新世纪计算机基础教育丛书 | 丛书主编 谭浩强

C程序设计（第三版）

谭浩强 著

发行800万册记录

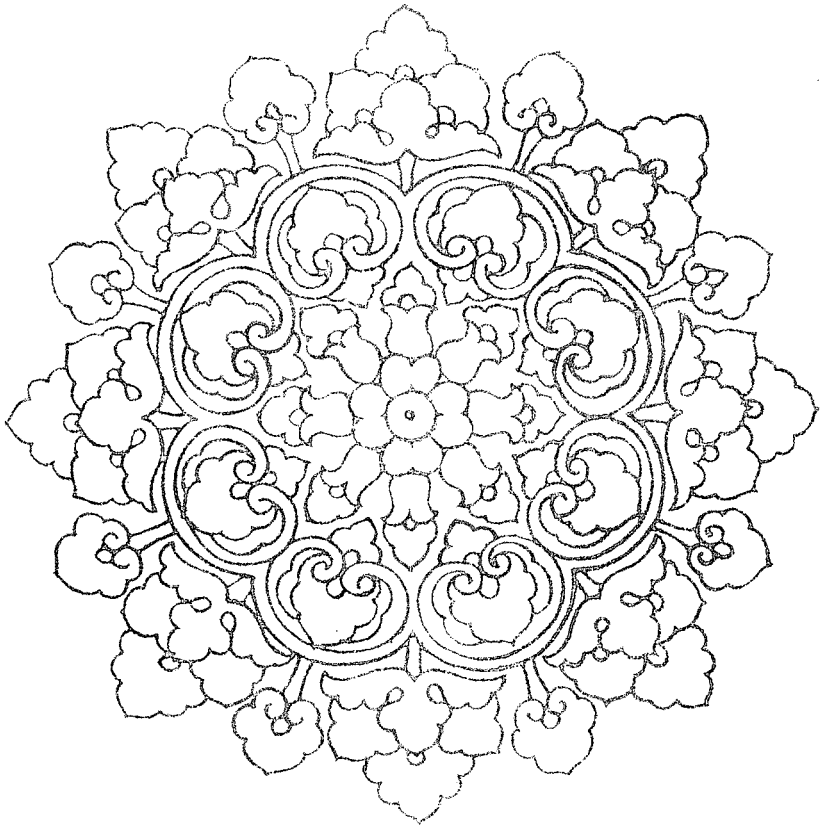
清华大学出版社

新世纪计算机基础教育丛书 | 丛书主编 谭浩强

C程序设计（第三版）

谭浩强 著

发行800万册记录



清华大学出版社
北京
www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

内 容 简 介

C 语言是国内外广泛使用的计算机语言，学会使用 C 语言进行程序设计是计算机工作者的一项基本功。

本书的第一版于 1991 年出版，第二版于 1999 年出版。由于本书作者具有丰富的教学经验和编写教材的经验，并针对初学者的特点，精心策划、准确定位，使得本书概念清晰、例题丰富、深入浅出，受到专家和读者的一致好评。本书被普遍认为是学习 C 语言的好教材，并被全国大多数高校选用。十多年来本书累计发行了 800 多万册，创同类书的全国最高记录，是学习 C 语言的主流用书。本书曾荣获原电子工业部优秀教材一等奖、高校出版社优秀畅销书特等奖、全国高等院校计算机基础教育研究会优秀教材一等奖。

根据发展的需要，作者对本书进行了再修订，使本书更加完善，更便于学习。书中全部例题中的程序均已调试通过。

本书内容新颖、体系合理、逻辑性强、文字流畅、通俗易懂，是学习 C 语言的理想教材。凡具有计算机初级知识的读者都能读懂本书。本书可作为高等学校各专业的正式教材，也是一本自学的好教材。另外本书还配有辅助教材《C 程序设计题解与上机指导(第三版)》。

图书在版编目(CIP)数据

C 程序设计/谭浩强著. —3 版. —北京：清华大学出版社，2005(2007 重印)
(新世纪计算机基础教育丛书/谭浩强主编)
ISBN 978-7-302-10853-5

Ⅰ. C…　Ⅱ. 谭…　Ⅲ. C 语言—程序设计　Ⅳ. TP312

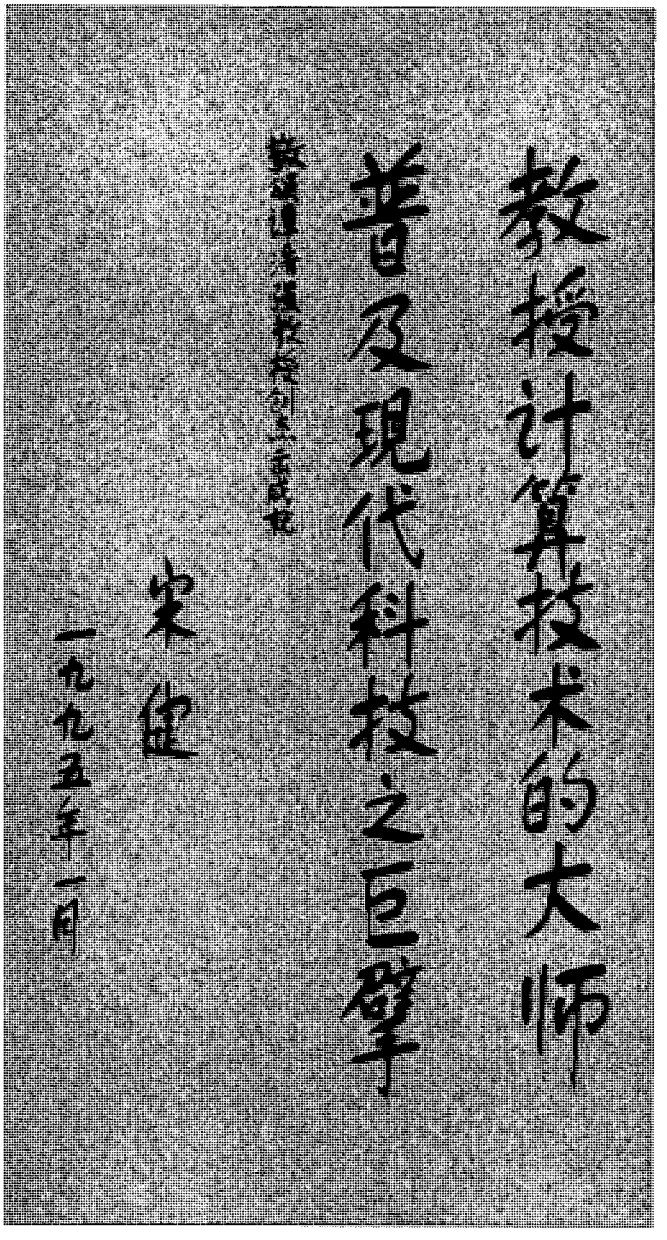
中国版本图书馆 CIP 数据核字(2007)第 014674 号

责任编辑：范素珍
封面设计：傅瑞学
责任印制：李红英

出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编**：100084
c-service@tup.tsinghua.edu.cn
社 总 机：010-62770175　　**邮购热线**：010-62786544
投稿咨询：010-62772015　　**客户服务**：010-62776969

印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：185×260　　**印张**：25.25　　**插页**：1　　**字数**：592 千字
版　　次：2005 年 7 月第 3 版　　**印次**：2007 年 2 月第 13 次印刷
印　　数：910001～940000
定　　价：26.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：010-62770177 转 3103　　产品编号：018602-01/TP



▲ 原全国政协副主席、国务委员、国家科委主任、
中国工程院院长宋健同志给谭浩强教授的题词

▲ 原全国人民代表大会副委员长、中国科学院院长卢嘉锡给谭浩强教授的题词

丛书序言

Preface Preface Preface Preface

现代科学技术的飞速发展，改变了世界，也改变了人类的生活。作为新世纪的大学生，应当站在时代发展的前列，掌握现代科学技术知识，调整自己的知识结构和能力结构，以适应社会发展的要求。新世纪需要具有丰富的现代科学知识，能够独立完成面临的任务，充满活力，有创新意识的新型人才。

掌握计算机知识和应用，无疑是培养新型人才的一个重要环节。计算机技术已深入到人类生活的各个角落，与其他学科紧密结合，成为推动各学科飞速发展的有力的催化剂。无论学什么专业的学生，都必须具备计算机的基础知识和应用能力。计算机既是现代科学技术的结晶，又是大众化的工具。学习计算机知识，不仅能够掌握有关的知识，而且能培养人们的信息素养。它是高等学校全面素质教育中极为重要的一部分。

高校计算机基础教育应当遵循的理念是：面向应用需要，采用多种模式，启发自主学习，重视实践训练，加强创新意识，树立团队精神，培养信息素养。

计算机应用人才的队伍由两部分人组成：一部分是计算机专业出身的计算机专业人才，他们是计算机应用人才队伍中的骨干力量；另一部分是各行各业中应用计算机的人员。这后一部分人一般并非计算机专业毕业。他们人数众多，既熟悉自己所从事的专业，又掌握计算机的应用知识，善于用计算机作为工具解决本领域中的问题。他们是计算机应用人才队伍中的基本力量。事实上，大部分应用软件都是由非计算机专业出身的计算机应用人员研制的。他们具有的这个优势是其他人难以代替的。从这个事实可以看到在非计算机专业中深入进行计算机教育的必要性。

非计算机专业中的计算机教育，无论目的、内容、教学体系、教材、教学方法等各方面都与计算机专业有很大的不同，绝不能照搬计算机专业的教学模式和做法。全国高等院校计算机基础教育研究会自 1984 年成立以来，始终不渝地探索高校计算机基础教育的特点和规律。2004 年，全国高等院校计算机基础教育研究会与清华大学出版社共同推出了《中国高等院校计算机基础教育课程体系 2004》(简称 CFC2004)；2006 年，又共同推出了《中国高等院校计算机基础教育课程体系 2006》(简称 CFC2006)，由清华大学出版社正式出版发行。

1988 年起，我们根据教学实际的需要，组织编写了《计算机基础教育丛书》，邀请有丰富教学经验的专家、学者先后编写了多种教材，由清华大学出版社出版。丛书出版后，迅速受到广大高校师生的欢迎，对高等学校

的计算机基础教育起到了积极的推动作用。广大读者反映这套教材定位准确，内容丰富，通俗易懂，符合大学生的特点。

1999 年，根据新世纪的需要，我们又在原有基础上组织出版了《新世纪计算机基础教育丛书》。由于内容符合需要，质量较高，该套丛书已被许多高校选为教材。丛书总发行量达 1000 多万册，这在国内是罕见的。

最近，我们又对丛书做了进一步的修订，根据发展的需要，增加了新的书目和内容。本丛书有以下特点：

(1) 内容新颖。根据 21 世纪的需要，重新确定了丛书的内容，以符合计算机科学技术的发展和教学改革的要求。本丛书除保留了原丛书中经过实践考验且深受群众欢迎的优秀教材外，还编写了许多新的教材。在这些教材中反映了近年来迅速得到推广应用的一些计算机新技术，以后还将根据发展不断补充新的内容。

(2) 适合不同学校组织教学的需要。本丛书采用模块形式，提供了各种课程的教材，内容覆盖高校计算机基础教育的各个方面。丛书中既理工类专业的教材，也有文科和经济类专业的教材；既有必修课的教材，也包括一些选修课的教材。各类学校都可以从中选择到合适的教材。

(3) 符合初学者的特点。本丛书针对初学者的特点，以应用为目的，以应用为出发点，强调实用性。本丛书的作者都是长期在第一线从事高校计算机基础教育的教师，对学生的基础、特点和认识规律有深入的研究，在教学实践中积累了丰富的经验。可以说，每一本教材都是他们长期教学经验的总结。在教材的写法上，既注意概念的严谨和清晰，又特别注意采用读者容易理解的方法阐明看似深奥难懂的问题，力求做到例题丰富，通俗易懂，便于自学。这一点是本丛书一个十分重要的特点。

(4) 采用多样化的形式。除了教材这一基本形式外，有些教材还配有习题解答和上机指导，并提供电子教案。

总之，本丛书的指导思想是内容新颖、概念清晰、实用性强、通俗易懂、教材配套。简单概括为："新颖、清晰、实用、通俗、配套"。我们经过多年实践形成的这一套行之有效的创作风格，相信会受到广大读者的欢迎。

本丛书多年来得到了各方面人士的指导、支持和帮助，尤其是得到了全国高等院校计算机基础教育研究会各位专家和各高校老师们的支持和帮助，我们在此表示由衷的感谢。

本丛书肯定有不足之处，竭诚希望得到广大读者的批评指正。

欢迎访问谭浩强网站：http：//www. tanhaoqiang. com

丛 书 主 编
全国高等院校计算机基础教育研究会会长
谭 浩 强

前言

Foreword Foreword Foreword Foreword

C语言是在国内外广泛使用的一种计算机语言。C语言功能丰富、表达能力强、使用灵活方便、应用面广、目标程序效率高、可移植性好，既具有高级语言的优点，又具有低级语言的许多特点，因此特别适合于编写系统软件。C语言诞生后，许多原来用汇编语言编写的软件，现在可以用C语言编写了(例如，著名的UNIX操作系统就是用C语言编写的)，而学习和使用C语言要比学习和使用汇编语言容易得多。

近年来，C语言不仅为计算机专业工作者所使用，而且为广大计算机应用人员(包括大量原来是非计算机专业的使用计算机的人们)所喜爱和使用。许多高等学校，不仅在计算机专业开设了C语言课程，而且在非计算机专业也开设了C语言课程。全国计算机等级考试、全国计算机应用技术证书考试(NIT)和全国各地区组织的大学生计算机统一考试都将C语言列入了考试范围。许多人用它编写应用软件。学习C语言已经成为广大计算机应用人员和广大青年学生的迫切要求。

由于C语言牵涉的概念比较复杂，规则繁多，使用灵活，容易出错，不少初学者感到困难，迫切希望有一本容易入门、容易学习的C语言教材。在许多同志的鼓励和督促下，作者于1991年编写了《C程序设计》，由清华大学出版社出版，以期抛砖引玉。该书针对初学者的特点和认知规律，精选内容，分散难点，降低台阶，例题丰富，通过深入浅出地叙述，阐明了复杂的概念，力求做到内容新颖、概念清晰、实用性强、通俗易懂。该书出版后受到广大读者的热烈欢迎，许多读者说“C语言原来是比较难学的，但自从《C程序设计》出版后，C语言变得不难学了”，“作者深入浅出地叙述，使我们对C语言由害怕到兴趣盎然”。1999年，作者对本书进行了修订，使之进一步完善，出版了本书的第二版。十多年来，该书累计发行了700多万册，平均每年印刷50万册，居全国同类书的首位。全国大多数高校把本书作为正式教材。许多高校的研究生入学考试都指定本书为必读教材，国内许多介绍C语言的书籍以本书为蓝本，许多在职干部和计算机爱好者通过自学本书掌握了C语言程序设计。在推广普及计算机程序设计过程中，本书成为广大初学者学习C语言程序设计的主流用书。本书

曾荣获原电子工业部优秀教材一等奖、全国高等院校计算机基础教育研究会优秀教材一等奖、高校出版社优秀畅销书特等奖。这是对我的莫大鼓励和鞭策。我深切地感受到广大读者对作者的殷切期望。

根据计算机科学技术的发展和教学实践的需要，作者对《C 程序设计》一书再次进行了修订，出版第三版。第三版保持了第一版的写作风格，保留了通俗易懂的特点，并在以下几方面做了修改。

(1) 在本书的前两版中，采用 Turbo C 2.0 对程序进行编译。由于近年来，大多数人习惯使用 Windows 操作系统的图形界面，用鼠标进行操作，感到用 Turbo C 2.0 不大方便。在第三版中改用 Turbo C++ 3.0 作为编译工具。Turbo C++ 3.0 虽然是基于 DOS 界面的，但它支持鼠标操作，可以在 Windows 环境下方便地使用。它的使用方法与 Turbo C 类似。用过 Turbo C 的人很容易掌握 Turbo C++ 3.0。Turbo C++ 本来是用来编译 C++ 程序的，由于 C++ 是从 C 语言发展而来的，C++ 对 C 语言是兼容的，用 C 语言编写的程序可以用 C++ 编译系统进行编译。在与本书配套的《C 程序设计题解与上机指导》(第三版)中还介绍了 Visual C++ 6.0 对 C 程序的编译方法。读者也可以使用 Visual C++ 6.0。这样也有利于读者今后向 C++ 过渡。

本书的例题程序是用 Turbo C++ 3.0 或 Visual C++ 6.0 进行编译的。用 C++ 编译系统时，对程序要求更加规范。例如，在定义和声明函数时，必须指定函数类型；程序中如果用到系统提供的库函数(包括 printf 和 scanf 函数)，都必须在程序文件的开头用 #include 命令将有关头文件包含进来。因此，本书的程序基本上采用下面的形式：

```
#include <stdio.h>          /* 如果程序中用到系统提供的输入输出函数 */
void main()                 /* 要求指定 main 函数的类型 */
{
⋮
}
```

(2) 对各章内容和表述进行了细致的修改，调整了部分内容和例题，使读者更容易理解。

(3) 本书第二版的 14 章和 15 章介绍 C++ 的初步知识。由于作者已出版了《C++ 程序设计》(清华大学出版社出版)，对 C++ 做了全面、详细的介绍，读者如需了解和学习 C++，可以参考该书，同时为了压缩篇幅，在第三版中不再保留这两章的内容。

相信修订后的第三版会更加符合读者的需要。

关于怎样学习程序设计，作者提出以下几点看法。

(1) 近年来,有一些面向对象的计算机语言陆续问世,受到欢迎。有些人认为面向过程的C语言已经过时了,不必学了。这是一种误解。不应把面向对象和面向过程对立起来,在面向对象程序设计中仍然要用到面向过程的知识。作为计算机程序开发人员,既要掌握面向对象程序设计的知识,又要掌握面向过程程序设计的知识。面向过程程序设计仍然是计算机工作者的基本功。

C++是为开发大型程序而研制的,它比C语言要复杂得多,学习C++也比学习C语言困难得多。事实上,将来并不是每个人都需要用C++编制大型程序。对于计算机专业人员,学习和使用C++是理所当然的。对于非计算机专业人员,可以先学习C语言,打下程序设计的基础,在以后需要时再学习和使用C++。有了C语言基础,再学习C++就容易多了。也可以在学习C语言的基础上,再学习一些面向对象程序设计的初步知识,为以后进一步学习和使用C++打下基础。

目前,国内外的高等学校都把C语言程序设计作为一门重要的课程,我国各种计算机统一考试都包括C语言程序设计的科目。

现在大多数高校把C语言作为第一门计算机语言进行教学,这是可行的,学生是能够学习好的。

(2) 在学校中,学习程序设计课程的目的是掌握设计程序的思路,学会用计算机语言编写程序,以实现所需处理的任务。要正确处理算法与语法的关系,算法是程序的核心、是灵魂,语法是外壳、是工具。不应把学习重点放在语法规则上,语法是重要的,不掌握语法规则就无法编写出正确的程序,但是只学会语法,甚至能把语法背得滚瓜烂熟,也不可能编写出好的程序。一定要把重点放在解题的思路上,通过大量的例题学习怎样设计一个算法,构造一个程序。在学习开始时更不要在语法细节上死背死抠。请记住:重要的是学会编程序,而不是背语法。一开始就要学会看懂程序,编写简单的程序,然后逐步深入。有一些语法细节是需要通过较长期的实践才能熟练地掌握的。初学时,切忌过早地滥用C语言的某些容易引起错误的细节(如不适当地使用++和--的副作用)。

(3) 不能设想今后一辈子只使用在学校里学过的某一种语言。但是,无论用哪一种语言进行程序设计,其基本规律是一样的。在学习时一定要学活用活,举一反三,掌握规律,在以后需要时能很快地掌握其他新的语言进行工作。

(4) 在学校学习阶段,主要是学习程序设计的方法,进行程序设计的基本训练,打下将来进一步学习的基础。对多数学生来说,不可能通过几十小时的学习,由一个门外汉变成编程高手,编写出大型而实用的程序。学习程序设计课程时,应该把精力放在最基本、最常用的内容上,学好基

本功。如果对学生有较高的程序设计要求，应当在学习本课程后，安排一次集中的课程设计环节，按照实际工作的要求，完成有一定规模的程序设计。

(5) 程序设计是一门实践性很强的课程，既要掌握概念，又要动手编程，还要上机调试运行，希望读者一定要重视实践环节，包括编程和上机。既会编写程序，又会调试程序。衡量这门课学习的好坏，不是看你“知不知道”，而是“会不会干”。考核的方法不能主要用是非题和选择题，而应当把重点放在编制程序和调试程序上。

(6) 使用哪一种编译系统并不是原则问题，重要的是编程能力的培养。程序编好以后，用哪一种编译系统进行编译都可以。读者不应该只会用某一种编译环境，应当了解、接触和使用不同的编译环境。不同的编译系统，其功能和使用方法有些不同，编译时给出的信息也不完全相同，要注意参阅使用说明书，特别要在使用中积累经验，举一反三。

为了帮助读者学习本书，作者还编了一本《C 程序设计题解与上机指导》(第三版)，提供本书中各章习题的参考答案，以及上机实习指导。该书由清华大学出版社于 2005 年出版。

作者从事计算机教育和计算机普及工作二十多年，最深刻的体会是：作者心中要永远装着读者，要处处为读者考虑，要和读者将心比心。我的心中经常浮现出千万读者殷切期望的目光。读者热切地期望作者能为他们写出一批好书，使他们的学习能事半功倍。作者多年来以此来鞭策自己，希望能摸索出一些能减少初学者困难的方法，并做了一些探索和尝试。要写好一本书，是不容易的。要深入了解自己工作的对象，有的放矢，准确定位；要根据应用的需要，合理取舍，精选内容；要认真研究学习者的认识规律，采用读者容易理解的方法，深入浅出，通俗易懂；要善于把复杂问题简单化，而不能把简单问题复杂化。写书不仅是简单地把有关的技术内容告诉读者，而且要考虑怎样写才能使读者容易理解。要下很大的功夫，有时为了找到一个好的例子或一个通俗的比喻，苦苦思索好几天，每一句话都要反复斟酌推敲，总是努力把每一本书都做成精品。作为教师和作者，可以不计较自己的作品是否获奖，但是应当努力使自己的作品得到千万读者的认可和赞誉，成为事实上的精品，这才是最高的奖赏。

最后，对多年来关心支持本书和本书作者的领导和朋友们表示由衷地感谢。尤其是原全国政协副主席、国务委员、科委主任、中国工程院院长宋健院士，中国计算机学会名誉理事长、中国科学院资深院士张效祥先生等前辈给予作者有力地支持和指导。全国高等院校计算机基础教育研究会多年来始终全力支持和帮助作者在计算机教育和计算机普及领域所从事的工作。全国高校广大教师多年来和我共同奋斗，千万读者每时每

刻都给予我巨大的、宝贵的关心和支持。清华大学出版社十几年始终密切合作与支持。没有这一切，我不可能取得今天的成就。我永远感谢曾经帮助和支持过我的、相识的和不相识的同志和朋友。

谭亦峰工程师参加了本书部分章节的编写和程序调试工作。由于作者水平有限，本书肯定会有不少缺点和不足，热切期望得到专家和读者的批评指正。

谭浩强

2005 年春节于清华园

本书曾荣获：

原电子工业部优秀教材一等奖
全国高等院校计算机基础教育研究会优秀教材一等奖
高校出版社优秀畅销书特等奖

目 录

Catalog Catalog Catalog Catalog

5 选择结构程序设计

6 循环控制

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

第1章 C语言概述

1.1 C语言出现的历史背景

C语言是国际上广泛流行的计算机高级语言。它适合作为系统描述语言，既可以用来编写系统软件，也可用来编写应用软件。

早期的操作系统软件主要是用汇编语言编写的(包括UNIX操作系统在内)。由于汇编语言依赖于计算机硬件，程序的可读性和可移植性都比较差，所以为了提高系统软件的可读性和可移植性，最好改用高级语言。但是，一般的高级语言难以实现汇编语言的某些功能(汇编语言可以直接对硬件进行操作，例如对内存地址的操作、位操作等)。人们希望找到一种兼具一般高级语言和低级语言优点的语言，于是，C语言就在这种情况下应运而生了。

C语言是在B语言的基础上发展起来的，它的根源可以追溯到ALGOL 60。1960年出现的ALGOL 60是一种面向问题的高级语言，它离硬件比较远，不宜用来编写系统程序。1963年英国剑桥大学推出了CPL (combined programming language)语言。CPL语言在ALGOL 60的基础上接近硬件一些，但规模比较大，难以实现。1967年英国剑桥大学的Martin Richards对CPL语言做了简化，推出了BCPL (basic combined programming language)语言。1970年美国贝尔实验室的Ken Thompson以BCPL语言为基础，又做了进一步简化，设计出了很简单的而且很接近硬件的B语言(取BCPL的第一个字母)，并用B语言编写了第一个UNIX操作系统，在PDP 7上实现。1971年在PDP 11/20上实现了B语言，并编写了UNIX操作系统，此时的B语言过于简单，功能有限。1972年至1973年间，贝尔实验室的D. M. Ritchie在B语言的基础上设计出了C语言(取BCPL的第二个字母)。C语言既保持了BCPL和B语言的优点(精练，接近硬件)，又克服了它们的缺点(过于简单，数据无类型等)。最初的C语言只是为描述和实现UNIX操作系统提供一种工作语言而设计的。1973年，Ken Thompson和D. M. Ritchie合作把UNIX的90%以上用C语言改写(即UNIX第5版。原来的UNIX操作系统是1969年由美国的贝尔实验室的Ken Thompson和D. M. Ritchie开发成功的，是用汇编语言编写的)。

后来，C语言多次做了改进，但主要还是在贝尔实验室内部使用。直到1975年UNIX第6版发布后，C语言的突出优点才引起人们的普遍注意。1977年出现了不依赖于具体机器的C语言编译文本《可移植C语言编译程序》，使C语言移植到其他机器时所需做的工作大大简化了，这也推动了UNIX操作系统迅速地在各种机器上实现。例如VAX、AT&T等计算机系统都相继开发了UNIX。随着UNIX的日益广泛使用，C语言也迅速得到推广。C语言和UNIX可以说是一对孪生兄弟，在发展过程中相辅相成。

1978 年以后，C 语言先后移植到大、中、小、微型计算机上，已独立于 UNIX 和 PDP 了。C 语言便很快风靡全世界，成为世界上应用最广泛的几种计算机语言之一。

以 1978 年发布的 UNIX 第 7 版中的 C 语言编译程序为基础，Brian W. Kernighan 和 Dennis M. Ritchie（合称 K & R）合著了影响深远的名著《The C Programming Language》，这本书中介绍的 C 语言成为后来广泛使用的 C 语言版本的基础，它被称为标准的 C。1983 年，美国国家标准化协会（ANSI）根据 C 语言问世以来各种版本对 C 语言的发展和扩充，制定了新的标准，称为 ANSI C。ANSI C 比原来的标准 C 有了很大的发展。K & R 在 1988 年修改了他们的经典著作《The C Programming Language》，按照 ANSI C 标准重新写了该书。1987 年，ANSI 又公布了新标准——87 ANSI C。1990 年，国际标准化组织 ISO（International Standard Organization）接受 87 ANSI C 为 ISO C 的标准（ISO9899—1990）。1994 年，ISO 修订了 C 语言标准。目前流行的 C 语言编译系统大多是以 ANSI C 为基础进行开发的，但不同版本的 C 编译系统所实现的语言功能和语法规则又略有差别，因此读者应了解所用的 C 语言编译系统的特点（可以参阅有关手册）。本书的叙述基本上以 ANSI C 为基础。

1.2 C 语言的特点

一种语言之所以能存在和发展，并具有较强的生命力，总是有其不同于（或优于）其他语言的特点。C 语言的主要特点如下。

（1）语言简洁、紧凑，使用方便、灵活。C 语言一共有 32 个关键字（见附录 B）、9 种控制语句，程序书写形式自由，主要用小写字母表示，压缩了一切不必要的成分。C 语言程序比其他许多高级语言简练，源程序短，因此输入程序时工作量少。

（2）运算符丰富。C 语言的运算符包含的范围很广泛，共有 34 种运算符（见附录 C）。C 语言把括号、赋值、强制类型转换等都作为运算符处理，从而使 C 语言的运算类型极其丰富，表达式类型多样化。灵活使用各种运算符可以实现在其他高级语言中难以实现的运算。

（3）数据类型丰富，具有现代语言的各种数据结构。C 语言提供的数据类型有：整型、浮点型、字符型、数组类型、指针类型、结构体类型、共用体类型等，能用来实现各种复杂的数据结构（如链表、树、栈等）的运算。尤其是指针类型数据，使用十分灵活和多样化。

（4）具有结构化的控制语句（如 if…else 语句、while 语句、do…while 语句、switch 语句、for 语句）。用函数作为程序的模块单位，便于实现程序的模块化。C 语言是完全模块化和结构化的语言。

（5）语法限制不太严格，程序设计自由度大。例如，对数组下标越界不做检查，由程序编写者自己保证程序的正确。对变量的类型使用比较灵活，例如，整型量与字符型数据以及逻辑型数据可以通用。一般的高级语言语法检查比较严，能检查出几乎所有的语法错误，而 C 语言允许程序编写者有较大的自由度，因此放宽了语法检查。程序员应当仔细检查程序，保证其正确，而不要过分依赖 C 语言编译程序去查错。“限制”与“灵活”是一对矛盾。限制严格，就失去灵活性；而强调灵活，就必然放松限制。一个不熟练的人员，

编一个正确的C语言程序可能会比编一个其他高级语言程序难一些。也就是说，对用C语言的人，要求对程序设计更熟练一些。

(6) C语言允许直接访问物理地址，能进行位(bit)操作，能实现汇编语言的大部分功能，可以直接对硬件进行操作。因此C语言既具有高级语言的功能，又具有低级语言的许多功能，可用来编写系统软件。C语言的这种双重性，使它既是成功的系统描述语言，又是通用的程序设计语言。有人把C语言称为“高级语言中的低级语言”或“中级语言”，意为兼有高级和低级语言的特点，但一般仍习惯将C语言称为高级语言。因为C语言程序也要通过编译、连接才能得到可执行的目标程序，这是和其他高级语言相同的。

(7) 生成目标代码质量高，程序执行效率高。C语言一般只比汇编程序生成的目标代码效率低10%～20%。

(8) 用C语言编写的程序可移植性好(与汇编语言比)。基本上不做修改就能用于各种型号的计算机和各种操作系统。

上面只介绍了C语言的最容易理解的一般特点，至于C语言内部的其他特点将结合以后各章的内容作介绍。由于C语言的这些优点，使C语言应用面很广。许多大的软件都用C语言编写，这主要是由于C语言的可移植性好和硬件控制能力高，表达和运算能力强。许多以前只能用汇编语言处理的问题，现在可以改用C语言来处理了。

C语言的以上特点，读者现在也许还不能深刻理解，待学完C语言以后再回顾一下，就会有比较深的体会。

下面从应用的角度，对C语言和其他高级语言作一简单比较。

从掌握语言的难易程度来看，C语言比其他语言难一些。BASIC是初学者较好的入门语言，FORTRAN也比较好掌握。对科学计算多用FORTRAN语言；对商业和管理等数据处理领域，用COBOL为宜，C语言虽然也可用于科学计算和管理领域，但是并不理想，C语言的特长不在这里。对操作系统和系统实用程序以及需要对硬件进行操作的场合，用C语言明显地优越于其他高级语言，有的大型应用软件也用C语言编写。从教学角度看，由于PASCAL语言是世界上第一个结构化语言，曾被认为是计算机专业的比较理想的教学语言，但PASCAL语言难以推广到各实际应用领域。C语言也是很好的结构化语言，且描述能力强，同样适于教学，如“操作系统”课程多结合UNIX讲解，而UNIX与C语言不可分。因此，大多数高校已用C语言取代了PASCAL语言。在“数据结构”课程中也已广泛采用C语言作为背景语言。C语言除了用于教学外，还有广泛的应用领域，因此更有生命力。自20世纪90年代初以来，我国学习和使用C语言的人越来越多，C语言成了学习和使用人数最多的一种计算机语言，熟练掌握C语言成为计算机开发人员的一项基本功。

由于开发大型软件的需要，近年来，面向对象的C++语言在我国逐步得到推广，但是不应得出这样的结论：“C语言已经过时了，人们都应当去学习C++，而不必学习C语言了”。不应把面向对象的程序设计和面向过程的程序设计对立起来，面向对象的基础是面向过程。C++是为解决编写大型软件的问题而产生的，学起来比C语言困难得多，将来并不是所有的人都去编写大型软件。因此，在发达国家的大学中，C语言仍然是一门重要的课程，是大学生的一种基本的选择。掌握了C语言，日后再进一步学习C++是不会太困难的。

1.3 简单的C语言程序介绍

下面先介绍几个简单的C语言程序，然后从中分析C语言程序的特点。

例 1.1 输出一行信息。

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

本程序的作用是输出以下一行信息：

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

先看第2行，其中 main 是函数的名字，表示“主函数”，main 前面的 void 表示此函数是“空类型”，void 是“空”的意思，即执行此函数后不产生一个函数值（有的函数在执行后会得到一个函数值，例如正弦函数 sin(x)）。每一个C语言程序都必须有一个 main 函数。函数体由花括号{ }括起来。本例中主函数内只有一个输出语句，printf 是C编译系统提供的标准函数库中的输出函数（详见第4章）。程序第4行 printf 语句中双撇号内的字符串按原样输出。“\n”是换行符，即在输出“This is a C program.”后回车换行。语句最后有一个分号。

在使用标准函数库中的输入输出函数时，编译系统要求程序提供有关的信息（例如对这些输入输出函数的声明），程序第1行“#include <stdio.h>”的作用就是用来提供这些信息的，stdio.h 是C编译系统提供的一个文件名，stdio 是“standard input & output”的缩写，即有关标准输入输出的信息。对此读者可不必深究，在第9章中有详细的介绍，在此只须记住：在程序中用到系统提供的标准函数库中的输入输出函数时，应在程序的开头写上下面一行：

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

例 1.2 求两数之和。

```
#include <stdio.h>
void main( )                          /*求两数之和*/
{
  int a,b,sum;                        /*这是声明部分，定义变量a、b、sum为整型*/
  a=123; b=456;                       /*以下3行为C语句 */
  sum=a+b;
  printf("sum is %d\n",sum);
}
```

本程序的作用是求两个整数 a 和 b 之和 sum。各行右侧的 /*……*/ 表示注释部分。注释可以用汉字或英文字符表示。注释只是给人看的，对编译和运行不起作用。注释可以出现在一行中的最右侧，也可以单独成为一行，可以根据需要写在程序中的任何一行中。第4行是声明部分，定义变量 a 和 b，指定 a 和 b 为整型(int)变量。第5行是两个

赋值语句，使 a 和 b 的值分别为 123 和 456。第 6 行使 sum 的值为 a＋b，第 7 行中“%d”是输入输出的“格式字符串”，“%d”表示“以十进制整数类型”，用来指定输入输出时的数据类型和格式(详见第 4 章)。在执行输出时，双撇号括起来的“sum is ”按原样输出，在“%d”的位置上代以一个十进制整数值，printf 函数中括号内逗号的右端 sum 是要输出的变量，现在它的值为 579(即 123 与 456 之和)，它应出现在“%d”的位置上，“\n”是换行符，实现换行。因此程序运行时输出以下信息：

```
sum is 579
```

例 1.3 求 2 个数中较大者。

```
#include <stdio.h>
void main( )                    /* 主函数 */
{
  int max(int x,int y);         /* 对被调用函数 max 的声明 */
  int a, b, c;                  /* 定义变量 a、b、c */
  scanf("%d,%d",&a,&b);         /* 输入变量 a 和 b 的值 */
  c=max(a,b);                   /* 调用 max 函数，将得到的值赋给 c */
  printf("max=%d\n",c);         /* 输出 c 的值 */
}

int max(int x,int y)            /* 定义 max 函数，函数值为整型，形式参数 x、y 为整型 */
{
  int z;                        /* max 函数中的声明部分，定义本函数中用到的变量 z 为整型 */
  if (x>y) z=x;
  else z=y;
  return(z);                    /* 将 z 的值返回，通过 max 带回到调用函数的位置 */
}
```

本程序包括两个函数：主函数 main 和被调用的函数 max。max 函数的作用是将 x 和 y 中较大者的值赋给变量 z。return 语句将 z 的值返回给主调函数 main。返回值是通过函数名 max 带回到 main 函数中的调用 max 函数的位置。程序第 4 行是对被调用函数 max 的声明，由于在 main 函数中要调用 max 函数，而 max 函数的位置在 main 函数之后，为了使编译系统能够正确识别和调用 max 函数，必须在调用 max 函数之前对 max 函数进行声明。有关函数的声明详见第 8 章，在此只初步了解即可。

main 函数中的 scanf 是“输入函数”的名字(scanf 和 printf 都是 C 的标准输入输出函数)。程序中 scanf 函数的作用是输入 a 和 b 的值。&a 和 &b 中的“&”的含义是“取地址”，此 scanf 函数的作用是：将两个数值分别输入到变量 a 和 b 的地址所标志的单元中，也就是输入给变量 a 和 b。scanf 函数中双撇号括起来的“%d,%d”的含义与前相同，只是现在用于“输入”。它指定输入的两个数据按十进制整数形式输入。关于 scanf 函数详见第 4 章。

在程序第 7 行中调用 max 函数，在调用时将实际参数 a 和 b 的值分别传送给 max 函数中的参数 x 和 y(称为形式参数)。经过执行 max 函数得到一个返回值(即 max 函数中变量 z 的值)，这个值返回到调用 max 函数的位置，即程序第 7 行“＝”的右侧，然后把这个值赋给变量 c。第 8 行输出变量 c 的值。在执行 printf 函数时，对双撇号括起来的“max＝%d\n”是这样处理的：将“max ＝”原样输出，“%d”将由 c 的值取代之，“\n”执行换行。程序运行情况如下：

8,5↙　　(输入 8 和 5 赋给 a 和 b)
max=8　　(输出 c 的值)

为了在分析运行情况时便于区别输入和输出的信息，本书对输入的信息加了下划线，如上面运行情况的第 1 行表示：从键盘输入 8 和 5，然后按 Enter 键。这样就把 8 和 5 分别送到计算机内存中(本例是送给变量 a 和 b)。第 2 行则是从计算机输出的信息，显示在屏幕上。

本例用到了函数调用、实际参数和形式参数等概念，在此只做了很简单的解释。读者如对此不大理解，可以先不予以深究，在学到以后有关章节时问题自然迎刃而解。在此介绍此例子，无非是使读者对 C 程序的组成和形式有一个初步的了解。

通过以上几个例子，可以看到：

(1) C 程序是由函数构成的。一个 C 源程序至少且仅包含一个 main 函数，也可以包含一个 main 函数和若干个其他函数。因此，函数是 C 程序的基本单位。被调用的函数可以是系统提供的库函数(例如 printf 和 scanf 函数)，也可以是用户根据需要自己编制设计的函数(例如，例 1.3 中的 max 函数)。C 的函数相当于其他语言中的子程序。用函数来实现特定的功能。程序全部工作都是由各个函数分别完成的。编写 C 程序就是编写一个个的函数。C 的函数库十分丰富，ANSI C 提供一百多个库函数，Turbo C 提供三百多个库函数。

C 语言的这种特点使得容易实现程序的模块化。

(2) 一个函数由两部分组成：

① 函数的首部，即函数的第 1 行，包括函数名、函数类型、函数属性、函数参数(形式参数)名、参数类型。

例如，例 1.3 中的 max 函数的首部为：

int	max	(int	x,	int	y)
↓	↓	↓	↓	↓	↓
函数类型	函数名	函数参数类型	函数参数名	函数参数类型	函数参数名

一个函数名后面必须跟一对圆括号，括号内写函数的参数名及其类型。函数可以没有参数，如 main( )。

② 函数体，即函数首部下面的花括号内的部分。如果一个函数内有多个花括号，则最外层的一对花括号为函数体的范围。

函数体一般包括以下两部分。

- 声明部分。在这部分中定义所用到的变量和对所调用函数的声明。如例 1.3 中 main 函数中对变量的定义“int a,b,c;” 和对所调用的函数的声明“int max(int x, int y); ”。
- 执行部分。由若干个语句组成。

当然，在某些情况下也可以没有声明部分(例如，例 1.1)，甚至可以既无声明部分也无执行部分。如：

```
void dump ( )
{
}
```

它是一个空函数，什么也不做，但这是合法的。

(3) 一个 C 程序总是从 main 函数开始执行的，而不论 main 函数在整个程序中的位置如何(main 函数可以放在程序最前头，也可以放在程序最后，或在一些函数之前，或在另一些函数之后)。

(4) C 程序书写格式自由，一行内可以写几个语句，一个语句可以分写在多行上，C 程序没有行号。

(5) 每个语句和数据声明的最后必须有一个分号。分号是 C 语句的必要组成部分。例如：

```
c=a+b;
```

分号是不可缺少的。即使是程序中最后一个语句也应包含分号。

(6) C 语言本身没有输入输出语句。输入和输出的操作是由库函数 scanf 和 printf 等函数来完成的。C 对输入输出实行“函数化”。由于输入输出操作牵涉具体的计算机设备，把输入输出操作放在函数中处理，就可以使 C 语言本身的规模较小，编译程序简单，很容易在各种机器上实现，程序具有可移植性。不同计算机系统除了提供标准函数外，还提供一些专门的函数，因此不同计算机系统中所提供的函数个数和功能是有所不同的。

(7) 可以用 /*……*/ 对 C 程序中的任何部分做注释。一个好的、有使用价值的源程序都应当加上必要的注释，以增加程序的可读性。

1.4 运行 C 程序的步骤与方法

1.4.1 运行 C 程序的步骤

前面已经列出了几个用 C 语言编写的程序。为了使计算机能按照人的意志进行工作，必须根据问题的要求，编写出相应的程序。所谓程序，就是一组计算机能识别和执行的指令。每一条指令使计算机执行特定的操作。用高级语言编写的程序称为“源程序(source program)”。实际上，计算机只能识别和执行由 0 和 1 组成的二进制的指令，而不能识别和执行用高级语言写的指令。为了使计算机能执行高级语言源程序，必须先用一种称为“编译程序”的软件，把源程序翻译成二进制形式的“目标程序(object program)”，然后再将该目标程序与系统的函数库以及其他目标程序连接起来，形成可执行的目标程序。

在编好一个 C 源程序后，如何上机运行呢？在纸上写好一个程序后，要经过这样几个步骤：上机输入与编辑源程序→对源程序进行编译→与库函数连接→运行目标程序。以上过程如图 1-1 所示。其中实线表示操作流程，虚线表示文件的输入输出。例如，编辑后得到一个源程序文件 f.c，然后在进行编译时再将源程序文件 f.c 输入，经过编译得到目标程序文件 f.obj，再将目标程序 f.obj 输入内存，与系统提供的库函数等连接，得到可执行的目标程序 f.exe，最后把 f.exe 调入内存并使之运行。

在了解了 C 语言的初步知识后，读者最好在计算机上运行一个 C 程序，以建立对 C 程序的初步认识。

1.4.2 上机运行 C 程序的方法

为了编译、连接和运行 C 程序，必须要有相应的 C 编译系统。目前使用的大多数 C 编译系统都是集成环境(IDE)的，把程序的编辑、编译、连接和运行等操作全部集中在一个界面上进行，功能丰富，使用方便，直观易用。

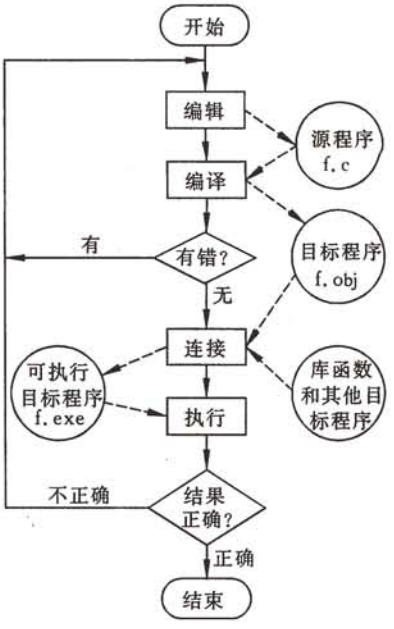


图 1-1 运行 C 程序的流程图

可以用不同的编译系统对 C 程序进行操作，常用的有 Turbo C 2.0、Turbo C++ 3.0、Visual C++ 等。前一段时间，Turbo C 2.0 用得比较多，但 Turbo C 2.0 是用于 DOS 环境的，在进入 Turbo C 集成环境后，不能用鼠标进行操作，主要通过键盘选择菜单，不大方便。近年来，不少人使用 Turbo C++ 3.0。Turbo C++ 3.0 也是一个集成环境，把程序的编辑、编译、连接和运行等操作全部集中在一个界面上进行，它具有方便、直观和易用的界面，虽然它也是 DOS 环境下的集成环境，但是可以把启动 Turbo C++ 3.0 集成环境的 DOS 执行文件 tc.exe 生成快捷方式，并以图标形式放在 Windows 桌面上，只要双击该图标，就能进入 Turbo C++ 3.0 集成环境，此外，它可以用鼠标操作菜单，因此在 Windows 环境下使用也感到十分方便。也可以用 Visual C++ 对 C 程序进行编译，由于目前学习 C++ 的人多数使用 Visual C++ 6.0，因此在学习时使用 Visual C++ 集成环境，也有利于今后进一步学习 C++ 语言。

学习本课程的目的主要是掌握 C 语言并利用它编制程序，写出源程序后可以用任何一种编译系统对程序进行编译和连接工作，只要用户感到方便、有效即可。不应当只会使用一种编译系统，而对其他的一无所知。无论用哪一种编译系统，都应当能举一反三，在需要时会用其他编译系统进行工作。

本节主要介绍在 Turbo C++ 3.0 中怎样编辑、编译、连接和运行 C 程序。在与本书配套的《C 程序设计题解与上机指导》(第三版)一书中还介绍怎样利用 Turbo C 2.0 和 Visual C++ 6.0 集成环境对 C 程序进行编辑、编译、连接和运行。读者可以根据自己的情况选用。

Turbo C++ 3.0 是 Borland 公司为 C++ 程序的编辑、编译、连接和运行而研制的集成环境。由于 C++ 是从 C 语言发展而来的，C++ 对于 C 程序是兼容的，因此可以用C++ 的编译系统对 C 程序进行编译，或者说一个 C 程序可以在 C++ 集成环境中进行调试和运行。

为了能使用 Turbo C++ 3.0，必须先将 Turbo C++ 3.0 编译程序装入磁盘的某一目录下，例如放在 C 盘根目录下一级 TC3.0 子目录下。上机运行 C 程序有以下几个步骤。

1. 进入 Turbo C++ 3.0 集成环境

可以通过两种方法得到 Turbo C++ 3.0 集成环境。

(1) 在 DOS 环境下。如果用户的当前目录是 Turbo C++ 3.0 编译程序所在的子目

录(例如 C:\TC3.0),可以在 DOS 环境下用键盘输入 DOS 命令"tc"即可:

C:\TC3.0>tc ↙

这时就执行 C:\TC3.0 子目录中的 tc.exe 文件,屏幕上出现 Turbo C 集成环境,见图 1-2 所示。

(2) 在 Windows 环境下。先通过浏览找到 Turbo C ++ 3.0 集成环境所在的子目录(如 C:\TC3.0),从中找到可执行文件 tc.exe,创建其快捷方式,并拖曳到 Windows 桌面上,用一个图标表示。双击该图标,就可打开如图 1-2 所示的 Turbo C ++ 3.0 集成环境。

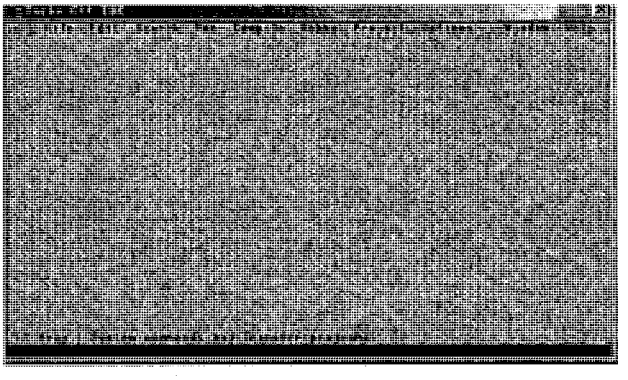


图 1-2 Turbo C ++ 3.0 集成环境

从图 1-2 可以看到:在集成环境的上部,有一行"主菜单",其中包括 10 个菜单项:

File Edit Search Run Compile Debug Project Options Window Help

用户可以通过以上菜单来选择使用集成环境所提供的 Turbo C ++ 3.0 的各项主要功能。以上 10 个菜单项分别代表:文件操作、编辑、寻找、运行、编译、调试、项目文件、选项、窗口、帮助。用鼠标可以选择菜单条中所需要的菜单项,单击此菜单项就会出现一个下拉菜单。

2. 编辑源文件

(1) 如果新建立一个源程序,可以用鼠标单击 File 菜单,然后在其下拉菜单选择 New(见图 1-3),表示要建立一个新的 C 源程序。屏幕上出现如图 1-4 所示的界面,上部是编辑窗口,供用户输入源程序。

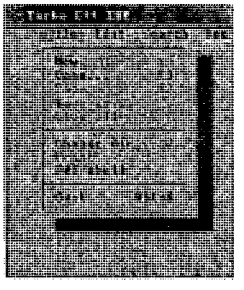


图 1-3 单击 File,选中 New

(2) 如果想对一个已有的源程序进行修改,应选择 File →Open(即单击 File 的下拉菜单中的 Open 项,下面均以此方法表示),此时屏幕上出现一个对话框,见图 1-5。用户可以在 Name 下面输入指定的文件路径和文件名(也可以只输入指定的文件路径并单击 Open 按钮,下面的文件列表中找到所需的文件名),然后单击 Open 按钮。系统会将此文件调入内存并显示在图 1-4 所示的编辑窗口中。此时集成环境自动设为编辑(Edit)状态。

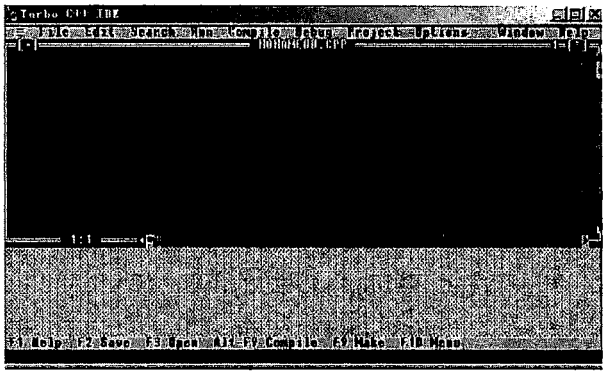


图 1-4　编辑源程序窗口

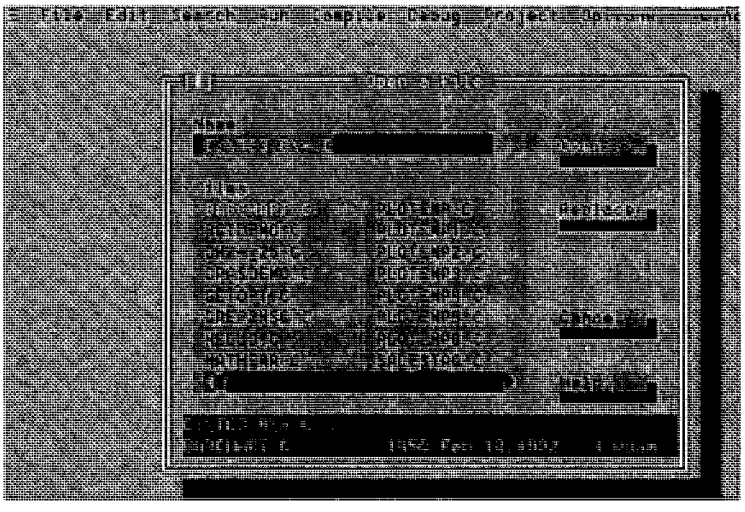


图 1-5　打开文件对话框

在编辑(Edit) 状态下,光标表示当前进行编辑的位置,在此位置可以进行插入、删除或修改,直到自己满意为止。在完成编辑之后,应当保存源程序。如果该源程序是已有的,则选择 File→Save 保存已修改过的源程序。如果该源程序是新输入的,则选择 File→Save ,并在弹出的 Save File As 对话框中的 Name 栏中输入文件路径和文件名,见图 1-6。

说明:

① C 程序的后缀应该是. c,如 c1. c、c2. c 等。由于现在是用 Turbo C ++ 3. 0 集成环境,它把源程序默认作 C ++ 程序。如果用户在保存源程序时文件名未加后缀,则系统会认为其是 C ++ 程序,自动加上后缀. cpp,如 c1. cpp、c2. cpp 等。cpp 是"C Plus Plus"的缩写,意为"C ++ "。如果在输入源程序后在保存时加上后缀. c,在编译时系统能识别并编译以. c 为后缀的 C 程序。也就是说,Turbo C ++ 3. 0 能编译以. cpp 为后缀的 C ++ 源程序,也能编译以. c 为后缀的 C 源程序(按照 Turbo C ++ 的语法规定进行编译)。如果用 Visual C ++ 6. 0,也按同样方法处理。如果用 Turbo 2. 0,则系统自动加上的后缀是. c。

② 如果在进入 Turbo C ++ 3. 0 集成环境后鼠标的形状不是箭头形状,而是实心矩

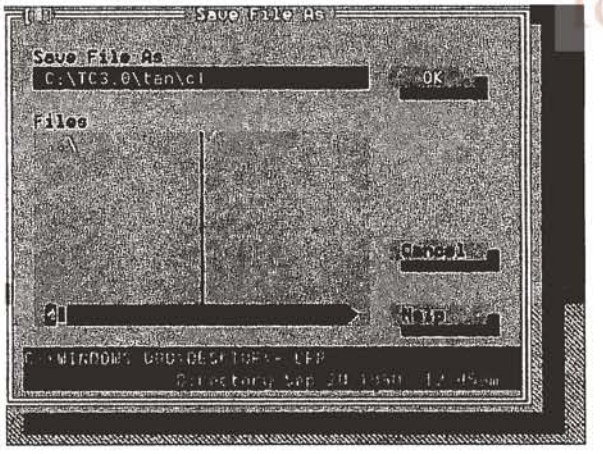


图 1-6　保存文件对话框

形，可以按“Alt＋Enter 组合键”使鼠标的形状改变为箭头形状。

3. 对源程序进行编译

选择 Compile 菜单，并在其下拉菜单中选择 Compile(也可以按 Alt＋F9)对源程序进行编译。屏幕上显示编译消息框，如图 1-7 所示。从图 1-7 可以看到：编译 c1. cpp 源程序，出现 1 个错误(error)，0 个警告(warning)。在按任何键后，消息框消失，在集成环境的下部消息 (Message)框中具体地指出在哪一行发生错误以及错误的原因，光标停留在与错误有关的行上，提醒用户改正错误。在修改程序后再进行编译，直到不出现错误和警告为止。此时两处所示得到一个后缀为. obj 的目标程序。

图 1-7　编译消息框

4. 将目标程序进行连接

假如一个程序包含多个文件，在分别对每个源程序进行编译并得到多个目标程序后，

要把这些目标程序连接起来，同时还要和系统提供的资源（如函数库）连接成为一个整体。方法是选择菜单 Compile →Link，如果不出现错误，会得到一个后缀为. exe 的可执行文件。应当说明的是：如果一个程序只包含一个文件，也必须进行连接，因为还要与系统提供的资源连接。

也可以将编译和连接合为一个步骤进行。选菜单 Compile →Make（或按 F9 键）即可一次完成编译和连接。在屏幕上会显示编译或连接时有无错误和有几个错误。按任何一个键，图 1-7 所显示的“编译消息框”会消失，屏幕上会恢复显示源程序。

5. 执行程序

选菜单 Run →Run（或按 Ctrl+F9 组合键），系统就会执行已编译和连接好的可执行文件。如果程序需要输入数据（如例 1.3），则屏幕会切换到运行窗口，等待用户输入数据，并输出结果。但在人们未来得及看清结果之前，屏幕很快又转回程序编辑窗口。为了能看清结果，可以按 Alt+F5 组合键，此时屏幕切换到运行窗口，用户可以充分观察和分析输出结果，最后按任何一个键，屏幕会切换到编辑窗口，见图 1-8。图 1-8 中第 1 行是用户输入给 a 的值，第 2 行是程序输出的结果。

图 1-8　编辑窗口

如果发现运行结果不对，要重新修改源程序，并重复上述 2、3、4 步骤，直至得到正确结果为止。

6. 退出 Turbo C++ 3.0 环境

在完成 C 语言作业后，可以选择 File→Quit，就会退出 Turbo C++ 3.0 环境，回到 Windows 环境。

以上简单介绍了 Turbo C++ 3.0 集成环境的使用，有了这些初步知识，就可以上机调试和运行 C 程序了，更深入地使用请参考 Turbo C++ 3.0 的使用手册或 Turbo C++ 3.0 的 Help 系统。

除了可以用 Turbo C++ 3.0 以外，还可以用 Visual C++ 6.0 对 C 程序进行编译和运行。由于篇幅关系，在本节中不介绍 Visual C++ 6.0 的使用方法，如果读者需要，可以参考作者编著的《C 程序设计题解与上机指导》（第三版）（清华大学出版社出版）。

习　　题

1.1　请根据自己的认识，写出 C 语言的主要特点。

1.2　C 语言的主要用途是什么？它和其他高级语言有什么异同？

1.3 写出一个C程序的构成。

1.4 C语言以函数为程序的基本单位，有什么好处？

1.5 请参照本章例题，编写一个C程序，输出以下信息：

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                        Very good!
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

1.6 编写一个C程序，输入a、b、c 3个值，输出其中最大者。

1.7 上机运行本章3个例题，熟悉所用系统的上机方法与步骤。

1.8 上机运行本章习题1.5和习题1.6。

第2章　程序的灵魂——算法

一个程序应包括以下两个方面的内容。

(1) 对数据的描述。在程序中要指定数据的类型和数据的组织形式，即数据结构(data structure)。

(2) 对操作的描述。即操作步骤，也就是算法(algorithm)。

数据是操作的对象，操作的目的是对数据进行加工处理，以得到期望的结果。打个比方，厨师制作菜肴，需要有菜谱，菜谱上一般应包括：①配料，指出应使用哪些原料；②操作步骤，指出如何使用这些原料按规定的步骤加工成所需的菜肴，没有原料是无法加工成所需菜肴的。面对同一些原料可以加工出不同风味的菜肴。作为程序设计人员，必须认真考虑和设计数据结构和操作步骤(即算法)。著名计算机科学家沃思(Nikiklaus Wirth)提出一个公式：

数据结构 + 算法 = 程序

实际上，一个程序除了以上两个主要要素之外，还应当采用结构化程序设计方法进行程序设计，并且用某一种计算机语言表示。因此，算法、数据结构、程序设计方法和语言工具4个方面是一个程序设计人员所应具备的知识。在设计一个程序时要综合运用这几方面的知识。在本书中不可能全面介绍这些内容，它们都属于有关的专门课程范畴。在这4个方面中，算法是灵魂，数据结构是加工对象，语言是工具，编程需要采用合适的方法。算法是解决“做什么”和“怎么做”的问题。程序中的操作语句，实际上就是算法的体现。显然，不了解算法就谈不上程序设计。本书不是一本专门介绍算法的教材，也不是一本只介绍C语言语法规则的使用说明。本书的目的是使读者通过学习，能够知道怎样编写一个C程序，并且能够编写出不太复杂的C程序。通过一些实例把以上4个方面的知识结合起来，介绍如何编写一个C程序。

由于算法的重要性，在本章中先介绍有关算法的初步知识，以便为后面各章的学习建立一定的基础。

2.1　算法的概念

做任何事情都有一定的步骤。例如，你想从北京去天津开会，首先要去买火车票，然后按时乘坐地铁到北京站，登上火车，到天津站后坐电车到会场，参加会议；你要买电视机，先要选好货物，然后开票，付款，拿发票，取货，打车回家；要考大学，首先要填报名单，交报名费，拿到准考证，按时参加考试，得到录取通知书，到指定学校报到注册等。这些步骤都是按一定的顺序进行的，缺一不可，次序错了也不行。我们从事各种工作和活动，都必须事先想好进行的步骤，然后按部就班地进行，才能避免产生错乱。实际上，在日常生

活中，由于已养成习惯，所以人们并不意识到每件事都需要事先设计出“行动步骤”。例如吃饭、上学、打球、做作业等，事实上都是按照一定的规律进行的，只是人们不必每次都重复考虑它而已。

不要认为只有“计算”的问题才有算法。广义地说，为解决一个问题而采取的方法和步骤，就称为“算法”。例如，描述太极拳动作的图解，就是“太极拳的算法”。一首歌曲的乐谱，也可以称为该歌曲的算法，因为它指定了演奏该歌曲的每一个步骤，按照它的规定就能演奏出预定的曲子。

对同一个问题，可以有不同的解题方法和步骤。例如，求 1＋2＋3＋…＋100，即 $\sum_{n=1}^{100} n$。有人可能先进行 1＋2，再加 3，再加 4，一直加到 100，而有的人采取这样的方法：$\sum_{n=1}^{100} n = 100+(1+99)+(2+98)+\cdots+(49+51)+50 = 100+49\times 100+50 = 5050$。还可以有其他的方法。当然，方法有优劣之分。有的方法只需进行很少的步骤，而有些方法则需要较多的步骤。一般说，希望采用方法简单、运算步骤少的方法。因此，为了有效地进行解题，不仅需要保证算法正确，还要考虑算法的质量，选择合适的算法。

本书所关心的当然只限于计算机算法，即计算机能执行的算法。例如，让计算机算 1×2×3×4×5，或将 100 个学生的成绩按高低分数的次序排列，是可以做到的，而让计算机去执行“替我理发”或“煎一份牛排”，是做不到的（至少目前如此）。

计算机算法可分为两大类别：数值运算算法和非数值运算算法。数值运算的目的是求数值解，例如求方程的根、求一个函数的定积分等，都属于数值运算范围。非数值运算包括的面十分广泛，最常见的是用于事务管理领域，例如图书检索、人事管理、行车调度管理等。目前，计算机在非数值运算方面的应用远远超过了在数值运算方面的应用。由于数值运算有现成的模型，可以运用数值分析方法，因此对数值运算的算法的研究比较深入，算法比较成熟。对各种数值运算都有比较成熟的算法可供选用。人们常常把这些算法汇编成册（写成程序形式），或者将这些程序存放在磁盘或磁带上，供用户调用。例如有的计算机系统提供“数学程序库”，使用起来十分方便。而非数值运算的种类繁多，要求各异，难以规范化，因此只对一些典型的非数值运算算法（例如排序算法）作比较深入的研究。其他的非数值运算问题，往往需要使用者参考已有的类似算法，重新设计解决特定问题的专门算法。本书不可能罗列所有算法，只是想通过一些典型算法的介绍，帮助读者了解如何设计一个算法，帮助读者举一反三。希望读者通过本章介绍的例子了解怎样提出问题，怎样思考问题，怎样表示一个算法。

2.2 简单算法举例

例 2.1 求 1×2×3×4×5。

可以用最原始的方法进行：

步骤 1：先求 1＊2，得到结果 2。

步骤 2：将步骤 1 得到的乘积 2 再乘以 3，得到结果 6。

步骤 3:将 6 再乘以 4,得 24。

步骤 4:将 24 再乘以 5,得 120。这就是最后的结果。

这样的算法虽然是正确的,但太繁琐。如果要求 $1\times2\times\cdots\times1000$,则要写 999 个步骤,显然是不可取的。而且每次都直接使用上一步骤的数值结果(如 2、6、24 等),也不方便。应当能找到一种通用的表示方法。

可以设两个变量:一个变量代表被乘数,一个变量代表乘数。不另设变量存放乘积结果,而直接将每一步骤的乘积放在被乘数变量中。今设 p 为被乘数,i 为乘数。用循环算法来求结果。可以将算法改写如下:

S1:使 p=1

S2:使 i=2

S3:使 p * i,乘积仍放在变量 p 中,可表示为:p * i⇒p

S4:使 i 的值加 1,即 i+1⇒i

S5:如果 i 不大于 5,返回重新执行步骤 S3 以及其后的步骤 S4 和 S5;否则,算法结束。最后得到 p 的值就是 5! 的值。

上面的 S1,S2…代表步骤 1,步骤 2…。S 是 Step(步)的缩写。这是写算法的习惯用法。

请读者仔细分析这个算法,能否得到预期的结果。显然这个算法比前面列出的算法简练。

如果题目改为:求 $1\times3\times5\times7\times9\times11$。

算法只需作很少的改动即可:

S1:1⇒p

S2:3⇒i

S3:p * i⇒p

S4:i+2⇒i

S5:若 i≤11,返回 S3;否则,结束。

(S5 步骤也可以表示为:"若 i>11,结束;否则返回 S3。"作用是相同的。)

可以看出用这种方法表示的算法具有通用性、灵活性。S3 到 S5 组成一个循环,在实现算法时,要反复多次执行 S3、S4、S5 步骤,直到某一时刻,执行 S5 步骤时经过判断,乘数 i 已超过规定的数值而不返回 S3 步骤为止。此时算法结束,变量 p 的值就是所求结果。

由于计算机是高速进行运算的自动机器,实现循环是轻而易举的,所有计算机高级语言中都有实现循环的语句,因此,上述算法不仅是正确的,而且是计算机能实现的较好的算法。

请读者仔细分析循环结束的条件,即 S5 步骤。如果在求 $1\times2\times\cdots\times11$ 时,将 S5 步骤写成:

S5:若 i<11,返回 S3。

这样会有什么问题?得到什么结果?

例 2.2 有 50 个学生,要求将他们之中成绩在 80 分以上的学号和成绩输出。用 n

表示学生学号，n_1 代表第一个学生学号，n_i 代表第 i 个学生学号。用 G 代表学生成绩，g_i 代表第 i 个学生成绩，算法可表示如下：

S1：1⇒i

S2：如果 $g_i \geqslant 80$，则输出 n_i 和 g_i；否则不输出

S3：i＋1⇒i

S4：如果 $i \leqslant 50$，返回 S2，继续执行；否则，算法结束。

本例中，变量 i 作为下标，用它来控制序号（第几个学生，第几个成绩）。当 i 超过 50 时，表示已对 50 个学生的成绩处理完毕，算法结束。

例 2.3 判定 2000—2500 年中的每一年是否闰年，将结果输出。

闰年的条件是：(1)能被 4 整除，但不能被 100 整除的年份都是闰年，如 1996 年、2004 年是闰年；(2)能被 100 整除，又能被 400 整除的年份是闰年，如 1600 年、2000 年是闰年。不符合这两个条件的年份不是闰年。

算法可表示如下：

设 y 为被检测的年份。可采取以下步骤：

S1：2000⇒y

S2：若 y 不能被 4 整除，则输出 y“不是闰年”。然后转到 S6

S3：若 y 能被 4 整除，不能被 100 整除，则输出 y“是闰年”。然后转到 S6

S4：若 y 能被 100 整除，又能被 400 整除，输出 y“是闰年”，然后转到 S6

S5：输出 y“不是闰年”

S6：y＋1⇒y

S7：当 $y \leqslant 2500$ 时，转 S2 继续执行，否则算法停止。

在这个算法中，采取了多次判断。先判断 y 能否被 4 整除，如不能，则 y 必然不是闰年。如 y 能被 4 整除，并不能马上决定它是否闰年，还要看它能否被 100 整除。如不能被 100 整除，则肯定是闰年（例如 1996 年）。如能被 100 整除，还不能判断它是否闰年，还要被 400 整除，如果能被 400 整除，则它是闰年；否则不是闰年。

在这个算法中，每做一步，都分别分离出一些范围（已能判定为闰年或非闰年），逐步缩小范围，使被判断的范围愈来愈小，直至执行 S5 时，只可能是非闰年，见图 2-1 示意。

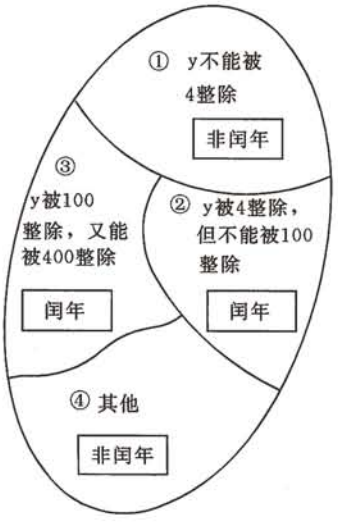


图 2-1

从图 2-1 可以看出：“其他”这一部分，包括能被 4 整除，又能被 100 整除，而不能被 400 整除的那些年份（如 1990 年），它们是非闰年。

在考虑算法时，应当仔细分析所需判断的条件，如何一步一步缩小被判断的范围。有的问题，判断的先后次序是无所谓的；而有的问题，判断条件的先后次序是不能任意颠倒的，读者可根据具体问题决定其逻辑。

例 2.4 求 $1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\cdots+\frac{1}{99}-\frac{1}{100}$。

算法可以表示如下：

S1：sign＝1

S2：sum＝1

S3：deno＝2

S4：sign＝(－1) * sign

S5：term＝sign * (1/deno)

S6：sum＝sum＋term

S7：deno＝deno＋1

S8：若 deno≤100 返回 S4；否则算法结束。

本例中用有含义的单词作变量名，以使算法更易于理解。sum 表示累加和，deno 是“分母”英文 denom inator 的缩写，sign 代表数值的符号，term 代表某一项。在步骤 S1 中先预设 sign(代表多项式中各项的符号，它的值为 1 或－1)。在步骤 S2 中使 sum 等于 1，相当于已将多项式中的第一项放到了 sum 中。在步骤 S3 中使分母的值为 2。在步骤 S4 中使 sign 的值变为－1。在步骤 S5 中求出多项式中第二项的值(－1/2)。在步骤 S6 中将刚才求出的第二项的值(－1/2)累加到 sum 中。至此，sum 的值是(1－1/2)。在步骤 S7 中使分母 deno 的值加 1(变成 3)。执行 S8 步骤，由于 deno≤100，故返回 S4 步骤，sign 的值改为 1，在 S5 中求出 term 的值为 1/3，在 S6 中将 1/3 累加到 sum 中。然后 S7 再使分母变为 4。按此规律反复执行 S4 到 S8 步骤，直到分母大于 100 为止。一共执行了 99 次循环，向 sum 累加入了 99 个分数。sum 最后的值就是多项式的值。

例 2.5 对一个大于或等于 3 的正整数，判断它是不是一个素数。

所谓素数，是指除了 1 和该数本身之外，不能被其他任何整数整除的数。例如，13 是素数，因为它不能被 2，3，4，…，12 整除。

判断一个数 $n(n\geqslant 3)$是否素数的方法是很简单的：将 n 作为被除数，将 2 到$(n-1)$各个整数先后作为除数，如果都不能被整除，则 n 为素数。

算法可以表示如下：

S1：输入 n 的值

S2：i＝2(i 作为除数)

S3：n 被 i 除，得余数 r

S4：如果 r＝0，表示 n 能被 i 整除，则输出 n“不是素数”，算法结束；否则执行 S5

S5：i＋1⇒i

S6：如果 i≤n－1，返回 S3；否则输出 n “是素数”，然后结束。

实际上，n 不必被 2 到$(n-1)$的整数除，只需被 2 到 $n/2$ 间整数除即可，甚至只需被 2 到$\sqrt{n}$之间的整数除即可。例如，判断 13 是否素数，只需将 13 被 2、3 除即可，如都除不尽，n 必为素数。S6 步骤可改为：

S6：如果 i≤$\sqrt{n}$，返回 S3；否则算法结束。

通过以上几个例子，可以初步了解怎样设计一个算法。

2.3 算法的特性

一个算法应该具有以下特点。

(1) 有穷性。一个算法应包含有限的操作步骤，而不能是无限的。例如 2.2 节中例 2.4 的算法，如果将 S8 步骤改为："若 deno＞0，返回 S4"，则循环永远不会停止。这不是有穷的步骤。事实上，"有穷性"往往指"在合理的范围之内"。如果让计算机执行一个历时 1000 年才结束的算法，这虽然是有穷的，但超过了合理的限度，人们也不把它视为有效算法。究竟什么算"合理限度"，并无严格标准，由人们的常识和需要而定。

(2) 确定性。算法中的每一个步骤都应当是确定的，而不应当是含糊的、模棱两可的。例如，有一个健身操的动作要领，其中有一个动作："手举过头顶"，这个步骤就是不确定的，含糊的。是双手都举过头？还是左手？或右手？举过头顶多少厘米？不同的人可以有不同的理解。算法中的每一个步骤应当不致被解释成不同的含义，而应是十分明确无误的。如例 2.5 中的 S3 步骤如果写成 "n 被一个整数除，得余数 r"，这也是"不确定"的，它没有说明 n 被哪个整数除，因此无法执行。也就是说，算法的含义应当是惟一的，而不应当产生"歧义性"。所谓"歧义性 "是指可以被理解为两种(或多种)的可能含义。

(3) 有零个或多个输入。所谓输入是指在执行算法时需要从外界取得必要的信息。例如，在执行例 2.5 算法时，需要输入 n 的值，然后判断 n 是否素数。也可以有两个或多个输入，例如，求两个整数 m 和 n 的最大公约数，则需要输入 m 和 n 的值。一个算法也可以没有输入，例如，例 2.1 在执行算法时不需要输入任何信息，就能求出 5!。

(4) 有一个或多个输出。算法的目的是为了求解，"解" 就是输出。如例 2.5 求素数的算法，最后输出的 n "是素数"或"不是素数"就是输出的信息。但算法的输出不一定就是计算机的打印输出，一个算法得到的结果就是算法的输出。没有输出的算法是没有意义的。

(5) 有效性。算法中的每一个步骤都应当能有效地执行，并得到确定的结果。例如，若 $b=0$ ，则执行 a/b 是不能有效执行的。

对于那些不熟悉计算机的人来说，可以使用别人已设计好的现成算法，只需根据算法的要求给予必要的输入，就能得到输出的结果。对他们来说，算法如同一个"黑箱子"一样，他们可以不了解"黑箱子"中的结构，只是从外部特性上了解算法的作用，即可方便地使用算法。例如，对一个"输入 3 个数，求其中最大值"的算法，可以用图 2-2 表示，只要输入 a、b、c 这 3 个数，执行算法后就能得到其中最大的数。但对于程序设计人员来说，必须会设计算法，并且根据算法编写程序。

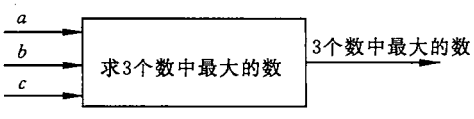


图 2-2

2.4 怎样表示一个算法

为了表示一个算法，可以用不同的方法。常用的方法有：自然语言、传统流程图、结构化流程图、伪代码、PAD 图等。

2.4.1 用自然语言表示算法

2.2 节介绍的算法是用自然语言来表示的，自然语言就是人们日常使用的语言，可以是汉语、英语，或其他语言。用自然语言表示通俗易懂，但文字冗长，容易出现歧义性。自然语言表示的含义往往不大严格，要根据上下文才能判断其正确含义。假如有这样一句话："张先生对李先生说他的孩子考上了大学"。请问是张先生的孩子考上大学呢还是李先生的孩子考上大学呢？光从这句话本身难以判断。此外，用自然语言来描述包含分支和循环的算法，不很方便（如例 2.5 的算法）。因此，除了那些很简单的问题以外，一般不用自然语言描述算法。

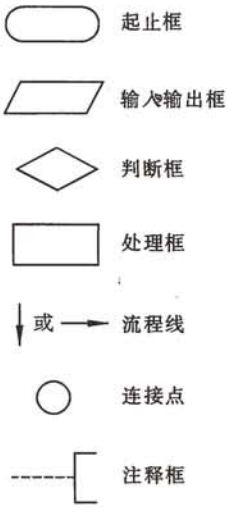


图 2-3

2.4.2 用流程图表示算法

流程图是用一些图框来表示各种操作。用图形表示算法，直观形象，易于理解。美国国家标准化协会 ANSI（American National Standard Institute）规定了一些常用的流程图符号（见图 2-3），已为世界各国程序工作者普遍采用。

图 2-3 中菱形框的作用是对一个给定的条件进行判断，根据给定的条件是否成立决定如何执行其后的操作。它有一个入口，两个出口。见图 2-4 示意。

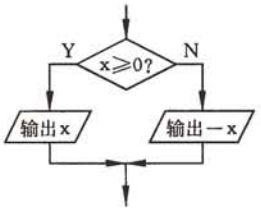


图 2-4

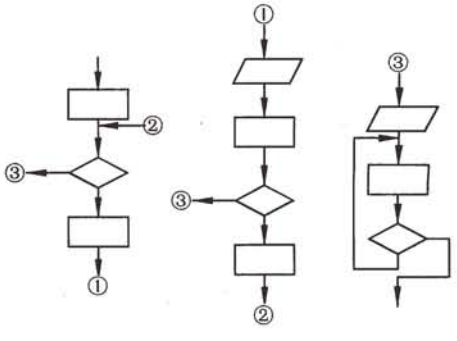


图 2-5

连接点（小圆圈）是用于将画在不同地方的流程线连接起来。如图 2-5 中有两个以 1 为标志的连接点（在连接点圈中写上"1"），它表示这两个点是互相连接在一起的，实际上它们是同一个点，只是画不下才分开来画。用连接点，可以避免流程线的交叉或过长，使

流程图清晰。注释框不是流程图中必要的部分，不反映流程和操作，只是为了对流程图中某些框的操作作必要的补充说明，以帮助阅读流程图的人更好地理解流程图的作用。

对 2.2 节中所举的几个算法例子，改用流程图表示。

例 2.6 将例 2.1 求 5！的算法用流程图表示，流程图见图 2-6。

如果需要将最后结果输出，可以在菱形框的下面再加一个输出框，见图 2-7。

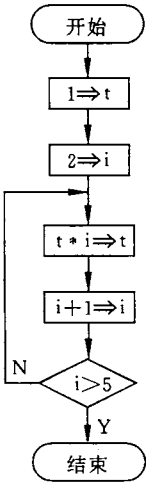


图 2-6

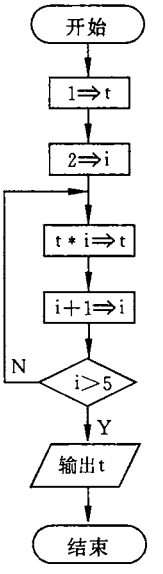


图 2-7

例 2.7 将例 2.2 的算法用流程图表示。将 50 名学生中成绩在 80 分以上者的学号和成绩输出。

见图 2-8，菱形框两侧的“Y”和“N”代表“是”(yes)和“否”(no)。在此算法中没有包括输入 50 个学生数据的部分。如果包括这个输入数据的部分，流程图如图 2-9 所示。

例 2.8 将例 2.3 判定闰年的算法用流程图表示。

见图 2-10，显然，用图 2-10 表示算法要比用文字描述算法逻辑清晰、易于理解。

请读者考虑，如果例 2.3 所表示的算法中，S2 步骤内没有最后“转到 S6”这一句话，而只是

S2：若 y 不能被 4 整除，则输出 y “不是闰年”

这样就意味着执行完 S2 步骤后，不论 S2 的执行情况如何都应执行 S3 步骤。请读者画出相应的流程图。请思考这样的算法在逻辑上有什么错误？从流程图上是很容易发现逻辑上的错误的。

例 2.9 将例 2.4 的算法用流程图表示。即求：$1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\cdots+\frac{1}{99}-\frac{1}{100}$，见图 2-11。

例 2.10 将例 2.5 判断素数的算法用流程图表示，见图 2-12。

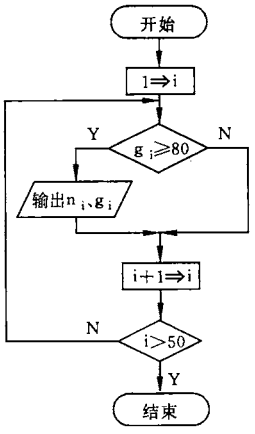


图 2-8

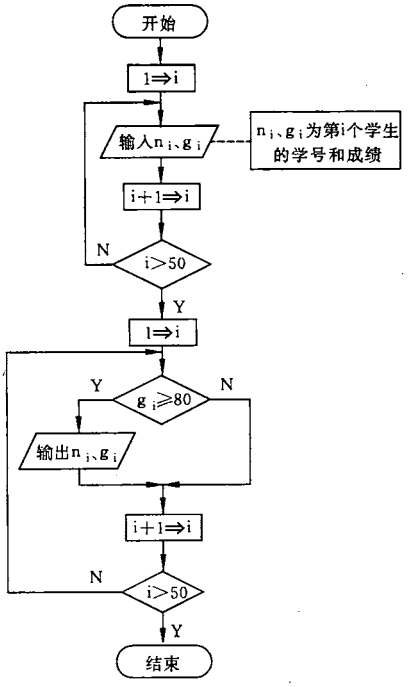


图 2-9

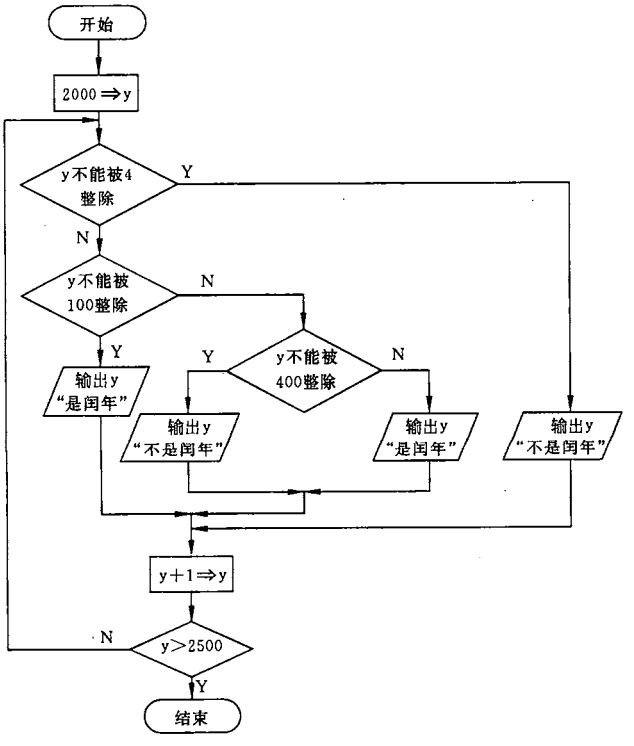


图 2-10

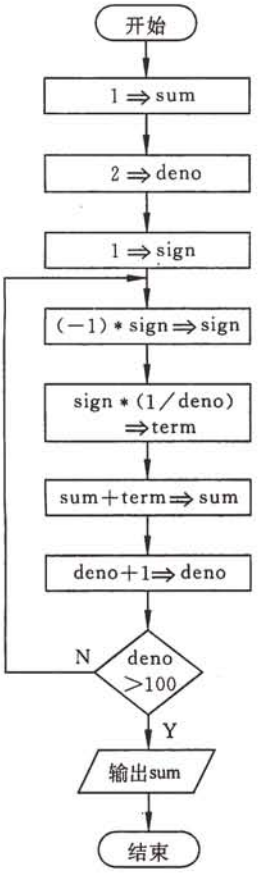


图 2-11

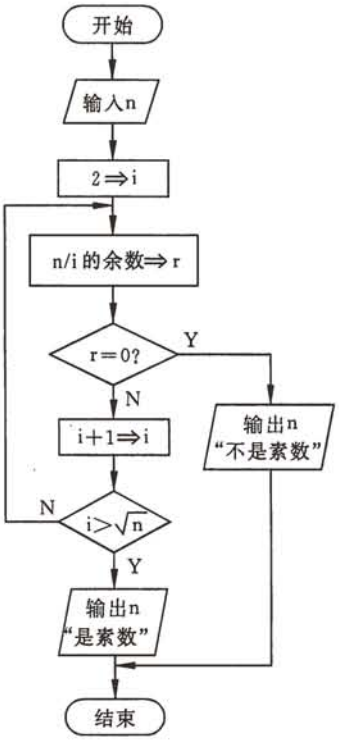


图 2-12

通过以上几个例子可以看出流程图是表示算法的较好的工具。一个流程图包括以下几部分。

(1) 表示相应操作的框；

(2) 带箭头的流程线；

(3) 框内外必要的文字说明。

需要提醒的是：流程线不要忘记画箭头，因为它是反映流程的执行先后次序的，如不画出箭头就难以判定各框的执行次序了。

用流程图表示算法直观形象，比较清楚地显示出各个框之间的逻辑关系。有一段时期国内外计算机书刊都广泛使用这种流程图表示算法。但是，这种流程图占用篇幅较多，尤其当算法比较复杂时，画流程图既费时又不方便。在结构化程序设计方法推广之后，许多书刊已用 N-S 结构化流程图代替这种传统的流程图。但是每一个程序编制人员都应当熟练掌握传统流程图，会看会画。

2.4.3 3种基本结构和改进的流程图

1. 传统流程图的弊端

传统的流程图用流程线指出各框的执行顺序，对流程线的使用没有严格限制。因此，使用者可以毫不受限制地使流程随意地转来转去，使流程图变得毫无规律，阅读者要花很大精力去追踪流程，使人难以理解算法的逻辑。这种情况如图 2-13 所示。这种如同乱麻一样的算法称为 BS 型算法，意为一碗面条(a bowl of spaghetti)，乱无头绪。

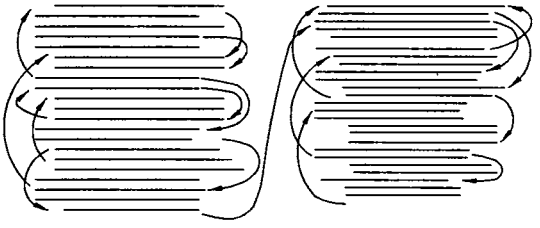


图 2-13

像图 2-13 这样的算法是不好的，难以阅读，也难以修改，从而使算法的可靠性和可维护性难以保证。如果写出的算法能限制流程的无规律任意转向，像一本书那样由各章各节顺序组成，那么阅读起来就很方便，不会有任何困难，只需从头到尾顺序地看下去即可。而如果一本书不是由各章节顺序组成，而是毫无规律地乱排(例如第 1 章从 36 页开始到 47 页，第 2 章从 98 页到 107 页，第 3 章从 19 页到 35 页……)，各章内各节也是毫无规律地乱排，阅读这种书是不会感到愉快的。

