;                /*p 重新指向第 1 位*/
29      for(i=0;i<=n; i++)
30          printf("%5d",*p++); /*输出*p，并右移指针*/
31  }
```

*7.4　二维数组与指针

7.4.1　二维数组的元素的地址

C 语言中，二维数组 a[M][N]共有 M×N 个元素，占据连续 M×N 个内存单元，各元素的排列顺序是 a[0][0]、a[0][1]、…、a[0][N−1]、a[1][0]、…、a[M−1][N−1]。若第一个元素元素 a[0][0]的位置为 0，则第 i 行第 j 列的元素 a[i][j]的位置是 i*N+j。

二维数组的每个元素都可以看做一个简单变量，可以使用取地址运算符“&”获取元素 a[i][j]的地址，写为&a[i][j]。若指针变量 p 的类型与二维数组 a 的类型相同，则 p 可指向任一数组元素，即 p=&a[i][j]。

若指针 p 指向二维数组的第一个元素，即 p=&a[0][0]，则指向元素 a[i][j]的指针为 p+i*N+j。

若指针 p 指向某一元素 a[i][j]，则 p++使 p 指向下一元素，可能是同一行的下一元素 a[i][j+1]（j<N−1 时），也可能是下一行的第一个元素 a[i+1][0]（j=N−1 时）。同样地，p−−使 p 指向 a[i][j−1]（j>0 时）或指向上一行的最后一个元素 a[i−1][N−1]（j=0 时）。

【例 7.9】　打印二维数组。

程序如下：

```
1   #include<stdio.h>
2   #define  N  3
3   void main()
4   {   int  i,a[N][N]={1,2,3,4,5,6,7,8,9};
5       int  *p;                        /*简单类型的指针可指向二维数组的元素*/
6       p=&a[0][0];
7       for(i=0;i<N*N;i++)
8       {   printf("%5d",*p++);       /*打印 p 所执行的元素，并移动指针*/
9           if((i+1)%N==0)printf("\n"); /*每行结束输出换行符*/
10      }
11  }
```

7.4.2　指向数组的指针

1．二维数组的行地址

我们知道，对于一维数组 int a[N]，若定义 int *p，可以令 p=a。那么，对于二维数组 int a[M][N],同样定义 int *p，是否可以令 p=a 呢？答案是否定的。为什么呢？因为二维数组名 a 对应的是二维数组的第一行的行地址，不是第一个元素的地址。

如图 7.6 所示，C 语言中，二维数组 a[M][N]可以看做是由 M 个一维数组组成的一维数组，该一维数组的元素是 a[0]，a[1]，…，a[M−1]。数组名称 a 是该一维数组的首地址，也就是第一元素 a[0]的地址。因此，a 是第 1 行的地址，a+1 是第 2 行的地址，以此类推。

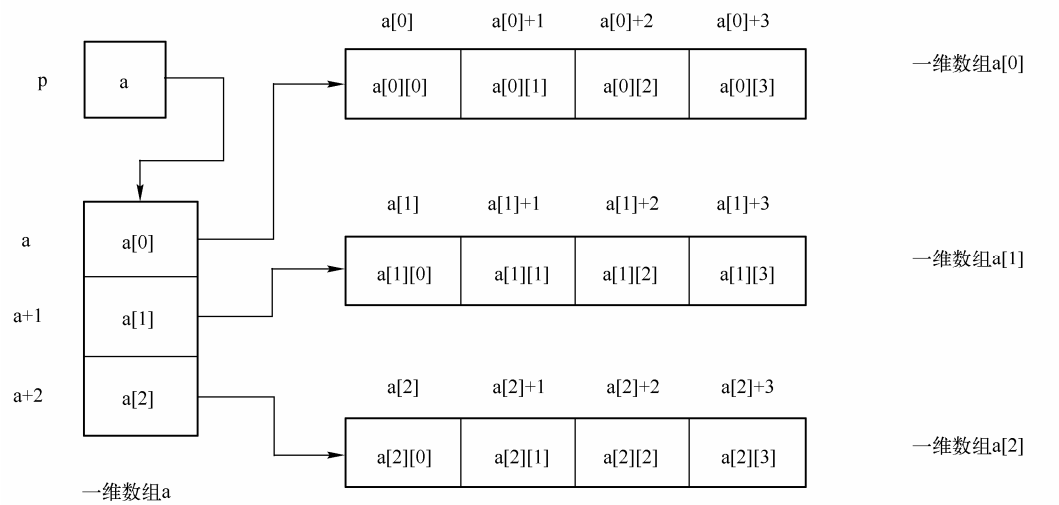


图 7.6　二维数组的行地址

二维数组的每一行也是一个一维数组，由 N 个元素组成。第 1 行对应的数组名为 a[0]，元素为 a[0][0]，a[0][1]，…，a[0][N−1]；第 2 行对应数组 a[1]，元素为 a[1][0]，a[1][1]，…，a[1][N−1]；等等。对于二维数组的任意一行 a[i]，a[i]表示与该行对应的一维数组的首地址，简称该行的行地址。该行的第 j 个元素的地址为 a[i]+j。

根据从一维数组学得的知识，既然 a[i]是数组 a 的第 i 个元素，就有 a[i]等价于*(a+i)。而 a[i][j]是数组 a[i]的第 j 个元素，所以 a[i][j]等价于*(a[i]+j)，也就等价于*(*(a+i)+j)。不难看出，a[i][j]也可以写为(*(a+i))[j]。

2. 指向数组的指针

如何定义与二维数组对应的指针呢？指针类型必须与数组元素一致，因此与二维数组对应的指针是指向一个一维数组的指针，定义格式为：

类型说明(*指针名称)[整型常数]

其中，圆括号不能省略，否则为定义指针组成的数组（见 7.4.4 节）。

例如：

```
int a[3][4],(*p)[4];
p=a+1;
```

数组 a 的每一行由 4 个整数组成，指针 p 可指向由 4 个整数组成的数组，二者是一致的。语句 p=a+1 将指针 p 指向数组 a 的第 2 行。图 7.6 中给出了指针 p 与数组 a 之间的关系。

若指针 p 指向二维数组 a 的第 i 行，即 p=a+i，则 p++使 p 指向下一行，p−−使 p 指向上一行。请读者对照 7.4.1 节进行学习。

7.4.3　指向指针的指针

如果指针的值是另一指针的地址，则该指针是指针的指针，定义为：

类型说明指针变量名**

例如：

```
1   #include<stdio.h>
2   void main()
3   {   int  x=3,*p1=&x;     /*p1 指向 x*/
4       int  **p2;           /*定义指针的指针*/
5       p2=&p1;              /*p2 指向变量 p1*/
6       printf("%d",**p2);   /*输出 x 的值*/
7   }
```

这里 p2 为指向指针 p1 的指针。变量 x 可以写为*p1，也可以写为**p2，所以程序的输出是 x 的值。

7.4.4　指针数组

如果数组的每个元素都是指针，则该数组为指针数组，定义为：

类型说明*数组名[数组长度]

例如：

```
1   #include<stdio.h>
2   void main()
3   {   int  i,x[5]={1,2,3,4,5};
4       int  *p1[3];                  /*p1 为指针数组*/
5       int  **p2;                    /*p2 为指向指针的指针*/
6       for(i=0;i<3;i++)
7           p1[i]=&x[2*i];            /*为数组 p1 的各元素赋值*/
8       for(p2=p1;p2<=p1+2;p2++)      /*p2 指向数组 p1*/
9           printf("%5d",**p2);       /*输出数组 p1 的每个元素所指向的数值*/
10  }
```

程序中，指针 p2、数组 p1 及数组 x 之间的关系可用图 7.7 表示。

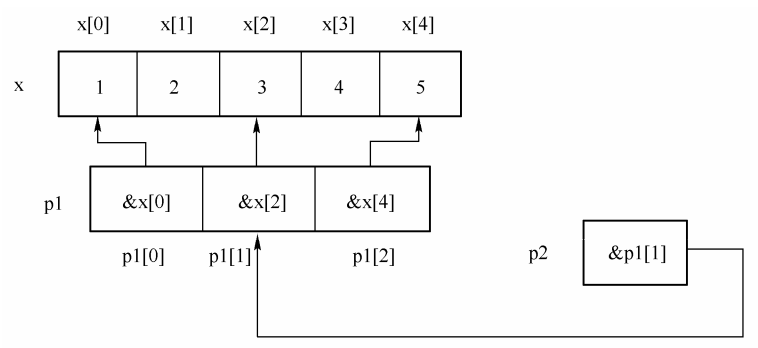


图 7.7　指针数组

7.5　指针的应用

C 语言中，常使用指针来操作数组。

【例 7.10】　数组中包含 *n* 个整数，试用指针求最大数的位置。

程序如下：

```
1   #include<stdio.h>
2   #define N 100
3   void main()
4   {   int i,n;
5       int iData[N],*p,*pMax;
6
```

```
7       printf("共有几个数（1～%d）？",N);
8       scanf("%d",&n);
9       printf("它们是：");
10      for(p=iData;p<iData+n;p++)
11          scanf("%d",p);
12
13      pMax=iData;
14      for(p=iData;p<iData+n;p++)
15          if(*p>*pMax)pMax=p;
16      i=pMax-iData+1;
17      printf("第%d 个数最大，为%d\n",i,*pMax);
18  }
```

第 7～11 行实现数据的输入，分为两步：先输入数值的实际个数，然后顺序输入这些数，并存放在数组 iData 中。第 13～16 行实现求最大值的位置（用 i 表示），指针 pMax 指向最大的数。

以下是某次运行程序的结果：

```
共有几个数（1～100）？10↙
它们是：5  3  6  8  4  7  9  0  10  2↙
第 9 个数最大，为 10
```

【例 7.11】　用选择法将数组 iData 中的数值排序。

分析：选择法排序的基本思路如下：①求第 1～*n* 个数中的最小值的位置 *i*，然后把该数与第 1 个数交换，这样，第 1 个数为最小，以后不再考虑；②求第 2～*n* 个数中的最小值的位置 *i*'，与第 2 个数交换，这样，第 2 个数为次最小，也不再考虑；③求第 3～*n* 个数中的最小值的位置 *i*"，与第 3 个数交换；以此类推。

程序如下：

```
1   #include<stdio.h>
2   #define N 100
3   void main()
4   {   int n;
5       int iTemp,iData[N];
6       int *p1,*p2,*pMin;
7
8       printf("共有几个数（1～%d）？",N);
9       scanf("%d",&n);
10      printf("它们是：");
11      for(p1=iData;p1<iData+n;p1++)
12          scanf("%d",p1);
13
14      for(p1=iData;p1<iData+n;p1++)
15      {   pMin=p1;
16          for(p2=p1+1;p2<iData+n;p2++)
```

```
17              if(*p2<*pMin)pMin=p2;
18          iTemp=*pMin;
19          *pMin=*p1;
20          *p1=iTemp;
21      }
22
23      printf("排序后: ");
24      for(p1=iData;p1<iData+n;p1++)
25          printf("%d",*p1);
26  }
```

【例 7.12】　将数组 iSrc 复制到数组 iDest 中。

程序如下：

```
1   #include<stdio.h>
2   #define N 100
3   void main()
4   {   int n;
5       int  iSrc[N],iDest[N];
6       int  *pSrc,*pDest;
7
8       printf("共有几个数（1～%d）？",N);
9       scanf("%d",&n);
10      printf("它们是: ");
11      for(pSrc=iSrc;pSrc<iSrc+n;pSrc++)
12          scanf("%d",pSrc);
13
14      pSrc=iSrc;
15      pDest=iDest;
16      while(pSrc<iSrc+n)
17      {   *pDest=*pSrc;
18          pSrc++;
19          pDest++;
20      }
21
22      printf("复制后: ");
23      for(pDest=iDest;pDest<iDest+n;pDest++)
24          printf("%d",*pDest);
25  }
```

第 14～20 行依次将数组 iSrc 的各元素复制到数组 iDest 中。可以将第 17～19 行的 3 个赋值语句合并，即将 16～20 行改为：

```
16      while( pSrc<iSrc+n )
17          *pDest++=*pSrc++;
```

【例 7.13】　求二维数组 iData[M][N]中最大元素所在的行和列。

分析：在这里将使用指向数组元素的指针，为此需要把该二维数组看做是由 M × N 个元素组成的连续序列。

对于有 m 行 n 列的二维数组，根据数组元素的存储顺序可知，若数组中的第 k 个元素为 iData[i][j]，则有 i=k/N，j=k−i*N=k%N，这里 i、j 都是整数，“%”为求余数运算。

程序如下：

```
1   #include<stdio.h>
2   #define M 3
3   #define N 4
4   void main()
5   {   int  i,j;
6       int  iData[M][N];
7       int  *p,*pMax;
8
9       printf("请按行输入%dx%d 数组的各个元素:\n",M,N);
10      for(p=&iData[0][0];p<=&iData[M-1][N-1];p++)
11          scanf("%d",p);
12
13      printf("输入的数组是: ");
14      for(p=iData[0];p<=iData[M-1]+N-1;p++)
15      {   if(((p-iData[0])%N)==0)printf("\n");     /*按行打印二维数组*/
16          printf("%5d",*p);
17      }
18
19      pMax=iData[0];
20      for(p=iData[0];p<=iData[0]+M*N-1;p++)
21          if(*p>*pMax)pMax=p;
22      i=(pMax-iData[0])/N;
23      j=(pMax-iData[0])%N;
24
25      printf("\n 最大值在第%d 行第%d 列\n",i+1,j+1);
26  }
```

程序中，将指针 p 定义为可指向任何一个数组元素。第 10 行中，分别用&iData[0][0]和&iData[M−1][N−1]表示第一个元素和最后一个元素的地址。在第 14 行中，则分别用 iData[0]和 iData[M−1]+N−1 表示，其中 iData[0]、iData[M−1]为第 1 行和最后一行的地址。在 20 行中，最后一个元素的地址表示为 iData[0]+M*N−1。

习　题　7

7.1　找出程序中的错误并改正。

（1）用起泡法排序。

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

```
#define N 10

void main()
{
    int  iData[N];
    int  *p,*q,*t;

    printf("顺序输入%d 个整数:",N);
    for(p=iData[0];p<iData+N;p++)scanf("%d",*p);
    printf("原序列是:");
    while(p<iData+N)printf("%5d",*p++);
    for(p=0;p<N-1;p++)
        for(q=p;q<iData+N;q++)
            if(*p>*q)
            {   *t=*p
                *p=*q;
                *q=*p;
            }
    printf("\n 排序后是:");
    while(p<iData+N)printf("%5d",++*p);
    printf("\n");
}
```

（2）将数组 a 按相反顺序复制到数组 b 中。

```
#include<stdio.h>
#define N 10

void main()
{
    int  a[N],b[N];
    int  *p,*q;

    printf("顺序输入%d 个整数:",N);
    for(p=a;p<=a+N-1;p++)scanf("%d",p);
    printf("\n 数组 a:");
    while(p<=a+N-1)printf("%5d",*p++);
    for(p=a,q=b+N-1;p<=a+N-1;)
        *q--=*p++;
    printf("\n 数组 b:");
    while(p<=b+N-1)printf("%5d",*p++);
}
```

7.2　程序填空。

（1）判断数组 a[N]是不是可逆的。

分析：如果数组 a 是可逆的，则第 1 个元素与倒数第 1 个元素相等，即 a[0]=a[n−1]；第

2 个元素与倒数第 2 个元素相等，即 a[1]=a[n−2]；以此类推

```
#include<stdio.h>
#define N 100
void main()
{   int  x[N],n;
    int  *p,*q;

    printf("数列中共有几个数（1～%d）？",N);
    scanf("%d",&n);
    printf("顺序输入%d 个整数：",n);
    ____①____
    while(p<=____②____)scanf("%d",____③____);

    ____④____
    q=x;
    while(q<p)
    {   if(____⑤____)break;
        ____⑥____;
        q++;
    }
    if(q____⑦____p)    printf("该数列是可逆的\n");
    else printf("该数列是不可逆的\n");
}
```

（2）判断 x 是不是数组 iData 中的最大数，注：x 可能不在数组中。

```
#include<stdio.h>
#define N 10

void main()
{
    int  iData[N];
    int  *p,*q,x;

    printf("顺序输入%d 个整数：",N);
    for(p=____①____;p<=____②____;p++)scanf("%d",p);
    printf("最大值可能是？");
    scanf("%d",&x );
    q=____③____;
    for(p=____④____;p<=____⑤____;p++)
    {   if(____⑥____)break;
        else if(*p==x)q=p;
    }
    if(q==NULL)printf("%d 不是数组中的最大数。\n",x);
    else if(____⑦____<N)printf("%d 在数组中但不是最大的\n",x);
```

```
20      else printf("%d 是最大值，是第%d 个数。\n",x,_____⑧_____);
21  }
```

7.3　判断两个数组是否相同。

7.4　统计数组中正数、负数和零的个数。

7.5　某数组中存放了 n 个整数，要删除其中的第 i（$1 \leqslant i \leqslant n$）个整数，试编程。

7.6　某数组中存放了若干个实数，编程把数组中大于阈值 x 的实数提取出来，存放在另一个数组中。

7.7　若数组是有序的（从小到大顺序排列），试用折半法查找数值 x 是否存在。

提示：用指针 p 指向所查找区域的开始，指针 q 指向的结束，指针 r 指向 p、q 之间的中间，若*r=x 则找到了，否则若*r>x，则令 p=r，若*r<x，则令 q=r，重复这一过程。

7.8　求 n 阶方阵 $\boldsymbol{A}=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$ 的转置矩阵。

提示：n 阶方阵 $\boldsymbol{A}$ 的转置矩阵是将 $\boldsymbol{A}$ 的元素 a_{ij} 和元素 a_{ji}（$1 \leqslant i$、$j \leqslant n$）互换得到的，因此 $\boldsymbol{A}$ 的转置矩阵为 $\begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}$。

7.9　某数组中存放了 n 个整数，将前面的各数向前推进 m 个位置，后面的 m 个数移到前面，如图 7.8 所示，试编程输出最后的数组。

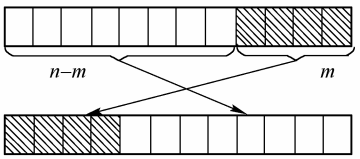


图 7.8　数字的移动

第 8 章　字　符　串

学习目标

通过本章学习，应该掌握：

- 字符串的概念及存储方法
- 存放字符串的字符数组、指向字符串的指针
- 字符串处理函数
- 字符串的应用

8.1　字符串的概念

8.1.1　字符与字符串

在计算机中，任何数据都是以二进制位形式存储和处理的，字符数据用对应的字符编码表示。随着计算机的广泛应用，计算机字符编码方案也在不断发展。在 Windows 平台上，常见的编码方案有 ASCII、GB-2312、GBK、BIG5、Unicode 等。不同编码方案中，字符个数、编码方法、字符编码都不尽相同，从而为计算机数据处理带来了不便。

本书只讨论基于 ASCII 编码的字符处理方法。

在 ASCII 编码中，每个字符用 1 个字节（8 个二进制位）表示，因此共有 2^8=256 个符号，它们的编码依次为 0～255，用十六进制表示为 0x00～0xFF。根据能否在屏幕上显示，可将这些字符分为两类——可打印字符和不可打印字符。根据以前的学习我们已经知道，在 C 语言程序中，可打印字符可以直接书写，而不可打印字符可以用对应的转义字符表示，例如用 '\n' 表示回车符。

在内存中，一个 ASCII 码字符占 1 个字节，因此可把字符处理看做是字节处理。计算机所处理的字符数据有时是分散的单一字符，如'a'、'0'、'\n'、'\x20'等，但更多的时候是若干字符的组合，如"Beijing"、"One World One Dream"等。通常把若干字符的组合看做一个整体，称为字符串。

8.1.2　字符串的存储方法

字符串由若干字符组成，在内存中占据相邻的存储单元。C 语言中，程序中的字符串使用双引号作为定界符，字符串以字符 '\0' 为结束标识，字符 '\0' 并不显式地出现在字符串中。图 8.1 给出了字符串"Olympic games"的存储格式。

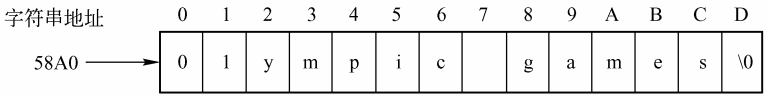


图 8.1　字符串"Olympic games"的存储方式

在内存中，字符串的首地址是该字符串的第一个字符的地址，其他字符按顺序依次存放在第一个字符的后边，最后是字符串的结束标识'\0'。字符串"Olympic games"包含 13 个字符，所占内存为 14 个字节。

该怎样理解字符串结束符呢？如果把图 8.1 中的第 8 个字符空格改为 '\0'，则内存中存放的字符串是"Olympic"，这个 '\0' 将作为新的结束符。

8.2　字符数组与指针

8.2.1　字符数组

字符数组是类型为字符的数组，其定义格式为：

char 数组名[数组长度]

可见，与其他类型的数组相比，字符数组的定义并没有特殊之处。字符数组中存放的字符可以组成字符串，此时至少有一个元素为 '\0'，从左边算起的第一个'\0'是该字符串的结束符。当然，字符数组中可以没有 '\0' 元素，也就不能按字符串进行处理了。

字符数组的初始化可采用下面的方法：

- char a[ ]={'C', ' ', 'L', 'a', 'n', 'g', 'u', 'a', 'g', 'e', '\0'};
- char b[ ]={67, 32, 76, 97, 110, 103, 117, 97, 103, 101, 0};
- char c[ ]={"C Language"};
- char d[ ]="C Language";

数组 a、b、c 和 d 是等价的，都存放了字符串"C Language"。但要注意，对于数组 a 和 b，最后一个元素是 '\0'。如果缺少了这个元素，不仅数组长度减少了 1，而且更严重的是数组的内容就不能看做字符串了。

若某字符数组中存放了一个字符串，程序中通常只会关心字符串的长度，而较少考虑数组长度。当然要存放一个字符串，数组长度必须足够。

下面的字符数组中存放的不是字符串，不能当做字符串处理：

- char x[ ]={'L', 'a', 'n', 'g', 'u', 'a', 'g', 'e'};
- char y[5]="world";

尽管数组 y 的长度小于字符串"world"，C 语言编译器通常也不会发现这个“错误”，但由于字符串"world"的结束符没有存入字符数组 y 中，程序运行时可能发生意想不到的错误。

【例 8.1】　求字符串的长度。

分析：字符串的长度是字符串中位于结束标识 '\0'（对应整数 0）之前的所有字符的个

数，也就是从数组的第一个元素开始到第一个 0 元素为止所有非 0 元素的个数。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   char str[]="Beijing Olympic games";
4       int i,iLength=0;
5       while(str[iLength]!='\0')iLength++;
6       printf("Length of string:\n");
7       for(i=0;i<iLength;i++)printf("%c",str[i]);
8       printf("\n is %d\n",iLength);
9   }
```

程序中，第 5 行的作用是求字符串长度（非 0 元素个数），第 7 行的作用是以字符方式输出字符串。

8.2.2　字符串的输入和输出

尽管可以以单字符方式输入和输出字符串，但使用起来非常不方便。C 语言专门为字符串的输入和输出定义了一系列库函数，常见的有 scanf 和 printf、gets 和 puts 等。

1．scanf函数和printf函数

使用 scanf 函数输入字符串的一般格式为：

scanf("%s",字符数组)

其中，%s 为字符串描述符，与之对应的输入项是字符数组的首地址。

例如：

```
char x[10];
scanf("%s",x)
```

其功能是从键盘缓冲区中读取一个字符串，并存入数组 x 中。不同于输入其他类型的数据，作为输入参数的数组名 x 的前面不能再有取地址运算符“&”。

值得注意的是，我们已经知道 scanf 函数使用空白（空格、制表符、回车符等）作为输入数据的分隔符，所以用 scanf 函数只能输入“单词”字符串：例如：

```
char a[10],b[10],c[10];
scanf("%s%s%s",a,b,c);
```

运行程序时，如果输入的是（“_”表示空格）：

```
_ _I'm_a_ _ student._ _
```

则数组 a、b、c 的内容分别是"I'm"、"a"、"student."，其中不包含任何空格。

printf 函数的用法与 scanf 函数类似，但输出参数可以是字符串常量。使用 printf 函数输出字符串的一般格式为：

printf("%s",字符数组名或字符串常量)

这里，与格式描述符%s 对应的仍然是数组名，与其他描述符不同。请读者对比%d、%f 等描述符的用法进行分析。

例如：

```
char  x[10]="Hi,all.\n";
printf("%s",x);
printf("%s","Hello.\n");
```

其中，x 是存放了一个字符串的字符数组。这里特别强调数组 x 中存放的是字符串，也就是必有一个元素为 '\0'，否则用%s 描述符输出会出现问题。

【例 8.2】　若从键盘输入的字符只有字母和空格，最后输入的是一个空格和字符“#”，试统计一共输入了几个单词（不包含字符“#”）。

分析：使用 scanf 函数读取每一个单词，每读取一个单词，计数一次，最后读入的单词为“#”。

程序如下：

```
1   #include<stdio.h>
2   #define N 20
3   void main()
4   {   int  i,word[N];
5       printf("输入一句话，以#结束：\n");
6       i=0;
7       scanf("%s",word);        /*先读一次，以确定是否只有“#”*/
8       while(word[0]!='#')      /*最后一个单词为“#”*/
9       {   i++;
10          scanf("%s",word);    /*度下一个单词 */
11      }
12      printf("共输入了%d 个单词\n",i);
13  }
```

为了容纳较长的单词，程序的第 2 行把存放单词的数组的长度定义为 20。以下是程序的运行结果：

```
输入一句话，以 # 结束:
Beijing  Olympic  games  start  at  8  aug,  2008  #↙
共输入了 8 个单词
```

程序中并未存储输入的单词，而是立即舍弃了，程序结束前，数组 word 中存放的是最后输入的单词“#”。

2．gets函数和puts函数

不同于 scanf 函数，使用 gets 函数可输入一句话（包含空格、制表符），其语法为：

gets(字符数组)

存入字符数组的是一个字符串（以 '\0' 结束），但不包括回车符。例如：

```
char s[100];
gets(s);
```

若输入是（“_”为空格）：

```
_ _I'm_a_sport_ _
```

则 s 的内容为 "_ _I'm_a_sport_ _"，显然与 scanf 函数有很大不同。

与 printf 函数不同，使用 puts 函数输出字符串，会在结束时自动换行，其语法为：

puts(字符数组或字符串常量)

【例 8.3】 输入一句话，统计字母的个数。

分析：方法是，将输入的句子存放在一个字符数组中，然后从第 1 个字符开始做逐一判断，以确定该字符是不是字母，直至句子结束。

程序如下：

```
1   #include<stdio.h>
2   #define N 256
3   void main()
4   {   char str[N];
5       int i,iCount=0;
6       puts("输入一句话：");
7       gets(str);
8       for(i=0;str[i]!=0;i++)
9           if(('A'<=str[i]&&str[i]<='Z')||('a'<=str[i]&&str[i]<='z'))iCount++;
10      puts("输入的句子是：");
11      puts(str);
12      printf("其中包含%d 个字母。\n",iCount);
13      }
```

第 8～9 行程序实现了字母个数的统计功能，循环的结束条件是数组元素为 '\0'。

程序的运行结果如下：

```
输入一句话:
I'm a sport,I like Beijing,I like Olympic.↙
输入的句子是:
I'm a sport,I like Beijing,I like Olympic.
其中包含 32 个字母。
```

不妨请大家想一想，能否将第 12 行的 printf 函数改用 puts 函数呢?