为了提高算法的质量，使算法的设计和阅读方便，必须限制箭头的滥用，即不允许无规律地使流程随意转向，只能顺序地进行下去。但是，算法上难免会包含一些分支和循环，而不可能全部由一个一个的框顺序组成。例如图 2-6 到图 2-12 都不是由各框顺序进行的，都包含一些流程的向前或向后的非顺序转向。为了解决这个问题，人们规定出几种基本结构，然后由这些基本结构按一定规律组成一个算法结构(如同用一些基本预制构件来搭成房屋一样)，如果能做到这一点，算法的质量就能得到保证和提高。

2. 3种基本结构

1966 年，Bohra 和 Jacopini 提出了以下 3 种基本结构，用这 3 种基本结构作为表示一个良好算法的基本单元。

(1) 顺序结构。如图 2-14 所示，虚线框内是一个顺序结构。其中 A 和 B 两个框是顺序执行的。即：在执行完 A 框所指定的操作后，必然接着执行 B 框所指定的操作。顺序结构是最简单的一种基本结构。

(2) 选择结构。选择结构又称选取结构或分支结构，如图 2-15 所示。虚线框内是一个选择结构。此结构中必包含一个判断框。根据给定的条件 p 是否成立而选择执行 A 框或 B 框。例如 p 条件可以是“$x \geqslant 0$”或“$x > y$”，“$a+b < c+d$”等，详见第 5 章。

注意：无论 p 条件是否成立，只能执行 A 框或 B 框之一，不可能既执行 A 框又执行

B 框。无论走哪一条路径，在执行完 A 或 B 之后，都经过 b 点，然后脱离本选择结构。A 或 B 两个框中可以有一个是空的，即不执行任何操作，如图 2-16 所示。

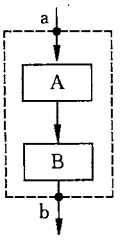


图 2-14

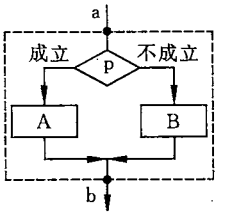


图 2-15

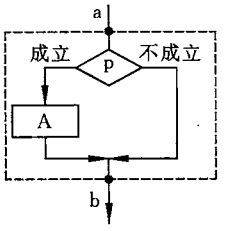


图 2-16

(3) 循环结构。它又称重复结构，即反复执行某一部分的操作。有两类循环结构。

① 当(while)型循环结构。当型循环结构如图 2-17(a)所示。它的功能是：当给定的条件 p_1 成立时，执行 A 框操作，执行完 A 后，再判断条件 p_1 是否成立，如果仍然成立，再执行 A 框，如此反复执行 A 框，直到某一次 p_1 条件不成立为止，此时不执行 A 框，而从 b 点脱离循环结构。

② 直到(until)型循环结构。直到型循环结构如图 2-17(b)所示。它的功能是：先执行 A 框，然后判断给定的 p_2 条件是否成立，如果 p_2 条件不成立，则再执行 A，然后再对 p_2 条件作判断，如果 p_2 条件仍然不成立，又执行 A……如此反复执行 A，直到给定的 p_2 条件成立为止，此时不再执行 A，从 b 点脱离本循环结构。

图 2-18 是当型循环的应用例子，图 2-19 是直到型循环的应用例子。

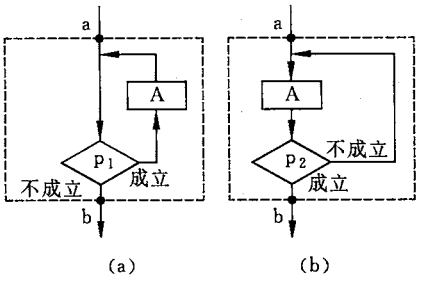


图 2-17

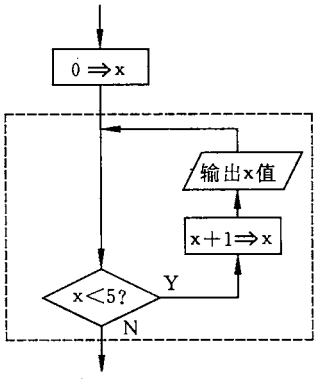


图 2-18

图 2-18 和图 2-19 的作用都是输出 5 个数：1,2,3,4,5。可以看到：对同一个问题既可以用当型循环来处理，也可以用直到型循环来处理。

以上 3 种基本结构，有以下共同特点：

(1) 只有一个入口。图 2-14～图 2-17 中的 a 点为入口点。

(2) 只有一个出口。图 2-14～图 2-17 中的 b 点为出口点。请注意，一个判断框有两个出口，而一个选择结构只有一个出口。不要将判断框的出口和选择结构的出口混淆。

(3) 结构内的每一部分都有机会被执行到。也就是说，对每一个框来说，都应当有一

条从入口到出口的路径通过它。图 2-20 中没有一条从入口到出口的路径通过 A 框。

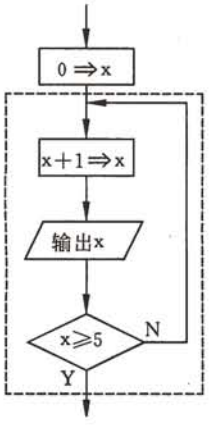


图 2-19

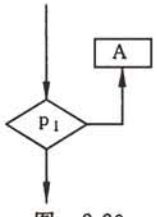


图 2-20

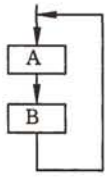


图 2-21

(4) 结构内不存在“死循环”(无终止的循环)。图 2-21 就是一个死循环。

已经证明,由以上 3 种基本结构顺序组成的算法结构,可以解决任何复杂的问题。由基本结构所构成的算法属于“结构化”的算法,它不存在无规律的转向,只在本基本结构内才允许存在分支和向前或向后的跳转。

其实,基本结构并不一定只限于上面 3 种,只要具有上述 4 个特点的都可以作为基本结构。人们可以自己定义基本结构,并由这些基本结构组成结构化程序。例如,也可以将图 2-22 和图 2-23 这样的结构定义为基本结构。图 2-23 所示的是一个多分支选择结构,根据给定的表达式的值决定执行哪一个框。图 2-22 和图 2-23 虚线框内的结构也只有一个入口和一个出口,并且具有上述全部的 4 个特点。由它们构成的算法结构也是结构化的算法。但是,可以认为像图 2-22 和图 2-23 那样的结构是由 3 种基本结构派生出来的。因此,人们普遍认为最基本的是本节介绍的 3 种基本结构。

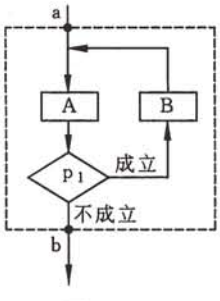


图 2-22

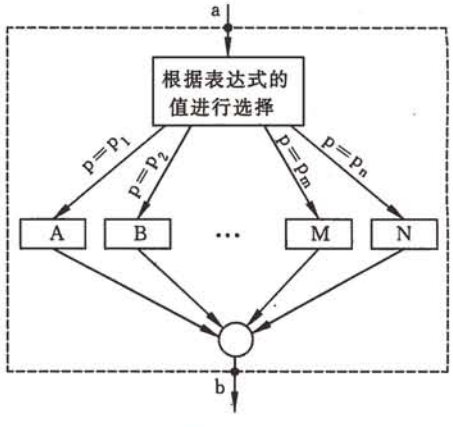


图 2-23

2.4.4 用 N-S 流程图表示算法

既然用基本结构的顺序组合可以表示任何复杂的算法结构，那么，基本结构之间的流程线就属多余的了。

1973 年美国学者 I. Nassi 和 B. Shneiderman 提出了一种新的流程图形式。在这种流程图中，完全去掉了带箭头的流程线。全部算法写在一个矩形框内，在该框内还可以包含其他的从属于它的框，或者说，由一些基本的框组成一个大的框。这种流程图又称 N-S 结构化流程图(N 和 S 是两位美国学者的英文姓氏的首字母)。这种流程图适于结构化程序设计，因而很受欢迎。

N-S 流程图用以下的流程图符号。

(1) 顺序结构。顺序结构用图 2-24 形式表示。A 和 B 两个框组成一个顺序结构。

(2) 选择结构。选择结构用图 2-25 表示。它与图 2-15 相应。当 p 条件成立时执行 A 操作，p 不成立则执行 B 操作。注意：图 2-25 是一个整体，代表一个基本结构。

(3) 循环结构。当型循环结构用图 2-26 形式表示。图 2-26 表示当 p_1 条件成立时反复执行 A 操作，直到 p_1 条件不成立为止。

图 2-24

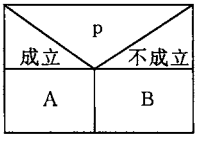


图 2-25

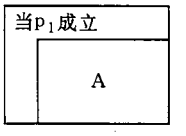


图 2-26

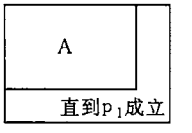


图 2-27

直到型循环结构用图 2-27 形式表示。

在初学时，为清楚起见，可如图 2-26 和图 2-27 那样，写明“当 p_1”或“直到 p_2”，待熟练之后，可以不写“当”和“直到”字样，只写“p_1”和“p_2”。从图的形状即可知道是当型或直到型。

用以上 3 种 N-S 流程图中的基本框可以组成复杂的 N-S 流程图，以表示算法。

应当说明，在图 2-24、图 2-25、图 2-26、图 2-27 中的 A 框或 B 框，可以是一个简单的操作(如读入数据或打印输出等)，也可以是 3 个基本结构之一。例如，图 2-24 所示的顺序结构，其中的 A 框可以又是一个选择结构，B 框可以又是一个循环结构。如图 2-28 所示那样，由 A 和 B 这两个基本结构组成一个顺序结构。

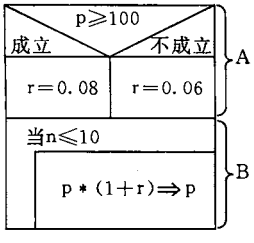


图 2-28

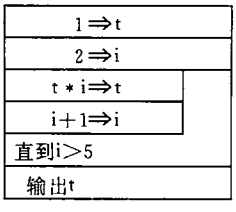


图 2-29

通过下面的几个例子，读者可以了解如何用 N-S 流程图表示算法。

例 2.11 将例 2.1 的求 5！算法用 N-S 图表示。见图 2-29，它和图 2-7 对应。

例 2.12 将例 2.2 的算法用 N-S 图表示。将 50 名学生中成绩高于 80 分的学号和成绩输出来。见图 2-30 和图 2-31，它和图 2-8 和图 2-9 对应。

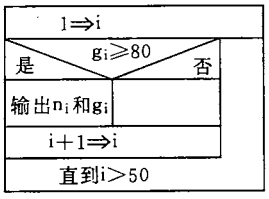


图 2-30

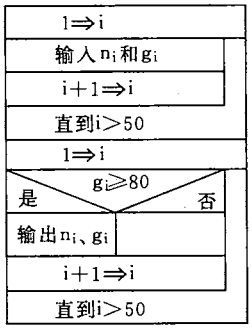


图 2-31

例 2.13 将例 2.3 判定闰年的算法用 N-S 图表示。见图 2-32，它和图 2-10 对应。

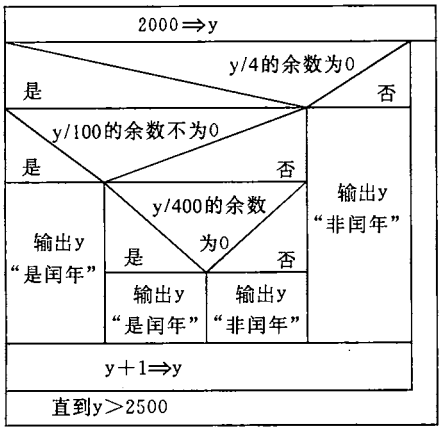


图 2-32

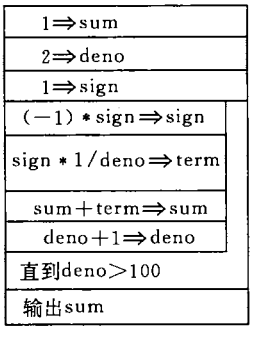


图 2-33

例 2.14 将例 2.4 的算法用 N-S 图表示。求 $1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\cdots+\frac{1}{99}-\frac{1}{100}$。见图 2-33，它和图 2-11 对应，只是最后加了一个“输出 sum”框。

例 2.15 将例 2.5 判别素数的算法用 N-S 流程图表示。

由图 2-12 可以看出，这个流程图不是由 3 种基本结构组成的。图中中间那个循环部分有两个出口（一个从第二个判断框下面出口，另一个在第一个判断框右边出口），不符合基本结构的特点。由于不能分解为 3 种基本结构，就无法直接用 N-S 流程图的 3 种基本结构的符号来表示。因此，应当先对图 2-12 作必要的变换。要将第一个判断框（“r=0”）的两个出口汇合在一点，以解决两个出口问题。当 r=0 时不直接输出 n“不是素数”，而只设一个标志值（变量 w），它的初始状态为 w=0。当 r=0 时（表示 n 为非素数）使 w 变

成 1。如果 r≠0 则保持 w=0。w 的作用如同一个开关一样，有两个工作状况：w=0 和 w=1。w=1 表示 n 为非素数。如果最终 w=0，则表示 n 为素数。变量 w 如同一个开关一样可以从一个状态转换到另一状态。然后将“1⇒w”框的出口线改为指向第二个判断框，同时将第二个判断框中的条件改为“$i\leqslant\sqrt{n}$和 w=0”，即只有当“$i\leqslant\sqrt{n}$”和“w=0”两个条件都满足时才继续执行循环。如果出现 $i>\sqrt{n}$或 w≠0 之一，都不会继续执行循环(见图 2-34)。如果在某一次 r=0，则应执行 1⇒w，然后，由第二个判断框判断为“条件不成立”，接着执行图下部的选择结构。此时，w≠0，表示 n 不是素数，应输出 n 不是素数的信息。如果 w=0，则表示在上面的每次循环中，n 都不能被每一个 i 整除，所以 n 是素数，故输出 n 是素数的信息。

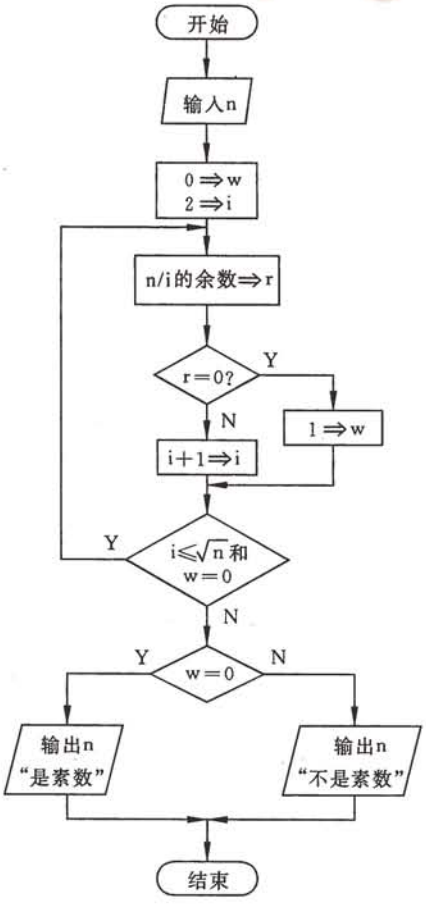


图 2-34

图 2-34 已由 3 种基本结构组成。可以用 N-S 图表示此算法，见图 2-35。注意，图 2-34 直到型循环的判断条件为：“直到 $i>\sqrt{n}$或 w≠0”，即只要“$i>\sqrt{n}$”或“w≠0”之一成立，就不再继续执行循环。对此务请读者弄清楚。

通过以上几个例子，可以看出用 N-S 图表示算法的优点。它比文字描述直观、形象、易于理解；比传统流程图紧凑易画，尤其是它废除了流程线，整个算法结构是由各个基本结构按顺序组成的，N-S 流程图中的上下顺序就是执行时的顺序，也就是图中位置在上面的先执行，位置在下面的后执行。写算法和看算法只需从上到下进行就可以了，十分方便。用 N-S 图表示的算法都是结构化的算法(它不可能出现流程无规律的跳转，而只能自上而下地顺序

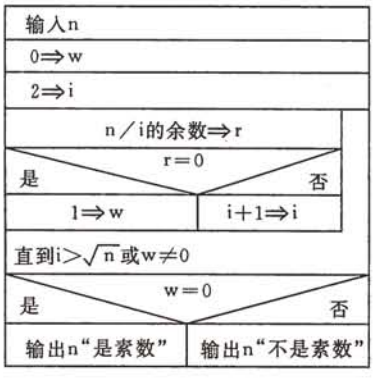


图 2-35

执行)。

归纳起来可知:一个结构化的算法是由一些基本结构顺序组成的;在基本结构之间不存在向前或向后的跳转,流程的转移只存在于一个基本结构范围之内(如循环中流程的跳转);一个非结构化的算法(如图 2-12)可以用一个等价的结构化算法(如图 2-35)代替,其功能不变。如果一个算法不能分解为若干个基本结构,则它必然不是一个结构化的算法。

N-S 图如同一个多层的盒子,又称盒图(box diagram)。

2.4.5 用伪代码表示算法

用传统的流程图和 N-S 图表示算法直观易懂,但画起来比较费事,在设计一个算法时,可能要反复修改,而修改流程图是比较麻烦的。因此,流程图适宜于表示一个算法,但在设计算法过程中使用不是很理想(尤其是当算法比较复杂、需要反复修改时)。为了设计算法时方便,常用一种称为伪代码(pseudo code)的工具。

伪代码是用介于自然语言和计算机语言之间的文字和符号来描述算法。它如同一篇文章一样,自上而下地写下来。每一行(或几行)表示一个基本操作。它不用图形符号,因此书写方便,格式紧凑,也比较好懂,也便于向计算机语言算法(即程序)过渡。

例如,"打印 x 的绝对值"的算法可以用伪代码表示如下:

```
if x is positive then
  print x
else
  print -x
```

它好像一个英语句子一样好懂,在西方国家用得比较普遍。也可以用汉字伪代码。例如:

```
若 x 为正
  输出 x
否则
  输出-x
```

也可以中英文混用,例如:

```
if x 为正
  print x
else
  print -x
```

即将计算机语言中的关键字用英文表示,其他的可用汉字。总之,以便于书写和阅读为原则。用伪代码写算法并无固定的、严格的语法规则,只要把意思表达清楚,并且书写的格式要写成清晰易读的形式。

将例 2.1 到例 2.5 的算法用伪代码表示如下。

例 2.16 求 5!。用伪代码表示的算法如下:

```
开始
```

```
  置 t 的初值为 1
  置 i 的初值为 2
  当 i<=5,执行下面操作:
    使 t=t*i
    使 i=i+1
    {循环体到此结束}
  输出 t 的值
结束
```

也可以写成以下形式:

```
begin     /*算法开始*/
  1⇒t
  2⇒i
  while i≤5
    {t*i⇒t
     i+1⇒i
    }
  print t
end     /*算法结束*/
```

在本算法中采用当型循环(第 3 行到第 5 行是一个当型循环)。while 意思为"当",它表示当 i≤5 时执行循环体(花括号中两行)的操作。

例 2.17 输出 50 个学生中成绩高于 80 分者的学号和成绩。

用伪代码表示算法如下:

```
begin     /*算法开始*/
  1⇒i
  while i≤50
  {
   input ni and gi
   i+1⇒i
  }
  1⇒i
while i≤50
    {
    if gi≥80 print ni and gi
    i+1⇒i
    }
end     /*算法结束*/
```

例 2.18 输出 2000—2500 年每一年是否闰年。

用伪代码表示算法如下:

```
begin     /*算法开始*/
  2000⇒y
```

```
while y≤2500
  {
    if y能被4整除
      if y不能被100整除
        print y;"是闰年"
      else
        if y能被400整除
          print y;"闰年"
        else
          print y;"非闰年"
        end if
      end if
    else
      print y;"非闰年"
    end if
    y+1⇒y
  }
end     /*算法结束*/
```

（内层选择结构、中层选择结构、外层选择结构）

例 2.19 求 $1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\cdots+\frac{1}{99}-\frac{1}{100}$。

用伪代码表示的算法如下：

```
begin     /*算法开始*/
  1⇒sum
  2⇒deno
  1⇒sign
  while deno ≤ 100
    {
     (-1) * sign⇒sign
     sign * 1/deno⇒term
     sum+term⇒sum
     deno+1⇒deno
    }
  print sum
end     /*算法结束*/
```

从以上例子可以看到：伪代码书写格式比较自由，可以随手写下去，容易表达出设计者的思想。同时，用伪代码写的算法很容易修改，例如加一行或删一行，或将后面某一部分调到前面某一位置，都是很容易做到的。而这却是用流程图表示算法时所不便处理的。用伪代码很容易写出结构化的算法。例如上面几个例子都是结构化的算法。但是用伪代码写算法不如流程图直观，可能会出现逻辑上的错误（例如循环或选择结构的范围搞错等）。

上面介绍了常用的表示算法的几种方法，在程序设计中读者可以根据需要和习惯任

意选用。软件专业人员一般习惯使用伪代码，考虑到国内广大初学人员的情况，为便于理解，在本书中主要采用形象化的 N-S 图表示算法。但是，读者应对其他方法也有所了解，以便在阅读其他书刊时不致发生困难。

2.4.6 用计算机语言表示算法

要完成一件工作，包括设计算法和实现算法两个部分。例如，作曲家创作一首乐谱就是设计一个算法，但它仅仅是一个乐谱，并未变成音乐，而作曲家的目的是希望使人们听到悦耳动人的音乐。由演奏家按照乐谱的规定进行演奏，就是“实现算法”。在没有人实现它时，乐谱是不会自动发声的。一个菜谱是一个算法，厨师炒菜就是在实现这个算法。设计算法的目的是为了实现算法。因此，不仅要考虑如何设计一个算法，也要考虑如何实现一个算法。

至此，只讲述了描述算法，即用不同的形式来表示操作的步骤。而要得到运算结果，就必须实现算法。在例 2.1、例 2.6、例 2.11 和例 2.16 中用不同的形式表示了求 5！的算法，但是并没有真正求出 5！的值。实现算法的方式可能不止一种。例如对例 2.1 表示的算法可以用人工心算的方式实现而得到结果。也可以用笔算或算盘、计算器来求出结果，这都是实现算法。

由于最终的目的是用计算机解题，也就是要用计算机实现算法，而计算机是无法识别流程图和伪代码的，它只有用计算机语言编写的程序才能被计算机执行，因此在用流程图或伪代码描述一个算法后，还要将它转换成计算机语言程序。

用计算机语言表示算法必须严格遵循所用的语言的语法规则，这是和伪代码不同的。下面将前面介绍过的算法用 C 语言表示。

例 2.20 将例 2.16 表示的算法(求 5!)用 C 语言表示。

```
#include <stdio.h>
void main( )
{
  int i,t;
  t=1;
  i=2;
  while(i<=5)
    {
     t=t*i;
     i=i+1;
    }
  printf("%d\n",t);
}
```

例 2.21 将例 2.19 表示的算法(求多项式的值)用 C 语言表示。

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

```
    int sign=1;
    float deno=2.0, sum=1.0, term;
    while (deno<=100)
      {
        sign=-sign;
        term=sign/deno;
        sum=sum+term;
        deno=deno+1;
      }
    printf("%f\n",sum);
}
```

在此，不打算仔细介绍以上程序的细节，读者只需大体看懂它即可。在以后各章中会详细介绍C语言有关的使用规则。

应当强调说明的是，写出了C程序，仍然只是描述了算法，并未实现算法。只有运行程序才是实现算法。应该说，用计算机语言表示的算法是计算机能够执行的算法。

2.5 结构化程序设计方法

前面介绍了结构化的算法和3种基本结构。一个结构化程序就是用高级语言表示的结构化算法。用3种基本结构组成的程序必然是结构化的程序，这种程序便于编写、便于阅读、便于修改和维护。这就减少了程序出错的机会，提高了程序的可靠性，保证了程序的质量。

结构化程序设计强调程序设计风格和程序结构的规范化，提倡清晰的结构。怎样才能得到一个结构化的程序呢？如果面临一个复杂的问题，是难于一下子写出一个层次分明、结构清晰、算法正确的程序的。结构化程序设计方法的基本思路是：把一个复杂问题的求解过程分阶段进行，每个阶段处理的问题都控制在人们容易理解和处理的范围内。

具体说，采取以下方法来保证得到结构化的程序：

(1) 自顶向下；

(2) 逐步细化；

(3) 模块化设计；

(4) 结构化编码。

在接受一个任务后应怎样着手进行呢？有两种不同的方法：一种是自顶向下，逐步细化；一种是自下而上，逐步积累。以写文章为例来说明这个问题。有的人胸有全局，先设想好整个文章分成哪几个部分，然后再进一步考虑每一部分分成哪几节，每一节分成哪几段，每一段应包含什么内容，如图2-36示意。

用这种方法逐步分解，直到作者认为可以直接将各小段表达为文字语句为止。这种方法就叫做“自顶向下，逐步细化”。

另有些人写文章时不拟提纲，如同写信一样提起笔就写，想到哪里就写到哪里，直到他认为把想写的内容都写出来了为止。这种方法叫做自下而上，逐步积累。

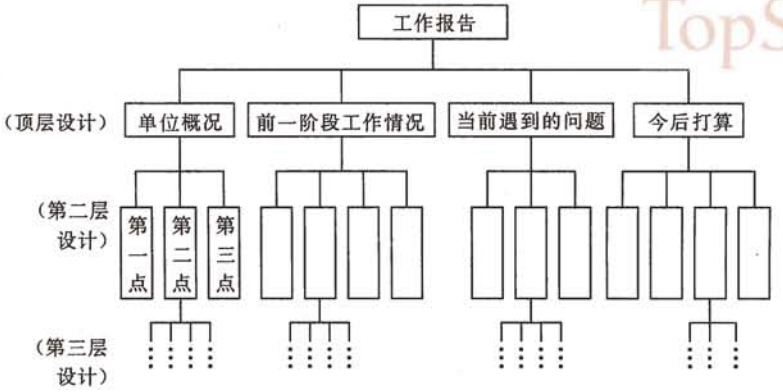


图 2-36

显然，用第一种方法考虑周全，结构清晰，层次分明，作者容易写，读者容易看。如果发现某一部分中有一段内容不妥，需要修改，只需找出该部分，修改有关段落即可，与其他部分无关。提倡用这种方法设计程序。这就是用工程的方法设计程序。

设计房屋就是用自顶向下、逐步细化的方法。先进行整体规划，然后确定建筑物方案，再进行各部分的设计，最后进行细节的设计（如门窗、楼道等），而决不会在没有整体方案之前先设计楼道和厕所。而在完成设计，有了图纸之后，在施工阶段则是自下而上地实施的，用一砖一瓦先实现一个局部，然后由各部分组成一个建筑物。

应当掌握自顶向下、逐步细化的设计方法。这种设计方法的过程是将问题求解由抽象逐步具体化的过程。如图 2-36 所示，最开始拿到的题目是作“工作报告”，这是一个很笼统而抽象的任务，经过初步考虑之后把它分成 4 个大的部分。这就比刚才具体一些了，但还不够具体。这一步只是粗略地划分，称为“顶层设计”。然后一步一步细化，依次称为第二层 、第三层设计，直到不需要细分为止。

用这种方法便于验证算法的正确性，在向下一层展开之前应仔细检查本层设计是否正确，只有上一层是正确的才能向下细化。如果每一层设计都没有问题，则整个算法就是正确的。由于每一层向下细化时都不太复杂，因此容易保证整个算法的正确性。检查时也是由上而下逐层检查，这样做，思路清楚，有条不紊地一步一步地进行，既严谨又方便。

在程序设计中常采用模块设计的方法，尤其当程序比较复杂时，更有必要。在拿到一个程序模块（实际上是程序模块的任务书）以后，根据程序模块的功能将它划分为若干个子模块，如果这些子模块的规模还嫌大，可以再划分为更小的模块。这个过程采用自顶向下的方法来实现。

程序中的子模块在 C 语言中通常用函数来实现（有关函数的概念将在第 8 章中介绍）。

程序中的子模块一般不超过 50 行，即输出时不超过一页，这样的规模便于组织，也便于阅读。划分子模块时应注意模块的独立性，即使用一个模块完成一项功能，耦合性愈少愈好。模块化设计的思想实际上是一种“分而治之”的思想，把一个大任务分为若干个子任务，每一个子任务就相对简单了。

结构化程序设计方法用来解决人脑思维能力的局限性和被处理问题的复杂性之间的矛盾。

在设计好一个结构化的算法之后，还要善于进行结构化编码（coding）。所谓编码就

是将已设计好的算法用计算机语言来表示，即根据已经细化的算法正确地写出计算机程序。结构化的语言（如 Pascal、C、QBASIC 等）都有与 3 种基本结构对应的语句，进行结构化编程序是不困难的。

本章的内容是十分重要的，是学习后面各章的基础。学习程序设计的目的不只是学习一种特定的语言，而是学习进行程序设计的一般方法。掌握了算法就是掌握了程序设计的灵魂，再学习有关的计算机语言的知识，就能够顺利地编写出任何一种语言的程序。脱离具体的语言去学习程序设计是困难的。但是，学习语言只是为了设计程序，它本身决不是目的。世界上高级语言有许多种，每种语言也都在不断发展，因而千万不能拘泥于一种具体的语言，而应当能举一反三。如前所述，关键是设计算法。有了正确的算法，用任何语言进行编码都不应当有什么困难。在本章中只是初步介绍了有关算法的基本知识，并没有深入介绍如何设计各种类型的算法。在以后各章中将结合程序实例陆续介绍有关的算法。

习　题

2.1　什么是算法？试从日常生活中找 3 个例子，描述它们的算法。

2.2　什么叫结构化的算法？为什么要提倡结构化的算法？

2.3　试述 3 种基本结构的特点，你能否自己另外设计两种基本结构（要符合基本结构的特点）。

2.4　用传统流程图表示求解以下问题的算法。

（1）有两个瓶子 A 和 B，分别盛放醋和酱油，要求将它们互换（即 A 瓶原来盛醋，现改盛酱油，B 瓶则相反）。

（2）依次将 10 个数输入，要求将其中最大的数输出。

（3）有 3 个数 a、b、c，要求按大小顺序把它们输出。

（4）求 $1+2+3+\cdots+100$。

（5）判断一个数 n 能否同时被 3 和 5 整除。

（6）将 100～200 之间的素数输出。

（7）求两个数 m 和 n 的最大公约数。

（8）求方程式 $ax^2+bx+c=0$ 的根。分别考虑：①有两个不等的实根；②有两个相等的实根；

2.5　用 N-S 图表示 2.4 题中各题的算法。

2.6　用伪代码表示 2.4 题中各题的算法。

2.7　什么叫结构化程序设计？它的主要内容是什么？

2.8　用自顶向下、逐步细化的方法进行以下算法的设计：

（1）输出 1900—2000 年中是闰年的年份，符合下面两个条件之一的年份是闰年。①能被 4 整除但不能被 100 整除；②能被 100 整除且能被 400 整除。

（2）求 $ax^2+bx+c=0$ 的根。分别考虑 $D=b^2-4ac$ 大于 0、等于 0 和小于 0 这 3 种情况。

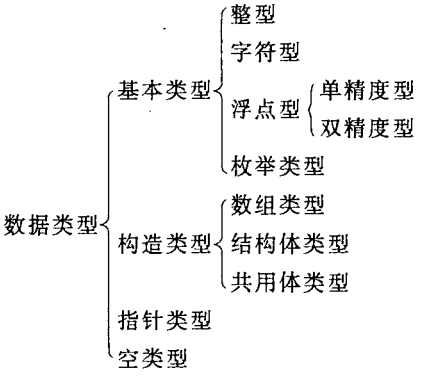
（3）输入 10 个数，输出其中最大的一个数。

第3章　数据类型、运算符与表达式

3.1　C语言的数据类型

在第2章中曾经提到过：算法处理的对象是数据，而数据是以某种特定的形式存在的（例如整数、实数、字符等形式）。不同的数据之间往往还存在某些联系（例如由若干个整数组成一个整数数组）。所谓数据结构指的是数据的组织形式。例如，数组就是一种数据结构。不同的计算机语言所允许定义和使用的数据结构是不同的。例如，C语言提供了“结构体”这样一种数据结构，而FORTRAN语言就不提供这种数据结构。处理同一类问题，如果数据结构不同，算法也会不同。例如：对10个整数排序和对由10个整数构成的数组排序的算法是不同的。因此，在考虑算法时，必须注意数据结构。应当综合考虑算法和数据结构，选择最佳的数据结构和算法。

C语言提供了以下一些数据类型，由这些数据类型可以构造出不同的数据结构。



在程序中对用到的所有数据都必须指定其数据类型。数据有常量与变量之分，它们分别属于以上这些类型。例如整型数据包括整型常量和整型变量。

利用以上数据类型还可以构成更复杂的数据结构。例如利用指针和结构体类型可以构成表、树、栈等复杂的数据结构。

在本章中主要介绍基本数据类型。

3.2　常量与变量

3.2.1　常量和符号常量

在程序运行过程中，其值不能被改变的量称为常量。常量区分为不同的类型，如12、0、－3为整型常量，4.6、－1.23为实型常量，'a'、'd'为字符常量。常量一般从其字面形式

即可判别。这种常量称为字面常量或直接常量。

也可以用一个标识符代表一个常量，如例 3.1。

例 3.1 符号常量的使用。

```
#define PRICE 30
#include <stdio.h>
void main ( )
{
  int num, total;
  num=10;
  total=num * PRICE;
  printf("total=%d\n",total);
}
```

程序中用 #define 命令行定义 PRICE 代表常量 30，此后凡在本程序中出现的 PRICE 都代表 30，可以和常量一样进行运算，程序运行结果为

```
total=300
```

有关 #define 命令行的详细用法在第 9 章中将详细介绍。

这种用一个标识符代表一个常量的符号，称为符号常量，即以标识符形式出现的常量。请注意符号常量与变量不同，符号常量的值在其作用域(在本例中为主函数)内不能改变，也不能再被赋值。如果再用赋值语句给 PRICE 赋值是错误的。

```
PRICE=40;        /* 错误，不能给符号常量赋值 */
```

习惯上，符号常量名用大写，变量名用小写，以示区别。

使用符号常量的好处如下。

(1) 含义清楚。如例 3.1 的程序中，看程序时从 PRICE 就可知道它代表价格。因此定义符号常量名时应考虑“见名知意”。在一个规范的程序中不提倡使用很多的常数，如：sum = 15 * 30 * 23.5 * 43，在检查程序时搞不清各个常数究竟代表什么。应尽量使用“见名知意”的变量名和符号常量。

(2) 在需要改变一个常量时能做到“一改全改”。例如在程序中多处用到某物品的价格，如果价格用常数表示，则在价格调整时，就需要在程序中作多处修改，若用符号常量 PRICE 代表价格，只需改动一处即可。如例 3.1 中的“#define PRICE 30”改为

```
#define PRICE 35
```

则在程序中所有以 PRICE 代表的价格就会一律自动改为 35。

3.2.2 变量

变量代表内存中具有特定属性的一个存储单元，它用来存放数据，也就是变量的值，在程序运行期间，这些值是可以改变的。一个变量应该有一个名字，以便被引用。请注意区分变量名和变量值这两个不同的概念，见图 3-1。变量名实际上是以一个名字对应，代

表一个地址。

在对程序编译连接时由编译系统给每一个变量名分配对应的内存地址。从变量中取值，实际上是通过变量名找到相应的内存地址，从该存储单元中读取数据。

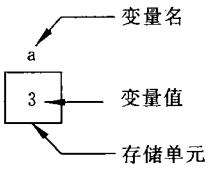


图 3-1

和其他高级语言一样，在C语言中用来对变量、符号常量、函数、数组、类型等数据对象命名的有效字符序列统称为标识符(identifier)。简单地说，标识符就是一个名字。

C语言规定标识符只能由字母、数字和下划线3种字符组成，且第一个字符必须为字母或下划线。下面列出的是合法的标识符，可以作为变量名：

sum, average, _total, Class, day, month, Student_name, tan, lotus_1_2_3, BASIC, li_ling

下面是不合法的标识符和变量名：

M. D. John, ¥123, #33, 3D64, a>b

注意，编译系统将大写字母和小写字母认为是两个不同的字符。因此，sum和SUM是两个不同的变量名，同样，Class和class也是两个不同的变量名。一般，变量名用小写字母表示，与人们日常习惯一致，以增加可读性。

ANSI C标准没有规定标识符的长度(字符个数)，但各个C编译系统都有自己的规定。有的系统(如IBM-PC的MS C)取8个字符，假如程序中出现的变量名长度大于8个字符，则只有前面8个字符有效，后面的不被识别。例如，有两个变量：student-name和student-number，由于二者的前8个字符相同，系统认为这两个变量是一回事而不加区别。可以将它们改为：stud-name和stud-num，以使之区别。Turbo C则允许变量名有32个字符。因此，在写程序时应了解所用系统对标识符长度的规定，以免出现上面的混淆，这种错误并不反映在编译过程中(即语法无错误)，但运行结果不对。为了程序的可移植性(即在甲计算机上运行的程序可以基本上不加修改，就能移到乙计算机上运行)以及阅读程序的方便，建议变量名的长度最好不要超过8个字符。

如前所述，在选择变量名和其他标识符时，应注意做到“见名知意”，即选有含义的英文单词(或其缩写)作标识符，如count、day、month、class、total、country等，除了数值计算程序外，一般不要用代数符号(如a、b、c、x1、y1等)作变量名，以增加程序的可读性。这是结构化程序的一个特征。本书在一些简单的举例中，为方便起见，仍用单字符的变量名(如a、b、c等)，请读者注意不要在其他所有程序中都如此。

在C语言中，要求对所有用到的变量作强制定义，也就是“先定义，后使用”，如例1.2、例1.3那样。这样做的目的如下。

(1) 凡未被事先定义的，系统不把它认作变量名，这就能保证程序中变量名使用得正确。例如，如果在声明部分有语句：

```
int student;
```

而在执行语句中错写成stadent。例如：

```
stadent=30;
```

在编译时检查出 stadent 未经定义，不作为变量名，因此输出“Undefined symbol stadent in function main”的信息，提醒用户检查错误，避免使用变量名时出错。

(2) 每一个变量被指定为一个确定类型，在编译时就能为其分配相应的存储单元。如指定 a、b 为 int 型，Turbo C 编译系统为 a 和 b 各分配两个字节，并按整数方式存储数据。

(3) 指定每一变量属于一个类型，这就便于在编译时据此检查在程序中要求对该变量进行的运算是否合法。例如，整型变量 a 和 b，可以进行求余运算：

```
a%b
```

%是“求余”(见 3.5 节)，得到 a/b 的整余数。如果将 a、b 指定为实型变量，则不允许进行“求余”运算，在编译时会给出有关“出错信息”。

下面各节分别介绍整型、实型(浮点型)和字符型数据。

3.3 整型数据

3.3.1 整型常量的表示方法

整型常量即整常数。在 C 语言中，整常数可用以下 3 种形式表示。

(1) 十进制整数，如 123、−456、4。

(2) 八进制整数，以 0 开头的数是八进制数。如 0123 表示八进制数 123，即$(123)_8$，其值为 $1\times8^2+2\times8^1+3\times8^0$，等于十进制数 83，−011 表示八进制数 −11，即十进制数 −9。

(3) 十六进制整数。以 0x 开头的数是十六进制数。如 0x123，代表十六进制数 123，即$(123)_{16}=1\times16^2+2\times16^1+3\times16^0=256+32+3=291$；−0x12 等于十进制数 −18。

3.3.2 整型变量

1. 整型数据在内存中的存放形式

数据在内存中是以二进制形式存放的。如果定义了一个整型变量 i：

```
int i;              /* 定义为整型变量 */
i=10;               /* 给 i 赋以整数 10 */
```

十进制数 10 的二进制形式为 1010，Turbo C 2.0 和 Turbo C ++ 3.0 为一个整型变量在内存中分配 2 个字节的存储单元(不同的编译系统为整型数据分配的字节数是不相同的，Visual C ++ 6.0 则分配 4 个字节。本书在举例时一般假定整型变量在内存中占 2 个字节)。图 3-2(a)是数据存放的示意图。图 3-2(b)是数据在内存中实际存放的情况。

实际上，数值是以补码(complement) 表示的。一个正整数的补码和该数的原码(即该数的二进制形式)相同。图 3-2(b) 就是用补码形式表示的。如果数值是负的，在内存中如何用补码形式表示呢？求负数的补码的方法是：将该数的绝对值的二进制形式，按位

取反再加 1。例如求－10 的补码的方法是：

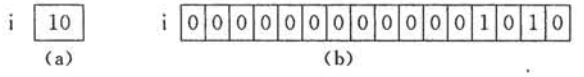


图 3-2　数据在内存中的存放

① 取－10 的绝对值 10；

② 10 的绝对值的二进制形式为 1010；

③ 对 1010 取反得 1111111111110101(一个整数占 16 位)；

④ 再加 1 得 1111111111110110，见图 3-3。

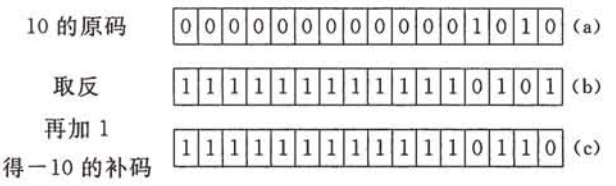


图　3-3

可知：在存放整数的存储单元中，最左面的一位是表示符号的，该位为 0，表示数值为正；该位为 1 则表示数值为负。

关于补码的知识不属于本书的范围，但学习 C 语言的读者应该比学习其他高级语言对数据在内存中的表示形式有更多的了解，这样才能理解不同类型数据间转换的规律。在本章稍后的叙述中还要接触到这方面的问题。

2. 整型变量的分类

整型变量的基本类型符为 int。可以根据数值的范围将变量定义为基本整型、短整型或长整型。在 int 之前可以根据需要分别加上修饰符(modifier)：short(短型)或 long(长型)。因此有以下 3 类整型变量：

(1) 基本整型，以 int 表示。

(2) 短整型，以 short int 表示，或以 short 表示。

(3) 长整型，以 long int 表示，或以 long 表示。

一个 int 型的变量值的范围为 $-2^{15}\sim(2^{15}-1)$，即－32768～32767。在实际应用中，变量的值常常是正的(如学号、库存量、年龄、存款额等)。为了充分利用变量的值的范围，此时可以将变量定义为“无符号”类型。对以上 3 类都可以加上修饰符 unsigned，以指定是“无符号数”。如果加上修饰符 signed，则指定是“有符号数”。如果既不指定为 signed，也不指定为 unsigned，则隐含为有符号(signed)。实际上 signed 是完全可以不写的。归纳起来，可以用以下 6 种整型变量。即：

- 有符号基本整型 [signed] int；
- 无符号基本整型 unsigned int；

- 有符号短整型 [signed] short [int];
- 无符号短整型 unsigned short [int];
- 有符号长整型 [signed] long [int];
- 无符号长整型 unsigned long [int]。

上面的方括号表示其中的内容是可选的,既可以有,也可以没有。

如果不指定 unsigned 或指定 signed,则存储单元中最高位代表符号(0 为正,1 为负)。如果指定 unsigned,为无符号型,存储单元中全部二进位(bit)用作存放数本身,而不包括符号。无符号型变量只能存放不带符号的整数,如 123、4687 等,而不能存放负数,如-123、-3。一个无符号整型变量中可以存放的正数的范围比一般整型变量中正数的范围扩大一倍。如果在程序中定义 a 和 b 两个变量:

```
int a;
unsigned int b;
```

则变量 a 的数值范围为-32768～32767,而变量 b 的数值范围为 0～65535。图 3-4(a) 表示有符号整型变量 a 的最大值(32767),图 3-4(b) 表示无符号整型变量 b 的最大值(65535)。

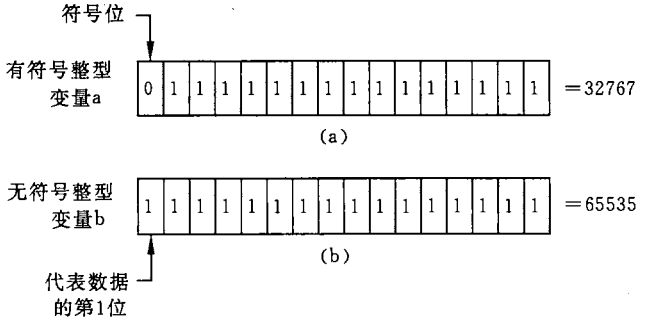


图 3-4

C 语言没有具体规定以上各类数据所占内存的字节数,只要求 long 型数据长度不短于 int 型,short 型不长于 int 型。具体如何实现,由各计算机系统自行决定。如在 Turbo C++ 中,int 型和 short 型数据都是 16 位(指二进制位,下同),而 long 型数据是 32 位。有的 C 语言编译系统则给 short 型数据分配 16 位,而 int 和 long 型数据都是 32 位。一般以一个机器字(word)存放一个 int 数据。前一时期,微型计算机的字长一般为 16 位,故以 16 位存放一个整数,但整数的范围太小,往往不够用,因此将 long 型数据定为 32 位。而有的计算机的字长为 32 位,以 32 位存放一个整数,范围可达正负 21 亿,已足够用了,不必再将 long 型定为 64 位。所以将 int 型和 long 型都定为 32 位。通常的做法是:把 long 定为 32 位,把 short 定为 16 位,而 int 可以是 16 位,也可以是 32 位。

表 3-1 列出用 Turbo C/Turbo C++ 时整数类型的有关数据。

表 3-1　整数类型的有关数据

类　　型	比特(位)数	取 值 范 围
[signed] int	16	−32768～32767 即 $-2^{15}\sim(2^{15}-1)$
Unsigned int	16	0～65535　即 $0\sim(2^{16}-1)$
[signed] short [int]	16	−32768～32767　即 $-2^{15}\sim(2^{15}-1)$
Unsigned short[int]	16	0～65535　即 $0\sim(2^{16}-1)$
long [int]	32	−2147483648～2147483647　即 $-2^{31}\sim(2^{31}-1)$
unsigned long [int]	32	0～4294967295　即 $0\sim(2^{32}-1)$

如前指出，方括号内的部分是可以省写的。例如 signed short int 与 short 等价。尤其是 signed 是完全多余的，一般都不写 signed。

注意：用不同的编译系统时，具体情况可能与表 3-1 有些差别，例如 Visual C++ 6.0 为整型数据分配 4 字节(32 位)，其取值范围为−2147483648～2147483647。

一个整数(以 13 为例)在存储单元中的存储情况如图 3-5 所示(假设使用的是 Turbo C)。

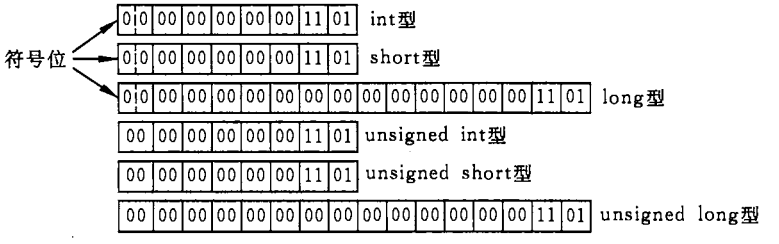


图　3-5

3. 整型变量的定义

前面已提到，C 语言程序中所有用到的变量都必须在程序中定义，即“强制类型定义”。例如：

```
int a,b;                        (指定变量 a、b 为整型)
unsigned short c,d;             (指定变量 c、d 为无符号短整型)
long e,f;                       (指定变量 e、f 为长整型)
```

对变量的定义一般是放在一个函数的开头部分的声明部分(也可以放在函数中某一段分程序内，但作用域只限它所在的分程序，这将在第 8 章介绍)。

例 3.2　整型变量的定义与使用。

```
#include <stdio.h>
void main()
{
  int a,b,c,d;                  /*指定 a、b、c、d 为整型变量*/
  unsigned u;                   /*指定 u 为无符号整型变量*/
  a=12;b=-24;u=10;
  c=a+u;d=b+u;
```

```
    printf("a+u=%d,b+u=%d\n",c,d);
}
```

运行结果为：

```
a+u=22,b+u=-14
```

可以看到不同种类的整型数据可以进行算术运算。在本例中是 int 型数据与 unsigned int 型数据进行相加相减运算(有关运算的规则在本章 3.7 节中介绍)。

4. 整型数据的溢出

在 Turbo C 和 Turbo C++ 中，一个 int 型的变量的最大允许值为 32767，如果再加 1，会出现什么情况？

例 3.3 整型数据的溢出(见图 3-6)。

```
#include <stdio.h>
void main()
{
  int a,b;
  a=32767;
  b=a+1;
  printf("%d,%d\n",a,b);
}
```

运行结果为：

```
32767,-32768
```

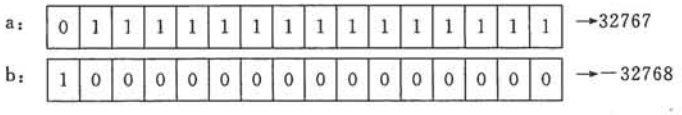


图 3-6

从图 3-6 可以看到：变量 a 的最高位为 0，后 15 位全为 1。加 1 后变成第 1 位为 1，后面 15 位全为 0。而它是－32768 的补码形式，所以输出变量 b 的值为－32768。请注意：一个整型变量只能容纳－32768～32767 范围内的数，无法表示大于 32767 或小于－32768 的数。遇到此情况就发生“溢出”。但运行时并不报错。它好像汽车里程表一样，达到最大值以后，又从最小值开始计数。所以，32767 加 1 得不到 32768，而得到－32768。这可能与程序编制者的原意不同。从这可以看到：C 语言的用法比较灵活，往往出现副作用，而系统又不给出“出错信息”，要靠程序员的细心和经验来保证结果的正确。将变量 b 改成 long 型就可得到预期结果 32768。

3.3.3 整型常量的类型

整型变量可分为 int、short int、long int 和 unsigned int、unsigned short、unsigned long 等类别，那么常量是否也有这些类别？在将一个整型常量赋值给上述几种类别的整

型变量时如何做到类型匹配？请注意以下几点(假定整型数据在内存中占 2 个字节)：

(1) 一个整数，如果其值在－32768～32767 范围内，认为它是 int 型，它可以赋值给 int 型和 long int 型变量。

(2) 一个整数，如果其值超过了上述范围，而在－2147483648～2147483647 范围内，则认为它是长整型。可以将它赋值给一个 long int 型变量。

(3) 如果所用的 C 语言版本(如 Turbo C)分配给 short int 与 int 型数据在内存中占据的长度相同，则它的表数范围与 int 型相同。因此一个 int 型的常量同时也是一个 short int 型常量，可以赋给 int 型或 short int 型变量。

(4) 一个整常量后面加一个字母 u 或 U，认为是 unsigned int 型，如 12345u 在内存中按 unsigned int 规定的方式存放(存储单元中最高位不作为符号位，而用来存储数据，见图 3-4(b))。如果写成－12345u，则先将－12345 转换成其补码 53191，然后按无符号数存储。

(5) 在一个整常量后面加一个字母 l 或 L，则认为是 long int 型常量，例如 123l、432L、0L 等，这往往用于函数调用中。如果函数的形参为 long int 型，则要求实参也为 long int 型。

3.4 浮点型数据

3.4.1 浮点型常量的表示方法

C 语言中的浮点数(floating point number) 就是平常所说的实数(real number)。浮点数有两种表示形式。

(1) 十进制小数形式。它由数字和小数点组成(注意必须有小数点)。0.123、123.、123.0、0.0 都是十进制小数形式。

(2) 指数形式。如 123e3 或 123E3 都代表 123×10^3。但注意字母 e(或 E)之前必须有数字，且 e 后面的指数必须为整数，如 e3、2.1e3.5、.e3、e 等都不是合法的指数形式。

一个浮点数可以有多种指数表示形式。例如 123.456 可以表示为：123.456e0、12.3456e1、1.23456e2、0.123456e3、0.0123456e4、0.00123456e5 等。其中的 1.23456e2 称为“规范化的指数形式”。即在字母 e(或 E)之前的小数部分中，小数点左边应有一位(且只能有一位)非零的数字。例如 2.3478e2、3.0999E5、6.46832e12 都属于规范化的指数形式，而 12.908e10、0.4578e3、756e0 则不属于规范化的指数形式。一个浮点数在用指数形式输出时，是按规范化的指数形式输出的。例如，若指定将实数 5689.65 按指数形式输出，输出的形式是 5.68965e+003，而不会是 0.568965e+004 或 56.8965e+002。

3.4.2 浮点型变量

1. 浮点型数据在内存中的存放形式

一个浮点型数据一般在内存中占 4 个字节(32 位)。与整型数据的存储方式不同，浮点型数据是按照指数形式存储的。系统把一个浮点型数据分成小数部分和指数部分，分

别存放。指数部分采用规范化的指数形式。实数 3.14159 在内存中的存放形式可以用图 3-7 示意。

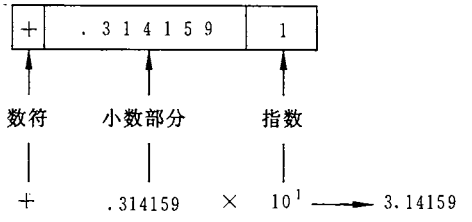


图 3-7

图 3-7 中是用十进制数来示意的，实际上在计算机中是用二进制数来表示小数部分以及用 2 的幂次来表示指数部分的。

在 4 个字节(32 位)中，究竟用多少位来表示小数部分，多少位来表示指数部分，标准并无具体规定，由各 C 语言编译系统自定。不少 C 语言编译系统以 24 位表示小数部分(包括符号)，以 8 位表示指数部分(包括指数的符号)。小数部分占的位(bit)数愈多，数的有效数字愈多，精度也就愈高。指数部分占的位数愈多，则能表示的数值范围愈大。

2. 浮点型变量的分类

浮点型变量分为单精度(float 型)、双精度(double 型)和长双精度(long double 型)3 类。在 Turbo C 中有关浮点型的数据见表 3-2。

表 3-2 浮点型数据

类 型	比特(位)数	有效数字	数值范围
float	32	6～7	$-3.4\times10^{-38}\sim3.4\times10^{38}$
double	64	15～16	$-1.7\times10^{-308}\sim1.7\times10^{308}$
long double	128	18～19	$-1.2\times10^{-4932}\sim1.2\times10^{4932}$

ANSI C 并未具体规定每种类型数据的长度、精度和数值范围。有的系统将 double 型所增加的 32 位全用于存放小数部分，这样可以增加数值的有效位数，减少舍入误差。有的系统则将所增加位(bit)的一部分用于存放指数部分，这样可以扩大数值的范围。表 3-2 列出的是 Turbo C、Turbo C++ 6.0、MS C 的情况，不同的系统会有差异。

对每一个浮点型变量都应在使用前加以定义。例如：

```
float x,y;          (指定 x、y 为单精度浮点数)
double z;           (指定 z 为双精度浮点数)
long double t;      (指定 t 为长双精度浮点数)
```

在初学阶段，对 long double 型用得较少，因此不作详细介绍。读者只要知道有此类型即可。

3. 浮点型数据的舍入误差

由于浮点型变量是由有限的存储单元组成的，因此能提供的有效数字总是有限的。在有效位以外的数字将被舍去。由此可能会产生一些误差，例如，a 加 20 的结果应该比 a 大。请分析下面的程序：

例 3.4 浮点型数据的舍入误差。

```
#include <stdio.h>
void main()
{
  float a,b;
  a = 123456.789e5;
  b = a + 20 ;
  printf("%f\n",b);
}
```

程序内 printf 函数中的“%f”是输出浮点数时指定的格式符，作用是指定该实数以小数形式输出。程序运行时输出 b 的值与 a 相等，原因是：a 的值比 20 大很多，a+20 的理论值应是 12345678920，而一个浮点型变量只能保证的有效数字是 7 位，后面的数字是无意义的，因此并不准确地表示该数。运行程序得到的 a 和 b 的值都是 12345678848.000000，可以看到，对这个数来说，前 8 位是准确的，后几位是不准确的，把 20 加在后几位上，是无意义的。应当避免将一个很大的数和一个很小的数直接相加或相减，否则就会“丢失”小的数。与此类似，用程序计算 1.0/3.0*3 的结果并不等于 1。

3.4.3 浮点型常量的类型

C 语言编译系统将浮点型常量作为双精度来处理。例如已定义一个浮点型变量 f，有如下语句：

```
f = 2.45678 * 4523.65;
```

系统先把 2.45678 和 4523.65 作为双精度数，然后进行相乘的运算，得到的乘积也是一个双精度数。最后取其前 7 位赋给浮点型变量 f。这样做可以使计算结果更精确。但是运算速度降低了。如果是在数的后面加字母 f 或 F(如 1.65f、654.87F)，这样编译系统就会把它们按单精度(32 位)处理。

一个浮点型常量可以赋给一个 float 型、double 型或 long double 型变量，根据变量的类型截取实型常量中相应的有效位数字。假如 a 已指定为单精度浮点型变量：

```
float a;
a=111111.111;
```

由于 float 型变量只能接收 7 位有效数字，因此最后两位小数不起作用。如果 a 改为 double 型，则能全部接收上述 9 位数字并存储在变量 a 中。

3.5 字符型数据

3.5.1 字符常量

C语言的字符常量是用单撇号括起来的一个字符。如'a'、'x'、'D'、'?'、'$'等都是字符常量。注意,'a'和'A'是不同的字符常量。

除了以上形式的字符常量外,C还允许用一种特殊形式的字符常量,就是以一个字符"\"开头的字符序列。例如,前面已经遇到过的,在printf函数中的"\n",它代表一个"换行"符。这是一种"控制字符",在屏幕上是不能显示的,在程序中也无法用一个一般形式的字符表示,只能采用特殊形式来表示。

常用的以"\"开头的特殊字符见表3-3。

表3-3 转义字符及其作用

字符形式	含　义	ASCII代码
\n	换行,将当前位置移到下一行开头	10
\t	水平制表(跳到下一个Tab位置)	9
\b	退格,将当前位置移到前一列	8
\r	回车,将当前位置移到本行开头	13
\f	换页,将当前位置移到下页开头	12
\\	代表一个反斜杠字符"\"	92
\'	代表一个单引号(撇号)字符	39
\"	代表一个双引号字符	34
\ddd	1到3位八进制数所代表的字符	
\xhh	1到2位十六进制数所代表的字符	

表3-3中列出的字符称为"转义字符",意思是将反斜杠"\"后面的字符转换成另外的意义。如"\n"中的"n"不代表字母n而作为"换行"符。

表3-3中倒数第二行是一个ASCII码(八进制数)表示一个字符,例如"\101"代表ASCII码(八进制数)为101的字符'A'。八进制数101相当于十进制数65,从附录A可以看到ASCII码(十进制数)为65的字符是大写字母'A'。"\012"代表八进制数12(即十进制数的10)的ASCII码所对应的字符"换行"符。用"\376"代表图形字符"■"。用表3-3中的方法可以表示任何可输出的字母字符、专用字符、图形字符和控制字符。请注意"\0"或"\000"是代表ASCII码为0的控制字符,即"空操作"字符。它常用在字符串中。

例3.5 转义字符的使用。

```
#include <stdio.h>
void main()
{
  printf("␣ab␣c\t␣de\rf\tg\n");
  printf("h\ti\b\bj␣k\n");
}
```

程序中没有设字符变量，用 printf 函数直接输出双引号内的各个字符。请注意其中的“转义字符”。第一个 printf 函数先在第一行左端开始输出“␣ab␣c”，然后遇到“\t”，它的作用是“跳格”，即跳到下一个“制表位置”，在所用系统中一个“制表区”占 8 列。“下一制表位置”从第 9 列开始，故在第 9～11 列上输出“␣de”。下面遇到“\r”，它代表“回车”（不换行），返回到本行最左端（第 1 列），输出字符“f”，然后遇“\t”再使当前输出位置移到第 9 列，输出“g”。下面是“\n”，作用是“使当前位置移到下一行的开头”。第二个 printf 函数先在第 1 列输出字符“h”，后面的“\t”使当前位置跳到第 9 列，输出字母“i”，然后当前位置应移到下一列（第 10 列）准备输出下一个字符。下面遇到两个“\b”，“\b”的作用是“退一格”，因此“\b\b”的作用是使当前位置回退到第 8 列，接着输出字符“jk”。

程序运行时输出以下结果：

```
fab␣c␣␣␣gde
h␣␣␣␣␣␣␣jik
```

注意在显示屏上最后看到的结果与上述输出的结果不同，是：

```
f␣␣␣␣␣␣␣␣gde
h␣␣␣␣␣␣␣j␣k
```

这是由于“\r”使当前位置回到本行开头，自此输出的字符（包括空格和跳格所经过的位置）将取代原来屏幕上该位置上显示的字符。所以原有的“␣ab␣c␣␣␣␣␣”被新的字符“f␣␣␣␣␣␣␣␣␣g”代替，其后的“de”未被新字符取代。换行后先输出“h␣␣␣␣␣␣␣␣␣i”，然后光标位置移到 i 右面一列处，再退两格后输出“␣j␣k”，j 后面的“␣”将原有的字符“i”取而代之。因此屏幕上看不到“i”。实际上，屏幕上完全按程序要求输出了全部的字符，只是因为在输出前面的字符后很快又输出后面的字符，在人们还未看清楚之前，新的已取代了旧的，所以误以为未输出应输出的字符。而在打印机输出时，不像显示屏那样会“抹掉”原字符，留下了不可磨灭的痕迹，它能真正反映输出的过程和结果。

3.5.2 字符变量

字符型变量用来存放字符常量，它只能放一个字符，不要以为在一个字符变量中可以放一个字符串（包括若干字符）。

字符变量的定义形式如下：

```
char c1,c2;
```

它表示 c1 和 c2 为字符型变量，各可以放一个字符，因此在本函数中可以用下面语句对 c1、c2 赋值：

```
c1='a';c2='b';
```

在所有的编译系统中都规定以一个字节来存放一个字符，或者说一个字符变量在内存中占一个字节。

3.5.3 字符数据在内存中的存储形式及其使用方法

将一个字符常量放到一个字符变量中，实际上并不是把该字符本身放到内存单元中去，而是将该字符的相应的 ASCII 代码放到存储单元中。例如字符'a'的 ASCII 代码为十进制数 97，'b'的 ASCII 代码为十进制数 98，在内存中变量 c1、c2 的值如图 3-8(a)所示。实际上是以二进制形式存放的，如图 3-8(b)所示。

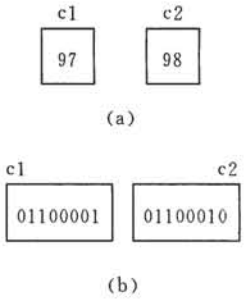


图 3-8

既然在内存中，字符数据以 ASCII 码存储，它的存储形式就与整数的存储形式类似。这样使字符型数据和整型数据之间可以通用。一个字符数据既可以以字符形式输出，也可以以整数形式输出。以字符形式输出时，需要先将存储单元中的 ASCII 码转换成相应字符，然后输出。以整数形式输出时，直接将 ASCII 码作为整数输出。也可以对字符数据进行算术运算，此时相当于对它们的 ASCII 码进行算术运算。

例 3.6 向字符变量赋予整数。

```
#include <stdio.h>
void main()
{
  char c1,c2;
  c1=97;
  c2=98;
  printf("%c %c\n",c1,c2);
  printf("%d %d\n",c1,c2);
}
```

c1、c2 被指定为字符变量。在第 5 和第 6 行中，将整数 97 和 98 分别赋给 c1 和 c2，它的作用相当于以下两个赋值语句：

```
c1='a';c2='b';
```

因为'a'和'b'的 ASCII 码为 97 和 98。在程序的第 5 和第 6 行是把 97 和 98 两个整数直接存放到 c1 和 c2 的内存单元中。而 c1='a'和 c2='b'则是先将字符'a'和'b'化成 ASCII 码 97 和 98，然后放到内存单元中。二者的作用和结果是相同的。第 7 行输出两个字符 a 和 b。"%c"是输出字符时使用的格式符。程序第 8 行输出两个整数 97 和 98。程序运行时输出结果如下：

```
a b
97 98
```

可以看到：字符型数据和整型数据是通用的。它们既可以用字符形式输出(用%c)，也可以用整数形式输出(用%d)，见图 3-9。但是应注意字符数据只占一个字节，它只能存放 0～255 范围内的整数。

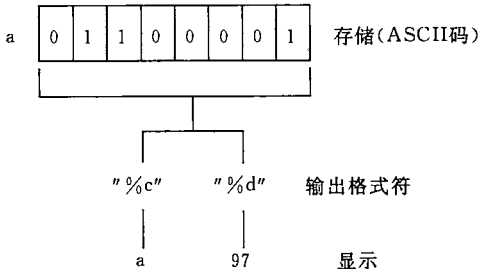


图 3-9

例 3.7 大小写字母的转换。

```
#include <stdio.h>
void main()
{
  char c1,c2;
  c1='a';
  c2='b';
  c1=c1-32;
  c2=c2-32;
  printf("%c %c",c1,c2);
}
```

运行结果为

```
A B
```

程序的作用是将两个小写字母 a 和 b 转换成大写字母 A 和 B。'a'的 ASCII 码为 97，而'A'为 65，'b'为 98，'B'为 66。从 ASCII 代码表中可以看到每一个小写字母比它相应的大写字母的 ASCII 码大 32。C 语言允许字符数据与整数直接进行算术运算。即'A'+32 会得到整数 97，'a'-32 会得到整数 65。

C 语言对字符数据做这种处理使程序设计时增大了自由度。例如对字符做各种转换就比较方便。

字符数据与整型数据可以互相赋值。例如：

```
int i;
char c;
i='a';
c=97;
```

是合法的。如果用格式符"%d"将 i 的值输出可得到 97。用"%c"输出 c 可得字符'a'。

如果在上面语句之后执行以下语句：

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

输出：

```
a,97
a,97
```

需要说明：有些系统(如 Turbo C)将字符变量定义为 signed char 型。其存储单元中的最高位作为符号位，它的取值范围是－128～127。如果在字符变量中存放一个 ASCII 码为 0～127 间的字符，由于字节中最高位为 0，因此用%d 输出字符变量时，输出的是一个正整数。如果在字符变量中存放一个 ASCII 码为 128～255 间的字符，由于在字节中最高位为 1，用%d 格式符输出时，就会得到一个负整数。例如：

```
char c=130;
printf("%d", c);
```

得到－126。如果不想按有符号处理，可以在程序中将字符变量定义为 unsigned char 类型，这时其取值范围是 0～255。signed char 和 unsigned char 的含义及用法与 signed int 和 unsigned int 相仿，但它只有一个字节。

3.5.4 字符串常量

前面已提到，字符常量是由一对单撇号括起来的单个字符。C 语言除了允许使用字符常量外，还允许使用字符串常量。字符串常量是一对双撇号括起来的字符序列。例如下面是合法的字符串常量：

```
"How do you do.", "CHINA","a","$123.45"
```

可以输出一个字符串，例如：

```
printf("How do you do.");
```

不要将字符常量与字符串常量混淆。'a'是字符常量，"a"是字符串常量，二者不同。假设 c 被指定为字符变量：

```
char c;
c='a';
```

是正确的。而

```
c="a";
```

是错误的。

```
c="CHINA";
```

也是错误的。不能把一个字符串常量赋给一个字符变量。

有人不能理解：'a'和"a"究竟有什么区别？C 规定：在每一个字符串常量的结尾加一个“字符串结束标志”，以便系统据此判断字符串是否结束。C 规定以字符'\0'作为字符串结束标志。'\0'是一个 ASCII 码为 0 的字符，从 ASCII 代码表中可以看到 ASCII 码为 0 的字符是“空操作字符”，即它不引起任何控制动作，也不是一个可显示的字符。如果有一

个字符串常量"CHINA"，实际上在内存中是：

C	H	I	N	A	\0

它占内存单元不是5个字符，而是6个字符，最后一个字符为'\0'。但在输出时不输出'\0'。例如printf("how do you do.")，从第一个字符开始逐个输出字符，直到遇到最后的'\0'字符，就知道字符串结束，停止输出。

注意，在写字符串时不必加'\0'，否则会画蛇添足。'\0'字符是系统自动加上的。字符串"a"实际上包含2个字符：'a'和'\0'，因此，想把它赋给只能容纳一个字符的字符变量c显然是不行的。

```
c="a";        /*错误,c是字符变量 */
```

在C语言中没有专门的字符串变量，如果想将一个字符串存放在变量中以便保存，必须使用字符数组，即用一个字符型数组来存放一个字符串，数组中每一个元素存放一个字符。这将在第7章中介绍。

3.6 变量赋初值

程序中常需要对一些变量预先设置初值。C语言允许在定义变量的同时使变量初始化。例如：

```
int a=3;              /* 指定a为整型变量,初值为3 */
float f=3.56;         /* 指定f为浮点型变量,初值为3.56 */
char c='a';           /* 指定c为字符变量,初值为'a' */
```

也可以使被定义的变量的一部分赋初值。例如：

```
int a,b,c=5;
```

表示指定a、b、c为整型变量，但只对c初始化，c的初值为5。

如果对几个变量赋予同一个初值，应写成：

```
int a=3,b=3,c=3;
```

表示a、b、c的初值都是3。不能写成：

```
int a=b=c=3;
```

初始化不是在编译阶段完成的(只有在第8章中介绍的静态存储变量和外部变量的初始化是在编译阶段完成的)，而是在程序运行时执行本函数时赋初值的，相当于有一个赋值语句。例如：

```
int a=3;
```

相当于：

```
int a;          /* 指定 a 为整型变量 */
a=3;            /* 赋值语句,将 3 赋给 a */
```

又如：

```
int a,b,c=5;
```

相当于：

```
int a,b,c;      /* 指定 a、b、c 为整型变量 */
c=5;            /* 将 5 赋给 c */
```

3.7 各类数值型数据间的混合运算

整型(包括 int、short、long)和浮点型(包括 float、double)可以混合运算。前已述及，字符型数据可以与整型通用，因此，整型、浮点型、字符型数据间可以混合运算。例如：

```
10+'a'+1.5-8765.1234 *'b'
```

是合法的。在进行运算时，不同类型的数据要先转换成同一类型，然后进行运算。转换的规则按图 3-10 所示。

图 3-10 中横向向左的箭头表示必定的转换，如字符数据必定先转换为整数，short 型转换为 int 型，float 型数据在运算时一律先转换成双精度型，以提高运算精度(即使是两个 float 型数据相加，也先都化成 double 型，然后再相加)。

纵向的箭头表示当运算对象为不同类型时转换的方向。例如 int 型与 double 型数据进行运算，先将 int 型的数据转换成 double 型，然后在两个同类型(double 型)数据间进行运算，结果为 double 型。

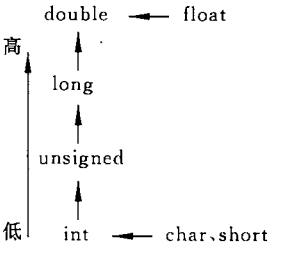


图 3-10

注意：箭头方向只表示数据类型级别的高低，由低向高转换。不要理解为 int 型先转换成 unsigned int 型，再转成 long 型，再转成 double 型。如果一个 int 型数据与一个 double 型数据运算，是直接将 int 型转成 double 型。同理，一个 int 型与一个 long 型数据运算，先将 int 型转换成 long 型。

换言之，如果有一个数据是 float 型或 double 型，则另一数据要先转换为 double 型，运算结果为 double 型。如果参加运算的两个数据中最高级别为 long 型，则另一数据先转换为 long 型，运算结果为 long 型。其他依此类推。

假设已指定 i 为整型变量，f 为 float 变量，d 为 double 型变量，e 为 long 型，有下面式子：

```
10+'a'+i*f-d/e
```

在计算机执行时从左至右扫描，运算次序为：

① 进行 10+'a'的运算，先将'a'转换成整数 97，运算结果为 107。

② 由于“*”比“+”优先，先进行 i*f 的运算。先将 i 与 f 都转成 double 型，运算结果为 double 型。

③ 整数 107 与 i*f 的积相加。先将整数 107 转换成双精度数(小数点后加若干个 0，即 107.000…00)，结果为 double 型。

④ 将变量 e 化成 double 型，d/e 结果为 double 型。

⑤ 将 10+'a'+i*f 的结果与 d/e 的商相减，结果为 double 型。

上述的类型转换是由系统自动进行的。

3.8 算术运算符和算术表达式

3.8.1 C 语言运算符简介

C 语言的运算符范围很宽，把除了控制语句和输入输出以外的几乎所有的基本操作都作为运算符处理，例如将赋值符“=”作为赋值运算符、方括号作为下标运算符等。C 的运算符有以下几类：

(1) 算术运算符　　(+ - * / %)
(2) 关系运算符　　(> < == >= <= !=)
(3) 逻辑运算符　　(! && ||)
(4) 位运算符　　(<< >> ~ | ^ &)
(5) 赋值运算符　　(=及其扩展赋值运算符)
(6) 条件运算符　　(?:)
(7) 逗号运算符　　(,)
(8) 指针运算符　　(* 和 &)
(9) 求字节数运算符　　(sizeof)
(10) 强制类型转换运算符　　((类型))
(11) 分量运算符　　(. ->)
(12) 下标运算符　　([])
(13) 其他　　(如函数调用运算符())

本章只介绍算术运算符和赋值运算符，在以后各章中结合有关内容将陆续介绍其他运算符。运算符见本书附录 C。

3.8.2 算术运算符和算术表达式

1. 基本的算术运算符

(1) +(加法运算符，或正值运算符，如 3+5、+3)；
(2) -(减法运算符，或负值运算符，如 5-2、-3)；
(3) *(乘法运算符，如 3*5)；
(4) /(除法运算符，如 5/3)；
(5) %(模运算符，或称求余运算符，%两侧均应为整型数据，如 7%4 的值为 3)。

需要说明，两个整数相除的结果为整数，如 5/3 的结果值为 1，舍去小数部分。但是，如果除数或被除数中有一个为负值，则舍入的方向是不固定的。例如，－5/3 在有的系统中得到的结果为－1，在有的系统中则得到结果为－2。多数 C 编译系统（如 Turbo C）采取“向零取整”的方法，即 5/3＝1，－5/3＝－1，取整后向零靠拢。

如果参加＋、－、＊、/运算的两个数中有一个数为实数或双精度数，则结果是 double 型，因为所有实数都按 double 型进行运算。

2. 算术表达式和运算符的优先级与结合性

用算术运算符和括号将运算对象（也称操作数）连接起来的、符合 C 语法规则的式子，称为 C 算术表达式。运算对象包括常量、变量、函数等。例如，下面是一个合法的 C 语言算术表达式：

```
a*b/c-1.5+'a'
```

C 语言规定了运算符的优先级和结合性。在表达式求值时，先按运算符的优先级别高低次序执行，例如先乘除后加减。如表达式 a－b＊c，b 的左侧为减号，右侧为乘号，而乘号优先于减号，因此，相当于 a－(b＊c)。如果在一个运算对象两侧的运算符的优先级别相同，如 a－b＋c，则按规定的“结合方向”处理。

C 语言规定了各种运算符的结合方向（结合性），算术运算符的结合方向为“自左至右”，即先左后右，因此 b 先与减号结合，执行 a－b 的运算，再执行加 c 的运算。“自左至右的结合方向”又称“左结合性”，即运算对象先与左面的运算符结合。以后可以看到有些运算符的结合方向为“自右至左”，即右结合性（例如，赋值运算符）。关于“结合性”的概念在其他一些高级语言中是没有的，是 C 的特点之一，希望能弄清楚。附录 C 列出了所有运算符以及它们的优先级别和结合性。

如果一个运算符的两侧的数据类型不同，则会按 3.7 节所述，先自动进行类型转换，使二者具有同一种类型，然后进行运算。

3. 强制类型转换运算符

可以利用强制类型转换运算符将一个表达式转换成所需类型。例如：

```
(double)a          (将 a 转换成 double 类型)
(int)(x+y)         (将 x+y 的值转换成整型)
(float)(5%3)       (将 5%3 的值转换成 float 型)
```

其一般形式为

(类型名)(表达式)

注意，表达式应该用括号括起来。如果写成

```
(int)x+y
```

则只将 x 转换成整型，然后与 y 相加。

需要说明的是，在强制类型转换时，得到一个所需类型的中间变量，原来变量的类型未发生变化。例如：

```
(int)x          (不要写成 int(x))
```

如果已定义 x 为 float 型，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 x 的整数部分，而 x 的类型不变(仍为 float 型)，见例 3.8。

例 3.8 强制类型转换。

```
#include <stdio.h>
void main()
  {
    float x;
    int i;
    x=3.6;
    i=(int)x;
    printf("x=%f, i=%d\n",x,i);
  }
```

运行结果如下：

```
x=3.600000, i=3
```

x 类型仍为 float 型，值仍等于 3.6。

从上可知，有两种类型转换，一种是在运算时不必用户指定，系统自动进行的类型转换，如 3+6.5。第二种是强制类型转换。当自动类型转换不能实现目的时，可以用强制类型转换。如“%”运算符要求其两侧均为整型量，若 x 为 float 型，则“x%3”不合法，必须用“(int)x % 3”。从附录 C 可以查到，强制类型转换运算优先于%运算，因此先进行(int)x 的运算，得到一个整型的中间变量，然后再对 3 求模。此外，在函数调用时，有时为了使实参与形参类型一致，可以用强制类型转换运算符得到一个所需类型的参数。

4. 自增、自减运算符

作用是使变量的值增 1 或减 1，例如：

```
++i,--i        (在使用 i 之前，先使 i 的值加(减)1)
i++,i--        (在使用 i 之后，使 i 的值加(减)1)
```

粗略地看，++i 和 i++的作用相当于 i=i+1。但++i 和 i++不同之处在于++i 是先执行 i=i+1 后，再使用 i 的值；而 i++是先使用 i 的值后，再执行 i=i+1。如果 i 的原值等于 3，请分析下面的赋值语句：

```
① j=++i;       (i 的值先变成 4，再赋给 j,j 的值为 4)
② j=i++;       (先将 i 的值 3 赋给 j,j 的值为 3,然后 i 变为 4)
```

又例如：

```
i=3;
```

```
printf("%d",++i);
```

输出“4”。若改为

```
printf("%d",i++);
```

则输出“3”。

注意：

(1) 自增运算符(++)和自减运算符(--)只能用于变量，而不能用于常量或表达式，如5++或(a+b)++都是不合法的。因为5是常量，常量的值不能改变。(a+b)++也不可能实现，假如a+b的值为5，那么自增后得到的6放在什么地方呢？无变量可供存放。

(2) ++和--的结合方向是“自右至左”，见附录C。前面已提到，算术运算符的结合方向为“自左而右”，这是大家所熟知的。如果有-i++，i的左面是负号运算符，右面是自加运算符。如果i的原值等于3，若按左结合性，相当于(-i)++，而(-i)++是不合法的，因为对表达式不能进行自加自减运算。

从附录C知道负号运算符和“++”运算符同优先级，而结合方向为“自右至左”(右结合性)，即它相当于-(i++)，如果有printf("%d",-i++)，则先取出i的值3，输出-i的值-3，然后i增值为4。注意 -(i++)是先用i的原值3加上负号输出-3，再对i加1，不要认为先加完1后再加负号，输出-4，这是不对的。

自增(减)运算符常用于循环语句中，使循环变量自动加1；也用于指针变量，使指针指向下一个地址。这些将在以后的章节中介绍。

5. 有关表达式使用中的问题说明

(1) C运算符和表达式使用灵活，利用这一点可以巧妙地处理许多在其他语言中难以处理的问题。但是应当注意：ANSI C并没有具体规定表达式中的子表达式的求值顺序，允许各编译系统自己安排。例如，对表达式

```
a = f1( )+f2( )
```

并不是所有的编译系统都先调用函数f1( )，然后调用函数f2( )。在一般情况下，先调用f1( )和先调用f2( )的结果可能是相同的。但是在有的情况下结果可能不同。有时会出现一些令人容易搞混的问题，因此务必要小心谨慎。

又如，i的初值为3，有以下表达式：

```
(i++)+(i++)+(i++)
```

表达式的值是多少呢？有的系统按照自左而右顺序求解括号内的运算，求完第1个括号的值后，实现i的自加，i值变为4，再求第2个括号的值，结果表达式相当于3+4+5，即12。而另一些系统(如Turbo C和MS C)把3作为表达式中所有i的值，因此3个i相加，得到表达式的值为9。在求出整个表达式的值后再实现自加3次，i的值变为6。

应该避免出现这种歧义性。如果编程者的原意是想得到12，可以写成下列语句：

```
i = 3;
```

```
a = i++;
b = i++;
c = i++;
d = a + b + c;
```

执行完上述语句后，d 的值为 12，i 的值为 6。虽然语句多了，但不会引起歧义，无论程序移植到哪一种 C 编译系统运行，结果都一样。

(2) C 语言中有的运算符为一个字符，有的运算符由两个字符组成，在表达式中如何组合呢？如 i+++j，是理解为(i++)+j 呢？还是 i+(++j)呢？C 编译系统在处理时尽可能多地(自左而右)将若干个字符组成一个运算符(在处理标识符、关键字时也按同一原则处理)，如 i+++j，将解释为(i++)+j，而不是 i+(++j)。为避免误解，最好采取大家都能理解的写法，不要写成 i+++j 的形式，而应写成(i++)+j 的形式。

(3) C 语言中类似上述这样的问题还有一些。例如，在调用函数时，实参数的求值顺序，C 标准并无统一规定。如 i 的初值为 3，如果有下面的函数调用：

```
printf("%d,%d",i,i++);
```

在有的系统中，从左至右求值，输出“3,3”。在多数系统中对函数参数的求值顺序是自右而左，上面 printf 函数中要输出两个表达式的值(i 和 i++分别是两个表达式)，先求出第 2 个表达式 i++的值 3(i 未自加时的值)，然后求第 1 个表达式的值，由于在求解第 2 个表达式后，执行 i++，使 i 加 1 变为 4，因此 printf 函数中第一个参数 i 的值为 4。所以上面 printf 函数输出的是“4,3”。

以上这种写法不宜提倡，最好改写成：

```
j = i++;
printf("%d, %d", j,i);
```

总之，不要写出别人看不懂的、也不知道系统会怎样执行的程序。

在看别人的程序时，应该考虑到在类似上述问题上，不同系统的处理方法不尽相同。应当知道使用 C 语言时可能出问题的地方，以免遇到问题时不知其所以然。

使用++和--时，常会出现一些人们“想不到”的副作用，初学者要慎用。

3.9 赋值运算符和赋值表达式

1. 赋值运算符

赋值符号“=”就是赋值运算符，它的作用是将一个数据赋给一个变量。如“a=3”的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。也可以将一个表达式的值赋给一个变量。

2. 类型转换

如果赋值运算符两侧的类型不一致，但都是数值型或字符型时，在赋值时要进行类型

转换。

(1) 将浮点型数据(包括单、双精度)赋给整型变量时,舍弃浮点数的小数部分。如 i 为整型变量,执行"i=3.56"的结果是使 i 的值为 3,以整数形式存储在整型变量中。

(2) 将整型数据赋给单、双精度变量时,数值不变,但以浮点数形式存储到变量中,如将 23 赋给 float 变量 f,即执行 f=23,先将 23 转换成 23.00000,再存储在 f 中。如将 23 赋给 double 型变量 d,即执行 d=23,则将 23 补足有效位数字为 23.00000000000000,然后以双精度浮点数形式存储到变量 d 中。

(3) 将一个 double 型数据赋给 float 变量时,截取其前面 7 位有效数字,存放到 float 变量的存储单元(4 个字节)中。但应注意数值范围不能溢出。例如:

```
float f;
double d=123.456789e100;
f=d;
```

就出现溢出的错误。

将一个 float 型数据赋给 double 变量时,数值不变,有效位数扩展到 16 位,在内存中以 8 个字节存储。

(4) 字符型数据赋给整型变量时,由于字符只占 1 个字节,而整型变量占 2 个字节,因此将字符数据(8 个二进制位)放到整型变量存储单元的低 8 位中。有两种情况如下。

① 如果所用系统将字符处理为无符号的字符类型,或程序已将字符变量定义为 unsigned char 型,则将字符的 8 位放到整型变量低 8 位,高 8 位补零。例如,将字符'\376'赋给 int 型变量 i,如图 3-11(a)所示。

② 如果所用系统(如 Turbo C++)将字符处理为带符号的(即 signed char),若字符最高位为 0,则整型变量高 8 位补 0;若字符最高位为 1,则高 8 位全补 1(见图 3-11(b))。这称为"符号扩展",这样做的目的是使数值保持不变,如变量 c(字符'\376')以整数形式输出为-2,i 的值也是-2。

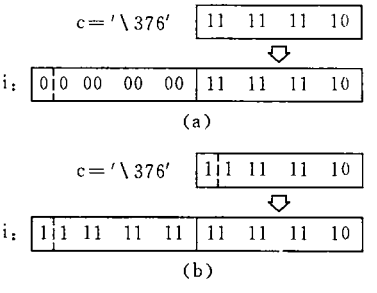


图 3-11

(5) 将一个 int、short、long 型数据赋给一个 char 型变量时,只将其低 8 位原封不动地送到 char 型变量(即截断)。例如:

```
int i=289;
char c='a';
c=i;
```

赋值情况见图 3-12。c 的值为 33,如果用"%c"输出 c,将得到字符"!"(其 ASCII 码为 33)。

(6) 将带符号的整型数据(int 型)赋给 long 型变量时,要进行符号扩展,将整型数的 16 位送到 long 型低 16 位中,如果 int 型数据为正值(符号位为 0),则 long 型变量的高 16 位补 0;如果 int 型变量为负值(符号位为 1),则 long 型变量的高 16 位补 1,以保持数值不改变。

反之，若将一个 long 型数据赋给一个 int 型变量，只将 long 型数据中低 16 位原封不动地送到整型变量（即截断）。例如：

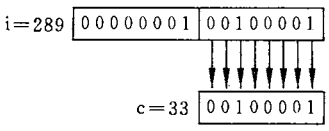


图 3-12

```
int a;
long b=8;
a=b;
```

赋值情况见图 3-13。如果 b=65536（八进制数 0200000），则赋值后 a 值为 0，见图 3-14。如果 b=020000000000（八进制数），赋值后 a 也为 0。请读者自己分析。

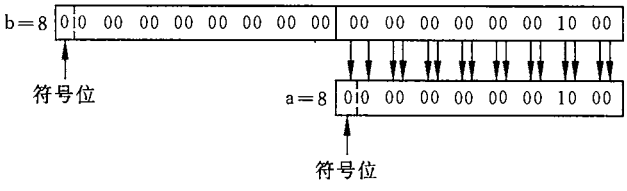


图 3-13

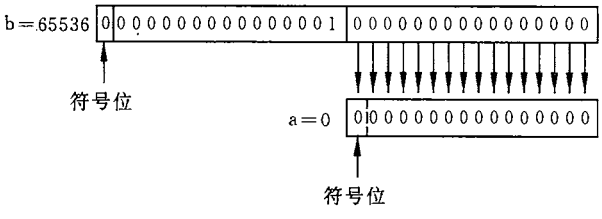


图 3-14

（7）将 unsigned int 型数据赋给 long int 型变量时，不存在符号扩展问题，只需将高位补 0 即可。将一个 unsigned 类型数据赋给一个占字节数相同的非 unsigned 型整型变量（例如：unsigned int⇒int，unsigned long⇒long，unsigned short⇒short），将 unsigned 型变量的内容原样送到非 unsigned 型变量中，但如果数据范围超过相应整型的范围，则会出现数据错误。例如：

```
unsigned int a=65535;
int b;
b=a;
```

将 a 整个送到 b 中（图 3-15），由于 b 是 int 型，第 1 位是符号位，因此 b 成了负数。根据补码知识可知，b 的值为－1，可以用

```
printf("%d",b);
```

来验证。

（8）将非 unsigned 型数据赋值给长度相同的 unsigned 型变量，也是原样赋值（连原有的符号位也作为数值一起传送），见例 3.9 的分析。

例 3.9 有符号数据传送给无符号变量。

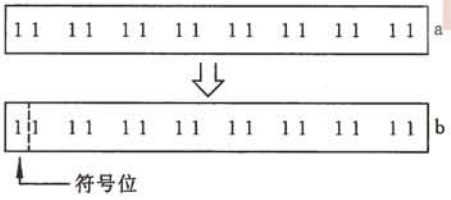


图 3-15

```
#include <stdio.h>
void main()
{
  unsigned a;
  int b=-1;
  a=b;
  printf("%u\n",a);
}
```

其中，“%u”是输出无符号数时所用的格式符。运行结果为

```
65535
```

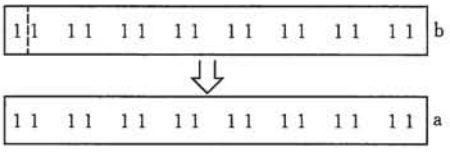


图 3-16

赋值情况见图 3-16。如果 b 为正值，且在 0～32767 之间，则赋值后数值不变。

以上的赋值规则看起来比较复杂，其实，不同类型的整型数据间的赋值归根到底就是一条：按存储单元中的存储形式直接传送。只要学过补码知识的，对以上规则是不难理解的。由于 C 语言使用灵活，在不同类型数据之间赋值时，常常会出现意想不到的结果，而编译系统并不提示出错，全靠程序员的经验来找出问题。这就要求编程人员对出现问题的原因有所了解，以便迅速排除故障。

根据作者的经验，相当多的初学者（甚至还有不少有一定编程经验的人）在这方面常出错，而且不能很快地找出原因。因此本书对此做了较详细的说明，使读者在编程序和对调试程序时有所依据。在学习本书时不必死记，这部分内容可以通过学生自学和上机实践来掌握。

3. 复合的赋值运算符

在赋值符“=”之前加上其他运算符，可以构成复合的运算符。如果在“=”前加一个“+”运算符就成了复合运算符“+=”。例如，可以有：

```
a+=3          等价于 a=a+3
```

x*=y+8　　等价于 x=x*(y+8)

x%=3　　等价于 x=x%3

以“a+=3”为例来说明，它相当于使 a 进行一次自加 3 的操作。即先使 a 加 3，再赋给 a。同样，“x*=y+8”的作用是使 x 乘以(y+8)，再赋给 x。

为便于记忆，可以这样理解：

① a += b　　(其中 a 为变量，b 为表达式)

② a+= b　　(将有下划线的“a+”移到“=”右侧)

③ a = a + b　　(在“=”左侧补上变量名 a)

注意，如果 b 是包含若干项的表达式，则相当于它有括号。例如，以下 3 种写法是等价的：

① x %= y+3

② x %= (y+3)

③ x = x %(y+3)　　(不要错写成 x=x%y+3)

凡是二元(二目)运算符，都可以与赋值符一起组合成复合赋值符。C 语言规定可以使用 10 种复合赋值运算符。即：

+=，-=，*=，/=，%=，<<=，>>=，&=，^=，|=

后 5 种是有关位运算的，将在第 12 章介绍。

C 语言采用这种复合运算符，一是为了简化程序，使程序精练，二是为了提高编译效率(这样写法与“逆波兰”式一致，有利于编译，能产生质量较高的目标代码。学过编译原理的读者对此容易理解，其他读者可不必深究)。专业人员喜欢使用复合运算符，程序显得专业一点，对初学者来说，不必多用，首要的是保持程序清晰易懂。

4. 赋值表达式

由赋值运算符将一个变量和一个表达式连接起来的式子称为“赋值表达式”。它的一般形式为

<变量><赋值运算符><表达式>

如“a=5”是一个赋值表达式。对赋值表达式求解的过程是：先求赋值运算符右侧的“表达式”的值，然后赋给赋值运算符左侧的变量。一个表达式应该有一个值，例如，赋值表达式“a=3*5”的值为 15，执行表达式后，变量 a 的值也是 15。赋值运算符左侧的标识符称为“左值”(left value，简写为 lvalue)。并不是任何对象都可以作为左值的，变量可以作为左值，而表达式 a+b 就不能作为左值，常变量也不能作为左值，因为常变量不能被赋值。出现在赋值运算符右侧的表达式称为“右值”(right value，简写为 rvalue)。显然左值也可以出现在赋值运算符右侧，因而凡是左值都可以作为右值。例如：

```
int a=3,b,c;
b=a;                /* b 是左值 */
c=b;                /* b 也是右值 */
```

赋值表达式中的“表达式”，又可以是一个赋值表达式。例如：

```
a=(b=5)
```

括号内的“b=5”是一个赋值表达式，它的值等于5。执行表达式“a=(b=5)”相当于执行“b=5”和“a=b”两个赋值表达式。因此a的值等于5，整个赋值表达式的值也等于5。从附录C可以知道赋值运算符按照“自右而左”的结合顺序，因此，“(b=5)”外面的括号可以不要，即“a=(b=5)”和“a=b=5”等价，都是先求“b=5”的值(得5)，然后再赋给a，下面是赋值表达式的例子：

```
a=b=c=5          (赋值表达式值为5,a、b、c值均为5)
a=5+(c=6)        (表达式值为11,a值为11,c值为6)
a=(b=4)+(c=6)    (表达式值为10,a值为10,b等于4,c等于6)
a=(b=10)/(c=2)   (表达式值为5,a等于5,b等于10,c等于2)
```

请分析下面的赋值表达式：

```
(a=3*5)=4*3
```

先执行括号内的运算，将15赋给a，然后执行4*3的运算，得12，再把12赋给a。最后a的值为12，整个表达式的值为12。读者可以看到：(a=3*5)出现在赋值运算符的左侧，因此赋值表达式(a=3*5)是左值。请注意，在对赋值表达式(a=3*5)求解后，变量a得到值15，此时赋值表达式(a=3*5)=4*3相当于(a)=4*3，在执行(a=3*5)=4*3时，实际上是将4*3的积12赋给变量a，而不是赋给3*5。正因为这样，赋值表达式才能够作为左值。

赋值表达式作为左值时应加括号，如果写成下面这样就出现语法错误：

```
a=3*5=4*3
```

因为3*5不是左值，不能出现在赋值运算符的左侧。

赋值表达式也可以包含复合的赋值运算符。例如：

```
a+=a-=a*a
```

也是一个赋值表达式。如果a的初值为12，此赋值表达式的求解步骤如下：

① 先进行“a-=a*a”的运算，它相当于a=a-a*a，a的值为12-144=-132。

② 再进行“a+=-132”的运算，相当于a=a+(-132)，a的值为-132-132=-264。

将赋值表达式作为表达式的一种，使赋值操作不仅可以出现在赋值语句中，而且可以以表达式形式出现在其他语句(如输出语句、循环语句等)中，例如：

```
printf("%d",a=b);
```

如果b的值为3，则输出a的值(也是表达式a=b的值)为3。在一个语句中完成了赋值和输出双重功能。这是C语言灵活性的一种表现。在第6章中将进一步看到这种应用及其优越性。

在第4章介绍“语句”之后，就可以了解到赋值表达式和赋值语句之间的联系和区

别了。

3.10 逗号运算符和逗号表达式

C语言提供一种特殊的运算符——逗号运算符。用它将两个表达式连接起来。例如：

```
3+5,6+8
```

称为逗号表达式，又称为“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

逗号表达式的求解过程是：先求解表达式1，再求解表达式2。整个逗号表达式的值是表达式2的值。例如，上面的逗号表达式“3+5，6+8”的值为14。又如，逗号表达式：

```
a=3*5,a*4
```

对此表达式的求解，读者可能会有两种不同的理解：一种认为“3*5，a*4”是一个逗号表达式，先求出此逗号表达式的值，如果a的原值为3，则逗号表达式的值为12，将12赋给a，因此最后a的值为12。另一种认为：“a=3*5”是一个赋值表达式，“a*4”是另一个表达式，二者用逗号相连，构成一个逗号表达式。这两者哪一个对呢？从附录C可知：赋值运算符的优先级别高于逗号运算符，因此应先求解a=3*5（也就是把“a=3*5”作为一个表达式）。经计算和赋值后得到a的值为15，然后求解a*4，得60。整个逗号表达式的值为60。

一个逗号表达式又可以与另一个表达式组成一个新的逗号表达式，例如：

```
(a=3*5,a*4),a+5
```

先计算出a的值为3*5，等于15，再进行a*4的运算得60（但a值未变，仍为15），再进行a+5得20，即整个表达式的值为20。

逗号表达式的一般形式可以扩展为

表达式1，表达式2，表达式3，…，表达式*n*

它的值为表达式*n*的值。

从附录C可知，逗号运算符是所有运算符中级别最低的。因此，下面两个表达式的作用是不同的：

① x=(a=3,6*3)

② x=a=3,6*3

第①个是一个赋值表达式，将一个逗号表达式的值赋给x，x的值等于18。第②个是逗号表达式，包括一个赋值表达式和一个算术表达式，x的值为3，整个逗号表达式的值为18。

其实，逗号表达式无非是把若干个表达式“串联”起来。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，逗号表达式最常用于循环语句（for语句）中，详见第6章。

注意，并不是任何地方出现的逗号都是作为逗号运算符。例如函数参数也是用逗号来间隔的。例如：

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

其中的"a,b,c"并不是一个逗号表达式，它是 printf 函数的 3 个参数，参数间用逗号间隔。有关函数的详细叙述见第 8 章。如果改写为

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

则"(a,b,c)"是一个逗号表达式，它的值等于 c 的值。括号内的逗号不是参数间的分隔符，而是逗号运算符。括号中的内容是一个整体，作为 printf 函数的一个参数。

C 语言表达能力强，其中一个重要方面就在于它的表达式类型丰富，运算符功能强，因而 C 语言使用灵活，适应性强。在后面几章中将会进一步看到这一点。

习　题

3.1 请将 C 语言的数据类型和其他高级语言的数据类型作比较，C 有哪些特点？

3.2 C 语言为什么规定对所有用到的变量要"先定义，后使用"，这样做有什么好处？

3.3 请将下面各数用八进制数和十六进制数表示：

(1) 10　　(2) 32　　(3) 75　　(4) −617

(5) −111　　(6) 2483　　(7) −28654　　(8) 21003

3.4 将以下 3 个整数分别赋给不同类型的变量，请将赋值后数据在内存中的存储形式填入表 3-4 中。(注：如没有学过二进制数和补码的，此题可不做。)

表 3-4　赋值后数据在内存中的存储形式

变量的类型	25	−2	32769
int 型(16 位)			
long 型(32 位)			
short 型(16 位)			
signed char(8 位)			
unsigned int 型			
unsigned long 型			
unsigned short 型			
unsigned char 型			

3.5 字符常量与字符串常量有什么区别？

3.6 写出以下程序的运行结果：

```
#include <stdio.h>
void main()
{
  char c1='a',c2='b',c3='c',c4='\101',c5='\116';
```

```
    printf("a%c b%c\tc%c\tabc\n",c1,c2,c3);
    printf("\t\b%c %c\n",c4,c5);
}
```

3.7 要将“China”译成密码，密码规律是：用原来的字母后面第 4 个字母代替原来的字母。例如，字母“A”后面第 4 个字母是“E”，用“E”代替“A”。因此，“China”应译为“Glmre”。请编一程序，用赋初值的方法使 c1、c2、c3、c4、c5 这 5 个变量的值分别为‘C’、‘h’、‘i’、‘n’、‘a’，经过运算，使 c1、c2、c3、c4、c5 分别变为‘G’、‘l’、‘m’、‘r’、‘e’，并输出。

3.8 例 3.6 能否改成如下：

```
#include <stdio.h>
void main()
{
    int c1,c2;          (原为 char c1,c2)
    c1=97;
    c2=98;
    printf("%c%c\n",c1,c2);
    printf("%d %d\n",c1,c2);
}
```

分析运行时会显示什么信息？为什么？

3.9 求下面算术表达式的值：

(1) x+a%3*(int)(x+y)%2/4

设 x=2.5,a=7,y=4.7

(2) (float)(a+b)/2+(int)x%(int)y

设 a=2,b=3,x=3.5,y=2.5

3.10 写出下面程序的运行结果：

```
#include <stdio.h>
void main()
{
    int i,j,m,n;
    i=8;
    j=10;
    m=++i;
    n=j++;
    printf("%d,%d,%d,%d\n",i,j,m,n);
}
```

3.11 写出下面赋值的结果。表 3-5 中所写出数值是要将它赋给其他类型变量，将所有空格填上赋值后的数值。

表 3-5

int	99					42	
char		'd'					
unsigned int			76				65535
float				53.65			
long int					68		

3.12 写出下面赋值表达式运算后 a 的值，设原来 a=12：

(1) a+=a　　(2) a-=2

(3) a*=2+3　　(4) a/=a+a

(5) a%=(n%=2)，n 的值等于 5　　(6) a+=a-=a*=a

第4章　最简单的C程序设计
——顺序程序设计

在第1章中介绍了几段简单的C程序，在第3章中介绍了程序中用到的一些基本要素（常量、变量、运算符、表达式等），它们是构成程序的基本成分。本章将介绍几种简单的C语句以及怎样利用它们编写简单的程序。

4.1　C语句概述

和其他高级语言一样，C语言的语句用来向计算机系统发出操作指令。一个语句经编译后产生若干条机器指令。一个实际的程序应当包含若干语句。应当指出，C语句都是用来完成一定操作任务的。声明部分的内容不应称为语句。如："int a;"不是一条C语句，它不产生机器操作，而只是对变量的定义。从第1章已知，一个函数包含声明部分和执行部分，执行部分是由语句组成的。C程序结构可以用图4-1表示。即一个C程序可以由若干个源程序文件（分别进行编译的文件模块）组成，一个源文件可以由若干个函数和预处理命令以及全局变量声明部分组成（关于"全局变量"见第8章，"预编译命令"见第9章），一个函数由数据声明部分和执行语句组成。

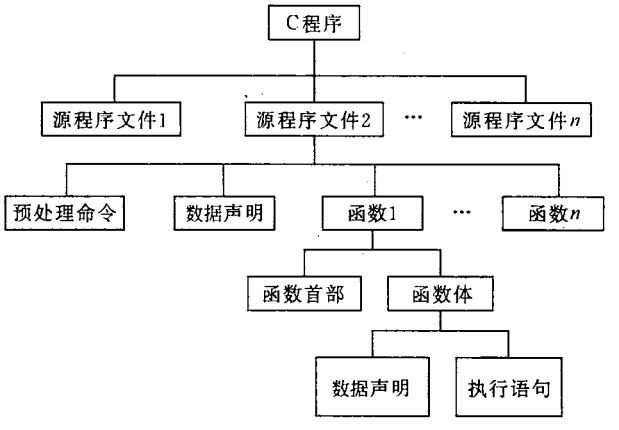


图　4-1

在第3章中已经说明，程序应该包括数据描述（由声明部分来实现）和数据操作（由语句来实现）。数据描述包括定义数据结构和在需要时对数据赋予初值。数据操作的任务是对已提供的数据进行加工。

C语句分为以下5类。

(1) 控制语句。控制语句用于完成一定的控制功能。C 只有 9 种控制语句,它们是:

① if()…else…　　(条件语句)
② for()…　　(循环语句)
③ while()…　　(循环语句)
④ do…while()　　(循环语句)
⑤ continue　　(结束本次循环语句)
⑥ break　　(中止执行 switch 或循环语句)
⑦ switch　　(多分支选择语句)
⑧ goto　　(转向语句)
⑨ return　　(从函数返回语句)

上面 9 种语句表示形式中的括号"()"表示括号中是一个"判别条件","…"表示内嵌的语句。例如:"if()…else…"的具体语句可以写成:

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

其中 x>y 是一个"判别条件",z=x;和 z=y;是语句,这两个语句是内嵌在 if…else 语句中的。这个 if…else 语句的作用是:先判别条件"x>y"是否成立,如果 x>y 成立,就执行内嵌语句"z=x;";否则就执行内嵌语句"z=y;"。

(2) 函数调用语句。函数调用语句由一个函数调用加一个分号构成,例如:

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

(3) 表达式语句。表达式语句由一个表达式加一个分号构成,最典型的是,由赋值表达式构成一个赋值语句。例如:

```
a=3
```

是一个赋值表达式,而

```
a=3;
```

是一个赋值语句。可以看到一个表达式的最后加一个分号就成了一个语句。一个语句必须在最后出现分号,分号是语句中不可缺少的组成部分,而不是两个语句间的分隔符号。例如:

```
i=i+1          (是表达式,不是语句)
i=i+1;         (是语句)
```

任何表达式都可以加上分号而成为语句,例如:

```
i++;
```

是一个语句,作用是使 i 值加 1。又例如:

```
x+y;
```

也是一个语句,作用是完成 x+y 的操作,它是合法的,但是并不把 x+y 的和赋给另一变

量，所以它并无实际意义。

表达式能构成语句是C语言的一个重要特色。其实“函数调用语句”也是属于表达式语句，因为函数调用(如sin(x))也属于表达式的一种。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”分开来说明。由于C程序中大多数语句是表达式语句(包括函数调用语句)，所以有人把C语言称做“表达式语言”。

(4) 空语句。下面是一个空语句：

```
;
```

即只有一个分号的语句，它什么也不做。有时用来作流程的转向点(流程从程序其他地方转到此语句处)，也可用来作为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)。

(5) 复合语句。可以用{}把一些语句括起来成为复合语句(又称分程序)。例如下面是一个复合语句：

```
{
  z=x+y;
  t=z/100;
  printf("%f",t);
}
```

注意：复合语句中最后一个语句中最后的分号不能忽略不写。

C语言允许一行写几个语句，也允许一个语句拆开写在几行上，书写格式无固定要求(FORTRAN语言和COBOL语言对书写格式有严格要求)。

在本章中将介绍几种顺序执行的语句，在执行这些语句的过程中不会发生流程的控制转移。

4.2 赋值语句

前面已经介绍，赋值语句是由赋值表达式加上一个分号构成。由于赋值语句应用十分普遍，所以专门再讨论一下。

C语言的赋值语句具有其他高级语言的赋值语句的一切特点和功能。但也应当注意到它们的不同。

(1) C语言中的赋值号“=”是一个运算符，在其他大多数语言中赋值号不是运算符。

(2) 关于赋值表达式与赋值语句的概念，其他多数高级语言没有“赋值表达式”这一概念。作为赋值表达式可以包括在其他表达式之中，例如：

```
if((a=b)>0)t=a;
```

按语法规定if后面的括号内是一个“条件”，例如可以是：“if(x>0)…”。现在在x的位置上换上一个赋值表达式“a=b”，其作用是：先进行赋值运算(将b的值赋给a)，然后判断a是否大于0，如大于0，执行t=a。在if语句中的“a=b”不是赋值语句而是赋值表达式，这

样写是合法的。如果写成：

```
if((a=b;)>0)t=a;
```

就错了。在 if 的条件中可以包含赋值表达式，但不能包含赋值语句。由此可以看到，C 语言把赋值语句和赋值表达式区别开来，增加了表达式的种类，使表达式的应用几乎“无孔不入”，能实现其他语言中难以实现的功能。

4.3 数据输入输出的概念及在 C 语言中的实现

在讨论输入输出时要注意以下几点。

(1) 所谓输入输出是以计算机主机为主体而言的。从计算机向外部输出设备(如显示器、打印机等)输出数据称为输出，从输入设备(如键盘、鼠标、扫描仪等)向计算机输入数据称为输入。

(2) C 语言本身不提供输入输出语句，输入和输出操作是由 C 函数库中的函数来实现的。在 C 标准函数库中提供了一些输入输出函数，例如，printf 函数和 scanf 函数。读者在使用它们时，千万不要误认为它们是 C 语言提供的“输入输出语句”。printf 和 scanf 不是 C 语言的关键字，而只是函数的名字。实际上完全可以不用 printf 和 scanf 这两个名字，而另外编两个输入输出函数，用其他的函数名。C 提供的函数以库的形式存放在系统中，它们不是 C 语言文本中的组成部分。

在第 1 章中曾介绍，不把输入输出作为 C 语句的目的是使 C 语言编译系统简单，因为将语句翻译成二进制的指令是在编译阶段完成的，没有输入输出语句就可以避免在编译阶段处理与硬件有关的问题，可以使编译系统简化，而且通用性强，可移植性好，在各种型号的计算机都能适用，便于在各种计算机上实现。各种 C 编译系统提供的系统函数库是各计算机厂商(或软件公司)根据用户的需要编写的，并且已编译成目标文件(.obj 文件)。它们在连接阶段与由源程序经编译而得到的目标文件(.obj 文件)相连接，生成一个可执行的目标程序(.exe 文件)。如果在源程序中有 printf 函数，在编译时并不把它翻译成目标指令，在连接阶段与系统函数库相连接后，在执行阶段中调用函数库中的 printf 函数。

不同的编译系统所提供的函数库中函数的数量、名字和功能是不完全相同的。不过，有些通用的函数(如 printf 和 scanf 等)，各种编译系统都提供，成为各种系统的标准函数。

C 语言函数库中有一批“标准输入输出函数”，它是以标准的输入输出设备(一般为终端设备)为输入输出对象的。其中有：putchar(输出字符)、getchar(输入字符)、printf(格式输出)、scanf(格式输入)、puts(输出字符串)、gets(输入字符串)。在本章中主要介绍前面 4 个最基本的输入输出函数。

(3) 在使用系统库函数时，要用预编译命令“#include”将有关的“头文件”包括到用户源文件中。在头文件中包含了调用函数时所需的有关信息。在使用标准输入输出库函数时，要用到“stdio.h”文件中提供的信息。文件后缀中“h”是 head 的缩写，#include 命令都是放在程序的开头，因此这类文件被称为“头文件”。在调用标准输入输出库函数时，文件开头应该有以下预编译命令：

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

或

```
#include "stdio.h"
```

stdio 是 standard input & output 的缩写，它包含了与标准 I/O 库有关的变量定义和宏定义以及对函数的声明。

有的 C 语言编译系统(如 Turbo C 2.0) 允许在使用 printf 和 scanf 函数时可不加 #include <stdio.h>命令，但有的编译系统(如 Turbo C ++ 3.0)则无此例外，因此应养成这样的习惯：只要在本程序文件中使用标准输入输出库函数时，一律加上“#include <stdio.h>”命令。

4.4 字符数据的输入输出

本节先介绍 C 标准 I/O 函数库中最简单的、也是最容易理解的字符输入输出函数 putchar 和 getchar，再介绍格式输入输出函数 printf 和 scanf 函数。

4.4.1 putchar 函数

putchar 函数(字符输出函数)的作用是向终端输出一个字符。其一般形式为

putchar(c)

它输出字符变量 c 的值，c 可以是字符型变量或整型变量。

例 4.1 输出单个字符。

```
#include <stdio.h>
void main()
{
  char a,b,c;
  a='B';b='O';c='Y';
  putchar(a);putchar(b);putchar(c); putchar ('\n');
}
```

运行结果：

```
BOY
```

用 putchar 函数可以输出能在屏幕上显示的字符，也可以输出控制字符，如 putchar('\n')的作用是输出一个换行符，使输出的当前位置移到下一行的开头。如果将例 4.1 程序最后一行改为：

```
putchar(a);putchar('\n');putchar(b);putchar('\n');putchar(c);putchar('\n');
```

则输出结果为：

```
B
```

O
Y

也可以输出其他转义字符，例如：

```
putchar('\101')          (输出字符'A')
putchar('\'')            (输出单撇号字符''')
putchar('\015')          (输出回车，不换行，使输出的当前位置移到本行开头)
```

4.4.2 getchar 函数

getchar 函数(字符输入函数)的作用是从终端(或系统隐含指定的输入设备)输入一个字符。getchar 函数没有参数，其一般形式为

getchar()

函数的值就是从输入设备得到的字符。

例 4.2 输入单个字符。

```
#include <stdio.h>
void main()
{
  char c;
  c=getchar();
  putchar(c);
  putchar('\n');          /* 换行 */
}
```

在运行时，如果从键盘输入字符'a'并按 Enter 键，就会在屏幕上看到输出的字符'a'。

```
a↙          (输入'a'后，按 Enter 键，字符才送到内存)
a           (输出变量 c 的值'a')
```

注意：getchar 函数只能接收一个字符。getchar 函数得到的字符可以赋给一个字符变量或整型变量，也可以不赋给任何变量，作为表达式的一部分。例如，例 4.2 第 4、5 行可以用下面一行代替：

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

因为 getchar 函数的值为'a'，因此 putchar 函数输出'a'。也可以用 printf 函数输出：

```
printf("%c",getchar());
```

4.5 格式输入与输出

C 语言的格式输入输出的规定比较繁琐，用的不对就得不到预期的结果，而输入输出又是最基本的操作，几乎每一个程序都包含输入输出，不少编程人员由于掌握不好这方面的知识而浪费了大量调试程序的时间。为了使读者对格式输入输出有全面的了解，在本

节中对格式输入与输出做了比较仔细的介绍，以便在编程时有所遵循。但是，在学习本书时不必花许多精力去死抠每一个细节，重点掌握最常用的一些规则即可。其他部分可在需要时随时查阅。这部分的内容建议自学和在计算机上练习，教师不必在课堂上一一细讲。应当通过编写和调试程序来逐步深入而自然地掌握输入输出的应用。

4.5.1 printf 函数

在前面各章节中已用到 printf 函数(格式输出函数)，它的作用是向终端(或系统隐含指定的输出设备)输出若干个任意类型的数据(putchar 只能输出字符，而且只能是一个字符，而 printf 可以输出多个数据，且为任意类型)。

1. printf 函数

printf 的一般格式为

printf(格式控制，输出表列)

例如：

```
printf("%d, %c\n",i,c)
```

括号内包括两部分：

(1)“格式控制”是用双撇号括起来的字符串，也称“转换控制字符串”，它包括两种信息。

① 格式说明。格式说明由“%”和格式字符组成，如%d、%f 等。它的作用是将输出的数据转换为指定的格式输出。格式说明总是由“%”字符开始的。

② 普通字符。普通字符即需要原样输出的字符。例如上面 printf 函数中双撇号内的逗号、空格和换行符。

(2)“输出表列”是需要输出的一些数据，可以是表达式。

下面是 printf 函数的例子：

```
printf("%d  %d", a,b)
        \/     |
    格式说明   输出表列
printf("a=%d  b=%d", a,b)
          \/      |
      格式说明 输出表列
```

在第二个 printf 函数中的双撇号内的字符，除了两个“%d”以外，还有非格式说明的普通字符，它们按原样输出。如果 a、b 的值分别为 3、4，则输出为

```
a=3␣b=4
```

其中有下划线的字符是 printf 函数中的“格式控制”字符串中的普通字符按原样输出的结果。3 和 4 是 a 和 b 的值(注意 3 和 4 无前导空格和尾随空格)，其数字位数由 a、b 值而

定。假如 a=12,b=123,则输出结果为

a=12␣b=123

由于 printf 是函数,因此,“格式控制”字符串和“输出表列”实际上都是函数的参数。printf 函数的一般形式可以表示为

printf(参数 1,参数 2,参数 3,…,参数 n)

printf 函数的功能是将参数 2～参数 n 按参数 1 给定的格式输出。

2. 格式字符

在输出时,对不同类型的数据要使用不同的格式字符。常用的有以下几种格式字符。

(1) d 格式符。用来输出十进制整数。有以下几种用法:

① %d。按十进制整型数据的实际长度输出。

② %md。m 为指定的输出字段的宽度。如果数据的位数小于 m,则左端补以空格,若大于 m,则按实际位数输出,例如:

```
printf("%4d,%4d",a,b);
```

若 a=123,b=12345,则输出结果为

␣123,12345

③ %ld。输出长整型数据,例如:

```
long a=135790;          /* 定义 a 为长整型变量 */
printf("%ld",a);
```

如果用%d 输出,就会发生错误,因为整型数据的范围为－32768～32767。对 long 型数据应当用%ld 格式输出。对长整型数据也可以指定字段宽度,如将上面 printf 函数中的“%ld”改为“%8ld”,则输出为

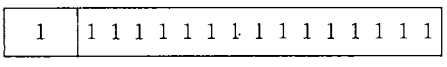
␣␣135790
8列

一个 int 型数据可以用%d 或%ld 格式输出。

(2) o 格式符。以八进制整数形式输出。由于是将内存单元中的各位的值(0 或 1)按八进制形式输出,因此输出的数值不带符号,即将符号位也一起作为八进制数的一部分输出。例如:

```
int a=－1;
printf("%d,%o",a,a);
```

－1 在内存单元中的存放形式(以补码形式存放)如下:



输出为

```
-1,177777
```

不会输出带负号的八进制整数。对长整数(long 型)可以用“%lo”格式输出。同样可以指定字段宽度,例如:

```
printf("%8o",a);
```

输出为

```
␣␣177777    (数字前有 2 个空格)
```

(3) x 格式符。以十六进制数形式输出整数。同样不会出现负的十六进制数。例如:

```
int a=-1;
printf("%x,%o,%d",a,a,a);
```

输出结果为

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

同样可以用“%lx”输出长整型数,也可以指定输出字段的宽度,如“%12x”。

(4) u 格式符。用来输出 unsigned 型数据,即无符号数,以十进制整数形式输出。

一个有符号整数(int 型)也可以用%u 格式输出;反之,一个 unsigned 型数据也可以用%d 格式输出。按相互赋值的规则处理(见第 3 章 3.9 节)。unsigned 型数据也可用%o 或%x 格式输出。

例 4.3 无符号数据的输出。

```
#include <stdio.h>
void main()
{
  unsigned int a=65535;
  int b=-2;
  printf("a=%d,%o,%x,%u\n",a,a,a,a);
  printf("b=%d,%o,%x,%u\n",b,b,b,b);
}
```

运行结果为:

```
a=-1,177777,ffff,65535
b=-2,177776,fffe,65534
```

请读者自己分析。

(5) c 格式符。用来输出一个字符。例如:

```
char c='a';
printf("%c",c);
```

输出字符'a'。请注意:“%c”中的 c 是格式符,逗号右边的 c 是变量名,不要搞混。

一个整数，只要它的值在0～255范围内，也可以用“%c”使之按字符形式输出，在输出前，系统会将该整数作为ASCII码转换成相应的字符；反之，一个字符数据也可以用整数形式输出。

例4.4 字符数据的输出。

```
#include <stdio.h>
void main()
{
  char c='a';
  int i=97;
  printf("%c,%d\n",c,c);
  printf("%c,%d\n",i,i);
}
```

运行结果为：

```
a,97
a,97
```

也可以指定输出字数宽度，如果有

```
printf("%3c",c);
```

则输出：“␣␣a”，即c变量输出占3列，前2列补空格。

(6) s格式符，用来输出一个字符串。有几种用法：

① %s。例如：

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

输出字符串“CHINA”(不包括双引号)。

② %ms，输出的字符串占m列，如字符串本身长度大于m，则突破m的限制，将字符串全部输出。若串长小于m，则左补空格。

③ %－ms，如果串长小于m，则在m列范围内，字符串向左靠，右补空格。

④ %m.ns，输出占m列，但只取字符串中左端n个字符。这n个字符输出在m列的右侧，左补空格。

⑤ %－m.ns，其中m、n含义同上，n个字符输出在m列范围的左侧，右补空格。如果n>m，则m自动取n值，即保证n个字符正常输出。

例4.5 字符串的输出。

```
#include <stdio.h>
void main()
{
  printf("%3s,%7.2s,%.4s,%－5.3s\n","CHINA","CHINA","CHINA", "CHINA");
}
```

输出如下：

CHINA,␣␣␣␣␣␣CH,CHIN,CHI␣␣

其中第3个输出项,格式说明为"%.4s",即只指定了n,没指定m,自动使m=n=4,故占4列。

(7) f格式符。用来输出实数(包括单、双精度),以小数形式输出。有以下几种用法。

① %f,不指定字段宽度,由系统自动指定,使整数部分全部输出,并输出6位小数。应当注意,在输出的数字中并非全部数字都是有效数字。单精度实数的有效位数一般为7位。

例4.6 输出实数时的有效位数。

```
#include <stdio.h>
void main()
{
  float x,y;
  x=111111.111;y=222222.222;
  printf("%f\n",x+y);
}
```

运行结果为:

```
333333.328125
```

显然,只有前7位数字是有效数字。千万不要以为凡是计算机输出的数字都是准确的。双精度数也可用%f格式输出,它的有效位数一般为16位,给出小数6位。

例4.7 输出双精度数时的有效位数。

```
#include <stdio.h>
void main()
{
  double x,y;
  x=1111111111111.111111111;
  y=2222222222222.222222222;
  printf("%f\n",x+y);
}
```

输出结果为:

```
3333333333333.333000
```

可以看到最后3位小数(超过16位)是无意义的。

② %m.nf,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。

③ %-m.nf与%m.nf基本相同,只是使输出的数值向左端靠,右端补空格。

例4.8 输出实数时指定小数位数。

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

```
void main()
{
  float f=123.456;
  printf("%f␣␣%10f␣␣%10.2f␣␣%.2f␣␣%-10.2f\n",f,f,f,f,f);
}
```

输出结果如下：

```
123.456001␣␣123.456001␣␣␣␣␣␣123.46␣␣123.46␣␣123.46␣␣␣␣
```

f 的值应为 123.456，但输出为 123.456001，这是由于实数在内存中的存储误差引起的。

(8) e 格式符，以指数形式输出实数。可用以下形式。

① %e，不指定输出数据所占的宽度和数字部分的小数位数，有的 C 编译系统自动指定给出数字部分的小数位数为 6 位，指数部分占 5 位(如 e+002)，其中"e"占 1 位，指数符号占 1 位，指数占 3 位。数值按规范化指数形式输出(即小数点前必须有而且只有 1 位非零数字)。例如

```
printf("%e",123.456);
```

输出如下：

```
1.234560e+002
  6列   5列
```

所输出的实数共占 13 列宽度。(注：不同系统的规定略有不同)

② %m.ne 和%-m.ne，m、n 和"-"字符的含义与前相同。此处 n 指拟输出的数据的小数部分(又称尾数)的小数位数。若 f=123.456，则：

```
printf("%e␣␣%10e␣␣%10.2e␣␣%.2e␣␣%-10.2e",f,f,f,f,f);
```

输出如下：

```
1.234560e+002␣␣1.234560e+002␣␣␣1.23e+002␣␣1.23e+002␣␣1.23e+002␣
   13列            13列            10列          9列          10列
```

第 2 个输出项按%10e 输出，即只指定了 m=10，未指定 n，凡未指定 n，自动使 n=6，整个数据长 13 列，超过给定的 10 列，乃突破 10 列的限制，按实际长度输出。第 3 个数据共占 10 列，小数部分占 2 列。第 4 个数据按"%.2e"格式输出，只指定 n=2，未指定 m，自动使 m 等于数据应占的长度，今为 9 列。第 5 个数据应占 10 列，数值只有 9 列，由于是"%-10.2e"，数值向左靠，右补一个空格。

注意：有的 C 系统的输出格式与此略有不同。

(9) g 格式符，用来输出实数，它根据数值的大小，自动选 f 格式或 e 格式(选择输出时占宽度较小的一种)，且不输出无意义的零。例如，若 f=123.468，则：

```
printf("%f␣␣%e␣␣%g",f,f,f);
```

输出如下：

123.468000␣␣1.234680e+002␣␣123.468␣␣␣

10列　　13列　　10列

用%f格式输出占10列，用%e格式输出占13列，用%g格式时，自动从上面两种格式中选择短者(今以%f格式为短)，故占10列，并按%f格式用小数形式输出，最后3个小数位为无意义的0，不输出，因此输出123.468，然后右补3个空格。%g格式用得较少。

以上介绍了9种格式符，归纳如表4-1所示。

表4-1　printf格式字符

格式字符	说　明
d, i	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, X	以十六进制无符号形式输出整数(不输出前导符0x)，用x则输出十六进制数的a～f时以小写形式输出。用X时，则以大写字母输出
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、双精度数，隐含输出6位小数
e, E	以指数形式输出实数，用e时指数以"e"表示(如1.2e+02)，用E时指数以"E"表示(如1.2E+02)
g, 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。用G时，若以指数形式输出，则指数以大写表示

在格式说明中，在%和上述格式字符间可以插入以下几种附加符号(又称修饰符)，见表4-2。

表4-2　printf的附加格式说明字符

字　符	说　明
l	用于长整型整数，可加在格式符d、o、x、u前面
m(代表一个正整数)	数据最小宽度
n(代表一个正整数)	对实数，表示输出n位小数；对字符串，表示截取的字符个数
-	输出的数字或字符在域内向左靠

在用printf函数输出时，务必注意数据类型应与上述格式说明匹配，否则将会出现错误。

对使用printf函数还要说明以下几点。

(1) 除了X、E、G外，其他格式字符必须用小写字母，如%d不能写成%D。

(2) 可以在printf函数中的"格式控制"字符串内包含第3章3.5节3.5.1小节中的"转义字符"，如"\n"、"\t"、"\b"、"\r"、"\f"、"\377"等。

(3) 上面介绍的d、o、x、u、c、s、f、e、g等字符，如用在"%"后面就作为格式符号。一个

格式说明以“%”开头，以上述 9 个格式字符之一为结束，中间可以插入附加格式字符(也称修饰符)。例如：

```
printf(" c=%cf=%fs=%s",c,f,s);
       格式说明 格式说明 格式说明
```

第一个格式说明为“%c”而不包括其后的 f；第二个格式说明为“%f”，不包括其后的 s；第三个格式说明为%s。其他的字符为原样输出的普通字符。

(4) 如果想输出字符“%”，则应该在“格式控制”字符串中用连续两个%表示，如：

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

输出：

```
0.333333%
```

4.5.2 scanf 函数

在第 1 章中已初步接触到了 scanf 函数(格式输入函数)，在本节中再作详细介绍。

1. 一般形式

scanf(格式控制，地址表列)

“格式控制”的含义同 printf 函数；“地址表列”是由若干个地址组成的表列，可以是变量的地址，或字符串的首地址。

例 4.9 用 scanf 函数输入数据。

```
#include <stdio.h>
void main()
{
  int a,b,c;
  scanf("%d%d%d",&a,&b,&c);
  printf("%d,%d,%d\n",a,b,c);
}
```

运行时按以下方式输入 a、b、c 的值：

```
3␣4␣5↙          (输入 a、b、c 的值)
3,4,5            (输出 a、b、c 的值)
```

&a、&b、&c 中的“&”是“地址运算符”，&a 指 a 在内存中的地址。上面 scanf 函数的作用是：按照 a、b、c 在内存的地址将 a、b、c 的值存进去，见图 4-2。变量 a、b、c 的地址是在编译连接阶段分配的。

“%d%d%d”表示要按十进制整数形式输入 3 个数据。输入数据时，在两个数据之间以一个或多个空格间隔，也可以用 Enter 键、Tab 键。下面输入均为合法：

① 3␣␣4␣␣␣␣5↙

② 3↙

4␣5↙

③ 3(按 Tab 键)4↙

5↙

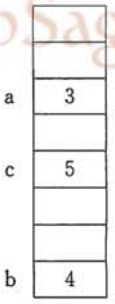


图 4-2

用“%d%d%d”格式输入数据时，不能用逗号作两个数据间的分隔符，如下面输入不合法：

3,4,5↙

2. 格式说明

与 printf 函数中的格式说明相似，以%开始，以一个格式字符结束，中间可以插入附加的字符。表 4-3 列出 scanf 用到的格式字符。表 4-4 列出 scanf 可以用的附加说明字符(修饰符)。

表 4-3 scanf 格式字符

格式字符	说　明
d,i	用来输入有符号的十进制整数
u	用来输入无符号的十进制整数
o	用来输入无符号的八进制整数
x, X	用来输入无符号的十六进制整数(大小写作用相同)
c	用来输入单个字符
s	用来输入字符串，将字符串送到一个字符数组中，在输入时以非空白字符开始，以第一个空白字符结束。字符串以串结束标志'\0'作为其最后一个字符
f	用来输入实数，可以用小数形式或指数形式输入
e, E, g, G	与 f 作用相同，e 与 f、g 可以互相替换(大小写作用相同)

表 4-4 scanf 的附加格式说明字符

字　符	说　明
l	用于输入长整型数据(可用%ld,%lo,%lx,%lu)以及 double 型数据(用%lf 或%le)
h	用于输入短整型数据(可用%hd,%ho,%hx)
域宽	指定输入数据所占宽度(列数)，域宽应为正整数
*	表示本输入项在读入后不赋给相应的变量

说明：

(1) 对 unsigned 型变量所需的数据，可以用%u、%d 或%o、%x 格式输入。

(2) 可以指定输入数据所占的列数，系统自动按它截取所需数据。例如：

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

输入：

```
123456↙
```

系统自动将 123 赋给变量 a，456 赋给变量 b。此方法也可用于字符型：

```
scanf("%3c",&ch);
```

如果从键盘连续输入 3 个字符“abc”，由于 ch 只能容纳一个字符，系统就把第一个字符'a'赋给字符变量 ch。

(3) 如果在%后有一个“*”附加说明符，表示跳过它指定的列数。例如：

```
scanf("%2d␣%*3d␣%2d",&a,&b);
```

如果输入如下信息：

```
12␣345␣67↙
```

系统会将 12 赋给整型变量 a，%*3d 表示读入 3 位整数但不赋给任何变量。然后再读入 2 位整数 67 赋给整型变量 b。也就是说第 2 个数据“345”被跳过。在利用现成的一批数据时，有时不需要其中某些数据，可用此法“跳过”它们。

(4) 输入数据时不能规定精度，例如：

```
scanf("%7.2f",&a);
```

是不合法的，不能企图用这样的 scanf 函数输入以下数据而使 a 的值为 12345.67。

```
1234567↙
```

3. 使用 scanf 函数时应注意的问题

(1) scanf 函数中的“格式控制”后面应当是变量地址，而不应是变量名。例如，若 a、b 为整型变量，则

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

是不对的，应将“a,b”改为“&a,&b”。这是 C 语言与其他高级语言不同之处。许多初学者常在此出错。

(2) 如果在“格式控制”字符串中除了格式说明以外还有其他字符，则在输入数据时在对应位置应输入与这些字符相同的字符。例如：

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

输入时应用如下形式：

```
3,4↙
```

注意：3 后面是逗号，它与 scanf 函数中的“格式控制”中的逗号对应。如果输入时不用逗号而用空格或其他字符是不对的：

3␣4↙　　（用空格分隔数据，与要求不符）

3:4↙　　（用冒号分隔数据，与要求不符）

如果是

```
scanf("%d␣␣%d",&a,&b);
```

由于在两个%d间有两个空格，因此在输入时，两个数据间应有2个或更多的空格字符。例如：

10␣␣34↙

或

10␣␣␣34↙

如果是

```
scanf("%d:%d:%d",&h,&m,&s);
```

输入应该用以下形式：

12:23:36↙

如果是

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

输入应为以下形式：

a=12,b=24,c=36↙

采用这种形式是为了使用户输入数据时添加必要的信息，使含义清楚，不易发生输入数据的错误。

(3) 在用"%c"格式输入字符时，空格字符和"转义字符"都作为有效字符输入，例如：

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

若输入

a␣b␣c↙

字符'a'送给c1，空格字符''送给c2，字符'b'送给c3。%c只要求读入一个字符，后面不需要用空格作为两个字符的间隔，因此''作为下一个字符送给c2。如果想将字符'a'、'b'、'c'分别赋给字符变量c1、c2、c3，正确的输入方法是：

abc↙　　（字符间没有空格）

(4) 在输入数据时，遇以下情况时认为该数据结束。

① 遇空格，或按"回车"或"跳格"(Tab)键；

② 按指定的宽度结束，如"%3d"，只取3列；

③ 遇非法输入。

例如：

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

若输入

```
1234  a  123o.26↙
 ↓    ↓    ↓
 a    b    c
```

第一个数据对应%d格式，在输入1234之后遇字符'a'，因此认为数值1234后已没有数字了，第一个数据到此结束，把1234送给变量a。字符'a'送给变量b，由于%c只要求输入一个字符，因此输入字符a之后不需要加空格，后面的数值应送给变量c。如果由于疏忽把本来应为1230.26错打成123o.26，由于123后面出现字符'o'，就认为该数值数据到此结束，将123送给c。

说明：在用Turbo C 2.0时，如果程序中包含printf和scanf函数，允许不加

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

例如，例4.9可以写为：

```
void main()
{
  int a,b,c;
  scanf("%d%d%d",&a,&b,&c);
  printf("%d,%d,%d\n",a,b,c);
}
```

在Turbo C 2.0中能通过编译。读者在阅读他人写的程序时，有可能遇到这种情况，对此应有所了解。但如果用Turbo C++ 3.0或Visual C++，程序是通不过编译的。为了稳妥和通用，建议读者在编程序时，只要用到系统库中的输入输出函数，都加上

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

4.6 顺序结构程序设计举例

下面介绍几个顺序程序设计的例子。

例4.10 输入三角形的三边长，求三角形面积。

为简单起见，设输入的三个边长a、b、c能构成三角形。从数学知识已知求三角形面积的公式为

$$\text{area}=\sqrt{s(s-a)(s-b)(s-c)}$$

其中$s=(a+b+c)/2$，据此编写程序如下：

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

```
void main()
    {
      float a,b,c,s,area;
      scanf("%f,%f,%f",&a,&b,&c);
      s=1.0/2*(a+b+c);
      area=sqrt(s*(s-a)*(s-b)*(s-c));
      printf("a=%7.2f,␣␣b=%7.2f,␣␣c=%7.2f,␣␣s=%7.2f\n",a,b,c,s);
      printf("area=%7.2f\n",area);
    }
```

程序第8行中sqrt函数是求平方根的函数。由于要调用数学函数库中的函数，必须在程序的开头加一条#include命令，把头文件“math.h”包含到程序中来。注意，以后凡在程序中要用到数学函数库中的函数，都应当“包含”math.h头文件。

运行情况如下：

```
3,4,6↙
a=␣␣␣3.00,␣␣b=␣␣␣4.00,␣␣c=␣␣␣6.00,␣␣s=␣␣␣6.50
area=␣␣␣5.33
```

例4.11 从键盘输入一个大写字母，要求改用小写字母输出。

前面已介绍过大小写字母间转换的方法，根据此思路编写出下面的程序：

```
#include <stdio.h>
void main()
{
    char c1,c2;
    c1=getchar();
    printf("%c,%d\n",c1,c1);
    c2=c1+32;
    printf("%c,%d\n",c2,c2);
}
```

运行情况如下：

```
A↙
A,65
a,97
```

用getchar函数得到从键盘上输入的字母'A'，赋给字符变量c1。将c1分别用字符形式('A')和整数形式(65)输出。再经过运算得到字母'a'，赋给字符变量c2，将c2分别用字符形式('a')和整数形式(97)输出。

例4.12 求$ax^2+bx+c=0$方程的根。a、b、c由键盘输入，设$b^2-4ac>0$。

众所周知，一元二次方程的根为：

$$x_1=\frac{-b+\sqrt{b^2-4ac}}{2a},\qquad x_2=\frac{-b-\sqrt{b^2-4ac}}{2a}$$

可以将上面的分式分为两项：

$$p=\frac{-b}{2a}\ ,\ q=\frac{\sqrt{b^2-4ac}}{2a}$$

$$x_1=p+q,\ x_2=p-q$$

据此编写程序如下：

```
#include <stdio.h>
#include <math.h>
void main()
{
    float a,b,c,disc,x1,x2,p,q;
    scanf("a=%f,b=%f,c=%f",&a,&b,&c);
    disc=b*b-4*a*c;
    p=-b/(2*a);
    q=sqrt(disc)/(2*a);
    x1=p+q; x2=p-q;
    printf("x1=%5.2f\nx2=%5.2f\n",x1,x2);
}
```

运行情况如下：

```
a=1,b=3,c=2↙
x1=-1.00
x2=-2.00
```

习　题

4.1　C语言中的语句有哪几类？C语句与其他语言中的语句有哪些异同？

4.2　怎样区分表达式和表达式语句？C语言为什么要设表达式语句？什么时候用表达式，什么时候用表达式语句？

4.3　C语言为什么要把输入输出的功能作为函数，而不作为语言的基本部分？

4.4　若 $a=3,b=4,c=5,x=1.2,y=2.4,z=-3.6,u=51274,n=128765,c1='a',c2='b'$。想得到以下的输出格式和结果，请写出程序（包括定义变量类型和设计输出）。

要求输出的结果如下：

```
a=␣3␣␣b=␣4␣␣c=␣5
x=1.200000,y=2.400000,z=-3.600000
x+y=␣3.60␣␣y+z=-1.20␣␣z+x=-2.40
u=␣51274␣␣n=␣␣␣128765
c1='a'␣or␣97(ASCII)
c2='B'␣or␣98(ASCII)
```

4.5　请写出下面程序的输出结果：

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

```
void main()
{
  int a=5,b=7;
  float x=67.8564, y=-789.124;
  char c='A';
  long n=1234567;
  unsigned u=65535;
  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,%x\n",n,n,n);
  printf("%u,%o,%x,%d\n",u,u,u,u);
  printf("%s,%5.3s\n","COMPUTER", "COMPUTER");
}
```

4.6 用下面的 scanf 函数输入数据，使 a=3，b=7，x=8.5，y=71.82，c1='A'，c2='a'。问在键盘上如何输入？

```
#include <stdio.h>
void main()
  {
    int a,b;
    float x,y;
    char c1,c2;
    scanf("a=%d␣b=%d",&a,&b);
    scanf("␣%f␣%e",&x,&y);
    scanf("␣%c␣%c",&c1,&c2);
  }
```

4.7 用下面的 scanf 函数输入数据，使 a=10，b=20，c1='A'，c2='a'，x=1.5，y=-3.75，z=67.8，请问在键盘上如何输入数据？

```
scanf("%5d%5d%c%c%f%f%*f,%f",&a,&b,&c1,&c2,&x,&y,&z);
```

4.8 设圆半径 $r=1.5$，圆柱高 $h=3$，求圆周长、圆面积、圆球表面积、圆球体积、圆柱体积。用 scanf 输入数据，输出计算结果，输出时要求有文字说明，取小数点后 2 位数字。请编程序。

4.9 输入一个华氏温度，要求输出摄氏温度。公式为

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

输出要有文字说明，取 2 位小数。

4.10 编程序，用 getchar 函数读入两个字符给 c1、c2，然后分别用 putchar 函数和 printf 函数输出这两个字符。思考以下问题：

(1) 变量 c1、c2 应定义为字符型或整型？或二者皆可？

(2) 要求输出 c1 和 c2 值的 ASCII 码，应如何处理？用 putchar 函数还是 printf 函数？

(3) 整型变量与字符变量是否在任何情况下都可以互相代替？如：

```
char c1,c2;
```

与

```
int c1,c2;
```

是否无条件地等价？

第5章　选择结构程序设计

在第2章中已介绍了选择结构，它是3种基本结构之一。在大多数程序中都会包含选择结构。它的作用是，根据所指定的条件是否满足，决定从给定的两组操作选择其一。在本章中介绍如何用C语言实现选择结构。

在C语言中选择结构是用if语句实现的。if语句最常用的形式如下：

if（关系表达式）语句1 else 语句2

例如：

```
if ( x>0 ) y=1; else y=-1;
```

其中x>0是一个关系表达式；“>”是一个关系运算符。

5.1　关系运算符和关系表达式

关系运算是逻辑运算中比较简单的一种。所谓“关系运算”实际上是“比较运算”。将两个值进行比较，判断其比较的结果是否符合给定的条件。例如，a>3是一个关系表达式，大于号(>)是一个关系运算符，如果a的值为5，则满足给定的“a>3”条件，因此关系表达式的值为“真”(即“条件满足”)；如果a的值为2，不满足“a>3”条件，则称关系表达式的值为“假”。

5.1.1　关系运算符及其优先次序

C语言提供6种关系运算符：

① <	(小于)	优先级相同（高）
② <=	(小于或等于)	
③ >	(大于)	
④ >=	(大于或等于)	
⑤ ==	(等于)	优先级相同（低）
⑥ !=	(不等于)	

关于优先次序：

(1) 前4种关系运算符(<，<=，>，>=)的优先级别相同，后2种也相同。前4种高于后2种。例如，“>”优先于“==”。而“>”与“<”优先级相同。

(2) 关系运算符的优先级低于算术运算符。

(3) 关系运算符的优先级高于赋值运算符。

以上关系见图5-1。

例如：

c>a+b 等效于 c>(a+b)

a>b==c 等效于 (a>b)==c

a==b<c 等效于 a==(b<c)

a=b>c 等效于 a=(b>c)

图 5-1

5.1.2 关系表达式

用关系运算符将两个表达式(可以是算术表达式或关系表达式、逻辑表达式、赋值表达式、字符表达式)连接起来的式子,称关系表达式。例如,下面都是合法的关系表达式:

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

关系表达式的值是一个逻辑值,即"真"或"假"。例如,关系表达式"5==3"的值为"假","5>=0"的值为"真"。C 语言没有逻辑型数据(C ++ 有逻辑型变量和逻辑型常量,以 True 表示"真",以 False 表示"假")。在 C 的逻辑运算中,以"1"代表"真",以"0"代表"假"。例如,a=3,b=2,c=1,则:

- 关系表达式"a>b"的值为"真",表达式的值为 1。
- 关系表达式"(a>b)==c"的值为"真"(因为 a>b 的值为 1,等于 c 的值),表达式的值为 1。
- 关系表达式"b+c<a"的值为"假",表达式的值为 0。

如果有以下赋值表达式:

- d=a>b　　则 d 的值为 1。
- f=a>b>c　　则 f 的值为 0(因为">"运算符是自左至右的结合方向,先执行"a>b"得值为 1,再执行关系运算"1>c",得值 0,赋给 f)。

5.2 逻辑运算符和逻辑表达式

用逻辑运算符将关系表达式或逻辑量连接起来的式子就是逻辑表达式。在 BASIC 和 Pascal 语言中有以下形式的逻辑表达式(AND 是逻辑运算符):

(a>b) AND (x>y)

如果 a>b 且 x>y,则上述逻辑表达式的值为"真"。下面介绍 C 语言中的逻辑运算符和逻辑运算。

5.2.1 逻辑运算符及其优先次序

C 语言提供 3 种逻辑运算符:

(1) && 逻辑与　　(相当于其他语言中的 AND)

(2) || 逻辑或　　(相当于其他语言中的 OR)

(3) ! 逻辑非　　(相当于其他语言中的 NOT)

“&&”和“||”是“双目(元)运算符”,它要求有两个运算量(操作数),如(a>b)&&(x>y),(a>b)||(x>y)。“!”是“一目(元)运算符”,只要求有一个运算量,如 !(a>b)。

逻辑运算举例如下:

a&&b　　若 a、b 为真,则 a&&b 为真。
a||b　　若 a、b 之一为真,则 a||b 为真。
!a　　若 a 为真,则!a 为假。

表 5-1 为逻辑运算的“真值表”。用它表示当 a 和 b 的值为不同组合时,各种逻辑运算所得到的值。

表 5-1　逻辑运算的真值表

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

在一个逻辑表达式中如果包含多个逻辑运算符,例如:

```
!a&&b||x>y&&c
```

按以下的优先次序:

(1) !(非)→&&(与)→||(或),即“!”为三者中最高的。

(2) 逻辑运算符中的“&&”和“||”低于关系运算符,“!”高于算术运算符,见图 5-2。

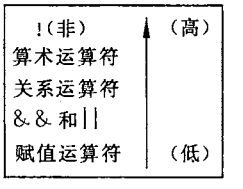


图　5-2

例如:

(a>b)&&(x>y)　　可写成 a>b&&x>y
(a==b)||(x==y)　　可写成 a==b||x==y
(!a)||(a>b)　　可写成 !a||a>b

5.2.2　逻辑表达式

如前所述,逻辑表达式的值应该是一个逻辑量“真”或“假”。C 语言编译系统在表示逻辑运算结果时,以数值 1 代表“真”,以 0 代表“假”,但在判断一个量是否为“真”时,以 0 代表“假”,以非 0 代表“真”。即将一个非零的数值认作为“真”。例如:

(1) 若 a=4,则!a 的值为 0。因为 a 的值为非 0,被认作“真”,对它进行“非”运算,得“假”,“假”以 0 代表。

(2) 若 a=4,b=5,则 a&&b 的值为 1。因为 a 和 b 均为非 0,被认为是“真”,因此 a&&b 的值也为“真”,值为 1。

(3) a、b 值分别为 4、5,a||b 的值为 1。

(4) a、b 值分别为 4、5,!a||b 的值为 1。

(5) 4&&0||2 的值为 1。

通过这几个例子可以看出，由系统给出的逻辑运算结果不是 0 就是 1，不可能是其他数值。而在逻辑表达式中作为参加逻辑运算的运算对象(操作数)可以是 0("假")或任何非 0 的数值(按"真"对待)。如果在一个表达式中不同位置上出现数值，应区分哪些是作为数值运算或关系运算的对象，哪些作为逻辑运算的对象。例如：

```
5>3 && 8<4-!0
```

表达式自左至右扫描求解。首先处理"5>3"(因为关系运算符优先于 &&)。在关系运算符两侧的 5 和 3 作为数值参加关系运算，"5>3"的值为 1(代表真)。再进行"1 && 8<4-! 0"的运算，8 的左侧为"&&"，右侧为"<"运算符，根据优先规则，应先进行"<"的运算，即先进行 8<4-! 0"的运算。现在 4 的左侧为"<"，右侧为"-"运算符，而"-"优先于"<"，因此应先进行"4-! 0"的运算，由于"!"的级别最高，因此先进行"! 0"的运算，得到结果 1。然后进行"4-1"的运算，得到结果 3，再进行"8<3"的运算，得 0，最后进行"1&&0"的运算，得 0。

实际上，逻辑运算符两侧的运算对象不但可以是 0 和 1，或者是 0 和非 0 的整数，也可以是字符型、实型或指针型等。系统最终以 0 和非 0 来判定它们属于"真"或"假"。例如：

```
'c' && 'd'
```

的值为 1(因为'c'和'd'的 ASCII 值都不为 0，按"真"处理)，所以 1 && 1 的值为 1。
可以将表 5-1 改写成表 5-2 形式。

表 5-2　逻辑运算的真值表

a	b	!a	!b	a && b	a || b
非 0	非 0	0	0	1	1
非 0	0	0	1	0	1
0	非 0	1	0	0	1
0	0	1	1	0	0

在逻辑表达式的求解中，并不是所有的逻辑运算符都被执行，只是在必须执行下一个逻辑运算符才能求出表达式的解时，才执行该运算符。举例如下。

(1) a && b && c 只有 a 为真(非 0)时，才需要判别 b 的值，只有 a 和 b 都为真的情况下才需要判别 c 的值。只要 a 为假，就不必判别 b 和 c(此时整个表达式已确定为假)。如果 a 为真，b 为假，不判别 c，见图 5-3。

(2) a||b||c 只要 a 为真(非 0)，就不必判断 b 和 c。只有 a 为假，才判别 b。a 和 b 都为假才判别 c，见图 5-4。

也就是说，对 && 运算符来说，只有 a≠0，才继续进行右面的运算。对||运算符来说，只有 a=0，才继续进行其右面的运算。因此，如果有下面的逻辑表达式：

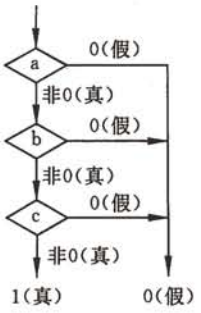


图 5-3

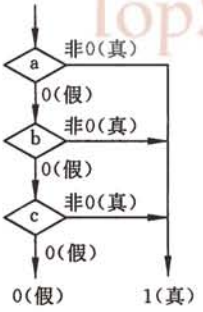


图 5-4

(m=a>b) && (n=c>d)

当 a=1,b=2,c=3,d=4,m 和 n 的原值为 1 时,由于"a>b"的值为 0,因此 m=0,而"n=c>d"不被执行,因此 n 的值不是 0 而仍保持原值 1。这点请读者注意。

熟练掌握 C 语言的关系运算符和逻辑运算符后,可以巧妙地用一个逻辑表达式来表示一个复杂的条件。

例如,要判别用 year 表示的某一年是否闰年。闰年的条件是符合下面二者之一:①能被 4 整除,但不能被 100 整除,如 2008。②能被 4 整除,又能被 400 整除,如 2000。可以用一个逻辑表达式来表示:

```
(year%4==0 && year%100!=0)||year%400==0
```

当 year 为某一整数值时,如果上述表达式值为真(1),则 year 为闰年;否则 year 为非闰年。

可以加一个"!"用来判别非闰年:

```
!((year%4==0 && year%100!=0)||year%400==0)
```

若此表达式值为真(1),year 为非闰年。也可以用下面逻辑表达式判别非闰年:

```
(year%4!=0)||(year%100==0 && year%400!=0)
```

若表达式值为真,year 为非闰年。请注意表达式中右面的括号内的不同运算符(%,!=,&&,==)的运算优先次序。

5.3 if 语 句

if 语句是用来判定所给定的条件是否满足,根据判定的结果(真或假)决定执行给出的两种操作之一。

5.3.1 if 语句的 3 种形式

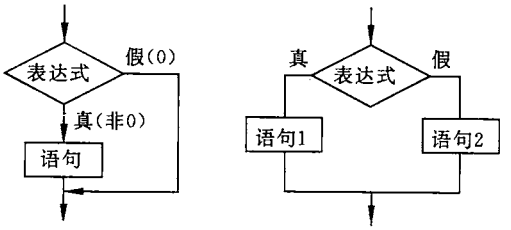
C 语言提供了 3 种形式的 if 语句。

1. if(表达式) 语句

例如：

```
if(x>y)printf("%d",x);
```

这种 if 语句的执行过程见图 5-5(a)。



(a)　　(b)

图　5-5

2. if(表达式)语句 1 else 语句 2

例如：

```
if (x>y)
  printf("%d",x);
else
  printf("%d",y);
```

见图 5-5(b)。

3. if(表达式 1) 语句 1
else if(表达式 2)语句 2
else if(表达式 3) 语句 3
⋮
else if(表达式 m) 语句 m
else 语句 n

流程图见图 5-6。

例如：

```
if (number>500) cost=0.15;
else if (number>300) cost=0.10;
else if (number>100) cost=0.075;
else if (number>50) cost=0.05;
else cost=0;
```

说明：

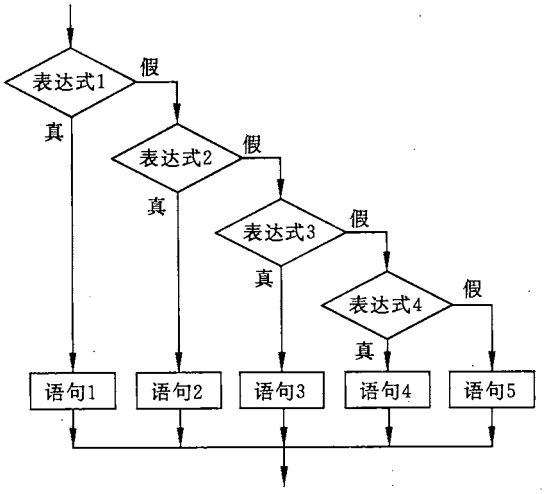


图 5-6

(1) 3 种形式的 if 语句中在 if 后面都有表达式，一般为逻辑表达式或关系表达式。例如：

```
if(a==b && x==y)printf("a=b,x=y");
```

在执行 if 语句时先对表达式求解，若表达式的值为 0，按“假”处理，若表达式的值为非 0，按“真”处理，执行指定的语句。假如有以下 if 语句：

```
if(3) printf("O. K.");
```

是合法的，执行结果输出“O. K.”，因为表达式的值为 3，按“真”处理。由此可见，表达式的类型不限于逻辑表达式，可以是任意的数值类型(包括整型、实型、字符型、指针型数据)。例如，下面的 if 语句也是合法的：

```
if('a') printf("%d",'a');
```

执行结果：输出'a'的 ASCII 码 97。

(2) 第二、第三种形式的 if 语句中，在每个 else 前面有一分号，整个语句结束处有一分号。例如：

```
if (x>0)
    print ("%f",x);
else                    各有一个分号;
    printf("%f", - x);
```

这是由于分号是 C 语句中不可缺少的部分，这个分号是 if 语句中的内嵌语句所要求的。如果无此分号，则出现语法错误。

注意：不要误认为上面是两个语句(if 语句和 else 语句)。它们都属于同一个 if 语句。else 子句不能作为语句单独使用，它必须是 if 语句的一部分，与 if 配对使用。

(3) 在 if 和 else 后面可以只含一个内嵌的操作语句(如上例),也可以有多个操作语句,此时用花括号"{}"将几个语句括起来成为一个复合语句。例如:

```
if (a+b>c && b+c>a && c+a>b)
  {
    s=0.5*(a+b+c);
    area=sqrt(s*(s-a)*(s-b)*(s-c));
    printf("area=%6.2f",area);
  }
else
  printf("it is not a trilateral");
```

注意在 else 上面的花括号"}"外面不需要再加分号。因为{}内是一个完整的复合语句,不需另附加分号。

例 5.1 输入两个实数,按代数值由小到大的顺序输出这两个数。

这个问题的算法很简单,只需要做一次比较即可。对类似这样简单的问题可以不必先写出算法或画流程图,而直接编写程序。或者说,算法在编程者的脑子里,相当于在算术运算中对简单的问题可以"心算"而不必在纸上写出来一样。

程序如下:

```
#include <stdio.h>
void main()
  {
    float a,b,t;
    scanf("%f,%f",&a,&b);
    if(a>b)
    {
      t=a;
      a=b;
      b=t;
    }
    printf("%5.2f,%5.2f\n",a,b);
  }
```

运行情况如下:

```
3.6,-3.2↙
-3.20,␣3.60
```

例 5.2 输入 3 个数 a、b、c,要求按由小到大的顺序输出。

解此题的算法比上一题稍复杂一些。可以用伪代码写出算法:

```
if a > b 将 a 和 b 对换        (a 是 a、b 中的小者)
if a > c 将 a 和 c 对换        (a 是 a、c 中的小者,因此 a 是三者中最小者)
if b > c 将 b 和 c 对换        (b 是 b、c 中的小者,也是三者中次小者)
```

然后顺序输出 a、b、c 即可。

按此算法编写程序：

```
#include <stdio.h>
void main()
{
    float a,b,c,t;
    scanf("%f,%f,%f",&a,&b,&c);
    if(a>b)
      {
        t=a;
        a=b;
        b=t;
      }                              /* 实现 a 和 b 的互换 */
    if(a>c)
      {
        t=a;
        a=c;
        c=t;
      }                              /* 实现 a 和 c 的互换 */
    if(b>c)
      {
        t=b;
        b=c;
        c=t;
      }                              /* 实现 b 和 c 的互换 */
    printf("%5.2f,%5.2f,%5.2f\n",a,b,c);
}
```

运行情况如下：

```
3,7,1↙
␣1.00,␣3.00,␣7.00
```

5.3.2 if 语句的嵌套

在 if 语句中又包含一个或多个 if 语句称为 if 语句的嵌套。一般形式如下：

```
if()
  if() 语句 1
  else 语句 2        内嵌 if
else
  if() 语句 3
  else 语句 4        内嵌 if
```

应当注意 if 与 else 的配对关系。else 总是与它上面的最近的未配对的 if 配对。假

如写成：

```
if()
  if()语句 1
else                    } 内嵌 if
  if() 语句 2
else 语句 3
```

编程序者把 else 写在与第一个 if(外层 if)同一列上，希望 else 与第一个 if 对应，但实际上 else 是与第二个 if 配对，因为它们相距最近。因此最好使内嵌 if 语句也包含 else 部分(如 5.3.2 小节最早列出的形式)，这样 if 的数目和 else 的数目相同，从内层到外层一一对应，不致出错。

如果 if 与 else 的数目不一样，为实现程序设计者的企图，可以加花括号来确定配对关系。例如：

```
if ()                   }
  {                     }
    if () 语句 1        } 内嵌 if
  }                     }
else      语句 2        }
```

这时“{ }”限定了内嵌 if 语句的范围，因此 else 与第一个 if 配对。

例 5.3 有一函数：

$$y=\begin{cases}-1 & (x<0)\\ 0 & (x=0)\\ 1 & (x>0)\end{cases}$$

编一程序，输入一个 x 值，输出 y 值。

可以先写出算法：

输入 x
若 x < 0，则 y = −1
若 x = 0，则 y = 0
若 x > 0，则 y = 1
输出 y

或：

输入 x
若 x < 0，则 y = −1
否则：
若 x = 0，则 y = 0
若 x > 0，则 y = 1
输出 y

也可以用流程图表示，见图 5-7。

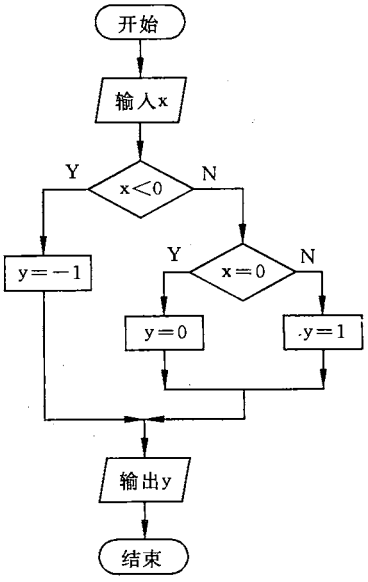


图 5-7

有以下几个程序，请读者判断哪个是正确的？

程序 1：

```
#include <stdio.h>
void main()
{
  int x,y;
  scanf("%d",&x);
  if(x<0)
    y=-1;
  else
    if(x==0) y=0;
    else y=1;
  printf("x=%d,y=%d\n",x,y);
}
```

程序 2：将上面程序的 if 语句(第 6～10 行)改为：

```
if (x>=0)
    if (x>0)  y= 1;
    else  y= 0;
else  y=-1;
```

程序 3：将上述 if 语句改为：

```
y=-1;
if(x! =0)
    if(x>0)  y=1;
else  y=0;
```

程序 4：

```
y=0;
if(x>=0)
    if(x>0) y=1;
else y=-1;
```

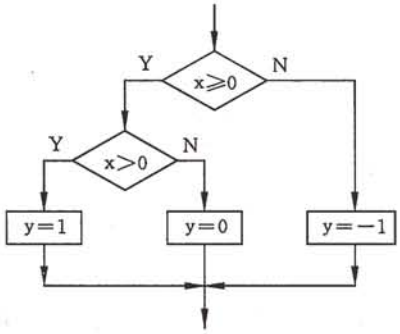


图 5-8

只有程序 1 和程序 2 是正确的。程序 1 体现了图 5-7 的流程，显然它是正确的。程序 2 的流程图见图 5-8，它也能实现题目的要求。程序 3 的流程图见图 5-9。程序 4 的流程图见图 5-10。它们不能实现题目的要求。请注意程序中的 else 与 if 的配对关系。例如程序 3 中的 else 子句是和它上一行的内嵌的 if 语句配对，而不与第 2 行的 if 语句配对。为了使逻辑关系清晰，避免出错，一般把内嵌的 if 语句放在外层的 else 子句中（如程序 1 那样），这样由于有外层的 else 相隔，内嵌的 else 不会被误认为和外层的 if 配对，而只能与内嵌的 if 配对，这样就不会搞混，如像程序 3 和程序 4 那样写就很容易出错。

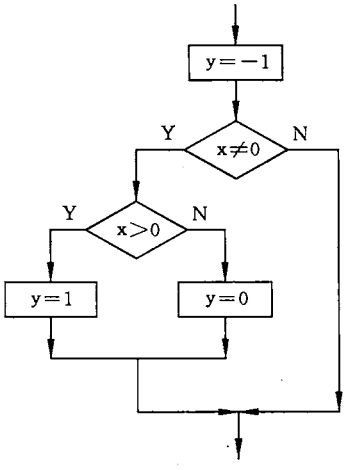


图 5-9

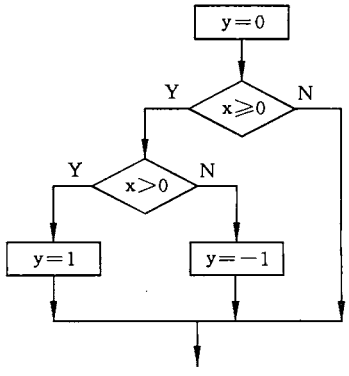


图 5-10

5.3.3 条件运算符

若在 if 语句中，当被判别的表达式的值为“真”或“假” 时，都执行一个赋值语句且向同一个变量赋值时，可以用一个条件运算符来处理。例如有以下 if 语句：

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

当 a＞b 时将 a 的值赋给 max，当 a≤b 时将 b 的值赋给 max，可以看到无论 a＞b 是否满足，都是向同一个变量赋值。可以用下面的条件运算符来处理：

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

其中“(a＞b)? a: b”是一个“条件表达式”。它是这样执行的：如果(a＞b)条件为真，则条件表达式取值 a；否则取值 b。

条件运算符要求有 3 个操作对象，称三目(元)运算符，它是 C 语言中惟一的一个三目运算符。条件表达式的一般形式为

表达式 1? 表达式 2: 表达式 3

它的执行过程见图 5-11。

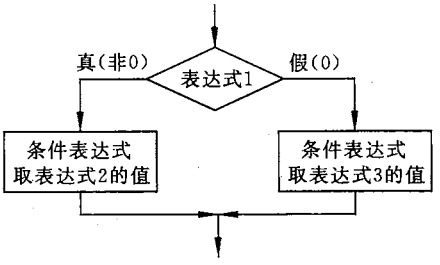


图 5-11

说明：

(1) 条件运算符的执行顺序：先求解表达式 1，若为非 0(真)则求解表达式 2，此时表达式 2 的值就作为整个条件表达式的值。若表达式 1 的值为 0(假)，则求解表达式 3，表达式 3 的值就是整个条件表达式的值。表达式

```
max=(a>b)? a: b
```

执行结果就是将条件表达式的值赋给 max，也就是将 a 和 b 二者中大者赋给 max。

(2) 条件运算符优先于赋值运算符，因此上面赋值表达式的求解过程是先求解条件表达式，再将它的值赋给 max。

条件运算符的优先级别比关系运算符和算术运算符都低。因此，

```
max=(a>b)? a: b
```

括号可以不要，可写成

```
max=a>b? a: b
```

如果有

```
a>b? a: b+1
```

相当于 a>b? a: (b+1)，而不相当于(a>b? a: b)+1。

(3) 条件运算符的结合方向为“自右至左”。如果有以下条件表达式：

```
a>b? a: c>d? c: d
```

相当于 a>b? a: (c>d? c: d)

如果 a=1，b=2，c=3，d=4，则条件表达式的值等于 4。

(4) 条件表达式还可以写成以下形式：

```
a>b? (a=100):(b=100)
```

或

```
a>b? printf("%d",a): printf ("%d", b)
```

即“表达式 2”和“表达式 3”不仅可以是数值表达式，还可以是赋值表达式或函数表达式。

上面第二个条件表达式相当于以下 if…else 语句：

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

(5) 条件表达式中，表达式 1 的类型可以与表达式 2 和表达式 3 的类型不同。例如：

```
x? 'a': 'b'
```

整型变量 x 的值若等于 0，则条件表达式的值为'b'。表达式 2 和表达式 3 的类型也可以不同，此时条件表达式的值的类型为二者中较高的类型。例如：

```
x>y? 1: 1.5
```

如果 x≤y，则条件表达式的值为 1.5；若 x>y，值应为 1，由于 1.5 是实型，比整型高，因此，将 1 转换成实型值 1.0。

例 5.4 输入一个字符，判别它是否大写字母，如果是，将它转换成小写字母；如果不是，不转换。然后输出最后得到的字符。

关于大小写字母之间的转换方法，在本书例 3.7 中已做了介绍，因此可直接编写程序：

```
#include <stdio.h>
void main()
  {
    char ch;
    scanf("%c",&ch);
    ch=(ch>='A'&& ch<='Z')? (ch+32):ch;
    printf("%c\n",ch);
  }
```

运行结果如下：

```
A↙
a
```

条件表达式"(ch>='A' && ch<='Z')? (ch+32): ch"的作用是：如果字符变量 ch 的值为大写字母，则条件表达式的值为(ch+32)，即相应的小写字母，32 是小写字母和大写字母 ASCII 码的差值。如果 ch 的值不是大写字母，则条件表达式的值为 ch，即不进行转换。

5.4 switch 语句

switch 语句是多分支选择语句。用来实现如图 2-23 所表示的多分支选择结构。if 语句只有两个分支可供选择，而实际问题中常常需要用到多分支的选择。例如，学生成绩

分类(85分以上为'A'等,70～84分为'B'等,60～69分为'C'等……);人口统计分类(按年龄分为老、中、青、少、儿童);工资统计分类;银行存款分类……

当然这些都可以用嵌套的if语句来处理,但如果分支较多,则嵌套的if语句层数多,程序冗长而且可读性降低。C语言提供switch语句直接处理多分支选择,它的一般形式如下:

```
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
  }
```

例如,要求按照考试成绩的等级输出百分制分数段,可以用switch语句实现:

```
switch(grade)
  {
    case 'A': printf("85～100\n");
    case 'B': printf("70～84\n");
    case 'C': printf("60～69\n");
    case 'D': printf("<60\n");
    default:  printf("error\n");
  }
```

说明:

(1) switch后面括号内的"表达式",ANSI标准允许它为任何类型。

(2) 当表达式的值与某一个case后面的常量表达式的值相等时,就执行此case后面的语句,若所有的case中的常量表达式的值都没有与表达式的值匹配的,就执行default后面的语句。

(3) 每一个case的常量表达式的值必须互不相同;否则就会出现互相矛盾的现象(对表达式的同一个值,有两种或多种执行方案)。

(4) 各个case和default的出现次序不影响执行结果。例如,可以先出现"default:…",再出现"case 'D':…",然后是"case 'A':…"。

(5) 执行完一个case后面的语句后,流程控制转移到下一个case继续执行。"case常量表达式"只是起语句标号作用,并不是在该处进行条件判断。在执行switch语句时,根据switch后面表达式的值找到匹配的入口标号,就从此标号开始执行下去,不再进行判断。例如,上面的例子中,若grade的值等于'A',则将连续输出:

```
85～100
70～84
60～69
<60
```

error

因此，应该在执行一个 case 分支后，使流程跳出 switch 结构，即终止 switch 语句的执行。可以用一个 break 语句来达到此目的。将上面的 switch 结构改写如下：

```
switch(grade)
  {
    case 'A': printf("85～100\n"); break;
    case 'B': printf("70～84\n");break;
    case 'C': printf("60～69\n"); break;
    case 'D': printf("＜60\n");break;
    default : printf("error\n");
  }
```

最后一个分支(default)可以不加 break 语句。如果 grade 的值为'B'，则只输出“70～84”，流程图见图 5-12。

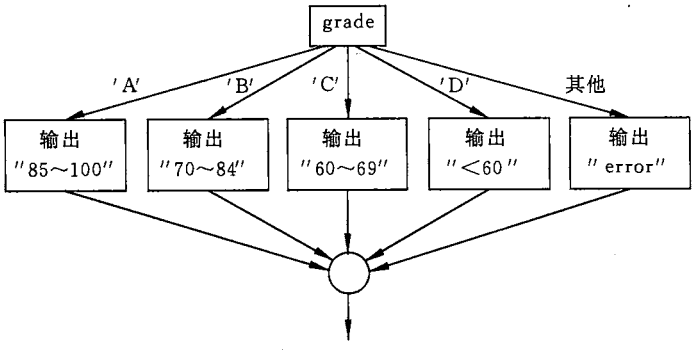


图 5-12

在 case 后面虽然包含了一个以上执行语句，但可以不必用花括号括起来，会自动顺序执行本 case 后面所有的执行语句。当然加上花括号也可以。

(6) 多个 case 可以共用一组执行语句，例如：

```
  ⋮
case 'A':
case 'B':
case 'C': printf(">60\n");
break;
  ⋮
```

grade 的值为'A'、'B'或'C'时都执行同一组语句。

5.5 程序举例

例 5.5 写程序，判断某一年是否闰年。

在第 2 章的例 2.3、例 2.13 曾介绍过判别闰年的方法。现在用图 5-13 来表示判别闰

年的算法。它的思路与图 2-32 相类似。只是以变量 leap 代表是否闰年的信息。若闰年，令 leap＝1；非闰年，leap＝0。最后判断 leap 是否为 1(真)，若是，则输出“闰年”信息。

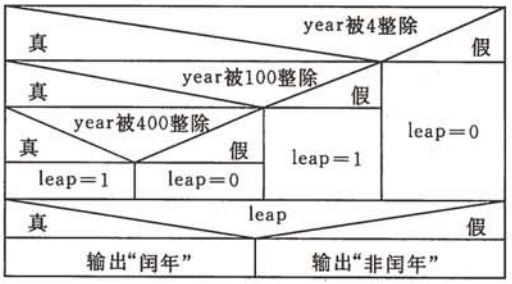


图　5-13

据此编写程序如下：

```
#include <stdio.h>
void main()
  {
    int year, leap;
    scanf("%d",&year);
    if (year%4==0)
      {
        if (year%100==0)
          {
            if (year%400==0)
                leap=1;
            else
                leap=0;
          }
        else
          leap=1;
      }
    else
        leap=0;
    if (leap)
        printf("%d is ",year);
    else
        printf("%d is not ",year);
    printf("a leap year.\n");
  }
```

运行情况如下：

① 1989↙

1989 is not a leap year.

② 2000↙

2000 is a leap year.