C 语言提供了另一个类似于 printf 的函数——sprintf 函数，可将要输出到终端的内容先以字符串方式存放在字符数组中，以备以后输出到终端。sprintf 函数的用法与 printf 函数基本相同，只是多了一个存放结果的字符数组，其一般格式如下：

sprintf(字符数组名,格式控制字符串,输出表列)

【例 8.4】　sprintf 函数的使用

```
#include<stdio.h>
void main()
{   int   x=5;
    float  y=3.4567;
    char  strBuffer[256];
    sprintf( strBuffer,"x=%3d\ty=%5.2f",x,y);/*将输出存入字符数组*/
    puts(strBuffer);                          /*输出字符串并换行 */
}
```

程序的运行结果如下（“_”代表空格）：

```
x=_ _5_ _ _ y=_3.46
```

8.2.3　字符指针

字符指针是指向字符串的指针，定义为：

```
char  *指针;
```

可以用字符数组为指针赋值，例如：

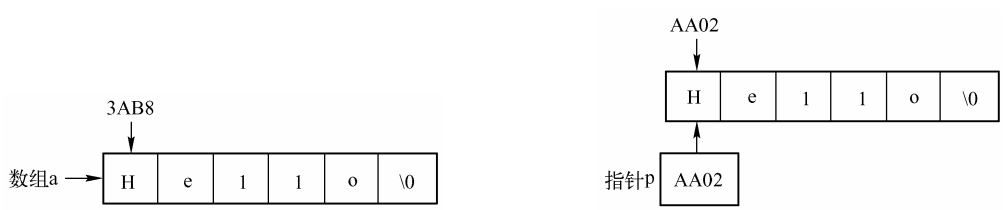
```
char  a[]="Welcome to Beijing.";
char  *p=a;
```

也可以用字符串常量的地址为指针赋值，例如：

```
char  *p="Hello";
```

此时，指针 p 指向字符串"Hello"，而不是存放"Hello"，指针与字符串之间的关系可用图 8.2 表示。由于 p 为指针变量，变量的值（指针指向）是可以改变的，例如：

```
char  *p="Hello",*q=p;
p="world";
```



（a）char a[]="Hello";　　（b）char *p="Hello";

图 8.2　字符数组和字符指针的关系

值得说明的是，如果 x 为字符数组，则令 x="abc"是错误的。

【例 8.5】　截取并输出任意字符串的前 *n* 个字符。

分析：若字符数组中存放了一个字符串，要截取前 *n* 个字符，只需将字符数组的第 *n* 个字符置为 '\0' 即可。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   char  cData[256];
4       char  *p=cData;          /*p 指向字符数组 cData*/
5       int  n;
6
7       puts("请输入一串字符：");
8       gets(p);                  /*输入的字符串存放在数组 cData 中*/
9       puts("要截取前几个字符？");
10      scanf("%d",&n);
11      if(n<0||n>256)
12      {   puts("输入参数错误！");
13          exit(0);
14      }
15      p=p+n;
16      *p='\0';                  /*将第 n 个字符置为 0*/
17      printf("截取结果为：%s",cData);
18  }
```

想一想，如果将第 4 行的初始化操作去掉，会发生什么问题呢？由于指针 p 未初始化，将导致程序错误。可见，程序中存放字符串离不开字符数组，因此，字符指针通常是跟字符数组一起使用的，除非程序中只有字符串常量。

程序的第 15～16 行可以改为“*(p+n)=0;”或者“p[n]=0;”结果都一样。

另外一个问题请大家来回答，如果输入的字符串的长度比 *n* 小，结果怎样？

【例 8.6】　输入一个字符串，然后把其中的大写字母转化为小写字母，输出转化后的字符串。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {   char  cData[256];
4       char  *p,*q=cData;
5       int  i=0;
6
7       printf("输入一串字符：");
8       gets(q);                      /*输入的字符串存放在字符数组 cData 中*/
9       for(p=q; *p!='\0'; p++)       /*用指针 p 逐一访问字符串的各个字符*/
10          if(*p>='A'&&*p<='Z')
11          {   i++;                  /*统计大写字母的个数 */
```

```
12              *p=*p+'a'-'A';       /*将大写字母转化为小写字母 */
13          }
14
15      printf("转换后为: %s\n",q);
16      printf("共转换了%d 个大写字母\n",i);
17  }
```

8.2.4　字符串数组

1. 字符串数组

生活中要处理的字符串通常不止一个，例如，某班同学的姓名、某本书的词汇表、某文档中的语句行等，这就需要使用字符串数组了。

字符串数组是存放了若干字符串的字符数组，字符串数组是一个二维数组。通常每个字符串占据一行，字符串的个数对应数组的行数，最长的字符串的长度对应数组的列数。因此，字符串数组的定义为：

char　数组名[字符串个数][字符串长度]

要引用字符串数组中的字符串，可使用行地址。第 *i* 个字符串的地址为：**数组名**[*i*]，也可写为：*(**数组名**+*i*)。

【例 8.7】　某班有 3 名同学参加考试，求成绩最高的学生的姓名。

分析：程序要处理的数据包括学生姓名、考试成绩。程序的核心是确定 10 个人谁的成绩最高，也就是求最大值问题。

程序如下：

```
1   #include<stdio.h>
2   #define N 3
3   void main()
4   {   char  sName[N][10]; /*假设姓名最多占 10 个字节(最多 4 个汉字或 9 个字母)*/
5       int  i,t,iScore[N];
6       printf("顺序输入每个同学的姓名和成绩(用空格分开): \n");
7       for(i=0;i<N;i++)
8           scanf("%s%d",sName[i],&iScore[i]);
9       t=0;
10      for(i=1;i<N;i++)        /*t 为最大值的下标*/
11          if(iScore[i]>iScore[t])t=i;
12      printf("%s 的成绩最好，为%d\n",sName[t],iScore[t]);
13  }
```

程序的运行结果如下：

```
顺序输入每个同学的姓名和成绩(用空格分开):
王刚  87↙
```

```
黎明　95↙
孙国民　　85↙
黎明的成绩最好，为 95
```

程序第 4 行假设所有同学的姓名不会超过 4 个汉字或 9 个字母，因而字符串数组 sName 的列宽为 10。

注意程序的第 8 行，引用 scanf 函数时，sName[i]是与第 i 个同学的姓名对应的行地址，不需要取地址运算符“&”，而 iScore[i]是该同学的成绩，用一个整数表示，需要使用取地址运算符“&”。第 12 行的语法也是同样道理。

***2．指向字符串数组的指针**

定义指向字符串数组的指针的方法是：

```
char(*指针)[字符串长度]
```

其中，“字符串长度”必须与字符串数组相同。

【例 8.8】　指向字符串数组的指针。

```
1   #include<stdio.h>
2   void main()
3   {   char sFruit[5][10]={"Apple","Pear","Banana","Peach","Plum"};
4       char  (*p)[10];      /*p 为指向字符串的指针
5       p=&sFruit[1];        /*p 指向第 2 行，即“Pear”*/
6       puts(*p);            /*输出 p 所指向的字符串*/
7       p=sFruit+3;          /*p 指向第 4 个字符串，即“Peach”*/
8       printf("%s",p);      /*输出 p 所指向的字符串*/
9   }
```

C 语言中有许多难于理解的地方，程序中的第 6 行、第 8 行就是一例，有兴趣的读者可以研究一下。

8.2.5　字符指针的数组

如果数组元素是字符指针，每个指针指向一个字符串，则该数组为字符指针的数组，定义为：

```
char  *指针[数组长度]
```

【例 8.9】　字符指针的数组。

```
1   #include<stdio.h>
2   void main()
3   {   char  *a="Beijing",*b="Shanghai";
4       char  *pArray[]={a,b};
5       int  i;
6       for(i=0;i<2;i++)
```

```
7           puts(pArray[i]);
8   }
```

程序中，pArray 为指针数组，有两个指针元素，分别指向由指针 a 和指针 b 对应的字符串。

请大家想一想，如果 p 的定义为 char *p[2]，则以下语句哪些是正确的?

① gets(p[0]);

② gets(*p);

③ gets(&p[0]);

语句①是错误的，因为 p[0]所指向的地址没有初始化，也就是“无指向”。语句②也是错误的，因为 p 是数组的首地址，*p 也就是 p[0]，错误原因与①相同。语句③还是错误的，因为&p[0]是数组的首地址，也就是 p，该数组是指针数组，不是字符串数组，不能用 gets 函数赋值。

8.3　字符串处理函数

C 语言函数库提供了大量的字符串处理函数，极大地丰富了处理字符串的手段，减缓了程序设计的劳动强度，也提高了程序的执行效率。

本节将详细介绍几种常用的字符串处理函数。需要说明的是，字符串处理函数的原型定义在头文件 string.h 中，因此，使用这些函数的时候，程序的开始位置需要书写：

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

8.3.1　复制与连接

1．Strcpy函数和strncpy函数

两个函数的一般形式是：

```
strcpy(字符数组 1,字符串 2)
strncpy(字符数组 1,字符串 2,整数 n)
```

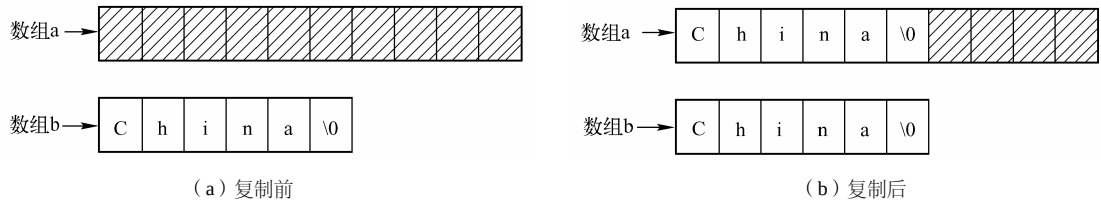
strcpy 函数的功能是将“字符串 2”对应的字符串全部复制到“字符数组 1”中，结果两个字符串相同。strncpy 函数最多将“字符串 2”的前 *n* 个字符复制到“字符数组 1”中，若“字符串 2”为"I'm a boy"，而 *n* 为 3，则仅复制"I'm"到“字符数组 1”中，并将第 4 个字符置为'\0'。

显然，“字符数组 1”的长度必须足够容纳复制后的字符串。“字符串 2”可以是存放了字符串的字符数组，也可以是字符串常量，或者指向字符串的指针。

例如：

```
char a[10],b[]="China";
strcpy(a,b);
```

执行后，字符数组 a 的内容为字符串“China”，如图 8.3 所示，最后 4 个字符位于字符串结束符 '\0' 之后，通常不会被处理，保持原样。



（a）复制前　　（b）复制后

图 8.3　strcpy 函数的使用

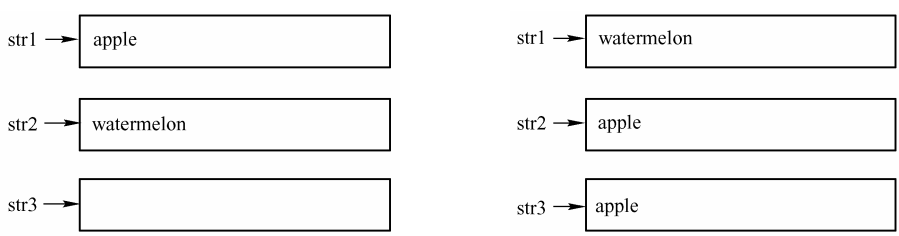
根据所学知识，由于 a 和 b 是数组名，是不能用赋值运算符完成复制操作的。使用表达式 a=b 将数组 b 中的字符串复制给数组 a 是非法的。

【例 8.10】　交换两个字符串。

方法一：程序中采用字符数组存放字符串。

```
1   #include<stdio.h>
2   #include<string.h>
3   #define N 40
4   void main()
5   {   char str1[N]="apple",str2[N]="watermelon";
6       char str3[N];
7
8       printf("(%s,%s)",str1,str2);
9       strcpy(str3,str1);
10      strcpy(str1,str2);
11      strcpy(str2,str3);
12      printf("=>(%s,%s)",str1,str2);
13  }
```

程序第 9～11 行实现了字符数组的内容互换，可用图 8.4 表示。



（a）交换前　　（b）交换后

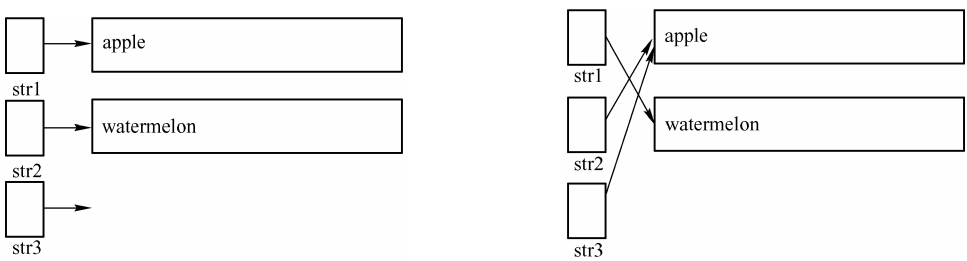
图 8.4　交换字符数组中的字符串

方法二：程序中采用字符指针指向字符串。

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

```
3   void main()
4   {   char*str1="apple",*str2="watermelon";
5       char*str3;
6
7       printf("(%s,%s)",str1,str2);
8       str3=str1;
9       str1=str2;
10      str2=str3;
11      printf("=>(%s,%s)",str1,str2);
12  }
```

在该程序中，交换的并不是字符串本身，而是指向两个字符串的指针，如图 8.5 所示。



（a）交换前　　（b）交换后

图 8.5　交换指向字符串的指针

有人会说，方法二的程序比方法一简单，效率高。此话不假，但方法一很容易扩充到对任意（从键盘输入）的两个字符串的处理，而方法二就不得不再定义其他字符数组了。

方法三：综合了方法一和方法二的优点，程序如下：

```
1   #include<stdio.h>
2   #include<string.h>
3   #define N 20
4   void main()
5   {   char  str1[N],str2[N];
6       char  *ps1=str1,*ps2=str2;
7       char  *ps3;
8
9       printf("输入两个字符串，用空格分开：");
10      scanf("%s%s",ps1,ps2);
11      ps3=ps1;
12      ps1=ps2;
13      ps2=ps3;
14      printf("交换后为：%s %s",ps1,ps2);
15  }
```

当然，字符数组 str1 和 str2 的内容在交换前后并没有改变，因此第 14 行若改为：

```
printf("交换后为：%s %s",str1,str2);
```

将输出错误的结果。

2．Strcat函数和strncat函数

两个函数的一般形式为：

strcat(字符数组 1,字符串 2)
strncat(字符数组 1,字符串 2,整数 *n*)

strcat 函数将“字符串 2”附加在“字符数组 1”中的字符串的后边，也就是把两个字符串连接起来再存入“字符数组 1”，因此，strcat 函数的功能是连接两个字符串。而 strncat 函数仅把“字符串 2”的前 *n* 个（不足 *n* 个，则为实际个数）字符连接到“字符数组 1”的字符串后边。

例如，对于：

```
char a[80]="I'm a",b[]="student",c[]="sport from China";
```

执行 strcat(a, b)后 a 的内容为"I'm a student"，而执行 strncat(a, c, 5)后 a 的内容为"I'm a sport"。

显然，“字符数组 1”中的字符串结束符在连接“字符串 2”后将被丢弃。

8.3.2 比较大小

字符串的大小由字符串中的字符的编码决定。对于任意两个字符串，从它们的第一个字符开始比较，若相同，则继续比较下一个字符，直到出现不同的字符或者都遇到结束符。若存在不同的字符，则编码较大的字符所在的字符串较大。如果同时遇到结束符，则两个字符串相等。例如，"ABC"小于"AbC"，"ABC"大于"AB"，"123"等于"\x31\x32\x33"，等等。

C 语言的关系运算符不能用来比较字符串的大小，要比较两个字符串的大小，只能使用库函数。

1．strcmp函数、strncmp函数和stricmp函数

这些函数的一般形式是：

strcmp(字符串 1,字符串 2)
strncmp(字符串 1,字符串 2,整数 *n*)
stricmp(字符串 1,字符串 2)

这些函数的函数值都是整型的，如果函数值为 0，表示两个字符串相等；如果函数值大于 0，则“字符串 1”大于“字符串 2”；如果函数值小于 0，则“字符串 1”小于“字符串 2”。

strcmp 函数是最常用的函数之一，比较时要从左到右逐一比较每个字符，直到遇到不同的字符或遇到字符串结束符为止。strncmp 函数只比较两个字符串的前 *n* 个字符，因此 strncmp("ABC", "AB", 2)的返回值为 0，表示"ABC"和"AB"相等。用 stricmp 函数比较两个字符串时，将忽略字母符号的大小写，因此"ABC"与"AbC"相等。

【例 8.11】　求若干字符串中的最大字符串。

分析：实际上，这也是一个求最大值问题，与求若干个数中的最大数的方法是一致的。

程序如下：

```
#include<stdio.h>
#include<string.h>
#define N 10
void main()
{   char  strWord[N][20];        /*存放输入的 N 个字符串*/
    int  i,iMax;
    /*输入*/
    printf("顺序输入%d 个字符串，用空格分开：\n",N);
    for(i=0;i<N;i++)scanf("%s",strWord[i]);
    /*加工*/
    iMax=0;
    for(i=1;i<N;i++)
        if(strcmp(strWord[i],strWord[iMax])>0)iMax=i;
    /*输出*/
    printf("第%d 个字符串最大，为：%s",iMax+1,strWord[iMax]);
}
```

2．strstr函数

strstr 函数的一般形式为：

strstr(字符串 1,字符串 2)

strstr 函数的功能是在“字符串 1”中查找“字符串 2”，如果找不到，则函数返回空指针 NULL，若找到了，则返回与“字符串 1”中所包含的“字符串 2”的首字符对应的指针。如果“字符串 1”中包含“字符串 2”，那么“字符串 2”常被称做“字符串 1”的子串。求子串是实现查询（检索）的基础。

【例 8.12】　判断一个字符串中是否包含一个子串。

程序如下：

```
#include<stdio.h>
#include<string.h>
void main()
{   char*pSrc="one world,one dream.";
    char*ps1="World",*ps2="one";
    char*p;
    printf("字符串：%s\n",pSrc);
    if((p=strstr(pSrc,ps1))==NULL)
        printf("未找到子串%s\n",ps1);
    else printf("子串%s 在第%d 位\n",ps1,p-pSrc+1);
    if((p=strstr(pSrc,ps2))==NULL)
```

```
12          printf("未找到子串%s\n",ps2);
13      else printf("子串%s 在第%d 位\n",ps2,p-pSrc+1);
14  }
```

程序的运行结果为：

```
字符串: one world,one dream.
未找到子串 World
子串 one 在第 1 位
```

8.3.3　变换

1．strlwr函数和strupr函数

两个函数的一般用法是：

```
strlwr(字符数组)
strupr(字符数组)
```

strlwr 函数将存放在“字符数组”中的字符串里的所有大写字母转化为小写字母，而 strupr 函数则相反，将小写字母转化为大写字母。转换后得到的字符串仍存放在“字符数组”中。

例如，对于 char s[]="thiS IS A samPle"，strlwr(s)将 s 转化为"this is a sample"，而 strupr(s)将 s 转化为"THIS IS A SAMPLE"。

2．atoi函数和atof函数

这两个函数的原型在 stdlib.h 中，一般用法是：

```
变量=atoi(字符串)
变量=atof(字符串)
```

atoi 函数将“字符串”转化成整数，并作为函数值返回，而 atof 函数则转化为实数。

【例 8.13】　任意输入一个加法算式，格式为“整数 1+整数 2=”，例如“23+32=”，试完成计算并打印出结果。注：算式中，开始位置、“+”和“=”的前后都可以有空格。

分析：输入时，如果“+”的后面必有空格，那么使用形如“scanf("%d+%d=", &a, &b)”的语句即可获得正确的输入。如果“+”的后面没有空格，那么用“scanf("%d%1s%d=", &a, s, &b)”（s 为字符数组名）也可以获得正确的输入。但这些方法都不能处理非法的输入。

本例的方法是：

① 将输入的“算式”存放在字符数组中；

② 用 atoi 函数提取字符串中的第 1 个加数，若提取失败则显示出错并中断程序运行；

③ 截取剩余的字符组成新的字符串；

④ 判断新的字符串的第一个非空格字符是否为“+”，如果不是，则显示出错并中断程

序运行；

⑤ 截取“+”后的字符串；

⑥ 用 atoi 函数提取第 2 个加数，若失败则显示出错并中断程序运行；

⑦ 截取剩余的字符串；

⑧ 若剩余字符串中的只有一个非空格字符“=”，则完成计算，否则显示出错。

程序如下：

```
#include<stdio.h>
#include<stdlib.h>
void main()
{   int a,b,c;
    char s[20],*p=s;
    printf("输入一个算式（形如“a+b=”）:");
    gets(p);
    if((a=atoi(p))==0)          /* atoi 函数的返回值为 0，表示转换失败*/
    {   printf("错误：不能提取整数\n");
        exit(0);
    }
    while(*p==''||(*p>='0'&&*p<='9'))p++;     /*跳过所有空格和数字*/
    if(*p!='+')
    {   printf("错误：请输入“+”\n");
        exit(0);
    }
    p++;
    if((b=atoi(p))==0)          /*atoi 函数的返回值为 0，表示转换失败*/
    {   printf("错误：不能提取整数\n");
        exit(0);
    }
    while(*p==''||(*p>='0'&&*p<='9'))p++;     /*跳过所有空格和数字*/
    if(*p!='=')
    {   printf("错误：请输入“=”\n");
        exit(0);
    }
    printf("%d\n",a+b);
}
```

8.3.4　其他函数

用 strlen 函数可计算字符串的长度，用法是：

变量=strlen(字符串)

strlen 函数的函数值是一个整数，表示“字符串”的长度，即结束符 '\0' 前的字符的个数。

与 sizeof(字符串)不同，用 strlen 函数求得的并不是数组长度（所占内存的字节数）。例如，对于 char s[10]="apple"，表达式 sizeof(s)的值为 10，而 strlen(s)的值为 5。

8.4　字符与字符串的应用

【例 8.14】 输入两个字符串，然后把它们连接起来形成一个字符串，再输出到屏幕上。

方法一（使用 strcat 函数）：

```
#include<stdio.h>
#include<string.h>
void main()
{   char s1[256],s2[256];
    puts("输入 2 个字符串：");
    gets(s1);
    gets(s2);
    strcat(s1,s2);
    printf("连接后为：%s\n",s1);
}
```

方法二（使用数组）：

```
#include<stdio.h>
void main()
{   char s1[256],s2[256];
    int i=0,j=0;
    puts("输入 2 个字符串：");
    gets(s1);   gets(s2);
    while(s1[i]!='\0')i++;          /*定位结束符位置*/
    while(s2[j]!='\0')              /*逐一复制 s2 中的字符*/
    {   s1[i]=s2[j];
        i++;j++;
    }
    s1[i]='\0';                     /*s1 中的字符串结束*/
    printf("连接后为：%s\n",s1);
}
```

程序的第 7 行用来定位结束符的位置，第 8～11 行将 s2 的字符逐一复制到 s1 中，第 12 行的作用是设置 s1 中字符串结束标志。

程序的第 9～11 行可以合并为“s1[i++]=s2[j++];”。进而，可以将 8～12 行合并为“while((s1[i++]=s1[j++]) != '\0');”，甚至进一步缩写为“while(s1[i++]=s1[j++]);”，请大家自己分析。

方法三（使用指针）：

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

```
3   {   char s1[256],s2[256];
4       char*p1=s1,*p2=s2;
5       puts("输入 2 个字符串：");
6       gets(s1);gets(s2);
7       while(*p1!='\0')p1++;
8       while(*p2!='\0')
9       {   *p1=*p2;
10          p1++;p2++;
11      }
12      *p1 = '\0';
13      printf("连接后为：%s\n",s1);
14  }
```

与方法二的分析相同，程序的第 8～12 行可以合并为“while(*p1++=*p2++);”。

【例 8.15】　将字符串倒置后再打印出来。

分析：不同于数值数组的倒置，对于存放了字符串的字符数组，倒置仅仅是对结束符前的字符进行的，因此若有 char s[10]="apple"，倒置后 s 中存放的字符串应为"elppa"，如图 8.6 所示。

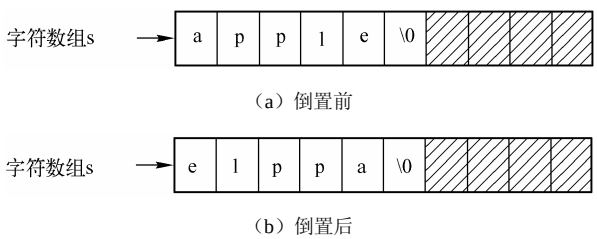


图 8.6　字符串的倒置

方法一（使用数组元素下标）：

```
1   #include<stdio.h>
2   void main()
3   {   char  strTemp[256],ch;
4       int  i,j;
5       printf("输入一串字符：");
6       gets(strTemp);
7       i=j=0;                          /*i 为第 1 个字符的下标*/
8       while(strTemp[j]!='\0')j++; /*获取'\0'的下标*/
9       j--;                            /*j 为'\0'前的最后一个字符的下标*/
10      while(i<j)                      /*交换前后两个字符*/
11      {   ch=strTemp[j];
12          strTemp[j]=strTemp[i];
13          strTemp[i]=ch;
14          i++;
15          j--;
```

```
    }
    printf("倒置后为：%s\n",strTemp);
}
```

方法二（使用指向字符串的指针）：

```
#include<stdio.h>
void main()
{   char  strTemp[256],ch;
    char  *p1,*p2;
    printf("输入一串字符：");
    gets(strTemp);
    p2=p1=strTemp;                    /*p1 指向第 1 个字符*/
    while(*p2!='\0')p2++;             /*p2 指向'\0'*/
    p2--;                             /*p2 指向'\0'前的最后一个字符*/
    while(p1<p2)                      /*交换 p1、p2 所指向的字符*/
    {   ch=*p1;
        *p1=*p2;
        *p2=ch;
        p1++;
        p2--;
    }
    printf("倒置后为：%s\n",strTemp);
}
```

【例 8.16】 将输入的若干单词按字典顺序排序，输出排序结果。

分析：数据排序的主要操作是通过交换数据改变数据的顺序，对字符串的排序也是如此。在 8.3.1 节的例 8.10 中讨论了两种交换字符串的策略，在这里将被应用于排序过程。

另外一个问题是字符串的输入。根据前面的学习，在输入单词时，使用 scanf 函数会比使用 gets 函数方便。

方法一：采用直接交换字符数组中的字符串的方法实现排序，如图 8.7（a）所示。

程序如下：

```
#include<stdio.h>
#include<string.h>
#define N 10
#define M 10
void main()
{   char sData[N][M],sTemp[M];
    int i,j,k,n;
    printf("输入字符串的个数(1～%d)：",N);
    do  scanf("%d",&n);
    while(n<0||n>N);
    printf("依次输入%d 个单词，用空格分隔：",n);
    for(i=0;i<n;i++)
```

```
13          scanf("%s",sData[i]);
14      /*用选择法排序*/
15      for(i=0;i<n-1;i++)
16      {   k=i;
17          for(j=i+1;j<n;j++)
18              if(strcmp(sData[k],sData[j])>0)k=j;
19          if(k!=i)    /*必要时交换两个字符串*/
20          {   strcpy(sTemp,sData[k]);
21              strcpy(sData[k],sData[i]);
22              strcpy(sData[i],sTemp);
23          }
24      }
25      /*输出*
26      printf("排序后：");
27      for(i=0;i<n;i++)
28          printf("%s",sData[i]);
29  }
```

程序运行结果为：

```
输入字符串的个数(1～10)：5↙
依次输入 5 个单词，用空格分隔：pear  peach  plum  apple  banana↙
排序后：apple banana peach pear plum
```

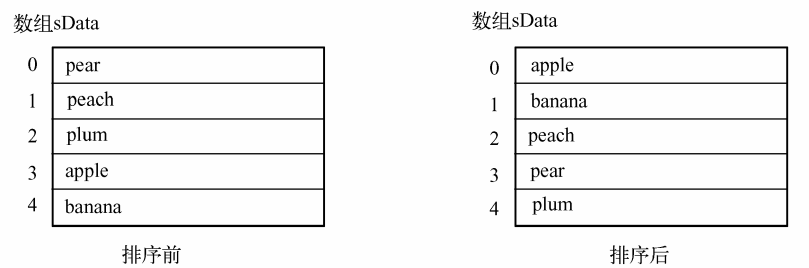
程序中采用选择法，而不是起泡法，减少了交换的次数。但由于字符串是相对复杂的数据结构，使用 strcpy 函数复制字符串比较费时，会造成程序的效率比较低。

方法二：用指针数组存放各字符串的地址，通过交换指针间接实现字符串的排序，如图 8.7（b）所示。

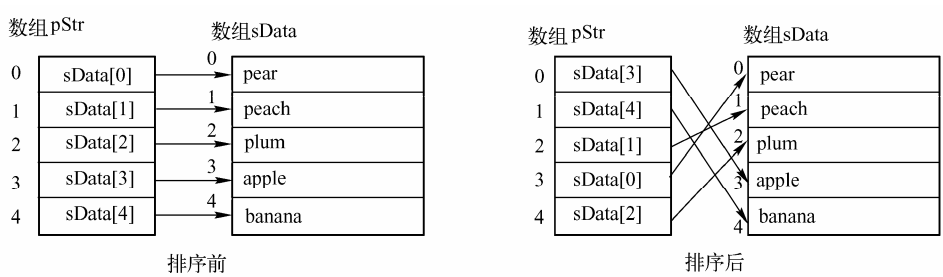
程序如下：

```
1   #include<stdio.h>
2   #include<string.h>
3   #define N 10
4   #define M 10
5   void main()
6   {   char sData[N][M];
7       char*pStr[N],*pTemp;
8       int i,j,k,n;
9       /*输入*/
10      printf("输入字符串的个数(1～%d)：",N);
11      do  scanf("%d",&n);
12      while(n<0||n>N);
13      printf("依次输入%d 个单词，用空格分隔：",n);
14      for(i=0;i<n;i++)
15      {   scanf("%s",sData[i]);       /*输入字符串*/
```

```
16          pStr[i]=sData[i];            /*存放字符串的地址*/
17      }
18      /*排序*/
19      for(i=0;i<n-1;i++)
20      {   k=i;
21          for(j=i+1;j<n;j++)
22              if(strcmp(pStr[k],pStr[j])>0)k=j;
23          if(k!=i)
24          {   pTemp=pStr[k];
25              pStr[k]=pStr[i];
26              pStr[i]=pTemp;
27          }
28      }
29      /*输出*/
30      printf("排序后: ");
31      for(i=0;i<n;i++)printf("%s",pStr[i]);
32  }
```



（a）直接交换两个字符串



（b）用指针数组实现间接排序

图 8.7　字符串的排序

程序第 16 行将输入的字符串的地址（二维数组 sData 的行地址）存放在一维数组 pStr 中，从而为这些字符串建立了索引结构。第 19～28 行的排序过程中，排序的依据是字符串的大小（根据索引获取），而不是数组 pStr 的元素的地址（为字符串的内存地址）的大小，但交换操作是针对数组 pStr 的元素的值进行的。

不同于方法一，方法二中没有字符串的交换，因而具有较高的执行效率。但方法二的算法比较复杂，使用了更复杂的数据结构，也不是完美无缺的。通常，只有在问题规模较大、

要处理的字符串较多时，才会考虑程序的效率问题。

习　题　8

8.1　程序改错。

（1）把字符串 s 中的所有字母改写成该字母的下一个字符，字母 'z' 改写成字母 'a' 。要求大写字母仍为大写字母，小写字母仍为小写字母，其他字符不做改变。

```
#include<stdio.h>
void main()
{   char  *s;
    int  i,k;
    gets(s);                    /*输入任意字符串*/

    while(s[k++]!=0);           /*k 为字符串长度*/
    for(i=0;i<k;i++)
        if('a'<=s[i]<='z'||'A'<=s[i]<='Z')
            s[i]++;
        else
            if(s[i]='z'||s[i]='Z')
                s[i]='a';

    puts(s);                    /*输出转化结果*/
}
```

（2）有近百名学生选修了某门课程，他们来源于不同的专业班级。请按班级、学号顺序打印学生的班级名称、学号和姓名。

提示：程序中包含了冒泡法排序过程，排序的依据是班级和学号，即先按班级排序，若班机相同则按学号排序。

```
#include<stdio.h>
#include<string.h>
#define  N  100
#define  LENGTH  10
void main()
{   char sName[N][LENGTH],sID[N][LENGTH],sClass[N][LENGTH];
    char sTemp;
    int i,j;

    printf("逐行输入学生的班级、学号和姓名：\n");
    for(i=0;i<N;i++)
        scanf("%s%s%s",&sClass[i],&sID[i][0],sName);
    /*冒泡法*/
    for(i=0;i<N;i++)
```

```
        for(j=i+1;j<N;j++)
            if((sClass[i]>sClass[j])||
                (sClass[i]=sClass[j])&&(sID[i]>sID[j]))
            {   sTemp=sName[i];
                sName[i]=sName[j];
                sName[j]=sTemp;
                sTemp=sID[i];
                sID[i]=sID[j];
                sID[j]=sTemp;

                strcpy(sClass[i],sClass[j]);
                strcpy(sClass[j],sClass[i]);
            }

    printf("排序后的班级、学号和姓名: \n");
    for(i=0;i<N;i++)
        printf("%s  %s  %s\n",sClass,sID,sName);
}
```

8.2　程序填空。

（1）输入一个字符串，存放在字符数组中，然后将某个字符插入到该字符串的某个位置。要求考虑输入的插入位置可能小于 0（在首部插入）或大于字符串长度（在尾部插入）等情况。

```
#include<stdio.h>
void main()
{   char s[256],*p,*q=s;
    char ch;
    int i;
    printf("输入一个字符串? ");
    gets(      ①      );
    printf("输入要插入的字符? ");
    ch=getchar();
    printf("插入在第几个字符前? ");
    scanf("%d",&i);
        ②
    while(      ③      )p++;                    /*p 指向串尾*/
    while((i-->0)&&      ④      )q++;          /*q 指向插入位置*/
    while(p>=q)
    {       ⑤      ;
        if(      ⑥      )p--;
        else  break;
    }
        ⑦
    printf("插入后的结果为: %s\n",      ⑧      );
```

```
22  }
```

（2）输入一个字符串，存放在字符数组 s1 中，然后再输入一个字符串 s2。在字符串 s1 中查找 s2，并将找到的 s2 删除，重复这一过程，直至删除了 s1 中包含的所有 s2。最后打印删除了所有 s2 的字符串 s1。

```
1   #include<stdio.h>
2   #include<string.h>
3   void main()
4   {   char s1[256],s2[256];
5       ____①____
6
7       printf("输入一串字符: ");
8       gets(s1);
9       printf("输入要查找的子串: ");
10      gets(s2);
11      ____②____
12      p1=strstr(p1,s2);
13      while(____③____)
14      {   for(q=s2;____④____;q++)  /*要删除的字符个数与 s2 的长度相同*/
15          {   p2=p1;                /*15～19 行可删除一个字符*/
16              while(____⑤____)
17              {   ____⑥____
18                  p2++;
19              }
20          }
21          ____⑦____
22      }
23      printf("删除子串后的结果为: %s\n",s1);
24  }
```

8.3　输入一句英文，然后将每个单词的第一个字母改为大写，输出最后结果。

8.4　截取某字符串的最后 *n* 个字符，并以字符串形式输出。要求不能定义第二个字符数组。

8.5　某字符串由若干“0”和“1”组成，试把它转化为十进制整数。

8.6　若输入的字符串可看做是一个十六进制整数，例如“30AeD”，试编程把它转化为十进制整数。

8.7　某字符串中包含若干数字，试将这些数字一一提取出来存放在一个字符串数组中，然后打印出这些数字。例如，从“2008 年 8 月，29 届奥运会在中国北京举行”中可提取 2008、8、29 三组数字，所提取的数字全部由数字字符组成。

8.8　在字符串数组中存放了若干单词，在一个指针数组中存放了它们的地址，试用起泡法实现对单词的排序，但不改变单词的存储顺序。

8.9　某字符串数组 s 中存放了若干字符串，若输入的命令字符串为“i5”则在 s 的第 5 行插入一空行，若输入的是“d3”则将 s 的第 3 行删除，若输入的是“q”则打印所有字符

串，并结束程序运行。试编程模拟这些操作。

提示：输入的命令的格式为：字母（i 或 d）+数字。若字母为“i”则为插入操作，若为“d”则为删除操作，若为“q”则退出程序，忽略其他字母。数字可为任何正整数。

8.10　输入两个字符串 s1 和 s2，在 s1 的指定位置插入 s2，试编程。例如，在“I'm a boy”的第 6 个字符位置插入“handsome ”，结果为“I'm a handsome boy”。

第 9 章　函　　数

学习目标

通过本章学习，应该掌握：

- 函数的定义方法、执行过程和返回值
- 函数的参数说明，实际参数和形式参数的对应关系，数值传递的含义和用法
- 程序的模块化，基于函数的结构化程序设计
- 变量的作用域
- 变量的存储类别

9.1　概　　述

本书前面的程序都很短，可以说程序的规划是相对容易的。但是专业化的应用程序往往有几万行甚至更多，试想如果这样的程序是一个不可拆分的整体，那么这样的程序如何构思？而一旦出了错误又如何调试？程序一旦要升级，又如何维护呢？

实际上，一个这样规模的程序都是需要精心规划的，比如整个程序分成多少模块，各个模块之间有怎样的连接关系等，都要事先进行规划。为了支持模块化，结构化程序设计语言中引入了子程序、子函数的概念。在 C 语言里，子程序、子函数统一称为函数。函数是 C 语言源程序的基本模块，通过对函数模块的调用实现特定的功能。

在第 1 章中就已经介绍过，C 语言源程序是由函数组成的。虽然在前面各章的程序中都只有一个主函数 main，但实际程序往往由多个函数组成。C 语言不仅提供了极为丰富的库函数，还允许程序员建立自己定义的函数。

有了函数，可以把一个庞大的问题分解，在较高层次上只考虑做什么，而把怎样做放到较低层次上考虑。通过函数的定义把实现的细节封闭起来，通过函数调用组合各种功能，就像用一个个构件盖房子似的。这也是结构化程序的基本要求。由于采用了函数模块式的结构，C 语言易于实现结构化程序设计，使程序的层次结构清晰，便于程序的编写、阅读、调试。

下面举一个例子来说明结构化程序的设计思想。

【例 9.1】　从键盘读入 10 个整数放入 a 数组，对其按照从大到小的顺序进行排序后输出。

分析：我们要做的工作如下：

① 建立数组，并分配内存；

② 从键盘读入数据，放入数组；

③ 对数组排序；

④ 输出排序后的数组。

假设将这 4 个工作分别用函数 init、readin、sort、output 完成，那么程序将变成类似于下面的形式：

```
/*下面的程序是伪程序，只是为了体现程序的流程*/
void main()
{
    init();
    readin();
    sort();
    output();
}
```

实际上，这个题目即使不用函数也是不难实现的。但是，可以看出，用函数将使得程序结构更加清晰。而且，假设有几个人一起编写这个程序，那么就可以给他们分别分配不同的模块，这样整个程序完成的时间就会缩短。

9.2　函数的定义

实际上前面已经学习了如何编写 C 语言函数——就是我们一直在使用的一个函数 main。任何函数的定义与创建形式都与 main 函数类似。

函数的定义形式如下：

```
数据类型 函数名(参数声明列表)
{
    说明与语句
}
```

其中有些部分是可以省略的。最简略的函数是下面的形式：

```
函数名(){}
```

这样的函数不做任何事情，称为空函数。实际上空函数也是有用的，例如，在程序开发过程初期可以占据一个位置，从而保持程序结构的完整性，便于程序调试。

在函数定义中，如果省略了函数的“数据类型”，则系统会隐含指定函数的类型是 int 型的。

程序实际上是一系列变量和函数的定义。函数之间进行通信，是通过函数的参数与返回值及外部变量完成的。

一个源程序可以分为若干函数，函数的定义在源程序中可以是任意的顺序。源程序可以分成多个文件。当然，一个函数不能在不同文件中定义。

函数是通过 return 语句返回给其调用者数值的。return 后面可以跟任何形式的表达式：

```
return 表达式;
```

其中，表达式可以转换成函数需要的返回类型。通常表达式还使用括号“()”，但是用不用括号是可以任选的。例如：

```
return 0;
return x;
```

```
return(x+y/2);
```

都是合法的 return 语句。

最后，需要指出的是：在 C 语言中，所有的函数定义，包括主函数 main 在内，都是平行的。也就是说，在一个函数的函数体内，不能再定义另一个函数，即不能嵌套定义。但是函数之间允许相互调用，也允许嵌套调用。习惯上把调用者称为主调函数，而把被调用者称为被调函数。函数还可以自己调用自己，称为递归调用。main 函数是主函数，它可以调用其他函数，而不允许被其他函数调用。因此，C 语言程序的执行总是从 main 函数开始，完成对其他函数的调用后再返回到 main 函数，最后由 main 函数结束整个程序。一个 C 语言源程序必须有也只能有一个主函数 main。

9.2.1　函数的命名

每个函数必须有一个名字。函数命名有一些简单的规则，这与变量的命名规则基本一致，即函数名必须以字母或下划线开头(_)，函数名其余部分可以是数字、字母或下划线。

编译器区分函数名的大小写。例如，thisFunc 和 ThisFunc 是不同的函数。

好的函数名最好能描述出函数的功能，不要让其含义含糊或者根本没有含义。例如，DisplayPicture、CloseWindow、InputUserName 等这些函数名就很好，因为这些函数无论对使用者还是维护者来说，都容易理解其含义。而像函数名 X1、G123、CCCC 等，尽管规则允许，但是其含义不清，会给使用和维护带来不便。

一个优秀的程序员是绝对不会给函数随便取名字的。

9.2.2　函数的执行

程序从调用函数的地方，跳到函数的开头执行。在函数结尾，程序返回到调用函数的地方，继续执行程序。

【例 9.2】　函数的执行顺序。

```
1   #include<stdio.h>
2   int max(int a,int b)
3   {
4       if(a>b)
5       return a;
6       else
7       return b;
8   }
9   void main()
10  {
11      int  x,y,z;
12      x=5;
13      y=8;
14      z=max(x,y);
```

```
15      printf("maximum=%d",z);
16  }
```

程序的运行结果如下：

```
maximum=8
```

程序的第 2～8 行为 max 函数定义，第 9～16 行为 main 函数的定义。习惯上，main 函数是程序的最后一个函数，当然也可以放在其他地方，以后再进行讨论。

程序执行都是从主函数 main 开始的，不管其所在的位置如何。程序第 14 行为调用 max 函数，并把 x、y 中的值传送给 max 的参数 a、b，然后开始执行从第 4～7 行的 max 函数的函数体，max 函数执行的结果（a 或 b）将返回给 main 函数的变量 z，最后由 main 函数第 14 行输出 z 的值。

9.2.3　函数的参数

函数的参数分为形式参数（简称形参）和实际参数（简称实参）两种。形参出现在函数定义中，在整个函数体内都可以使用，离开该函数则不能使用。实参出现在主调函数中，进入被调函数后，实参变量也不能使用。形参和实参的功能是做数据传送。发生函数调用时，主调函数把实参的值传送给被调函数的形参从而实现主调函数向被调函数的数据传送。

【例 9.3】　形参与实参。

```
1   #include<stdio.h>
2   void main()
3   {
4       int n;
5       int sum(int n);
6       printf("input number:\n");
7       scanf("%d",&n);
8       sum(n);
9       printf("in main function n=%d\n",n);
10  }
11  int sum(int n)
12  {
13      int i;
14      for (i=n-1;i>=1;i--)
15      n=n+i;
16      printf("in sum function n=%d\n",n);
17  }
```

程序的运行结果如下：

```
input number:
10 ↙
in sum function n=55
```

```
in main function n=10
```

程序为什么出现这样的结果呢？形参和实参是如何传递的呢？

本程序中定义了一个函数 sum，该函数的功能是求 $\sum_{i=1}^{n} i$ 的值。在主函数中输入 n 值，并作为实参，在第 8 行调用时传送给 sum 函数的形参 n（注意，本例的形参变量和实参变量的标识符都为 n，但这是两个不同的量，各自的作用域不同）。在主函数第 9 行中用 printf 语句输出一次 n 值，这个 n 值是实参 n 的值。在函数 sum 中第 16 行也用 printf 语句输出了一次 n 值，这个 n 值是形参最后取得的 n 值。从运行情况看，输入 n 值为 10，即实参 n 的值为 10，把此值在第 8 行传给函数 sum。第 11 行进入 sum 函数，形参 n 的初值也为 10，在执行函数过程中，形参 n 的值变为 55。返回主函数之后，输出实参 n 的值仍为 10。可见实参的值不随形参的变化而变化。

实际上，函数的形参和实参具有以下特点。

① 实参可以是常量、变量、表达式等。无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值，以便把这些值传送给形参。因此，应预先用赋值、输入等方法使实参获得确定值。

② 形参变量只有在被调用时才分配内存单元，在调用结束时，即刻释放所分配的内存单元。因此，形参只有在函数内部有效。函数调用结束返回主调函数后则不能再使用该形参变量。关于形参的有效范围问题详见 9.6 节。

③ 实参和形参在数量、类型、顺序上应严格一致，否则会发生“类型不匹配”的错误。

④ 函数调用中发生的数据传送是单向的，即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。因此，在函数调用过程中，形参的值发生改变，而实参中的值不会变化。

【例 9.4】　形参只有在函数内部有效。

```
1   #include<stdio.h>
2   int main()
3   {
4       int rows=20;
5       void ClearScreen(int maxScreenRows);
6       ClearScreen(10);
7       ClearScreen(rows);
8       printf("%d\n",maxScreenRows);
9       return 0;
10  }
11  void ClearScreen(int maxScreenRows)
12  {
13      int i;
14      for (i=0;i<maxScreenRows;i++)
15      {
16      printf("\n");
17      }
18  }
```

程序编译时出现如下错误提示：

```
Undefined symbol'maxScreenRows'in function main
```

表明第 8 行中出现的变量 maxScreenRows 没有定义。实际上，函数 ClearScreen 在第 6 行、第 7 行都得到了调用。第一次向 ClearScreen 函数传递了整数 10，函数调用期间 maxScreenRows 是形参，在函数内部 maxScreenRows 的值就是接收来的实参，即整数 10。第 2 次向函数传递了 rows 变量，rows 的值此时为 20，函数内部 maxScreenRows 的值就是整数 20。但是，到了第 8 行，实际上已经在函数 ClearScreen 的外面，此时的形参 maxScreenRows 不再有效，故出现了变量 maxScreenRows 没有定义的提示。

【例 9.5】　用错误的参数调用 ClearScreen 函数。

```
1   #include<stdio.h>
2   int main()
3   {
4       float badValue=20.5;
5       void ClearScreen(int maxScreenRows);
6       ClearScreen('C');
7       ClearScreen(badValue);
8       ClearScreen("If a function expects an int,never give it a string.");
9       ClearScreen(10,20);
10      return 0;
11  }
12  void ClearScreen(int maxScreenRows)
13  {
14      int i;
15      for(i=0;i<maxScreenRows;i++)
16      {
17          printf("\n");
18      }
19  }
```

例 9.5 在以下 4 处展示了对 ClearScreen 函数的调用。

① 第 6 行中，main 函数传递的实参是 1 个字符'C'，尽管与 void ClearScreen(int maxScreenRows)声明的参数类型不匹配，但是由于字符的本质是其 ASCII 码（是整数值），因此可以使用。

② 第 7 行中，对 ClearScreen 函数调用进行了另外的尝试，将浮点数作为其参数使用，一般编译器会给出警告信息，程序还是可以运行的。当然，此时传递的浮点数参数的小数部分会被截断。

③ 第 8 行中，对 ClearScreen 函数调用的尝试，是将字符串作为其参数使用，此时编译器会给出错误报告。

④ 第 9 行中，对 ClearScreen 函数调用的尝试，使用了两个参数，参数数量与定义不符，此时编译器会给出错误报告。

关于形参与实参单向数据传递的例子可参见例 9.3。

9.2.4　函数的返回值

要从函数传出一个数值，必须用到关键字 return，而且数据类型必须和函数的返回类型相匹配。

这里重新查看例 9.3 的函数 max：

```
1   int max(int a,int b)
2   {
3       if(a>b)
4           return a;
5       else
6           return b;
7   }
```

从第 1 行可以看出，函数 max 的返回类型为 int。在函数体的第 4 行和第 6 行函数返回，由于 a、b 也都是 int，所以其返回类型都与函数类型一致。

实际上，所有的函数都有一个返回值。

对于 void 类型的函数，函数不用写 return 语句或者 return 语句后面不带任何表达式。

对于 int 类型的函数，不写 return 语句时，相当于执行了“rerun 0;”语句。

其实，主函数 main 也是有返回类型的。有时将其定义 void main()，表明其返回值为 void。有时将主函数定义为 int main()，或者省略类型直接定义为 main()，都表示函数的返回类型为 int。main 函数的返回值是直接给操作系统的。一般情况下，如果程序正常结束，系统要求数值为 0，如果有异常，则要求一个非 0 值。

9.3　函 数 原 型

除了 main 函数外，每个函数都要有原型。原型声明函数的名称、返回类型和参数列表。原型的作用就是让 C 语言编译器在任何调用函数的时候，对函数的返回类型和参数进行检查。

9.3.1　自定义函数的原型

函数的原型看起来很像函数的第 1 行，但区别是原型是第 1 行用分号结束，而函数还要有函数体。

【例 9.6】　函数原型的使用。

```
1   #include<stdio.h>
2   void ClearScreen(int maxScreenRows);
3   int main()
4   {
5       int rows=20;
6       ClearScreen(rows);
7       return 0;
8   }
```

```
9   void ClearScreen(int maxScreenRows)
10  {
11      int i;
12      for(i=0;i<maxScreenRows;i++)
13      {
14          printf("\n");
15      }
16  }
```

本例中，第 2 行给出了 ClearScreen 函数的原型，而第 9～16 行定义了该函数。注意，第 2 行后面有分号，而第 9 行没有。

实际上，第 2 行还可以直接写成下面的形式：

```
void ClearScreen(int);
```

因为这样，对函数的返回类型和参数类型、个数、顺序等的声明也已经很明确了。

9.3.2　库函数的原型

在编写程序时，大量用到了库函数。实际上，已经在大量使用它们了。例如，一直使用的完成输入和输出的函数 scanf 与 printf，那么库函数的原型在哪里呢？一般来说，库函数的原型在系统提供的头文件中。表 9.1 给出的是常用的几个库函数及它们的原型所在的头文件名称。

表 9.1　常用的几个库函数的原型

函 数 原 型	函 数 功 能	函 数 原 型	函 数 功 能
string.h: 字符串函数		ctype.h: 字符判断函数	
char *strcat(char *, const char *);	连接两个字符串	isdigit(int);	是否为数字字符（0～9）
int strcmp(const char *, const char *);	比较字符串的大小	isalpha(int);	是否为英文字母
char *strcpy(char *, const char *);	字符串复制	isalnum(int);	是否为字母或数字
stdio.h: 输入输出函数		islower(int);	是否为小写字母
int getchar(void);	输入一个字符	isupper(int);	是否为大写字母
char * gets(char *);	输入一个字符串	math.h: 数学函数	
int printf(const char *, ...);	格式化输出	double sin(double);	求正弦
int putchar(int);	输出一个字符	double asin(double);	求反正弦，返回弧度
int scanf(const char *, ...);	格式化输入	double cos(double);	求余弦
int puts(const char *);	输出字符串	double acos(double);	求反余弦
stdlib.h: 杂类函数		double fabs(double);	求绝对值
void *malloc(size_t);	动态分配内存	double exp(double);	求自然指数
int rand(void);	返回一个随机整数	double sqrt(double);	求平方根
void srand(unsigned);	初始化随机数序列	double log(double);	求自然对数

【例 9.7】　库函数原型的使用。

```
1   #include<string.h>
2   #include<stdio.h>
3   int main()
4   {
5       char firstString[80];
6       char anotherString[80];
7       strcpy(firstString,"The first string.");
8       strcpy(anotherString,"The second string.");
9       strcat(firstString,anotherString);
10      printf("In the end the firstString=%s\n",firstString);
11      return 0;
12  }
```

程序运行的结果是：

```
In the end the firstString=The first string.The second string.
```

这里，使用了库函数 strcpy 和 strcat。其中，这两个库函数都是在 string.h 头文件里给出了原型。打开 string.h 头文件，找到这两个函数，可以看到这两个函数的原型是：

```
char*strcpy(char*,constchar*);
char*strcat(char*,constchar*);
```

这里，在第 1 行使用“#include <string.h>”就相当于给出了这两个函数的原型。

实际上，这种思想也可以借鉴来定义我们自己的函数原型。即将定义好的函数的原型放入一个“.h”的头文件中，然后在使用这些函数的模块里使用#include 预编译命令包含这个头文件，进而给出函数的原型。关于这一部分的内容在第 11 章预编译程序部分还有介绍。

9.4　基于函数的结构化设计

结构化程序设计的精髓就是问题的自顶向下逐步求精方法和程序的模块化。

9.4.1　自顶向下逐步求精方法

采用自顶向下的方法进行程序设计，有利于程序获得一个合理的宏观结构。它是依据人们考虑问题的思维方法，即思路，由粗到细、由抽象到具体、逐步细化的过程。程序的编写和测试也采用自顶向下的方法进行。与程序的宏观结构和自顶向下的设计思想一致，程序的编写过程也应遵循由粗到细、由抽象到具体的思想方法和工作过程，这就是逐步求精。这种方法先从模块总的功能出发，逐步细化，直到成为可以在计算机上执行的结构化程序为止。这种方法易于保证和验证程序的正确性。由于求精细化的过程是逐层进行的，每层的细化工作都不会太复杂，只要从问题开始的每一层设计，在检查正确之后再进行下一层的细化，确认每层设计都正确，就保证了整个程序是正确的。由于结构层次清晰，逐层检查验证或证明