也可以将程序中第 6～19 行改写成以下的 if 语句：

```
if (year%4! =0)
  leap=0;
else if(year%100! =0)
  leap=1;
else if(year%400! =0)
  leap=0;
else
  leap=1;
```

也可以用一个逻辑表达式包含所有的闰年条件，将上述 if 语句用下面的 if 语句代替：

```
if((year%4==0 && year%100!=0) || (year%400==0))
    leap=1;
else
    leap=0;
```

例 5.6 求 $ax^2+bx+c=0$ 方程的解。

例 4.12 曾介绍过基本的算法，实际上应该有以下几种可能。

① $a=0$，不是二次方程。

② $b^2-4ac=0$，有两个相等实根。

③ $b^2-4ac>0$，有两个不等实根。

④ $b^2-4ac<0$，有两个共轭复根。

画出 N-S 流程图表示算法(图 5-14)。

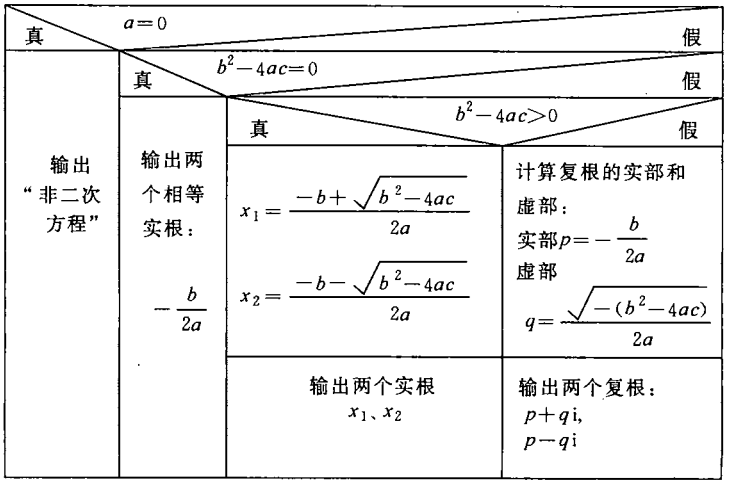


图 5-14

据此编写程序如下：

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

```
    float a,b,c,disc,x1,x2,realpart,imagpart;
    scanf("%f,%f,%f",&a,&b,&c);
    printf("The equation ");
    if(fabs(a)<=1e-6)
        printf("is not a quadratic\n");
    else
      {
        disc=b*b-4*a*c;
        if(fabs(disc)<=1e-6)
            printf("has two equal roots:%8.4f\n",-b/(2*a));
        else
          if(disc>1e-6)
           {
            x1=(-b+sqrt(disc))/(2*a);
            x2=(-b-sqrt(disc))/(2*a);
            printf("has distinct real roots:%8.4f and %8.4f\n",x1,x2);
           }
          else
           {
            realpart=-b/(2*a);
            imagpart=sqrt(-disc)/(2*a);
            printf(" has complex roots: \n");
            printf("%8.4f+%8.4fi\n",realpart,imagpart);
            printf("%8.4f-%8.4fi\n",realpart,imagpart);
           }
      }
}
```

程序中用 disc 代表 b^2-4ac，先计算 disc 的值，以减少以后的重复计算。对于判断 b^2-4ac 是否等于 0 时，要注意：由于 disc(即 b^2-4ac)是实数，而实数在计算和存储时会有一些微小的误差，因此不能直接进行如下判断："if(disc==0)…"，因为这样可能会出现本来是零的量，由于上述误差而被判别为不等于零而导致结果错误。所以采取的办法是判别 disc 的绝对值(fabs(disc))是否小于一个很小的数(例如 10^{-6})，如果小于此数，就认为 disc 等于 0。程序中以 realpart 代表实部 p，以 imagpart 代表虚部 q，以增加可读性。

运行情况如下：

① 1,2,1↙

The equation has two equal roots: -1.0000

② 1,2,2↙

The equation has complex roots:

-1.0000+1.0000i

-1.0000-1.0000i

③ 2,6,1↙

The equation has distinct real roots: -0.1771 and -2.8229

例 5.7 运输公司对用户计算运费。路程(skm)越远,每吨·千米运费越低。标准如下:

$s<250$	没有折扣
$250\leqslant s<500$	2%折扣
$500\leqslant s<1000$	5%折扣
$1000\leqslant s<2000$	8%折扣
$2000\leqslant s<3000$	10%折扣
$3000\leqslant s$	15%折扣

设每吨每千米货物的基本运费为 p(price 的缩写),货物重为 w(weight 的缩写),距离为 s,折扣为 d(discount 的缩写),则总运费 f(freight 的缩写)的计算公式为

$$f=p\times w\times s\times(1-d)$$

分析此问题,折扣的变化是有规律的:从图 5-15 可以看到,折扣的"变化点"都是 250 的倍数(250,500,1000,2000,3000)。利用这一特点,可以在横轴上加一种坐标 c,c 的值为 $s/250$。c 代表 250 的倍数。当 $c<1$ 时,表示 $s<250$,无折扣;$1\leqslant c<2$ 时,表示 $250\leqslant s<500$,折扣 $d=2\%$;$2\leqslant c<4$ 时,$d=5\%$;$4\leqslant c<8$ 时,$d=8\%$;$8\leqslant c<12$ 时,$d=10\%$;$c\geqslant 12$ 时,$d=15\%$。

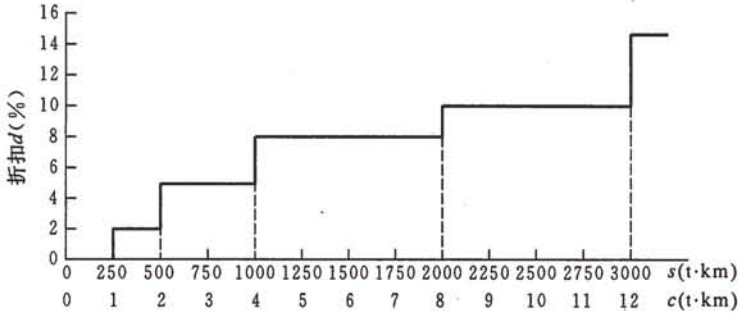


图 5-15

据此写出程序如下:

```
#include <stdio.h>
void main()
  {
    int c,s;
    float p,w,d,f;
    scanf("%f,%f,%d",&p,&w,&s);
    if(s>=3000) c=12;
    else c=s/250;
    switch(c)
    {
      case 0: d=0;break;
      case 1: d=2;break;
```

```
        case 2:
        case 3: d=5;break;
        case 4:
        case 5:
        case 6:
        case 7: d=8;break;
        case 8:
        case 9:
        case 10:
        case 11: d=10;break;
        case 12: d=15;break;
    }
    f=p*w*s*(1-d/100.0);
    printf("freight=%15.4f\n",f);
}
```

运行情况如下：

```
100,20,300↙
freight=␣␣␣␣588000.0000
```

注意：c、s是整型变量，因此c=s/250为整数。当s≥3000时，令c=12，而不使c随s增大，这是为了在switch语句中便于处理，用一个case可以处理所有s≥3000的情况。

习　题

5.1 什么是算术运算？什么是关系运算？什么是逻辑运算？

5.2 C语言中如何表示“真”和“假”？系统如何判断一个量的“真”和“假”？

5.3 写出下面各逻辑表达式的值。设a=3,b=4,c=5。

(1) a+b>c && b==c

(2) a||b+c && b-c

(3) !(a>b) && !c||1

(4) !(x=a) && (y=b) && 0

(5) !(a+b)+c-1 && b+c/2

5.4 有3个整数a、b、c，由键盘输入，输出其中最大的数。

5.5 有一个函数：

$$y=\begin{cases} x & x<1 \\ 2x-1 & 1\leqslant x<10 \\ 3x-11 & x\geqslant 10 \end{cases}$$

写一段程序，输入x，输出y值。

5.6 给出一百分制成绩，要求输出成绩等级'A'、'B'、'C'、'D'、'E'。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'。

5.7 给一个不多于 5 位的正整数，要求：

① 求出它是几位数；

② 分别输出每一位数字；

③ 按逆序输出各位数字，例如原数为 321，应输出 123。

5.8 企业发放的奖金根据利润提成。利润 I 低于或等于 100 000 元的，奖金可提 10%；利润高于 100 000 元，低于 200 000 元(100 000$<I\leqslant$200 000)时，低于 100 000 元的部分按 10%提成，高于 100 000 元的部分，可提成 7.5%；200 000$<I\leqslant$400 000 时，低于 200 000 元的部分仍按上述办法提成(下同)。高于 200 000 元的部分按 5%提成；400 000$<I\leqslant$600 000 元时，高于 400 000 元的部分按 3%提成；600 000$<I\leqslant$1000 000 时，高于 600 000 元的部分按 1.5%提成；$I>$1000 000 时，超过 100 0000 元的部分按 1%提成。从键盘输入当月利润 I，求应发奖金总数。

要求：

(1) 用 if 语句编程序；

(2) 用 switch 语句编程序。

5.9 输入 4 个整数，要求按由小到大的顺序输出。

5.10 有 4 个圆塔，圆心分别为(2,2)、(−2,2)、(−2,−2)、(2,−2)，圆半径为 1，见图 5-16。这 4 个塔的高度为 10m，塔以外无建筑物。今输入任一点的坐标，求该点的建筑高度(塔外的高度为零)。

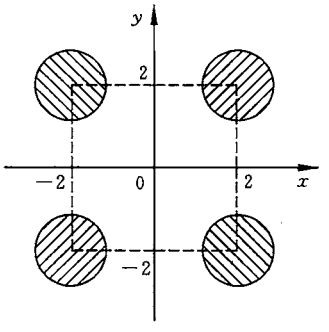


图 5-16

第6章 循环控制

6.1 概 述

在许多问题中需要用到循环控制。例如，要输入全校学生成绩；求若干个数之和；迭代求根等。绝大多数的应用程序都包含循环。循环结构是结构化程序设计的基本结构之一，它和顺序结构、选择结构共同作为各种复杂程序的基本构造单元。因此熟练掌握选择结构和循环结构的概念及使用是程序设计的最基本的要求。

6.2 goto 语句以及用 goto 语句构成循环

goto 语句为无条件转向语句，它的一般形式为

goto 语句标号；

语句标号用标识符表示，它的定名规则与变量名相同，即由字母、数字和下划线组成，其第一个字符必须为字母或下划线。不能用整数来作标号。例如：

```
goto label_1;
```

是合法的，而

```
goto 123;
```

是不合法的。结构化程序设计方法主张限制使用 goto 语句，因为滥用 goto 语句将使程序流程无规律、可读性差。但也不是绝对禁止使用 goto 语句。一般来说，有两种用途：

(1) 与 if 语句一起构成循环结构。

(2) 从循环体中跳转到循环体外，但在 C 语言中可以用 break 语句和 continue 语句(见 6.8 节)跳出本层循环和结束本次循环。goto 语句的使用机会已大大减少，只是需要从多层循环的内层循环跳到外层循环外时才用到 goto 语句。但是这种用法不符合结构化原则，一般不宜采用，只有在不得已时(例如能大大提高效率)才使用。

例 6.1 用 if 语句和 goto 语句构成循环，求 $\sum_{n=1}^{100} n$。

此问题的算法比较简单，可以直接写出程序：

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

```
    loop: if(i<=100)
          {
            sum=sum+i;
            i++;
            goto loop;
          }
    printf("%d\n",sum);
}
```

运行结果如下：

```
5050
```

这里用的是“当型”循环结构，当满足“i<=100”时执行花括号内的循环体。请读者自己画出流程图。

这个例子只是说明 goto 语句的用法以及可以用 goto 语句和 goto 语句构成循环，读者在阅读别人写的程序如遇到类似用法时，可心中有数。在一般情况下不提倡使用。

6.3 用 while 语句实现循环

while 语句用来实现“当型”循环结构。其一般形式如下：

while（表达式）语句

当表达式为非 0 值时，执行 while 语句中的内嵌语句，其流程图见图 6-1。其特点是：先判断表达式，后执行语句。

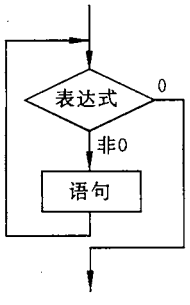


图 6-1

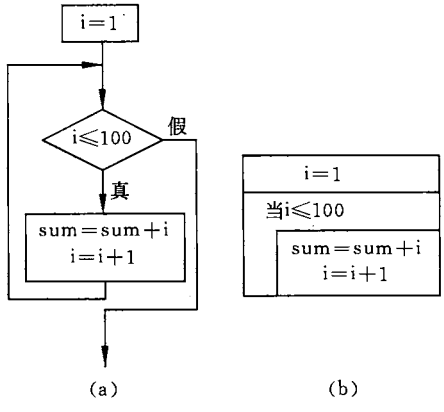


图 6-2

例 6.2 求 $\sum_{n=1}^{100} n$。

用传统流程图和 N-S 结构流程图表示算法，见图 6-2(a)和图 6-2(b)。

根据流程图写出程序：

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

```
void main()
{
  int i,sum=0;
  i=1;
  while (i<=100)
    {
      sum=sum+i;
      i++;
    }
  printf("%d\n",sum);
}
```

注意：

(1) 循环体如果包含一个以上的语句，应该用花括号括起来，以复合语句形式出现。如果不加花括号，则 while 语句的范围只到 while 后面第一个分号处。例如，本例中 while 语句中如无花括号，则 while 语句范围只到“sum=sum+i;”。

(2) 在循环体中应有使循环趋向于结束的语句。例如，在本例中循环结束的条件是“i>100”，因此在循环体中应该有使 i 增值以最终导致 i>100 的语句，今用“i++;”语句来达到此目的。如果无此语句，则 i 的值始终不改变，循环永不结束。

6.4 用 do…while 语句实现循环

do…while 语句的特点是先执行循环体，然后判断循环条件是否成立。其一般形式为：

do

循环体语句

while (表达式);

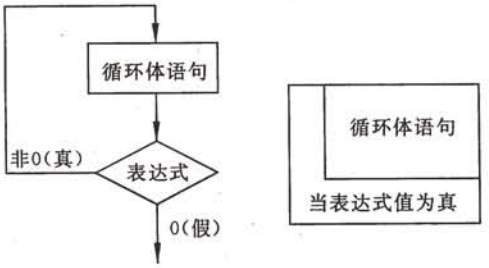
它是这样执行的：先执行一次指定的循环体语句，然后判别表达式，当表达式的值为非零(“真”) 时，返回重新执行循环体语句，如此反复，直到表达式的值等于 0 为止，此时循环结束。可以用图 6-3 表示其流程。请注意 do…while 循环用 N-S 流程图的表示形式(图 6-3(b))。

例 6.3 用 do…while 语句求 $\sum_{n=1}^{100} n$。

先画出流程图，见图 6-4。程序如下：

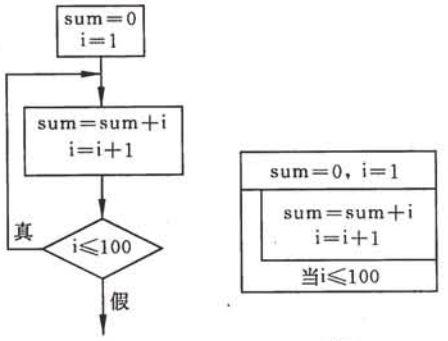
```
#include <stdio.h>
void main()
{
  int i,sum=0;
  i=1;
```

```
    do
        {
            sum=sum+i;
            i++;
        }
        while(i<=100);
        printf("%d\n",sum);
}
```



(a) (b)

图 6-3



(a) (b)

图 6-4

可以看到:对同一个问题可以用 while 语句处理,也可以用do…while语句处理。do…while语句结构可以转换成 while 结构。图 6-3 可以改画成图 6-5 形式,二者完全等价。而图 6-5 中线框部分就是一个 while 结构。可见,do…while 结构是由一个语句加一个 while 结构构成的。若图 6-1 中表达式值为真,则图 6-1 也与图 6-5 等价(因为都要先执行一次语句)。

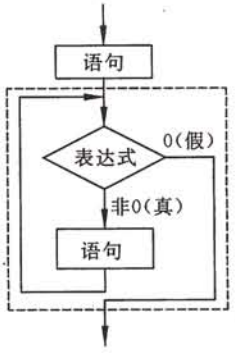


图 6-5

在一般情况下,用 while 语句和用 do…while 语句处理同一问题时,若二者的循环体部分是一样的,它们的结果也一样。如例 6.2 和例 6.3 程序中的循环体是相同的,得到结果也相同。但是如果 while 后面的表达式一开始就为假(0 值)时,两种循环的结果是不同的。

例 6.4 while 和 do…while 循环的比较。

(1)
```
#include <stdio.h>
void main ( )
{
  int sum=0,i;
  scanf("%d",&i);
  while (i<=10)
    {
      sum=sum+i;
      i++;
    }

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

运行情况如下：

1↙
sum=55

再运行一次：

11↙
sum=0

(2)
```
#include <stdio.h>
void main( )
{
  int sum=0,i;
  scanf("%d",&i);
  do
    {
      sum=sum+i;
      i++;
    }
  while (i<=10);
  printf("sum=%d\n",sum);
}
```

运行情况如下：

1↙
sum=55

再运行一次：

11↙
sum=11

可以看到：当输入 i 的值小于或等于 10 时，二者得到结果相同。而当 i>10 时，二者结果就不同了。这是因为此时对 while 循环来说，一次也不执行循环体(表达式“i<=10”为假)，而对 do…while 循环语句来说则要执行一次循环体。可以得到结论：当 while 后面的表达式的第一次的值为“真”时，两种循环得到的结果相同；否则，二者结果不相同(指二者具有相同的循环体的情况)。

do…while 循环是先执行循环体，后判断表达式的“当型”循环(因为当条件满足时才执行循环体)。但利用它可以方便地实现如第 2 章图 2-27 所示的典型的“直到型”循环结构。典型的直到型(until 型)循环结构是在表达式为真时结束循环(见图 2-29、图 2-30、图 2-31)。因此在将图 2-29 的算法用 do…while 循环表示时，应将条件取“反”，即将图 2-29 中的 “i>5”改为“i<=5”。因为“当 i≤5 时继续执行循环”和“直到 i>5 结束循环”是对同一问题的两种表述方式。可以用下面的 C 程序实现图 2-29 的算法：

```
#include <stdio.h>
void main( )
{
  int t=1,i=2;
  do
    {
     t=t*i;
     i++;
    }while (i<=5);
  printf("t=%d\n",t);
}
```

6.5 用 for 语句实现循环

C 语言中的 for 语句使用最为灵活，不仅可以用于循环次数已经确定的情况，而且可以用于循环次数不确定而只给出循环结束条件的情况，它完全可以代替 while 语句。

for 语句的一般形式为

for(表达式 1;表达式 2;表达式 3) 语句

它的执行过程如下：

(1) 先求解表达式 1。

(2) 求解表达式 2，若其值为真(值为非 0)，则执行 for 语句中指定的内嵌语句，然后执行下面第(3)步。若为假(值为 0)，则结束循环，转到第(5)步。

(3) 求解表达式 3。

(4) 转回上面第(2)步骤继续执行。

(5) 循环结束，执行 for 语句下面的一个语句。

可以用图 6-6 来表示 for 语句的执行过程。

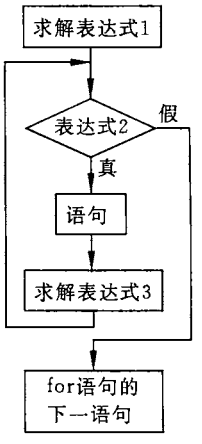


图 6-6

for 语句最简单的应用形式也就是最易理解的如下形式：

for(循环变量赋初值;循环条件;循环变量增值) 语句

例如：

```
for(i=1;i<=100;i++) sum=sum+i;
```

的执行过程与图 6-2 完全一样。它相当于以下语句：

```
i=1;
while(i<=100)
{
  sum=sum+i;
  i++;
}
```

显然，用 for 语句简单、方便。对于以上 for 语句的一般形式也可以改写为 while 循环的形式：

```
表达式 1;
while 表达式 2
{
  语句
  表达式 3;
}
```

说明：

(1) for 语句的一般形式中的“表达式 1”可以省略，此时应在 for 语句之前给循环变量赋初值。注意，省略表达式 1 时，其后的分号不能省略。例如：

```
for(;i<=100;i++) sum=sum+i;
```

执行时，跳过“求解表达式 1”这一步，其他不变。

(2) 如果表达式 2 省略，即不判断循环条件，循环无终止地进行下去。也就是认为表达式 2 始终为真，见图 6-7。

例如：

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

表达式 1 是一个赋值表达式，表达式 2 空缺。它相当于：

```
i=1;
while(1)
    {
      sum=sum+i;
      i++;
    }
```

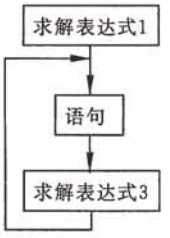


图 6-7

(3) 表达式 3 也可以省略，但此时程序设计者应另外设法保证循环能正常结束。例如：

```
for(i=1;i<=100;)
    {
      sum=sum+i;
      i++;
    }
```

在上面的 for 语句中只有表达式 1 和表达式 2，而没有表达式 3。i++的操作不放在 for 语句的表达式 3 的位置处，而作为循环体的一部分，效果是一样的，都能使循环正常结束。

(4) 可以省略表达式 1 和表达式 3，只有表达式 2，即只给循环条件。例如：

```
for(;i<=100;)                          while(i<=100)
  {                                      {
    sum=sum+i;          相当于             sum=sum+i;
    i++;                                   i++;
  }                                      }
```

在这种情况下，完全等同于 while 语句。可见 for 语句比 while 语句功能强，除了可以给出循环条件外，还可以赋初值，使循环变量自动增值等。

(5) 3 个表达式都可省略，例如：

```
for(; ;) 语句
```

相当于

```
while(1) 语句
```

即不设初值，不判断条件（认为表达式 2 为真值），循环变量不增值。无终止地执行循环体。

（6）表达式 1 可以是设置循环变量初值的赋值表达式，也可以是与循环变量无关的其他表达式。例如：

```
for (sum=0;i<=100;i++) sum=sum+i;
```

表达式 3 也可以是与循环控制无关的任意表达式。

表达式 1 和表达式 3 可以是一个简单的表达式，也可以是逗号表达式，即包含一个以上的简单表达式，中间用逗号间隔。例如：

```
for(sum=0,i=1;i<=100;i++) sum=sum+i;
```

或

```
for(i=0,j=100;i<=j;i++,j--) k=i+j;
```

表达式 1 和表达式 3 都是逗号表达式，各包含两个赋值表达式，即同时设两个初值，使两个变量增值，执行情况见图 6-8。

在逗号表达式内按自左至右顺序求解，整个逗号表达式的值为其中最右边的表达式的值。例如：

```
for(i=1;i<=100;i++,i++) sum=sum+i;
```

相当于

```
for(i=1;i<=100;i=i+2) sum=sum+i;
```

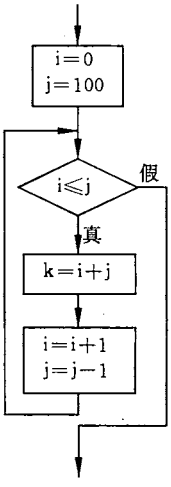


图 6-8

（7）表达式一般是关系表达式（如 i<=100）或逻辑表达式（如 a<b && x<y），但也可以是数值表达式或字符表达式，只要其值为非零，就执行循环体。分析下面两个例子：

```
① for(i=0;(c=getchar())! ='\n';i+=c);
```

在表达式 2 中先从终端接收一个字符赋给 c，然后判断此赋值表达式的值是否不等于'\n'（换行符），如果不等于'\n'，就执行循环体。此 for 语句的执行过程见图 6-9，它的作用是不断输入字符，将它们的 ASCII 码相加，直到输入一个“换行”符为止。

注意：此 for 语句的循环体为空语句，把本来要在循环体内处理的内容放在表达式 3 中，作用是一样的。可见 for 语句功能强，可以在表达式中完成本来应在循环体内完成的操作。

```
② for( ;(c=getchar())! ='\n';)
    printf("%c",c);
```

for 语句中只有表达式 2，而无表达式 1 和表达式 3。其作用是每读入一个字符后立即输出该字符，直到输入一个“换行”为止。请注意，从终端键盘向计算机输入时，是在按 Enter 键以后才将一批数据一起送到内存缓冲区中去的。

运行情况：

Computer↙　　（输入）
Computer　　（输出）

注意输出结果不是

Ccoommppuutteerr

即不是从终端敲入一个字符马上输出一个字符，而是按 Enter 键后数据送入内存缓冲区，然后每次从缓冲区读一个字符，再输出该字符。

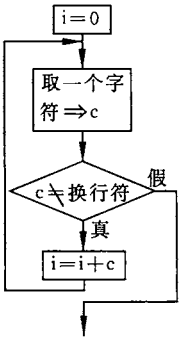


图　6-9

从上面介绍可以知道 C 语言中的 for 语句比其他语言（如 BASIC、Pascal）中的 for 语句功能强得多。可以把循环体和一些与循环控制无关的操作也作为表达式 1 或表达式 3 出现，这样程序可以短小简洁。但过分地利用这一特点会使 for 语句显得杂乱，可读性降低，最好不要把与循环控制无关的内容放到 for 语句中。

6.6　循环的嵌套

一个循环体内又包含另一个完整的循环结构，称为循环的嵌套。内嵌的循环中还可以嵌套循环，这就是多层循环。各种语言中关于循环的嵌套的概念都是一样的。

3 种循环（while 循环、do…while 循环和 for 循环）可以互相嵌套。例如，下面几种都是合法的形式：

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

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

```
(3) for(;;)
      {
       for(;;)
        {…}
      }
```

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

```
(5) for(;;)
      {⋮
       while()
          {…}
        ⋮
      }
```

```
(6) do
      {
        ⋮
       for(;;)
          {…}
      }
    while();
```

6.7 几种循环的比较

(1) 4 种循环都可以用来处理同一问题，一般情况下它们可以互相代替。但一般不提倡用 goto 型循环。

(2) 在 while 循环和 do…while 循环中，只在 while 后面的括号内指定循环条件，因此为了使循环能正常结束，应在循环体中包含使循环趋于结束的语句(如 i++，或 i=i+1 等)。

for 循环可以在表达式 3 中包含使循环趋于结束的操作，甚至可以将循环体中的操作全部放到表达式 3 中。因此 for 语句的功能更强，凡用 while 循环能完成的，用 for 循环都能实现。

(3) 用 while 和 do…while 循环时，循环变量初始化的操作应在 while 和 do…while 语句之前完成。而 for 语句可以在表达式 1 中实现循环变量的初始化。

(4) while 循环、do…while 循环和 for 循环，可以用 break 语句跳出循环，用 continue 语句结束本次循环(break 语句和 continue 语句见 6.8 节)。而对用 goto 语句和 if 语句构成的循环，不能用 break 语句和 continue 语句进行控制。

6.8 break 语句和 continue 语句

6.8.1 break 语句

在 5.4 节中已经介绍过用 break 语句可以使流程跳出 switch 结构，继续执行 switch 语句下面的一个语句。实际上，break 语句还可以用来从循环体内跳出循环体，即提前结束循环，接着执行循环下面的语句。例如：

```
float pi=3.14159;
for(r=1;r<=10;r++)
  {
    area=pi*r*r;
    if(area>100) break;
    printf("r=%f,area=%f\n",r,area);
  }
```

程序的作用是计算 r=1 到 r=10 时的圆面积，直到面积 area 大于 100 为止。从上面的 for 循环可以看到：当 area>100 时，执行 break 语句，提前结束循环，即不再继续执行其余的几次循环。

break 语句的一般形式为

break;

break 语句不能用于循环语句和 switch 语句之外的任何其他语句中。

6.8.2 continue 语句

一般形式为

continue;

其作用为结束本次循环，即跳过循环体中下面尚未执行的语句，接着进行下一次是否执行循环的判定。

continue 语句和 break 语句的区别是：continue 语句只结束本次循环，而不是终止整个循环的执行。而 break 语句则是结束整个循环过程，不再判断执行循环的条件是否成立。如果有以下两个循环结构：

```
(1) while(表达式 1)
    {
        ⋮
      if(表达式 2) break;
        ⋮
    }

(2) while(表达式 1)
    {
        ⋮
      if(表达式 2) continue;
        ⋮
    }
```

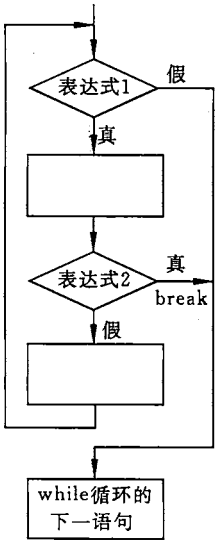


图 6-10

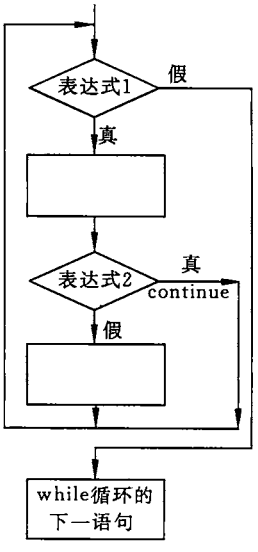


图 6-11

程序(1)的流程图如图 6-10 所示，而程序(2)的流程如图 6-11 所示。请注意图 6-10 和图 6-11 中当“表达式 2”为真时流程的转向。

例 6.5 把 100～200 之间的不能被 3 整除的数输出。

```
#include <stdio.h>
void main()
{
  int n;
  for (n=100;n<=200;n++)
    {if (n%3==0)
     continue;
     printf("%d  ",n);
    }
  printf("\n");
}
```

当 n 能被 3 整除时，执行 continue 语句，结束本次循环(即跳过 printf 函数语句)，只有 n 不能被 3 整除时才执行 printf 函数。

当然，例 6.5 中循环体也可以改用一个 if 语句处理：

```
if (n%3!=0) printf("%d  ",n);
```

在程序中用 continue 语句无非为了说明 continue 语句的作用。

6.9 程序举例

例 6.6 用 $\pi/4\approx1-1/3+1/5-1/7+\cdots$ 公式求 π 的近似值，直到某一项的绝对值小于 10^{-6} 为止。

用 N-S 结构化流程图表示算法(见图 6-12)。

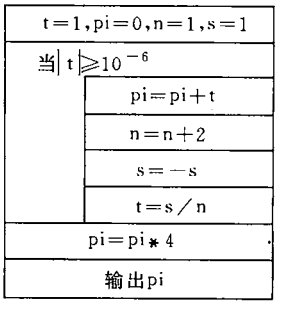


图 6-12

程序如下：

```
#include <stdio.h>
#include<math.h>
void main()
{
  int s;
  float n,t,pi;
  t=1;pi=0;n=1.0;s=1;
  while(fabs(t)>1e-6)          /*fabs是求绝对值函数,结果为浮点型*/
    {
      pi=pi+t;
      n=n+2;
      s=-s;
      t=s/n;
    }
 pi=pi*4;
 printf("pi=%10.6f\n",pi);
}
```

运行结果为：

pi=␣␣3.141593

例 6.7 求 Fibonacci 数列前 40 个数。这个数列有如下特点：第 1、2 两个数为 1、1。从第 3 个数开始，该数是其前面两个数之和。即：

$$\begin{cases} F_1=1 & (n=1) \\ F_2=1 & (n=2) \\ F_n=F_{n-1}+F_{n-2} & (n\geqslant 3) \end{cases}$$

这是一个有趣的古典数学问题：有一对兔子，从出生后第 3 个月起每个月都生一对兔子。小兔子长到第 3 个月后每个月又生一对兔子。假设所有兔子都不死，问每个月的兔子总数为多少？

可以从表 6-1 看出兔子繁殖的规律。

表 6-1 兔子繁殖的规律

第几个月	小兔子对数	中兔子对数	老兔子对数	兔子总数
1	1	0	0	1
2	0	1	0	1
3	1	0	1	2
4	1	1	1	3
5	2	1	2	5
6	3	2	3	8
7	5	3	5	13
⋮	⋮	⋮	⋮	⋮

注：不满 1 个月的为小兔子，满 1 个月不满 2 个月的为中兔子，满 3 个月以上的为老兔子。

可以看到每个月的兔子总数依次为 1,1,2,3,5,8,13…这就是 Fibonacci 数列。解此题的算法如图 6-13 所示。

程序如下：

```
#include <stdio.h>
void main()
{
  long int f1,f2;
  int i;
  f1=1;f2=1;
  for(i=1; i<=20; i++)
    {
      printf("%12ld␣%12ld ",f1,f2);
      if(i%2==0) printf("\n");
```

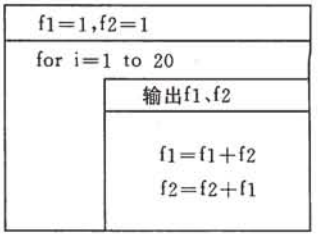


图 6-13

```
        f1=f1+f2;
        f2=f2+f1;
    }
}
```

运行结果为：

```
    1        1        2        3
    5        8       13       21
   34       55       89      144
  233      377      610      987
 1597     2584     4181     6765
10946    17711    28657    46368
75025   121393   196418   317811
514229  832040  1346269  2178309
3524578 57022887 9227465  14930352
24157817 39088169 63245986 102334155
```

程序中变量 f1 和 f2 用长整型，在 printf 函数中输出格式符用“%12ld”，而不是用“%12d”，这是由于在第 23 个数之后，整数值已超过整数最大值 32767，因此必须用长整型变量才能容纳，并用“%ld”格式输出。

if 语句的作用是使输出 4 个数后换行。i 是循环变量，当 i 为偶数时换行，而 i 每增值 1，就要计算和输出 2 个数(f1,f2)，因此 i 每隔 2 换一次行相当于每输出 4 个数后换行输出。

例 6.8 判断 m 是否素数。

算法如图 6-14 所示。

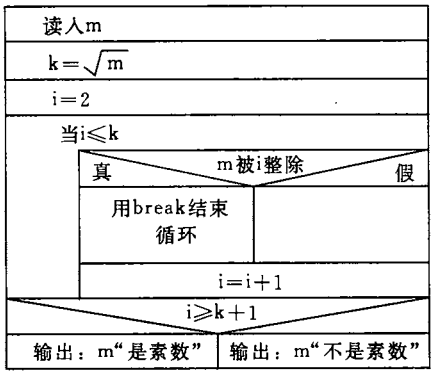


图 6-14

采用的算法如下：让 m 被 2 到$\sqrt{m}$除，如果 m 能被 2～$\sqrt{m}$之中任何一个整数整除，则提前结束循环，此时 i 必然小于或等于 k(即$\sqrt{m}$)；如果 m 不能被 2～k(即$\sqrt{m}$)之间的任一整数整除，则在完成最后一次循环后，i 还要加 1，因此 i=k+1，然后才终止循环。在循环之后判别 i 的值是否大于或等于 k+1，若是，则表明未曾被 2～k 之间任一整数整除过，

因此输出“是素数”。

程序如下：

```
#include <stdio.h>
#include <math.h>
void main()
{
  int m,i,k;
  scanf("%d",&m);
  k=sqrt(m);
  for (i=2;i<=k;i++)
    if(m%i==0)break;
  if(i>k)printf("%d is a prime number\n",m);
  else printf("%d is not a prime number\n",m);
}
```

运行情况如下：

```
17↙
17 is a prime number
```

例 6.9 求 100～200 间的全部素数。

在例 6.8 的基础上，对本题用一个嵌套的 for 循环即可处理。程序如下：

```
#include <stdio.h>
#include <math.h>
void main()
{
  int m,k,i,n=0;
  for(m=101;m<=200;m=m+2)
    {
      k=sqrt(m);
      for (i=2;i<=k;i++)
          if (m%i==0)
              break;
      if (i>=k+1)
          {
            printf("%d ",m);
            n=n+1;
          }
      if(n%10==0)
        printf("\n");
    }
  printf ("\n");
}
```

运行结果如下：

```
101 103 107 109 113 127 131 137 139 149
151 157 163 167 173 179 181 191 193 197
199
```

n 的作用是累计输出素数的个数，控制每行输出 10 个数据。

例 6.10 译密码。为使电文保密，往往按一定规律将其转换成密码，收报人再按约定的规律将其译回原文。例如，可以按以下规律将电文变成密码：

将字母 A 变成字母 E，a 变成 e，即变成其后的第 4 个字母，W 变成 A，X 变成 B，Y 变成 C，Z 变成 D，见图 6-15。

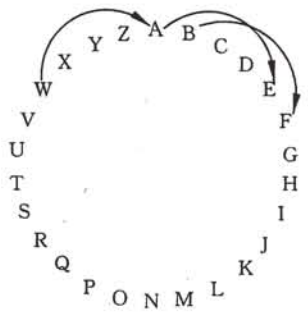


图 6-15

字母按上述规律转换，非字母字符不变。例如“China!”转换为“Glmre!”。

输入一行字符，要求输出其相应的密码。

程序如下：

```
#include <stdio.h>
void main()
{
  char c;
  while((c=getchar())!='\n')
    {
      if((c>='a' && c<='z') || (c>='A' && c<='Z'))
       {
        c=c+4;
        if(c>'Z' && c<='Z'+4 || c>'z')
          c=c-26;
       }
      printf("%c",c);
    }
  printf("\n");
}
```

运行结果如下：

```
China! ↙
Glmre!
```

程序中对输入的字符处理办法是：先判定它是否大写字母或小写字母，若是，则将其值加 4(变成其后的第 4 个字母)。如果加 4 以后字符值大于'Z'或'z'，则表示原来的字母在 V(或 v)之后，应按图 6-15 所示的规律将它转换为 A～D(或 a～d)之一。办法是使字符变量 c 的值减 26，如果读者对此还有疑问，请查 ASCII 码表即可清楚。

还有一点请读者注意：内嵌的 if 语句不能写成：

```
if(c>'Z'|| c>'z')
  c=c-26;
```

因为当字母为小写时都满足“c>'Z'”条件，从而也执行“c=c-26;”语句，这就会出错。因此必须限制其范围为“c>'Z' && c<='Z'+4”，即原字母为 W 到 Z，在此范围以外的不是大写字母 W～Z，不应按此规律转换。请考虑：为什么对小写字母不按此处理，即没有写成“c>'z' && c<='z'+4”，而只写成“c>'z'”。

习　题

6.1 输入两个正整数 m 和 n，求其最大公约数和最小公倍数。

6.2 输入一行字符，分别统计出其中英文字母、空格、数字和其他字符的个数。

6.3 求 $S_n=a+aa+aaa+\cdots+\overbrace{aa\cdots a}^{n个a}$ 之值，其中 a 是一个数字，n 表示 a 的位数，例如：

2+22+222+2222+22222（此时 $n=5$）

n 由键盘输入。

6.4 求 $\sum_{n=1}^{20} n!$（即求 1! +2! +3! +4! +…+20!）。

6.5 求 $\sum_{k=1}^{100} k+\sum_{k=1}^{50} k^2+\sum_{k=1}^{10} \frac{1}{k}$。

6.6 输出所有的“水仙花数”，所谓“水仙花数”是指一个 3 位数，其各位数字立方和等于该数本身。例如，153 是一水仙花数，因为 $153=1^3+5^3+3^3$。

6.7 一个数如果恰好等于它的因子之和，这个数就称为“完数”。例如，6 的因子为 1、2、3，而 6=1+2+3，因此 6 是“完数”。编程序找出 1000 之内的所有完数，并按下面格式输出其因子：

6 its factors are 1,2,3

6.8 有一分数序列

2/1,3/2,5/3,8/5,13/8,21/13,…

求出这个数列的前 20 项之和。

6.9 一个球从 100m 高度自由落下，每次落地后反跳回原高度的一半，再落下，再反弹。求它在第 10 次落地时，共经过多少米？第 10 次反弹多高？

6.10 猴子吃桃问题。猴子第一天摘下若干个桃子，当即吃了一半，还不过瘾，又多吃了一个。第二天早上又将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半零一个。到第 10 天早上想再吃时，就只剩一个桃子了。求第一天共摘多少个桃子。

6.11 用迭代法求 $x=\sqrt{a}$。求平方根的迭代公式为：

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

要求前后两次求出的 x 的差的绝对值小于 10^{-5}。

6.12 用牛顿迭代法求下面方程在 1.5 附近的根：

$$2x^3 - 4x^2 + 3x - 6 = 0$$

6.13 用二分法求下面方程在(−10,10)之间的根：

$$2x^3 - 4x^2 + 3x - 6 = 0$$

6.14 输出以下图案：

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

6.15 两个乒乓球队进行比赛，各出 3 人。甲队为 A、B、C 3 人，乙队为 X、Y、Z 3 人。已抽签决定比赛名单。有人向队员打听比赛的名单，A 说他不和 X 比，C 说他不和 X、Z 比，请编程序找出 3 对赛手的名单。

第7章 数 组

迄今为止，本书使用的都是属于基本类型（整型、字符型、实型）的数据，C 语言还提供了构造类型的数据，它们有数组类型、结构体类型和共用体类型。构造类型数据是由基本类型数据按一定规则组成的，因此它们又被称为“导出类型”。

本章只介绍数组。数组是有序数据的集合。数组中的每一个元素都属于同一个数据类型。用一个统一的数组名和下标来惟一地确定数组中的元素。例如，一个班有 30 个学生，可以用 s_1，s_2，s_3，…，s_{30} 代表 30 个学生的成绩。s 是数组名，下标代表学生的序号。s_{15} 代表第 15 个学生的成绩。在 C 语言中无法表示上下标，就用方括号表示下标，如用 s[15]表示 s_{15}，即第 15 个学生的成绩。将数组与循环结合起来，可以有效地处理大批量的数据，大大提高了工作效率，十分方便。

本章介绍在 C 语言中怎样定义和使用数组。

7.1 一维数组的定义和引用

7.1.1 一维数组的定义

一维数组的定义方式为

类型说明符 数组名[常量表达式];

例如：

```
int a[10];
```

它表示定义了一个整型数组，数组名为 a，此数组有 10 个元素。

说明：

（1）数组名的命名规则和变量名相同，遵循标识符命名规则。

（2）在定义数组时，需要指定数组中元素的个数，方括号中的常量表达式用来表示元素的个数，即数组长度。例如，指定 a[10]，表示 a 数组有 10 个元素。注意，下标是从 0 开始的，这 10 个元素是：a[0]，a[1]，a[2]，a[3]，a[4]，a[5]，a[6]，a[7]，a[8]，a[9]。请特别注意，按上面的定义，不存在数组元素 a[10]。

（3）常量表达式中可以包括常量和符号常量，不能包含变量。也就是说，C 语言不允许对数组的大小作动态定义，即数组的大小不依赖于程序运行过程中变量的值。例如，下面这样定义数组是不行的：

```
int n;
scanf("%d",&n);          /*在程序中临时输入数组的大小*/
int a[n];
```

7.1.2 一维数组元素的引用

数组必须先定义，然后使用。C 语言规定只能逐个引用数组元素而不能一次引用整个数组。

数组元素的表示形式为

数组名[下标]

下标可以是整型常量或整型表达式。例如：

```
a[0]=a[5]+a[7]-a[2*3]
```

注意：定义数组时用到的“数组名[常量表达式]”和引用数组元素时用到的“数组名[下标]”的区别，例如：

```
int a[10];          /* 定义数组长度为 10 */
t=a[6];             /* 引用 a 数组中序号为 6 的元素。此时 6 不代表数组长度 */
```

例 7.1 数组元素的引用。

```
#include <stdio.h>
void main()
{
  int i,a[10];
  for (i=0; i<=9;i++)
    a[i]=i;
  for(i=9;i>=0; i--)
    printf("%d ",a[i]);
  printf("\n");
}
```

运行结果如下：

```
9 8 7 6 5 4 3 2 1 0
```

程序使 a[0]～a[9]的值为 0～9，然后按逆序输出。

7.1.3 一维数组的初始化

对数组元素的初始化可以用以下方法实现。

(1) 在定义数组时对数组元素赋予初值。例如：

```
int a[10]={0,1,2,3,4,5,6,7,8,9};
```

将数组元素的初值依次放在一对花括号内。经过上面的定义和初始化之后，a[0]=0，a[1]=1，a[2]=2，a[3]=3，a[4]=4，a[5]=5，a[6]=6，a[7]=7，a[8]=8，a[9]=9。

(2) 可以只给一部分元素赋值。例如：

```
int a[10]={0,1,2,3,4};
```

定义 a 数组有 10 个元素，但花括号内只提供 5 个初值，这表示只给前面 5 个元素赋初值，后 5 个元素值为 0。

(3) 如果想使一个数组中全部元素值为 0，可以写成：

```
int a[10]={0,0,0,0,0,0,0,0,0,0};
```

或

```
int a[10]={0};
```

不能写成

```
int a[10]={0*10};
```

这是与 FORTRAN 语言不同的，不能给数组整体赋初值。

(4) 在对全部数组元素赋初值时，由于数据的个数已经确定，因此可以不指定数组长度。例如：

```
int a[5]={1,2,3,4,5};
```

可以写成

```
int a[]={1,2,3,4,5};
```

在第二种写法中，花括号中有 5 个数，系统就会据此自动定义 a 数组的长度为 5。但若数组长度与提供初值的个数不相同，则数组长度不能省略。例如，想定义数组长度为 10，就不能省略数组长度的定义，而必须写成：

```
int a[10]={1,2,3,4,5};
```

只初始化前 5 个元素，后 5 个元素为 0。

7.1.4 一维数组程序举例

例 7.2 用数组来处理求 Fibonacci 数列问题。

程序如下：

```
#include <stdio.h>
void main()
  {
    int i;
    int f[20]={1,1};
    for(i=2;i<20;i++)
      f[i]=f[i-2]+f[i-1];
    for(i=0;i<20;i++)
      {
        if(i%5==0)printf("\n");
        printf("%12d",f[i]);
      }
```

```
    printf("\n");
}
```

运行结果如下：

```
    1       1       2       3       5
    8      13      21      34      55
   89     144     233     377     610
  987    1597    2584    4181    6765
```

if 语句用来控制换行，每行输出 5 个数据。

例 7.3 用起泡法对 10 个数排序(由小到大)。

起泡法的思路是：将相邻两个数比较，将小的调到前头，见图 7-1。

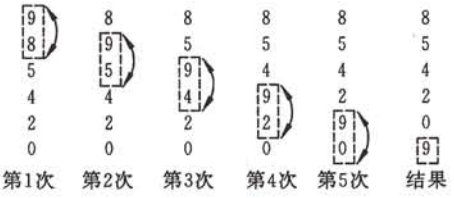


图 7-1

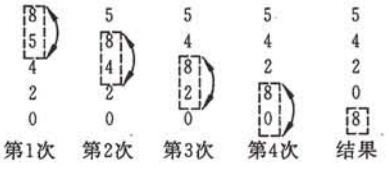


图 7-2

若有 6 个数。第一次将 8 和 9 对调，第二次将第 2 和第 3 个数(9 和 5)对调……如此共进行 5 次，得到 8—5—4—2—0—9 的顺序，可以看到：最大的数 9 已“沉底”，成为最下面一个数，而小的数“上升”。最小的数 0 已向上“浮起”一个位置。经第一趟(共 5 次比较与交换)后，已得到最大的数。然后进行第二趟比较，对余下的前面 5 个数按上法进行比较，见图 7-2。经过 4 次比较与交换，得到次大的数 8。如此进行下去，可以推知，对 6 个数要比较 5 趟，才能使 6 个数按大小顺序排列。在第一趟中要进行两个数之间的比较，共 5 次，在第二趟中比较 4 次……第 5 趟比较 1 次。如果有 n 个数，则要进行 n—1 趟比较。在第 1 趟比较中要进行 n—1 次两两比较，在第 j 趟比较中要进行 n—j 次两两比较。据此画出流程图(见图 7-3，设 n=10)。

根据流程图写出程序(今设 n=10)。

```
#include <stdio.h>
void main()
{
  int a[10];
  int i,j,t;
```

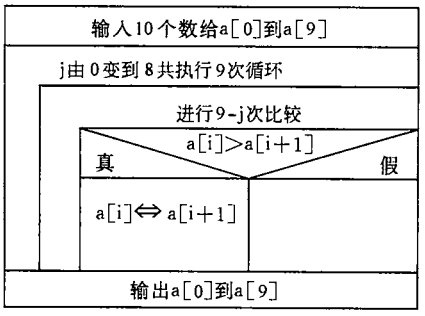


图 7-3

```
    printf("input 10 numbers :\n");
    for (i=0;i<10;i++)
      scanf("%d",&a[i]);
    printf("\n");
    for(j=0;j<9;j++)                    /*进行9次循环,实现9趟比较*/
      for(i=0;i<9-j;i++)                /*在每一趟中进行9-j次比较*/
        if (a[i]>a[i+1])                /*相邻两个数比较*/
          {
            t=a[i];
            a[i]=a[i+1];
            a[i+1]=t;
          }
        printf("the sorted numbers :\n");
        for(i=0;i<10;i++)
          printf("%d ",a[i]);
    printf("\n");
}
```

运行情况如下：

```
input 10 numbers:
1 0 4 8 12 65 -76 100 -45 123↙

the sorted numbers:
-76 -45 0 1 4 8 12 65 100 123
```

7.2 二维数组的定义和引用

7.2.1 二维数组的定义

二维数组定义的一般形式为

类型说明符 数组名[常量表达式][常量表达式];

例如：

```
float a[3][4],b[5][10];
```

定义 a 为 3×4(3 行 4 列)的数组，b 为 5×10(5 行 10 列)的数组。注意，不能写成：

```
float a[3,4],b[5,10];
```

C 语言对二维数组采用这样的定义方式，使得二维数组可被看作是一种特殊的一维数组：它的元素又是一个一维数组。例如，可以把 a 看作是一个一维数组，它有 3 个元素：a[0]、a[1]、a[2]，每个元素又是一个包含 4 个元素的一维数组，见图 7-4。

可以把 a[0]、a[1]、a[2]看作是 3 个一维数组的名字。上面定义的二维数组可以理解为定义了 3 个一维数组，即相当于：

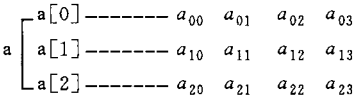


图 7-4

```
float a[0][4],a[1][4],a[2][4];
```

此处把 a[0]、a[1]、a[2]看作一维数组名。C 语言的这种处理方法在数组初始化和用指针表示时显得很方便，这在以后会体会到。

C 语言中，二维数组中元素排列的顺序是按行存放的，即在内存中先顺序存放第一行的元素，再存放第二行的元素。图 7-5 表示对 a[3][4]数组存放的顺序。

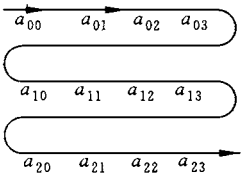


图 7-5

C 语言允许使用多维数组。有了二维数组的基础，再掌握多维数组是不困难的。例如，定义三维数组的方法如下：

```
float a[2][3][4];
```

多维数组元素在内存中的排列顺序：第一维的下标变化最慢，最右边的下标变化最快。例如，上述三维数组的元素排列顺序为：

a[0][0][0]→a[0][0][1]→a[0][0][2]→a[0][0][3]→a[0][1][0]→a[0][1][1]→a[0][1][2]→a[0][1][3]→a[0][2][0]→a[0][2][1]→a[0][2][2]→a[0][2][3]→a[1][0][0]→a[1][0][1]→a[1][0][2]→a[1][0][3]→a[1][1][0]→a[1][1][1]→a[1][1][2]→a[1][1][3]→a[1][2][0]→a[1][2][1]→a[1][2][2]→a[1][2][3]

7.2.2 二维数组的引用

二维数组元素的表示形式为

数组名[下标][下标]

例如 a[2][3]。下标可以是整型表达式，如 a[2－1][2＊2－1]。不要写成 a[2,3]、a[2－1,2＊2－1]形式。

数组元素可以出现在表达式中，也可以被赋值，例如：

```
b[1][2]=a[2][3]/2
```

在使用数组元素时应该注意，下标值应在已定义的数组大小的范围内。常出现的错误，例如：

```
int a[3][4];          /* 定义 a 为 3×4 的数组 */
⋮
a[3][4]=3;
```

按以上的定义，数组 a 可用的行下标的范围为 0～2，列下标的范围为 0～3。用 a[3][4]超过了数组的范围。

请读者严格区分在定义数组时用的 a[3][4]和引用元素时的 a[3][4]的区别。前者用 a[3][4]来定义数组的维数和各维的大小，后者 a[3][4]中的 3 和 4 是数组元素的下标值，a[3][4]代表行序号为 3、列序号为 4 的元素(行序号和列序号均从 0 起算)。

7.2.3 二维数组的初始化

可以用下面的方法对二维数组初始化。

(1) 分行给二维数组赋初值。例如：

```
int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

这种赋初值方法比较直观，把第 1 个花括号内的数据给第 1 行的元素，第 2 个花括号内的数据赋给第 2 行的元素……即按行赋初值。

(2) 可以将所有数据写在一个花括号内，按数组排列的顺序对各元素赋初值。例如：

```
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

效果与前相同。但以第 1 种方法为好，一行对一行，界限清楚。用第(2)种方法如果数据多，写成一大片，容易遗漏，也不易检查。

(3) 可以对部分元素赋初值。例如：

```
int a[3][4]={{1},{5},{9}};
```

它的作用是只对各行第 1 列(即序号为 0 的列)的元素赋初值，其余元素值自动为 0。赋初值后数组各元素为

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 \end{bmatrix}$$

也可以对各行中的某一元素赋初值，例如：

```
int a[3][4]={{1},{0,6},{0,0,11}};
```

初始化后的数组元素如下：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 11 & 0 \end{bmatrix}$$

这种方法对非 0 元素少时比较方便，不必将所有的 0 都写出来，只需输入少量数据。也可以只对某几行元素赋初值：

```
int a[3][4]={{1},{5,6}};
```

数组元素为

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 5 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

第 3 行不赋初值。也可以对第 2 行不赋初值，例如：

```
int a[3][4]={{1},{},{9}};
```

(4) 如果对全部元素都赋初值(即提供全部初始数据)，则定义数组时对第一维的长度可以不指定，但第二维的长度不能省。例如：

```
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

与下面的定义等价：

```
int a[][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

系统会根据数据总个数和第二维的长度算出第一维的长度。数组一共有 12 个元素，每行 4 列，显然可以确定行数为 3。

在定义时也可以只对部分元素赋初值而省略第一维的长度，但应分行赋初值。例如：

```
int a[][4]={{0,0,3},{},{0,10}};
```

这样的写法，能通知编译系统；数组共有 3 行。数组各元素为

$$\begin{bmatrix} 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \end{bmatrix}$$

从本节的介绍中可以看到：C 语言在定义数组和表示数组元素时采用 a[][]这种两个方括号的方式，对数组初始化时十分有用，它使概念清楚，使用方便，不易出错。

7.2.4 二维数组程序举例

例 7.4 将一个二维数组行和列的元素互换，存到另一个二维数组中。例如：

$$a=\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad b=\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

程序如下：

```
#include <stdio.h>
void main()
{
    int a[2][3]={{1,2,3},{4,5,6}};
    int b[3][2],i,j;
    printf("array a:\n");
    for (i=0;i<=1;i++)
    {
```

```
        for (j=0;j<=2;j++)
          {
            printf("%5d",a[i][j]);
            b[j][i]=a[i][j];
          }
        printf("\n");
      }
      printf("array b:\n");
      for (i=0;i<=2;i++)
        {
          for(j=0;j<=1;j++)
            printf("%5d",b[i][j]);
          printf("\n");
        }
}
```

运行结果如下：

```
array a:
    1   2   3
    4   5   6
array b:
    1   4
    2   5
    3   6
```

例 7.5 有一个 3×4 的矩阵，要求编程序求出其中值最大的那个元素的值，以及其所在的行号和列号。

先用 N-S 流程图表示算法，见图 7-6。

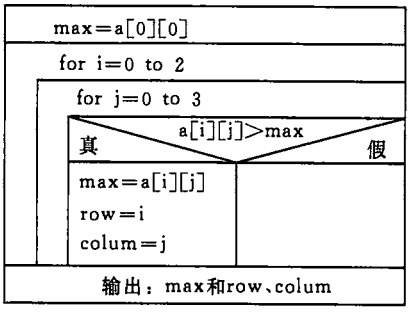


图 7-6

据此写出以下程序：

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

```
    int i,j,row=0,colum=0,max;
    int a[3][4]={{1,2,3,4},{9,8,7,6},{-10,10,-5,2}};
    max=a[0][0];
    for (i=0;i<=2;i++)
      for (j=0;j<=3;j++)
          if (a[i][j]>max)
            {
              max=a[i][j];
              row=i;
              colum=j;
            }
    printf("max=%d,row=%d,colum=%d\n",max,row,colum);
}
```

输出结果为：

```
max=10,row=2,colum=1
```

7.3 字符数组

用来存放字符数据的数组是字符数组。字符数组中的一个元素存放一个字符。

7.3.1 字符数组的定义

字符数组的定义方法与前面介绍的类似。例如：

```
char c[10];
c[0]='I';c[1]=' ';c[2]='a';c[3]='m';c[4]=' ';
c[5]='h';c[6]='a';c[7]='p';c[8]='p';c[9]='y';
```

以上定义了 c 为字符数组，包含 10 个元素。赋值以后数组的状态如图 7-7 所示。

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	c[8]	c[9]
I	␣	a	m	␣	h	a	p	p	y

图 7-7

由于字符型与整型是互相通用的，因此也可以定义一个整型数组，用它存放字符数据，例如：

```
int c[10];
c[0]='a';     /*合法，但浪费存储空间*/
```

7.3.2 字符数组的初始化

对字符数组初始化，最容易理解的方式是逐个字符赋给数组中各元素。例如：

```
char c[10]={'I',' ','a','m',' ','h','a','p','p','y'};
```

把 10 个字符分别赋给 c[0]～c[9]这 10 个元素。

如果在定义字符数组时不进行初始化，则数组中各元素的值是不可预料的。如果花括号中提供的初值个数(即字符个数)大于数组长度，则按语法错误处理。如果初值个数小于数组长度，则只将这些字符赋给数组中前面那些元素，其余的元素自动定为空字符(即'\0')。例如：

```
char c[10]={'c',' ','p','r','o','g','r','a','m'};
```

数组状态如图 7-8 所示。

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	c[8]	c[9]
c	␣	p	r	o	g	r	a	m	\0

图 7-8

如果提供的初值个数与预定的数组长度相同，在定义时可以省略数组长度，系统会自动根据初值个数确定数组长度。例如：

```
char c[]={'I',' ','a','m',' ','h','a','p','p','y'};
```

数组 c 的长度自动定为 10。用这种方式可以不必人工去数字符的个数，尤其在赋初值的字符个数较多时，比较方便。

也可以定义和初始化一个二维字符数组，例如：

```
char diamond[5][5]={{' ',' ','*'},{' ','*',' ','*'},{'*',' ',' ',' ','*'},{' ','*',' ',
'*'},{' ',' ','*'}};
```

用它代表一个菱形的平面图形，见图 7-9。完整的程序见例 7.7。

7.3.3 字符数组的引用

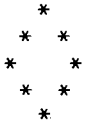


图 7-9

可以引用字符数组中的一个元素，得到一个字符。

例 7.6 输出一个字符串。

```
#include <stdio.h>
void main()
{
  char c[10]={'I',' ','a','m',' ','a',' ','b','o','y'};
  int i;
  for(i=0;i<10;i++)
      printf("%c",c[i]);
  printf("\n");
}
```

运行结果：

```
I am a boy
```

例 7.7 输出一个菱形图。

```
#include <stdio.h>
void main()
{
  char diamond[][5]={{' ',' ','*'},{' ','*',' ','*'},{'*',' ',' ',' ','*'},{' ','*',' ',
                    '*'},{' ',' ','*'}};
  int i,j;
  for (i=0;i<5;i++)
    {
      for (j=0;j<5;j++)
          printf("%c",diamond[i][j]);
      printf("\n");
    }
}
```

运行结果为：

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

7.3.4 字符串和字符串结束标志

在 C 语言中，是将字符串作为字符数组来处理的。例 7.6 就是用一个一维的字符数组来存放字符串"I am a boy"的，字符串中的字符是逐个存放到数组元素中的。这个字符串的实际长度与数组长度相等。在实际工作中，人们关心的往往是字符串的有效长度而不是字符数组的长度。例如，定义一个字符数组长度为 100，而实际有效字符只有 40 个。为了测定字符串的实际长度，C 语言规定了一个"字符串结束标志"，以字符'\0'作为标志。如果有一个字符串，前面 9 个字符都不是空字符(即'\0')，而第 10 个字符是'\0'，则此字符串的有效字符为 9 个。也就是说，在遇到字符'\0'时，表示字符串结束，由它前面的字符组成字符串。

系统对字符串常量也自动加一个'\0'作为结束符。例如"C␣Program"共有 9 个字符，但在内存中占 10 个字节，最后一个字节'\0'是由系统自动加上的。字符串作为一维数组存放在内存中。

有了结束标志'\0'后，字符数组的长度就显得不那么重要了。在程序中往往依靠检测'\0'的位置来判定字符串是否结束，而不是根据数组的长度来决定字符串长度。当然，在定义字符数组时应估计实际字符串长度，保证数组长度始终大于字符串实际长度。如果在一个字符数组中先后存放多个不同长度的字符串，则应使数组长度大于最长的字符串的长度。

说明：'\0'代表 ASCII 码为 0 的字符，从 ASCII 码表中可以查到，ASCII 码为 0 的字符不是一个可以显示的字符，而是一个"空操作符"，即它什么也不做。用它来作为字符串结束标志不会产生附加的操作或增加有效字符，只起一个供辨别的标志。

前面曾用过以下语句输出一个字符串。

```
printf("How do you do? \n");
```

在执行此语句时系统怎么知道应该输出到哪里为止呢？实际上，在内存中存放时，系统自动在最后一个字符'\n'的后面加了一个'\0'作为字符串结束标志，在执行 printf 函数时，每输出一个字符检查一次，看下一个字符是否'\0'，遇'\0'就停止输出。

对 C 语言处理字符串的方法有以上的了解后，再对字符数组初始化的方法补充一种方法，即用字符串常量来使字符数组初始化。例如：

```
char c[]={"I am happy"};
```

也可以省略花括号，直接写成：

```
char c[]="I am happy";
```

不像例 7.6 那样用单个字符作为字符数组的初值，而是用一个字符串（注意字符串的两端是用双撇号而不是单撇号括起来的）作为初值。显然，这种方法直观、方便、符合人们的习惯。数组 c 的长度不是 10，而是 11，这点务请注意。因为字符串常量的最后由系统加上一个'\0'。因此，上面的初始化与下面的初始化等价。

```
char c[] ={'I',' ','a','m',' ','h','a','p','p','y','\0'};
```

而不与下面的等价：

```
char c[]={'I',' ','a','m',' ','h','a','p','p','y'};
```

前者的长度为 11，后者的长度为 10。如果有：

```
char c[10]={"China"};
```

数组 c 的前 5 个元素为

'C'、'h'、'i'、'n'、'a'，第 6 个元素为'\0'，后 4 个元素也设定为空字符，见图 7-10。

C	h	i	n	a	\0	\0	\0	\0	\0

图 7-10

需要说明的是：字符数组并不要求它的最后一个字符为'\0'，甚至可以不包含'\0'。像以下这样写完全是合法的：

```
char c[5]={'C','h','i','n','a'};
```

是否需要加'\0'，完全根据需要决定。但是由于系统对字符串常量自动加一个'\0'，因此，为了使处理方法一致，便于测定字符串的实际长度，以及在程序中作相应的处理，在字符数组也常常人为地加上一个'\0'，例如：

```
char c[6]={'C','h','i','n','a','\0'};
```

这样做，便于引用字符数组中的字符串。如定义了以下的字符数组：

```
char c[]={"Pascal program"};
```

若想用一个新的字符串代替原有的字符串"Pascal program",从键盘向字符数组输入:

```
Hello
```

如果不加'\0'的话,字符数组中的字符如下:

```
Hellol program
```

新字符串和老字符串连成一片,无法区分开。如果想输出字符数组中的字符串,则会连续输出 Hellol program。现在,系统在"Hello"后面加了一个'\0',它取代了第 6 个字符"l",它是字符串结束标志,在输出字符数组中的字符串时,遇'\0'就停止输出,因此只输出了字符串"Hello"。从这里可以看到在字符串末尾加'\0'的作用。

7.3.5 字符数组的输入输出

字符数组的输入输出可以有两种方法。

(1) 逐个字符输入输出。用格式符"%c"输入或输出一个字符,如例 7.6。

(2) 将整个字符串一次输入或输出。用"%s"格式符,意思是对字符串(string)的输入输出。例如:

```
char c[]={"China"};
printf("%s",c);
```

在内存中数组 c 的状态如图 7-11 所示。输出时,遇结束符'\0'就停止输出。输出结果为

```
China
```

C	h	i	n	a	\0

图 7-11

注意:

(1) 输出字符不包括结束符'\0'。

(2) 用"%s"格式符输出字符串时,printf 函数中的输出项是字符数组名,而不是数组元素名。写成下面这样是不对的:

```
printf("%s",c[0]);
```

(3) 如果数组长度大于字符串的实际长度,也只输出到遇'\0'结束。例如:

```
char c[10]={"China"};      /* 字符串长度为 5,连'\0'共占 6 个字节 */
printf("%s",c);
```

也只输出字符串的有效字符"China",而不是输出 10 个字符。这就是用字符串结束标志的好处。

(4) 如果一个字符数组中包含一个以上'\0',则遇第一个'\0'时输出就结束。

(5) 可以用 scanf 函数输入一个字符串。例如:

```
scanf("%s",c);
```

scanf 函数中的输入项 c 是已定义的字符数组名,输入的字符串应短于已定义的字符数组的长度。例如,已定义:

```
char c[6];
```

从键盘输入:

China↙

系统自动在 China 后面加一个'\0'结束符。如果利用一个 scanf 函数输入多个字符串,则在输入时以空格分隔。例如:

```
char str1[5],str2[5],str3[5];
scanf("%s%s%s",str1,str2,str3);
```

输入数据:

How are you?↙

输入后 str1、str2、str3 数组状态见图 7-12。数组中未被赋值的元素的值自动置'\0'。若改为:

```
char str[13];
scanf("%s",str);
```

H	o	w	\0	\0
a	r	e	\0	\0
y	o	u	?	\0

图 7-12

如果输入以下 12 个字符:

How␣are␣you?↙

由于系统把空格字符作为输入的字符串之间的分隔符,因此只将空格前的字符"How"送到 str 中。由于把"How"作为一个字符串处理,故在其后加'\0'。str 数组状态见图 7-13。

H	o	w	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0

图 7-13

注意:scanf 函数中的输入项如果是字符数组名,不要再加地址符 &,因为在 C 语言中数组名代表该数组的起始地址。下面写法不正确:

```
scanf("%s",&str);
```

分析图 7-14 所示的字符数组,数组名为 c,占 6 个字节。数组名 c 代表地址 2000。可以用下面的输出语句得到数组的起始地址。

```
printf("%o",c);          /* 用八进制形式输出数组 c 的起始地址 */
```

输出数组 c 的起始地址 2000。可知数组名 c 代表数组起始地址。见图 7-14。

(6) 前面介绍的输出字符串的方法：

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

实际上是这样执行的：按字符数组名 c 找到其数组起始地址，然后逐个输出其中的字符，直到遇'\0'为止。

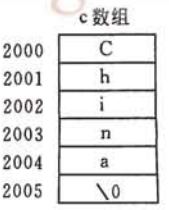


图 7-14

7.3.6 字符串处理函数

在 C 函数库中提供了一些用来处理字符串的函数，使用方便。几乎所有版本的 C 语言编译系统都提供这些函数。下面介绍几种常用的函数。

1. puts 函数

其一般形式为

puts（字符数组）

其作用是将一个字符串（以'\0'结束的字符序列）输出到终端。假如已定义 str 是一个字符数组名，且该数组已被初始化为"China"。则执行：

```
puts(str);
```

其结果是在终端上输出"China"。由于可以用 printf 函数输出字符串，因此 puts 函数用的不多。

用 puts 函数输出的字符串中可以包含转义字符。例如：

```
char str[]={"China\n Bei jing"};
puts(str);
```

输出：

```
China
Bei jing
```

在输出时将字符串结束标志'\0'转换成'\n'，即输出完字符串后换行。

2. gets 函数

其一般形式为

gets(字符数组)

其作用是从终端输入一个字符串到字符数组，并且得到一个函数值。该函数值是字符数组的起始地址。如执行下面的函数：

```
gets(str)
```

从键盘输入：

```
Computer↙
```

将输入的字符串"Computer"送给字符数组 str(请注意,送给数组的共有 9 个字符,而不是 8 个字符),函数值为字符数组 str 的起始地址。一般利用 gets 函数的目的是向字符数组输入一个字符串,而不大关心其函数值。

注意:用 puts 和 gets 函数只能输出或输入一个字符串,不能写成

```
puts(str1,str2)
```

或

```
gets(str1,str2)
```

3. strcat 函数

其一般形式为

strcat(字符数组 1,字符数组 2)

strcat 是 STRing CATenate(字符串连接)的缩写。其作用是连接两个字符数组中的字符串,把字符串 2 接到字符串 1 的后面,结果放在字符数组 1 中,函数调用后得到一个函数值——字符数组 1 的地址。例如:

```
char str1[30]={"People's Republic of "};
char str2[]={"China"};
printf("%s",strcat(str1,str2));
```

输出:

```
People's Republic of China
```

连接前后的状况见图 7-15 所示。

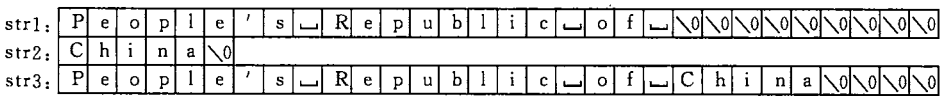


图 7-15

说明:

(1) 字符数组 1 必须足够大,以便容纳连接后的新字符串。本例中定义 str1 的长度为 30,是足够大的,如果在定义时改用 str1[]={"People's Republic of"};就会出问题,因长度不够。

(2) 连接前两个字符串的后面都有'\0',连接时将字符串 1 后面的'\0'取消,只在新串最后保留'\0'。

4. strcpy 和 strncpy 函数

其一般形式为

strcpy(字符数组 1,字符串 2)

strcpy 是 STRingCoPY(字符串复制)的简写。它是"字符串复制函数",作用是将字

符串 2 复制到字符数组 1 中去。例如：

```
char str1[10]='',str2[]={"China"};
strcpy(str1,str2);
```

执行后，str1 的状态如图 7-16 所示。

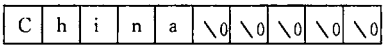


图 7-16

说明：

(1) 字符数组 1 必须定义得足够大，以便容纳被复制的字符串。字符数组 1 的长度不应小于字符串 2 的长度。

(2) “字符数组 1”必须写成数组名形式(如 str1)，“字符串 2”可以是字符数组名，也可以是一个字符串常量。例如：

```
strcpy(str1,"China");
```

作用与前面相同。

(3) 如果在复制前未对 str1 数组赋值，则 str1 各字节中的内容是无法预知的，复制时将 str2 中的字符串和其后的'\0'一起复制到字符数组 1 中，取代字符数组 1 中的前面 6 个字符，最后 4 个字符并不一定是斜杠零，而是 str1 中原有的最后 4 个字节的内容。

(4) 不能用赋值语句将一个字符串常量或字符数组直接给一个字符数组。如下面两行都是不合法的：

```
str1="China";
str1=str2;
```

而只能用 strcpy 函数将一个字符串复制到另一个字符数组中去。用赋值语句只能将一个字符赋给一个字符型变量或字符数组元素。如下面的语句是合法的：

```
char a[5],c1,c2;
c1='A';c2='B';
a[0]='C';a[1]='h';a[2]='i';a[3]='n';a[4]='a';
```

(5) 可以用 strncpy 函数将字符串 2 中前面 n 个字符复制到字符数组 1 中去。例如：

```
strncpy(str1,str2,2);
```

作用是将 str2 中最前面 2 个字符复制到 str1 中，取代 str1 中原有的最前面 2 个字符。但复制的字符个数 n 不应多于 str1 中原有的字符(不包括'\0')。

5. strcmp 函数

其一般形式为

strcmp(字符串 1,字符串 2)

strcmp 是 STRing CoMPare(字符串比较)的缩写。它的作用是比较字符串 1 和字符串 2。例如：

```
strcmp(str1,str2);
strcmp("China","Korea");
strcmp(str1,"Bei jing");
```

字符串比较的规则与其他语言中的规则相同，即对两个字符串自左至右逐个字符相比(按 ASCII 码值大小比较)，直到出现不同的字符或遇到'\0'为止。如全部字符相同，则认为相等；若出现不相同的字符，则以第一个不相同的字符的比较结果为准。例如：

"A"<"B", "a">"A","computer">"compare","these">"that","36+54">"! $ & #","CHINA">"CANADA","DOG"<"cat"。

如果参加比较的两个字符串都由英文字母组成，则有一个简单的规律：在英文字典中位置在后面的为“大”。例如 computer 在字典中的位置在 compare 之后，所以"computer">"compare"。但应注意小写字母比大写字母“大”，所以“DOG”<“cat”。

比较的结果由函数值带回。

(1) 如果字符串 1=字符串 2，则函数值为 0。

(2) 如果字符串 1>字符串 2，则函数值为一个正整数。

(3) 如果字符串 1<字符串 2，则函数值为一个负整数。

注意：对两个字符串比较，不能用以下形式：

```
if(str1>str2)
  printf("yes");
```

而只能用

```
if(strcmp(str1,str2)>0)
  printf("yes");
```

6. strlen 函数

其一般形式为

strlen (字符数组)

strlen 是 STRing LENgth(字符串长度)的缩写。它是测试字符串长度的函数。函数的值为字符串中的实际长度(不包括'\0'在内)。例如：

```
char str[10]={"China"};
printf("%d",strlen(str));
```

输出结果不是 10，也不是 6，而是 5。也可以直接测试字符串常量的长度，例如：

```
strlen("China");
```

7. strlwr 函数

其一般形式为

strlwr (字符串)

strlwr 是 STRing LoWeRcase（字符串小写）的缩写。函数的作用是将字符串中大写字母换成小写字母。

8. strupr 函数

其一般形式为

strupr（字符串）

strupr 是 STRing UPpeRcase（字符串大写）的缩写。函数的作用是将字符串中小写字母换成大写字母。

以上介绍了常用的 8 种字符串处理函数，应当再次强调：库函数并非 C 语言本身的组成部分，而是 C 语言编译系统为方便用户使用而提供的公共函数。不同的编译系统提供的函数数量和函数名、函数功能都不尽相同，使用时要小心，必要时查一下库函数手册。当然，有一些基本的函数（包括函数名和函数功能），不同的系统所提供的是相同的，这就为程序的通用性提供了基础。

7.3.7 字符数组应用举例

例 7.8 输入一行字符，统计其中有多少个单词，单词之间用空格分隔开。

程序如下：

```
#include <stdio.h>
void main()
{
  char string[81];
  int i,num=0,word=0;
  char c;
  gets(string);
  for (i=0;(c=string[i])!='\0';i++)
    if(c==' ') word=0;
    else if(word==0)
      {
        word=1;
        num++;
      }
  printf("There are %d words in the line.\n",num);
}
```

运行情况如下：

```
I am a boy.↙
There are 4 words in the line.
```

程序中变量 i 作为循环变量，num 用来统计单词个数，word 作为判别是否单词的标志，若 word=0 表示未出现单词，如出现单词 word 就置成 1。算法见图 7-17 所示。

解题思路：单词的数目可以由空格出现的次数决定（连续的若干个空格作为出现一次

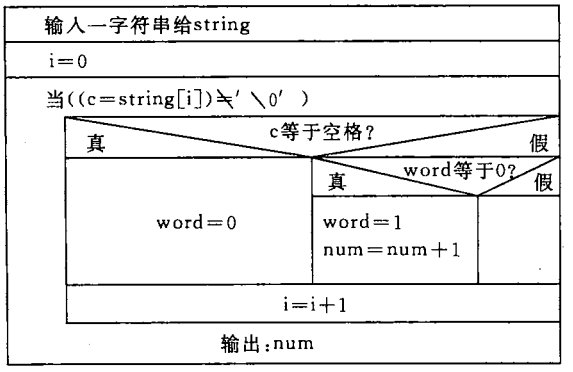


图 7-17

空格;一行开头的空格不统计在内)。如果测出某一个字符为非空格,而它的前面的字符是空格,则表示"新的单词开始了",此时使 num(单词数)累加 1。如果当前字符为非空格而其前面的字符也是非空格,则意味着仍然是原来那个单词的继续,num 不应再累加 1。前面一个字符是否空格可以从 word 的值看出来,若 word 等于 0,则表示前一个字符是空格;如果 word 等于 1,意味着前一个字符为非空格,可以用图 7-18 表示。

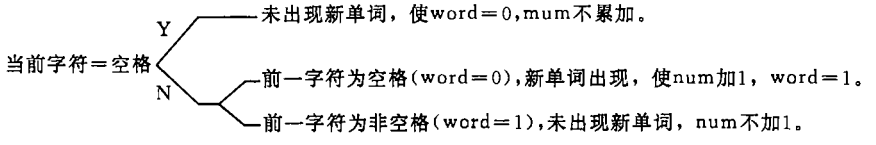


图 7-18

如果输入为"I am a boy."对每个字符的有关参数的状态见表 7-1 所示。

表 7-1 输入"I am a boy."后有关参数的状态

当前字符	␣	I	␣	a	m	␣	a	␣	b	o	y	.
是否空格	是	否	是	否	否	是	否	是	否	否	否	否
word 原值	0	0	1	0	1	1	0	1	0	1	1	1
新单词开始否	未	是	未	是	未	未	是	未	是	未	未	未
word 新值	0	1	0	1	1	0	1	0	1	1	1	1
num 值	0	1	1	2	2	2	3	3	4	4	4	4

程序中 for 语句中的"循环条件"为

```
(c=string[i])!='\0'
```

它的作用是先将字符数组的某个元素(一个字符)赋给字符变量 c。此时赋值表达式的值就是该字符,然后再判定它是否结束符。这个"循环条件"包含了一个赋值操作和一个关系运算。

例 7.9 有 3 个字符串,要求找出其中最大者。

今设一个二维的字符数组 str，大小为 3×20，即有 3 行 20 列，每一行可以容纳 20 个字符。图 7-19 表示此二维数组的情况。

str[0]：	C	h	i	n	a	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
str[1]：	J	a	p	a	n	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
str[2]：	I	n	d	i	a	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0

图 7-19

如前所述，可以把 str[0]、str[1]、str[2]看作 3 个一维字符数组，它们各有 20 个元素。可以把它们如同一维数组那样进行处理。今用 gets 函数分别读入 3 个字符串。经过两次比较，就可得到值最大者，把它放在一维字符数组 string 中。

程序如下：

```
#include<stdio.h>
#include<string.h>
void main ( )
{
  char string[20];
  char str[3][20];
  int i;
  for (i=0;i<3;i++)
       gets (str[i]);
  if (strcmp(str[0],str[1])>0)
      strcpy(string,str[0]);
  else
      strcpy(string,str[1]);
  if (strcmp(str[2],string)>0)
      strcpy(string,str[2]);
  printf("\nthe largest string is: \n%s\n",string);
}
```

运行结果如下：

```
CHINA↙
HOLLAND↙
AMERICA↙

the largest string is:
    HOLLAND
```

当然，这个题目也可以不采用二维数组，而设 3 个一维字符数组来处理。读者可自己完成。

习　题

7.1　用筛选法求 100 之内的素数。

7.2　用选择法对 10 个整数排序。

7.3 求一个 3×3 的整型矩阵对角线元素之和。

7.4 已有一个已排好序的数组，要求输入一个数后，按原来排序的规律将它插入数组中。

7.5 将一个数组中的值按逆序重新存放。例如，原来顺序为 8,6,5,4,1。要求改为 1,4,5,6,8。

7.6 输出以下的杨辉三角形(要求输出 10 行)。

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5  10  10  5  1
⋮  ⋮  ⋮  ⋮  ⋮  ⋮
```

7.7 输出“魔方阵”。所谓魔方阵是指这样的方阵，它的每一行、每一列和对角线之和均相等。例如，三阶魔方阵为：

```
8  1  6
3  5  7
4  9  2
```

要求输出 $1\sim n^2$ 的自然数构成的魔方阵。

7.8 找出一个二维数组中的鞍点，即该位置上的元素在该行上最大、在该列上最小。也可能没有鞍点。

7.9 有 15 个数按由大到小顺序存放在一个数组中，输入一个数，要求用折半查找法找出该数是数组中第几个元素的值。如果该数不在数组中，则输出“无此数”。

7.10 有一篇文章，共有 3 行文字，每行有 80 个字符。要求分别统计出其中英文大写字母、小写字母、数字、空格以及其他字符的个数。

7.11 输出以下图案：

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

7.12 有一行电文，已按下面规律译成密码：

```
A→Z  a→z
B→Y  b→y
C→X  c→x
⋮    ⋮
```

即第 1 个字母变成第 26 个字母，第 i 个字母变成第(26－i＋1)个字母。非字母字符不变。要求编程序将密码译回原文，并输出密码和原文。

7.13 编一程序，将两个字符串连接起来，不要用 strcat 函数。

7.14 编一个程序，将两个字符串 s1 和 s2 比较，若 s1＞s2，输出一个正数；若 s1＝s2，输出 0；若 s1＜s2，输出一个负数。不要用 strcpy 函数。两个字符串用 gets 函数读入。输出的正数或负数的绝对值应是相比较的两个字符串相应字符的 ASCII 码的差值。例如，"A"与"C"相比，由于"A"＜"C"，应输出负数，同时由于'A'与'C'的 ASCII 码差值为 2，因此应输出"－2"。同理："And"和"Aid"比较，根据第 2 个字符比较结果，"n"比"i"大 5，因此应输出"5"。

7.15 编写一个程序，将字符数组 s2 中的全部字符复制到字符数组 s1 中。不用 strcpy 函数。复制时，'\0'也要复制过去。'\0'后面的字符不复制。

第8章 函 数

8.1 概 述

一个较大的程序一般应分为若干个程序模块，每一个模块用来实现一个特定的功能。所有的高级语言中都有子程序这个概念，用子程序实现模块的功能。在C语言中，子程序的作用是由函数来完成的。一个C程序可由一个主函数和若干个其他函数构成。由主函数调用其他函数，其他函数也可以互相调用。同一个函数可以被一个或多个函数调用任意多次。图8-1是一个程序中函数调用的示意图。

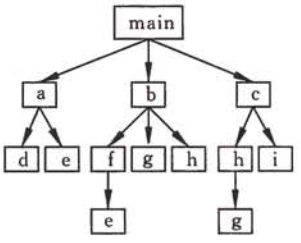


图 8-1

在程序开发中，常将一些常用的功能模块编写成函数，放在公共函数库中供大家选用。程序设计人员要善于利用函数，以减少重复编写程序段的工作量。

先举一个函数调用的简单例子。

例8.1 函数调用的简单例子。

```
#include <stdio.h>
void main()
{
  void printstar();                    /* 对 printstar 函数进行声明 */
  void print_message();                /* 对 print_message 函数进行声明 */
  printstar();                         /* 调用 printstar 函数 */
  print_message();                     /* 调用 print_message 函数 */
  printstar();                         /* 调用 printstar 函数 */
}

void printstar()                       /* 定义 printstar 函数 */
{
  printf("* * * * * * * * * * * * * * * * * * * * *\n");
}
void print_message()                   /* 定义 print_message 函数 */
{
  printf("␣␣How do you do! \n");
}
```

运行情况如下：

```
* * * * * * * * * * * * * * * * * * * *
    H o w   d o   y o u   d o !
* * * * * * * * * * * * * * * * * * * *
```

printstar 和 print_message 都是用户定义的函数名，分别用来输出一排“*”号和一行信息。在定义这两个函数时指定函数的类型为 void，意为函数无类型，即无函数值，也就是说，执行这两个函数后不会把任何值带回 main 函数。

说明：

(1) 一个 C 程序由一个或多个程序模块组成，每一个程序模块作为一个源程序文件。对较大的程序，一般不希望把所有内容全放在一个文件中，而是将它们分别放在若干个源文件中，再由若干个源程序文件组成一个 C 程序。这样便于分别编写、分别编译，提高调试效率。一个源程序文件可以为多个 C 程序共用。

(2) 一个源程序文件由一个或多个函数以及其他有关内容(如命令行、数据定义等)组成。一个源程序文件是一个编译单位，在程序编译时是以源程序文件为单位进行编译的，而不是以函数为单位进行编译的。

(3) C 程序的执行是从 main 函数开始的，如是在 main 函数中调用其他函数，在调用后流程返回到 main 函数，在 main 函数中结束整个程序的运行。

(4) 所有函数都是平行的，即在定义函数时是分别进行的，是互相独立的。一个函数并不从属于另一个函数，即函数不能嵌套定义。函数间可以互相调用，但不能调用 main 函数。main 函数是系统调用的。

(5) 从用户使用的角度看，函数有两种。

① 标准函数。标准函数即库函数，它是由系统提供的，用户不必自己定义而直接使用它们。应该说明，不同的 C 语言编译系统提供的库函数的数量和功能会有一些不同，当然许多基本的函数是共同的。

② 用户自己定义的函数。它是用以解决用户专门需要的函数。

(6) 从函数的形式看，函数分两类。

① 无参函数。如例 8.1 中的 printstar 和 print_message 就是无参函数。在调用无参函数时，主调函数不向被调用函数传递数据。无参函数一般用来执行指定的一组操作。例如，例 8.1 程序中的 printstar 函数的作用是输出 16 个星号。无参函数可以带回或不带回函数值，但一般以不带回函数值的居多。

② 有参函数。在调用函数时，主调函数在调用被调用函数时，通过参数向被调用函数传递数据，一般情况下，执行被调用函数时会得到一个函数值，供主调函数使用。

8.2 函数定义的一般形式

8.2.1 无参函数定义的一般形式

定义无参函数的一般形式为：

类型标识符 函数名()

{

声明部分

语句部分

}

例 8.1 中的 printstar 和 print_message 函数都是无参函数。

在定义函数时要用“类型标识符”指定函数值的类型，即函数带回来的值的类型。例 8.1 中的 printstar 和 print_message 函数为 void 类型，表示不需要带回函数值。

8.2.2 有参函数定义的一般形式

定义有参函数的一般形式为：

类型标识符 函数名(形式参数表列)

{

声明部分

语句部分

}

例如：

```
int max(int x,int y)
    {
      int z;                          /* 函数体中的声明部分 */
      z=x>y? x: y;
      return(z);
    }
```

这是一个求 x 和 y 二者中大者的函数，第 1 行第一个关键字 int 表示函数值是整型的。max 为函数名。括号中有两个形式参数 x 和 y，它们都是整型的。在调用此函数时，主调函数把实际参数的值传递给被调用函数中的形式参数 x 和 y。大括号内是函数体，它包括声明部分和语句部分。声明部分包括对函数中用到的变量进行定义以及对要调用的函数进行声明(见 8.4.3 小节)等内容。在函数体的语句中求出 z 的值(为 x 与 y 中大者)，return(z)的作用是将 z 的值作为函数值带回到主调函数中。return 后面的括号中的值(z)作为函数带回的值(称函数返回值)。在函数定义时已指定 max 函数为整型，在函数体中定义 z 为整型，二者是一致的，将 z 作为函数 max 的值带回调用函数(见例 8.2)。

如果在定义函数时不指定函数类型，系统会隐含指定函数类型为 int 型。因此上面定义的 max 函数左端的 int 可以省写。

8.2.3 空函数

在程序设计中有时会用到空函数，它的形式为：

类型说明符 函数名( )

{ }

例如：

```
void dummy()
  {    }
```

调用此函数时，什么工作也不做，没有任何实际作用。在主调函数中写上“dummy();”表明“这里要调用一个函数”，而现在这个函数没有起作用，等以后扩充函数功能时补充上。

在程序设计中往往根据需要确定若干模块，分别由一些函数来实现。而在第一阶段只设计最基本的模块，其他一些次要功能或锦上添花的功能则在以后需要时陆续补上。在编写程序的开始阶段，可以在将来准备扩充功能的地方写上一个空函数（函数名取将来采用的实际函数名（如用 merge()，matproduct()，concatenate()，shell()等，分别代表合并、矩阵相乘、字符串连接、希尔法排序等），只是这些函数未编好，先占一个位置，以后用一个编好的函数代替它。这样做，程序的结构清楚，可读性好，以后扩充新功能方便，对程序结构影响不大。空函数在程序设计中常常是有用的。

8.3 函数参数和函数的值

8.3.1 形式参数和实际参数

在调用函数时，大多数情况下，主调函数和被调用函数之间有数据传递关系。这就是前面提到的有参函数。前面已提到：在定义函数时函数名后面括号中的变量名称为“形式参数”（简称“形参”），在主调函数中调用一个函数时，函数名后面括号中的参数（可以是一个表达式）称为“实际参数”（简称“实参”）。

例 8.2 调用函数时的数据传递。

```
#include <stdio.h>
void main()
{
  int max(int x,int y);                /* 对 max 函数的声明 */
  int a,b,c;
  scanf("%d,%d",&a,&b);
  c=max(a,b);
  printf("Max is %d",c);
}
int max(int x,int y)                   /*定义有参函数 max */
{
  int z;
  z=x>y? x: y;
  return(z);
}
```

运行情况如下：

```
7,8↙
Max is 8
```

程序中第 10～15 行是一个函数定义(注意第 10 行的末尾没有分号)。第 10 行定义了一个函数名 max 和指定两个形参名 x、y 及形参类型。程序第 7 行包含一个函数调用,max 后面括号内的 a、b 是实参。a 和 b 是在 main 函数中定义的变量,x 和 y 是函数 max 中的形式参数。通过函数调用,使两个函数中的数据发生联系,见图 8-2。

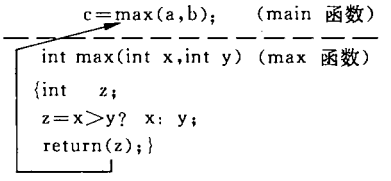


图 8-2

关于形参与实参的说明:

(1) 在定义函数中指定的形参,在未出现函数调用时,它们并不占内存中的存储单元。只有在发生函数调用时,函数 max 中的形参才被分配内存单元。在调用结束后,形参所占的内存单元也被释放。

(2) 实参可以是常量、变量或表达式,例如:

```
max(3,a+b);
```

但要求它们有确定的值。在调用时将实参的值赋给形参。

(3) 在被定义的函数中,必须指定形参的类型(见例 8.2 程序第 10 行)。

(4) 实参与形参的类型应相同或赋值兼容。例 8.2 中实参和形参都是整型,这是合法的、正确的。如果实参为整型而形参 x 为实型,或者相反,则按第 3 章介绍的不同类型数值的赋值规则进行转换。例如实参值 a 为 3.5,而形参 x 为整型,则将实数 3.5 转换成整数 3,然后送到形参 x。字符型与整型可以互相通用。

(5) 在 C 语言中,实参向形参的数据传递是"值传递",单向传递,只由实参传给形参,而不能由形参传回来给实参。在内存中,实参单元与形参单元是不同的单元,如图 8-3 所示。

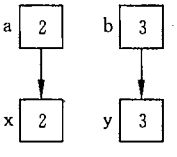


图 8-3

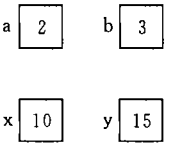


图 8-4

在调用函数时,给形参分配存储单元,并将实参对应的值传递给形参,调用结束后,形参单元被释放,实参单元仍保留并维持原值。因此,在执行一个被调用函数时,形参的值如果发生改变,并不会改变主调函数的实参的值。例如,若在执行函数过程中 x 和 y 的值变为 10 和 15,而 a 和 b 仍为 2 和 3,见图 8-4。

8.3.2 函数的返回值

通常，希望通过函数调用使主调函数能得到一个确定的值，这就是函数的返回值。例如，例 8.2 中，max(2,3)的值是 3，max(5,2)的值是 5。赋值语句将这个函数值赋给变量 c。下面对函数值作一些说明。

(1) 函数的返回值是通过函数中的 return 语句获得的。return 语句将被调用函数中的一个确定值带回主调函数中去。见图 8-2 中从 return 语句返回的箭头。如果需要从被调用函数带回一个函数值(供主调函数使用)，被调用函数中必须包含 return 语句。如果不需要从被调用函数带回函数值可以不要 return 语句。

一个函数中可以有一个以上的 return 语句，执行到哪一个 return 语句，哪一个语句起作用。

return 语句后面的括号也可以不要，如“return z;”它与“return(z);”等价。

return 后面的值可以是一个表达式。例如，例 8.2 中的函数 max 可以改写如下：

```
max(int x,int y)
  {
    return(x>y? x: y);
  }
```

这样的函数体更为简短，只用一个 return 语句就把求值和返回都解决了。

(2) 函数值的类型。既然函数有返回值，这个值当然应属于某一个确定的类型，应当在定义函数时指定函数值的类型。例如下面是 3 个函数的首行：

```
int max(float x,float y)                    /* 函数值为整型 */
char letter(char c1,char c2)                /* 函数值为字符型 */
double min(int x,int y)                     /* 函数值为双精度型 */
```

在 C 语言中，凡不加类型说明的函数，自动按整型处理。例 8.2 中的 max 函数首行的函数类型 int 可以省写，用 Turbo C 2.0 编译程序时能通过，但用 Turbo C++ 3.0 编译程序时不能通过，因为 C++ 要求所有函数都必须指定函数类型。建议读者在定义时对所有函数都指定函数类型。

(3) 在定义函数时指定的函数类型一般应该和 return 语句中的表达式类型一致。例如，例 8.2 中指定 max 函数值为整型，而变量 z 也被指定为整型，通过 return 语句把 z 的值作为 max 的函数值，由 max 带回主调函数。z 的类型与 max 函数的类型是一致的，是正确的。

如果函数值的类型和 return 语句中表达式的值不一致，则以函数类型为准。对数值型数据，可以自动进行类型转换。即函数类型决定返回值的类型。

例 8.3 返回值类型与函数类型不同。将例 8.2 稍作改动(注意：是变量的类型改动)。

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

```
{
    int max(float x,float y);
    float a,b;
    int c;
    scanf("%f,%f,",&a,&b);
    c=max(a,b);
    printf("Max is %d\n",c);
}
int max(float x,float y)
{
    float z;                    /* z为实型变量 */
    z=x>y? x: y;
    return(z);
}
```

运行情况如下：

```
1.5, 2.5↙
Max is 2
```

函数 max 定义为整型，而 return 语句中的 z 为实型，二者不一致，按上述规定，先将 z 转换为整型，然后 max(x,y)带回一个整型值 2 返回主调函数 main。如果将 main 函数中的 c 定义为实型，用%f 格式符输出，也是输出 2.000000。

有时，可以利用这一特点进行类型转换，如在函数中进行实型运算，希望返回的是整型量，可让系统自动完成类型转换。但这种做法往往使程序不清晰，可读性降低，容易弄错，而且并不是所有的类型都能互相转换的(如实数与字符类型数据之间)。因此建议初学者不要采用这种方法，而应做到使函数类型与 return 返回值的类型一致。

(4) 对于不带回值的函数，应当用“void”定义函数为“无类型”(或称“空类型”)。这样，系统就保证不使函数带回任何值，即禁止在调用函数中使用被调用函数的返回值。此时在函数体中不得出现 return 语句。

8.4 函数的调用

8.4.1 函数调用的一般形式

函数调用的一般形式为

函数名(实参表列)；

如果是调用无参函数，则“实参表列”可以没有，但括号不能省略，见例 8.1。如果实参表列包含多个实参，则各参数间用逗号隔开。实参与形参的个数应相等，类型应匹配。实参与形参按顺序对应，一一传递数据。但应说明，如果实参表列包括多个实参，对实参求值的顺序并不是确定的，有的系统按自左至右顺序求实参的值，有的系统则按自右至左顺序。许多 C 版本(例如 Turbo C++)是按自右而左的顺序求值。

例 8.4 实参求值的顺序。

```
#include <stdio.h>
void main()
{
    int f(int a,int b);                    /* 函数声明 */
    int i=2,p;
    p=f(i,++i);                            /* 函数调用 */
    printf("%d\n",p);
}

int f(int a,int b)                         /* 函数定义 */
{
    int c;
    if(a>b) c=1;
    else if(a==b) c=0;
    else c=-1;
    return(c);
}
```

在 Turbo C 2.0 和 Turbo C++ 3.0 系统上运行的结果如下：

```
0
```

如果按自左至右顺序求实参的值，则函数调用相当于 f(2,3)，程序运行应得结果为“-1”。若按自右至左顺序求实参的值，则它相当于 f(3,3)，程序运行结果为“0”。如果读者用的是其他编译系统，可以用上面的程序测试一下，以便知道它所处理的方法。由于存在上述情况，使程序通用性受到影响。因此应当避免这种容易引起不同理解的情况。如果本意是按自左而右顺序求实参的值，可以改写为：

```
j=i;
k=++i;
p=f(j,k);
```

如果本意是自右而左求实参的值，可改写为：

```
j=++i;
p=f(j,j);
```

这种情况在 printf 函数中也同样存在，例如：

```
printf("%d,%d",i,i++);
```

也会发生上述同样的问题，若 i 的原值为 3，在 Turbo C 2.0 和 Turbo C++ 3.0 上运行结果为 4,3。请读者务必注意，应该避免这种容易混淆的用法。

8.4.2 函数调用的方式

按函数在程序中出现的位置来分，可以有以下 3 种函数调用方式。

1. 函数语句

把函数调用作为一个语句。如例 8.1 中的“printstar();”,这时不要求函数带回值,只要求函数完成一定的操作。

2. 函数表达式

函数出现在一个表达式中,这种表达式称为函数表达式。这时要求函数带回一个确定的值以参加表达式的运算。例如:

```
c=2 * max(a,b);
```

函数 max 是表达式的一部分,它的值乘 2 再赋给 c。

3. 函数参数

函数调用作为一个函数的实参。例如:

```
m=max(a,max(b,c));
```

其中 max(b,c)是一次函数调用,它的值作为 max 另一次调用的实参。m 的值是 a、b、c 三者中的最大者。又如:

```
printf ("%d", max (a,b));
```

也是把 max(a,b)作为 printf 函数的一个参数。

函数调用作为函数的参数,实质上也是函数表达式形式调用的一种,因为函数的参数本来就要求是表达式形式。

8.4.3 对被调用函数的声明和函数原型

在一个函数中调用另一个函数(即被调用函数)需要具备的条件如下。

(1) 首先被调用的函数必须是已经存在的函数(是库函数或用户自己定义的函数)。但光有这一条件还不够。

(2) 如果使用库函数,还应该在本文件开头用 #include 命令将调用有关库函数时所需用到的信息“包含”到本文件中来。例如,前几章中已经用过的命令:

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

其中“stdio.h”是一个“头文件”。在 stdio.h 文件中包含了输入输出库函数所用到的一些宏定义信息。如果不包含“stdio.h”文件中的信息,就无法使用输入输出库中的函数。同样,使用数学库中的函数,应该用 #include <math.h>。其中,.h 是头文件所用的后缀,标志头文件(header file)。有关宏定义等概念请见第 9 章。

(3) 如果使用用户自己定义的函数,而该函数的位置在调用它的函数(即主调函数)的后面(在同一个文件中),应该在主调函数中对被调用的函数作声明。在前面的例子中已出现过函数的声明,今再进一步作些说明。

“声明”一词的原文是 declaration，过去在许多书中把它译为“说明”，近年来，愈来愈多的计算机专家提出应译为声明，作者也认为译为“声明”更确切，表意更清楚。声明的作用是把函数名、函数参数的个数和参数类型等信息通知编译系统，以便在遇到函数调用时，编译系统能正确识别函数并检查调用是否合法。

例 8.5 对被调用的函数作声明。

```
#include <stdio.h>
void main()
  {
    float add(float x, float y);          /*对被调用函数 add 的声明*/
    float a,b,c;
    scanf("%f,%f",&a,&b);
    c=add(a,b);
    printf("sum is %f\n",c);
  }
float add(float x,float y)                /*函数首部*/
  {
    float z;                              /* 函数体 */
    z=x+y;
    return(z);
  }
```

运行情况如下：

```
3.6,6.5↙
sum is 10.100000
```

这是一个很简单的函数调用，函数 add 的作用是求两个实数之和，得到的函数值也是实型。程序第 3 行：

```
float add(float x, float y);
```

是对被调用的 add 函数作声明。其实，在函数声明中也可以不写形参名，而只写形参的类型，如上面的声明可以写为

```
float add(float,float);
```

编译系统只检查参数个数和参数类型，而不检查参数名。

以上的函数声明称为函数原型(function prototype)。使用函数原型是 ANSI C 的一个重要特点。从例 8.5 中可以看到 main 函数的位置在 add 函数的前面，而在进行编译时是从上到下逐行进行的，如果没有对函数的声明，当编译到程序第 6 行时，编译系统无法确定 add 是不是函数名，也无法判断实参(a 和 b)的类型和个数是否正确，因而无法进行正确性的检查。如果不作检查，在运行时才发现实参与形参的类型或个数不一致，出现运行错误。但是在运行阶段发现错误并重新调试程序，是比较麻烦的，工作量也较大。应当在编译阶段尽可能多地发现错误，随之纠正错误。

现在，在函数调用之前用函数原型做了函数声明。因此编译系统记下了所需调用的函数的有关信息，在对“c＝add(a,b);”进行编译时就“有章可循”了。编译系统根据函数的原型对函数的调用的合法性进行全面的检查。与函数原型不匹配的函数调用会导致编译出错，它属于语法错误。用户根据屏幕显示的出错信息很容易发现和纠正错误。

注意：对函数的“定义”和“声明”不是一回事。函数的定义是指对函数功能的确立，包括指定函数名、函数值类型、形参及其类型、函数体等，它是一个完整的、独立的函数单位。而函数的声明的作用则是把函数的名字、函数类型以及形参的类型、个数和顺序通知编译系统，以便在调用该函数时系统按此进行对照检查(例如，函数名是否正确，实参与形参的类型和个数是否一致)。

从程序中可以看到对函数的声明与函数定义中的第 1 行(函数首部)基本上是相同的，只差一个分号。因此可以简单地照写已定义的函数的首部，再加一个分号，就成为了对函数的“声明”。

函数原型的一般形式有两种，分别为

(1) **函数类型 函数名(参数类型 1,参数类型 2,…,参数类型 n);**

(2) **函数类型 函数名(参数类型 1,参数名 1,参数类型 2,参数名 2,…,参数类型 n,参数名 n);**

第(1)种形式是基本的形式。为了便于阅读程序，也允许在函数原型中加上参数名，就成了第(2)种形式。但编译系统不检查参数名。因此参数名是什么都无所谓。上面程序中的声明也可以写成：

```
float add(float a, float b);      /* 参数名不用 x、y,而用 a、b */
```

效果完全相同。

应当保证函数原型与函数首部写法上的一致，即函数类型、函数名、参数个数、参数类型和参数顺序必须相同。函数调用时函数名、实参个数应与函数原型一致。实参类型必须与函数原型中的形参类型赋值兼容，按第 3 章介绍的赋值规则进行类型转换。如果不是赋值兼容，就按出错处理。

用函数原型来声明函数，能减少编写程序时可能出现的错误。由于函数声明的位置与函数调用语句的位置比较近，因此在写程序时便于就近参照函数原型来书写函数调用，不易出错。

说明：

(1) 以前的 C 语言版本的函数声明方式不是采用函数原型，而只声明函数名和函数类型。例如在例 8.5 中也可以采用下面的函数声明形式：

```
float add( );
```

不包括参数类型和参数个数。系统不检查参数类型和参数个数。新版本也兼容这种用法，但不提倡这种用法，因为它未对函数调用的合法性进行全面的检查。

(2) 如果被调用函数的定义出现在主调函数之前，可以不必加以声明。因为编译系统已经先知道了已定义函数的有关情况，会根据函数首部提供的信息对函数的调用作正确性检查。

如果把例 8.5 改写如下(即把 main 函数放在 add 函数的后面),就不必在 main 函数中对 add 声明。

```
#include <stdio.h>
float add(float x,float y)                    /* 函数定义 */
{
  float z;
  z=x+y;
  return(z);
}
void main()
{                                             /* 不必对 add 函数作声明 */
  float a,b,c;
  scanf("%f,%f",&a,&b);
  c=add(a,b);
  printf("%f\n",c);
}
```

(3) 如果已在文件的开头(在所有函数之前),已对本文件中所调用的函数进行了声明,则在各函数中不必对其所调用的函数再作声明 。例如:

```
char letter(char, char);              /* 以下 3 行在所有函数之前,且在函数外部 */
float f(float, float);
int i(float, float);

void main()                           /* 在 main 函数中要调用 letter、f 和 i 函数 */
{                                     /* 不必对它所调用的这 3 个函数进行声明 */
  ⋮
}
/* 下面定义被 main 函数调用的 3 个函数 */
char letter(char c1,char c2)          /* 定义 letter 函数 */
{
  ⋮
}
float f(float x,float y)              /* 定义 f 函数 */
{
  ⋮
}
int i(float j,float k)                /* 定义 i 函数 */
{
  ⋮
}
```

由于在文件的开头已对要调用的函数进行了声明,因此编译系统从声明中已知道了函数的有关情况,所以不必在主调函数中再进行声明。

(4) 如果被调用的函数类型为整型,C 语言允许在调用函数前不必作函数原型声明。如例 8.4 可以写成以下形式:

```
void main()
{                              /* 可以没有函数原型声明 */
  int i=2,p;
  p=f(i,++i);                  /* 函数调用 */
  printf("%d\n",p);
}

f(int a,int b)                 /* 定义整型函数,省写了类型标识符 int */
{
  int c;
  if(a>b)
    c=1;
  else
    if(a==b)
      c=0;
    else
      c=-1;
  return(c);
}
```

但是使用这种方法,系统无法对函数参数的个数和类型进行检查。若调用函数时参数使用不当,在编译时也不会报错,而在运行时出错。而且,用 Turbo C 和 Visual C++ 时,要求对所有被调用的函数进行声明,上面的程序在 Turbo C 和 Visual C++ 中通不过。因此,为了程序清晰和安全以及通用性,编写程序时最好都加上函数原型声明(如例 8.2 那样)。读者在看已有的 C 程序时,可能会遇到类似上面的程序,应能理解和修改。

8.5 函数的嵌套调用

C 语言的函数定义是互相平行、独立的,也就是说,在定义函数时,一个函数内不能包含另一个函数,这是和 Pascal 不同的(Pascal 允许在定义一个函数时,其函数体内又包含另一个函数的完整定义,这就是嵌套定义。这个内嵌的函数只能被包含它的函数所调用,其他函数不能调用它)。

C 语言不能嵌套定义函数,但可以嵌套调用函数,也就是说,在调用一个函数的过程中,又调用另一个函数,见图 8-5。

图 8-5 表示的是两层嵌套(包括 main 函数共 3 层函数),其执行过程是:

(1) 执行 main 函数的开头部分;

(2) 遇函数调用语句,调用函数 a,流程转去 a 函数;

(3) 执行 a 函数的开头部分;

(4) 遇函数调用语句,调用函数 b,流程转去函数 b;

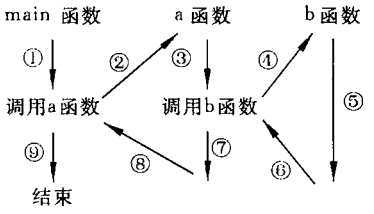


图 8-5

(5) 执行 b 函数，如果再无其他嵌套的函数，则完成 b 函数的全部操作；

(6) 返回到 a 函数中调用 b 函数的位置；

(7) 继续执行 a 函数中尚未执行的部分，直到 a 函数结束；

(8) 返回 main 函数中调用 a 函数的位置；

(9) 继续执行 main 函数的剩余部分直到结束。

例 8.6 用弦截法求方程 $f(x)=x^3-5x^2+16x-80=0$ 的根。方法如下。

(1) 取两个不同点 x_1、x_2，如果 $f(x_1)$和 $f(x_2)$符号相反，则(x_1,x_2)区间内必有一个根。如果 $f(x_1)$与 $f(x_2)$同符号，则应改变 x_1、x_2，直到 $f(x_1)$、$f(x_2)$异号为止。注意 x_1、x_2 的值不应差太大，以保证(x_1,x_2)区间内只有一个根。

(2) 连接$(x_1,f(x_1))$和$(x_2,f(x_2))$两点，此线(即弦)交 x 轴于 x，见图 8-6。

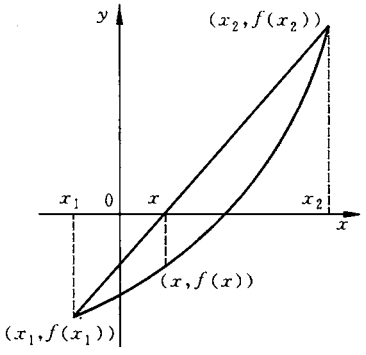


图 8-6

x 点坐标可用下式求出：

$$x=\frac{x_1f(x_2)-x_2f(x_1)}{f(x_2)-f(x_1)}$$

再从 x 求出 $f(x)$。

(3) 若 $f(x)$与 $f(x_1)$同符号，则根必在(x,x_2)区间内，此时将 x 作为新的 x_1。如果 $f(x)$与 $f(x_2)$同符号，则表示根在(x_1,x)区间内，将 x 作为新的 x_2。

(4) 重复步骤(2)和步骤(3)，直到 $|f(x)|<\varepsilon$ 为止，ε 为一个很小的数，例如 10^{-6}。此时认为 $f(x)\approx 0$。

根据上述思路画出 N-S 流程图，见图 8-7。

分别用几个函数来实现各部分功能：

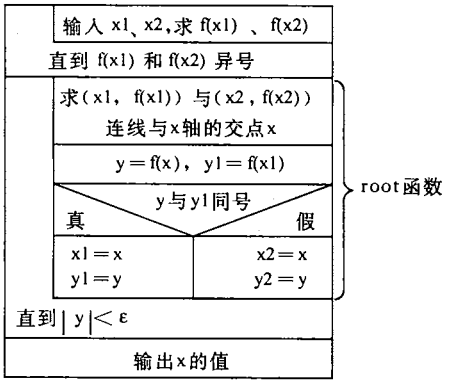


图 8-7

(1) 用函数 $f(x)$代表 x 的函数：$x^3-5x^2+16x-80$。

(2) 用函数调用 xpoint (x1,x2)来求$(x_1,f(x_1))$和$(x_2,f(x_2))$的连线与 x 轴的交点 x 的坐标。

(3) 用函数调用 root (x1,x2)来求(x_1,x_2)区间的那个实根。显然,执行 root 函数过程中要用到函数 xpoint,而执行 xpoint 函数过程中要用到 f 函数。

请读者先分析下面的程序。

```
#include <stdio.h>
#include <math.h>
float f(float x)                    /* 定义 f 函数,以实现 f(x)=x³-5x²+16x-80 */
{
  float y;
  y=((x-5.0)*x+16.0)*x-80.0;
  return(y);
}

float xpoint(float x1,float x2)     /* 定义 xpoint 函数,求出弦与 x 轴交点 */
{
  float y;
  y=(x1*f(x2)-x2*f(x1))/(f(x2)-f(x1));
  return(y);
}

float root(float x1,float x2)       /* 定义 root 函数,求近似根 */
{
  float x,y,y1;
  y1=f(x1);
  do
    {
      x=xpoint(x1,x2);
```

```
        y=f(x);
        if(y*y1>0)                    /*f(x)与f(x1)同符号*/
          {
            y1=y;
            x1=x;
          }
        else
          x2=x;
        }while(fabs(y)>=0.0001);
        return(x);
    }

    void main()                       /*主函数*/
    {
      float x1,x2,f1,f2,x;
      do
        {
          printf("input x1,x2:\n");
          scanf("%f,%f",&x1,&x2);
          f1=f(x1);
          f2=f(x2);
        }
      while(f1*f2>=0);
      x=root(x1,x2);
      printf("A root of equation is%8.4f\n",x);
    }
```

运行情况如下：

```
input x1,x2:
2,6↙
A root of equation is 5.0000
```

从程序中可以看到：

(1) 在定义函数时，3 个函数 f、xpoint、root 是互相独立的，并不互相从属。这 3 个函数均定为实型。

(2) 3 个被调用的函数的定义位置均在调用它的函数之前，例如在 main 函数中要调用 f 函数和 root 函数，而 f 函数和 root 函数是在 main 函数出现之前定义的；在 root 函数中要调用 f 函数和 xpoint 函数，而 f 函数和 xpoint 函数是在 root 函数出现之前定义的；在 xpoint 函数中要调用 f 函数，而 f 函数是在 xpoint 函数出现之前定义的。因此不必对这 3 个函数进行声明。

(3) 程序从 main 函数开始执行。先执行一个 do…while 循环，其作用是输入 x1 和 x2，判别 f(x1)和 f(x2)是否异号。如果不是异号，则重新输入 x1 和 x2，直到满足 f(x1)与 f(x2)异号为止。然后调用函数 root(x1,x2)求根 x。调用 root 函数过程中，要调用

xpoint 函数来求(x1,f(x1))和(x2,f(x2))连线与 x 轴的交点 x。在调用 xpoint 函数过程中要用到函数 f 来求 x1 和 x2 的相应的函数值 f(x1)和 f(x2)。这就是函数的嵌套调用，见图 8-8。

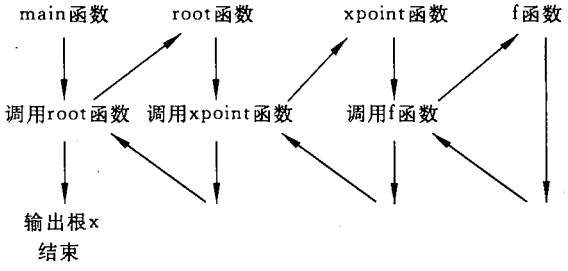


图 8-8

(4) 在 root 函数中要用到求绝对值的函数 fabs，它是对实型数求绝对值的标准函数，属于数学函数库，故在文件开头用

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

把使用数学库函数时所需的信息包含进来。

8.6 函数的递归调用

在调用一个函数的过程中又出现直接或间接地调用该函数本身，称为函数的递归调用。C 语言的特点之一就在于允许函数的递归调用。例如：

```
int f(int x)
  {
    int y,z;
    z=f(y);
    return(2*z);
  }
```

在调用函数 f 的过程中，又要调用 f 函数，这是直接调用本函数，见图 8-9。

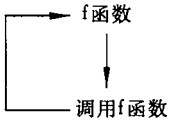


图 8-9

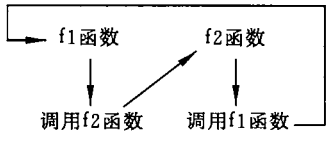


图 8-10

如果在调用 f1 函数过程中要调用 f2 函数，而在调用 f2 函数过程中又要调用 f1 函数，就是间接调用本函数，见图 8-10。

从图 8-9 和图 8-10 可以看到，这两种递归调用都是无终止的自身调用。显然，程序中不应出现这种无终止的递归调用，而只应出现有限次数的、有终止的递归调用，这可以

用 if 语句来控制，只有在某一条件成立时才继续执行递归调用；否则就不再继续。

关于递归的概念，有些初学者感到不好理解，下面用一个通俗的例子来说明。

例 8.7 有 5 个人坐在一起，问第 5 个人多少岁？他说比第 4 个人大 2 岁。问第 4 个人岁数，他说比第 3 个人大 2 岁。问第 3 个人，又说比第 2 个人大 2 岁。问第 2 个人，说比第 1 个人大 2 岁。最后问第 1 个人，他说是 10 岁。请问第 5 个人多大。

显然，这是一个递归问题。要求第 5 个人的年龄，就必须先知道第 4 个人的年龄，而第 4 个人的年龄也不知道，要求第 4 个人的年龄必须先知道第 3 个人的年龄，而第 3 个人的年龄又取决于第 2 个人的年龄，第 2 个人的年龄取决于第 1 个人的年龄。而且每一个人的年龄都比其前 1 个人的年龄大 2。即：

age(5)＝age(4)＋2

age(4)＝age(3)＋2

age(3)＝age(2)＋2

age(2)＝age(1)＋2

age(1)＝10

可以用数学公式表述如下：

$$\text{age}(n)=\begin{cases}10 & n=1\\ \text{age}(n-1)+2 & n>1\end{cases}$$

可以看到，当 $n>1$ 时，求第 n 个人的年龄的公式是相同的。因此可以用一个函数表示上述关系。图 8-11 表示求第 5 个人年龄的过程。

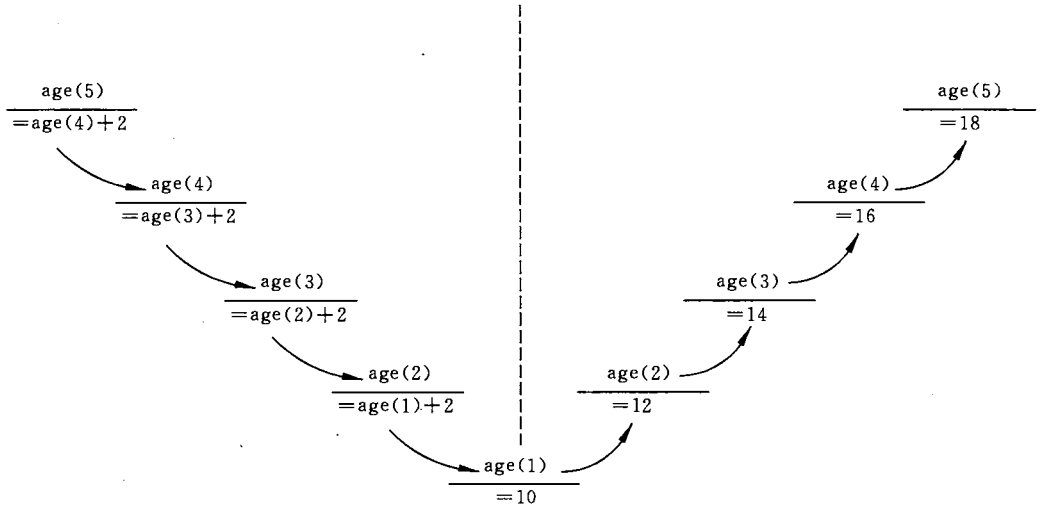


图 8-11

从图 8-11 可知，求解可分成两个阶段：第一阶段是“回推”，即将第 n 个人的年龄表示为第(n－1)个人年龄的函数，而第(n－1)个人的年龄仍然不知道，还要“回推”到第(n－2)个人的年龄……直到第 1 个人的年龄。此时 age(1)已知，不必再向前推了。然后开始第二阶段，采用递推方法，从第 1 个人的已知年龄推算出第 2 个人的年龄(12 岁)，从第 2

个人的年龄推算出第 3 个人的年龄(14 岁)……一直推算出第 5 个人的年龄(18 岁)为止。也就是说,一个递归的问题可以分为“回推”和“递推”两个阶段。要经历若干步才能求出最后的值。显而易见,如果要求递归过程不是无限制进行下去,必须具有一个结束递归过程的条件。例如,age(1)=10,就是使递归结束的条件。

可以用一个函数来描述上述递归过程:

```
int age(int n)                /*求年龄的递归函数*/
{
  int c;                      /* c用作存放函数的返回值的变量 */
  if(n==1)
      c=10;
  else
      c=age(n-1)+2;
  return(c);
}
```

用一个主函数调用 age 函数,求得第 5 人的年龄:

```
#include <stdio.h>
void main()
{
  printf("%d\n",age(5));
}
```

运行结果如下:

```
18
```

main 函数中只有一个语句。整个问题的求解全靠一个 age(5)函数调用来解决。函数调用过程如图 8-12 所示。

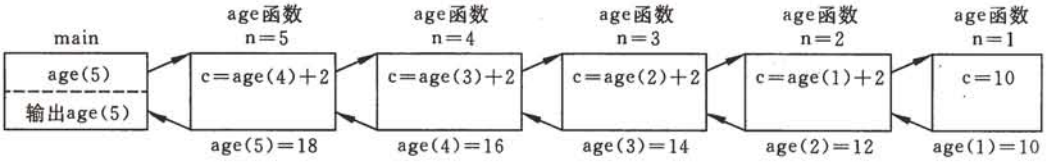


图 8-12

从图 8-12 可以看到:age 函数共被调用 5 次,即 age(5)、age(4)、age(3)、age(2)、age(1)。其中 age(5)是 main 函数调用的,其余 4 次是在 age 函数中调用的,即递归调用 4 次。请读者仔细分析调用的过程。应当强调说明的是在某一次调用 age 函数时并不是立即得到 age(n)的值,而是一次又一次地进行递归调用,到 age(1)时才有确定的值,然后再递推出 age(2)、age(3)、age(4)、age(5)。请读者将程序和图 8-11 和图 8-12 结合起来认真分析。

例 8.8 用递归方法求 $n!$。

求 $n!$ 可以用递推方法，即从 1 开始，乘 2，再乘 3……一直乘到 n。这种方法容易理解，也容易实现。递推法的特点是从一个已知的事实出发，按一定规律推出下一个事实，再从这个新的已知的事实出发，再向下推出一个新的事实……这是和递归不同的。

求 $n!$ 也可以用递归方法，即 5! 等于 4! ×5，而 4! =3! ×4…1! =1。可用下面的递归公式表示：

$$n! = \begin{cases} 1 & n=0,1 \\ n \cdot (n-1)! & n>1 \end{cases}$$

有了例 8.7 的基础，很容易写出本题的程序：

```
#include <stdio.h>
void main()
{
  float fac(int n);                    /* 对 fac 函数的声明 */
  int n;
  float y;
  printf("input an integer number:");
  scanf("%d",&n);
  y=fac(n);
  printf("%d!=%10.0f\n",n,y);
}

float fac(int n)                       /* 定义 fac 函数 */
{
  float f;
  if(n<0)
    {
      printf("n<0,dataerror!");
    }
  else
      if(n==0||n==1)
      f=1;
  else
      f=fac(n-1)*n;
  return(f);
}
```

运行情况如下：

```
input an integer number:10↙
10!=␣␣3628800.
```

例 8.9 Hanoi(汉诺)塔问题。这是一个古典的数学问题，是一个用递归方法解题的典型例子。问题是这样的：古代有一个梵塔，塔内有 3 个座 A、B、C，开始时 A 座上有 64 个盘子，盘子大小不等，大的在下，小的在上(见图 8-13)。有一个老和尚想把这 64 个盘子

从 A 座移到 C 座，但每次只允许移动一个盘，且在移动过程中在 3 个座上都始终保持大盘在下，小盘在上。在移动过程中可以利用 B 座，要求编程序输出移动的步骤。

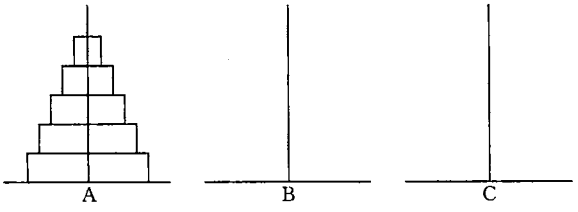


图 8-13

读者是不大可能直接写出移动盘子的每一个具体步骤的。请读者试验一下按上面的规律将 5 个盘子从 A 座移到 C 座，能否直接写出每一步骤？

老和尚自然会这样想：假如有另外一个和尚能有办法将 63 个盘子从一个座移到另一座。那么，问题就解决了。此时老和尚只需这样做：

(1) 命令第 2 个和尚将 63 个盘子从 A 座移到 B 座；

(2) 自己将 1 个盘子(最底下的、最大的盘子)从 A 座移到 C 座；

(3) 再命令第 2 个和尚将 63 个盘子从 B 座移到 C 座。

至此，全部任务完成了。这就是递归方法。但是，有一个问题实际上未解决：第 2 个和尚怎样才能将 63 个盘子从 A 座移到 B 座？

为了解决将 63 个盘子从 A 座移到 B 座，第 2 个和尚又想：如果有人能将 62 个盘子从一个座移到另一座，我就能将 63 个盘子从 A 座移到 B 座，他是这样做的：

(1) 命令第 3 个和尚将 62 个盘子从 A 座移到 C 座；

(2) 自己将 1 个盘子从 A 座移到 B 座；

(3) 再命令第 3 个和尚将 62 个盘子从 C 座移到 B 座。

再进行一次递归。如此“层层下放”，直到后来找到第 63 个和尚，让他完成将 2 个盘子从一个座移到另一座，进行到此，问题就接近解决了。最后找到第 64 个和尚，让他完成将 1 个盘子从一个座移到另一座，至此，全部工作都已落实，都是可以执行的。

可以看出，递归的结束条件是最后一个和尚只需移一个盘子；否则递归还要继续进行下去。

应当说明，只有第 64 个和尚的任务完成后，第 63 个和尚的任务才能完成。只有第 2 个到第 64 个和尚任务完成后，第 1 个和尚的任务才能完成。这是一个典型的递归的问题。

为便于理解，先分析将 A 座上 3 个盘子移到 C 座上的过程：

(1) 将 A 座上 2 个盘子移到 B 座上(借助 C)；

(2) 将 A 座上 1 个盘子移到 C 座上；

(3) 将 B 座上 2 个盘子移到 C 座上(借助 A)。

其中第(2)步可以直接实现。第(1)步又可用递归方法分解为：

1.1 将 A 上 1 个盘子从 A 移到 C；

1.2 将 A 上 1 个盘子从 A 移到 B；

1.3 将 C 上 1 个盘子从 C 移到 B。

第(3)步可以分解为：

3.1 将 B 上 1 个盘子从 B 移到 A 上；

3.2 将 B 上 1 个盘子从 B 移到 C 上；

3.3 将 A 上 1 个盘子从 A 移到 C 上。

将以上综合起来，可得到移动 3 个盘子的步骤为：

A→C，A→B，C→B，A→C，B→A，B→C，A→C。

共经历 7 步。由此可推出：移动 n 个盘子要经历 2^n-1 步。如移 4 个盘子经历 15 步，移 5 个盘子经历 31 步，移 64 个盘子经历 $2^{64}-1$ 步。

由上面的分析可知：将 n 个盘子从 A 座移到 C 座可以分解为以下 3 个步骤：

(1) 将 A 上 $n-1$ 个盘借助 C 座先移到 B 座上；

(2) 把 A 座上剩下的一个盘移到 C 座上；

(3) 将 $n-1$ 个盘从 B 座借助于 A 座移到 C 座上。

上面第(1)步和第(3)步，都是把 $n-1$ 个盘从一个座移到另一个座上，采取的办法是一样的，只是座的名字不同而已。为使之一般化，可以将第(1)步和第(3)步表示为：

将“one”座上 $n-1$ 个盘移到“two”座(借助“three”座)。只是在第(1)步和第(3)步中，one、two、three 和 A、B、C 的对应关系不同。对第(1)步，对应关系是 one 对应 A，two 对应 B，three 对应 C。对第(3)步，是：one 对应 B，two 对应 C，three 对应 A。

因此，可以把上面 3 个步骤分成两类操作：

(1) 将 $n-1$ 个盘从一个座移到另一个座上($n>1$)。这就是大和尚让小和尚做的工作，它是一个递归的过程，即和尚将任务层层下放，直到第 64 个和尚为止。

(2) 将 1 个盘子从一个座上移到另一座上。这是大和尚自己做的工作。

下面编写程序。分别用两个函数实现以上的两类操作，用 hanoi 函数实现上面第 1 类操作(即模拟小和尚的任务)，用 move 函数实现上面第 2 类操作(模拟大和尚自己移盘)，函数调用 hanoi(n,one,two,three)表示将 n 个盘子从“one”座移到“three”座的过程(借助“two”座)。函数调用 move(x,y)表示将 1 个盘子从 x 座移到 y 座的过程。x 和 y 是代表 A、B、C 座之一，根据每次不同情况分别取 A、B、C 代入。

程序如下：

```
#include <stdio.h>
void main()
{
  void hanoi(int n,char one,char two,char three);        /* 对 hanoi 函数的声明 */
  int m;
  printf("input the number of diskes:");
  scanf("%d",&m);
  printf("The step to moveing %d diskes:\n",m);
  hanoi(m,'A','B','C');
}
```

```
void hanoi(int n,char one,char two,char three)          /* 定义 hanoi 函数 */
                                        /* 将 n 个盘从 one 座借助 two 座,移到 three 座 */
{
    void move(char x,char y);                           /* 对 move 函数的声明 */
    if(n==1)
        move(one,three);
    else
      {
        hanoi(n-1,one,three,two);
        move(one,three);
        hanoi(n-1,two,one,three);
      }
}

  void move(char x,char y)                              /* 定义 move 函数 */
  {
    printf("%c-->%c\n",x,y);;
  }
```

运行情况如下：

```
input the number of diskes:3 ↙
The steps to moving 3 diskes:
A-->C
A-->B
C-->B
A-->C
B-->A
B-->C
A-->C
```

在本程序中 move 函数并未真正移动盘子，而只是输出移盘的方案(从哪一个座移到哪一个座)。

可以看到，将 3 个盘子从 A 座移到 C 座需要移 7 次，如果将 64 个盘子从 A 座移到 C 座需要移$(2^{64}-1)$次，假设和尚每次移动 1 个盘子用 1 秒钟，则移动$(2^{64}-1)$次需要$(2^{64}-1)$秒，大约相当于6×10^{11}年，即大约 600 亿年，所以有人戏称，当老和尚移完 64 个盘子之时，“世界末日”也到了。

由于篇幅关系，不再对上述程序做过多解释，请读者仔细分析理解。

8.7 数组作为函数参数

前面已经介绍了可以用变量作函数参数，显然，数组元素也可以作函数实参，其用法与变量相同。此外，数组名也可以作实参和形参，传递的是数组首元素的地址。

8.7.1 数组元素作函数实参

由于实参可以是表达式，而数组元素可以是表达式的组成部分，因此数组元素当然可以作为函数的实参，与用变量作实参一样，是单向传递，即“值传送”方式。

例 8.10 有两个数组 a 和 b，各有 10 个元素，将它们对应地逐个比较（即 a[0]与 b[0]比，a[1]与 b[1]比……）。如果 a 数组中的元素大于 b 数组中的相应元素的数目多于 b 数组中元素大于 a 数组中相应元素的数目（例如，a[i]>b[i]6 次，b[i]>a[i]3 次，其中 i 每次为不同的值），则认为 a 数组大于 b 数组，并分别统计出两个数组相应元素大于、等于、小于的次数。

程序如下：

```
#include <stdio.h>
void main()
{
    int large(int x,int y);                          /* 函数声明 */
    int a[10],b[10],i,n=0,m=0,k=0;
    printf("enter array a: \n");
    for(i=0;i<10;i++)
        scanf("%d",&a[i]);
    printf("\n");
    printf("enter array b: \n");
    for(i=0;i<10;i++)
        scanf("%d",&b[i]);
    printf("\n");
    for(i=0;i<10;i++)
      {
        if(large(a[i],b[i])==1)
            n=n+1;
        else
          if(large(a[i],b[i])==0)
            m=m+1;
          else
            k=k+1;
      }
    printf ("a[i]>b[i] %d times\na[i]=b[i] %d times\na[i]<b[i] %d times\n", n,
        m, k);
    if(n>k)
        printf("array a is larger than array b\n");
    else
        if (n<k) printf("array a is smaller than array b\n");
    else
        printf("array a is equal to array b\n");
}

large(int x,int y)
```

```
{
  int flag;
  if(x>y)
    flag=1;
  else if(x<y)
    flag=-1;
  else flag=0;
  return(flag);
}
```

运行情况如下：

```
enter array a:
1 3 5 7 9 8 6 4 2 0↙
enter array b:
5 3 8 9-1-3 5 6 0 4↙
a[i]>b[i] 4 times
a[i]=b[i] 1 times
a[i]<b[i] 5 times
array a is smaller than array b
```

8.7.2 数组名作函数参数

可以用数组名作函数参数，此时形参应当用数组名或用指针变量(见第 10 章)。

例 8.11 有一个一维数组 score，内放 10 个学生成绩，求平均成绩。

程序如下：

```
#include <stdio.h>
void main()
{
  float average(float array[10]);                    /* 函数声明 */
  float score[10],aver;
  int i;
  printf("input 10 scores:\n");
  for(i=0;i<10;i++)
      scanf("%f",&score[i]);
  printf("\n");
  aver=average(score);
  printf("average score is %5.2f\n",aver);
}

float average(float array[10])
{
  int i;
  float aver,sum=array[0];
  for(i=1;i<10;i++)
```

```
    sum=sum+array[i];
  aver=sum/10;
  return(aver);
}
```

运行情况如下：

```
input 10 scores:
100 56 78 98.5 76 87 99 67.5 75 97↙
average score is 83.40
```

说明：

(1) 用数组名作函数参数，应该在主调函数和被调用函数分别定义数组，例中 array 是形参数组名，score 是实参数组名，分别在其所在函数中定义，不能只在一方定义。

(2) 实参数组与形参数组类型应一致(今都为 float 型)，如不一致，结果将出错。

(3) 在被调用函数中声明了形参数组的大小为 10，但在实际上，指定其大小是不起任何作用的，因为 C 语言编译对形参数组大小不做检查，只是将实参数组的首元素的地址传给形参数组。因此，形参数组名获得了实参数组的首元素的地址，前已说明，数组名代表数组的首元素的地址，因此，形参数组首元素(array[0])和实参数组首元素(score[0])具有同一地址，它们共占同一存储单元，score[n]和 array[n]指的是同一单元。score[n]和 array[n]具有相同的值。

(4) 形参数组可以不指定大小，在定义数组时在数组名后面跟一个空的方括号。有时为了在被调用函数中处理数组元素的需要，可以另设一个形参，传递需要处理的数组元素的个数，例 8.11 可以改写为例 8.12 的形式。

例 8.12 形参数组不定义长度。

```
#include <stdio.h>
void main()
{
  float average(float array[ ],int n);
  float score_1[5]={98.5,97,91.5,60,55};
  float score_2[10]={67.5,89.5,99,69.5,77,89.5,76.5,54,60,99.5};
  printf("the average of class A is %6.2f\n",average(score_1,5));
  printf("the average of class B is %6.2f\n",average(score_2,10));
}

float average(float array[ ],int n)
{
  int i;
  float aver,sum=array[0];
  for(i=1;i<n;i++)
    sum=sum+array[i];
  aver=sum/n;
```

```
    return(aver);
}
```

程序的作用是分别求出数组 score_1(有 5 个元素) 和数组 score_2(有 10 个元素)各元素的平均值。两次调用 average 函数时需要处理的数组元素个数是不同的,在第一次调用时将实参(值为 5)传递给形参 n,表示求 5 个学生的平均分。第二次调用时,求 10 个学生的平均分。

运行结果如下:

```
the average of class A is 80.40
the average of class B is 78.20
```

(5) 最后应当强调说明一点:用数组名作函数实参时,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,见图 8-14。假若 a 的首元素的地址为 1000,则 b 数组首元素的地址也是 1000,显然,a[0]与 b[0]同占一个单元……假如改变了 b[0] 的值,也就意味着 a[0] 的值也改变了。也就是说,形参数组中各元素的值如发生变化会使实参数组元素的值同时发生变化,从图 8-14 看是很容易理解的。这一点是与变量作函数参数的情况不相同,务请注意。在程序设计中可以有意识地利用这一特点改变实参数组元素的值(如排序)。

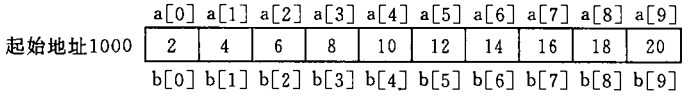


图 8-14

关于数组名作为函数参数,将在第 10 章介绍完指针变量后作进一步的说明。

例 8.13 用选择法对数组中 10 个整数按由小到大排序。所谓选择法就是先将 10 个数中最小的数与 a[0]对换;再将 a[1]到 a[9]中最小的数与 a[1]对换……每比较一轮,找出一个未经排序的数中最小的一个。共比较 9 轮。

下面以 5 个数为例说明选择法的步骤。

a[0]	a[1]	a[2]	a[3]	a[4]	
3	6	1	9	4	未排序时的情况
1	6	3	9	4	将 5 个数中最小的数 1 与 a[0]对换
1	3	6	9	4	将余下的 4 个数中最小的数 3 与 a[1]对换
1	3	4	9	6	将余下的 3 个数中最小的数 4 与 a[2]对换
1	3	4	6	9	将余下的 2 个数中最小的数 6 与 a[3]对换,至此完成排序

根据此思路编写程序如下:

```
#include <stdio.h>
void main()
{
  void sort(int array[],int n);
```

```
    int a[10],i;
    printf("enter the array\n");
    for(i=0;i<10;i++)
        scanf("%d",&a[i]);
    sort(a,10);
    printf("the sorted array: \n");
    for(i=0;i<10;i++)
        printf("%5d",a[i]);
    printf("\n");
}

void sort(int array[],int n)
  {
    int i,j,k,t;
    for(i=0;i<n-1;i++)
        {
          k=i;
          for(j=i+1;j<n;j++)
              if(array[j]<array[k])
                  k=j;
          t=array[k];array[k]=array[i];array[i]=t;
        }
  }
```

可以看到在执行函数调用语句“sort(a,10);”之前和之后，a 数组中各元素的值是不同的。原来是无序的，执行“sort(a,10);”后，a 数组已经排好序了，这是由于形参数组 array 已用选择法进行排序了，形参数组改变也使实参数组随之改变。

请读者自己画出调用 sort 函数前后实参数组中各元素的值。

8.7.3 多维数组名作函数参数

多维数组元素可以作函数参数，这点与前述的情况类似。

可以用多维数组名作为函数的实参和形参，在被调用函数中对形参数组定义时可以指定每一维的大小，也可以省略第一维的大小说明。例如：

```
int array[3][10];
```

或

```
int array[ ][10];
```

二者都合法而且等价。但是不能把第二维以及其他高维的大小说明省略。如下面的定义是不合法的：

```
int array[ ][ ];
```

前已说明，二维数组是由若干个一维数组组成的，在内存中，数组是按行存放的，因此，在定义二维数组时，必须指定列数（即一行中包含几个元素），由于形参数组与实参数组类型相同，所以它们是由具有相同长度的一维数组所组成的。不能只指定第一维（行

数)而省略第二维(列数),下面的写法是错误的:

```
int array[3][];
```

在第二维大小相同的前提下,形参数组的第一维可以与实参数组不同。例如,实参数组定义为

```
int score[5][10];
```

而形参数组定义为

```
int array[3][10];
```

或

```
int array[8][10];
```

均可以。这时形参数组和实参数组都是由相同类型和大小的一维数组组成的。C 语言编译系统不检查第一维的大小。在学习指针以后,对此会有更深入的认识。

例 8.14 有一个 3×4 的矩阵,求所有元素中的最大值。

解此题的算法是:先使变量 max 的初值为矩阵中第一个元素的值,然后将矩阵中各个元素的值与 max 相比,每次比较后都把"大者"存放在 max 中,全部元素比较完后,max 的值就是所有元素的最大值。

程序如下:

```
#include <stdio.h>
void main()
  {
    int max_value(int array[][4]);
    int a[3][4]={{1,3,5,7},{2,4,6,8},{15,17,34,12}};
    printf("max value is %d\n",max_value(a));
  }

int max_value(int array[][4])
  {
    int i,j,max;
    max=array[0][0];
    for(i=0;i<3;i++)
      for(j=0;j<4;j++)
        if(array[i][j]>max)
          max=array[i][j];
    return (max);
  }
```

运行结果如下:

```
max value is 34
```

8.8 局部变量和全局变量

8.8.1 局部变量

在一个函数内部定义的变量是内部变量，它只在本函数范围内有效，也就是说只有在本函数内才能使用它们，在此函数以外是不能使用这些变量的。这称为“局部变量”。例如：

```
float f1( int a)                    /* 函数 f1 */
{int b,c;                           }
 ⋮                                  } a、b、c 有效
}                                   }
char f2(int x,int y)                /* 函数 f2 */
{int i,j;                           }
}                                   } x、y、i、j 有效
void main( )                        /* 主函数 */
{int m,n;                           }
 ⋮                                  } m、n 有效
}                                   }
```

说明：

(1) 主函数中定义的变量(m,n)也只在主函数中有效，而不因为在主函数中定义而在整个文件或程序中有效。主函数也不能使用其他函数中定义的变量。

(2) 不同函数中可以使用相同名字的变量，它们代表不同的对象，互不干扰。例如，上面在 f1 函数中定义了变量 b 和 c，倘若在 f2 函数中也定义变量 b 和 c，它们在内存中占不同的单元，互不混淆。

(3) 形式参数也是局部变量。例如上面 f1 函数中的形参 a，也只在 f1 函数中有效。其他函数可以调用 f1 函数，但不能引用 f1 函数的形参 a。

(4) 在一个函数内部，可以在复合语句中定义变量，这些变量只在本复合语句中有效，这种复合语句也称为“分程序”或“程序块”。

```
void main ( )
{                                                    }
  int a,b;                                           }
    ⋮                                                }
    {                                                }
      int c;                                         } a、b 在此范围内有效
      c=a+b;  } c 在此范围内有效                     }
       ⋮      }                                      }
    }                                                }
    ⋮                                                }
}                                                    }
```

变量 c 只在复合语句(分程序)内有效,离开该复合语句该变量就无效,释放内存单元。

8.8.2 全局变量

前已介绍,程序的编译单位是源程序文件,一个源文件可以包含一个或若干个函数。在函数内定义的变量是局部变量,而在函数之外定义的变量称为外部变量,外部变量是全局变量(也称全程变量)。全局变量可以为本文件中其他函数所共用。它的有效范围为从定义变量的位置开始到本源文件结束。例如:

```
int p=1,q=5;                    /* 外部变量 */
float f1(int a)                 /* 定义函数 f1 */
  {
    int b,c;
    ⋮
  }
char c1,c2;                  /* 外部变量 */
char f2 (int x, int y)       /* 定义函数 f2 */
  {
    int i,j;
    ⋮
  }
void main ( ) /* 主函数 */
  {
    int m,n;
    ⋮
  }
```

全局变量 p、q 的作用范围

全局变量 c1、c2 的作用范围

p、q、c1、c2 都是全局变量,但它们的作用范围不同,在 main 函数和 f2 函数中可以使用全局变量 p、q、c1、c2,但在函数 f1 中只能使用全局变量 p、q,而不能使用 c1 和 c2。

在一个函数中既可以使用本函数中的局部变量,又可以使用有效的全局变量。打个通俗的比方:国家有统一的法律和法规,各省还可以根据需要制定地方的法律和法规。在甲省,国家统一的法律法规和甲省的法律法规都是有效的,而在乙省,则国家统一的和乙省的法律法规有效。显然,甲省的法律法规在乙省无效。

说明:

(1) 设置全局变量的作用是增加了函数间数据联系的渠道。由于同一文件中的所有函数都能引用全局变量的值,因此如果在一个函数中改变了全局变量的值,就能影响到其他函数,相当于各个函数间有直接的传递通道。由于函数的调用只能带回一个返回值,因此有时可以利用全局变量增加函数间的联系渠道,通过函数调用能得到一个以上的值。

为了便于区别全局变量和局部变量,在 C 程序设计人员中有一个不成文的约定(但非规定),将全局变量名的第一个字母用大写表示。

例 8.15 有一个一维数组,内放 10 个学生成绩,写一个函数,求出平均分、最高分和最低分。

显然希望通过函数调用得到3个结果值，除了可以从函数得到一个函数返回值外，还可以利用全局变量。

```
#include <stdio.h>
float Max=0,Min=0;                    /*全局变量*/
void main()
  {
    float average(float array[ ],int n);
    float ave,score[10];
    int i;
    for(i=0;i<10;i++)
        scanf("%f",&score[i]);
    ave=average(score,10);
    printf("max=%6.2f\nmin=%6.2f\naverage=%6.2f\n",Max,Min,ave);
  }

float average(float array[ ],int n)              /* 定义函数,形参为数组 */
  {
    int i;
    float aver,sum=array[0];
    Max=Min=array[0];
    for(i=1;i<n;i++)
      {
        if(array[i]>Max) Max=array[i];
        else if(array[i]<Min) Min=array[i];
        sum=sum+array[i];
      }
    aver=sum/n;
    return(aver);
  }
```

运行情况如下：

```
99 45 78 97 100 67.5 89 92 66 43↙
max=100.00
min=43.00
average=77.65
```

函数average中和外界有联系的变量与外界的联系如图8-15所示。可以看出形参array和n的值由main函数传递给形参，函数average中aver的值通过return语句带回main函数。Max和Min是全局变量，是公用的，它们的值可以供各函数使用，如果在一个函数中，改变了它们的值，在其他函数中也可以使用这个已改变的值。

由此看出，可以利用全局变量以减少函数实参与形参的个数，从而减少内存空间以及传递数据时的时间消耗。

(2) 建议不在必要时不要使用全局变量，原因如下。

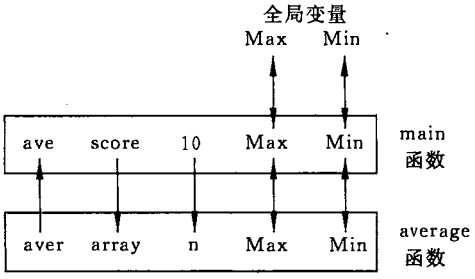


图 8-15

① 全局变量在程序的全部执行过程中都占用存储单元，而不是仅在需要时才开辟单元。

② 它使函数的通用性降低了，因为函数在执行时要依赖于其所在的外部变量。如果将一个函数移到另一个文件中，还要将有关的外部变量及其值一起移过去。但若该外部变量与其他文件的变量同名时，就会出现问题，降低了程序的可靠性和通用性。在程序设计中，在划分模块时要求模块的“内聚性”强、与其他模块的“耦合性”弱。即模块的功能要单一(不要把许多互不相干的功能放到一个模块中)，与其他模块的相互影响要尽量少，而用全局变量是不符合这个原则的。一般要求把 C 程序中的函数做成一个封闭体，除了可以通过“实参—形参”的渠道与外界发生联系外，没有其他渠道。这样的程序移植性好，可读性强。

③ 使用全局变量过多，会降低程序的清晰性，人们往往难以清楚地判断出每个瞬时各个外部变量的值。在各个函数执行时都可能改变外部变量的值，程序容易出错。因此，要限制使用全局变量。

(3) 如果在同一个源文件中，外部变量与局部变量同名，则在局部变量的作用范围内，外部变量被“屏蔽”，即它不起作用。

例 8.16 外部变量与局部变量同名。

```
#include <stdio.h>
int a=3,b=5;                    /* a、b 为外部变量 */
void main()
  {
    int max(int a,int b);       /* 本行是函数声明，a 和 b 为形参名 */
    int a=8;                    /* a 为局部变量 */        局部变量 a 作用范围
    printf("%d\n",max(a,b));                              全局变量 b 的作用范围
  }

int max(int a,int b)            /* a、b 为形参局部变量 */
  {                                                       形参 a、b 作用范围
    int c;
    c=a>b? a:b;
    return(c);
  }
```

运行结果为

8

在此，故意重复使用 a、b 作变量名，请读者区别不同的 a、b 的含义和作用范围。程序第 2 行定义了外部变量 a 和 b，并使之初始化。第 3 行是 main 函数，在 main 函数中定义了一个局部变量 a，因此全局变量 a 在 main 函数范围内不起作用，而全局变量 b 在此范围内有效。因此 printf 函数中的 max(a,b)相当于 max(8,5)。第 8 行开始定义函数 max，a 和 b 是形参，形参也是局部变量。函数 max 中的 a、b 不是外部变量 a、b，它们的值是由实参传给形参的，外部变量 a、b 在 max 函数范围内不起作用。

8.9 变量的存储类别

8.9.1 动态存储方式与静态存储方式

上一节已介绍了，从变量的作用域(即从空间)角度来分，可以分为全局变量和局部变量。

可以从另一个角度，从变量值存在的时间(即生存期)角度来分，可以分为**静态存储方式和动态存储方式**。

所谓静态存储方式是指在程序运行期间由系统分配固定的存储空间的方式。而动态存储方式则是在程序运行期间根据需要进行动态的分配存储空间的方式。

先看一下内存中的供用户使用的存储空间的情况。这个存储空间可以分为三部分，见图 8-16。

(1) 程序区；

(2) 静态存储区；

(3) 动态存储区。

图 8-16

数据分别存放在静态存储区和动态存储区中。全局变量全部存放在静态存储区中，在程序开始执行时给全局变量分配存储区，程序执行完毕就释放。在程序执行过程中它们占据固定的存储单元，而不是动态地进行分配和释放。

在动态存储区中存放以下数据：

① 函数形式参数。在调用函数时给形参分配存储空间。

② 自动变量(未加 static 声明的局部变量，详见后面的介绍)。

③ 函数调用时的现场保护和返回地址等。

对以上这些数据，在函数调用开始时分配动态存储空间，函数结束时释放这些空间。在程序执行过程中，这种分配和释放是动态的，如果在一个程序中两次调用同一函数，分配给此函数中局部变量的存储空间地址可能是不相同的。如果一个程序包含若干个函数，每个函数中的局部变量的生存期并不等于整个程序的执行周期，它只是程序执行周期的一部分。根据函数调用的需要，动态地分配和释放存储空间。

在 C 语言中，每一个变量和函数有两个属性：数据类型和数据的存储类别。对数据

类型，读者已熟悉(如整型、字符型等)。存储类别指的是数据在内存中存储的方式。存储方式分为两大类：静态存储类和动态存储类。具体包含 4 种：自动的(auto)、静态的(static)、寄存器的(register)、外部的(extern)。根据变量的存储类别，可以知道变量的作用域和生存期。下面分别作介绍。

8.9.2 auto 变量

函数中的局部变量，如果不专门声明为 static 存储类别，都是动态地分配存储空间的，数据存储在动态存储区中。函数中的形参和在函数中定义的变量(包括在复合语句中定义的变量)，都属此类，在调用该函数时系统会给它们分配存储空间，在函数调用结束时就自动释放这些存储空间。因此这类局部变量称为自动变量。自动变量用关键字 auto 作存储类别的声明。例如：

```
int f(int a)                    /*定义 f 函数,a 为形参 */
{
  auto int b,c=3;               /*定义 b、c 为自动变量 */
  ⋮
}
```

其中，a 是形参，b、c 是自动变量，对 c 赋初值 3。执行完 f 函数后，自动释放 a、b、c 所占的存储单元。

实际上，关键字“auto”可以省略，auto 不写则隐含确定为“自动存储类别”，它属于动态存储方式。程序中大多数变量属于自动变量。前面介绍的函数中定义的变量都没有声明为 auto，其实都隐含指定为自动变量。例如，在函数体中：

```
auto int b,c=3;
```

与

```
int b,c=3;
```

二者等价。

8.9.3 用 static 声明局部变量

有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值，即其占用的存储单元不释放，在下一次该函数调用时，该变量已有值，就是上一次函数调用结束时的值。这时就应该指定该局部变量为“静态局部变量”，用关键字 static 进行声明。通过下面简单的例子可以了解它的特点。

例 8.17 考察静态局部变量的值。

```
#include <stdio.h>
void main()
  {
    int f(int);
    int a=2,i;
```

```
    for(i=0;i<3;i++)
        printf("%d ",f(a));
  }
int f(int a)
  {
    auto int b=0;
    static int c=3;
    b=b+1;
    c=c+1;
    return(a+b+c);
  }
```

运行结果为：

7 8 9

在第 1 次调用 f 函数时，b 的初值为 0，c 的初值为 3，第 1 次调用结束时，b=1，c=4，a+b+c=7。由于 c 是静态局部变量，在函数调用结束后，它并不释放，仍保留 c=4。在第 2 次调用 f 函数时，b 的初值为 0，而 c 的初值为 4（上次调用结束时的值），见图 8-17。先后 3 次调用 f 函数时，b 和 c 的值见表 8-1 所示。

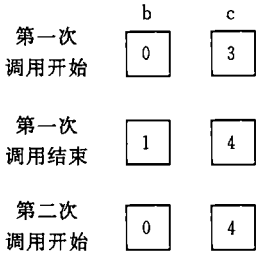


图 8-17

表 8-1

第几次调用	调用时初值		调用结束时的值		
	b	c	b	c	a+b+c
第 1 次	0	3	1	4	7
第 2 次	0	4	1	5	8
第 3 次	0	5	1	6	9

对静态局部变量的说明：

(1) 静态局部变量属于静态存储类别，在静态存储区内分配存储单元。在程序整个运行期间都不释放。而自动变量（即动态局部变量）属于动态存储类别，占动态存储区空间而不占静态存储区空间，函数调用结束后即释放。

(2) 对静态局部变量是在编译时赋初值的，即只赋初值一次，在程序运行时它已有初值。以后每次调用函数时不再重新赋初值而只是保留上次函数调用结束时的值。而对自

动变量赋初值，不是在编译时进行的，而是在函数调用时进行，每调用一次函数重新给一次初值，相当于执行一次赋值语句。

(3) 如在定义局部变量时不赋初值的话，则对静态局部变量来说，编译时自动赋初值0(对数值型变量)或空字符(对字符变量)。而对自动变量来说，如果不赋初值则它的值是一个不确定的值。这是由于每次函数调用结束后存储单元已释放，下次调用时又重新另分配存储单元，而所分配的单元中的值是不确定的。

(4) 虽然静态局部变量在函数调用结束后仍然存在，但其他函数是不能引用它的。

需要用局部静态变量的情况如下。

(1) 需要保留函数上一次调用结束时的值。例如可以用下面方法求 $n!$。

例 8.18 输出 1 到 5 的阶乘值。

```
#include <stdio.h>
void main()
  {
    int fac(int n);
    int i;
    for(i=1;i<=5;i++)
        printf("%d!=%d\n",i,fac(i));
  }

int fac(int n)
  {
    static int f=1;
    f=f*n;
    return(f);
  }
```

运行结果为：

```
1!=1
2!=2
3!=6
4!=24
5!=120
```

每次调用 fac(i)，输出一个 $i!$，同时保留这个 $i!$ 的值以便下次再乘($i+1$)。

(2) 如果初始化后，变量只被引用而不改变其值，则这时用静态局部变量比较方便，以免每次调用时重新赋值。

但是应该看到，用静态存储要多占内存(长期占用不释放，而不能像动态存储那样一个存储单元可供多个变量使用，节约内存)，而且降低了程序的可读性，当调用次数多时往往弄不清静态局部变量的当前值是什么。因此，若非必要，不要多用静态局部变量。

8.9.4 register 变量

一般情况下，变量(包括静态存储方式和动态存储方式)的值是存放在内存中的。当

程序中用到哪一个变量的值时，由控制器发出指令将内存中该变量的值送到运算器中。经过运算器进行运算，如果需要存数，再从运算器将数据送到内存存放，见图 8-18。

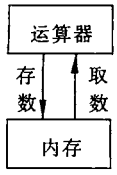


图 8-18

如果有一些变量使用频繁(例如，在一个函数中执行 10000 次循环，每次循环中都要引用某局部变量)，则为存取变量的值要花费不少时间。为提高执行效率，C 语言允许将局部变量的值放在 CPU 中的寄存器中，需要用时直接从寄存器取出参加运算，不必再到内存中去存取。由于对寄存器的存取速度远高于对内存的存取速度，因此这样做可以提高执行效率。这种变量叫做寄存器变量，用关键字 register 作声明。例如，例 8.19 中的程序是输出 1～n 阶乘的值。

例 8.19 使用寄存器变量。

```
#include <stdio.h>
void main()
{
  long fac(long);
  long i,n;
  scanf("%ld",&n);
  for(i=1;i<=n;i++)
    printf("%ld!=%ld\n",i,fac(i));
}

long fac(long n)
{
  register long i,f=1;          /*定义寄存器变量*/
  for (i=1;i<=n;i++)
    f=f*i;
  return (f);
}
```

运行时输入 n 的值，就能输出 1! 到 n! 的值。在 fac 函数中定义的局部变量 f 和 i 是寄存器变量，如果 n 的值大，则能节约许多执行时间。

说明：

(1) 只有局部自动变量和形式参数可以作为寄存器变量，其他(如全局变量)不行。在调用一个函数时占用一些寄存器以存放寄存器变量的值，函数调用结束释放寄存器。此后，在调用另一个函数时又可以利用它来存放该函数的寄存器变量。

(2) 一个计算机系统中的寄存器数目是有限的，不能定义任意多个寄存器变量。不同的系统允许使用的寄存器个数是不同的，而且对 register 变量的处理方法也是不同的，有的系统对 register 变量当作自动变量处理，分配内存单元，并不真正把它们存放在寄存器中，有的系统只允许将 int、char 和指针型变量定义为寄存器变量。

(3) 局部静态变量不能定义为寄存器变量。不能写成

```
rcgister static int a,b,c;
```

不能把变量 a、b、c 既放在静态存储区中，又放在寄存器中，二者只能居其一。对一个变量只能声明为一种存储类别。

当今的优化编译系统能够识别使用频繁的变量，从而自动地将这些变量放在寄存器中，而不需要程序设计者指定。因此，实际上用 register 声明变量是不必要的。读者对它有一定了解即可，以便在阅读他人写的程序时遇到 register 不致感到茫然。

8.9.5 用 extern 声明外部变量

外部变量是在函数的外部定义的全局变量，它的作用域是从变量的定义处开始，到本程序文件的末尾。在此作用域内，全局变量可以为程序中各个函数所引用。编译时将外部变量分配在静态存储区。

有时需要用 extern 来声明外部变量，以扩展外部变量的作用域。

1. 在一个文件内声明外部变量

如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件结束。如果在定义点之前的函数想引用该外部变量，则应该在引用之前用关键字 extern 对该变量作“外部变量声明”，表示该变量是一个已经定义的外部变量。有了此声明，就可以从“声明”处起，合法地使用该外部变量。例如：

例 8.20 用 extern 声明外部变量，扩展它在程序文件中的作用域。

```
#include <stdio.h>
void main()
{
  int max(int,int);
  extern A,B;                    /*外部变量声明*/
  printf("%d\n",max(A,B));
}

int A=13,B=-8;                   /*定义外部变量*/
int max(int x,int y)             /*定义 max 函数 */
{
  int z;
  z=x>y? x:y;
  return(z);
}
```

运行结果如下：

```
13
```

定义外部变量 A 和 B 的位置在函数 main 之后，因此在 main 函数中不能引用外部变量 A 和 B。在 main 函数的第 3 行用 extern 对 A 和 B 进行“外部变量声明”，表示 A 和 B 是已经定义的外部变量(但定义的位置在后面)。这样在 main 函数中就可以合法地使用

全局变量 A 和 B 了。如果不作 extern 声明，编译时出错，系统不会认为 A、B 是已定义的外部变量。一般做法是外部变量的定义放在引用它的所有函数之前，这样可以避免在函数中多加一个 extern 声明。

用 extern 声明外部变量时，类型名可以写也可以省写。例如，“extern int A,B;”也可以写成：“extern A,B;”。

2. 在多文件的程序中声明外部变量

一个 C 程序可以由一个或多个源程序文件组成。如果程序只由一个源文件组成，使用外部变量的方法前面已经介绍。如果程序由多个源程序文件组成，那么在一个文件中想引用另一个文件中已定义的外部变量，有什么办法呢？

如果一个程序包含两个文件，在两个文件中都要用到同一个外部变量 Num，不能分别在两个文件中各自定义一个外部变量 Num，否则在进行程序的连接时会出现“重复定义”的错误。正确的做法是：在任一个文件中定义外部变量 Num，而在另一文件中用 extern 对 Num 作“外部变量声明”。即“extern Num; ”。在编译和连接时，系统会由此知道 Num 是一个已在别处定义的外部变量，并将在另一文件中定义的外部变量的作用域扩展到本文件，在本文件中可以合法地引用外部变量 Num。

下面举一个简单的例子来说明这种引用。

例 8.21 用 extern 将外部变量的作用域扩展到其他文件。

本程序的作用是给定 b 的值，输入 a 和 m，求 a * b 和 a^m 的值。

文件 file1. c 中的内容为：

```
#include <stdio.h>
int A;                                  /*定义外部变量*/
void main()
{
  int power(int);                       /*函数声明*/
  int b=3,c,d,m;
  printf("enter the number a and its power m:\n");
  scanf("%d,%d",&A,&m);
  c=A*b;
  printf("%d*%d=%d\n",A,b,c);
  d=power(m);
  printf("%d**%d=%d\n",A,m,d);
}
```

文件 file2. c 中的内容为：

```
extern A;                       /*声明 A 为一个已定义的外部变量*/
int power(int n)
{
  int i,y=1;
  for(i=1;i<=n;i++)
```

```
    y*=A;
  return(y);
}
```

可以看到，file2.c 文件中的开头有一个 extern 声明，它声明在本文件中出现的变量 A 是一个已经在其他文件中定义过的外部变量，本文件不必再为它分配内存。本来外部变量 A 的作用域是 file1.c，但现在用 extern 声明将其作用域扩大到 file2.c 文件。假如程序有 5 个源文件，在一个文件中定义外部整型变量 A，其他 4 个文件都可以引用 A，但必须在每一个文件中都加上一个“extern A;”声明。在各文件经过编译后，将各目标文件连接成一个可执行的目标文件。

但是用这样的全局变量应十分慎重，因为在执行一个文件中的函数时，可能会改变了该全局变量的值，从而会影响到另一文件中的函数执行结果。

有的读者可能会问：extern 既可以用来扩展外部变量在本文件中的作用域，又可以使外部变量的作用域从一个文件扩展到程序中的其他文件，那么系统怎么区别处理呢？实际上，在编译时遇到 extern 时，先在本文件中找外部变量的定义，如果找到，就在本文件中扩展作用域；如果找不到，就在连接时从其他文件中找外部变量的定义。如果从其他文件中找到了，就将作用域扩展到本文件；如果再找不到，就按出错处理。

8.9.6 用 static 声明外部变量

有时在程序设计中希望某些外部变量只限于被本文件引用，而不能被其他文件引用。这时可以在定义外部变量时加一个 static 声明。

例如：

file1.c

```
static int A;
void main ( )
{
    ⋮
}
```

file2.c

```
extern int A;
void fun (int n)
{
    ⋮
  A=A*n;
    ⋮
}
```

在 file1.c 中定义了一个全局变量 A，但它用 static 声明，因此只能用于本文件，虽然在 file2.c 文件中用了“extern int A;”，但 file2.c 文件中无法使用 file1.c 中的全局变量 A。

这种加上 static 声明、只能用于本文件的外部变量称为静态外部变量。在程序设计中，常由若干人分别完成各个模块，各人可以独立地在其设计的文件中使用相同的外部变量名而互不相干。只需在每个文件中的外部变量前加上 static 即可。这就为程序的模块化、通用性提供方便。如果其他文件不需要引用本文件的外部变量，可以对本文件中的外部变量都加上 static，成为静态外部变量，以免被其他文件误用。

需要指出：不要误认为对外部变量加 static 声明后才是静态存储方式（存放在静态

存储区中),而不加 static 的是动态存储(存放在动态存储区)。两种形式的外部变量都是静态存储方式(存放在静态存储区),只是作用范围不同而已,都是在编译时分配内存的。

8.9.7 关于变量的声明和定义

在第 3 章中介绍了如何定义一个变量。在本章中又介绍了如何对一个变量的存储类别作声明。可能有些读者弄不清楚定义与声明有什么区别,它们是否一回事。在 C 语言的学习中,关于定义与声明这两个名词的使用上始终存在着混淆。不仅许多初学者没有搞清楚,连不少介绍 C 语言的教材和书刊也没有给出准确的介绍。

从第 3 章已经知道,一个函数一般由两部分组成:声明部分和执行语句。声明部分的作用是对有关的标识符(如变量、函数、结构体、共用体等)的属性进行说明。对于函数,声明和定义的区别是明显的,在本章 8.4.3 小节中已说明,函数的声明是函数的原型,而函数的定义是函数的本身。对被调用函数的声明是放在主调函数的声明部分中的,而函数的定义显然不在声明部分的范围内,它是一个独立的模块。

对变量而言,声明与定义的关系稍微复杂一些。在声明部分出现的变量有两种情况:一种是需要建立存储空间的(如:int a;),另一种是不需要建立存储空间的(如:extern a;)。前者称为"定义性声明"(defining declaration),或简称定义(definition)。后者称为"引用性声明"(referencing declaration)。广义地说,声明包括定义,但并非所有的声明都是定义。对"int a;"而言,它既是声明,又是定义。而对"extern a;"而言,它是声明而不是定义。一般为了叙述方便,把建立存储空间的声明称定义,而把不需要建立存储空间的声明称为声明。显然这里指的声明是狭义的,即非定义性声明。例如:

```
void main()
  {
    extern A;              /* 是声明,不是定义。声明 A 是一个已定义的外部变量 */
    ⋮
  }
int A;                     /* 是定义,定义 A 为整型外部变量 */
```

外部变量定义和外部变量声明的含义是不同的。外部变量的定义只能有一次,它的位置在所有函数之外,而同一文件中的外部变量的声明可以有多次,它的位置可以在函数之内(哪个函数要用就在哪个函数中声明),也可以在函数之外(在外部变量的定义点之前)。系统根据外部变量的定义(而不是根据外部变量的声明)分配存储单元。对外部变量的初始化只能在"定义"时进行,而不能在"声明"中进行。所谓"声明",其作用是声明该变量是一个已在后面(或在其他文件中)已定义的外部变量,仅仅是为了扩展该变量的作用范围而作的"声明"。extern 只用作声明,而不用于定义。

用 static 来声明一个变量的作用有二:

(1) 对局部变量用 static 声明,则使该变量在整个程序执行期间不释放,为其分配的空间始终存在。

(2) 全局变量用 static 声明,则该变量的作用域只限于本文件模块(即被声明的文件中)。

注意：用 auto、register、static 声明变量时，是在定义变量的基础上加上这些关键字，而不能单独使用。下面用法不对：

```
int a;                    /* 先定义整型变量 a */
static a;                 /* 再对变量 a 声明为静态变量 */
```

编译时会被认为“重新定义”。

8.9.8 存储类别小结

从上可知，对一个数据的定义，需要指定两种属性：数据类型和存储类别，分别使用两个关键字。例如：

```
static int a;             /* 静态内部整型变量或静态外部整型变量 */
auto char c;              /* 自动变量，在函数内定义 */
register int d;           /* 寄存器变量，在函数内定义 */
```

此外，可以用 extern 声明变量为已定义的外部变量，例如：

```
extern b;                 /* 声明 b 是一个已被定义的外部变量 */
```

下面从不同角度做些归纳：

(1) 从作用域角度分，有局部变量和全局变量。它们采用的存储类别如下：

- 按作用域角度分
  - 局部变量
    - 自动变量，即动态局部变量(离开函数，值就消失)
    - 静态局部变量(离开函数，值仍保留)
    - 寄存器变量(离开函数，值就消失)
    - (形式参数可以定义为自动变量或寄存器变量)
  - 全局变量
    - 静态外部变量(只限本文件引用)
    - 外部变量(即非静态的外部变量，允许其他文件引用)

(2) 从变量存在的时间(生存期)来区分，有动态存储和静态存储两种类型。静态存储是程序整个运行时间都存在，而动态存储则是在调用函数时临时分配单元。

- 按变量存在的时间分
  - 动态存储
    - 自动变量(本函数内有效)
    - 寄存器变量(本函数内有效)
    - 形式参数(本函数内有效)
  - 静态存储
    - 静态局部变量(函数内有效)
    - 静态外部变量(本文件内有效)
    - 外部变量(其他文件可引用)

(3) 从变量值存放的位置来区分，可分为：

- 按变量值存放的位置分
  - 内存中静态存储区
    - 静态局部变量
    - 静态外部变量(函数外部静态变量)
    - 外部变量(可为其他文件引用)
  - 内存中动态存储区：自动变量和形式参数
  - CPU 中的寄存器：寄存器变量

(4) 关于作用域和生存期的概念。从前面叙述可以知道，对一个变量的性质可以从两个方面分析，一是变量的作用域，一是变量值存在时间的长短，即生存期。前者是从空间的角度，后者是从时间的角度。二者有联系但不是同一回事。图 8-19 是作用域的示意图，图 8-20 是生存期的示意图。

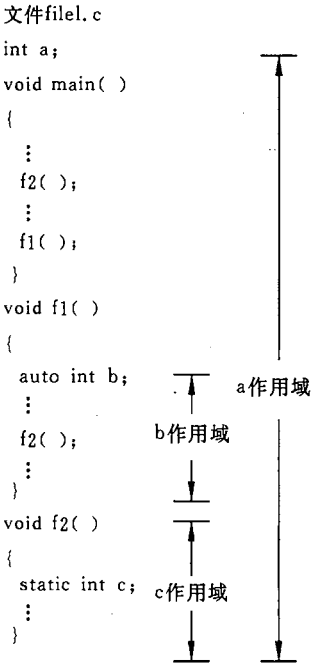


图 8-19

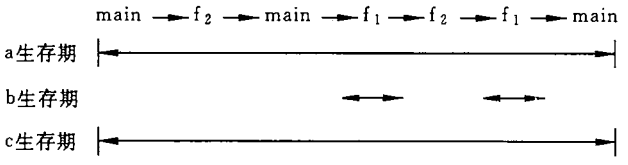


图 8-20

如果一个变量在某个文件或函数范围内是有效的，则称该文件或函数为该变量的作用域，在此作用域内可以引用该变量，所以又称变量在此作用域内“可见”，这种性质又称为变量的可见性，例如图 8-19 中变量 a 和 b 在函数 f1 中“可见”。如果一个变量值在某一时刻是存在的，则认为这一时刻属于该变量的“生存期”，或称该变量在此时刻“存在”。表 8-2 表示各种类型变量的作用域和存在性的情况。

表 8-2 各种类型变量的作用域和存在性的情况

变量存储类别	函数内		函数外	
	作用域	存在性	作用域	存在性
自动变量和寄存器变量	√	√	×	×
静态局部变量	√	√	×	√
静态外部变量	√	√	√(只限本文件)	√
外部变量	√	√	√	√

表 8-2 中“√”表示“是”，“×”表示“否”。可以看到自动变量和寄存器变量在函数内外的“可见性”和“存在性”是一致的，即离开函数后，值不能被引用，值也不存在。静态外

部变量和外部变量的可见性和存在性也是一致的，在离开函数后变量值仍存在，且可被引用，而静态局部变量的可见性和存在性不一致，离开函数后，变量值存在，但不能被引用。

(5) static 对局部变量和全局变量的作用不同。对局部变量来说，它使变量由动态存储方式改变为静态存储方式。而对全局变量来说，它使变量局部化(局部于本文件)，但仍为静态存储方式。从作用域角度看，凡有 static 声明的，其作用域都是局限的，或者是局限于本函数内(静态局部变量)，或者局限于本文件内(静态外部变量)。

8.10 内部函数和外部函数

函数本质上是全局的，因为一个函数要被另外的函数调用，但是，也可以指定函数不能被其他文件调用。根据函数能否被其他源文件调用，将函数区分为内部函数和外部函数。

8.10.1 内部函数

如果一个函数只能被本文件中其他函数所调用，它称为内部函数。在定义内部函数时，在函数名和函数类型的前面加 static，即：

```
static 类型标识符 函数名(形参表);
```

例如：

```
static int fun(int a,int b);
```

内部函数又称静态函数，因为它是用 static 声明的。使用内部函数，可以使函数的作用域只局限于所在文件，在不同的文件中有同名的内部函数，互不干扰。这样不同的人可以分别编写不同的函数，而不必担心所用函数是否会与其他文件中函数同名，通常把只能由同一文件使用的函数和外部变量放在一个文件中，在它们前面都冠以 static 使之局部化，其他文件不能引用。

8.10.2 外部函数

(1) 在定义函数时，如果在函数首部的最左端加关键字 extern，则表示此函数是外部函数，可供其他文件调用。

如函数首部可以写为

```
extern int fun (int a, int b);
```

这样，函数 fun 就可以为其他文件调用。C 语言规定，如果在定义函数时省略 extern，则隐含为外部函数。本书前面所用的函数都是外部函数。

(2) 在需要调用此函数的文件中，用 extern 对函数作声明，表示该函数是在其他文件中定义的外部函数。

例 8.22 有一个字符串，内有若干个字符，今输入一个字符，要求程序将字符串中该字符删去。用外部函数实现。

file1.c(文件 1)

```
#include <stdio.h>
void main()
{
  extern void enter_string(char str[]);
  extern void delete_string(char str[],char ch);
  extern void print_string(char str[]);
  /*以上3行声明在本函数中将要调用的在其他文件中定义的3个函数*/

  char c;
  char str[80];
  enter_string(str);
  scanf("%c",&c);
  delete_string(str,c);
  print_string(str);
}
```

file2.c(文件 2)

```
#include <stdio.h>
void enter_string(char str[80]) /* 定义外部函数 enter_string */
{
  gets(str);                    /*向字符数组输入字符串*/
}
```

file3.c(文件 3)

```
#include <stdio.h>
void delete_string(char str[],char ch) /*定义外部函数 delete_string */
{
  int i,j;
  for(i=j=0;str[i]!='\0';i++)
      if(str[i]!=ch)
          str[j++]=str[i];
  str[j]='\0';
}
```

file4.c(文件 4)

```
#include <stdio.h>
void print_string(char str[])          /*定义外部函数 print_string*/
{
  printf("%s\n",str);
}
```

运行情况如下:

```
abcdefgc↙          (输入 str)
c↙                 (输入要删去的字符)
abdefg             (输出已删去指定字符的字符串)
```

整个程序由 4 个文件组成。每个文件包含一个函数。主函数是主控函数，除声明部分外，由 4 个函数调用语句组成。其中 scanf 是库函数，另外 3 个是用户自己定义的函数。函数 delete_string 的作用是根据给定的字符串和要删除的字符 ch，对字符串作删除处理。

算法是这样的：对 str 数组的字符逐个检查，如果不是被删除的字符就将它存放在数组中，见图 8-21 示意(设删除空格)。从 str[0]开始逐个检查数组元素值是否等于指定要删除的字符，若不是就留在数组中；若是就不保留。从图 8-21 中可以看到，应该使 str[0]赋给 str[0]，str[1]⇒str[1]，str[2]⇒str[2]，str[3]⇒str[3]，然后 str[5]⇒str[4]……请读者注意分析如何控制 i 和 j 的变化，以便使被删除的字符不保留在原数组中。这个题目当然可以设两个数组，把不删除的字符一一赋给新数组。但我们只用一个数组，只把不被删除的字符保留下来。由于 i 总是大于或等于 j，因此最后保留下来的字符不会覆盖未被检测处理的字符。最后将结束符'\0'也复制到被保留的字符后面。

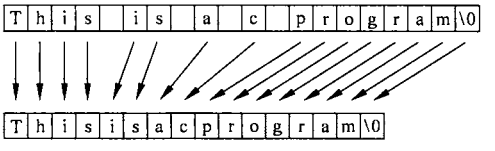


图 8-21

程序中 3 个函数都定义为外部函数。在 main 函数中用 extern 声明在 main 函数中用到的 enter_string、delete_string、print_string 是在其他文件中定义的外部函数。

通过此例可知：使用 extern 声明就能够在一个文件中调用其他文件中定义的函数，或者说把该函数的作用域扩展到本文件。extern 声明的形式就是在函数原型基础上加关键字 extern(见本例 main 函数中的声明形式)。由于函数在本质上是外部的，在程序中经常要调用外部函数，为方便编程，C 语言允许在声明函数时省写 extern。例 8.21 用程序中 main 函数中对 power 函数的声明就没有用 extern，但作用相同。一般都省写 extern，例如例 8.22 程序中 main 函数中的第一个函数声明可写成：

```
enter_string(char str[]);
```

这就是多次用过的函数原型。

由此可以进一步理解函数原型的作用。用函数原型能够把函数的作用域扩展到定义该函数的文件之外(不必使用 extern)。只要在使用该函数的每一个文件中包含该函数的函数原型即可。函数原型通知编译系统：该函数在本文件中稍后定义，或在另一文件中定义。

利用函数原型扩展函数作用域最常见的例子是 # include 命令的应用。在前面几章中曾多次使用过 # include 命令，并提到过：在 # include 命令所指定的“头文件”中包含调

用库函数时所需的信息。例如，在程序中需要调用 sin 函数，但三角函数并不是由用户在本文件中定义的，而是存放在数学函数库中的。按以上的介绍，必须在本文件中写出 sin 函数的原型，否则无法调用 sin 函数。sin 函数的原型是

```
double sin(double x);
```

显然，要求程序设计者在调用库函数时先从手册中查出所用的库函数的原型，并在程序中一一写出来是十分麻烦而困难的。为减少程序设计者的困难，在头文件 math.h 中包括了所有数学函数的原型和其他有关信息，用户只需用以下 #include 命令：

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

在该文件中就能合法地调用各数学库函数了。

有关多文件程序的编译、连接和运行的方法，可参考作者编著的《C 程序设计题解与上机指导(第三版)》中的第二部分。

习　题

8.1 写两个函数，分别求两个整数的最大公约数和最小公倍数，用主函数调用这两个函数，并输出结果。两个整数由键盘输入。

8.2 求方程 $ax^2+bx+c=0$ 的根，用 3 个函数分别求当 b^2-4ac 大于 0、等于 0 和小于 0 时的根并输出结果。从主函数输入 a、b、c 的值。