程序的正确性也就方便、容易多了。

下面通过一个掷色子游戏程序的设计对这一思想进行说明。

【例 9.8】　掷色子游戏程序。

本游戏程序根据用户掷色子的值，显示相应个数的星号（*）。

设计思路：

① 游戏用一个随机函数模拟用户掷色子的过程。因为色子的点数总共有 1～6 共 6 种情况，因此应将随机函数产生的数据限制在此范围内。设置此掷色子的函数为 randn。

② 色子数为 1～6 的情况可以分别用一个显示星号的函数完成，分别为 showone、showtwo、showthree、showfour、showfive 和 showsix。

③ 为了程序能够多次运行，还应该增加一个控制部分，根据用户的输入确定是否退出程序，设置此函数为 getans。

至于这些函数的具体实现，在设计阶段可以不去关心这些细节。用这种方式编写的函数，称其为黑盒。黑盒表示函数完成任务时我们不知道它是如何进行的。黑盒的方法有助于简化程序的设计过程，这种思维方式在一个项目组里从事软件架构的设计很有意义。

在此基础上，整个程序的框架便很容易生成，见图 9.1。

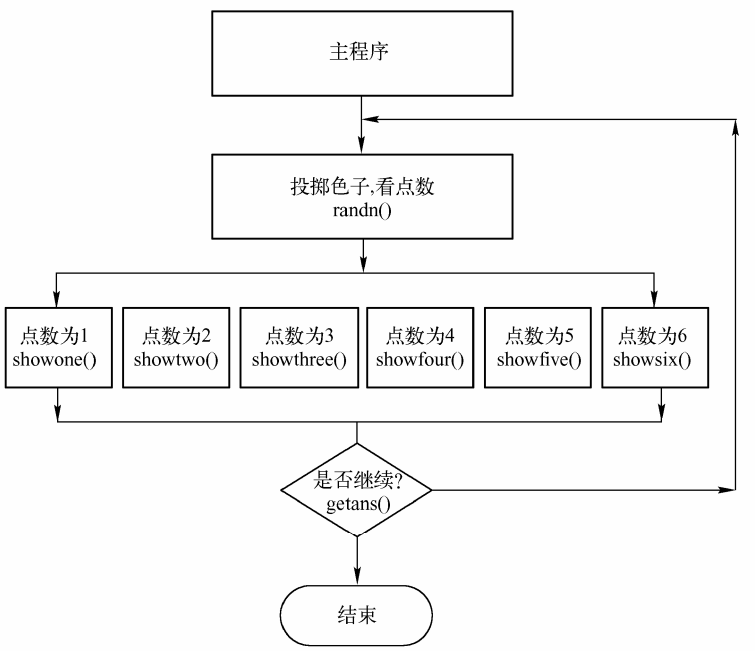


图 9.1　掷色子游戏的框架图

9.4.2　程序模块化

这种方法就是把一个大型程序划分成一些较小的部分，每一个小部分就是一个模块。分解模块应具有简单性、独立性和完整性，这样才能使程序具有较高的可靠性和灵活性，同时便于程序的测试和维护。

划分模块应有如下性质：模块与环境的联系仅限于明确定义了的输入参数和输出参数，其内部结构与调用它的程序无关；一个模块必须通过它的名字来调用；一个模块可以被其他模块调用，被调用模块必须返回到被调用前的模块；一个模块只应具有单一的入口和出口，即模块是封闭的，不对环境产生任何副作用；模块包括的语句不宜过多，功能要单纯，最好一个模块只执行一个功能。

接着看例 9.8 的掷色子游戏程序，程序的功能在前面已经确定了。由于程序功能比较简单，各个小部分都适合于用一个函数来实现，下面直接给出其源程序。

源程序如下：

```
#include<stdio.h>
void showone();
void showtwo();
void showthree();
void showfour();
void showfive();
void showsix();
int randn();
char getans();
void blines();
void main()
{
    int r;
    char ans;
    ans=getans();
    srand(17);
    while(ans=='y')
    {
        r=randn(6);
        blines(2);
        switch(r)
        {
            case 1:
                showone();
                break;
            case 2:
                showtwo();
                break;
            case 3:
                showthree();
                break;
            case 4:
                showfour();
                break;
            case 5:
```

```
                showfive();
                break;
            case 6:
                showsix();
        }
        blines(2);
        ans=getans();
    }
}
int randn(int n)
{
    return rand()%n+1;
}
char getans()
{
    int ans;
    printf("Throw y/n?");
    ans=-1;
    while(ans==-1)
    {
        ans=getche();
    }
    return ans;
}
void blines(int n)
{
    int i;
    for(i=1;i<=n;i++)printf("\n");
}
void showone()
{
    printf("\n*\n");
}
void showtwo()
{
    printf("*\n");
    printf("*\n");
}
void showthree()
{
    printf("*   \n");
    printf(" *  \n");
    printf("  * \n");
}
void showfour()
```

```
81  {
82      printf("**\n\n");
83      printf("**\n");
84  }
85  void showfive()
86  {
87      printf("**\n");
88      printf(" *\n");
89      printf("**\n");
90  }
91  void showsix()
92  {
93      int i;
94      for(i=1;i<=3;i++)
95          printf("**\n");
96  }
```

程序的运行结果如下：

```
Throw y/n?y

 *
 *

Throw y/n?y

 *

Throw y/n?y

 *

Throw y/n?n
```

这里涉及随机数产生的问题。尽管在计算机中并没有一个真正的随机数发生器，但是可以做到使所产生的数字的重复率很低，以至于它们看起来是随机的。实现这一功能的程序叫做伪随机数发生器。

这里要说的是，不管用什么办法实现随机数发生器，都必须给它提供一个被称为“种子（seed）”的初始值，而且这个值最好是随机的，或者至少是伪随机的。

现在的 C 语言编译程序都提供了一个基于 ANSI 标准的伪随机数发生器函数，用来生成随机数。Visual C++ 6.0 是通过 rand 函数和 srand 函数来支持这种标准的，它们的工作过程如下：

① 给 srand 函数提供一个“种子”，它是 unsigned int 类型；

② 调用 rand 函数，它会根据提供给 srand 函数的“种子”值返回一个随机数；

③ 根据需要多次调用 rand 函数，从而不断地得到新的随机数；

④ 无论什么时候，都可以给 srand 函数提供一个新的“种子”，从而进一步“随机化” rand 函数的输出结果。

这个过程看起来很简单，问题是如果每次调用 srand 函数时都提供相同的“种子”值，那么将会得到相同的“随机”数序列。例如，在以 17 为“种子”值调用 srand 函数之后，在首次调用 rand 函数时，将得到随机数 94；在第二次和第三次调用 rand 函数时，将分别得到 26602 和 30017。这些数看上去是相当随机的（尽管这只是一个很小的数据点集合），但是，再次以 17 为“种子”值调用 srand 函数之后，在对 rand 函数的前三次调用中，所得到的返回值仍然是 94、26602 和 30017。因此，只有再次给 srand 函数提供一个随机的“种子”值，才能再次得到一个随机数。

9.5　函数的递归调用

一个函数在它的函数体内调用它自身称为递归调用。这种函数称为递归函数。C 语言允许函数的递归调用。例如，有函数 f 如下：

```
int f(int x)
{
    int y;
    z=f(y);
    return z;
}
```

这个函数是一个递归函数。 但是运行该函数将无休止地调用其自身，这当然是不正确的。为了防止递归调用无休止地进行，必须在函数内有终止递归调用的手段。常用的办法是加条件判断，满足某种条件后就不再进行递归调用，然后逐层返回。

递归是设计和描述算法的一种有力的工具，由于它在复杂算法的描述中被经常采用。能采用递归描述的算法通常有这样的特征：为求解规模为 N 的问题，设法将它分解成规模较小的问题，然后从这些小问题的解方便地构造出大问题的解，并且这些规模较小的问题也能采用同样的分解和综合方法，分解成规模更小的问题，并从这些更小问题的解构造出规模较大问题的解。特别地，当规模 N=1 时，能直接得解。

下面举例说明递归调用的执行过程。

【例 9.9】　用递归法计算 n!。

用递归法计算 n!可用以下公式表示：

$$n!=\begin{cases}1, & (n=0,1)\\ n\times(n-1)!, & (n>1)\end{cases}$$

按公式可编程如下：

```
1   #include<stdio.h>
2   long factorial(int n)
3   {
4       long f;
5       if(n<=1)f=1;
```

```
6       else f=n*factorial(n-1);
7       return(f);
8   }
9   void main()
10  {
11      int n;
12      long y;
13      printf("\ninput a integer number:\n");
14      scanf("%d",&n);
15      y= factorial(n);
16      printf("%d!=%ld",n,y);
17  }
```

运行结果如下：

```
input a integer number:
5 ↙
5!=120
```

程序中给出的函数 factorial 是一个递归函数。主函数调用 factorial 后即进入函数 factorial 执行，如果 n<0、n==0 或 n=1 时都将结束函数的执行，否则就递归调用 factorial 函数自身。由于每次递归调用的实参为 n–1，即把 n–1 的值赋予形参 n，最后当 n–1 的值为 1 时再进行递归调用，形参 n 的值也为 1，将使递归终止，然后可逐层退回。

下面再举例说明该过程。设执行本程序时输入为 5，即求 5!。在主函数中的调用语句即为 y=factorial(5)，进入 factorial 函数后，由于 n=5，不等于 0 或 1，故应执行 f=factorial(n–1)*n，即 f=5*factorial(5–1)。该语句对 factorial 做递归调用即 factorial(4)。逐次递归展开如图 9.2 所示。

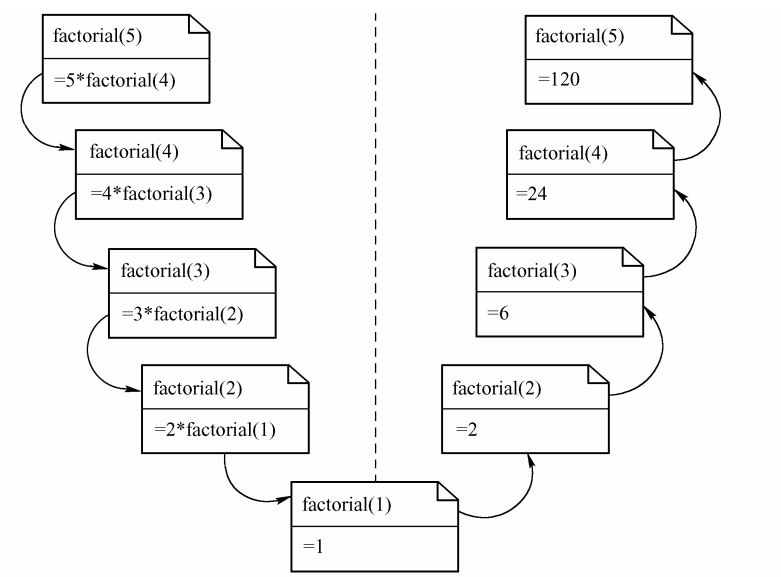


图 9.2　求 *n*!递归过程

进行 4 次递归调用后，factorial 函数形参取得的值变为 1，故不再继续递归调用而开始逐层返回主调函数。factorial(1)的函数返回值为 1，factorial(2)的返回值为 2*1=2，factorial(3)的返回值为 3*2=6，factorial(4) 的返回值为 4*6=24，最后返回值 factorial (5)为 5*24=120。

【例 9.10】　快速排序问题。

这是 C.A.R. Hoare 在 1962 年提出的一种排序算法。其基本原理是：给定一个数组，选定其中一个元素作为基准元素，其余的元素分成两个子集——其中一个子集存放小于选定元素的元素，另一个子集放的是大于或等于选定元素的元素。然后，这一过程递归应用到两个子集中。当子集小于两个元素时，停止递归过程。

下面的程序是采用数组的中间元素作为选定元素，然后以此为基础，将小于此元素者归为左子集，大于等于者为右子集，排序采用升序排列。

```
#include<stdio.h>

/*qsort: sort v[left]...v[right]into increasing order*/
void qsort(int v[],int left,int right)
{
    int i,last;
    void swap(int v[],int i,int j);

    if(left>=right)             /*do nothing if array contains*/
        return;                 /*fewer than two elements*/
    swap(v,left,(left+right)/2);    /*move partition elem*/
    last=left;                      /*to v[0]*/
    for (i=left+1;i<=right;i++)     /*partition*/
        if(v[i]<v[left])
            swap(v,++last,i);
    swap(v,left,last);              /*restore partition  elem*/
    qsort(v,left,last-1);
    qsort(v,last+1,right);
}

/*swap: interchange v[i]and v[j]*/
void swap(int v[],int i,int j)
{
    int temp;
    temp=v[i];
    v[i]=v[j];
    v[j]=temp;
}

void main()
{
    int a[10]={1,0,4,8,12,65,-76,100,-45,123};
    int i;
```

```
34      for(i=0;i<=9;i++)
35          printf("%d",a[i]);
36      qsort(a,0,9);
37      printf("\nThe sorted numbers:\n");
38      for(i=0;i<=9;i++)
39          printf("%d",a[i]);
40      printf("\n");
41  }
```

运行结果如下：

```
1 0 4 8 12 65 -76 100 -45 123
The sorted numbers:
 -76 -45 0 1 4 8 12 65 100 123
```

程序第 4 行定义 qsort 函数：“void qsort(int v[], int left,int right)”，有 3 个参数，v[]表示要排序的数组，left 表示排序数组的起始元素下标，right 表示排序的终止元素下标。其次，因为函数中多次用到数组元素的交换，故将交换定义为专门的函数 swap。第 22 行定义的 swap 函数 void swap(int v[], int i,int j)也有 3 个参数，v[]表示要交换元素的数组，i、j 表示数组中互换元素的下标。这里 qsort 函数和 swap 函数都涉及数组名字作为参数的问题。

前面曾介绍过，数组名就是数组的首地址。需要强调一点：用数组名字作为实参时，不是把数组元素的值传递给形参，而是把实参数组的首元素的地址传递给形参数组，这样两个数组就共占同一段内存单元，如图 9.3 所示。因此，任何在函数体内对形参数组的改变就会直接反映在实参数组上。

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
起始地址	1	0	4	8	12	65	−76	100	−45	123
	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]

图 9.3　数组名作为实参时内存分配情况

为了更好地理解排序的过程，下面执行一次 qsort，看是如何将数组分成左右两个子集的。

起始时，left=0，(left+right)/2=4，因而执行第 11 行后，a[0]与 a[4]元素互换；执行第 12 行，last=0，实际上是将基准元素放入 a[0]；第 13～15 行执行的是一个循环过程，将数组的 a[1] ～a[9]与基准元素比较，小于基准元素者放入数组的前端，放入的位置由 last 变量记录，每来一个小于基准元素，先将 last 加 1，然后将新来元素与 last 位置处元素互换。实际上，此时 last 记录了数组中小于基准元素的元素数，同时这些元素也被放入了数组的 1～last 的下标位置。

当 i=1、2、3、4 时由于 v[1]=0、v[2]=4、[3]=8、v[4]=1 都满足 v[i] < v[left]，因此都将执行第 15 行程序，被依次与 v[last+1]交换，实际上这时由于 last 与 i 相等，并没有真正交换，只是将 last 不断加 1。i=4 循环完成后，last=4。

当 i=5 时，由于 v[5]=65，不满足 v[i] < v[left]，不做任何处理。

当 i=6 时，v[6]=−76，满足 v[i] < v[left]，此时将执行第 15 行程序，将 v[6]与 v[5]交换，

last=5。

当 i=7 时，由于 v[7]=100，不满足 v[i] < v[left]，不做任何处理。

当 i=8 当，v[8]=–45，满足 v[i] < v[left]，此时将执行第 15 行程序，将 v[8]与 v[6]交换。

当 i=9 时，由于 v[9]=123，不满足 v[i] < v[left]，不做任何处理。

第 16 行执行将基准元素放入合适的位置的操作，此时 last=6，最后得出的小于基准元素的元素将与基准元素交换位置，放入第 0 个元素处。这样，0～last–1 下标的元素就成了小于基准元素的子集，last 位置放入的是基准元素，而 last+1 至最后的元素都归为大于等于基准元素的子集。详细的交换过程见表 9.2。

表 9.2　快速排序执行过程数组变化情况

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
起始	1	0	4	8	12	65	–76	100	–45	123
第 9 行后	12	0	4	8	1	65	–76	100	–45	123
i=1 后	12	0	4	8	1	65	–76	100	–45	123
i=2 后	12	0	4	8	1	65	–76	100	–45	123
i=3 后	12	0	4	8	1	65	–76	100	–45	123
i=4 后	12	0	4	8	1	65	–76	100	–45	123
i=5 后	12	0	4	8	1	65	–76	100	–45	123
i=6 后	12	0	4	8	1	–76	65	100	–45	123
i=7 后	12	0	4	8	1	–76	65	100	–45	123
i=8 后	12	0	4	8	1	–76	–45	100	65	123
i=9 后	12	0	4	8	1	–76	–45	100	65	123
第 14 行后	–45	0	4	8	1	–76	12	100	65	123

第 17、18 行再分别完成对左右两个子集的递归排序，都直到满足第 9 行的条件为止，从第 10 行返回，从而完成整个排序。后面的数组变化表 9.2 没有给出。

实际上，在 C 语言标准库里也有一个 qsort 函数，它可以对于任意类型的数据进行排序。关于这个标准函数的使用这里不再举例。

9.6　变量的作用域

在讨论函数的形参变量时曾经提到，形参变量只在被调用期间才分配内存单元，调用结束立即释放。这一点表明，形参变量只有在函数内才是有效的，离开该函数就不能再使用了。这种变量有效性的范围称为变量的作用域。

不仅形参变量，C 语言中所有的变量都有自己的作用域。变量说明的方式不同，其作用域也不同。C 语言中的变量，按作用域范围可分为两种，即局部变量和全局变量。

【例 9.11】　变量的作用域。

以下程序内有多种类型的变量定义形式，下面将结合程序，对这些变量定义一一说明。

```
1   #include<stdio.h>
2   int count;              /*This is a global variable*/
3   void main( )
```

```
4   {
5       int index;      /*This variable is available only in main*/
6       head1();
7       head2();
8       head3();
9       /*main"for"loop of this program*/
10      for(index=8;index>0;index--)
11      {
12          int stuff;   /*This variable is only available in these braces*/
13          for(stuff=0;stuff<=6;stuff++)
14              printf("%d",stuff);
15          printf("index is now %d\n",index);
16      }
17  }
18  int counter;        /*This is available from this point on*/
19  void head1()
20  {
21      int index;      /*This variable is available only in head1*/
22      index=23;
23      printf("The header1 value is %d\n",index);
24  }
25  void head2()
26  {
27      int count;      /*This variable is available only in head2*/
28      /*and it displaces the global of the same name*/
29      count=53;
30      printf("The header2 value is %d\n",count);
31      counter=77;
32  }
33  void head3()
34  {
35      printf("The header3 value is %d\n",counter);
36  }
```

第 1 个变量 count（第 2 行）是一个全局变量，它在程序的任意函数内都是有效的。因为它定义在所有函数的前面。变量 counter（第 18 行）也是一个全局变量，但是它在 main 函数中是无效的，因为它在 main 函数后才定义。

全局变量可以是定义在任何函数之外的任意类型变量。这些变量有时也被称为外部（extern）变量，原因就是它们在任何函数体之外。

全局变量不属于哪一个函数，它属于一个源程序文件，其作用域是整个源程序。在函数中使用全局变量，一般应做全局变量说明，只有在函数内经过说明的全局变量才能使用。全局变量的说明符为 extern。在一个函数之前定义的全局变量，在该函数内使用可不再加以说明。

在 main 函数中，第 5 行定义了整型变量 index，因为它是在 main 函数内定义的，此变量只在 main 函数内有效。另外，它是一个自动型（automatic）变量。自动型变量意味着它

只在定义它的函数体内出现时被激活，而函数结束时将消失。当然，在 main 函数里，这并没有什么意义，因为 main 函数始终处于运行状态，即使程序流程转到其他函数时都是如此。

另一个定义的整型变量 stuff（第 12 行）是在 for 括号内定义的。任何一对大括号内定义的变量只在本括号内的执行语句中有效。此变量也是自动型变量，当程序执行完离开大括号时自动失效。定义此类变量，在像 for 这种类型的循环中作为计数变量或在其他非常局部化的变量定义中非常方便。

其他函数体定义的局部变量在 9.2 节已有说明，在此不再赘述。

整个程序的运行结果如下：

```
The header1 value is 23
The header2 value is 53
The header3 value is 77
0123456 index is now 8
0123456 index is now 7
0123456 index is now 6
0123456 index is now 5
0123456 index is now 4
0123456 index is now 3
0123456 index is now 2
0123456 index is now 1
```

关于局部变量的作用域还要说明如下。

① 主函数中定义的变量只能在主函数中使用，不能在其他函数中使用。同时，主函数中也不能使用其他函数中定义的变量。因为主函数也是一个函数，它与其他函数是平行关系。这一点是与其他语言不同的，应予以注意。

② 形参变量是属于被调函数的局部变量，实参变量是属于主调函数的局部变量。

③ 允许在不同的函数中使用相同的变量名，它们代表不同的对象，分配不同的单元，互不干扰，也不会发生混淆。

④ 在同一源文件中，允许全局变量和局部变量同名。在局部变量的作用域内，全局变量不起作用。

⑤ 在复合语句中也可定义变量，其作用域只在复合语句范围内，如图 9.4 所示。

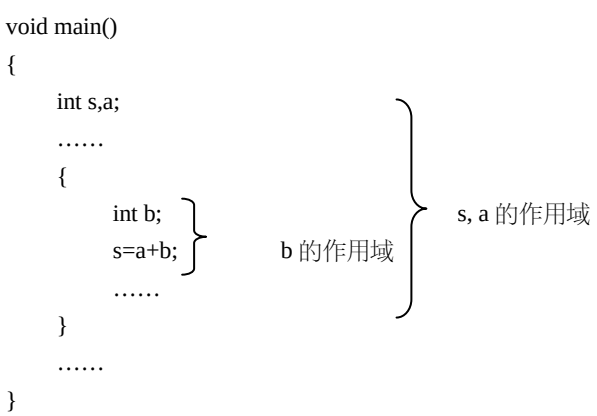


图 9. 4　在复合语句中定义变量

9.7　变量的存储类型

所谓存储类型是指变量占用内存空间的方式，也称为存储方式。变量的存储方式可分为“静态存储”和“动态存储”两种。

静态存储变量通常是在变量定义时就分定存储单元并一直保持不变，直至整个程序结束。9.6 节中介绍的全局变量即属于此类存储方式。

动态存储变量是在程序执行过程中，使用它时才分配存储单元，使用完毕立即释放。典型的例子是函数的形式参数，在函数定义时并不给形参分配存储单元，只是在函数被调用时，才予以分配，调用函数完毕立即释放。如果一个函数被多次调用，则反复地分配、释放形参变量的存储单元。

从以上分析可知，静态存储变量是一直存在的，而动态存储变量则时而存在、时而消失。把这种由于变量存储方式不同而产生的特性称为变量的生存期。生存期表示了变量存在的时间。生存期和作用域从时间和空间这两个不同的角度来描述变量的特性，这两者既有联系，又有区别。一个变量究竟属于哪一种存储方式，并不能仅从其作用域来判断，还应有明确的存储类型说明。

在 C 语言中，对变量的存储类型说明有以下 4 种：

- auto：自动变量。
- register：寄存器变量。
- extern：外部变量。
- static：静态变量。

自动变量和寄存器变量属于动态存储方式，外部变量和静态变量属于静态存储方式。在介绍了变量的存储类型之后，可以知道对一个变量的说明不仅应说明其数据类型，还应说明其存储类型。

因此，变量说明的完整形式应为：

存储类型说明符 数据类型说明符 变量名,变量名……;

例如：

① static int a,b;　　说明 a, b 为静态类型变量。
② auto char c1,c2;　　说明 c1, c2 为自动字符变量。
③ static int a[5]={1,2,3,4,5};　　说明 a 为静态整型数组。
④ extern int x,y;　　说明 x, y 为外部整型变量。

9.7.1　auto变量

这种存储类型是 C 语言程序中使用最广泛的一种类型。C 语言规定，函数内凡未加存储类型说明的变量均视为自动变量，也就是说自动变量可以省去说明符 auto。在前面各章的程序中所定义的变量凡未加存储类型说明符的都是自动变量。

自动变量具有以下特点。

① 自动变量的作用域仅限于定义该变量的个体内。在函数中定义的自动变量，只在该

函数内有效。在复合语句中定义的自动变量只在该复合语句中有效。

② 自动变量属于动态存储方式，只有在使用它，即定义该变量的函数被调用时，才给它分配存储单元，开始它的生存期。函数调用结束时，释放存储单元，结束生存期。因此，函数调用结束之后，自动变量的值不能保留。在复合语句中定义的自动变量，在退出复合语句后也不能再使用，否则将引起错误。

③ 由于自动变量的作用域和生存期都局限于定义它的个体内（函数或复合语句内），因此不同的个体中允许使用同名的变量而不会混淆。即使在函数内定义的自动变量也可以与该函数内部的复合语句中定义的自动变量同名。

例 9.11 的 head1 函数中的 index 和 main 函数中 index，都是自动变量，但是它们本身是两个不同的实体。

9.7.2　extern变量

在前面介绍全局变量时已介绍过外部变量。下面再补充说明外部变量的几个特点。

① 外部变量和全局变量是对同一类变量的两种不同角度的提法。全局变量是从它的作用域提出的，外部变量是从它的存储方式提出的，表示了它的生存期。

② 当一个源程序由若干个源文件组成时，在一个源文件中定义的外部变量在其他的源文件中也有效。例如，有一个源程序由源文件 F1.c 和 F2.c 组成。

F1.c 文件如下：

```
int a,b;                /*外部变量定义*/
char c;                 /*外部变量定义*/
void main()
{
……
}
```

F2.c 文件如下：

```
extern int a, b;        /*外部变量说明*/
extern char c;          /*外部变量说明*/
void func (int x, y)
{
……
}
```

在 F1.c 和 F2.c 两个文件中都要使用 a、b、c 三个变量。在 F1.c 文件中把 a、b、c 都定义为外部变量。在 F2.c 文件中用 extern 把三个变量说明为外部变量，表示这些变量已在其他文件中定义，编译系统不再为它们分配内存空间。对构造类型的外部变量，如数组等，可以在说明时初始化赋值，若不赋初值，则系统自动定义它们的初值为 0。

9.7.3　static变量

将保留字 static 加在一个函数体定义的变量前，那么这个变量就成为静态局部变量，它在这个函数的每次调用时都占用同一块内存区域。因此，当多次调用一个函数且要求在调用

之间保留某些变量的值时，可以考虑采用静态局部变量，有时这种定义方式很有用。

将保留字 static 加在一个外部变量前，则将会使这个外部变量“私有化”，就是说将会使这个外部变量只能在本文件之内使用，在其他文件内即使使用 extern 说明了这个变量，它仍然不可以被访问。