8.3 写一个判素数的函数，在主函数输入一个整数，输出是否素数的信息。

8.4 写一个函数，使给定的一个 3×3 的二维整型数组转置，即行列互换。

8.5 写一个函数，使输入的一个字符串按反序存放，在主函数中输入和输出字符串。

8.6 写一个函数，将两个字符串连接。

8.7 写一个函数，将一个字符串中的元音字母复制到另一字符串，然后输出。

8.8 写一个函数，输入一个 4 位数字，要求输出这 4 个数字字符，但每两个数字间空一个空格。如输入 1990，应输出“1 ␣ 9 ␣ 9 ␣ 0”。

8.9 编写一个函数，由实参传来一个字符串，统计此字符串中字母、数字、空格和其他字符的个数，在主函数中输入字符串以及输出上述的结果。

8.10 写一个函数，输入一行字符，将此字符串中最长的单词输出。

8.11 写一个函数，用“起泡法”对输入的 10 个字符按由小到大顺序排列。

8.12 用牛顿迭代法求根。方程为 $ax^3+bx^2+cx+d=0$，系数 a、b、c、d 的值依次为 1、2、3、4，由主函数输入。求 x 在 1 附近的一个实根。求出根后由主函数输出。

8.13 用递归方法求 n 阶勒让德多项式的值，递归公式为：

$$p_n(x)=\begin{cases}1 & n=0\\ x & n=1\\ ((2n-1)\cdot x-p_{n-1}(x)-(n-1)*p_{n-2}(x))/n & n\geqslant 1\end{cases}$$

8.14 输入 10 个学生 5 门课的成绩，分别用函数实现下列功能：

① 计算每个学生平均分；

② 计算每门课的平均分；

③ 找出所有 50 个分数中最高的分数所对应的学生和课程；

④ 计算平均分方差：

$$\sigma = \frac{1}{n}\sum x_i^2 - \left(\frac{\sum x_i}{n}\right)^2$$

其中，x_i 为某一学生的平均分。

8.15 写几个函数：

① 输入 10 个职工的姓名和职工号；

② 按职工号由小到大顺序排序，姓名顺序也随之调整；

③ 要求输入一个职工号，用折半查找法找出该职工的姓名，从主函数输入要查找的职工号，输出该职工姓名。

8.16 写一个函数，输入一个十六进制数，输出相应的十进制数。

8.17 用递归法将一个整数 n 转换成字符串。例如，输入 483，应输出字符串“483”。n 的位数不确定，可以是任意位数的整数。

8.18 给出年、月、日，计算该日是该年的第 n 天。

第9章 预处理命令

ANSI C标准规定可以在C源程序中加入一些“预处理命令”(preprocessor directives),以改进程序设计环境,提高编程效率。这些预处理命令是由ANSI C统一规定的,但是它不是C语言本身的组成部分,不能直接对它们进行编译(因为编译程序不能识别它们)。必须在对程序进行通常的编译(包括词法和语法分析、代码生成、优化等)之前,先对程序中这些特殊的命令进行“预处理”,即根据预处理命令对程序作相应的处理(例如,若程序中用#define命令定义了一个符号常量A,则在预处理时将程序中所有的A都置换为指定的字符串。若程序中用#include命令包含一个文件“stdio.h”,则在预处理时将stdio.h文件中的实际内容代替该命令)。

经过预处理后的程序不再包括预处理命令了,最后再由编译程序对预处理后的源程序进行通常的编译处理,得到可供执行的目标代码。现在使用的许多C编译系统都包括了预处理、编译和连接等部分,在进行编译时一气呵成。因此不少用户误认为预处理命令是C语言的一部分,甚至以为它们是C语句,这是不对的。必须正确区别预处理命令和C语句,区别预处理和编译,才能正确使用预处理命令。C语言与其他高级语言的一个重要区别是可以使用预处理命令和具有预处理的功能。

C提供的预处理功能主要有以下3种:

1. 宏定义
2. 文件包含
3. 条件编译

分别用宏定义命令、文件包含命令、条件编译命令来实现。为了与一般C语句相区别,这些命令以符号“#”开头。

9.1 宏 定 义

9.1.1 不带参数的宏定义

用一个指定的标识符(即名字)来代表一个字符串,它的一般形式为

#define 标识符 字符串

这就是已经介绍过的定义符号常量,例如:

```
#define PI 3.1415926
```

它的作用是在本程序文件中用指定的标识符PI来代替“3.1415926”这个字符串,在编译预处理时,将程序中在该命令以后出现的所有的PI都用“3.1415926”代替。这种方法使

用户能以一个简单的名字代替一个长的字符串，因此把这个标识符（名字）称为“宏名”，在预编译时将宏名替换成字符串的过程称为“宏展开”。#define 是宏定义命令。

例 9.1 使用不带参数的宏定义。

```
#include <stdio.h>
#define PI 3.1415926
void main()
  { float l,s,r,v;
    printf("input radius:");
    scanf("%f",&r);
    l=2.0*PI*r;
    s=PI*r*r;
    v=4.0/3*PI*r*r*r;
    printf("l=%10.4f\ns=%10.4f\nv=%10.4f\n",l,s,v);
  }
```

运行情况如下：

```
input radius:4↙
l=25.1327
s=50.2655
v=268.0826
```

说明：

（1）宏名一般习惯用大写字母表示，以便与变量名相区别，但这并非规定，也可用小写字母。

（2）使用宏名代替一个字符串，可以减少程序中重复书写某些字符串的工作量。例如，若不定义 PI 代表 3.1415926，则在程序中要多处出现 3.1415926，不仅麻烦，而且容易写错（或敲错），用宏名代替，简单而不易出错，因为记住一个宏名（它的名字往往用容易理解的单词表示）要比记住一个无规律的字符串容易，而且在读程序时能立即知道它的含义，当需要改变某一个常量时，可以只改变#define 命令行，一改全改。例如，定义数组大小，可以用：

```
#define array_ size 1000
int array[array_size];
```

先指定 array_size 代表常量 1000，因此数组 array 的大小为 1000。如果需要改变数组大小为 500，只需改#define 行：

```
#define array_size 500
```

这样在程序中所有以 array_size 代表的 1000 都全改为 500 了。使用宏定义，可以提高程序的通用性。

（3）宏定义是用宏名代替一个字符串，也就是作简单的置换，不作正确性检查。如果写成

```
#define PI 3.l4l59
```

即把数字1写成小写字母l，预处理时也照样代入，不管是否符合用户原意，也不管含义是否有意义。预编译时不作任何语法检查。只有在编译已被宏展开后的源程序时才会发现语法错误并报错。

(4) 宏定义不是C语句，不必在行末加分号。如果加了分号则会连分号一起进行置换。如：

```
#define PI 3.1415926;
area=PI*r*r;
```

经过宏展开后，该语句为

```
area=3.1415926;*r*r;
```

显然出现语法错误。

(5) #define命令出现在程序中函数的外面，宏名的有效范围为定义命令之后到本源文件结束。通常，#define命令写在文件开头，函数之前，作为文件一部分，在此文件范围内有效。

(6) 可以用#undef命令终止宏定义的作用域。例如：

```
#define G 9.8
void main()
  {
   ⋮
  }
#undef G
f1()
  {
   ⋮
  }
```

由于#undef的作用，使G的作用范围到#undef行终止，因此在f1函数中，G不再代表9.8。这样可以灵活控制宏定义的作用范围。

(7) 在进行宏定义时，可以引用已定义的宏名，可以层层置换。

例9.2 在宏定义中引用已定义的宏名。

```
#include <stdio.h>
#define R 3.0
#define PI   3.1415926
#define L    2*PI*R
#define S    PI*R*R
void main()
  {
    printf("L=%f\nS=%f\n",L,S);
  }
```

运行情况如下：

```
L=18.849556
S=28.274333
```

经过宏展开后，printf 函数中的输出项 L 被展开为 2 * 3.1415926 * 3.0，S 展开为 3.1415926 * 3.0 * 3.0，printf 函数调用语句展开为

```
printf("L=%f\nS=%f\n",2 * 3.1415926 * 3.0,3.1415926 * 3.0 * 3.0);
```

(8) 对程序中用双撇号括起来的字符串内的字符，即使与宏名相同，也不进行置换。例如例 9.2 中的 printf 函数内有两个 L 字符，一个在双撇号内，它不被宏置换，另一个在双撇号外，被宏置换展开。

(9) 宏定义是专门用于预处理命令的一个专用名词，它与定义变量的含义不同，只作字符替换，不分配内存空间。

9.1.2 带参数的宏定义

带参数的宏定义不是进行简单的字符串替换，还要进行参数替换。其定义的一般形式为

#define 宏名(参数表) 字符串

字符串中包含在括号中所指定的参数。例如：

```
#define S(a,b) a * b
    ⋮
area=S(3,2);
```

定义矩形面积 S，a 和 b 是边长。在程序中用了 S(3,2)，把 3 和 2 分别代替宏定义中的形式参数 a、b，即用 3 * 2 代替 S(3,2)。因此赋值语句展开为

```
area=3 * 2;
```

对带参数的宏定义是这样展开置换的：在程序中如果有带实参的宏(如 S(3,2))，则按 #define 命令行中指定的字符串从左到右进行置换。如果串中包含宏中的形参(如 a、b)，则将程序语句中相应的实参(可以是常量、变量或表达式)代替形参。如果宏定义中的字符串中的字符不是参数字符(如 a * b 中的 * 号)，则保留。这样就形成了置换的字符串，见图 9-1。

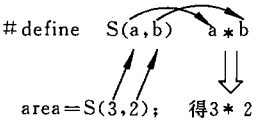


图 9-1

例 9.3 使用带参数的宏。

```
#include <stdio.h>
#define PI 3.1415926
#define S(r) PI * r * r
void main()
 { float a,area;
```

```
    a=3.6;
    area=S(a);
    printf("r=%f\narea=%f\n",a,area);
}
```

运行结果如下：

```
r=3.600000
area=40.715038
```

赋值语句“area=S(a);”经宏展开后为

```
area=3.1415926*a*a;
```

说明：

(1) 对带参数的宏的展开只是将语句中的宏名后面括号内的实参字符串代替#define命令行中的形参。例 9.3 中语句中有 S(a)，在展开时，找到#define 命令行中的 S(r)，将 S(a)中的实参 a 代替宏定义中的字符串“PI*r*r”中的形参 r，得到 PI*a*a。这是容易理解而且不会发生什么问题的。但是，如果有以下语句：

```
area=S(a+b);
```

这时把实参 a+b 代替 PI*r*r 中的形参 r，成为

```
area=PI*a+b*a+b;
```

请注意在 a+b 外面没有括号，显然这与程序设计者的原意不符。原意希望得到

```
area=PI*(a+b)*(a+b);
```

为了得到这个结果，应当在定义时，在字符串中的形式参数外面加一个括号。即

```
#define S(r) PI*(r)*(r)
```

在对 S(a+b)进行宏展开时，将 a+b 代替 r，就成了

```
PI*(a+b)*(a+b)
```

这就达到了目的。

(2) 在宏定义时，在宏名与带参数的括号之间不应加空格；否则将空格以后的字符都作为替代字符串的一部分。例如，如果有

```
#define S (r) PI*r*r
```

被认为 S 是符号常量(不带参数的宏名)，它代表字符串"(r) PI*r*r"。如果在程序中有语句

```
area=S(a);
```

则被展开为

```
area=(r) PI*r*r (a);
```

显然不对了。

有些读者容易把带参数的宏和函数混淆。的确，它们之间有一定类似之处，在调用函数时也是在函数名后的括号内写实参，也要求实参与形参的数目相等。但是带参数的宏定义与函数是不同的。主要有：

(1) 函数调用时，先求出实参表达式的值，然后代入形参。而使用带参数的宏只是进行简单的字符替换。例如上面的 S(a+b)，在宏展开时并不求 a+b 的值，而只将实参字符"a+b"代替形参 r。

(2) 函数调用是在程序运行时处理的，为形参分配临时的内存单元。而宏展开则是在编译前进行的，在展开时并不分配内存单元，不进行值的传递处理，也没有"返回值"的概念。

(3) 对函数中的实参和形参都要定义类型，二者的类型要求一致，如不一致，应进行类型转换。而宏不存在类型问题，宏名无类型，它的参数也无类型，只是一个符号代表，展开时代入指定的字符串即可。宏定义时，字符串可以是任何类型的数据。例如：

```
#define CHAR1 CHINA        (字符)
#define a 3.6              (数值)
```

CHAR1 和 a 不需要定义类型，它们不是变量，在程序中凡遇 CHAR1 均以 CHINA 代之；凡遇 a 均以 3.6 代之，显然不需定义类型。同样，对带参数的宏：

```
#define s(r) PI*r*r
```

r 也不是变量，如果在语句中有 S(3.6)，则展开后为 PI * 3.6 * 3.6，语句中并不出现 r。当然也不必定义 r 的类型。

(4) 调用函数只可得到一个返回值，而用宏可以设法得到几个结果。

例 9.4 通过宏展开得到若干个结果。

```
#include <stdio.h>
#define PI 3.1415926
#define CIRCLE(R,L,S,V) L=2*PI*R;S=PI*R*R;V=4.0/3.0*PI*R*R*R
void main()
  {float r,l,s,v;
   scanf("%f",&r);
   CIRCLE(r,l,s,v);
   printf("r=%6.2f,l=%6.2f,s=%6.2f,v=%6.2f\n",r,l,s,v);
  }
```

对宏进行预编译，展开后的 main 函数如下：

```
void main()
  { float r,l,s,v;
    scanf("%f",&r);
    l=2*3.1415926*r; s=3.1415926*r*r; v=4.0/3.0*3.1415926*r*r*r;
    printf("r=%6.2f,l=%6.2f,s=%6.2f,v=%6.2f\n",r,l,s,v);
  }
```

运行情况如下：

```
3.5↙
r= 3.50,l= 21.99,s= 38.48,v= 179.59
```

请注意，只要已知实参 r 的值，就可以从宏 CIRCLE 的展开中得到 3 个值(l,s,v)。其实，这只不过是字符代替而已，将字符 r 代替宏定义中的 R，l 代替 L，s 代替 S，v 代替 V，而并未在宏展开时求出 l、s、v 的值。

(5) 使用宏次数多时，宏展开后源程序变长，因为每展开一次都使程序增长，而函数调用不会使源程序变长。

(6) 宏替换不占运行时间，只占编译时间。而函数调用则占运行时间(分配单元、保留现场、值传递、返回)。

一般用宏来代表简短的表达式比较合适。有些问题，用宏和函数都可以。例如：

```
#define MAX(x,y) (x)>(y)? (x) : (y)
void main()
  { int a,b,c,d,t;
      ⋮
    t=MAX(a+b,c+d);
      ⋮
  }
```

赋值语句展开后为

```
t=(a+b)>(c+d)? (a+b):(c+d);
```

注意：MAX 不是函数，这里只有一个 main 函数，在 main 函数中就能求出 t 的值。

这个问题也可以用函数来解决。可以定义求两个数中大者的函数 max：

```
int max(int x,int y)      /* 定义 max 函数 */
  {return(x>y? x:y);}
```

在主函数中调用 max 函数：

```
void main()
{ int a,b,c,d,t;
    ⋮
  t=max(a+b,c+d);   /* 调用 max 函数 */
    ⋮
}
```

请仔细分析以上两种方法。

如果善于利用宏定义，可以实现程序的简化，可以事先将程序中的“输出格式”定义好，以减少在输出语句中每次都要写出具体的输出格式的麻烦。

例 9.5 用宏代表输出格式。

```
#include <stdio.h>
#define PR printf
#define NL "\n"
#define D "%d"
#define D1 D NL
#define D2 D D NL
#define D3 D D D NL
#define D4 D D D D NL
#define S "%s"
void main()
 { int a,b,c,d;
   char string[]="CHINA";
   a=1;b=2;c=3;d=4;
   PR(D1,a);
   PR(D2,a,b);
   PR(D3,a,b,c);
   PR(D4,a,b,c,d);
   PR(S,string);
 }
```

运行时输出以下结果：

```
1
12
123
1234
CHINA
```

程序中用 PR 代表 printf；以 NL 代表执行一次“换行”操作；以 D 代表输出一个整型数据的格式符"%d"；以 D1 代表输出完 1 个整数后换行；D2 代表输出 2 个整数后换行；D3 代表输出 3 个整数后换行；D4 代表输出 4 个整数后换行；以 S 代表输出一个字符串的格式符"%s"。可以看到，程序中写输出语句就比较简单了，只要根据需要选择已定义的输出格式即可，连 printf 都可以简写为 PR。

可以参照例 9.5，写出各种输入输出的格式（例如单精度浮点型、双精度浮点型、长整型、十六进制整数、八进制整数、字符型等），把它们单独编成一个文件，它相当一个“格式库”，用 #include 命令“包括”到自己所编的程序中，用户就可以根据情况各取所需了。显然在写大程序时，这样做是很方便的。

9.2 “文件包含”处理

所谓“文件包含”处理是指一个源文件可以将另外一个源文件的全部内容包含进来，即将另外的文件包含到本文件之中。C 语言提供了 #include 命令用来实现“文件包含”的操作。其一般形式为

#include "文件名"

或

#include <文件名>

图 9-2 表示“文件包含”的含义。图 9-2(a)为文件 file1.c,它有一个#include <file2.c>命令,然后还有其他内容(以 A 表示)。图 9-2(b)为另一文件 file2.c,文件内容以 B 表示。在编译预处理时,要对#include 命令进行“文件包含”处理:将 file2.c 的全部内容复制插入到#include <file2.c>命令处,即 file2.c 被包含到 file1.c 中,得到图 9-2(c)所示的结果。在编译时,对将经编译预处理的 file1.c(即图 9-2(c)所示)作为一个源文件单位进行编译。

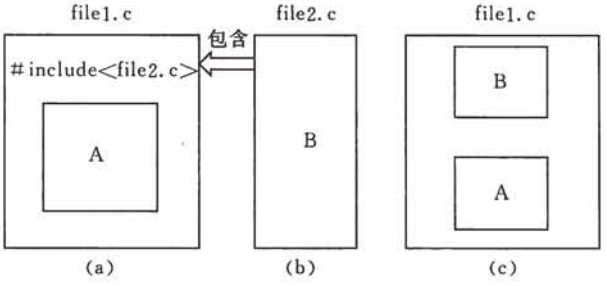


图 9-2

“文件包含”命令是很有用的,它可以节省程序设计人员的重复劳动。例如,某单位的人员往往使用一组固定的符号常量(如 g=9.81,pi=3.1415926,e=2.718,c=……),可以把这些宏定义命令组成一个头文件,然后各人都可以用#include 命令将这些符号常量包含到自己所写的源文件中。这样每个人就可以不必重复定义这些符号常量,相当于工业上的标准零件,拿来就用。

例 9.6 将例 9.5 的格式宏做成头文件,把它包含在用户程序中。

(1) 将格式宏做成头文件 format.h

```
#define PR printf
#define NL "\n"
#define D "%d"
#define D1 D NL
#define D2 D D NL
#define D3 D D D NL
#define D4 D D D D NL
#define S "%s"
```

(2) 主文件 file1.c

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

```
{ int a,b,c,d;
  char string[]="CHINA";
  a=1;b=2;c=3;d=4;
  PR(D1,a);
  PR(D2,a,b);
  PR(D3,a,b,c);
  PR(D4,a,b,c,d);
  PR(S,string);
}
```

注意：在编译时并不是对两个文件分别进行编译，然后再将它们的目标程序连接的，而是在经过编译预处理后将头文件 format.h 包含到主文件中，得到一个新的源程序，然后对这个文件进行编译，得到一个目标(.obj)文件。被包含的文件成为新的源文件的一部分，而单独生成目标文件。

这种常用在文件头部的被包含的文件称为“标题文件”或“头文件”，常以“.h”为后缀(h 为 head(头)的缩写)，如“format.h”文件。当然不用“.h”为后缀，而用“.c”为后缀或者没有后缀也是可以的，但用“.h”作后缀更能表示此文件的性质。

如果需要修改程序中常用的一些参数，可以不必修改每个程序，只需把这些参数放在一个头文件中，在需要时修改头文件即可。但是应当注意，被包含文件修改后，凡包含此文件的所有文件都要全部重新编译。

头文件除了可以包括函数原型和宏定义外，也可以包括结构体类型定义(见第 11 章)和全局变量定义等。

说明：

(1) 一个#include 命令只能指定一个被包含文件，如果要包含 n 个文件，要用 n 个#include命令。

(2) 如果文件 1 包含文件 2，而在文件 2 中要用到文件 3 的内容，则可在文件 1 中用两个 include 命令分别包含文件 2 和文件 3，而且文件 3 应出现在文件 2 之前，即在 file1.c 中定义：

```
#include "file3.h"
#include "file2.h"
```

这样，file1 和 file2 都可以用 file3 的内容。在 file2 中不必再用#include "file3.h"了(以上是假设 file2.h 在本程序中只被 file1.c 包含，而不出现在其他场合)。

(3) 在一个被包含文件中又可以包含另一个被包含文件，即文件包含是可以嵌套的。例如，上面的问题也可以这样处理，见图 9-3。

它的作用与图 9-4 所示相同。

(4) 在#include 命令中，文件名可以用双撇号或尖括号括起来，如可以在 file1.c 中用

```
#include <file2.h>
```

或

#include "file2.h"

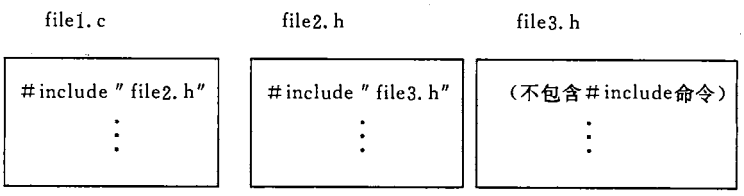


图 9-3

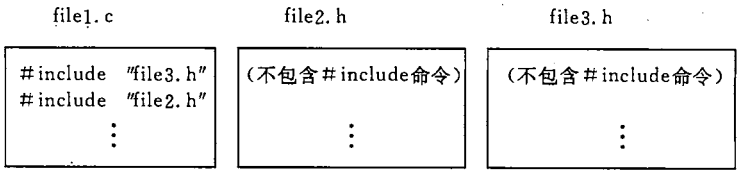


图 9-4

都是合法的。二者的区别是用尖括号(如<stdio.h>形式)时,系统到存放C库函数头文件的目录中寻找要包含的文件,这称为标准方式。用双撇号(即"file2.h"形式)时,系统先在用户当前目录中寻找要包含的文件,若找不到,再按标准方式查找(即再按尖括号的方式查找)。一般来说,如果为调用库函数而用#include命令来包含相关的头文件,则用尖括号,以节省查找时间。如果要包含的是用户自己编写的文件(这种文件一般都在用户当前目录中),一般用双撇号。若文件不在当前目录中,在双撇号内应给出文件路径(如#include"C:\wang\file2.h")。

(5) 被包含文件(file2.h)与其所在的文件(即用#include命令的源文件file1.c),在预编译后已成为同一个文件(而不是两个文件)。因此,如果file2.h中有全局静态变量,它也在file1.c文件中有效,不必用extern声明。

9.3 条件编译

一般情况下,源程序中所有行都参加编译。但是有时希望程序中一部分内容只在满足一定条件时才进行编译,也就是对这一部分内容指定编译的条件,这就是"条件编译"。有时,希望在满足某条件时对某一组语句进行编译,而当条件不满足时则编译另一组语句。

条件编译命令有以下几种形式:

(1) **#ifdef 标识符**
  程序段1
#else
  程序段2
#endif

它的作用是若所指定的标识符已经被 # define 命令定义过，则在程序编译阶段编译程序段 1；否则编译程序段 2。其中 # else 部分可以没有，即

```
#ifdef 标识符
    程序段 1
#endif
```

这里的“程序段”可以是语句组，也可以是命令行。这种条件编译对于提高 C 源程序的通用性是很有好处的。如果一个 C 源程序在不同计算机系统上运行，而不同的计算机又有一定的差异（例如，有的机器以 16 位（2 个字节）来存放一个整数，而有的则以 32 位存放一个整数），这样往往需要对源程序作必要的修改，这就降低了程序的通用性。可以用以下的条件编译来处理：

```
#ifdef COMPUTER_A
    #define INTEGER_SIZE 16
#else
    #define INTEGER_SIZE 32
#endif
```

如果在这组条件编译命令之前曾出现以下命令行：

```
#define COMPUTER_A 0
```

或将 COMPUTER_A 定义为任何字符串，甚至是

```
#define COMPUTER_A
```

即只要 COMPUTER_A 已被定义过，则在程序编译时就编译下面的命令行：

```
#define INTEGER_SIZE 16
```

否则，就编译下面的命令行：

```
#define INTEGER_SIZE 32
```

则预编译后程序中的 INTEGER_SIZE 都用 16 代替；否则都用 32 代替。

这样，源程序可以不必做任何修改就可以用于不同类型的计算机系统。当然以上介绍的只是一种简单的情况，读者可以根据此思路设计出其他的条件编译。

例如，在调试程序时，常常希望输出一些所需的信息，而在调试完成后不再输出这些信息。可以在源程序中插入以下的条件编译段：

```
#ifdef DEBUG
    printf("x=%d,y=%d,z=%d\n",x,y,z);
#endif
```

如果在它的前面有以下命令行：

```
#define DEBUG
```

则在程序运行时输出 x、y、z 的值，以便调试时分析。调试完成后只需将这个 # define 命令行删去即可。有人可能觉得不用条件编译也可达此目的，即在调试时加一批 printf 语句，调试后一一将 printf 语句删去。的确，这是可以的。但是，当调试时加的 printf 语句比较多时，修改的工作量是很大的。用条件编译，则不必一一删改 printf 语句，只需删除前面的一条“# define DEBUG”命令即可，这时所有的用 DEBUG 作标识符的条件编译段都使其中的 printf 语句不起作用，即起统一控制的作用，如同一个“开关”一样。

(2) **# ifndef 标识符**

程序段 1

 # else

程序段 2

 # endif

只是第一行与第一种形式不同：将“ifdef”改为“ifndef”。它的作用是若标识符未被定义过则编译程序段 1；否则编译程序段 2。这种形式与第一种形式的作用相反。

以上两种形式的用法差不多，根据需要任选一种，视方便而定。例如，上面调试时输出信息的条件编译段也可以改为

```
# ifndef RUN
printf("x=%d,y=%d,z=%d\n",x,y,z);
# endif
```

如果在此之前未对 RUN 定义，则输出 x、y、z 的值。调试完成后，在运行之前，加以下命令行：

```
# define RUN
```

则不再输出 x、y、z 的值。

(3) **# if 表达式**

程序段 1

 # else

程序段 2

 # endif

它的作用是当指定的表达式值为真(非零)时就编译程序段 1；否则编译程序段 2。可以事先给定条件，使程序在不同的条件下执行不同的功能。

例 9.7 输入一行字母字符，根据需要设置条件编译，使之能将字母全改为大写输出，或全改为小写字母输出。

程序如下：

```
# include <stdio. h>
# define LETTER 1
void main()
    { char str[20]="C Language",c;
```

```
    int i;
    i=0;
    while((c=str[i])!='\0')
    { i++;
      #if LETTER
        if(c>='a' && c<='z')
          c=c-32;
      #else
        if(c>='A' && c<='Z')
          c=c+32;
      #endif
      printf("%c",c);
    }
    printf("\n")
}
```

条件编译

运行结果为：

C LANGUAGE

现在先定义 LETTER 为 1，这样在对条件编译命令进行预处理时，由于 LETTER 为真(非零)，则对第一个 if 语句进行编译，运行时使小写字母变为大写。如果将程序第一行改为

#define LETTER 0

则在预处理时，对第二个 if 语句进行编译处理，使大写字母变成小写字母(大写字母与相应的小写字母的 ASCII 代码差为 32)。此时运行情况为

c language

有的读者可能会问，不用条件编译命令而直接用 if 语句也能达到要求，用条件编译命令有什么好处呢？的确，对这个问题完全可以不用条件编译处理而用 if 语句处理，但那样做，目标程序长(因为所有语句都编译)，运行时间长(因为在程序运行时对 if 语句进行测试)。而采用条件编译，可以减少被编译的语句，从而减少目标程序的长度，减少运行时间。当条件编译段比较多时，目标程序长度可以大大减少。以上举例是最简单的情况，只是为了说明怎样使用条件编译，有人会觉得其优越性不太明显，但是如果程序比较复杂而善于使用条件编译，其优越性是比较明显的。

本章介绍的预编译功能是 C 语言特有的，有利于程序的可移植性，增加程序的灵活性。

习　题

9.1 定义一个带参数的宏，使两个参数的值互换，并写出程序，输入两个数作为使用宏时的实参。输出已交换后的两个值。

9.2 输入两个整数，求它们相除的余数。用带参数的宏来实现，编程序。

9.3 三角形的面积为

$$\text{area}=\sqrt{s(s-a)(s-b)(s-c)}$$

其中 $s=\frac{1}{2}(a+b+c)$。a、b、c 为三角形的三边。定义两个带参数的宏，一个用来求 s，另一个宏用来求 area。写程序，在程序中用带实参的宏名来求面积 area。

9.4 给年份 year 定义一个宏，以判别该年份是否闰年。

提示：宏名可定为 LEAP_YEAR，形参为 y，即定义宏的形式为

#define LEAP_YEAR(y)（读者设计的字符串）

在程序中用以下语句输出结果：

```
if (LEAP_YEAR(year)) printf ("%d is a leap year.\n",year);
else printf ("%d is not a leap year.\n",year);
```

9.5 请分析以下一组宏所定义的输出格式：

```
#define NL putchar ('\n')
#define PR(format,value) printf("value=%format\t",(value))
#define PRINT1(f,x1) PR(f,x1);NL
#define PRINT2(f,x1,x2)PR(f,x1);PRINT1(f,x2)
```

如果在程序中有以下的宏引用：

```
PR (d,x);
PRINT1(d,x);
PRINT2(d,x1,x2);
```

写出宏展开后的情况，并写出应输出的结果，设 x=5，x1=3，x2=8。

9.6 请设计输出实数的格式，实数用“%6.2f”格式输出。要求：

(1) 一行输出 1 个实数；

(2) 一行内输出 2 个实数；

(3) 一行内输出 3 个实数。

9.7 设计所需的各种各样的输出格式(包括整数、实数、字符串等)，用一个文件名"format.h"把这些信息都放到此文件内，另编一个程序文件，用#include "format.h"命令，以确保能使用这些格式。

9.8 分别用函数和带参数的宏，从 3 个数中找出最大数。

9.9 试述“文件包含”和程序文件的连接(link)的概念，二者有何不同？

9.10 用条件编译方法实现以下功能：

输入一行电报文字，可以任选两种输出：一为原文输出；一为将字母变成其下一字母(如'a'变成'b'，……，'z'变成'a')，其他非字母字符不变。用#define 命令来控制是否要译成密码。例如：

```
#define CHANGE    1
```

则输出密码。若

```
#define CHANGE    0
```

则不译成密码，按原码输出。

第10章 指　　针

指针是C语言中的一个重要概念，也是C语言的一个重要特色。正确而灵活地运用它，可以有效地表示复杂的数据结构；能动态分配内存；方便地使用字符串；有效而方便地使用数组；在调用函数时能获得1个以上的结果；能直接处理内存单元地址等，这对设计系统软件是非常必要的。掌握指针的应用，可以使程序简洁、紧凑、高效。每一个学习和使用C语言的人，都应当深入地学习和掌握指针。可以说，不掌握指针就是没有掌握C的精华。

指针的概念比较复杂，使用也比较灵活，因此初学时常会出错，务请在学习本章内容时十分小心，多思考、多比较、多上机，在实践中掌握它。我们在叙述时也力图用通俗易懂的方法使读者易于理解。

10.1 地址和指针的概念

为了说清楚什么是指针，必须弄清楚数据在内存中是如何存储的，又是如何读取的。

如果在程序中定义了一个变量，在对程序进行编译时，系统就会给这个变量分配内存单元。编译系统根据程序中定义的变量类型，分配一定长度的空间。例如，一般为整型变量分配2个字节，对单精度浮点型变量分配4个字节，对字符型变量分配1个字节。内存区的每一个字节有一个编号，这就是“地址”，它相当于旅馆中的房间号。在地址所标志的内存单元中存放数据，这相当于旅馆房间中居住的旅客一样。

请务必弄清楚一个内存单元的地址与内存单元的内容这两个概念的区别，假设程序已定义了3个整型变量i、j、k，编译时系统分配2000和2001两个字节给变量i，2002、2003字节给j，2004、2005给k。在程序中一般是通过变量名来对内存单元进行存取操作的。其实程序经过编译以后已经将变量名转换为变量的地址，对变量值的存取都是通过地址进行的。假如有输出语句

```
printf("%d",i);
```

它是这样执行的：根据变量名与地址的对应关系(这个对应关系是在编译时确定的)，找到变量i的地址2000，然后从由2000开始的两个字节中取出数据(即变量的值3)，把它输出。假如有输入语句

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

在执行时，把从键盘输入的值送到地址为2000开始的整型存储单元中。如果有语句

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

则从 2000、2001 字节取出 i 的值(3),再从 2002、2003 字节取出 j 的值(6),将它们相加后再将其和(9)送到 k 所占用的 2004、2005 字节单元中。这种按变量地址存取变量值的方式称为"直接访问"方式。

还可以采用另一种称之为"间接访问"的方式,将变量 i 的地址存放在另一个变量中。按 C 语言的规定,可以在程序中定义整型变量、实型变量、字符变量等,也可以定义这样一种特殊的变量,它是存放地址的。假设我们定义了一个变量 i_pointer,用来存放整型变量的地址,它被分配为 3010、3011 两个字节。可以通过下面语句将 i 的地址(2000)存放到 i_pointer 中。

```
i_pointer=&i;
```

这时,i_pointer 的值就是 2000,即变量 i 所占用单元的起始地址。要存取变量 i 的值,也可以采用间接方式:先找到存放"i 的地址"的变量 i_pointer,从中取出 i 的地址(2000),然后到 2000、2001 字节取出 i 的值(3),见图 10-1。

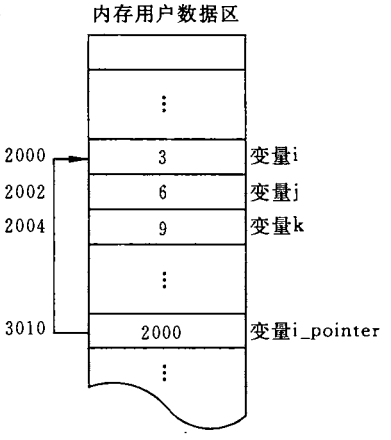


图 10-1

打个比方,为了开一个 A 抽屉,有两种办法,一种是将 A 钥匙带在身上,需要时直接找出该钥匙打开抽屉,取出所需的东西。另一种办法是:为安全起见,将该 A 钥匙放到另一抽屉 B 中锁起来。如果需要打开 A 抽屉,就需要先找出 B 钥匙,打开 B 抽屉,取出 A 钥匙,再打开 A 抽屉,取出 A 抽屉中之物,这就是"间接访问"。

图 10-2(a)表示直接访问,已经知道变量 i 的地址,根据此地址直接对变量 i 的存储单元进行存取访问。图 10-2(b) 表示间接访问,先找到存放变量 i 地址的变量 i_pointer,从其中得到变量 i 的地址,然后找到变量 i 的存储单元,对它进行存取访问。

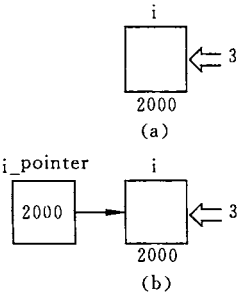


图 10-2

为了表示将数值 3 送到变量中,可以有两种表达方法:

(1) 将 3 送到变量 i 所标志的单元中,见图 10-2(a)。

(2) 将 3 送到变量 i_pointer 所指向的单元(即 i 所标志的单元)中,见图 10-2(b)。

所谓指向就是通过地址来体现的。假设 i_pointer 中的值为 2000,它是变量 i 的地址,这样就在 i_pointer 和变量 i 之间建立起一种联系,即通过 i_pointer 能知道 i 的地址,从而找到变量 i 的内存单元。图 10-2 中以箭头表示这种"指向"关系。

由于通过地址能找到所需的变量单元,我们可以说,地址指向该变量单元(如同说,一个房间号"指向"某一房间一样)。因此在 C 语言中,将地址形象化地称为"指针"。意思是通过它能找到以它为地址的内存单元(例如根据地址 2000 就能找到变量 i 的存储单元,从而读取其中的值)。

一个变量的地址称为该变量的“指针”。例如，地址 2000 是变量 i 的指针。如果有一个变量专门用来存放另一变量的地址(即指针)，则它称为“指针变量”。上述的 i_pointer 就是一个指针变量。指针变量的值(即指针变量中存放的值)是地址(即指针)。请区分“指针”和“指针变量”这两个概念。例如，可以说变量 i 的指针是 2000，而不能说 i 的指针变量是 2000。指针是一个地址，而指针变量是存放地址的变量。

10.2 变量的指针和指向变量的指针变量

如前所述，变量的指针就是变量的地址。存放变量地址的变量是指针变量，它用来指向另一个变量。为了表示指针变量和它所指向的变量之间的联系，在程序中用“*”符号表示“指向”，如果已定义 i_pointer 为指针变量，则(*i_pointer)是 i_pointer 所指向的变量，见图 10-3。

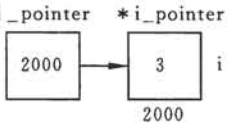


图 10-3

可以看到，*i_pointer 也代表一个变量，它和变量 i 是同一回事。下面两个语句作用相同：

① i=3;

② *i_pointer=3;

第②个语句的含义是将 3 赋给指针变量 i_pointer 所指向的变量。

10.2.1 定义一个指针变量

C 语言规定所有变量在使用前必须定义，指定其类型，并按此分配内存单元。指针变量不同于整型变量和其他类型的变量，它是用来专门存放地址的，必须将它定义为“指针类型”。先看一个具体例子：

```
int i,j;
int *pointer_1,*pointer_2;
```

第 1 行定义了两个整型变量 i 和 j，第 2 行定义了两个指针变量：pointer_1 和 pointer_2，它们是指向整型变量的指针变量。左端的 int 是在定义指针变量时必须指定的“基类型”。指针变量的基类型用来指定该指针变量可以指向的变量的类型。例如，上面定义的基类型为 int 的指针变量 pointer_1 和 pointer_2，可以用来指向整型的变量 i 和 j，但不能指向浮点型变量 a 和 b。

定义指针变量的一般形式为

基类型 *指针变量名;

下面都是合法的定义：

```
float *pointer_3;      (pointer_3 是指向 float 型变量的指针变量)
char *pointer_4;       (pointer_4 是指向字符型变量的指针变量)
```

那么，怎样使一个指针变量指向另一个变量呢？可以用赋值语句使一个指针变量得到另一个变量的地址，从而使它指向一个该变量。例如：

```
pointer_1=&i;
pointer_2=&j;
```

将变量 i 的地址存放到指针变量 pointer_1 中，因此 pointer_1 就“指向”了变量 i。同样，将变量 j 的地址存放到指针变量 pointer_2 中，因此 pointer_2 就“指向”了变量 j，见图 10-4。

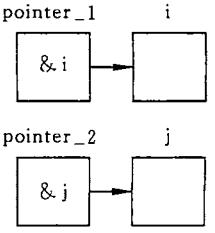


图 10-4

在定义指针变量时要注意两点：

(1) 指针变量前面的“*”表示该变量的类型为指针型变量。指针变量名是 pointer_1、pointer_2，而不是 *pointer_1、*pointer_2。这是与定义整型或浮点型变量的形式不同的。

(2) 在定义指针变量时必须指定基类型。有的读者认为既然指针变量是存放地址的，那么只需要指定其为“指针型变量”即可，为什么还要指定基类型呢？要知道不同类型的数据在内存中所占的字节数是不相同的(例如整型数据占 2 字节，字符型数据占 1 字节)，在本章的稍后将要介绍指针的移动和指针的运算(加、减)，例如“使指针移动 1 个位置”或“使指针值加 1”，这个 1 代表什么呢？如果指针是指向一个整型变量的，那么“使指针移动 1 个位置”意味着移动 2 个字节，“使指针加 1”意味着使地址值加 2 个字节。如果指针是指向一个浮点型变量的，则增加的不是 2 而是 4。因此必须指定指针变量所指向的变量的类型，即基类型。一个指针变量只能指向同一个类型的变量，不能忽而指向一个整型变量，忽而指向一个实型变量。在前面定义的 pointer_1 和 pointer_2 只能指向整型数据。

对上述指针变量的定义也可以这样理解：“int *pointer_1, *pointer_2;”定义了 *pointer_1和，*pointer_2 是整型变量，如同：“int a,b;”定义了 a 和 b 是整型变量一样。而 *pointer_1 和 *pointer_2 是 pointer_1 和 pointer_2 所指向的变量，pointer_1 和 pointer_2 是指针变量。

需要特别注意的是，只有整型变量的地址才能放到指向整型变量的指针变量中。下面的赋值是错误的：

```
float a;                /* 定义 a 为 float 型变量 */
int *pointer_1;         /* 定义 pointer_1 为基类型的 int 的指针变量 */
pointer_1=&a;           /* 将 float 型变量的地址放到指向整型变量的指针变量中，错误 */
```

10.2.2 指针变量的引用

请牢记，指针变量中只能存放地址(指针)，不要将一个整数(或任何其他非地址类型的数据)赋给一个指针变量。下面的赋值是不合法的：

```
*pointer_1=100;        /* pointer_1 为指针变量，100 为整数 */
```

有两个有关的运算符：

(1) &：取地址运算符。

(2) *：指针运算符(或称“间接访问”运算符)，取其指向的内容。

例如，&a 为变量 a 的地址，*p 为指针变量 p 所指向的存储单元的内容(即 p 所指向

的变量的值)。

例 10.1 通过指针变量访问整型变量。

```
#include <stdio.h>
void main()
 { int a,b;
   int *pointer_1, *pointer_2;
   a=100;b=10;
   pointer_1=&a;      /*把变量 a 的地址赋给 pointer_1 */
   pointer_2=&b;      /*把变量 b 的地址赋给 pointer_2 */
   printf("%d,%d\n",a,b);
   printf("%d,%d\n",*pointer_1,*pointer_2);
 }
```

运行结果为:

```
100,10
100,10
```

对程序的说明:

(1) 在开头处虽然定义了两个指针变量 pointer_1 和 pointer_2,但它们并未指向任何一个整型变量,只是提供两个指针变量,规定它们可以指向整型变量,至于指向哪一个整型变量,要在程序语句中指定。程序第 6、7 行的作用就是使 pointer_1 指向 a,pointer_2 指向 b,见图 10-5。此时 pointer_1 的值为 &a(即 a 的地址),pointer_2 的值为 &b。

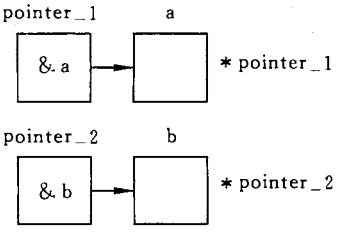


图 10-5

(2) 最后一行的 *pointer_1 和 *pointer_2 就是变量 a 和 b。最后两个 printf 函数作用是相同的。

(3) 程序中有两处出现 *pointer_1 和 *pointer_2,请区分它们的不同含义。程序第 4 行的 *pointer_1 和 *pointer_2 表示定义两个指针变量 pointer_1、pointer_2。它们前面的"*"只是表示该变量是指针变量。程序最后一行 printf 函数中的 *pointer_1 和 *pointer_2则代表 pointer_1 和 pointer_2 所指向的变量。

(4) 第 6、7 行"pointer_1=&a;"和"pointer_2=&b;"是将 a 和 b 的地址分别赋给 pointer_1 和 pointer_2。注意不应写成:"*pointer_1=&a;"和"*pointer_2=&b;"。因为 a 的地址是赋给指针变量 pointer_1,而不是赋给 *pointer_1(即变量 a)。请对照图 10-5分析。

下面对"&"和"*"运算符再作些说明:

如果已执行了语句

```
pointer_1=&a;
```

(1) &*pointer_1 的含义是什么?"&"和"*"两个运算符的优先级别相同,但按自

右而左方向结合，因此先进行 * pointer_1 的运算，它就是变量 a，再执行 & 运算。因此，& * pointer_1 与 &a 相同，即变量 a 的地址。

如果有

```
pointer_2=& * pointer_1;
```

它的作用是将 &a(a 的地址)赋给 pointer_2，如果 pointer_2 原来指向 b，经过重新赋值后它已不再指向 b 了，而指向了 a，见图 10-6。图 10-6(a)是原来的情况，图 10-6(b)是执行上述赋值语句后的情况。

(2) * &a 的含义是什么？先进行 &a 运算，得 a 的地址，再进行 * 运算，即 &a 所指向的变量，也就是变量 a。 * &a 和 * pointer_1 的作用是一样的，它们都等价于变量 a，即 * &a 与 a 等价，见图 10-7。

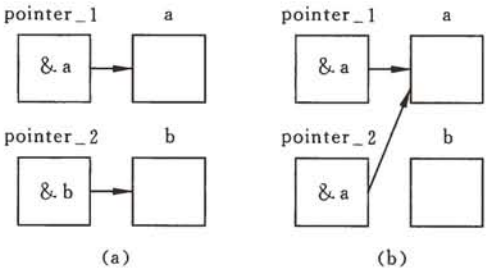


图 10-6

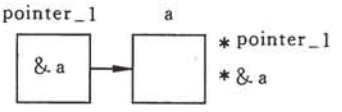


图 10-7

(3)(* pointer_1)++相当于 a++。注意括号是必要的，如果没有括号，就成为了 * pointer_1++，从附录可知：++和 * 为同一优先级别，而结合方向为自右而左，因此它相当于 *(pointer_1++)。由于++在 pointer_1 的右侧，是“后加”，因此先对 pointer_1的原值进行 * 运算，得到 a 的值，然后使 pointer_1 的值改变，这样 pointer_1 不再指向 a 了。

下面举一个指针变量应用的例子。

例 10.2 输入 a 和 b 两个整数，按先大后小的顺序输出 a 和 b。

```
#include <stdio.h>
void main()
  { int *p1,*p2,*p,a,b;
    scanf("%d,%d",&a,&b);
    p1=&a;p2=&b;
    if(a<b)
      {p=p1;p1=p2;p2=p;}
    printf("a=%d,b=%d\n\n",a,b);
    printf("max=%d,min=%d\n",*p1,*p2);
  }
```

运行情况如下：

```
5,9↙
```

a=5,b=9
max=9,min=5

当输入 a=5,b=9 时,由于 a<b,将 p1 和 p2 交换。交换前的情况见图 10-8(a),交换后的情况见图 10-8(b)。

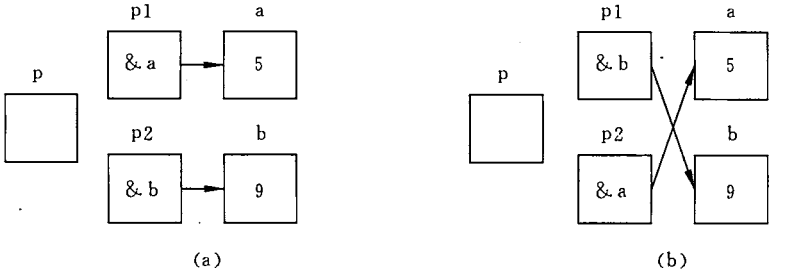


图 10-8

请注意,a 和 b 并未交换,它们仍保持原值,但 p1 和 p2 的值改变了。p1 的值原为 &a,后来变成 &b,p2 原值为 &b,后来变成 &a。这样在输出 * p1 和 * p2 时,实际上是输出变量 b 和 a 的值,所以先输出 9,然后输出 5。

这个问题的算法是不交换整型变量的值,而是交换两个指针变量的值(即 a 和 b 的地址)。

10.2.3 指针变量作为函数参数

函数的参数不仅可以是整型、浮点型、字符型等数据,还可以是指针类型。它的作用是将一个变量的地址传送到另一个函数中。

下面通过一个例子来说明。

例 10.3 题目要求同例 10.2,即对输入的两个整数按大小顺序输出。

现用函数处理,而且用指针类型的数据作函数参数。程序如下:

```
#include <stdio.h>
void main()
  { void swap(int * p1,int * p2);
    int a,b;
    int * pointer_1, * pointer_2;
    scanf("%d,%d",&a,&b);
    pointer_1=&a; pointer_2=&b;
    if(a<b) swap(pointer_1,pointer_2);
    printf("\n%d,%d\n",a,b);
  }
void swap(int * p1,int * p2)
  {int temp;
   temp= * p1;
   * p1= * p2;
   * p2=temp;
```

```
}
```

运行情况如下：

```
5,9↙
9,5
```

对程序的说明：swap 是用户定义的函数，它的作用是交换两个变量(a 和 b)的值。swap 函数的两个形参 p1、p2 是指针变量。程序运行时，先执行 main 函数，输入 a 和 b 的值(现输入 5 和 9)。然后将 a 和 b 的地址分别赋给指针变量 pointer_1 和 pointer_2，使 pointer_1 指向 a，pointer_2 指向 b，见图 10-9(a)。接着执行 if 语句，由于 a<b，因此执行 swap 函数。注意实参 pointer_1 和 pointer_2 是指针变量，在函数调用时，将实参变量的值传送给形参变量，采取的依然是“值传递”方式。因此虚实结合后形参 p1 的值为 &a，p2 的值为 &b，见图 10-9(b)。这时 p1 和 pointer_1 都指向变量 a，p2 和 pointer_2 都指向 b。接着执行 swap 函数的函数体，使 * p1 和 * p2 的值互换，也就是使 a 和 b 的值互换。互换后的情况见图 10-9(c)。函数调用结束后，形参 p1 和 p2 不复存在(已释放)，情况如图 10-9(d)所示。最后在 main 函数中输出的 a 和 b 的值已是经过交换的值(a=9，b=5)。

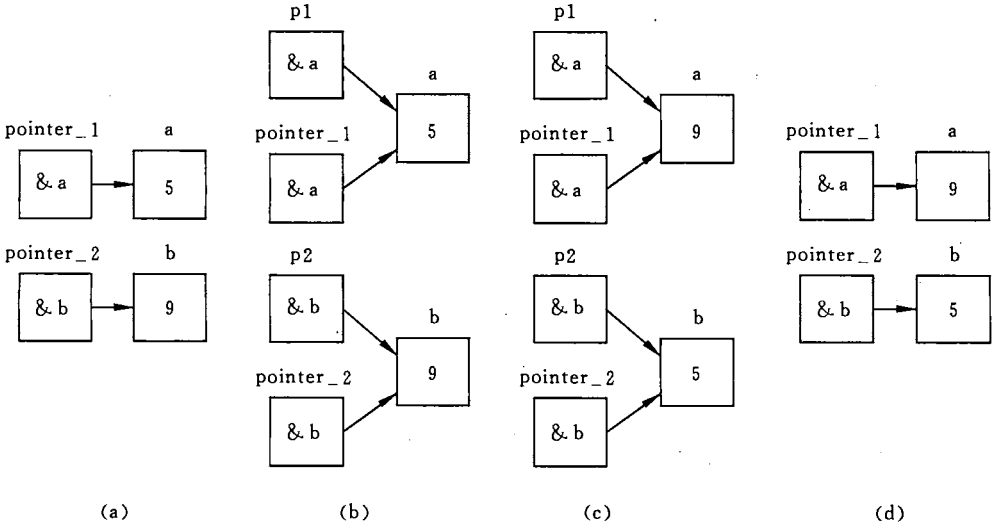


图 10-9

请注意交换 * p1 和 * p2 的值是如何实现的。如果写成以下这样就有问题了：

```
void swap(int *p1,int *p2)
  {int *temp;
   *temp=*p1;    /*此语句有问题*/
   *p1=*p2;
   *p2=*temp;
  }
```

* p1 就是 a，是整型变量。而 * temp 是指针变量 temp 所指向的变量。但 temp 中并

无确定的值(它的值是不可预见的),因此 temp 所指向的单元也是不可预见的。所以,对 * temp 赋值有可能给一个存储着重要数据的存储单元赋值,这样就会破坏系统的正常工作状况。应该将 * p1 的值赋给一个整型变量,如例 10.3 程序所示那样,用整型变量 temp 作为临时辅助变量实现 * p1 和 * p2 的交换。

注意:本例采取的方法是交换 a 和 b 的值,而 p1 和 p2 的值不变。这恰和例 10.2 相反。

可以看到,在执行 swap 函数后,变量 a 和 b 的值改变了。请仔细分析,这个改变是怎么实现的。这个改变不是通过将形参值传回实参来实现的。请读者考虑一下能否通过下面的函数实现 a 和 b 互换。

```
void swap(int x,int y)
  { int temp;
    temp=x;
    x=y;
    y=temp;
  }
```

如果在 main 函数中调用 swap 函数:

```
swap(a,b);
```

会有什么结果呢? 如图 10-10 所示。在函数调用时,a 的值传送给 x,b 的值传送给 y,见图 10-10(a)。执行完 swap 函数后,x 和 y 的值是互换了,但并未影响到 a 和 b 的值。在函数结束时,变量 x 和 y 释放了,main 函数中的 a 和 b 并未互换,见图 10-10(b)。也就是说,由于"单向传送"的"值传递"方式,形参值的改变不能使实参的值随之改变。

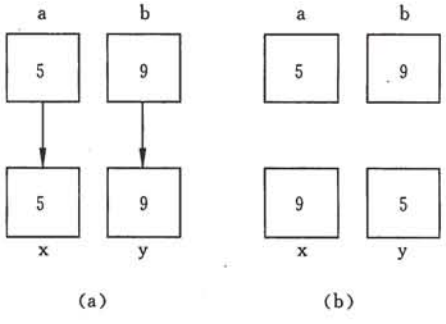


图 10-10

为了使在函数中改变了的变量值能被 main 函数所用,不能采取上述把要改变值的变量作为参数的办法,而应该用指针变量作为函数参数,在函数执行过程中使指针变量所指向的变量值发生变化,函数调用结束后,这些变量值的变化依然保留下来,这样就实现了"通过调用函数使变量的值发生变化,在主调函数(如 main 函数)中可以使用这些改变了的值"的目的。

如果想通过函数调用得到 n 个要改变的值,可以:

① 在主调函数中设 n 个变量,用 n 个指针变量指向它们;

② 然后将指针变量作实参,将这 n 个变量的地址传给所调用的函数的形参;

③ 通过形参指针变量,改变该 n 个变量的值;

④ 主调函数中就可以使用这些改变了值的变量。

请读者按此思路仔细理解例 10.3 程序。

请注意,不能企图通过改变指针形参的值而使指针实参的值改变。请看下面的程序:

```
#include <stdio.h>
void main()
  { void swap(int *p1,int *p2);
    int a,b;
    int *pointer_1, *pointer_2;
    scanf("%d,%d",&a,&b);
    pointer_1=&a;
    pointer_2=&b;
    if(a<b) swap(pointer_1,pointer_2);
    printf("\n%d,%d\n", *pointer_1, *pointer_2);
  }
void swap(int *p1,int *p2)
  { int *p;
    p=p1;
    p1=p2;
    p2=p;
  }
```

程序编写者的意图是:交换 pointer_1 和 pointer_2 的值,使 pointer_1 指向值大的变量。其设想是:

① 先使 pointer_1 指向 a,pointer_2 指向 b,见图 10-11(a)。

② 调用 swap 函数,将 pointer_1 的值传给 p1,pointer_2 传给 p2,见图 10-11(b)。

③ 在 swap 函数中使 p1 与 p2 的值交换,见图 10-11(c)。

④ 形参 p1、p2 将地址传回实参 pointer_1 和 pointer_2,使 pointer_1 指向 b,pointer_2 指向 a,见图 10-11(d)。然后输出 *pointer_1 和 *pointer_2,想得到输出"9,5"。

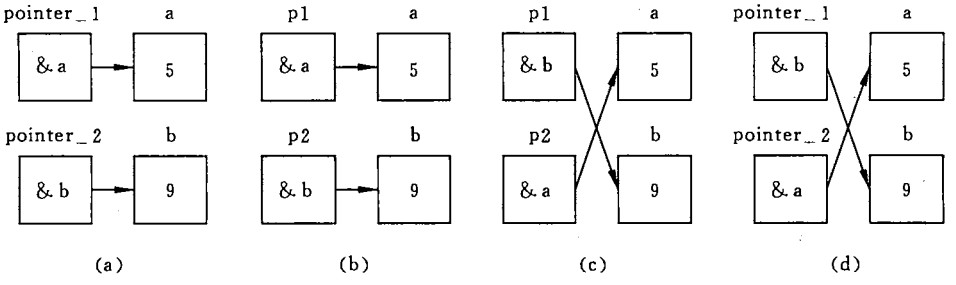


图 10-11

但是,这是办不到的,程序实际输出为"5,9"。问题出在第④步。C 语言中实参变量和形参变量之间的数据传递是单向的"值传递"方式。指针变量作函数参数也要遵循这一规则。不可能通过调用函数来改变实参指针变量的值,但可以改变实参指针变量所指变量的值。我们知道,函数的调用可以(而且只可以)得到一个返回值(即函数值),而运用指针变量作参数,可以得到多个变化了的值。如果不用指针变量是难以做到这一点的。

例 10.4 输入 a、b、c 这 3 个整数,按大小顺序输出。

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

```
void main()
  { void exchange(int *q1, int *q2, int *q3);
    int a,b,c, *p1, *p2, *p3;
    scanf("%d,%d,%d",&a,&b,&c);
    p1=&a;p2=&b;p3=&c;
    exchange(p1,p2,p3);
    printf("\n%d,%d,%d\n",a,b,c);
  }
    void exchange(int *q1, int *q2, int *q3)
  { void swap(int *pt1, int *pt2);
    if(*q1<*q2) swap(q1,q2);
    if(*q1<*q3) swap(q1,q3);
    if(*q2<*q3) swap(q2,q3);
  }
    void swap(int *pt1, int *pt2)
  { int temp;
    temp=*pt1;
    *pt1=*pt2;
    *pt2=temp;
  }
```

运行情况如下：

```
9,0,10↙
10,9,0
```

10.3 数组与指针

一个变量有地址，一个数组包含若干元素，每个数组元素都在内存中占用存储单元，它们都有相应的地址。指针变量既然可以指向变量，当然也可以指向数组元素（把某一元素的地址放到一个指针变量中）。所谓数组元素的指针就是数组元素的地址。

引用数组元素可以用下标法（如 a[3]），也可以用指针法，即通过指向数组元素的指针找到所需的元素。使用指针法能使目标程序质量高（占内存少，运行速度快）。

10.3.1 指向数组元素的指针

定义一个指向数组元素的指针变量的方法，与以前介绍的指向变量的指针变量相同。例如：

```
int a[10];      （定义 a 为包含 10 个整型数据的数组）
int *p;         （定义 p 为指向整型变量的指针变量）
```

应当注意，如果数组为 int 型，则指针变量的基类型也应为 int 型。下面是对该指针变量赋值：

```
p=&a[0];
```

把 a[0]元素的地址赋给指针变量 p。也就是使 p 指向 a 数组的第 0 号元素，见图 10-12。

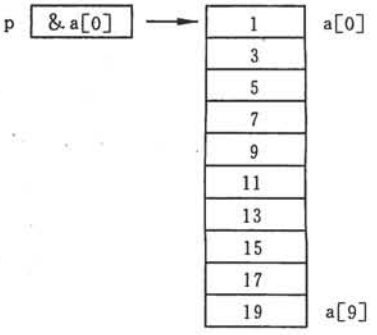


图 10-12

C 语言规定数组名(不包括形参数组名，形参数组并不占据实际的内存单元)代表数组中首元素(即序号为 0 的元素)的地址。因此，下面两个语句等价：

```
p=&a[0];
p=a;
```

注意数组名 a 不代表整个数组，上述“p=a;”的作用是“把 a 数组的首元素的地址赋给指针变量 p”，而不是“把数组 a 各元素的值赋给 p”。

在定义指针变量时可以对它赋予初值：

```
int *p=&a[0];
```

它等效于下面两行：

```
int *p;
p=&a[0];      /* 注意，不是 *p=&a[0]; */
```

当然定义时也可以写成

```
int *p=a;
```

它的作用是将 a 数组首元素(即 a[0])的地址赋给指针变量 p(而不是赋给 *p)。

10.3.2 通过指针引用数组元素

假设 p 已定义为一个指向整型数据的指针变量，并已给它赋了一个整型数组元素的地址，使它指向某一个数组元素。如果有以下赋值语句：

```
*p=1;
```

表示将 1 赋给 p 当前所指向的数组元素。

按 C 语言的规定：如果指针变量 p 已指向数组中的一个元素，则 p+1 指向同一数组中的下一个元素，而不是将 p 的值(地址)简单地加 1。例如，数组元素是 float 型，每个元素占 4 个字节，则 p+1 意味着使 p 的值(是地址)加 4 个字节，以使它指向下一元素。p+1所代表的地址实际上是 p+1×d，d 是一个数组元素所占的字节数(在 Turbo C++ 中，对 int 型，d=2；对 float 和 long 型，d=4；对 char 型，d=1。在 Visual C++ 6.0 中，对 int、long 和 float 型，d=4；对 char 型，d=1)。

如果 p 的初值为 &a[0]，则：

(1) p+i 和 a+i 就是 a[i]的地址，或者说，它们指向 a 数组的第 i 个元素，见图10-13。这里需要特别注意的是 a 代表数组首元素的地址，a+i 也是地址，它的计算方法同 p+i，即它的实际地址为 a+i×d。例如，p+9 和 a+9 的值是 &a[9]，它指向 a[9]，如图 10-13 所示。

(2) *(p+i)或*(a+i)是 p+i 或 a+i 所指向的数组元素，即 a[i]。例如，*(p+5)或*(a+5)就是 a[5]。即*(p+5)、*(a+5)、a[5]三者等价。实际上，在编译时，对数组元素 a[i]就是按*(a+i)处理的，即按数组首元素的地址加上相对位移量得到要找的元素的地址，然后找出该单元中的内容。若数组 a 的首元素的地址为 1000，设数组为 float 型，则 a[3]的地址是这样计算的：1000+3×4=1012，然后从 1012 地址所指向的 float 型单元取出元素的值，即 a[3]的值。可以看出，[ ]实际上是**变址运算符**，即将 a[i]按 a+i 计算地址，然后找出此地址单元中的值。

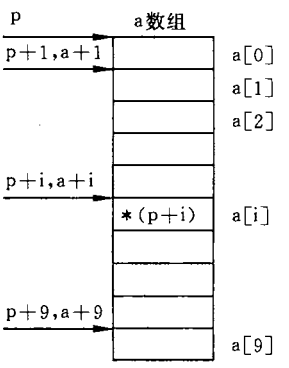


图 10-13

(3) 指向数组的指针变量也可以带下标，如 p[i]与*(p+i)等价。

根据以上叙述，引用一个数组元素，可以用：

(1) 下标法，如 a[i]形式；

(2) 指针法，如*(a+i)或*(p+i)。其中 a 是数组名，p 是指向数组元素的指针变量，其初值 p=a。

例 10.5 输出数组中的全部元素。

假设有一个 a 数组，整型，有 10 个元素。要输出各元素的值有 3 种方法。

(1) 下标法。

```
#include <stdio.h>
void main()
  {int a[10];
   int i;
   for(i=0;i<10;i++)
     scanf("%d",&a[i]);
   printf("\n");
   for(i=0;i<10;i++)
     printf("%d ",a[i]);
   printf("\n");
  }
```

(2) 通过数组名计算数组元素地址，找出元素的值。

```
#include <stdio.h>
void main()
  {int a[10];
   int i;
   for(i=0;i<10;i++)
     scanf("%d",&a[i]);
   printf("\n");
   for(i=0;i<10;i++)
```

```
    printf("%d ",*(a+i));
  printf("\n");
  }
```

(3) 用指针变量指向数组元素。

```
#include <stdio.h>
void main()
  { int a[10];
    int *p,i;
    for(i=0;i<10;i++)
      scanf("%d",&a[i]);
    printf("\n");
    for(p=a;p<(a+10);p++)
      printf("%d ",*p);
    printf("\n");
  }
```

以上 3 个程序的运行情况均如下：

```
1 2 3 4 5 6 7 8 9 0↙
1 2 3 4 5 6 7 8 9 0
```

对 3 种方法的比较：

① 例 10.5 的第①和第②种方法执行效率是相同的。C 编译系统是将 a[i]转换为 *(a+i)处理的，即先计算元素地址。因此用第①和第②种方法找数组元素费时较多。

② 第③种方法比第①、第②方法快，用指针变量直接指向元素，不必每次都重新计算地址，像 p++这样的自加操作是比较快的。这种有规律地改变地址值(p++)能大大提高执行效率。

③ 用下标法比较直观，能直接知道是第几个元素。例如，a[5]是数组中序号为 5 的元素(注意序号从 0 算起)。用地址法或指针变量的方法不直观，难以很快地判断出当前处理的是哪一个元素。例如，例 10.5 第③种方法所用的程序，要仔细分析指针变量 p 的当前指向，才能判断当前输出的是第几个元素。

在使用指针变量指向数组元素时，有以下几个问题要注意：

① 可以通过改变指针变量的值指向不同的元素。例如，上述第③种方法是用指针变量 p 来指向元素，用 p++使 p 的值不断改变从而指向不同的元素，这是合法的。如果不用 p 而使数组名 a 变化(例如，用 a++)行不行呢？假如将上述第③种方法中的程序的最后两行改为：

```
for(p=a;a<(p+10);a++)
    printf("%d",*a);
```

是不行的。因为数组名 a 代表数组首元素的地址，它是一个指针常量，它的值在程序运行期间是固定不变的。既然 a 是常量，所以 a++是无法实现的。

② 要注意指针变量的当前值。请看下面的例子。

例 10.6 通过指针变量输出 a 数组的 10 个元素。

有人编写出以下程序：

```
#include <stdio.h>
void main()
  { int *p,i,a[10];
    p=a;
    for(i=0;i<10;i++)
      scanf("%d",p++);
    printf("\n");
    for(i=0;i<10;i++,p++)
      printf("%d ",*p);
    printf("\n");
  }
```

这个程序乍看起来好像没有什么问题。有的人即使已被告知此程序有问题，还是找不出它有什么问题。我们先看一下运行情况：

```
1 2 3 4 5 6 7 8 9 0↙
22153 234 0 0 30036 25202 11631 8259 8237 28483
```

（在不同的环境中运行时显示的数据可能与上面的有所不同）

显然输出的数值并不是 a 数组中各元素的值。原因是指针变量的初始值为 a 数组首元素（即 a[0]）的地址（见图 10-14 中的①），但经过第一个 for 循环读入数据后，p 已指向 a 数组的末尾（见图 10-14 中②）。因此，在执行第二个 for 循环时，p 的起始值不是 &a[0] 了，而是 a＋10。由于执行第二个 for 循环时，每次要执行 p＋＋，因此 p 指向的是 a 数组下面的 10 个元素，而这些存储单元中的值是不可预料的。

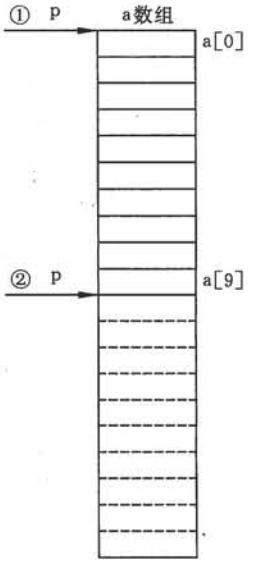


图 10-14

解决这个问题的办法是，只要在第二个 for 循环之前加一个赋值语句：

```
p=a;
```

使 p 的初始值回到 &a[0]，这样结果就对了。程序为：

```
#include <stdio.h>
void main()
  { int *p,i,a[10];
    p=a;
    for(i=0;i<10;i++)
      scanf("%d",p++);
    printf("\n");
    p=a;
    for(i=0;i<10;i++,p++)
```

```
        printf("%d ", *p);
    printf("\n");
}
```

运行情况如下：

```
1 2 3 4 5 6 7 8 9 0↙
1 2 3 4 5 6 7 8 9 0
```

③ 从上例可以看到，虽然定义数组时指定它包含 10 个元素，并用 p 指向某一数组元素，但是实际上指针变量 p 可以指向数组以后的内存单元。如果引用数组元素 a[10]，C 编译程序并不认为非法，系统把它按 *(a+10)处理，即先找出(a+10)的值(是一个地址)，然后找出它指向的单元的内容。这样做虽然是合法的(在编译时不出错)，但应避免出现这样的情况，这会使程序得不到预期的结果。这种错误比较隐蔽，初学者往往难以发现。在使用指针变量指向数组元素时，应切实保证指向数组中有效的元素。

④ 注意指针变量的运算。如果先使 p 指向数组 a 的首元素(即 p=a)，请分析：

(i) p++(或 p+=1)。使 p 指向下一元素，即 a[1]。若再执行 *p，则得到下一个元素 a[1]的值。

(ii) *p++。由于++和 * 同优先级，结合方向为自右而左，因此它等价于 *(p++)。作用是先得到 p 指向的变量的值(即 *p)，然后再使 p+1⇒p。

例 10.6 最后一个程序中最后一个 for 语句：

```
for(i=0;i<10;i++,p++)
    printf("%d", *p);
```

可以改写为

```
for(i=0;i<10;i++)
    printf("%d", *p++);
```

作用完全一样。它们的作用都是先输出 *p 的值，然后使 p 值加 1。这样下一次循环时，*p 就是下一个元素的值。

(iii) *(p++)与 *(++p)作用不同。前者是先取 *p 值，然后使 p 加 1。后者是先使 p 加 1，再取 *p。若 p 初值为 a(即 &a[0])，则 *(p++)为 a[0]，而 *(++p)为a[1]。

(iv) (*p)++表示 p 所指向的元素值加 1，如果 p=a，则(*p)++相当于(a[0])++，若 a[0]=3，则(*p)++(即(a[0])++)的值为 4。注意：是元素值加 1，而不是指针值加 1。

(v) 如果 p 当前指向 a 数组中第 i 个元素，则：

(p--)相当于 a[i--]，先对 p 进行“”运算，再使 p 自减。

*(++p)相当于 a[++i]，先使 p 自加，再作 * 运算。

*(--p)相当于 a[--i]，先使 p 自减，再作 * 运算。

将++和--运算符用于指针变量十分有效，可以使指针变量自动向前或向后移动，

指向下一个或上一个数组元素。例如，想输出 a 数组的 100 个元素，可以用下面的方法：

```
p=a;                                    p=a;
while(p<a+100)          或              while(p<a+100)
  printf("%d", *p++);                     {printf("%d", *p);p++;}
```

但如果不小心，很容易弄错。因此在用 *p++形式的运算时，一定要十分小心，弄清楚先取 p 值还是先使 p 加 1。

10.3.3 用数组名作函数参数

在第 8 章 8.7 节中介绍过可以用数组名作函数的参数。例如：

```
void main()                          void f(int arr[ ],int n)
  {void f(int arr[],int n);            {
   int array[10];                        ⋮
     ⋮                                 }
   f(array,10);
     ⋮
  }
```

array 为实参数组名，arr 为形参数组名。在 8.7 节已知，当用数组名作参数时，如果形参数组中各元素的值发生变化，实参数组元素的值随之变化。这是为什么？在学习指针以后，对此问题就容易理解了。

先看数组元素作实参时的情况。如果已定义一个函数，其原型为

```
void swap(int x,int y);
```

假设函数的作用是将两个形参(x,y) 的值交换，今有以下的函数调用：

```
swap(a[1],a[2]);
```

用数组元素 a[1]、a[2]作实参的情况与用变量作实参时一样，是“值传递”方式，将 a[1]和 a[2]的值单向传递给 x 和 y。当 x 和 y 的值改变时 a[1]和 a[2]的值并不改变。

再看用数组名作函数参数的情况。前已介绍，实参数组名代表该数组首元素的地址，而形参是用来接收从实参传递过来的数组首元素地址的。因此，形参应该是一个指针变量(只有指针变量才能存放地址)。实际上，C 编译都是将形参数组名作为指针变量来处理的。例如，上面给出的函数 f 的形参是写成数组形式的：

```
f(int arr[ ], int n)
```

但在编译时是将 arr 按指针变量处理的，相当于将函数 f 的首部写成

```
f(int *arr, int n)
```

以上两种写法是等价的。在该函数被调用时，系统会建立一个指针变量 arr，用来存放从主调函数传递过来的实参数组首元素的地址。如果在 f 函数中用 sizeof(arr)测定

arr 所占的字节数，结果为 2(用 Turbo C++ 时)或 4(用 Visual C++ 时)。这就证明了系统是把 arr 作为指针变量来处理的(指针变量在 Turbo C++ 中占 2 个字节，在 Visual C++ 中占 4 个字节)。

当 arr 接收了实参数组的首元素地址后，arr 就指向实参数组首元素，也就是指向 array[0]。因此，* arr 就是 array[0]。arr+1 指向 array[1]，arr+2 指向 array[2]，arr+3 指向 array[3]。也就是说，*(arr+1)、*(arr+2)、*(arr+3)分别是 array[1]、array[2]、array[3]。根据前面介绍过的知识，*(arr+i)和 arr[i]是无条件等价的。因此，在调用函数期间，arr[0]和 * arr 以及 array[0]都代表数组 array 序号为 0 的元素，依此类推，arr[3]、*(arr+3)、array[3]都代表 array 数组序号为 3 的元素，见图 10-15。这个道理与 10.2 节 10.2.3 段中的叙述是类似的。

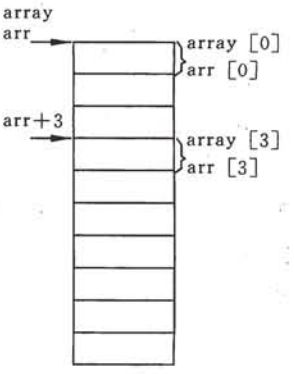


图 10-15

常用这种方法通过调用一个函数来改变实参数组的值。

下面把用变量名作为函数参数和用数组名作为函数参数做一比较，见表 10-1。

表 10-1 以变量名和数组名作为函数参数的比较

实参类型	变量名	数组名
要求形参的类型	变量名	数组名或指针变量
传递的信息	变量的值	实参数组首元素的地址
通过函数调用能否改变实参的值	不能	能

需要说明的是：C 语言调用函数时虚实结合的方法都是采用“值传递”方式，当用变量名作为函数参数时传递的是变量的值，当用数组名作为函数参数时，由于数组名代表的是数组首元素地址，因此传递的值是地址，所以要求形参为指针变量。

在用数组名作为函数实参时，既然实际上相应的形参是指针变量，为什么还允许使用形参数组的形式呢？这是因为在 C 语言中用下标法和指针法都可以访问一个数组(如果有一个数组 a，则 a[i]和 *(a+i)无条件等价)，用下标法表示比较直观，便于理解。因此许多人愿意用数组名作形参，以便与实参数组对应。从应用的角度看，用户可以认为有一个形参数组，它从实参数组那里得到起始地址，因此形参数组与实参数组共占同一段内存单元，在调用函数期间，如果改变了形参数组的值，也就是改变了实参数组的值。当然在主调函数中可以利用这些已改变的值。对 C 语言比较熟练的专业人员往往喜欢用指针变量作形参。

应该注意：实参数组名代表一个固定的地址，或者说是指针常量，但形参数组并不是一个固定的地址值，而是作为指针变量，在函数调用开始时，它的值等于实参数组首元素的地址，在函数执行期间，它可以再被赋值。例如：

```
void f(arr[ ],int n)
  { printf("%d\n", *arr);          /*输出 array[0]的值,*/
```

```
    arr=arr+3;
    printf("%d\n", *arr);              /*输出 array[3]的值,*/
  }
```

例 10.7 将数组 a 中 n 个整数按相反顺序存放,见图 10-16 示意。

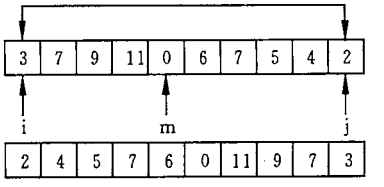


图 10-16

解此题的算法为:将 a[0]与 a[n−1]对换,再将 a[1]与 a[n−2]对换……直到将 a[int(n−1)/2]与 a[n−int((n−1)/2)−1]对换。今用循环处理此问题,设两个"位置指示变量"i 和 j,i 的初值为 0,j 的初值为 n−1。将 a[i]与 a[j]交换,然后使 i 的值加 1,j 的值减 1,再将 a[i]与 a[j]对换,直到 i=(n−1)/2 为止。

程序如下:

```
#include <stdio.h>
void main()
 { void inv(int x[],int n);
   int i,a[10]={3,7,9,11,0,6,7,5,4,2};
   printf("The original array:\n");
   for(i=0;i<10;i++)
     printf("%d,",a[i]);
   printf("\n");
   inv(a,10);
   printf("The array has been inverted:\n");
   for(i=0;i<10;i++)
     printf("%d,",a[i]);
   printf("\n");
 }
void inv(int x[],int n)          /*形参 x 是数组名*/
{int temp,i,j,m=(n-1)/2;
 for(i=0;i<=m;i++)
  {j=n-1-i;
   temp=x[i];x[i]=x[j];x[j]=temp;
  }
 return;
}
```

运行情况如下:

```
The original array:
3,7,9,11,0,6,7,5,4,2
The array has been inverted:
2,4,5,7,6,0,11,9,7,3
```

主函数中数组名为 a,对各元素赋予初值。函数 inv 中的形参数组名为 x。在 inv 函

数中可以不指定数组元素的个数。因为形参数组名实际上是一个指针变量，并不是真正地开辟一个数组空间(定义实参数组时必须指定数组大小，因为要开辟相应的存储空间)。函数形参 n 用来接收实际上需要处理的元素的个数。如果在 main 函数中有函数调用语句"inv(a,10);"，表示要求对 a 数组的 10 个元素实行题目要求的颠倒排列。如果改为"inv(a,5);"，则表示要求将 a 数组的前 5 个元素实行颠倒排列，此时，函数 inv 只处理 5 个数组元素。函数 inv 中的 m 是 i 值的上限，当 i≤m 时，循环继续执行；当 i>m 时，则结束循环过程。例如，若 n=10，则 m=4，最后一次 a[i]与 a[j]的交换是 a[4]与 a[5]交换。

对这个程序可以作一些改动。将函数 inv 中的形参 x 改成指针变量。实参为数组名 a，即数组 a 首元素的地址，将它传给形参指针变量 x，这时 x 就指向 a[0]。x+m 是 a[m]元素的地址。设 i 和 j 以及 p 都是指针变量，用它们指向有关元素。i 的初值为 x，j 的初值为 x+n−1，见图 10-17。使 *i 与 *j 交换就是使 a[i]与 a[j]交换。

程序如下：

```
#include <stdio.h>
void main()
 {void inv(int *x,int n);
  int i,a[10]={3,7,9,11,0,6,7,5,4,2};
  printf("The original array:\n");
  for(i=0;i<10;i++)
    printf("%d,",a[i]);
  printf("\n");
  inv(a,10);
  printf("The array has been inverted:\n");
  for(i=0;i<10;i++)
    printf("%d,",a[i]);
  printf("\n");
 }

void inv(int *x,int n)              /*形参 x 为指针变量*/
 {int *p,temp,*i,*j,m=(n-1)/2;
  i=x;j=x+n-1;p=x+m;
  for(;i<=p;i++,j--)
      {temp=*i;*i=*j;*j=temp;}
  return;
 }
```

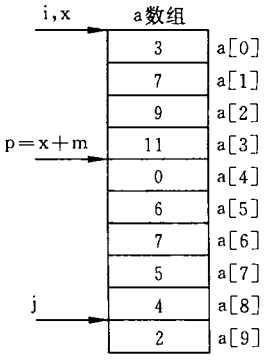


图 10-17

运行情况与前一程序相同。

归纳起来，如果有一个实参数组，要想在函数中改变此数组中的元素的值，实参与形参的对应关系有以下 4 种情况。

(1) 形参和实参都用数组名，例如：

```
void main()                    void f(int x[ ],int n)
 {int a[10];                   {
    ⋮                              ⋮
```

```
    f(a,10);                    }
      ⋮
}
```

由于形参数组名接收了实参数组首元素的地址，因此可以认为在函数调用期间，形参数组与实参数组共用一段内存单元，这种形式比较好理解，见图 10-18。例 10.7 第一个程序即属此情况。

(2) 实参用数组名，形参用指针变量。例如：

```
void main()                     void f(int * x,int n)
 {int a[10];                     {
    ⋮                               ⋮
  f(a,10);                       }
    ⋮
}
```

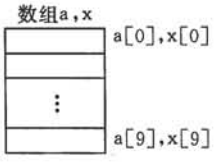


图 10-18

实参 a 为数组名，形参 x 为指向整型变量的指针变量，函数开始执行时，x 指向 a[0]，即 x=&a[0]，见图 10-19。通过 x 值的改变，可以指向 a 数组的任一元素。例 10.7 的第二个程序就属此类。

(3) 实参形参都用指针变量。例如：

```
void main()                     void f(int * x,int n)
 {int a[10], * p=a;              {
    ⋮                               ⋮
  f(p,10);                       }
    ⋮
}
```

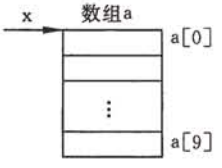


图 10-19

实参 p 和形参 x 都是指针变量。先使实参指针变量 p 指向数组 a，p 的值是 &a[0]。然后将 p 的值传给形参指针变量 x，x 的初始值也是 &a[0]，见图 10-20。通过 x 值的改变可以使 x 指向数组 a 的任一元素。

(4) 实参为指针变量，形参为数组名。例如：

```
void main()                     void f(int x[ ],int n)
 {int a[10], * p=a;              {
    ⋮                               ⋮
  f(p,10);                       }
    ⋮
}
```

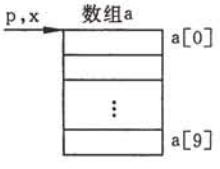


图 10-20

实参 p 为指针变量，它指向 a[0]。形参为数组名 x，编译系统把 x 作为指针变量处理，今将 a[0]的地址传给形参 x，使指针变量 x 指向 a[0]。也可以理解为形参数组 x 和 a 数组共用同一段内存单元，见图 10-21。在函数执行过程中可以使 x[i]的值发生变化，而 x[i]就是 a[i]。这样，主函数就可以使用变化了的数组元素值。例 10.7 的程序可以改写为例 10.8。

例 10.8 用实参指针变量改写例 10.7。

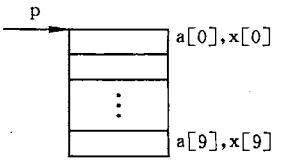


图 10-21

```
#include <stdio.h>
void main()
  {void inv(int * x,int n);
   int i,arr[10], * p=arr;
   printf("The original array:\n");
   for(i=0;i<10;i++,p++)
     scanf("%d",p);
   printf("\n");
   p=arr;
   inv(p,10);                    /* 实参为指针变量 */
   printf("The array has been inverted:\n");
   for(p=arr;p<arr+10;p++)
     printf("%d ", * p);
   printf("\n");
  }
void inv(int * x,int n)
  {int * p,m,temp, * i, * j;
   m=(n-1)/2;
   i=x;j=x+n-1;p=x+m;
   for(;i<=p;i++,j--)
     {temp= * i; * i= * j; * j=temp;}
   return;
  }
```

注意,上面的 main 函数中的指针变量 p 是有确定值的。如果在 main 函数中不设数组,只设指针变量,就会出错,假如把程序修改如下:

```
#include <stdio.h>
void main()
  { int i, * arr;
    printf("The original array:\n");
    for(i=0;i<10;i++)
       scanf("%d",arr+i);
    printf("\n");
    inv(arr,10);                  /* 实参为指针变量,但未被赋值 */
    printf("The array has been inverted:\n");
    for(i=0;i<10;i++)
        printf("%d", * (arr+i));
    printf("\n");
  }
```

编译时出错,原因是指针变量 arr 没有确定值,谈不上指向哪个变量。下面的使用是不正确的:

```
void main()                    f(x[ ],int n)
 {int * p;                      {
  f(p,10);                       ⋮
    ⋮                           }
 }
```

应注意，如果用指针变量作实参，必须先使指针变量有确定值，指向一个已定义的单元。

以上 4 种方法，实质上都是地址的传递。其中(3)、(4)两种只是形式上不同，实际上形参都是使用指针变量。

例 10.9 用选择法对 10 个整数按由大到小顺序排序。

程序如下：

```
#include <stdio.h>
void main()
 { void sort(int x[ ],int n);
   int * p,i,a[10];
   p=a;
   for(i=0;i<10;i++)
     scanf("%d",p++);
   p=a;
   sort(p,10);
   for(p=a,i=0;i<10;i++)
     {printf("%d ", * p);p++;}
 }

void sort(int x[],int n)
  { int i,j,k,t;
    for(i=0;i<n-1;i++)
      {k=i;
       for(j=i+1;j<n;j++)
         if(x[j]>x[k]) k=j;
       if(k!=i)
         {t=x[i];x[i]=x[k];x[k]=t;}
      }
  }
```

为了便于理解，函数 sort 中用数组名作为形参，用下标法引用形参数组元素，这样的程序很容易看懂。当然也可以改用指针变量，这时 sort 函数的首部可以改为

```
sort(int * x,int n)
```

其他不改，程序运行结果不变。可以看到，即使在函数 sort 中将 x 定义为指针变量，在函数中仍可用 x[i]、x[k]这样的形式表示数组元素，它就是 x+i 和 x+k 所指的数组元素。

上面的 sort 函数等价于：

```
void sort(int * x,int n)
 { int i,j,k,t;
   for(i=0;i<n-1;i++)
     { k=i;
       for(j=i+1;j<n;j++)
         if(*(x+j)>*(x+k)) k=j;
       if(k!=i)
         {t=*(x+i);*(x+i)=*(x+k);*(x+k)=t;}
     }
 }
```

请读者自己理解消化程序。

10.3.4 多维数组与指针

用指针变量可以指向一维数组中的元素，也可以指向多维数组中的元素。但在概念上和使用上，多维数组的指针比一维数组的指针要复杂一些。

1. 多维数组元素的地址

为了说清楚指向多维数组元素的指针，先回顾一下多维数组的性质。今以二维数组为例，设有一个二维数组 a，它有 3 行 4 列。

它的定义为

```
int a[3][4]={{1,3,5,7},{9,11,13,15},{17,19,21,23}};
```

a 是一个数组名。a 数组包含 3 行，即 3 个元素：a[0]、a[1]、a[2]。而每一个元素又是一个一维数组，它包含 4 个元素（即 4 个列元素），例如，a[0]所代表的一维数组又包含 4 个元素：a[0][0]、a[0][1]、a[0][2]、a[0][3]，见图 10-22。可以认为二维数组是“数组的数组”，即二维数组 a 是由 3 个一维数组所组成的。