静态局部变量属于静态存储方式，它具有以下特点。

① 静态局部变量在函数内定义，但不像自动变量那样，当调用时就存在，退出函数时就消失。静态局部变量始终存在着，也就是说它的生存期为整个源程序。

② 静态局部变量的生存期虽然为整个源程序，但是其作用域仍与自动变量相同，即只能在定义该变量的函数内使用该变量。退出该函数后，尽管该变量还继续存在，但不能使用它。

③ 允许对构造类静态局部变量赋初值。在第 6 章中介绍数组初始化时已做过说明。若未赋初值，则由系统自动赋 0 值。

④ 对基本类型的静态局部变量若在说明时未赋初值，则系统自动赋 0 值。而对自动变量不赋初值，则其值是不定的。根据静态局部变量的特点，可以看出它是一种生存期为整个源程序的量。虽然离开定义它的函数后不能使用，但再次调用定义它的函数时，它又可继续使用，而且保存了前次被调用后留下的值。虽然用全局变量也可以达到上述目的，但全局变量有时会造成意外的副作用，因此仍以采用局部静态变量为宜。

【例 9.12】　静态变量的使用。

在许多计算机构成的数据采集控制系统中，要求每隔一定时间完成一些数据采集或控制之类的任务。那么该如何实现呢？下面是一个模拟数据采集的小程序。函数 f 是数据采集程序。正常情况下，它的触发是由计算机硬件每隔一定时间自动完成的，然后每触发 3 次完成一次真正的数据采集。

```
1   #include <stdio.h>
2   void main()
3   {
4       int i;
5       void f();
6       for (i=1;i<=10;i++)
7           f();
8   }
9   void f()
10  {
11      static int j=0, i=1;
12      ++j;
13      printf("%d\t", j);
14      if ( j%3==0 )
15      {
16          printf("the %d times data gathering!\n", i);
17          i++;
18      }
19  }
```

运行结果如下：

```
1      2      3      the 1 times data gathering!
4      5      6      the 2 times data gathering!
7      8      9      the 3 times data gathering!
10
```

由于 f 函数的 j、i 为静态变量，能在每次调用后保留其值并在下一次调用时继续使用，所以输出值成为累加的结果。读者可自行分析其执行过程。

9.7.4　register变量

通常，变量都存放在存储器内，因此当对一个变量频繁读写时，必须要反复访问内存储器，从而花费大量的存取时间。为此，C 语言提供了另一种变量，即寄存器变量。这种变量存放在 CPU 的寄存器中，使用时，不需要访问内存，而直接从寄存器中读写，这样变量的存取效率可大大提高。寄存器变量的说明符是 register。由于寄存器的特点，只有整型或者字符型变量可以声明为寄存器变量。其他类似的类型如 unsigned、long 或 short 也可以声明为寄存器变量。

实际上，任何编译器都只允许使用有限个寄存器变量，而且一旦没有寄存器可以使用，编译器将忽略寄存器变量，而只把它当成普通的变量。因此，通过使用寄存器变量获得的速度提高是有限的。

习　题　9

9.1　编写一个程序，使用 Square 函数来计算 2、10、623 的平方和。Square 函数可以返回数的平方，数的平方是这个数与自身相乘。

9.2　编写一个程序，使用函数调用 Power。Power 函数应符合以下原型：

```
int Power(int aNumber, int thePower);
```

Power 函数应能对参数 aNumber 求幂，幂指数为参数 thePower。

9.3　编写函数 replace，使用字符串指针作为参数。函数的功能是用“-”替换字符串中所有空格，并返回替换的空格个数。例如：

```
char *cat = "The cat sat";
n = replace(cat);
```

那么，cat 将为"The-cat-sat"，　n 为 2。

9.4　直线的表达式为：

$$y=mx+b$$

式中，*m* 是直线的斜率，*b* 是截距（即与 *y* 轴的交点）。编写函数能计算并返回直线上任意给定 *x* 值对应的 *y* 坐标。函数符合以下原型：

```
double FindYValue(double slope,double xValue,double yIntercept);
```

编写一个程序，调用该函数输出直线上的若干个点对应的（x，y）值。

9.5　希腊数学家 Euclid 研究了求两个整数的最大公约数的算法。对于两个整数 integer1 和 integer2，算法如下：

① 如果 integer1/integer2 的余数为 0，那么 integer2 就是最大公约数；

② 如果余数不为 0，那么将 integer2 赋值给 integer1，余数赋值给 integer2；

③ 从步骤①重复执行。

编写一个程序来实现这个算法。它使用两个整型参数，并返回最大公约数。

9.6　编写计算 Fibonacci 数列的第 n 项函数 fib(n)。

Fibonacci 数列为：0，1，1，2，3，…，即

fib(0)=0

fib(1)=1

fib(n)=fib(n–1)+fib(n–2)，n>1

9.7　用递归方法求 n 阶勒让德多项式的值，递归公式为：

$$p_n(x)=\begin{cases}1, & n=0\\ x, & n=1\\ ((2n-1)*x-p_{n-1}(x)-(n-1)*p_{n-2}(x))/2, & n>1\end{cases}$$

9.8　三维计算机图形程序通常用矩阵乘法进行移动、旋转和缩放绘制 3D 图形。具体来说，采用 4×4 的变换矩阵乘以图形的每个顶点。这个乘法的形式为：

$$\begin{bmatrix}X_P & Y_P & Z_P & 1\end{bmatrix}\cdot\begin{bmatrix}M_{00} & M_{01} & M_{02} & M_{03}\\ M_{10} & M_{11} & M_{12} & M_{13}\\ M_{20} & M_{21} & M_{22} & M_{23}\\ M_{30} & M_{31} & M_{32} & M_{33}\end{bmatrix}=\begin{bmatrix}X_R & Y_R & Z_R & 1\end{bmatrix}$$

在公式中 X_P、Y_P、Z_P 代表被变换点的 x、y、z 坐标，X_R、Y_R、Z_R 代表被变换后的点的 x、y、z 坐标。编写一个函数，它能用以上的公式变换一个点。函数应符合以下的原型：

```
void TransformPoint(double thePoint[3];
double tranMatrix[4][4];
double the resPoint[3]);
```

编写程序对点(10,10,10)、(40,30,20)、(30,30,40)各使用以下 3 个变换矩阵变换一次，看其新的坐标位置。

$$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\quad\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 5 & 9 & 3 & 1\end{bmatrix}\quad\begin{bmatrix}2 & 0 & 0 & 0\\ 0 & 3 & 0 & 0\\ 0 & 0 & 4 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

第 10 章　自定义数据类型

学习目标

通过本章学习，应该掌握：

- 结构体、共用体的有关概念和定义方法
- 结构体数组与结构体指针
- 链表的有关概念
- 自定义数据类型的方法

10.1　概　　述

到目前为止，程序用到的数据类型都是基本数据类型（如整型、字符型、浮点型等）或者由基本类型数据构成的数组。但是，有的时候需要将多种类型的数据联系在一起，构成更加复杂的数据类型。

例如，要设计一个学生成绩管理系统，需要管理每个同学的成绩，见表 10.1。尽管学生姓名可以用字符数组表示，课程成绩也可以用浮点型数据表示，然而多门课程的成绩与学生的姓名之间是有对应关系的，因此为了保持多个数据之间的关系，编程者必须考虑相应的策略，否则数据之间关系容易出现混乱。

表 10.1　学生成绩登记

学　号	姓　名	数　学	英　语	物　理	体　育	C 语 言
706001	张大伟	92	87	93	80	95
706002	李丽	86	92	85	82	91
……	……	……	……	……	……	……

实际上，C 语言提供了自己定义数据类型的机制，在这种机制下，数据管理将会非常方便。另外，这些自定义的数据类型还便于数据的封装与隐藏，这些都为我们写出更好的软件奠定了基础。

本章将主要介绍结构体、共用体、枚举等自定义数据类型。

10.2　结　构　体

结构体是 C 语言提供的创建自己的数据类型的最有力的手段。C 语言结构体可以包含若干数据项，每个数据项被称为“成员”。所有这些数据项在一起成为“成员数据”。

10.2.1　结构体的定义与应用

结构体定义的一般形式如下：

```
struct  结构体类型名
{
    数据类型  成员 1;
    数据类型  成员 2;
    ……
    数据类型  成员 n;
};
```

struct 关键字后面是结构体类型名，接着是一对大括号“{}”，它们表示结构体类型定义的开始和结束。括号后面要有分号“;”，表示整个定义结束。在括号中，与声明程序变量一样声明结构体的数据成员。在结构体中的成员还可以使用用户自定义的类型，包括结构体类型。

使用 struct 关键字来定义结构体类型时不分配任何存储区。只是在声明结构体变量时，才分配存储空间，实际上这与其他简单类型变量是一样的。相比之下，结构体变量的使用比其他简单类型的变量更复杂一些，因为它不仅本身可以作为一个变量被访问，同时其数据成员也要被访问。

例 10.1 给出了结构体的定义与使用，通过例 10.1 可以加深对结构体的理解。

【例 10.1】　结构体变量的定义与使用。

程序如下：

```
1   #include <stdio.h>
2   struct child
3   {
4       char initial;   /*last name initial*/
5       int age;        /*childs age*/
6       int grade;      /*childs grade in school*/
7   };
8   void main()
9   {
10      struct child boy,girl,copyboy;
11      boy.initial='R';
12      boy.age=15;
13      boy.grade=75;
14      girl.age=boy.age-1;     /*she is one year younger*/
15      girl.grade=82;
16      girl.initial='H';
17      copyboy=boy;            /*copy boy's data to copyboy*/
18      printf("girl:%c is%d years old and got a grade of%d\n",
19              girl.initial,girl.age,girl.grade);
20      printf("boy:%c is%d years old and got a grade of%d\n",
21              boy.initial,boy.age,boy.grade);
22      printf("copyboy:%c is%d years old and got a grade of%d\n",
```

```
23              copyboy.initial,copyboy.age,copyboy.grade);
24  }
```

程序运行结果如下：

```
girl:H is 14 years old and got a grade of 82
boy:R is 15 years old and got a grade of 75
copyboy:R is 15 years old and got a grade of 75
```

第 2～7 行定义了结构体 child，第 10 行定义了 3 个结构体类型的变量。注意，结构体类型的变量定义时前面还要加关键字 struct。

第 11～13 行给出的是变量 boy 的赋值过程，从中可以看出结构体成员变量的访问只需在变量后面加一个“.”即可，结构体成员变量的赋值与一般简单变量一样。第 14～16 行给出的是 girl 的赋值，第 14 行更是直接将 boy.age 的值直接拿来使用。实际上，对于 boy 和 girl 的 3 个成员变量来说，它们就是普通的简单变量，可以在程序的任何地方使用，其赋值也可以在任何地方以任何顺序完成。但脱离了结构体变量的成员变量是不能使用的，例如，如果本例中直接使用“age”就是非法的。

第 17 行完成的是 copyboy 的赋值，直接将 boy 赋值给了 copyboy。需要说明的是，尽管在这里这种赋值是支持的，但当结构体内有指针或数组成员时，这种赋值会带来一些问题。具体可参见例 10.2。

第 18～23 行完成了 3 个变量内成员变量的输出。

除了例 10.1 给出的先定义结构体类型再定义变量名的定义方法外，还有其他两种方法。

（1）在声明类型的同时定义变量。

例如：

```
struct child
{
    char initial;   /*last name initial*/
    int age;        /*childs age*/
    int grade;      /*childs grade in school*/
}boy,girl;
```

（2）直接定义结构体变量，例如：

```
struct
{
    char initial;   /*last name initial*/
    int age;        /*childs age*/
    int grade;      /*childs grade in school*/
}boy,girl;
```

【例 10.2】　结构体变量的初始化。

与其他类型变量一样，结构体变量也可以在定义时指定初值。

程序如下：

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

```
2   void main()
3   {
4       struct
5       {
6           char initial;
7           int age;
8           int grade;
9       } boy={'A',15,90};
10      printf("%c is%d years old and got a grade of%d\n",
11              boy.initial,boy.age,boy.grade);
12  }
```

程序运行结果是：

```
A is 15 years old and got a grade of 90
```

本例中，第 4～9 行采用直接定义结构体变量的方法，没有定义结构体的类型。同时在第 9 行直接为结构体变量 boy 赋了初值。赋初值的方法从运行结果来看一目了然。

10.2.2　结构体数组与指针

结构体类型定义后，就可以像基本数据类型那样使用它了，因此结构体数组和指针的定义就没有特别之处。只是在使用的时候，由于要用到结构体的数据成员，其使用方法比一般的基本类型的数组与指针略为复杂。下面通过一个例子介绍结构体数组和指针的使用。

【例 10.3】　结构体数组和指针的使用。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {
4       struct
5       {
6           char initial;
7           int age;
8           int grade;
9       }kids[8],*point;
10      int index;
11      for(index=0;index<8;index++)
12      {
13          point=kids+index;
14          point->initial='A'+index;
15          point->age=16;
16          point->grade=87;
17      }
18      kids[3].age=kids[5].age=17;
```

```
19      kids[2].grade=kids[6].grade=92;
20      kids[4].grade=57;
21      for(index=0;index<8;index++)
22      {
23          point=kids+index;
24          printf("%c is%d years old and got a grade of%d\n",
25              (*point).initial,kids[index].age,
26              point->grade);
27      }
28  }
```

运行结果如下：

```
A is 16 years old and got a grade of 87
B is 16 years old and got a grade of 87
C is 16 years old and got a grade of 92
D is 17 years old and got a grade of 87
E is 16 years old and got a grade of 57
F is 17 years old and got a grade of 87
G is 16 years old and got a grade of 92
H is 16 years old and got a grade of 87
```

程序第 9 行定义了结构体数组 kids[]和指针 point。第 11～17 行完成结构体数组元素的赋值。赋值时采用指针指向结构体数组变量。它们之间的对应关系可以用图 10.1 表示。图中对 kids[1]的各个成员在右边进行了展示，其实其他结构体数组的元素都是一样的。

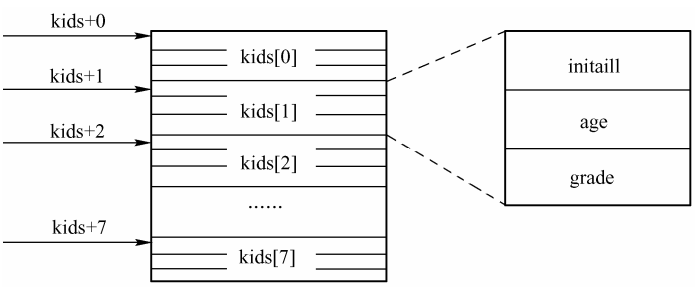


图 10.1　结构体数组与指针

第 13 行表示 point 指向数组的第 index 个元素，14～16 行表示的是一种通过指针访问结构体变量的数据成员的方法。

程序第 18～20 行对结构体数组 kids[]的某些元素的数据成员进行了修改。第 21～27 行对整个数组的元素进行了显示。第 25～26 行展示了指针的两种用法，这时指针指向数组的第 index 个元素。“(*point).initial”与“point->initial”的意思是一致的。这两种指针的用法和第 7 章中所讲的指针用法是一致的，唯一的不同之处在于这里访问的是结构体数组的成员变量。

通过结构体指针变量访问结构体成员的两种方法如下。

（1）使用点操作符：

(*结构体指针变量名).成员名

（2）使用箭头操作符：

结构体指针变量名->成员名

10.2.3　结构体的嵌套与指针成员

所谓结构体嵌套是指结构体的数据成员仍然是结构体类型。实际上，定义过的结构体类型本身也是一种数据类型，因此，这种嵌套定义的结构体类型也就不足为怪了。只是由于成员本身是结构体类型，所以在访问成员时，需要按照结构体成员的访问方法。

【例 10.4】　嵌套结构体的定义与访问。

程序如下：

```
#include<stdio.h>
void main()
{
    struct person{
        char name[25];
        int age;
        char status;        /*M=married,S=single*/
    };
    struct alldat{
        int grade;
        struct person descrip;
        char lunch[25];
    } student[3];
    struct alldat teacher;
    int i;
    teacher.grade=94;
    teacher.descrip.age=34;
    teacher.descrip.status='M';
    strcpy(teacher.descrip.name,"Mary Smith");
    strcpy(teacher.lunch,"Baloney Sandwich");
    for(i=0;i<=2;i++)
    {
        student[i].descrip.age=15;
        student[i].descrip.status='S';
        strcpy(student[i].lunch,"Peanut Butter");
        student[i].grade=77;
    }
    student[1].grade=87;
    student[1].descrip.age=14;
    strcpy(student[0].descrip.name,"Todd White");
    strcpy(student[1].descrip.name,"Peter Bush");
```

```
32      strcpy(student[2].descrip.name,"Tiger Black");
33      printf("the status of the teacher and students:\n");
34      printf("name    \tage\tstatus\tgrade\t  lunch\n");
35
        printf("================================================\n");
36      printf("%10s\t%d\t%c\t%d\t%10s\n",
37              teacher.descrip.name,
38              teacher.descrip.age,
39              teacher.descrip.status,
40              teacher.grade,
41              teacher.lunch);
42      for(i=0;i<=2;i++)
43          printf("%s\t%d\t%c\t%d\t%s\n",
44                  student[i].descrip.name,
45                  student[i].descrip.age,
46                  student[i].descrip.status,
47                  student[i].grade,
48                  student[i].lunch);
49  }
```

运行结果如下：

```
the status of the teacher and students:
name            age status  grade    lunch
=======================================================
Mary Smith      34  M       94       Baloney Sandwich
Todd White      15  S       77       Peanut Butter
Peter Bush      14  S       87       Peanut Butter
Tiger Black     15  S       77       Peanut Butter
```

第 4～8 行定义了 person 结构体。第 9～13 行定义了 alldat 结构体，在这个结构体里有一个成员 descrip 就是 person 结构体类型的，见第 11 行。

第 13 行定义了 alldat 结构体类型的数组 student，第 14 行定义了 alldat 结构体类型的变量 teacher。第 16～20 行对 teacher 变量进行了赋值。其中，第 17～19 行完成了结构体类型成员变量的赋值。另外，还要注意字符数组的赋值是使用函数完成的。

第 21～27 行的循环完成了对结构型数组 student 的赋值。第 28～32 行对不便于循环赋值的数组成员变量单个进行了赋值。后面的语句完成了 teacher 和 student 内容的输出。

结构体的成员的数据类型为指针类型是完全允许的。但是要注意，指针类型的成员变量赋值时要分配内存空间，否则会出现内存混乱。

【例 10.5】 指针类型结构体成员变量。

程序如下：

```
1   void main()
2   {
```

```
3       struct person{
4           char*name;
5           int age;
6           char status;                /*M=married,S=single*/
7       };
8       struct person student,teacher,*point;
9       point=malloc(sizeof(struct person)); /*allocate memory for the point*/
10      printf("student's address is:%ld\n",&student);
11      printf("teacher's address is:%ld\n",&teacher);
12      printf("point is pointed to address:%ld\n",point);
13      printf("------before allocate memory------\n");
14      printf("student's name is pointed to address:%ld\n",student.name);
15      printf("teacher's name is pointed to address:%ld\n",teacher.name);
16      printf("point pointed name is pointed to address:%ld\n", point->name);
17      student.name=malloc(sizeof(char)*10);       /*allocate 10 bytes
                                                    for the name*/
18      strcpy(student.name,"Peter Nixon");
19      student.age=20;
20      student.status='S';
21      teacher.name=malloc(sizeof(char)*10);       /*allocate 10 bytes
                                                    for the name*/
22      strcpy(teacher.name,"Dan Bush");
23      teacher.age=36;
24      teacher.status='M';
25      point->name=malloc(sizeof(char)*10);        /*allocate 10 bytes
                                                    for the name*/
26      strcpy(point->name,"Tom Black");
27      point->age=26;
28      point->status='M';
29      printf("------after allocate memory------\n");
30      printf("student's name is pointed to address:%ld\n",
        student.name);
31      printf("teacher's name is pointed to address:%ld\n",teacher.name);
32      printf("point pointed name is pointed to address:%ld\n",point->name);
33      printf("name   \t age\tstatus\n");
34      printf("==================================\n");
35      printf("%s\t%d\t\%c\n",student.name,student.age,student.status);
36      printf("%s\t%d\t\%c\n",teacher.name,teacher.age,teacher.status);
37      printf("%s\t%d\t\%c\n",point->name,point->age,point->status);
38      free(student.name);
39      free(teacher.name);
40      free(point->name);
41      free(point);
42  }
```

程序的运行结果如下：

```
student's address is:98697156
teacher's address is:98697162
point is pointed to address:98634946
------before allocate memory------
student's name is pointed to address:98631680
teacher's name is pointed to address:98631680
point pointed name is pointed to address:98631680
------after allocate memory------
student's name is pointed to address:98634962
teacher's name is pointed to address:98634978
point pointed name is pointed to address:98634994
name        age status
==================================
Peter Nixon 20  S
Dan Bush    36  M
Tom Black   26  M
```

程序第 3～7 行定义了结构体类型 person，其中的成员变量 name 是字符串指针型变量。第 8 行定义了 person 结构型变量 student、teacher，同时还定义了这种类型的指针变量 point。

第 9 行为指针型变量 point 申请了内存空间。注意，malloc 函数的参数 sizeof（struct person）表示要申请的内存空间的大小。这也保证了指针型变量赋值前有了其对应的内存空间。

第 10～12 行显示了 student、teacher 的地址及 point 指针指向的内存地址。

第 13～16 行显示了对 student、teacher 及 point 指针所对应的成员变量 name 所指向的内存单元地址。注意，这时由于没有给 name 指针分配地址，因此 3 个变量的 name 指向了同一地址。

第 17～18 行给出了对 student 成员变量 name 的赋值方法。注意，这种类型的成员变量的赋值必须先申请内存空间。同样的情况在第 21～21 行及第 25～26 行，完成了对其他两个结构体变量的指针类型成员 name 的赋值。第 29～32 行显示了经过分配内存之后 3 个结构体变量的成员 name 的指针值已经发生了改变，它们各自指向了不同的内存区域。

第 33～37 行完成了对 3 个结构体变量的数据的显示。

第 38～40 行完成了 3 个结构体变量的指针型成员变量申请的内存空间的释放。第 41 行完成了对 point 申请空间的释放。注意第 40 行和第 41 行的顺序，如果先释放了 point，那么在释放 point->name 时就会因为空指针而出现错误。

从这个例子中，我们需要了解：如果结构体的数据成员是指针类型的，要注意指针类型的成员变量要赋值时要分配内存空间，如果是空间分配通过内存分配函数完成的，最后还要释放这块内存区域。

10.2.4　链表

链表是一种重要的数据结构。它是动态进行存储分配的一种结构。与结构数组相比，它的最大优势在于不用事先知道元素的个数。这在大型数据管理中是非常有用的。例如，在系

统中建立一个数组“struct Student stu[30000];”，还要寄希望于学生数不要超过 30000，否则程序会出现错误。但当学生数很少时，又会造成资源的浪费，降低软件的性能。链表可以很好地解决这一问题。

图 10.2 所示为单向链表和双向链表的示意图。

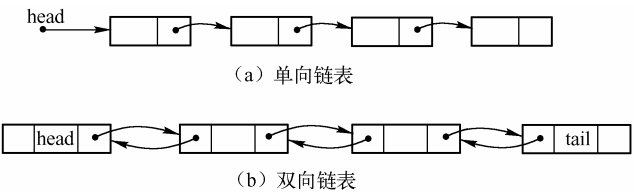


图 10.2　链表

链表看起来很复杂，实际上是一种间接寻址的形式。就像我们身边经常发生的事情，老师问班长“张三去哪里了？”，班长问组长“张三去哪里了？”，组长再问别的同学，最后找到了张三。

链表的每个元素称为一个结点。单向链表中，每个结点包括两个部分，即数据部分和指针部分，指针部分指向下一个结点，链表的末尾结点指针则为 NULL。双向链表中，每个结点包括 3 个部分，即数据部分、前指针和后指针，前指针指向前一个结点，后指针指向后一个结点。

关于链表的操作主要有 3 个，即查找、插入和删除。查找主要根据结点的数据部分与要找的数据比较而得。插入则是根据新插入结点的数值，确定要插入的位置，调整前后结点的指针完成。删除操作，则是将前后结点指针调整后，释放要删除结点的内存即可。

【例 10.6】　单向链表。

程序如下：

```
1   #include"stdio.h"            /*this is needed only to define the NULL*/
2   #include"stdlib.h"           /*this is needed for the function itoa()*/
3   #include"string.h"           /*this is needed for the function
                                  strcpy()*/
4   #define RECORDS 3
5   void main()
6   {
7       struct animal
8       {
9           char name[25];       /*The animals name*/
10          char breed[25];      /*The type of animal*/
11          int age;             /*The animals age*/
12          struct animal*next;/*a pointer to another record of this type*/
13      }*point,*start,*prior;  /*this defines 3 pointers,no
                                          variables*/
14      int index;
15      char temp[5];
16      /*the first record is always a special case*/
17      start=(struct animal*)malloc(sizeof(struct animal));
```

```
18      strcpy(start->name,"general");
19      strcpy(start->breed,"Mixed Breed");
20      start->next=NULL;
21      prior=start;
22      /*a loop can be used to fill in the rest once it is started*/
23      for (index=0;index<RECORDS;index++)
24      {
25          point=(struct animal*)malloc(sizeof(struct animal));
26          strcpy(point->name,"Frank");
27          strcat(point->name,itoa(index,temp,10));
28          strcpy(point->breed,"Laborador Retriever");
29          point->age=index;
30          prior->next=point;  /*point last"next"to this record*/
31          point->next=NULL;   /*point this"next"to NULL*/
32          prior=point;        /*this is now the prior record*/
33      }
34      /*now print out the data described above*/
35      point=start;
36      do
37      {
38          prior=point->next;
39          printf("%s is a%s,and is%d years old.\n",point->name,
40                  point->breed,point->age);
41          point = point->next;
42      }while (prior!=NULL);
43      /*good programming practice dictates that we free up the*/
44      /*dynamically allocated space before we quit*/
45      point=start;            /*first block of group*/
46      do
47      {
48          prior=point->next;  /*next block of data*/
49          free(point);         /*free present block*/
50          point=prior;        /*point to next*/
51      }while(prior!=NULL);    /*quit when next is NULL*/
52  }
```

运行结果如下：