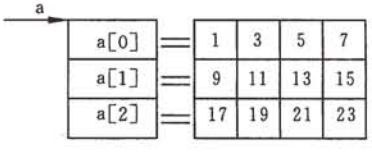


图 10-22

从二维数组的角度来看，a 代表二维数组首元素的地址，现在的首元素不是一个简单的整型元素，而是由 4 个整型元素所组成的一维数组，因此 a 代表的是首行（即第 0 行）的首地址。a+1 代表第 1 行的首地址。如果二维数组的首行的首地址为 2000，则在 Turbo C 中，a+1 为 2008，因为第 0 行有 4 个整型数据，因此 a+1 的含义是 a[1]的地址，即 a+4×2=2008。a+2 代表 a[2]的首地址，它的值是 2016，见图 10-23。

a[0]、a[1]、a[2]既然是一维数组名，而 C 语言又规定了数组名代表数组首元素地址，因此 a[0]代表一维数组 a[0]中第 0 列元素的地址，即 &a[0][0]。a[1]的值是 &a[1][0]，a[2]的值是 &a[2][0]。

请考虑 0 行 1 列元素的地址怎么表示？a[0]为一维数组名，该一维数组中序号为 1 的元素的地址显然应该用 a[0]+1 来表示，见图 10-24。此时“a[0]+1”中的 1 代表 1 个

列元素的字节数，即 2 个字节。今 a[0]的值是 2000，a[0]+1 的值是 2002(而不是 2008)。这是因为现在是在一维数组范围内讨论问题的，正如有一个一维数组 x，x+1 是其第 1 个元素 x[1]的地址一样。a[0]+0、a[0]+1、a[0]+2、a[0]+3 分别是 a[0][0]、a[0][1]、a[0][2]、a[0][3]元素的地址(即 &a[0][0]、&a[0][1]、&a[0][2]、&a[0][3])。

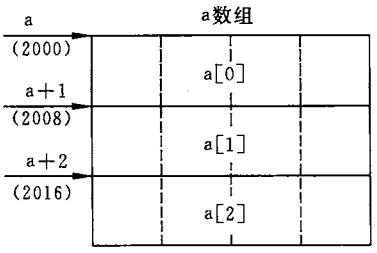


图 10-23

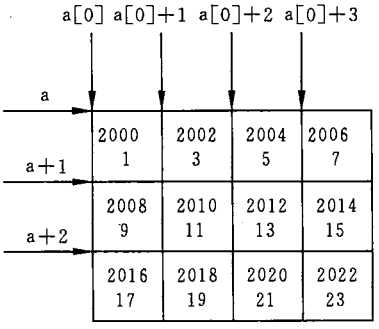


图 10-24

前已述及，a[0]和 *(a+0)等价，a[1]和 *(a+1)等价，a[i]和 *(a+i)等价。因此，a[0]+1和 *(a+0)+1 都是 &a[0][1](即图 10-24 中的 2002)。a[1]+2 和 *(a+1)+2 的值都是 &a[1][2](即图中的 2012)。请注意不要将 *(a+1)+2 错写成 *(a+1+2)，后者变成 *(a+3)了，相当于 a[3]。

进一步分析，欲得到 a[0][1]的值，用地址法怎么表示呢？既然 a[0]+1 和 *(a+0)+1 是 a[0][1]的地址，那么，*(a[0]+1)就是 a[0][1]的值。同理，*(*(a+0)+1)或 *(*a+1)也是 a[0][1]的值。*(a[i]+j)或 *(*(a+i)+j)是 a[i][j]的值。务请记住 *(a+i)和 a[i]是等价的。

有必要对 a[i]的性质作进一步说明。a[i]从形式上看是 a 数组中序号为 i 的元素。如果 a 是一维数组名，则 a[i]代表 a 数组序号为 i 的元素所占的内存单元的内容。a[i]是有物理地址的，是占内存单元的。但如果 a 是二维数组，则 a[i]是代表一维数组名。它只是一个地址，并不代表某一元素的值(如同一维数组名只是一个指针常量一样)。a、a+i、a[i]、*(a+i)、*(a+i)+j、a[i]+j 都是地址。而 *(a[i]+j)和 *(*(a+i)+j)是二维数组元素 a[i][j]的值，见表 10-2。

表 10-2 数组 a 的性质

表示形式	含义	地址
a	二维数组名，指向一维数组 a[0]，即 0 行首地址	2000
a[0]，*(a+0)，*a	0 行 0 列元素地址	2000
a+1，&a[1]	1 行首地址	2008
a[1]，*(a+1)	1 行 0 列元素 a[1][0]的地址	2008
a[1]+2，*(a+1)+2，&a[1][2]	1 行 2 列元素 a[1][2]的地址	2012
(a[1]+2)，(*(a+1)+2)，a[1][2]	1 行 2 列元素 a[1][2]的值	元素值为 13

有些读者可能不理解为什么 a+1 和 *(a+1)都是 2008 呢？他们想“a+1(是地址)和 *(a+1)(是内容)怎么都是同一个值呢”？的确，二维数组中有些概念比较复杂难懂，要反复思考。

首先说明，a+1 是二维数组 a 中序号为 1 的行的首地址(序号从 0 起算)，而 *(a+1)并不是 a+1 单元的内容(值)，因为 a+1 并不是一个变量的存储单元，也就谈不上它的内容了。 *(a+1)就是 a[1]，而 a[1]是一维数组名，所以也是地址，它指向 a[1][0]。以上各种形式都是地址计算的不同表示。

为了说明这个容易搞混的问题，可以用军训中排队点名的例子来说明。班长逐个检查本班战士是否在队列中，班长每移动一步，走过一个战士。而排长点名时只检查本排各班是否到齐。排长从第 0 班的起始位置走到第 1 班的起始位置，看来只走了一步，但实际上它跳过了 10 个战士。这相当于 a+1(见图 10-25)。班长面对的是战士，排长面对的是班，班长相当于列指针，排长相当于行指针。

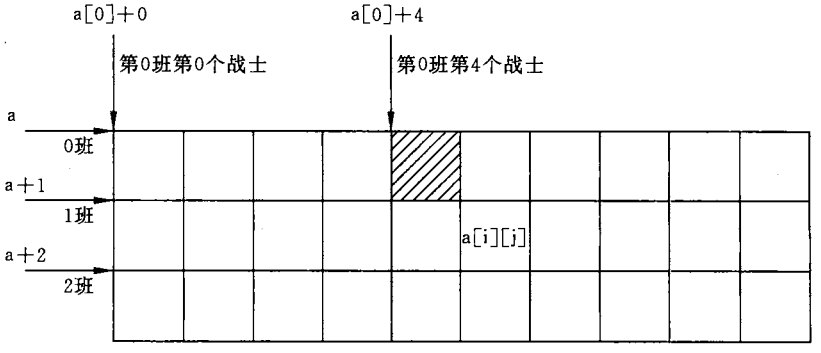


图 10-25

为了找到某一班内某一个战士，必须给两个参数，即第 i 班第 j 个战士，先找到第 i 班，然后由该班班长在本班范围内找第 j 个战士。这个战士的位置就是 a[i]+j(这是一个地址)。开始时班长面对第 0 个战士。注意，排长和班长的初始位置是相同的(如图 10-24 中的 a 和 a[0]都是 2000)，但它们的“指向”是不同的。排长“指向”班，在图 10-25 中是纵向管理，他走一步就跳过 1 个班，而班长“指向”战士，在图上是横向管理，走一步只是指向下一个战士。二维数组 a 相当于排长，每一行(即一维数组 a[0]、a[1]、a[2])相当于班长，每一行中的元素(如 a[1][2])相当于战士。

a+1 与 a[0]+1 是不同的，a+1 是序号为 1 的行的首地址，a+1 指向序号为 1 的行(相当于排长走到第 1 班的开头)，而 *(a+1)或 a[1]或 a[1]+0 都指向 1 行 0 列元素(相当于第 1 班第 0 个战士)，二者地址虽相同，但含义不同了。a、a[0]的值虽然相同(等于 2000)，但是由于指针的类型不同(相当于排长和班长面对的对象一样，a 是指向一维数组，a[0]指向 a[0][0]元素)，因此，对这些指针进行加 1 的运算，得到的结果是不同的。

再一次强调，二维数组名(如 a)是指向行的。因此 a+1 中的“1”代表一行中全部元素所占的字节数(图 10-24 表示为 8 个字节)。一维数组名(如 a[0]、a[1])是指向列元素的。a[0]+1 中的 1 代表一个元素所占的字节数(图 10-24 表示为 2 个字节)。在指向行

的指针前面加一个 *，就转换为指向列的指针。例如，a 和 a+1 是指向行的指针，在它们前面加一个 * 就是 * a 和 *(a+1)，它们就成为指向列的指针，分别指向 a 数组 0 行 0 列的元素和 1 行 0 列的元素。反之，在指向列的指针前面加 &，就成为指向行的指针。例如 a[0] 是指向 0 行 0 列元素的指针，在它前面加一个 &，得 &a[0]，由于 a[0] 与 *(a+0) 等价，因此 &a[0] 与 & * a 等价，也就是与 a 等价，它指向二维数组的 0 行。

不要把 &a[i]简单地理解为 a[i]单元的物理地址，因为并不存在 a[i]这样一个实际的变量。它只是一种地址的计算方法，能得到第 i 行的首地址，&a[i]和 a[i]的值是一样的，但它们的含义是不同的。&a[i]或 a+i 指向行，而 a[i]或 *(a+i)指向列。当列下标 j 为 0 时，&a[i]和 a[i](即 a[i]+j)值相等，即它们具有同一地址值，但应注意它们所指向的对象是不同的，即指针的类型是不同的。

*(a+i)只是 a[i]的另一种表示形式，不要简单地认为 *(a+i)是“a+i 所指单元中的内容”。在一维数组中 a+i 所指的是一个数组元素的存储单元，在该单元中有具体值，上述说法是正确的。而对二维数组，a+i 不是指向具体存储单元而指向行。在二维数组中，a+i、a[i]、*(a+i)、&a[i]、&a[i][0]的值相等，即它们是同一地址值。请读者仔细琢磨其概念。

请分析下面的程序，以加深对上面叙述的理解。

例 10.10 输出二维数组有关的值。

```
#include <stdio.h>
#define FORMAT "%d,%d\n"
void main()
 { int a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};
   printf(FORMAT,a, * a);
   printf(FORMAT,a[0], *(a+0));
   printf(FORMAT,&a[0],&a[0][0]);
   printf(FORMAT,a[1],a+1);
   printf(FORMAT,&a[1][0], *(a+1)+0);
   printf(FORMAT,a[2], *(a+2));
   printf(FORMAT,&a[2],a+2);
   printf(FORMAT,a[1][0], *( *(a+1)+0));
 }
```

某一次运行结果如下：

```
158,158                    (0 行首地址和 0 行 0 列元素地址)
158,158                    (0 行 0 列元素地址)
158,158                    (0 行首地址和 0 行 0 列元素地址)
166,166                    (1 行 0 列元素地址和 1 行首地址)
166,166                    (1 行 0 列元素地址)
174,174                    (2 行 0 列元素地址)
174,174                    (2 行首地址)
9,9                        (1 行 0 列元素的值)
```

a是二维数组名，代表数组首行的起始地址。*a相当于*(a+0)，即a[0]，它是0行0列元素地址(本次程序运行时输出a，a[0]和*a的值都是158，都是地址。请注意：每次编译分配的地址是不同的)。a是指向一维数组(即指向行)的指针，*a是指向列元素的指针，指向0行0列元素，**a是0行0列元素的值。同样，a+1是1行的首地址，但不能企图用*(a+1)得到a[1][0]的值，而应该用*(*(a+1))求a[1][0]元素的值。*(*(a+1))可以写成**(a+1)。

2. 指向多维数组元素的指针变量

在了解上面的概念后，可以用指针变量指向多维数组的元素。

(1) 指向数组元素的指针变量

例10.11 用指针变量输出二维数组元素的值。

```
#include <stdio.h>
void main()
  { int a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};
    int *p;
    for(p=a[0];p<a[0]+12;p++)
      { if((p-a[0])%4==0) printf("\n");
        printf("%4d",*p);
      }
    printf("\n");
  }
```

运行结果如下：

```
 1   3   5   7
 9  11  13  15
17  19  21  23
```

p是一个指向整型变量的指针变量，它可以指向一般的整型变量，也可以指向整型的数组元素。每次使p值加1，使p指向下一元素。if语句的作用是使输出4个数据后换行。如果读者对p的值还缺乏具体概念的话，可以把p的值(即数组元素的地址)输出。可将程序最后两个语句改为

```
printf("addr=%o,value=%2d\n",p,*p);
```

在Turbo C++环境下某一次运行时输出如下：

```
addr=236,value=1
addr=240,value=3
addr=242,value=5
addr=244,value=7
addr=246,value=9
addr=250,value=11
addr=252,value=13
```

```
addr=254,value=15
addr=256,value=17
addr=260,value=19
addr=262,value=21
addr=264,value=23
```

注意地址是以八进制数表示的(输出格式符为%o)。

上例是顺序输出数组中各元素之值,比较简单。如果要输出某个指定的数值元素(例如 a[1][2]),则应事先计算该元素在数组中的相对位置(即相对于数组起始位置的相对位移量)。计算 a[i][j]在数组中的相对位置的计算公式为

```
i*m+j
```

其中 m 为二维数组的列数(二维数组大小为 n×m)。例如,对上述 3×4 的二维数组,它的 2 行 3 列元素 a[2][3]对 a[0][0]的相对位移量为 2*4+3=11 元素。如果开始时使指针变量 p 指向 a[0][0],为了得到 a[2][3]的值,可以用 *(p+2*4+3)表示。(p+11)是 a[2][3]的地址。a[i][j]的地址为 &a[0][0]+i*m+j。

下面来说明上述(&a[0][0]+i*m+j)中的 i*m+j 公式的含义。从图 10-26 可以看到在 a[i][j]元素之前有 i 行元素(每行有 m 个元素),在 a[i][j]所在行,a[i][j]的前面还有 j 个元素,因此 a[i][j]之前共有 i×m+j 个元素。例如,a[2][3]的前面有两行(共 2×4=8 个)元素,在它本行内还有 3 个元素在它前面,故共有 8+3=11 个元素在它之前。可用 p+11 表示其相对位置。

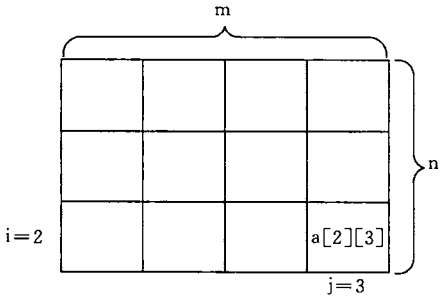


图 10-26

可以看到,C 语言规定数组下标从 0 开始,对计算上述相对位置比较方便,只要知道 i 和 j 的值,就可以直接用 i×m+j 公式计算出 a[i][j]相对于数组开头的相对位置。如果规定下标从 1 开始(如 FORTRAN 语言),则为计算 a[i][j]的相对位置所用的公式就要改为

```
(i-1)×m+(j-1)
```

这就增加了计算的工作量,而且不直观。

(2) 指向由 m 个元素组成的一维数组的指针变量

上例的指针变量 p 是用"int *p;"定义的,它是指向整型数据的,p+1 所指向的元素是 p 所指向的元素的下一元素。

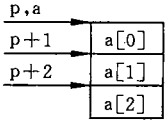


图 10-27

可以改用另一方法,使 p 不是指向整型变量,而是指向一个包含 m 个元素的一维数组。这时,如果 p 先指向 a[0](即 p=&a[0]),则 p+1 不是指向 a[0][1],而是指向 a[1],p 的增值以一维数组的长度为单位,见图 10-27。

例 10.12 输出二维数组任一行任一列元素的值。

```
#include <stdio.h>
void main()
  { int a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};
    int (*p)[4],i,j;
    p=a;
    scanf("i=%d,j=%d",&i,&j);
    printf("a[%d,%d]=%d\n",i,j,*(*(p+i)+j));
  }
```

运行情况如下：

```
i=1,j=2↙                    (本行为键盘输入)
a[1,2]=13
```

注意应输入“i=1,j=2”，以与 scanf 函数中指定的字符串相对应。

程序第 4 行中“int(*p)[4]”表示 p 是一个指针变量，它指向包含 4 个整型元素的一维数组。注意 *p 两侧的括号不可缺少，如果写成 *p[4]，由于方括号[ ]运算级别高，因此 p 先与[4]结合，p[4]是定义数组的形式，然后再与前面的 * 结合，*p[4]就是指针数组(见 10.7 节)。有的读者感到“(*p)[4]”这种形式不好理解。可以对下面二者做比较：

① int a[4]；(a 有 4 个元素，每个元素为整型)

② int (*p)[4]；

第②种形式表示 *p 有 4 个元素，每个元素为整型。也就是 p 所指的对象是有 4 个整型元素的数组，即 p 是指向一维数组的指针，见图 10-28。应该记住，此时 p 只能指向一个包含 4 个元素的一维数组，p 的值就是该一维数组的起始地址。p 不能指向一维数组中的某一元素。请务必注意指针变量的类型。

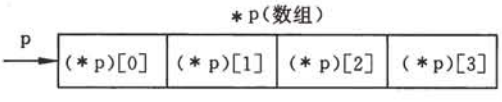


图 10-28

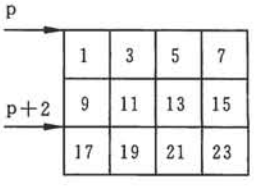


图 10-29

程序中的 p+i 是二维数组 a 的 i 行的起始地址(由于 p 是指向一维数组的指针变量，因此 p 加 1，就指向下一行)，见图 10-29。语分析 *(p+2)+3 是什么？由于 p=a，因此 *(p+2) 就是 a[2]，*(p+2)+3 就是 a[2]+3，而 a[2] 的值是 a 数组中 2 行 0 列元素 a[2][0]的地址(即 &a[2][0])，因此 *(p+2)+3 就是 a 数组 2 行 3 列元素的地址，这是指向列元素的指针，由此不难理解：*(*(p+2)+3)是 a[2][3]的值。

有的读者可能会想，*(p+2)是 a 数组 2 行 0 列元素的地址，而 p+2 是 a 数组 2 行起始地址，二者的值相同，*(p+2)+3 能否写成(p+2)+3 呢？显然不行。不能作简单的数值替换。(p+2)+3 就成了(p+5)了，是 a 数组 5 行的起始地址了。要注意指针变量的类型，由于 p 被定义为指向一维数组的指针变量，因此“*(p+2)+3”括号中的 2 是以一维数组的长度为单位的，即 p 每加 1，地址就增加 8 个字节(4 个元素，每个元素 2 个

字节),而“*(p+2)+3”括号外的数字3,不是以p所指向的一维数组为长度单位的。由于经过*(p+2)的运算,得到a[2],即&a[2][0],它已经转化为指向列元素的指针了,因此加3是以元素的长度为单位的,加3就是加(3×2)个字节。虽然p+2和*(p+2)具有相同的值,但由于它们所指向的对象的长度不同,因此(p+2)+3和*(p+2)+3的值就不相同了。这和上一节所叙述的概念是一致的。

3. 用指向数组的指针作函数参数

一维数组名可以作为函数参数传递,多维数组名也可作函数参数传递。在用指针变量作形参以接受实参数组名传递来的地址时,有两种方法:①用指向变量的指针变量;②用指向一维数组的指针变量。

例10.13 有一个班,3个学生,各学4门课,计算总平均分数以及第n个学生的成绩。

这个题目是很简单的。只是为了说明用指向数组的指针作函数参数而举的例子。用函数average求总平均成绩,用函数search找出并输出第i个学生的成绩。

程序如下:

```
#include <stdio.h>
void main()
  { void average(float *p,int n);
    void search(float (*p)[4],int n);
    float score[3][4]={{65,67,70,60},{80,87,90,81},{90,99,100,98}};
    average(*score,12);                    /*求12个分数的平均分*/
    search(score,2);                       /*求序号为2的学生的成绩*/
  }

void average(float *p,int n)
  { float *p_end;
    float sum=0,aver;
    p_end=p+n-1;
    for(;p<=p_end;p++)
      sum=sum+(*p);
    aver=sum/n;
    printf("average=%5.2f\n",aver);
  }

void search(float (*p)[4],int n) /* p是指向具有4个元素的一维数组的指针 */
  { int i;
    printf("the score of No.%d are:\n",n);
    for(i=0;i<4;i++)
      printf("%5.2f ",*(*(p+n)+i));
  }
```

程序运行结果如下:

```
average=82.25
```

```
the score of No. 2 are:
90.00 99.00 100.00 98.00
```

在 main 函数中，先调用 average 函数以求总平均值。在函数 average 中形参 p 被声明为指向一个 float 型变量的指针变量。实参用 * score，即 score[0]，也就是 &score[0][0]，即 score [0][0]的地址。把 score [0][0]的地址传给 p，使 p 指向 score [0][0] 。然后使 p 先后指向二维数组的各个元素，p 每加 1 就改为指向 score 数组的下一个元素，见图 10-30。形参 n 代表需要求平均值的元素的个数，实参 12 表示要求 12 个元素值的平均值。p_end 是最后一个元素的地址。sum 是累计总分，aver 是平均值。在函数中输出 aver 的值，函数无需返回值。

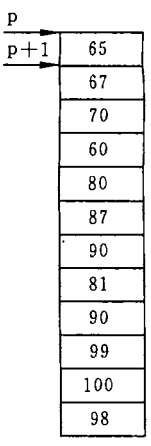


图 10-30

函数 search 的形参 p 不是指向一般变量的指针变量，而是指向包含 4 个元素的一维数组的指针变量。实参传给形参 n 的值为 2，即找序号为 2 的学生的成绩(3 个学生的序号分别为 0、1、2)。函数调用开始时，将实参 score 的值(代表该数组 0 行起始地址)传给 p，使 p 也指向 score[0]。p+n 是 score[n]的起始地址，*(p+n)+i 是 score[n][i]的地址，*(*(p+n)+i)是 score[n][i]的值。现在 n=2，i 由 0 变到 3，for 循环输出 score[2][0]到 score[2][3]的值。

例 10.14 在上题基础上，查找有一门以上课程不及格的学生，输出他们的全部课程的成绩。

程序如下：

```
#include <stdio.h>
void main()
  { void search(float (*p)[4],int n);              /*函数声明*/
    float score[3][4]={{65,57,70,60},{58,87,90,81},{90,99,100,98}};
    search(score,3);
  }

void search(float (*p)[4],int n)
  { int i,j,flag;
    for(j=0;j<n;j++)
      { flag=0;
        for(i=0;i<4;i++)
          if(*(*(p+j)+i)<60) flag=1;
        if(flag==1)
          { printf("No. %d fails,his scores are:\n",j+1);
            for(i=0;i<4;i++)
              printf("%5.1f",*(*(p+j)+i));
            printf("\n");
          }
      }
  }
```

程序运行结果如下：

```
No. 1 fails,his scores are:
65.0 57.0 70.0 60.0
No. 2 fails,his scores are:
58.0 87.0 90.0 81.0
```

在函数 search 中，flag 是作为标志不及格的变量。先使 flag＝0，若发现某一学生有一门不及格，则使 flag＝1。最后用 if 语句检查 flag，如为 1，则表示该学生有不及格的记录，输出其全部课程成绩。变量 j 代表学生号，i 代表课程号。

通过指针变量存取数组元素速度快，且程序简明。用指针变量作形参，所处理的数组大小可以变化。因此数组与指针常常是紧密联系的，使用熟练的话可以使程序质量提高，且编写程序方便灵活。

10.4 字符串与指针

10.4.1 字符串的表示形式

在 C 程序中，可以用两种方法访问一个字符串。

(1) 用字符数组存放一个字符串，然后输出该字符串。

例 10.15 定义一个字符数组，对它初始化，然后输出该字符串。

```
#include <stdio.h>
void main()
  { char string[]="I love China!";
    printf("%s\n",string);
  }
```

运行时输出：

```
I love China!
```

和前面介绍的数组属性一样，string 是数组名，它代表字符数组的首元素的地址(见图 10-31)。string[4]代表数组中序号为 4 的元素(它的值是字母 v)，实际上 string[4]就是 *(string＋4)，string＋4 是一个地址，它指向字符“v”。

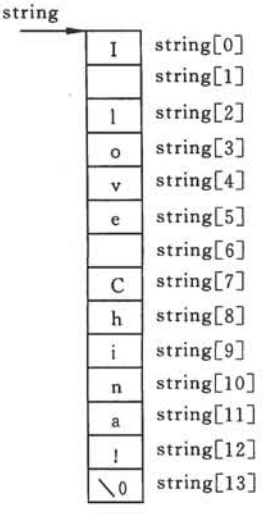


图 10-31

(2) 用字符指针指向一个字符串。

可以不定义字符数组，而定义一个字符指针。用字符指针指向字符串中的字符。

例 10.16 定义字符指针。

```
#include <stdio.h>
void main()
 { char *string="I love China!";
```

```
    printf("%s\n",string);
}
```

在这里没有定义字符数组，在程序中定义了一个字符指针变量 string，用字符串常量“I love China!”对它初始化。C 语言对字符串常量是按字符数组处理的，在内存中开辟了一个字符数组用来存放该字符串常量。对字符指针变量 string 初始化，实际上是把字符串第 1 个元素的地址（即存放字符串的字符数组的首元素地址）赋给 string（见图 10-32）。有人误认为 string 是一个字符串变量，以为在定义时把“I love China!”这几个字符赋给该字符串变量，这是不对的。定义 string 的部分：

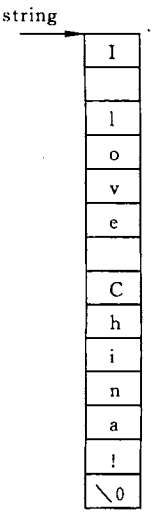


图 10-32

```
char * string="I love China!";
```

等价于下面两行：

```
char * string;
string="I love China!";
```

可以看到 string 被定义为一个指针变量，指向字符型数据，请注意它只能指向一个字符变量或其他字符类型数据，不能同时指向多个字符数据，更不是把“I love China!”这些字符存放到 string 中（指针变量只能存放地址），也不是把字符串赋给 * string。只是把“I love China!”的第 1 个字符的地址赋给指针变量 string。不要认为上述定义行等价于

```
char * string;
 * string="I love China!";
```

在输出时，要用

```
printf("%s\n",string);
```

%s 是输出字符串时所用的格式符，在输出项中给出字符指针变量名 string，则系统先输出它所指向的一个字符数据，然后自动使 string 加 1，使之指向下一个字符，然后再输出一个字符……如此直到遇到字符串结束标志'\0'为止。注意，在内存中，字符串的最后被自动加了一个'\0'（如图 10-32 所示），因此在输出时能确定字符串的终止位置。

说明：通过字符数组名或字符指针变量可以输出一个字符串。而对一个数值型数组，是不能企图用数组名输出它的全部元素的。例如：

```
int i[10];
   ⋮
printf("%d\n",i);
```

是不行的，只能逐个元素输出。

显然，用%s 可以对一个字符串进行整体的输入输出。

对字符串中字符的存取，可以用下标方法，也可以用指针方法。

例 10.17 将字符串 a 复制为字符串 b。

```
#include <stdio.h>
void main()
  { char a[]="I am a boy.",b[20];
    int i;
    for(i=0;*(a+i)!='\0';i++)
      *(b+i)=*(a+i);
    *(b+i)='\0';
    printf("string a is:%s\n",a);
    printf("string b is:");
    for(i=0;b[i]!='\0';i++)
      printf("%c",b[i]);
    printf("\n");
  }
```

程序运行结果为：

```
string a is:I am a boy.
string b is:I am a boy.
```

程序中 a 和 b 都定义为字符数组，可以通过地址访问其数组元素。在 for 语句中，先检查 a[i]是否为'\0'(今 a[i]是以 *(a+i)形式表示的)。如果不等于'\0'，表示字符串尚未处理完，就将 a[i]的值赋给 b[i]，即复制一个字符。在 for 循环中将 a 串全部复制给了 b 串。最后还应将'\ 0'复制过去，故有

```
*(b+i)='\0';
```

此时 i 的值是字符串有效字符的个数 n 加 1。第二个 for 循环中用下标法表示一个数组元素(即一个字符)。

也可以设指针变量，用它的值的改变来指向字符串中的不同的字符。

例 10.18 用指针变量来处理例 10.17 问题。

```
#include <stdio.h>
void main()
 { char a[]="I am a boy.",b[20],*p1,*p2;
   int i;
   p1=a;p2=b;
   for(;*p1!='\0';p1++,p2++)
     *p2=*p1;
   *p2='\0';
   printf("string a is:%s\n",a);
   printf("string b is:");
   for(i=0;b[i]!='\0';i++)
     printf("%c",b[i]);
   printf("\n");
 }
```

p1、p2是指向字符型数据的指针变量。先使p1和p2的值分别为字符串a和b第1个字符的地址。*p1最初的值为'I',赋值语句"*p2=*p1;"的作用是将字符'I'(a串中第1个字符)赋给p2所指向的元素,即b[0]。然后p1和p2分别加1,指向其下面的一个元素,直到*p1的值为'\0'止。注意p1和p2的值是不断在改变的,见图10-33的虚线和p1'、p2'。程序必须保证使p1和p2同步移动。

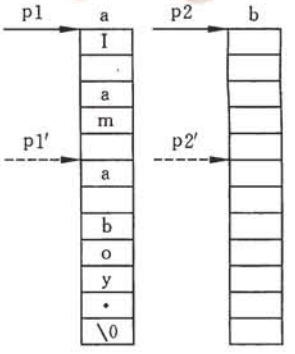


图 10-33

10.4.2 字符指针作函数参数

将一个字符串从一个函数传递到另一个函数,可以用地址传递的办法,即用字符数组名作参数,也可以用指向字符的指针变量作参数。在被调用的函数中可以改变字符串的内容,在主调函数中可以得到改变了的字符串。

例10.19 用函数调用实现字符串的复制。

(1) 用字符数组作参数

```
#include <stdio.h>
void main()
  { void copy_string(char from[ ], char to[ ]);
    char a[ ]="I am a teacher.";
    char b[ ]="You are a student.";
    printf("string a=%s\nstring b=%s\n",a,b);
    printf("copy string a to string b:\n");
    copy_string(a,b);
    printf("\nstring a=%s\nstring b=%s\n",a,b);
  }

void copy_string(char from[ ], char to[ ])
  { int i=0;
    while(from[i]!='\0')
      {to[i]=from[i];i++;}
    to[i]='\0';
  }
```

程序运行结果如下:

```
string a=I am a teacher.
string b=You are a student.
copy string a to string b:
string a=I am a teacher.
string b=I am a teacher.
```

a和b是字符数组。初值如图10-34(a)所示。copy_string函数的作用是将from[i]赋给to[i],直到from[i]的值等于'\0'为止。在调用copy_string函数时,将a和b第1个字符的地址分别传递给形参数组名from和to。因此from[i]和a[i]是同一个单元,to[i]和b[i]是同一个单元。程序执行完以后,b数组的内容如图10-34(b)所示。可以看到,由于

b 数组原来的长度大于 a 数组，因此在将 a 数组复制到 b 数组后，未能全部覆盖 b 数组原有内容。b 数组最后 3 个元素仍保留原状。在输出 b 时由于按%s(字符串)输出，遇'\0'即告结束，因此第一个'\0'后的字符不输出。如果不采取%s 格式输出而用%c 逐个字符输出是可以输出后面这些字符的。

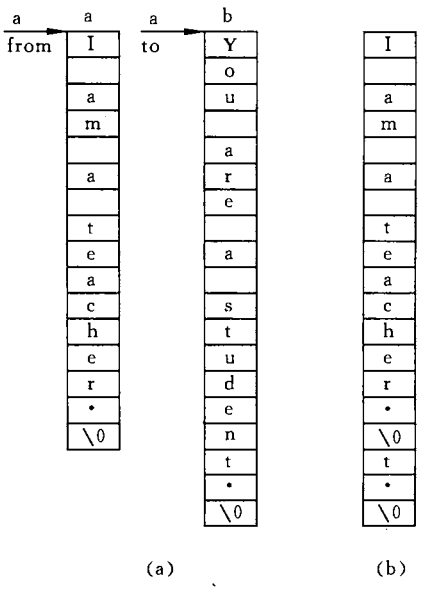


图 10-34

在 main 函数中也可以用字符型指针变量作实参，先使指针变量 a 和 b 分别指向两个字符串。main 函数可改写如下：

```
#include <stdio.h>
void main()
{ void copy_string(char from[ ], char to[ ]);
  char from[]="I am a teacher.";
  char to[]="You are a student.";
  char *a=from, *b=to;
  printf("string a=%s\nstring b=%s\n",a,b);
  printf("copy string a to string b:\n");
  copy_string(a,b);
  printf("\string a=%s\nstring b=%s\n",a,b);
}

void copy_string (char from[], char to [])
{ int i=0;
  while (from[i]!='\0')
  {to[i]=from[i]; i++;}
  to[i]='\0';
}
```

运行结果与上面程序相同。

(2) 形参用字符指针变量。

程序如下：

```
#include <stdio.h>
void main()
{ void copy_string(char from[], char to[]);
  char from[]="I am a teacher.";
  char to[]="You are a student.";
  char *a=from, *b=to;
  printf("string a=%s\nstring b=%s\n",a,b);
  printf("copy string a to string b:\n");
  copy_string(a,b);
  printf("string a=%s\nstring b=%s\n",a,b);
}

void copy_string(char *from,char *to)
{ for(;*from!='\0';from++,to++)
```

```
    * to= * from;
  * to='\0';
}
```

形参 from 和 to 是字符指针变量。它们相当于例 10.18 中的 p1 和 p2。算法也与例 10.18 完全相同。在调用 copy_string 时,将数组 a 首元素的地址传给 from,把数组 b 首元素的地址传给 to。在函数 copy_string 中的 for 循环中,每次将 * from 赋给 * to,第 1 次就是将 a 数组中第 1 个字符赋给 b 数组的第 1 个字符。在执行 from++和 to++以后,from 和 to 就分别指向 a[1]和 b[1]。再执行 * to= * from,就将 a[1]赋给 b[1]……最后将'\0'赋给 * to,注意此时 to 指向哪个单元。

(3) 对 copy_string 函数还可作简化。

① 将 copy_string 函数改写为:

```
void copy_string(char * from,char * to)
  {while(( * to= * from)!='\0')
      {to++;from++;}
  }
```

请与上面一个程序对比。在本程序中将" * to= * from"的操作放在 while 语句括号内的表达式中,而且把赋值运算和判断是否为'\0'的运算放在一个表达式中,先赋值后判断。在循环体中使 to 和 from 增值,指向下一个元素……直到 * from 的值为'\0'为止。

② copy_string 函数的函数体还可改为:

```
{
  while(( * to++= * from++)!='\0');
}
```

把上面程序的 to++和 from++运算与 * to= * from 合并,它的执行过程是,先将 * from 赋给 * to,然后使 to 和 from 增值。显然这又简化了。

③ copy_string 函数的函数体还可写成:

```
{
 while( * from!='\0')
    * to++= * from++;
 * to='\0';
}
```

当 * from 不等于'\0'时,将 * from 赋给 * to,然后使 to 和 from 增值。

字符可以用其 ASCII 码来代替。例如,"ch='a'"可以用"ch=97"代替,"while(ch!='a')"可以用"while(ch!=97)"代替。因此,"while(* from!='\0')"可以用"while(* from!=0)"代替('\0'的 ASCII 代码为 0)。而关系表达式" * from!=0"又可简化为" * from",这是因为若 * from 的值不等于 0,则表达式" * from"为真,同时" * from!=0" 也为真。因此"while(* from!=0)"和"while(* from)"是等价的。所以函数体可简化为:

```
{while( * from)
    * to++= * from++;
```

```
    * to='\0';
}
```

④ 上面的 while 语句还可以进一步简化为下面的 while 语句：

```
while( * to++= * from++);
```

它与下面语句等价：

```
while(( * to++= * from++)!='\0');
```

将 * from 赋给 * to，如果赋值后的 * to 值等于'\0'，则循环终止('\0'已赋给 * to)。

⑤ 函数体中 while 语句也可以改用 for 语句：

```
for(;( * to++= * from++)!=0;);
```

或

```
for(; * to++= * from++;);
```

⑥ 也可用指针变量，函数 copy_string 可写为：

```
void copy_string(char from[ ],char to[ ])
  { char * p1, * p2;
    p1=from;p2=to;
    while(( * p2++= * p1++)!='\0');
  }
```

以上各种用法，变化多端，使用十分灵活，初看起来不太习惯，含义不直观。初学者会有些困难，也容易出错。但对 C 熟练之后，以上形式的使用是比较多的，读者应逐渐熟悉它，掌握它。

归纳起来，作为函数参数，有以下几种情况：见表 10-3。

表 10-3

实　参	形　参	实　参	形　参
数组名	数组名	字符指针变量	字符指针变量
数组名	字符指针变量	字符指针变量	数组名

10.4.3 对使用字符指针变量和字符数组的讨论

虽然用字符数组和字符指针变量都能实现字符串的存储和运算，但它们二者之间是有区别的，不应混为一谈，主要有以下几点。

(1) 字符数组由若干个元素组成，每个元素中放一个字符，而字符指针变量中存放的是地址(字符串第 1 个字符的地址)，决不是将字符串放到字符指针变量中。

(2) 赋值方式。对字符数组只能对各个元素赋值，不能用以下办法对字符数组赋值：

```
char str[14];
str="I love China!";
```

而对字符指针变量，可以采用下面方法赋值：

```
char * a;
a="I love China!";
```

但注意赋给 a 的不是字符，而是字符串第一个元素的地址。

(3) 对字符指针变量赋初值：

```
char * a="I love China!";
```

等价于

```
char * a;
a="I love China!";
```

而对数组的初始化：

```
char str[14]={"I love China!"};
```

不能等价于

```
char str[14];
str[ ]="I love China!";
```

即数组可以在定义时整体赋初值，但不能在赋值语句中整体赋值。

(4) 如果定义了一个字符数组，在编译时为它分配内存单元，它有确定的地址。而定义一个字符指针变量时，给指针变量分配内存单元，在其中可以放一个字符变量的地址，也就是说，该指针变量可以指向一个字符型数据，但如果未对它赋予一个地址值，则它并未具体指向一个确定的字符数据。例如：

```
char str[10];
scanf("%s",str);
```

是可以的。而常有人用下面的方法：

```
char * a;
scanf("%s",a);
```

目的是想输入一个字符串，虽然一般也能运行，但这种方法是危险的，决不应提倡。因为编译时虽然给指针变量 a 分配了内存单元，a 的地址(即 &a)是已指定了，但 a 的值并未指定，在 a 单元中是一个不可预料的值。在执行 scanf 函数时要求将一个字符串输入到 a 所指向的一段内存单元(即以 a 的值(地址)开始的一段内存单元)中。而 a 的值如今却是不可预料的，它可能指向内存中空白的(未用的)用户存储区中(这是好的情况)，也有可能指向已存放指令或数据的有用内存段，这就会破坏了程序，甚至破坏了系统，会造成严重的后果。在程序规模小时，由于空白地带多，往往可以正常运行，而程序规模大时，出现上述“冲突”的可能性就大多了。应当这样：

```
char * a,str[10];
a=str;
```

```
scanf("%s",a);
```

先使 a 有确定值，也就是使 a 指向一个数组的首元素，然后输入一个字符串，把它存放在以该地址开始的若干单元中。

(5) 指针变量的值是可以改变的，见下例。

例 10.20 改变指针变量的值。

```
#include <stdio.h>
void main()
  { char *a="I love China!";
    a=a+7;
    printf("%s",a);
  }
```

运行结果如下：

```
China!
```

指针变量 a 的值是可以变化的，输出字符串时从 a 当时所指向的单元开始输出各个字符，直到遇'\0'为止。而数组名虽然代表地址，但它是常量，它的值是不能改变的。下面是错的：

```
char str[ ]={"I love China!"};
str=str+7;
printf("%s",str);
```

需要说明，若定义了一个指针变量，并使它指向一个字符串，就可以用下标形式引用指针变量所指的字符串中的字符。

例 10.21 用带下标的字符指针变量引用字符串中的字符。

```
#include <stdio.h>
void main()
  { char *a="I love China!";
    int i;
    printf("The sixth character is %c\n",a[5]);
    for(i=0;a[i]!='\0';i++)
      printf("%c",a[i]);
    printf("\n");
  }
```

运行结果如下：

```
The sixth character is e
I love China!
```

程序中虽然并未定义数组 a，但字符串在内存中是以字符数组形式存放的。a[5]按 *(a+5)处理，即从 a 当前所指向的元素下移 5 个元素位置，取出其单元中的值。

(6) 用指针变量指向一个格式字符串，可以用它代替 printf 函数中的格式字符串。

例如：

```
char * format;
format="a=%d,b=%f\n";
printf(format,a,b);
```

它相当于

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

因此只要改变指针变量 format 所指向的字符串，就可以改变输入输出的格式。这种 printf 函数称为可变格式输出函数。也可以用字符数组实现。例如：

```
char format[ ]="a=%d,b=%f\n";
printf(format,a,b);
```

但由于不能采用赋值语句对数组整体赋值，例如：

```
char format[];
format="a=%d,b=%d\n";
```

因此，用指针变量指向字符串的方式更为方便。

10.5 指向函数的指针

10.5.1 用函数指针变量调用函数

可以用指针变量指向整型变量、字符串、数组，也可以指向一个函数。一个函数在编译时被分配给一个入口地址。这个函数的入口地址就称为函数的指针。可以用一个指针变量指向函数，然后通过该指针变量调用此函数。先通过一个简单的例子来回顾一下函数的调用情况。

例 10.22 求 a 和 b 中的大者。先列出按一般方法的程序。

```
#include <stdio.h>
void main()
  { int max(int,int);
    int a,b,c;
    scanf("%d,%d",&a,&b);
    c=max(a,b);
    printf("a=%d,b=%d,max=%d\n",a,b,c);
  }

  int max(int x,int y)
  { int z;
    if(x>y)z=x;
    else z=y;
    return(z);
  }
```

main 函数中的“c=max(a,b);”包括了一次函数调用(调用 max 函数)。每一个函数都占用一段内存单元,它们有一个起始地址。因此,可以用一个指针变量指向一个函数,通过指针变量来访问它指向的函数。

将 main 函数改写为:

```
#include <stdio.h>
void main()
  { int max(int,int);
    int (*p)(int,int);
    int a,b,c;
    p=max;
    scanf("%d,%d",&a,&b);
    c=(*p)(a,b);
    printf("a=%d,b=%d,max=%d\n",a,b,c);
  }
```

第 4 行“int(*p)(int,int);”用来定义 p 是一个指向函数的指针变量,该函数有两个整型参数,函数值为整型。注意 *p 两侧的括号不可省略,表示 p 先与 * 结合,是指针变量,然后再与后面的()结合,表示此指针变量指向函数,这个函数值(即函数返回的值)是整型的。如果写成“int *p(int,int);”,则由于()优先级高于 *,它就成了声明一个 p 函数了(这个函数的返回值是指向整型变量的指针)。

赋值语句“p=max;”的作用是将函数 max 的入口地址赋给指针变量 p。和数组名代表数组首元素地址类似,**函数名代表该函数的入口地址**。这时,p 就是指向函数 max 的指针变量,此时 p 和 max 都指向函数的开头,见图 10-35。调用 *p 就是调用 max 函数。请注意 p 是指向函数的指针变量,它只能指向函数的入口处而不可能指向函数中间的某一条指令处,因此不能用 *(p+1)来表示函数的下一条指令。

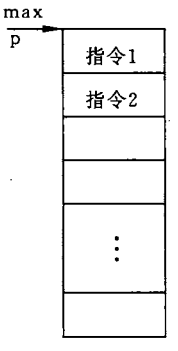


图 10-35

在 main 函数中有一个赋值语句:

```
c=(*p)(a,b);
```

它和“c=max(a,b);”等价。这就是用指针形式实现函数的调用。以上用两种方法实现函数的调用,结果是一样的。

说明:

(1) 指向函数的指针变量的一般定义形式为

数据类型 (*指针变量名)(函数参数表列);

这里的“数据类型”是指函数返回值的类型。

(2) 函数的调用可以通过函数名调用,也可以通过函数指针调用(即用指向函数的指针变量调用)。

(3) “int (*p)(int,int);”表示定义一个指向函数的指针变量 p,它不是固定指向哪一个函数的,而只是表示定义了这样一个类型的变量,它是专门用来存放函数的入口地址的。在程序中把哪一个函数(该函数的值应是整型的,且有两个整型参数)的地址赋给它,它就指向哪一个函数。在一个程序中,一个指针变量可以先后指向同类型的不同函数。

(4) 在给函数指针变量赋值时,只需给出函数名而不必给出参数,例如:

```
p=max;
```

因为是将函数入口地址赋给 p,而不牵涉实参与形参的结合问题。不能写成

```
p=max(a,b);
```

(5) 用函数指针变量调用函数时,只需将(*p)代替函数名即可(p 为指针变量名),在(*p)之后的括号中根据需要写上实参。例如:

```
c=(*p)(a,b);
```

表示“调用由 p 指向的函数,实参为 a、b。得到的函数值赋给 c”。注意函数的返回值是什么类型?从上例对指针变量 p 的定义可以知道,函数的返回值是整型的,因此将其值赋给整型变量 c 是合法的。

(6) 对指向函数的指针变量,像 p+n、p++、p--等运算是无意义的。

10.5.2 用指向函数的指针作函数参数

函数指针变量通常的用途之一是把指针作为参数传递到其他函数。这个问题是 C 语言应用的一个比较深入的部分,在本书中只作简单的介绍,以便在今后用到时不致感到困惑。进一步的理解和掌握有待于读者今后深入的学习和提高。

前面介绍过,函数的参数可以是变量、指向变量的指针变量、数组名、指向数组的指针变量等。现在介绍指向函数的指针也可以作为参数,以实现函数地址的传递,这样就能够在被调用的函数中使用实参函数。

它的原理可以简述如下:有一个函数(假设函数名为 sub),它有两个形参(x1 和 x2),定义 x1 和 x2 为指向函数的指针变量。在调用函数 sub 时,实参为两个函数名 f1 和 f2,给形参传递的是函数 f1 和 f2 的地址。这样在函数 sub 中就可以调用 f1 和 f2 函数了。例如:

```
实参函数名   f1              f2
             ↓               ↓
void sub(int (*x1)(int),int (*x2)(int,int))
    { int a,b,i,j;
      a=(*x1)(i);                    /*调用 f1 函数*/
      b=(*x2)(i,j);                  /*调用 f2 函数*/
          ⋮
    }
```

在函数首部定义 x1、x2 为函数指针变量，x1 指向的函数有一个整型形参，x2 指向的函数有两个整型形参。i 和 j 是函数 f1 和 f2 所要求的参数。函数 sub 的形参 x1、x2(指针变量)在函数 sub 未被调用时并不占内存单元，也不指向任何函数。在 sub 被调用时，把实参函数 f1 和 f2 的入口地址传给形参指针变量 x1 和 x2，使 x1 和 x2 指向函数 f1 和 f2，见图 10-36。这时，在函数 sub 中，用 * x1 和 * x2 就可以调用函数 f1 和 f2。(* x1)(i)就相当于 f1(i)，(* x2)(i,j)就相当于 f2(i,j)。

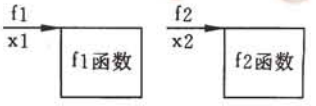


图 10-36

有人可能会问，既然在 sub 函数中要调用 f1 和 f2 函数，为什么不直接调用 f1 和 f2 而要用函数指针变量呢？何必绕这样一个圈子呢？的确，如果只是用到 f1 和 f2 函数，完全可以在 sub 函数中直接调用 f1 和 f2，而不必设指针变量 x1、x2。但是，如果在每次调用 sub 函数时，要调用的函数不是固定的，这次调用 f1 和 f2，而下次要调用 f3 和 f4，第三次要调用的是 f5 和 f6。这时，用指针变量就比较方便了。只要在每次调用 sub 函数时给出不同的函数名作为实参即可，sub 函数不必做任何修改。这种方法是符合结构化程序设计方法原则的，是程序设计中常使用的。

下面通过一个简单的例子来说明这种方法的应用。

例 10.23 设一个函数 process，在调用它的时候，每次实现不同的功能。输入 a 和 b 两个数，第一次调用 process 时找出 a 和 b 中大者，第二次找出其中小者，第三次求 a 与 b 之和。

程序如下：

```
#include <stdio.h>
void main()
  { int max(int,int);                              /* 函数声明 */
    int min(int,int);                              /* 函数声明 */
    int add(int,int);                              /* 函数声明 */
    void process (int,int,int( * fun)(int,int));   /* 函数声明 */
    int a,b;
    printf("enter a and b:");
    scanf("%d,%d",&a,&b);
    printf("max=");
    process(a,b,max);
    printf("min=");
    process(a,b,min);
    printf("sum=");
    process(a,b,add);
  }
int max(int x,int y)                               /* 函数定义 */
  { int z;
    if(x>y)z=x;
    else z=y;
```

```
    return(z);
  }
int min(int x,int y)                          /* 函数定义 */
  { int z;
    if(x<y)z=x;
    else z=y;
    return(z);
  }
int add(int x,int y)                          /* 函数定义 */
  {int z;
   z=x+y;
   return(z);
  }
void process(int x,int y,int (*fun)(int,int))   /* 函数定义 */
  { int result;
    result=(*fun)(x,y);
    printf("%d\n",result);
  }
```

运行情况如下：

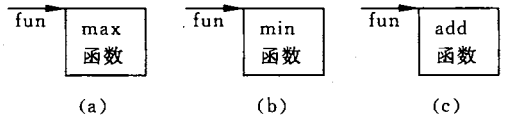
```
enter a and b:2,6↙
max=6
min=2
sum=8
```

在定义 process 函数时，在函数首部用“int(*fun)(int,int)”表示 fun 是指向函数的指针，该函数是一个整型函数，有两个整型形参。max、min 和 add 是已定义的 3 个函数，分别用来实现求大数、求小数和求和的功能。

在 main 函数中第一次调用 process 函数时，除了将 a 和 b 作为实参，将两个整数传给 process的形参 x、y 外，还将函数名 max 作为实参将其入口地址传送给 process 函数中的形参fun(fun 是指向函数的指针变量)，见图 10-37(a)。这时，process 函数中的(*fun)(x,y)相当于 max(x,y)，执行 process 可以输出 a 和 b 中大者。

在 main 函数第二次调用时，改以函数名 min 作实参，此时 process 函数的形参 fun 指向函数min，见图 10-37(b)，在 process 函数中的函数调用(*fun)(x,y)相当于 min(x,y)。同理，第三次调用 process 函数时，情况见图 10-37(c)。(*fun)(x,y)相当于add(x,y)。

从本例可以清楚地看到，不论调用 max、min 或 add，函数 process 一点都没有改动，只是在调用 process 函数时改变实参函数名而已。这就增加了函数使用的灵活性。可以编写一个通用的函数来实现各种专用的功能。需要注意的是，对作为实参的函



(a) (b) (c)

图 10-37

数，应在主调函数中用函数原型作函数声明。例如，main 函数中第 3 行到第 5 行的函数声明是不可少的。

有了以上基础，就可以编写出较为复杂的程序。例如，编写一个求定积分的通用函数，用它分别求以下 5 个函数的定积分：

$$\int_a^b (1+x)\mathrm{d}x, \int_a^b (2x+3)\mathrm{d}x, \int_a^b (e^x+1)\mathrm{d}x, \int_a^b (1+x)^2\mathrm{d}x, \int_a^b x^3\mathrm{d}x$$

可以看出，每次需要求定积分的函数是不一样的。可以编写一个求定积分的通用函数 integral，它有 3 个形参：下限 a、上限 b 以及指向函数的指针变量 fun。函数原型可写为

```
float integral (float a, float b, float ( * fun)(float));
```

分别定义 5 个 C 函数 f1、f2、f3、f4、f5，用来求上面 5 个函数：$1+x$、$2x+3$、e^x+1、$(1+x)^2$、x^3。然后先后调用 integral 函数 5 次，每次调用时把 a、b 以及一个函数名（f1、f2、f3、f4、f5 之中的一个）作为实参，即把上限、下限以及有关函数的入口地址传送给形参 fun。在执行 integral 函数过程中求出定积分的值。请读者根据以上思路，编写出完整的程序。

10.6 返回指针值的函数

一个函数可以返回一个整型值、字符值、实型值等，也可以返回指针型的数据，即地址。其概念与以前类似，只是返回的值的类型是指针类型而已。

这种返回指针值的函数，一般定义形式为

类型名 * 函数名(参数表列);

例如：

```
int * a(int x,int y);
```

a 是函数名，调用它以后能得到一个指向整型数据的指针（地址）。x、y 是函数 a 的形参，为整型。请注意在 * a 两侧没有括号，在 a 的两侧分别为 * 运算符和()运算符。而()优先级高于 *，因此 a 先与()结合。显然这是函数形式。这个函数前面有一个 *，表示此函数是指针型函数（函数值是指针）。最前面的 int 表示返回的指针指向整型变量。对初学 C 语言的人来说，这种定义形式可能不大习惯，容易弄错，用时要十分小心。

例 10.24 有若干个学生的成绩（每个学生有 4 门课程），要求在用户输入学生序号以后，能输出该学生的全部成绩。用指针函数来实现。

程序如下：

```
#include <stdio.h>
void main()
  {float score[ ][4]={{60,70,80,90},{56,89,67,88},{34,78,90,66}};
   float * search(float ( * pointer)[4],int n);
   float * p;
   int i,m;
```

```
    printf("enter the number of student:");
    scanf("%d",&m);
    printf("The scores of No. %d are:\n",m);
    p=search(score,m);
    for(i=0;i<4;i++)
      printf("%5.2f\t",*(p+i));
    printf("\n");
  }
float *search(float (*pointer)[4],int n)
  {float *pt;
   pt=*(pointer+n);
   return(pt);
  }
```

运行情况如下：

```
enter the number of student:1↙
The scores of No. 1 are:
56.00   89.00   67.00   88.00
```

注意：学生序号是从 0 号算起的。函数 search 被定义为指针型函数，它的形参 pointer 是指向包含 4 个元素的一维数组的指针变量。pointer+1 指向 score 数组序号为 1 的行，见图 10-38。*(pointer+1)指向 1 行 0 列元素，加了“*”号后，指针从行控制转化为列控制了。search 函数中的 pt 是指针变量，它指向实型变量(而不是指向一维数组)。main 函数调用 search 函数，将 score 数组首行地址传给形参 pointer(注意 score 也是指向行的指针，而不是指向列元素的指针)。m 是要查找的学生序号。调用 search 函数后，得到一个地址(指向第 m 个学生第 0 门课程)，赋给 p。然后将此学生的 4 门课程的成绩输出。注意 p 是指向列元素的指针变量，*(p+i)表示该学生第 i 门课程的成绩。

请注意指针变量 p、pt 和 pointer 的区别。

如果将 search 函数中的语句

```
pt=*(pointer+n);
```

改为

```
pt=(*pointer+n);
```

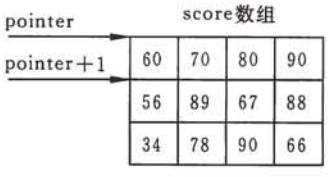


图 10-38

运行结果为：

```
enter the number of student:1↙
The scores of No. 1 are:
70.00   80.00   90.00   56.00
```

得到的不是第一个学生的成绩，而是二维数组中 a[0][1]开始的 4 个元素的值。为什么？请读者分析。

例 10.25 对上例中的学生，找出其中不及格课程的学生及其学生号。

程序如下：

```
#include <stdio.h>
void main()
  {float score[ ][4]={{60,70,80,90},{56,89,67,88},{34,78,90,66}};
   float *search(float (*pointer)[4]);
   float *p;
   int i,j;
   for(i=0;i<3;i++)
     {p=search(score+i);
      if(p==*(score+i))
         {printf("No.%d scores:",i);
          for(j=0;j<4;j++)
            printf("%5.2f",*(p+j));
          printf("\n");}
     }
  }
float *search(float (*pointer)[4])
  { int i;
    float *pt;
    pt=*(pointer+1);
    for(i=0;i<4;i++)
      if(*(*pointer+i)<60) pt=*pointer;
    return(pt);
  }
```

程序运行结果为：

```
No.1 scores: 56.00 89.00 67.00 88.00
No.2 scores: 34.00 78.00 90.00 66.00
```

函数 search 的作用是检查一个学生有无不及格的课程。在 search 函数中的 pointer 是指针变量，指向一维数组(有 4 个元素)。pt 为指向实型变量的指针变量。从实参传给形参 pointer 的是 score+i，它是 score 第 i 行的首地址，见图 10-39(a)。

在 search 函数中，先使 pt=*(pointer+1)，即把 pt 指向本行之末尾，见图 10-39(b)。用 pt 作为区分有无不及格课程的标志。若经检查 4 门课中有不及格的，就使 pt 指向本行 0 列元素；若无不及格则保持 pt 指向本行末尾(即下一行 0 列元素)。将 pt 返回 main 函数。在 main 函数中，把调用 search 得到的函数值(指针变量 pt 的值)，赋给 p。用 if 语句判断 p 是否等于*(score+i)，若相等，表示所查的序号为 i 的学生有不及格课程(p 的值为*(score+i)，即 p 指向第 i 行的 0 列元素)；若无不及格，p 的值是*(score+i+1)，因为在函数 search 中已使它指向本行的末尾，也就是下一行开头。如果 p 等于*(score+i)，就输出该学生(有不及格课程的学生)4 门课成绩。

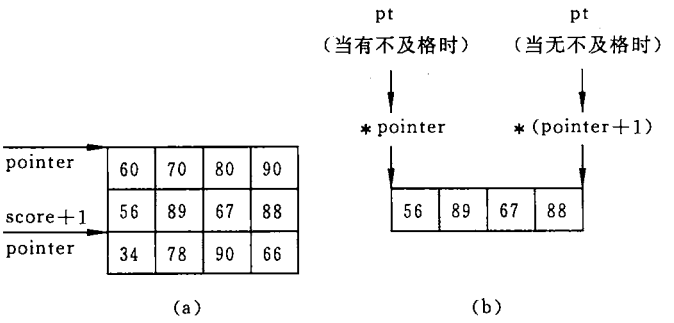


图 10-39

请读者仔细消化本例中指针变量的含义和用法。

10.7 指针数组和指向指针的指针

10.7.1 指针数组的概念

一个数组，若其元素均为指针类型数据，称为指针数组，也就是说，指针数组中的每一个元素都相当于一个指针变量。一维指针数组的定义形式为

类型名 * 数组名[数组长度]；

例如：

```
int * p[4];
```

由于[ ]比 * 优先级高，因此 p 先与[4]结合，形成 p[4]形式，这显然是数组形式，它有 4 个元素。然后再与 p 前面的“ * ”结合，“ * ”表示此数组是指针类型的，每个数组元素（相当于一个指针变量）都可指向一个整型变量。

注意不要写成

```
int( * p)[4];
```

这是指向一维数组的指针变量。这在前面已介绍过了。

为什么要用到指针数组呢？这是因为它比较适合于用来指向若干个字符串，使字符串处理更加方便灵活。例如，图书馆有若干本书，想把书名放在一个数组中（图 10-40(a)），然后要对这些书目进行排序和查询。按一般方法，字符串本身就是一个字符数组。因此要设计一个二维的字符数组才能存放多个字符串。但在定义二维数组时，需要指定列数，也就是说二维数组中每一行中包含的元素个数（即列数）相等。而实际上各字符串（书名）长度一般是不相等的。如按最长的字符串来定义列数，则会浪费许多内存单元，见图 10-40(b)。

可以分别定义一些字符串，然后用指针数组中的元素分别指向各字符串，见图 10-40(c)。如果想对字符串排序，不必改动字符串的位置，只需改动指针数组中各元素的指向（即改变各元素的值，这些值是各字符串的首地址）。这样，各字符串的长度可以不同，而且移动指针变量的值（地址）要比移动字符串所花的时间少得多。

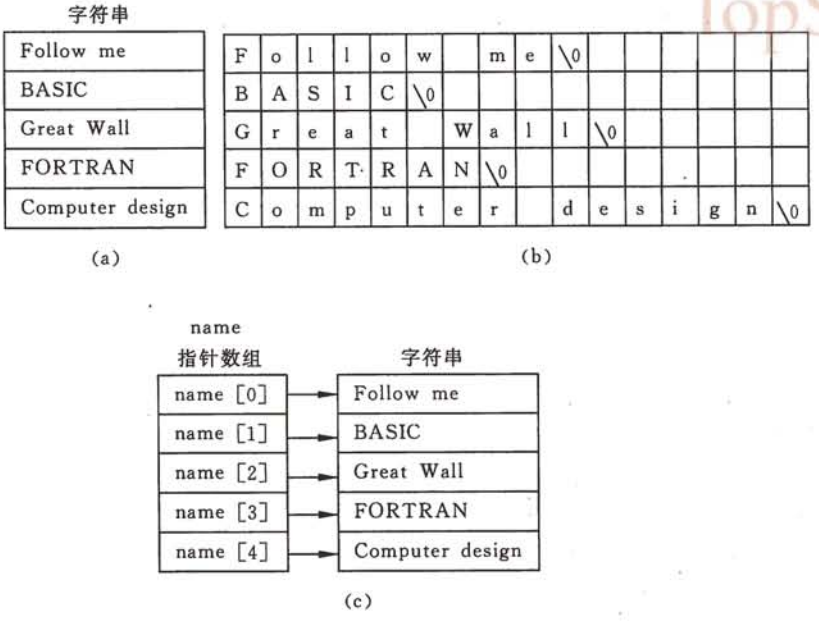


图 10-40

例 10.26 将若干字符串按字母顺序(由小到大)输出。

```
#include <stdio.h>
#include <string.h>
void main()
  { void sort(char *name[ ],int n);
    void print(char *name[ ],int n);
    char *name[ ]={"Follow me","BASIC","Great Wall","FORTRAN","Computer design"};
    int n=5;
    sort(name,n);
    print(name,n);
  }
void sort(char *name[ ],int n)
  { char *temp;
    int i,j,k;
    for(i=0;i<n-1;i++)
      { k=i;
        for(j=i+1;j<n;j++)
          if(strcmp(name[k],name[j])>0) k=j;
        if(k!=i)
          {temp=name[i]; name[i]=name[k]; name[k]=temp;}
      }
  }
  void print(char *name[ ],int n)
```

```
{int i;
 for(i=0;i<n;i++)
   printf("%s\n",name[i]);
}
```

运行结果为：

```
BASIC
Computer design
FORTRAN
Follow me
Great Wall
```

在 main 函数中定义指针数组 name，它有 5 个元素，其初值分别是"Follow me"、"BASIC"、"Great Wall"、"FORTRAN"和"Computer design"的起始地址，见图 10-40(c)。这些字符串是不等长的(并不是按同一长度定义的)。

sort 函数的作用是对字符串排序。sort 函数的形参 name 也是指针数组名，接受实参传过来的 name 数组 0 行的地址，因此形参 name 数组和实参 name 数组指的是同一数组。用选择法对字符串排序。strcmp 是字符串比较函数，name[k]和 name[j]是第 k 个和第 j 个字符串的起始地址。strcmp(name[k],name[j])的值为：如果 name[k]所指的字符串大于 name[j]所指的字符串，则此函数值为正值；若相等，则函数值为 0；若小于，则函数值为负值。if 语句的作用是将两个串中"小"的那个串的序号(k 或 j 之一)保留在变量 k 中。当执行完内循环 for 语句后，从第 i 串到第 n 串这些字符串中，第 k 串最"小"。若 k≠i 就表示最小的串不是第 i 串。故将 name[i]和 name[k]对换，也就是将指向第 i 串的数组元素(是指针型元素)与指向第 k 串的数组元素对换。执行完 sort 函数后指针数组的情况如图 10-41 所示。

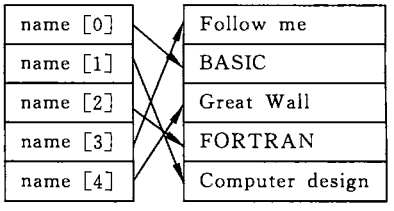


图 10-41

print 函数的作用是输出各字符串。name[0]到 name[4]分别是各字符串(按从小到大顺序排好序的各字符串)的首地址(按字符串从小到大顺序，name[0]指向最小的串)，用"%s"格式符输出，就得到这些字符串。

print 函数也可改写为以下形式：

```
void print(char * name[ ],int n)
  { int i=0;
    char * p;
    p=name[0];
    while(i<n)
      {p= *(name+i++);
       printf("%s\n",p);}
  }
```

其中“*(name+i++)”表示先求*(name+i)的值，即name[i](它是一个地址)，然后使i加1。在输出时，按字符串形式输出从p地址开始的字符串。

请注意sort函数中的第一个if语句中的逻辑表达式的正确用法。如果写成下面形式是不对的：

```
if(*name[k]>*name[j]) k=j;
```

这样只比较name[k]和name[j]所指向的字符串中的第一个字符。字符串比较应当用strcmp函数。

10.7.2 指向指针的指针

在掌握了指针数组的概念的基础上，下面介绍指向指针数据的指针变量，简称为指向指针的指针。从图10-42可以看到，name是一个指针数组，它的每一个元素是一个指针型数据，其值为地址。name是一个数组，它的每一元素都有相应的地址。数组名name代表该指针数组首元素的地址。name+i是name[i]的地址。name+i就是指向指针型数据的指针。还可以设置一个指针变量p，它指向指针数组的元素(见图10-43)。p就是指向指针型数据的指针变量。

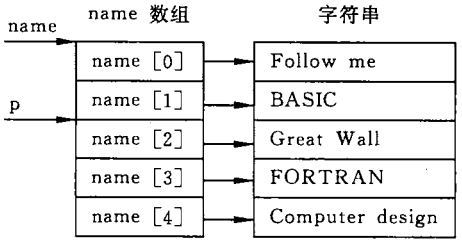


图 10-42

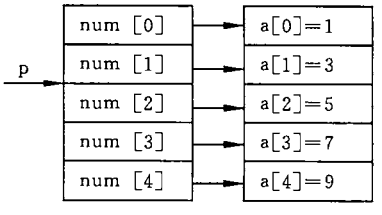


图 10-43

怎样定义一个指向指针数据的指针变量呢？例如：

```
char **p;
```

p的前面有两个*号。从附录Ⅲ可以知道，*运算符的结合性是从右到左，因此**p相当于*(*p)，显然*p是指针变量的定义形式。如果没有最前面的*，那就是定义了一个指向字符数据的指针变量。现在它前面又有一个*号，表示指针变量p是指向一个字符指针变量(即指向字符型数据的指针变量)的。*p就是p所指向的另一个指针变量，如果有：

```
p=name+2;
printf("%o\n",*p);
printf("%s\n",*p);
```

第一个printf函数语句输出name[2]的值(它是一个地址)，第二个printf函数语句以字符串形式(%s)输出字符串"Great Wall"。

例10.27 使用指向指针的指针。

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

```
void main()
  { char * name[]={"Follow me","BASIC","Great Wall","FORTRAN","Computer design"};
    char ** p;
    int i;
    for(i=0;i<5;i++)
      {p=name+i;
       printf("%s\n", * p);
      }
  }
```

运行结果如下：

```
Follow me
BASIC
Great Wall
FORTRAN
Computer design
```

p 是指向指针的指针变量，在第一次执行循环体时，赋值语句“p=name+i;”使 p 指向 name 数组的 0 号元素 name[0]，* p 是 name[0]的值，即第一个字符串的起始地址，用 printf 函数输出第一个字符串(格式符为%s)。执行 5 次循环体，依次输出 5 个字符串。

指针数组的元素也可以不指向字符串，而指向整型数据或实型数据等，例如：

```
int a[5]={1,3,5,7,9};
int * num[5],i;
int ** p;
for (i=0; i<5; i++)
  num[i]=&a[i];
```

此时为了得到 a[2]中的数据 5，可以先使 p=num+2，然后输出 ** p。注意 * p 是 p 间接指向的对象的地址 num[2]。而 ** p 是 p 间接指向的对象的值，即 * num[2]，也就是 a[2]的值 5，见图 10-43。

例 10.28 指针数组的元素指向整型数据。这是一个简单例子，目的是为了说明它的用法。

```
#include <stdio.h>
void main()
{   int a[5]={1,3,5,7,9};
    int * num[5]={&a[0],&a[1],&a[2],&a[3],&a[4]};
    int ** p,i;
    p=num;
    for(i=0;i<5;i++)
      {printf("%d  ", ** p);
       p++;
      }
    printf("\n");
}
```

运行结果为：

```
1 3 5 7 9
```

请不要把第 3、4 行错写为

```
int * num[5]={1,3,5,7,9};
```

指针数组的元素只能存放地址。

读者可在此例基础上实现对各数排序。

在本章开头已经提到了“间接访问”变量的方式。利用指针变量访问另一个变量就是“间接访问”。如果在一个指针变量中存放一个目标变量的地址，这就是“单级间址”，见图 10-44(a)。指向指针的指针用的是“二级间址”方法，见图 10-44(b)。从理论上说，间址方法可以延伸到更多的级，见图 10-44(c)。但实际上在程序中很少有超过二级间址的。级数愈多，愈难理解，容易产生混乱，出错机会也多。

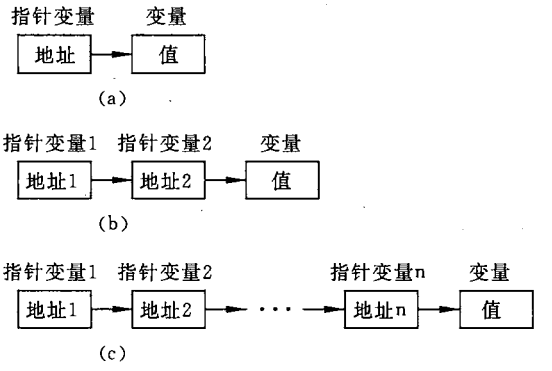


图 10-44

10.7.3 指针数组作 main 函数的形参

指针数组的一个重要应用是作为 main 函数的形参。在以往的程序中，main 函数的第一行一般写成以下形式：

void main()

括号中是空的。实际上，main 函数可以有参数，例如：

void main(int argc,char * argv[ ])

argc 和 argv 就是 main 函数的形参。main 函数是由操作系统调用的。当处于操作命令状态下，输入 main 所在的文件名(经过编译、连接后得到的可执行文件名，后缀为. exe)，操作系统就调用 main 函数。那么，main 函数的形参的值从何处得到呢？显然不可能在程序中得到。实际上实参是和命令一起给出的，也就是在一个命令行中包括命令名和需要传给 main 函数的参数。命令行的一般形式为

命令名 参数 1　参数 2……参数 n

命令名和各参数之间用空格分隔。命令名是 main 所在的执行文件名，假设为 file1，今想将两个字符串“China”，“Beijing”作为传送给 main 函数的参数。参数可以写成以下形式：

```
file1 China Beijing
```

实际上，文件名应包括盘符、路径以及文件的扩展名，今为简化起见，用 file1 来代表。

请注意以上参数与 main 函数中形参的关系。main 函数中形参 argc 是指命令行中参数的个数(注意，文件名也作为一个参数。例如，本例中“file1” 也是一个参数)，现在，argc 的值等于 3(有 3 个命令行参数：file1、China、Beijing)。main 函数的第二个形参 argv 是一个指向字符串的指针数组，也就是说，带参数的 main 函数原型是

```
void main(int argc,char * argv[ ]);
```

命令行参数应当都是字符串(例如，上面命令行中的“file1”、“China”、“Beijing”都是字符串)，这些字符串的首地址构成一个指针数组，见图 10-45。

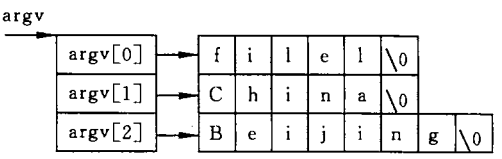


图 10-45

指针数组 argv 中的元素 argv[0]指向字符串"file1"(或者说 argv[0]的值是字符串"file1"的首地址)，argv[1]指向字符串"China"，argv[2]指向字符串"Beijing"。

如果有一个名为 file1 的文件，它包含以下的 main 函数：

```
void main(int argc,char * argv[ ])
{while(argc>1)
    {++argv;
     printf("%s\n", * argv);
     --argc;
    }
}
```

在 DOS 命令状态下输入的命令行为

```
file1 China Beijing↙
```

则执行以上命令行将会输出以下信息：

```
China
Beijing
```

上面 main 函数可以改写为

```
void main(int argc,char * argv[ ])
  {while(argc-->1)
     printf("%s\n", * ++argv);
  }
```

其中 * ++argv 是先进行++argv 的运算，使 argv 指向下一个元素，然后进行 * 的运

算，找到 argv 当前指向的字符串，输出该字符串。在开始时，argv 指向字符串"file1"，++argv使之指向"China"，所以第一次输出的是"China"，第二次输出"Beijing"。

许多操作系统提供了 echo 命令，它的作用是实现"参数回送"，即将 echo 后面的各参数(各字符串)在同一行上输出。实现"参数回送"的 C 程序(文件名为 echo.c)如下：

```
#include <stdio.h>
void main(int argc,char *argv[])
 {while(--argc>0)
     printf("%s%c",*++argv,(argc>1)?' ':'\n');
 }
```

如果命令行输入：

```
c>echo Computer and C Language↙
```

会在显示屏上输出：

```
Computer and C Language
```

这个程序与前面的差别在于：①将 while 语句中的(argc-->1)改为(--argc>0)，作用显然是一样的。②当 argc>1 时，在输出的两个字符串间输出一个空格，当 argc=1时输出一个换行。程序不输出命令名"echo"。

为便于理解，echo 程序也可写成以下形式：

```
#include <stdio.h>
void main(int argc,char *argv[])
    { int i;
      for(i=1;i<argc;i++)
         printf("%s%c",argv[i],(i<argc-1)?' ':'\n');
    }
```

main 函数中的形参不一定命名为 argc 和 argv，可以是任意的名字，只是人们习惯用 argc 和 argv 而已。

利用指针数组作 main 函数的形参，可以向程序传送命令行参数(这些参数是字符串)，这些字符串的长度事先并不知道，而且各参数字符串的长度一般并不相同，命令行参数的数目也是可以任意的。用指针数组能够较好地满足上述要求。

关于指向指针的指针是 C 语言中比较深入的概念，在此只作简单的介绍，以便为读者提供今后进一步学习的基础。

10.8 有关指针的数据类型和指针运算的小结

前面已经介绍了有关指针的数据类型和指针的运算，为使读者有一系统完整的概念，现在作一小结。

10.8.1 有关指针的数据类型的小结

表 10-4 是有关指针的数据类型的小结，为便于比较，我们把其他一些类型的定义也列在一起。

表 10-4

定　义	含　义
int i;	定义整型变量 i
int * p;	p 为指向整型数据的指针变量
int a[n];	定义整型数组 a，它有 n 个元素
int * p[n];	定义指针数组 p，它由 n 个指向整型数据的指针元素组成
int (* p)[n];	p 为指向含 n 个元素的一维数组的指针变量
int f();	f 为返回整型函数值的函数
int * p();	p 为返回一个指针的函数，该指针指向整型数据
int (* p)();	p 为指向函数的指针，该函数返回一个整型值
int ** p;	p 是一个指针变量，它指向一个指向整型数据的指针变量

10.8.2 指针运算小结

前面已用过一些指针运算(如 p++，p+i 等)，今把全部的指针运算列出如下：

(1) 指针变量加(减)一个整数

例如：p++、p--、p+i、p-i、p+=i、p-=i 等均是指针变量加(减)一个整数。

C 语言规定，一个指针变量加(减)一个整数并不是简单地将指针变量的原值加(减)一个整数，而是将该指针变量的原值(是一个地址)和它指向的变量所占用的内存单元字节数相加(减)。如 p+i 代表地址计算：p+c * i。c 为字节数，在 Turbo C 中，数据类型为整型时，c 为 2，实型时 c 为 4，字符型时 c 为 1。这样才能保证 p+i 指向 p 下面的第 i 个元素，它才有实际意义。

(2) 指针变量赋值

将一个变量地址赋给一个指针变量。例如：

```
p=&a;                (将变量 a 的地址赋给 p)
p=array;             (将数组 array 首元素地址赋给 p)
p=&array[i];         (将数组 array 第 i 个元素的地址赋给 p)
p=max;               (max 为已定义的函数，将 max 的入口地址赋给 p)
p1=p2;               (p1 和 p2 都是指针变量，将 p2 的值赋给 p1)
```

注意：不应把一个整数赋给指针变量。例如：

```
p=1000;
```

有人以为这样可以将地址 1000 赋给 p，但实际上是做不到的。只能将变量已分配的地址赋给指针变量。

同样也不应把指针变量 p 的值(地址)赋给一个整型变量 i：

```
i=p;
```

(3) 指针变量可以有空值，即该指针变量不指向任何变量，可以这样表示：

```
p=NULL;
```

其中 NULL 是整数 0，它使 p 的存储单元中所有二进位均为 0，也就是使 p 指向地址为 0 的单元。系统保证使该单元不作它用(不存放有效数据)，即有效数据的指针不指向 0 单元。实际上是先定义 NULL，即：

```
#define NULL 0
   ⋮
p=NULL;
```

在 stdio.h 头文件中就有以上的 NULL 定义，它是一个符号常量。用“p=NULL;”表示 p 不指向任一有用单元。应注意，p 的值为 NULL 与未对 p 赋值是两个不同的概念。前者是有值的(值为 0)，不指向任何程序变量，后者虽未对 p 赋值但并不等于 p 无值，只是它的值是一个无法预料的值，也就是 p 可能指向一个事先未指定的单元。这种情况是很危险的。因此，在引用指针变量之前应对它赋值。

任何指针变量或地址都可以与 NULL 作相等或不相等的比较，例如：

```
if(p==NULL)…
```

(4) 两个指针变量可以相减

如果两个指针变量都指向同一个数组中的元素，则两个指针变量值之差是两个指针之间的元素个数，见图 10-46。

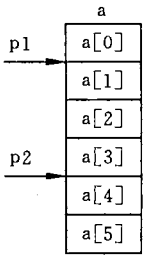


图 10-46

假如 p1 指向 a[1]，p2 指向 a[4]，则 p2－p1＝4－1＝3。但 p1＋p2 并无实际意义。

(5) 两个指针变量比较

若两个指针指向同一个数组的元素，则可以进行比较。指向前面的元素的指针变量“小于”指向后面元素的指针变量。如图 10-46 中，p1＜p2，或者说，表达式“p1＜p2”的值为 1(真)，而“p2＜p1”的值为 0(假)。注意，如果 p1 和 p2 不指向同一数组则比较无意义。

10.8.3 void 指针类型

ANSI 新标准增加了一种“void”指针类型，即可定义一个指针变量，但不指定它是指向哪一种类型数据的。ANSI C 标准规定用动态存储分配函数时返回 void 指针(见附录 V 中的第 4 点)，它可以用来指向一个抽象的类型的数据，在将它的值赋给另一指针变量时要进行强制类型转换使之适合于被赋值的变量的类型。例如：

```
char * p1;
void * p2;
   ⋮
p1=(char *)p2;
```

同样可以用(void *)p1 将 p1 的值转换成 void * 类型。例如：

```
p2=(void *)p1;
```

也可以将一个函数定义为 void * 类型，例如：

```
void * fun(char ch1,char ch2)
```

表示函数 fun 返回的是一个地址，它指向“空类型”，如果需要引用此地址，需要根据情况对之进行类型转换，例如可以对该函数调用得到的地址进行以下的转换：

```
p1=(char *)fun(ch1,ch2);
```

void * 指针类型是新标准新增加的，欲详细了解可参阅有关手册。

在本章中介绍了指针的基本概念和初步应用。应该说明，指针是 C 语言中重要的概念，是 C 的一个特色。使用指针的优点：①提高程序效率；②在调用函数时变量改变了的值能够为主调函数使用，即可以从函数调用得到多个可改变的值；③可以实现动态存储分配。

但是同时应该看到，指针使用实在太灵活，对熟练的程序人员来说，可以利用它编写出颇有特色的、质量优良的程序，实现许多用其他高级语言难以实现的功能，但也十分容易出错，而且这种错误往往比较隐蔽。由于指针运用的错误可能会使整个程序遭受破坏，比如由于未对指针变量 p 赋值就向 * p 赋值，就可能破坏了有用的单元的内容。有人说指针是有利有弊的工具，如果使用指针不当，(例如赋予它一个错误的值)，会出现隐蔽的、难以发现和排除的故障。因此，使用指针要十分小心谨慎，要多上机调试程序，以弄清一些细节，并积累经验。

习　　题

本章习题均要求用指针方法处理。

10.1　输入 3 个整数，按由小到大的顺序输出。

10.2　输入 3 个字符串，按由小到大的顺序输出。

10.3　输入 10 个整数，将其中最小的数与第一个数对换，把最大的数与最后一个数对换。写 3 个函数：①输入 10 个数；②进行处理；③输出 10 个数。

10.4　有 n 个整数，使前面各数顺序向后移 m 个位置，最后 m 个数变成最前面 m 个数，见图 10-47。写一函数实现以上功能，在主函数中输入 n 个整数和输出调整后的 n 个数。

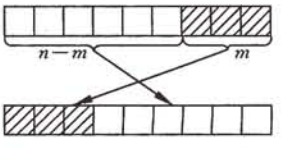


图　10-47

10.5　有 n 个人围成一圈，顺序排号。从第 1 个人开始报数(从 1 到 3 报数)，凡报到 3 的人退出圈子，问最后留下的

是原来第几号的那位。

10.6 写一函数，求一个字符串的长度。在 main 函数中输入字符串，并输出其长度。

10.7 有一字符串，包含 n 个字符。写一函数，将此字符串中从第 m 个字符开始的全部字符复制成为另一个字符串。

10.8 输入一行文字，找出其中大写字母、小写字母、空格、数字以及其他字符各有多少？

10.9 写一函数，将一个 3×3 的整型矩阵转置。

10.10 将一个 5×5 的矩阵中最大的元素放在中心，4 个角分别放 4 个最小的元素（顺序为从左到右，从上到下顺序依次从小到大存放），写一函数实现之。用 main 函数调用。

10.11 在主函数中输入 10 个等长的字符串。用另一函数对它们排序。然后在主函数输出这 10 个已排好序的字符串。

10.12 用指针数组处理上一题目，字符串不等长。

10.13 写一个用矩形法求定积分的通用函数，分别求

$$\int_0^1 \sin x\,dx,\ \int_0^1 \cos x\,dx,\ \int_0^1 e^x\,dx$$

说明：sin、cos、exp 函数已在系统的数学函数库中，程序开头要用 #include <math.h>）。

10.14 将 n 个数按输入时顺序的逆序排列，用函数实现。

10.15 有一个班 4 个学生，5 门课程。①求第一门课程的平均分；②找出有 2 门以上课程不及格的学生，输出他们的学号和全部课程成绩及平均成绩；③找出平均成绩在 90 分以上或全部课程成绩在 85 分以上的学生。分别编 3 个函数实现以上 3 个要求。

10.16 输入一个字符串，内有数字和非数字字符，例如：

a123x456 17960? 302tab5876

将其中连续的数字作为一个整数，依次存放到一数组 a 中。例如，123 放在 a[0]，456 放在 a[1]……统计共有多少个整数，并输出这些数。

10.17 写一函数，实现两个字符串的比较。即自己写一个 strcmp 函数，函数原型为

```
int strcmp(char *p1,char *p2);
```

设 p1 指向字符串 s1，p2 指向字符串 s2。要求当 s1=s2 时，返回值为 0；若 s1≠s2，返回它们二者第一个不同字符的 ASCII 码差值（如"BOY"与"BAD"，第二个字母不同，"O"与"A"之差为 79－65＝14）。如果 s1>s2，则输出正值；如果 s1<s2，则输出负值。

10.18 编一程序，输入月份号，输出该月的英文月名。例如，输入"3"，则输出"March"，要求用指针数组处理。

10.19 编写一个函数 alloc(n)，用来在内存区新开辟一个连续的空间（n 个字节）。此函数的返回值是一个指针，指向新开辟的连续空间的起始地址。再写一个函数 free(p)，将地址 p 开始的各单元释放（不能再被程序使用，除非再度开辟）。

提示：先在内存区确定出一片相当大的连续空间（例如 1000 个字节）。然后开辟与释放都在此空间内进行。假设指针变量 p 原已指向未用空间的开头，调用 alloc(n)后，开辟了 n 个字节可供程序使用（例如，可以赋值到这些单元中）。现在需要使 p 的值变成

p+n，表示空白未用区从 p+n 地址开始，同时要将新开辟区的起始位置(p)作为函数值返回，以表示可以利用从此点开始的单元。如果要新开辟的区太大(n 大)，超过了预设的空间——1000 字符，则 alloc(n)函数返回指针 NULL，表示开辟失败。

alloc(n)应返回一个指向字符型数据的指针(因为开辟的区间是以字节为单位被利用的)。

10.20 用指向指针的指针的方法对 5 个字符串排序并输出。

10.21 用指向指针的指针的方法对 *n* 个整数排序并输出。要求将排序单独写成一个函数。*n* 个整数在主函数中输入，最后在主函数中输出。

第 11 章　结构体与共用体

11.1　概　　述

迄今为止，已介绍了基本类型（或称简单类型）的变量（如整型、实型、字符型变量等），也介绍了一种构造类型数据——数组，数组中的各元素是属于同一个类型的。

但是只有这些数据类型是不够的。有时需要将不同类型的数据组合成一个有机的整体，以便于引用。这些组合在一个整体中的数据是互相联系的，例如，一个学生的学号、姓名、性别、年龄、成绩、家庭地址等项。这些项都与某一学生相联系，见图 11-1。可以看到性别（sex）、年龄（age）、成绩（score）、地址（addr）是属于学号为 10010 和名为“Li Fun”的学生的。如果将 num、name、sex、age、score、addr 分别定义为互相独立的简单变量，难以反映它们之间的内在联系，应当把它们组织成一个组合项，在一个组合项中包含若干个类型不同（当然也可以相同）的数据项。C 语言允许用户自己指定这样一种数据结构，它称为结构体（structure）。它相当于其他高级语言中的“记录”。

num	name	sex	age	score	addr
10010	Li Fun	M	18	87.5	Beijing

图　11-1

假设程序中要用到图 11-1 所表示的数据结构，但是 C 语言没有提供这种现成的数据类型，因此用户必须要在程序中建立所需的结构体类型。

例如：

```
struct student
{int num;
 char name[20];
 char sex;
 int age;
 float score;
 char addr[30];
};
```

注意不要忽略最后的分号。上面由程序设计者指定了一个新的结构体类型 struct student（struct 是声明结构体类型时所必须使用的关键字，不能省略），它向编译系统声明这是一个“结构体类型”，它包括 num、name、sex、age、score、addr 等不同类型的数据项。应当说明 struct student 是一个类型名，它和系统提供的标准类型（如 int、char、float、double 等）一样具有同样的作用，都可以用来定义变量的类型，只不过结构体类型需要由

用户自己指定而已。

声明一个结构体类型的一般形式为：

struct　结构体名

{成员表列};