```
general is a Mixed Breed,and is 0 years old.
Frank0 is a Laborador Retriever,and is 0 years old.
Frank1 is a Laborador Retriever,and is 1 years old.
Frank2 is a Laborador Retriever,and is 2 years old.
```

程序的第 1～3 行包含了必要的头文件。因为空指针 NULL 的定义在 stdio.h 文件中，而函数 itoa 完成整数到字符串的转化，函数的定义在 stdlib.h 中，函数 strcpy 的定义在 string.h 中。

第 7～13 行定义了结构体 animal，其中成员 next 为指向这个类型结构体的指针，是构成

单向链表的指针。第 13 行还定义了 3 个此类型结构体的指针变量 point、start、prior。

第 16～21 行完成了链表的首个元素的赋值。第 17 行首先为首个结构体元素分配必要的内存空间，然后由 start 指针指向此空间。第 18～19 行对此结构体元素赋值。由于此时下一个链表的元素位置还未确定，因此第 20 行里将元素的 next 成员赋值为 NULL。第 21 行将此时的 start 指针临时保护起来赋给变量 prior，目的是在后面链表的元素获得内存空间后再使此元素的 next 成员指向它。

第 22～33 行循环完成后面各元素加入链表的过程。为了区别新加入的 animal，在第 27 行将其 name 加入了后缀。在第 30 行将该结点赋给了 prior->next，完成了链表的一个结点加入。第 31 行将该结点的指针赋予 NULL 值，这样就保证了，即使该结点为最后结点，链表仍是完整的。第 32 行则将该结点作为下次插入结点的 prior 结点，为下次循环做好准备。注意，第 30～32 行在程序中的顺序。

第 34～42 行完成了整个链表的元素的输出。第 43～51 行当链表不再使用时，要将其占用的内存空间释放。

实际上，在数据结构中，还有像队列、堆栈、树等结构，各元素之间有着多种链接形式，结构体内的指针可以很好地表现这些情况。有兴趣的读者可以查看数据结构书籍。

10.3　共　用　体

共用体（union）是将不同的数据类型组合在一起，共同占用同一段内存的用户自定义数据类型。共用体的声明方法与结构体类似，只是将关键字 struct 改为了 union。图 10.3 所示为共用体数据类型 Number 的定义方法和内存占用情况示意图。

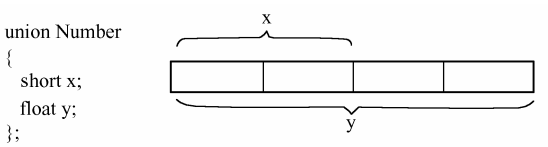


图 10.3　共用体数据类型定义及内存分配示意图

从中可以了解到共用体共有两个成员 short x 和 float y。但与结构体的不同之处在于这两个成员共用同一段内存区域。实际上，对于成员 x 的改变会影响到 y，当然对 y 的改变也可能影响到 x。共用体 Number 总共占用的内存长度为 4（最长的成员 y 的长度），而不是 6。

对比结构体和共用体，可以发现：

① 结构体和共用体都是由多个不同的数据类型成员组成的，但在任何同一时刻，共用体中只存放了一个被选中的成员，而结构体的所有成员都存在；

② 对于共用体的不同成员赋值，将会对其他成员重写，原来成员的值就不存在了，而对于结构体的不同成员赋值是互不影响的。

【例 10.7】　共用体。

程序如下：

```
1   void main()
2   {
3       union
```

```
4       {
5           short value;        /*This is the first part of the union*/
6           struct
7           {
8               char first;     /*These two values are the second*/
9               char second;
10          }half;
11      }number;
12      number.value=0x4241;    /*assign value to the first part of union*/
13      printf("%c%c\n",        /*means the second part is also changed*/
14              number.half.first,number.half.second);
15      number.half.first='a';  /*assign value to the second part of union*/
16      number.half.second='b'; /*means the first part is also changed*/
17      printf("%x\n",number.value);
18      getch();
19  }
```

程序的运行结果为：

```
AB
6261
```

例中第 3～11 行定义了共同体类型的变量 number。此共同体类型包括两个成员：成员 value 是一个整型变量，整型变量占用 2 个字节（16 个二进制位）；成员 half 是一个结构体变量，其中结构体又包括两个成员 first 和 second，由于结构体的两个成员都是字符型变量，每个字符型变量占用 1 个字节，因此整个结构体变量 half 也占用 2 个字节。因此，整个共用体变量 number 实际上就占用 2 字节，从 value 角度看它是一个整型变量，从 half 角度看它一个包括 2 个字符变量的结构体变量。

第 12 行完成对 number 变量的 value 成员的赋值。第 13～14 行，从 half 成员的角度进行了查看。0x4241 意味着低字节为 0x41，高字节为 0x42，它们正是字符 A、B 的 ASCII 码，显示结果也说明了这一点。

第 15～16 行对 number 的 half 成员进行赋值，由于字符 a、b 的 ASCII 码分别是 0x61 和 0x62，因此在第 17 行以十六进制数显示 number 的成员 value 时，可看出其值为 6261。

10.4　用typedef定义数据类型

除了使用 C 语言提供的标准类型名（如 int、char、float、double 等）、结构体和共用体之外，还可以用 typedef 声明新的类型名来代替已有的类型名。一般来说，typedef 新定义的类型用大写字母表示，以区别于原有的数据类型。

下面是几个例子：

```
typedef int INTEGER;
typedef float REAL;
typedef struct
{
```

```
    int month;
    int day;
    int year;
}DATE;
INTEGER iValue;
REAL rValue;
DATE birthDay;
```

这里，用 INTEGER 代替 int 类型，用 REAL 代替 float 类型，DATE 类型是一个结构体类型。程序段的最后，采用了新的类型名声明了 3 个变量。特别要注意，最后一行不要写成“struct DATE birthday”。

【例 10.8】 设有一个教师与学生通用的表格，教师数据有姓名、年龄、职业、办公室 4 项。学生有姓名、年龄、职业、班级 4 项。其中，学生的职业为“S”，教师职业为“T”。

程序如下：

```
1   void main()
2   {
3       typedef struct
4       {
5           char name[20];
6           int age;
7           char job;
8           union
9           {
10              int class;
11              char office[20];
12          }depa;
13      }PERSON;
14      PERSON body[2];
15      int i;
16      strcpy(body[0].name,"Bill Chan");
17      strcpy(body[1].name,"Robert Wilson");
18      body[0].age=18;
19      body[1].age=38;
20      body[0].job='S';
21      body[1].job='T';
22      body[0].depa.class=2;
23      strcpy(body[1].depa.office,"B1-101");
24      printf("name    \tage\tjob\tclass/office\n");
25      printf("=============================================\n");
26      for(i=0;i<2;i++)
27      {
28          if(body[i].job=='S')
29              printf("%s\t%d\t%c\t%d\n",body[i].name,body[i].age,
30                      body[i].job,body[i].depa.class);
```

```
31          else
32              printf("%s\t%d\t%c\t%s\n",body[i].name,body[i].age,
33                      body[i].job,body[i].depa.office);
34      }
35  }
```

程序运行结果如下：

```
name            age job class/office
=============================================
Bill Chan       18  S   2
Robert Wilson   38  T   B1-101
```

程序第 3～13 行用 typedef 定义了结构体类型 PERSON，其中第 8～12 行的共用体给出了学生和教师的不同成员变量。第 14 行声明了 PERSON 类型的数组 body。

第 16～23 行对 body 数组的两个元素进行赋值。其中第 22～23 行给出了学生和教师在 depa 成员的不同。

第 24～34 行对 body 数组的两个元素进行了展示。

10.5　枚 举 类 型

在实际问题中，有些变量的取值被限定在一个有限的范围内。例如，一个星期内只有 7 天，一年只有 12 个月等。为此，C 语言提供了一种称为“枚举”的类型。在枚举类型的定义中列举出所有可能的取值，被说明为该枚举类型的变量取值不能超过定义的范围。

10.5.1　枚举类型的定义

枚举类型定义的一般形式如下：

```
enum 枚举名
{
    枚举值表
};
```

在“枚举值表”中应罗列出所有可用值。这些值也称为枚举元素。

例如：

```
enum weekday
{sun,mou,tue,wed,thu,fri,sat};
```

该枚举名为 weekday，枚举值共有 7 个，即一周中的 7 天。凡被说明为 weekday 类型变量的取值只能是 7 天中的某一天。

如同结构体和共用体一样，枚举变量也可以用不同的方式说明，即先定义后说明、同时定义说明或者直接说明。设有变量 a、b、c 被说明为上述的 weekday，可以采用以下任意一种方式：

```
enum weekday
{   ......
};
enum weekday a,b,c;
```

或者：

```
enum weekday
{   ......
}a,b,c;
```

或者：

```
enum
{   ......
}a,b,c;
```

10.5.2　枚举类型变量的使用

枚举类型在使用中有以下规定。

① 枚举值是常量，不是变量，不能在程序中用赋值语句再对它赋值。例如，对枚举 weekday 的元素再进行以下赋值：“sun=5;”、“mon=2;”、“sun=mon;”是错误的。

② 枚举元素本身由系统定义了一个表示序号的数值，从 0 开始顺序定义为 0, 1, 2,…。如在 weekday 中，sun 值为 0，mon 值为 1，…，sat 值为 6。

【例 10.9】　枚举类型实质上属于整型的一种特殊情况。

程序如下：

```
1   void main()
2   {
3       enum weekday
4       {sun,mon,tue,wed,thu,fri,sat}a,b,c;
5       a=sun;
6       b=mon;
7       c=tue;
8       printf("%d,%d,%d",a,b,c);
9   }
```

运行结果如下：

```
0,1,2
```

③ 只能把枚举值赋予枚举变量，不能把元素的数值直接赋予枚举变量。

例如，“a=sum; b=mon;”是正确的，而“a=0; b=1;”是错误的。例如，一定要把数值赋予枚举变量，则必须用强制类型转换，如“a=(enum weekday)2;”，其意义是将顺序号为 2 的枚举元素赋予枚举变量 a，相当于“a=tue;”。

还应该说明的是枚举元素既不是字符常量，也不是字符串常量，使用时不要加单、双引

号。

④ 枚举类型也可以使用 typedef 重新定义。

【例 10.10】　枚举类型元素定义可以从任一整数开始。

程序如下：

```
1   void main()
2   {
3       typedef enum
4       {
5           Jan=1,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec
6       }MONTH;
7       MONTH m;
8       for(m=Jan;m<=Dec;m++)
9           printf("%d,",m);
10      printf("\n");
11      }
```

运行结果如下：

```
1,2,3,4,5,6,7,8,9,10,11,12,
```

默认的枚举类型的第 1 个元素对应整数值为 0，实际上还可以指定起始值。例如，在第 5 行中，Jan=1 表示枚举类型值从 1 开始。

习　题　10

10.1　学生成绩管理时，每个学生的数据包括 num、name、score。要求利用结构体定义数据类型，编写一个程序将 5 个学生的信息按学生成绩的高低排列输出。

10.2　结构体结构与变量定义如下：

```
struct person{
    int num;
    char *department;
}a,b;
```

编写一个程序给变量 a 赋值，并将 a 的值赋给 b。提示：使用内存动态分配函数。

10.3　有复数结构体，其中变量 r 和 i 分别表示复数的实部和虚部：

```
struct complex_number{
    double r,i;
};
```

编写函数 AddComplexNumber 能够完成两个复数的相加。提示：复数的加法规则是实部和实部相加得实部，虚部和虚部相加得虚部。

10.4　定义一个共用体 int_to_char，这个共用体有整型和字符型两个数据成员。声明一个这种类型的变量，把 49、65、97 以整型存储到变量中，以字符型重新读取这个值，并打

印出来。

10.5　创建 BOOL 枚举类型。BOOL 型中有 FALSE、TRUE 两个常量与之关联。FALSE 的值为 0，TRUE 的值为 1。编写一个程序，演示你的枚举类型可以正常工作。

10.6　使用 typedef 和 struct 语句创建称为 date_type 的日期类型。date_type 应包含 year、month、day 的整数域。编写一个程序，输入两个日期，然后比较它们，并按照日期的顺序输出它们。

10.7　一个 chain 类型结构体定义如下：

```
struct chain{
    int num;
    chain*next;
};
```

要求从键盘输入 *n* 个整数，建立一个顺序链表。当再次输入一个整数时，则从链表中查找该数据，找到后显示链表中该结点的数据及后面结点的数据，如果没有找到则显示“NO FOUND”。

10.8　有两个链表 a 和 b，设结点中包含学号和姓名。从 a 链表中删去与 b 链表中有相同学号的那些结点。

第 11 章　预处理命令与程序组织

学习目标

通过本章学习，应该掌握：

- C 语言的预处理命令，宏替换的使用方法
- 程序的组织方式

11.1　概　　述

在前面各章中，已多次使用过以“#”开头的预处理命令。例如，包含命令#include，宏定义命令#define 等。读者可能已经注意到这些命令不同于一般的 C 语言语句，每个命令后面并没有 C 语言语句的结束标志“;”，它们称为预处理命令。

所谓预处理是指在进行编译的第一遍扫描（词法扫描和语法分析）之前所做的工作。预处理是 C 语言的一个重要功能，它由预处理程序负责完成。当对一个源文件进行编译时，系统将自动引用预处理程序对源程序中的预处理部分进行处理，处理完毕自动进入对源程序的编译。虽然预处理命令实际上不是 C 语言的一部分，但却扩展了 C 语言程序设计的环境。

图 11.1 所示为创建计算机应用程序的完整过程。

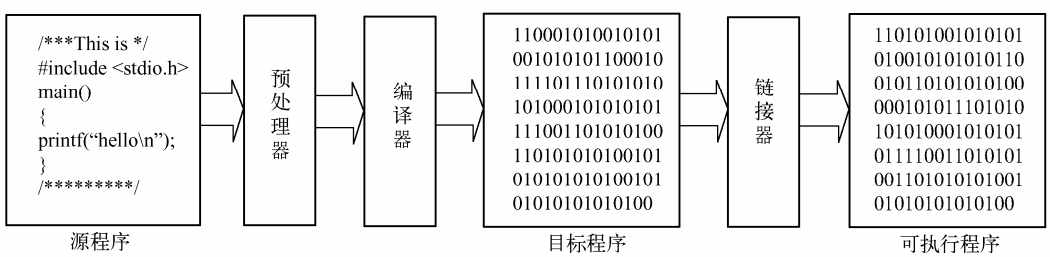


图 11.1　创建计算机应用程序的完整过程

C 语言提供了多种预处理功能，如宏定义、文件包含、条件编译等。合理地使用预处理功能编写的程序便于阅读、修改、移植和调试，也有利于模块化程序设计。表 11.1 所示为 ANSI 标准定义的 C 语言预处理命令，本章介绍常用的几种预处理功能。

表 11.1　ANSI 标准定义的 C 语言预处理命令

预处理命令	含　义
#define	定义命名常量和宏
#error	强迫编译程序停止编译，主要用于程序调试
#include	使编译程序将另一源文件嵌入，被读入的源文件名必须用双引号或尖括号括起来
#if	如果#if 后面的常量表达式为真，则编译它与#endif 之间的代码，否则跳过这些代码
#else	功能有点像 C 语言中的“else;”，#else 建立另一选择（在#if 失败的情况下）

续表

预处理命令	含　义
#elif	意义与 else if 相同，它形成一个 if else-if 阶梯状语句，可进行多种编译选择
#endif	标识一个#if 块的结束
#ifdef	条件编译的另一种方法是用#ifdef 命令，表示"如果有定义"
#ifndef	条件编译的另一种方法是用#ifndef 命令，表示 "如果无定义"
#undef	取消其后那个前面已定义过有宏名定义，该命令后前面的定义失效
#line	改变__LINE__与__FILE__的内容，它们是在编译程序中预先定义的标识符，分别表示源程序中当前行号和源文件名
#pragma	设定编译器的状态或者指示编译器完成一些特定的动作。#pragma 指令对每个编译器给出了一个方法，在保持与 C 语言和 C++语言完全兼容的情况下，给出主机或操作系统专有的特征

11.2　#define定义宏

#define 命令定义了一个标识符及一个串，可以用来创建命名常量和宏。在源程序中每次遇到该标识符时，均以定义的串代换它。ANSI 标准将标识符定义为宏名，将替换过程称为宏替换。关于命名常量的定义在前面的章节已有叙述，这里不再赘述。

在 C 语言源程序中允许用一个标识符来表示一个字符串，称为宏。被定义为宏的标识符称为宏名。在编译预处理时，对程序中所有出现的宏名，都用宏定义中的字符串去代换，称为宏代换或宏展开。宏代换是由预处理程序自动完成的。在 C 语言中，宏看起来像一个函数，通常用法也很相似，也分为有参数和无参数两种。但是，它不是一个函数，实际上在预处理器中找到宏的地方，就把宏的代码插入到程序中了。

#undef 用来取消宏定义，它与#define 对立：

#undef　标识符

如果被取消的宏实际上没有被#define 所定义，针对它的#undef 并不会产生错误。当一个宏定义被取消后，可以再度定义它。

【例 11.1】　创建一个求两个数中较大数的宏。

程序如下：

```
1   #define Max(value1,value2)(value1>value2)?value1:value2
2   void main()
3   {
4       int iVar1=3,iVar2=8;
5       float fVar1=3.5,fVar2=10.6;
6       double dVar1=10.0,dVar2=100.0;
7       printf("Two integers compare:\n");
8       printf("Max(%d,%d)=%d\n",iVar1,iVar2,Max(iVar1,iVar2));
9       printf("Two floats compare:\n");
10      printf("Max(%f,%f)=%f\n",fVar1,fVar2,Max(fVar1,fVar2));
11      printf("Two doubles compare:\n");
12      printf("Max(%f,%f)=%f\n",dVar1,dVar2,Max(dVar1,dVar2));
```

```
13  }
```

程序的第 1 行定义了宏 Max，在第 8、10、12 行完成了其调用。Max 宏接受两个参数，并比较两个参数的大小，返回最大的值。它与函数的最大不同之处在于其参数没有设置类型，实际上，如果用函数实现此程序的功能则需要设置 3 个函数，因为在调用时参数的类型是不一样的。

运行结果如下：

```
Two integers compare:
Max(3,8)=8
Two floats compare:
Max(3.500000,10.600000)=10.600000
Two doubles compare:
Max(10.000000,100.000000)=100.000000
```

当然，对参数不做类型检查可以看成宏的一个优点，有时也是一个缺点，有些编译器对于宏调用出现的语法问题无法发现。

如果使用连接符（即反斜杆“\”），宏可以超过 1 行。

【例 11.2】　超过 1 行的宏。

程序如下：

```
1   int func_swap(int A,int B)
2   {
3       int temp;
4       temp=A;
5       A=B;
6       B=temp;
7       return temp;
8   }
9   #define SWAP(A,B,TYPE)\
10  {       \
11      TYPE temp=A;\
12      A=B;        \
13      B=temp; \
14  }
15  void main()
16  {
17      int iVar1=10,iVar2=20;
18      printf("Before func_swap:iVar1=%d,iVar2=%d\n",iVar1,iVar2);
19      func_swap(iVar1,iVar2);
20      printf("After func_swap:iVar1=%d,iVar2=%d\n",iVar1,iVar2);
21      printf("Before SWAP:iVar1=%d,iVar2=%d\n",iVar1,iVar2);
22      SWAP(iVar1,iVar2,int);
23      printf("After SWAP:iVar1=%d,iVar2=%d\n",iVar1,iVar2);
24  }
```

程序的第 1～8 行定义了一个函数 func_swap，第 9～14 行定义了 1 个多行的宏 SWAP，读者可能很容易发现它们的相似之处，但是实际上却有很大不同。函数有返回值，而宏没有。其次，关于函数 func_swap 的形参和实参问题，使得本函数没有真正完成两个数据的交换，而对于宏 SWAP，因为它的调用本身是代码的替换，并没有为所谓的“形参”设置新的变量空间，所以其参数的互换可以完成，读者可以从运行结果发现这一点。

运行结果如下：

```
Before func_swap:iVar1=10,iVar2=20
After func_swap:iVar1=10,iVar2=20
Before SWAP:iVar1=10,iVar2=20
After SWAP:iVar1=20,iVar2=10
```

程序中的 SWAP 宏有参数 TYPE，通过调用时设置 SWAP (A, B, float)和 SWAP(A, B, double)可以完成 float 类型及 double 类型的变量替换。

另外，考虑到宏使用的特点是替换，因此，需要注意替换时可能引起的优先级问题。最好的办法就是将宏的参数直接用括号括起来。

【例 11.3】　宏替换引起的优先级问题。

程序如下：

```
1   #define SQUARE(x)x*x
2   #define SQUARE_M(x)(x)*(x)
3   void main()
4   {
5       int iVar=5;
6       printf("SQUARE(%d+3)=%d\n",iVar,SQUARE(iVar+3));
7       printf("SQUARE_M(%d+3)=%d\n",iVar,SQUARE_M(iVar+3));
8   }
```

程序运行结果如下：

```
SQUARE(5+3)=23
SQUARE_M(5+3)=64
```

第 1 行和第 2 行分别定义了两个宏，粗略感觉两个宏是一样的。但是在替换时却引起了不同的优先级问题，从而导致结果不一致。显然，SQUARE_M 得到的结果是我们期望的。实际上，SQUARE 宏在进行替换时，SQUARE(iVar+3)替换成了“5+3*5+3”，因而结果为“23”，而 SQUARE_M(iVar+3)替换成了“(5+3)*(5+3)”，结果为“64”。

11.3　预 定 义 宏

在 C 语言中预定义了一些有用的宏，见表 11.2。这些宏主要是提供当前编译的信息。宏__LINE__和__STDC__是整型常量，其他 3 个宏是字符串量。

表 11.2　C 语言的预定义宏

预 定 义 宏	描　　述
__LINE__	当前源代码中的行号，以十进制整数标注
__FILE__	被编译的文件的名字，以字符串常量标注
__DATE__	编译的日期，以"mm dd yyyy"格式的字符串标注，如"Nov 15 2007"
__TIME__	编译的时间，以"hh:mm:ss"格式的字符串标注
__STDC__	如果编译器接受 ANSI C 标准，那么值为 1

有时，通过这样的信息可以帮助区分不同程序或者同一程序的不同版本。

【例 11.4】　预定义宏。

程序如下：

```
1   void main()
2   {
3       printf("The File is: %s\n",__FILE__);
4       printf("It is compiled on%s at%s\n",__DATE__,__TIME__);
5       printf("The line number of this line is:%d\n",__LINE__);
6   }
```

运行结果如下：

```
The File is:SOURCE\11_4.C
It is compiled on Nov 15 2007 at 16:57:06
The line number of this line is:5
```

11.4　#include包含

预处理器发现#include 命令后，就会寻找后面跟着的文件名并把这个文件的内容包含到当前文件中。目前，我们的程序大多都使用了#include 命令，此命令用于文件包含。

为什么要包含文件呢？因为这些文件包含了编译器所需的信息。例如，stdio.h 文件通常包含 EOF、NULL、getchar 函数和 putchar 函数的定义。包含大型头文件并不一定显著增加程序的大小。很多情况下，头文件中的内容是编译器产生最终代码所需的信息，而不是加到最终代码里的具体语句。

#include 的格式如下：

```
#include"文件名"
#include<文件名>
#include 预处理标记
```

注意，所在的行不能含有除注释和空白符之外的其他任何内容。用尖括号“<”和“>”括起来表明这个文件是一个工程或标准头文件。查找过程会检查预定义的目录。可以通过设置搜索路径环境变量或命令行选项来修改这些目录。如果文件名用一对引号括起来则表明该文件是用户提供的头文件，查找该文件时将从当前文件目录（或文件名指定的其他目录）中寻找文件，然后再在标准位置寻找文件。

前面两种形式大家都很熟悉，“#include 预处理标记”中，“预处理标记”会被预处理器替换，替换的结果必须符合前两种形式中的某一种。

11.5　条 件 编 译

编译命令#if、#elif、#else、#endif 用于条件编译，基本格式如下：

```
#if 常量表达式 1
    语句……
#elif 常量表达式 2
    语句……
#elif 常量表达式 3
    语句……
#else
    语句……
#endif
```

#if 和#else 分别相当于 C 语言语句中的 if 和 else，它们根据常量表达式的值来判别是否编译后面的语句。#elif 相当于 C 语言中的 else if，#else 之后不带常量表达式。

常量表达式可以是包含宏、算术运算、逻辑运算等的合法 C 常量表达式。如果常量表达式为一个未定义的宏，那么它的值被视为 0。

使用这些条件编译命令可以方便地实现对源代码内容的控制。使用它们可以提升代码的可移植性，针对不同的平台使用不同的执行语句，也经常用于大段代码注释。

在判断某个宏是否被定义时，应当避免使用#if（因为该宏的值可能就是被定义为 0），而应当使用下一节介绍的#ifdef 或#ifndef。注意：#if、#elif、#else 之后的宏只能是对象宏，而不能是函数宏。

【例 11.5】　条件编译。

程序如下：

```
1   #include<stdio.h>
2   #define MY_VERSION 1
3   void main()
4   {
5   #if MY_VERSION==1
6       printf("This is Version 1.\n");
7   #elif MY_VERSION==2
8       printf("This is Version 2.\n");
9   #elif MY_VERSION==3
10      printf("This is Version 3.\n");
11  #else
12      printf("The Version not 1,2,3.\n");
13  #endif
14      printf("This line is excuted no matter how the version it is.\n");
15  }
```

运行结果如下：

```
This is Version 1.
This line is excuted no matter how the version it is.
```

程序的第 2 行定义了一个版本常量 MY_VERSION 的值为 1，因此，第 5 行的条件编译满足，故第 6 行程序得到执行。如果将第 2 行的常量定义分别为 2、3 或其他时，则第 8、10、12 行分别被执行，而第 14 行程序不管版本常量如何，都会被执行。有兴趣的读者，可以将第 2 行常量定义改为其他值一试。

本例对于编制在不同的平台下可以运行的程序有借鉴意义。类似于第 2 行中使用的版本常量，只需把平台的特征信息在程序中标出，即可保证程序在不同平台下执行不同的代码。

另外，预编译命令#ifdef、#ifndef 经常与#define 命令一起组成条件编译命令，具体的使用见 11.6 节。

11.6　程 序 组 织

尽管本书中的程序的源程序基本都只有 1 个文件。但是，对于大多数实际应用的程序，源程序一般需要分成若干文件。对于团队开发的较大程序，更是如此。那么如何对于多个源程序进行组织就是我们必须考虑的问题。

一般说来，团队成员必须达成一致，他们使用相同的方法组织他们的文件。为了保持一致性，通常把共享类型的定义放到头文件中，头文件以“.h”作为扩展名。函数的实现放到“.c”文件中。利用这种风格，类型的定义可以保持更好的独立，也容易被其他程序应用。

11.6.1　头文件

头文件是.c 文件之间共享信息的最常用工具。如果用得恰当，对于写出移植性好、可重用性好的程序很有帮助。

头文件应该包含需要共享的信息。当要确定哪些内容放到头文件中时，就需要问一下自己这些内容是否不止一个文件需要它。如果不是，它就不适合放到头文件中。

表 11.3 给出了常出现在头文件中的项目和一般在头文件中的习惯顺序。当然，顺序并不是什么规则，只是大家的一种习惯。而且这样的顺序便于在后面的定义中使用前面的定义。

表 11.3　头文件中的项目与习惯顺序

项　　目	说　　明
包含文件	用到的其他头文件
全局常量	在整个程序中有效的全局常量
类型定义	自定义的数据类型
枚举类型	自定义的枚举类型
结构体和共用体	自定义的结构体和共用体类型
宏	自定义的宏
原型	本头文件中使用的函数原型

另外，尽管#include 命令也可以包含.c 文件，但是一般不这样做。主要原因是，当该头文件被多个源程序包含时，会将该.c 文件多次编译。同样，将函数体放在头文件中也会引发这类问题。

11.6.2　程序组织与条件编译

有的时候，编制的.h 头文件之间可能会形成循环包含的现象，图 11.2 就给出了这样的一个例子。

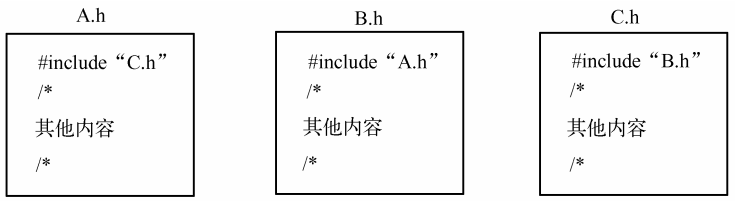


图 11.2　循环包含

C 语言编译器不允许这样做。如果碰到这种情况，一般来说意味着设计有缺陷，需要重新设计它们之间的关系。常用的解决思路是调整.h 文件的内容，将其中的某个.h 文件中的内容合并到别的.h 文件中，当然合并时需要理清它们之间的先后关系。

有些程序员把所有的头文件包含到一个文件中，并且借助条件编译命令避免头文件的重复包含。一般来说，使用#ifdef、#ifndef、defined 用来测试某个宏是否被定义，进而确定是否进行头文件包含。一般格式如下：

```
#ifndef __FILE_H__
#define __FILE_H__
#include"file.h"
#endif
```

针对图 11.2 给出的循环包含现象，采用的方法见例 11.6，这里增加了 1 个头文件 D.h。

【例 11.6】　条件编译与循环包含问题解决。

程序如下：

```
1   /*This is File A.h*/
2   #define A_H
3   #if!defined(D_H)
4   #include "D.h"
5   #endif
6   /*More stuff here*/
7   /*End of A.h*/

1   /*This is File  B.h*/
2   #define B_H
3   #if!defined(D_H)
4   #include"D.h"
```

```
#endif
/*More stuff here*/
/*End of B.h*/
```

```
/*This is File  C.h*/
#define C_H
#if!defined(D_H)
#include "D.h"
#endif
/*More stuff here*/
/*End of C.h*/
```

```
/*This is File  D.h*/
#define D_H
#ifndef A_H
#include"A.h"
#endif

#ifndef B_H
#include"B.h"
#endif

#ifndef C_H
#include "C.h"
#endif

/*More stuff here*/
/*End of D.h*/
```

这样，只要.c 文件都包含 D.h 头文件，就会自动包含 A.h、B.h、C.h 头文件，而且即使有多个.c 文件都包含 D.h 头文件，A.h、B.h、C.h 头文件也只包含 1 次。当然，这样的编译会增加一定的编译时间。

习　题　11

11.1　说明在程序中使用宏而不用函数有什么优缺点。

11.2　说明怎样使用条件编译编写移植性高的宏。

11.3　分析下面程序的执行结果，从中体会宏定义的有效范围。

```
#define E 2.71828
void fun();
void main()
{
    fun();
```

```
6       printf("%f",E);
7   }
8   #undef E
9   #define E"hello"
10  void fun()
11  {
12      char*s=E;
13      printf("%s\n",s);
14  }
```

11.4　用条件编译编写一个宏 DebugPause。如果常量 DEBUG 在程序中定义了，则执行 DebugPause 宏暂停程序执行，直至用户按下回车键。如果常量 DEBUG 没有定义，则不调用 DebugPause 宏。

11.5　定义常量 OS_VERSION 标识程序的操作系统版本，程序版本有可能是 UNIX、Windows、DOS，对于不同的版本系统给出相应的提示，要求使用条件编译。

第 12 章　文 件 操 作

学习目标

通过本章学习，要求掌握：

- 文件的概念与文件的打开方式
- 文本文件、二进制文件的读、写操作
- 标准文件的概念与使用

12.1　概　　述

文件是以整体形式存储在计算的磁盘、磁带、光盘上的信息的集合，是在计算机中以实现某种功能或某个软件的部分功能为目的而定义的一个单位。文件操作的目的就是读取或者保存信息。在 C 语言中，提供了很多文件操作类的函数，借助于这些函数，使得我们操作文件更加简单、方便。

在 C 语言中，文件也被称做流。从本质上来说，文件都是以编码形式存储的信息流，只是由于编码的差异，使得我们把文件分为文本文件和二进制文件。文本文件是指以 ASCII 码或者其他文字语言的交换编码存储的文件，可以直接在屏幕或打印机输出为人们识别的信息。而二进制文件可以包含任意有效的二进制数，而对于这些二进制数的含义则因为编码的差异而不同。例如，有的二进制文件是由可执行的计算机机器指令组成的，可以称为可执行文件等。

就像每个人都有一个名字一样，一般来说文件都有一个名字。而对应文件的操作，一般来说都是通过名字和文件的性质打开文件，得到一个文件句柄的指针，然后，以此为基础对文件进行读写，操作完成之后还要关闭文件。所有对于文件的操作都离不开文件句柄。图 12.1 给出了文件操作的基本步骤。

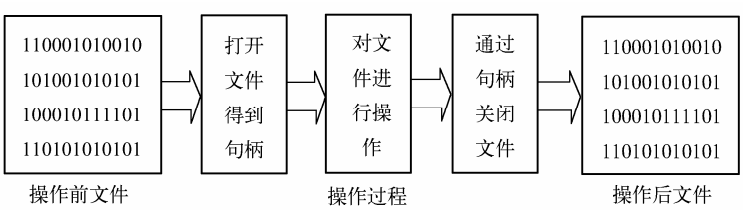


图 12.1　文件操作的基本步骤

12.2　文件句柄与文件打开和关闭

文件系统中，文件句柄是一个重要的概念。实际上，每个被使用的文件，都在内存中开辟一个区域，用来存放文件的有关信息，例如，文件的状态以及当前位置等。这些信息保存在一个结构体变量中，结构体的类型是由系统定义的。在 C 语言中，该结构体取名为 FILE。

一般来说，不需要知道 FILE 结构的细节，这些信息都是给操作系统为操作文件提供的。

要处理文件，程序必须创建文件句柄指针变量并打开文件，然后用输入函数从文件中读取数据，用输出函数向文件中写入数据。操作结束时，还要关闭文件。例 12.1 演示了文件操作的过程。

【例 12.1】　文件操作过程演示。

程序如下：

```
#include<stdio.h>
void main()
{
    FILE*filePointer=NULL;      /*定义文件句柄指针*/
    filePointer=fopen("A1.txt","r");/*打开指定文件，获得文件句柄指针*/
    if(filePointer)                 /*如果指针不为空，则打开成功*/
    {
        /*相应的文件操作*/
        fclose(filePointer);            /*关闭文件*/
    }
}
```

不论对哪种类型的文件，程序都调用 fopen 函数打开文件。fopen 函数的一般格式为：

文件句柄指针=fopen(文件名字符串，文件打开方式字符串)

fopen 函数的第 1 个参数是包含文件名的字符串，第 2 个参数是包含一个或多个文件说明符的字符串。可用的文件打开方式说明符如表 12.1 所示。

表 12.1　文件打开方式

说明符字符串	描　述
"r"　（只读）	以只读方式打开文本文件，如果文件不存在，也不创建它
"w"　（只写）	以只写方式打开文本文件，如果文件已存在，则把它清空，若文件不存在，则创建它
"a"　（追加）	以写的方式打开文本文件，新的数据追加到文件尾。如果文件不存在，则创建它
"r+"　（读写）	以读写方式打开文本文件，假定先读，所以当前位置指针在文件开头。如果文件不存在，则创建它
"w+"　（写读）	以写读方式打开文本文件，假定先写。若文件已存在，则把它清空；若文件不存在，则创建它
"a+"　（读写）	以读写的方式打开文本文件，假定先写，当前位置指针在文件尾，整个文件都可以读，但已经存在的数据不能覆盖。如果文件不存在，则创建它
"rb"　（只读）	和"r"一样，但用于二进制文件
"wb"　（只写）	和"w"一样，但用于二进制文件
"ab"　（追加）	和"a"一样，但用于二进制文件
"rb+"　（读写）	和"r+"一样，但用于二进制文件
"wb+"　（写读）	和"w+"一样，但用于二进制文件
"ab+"　（读写）	和"a+"一样，但用于二进制文件

在这里，读者可能会注意到文件位置指针的概念，我们不必担心对文件的位置指针的管理，实际上标准的 C 语言库函数为我们提供了这样的函数。

不论对哪种类型的文件，程序都调用 fclose 函数关闭文件。fclose 函数的一般格式为：

```
fclose(文件句柄指针);
```

应该养成结束文件操作后关闭文件的习惯，如果不关闭文件，可能会发生丢失数据的现象。

12.3　文本文件的操作

严格说来，文本文件也是二进制文件。但由于存储的信息是采用 ASCII 编码或其他文字语言的交换编码存储的文件，其信息一般可以直接显示，故而成为单独的一类。

一般来说，程序从头处理文本文件到文件尾，而我们可以通过调用 feof 函数知道哪里是文件尾。

关于文本文件的操作过程在 12.2 节已有介绍，表 12.2 给出的是几个常用的输入、输出函数，表中省略了各个函数的具体参数。

表 12.2　文本文件的输入、输出函数

函　数	描　述
fgetc()	从文件中获得一个字符
fgets()	从文件中获得一个字符串
fscanf()	和 scanf()一样，但用于文件
fputc()	向文件写入一个字符
fputs()	向文件写入一个字符串
fprintf()	和 printf()一样，但用于文件

【例 12.2】　显示一个文本文件内容。假设磁盘上的文本文件是“D:\TC\SOURCE\temp.c”。程序如下：

```
1   #include<stdio.h>
2   void main()
3   {
4       FILE*fp;
5       int tempChar;
6       fp=fopen("D:\\TC\\SOURCE\\temp.c","r");
7       if(fp==NULL)
8       {
9           printf("Open File error,Please check if the file exists or
not!\n");
10          return;
11      }
12      else
13      {
```

```
14          while(!feof(fp))
15          {
16              tempChar=fgetc(fp);
17              if(!feof(fp))
18                  putchar(tempChar);
19          }
20          fclose(fp);
21      }
22  }
```

程序的第 4 行定义了文件句柄 fp，在第 6 行以只读形式打开文件，文件名给出的是全路径。注意，给出路径时因为有“\”而使用了转义符“\”。第 7 行判断文件是否存在，如果 fp 为 NULL，则文件不存在。第 14～19 行则是每次读出一个字符，并进行显示。其中，feof 函数判断文件是否已到文件尾，此时文件指针根据文件读出情况自动调整。读者也许会问第 17 行的判断是否有必要，可以告诉大家这是必要的！因为文件结束标志也是一个字符，但是是不可显示的字符，当文件指针到达结束标志时，while 循环的条件仍然满足，此时变量 tempChar 就是这个标志，因此，如果在第 17 行不进行判断而直接显示，就会有些问题。第 20 行关闭文件。

运行结果显示了文件的具体内容，这里不妨假设文件内容如下：

```
void main()
{
    int n;
    printf("Input number:\n");
    scanf("%d",&n);
    printf("The number is%d\n",n);
}
```

把例题 12.2 改写一下，使用 fgets 函数读入文本，见例 12.3。

【例 12.3】　使用 fgets 函数读取文本。

程序如下：

```
1   #include<stdio.h>
2   #define MAX_STRINGLENGTH 100
3   void main()
4   {
5       FILE*fp;
6       char tempString[MAX_STRINGLENGTH];
7       fp=fopen("D:\\TC\\SOURCE\\temp.c","r");
8       if(fp==NULL)
9       {
10          printf("Open File error,Please check if the file exists or
not!\n");
11          return;
12      }
```

```
13      else
14      {
15          while(!feof(fp))
16          {
17              if(fgets(tempString,MAX_STRINGLENGTH,fp)!=NULL)
18                  puts(tempString);
19          }
20          fclose(fp);
21      }
22  }
```

程序的第 2 行定义了一个常量，设置每行读取字符串的最大长度。程序的第 17 行展示了 fgets 函数的使用。函数的第 1 个参数是保存文本的数组；第 2 个参数是能保存在数组中的最大字符数，如果字符串长度超过这个长度，那么只包含这个长度的字符数；第 3 个参数是文件的句柄指针。第 18 行为屏幕显示这个刚读入的字符串。

运行结果如下：

```
void main()

{

    int n;

    printf("Input number:\n");

    scanf("%d",&n);

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

}
```

细心的读者会注意到，这里的输出文本行之间都有一个空行。这是由 fgets 函数和 puts 函数共同造成的。fgets 函数从文本文件读入一行完整文本，包含了'\n'，并在后面追加一个字符串结束符'\0'，而 puts 函数打印字符串后再打印一个'\n'，从而导致多出了空行。

【例 12.4】 使用 fscanf 和 fprintf 函数完成文本文件的输入和输出。从键盘输入 3 个学生的姓名与成绩，将其保存在磁盘上，然后读出来显示。

程序如下：

```
1   #include<stdio.h>
2   #define MAX_NAMELENGTH 10
3   #define MAX_NUMBER 3
4   void main()
5   {
6       FILE*fp;
7       char tempName[MAX_NAMELENGTH];
```

```
8       double tempGrade;
9       int i;
10      fp=fopen("D:\\TC\\SOURCE\\temp12-4.txt","w");
11      if(fp==NULL)
12      {
13          printf("Create File error!\n");
14          return;
15      }
16      else
17      {
18          for(i=0;i<MAX_NUMBER;i++)
19          {
20              printf("Name and Grade:");
21              scanf("%s%lf",tempName,&tempGrade);
22              fprintf(fp,"%s%lf\n",tempName,tempGrade);
23          }
24          fclose(fp);
25      }
26      fp=fopen("D:\\TC\\SOURCE\\temp12-4.txt","r");
27      if(fp==NULL)
28      {
29          printf("Create Read error!\n");
30          return;
31      }
32      else
33      {
34          for(i=0;i<MAX_NUMBER;i++)
35          {
36              fscanf(fp,"%s%lf",tempName,&tempGrade);
37              printf("%s%f\n",tempName,tempGrade);
38          }
39          fclose(fp);
40      }
41  }
```

程序运行结果如下：

```
Name and Grade:zhu 100
Name and Grade:Gao 90.5
Name and Grade:Huang 80
zhu 100.000000
Gao 90.500000
Huang 80.000000
```

第 11～25 行完成了文件的写入工作，而第 26～40 行完成了文件内容的读出工作。需要说明的是，fscanf 和 fprintf 函数的第 1 个参数是指向要读取文本文件的句柄，其他参数都与

scanf 函数和 printf 函数完全一样。还要注意，与 scanf 函数一样，fscanf 函数在遇到空白字符时停止读入，包括空格和回车。

12.4　二进制文件的操作

二进制文件的处理很多类似于文本文件，但是在文件打开时，需要指明打开方式里包含“b”。表 12.3 给出了二进制文件的输入/输出函数。

表 12.3　二进制文件的输入/输出函数

函　数	描　述
fgetc()	从文件中获得一个字节
fread ()	从文件中获得一数据块
fputc()	向文件写入一个字节数据
fwrite()	向文件写入一个数据块

对于二进制文件，程序很少按顺序将文件从头到尾进行访问。更常见的是，随机访问文件的内容，这需要对文件的位置进行控制。表 12.4 给出了文件定位的函数。需要说明的是文件定位的函数虽然在本节给出，但对于文本文件也是适用的。例如，对于例 12.4 给出的程序，将第 10 行的文件打开方式改为“w+”，则在文件内容写入完成后，不用关闭文件，增加语句 rewind(fp)就可以直接读入文件内容进行显示，有兴趣的读者不妨一试。

表 12.4　文件定位函数

函　数	描　述
fseek()	将文件位置指针移动到指定的位置。其中，位置说明符为：SEEK_SET 从文件头查找，SEEK_CUR 从当前位置指针的当前位置找，SEEK_END 从文件尾查找
ftell ()	返回文件的当前位置指针的字节偏移数
rewind()	将文件的当前位置指针指向文件头

【例 12.5】　从键盘输入两个学生数据，写入一个文件中，再读出这两个学生的数据显示在屏幕上。

程序如下：

```
1   #include<stdio.h>
2   struct stu
3   {
4       char name[10];
5       int num;
6       int age;
7       char addr[15];
8       }boya[2],boyb[2],*pSource,*pDest;
9   void main()
10  {
11      FILE*fp;
```

```
12      int i;
13      pSource=boya;
14      pDest=boyb;
15      if((fp=fopen("temp12-5.dat","wb+"))==NULL)
16      {
17          printf("Cannot create file!");
18          exit(1);
19      }
20      printf("Input The Student's Name,Num,Age,Adress\n");
21      for(i=0;i<2;i++,pSource++)
22          scanf("%s%d%d%s",pSource->name,&pSource->num,
23                  &pSource->age, pSource->addr);
24      pSource=boya;
25      fwrite(pSource,sizeof(struct stu),2,fp);
26      rewind(fp);
27      fread(pDest,sizeof(struct stu),2,fp);
28      printf("\nname\tnumber\tage\t addr\n");
29      for(i=0;i<2;i++,pDest++)
30          printf("%s\t%5d\t%3d\t%s\n",pDest->name,pDest->num,
31                  pDest->age,pDest->addr);
32      fclose(fp);
33  }
```

运行结果如下：

```
Input The Student's Name,Num,Age,Adress
Wang 1001 21 Beijing↙
Li 1002 20 Shanghai↙

name    number  age addr
Wang    1001    21  Beijing
Li      1002    20  Shanghai
```

程序定义了一个结构体 stu，说明了两个结构体数组 boya 和 boyb，以及两个结构体指针变量 pSource 和 pDest。pSource 指向 boya，pDest 指向 boyb。程序第 15 行以读写方式打开二进制文件“temp12-5.dat”，输入两个学生数据之后，写入该文件中，然后第 26 行把文件内部位置指针移到文件首，读出两个学生数据后，在屏幕上显示。注意这里 fread 和 fwrite 函数对数据块的设置形式。

12.5　标 准 文 件

前面介绍的文本文件和二进制文件都是指存储在磁盘或磁带上的信息流。实际上，针对每个程序，系统还打开了 3 个标准文件，它们是 stdin、stdout、stderr，分别是 standard input（标准输入）、standard output（标准输出）、standard error（标准出错）的缩写。文件 stdin 是程序可以读取其输入的位置，默认情况下，从键盘读取。stdout 是程序写入其输出的位置，

除非将其定向到其他地方，否则标准输出通常出现在显示器上。stderr 是程序发出错误和特殊消息的地方，除非将其定向到其他地方，否则标准错误通常出现在显示器上。实际上，不论在标准文件系统还是非标准文件系统中，文件结构只需要用文件句柄或文件代号代替，前面介绍的文件操作函数也均适用。

【例 12.6】　标准文件使用举例。

程序如下：

```
1   #include<stdio.h>
2   #include<io.h>
3   #include<stdlib.h>
4   int main(void)
5   {
6       FILE*fp;
7       chars[80];
8       int t;
9       if((fp=fopen("test","w"))==NULL)
10      {
11          printf("Cannot open file.\n");
12          exit(1);
13      }
14      printf("Enter a string and a number:");
15      fscanf(stdin,"%s%d",s,&t);          /*read from keyboard*/
16      fprintf(fp,"%s%d",s,t);             /*write to file*/
17      fclose(fp);
18      if((fp=fopen("test","r"))==NULL)
19      {
20          fprintf(stderr,"Cannot open file.\n");
21          exit(1);
22      }
23      fscanf(fp,"%s%d",s,&t);             /*read from file*/
24      fprintf(stdout,"%s%d",s,t);         /*print on screen*/
25      fclose(fp);
26      return 0;
27  }
```

运行结果如下：

```
Enter a string and a number: hello 100 ↙
hello 100
```

程序在第 15、20 行和第 24 行分别使用了标准文件，而在第 9、17 行打开和关闭普通文件。可以看出标准文件在使用时是不用打开和关闭的，因为这些工作由系统在运行程序时自动完成。

12.6　其他文件操作函数

除了前面的文件操作函数外，C 语言标准库还提供了其他的文件操作函数。这里介绍其中的 3 个，见表 12.5。

表 12.5　几个常用的文件函数

函　数	描　述
fflush()	强迫将缓冲区内的数据写回参数指定的文件中
ferror ()	检测流上的错误
uugetc()	将字符写回参数所指定的文件流。这个写回的字符会由下一个读取文件流的函数取得

下面通过实例对这几个函数的用法进行说明。

【例 12.7】　检测流上的错误。

程序如下：

```
1   #include<stdio.h>
2   int main()
3   {
4       FILE*stream;
5       /*open a file for writing*/
6       stream=fopen("DUMMY.FIL","w");
7       /*force an error condition by attempting to read*/
8       (void)fgetc(stream);
9       if(ferror(stream))      /*test for an error on the stream*/
10      {
11          /*display an error message*/
12          printf("Error reading from DUMMY.FIL\n");
13          /*reset the error and EOF indicators*/
14          clearerr(stream);
15      }
16      fclose(stream);
17      return 0;
18  }
```

运行结果如下：

```
Error reading from DUMMY.FIL
```

程序的第 8 行从文件流中读取一个字符，从而为可能产生流上的错误提供了条件。因为第 6 行打开的文件是只写的，所以这里会产生一个错误，这个错误在第 9～15 行中进行处理。第 14 行将错误标记复位。

【例 12.8】　fflush 函数的使用。

程序如下：

```
#include<stdio.h>
char mybuffer[80];
int main()
{
    FILE*pFile;
    pFile=fopen("D:\\TC\\source\\temp12-8.txt","r+");
    if (pFile==NULL) perror ("Error opening file");
    else
    {
        fputs("test",pFile);
        /*flushing or repositioning required*/
        fflush (pFile);
        fgets(mybuffer,80,pFile);
        puts(mybuffer);
        fclose(pFile);
        return 0;
    }
}
```

程序的第 2 行开辟了一个 80 字节的缓冲区 mybuffer，第 6 行以读写形式打开文本文件。第 10 行将字符串“test”写入文件，由于此时文件位置指针在文件头，故文件前面的字符被替换成“test”。第 12 行清空文件缓冲区，并将缓冲区的修改写回文件。当第 13 行再次执行读取字符串时，什么也读取不到，当然在第 14 行也就显示不出来。此外，第 7 行还有函数 perror，这个函数除了显示作为参数的字符串外，还显示正常的错误信息。

【例 12.9】　uugetc 函数的使用。

程序如下：

```
#include<stdio.h>
#include<ctype.h>
int main()
{
    int i=0;
    char ch;
    puts("Input an integer followed by a char:");
    /*read chars until non digit or EOF*/
    while((ch=getchar())!=EOF&&isdigit(ch))
        i=10*i+ch-48;   /*convert ASCII into int value*/
    /*if non digit char was read,push it back into input buffer*/
    if(ch!=EOF)
        ungetc(ch,stdin);
    printf("i=%d,next char in buffer=%c\n",i,getchar());
    return 0;
}
```

运行结果如下：

```
Input an integer followed by a char:
123g↙
i=123,next char in buffer=g
```

程序的第 9～10 行循环读取缓冲区内的数字，并将其转换成整数，当缓冲区中遇到非数字时结束循环。当程序执行到第 13 行时，ch 中为第 1 个非数字的字符，此时执行 uugetc 函数，将此字符写回 stdin（即键盘缓冲区），因此在第 14 行用 getchar 读取字符时，仍然从缓冲区读取了该字符。另外，第 13 行还使用了标准文件 stdin。

习　题　12

12.1　如果计算机的所有信息最终都是以二进制保存的，为什么 C 语言中还有二进制文件和文本文件之分？

12.2　解释顺序访问和随机访问之间的区别。

12.3　文件的打开和关闭的含义是什么？为什么要打开和关闭文件？

12.4　编写程序将两个文本文件的内容合成一个新的文本文件。

12.5　编写程序统计文本文件中每个单词出现的次数，并把统计结果追加在原文本文件末尾。

12.6　有 5 个学生，每个学生有 3 门课的成绩，从键盘输入学生数据（学生姓名和 3 门课成绩），然后计算出各自的总成绩，并将原有学生信息及总成绩以二进制文件形式存入磁盘。

12.7　磁盘文件中有一批学生数据，包括姓名、3 门课的成绩及总成绩，要求将这批学生按照分数高低排序，然后将排过序的学生成绩写回原磁盘文件。

12.8　磁盘文件中有一批学生数据，包括姓名、3 门课的成绩及总成绩，要求将这批学生中有不及格课程的学生信息删除，然后把其余学生的信息写回原磁盘文件。

第 13 章　位　操　作

学习目标

通过本章学习，要求掌握：

- 位运算的有关概念和运算规律
- 位段的概念

13.1　概　　述

计算机中的所有信息都是由二进制数据构成的，信息的实质是编码。对编码的不同解释决定了它们各自代表的意义。二进制信息的每一位或 0 或 1，8 个二进制位组成 1 个字节。一般来说，计算机中的信息多数是以字节为单位进行存储的。例如，字符类型数据存放的是 ASCII 码，在计算机中占用 1 个字节。而 16 位整数，则存放该整数的补码，其中高位表示符号，0 表示正数，1 表示负数等。

C 语言是为描述系统设计的，因此它具有很多低级语言所能完成的功能，因此也提供了许多二进制数据操作的手段。当然，要很好地使用它们，还需要对信息本身的意义有较好的理解。例如，C 语言提供了很多硬件操作的方法，要使用这些方法就必须懂得硬件操作的需要是什么。

13.2　位运算符和位运算

C 语言提供的位运算符如表 13.1 所示。

表 13.1　C 语言提供的位运算符

运　算　符	含　　义	运　算　符	含　　义
&	按位与	^	按位异或
|	按位或	<<	左移位
~	按位取反	>>	右移位

需要说明的是，这些运算符除了“~”取反是单目运算符以外，其余都是双目运算符。另外，参与运算的只能是整型数或字符型数，浮点数不能参与运算。

13.2.1　移位运算

移位包括左移和右移。

1．<<左移位

左移位就是将一个数的各二进制位左移指定位数，右边补 0，高位左移溢出后，舍弃不用。把一个数左移 1 位，相当于用该数乘以 2，当然也存在溢出的问题。

2．>>右移位

右移位就是将一个数的各二进制位右移指定位数，左边补该数的符号位，低位右移溢出后，舍弃不用。把一个数右移 1 位，相当于用该数除以 2，当然也存在舍弃小数的问题。

需要说明的是，对于无符号数，高位不再代表符号，因此在右移时，高位会补 0。

通过下面的示例，读者可以加深对移位的理解。

【例 13.1】　左移位。

程序如下：

```
#include<stdio.h>
void main()
{
    int a=-1;
    int i=0;
    for (i=0;i<17;i++)
    {
        printf("%d<<%d=%d\t",a,i,a<<i);
        printf("0x%x<<%d=0x%x\n",a,i,a<<i);
    }
}
```

运行结果如下：

```
-1<<0=-1        0xffff<<0=0xffff
-1<<1=-2        0xffff<<1=0xfffe
-1<<2=-4        0xffff<<2=0xfffc
-1<<3=-8        0xffff<<3=0xfff8
-1<<4=-16       0xffff<<4=0xfff0
-1<<5=-32       0xffff<<5=0xffe0
-1<<6=-64       0xffff<<6=0xffc0
-1<<7=-128      0xffff<<7=0xff80
-1<<8=-256      0xffff<<8=0xff00
-1<<9=-512      0xffff<<9=0xfe00
-1<<10=-1024    0xffff<<10=0xfc00
-1<<11=-2048    0xffff<<11=0xf800
-1<<12=-4096    0xffff<<12=0xf000
-1<<13=-8192    0xffff<<13=0xe000
-1<<14=-16384   0xffff<<14=0xc000
-1<<15=-32768   0xffff<<15=0x8000
-1<<16=0        0xffff<<16=0x0
```

可以看出，整型数据是采用补码表示的。每左移 1 位相当于数据乘以 2，当移位结果超出数据表示范围时，结果就不再正确。

【例 13.2】　右移位。

程序如下：

```
#include<stdio.h>
void main()
{
    int a=0xc57f;
    int i;
    for(i=0;i<16;i++)
    {
        printf("%d>>%d=%d\t",a,i,a>>i);
        printf("0x%x>>%d=0x%x\n",a,i,a>>i);
    }
}
```

运行结果如下：

```
-14977>>0=-14977    0xc57f>>0=0xc57f
-14977>>1=-7489     0xc57f>>1=0xe2bf
-14977>>2=-3745     0xc57f>>2=0xf15f
-14977>>3=-1873     0xc57f>>3=0xf8af
-14977>>4=-937      0xc57f>>4=0xfc57
-14977>>5=-469      0xc57f>>5=0xfe2b
-14977>>6=-235      0xc57f>>6=0xff15
-14977>>7=-118      0xc57f>>7=0xff8a
-14977>>8=-59       0xc57f>>8=0xffc5
-14977>>9=-30       0xc57f>>9=0xffe2
-14977>>10=-15      0xc57f>>10=0xfff1
-14977>>11=-8       0xc57f>>11=0xfff8
-14977>>12=-4       0xc57f>>12=0xfffc
-14977>>13=-2       0xc57f>>13=0xfffe
-14977>>14=-1       0xc57f>>14=0xffff
-14977>>15=-1       0xc57f>>15=0xffff
```

可以看出，整数−14977 的补码是 0xc57f。整型数据每右移 1 位相当于数据除以 2，同时还应注意到−1/2=−1。

13.2.2　其他位运算

除了移位运算，二进制数据的位运算还包括与、或、非、异或 4 种类型。这些操作都是按二进制位进行的，见表 13.2。

表 13.2　位运算真值表

位 运 算 符	运 算 结 果
与&	0&0=0，0&1=0，1&0=0，1&1=1
或|	0|0=0，0|1=1，1|0=1，1|1=1
非~	~0=1，~1=0
异或^	0^0=0，0^1=1，1^0=1，1^1=0

从数据本身的角度来看，位运算本身的意义不大，但是在与计算机硬件相关的很多操作中，位运算有其他运算无可比拟的优势。例如，用一个 16 位数据代表设备的多个状态，如果只对某些状态感兴趣，那么可以设置一个数据与设备的状态进行与运算，通过得到的结果即可改变设备的状态。再例如，用一个 16 位数据代表多个命令，如果只想发送几个命令，可以通过将这几个命令的对应位置 1，然后作为命令字发送出去，再把命令字的对应位设为 1 与原命令字相或就可以得到新命令字。当然，具体情况的确定需要对硬件的要求比较清楚才可以完成。下面纯粹从数据处理的角度举例说明位运算的使用。

【例 13.3】　将数据的最末位置位为 0。

程序如下：

```
1   #include<stdio.h>
2   void main()
3   {
4       int aValue;
5       int i;
6       for(i=0;i<5;i++)
7       {
8           printf("Please input a integer:");
9           scanf("%d",&aValue);
10          aValue=aValue&0xfffe;
11          printf("The new value is:%d\n",aValue);
12      }
13  }
```

运行结果如下：

```
Please input a integer:19↙
The new value is:18
Please input a integer:1221↙
The new value is:1220
Please input a integer:12↙
The new value is:12
Please input a integer:17↙
The new value is:16
Please input a integer:400↙
The new value is:400
```

本程序完成了整型数据最低位置 0，不管数据原来最低位是 0 还是 1。这里可以把 0xfffe 看成一个屏蔽码，它屏蔽了 aValue 的最低位数据。屏蔽码在计算机硬件相关的编程中经常使用。关于数据运算的过程，图 13.1 所示为数据 19 被屏蔽之后变成 18 的情况，其他数据运算过程读者可以自己演习一下。

```
    0000000000010011      ……19 的补码
&   1111111111111110      ……0xfffe
    0000000000010010      ……18 的补码
```

图 13.1　数据运算过程

【例 13.4】　字符串的简单加密。

程序如下：

```
1   #include<stdio.h>
2   #include<string.h>
3   void main()
4   {
5       char maskChar=0x55;
6       char *sourceString="Hello";
7       char *destString1;
8       char *destString2;
9       int i;
10      destString1=malloc(strlen(sourceString));
11      destString2=malloc(strlen(sourceString));
12      for (i=0;i<strlen(sourceString); i++)
13          destString1[i]=sourceString[i]^maskChar;
14      destString1[i]='\0';
15      for(i=0;i<strlen(sourceString);i++)
16          destString2[i]=destString1[i]^maskChar;
17      destString2[i]='\0';
18      printf("sourceString=%s\n",sourceString);
19      printf("destString1=%s\n",destString1);
20      printf("destString2=%s\n",destString2);
21  }
```

程序的第 2 行定义了一个源字符串，第 5 行定义了一个屏蔽码。程序第 12～13 行的循环完成了源字符串的加密，得到密文字符串 destString1。程序第 15～16 行的循环将密文字符串还原回明文字符串 destString2。这里使用了异或操作，加密和解密都是使用同一个屏蔽码。这种技术非常简单但又非常有意义。因为，加密的过程是为了让别人无法看清楚，而解密则是自己的程序必须用到的原来的信息。

运行结果如下：

```
sourceString=Hello
destString1=↔099:
destString2=Hello
```

【例 13.5】　AD574 是一种 A/D 转换器，它的作用就是将模拟信号转换成数字信号。它对外有 2 个端口，分别叫做偶地址端口和奇地址端口。向偶地址端口发送命令可以启动 A/D 转换。从偶地址端口读取数据可以获取 A/D 转换是否完成的状态。在 A/D 完成后，从偶地址可以获取转换数据的高 8 位数据，从奇地址可以获取低 4 位数据。假设根据硬件连接情况，得到端口地址 0x500 和 0x501。在 C 语言的库函数中，有访问外设端口的 outportb 函数，它将 1 个字节的数据写入外设端口寄存器。inportb 函数把端口寄存器中 1 个字节读入 CPU。下面的程序给出了一个数据采集函数。

程序如下：

```
1   unsigned  ad_convert()
2   {
3       unsigned  ad_h8=0;
4       unsigned  ad_L4=0;
5       unsigned  ad_result=0;
6       outportb(0x500,0xff);
7       while(inportb(0x500)&&0x80);
8       ad_h8=(unsigned)inportb(0x500);
9       ad_L4=(unsigned)inportb(0x501);
10      ad_h8=ad_h8<<4;
11      ad_L4=ad_L4>>4;
12      ad_result=ad_h8+ad_L4;
13      return(ad_result);
14  }
```

这里是一个采集数据的函数，它的返回值是一个无符号的整数，包含采集到的 12 位数据，这个数据代表输入的模拟信号的大小。

程序的第 2～5 行定义了 3 个变量，分别对应采集出来的高 8 位数据，低 4 位数据和最终的合成结果。第 6 行向偶端口输出了一个数 0xff，相当于向 A/D 转换器发送了启动 A/D 转换的命令。程序的第 7 行是一个循环，循环的目的是等待 A/D 转换的完成。这里使用了逻辑与运算。实际上 A/D 转换完成以后，会在 A/D 转换器的状态端口有一个标志，这里的标志就是 8 位数据的最高位，如果最高位为 0，表示转换已经完成，否则表示转换没有完成，需要继续等待。这里将获取的数据与 0x80 相与就可以屏蔽其他标志，而只得到这个标志。这个循环就是等待转换完成的。程序的第 8、9 行分别从数据口获取 12 位数据的高 8 位和低 4 位，当然此时转换已经完成。程序的第 10～12 行则是对得到的结果进行组合，得到最后的结果。因为高 8 位的数据应放置在结果数据的第 9～12 位，所以代表的数值需要左移 4 位，而低 4 位数据本身在采集时处在 8 位数据的高 4 位，所以需要右移 4 位得到它代表的真实数据。

需要说明的是，对于数据采集及控制本身由于与硬件连接和使用规范有直接关系，具体情况还要具体对待。但是 C 语言的功能确实提供了比汇编语言更方便的操作手段。

13.3　位　　段

我们知道计算机中信息的存储一般是以字节为单位的。实际上，有时一个信息本身可能用不到那么多位。例如，要表示人的性别只需要 1 位二进制数据就可以了， 0 代表“男”，1 代表“女”。再如，在过程控制领域，1 个字节的数据可以表示多个命令或设备的多个状态特征。虽然可以用移位或其他位运算组合成我们需要的信息，但是，C 语言提供了更方便的利用位段结构处理信息的方法。

【例 13.6】　定义位段。

程序如下：

```
1   typedef struct
2   {
```

```
3       unsigned a:3;
4       unsigned b:1;
5       unsigned c:7;
6   }Some_bits;
7   #include<stdio.h>
8   void main()
9   {
10      Some_bits theseBits;
11      unsigned int*pInteger;
12      pInteger=(unsigned int*)&theseBits;
13      *pInteger=0;
14      theseBits.a=6;
15      theseBits.b=1;
16      theseBits.c=16;
17      printf("sizeof(Some_bits)=%d\n",sizeof(Some_bits));
18      printf("*pInteger=%x\n",*pInteger);
19      printf("%u,%u,%u\n",theseBits.a,theseBits.b,theseBits.c);
20  }
```

运行结果如下：

```
sizeof(Some_bits)=2
*pInteger=10e
6,1,16
```

本例程序定义了一个结构 Some_bits，包含了成员 a，这个成员 3bit 宽。还包含了成员 b 和 c，它们的宽度分别是 1bit 和 7bit。类型 Some_bits 的总宽度是 11bit。表面上定义位段可以节省空间，但是事实上往往不是这样的。对于多数计算机，最小的数据也是一个整数或字节。因此，这里的结构 Some_bits 也要占用 2 个字节，当然有些位是没有使用的。

这里，我们定义的 Some_bits 结构的格式如图 13.2 所示。

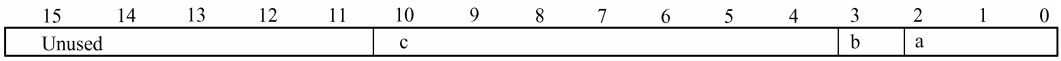


图 13.2　Some_bits 结构的格式

注意，在这一定义中，有定义的位段位于整个结构数据的低位。

读者可能已经注意到，在程序的第 18 行，输出的*pInteger 的值是 0x10e，它是如何得到的呢？把各个位段的数值用二进制表示出来，如图 13.3 所示，读者就会容易看出来了。

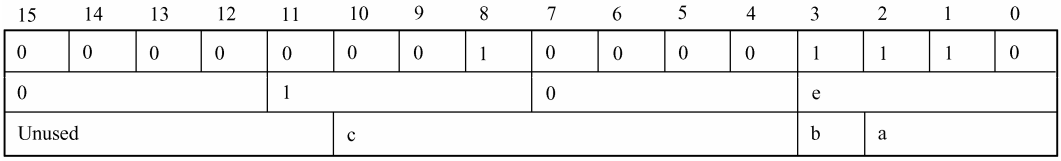


图 13.3　二进制表示

位段的访问方法和其他结构一样，这里就不再赘述了。

此外，还可以用指定宽度的未命名位段填充位段结构中的位，用零宽度的未命名位段可以强制使下一个位段与下一个整数所占内存的边界对齐。例 13.7 演示了未命名位段的用法。

【例 13.7】　未命名位段举例。

程序如下：

```
1   typedef struct
2   {
3       unsigned a:3;
4       unsigned :2;
5       unsigned b:1;
6       unsigned :0;
7       unsigned c:7;
8   }Unamed_bits;
9   #include<stdio.h>
10  void main()
11  {
12      Unamed_bits theseBits;
13      unsigned int*pInteger;
14      pInteger=(unsigned int*)&theseBits;
15      *pInteger=0;
16      theseBits.a=6;
17      theseBits.b=1;
18      theseBits.c=16;
19      printf("sizeof(Unamed_bits)=%d\n",sizeof(Unamed_bits));
20      printf("*pInteger=%x\n",*pInteger);
21      printf("%u,%u,%u\n",theseBits.a,theseBits.b,theseBits.c);
22  }
```

程序的运行结果如下：

```
sizeof(Unamed_bits)=2
*pInteger=1026
6,1,16
```

可见，程序的运行结果与例 13.6 是相同的。但这里的位段的结构是不同的，其定义格式如图 13.4 所示。

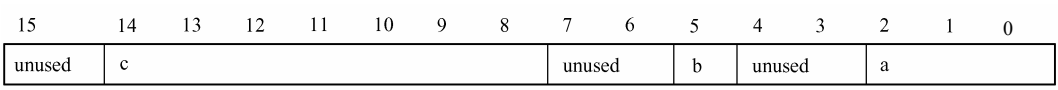


图 13.4　位段定义格式

在程序的第 20 行，输出的*pInteger 的值是 0x1026，请读者自己分析如何获得这个数据，这里不再赘述。

还有一点需要说明，由于计算机硬件的不同，位域的设置可能在不同的 C 语言环境下有

所差异，读者在使用时需要注意。本章的例子是在 Turbo C 2.0 环境下适用的。

习　题　13

13.1　编写一个函数，将 16 位无符号整数按从高到低将各位显示出来。

13.2　编写程序，将二进制文件按字节显示其内容，每行显示 16 个字节。

13.3　编写一个程序用来文件的加密与解密。并分三列把文件的原内容、加密后的内容和解密后的内容展示出来，每列显示 10 个字符。

13.4　编写一个函数用来实现循环移位。函数名为 move，调用方法为 move(value,*n*)，其中 value 为要移位的整数，*n* 为移位的位数。当 *n*<0 表示左移，*n*>0 为右移。

13.5　编写一个函数把一个包含二进制数字的字符串转换成十六进制数。例如输入“10001010”，返回值为 8A。并编写程序证明函数能工作。

13.6　把下面的 8×8 的二进制位图信息在屏幕显示在一个 8×8 的方框里。其中，对应位为 0 则显示空格，对应位为 1 则显示*。

```
00000000
00001000
00011100
00011110
00011100
00011100
00011100
00000000
```

13.7　给出 64 位结构 hardware_config 的定义，它包含如下内容。

（1）一个成员指示安装了哪些可拆卸的媒体驱动器。0x0 表示没有驱动器，0x1 表示有 1 个或多个软盘，0x2 表示有 1 个或多个 CD-ROM 驱动器，0x4 表示有 1 个或多个 Zip 驱动器。

（2）1 个未用字节的位域。

（3）一个成员指示安装的 RAM 容量，它可以是 0～1GB 之间的数。

（4）一个成员指示是否有游戏端口。0 表示没有，1 表示有。

（5）一个成员指示是否有网卡。0 表示没有，1 表示有。

（6）一个成员指示安装的串口的数量，范围是 0～15。

（7）一个成员指示安装的并口的数量，范围是 0～3。

（8）1 个未用字节的位域。

附录A　常用字符的ASCII编码

表 A.1　常用字符的 ASCII 编码

ASCII 码			字符	ASCII 码			字符	ASCII 码			字符	ASCII 码			字符
十进制	八进制	十六进制		十进制	八进制	十六进制		十进制	八进制	十六进制		十进制	八进制	十六进制	
0	0	0	(Null)	32	40	20	空格	64	100	40	@	96	140	60	`
1	1	1	☺	33	41	21	!	65	101	41	A	97	141	61	a
2	2	2	☻	34	42	22	"	66	102	42	B	98	142	62	b
3	3	3	♥	35	43	23	#	67	103	43	C	99	143	63	c
4	4	4	♦	36	44	24	$	68	104	44	D	100	144	64	d
5	5	5	♣	37	45	25	%	69	105	45	E	101	145	65	e
6	6	6	♠	38	46	26	&	70	106	46	F	102	146	66	f
7	7	7	(Beep)	39	47	27	'	71	107	47	G	103	147	67	g
8	10	8	◘	40	50	28	(	72	110	48	H	104	150	68	h
9	11	9	(Tab)	41	51	29	)	73	111	49	I	105	151	69	i
10	12	A	换行	42	52	2A	*	74	112	4A	J	106	152	6A	j
11	13	B	(VT)	43	53	2B	+	75	113	4B	K	107	153	6B	k
12	14	C	(FF)	44	54	2C	,	76	114	4C	L	108	154	6C	l
13	15	D	回车	45	55	2D	-	77	115	4D	M	109	155	6D	m
14	16	E	♫	46	56	2E	.	78	116	4E	N	110	156	6E	n
15	17	F	☼	47	57	2F	/	79	117	4F	O	111	157	6F	o
16	20	10	►	48	60	30	0	80	120	50	P	112	160	70	p
17	21	11	◄	49	61	31	1	81	121	51	Q	113	161	71	q
18	22	12	↕	50	62	32	2	82	122	52	R	114	162	72	r
19	23	13	‼	51	63	33	3	83	123	53	S	115	163	73	s
20	24	14	¶	52	64	34	4	84	124	54	T	116	164	74	t
21	25	15	§	53	65	35	5	85	125	55	U	117	165	75	u
22	26	16	▬	54	66	36	6	86	126	56	V	118	166	76	v
23	27	17	↨	55	67	37	7	87	127	57	W	119	167	77	w
24	30	18	↑	56	70	38	8	88	130	58	X	120	170	78	x
25	31	19	↓	57	71	39	9	89	131	59	Y	121	171	79	y
26	32	1A	→	58	72	3A	:	90	132	5A	Z	122	172	7A	z
27	33	1B	←	59	73	3B	;	91	133	5B	[	123	173	7B	{
28	34	1C	∟	60	74	3C	<	92	134	5C	\	124	174	7C	|
29	35	1D	↔	61	75	3D	=	93	135	5D	]	125	175	7D	}
30	36	1E	▲	62	76	3E	>	94	136	5E	^	126	176	7E	~
31	37	1F	▼	63	77	3F	?	95	137	5F	_	127	177	7F	·

附录B　计算机中数的表示

1．进位计数制

所谓进位计数制，是指用进位的方法进行计数的数制，简称进制。一般情况下，人们习惯于用十进制来表示数，有时也使用其他进制，如用六十进制计时，用七进制计算星期等。在计算机内，则采用二进制表示各种数据。在 C 语言中，还采用八进制和十六进制。这样就存在着同一个数可以用不同的数制表示及它们相互之间转换的问题。

下面简单介绍一下各种数制的转换规律。

（1）二进制、八进制、十六进制数转化为十进制数

对于任何一个二进制数、八进制数、十六进制数，可以写出它的按权展开式，再按十进制数运算规则进行计算即可得到对应的十进制数。

例如：

$(1\ 111.11)_2 = 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 + 1\times2^{-1} + 1\times2^{-2} = 15.75$

$(A10B.8)_{16} = 10\times16^3 + 1\times16^2 + 0\times16^1 + 11\times16^0 + 8\times16^{-1} = 41\ 227.5$

（2）十进制数转化为二进制数

要将十进制数转化为二进制数，十进制数的整数部分和小数部分在转换时需作不同的计算，分别求值后再组合。对于整数部分采用除 2 取余数法，即逐次除以 2，直至商为 0，得出的余数倒排，即为二进制各位的数码。小数部分采用乘 2 取整法，即逐次乘以 2，从每次乘积的整数部分得到二进制数各位的数码。

例如，将十进制数 123.125 转化为二进制数。

先对整数 123 进行转换：

2 | 123		
2 | 61	1	↑
2 | 30	1	
2 | 15	0	
2 | 7	1	
2 | 3	1	
2 | 1	1	
0	1	

因此，$123 = (1111011)_2$。

再对小数部分 0.125 进行转换：

0.125×2	$= 0.25$	0
0.25×2	$= 0.5$	0
0.5×2	$= 1$	1 ↓

即，$0.125 = (0.001)_2$。

最后，将整数和小数部分组合，得出 $123.125 = (1100100.001)_2$。

（3）二进制数、八进制数和十六进制数的相互转换

二进制数转换成八进制数的方法是：将二进制数的整数部分从小数点开始向左每 3 位分成一组，不足 3 位时在左边补 0；对二进制数的小数部分从小数点向右每 3 位分成一组，不

足 3 位时在右边补 0；将每一组的 3 位二进制数，分别转换成对应的八进制数即可。

例如，把二进制数 11101.10101 转化为八进制数。

方法为：11101.10101 = 011 101.101 010 =（35.52）$_8$。

反过来，将八进制数转换成二进制数，只要将每一位八进制数转换成相应的 3 位二进制数，依次连接起来，再去掉小数点左边最高位的 0 和小数点右边最低位的 0 即可。

二进制数与十六进制数的相互转换规律与此类似，读者可以自己试一试。

2．计算机中整数的表示

计算机中的整数采用二进制数形式表示，如果不考虑正负号，将十进制数整数直接转化为二进制数即可。如果考虑正负号，则需要转换为二进制补码形式。

（1）符号位的表示

在计算机中，数的正负号也用 0 和 1 表示。通常，把一个数的最高位作为符号位，0 表示正，1 表示负。对于 16 位整数，最高位是最左边的第 16 位，因为有了符号位，能够表示整数的值的位数就只有 15 位了。

（2）二进制数的原码、反码和补码

如果要处理的整数是有符号的，在计算机中就必须考虑符号位的处理，为此引入了原码、反码和补码的概念。在计算机中，数据的表示采用二进制的补码形式。

通常，把用正负号表示的数称做真值。求整数的补码的步骤如下。

① 求原码：将整数的真值转化为对应的二进制形式，左边补 0 以凑足整数的位数，如果为正数，符号位取 0，若为负数则取 1。

② 求反码：如果数的原码的符号位为 0，则不用改变；若为 1，则把除符号位外的各位求反，即 0 变 1，1 变 0。

③ 求补码：如果数的反码的符号位为 0，则不用改变；若为 1，则在最低位上加 1。

例如，如果整数位数为 16 位，求负数−123 的补码。

解：把−123 转化为二进制数：−123=（−1111011）$_2$，得到原码 1000 0000 0111 1011，符号位为 1。其反码为 1111 1111 1000 0100，补码是 1111 1111 1000 0101，十六进制表示为 FF85。

表 B.1 所示为几个典型的整数的补码（为了方便，这里假设整数是 8 位的）。

表 B.1　几个典型的整数的补码

十进制数（真值）	二进制数（真值）	原　码	反　码	补　码
+0	+0	0000 0000	0000 0000	0000 0000
−0	−0	1000 0000	1111 1111	0000 0000
+1	+1	0000 0001	0000 0001	0000 0001
−1	−1	1000 0001	1111 1110	1111 1111
+127	+111 1111	0111 1111	0111 1111	0111 1111
−127	−111 1111	1111 1111	1000 0000	1000 0001
−128				1000 0000

可见，采用补码表示后，没有了+0、−0 的问题，它们的补码相同。但出现了一个特殊的数的补码——“符号位为 1，其他位为 0”，对应的十进制数是−128（8 位整数）。

（3）补码的运算规律

采用补码表示的数的运算具有以下规律：

① 减法运算可以用加法来实现，即用求和代替求差；

② 数的符号位可以同数值部分作为一个整体参与运算；

③ 两数的补码之和（差）等于两数和（差）的补码。

同时，由于根据补码求真值的过程与求补码的过程相同，即把补码当成“原码”，再求其“补码”即可，大大简化了计算机硬件系统的设计，所以现代计算机内部大都用补码表示数值，运算结果也用补码表示。

3. 计算机中小数的表示

在计算机中表示小数，根据其小数点位置是否固定，分为定点数表示法和浮点数表示法。

定点数：计算机中小数点的位置是固定的，运算时不再考虑小数点的位置。

浮点数：计算机中小数点的位置是浮动的。通常，可将小数表示为 $m \times 2^n$ 的形式，其中 m、n 都是二进制的，n 为整数，m 则采用小数点左边只有一个 1 的标准化形式。

小数的两种表示方法不仅关系到小数点的问题，而且关系到数的表示范围、精度及计算机内部电路的复杂程度。与定点数相比较，浮点数的表示范围要大得多，处理难度也大得多。

附录C　C语言的运算符

表 C.1　C 语言的运算符

优　先　级	运　算　符	含　　义	运算数个数	结 合 方 向
1	( )	括号		
2	[ ]	元素下标运算符	2	自左向右
	–>	指向结构体成员运算符		
	.	结构体成员运算符		
3	!	逻辑非	1 （单目运算符）	自右向左
	~	按位取反运算符		
	++	自增运算符		
	––	自减运算符		
	+ –	正、负号运算符		
	(类型说明符)	强制类型转换运算符		
	*	指针运算符		
	&	取地址运算符		
	sizeof()	长度运算符		
4	*	乘法运算符	2 （双目运算符）	自左向右
	/	除法运算副		
	%	模运算符（求余数）		
5	+	加法运算符		
	–	减法运算符		
6	<<	左移位运算符		
	>>	右移位运算符		
7	< <= => >	关系运算符		
8	==	等于运算符		
	!=	不等于运算符		
9	&	按位与运算符		
10	^	按位异或运算符		
11	|	按位或运算符		
12	&&	逻辑与运算符		
13	||	逻辑或运算符		
14	? :	条件运算符	3 （三目运算符）	自右向左
15	=	赋值运算符	2 （双目运算符）	自右向左
	+= –= *= /= %=			
	>>= <<= &= ^= |=			
16	,	逗号运算符		自左向右

说明：

① 括号“( )”是一个特殊的运算符，用于提升括号内的运算的优先级。也就是说，当表达式中有括号时，按照先括号内后括号外的顺序执行。如果括号是嵌套的，则从最内层括号开始。

② 单目运算符都是自右向左结合的，通常位于操作数的左边，只有自增（++）和自减（--）运算符既可以位于操作数的左边也可以位于右边。自增、自减运算符的位置不同，运算规律也不同。当运算符在左边时，表达式的值是执行了自增、自减运算后的变量的值，而在右边时，表达式的值是自增、自减前的值。如果表达式中还有其他运算符，应考虑运算符的优先级。注意，表达式“a+++b”等价于“(a++)+b”，不同于“++a+b”或“a+(++b)”。

③ 算术运算符和位运算符（按位取非“~”是单目运算符，除外）可以与赋值运算符“=”组合在一起，形成复合赋值运算符，它们的优先级与赋值运算符“=”相同，结合方向都是从右到左。例如，“x/=y>>=2”的运算过程是先计算“y=y>>2”，然后计算“x=x/y”。

④ sizeof()是个特殊的“运算符”，其操作数可以是变量、常量、基本类型说明符、用户定制的类型说明符等。

参 考 文 献

[1] 谭浩强. C 程序设计（第三版）. 北京：清华大学出版社，2005

[2] Steve McConnell. 代码大全（第 2 版・英文版）. 北京：电子工业出版社，2006

[3] David Conger. 软件开发：编程与设计（C 语言版）. 朱建平. 北京：清华大学出版社，2006

[4] 裘宗燕. 从问题到程序. 北京：机械工业出版社，2006

[5] 苏晓红，陈惠鹏，孙志刚. C 语言大学实用教程. 北京：电子工业出版社，2005

[6] Brian W. Kernighan，Dennis M. Ritchie. C 语言程序设计（第 2 版・新版）. 徐宝文，李志. 北京：机械工业出版社，2004

[7] Brian W. Kernighan. 程序设计实践. 裘宗燕. 北京：机械工业出版社，2003

[8] Carroll Morgan. 从规范出发的程序设计. 裘宗燕. 北京：机械工业出版社，2002

反侵权盗版声明