“结构体名”用作结构体类型的标志，它又称“结构体标记”(structure tag)。上面的结构体声明中 student 就是结构体名(结构体标记)。花括号内是该结构体中的各个成员，由它们组成一个结构体。例如，上例中的 num、name、sex 等都是成员。对各成员都应进行类型声明，即

类型名　成员名;

也可以把“成员表列”(member list)称为“域表”(field list)。每一个成员也称为结构体中的一个域。成员名命名规则与变量名相同。

“结构体”这个词是根据英文单词 structure 译出的。有些 C 语言书把 structure 直译为“结构”。作者认为译作“结构”会与一般含义上的“结构”混淆(例如，数据结构、程序结构、控制结构等)。日本把 structure 译作“结构体”或“构造体”，作者认为译作“结构体”比译作“结构”更确切一些，不致与一般含义上的“结构”混淆。

11.2　定义结构体类型变量的方法

前面只是指定了一个结构体类型，它相当于一个模型，但其中并无具体数据，系统对之也不分配实际的内存单元。为了能在程序中使用结构体类型的数据，应当定义结构体类型的变量，并在其中存放具体的数据。可以采取以下 3 种方法定义结构体类型变量。

1. 先声明结构体类型再定义变量名

如上面已定义了一个结构体类型 struct student，可以用它来定义变量。例如：

struct student　student1, student2;

(struct student：结构体类型名；student1, student2：结构体变量名)

定义了 student1 和 student2 为 struct student 类型的变量，即它们具有 struct student 类型的结构，如图 11-2 所示。

student1:	10001	Zhang Xin	M	19	90.5	Shanghai
student2:	10002	Wang Li	F	20	98	Beijing

图　11-2

在定义了结构体变量后，系统会为之分配内存单元。例如，student1 和 student2 在内存中各占 59 个字节(2+20+1+2+4+30=59)。

应当注意，将一个变量定义为标准类型(基本数据类型)与定义为结构体类型不同之处在于后者不仅要求指定变量为结构体类型，而且要求指定为某一特定的结构体类型(例

如 struct student 类型),因为可以定义出许许多多种具体的结构体类型。而在定义变量为整型时,只需简单化指定为 int 型即可。

如果程序规模比较大,往往将对结构体类型的声明集中放到一个头文件(以.h 为后缀)中。哪个源文件需用到此结构体类型则可用 #include 命令将该头文件包含到本文件中。这样做便于装配,便于修改,便于使用。

2. 在声明类型的同时定义变量

例如:

```
struct student
{int num;
 char name[20];
 char sex;
 int age;
 float score;
 char addr[30];
}student1,student2;
```

它的作用与第一种方法相同,即定义了两个 struct student 类型的变量 student1、student2。这种形式的定义的一般形式为:

struct 结构体名
{
成员表列
}**变量名表列**;

3. 直接定义结构体类型变量

其一般形式为:

struct
{
成员表列
}**变量名表列**;

即不出现结构体名。

关于结构体类型,有几点要说明:

(1) 类型与变量是不同的概念,不要混同。只能对变量赋值、存取或运算,而不能对一个类型赋值、存取或运算。在编译时,对类型是不分配空间的,只对变量分配空间。

(2) 对结构体中的成员(即"域"),可以单独使用,它的作用与地位相当于普通变量。关于对成员的引用方法见 11.3 节。

(3) 成员也可以是一个结构体变量。

例如:

```
struct date                    /*声明一个结构体类型*/
  {int month;
```

```
    int day;
    int year;
  };
struct student
  {int num;
   char name[20];
   char sex;
   int age;
   struct date birthday;        /* birthday 是 struct date 类型 */
   char addr[30];
  }student1,student2;
```

先声明一个 struct date 类型，它代表“日期”，包括 3 个成员：month（月）、day（日）、year（年）。然后在声明 struct student 类型时，将成员 birthday 指定为 struct date 类型。struct student 的结构见图 11-3 所示。已声明的类型 struct date 与其他类型（如 int、char）一样可以用来定义成员的类型。

num	name	sex	age	birthday			addr
				month	day	year	

图 11-3

（4）成员名可以与程序中的变量名相同，二者不代表同一对象。例如，程序中可以另定义一个变量 num，它与 struct student 中的 num 是两回事，互不干扰。

11.3 结构体变量的引用

在定义了结构体变量以后，当然可以引用这个变量。但应遵守以下规则：

（1）不能将一个结构体变量作为一个整体进行输入和输出。例如，已定义 student1 和 student2 为结构体变量并且它们已有值。不能这样引用：

```
printf ("%d,%s,%c,%d,%f,%s\n",student1);
```

只能对结构体变量中的各个成员分别进行输入和输出。引用结构体变量中成员的方式为

结构体变量名.成员名

例如，student1.num 表示 student1 变量中的 num 成员，即 student1 的 num（学号）项。可以对变量的成员赋值，例如：

```
student1.num=10010;
```

“.”是成员（分量）运算符，它在所有的运算符中优先级最高，因此可以把 student1.num 作为一个整体来看待。上面赋值语句的作用是将整数 10010 赋给 student1 变量中的成员 num。

（2）如果成员本身又属一个结构体类型，则要用若干个成员运算符，一级一级地找到最低的一级的成员。只能对最低级的成员进行赋值或存取以及运算。例如，对上面定义

的结构体变量 student1，可以这样访问各成员：

```
student1. num
student1. birthday. month
```

注意：不能用 student1. birthday 来访问 student1 变量中的成员 birthday，因为 birthday 本身是一个结构体变量。

(3) 对结构体变量的成员可以像普通变量一样进行各种运算(根据其类型决定可以进行的运算)。例如：

```
student2. score=student1. score;
sum=student1. score+student2. score;
student1. age++;
++student1. age;
```

由于“.”运算符的优先级最高，因此 student1. age++是对 student1. age 进行自加运算，而不是先对 age 进行自加运算。

(4) 可以引用结构体变量成员的地址，也可以引用结构体变量的地址。例如：

```
scanf("%d",&student1. num);          (输入 student1. num 的值)
printf("%o",&student1);              (输出 student1 的首地址)
```

但不能用以下语句整体读入结构体变量，例如：

```
scanf("%d,%s,%c,%d,%f,%s",&student1);
```

结构体变量的地址主要用作函数参数，传递结构体变量的地址。

11.4 结构体变量的初始化

和其他类型变量一样，对结构体变量可以在定义时指定初始值。

例 11.1 对结构体变量初始化。

```
#include <stdio. h>
void main()
{struct student
  {long int num;
   char name[20];
   char sex;
   char addr[20];
  }a={10101,"Li Lin",'M',"123 Beijing Road"};        /* 对结构体变量 a 赋初值 */
 printf("No. :%ld\nname:%s\nsex:%c\naddress:%s\n",a. num,a. name,a. sex,a. addr);
}
```

运行结果如下：

```
No. :10101
name:Li Lin
```

sex:M
address:123 Beijing Road

11.5 结构体数组

一个结构体变量中可以存放一组数据(如一个学生的学号、姓名、成绩等数据)。如果有 10 个学生的数据需要参加运算,显然应该用数组,这就是结构体数组。结构体数组与以前介绍过的数值型数组不同之处在于每个数组元素都是一个结构体类型的数据,它们都分别包括各个成员(分量)项。

11.5.1 定义结构体数组

和定义结构体变量的方法相仿,只需说明其为数组即可。例如:

```
struct student
 {int num;
  char name[20];
  char sex;
  int age;
  float score;
  char addr[30];
 };
struct student stu[3];
```

以上定义了一个数组 stu,数组有 3 个元素,均为 struct student 类型数据。也可以直接定义一个结构体数组,例如:

```
struct student
 {int num;
     ⋮
 }stu[3];
```

或

```
struct
 {int num;
     ⋮
 }stu[3];
```

见图 11-4。

	num	name	sex	age	score	addr
stu[0]	10101	Li Lin	M	18	87.5	103 Beijing Road
stu[1]	10102	Zhang Fun	M	19	99	130 Shanghai Road
stu[2]	10104	Wang Min	F	20	78.5	1010 Zhongshan Road

图 11-4

数组各元素在内存中连续存放，见图 11-5 示意。

11.5.2 结构体数组的初始化

与其他类型的数组一样，对结构体数组可以初始化。例如：

```
struct student
{int num;
 char name[20];
 char sex;
 int age;
 float score;
 char add[30];
}stu[3]={{10101,"Li Lin",'M',18,87.5,"103 Beijing
         Road"},{10102,"Zhang Fun",'M',19,99,"130
         Shanghai Road"},{10104,"Wang Min",'F',
         20,78.5,"1010 Zhongshan Road"}};
```

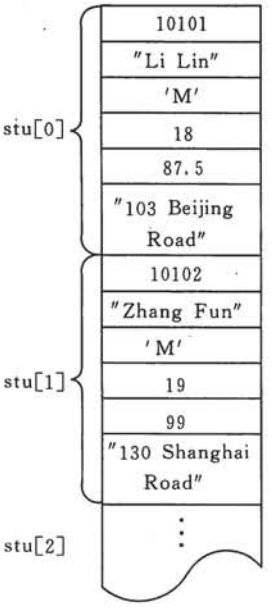


图 11-5

定义数组 stu 时，元素个数可以不指定，即写成以下形式：

```
stu[ ]= {{…},{…},{…} };
```

编译时，系统会根据给出初值的结构体常量的个数来确定数组元素的个数。一个结构体常量包括结构体中全部成员的值。

当然，数组的初始化也可以用以下形式：

```
struct student
{int num;
  ⋮
};
struct student stu[ ]={{…},{…},{…}};
```

即先声明结构体类型，然后定义数组为该结构体类型，在定义数组时初始化。

从以上可以看到，结构体数组初始化的一般形式是在定义数组的后面加上“=｛初值表列｝;”。

11.5.3 结构体数组应用举例

下面举一个简单的例子来说明结构体数组的定义和引用。

例 11.2 对候选人得票的统计程序。设有 3 个候选人，每次输入一个得票的候选人的名字，要求最后输出各人得票结果。

程序如下：

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

```
 {char name[20];
  int count;
 }leader[3]={"Li",0,"Zhang",0,"Fun",0};
void main()
{int i,j;
 char leader_name[20];
 for (i=1;i<=10;i++)
   {scanf("%s",leader_name);
    for(j=0;j<3;j++)
      if(strcmp(leader_name,leader[j].name)==0) leader[j].count++;
   }
 printf("\n");
 for(i=0;i<3;i++)
    printf("%5s:%d\n",leader[i].name,leader[i].count);
}
```

运行情况如下：

```
Li↙
Li↙
Fun↙
Zhang↙
Zhang↙
Fun↙
Li↙
Fun↙
Zhang↙
Li↙
Li:4
Zhang:3
Fun:3
```

程序定义一个全局的结构体数组 leader，它有 3 个元素，每一个元素包含两个成员 name(姓名)和 count(票数)。在定义数组时使之初始化，使 3 位候选人的票数都先置零，见图 11-6。

name	count
Li	0
Zhang	0
Fun	0

图 11-6

在主函数中定义字符数组 leader_name，它代表被选人的姓名，在 10 次循环中每次先输入一个被选人的具体人名，然后把它与 3 个候选人姓名相比，看它和哪一个候选人的名字相同。注意 leader_name 是和 leader[j].name 相比，leader[j]是数组 leader 的第 j 个元素，它包含两个成员项，leader_name 应该和 leader 数组第 j 个元素的 name 成员相比。若 j 为某一值时，输入的姓名与 leader[j].name 相等，就执行“leader[j].count++”，由于成员运算符“.”优先于自增运算符“++”，因此它

相当于(leader[j].count)++,使 leader[j]的成员 count 的值加 1。在输入和统计结束之后,将 3 人的名字和得票数输出。

11.6 指向结构体类型数据的指针

一个结构体变量的指针就是该变量所占据的内存段的起始地址。可以设一个指针变量,用来指向一个结构体变量,此时该指针变量的值是结构体变量的起始地址。指针变量也可以用来指向结构体数组中的元素。

11.6.1 指向结构体变量的指针

下面通过一个简单例子来说明指向结构体变量的指针变量的应用。

例 11.3 指向结构体变量的指针的应用。

```
#include <stdio.h>
#include <string.h>
void main()
{struct student
  {long num;
   char name[20];
   char sex;
   float score;
  };
 struct student stu_1;
 struct student * p;
 p=&stu_1;
 stu_1.num=89101;
 strcpy(stu_1.name,"Li Lin");
 stu_1.sex='M';
 stu_1.score=89.5;
 printf("No.:%ld\nname:%s\nsex:%c\nscore:%f\n",stu_1.num,stu_1.name,stu_1.sex,
       stu_1.score);
 printf("No.:%ld\nname:%s\nsex:%c\nscore:%f\n",(*p).num,(*p).name,(*p).sex,
       (*p).score);
}
```

在主函数中声明了 struct student 类型,然后定义一个 struct student 类型的变量 stu_1。同时又定义一个指针变量 p,它指向一个 struct student 类型的数据。在函数的执行部分将结构体变量 stu_1 的起始地址赋给指针变量 p,也就是使 p 指向 stu_1(见图 11-7),然后对 stu_1 的各成员赋值。第一个 printf 函数是输出 stu_1 的各个成员的值。用 stu_1.num 表示 stu_1 中的成员 num,依此类推。第二个 printf 函数也是用来输出 stu_1 各成员的值,但使用的是(*p).num 这样的形式。(*p)表示 p 指向的结构体变量,(*p).num 是 p 指向的结构体变量中

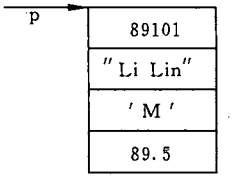


图 11-7

的成员 num。注意 * p 两侧的括号不可省，因为成员运算符“.”优先于“*”运算符，* p. num就等价于 *(p. num)了。

程序运行结果如下：

```
No. :89101
name:Li Lin
sex:M
score:89.500000

No:89101
name:Li Lin
sex:M
score:89.500000
```

可见两个 printf 函数输出的结果是相同的。

为了使用方便和使之直观，可以把(* p). num 改用 p—>num 来代替，它表示 p 所指向的结构体变量中的 num 成员。同样，(* p). name 等价于 p—>name。也就是说，以下 3 种形式等价：

① 结构体变量. 成员名

② (* p). 成员名

③ p—>成员名

上面程序中最后一个 printf 函数中的输出项表列可以改写为

```
p->num,p->name,p->sex,p->score
```

其中—>称为**指向运算符**。

请分析以下几种运算：

p—>n	得到 p 指向的结构体变量中的成员 n 的值。
p—>n++	得到 p 指向的结构体变量中的成员 n 的值，用完该值后使它加 1。
++p—>n	得到 p 指向的结构体变量中的成员 n 的值加 1，然后再使用它。

11.6.2 指向结构体数组的指针

已经介绍过可以使用指向数组或数组元素的指针和指针变量，同样，对结构体数组及其元素也可以用指针或指针变量来指向。

例 11.4 指向结构体数组的指针的应用。

```
#include <stdio.h>
struct student
{int num;
 char name[20];
 char sex;
 int age;
};
struct student stu[3] = {{10101,"Li Lin",'M',18},{10102,"Zhang Fun",'M',19},{10104,
                         "Wang Min",'F',20}};
```

```
void main()
 {struct student * p;
  printf(" No.     Name            sex          age\n");
  for (p=stu;p<stu+3;p++)
     printf ("%5d %-20s %2c %4d\n",p->num, p->name, p->sex, p->age);
 }
```

运行结果如下：

No.	Name	sex	age
10101	Li Lin	M	18
10102	Zhang Fun	M	19
10104	Wang Min	F	20

p 是指向 struct student 结构体类型数据的指针变量。在 for 语句中先使 p 的初值为 stu，也就是数组 stu 第一个元素的起始地址，见图 11-8 中 p 的指向。在第一次循环中输出 stu[0]的各个成员值。然后执行 p++，使 p 自加 1。p 加 1 意味着 p 所增加的值为结构体数组 stu 的一个元素所占的字节数(在本例中为 2+20+1+2=25 字节)。执行 p++后 p 的值等于 stu+1，p 指向 stu[1]，见图 11-8 中 p′的指向。在第二次循环中输出 stu[1]的各成员值。在执行 p++后，p 的值等于 stu+2，它的指向见图 11-8 中的 p″，再输出 stu[2]的各成员值。在执行 p++后，p 的值变为 stu+3，已不再小于 stu+3 了，不再执行循环。

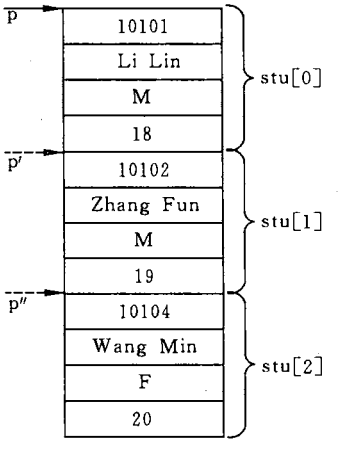


图 11-8

注意以下两点：

(1) 如果 p 的初值为 stu，即指向第一个元素，则 p 加 1 后 p 就指向下一个元素。例如：

(++p)->num 先使 p 自加 1，然后得到它指向的元素中的 num 成员值(即 10102)。

(p++)->num 先得到 p->num 的值(即 10101)，然后使 p 自加 1，指向 stu[1]。

请注意以上二者的不同。

(2) 程序已定义了 p 是一个指向 struct student 类型数据的指针变量，它用来指向一个 struct student 类型的数据(在例 11.4 中 p 的值是 stu 数组的一个元素(如 stu[0]、stu[1])的起始地址)，不应用来指向 stu 数组元素中的某一成员。例如，下面的用法是不对的：

```
p=stu[1].name;
```

编译时将给出“警告”信息，表示地址的类型不匹配。不要认为反正 p 是存放地址的，可以将任何地址赋给它。如果要将某一成员的地址赋给 p，可以用强制类型转换，先将成员的地址转换成 p 的类型。例如：

```
p=(struct student *)stu[0].name;
```

此时，p 的值是 stu[0] 元素的 name 成员的起始地址。可以用“printf("%s",p);”输出 stu[0] 中成员 name 的值。但是，p 仍保持原来的类型。如果执行“printf("%s",p+1);”，则会输出

stu[1]中 name 的值。执行 p+1 时，p 的值增加了结构体 struct student 的长度。

11.6.3 用结构体变量和指向结构体的指针作函数参数

将一个结构体变量的值传递给另一个函数，有 3 个方法：

(1) 用结构体变量的成员作参数。例如，用 stu[1].num 或 stu[2].name 作函数实参，将实参值传给形参。用法和用普通变量作实参是一样的，属于“值传递”方式。应当注意实参与形参的类型保持一致。

(2) 用结构体变量作实参。用结构体变量作实参时，采取的也是“值传递”的方式，将结构体变量所占的内存单元的内容全部顺序传递给形参，形参也必须是同类型的结构体变量。在函数调用期间形参也要占用内存单元。这种传递方式在空间和时间上开销较大，如果结构体的规模很大时，开销是很可观的。此外，由于采用值传递方式，如果在执行被调用函数期间改变了形参(也是结构体变量)的值，该值不能返回主调函数，这往往造成使用上的不便。因此一般较少用这种方法。

(3) 用指向结构体变量(或数组)的指针作实参，将结构体变量(或数组)的地址传给形参。

例 11.5 有一个结构体变量 stu，内含学生学号、姓名和 3 门课程的成绩。要求在 main 函数中赋予值，在另一函数 print 中将它们输出。

今用结构体变量作函数参数。

```
#include <stdio.h>
#include <string.h>
#define FORMAT "%d\n%s\n%f\n%f\n%f\n"
struct student
{int num;
 char name[20];
 float score[3];
};
void main()
  {void print(struct student);
   struct student stu;
   stu.num=12345;
   strcpy(stu.name,"Li Li");
   stu.score[0]=67.5;
   stu.score[1]=89;
   stu.score[2]=78.6;
   print(stu);
  }
void print(struct student stu)
  {printf(FORMAT,stu.num,stu.name, stu.score[0], stu.score[1],stu.score[2]);
   printf("\n");
  }
```

运行结果为：

```
12345
Li Li
67.500000
89.000000
78.599998
```

struct student 被定义为外部的类型，这样，同一源文件中的各个函数都可以用它来定义变量。main 函数中的 stu 定义为 struct student 类型变量，print 函数中的形参 stu 也定义为 struct student 类型变量。在 main 函数中对 stu 的各成员赋值。在调用 print 函数时，以 stu 为实参向形参 stu 实行“值传递”。在 print 函数中输出结构体变量 stu 各成员的值。

例 11.6 将上题改用指向结构体变量的指针作实参。

可以在上面程序的基础上作少量修改即可。请注意程序的注释。

```
#include<stdio.h>
#define FORMAT "%d\n%s\n%f\n%f\n%f\n"
struct student
{int num;
 char name[20];
 float score[3];
}stu={12345,"Li Li",67.5,89,78.6};

void main()
  { void print(struct student *);                    /*形参类型修改成指向结构体的指针变量*/
    print(&stu);                                     /*实参改为 stu 的起始地址*/
  }

void print(struct student *p)                        /*形参类型修改了*/
  {printf(FORMAT,p->num,p->name,p->score[0],
          p->score[1],p->score[2]);                  /*用指针变量调用各成员的值*/
   printf("\n");
  }
```

此程序改用在定义结构体变量 stu 时赋初值，这样程序可简化些。print 函数中的形参 p 被定义为指向 struct student 类型数据的指针变量。注意在调用 print 函数时，用结构体变量 stu 的起始地址 &stu 作实参。在调用函数时将该地址传送给形参 p(p 是指针变量)。这样 p 就指向 stu，见图 11-9。在 print 函数中输出 p 所指向的结构体变量的各个成员值，它们也就是 stu 的成员值。

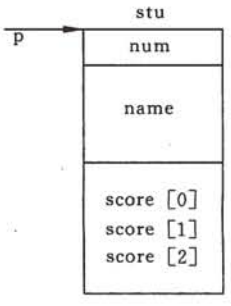


图 11-9

main 函数中的对各成员赋值也可以改用 scanf 函数输入，即用

```
scanf("%d%s%f%f%f",&stu.num,stu.name,&stu.score[0],
      &stu.score[1],&stu.score[2]);
```

输入时用下面形式输入：

12345 Li_Li 67.5 89 78.6↙

注意:输入项表列中 stu. name 前没有“&”符号,因为 stu. name 是字符数组名,本身代表地址,不应写成 &stu. name。

用指针作函数参数比较好,能提高运行效率。

11.7 用指针处理链表

11.7.1 链表概述

链表是一种常见的重要的数据结构。它是动态地进行存储分配的一种结构。我们知道,用数组存放数据时,必须事先定义固定的长度(即元素个数)。比如,有的班级有 100 人,而有的班级只有 30 人,如果要用同一个数组先后存放不同班级的学生数据,则必须定义长度为 100 的数组。如果事先难以确定一个班的最多人数,则必须把数组定得足够大,以便能存放任何班级的学生数据,显然这将会浪费内存。链表则没有这种缺点,它根据需要开辟内存单元。图 11-10 表示最简单的一种链表(单向链表)的结构。

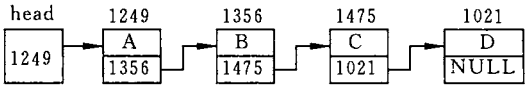


图 11-10

链表有一个“头指针”变量,图中以 head 表示,它存放一个地址,该地址指向一个元素。链表中每一个元素称为“结点”,每个结点都应包括两个部分:用户需要用的实际数据和下一个结点的地址。可以看出,head 指向第一个元素;第一个元素又指向第二个元素……直到最后一个元素,该元素不再指向其他元素,它称为“表尾”,它的地址部分放一个“NULL”(表示“空地址”),链表到此结束。

可以看到链表中各元素在内存中可以不是连续存放的。要找某一元素,必须先找到上一个元素,根据它提供的下一元素地址才能找到下一个元素。如果不提供“头指针”(head),则整个链表都无法访问。链表如同一条铁链一样,一环扣一环,中间是不能断开的。打个通俗的比方:幼儿园的老师带领孩子出来散步,老师牵着第一个小孩的手,第一个小孩的另一只手牵着第二个孩子……这就是一个“链”,最后一个孩子有一只手空着,他是“链尾”。要找这个队伍,必须先找到老师,然后顺序找到每一个孩子。

可以看到,这种链表的数据结构,必须利用指针变量才能实现,即一个结点中应包含一个指针变量,用它存放下一结点的地址。

前面介绍了结构体变量,用它作链表中的结点是最合适的。一个结构体变量包含若干成员,这些成员可以是数值类型、字符类型、数组类型,也可以是指针类型。我们用这个指针类型成员来存放下一个结点的地址。例如,可以设计这样一个结构体类型:

```
struct student
{int num;
 float score;
```

```
 struct student * next;
};
```

其中成员 num 和 score 用来存放结点中的有用数据(用户需要用到的数据),相当于图11-10结点中的 A、B、C、D。next 是指针类型的成员,它指向 struct student 类型数据(这就是 next 所在的结构体类型)。一个指针类型的成员既可以指向其他类型的结构体数据,也可以指向自己所在的结构体类型的数据。现在,next 是 struct student 类型中的一个成员,它又指向 struct student 类型的数据。用这种方法就可以建立链表,见图 11-11。

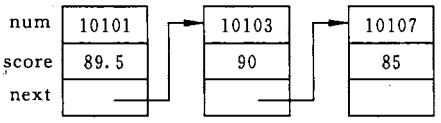


图 11-11

图 11-11 中每一个结点都属于 struct student 类型,它的成员 next 存放下一结点的地址,程序设计人员可以不必具体知道各结点的地址,只要保证将下一个结点的地址放到前一结点的成员 next 中即可。

请注意:上面只是定义了一个 struct student 类型,并未实际分配存储空间。只有定义了变量才分配内存单元。

11.7.2 简单链表

下面通过一个例子来说明如何建立和输出一个简单链表。

例 11.7 建立一个如图 11-11 所示的简单链表,它由 3 个学生数据的结点组成。输出各结点中的数据。

```
#include <stdio.h>
#define NULL 0
struct student
{long num;
 float score;
 struct student * next;
};

void main()
{ struct student a,b,c, * head, * p;
  a.num=10101; a.score=89.5;
  b.num=10103; b.score=90;
  c.num=10107; c.score=85;   /*对结点的 num 和 score 成员赋值*/
  head=&a;                   /*将结点 a 的起始地址赋给头指针 head*/
  a.next=&b;                 /*将结点 b 的起始地址赋给 a 结点的 next 成员*/
  b.next=&c;                 /*将结点 c 的起始地址赋给 b 结点的 next 成员*/
  c.next=NULL;               /*c 结点的 next 成员不存放其他结点地址*/
  p=head;                    /*使 p 指针指向 a 结点*/
  do
    {printf("%ld %5.1f\n",p->num,p->score);
                             /*输出 p 指向的结点的数据*/
     p=p->next;              /*使 p 指向下一结点*/
    } while(p!=NULL);        /*输出完 c 结点后 p 的值为 NULL*/
```

```
}
```

运行结果为：

```
10101    89.5
10103    90.0
10107    85.0
```

请读者仔细考虑：①各个结点是怎样构成链表的。②没有头指针 head 行不行？③p 起什么作用？没有它行不行？

开始时使 head 指向 a 结点，a. next 指向 b 结点，b. next 指向 c 结点，这就构成链表关系。“c. next＝NULL”的作用是使 c. next 不指向任何有用的存储单元。在输出链表时要借助 p，先使 p 指向 a 结点，然后输出 a 结点中的数据，“p＝p－＞next”是为输出下一个结点作准备。p－＞next 的值是 b 结点的地址，因此执行“p＝p－＞next”后 p 就指向 b 结点，所以在下一次循环时输出的是 b 结点中的数据。

本例是比较简单的，所有结点都是在程序中定义的，不是临时开辟的，也不能用完后释放，这种链表称为“静态链表”。

11.7.3 处理动态链表所需的函数

前面讲过，链表结构是动态地分配存储的，即在需要时才开辟一个结点的存储单元。怎样动态地开辟和释放存储单元呢？C 语言编译系统的库函数提供了以下有关函数。

1. malloc 函数

其函数原型为

```
void * malloc(unsigned int size);
```

其作用是在内存的动态存储区中分配一个长度为 size 的连续空间。此函数的值（即“返回值”）是一个指向分配域起始地址的指针（类型为 void）。如果此函数未能成功地执行（例如内存空间不足），则返回空指针（NULL）。

2. calloc 函数

其函数原型为

```
void * calloc(unsigned n,unsigned size);
```

其作用是在内存的动态存储区中分配 n 个长度为 size 的连续空间。函数返回一个指向分配域起始地址的指针；如果分配不成功，返回 NULL。

用 calloc 函数可以为一维数组开辟动态存储空间，n 为数组元素个数，每个元素长度为 size。

3. free 函数

其函数原型为

```
void free(void * p);
```

其作用是释放由 p 指向的内存区，使这部分内存区能被其他变量使用。p 是最近一次调用 calloc 或 malloc 函数时返回的值。free 函数无返回值。

以前的 C 版本提供的 malloc 和 calloc 函数得到的是指向字符型数据的指针。ANSI C 提供的 malloc 和 calloc 函数规定为 void * 类型。

有了本节所介绍的初步知识，下面就可以对链表进行操作了（包括建立链表、插入或删除链表中的一个结点等）。有些概念需要在后面的应用中逐步建立和掌握。

11.7.4 建立动态链表

所谓建立动态链表是指在程序执行过程中从无到有地建立起一个链表，即一个一个地开辟结点和输入各结点数据，并建立起前后相链的关系。

例 11.8 写一函数建立一个有 3 名学生数据的单向动态链表。

先考虑实现此要求的算法（见图 11-12）。

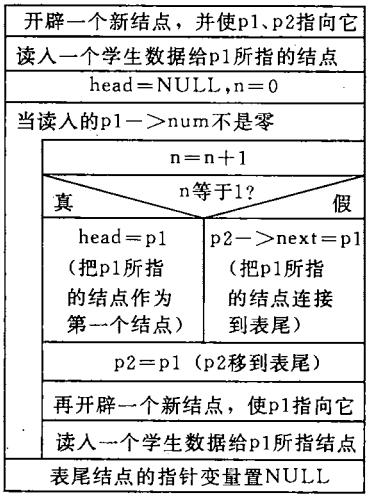


图 11-12

设 3 个指针变量：head、p1、p2，它们都是用来指向 struct student 类型数据的。先用 malloc 函数开辟第一个结点，并使 p1、p2 指向它。然后从键盘读入一个学生的数据给 p1 所指的第一个结点。我们约定学号不会为零，如果输入的学号为 0，则表示建立链表的过程完成，该结点不应连接到链表中。先使 head 的值为 NULL（即等于 0），这是链表为“空”时的情况（即 head 不指向任何结点，即链表中无结点），当建立第一个结点就使 head 指向该结点。

如果输入的 p1->num 不等于 0，则输入的是第一个结点数据（n=1），令 head=p1，即把 p1 的值赋给 head，也就是使 head 也指向新开辟的结点（图 11-13）。p1 所指向的新开辟的结点就成为链表中第一个结点。然后再开辟另一个结点并使 p1 指向它，接着输入该结点的数据（见图 11-14(a)）。如果输入的 p1->num≠0，则应链入第 2 个结点（n=2），由于 n≠1，则将 p1 的值赋给 p2->next，此时 p2 指向第一个结点，因此执行“p2->next=p1”就将新结点的地址赋给第一个结点的 next 成员，使第一个结点的 next 成员指向第二个结点（见图 11-14(b)）。接着使 p2=p1，也就是使 p2 指向刚才建立的结点，见图 11-14(c)。接着再开辟一个结点并使 p1 指向它，并输入该结点的数据（见图 11-15(a)）。在第三次循环中，由于 n=3(n≠1)，又将 p1 的值赋给 p2->next，也就是将第 3 个结点连接到第 2 个结点之后，并使 p2=p1，使 p2 指向最后一个结点（见图11-15(b)）。

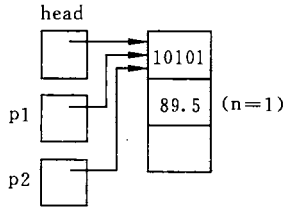


图 11-13

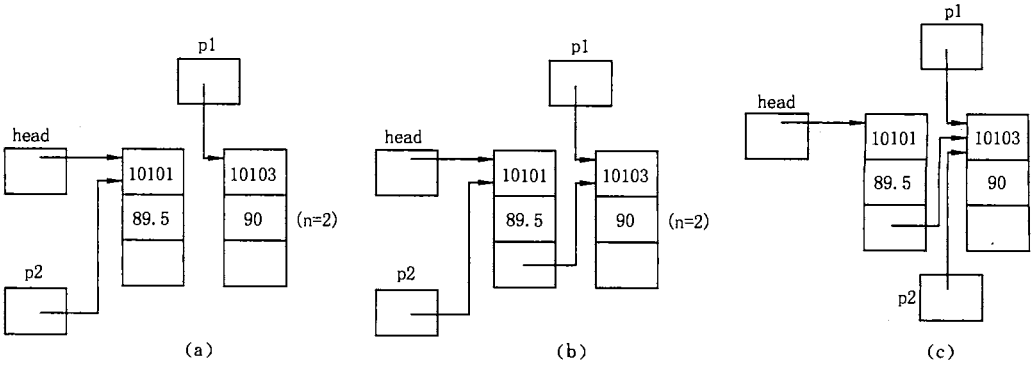


图 11-14

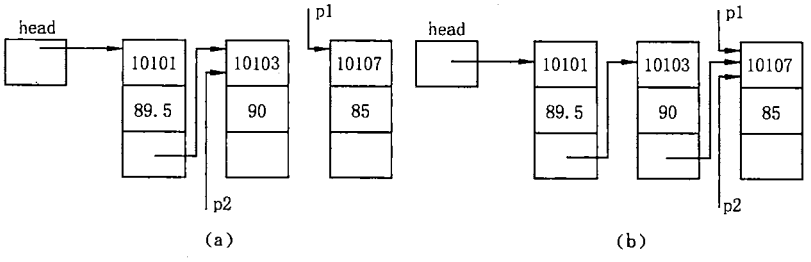


图 11-15

再开辟一个新结点，并使 p1 指向它，输入该结点的数据(见图 11-16(a))。由于 p1 ->num的值为 0，不再执行循环，此新结点不应被连接到链表中。此时将 NULL 赋给 p2 ->next，见图 11-16(b)。建立链表过程至此结束，p1 最后所指的结点未链入链表中，第三个结点的 next 成员的值为 NULL，它不指向任何结点。虽然 p1 指向新开辟的结点，但从链表中无法找到该结点。

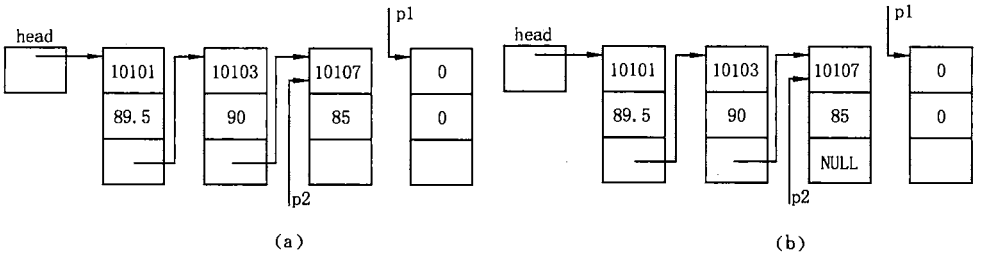


图 11-16

建立链表的函数如下：

```
#include <stdio.h>
#include <malloc.h>
#define NULL 0
#define LEN sizeof(struct student)
struct student
```

```
{long num;
 float score;
 struct student *next;
};
int n;                              /*n为全局变量,本文件模块中各函数均可使用它*/
struct student *creat(void)         /*定义函数。此函数带回一个指向链表头的指针*/
{struct student *head;
 struct student *p1,*p2;
 n=0;
 p1=p2=(struct student*) malloc(LEN); /*开辟一个新单元*/
 scanf("%ld,%f",&p1->num,&p1->score);
 head=NULL;
 while(p1->num!=0)
    {n=n+1;
     if(n==1)head=p1;
     else p2->next=p1;
     p2=p1;
     p1=(struct student*)malloc(LEN);
     scanf("%ld,%f",&p1->num,&p1->score);
    }
 p2->next=NULL;
 return(head);
}
```

函数首部在括号内写 void,表示本函数没有形参,不需要进行数据传递。

可以在 main 函数中调用 creat 函数:

```
void main()
{  ⋮
 creat();        /*调用 creat 函数后建立了一个单向动态链表*/
}
```

调用 creat 函数后,函数的值是所建立的链表的第一个结点的地址(请查看 return 语句)。

请注意:

(1) 第 3 行为 # define 命令行,令 NULL 代表 0,用它表示"空地址"。第 4 行令 LEN 代表 struct student 类型数据的长度,sizeof 是"求字节数运算符"。

(2) 第 11 行定义一个 creat 函数,它是指针类型,即此函数带回一个指针值,它指向一个 struct student 类型数据。实际上此 creat 函数带回一个链表起始地址。

(3) malloc(LEN)的作用是开辟一个长度为 LEN 的内存区,LEN 已定义为 sizeof (struct student),即结构体 struct student 的长度。malloc 带回的是不指向任何类型数据的指针(void *)。而 p1、p2 是指向 struct student 类型数据的指针变量,因此必须用强制类型转换的方法使指针的基类型改变为 struct student 类型,在 malloc(LEN)之前加了"(struct student *)",它的作用是使 malloc 返回的指针转换为指向 struct student 类型数据的指针。注意"*"号不可省略,否则变成转换成 struct student 类型了,而不是指针

类型了。

(4) 最后一行 return 后面的参数是 head(head 已定义为指针变量,指向 struct student 类型数据)。因此函数返回的是 head 的值,也就是链表的头地址。

(5) n 是结点个数。

(6) 这个算法的思路是让 p1 指向新开辟的结点,p2 指向链表中最后一个结点,把 p1 所指的结点连接在 p2 所指的结点后面,用"p2->next=p1"来实现。

我们对建立链表过程做了比较详细的介绍,读者如果对建立链表的过程比较清楚的话,对下面介绍的删除和插入过程也就比较容易理解了。

11.7.5 输出链表

将链表中各结点的数据依次输出。这个问题比较容易处理。例 11.7 中已初步介绍了输出链表的方法。首先要知道链表第一个结点的地址,也就是要知道 head 的值。然后设一个指针变量 p,先指向第一个结点,输出 p 所指的结点,然后使 p 后移一个结点,再输出,直到链表的尾结点。

例 11.9 编写一个输出链表的函数 print。

```
void print(struct student * head)
 {struct student * p;
  printf("\nNow,These %d records are:\n",n);
  p=head;
  if(head!=NULL)
    do
      {printf("%ld %5.1f\n",p->num,p->score);
       p=p->next;
      }while(p!=NULL);
 }
```

算法可用图 11-17 表示。其过程可用图 11-18 表示。p 先指向第一个结点,在输出完第一个结点之后,p 移到图中 p′虚线位置,指向第二个结点。程序中"p=p->next;"的作用是将 p 原来所指向的结点中 next 的值赋给 p,而 p->next 的值就是第二个结点的起始地址。将它赋给 p,就是使 p 指向第二个结点。

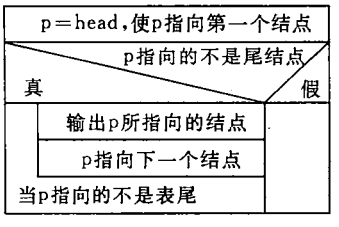


图 11-17

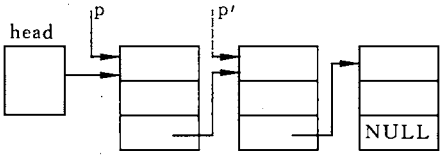
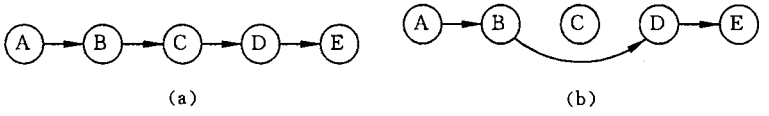


图 11-18

head 的值由实参传过来，也就是将已有的链表的头指针传给被调用的函数，在 print 函数中从 head 所指的第一个结点出发顺序输出各个结点。

11.7.6 对链表的删除操作

已有一个链表，希望删除其中某个结点。怎样考虑此问题的算法呢？先打个比方：一队小孩(A、B、C、D、E)手拉手，如果某一小孩(C)想离队有事，而队形仍保持不变。只要将 C 的手从两边脱开，B 改为与 D 拉手即可，见图 11-19。图 11-19(a)是原来的队伍，图 11-19(b)是 C 离队后的队伍。



(a) (b)

图 11-19

与此相仿，从一个动态链表中删去一个结点，并不是真正从内存中把它抹掉，而是把它从链表中分离开来，只要撤销原来的链接关系即可。

例 11.10 写一函数以删除动态链表中指定的结点。

以指定的学号作为删除结点的标志。例如，输入 10103 表示要求删除学号为 10103 的结点。解题的思路是这样的：从 p 指向的第一个结点开始，检查该结点中的 num 值是否等于输入的要求删除的那个学号。如果相等就将该结点删除，如不相等，就将 p 后移一个结点，再如此进行下去，直到遇到表尾为止。

可以设两个指针变量 p1 和 p2，先使 p1 指向第一个结点(图 11-20(a))。如果要删

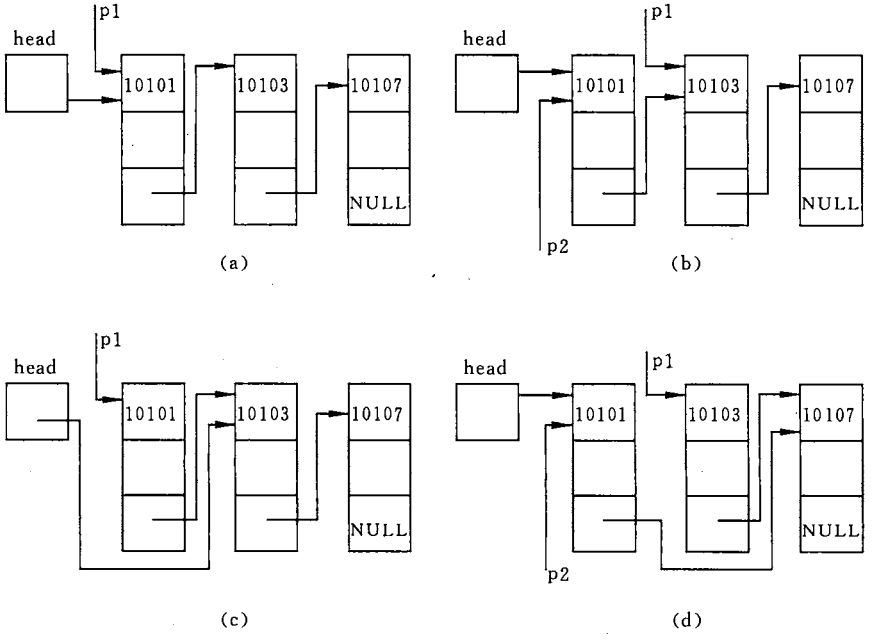


图 11-20

除的不是第一个结点，则使 p1 后移指向下一个结点（将 p1－＞next 赋给 p1），在此之前应将 p1 的值赋给 p2，使 p2 指向刚才检查过的那个结点，见图 11-20(b)。如此一次一次地使 p1 后移，直到找到所要删除的结点或检查完全部链表都找不到要删除的结点为止。如果找到某一结点是要删除的结点，还要区分两种情况：①要删的是第一个结点（p1 的值等于 head 的值，如图 11-20(a)那样），则应将 p1－＞next 赋给 head，见图 11-20(c)。这时 head 指向原来的第二个结点。第一个结点虽然仍存在，但它已与链表脱离，因为链表中没有一个结点或头指针指向它。虽然 p1 还指向它，它仍指向第二个结点，但仍无济于事，现在链表的第一个结点是原来的第二个结点，原来第一个结点已“丢失”，即不再是链表中的一部分了。②如果要删除的不是第一个结点，则将 p1－＞next 赋给 p2－＞next，见图11-20(d)。p2－＞next 原来指向 p1 指向的结点（图中第二个结点），现在 p2－＞next 改为指向 p1 －＞next 所指向的结点（图中第三个结点）。p1 所指向的结点不再是链表的一部分。

还需要考虑链表是空表（无结点）和链表中找不到要删除的结点的情况。

图 11-21 表示解此题的算法。

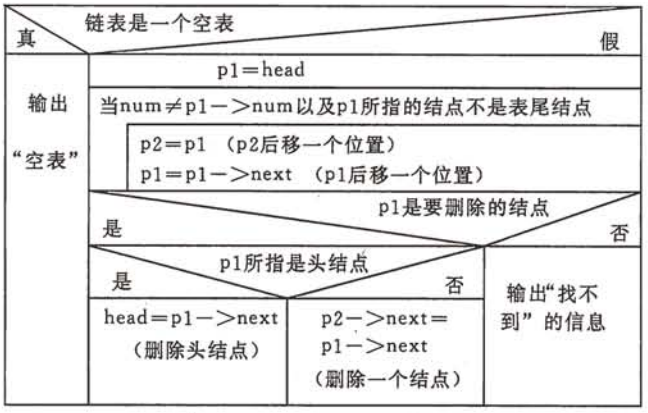


图 11-21

删除结点的函数 del 如下：

```
struct student *del(struct student *head,long num)
{struct student *p1,*p2;
 if (head==NULL)  {printf("\nlist null! \n");goto end;}
 p1=head;
 while(num!=p1->num && p1->next!==NULL)
                  /*p1 指向的不是所要找的结点,并且后面还有结点*/
    {p2=p1;p1=p1->next;}  /*p1 后移一个结点 */
 if (num==p1->num)          /*找到了*/
    {if(p1==head)head=p1->next;
                  /*若 p1 指向的是首结点,把第二个结点地址赋予 head*/
     else p2->next=p1->next;      /*否则将下一结点地址赋给前一结点地址*/
     printf("delete:%ld\n",num);
     n=n-1;
```

```
    }
  else printf("%ld not been found! \n",num); /* 找不到该结点 */
  end;
  return(head);
}
```

函数的类型是指向 struct student 类型数据的指针，它的值是链表的头指针。函数参数为 head 和要删除的学号 num。head 的值可能在函数执行过程中被改变(当删除第一个结点时)。

11.7.7 对链表的插入操作

对链表的插入是指将一个结点插入到一个已有的链表中。

若已有一个学生链表，各结点是按其成员项 num(学号)的值由小到大顺序排列的。今要插入一个新生的结点，要求按学号的顺序插入。

为了能做到正确插入，必须解决两个问题：① 怎样找到插入的位置；② 怎样实现插入。

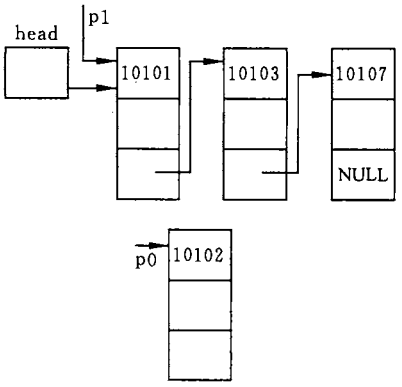
如果有一群小学生，按身高顺序(由低到高)手拉手排好队。现在来了一名新同学，要求按身高顺序插入队中。首先要确定插到什么位置。可以将新同学先与队中第 1 名小学生比身高，若新同学比第 1 名学生高，就使新同学后移一个位置，与第 2 名学生比，如果仍比第 2 名学生高，再往后移，与第 3 名学生比……直到出现比第 i 名学生高，比第 i+1 名学生低的情况为止。显然，新同学的位置应该在第 i 名学生之后，在第 i+1 名学生之前。在确定了位置之后，让第 i 名学生与第 i+1 名学生的手脱开，然后让第 i 名学生的手去拉新同学的手，让新同学另外一只手去拉第 i+1 名学生的手。这样就完成了插入，形成了新的队列。

根据这个思路来实现链表的插入操作。先用指针变量 p0 指向待插入的结点，p1 指向第一个结点，见图 11-22(a)。将 p0－＞num 与 p1－＞num 相比较，如果 p0－＞num＞p1－＞num，则待插入的结点不应插在 p1 所指的结点之前。此时将 p1 后移，并使 p2 指向刚才 p1 所指的结点，见图 11-22(b)。再将 p1－＞num 与 p0－＞num 比，如果仍然是 p0 －＞num 大，则应使 p1 继续后移，直到 p0－＞num≤p1－＞num 为止。这时将 p0 所指的结点插到 p1 所指结点之前。但是如果 p1 所指的已是表尾结点，则 p1 就不应后移了。如果 p0－＞num 比所有结点的 num 都大，则应将 p0 所指的结点插到链表末尾。

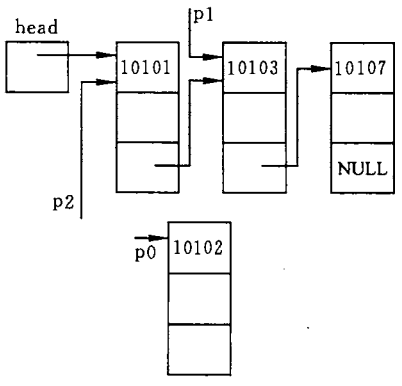
如果插入的位置既不在第一个结点之前，又不在表尾结点之后，则将 p0 的值赋给 p2－＞next，使 p2－＞next 指向待插入的结点，然后将 p1 的值赋给 p0－＞next，使得 p0－＞next指向 p1 指向的变量，见图 11-22(c)。可以看到，在第一个结点和第二个结点之间已插入了一个新的结点。

如果插入位置为第一个结点之前(即 p1 等于 head 时)，则将 p0 赋给 head，将 p1 赋给 p0－＞next，见图 11-22(d)。如果要插到表尾之后，应将 p0 赋给 p1－＞next，NULL 赋给 p0－＞next，见图 11-22(e)。

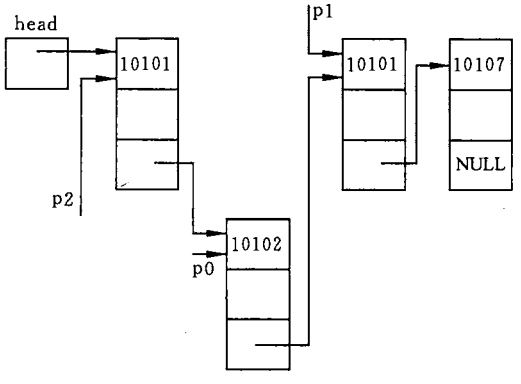
以上算法可用图 11-23 表示。



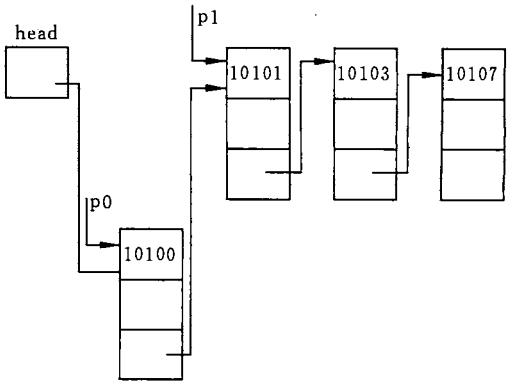
(a)



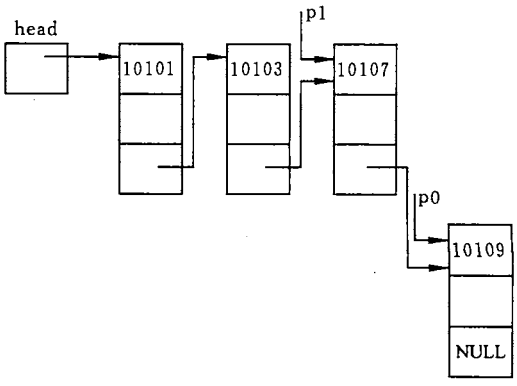
(b)



(c)



(d)



(e)

图 11-22

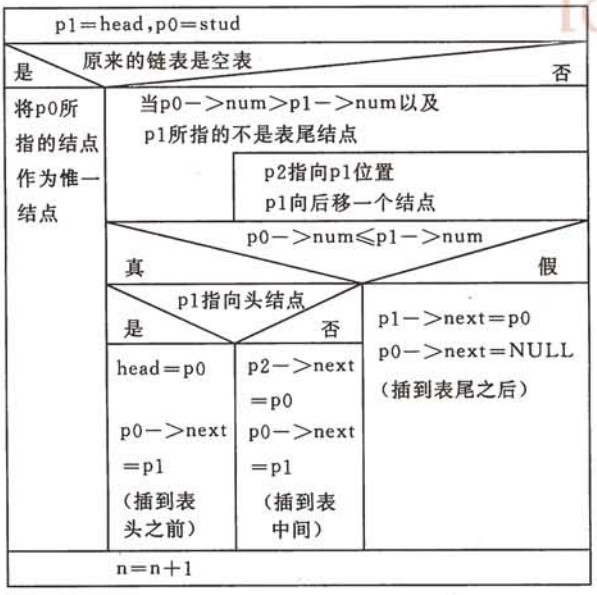


图 11-23

例 11.11 插入结点的函数 insert 如下。

```
struct student * insert(struct student * head, struct student * stud)
  {struct student * p0, * p1, * p2;
   p1=head;                          /* 使 p1 指向第一个结点 */
   p0=stud;                          /* p0 指向要插入的结点 */
   if(head==NULL)                    /* 原来的链表是空表 */
     {head=p0; p0->next=NULL;} /* 使 p0 指向的结点作为头结点 */
   else
     {while((p0->num>p1->num) && (p1->next!=NULL))
       {p2=p1;                       /* 使 p2 指向刚才 p1 指向的结点 */
        p1=p1->next;}                /* p1 后移一个结点 */
        if(p0->num<=p1->num)
          {if(head==p1) head=p0; /* 插到原来第一个结点之前 */
           else p2->next=p0;         /* 插到 p2 指向的结点之后 */
           p0->next=p1;}
        else
          {p1->next=p0; p0->next=NULL;} /* 插到最后的结点之后 */
       }
      n=n+1;                         /* 结点数加 1 */
      return(head);
     }
```

函数参数是 head 和 stud。stud 也是一个指针变量，从实参传来待插入结点的地址给 stud。语句“p0=stud;”的作用是使 p0 指向待插入的结点。函数类型是指针类型，函数值是链表起始地址 head。

11.7.8 对链表的综合操作

将以上建立、输出、删除、插入的函数组织在一个 C 程序中，即将例 11.8～例 11.11

中的4个函数顺序排列，用main函数作主调函数。可以写出以下main函数(main函数的位置在以上各函数的后面)。

```
void main()
  {struct student *head,stu;
   long del_num;
   printf("input records:\n");
   head=creat();                          /*建立链表,返回头指针*/
   print(head);                           /*输出全部结点*/
   printf("\ninput the deleted number:");
   scanf("%ld",&del_num);                 /*输入要删除的学号*/
   head=del(head,del_num);                /*删除后链表的头地址*/
   print(head);                           /*输出全部结点*/
   printf("\ninput the inserted record:"); /*输入要插入的结点*/
   scanf("%ld,%f",&stu.num,&stu.score);
   head=insert(head,&stu);                /*插入一个结点,返回头结点地址*/
   print(head);                           /*输出全部结点*/
  }
```

此程序运行结果是正确的。它只删除一个结点，插入一个结点。但如果想再插入一个结点，重复写上程序最后4行，共插入两个结点，运行结果却是错误的。

运行结果如下：

```
input records:(建立链表)
10101,90↙
10103,98↙
10105,76↙
0,0↙

Now,These 3 records are:
10101    90.0
10103    98.0
10105    76.0

input the deleted number:10103↙  (删除)
delete:10103

Now,These 2 records are:
10101    90.0
10105    76.0

input the inserted record:10102,90↙   (插入第一个结点)

Now,These 3 records are:
10101    90.0
10102    90.0
10105    76.0

input the inserted record:10104,99↙  (插入第二个结点)

Now,These 4 records are:
10101    90.0
```

```
10104    99.0
10104    99.0
10104    99.0
  ⋮
(无终止地输出 10104 的结点数据)
```

请读者将 main 与 insert 函数结合起来考察为什么会产生以上运行结果。

出现以上结果的原因是:stu 是一个有固定地址的结构体变量。第一次把 stu 结点插入到链表中,第二次若再用它来插入第二个结点,就把第一次结点的数据冲掉了,实际上并没有开辟两个结点。读者可根据 insert 函数画出此时链表的情况。为了解决这个问题,必须在每插入一个结点时新开辟一个内存区。我们修改 main 函数,使之能删除多个结点(直到输入要删的学号为 0),能插入多个结点(直到输入要插入的学号为 0)。

main 函数如下:

```
void main()
  {struct student *head, *stu;
   long del_num;
   printf("input records:\n");
   head=creat();
   print(head);
   printf("\ninput the deleted number:");
   scanf("%ld",&del_num);
   while(del_num! =0)
     {head=del(head,del_num);
      print(head);
      printf("input the deleted number:");
      scanf("%ld",&del_num);}
   printf("\ninput the inserted record:");
   stu=(struct student *)malloc(LEN);
   scanf("%ld,%f",&stu->num,&stu->score);
   while(stu->num!=0)
     {head=insert(head,stu);
      print(head);
      printf("input the inserted record:");
      stu=(struct student *)malloc(LEN);
      scanf("%ld,%f",&stu->num,&stu->score);
     }
  }
```

stu 定义为指针变量,在需要插入时先用 malloc 函数开辟一个内存区,将其起始地址经强制类型转换后赋给 stu,然后输入此结构体变量中各成员的值。对不同的插入对象,stu 的值是不同的,每次指向一个新的 struct student 变量。在调用 insert 函数时,实参为 head 和 stu,将已建立的链表起始地址传给 insert 函数的形参,将 stu(即新开辟的单元的地址)传给形参 stud,返回的函数值是经过插入之后的链表的头指针(地址)。

运行情况如下:

```
input records:
10101,99↙
10103,87↙
10105,77↙
0,0↙

Now,These 3 records are:
10101    99.0
10103    87.0
10105    77.0

input the deleted number:10103↙
delete:10103

Now,These 2 records are:
10101    99.0
10105    77.0

input the deleted number:10105↙
delete:10105

Now,These 1 records are:
10101    99.0

input the deleted number:0↙

input the inserted record:10104,87↙

Now,These 2 records are:
10101    99.0
10104    87.0

input the inserted record:10106,65↙

Now,These 3 records are:
10101    99.0
10104    87.0
10106    65.0

input the inserted record:0,0↙
```

对这个程序请读者仔细消化。

结构体和指针的应用领域很宽广，除了单向链表之外，还有环形链表和双向链表。此外还有队列、树、栈、图等数据结构。有关这些问题的算法可以学习“数据结构”课程，在此不作详述。

11.8 共 用 体

11.8.1 共用体的概念

有时需要使几种不同类型的变量存放到同一段内存单元中。例如，可把一个整型变量、一个字符型变量、一个实型变量放在同一个地址开始的内存单元中(见图 11-24)。以上 3 个变量在内存中占的字节数不同，但都从同一地址开始(图中设地址为 1000)存放。

也就是使用覆盖技术，几个变量互相覆盖。这种使几个不同的变量共占同一段内存的结构，称为“共用体”类型的结构。

定义共用体类型变量的一般形式为：

union 共用体名

{ 成员表列

}变量表列；

例如：

```
union data
{int i;
 char ch;
 float f;
}a,b,c;
```

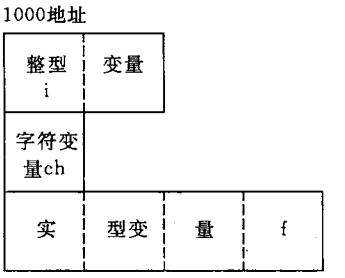


图 11-24

也可以将类型声明与变量定义分开：

```
union data
{int i;
 char ch;
 float f;
};
union data a,b,c;
```

即先声明一个 union data 类型，再将 a、b、c 定义为 union data 类型。当然也可以直接定义共用体变量，例如：

```
union
{int i;
 char ch;
 float f;
}a,b,c;
```

可以看到，“共用体”与“结构体”的定义形式相似。但它们的含义是不同的。

结构体变量所占内存长度是各成员占的内存长度之和。每个成员分别占有其自己的内存单元。

共用体变量所占的内存长度等于最长的成员的长度。例如，上面定义的“共用体”变量 a、b、c 各占 4 个字节(因为一个实型变量占 4 个字节)，而不是各占 2+1+4=7 个字节。

国内有些 C 语言的书把 union 直译为“联合”。作者认为，译为“共用体”更能反映这种结构的特点，即几个变量共用一个内存区。而“联合”这一名词，在一般意义上容易被理解为“将两个或若干个变量联结在一起”，难以表达这种结构的特点。日本就是用“共用体”这一术语的。但是读者应当知道“共用体”在一些书中也被称为“联合”。在阅读其他书籍时如遇“联合”一词，应理解为“共用体”。

11.8.2 共用体变量的引用方式

只有先定义了共用体变量才能引用它，而且不能引用共用体变量，而只能引用共用体

变量中的成员。例如，前面定义了 a、b、c 为共用体变量，下面的引用方式是正确的：

```
a.i    (引用共用体变量中的整型变量 i)
a.ch   (引用共用体变量中的字符变量 ch)
a.f    (引用共用体变量中的实型变量 f)
```

不能只引用共用体变量，例如：

```
printf("%d",a);
```

是错误的，a 的存储区有好几种类型，分别占不同长度的存储区，仅写共用体变量名 a，难以使系统确定究竟输出的是哪一个成员的值。应该写成“printf("%d",a.i);”或“printf("%c",a.ch);”等。

11.8.3 共用体类型数据的特点

在使用共用体类型数据时要注意以下一些特点：

(1) 同一个内存段可以用来存放几种不同类型的成员，但在每一瞬时只能存放其中一种，而不是同时存放几种。也就是说，每一瞬时只有一个成员起作用，其他的成员不起作用，即不是同时都存在和起作用。

(2) 共用体变量中起作用的成员是最后一次存放的成员，在存入一个新的成员后原有的成员就失去作用。例如有以下赋值语句：

```
a.i=1;
a.c='a';
a.f=1.5;
```

在完成以上 3 个赋值运算以后，只有 a.f 是有效的，a.i 和 a.c 已经无意义了。此时用“printf("%d",a.i);”是不行的，而用“printf("%f",a.f);”是可以的，因为最后一次的赋值是向 a.f 赋值。因此在引用共用体变量时应十分注意当前存放在共用体变量中的究竟是哪个成员。

(3) 共用体变量的地址和它的各成员的地址都是同一地址。例如，&a、&a.i、&a.c、&a.f 都是同一地址值，其原因是显然的。

(4) 不能对共用体变量名赋值，也不能企图引用变量名来得到一个值，又不能在定义共用体变量时对它初始化。例如，下面这些都是不对的：

```
① union
  {int i;
   char ch;
   float f;
  }a={1,'a',1.5};   (不能初始化)
② a=1;              (不能对共用体变量赋值)
③ m=a;              (不能引用共用体变量名以得到一个值)
```

(5) 不能把共用体变量作为函数参数，也不能使函数带回共用体变量，但可以使用指向共用体变量的指针(与结构体变量这种用法相仿)。

(6) 共用体类型可以出现在结构体类型定义中,也可以定义共用体数组。反之,结构体也可以出现在共用体类型定义中,数组也可以作为共用体的成员。

num	name	sex	job	class(班) / position(职务)
101	Li	f	s	501
102	Wang	m	t	prof

图 11-25

例 11.12 设有若干个人员的数据,其中有学生和教师。学生的数据中包括:姓名、号码、性别、职业、班级。教师的数据包括:姓名、号码、性别、职业、职务。可以看出,学生和教师所包含的数据是不同的。现要求把它们放在同一表格中,见图 11-25。如果 job 项为 s(学生),则第 5 项为 class(班)。即 Li 是 501 班的。如果 job 项是 t(教师),则第 5 项为 position(职务)。Wang 是 prof(教授)。显然对第 5 项可以用共用体来处理(将 calss 和 position 放在同一段内存中)。

要求输入人员的数据,然后再输出。可以写出算法(见图11-26)。按此写出程序。为简化起见,只设两个人(一个学生、一个教师)。

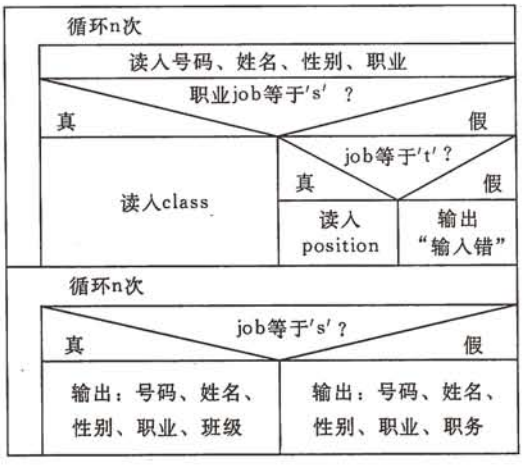


图 11-26

```
#include<stdio.h>
struct
{ int num;
  char name[10];
  char sex;
  char job;
  union
   { int banji;
     char position[10];
   }category;
}person[2];

void main()
{ int i;
  for(i=0;i<2;i++)
    {
      scanf("%d %s %c %c",&person[i].num, &person[i].name,
            &person[i].sex, &person[i].job);
      if(person[i].job=='s')
         scanf("%d", &person[i].category.banji);
      else if (person[i].job=='t')
         scanf("%s",person[i].category.position);
      else printf("input error!");
    }
  printf("\n");
  printf("No.    Name sex job class/position\n");
  for(i=0;i<2;i++)
    { if (person[i].job=='s')
```

```
            printf("%-6d%-10s%-3c%-3c%-6d\n", person[i].num, person[i]. name,
                person[i].sex,person[i].job,person[i].category.banji);
        else
            printf("%-6d%-10s%-3c%-3c%-6s\n", person[i].num, person[i].name,
                person[i].sex,person[i].job,person[i].category.position);
    }
}
```

运行情况如下：

```
101 Li f s 501↙
102 Wang m t professor↙
No.    Name    sex  job  class/position
101    Li      f    s    501
102    Wang    m    t    professor
```

可以看到：在 main 函数之前定义了外部的结构体数组 person，在结构体类型声明中包括了共用体类型，category(分类)是结构体中一个成员名，在这个共用体中成员为 class 和 position，前者为整型，后者为字符数组(存放“职务”的值——字符串)。

11.9 枚举类型

枚举类型是 ANSI C 新标准所增加的。

如果一个变量只有几种可能的值，则可以定义为枚举类型。所谓“枚举”是指将变量的值一一列举出来，变量的值只限于列举出来的值的范围内。

声明枚举类型用 enum 开头。例如：

```
enum weekday{sun,mon,tue,wed,thu,fri,sat};
```

声明了一个枚举类型 enum weekday，可以用此类型来定义变量。例如：

```
enum weekday workday,week_end;
```

workday 和 week_end 被定义为枚举变量，它们的值只能是 sun 到 sat 之一。例如：

```
workday=mon;
week_end=sun;
```

是正确的。当然，也可以直接定义枚举变量，例如：

```
enum{sun,mon,tue,wed,thu,fri,sat} workday,week_end;
```

其中 sun、mon、…、sat 称为枚举元素或枚举常量。它们是用户定义的标识符。这些标识符并不自动地代表什么含义。例如，不能因为写成 sun，就自动代表“星期天”。其实不写 sun 而写成 sunday 也可以。用什么标识符代表什么含义，完全由程序员决定，并在程序中作相应处理。

说明：

(1) 在 C 编译中，对枚举元素按常量处理，故称枚举常量。它们不是变量，不能对它

们赋值。例如：

```
sun=0;mon=1;
```

是错误的。

(2) 枚举元素作为常量，它们是有值的，C 语言编译按定义时的顺序使它们的值为 0，1,2…

在上面定义中，sun 的值为 0，mon 的值为 1……sat 的值为 6。如果有赋值语句：

```
workday=mon;
```

workday 变量的值为 1。这个整数是可以输出的。例如：

```
printf("%d",workday);
```

将输出整数 1。

也可以改变枚举元素的值，在定义时由程序员指定，例如：

```
enum weekday{sun=7,mon=1,tue,wed,thu,fri,sat}workday,week_end;
```

定义 sun 为 7，mon 为 1，以后顺序加 1，sat 为 6。

(3) 枚举值可以用来作判断比较。例如：

```
if(workday==mon)…
if(workday>sun)…
```

枚举值的比较规则是按其在定义时的顺序号比较的。如果定义时未人为指定，则第一个枚举元素的值认作 0，故 mon>sun，sat>fri。

(4) 一个整数不能直接赋给一个枚举变量。例如：

```
workday=2;
```

是不对的。它们属于不同的类型。应先进行强制类型转换才能赋值。例如：

```
workday=(enum weekday)2;
```

它相当于将顺序号为 2 的枚举元素赋给 workday，相当于

```
workday=tue;
```

甚至可以是表达式。例如：

```
workday=(enum weekday)(5-3);
```

例 11.13 口袋中有红、黄、蓝、白、黑 5 种颜色的球若干个。每次从口袋中先后取出 3 个球，问得到 3 种不同色的球的可能取法，输出每种排列的情况。

球只能是 5 种色之一，而且要判断各球是否同色，应该用枚举类型变量处理。

设取出的球为 i、j、k。根据题意，i、j、k 分别是 5 种色球之一，并要求 i≠j≠k。可以用穷举法，即一种可能一种可能地试，看哪一组符合条件。

算法可用图 11-27 表示。

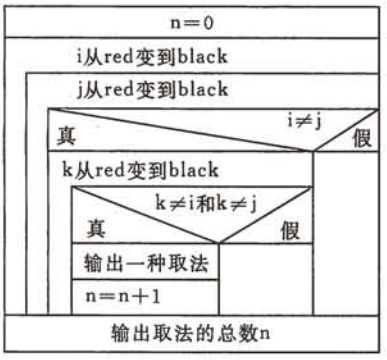


图 11-27

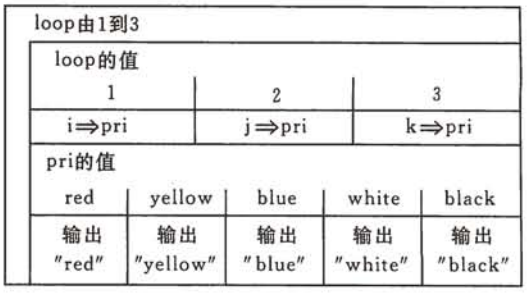


图 11-28

用 n 累计得到 3 种不同色球的次数。外循环使第 1 个球 i 从 red 变到 black。中循环使第 2 个球 j 也从 red 变到 black。如果 i 和 j 同色则不可取，只有 i、j 不同色(i≠j)时才需要继续找第 3 个球，此时第 3 个球 k 也有 5 种可能(red 到 black)，但要求第 3 个球不能与第 1 个球或第 2 个球同色，即 k≠i，k≠j。满足此条件就得到 3 种不同色的球。输出这种 3 色组合方案，然后使 n 加 1。外循环全部执行完后，全部方案就已输出完了。最后输出总数 n。

下面的问题是如何实现图 11-28 中的“输出一种取法”。这里有一个问题：如何输出 red、blue……单词。不能写成 printf("%s",red)来输出“red”字符串。可以采用图 11-28 的方法。

为了输出 3 个球的颜色，显然应经过 3 次循环，第 1 次输出 i 的颜色，第 2 次输出 j 的颜色，第 3 次输出 k 的颜色。在 3 次循环中先后将 i、j、k 赋予 pri。然后根据 pri 的值输出颜色信息。在第 1 次循环时，pri 的值为 i，如果 i 的值为 red，则输出字符串“red”，其他的类推。

程序如下：

```
#include <stdio.h>
void main()
{enum color {red,yellow,blue,white,black};
 enum color i,j,k,pri;
 int n,loop;
 n=0;
 for (i=red;i<=black;i++)
   for (j=red;j<=black;j++)
     if (i!=j)
       { for (k=red;k<=black;k++)
         if ((k!=i) && (k!=j))
           {n=n+1;
            printf("%-4d",n);
            for (loop=1;loop<=3;loop++)
              {switch (loop)
               {case 1: pri=i;break;
```

```
            case 2: pri=j;break;
            case 3: pri=k;break;
            default:break;
           }
          switch (pri)
            {case red:    printf("%-10s","red"); break;
             case yellow: printf("%-10s","yellow"); break;
             case blue:   printf("%-10s","blue"); break;
             case white:  printf("%-10s","white"); break;
             case black:  printf("%-10s","black"); break;
             default :    break;
            }
          }
        printf("\n");
       }
    }
 printf("\ntotal:%5d\n",n);
}
```

运行结果如下：

```
1       red         yellow      blue
2       red         yellow      white
3       red         yellow      black
⋮       ⋮           ⋮           ⋮
58      black       white       red
59      black       white       yellow
60      black       white       blue
total:    60
```

有人说，不用枚举常量而用常数 0 代表“红”，1 代表“黄”……不也可以吗？是的，完全可以。但显然用枚举变量更直观，因为枚举元素都选用了令人“见名知义”的标识符，而且枚举变量的值限制在定义时规定的几个枚举元素范围内，如果赋予它一个其他的值，就会出现出错信息，便于检查。

11.10 用 typedef 定义类型

除了可以直接使用 C 提供的标准类型名（如 int、char、float、double、long 等）和自己声明的结构体、共用体、指针、枚举类型外，还可以用 typedef 声明新的类型名来代替已有的类型名。例如：

```
typedef int         INTEGER;
typedef float       REAL;
```

指定用 INTEGER 代表 int 类型，REAL 代表 float。这样，以下两行等价：

① int i,j; float a,b;

② INTEGER i,j; REAL a,b;

这样可以使熟悉 FORTRAN 的人能用 INTEGER 和 REAL 定义变量，以适应他们的习惯。

如果在一个程序中，一个整型变量用来计数，则：

```
typedef int COUNT;
COUNT i,j;
```

即将变量 i、j 定义为 COUNT 类型，而 COUNT 等价于 int，因此 i、j 是整型。在程序中将 i、j 定为 COUNT 类型，可以使人更一目了然地知道它们是用于计数的。

可以声明结构体类型：

```
typedef struct
{int month;
 int day;
 int year;
}DATE;
```

声明新类型名 DATE，它代表上面指定的一个结构体类型。这时就可以用 DATE 定义变量：

```
DATE  birthday;   (不要写成 struct DATE birthday;)
DATE  *p;         (p 为指向此结构体类型数据的指针)
```

还可以进一步：

```
① typedef int NUM[100];  (声明 NUM 为整型数组类型)
   NUM n;                (定义 n 为整型数组变量)
② typedef char *STRING;  (声明 STRING 为字符指针类型)
   STRING p,s[10];       (p 为字符指针变量，s 为指针数组)
③ typedef int (*POINTER)()(声明 POINTER 为指向函数的指针类型，该函数返回整型值)
   POINTER p1,p2;        (p1、p2 为 POINTER 类型的指针变量)
```

归纳起来，声明一个新的类型名的方法是：

① 先按定义变量的方法写出定义体(如：int i;)。

② 将变量名换成新类型名(例如：将 i 换成 COUNT)。

③ 在最前面加 typedef(例如：typedef int COUNT)。

④ 然后可以用新类型名去定义变量。

再以定义上述的数组类型为例来说明：

① 先按定义数组变量形式书写：int n[100];

② 将变量名 n 换成自己指定的类型名：int NUM[100];

③ 在前面加上 typedef，得到 typedef int NUM[100];

④ 用来定义变量：NUM n;

同样，对字符指针类型，也是：①char *p; ②char *STRING; ③typedef *STRING; ④STRING p,s[10];

习惯上常把用 typedef 声明的类型名用大写字母表示，以便与系统提供的标准类型

标识符相区别。

说明：

(1) 用 typedef 可以声明各种类型名，但不能用来定义变量。用 typedef 可以声明数组类型、字符串类型，使用比较方便。例如定义数组，原来是用

```
int a[10],b[10],c[10],d[10];
```

由于都是一维数组，大小也相同，可以先将此数组类型声明为一个名字：

```
typedef int ARR[10];
```

然后用 ARR 去定义数组变量：

```
ARR a,b,c,d;
```

ARR 为数组类型，它包含 10 个元素。因此，a、b、c、d 都被定义为一维数组，含 10 个元素。

可以看到，用 typedef 可以将数组类型和数组变量分离开来，利用数组类型可以定义多个数组变量。同样可以定义字符串类型、指针类型等。

(2) 用 typedef 只是对已经存在的类型增加一个类型名，而没有创造新的类型。例如，前面声明的整型类型 COUNT，它无非是对 int 型另给一个新名字。又如：

```
typedef int NUM[10];
```

无非是把原来用“int n[10];”定义的数组变量的类型用一个新的名字 NUM 表示出来。无论用哪种方式定义变量，效果都是一样的。

(3) typedef 与 #define 有相似之处，例如：

```
typedef int COUNT;
```

和

```
#define COUNT int
```

的作用都是用 COUNT 代表 int。但事实上，它们二者是不同的。#define 是在预编译时处理的，它只能作简单的字符串替换，而 typedef 是在编译时处理的。实际上它并不是作简单的字符串替换，例如：

```
typedef int NUM[10];
```

并不是用“NUM[10]”去代替“int”，而是采用如同定义变量的方法那样来声明一个类型（就是前面介绍过的将原来的变量名换成类型名）。

(4) 当不同源文件中用到同一类型数据（尤其是像数组、指针、结构体、共用体等类型数据）时，常用 typedef 声明一些数据类型，把它们单独放在一个文件中，然后在需要用到它们的文件中用 #include 命令把它们包含进来。

(5) 使用 typedef 有利于程序的通用与移植。有时程序会依赖于硬件特性，用 typedef便于移植。例如，有的计算机系统 int 型数据用两个字节，数值范围为 −32768～32767，而另外一些机器则以 4 个字节存放一个整数，数值范围为±21 亿。如果

把一个C程序从一个以4个字节存放整数的计算机系统移植到以2个字节存放整数的系统，按一般办法需要将定义变量中的每个int改为long。例如，将“int a,b,c;”改为“long a,b,c;”，如果程序中有多处用int定义变量，则要改动多处。现可以用一个INTEGER来声明int：

```
typedef int INTEGER;
```

在程序中所有整型变量都用INTEGER定义。在移植时只需改动typedef定义体即可：

```
typedef long INTEGER;
```

习　题

11.1 定义一个结构体变量(包括年、月、日)。计算该日在本年中是第几天？注意闰年问题。

11.2 写一个函数days，实现题11.1的计算。由主函数将年、月、日传递给days函数，计算后将日子数传回主函数输出。

11.3 编写一个函数print，打印一个学生的成绩数组，该数组中有5个学生的数据记录，每个记录包括num、name、score[3]，用主函数输入这些记录，用print函数输出这些记录。

11.4 在题11.3的基础上，编写一个函数input，用来输入5个学生的数据记录。

11.5 有10个学生，每个学生的数据包括学号、姓名、3门课程的成绩，从键盘输入10个学生数据，要求输出3门课程总平均成绩，以及最高分的学生的数据(包括学号、姓名、3门课程成绩、平均分数)。

11.6 编写一个函数new，对n个字符开辟连续的存储空间，此函数应返回一个指针(地址)，指向字符串开始的空间。new(n)表示分配n个字节的内存空间，见图11-29。

11.7 写一函数free，将题11.6用new函数占用的空间释放。free(p)表示将p(地址)指向的单元以后的内存段释放。

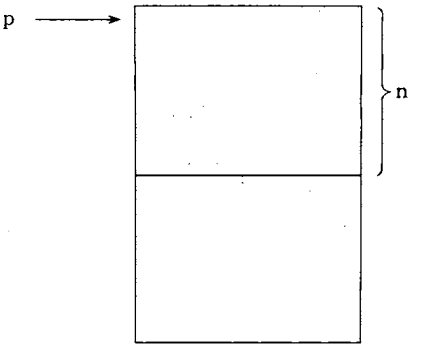


图　11-29

11.8 已有a、b两个链表，每个链表中的结点包括学号、成绩。要求把两个链表合并，按学号升序排列。

11.9 13个人围成一圈，从第1个人开始顺序报号1、2、3。凡报到3者退出圈子。找出最后留在圈子中的人原来的序号。

11.10 有两个链表a和b，设结点中包含学号、姓名。从a链表中删去与b链表中有相同学号的那些结点。

11.11 建立一个链表，每个结点包括：学号、姓名、性别、年龄。输入一个年龄，如果链表中的结点所包含的年龄等于此年龄，则将此结点删去。

11.12 将一个链表按逆序排列，即将链头当链尾，链尾当链头。

第12章 位 运 算

第10章介绍的指针运算和本章将介绍的位运算很适合于编写系统软件的需要，是C语言的重要特色。在计算机用于检测和控制领域中要用到位运算的知识，因此读者应当学习和掌握本章的内容。

所谓位运算是指进行二进制位的运算。在系统软件中，常要处理二进制位的问题。例如，将一个存储单元中的各二进制位左移或右移一位，两个数按位相加等。C语言提供位运算的功能，与其他高级语言(如PASCAL)相比，它显然具有很大的优越性。

12.1 位运算符和位运算

C语言提供如表12-1所列出的位运算符。

表 12-1

运算符	含 义	运算符	含 义
&	按位与	~	取反
|	按位或	<<	左移
∧	按位异或	>>	右移

说明：

(1) 位运算符中除~以外，均为二目(元)运算符，即要求两侧各有一个运算量。

(2) 运算量只能是整型或字符型的数据，不能为实型数据。

下面对各运算符分别介绍。

12.1.1 “按位与”运算符(&)

参加运算的两个数据，按二进制位进行“与”运算。如果两个相应的二进制位都为1，则该位的结果值为1；否则为0。即

0&0=0， 0&1=0， 1&0=0， 1&1=1

例如，3&5并不等于8，应该是按位与：

```
    00000011    (3)
(&) 00000101    (5)
    --------
    00000001    (1)
```

因此，3&5的值得1。如果参加&运算的是负数(如-3 & -5)，则以补码形式表示为二进制数，然后按位进行“与”运算。

按位与有一些特殊的用途：

(1) 清零。如果想将一个单元清零，即使其全部二进制位为 0，只要找一个二进制数，其中各个位符合以下条件：原来的数中为 1 的位，新数中相应位为 0。然后使二者进行 & 运算，即可达到清零目的。

例如，原有数为 00101011，另找一个数，设它为 10010100，它符合以上条件，即在原数为 1 的位置上，它的位值均为 0。将两个数进行 & 运算：

```
    00101011
(&) 10010100
    --------
    00000000
```

其道理是显然的。当然也可以不用 10010100 这个数，而用其他数(如 01000100)也可以，只要符合上述条件即可。

(2) 取一个数中某些指定位。如有一个整数 a(2 个字节)，想要其中的低字节。只需将 a 与$(377)_8$按位与即可，见图 12-1。

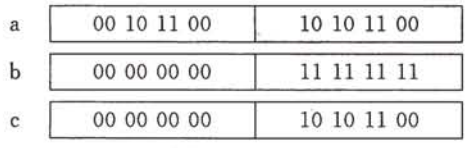


图 12-1

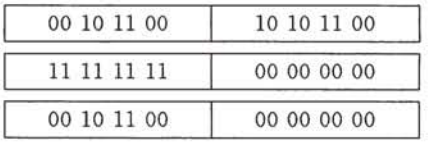


图 12-2

c=a&b，b 为八进制数的 377，运算后 c 只保留 a 的低字节，高字节为 0。

如果想取两个字节中的高字节，只需 c=a & 0177400 (0177400 表示八进制数的 177400)，见图 12-2。

(3) 要想将哪一位保留下来，就与一个数进行 & 运算，此数在该位取 1。例如，有一数 01010100，想把其中左面第 3、4、5、7、8 位保留下来，可以这样运算：

```
    01010100  (十进制数 84)
(&) 00111011  (十进制数 59)
    --------
    00010000  (十进制数 16)
```

即 a=84，b=59，c=a&b=16。

12.1.2 “按位或”运算符(|)

两个相应的二进制位中只要有一个为 1，该位的结果值为 1。即 0|0=0，0|1=1，1|0=1，1|1=1。例如：

```
060|017
```

将八进制数 60 与八进制数 17 进行按位或运算。

```
    00110000
(|) 00001111
    --------
    00111111
```

低 4 位全为 1。如果想使一个数 a 的低 4 位改为 1，只需将 a 与 017 进行按位或运算即可。

按位或运算常用来对一个数据的某些位定值为 1。例如，a 是一个整数(16 位)，有

表达式：

```
a | 0377
```

则低 8 位全置为 1，高 8 位保留原样。

12.1.3 “异或”运算符(∧)

异或运算符∧也称 XOR 运算符。它的规则是：若参加运算的两个二进制位同号，则结果为 0(假)；异号则为 1(真)。即 0∧0=0，0∧1=1，1∧0=1，1∧1=0。例如：

```
  00111001  (十进制数 57，八进制数 071)
(∧) 00101010  (十进制数 42，八进制数 052)
  00010011  (十进制数 19，八进制数 023)
```

即 071∧052，结果为 023(八进制数)。

“异或”的意思是判断两个相应的位值是否为“异”，为“异”(值不同)就取真(1)；否则为假(0)。

下面举例说明∧运算符的应用：

(1) 使特定位翻转

假设有 01111010，想使其低 4 位翻转，即 1 变为 0，0 变为 1。可以将它与 00001111 进行∧运算，即

```
  01111010
(∧) 00001111
  01110101
```

结果值的低 4 位正好是原数低 4 位的翻转。要使哪几位翻转就将与其进行∧运算的该几位置为 1 即可。这是因为原数中值为 1 的位与 1 进行∧运算得 0，原数中的位值 0 与 1 进行∧运算的结果得 1。

(2) 与 0 相∧，保留原值

例如：012∧00=012

```
  00001010
(∧) 00000000
  00001010
```

因为原数中的 1 与 0 进行∧运算得 1，0∧0 得 0，故保留原数。

(3) 交换两个值，不用临时变量

假如 a=3，b=4。想将 a 和 b 的值互换，可以用以下赋值语句实现：

```
a=a∧b;
b=b∧a;
a=a∧b;
```

可以用下面的竖式来说明：

```
        a= 011
(∧)     b= 100
        a= 111   (a∧b 的结果,a 已变成 7)
(∧)     b= 100
        b= 011   (b∧a 的结果,b 已变成 3)
(∧)     a= 111
        a= 100   (a∧b 的结果,a 已变成 4)
```

即等效于以下两步：

① 执行前两个赋值语句："a=a∧b;"和"b=b∧a;"相当于 b=b∧(a∧b)。而 b∧a∧b等于 a∧b∧b。b∧b 的结果为 0,因为同一个数与本身相∧,结果必为 0。因此 b 的值等于 a∧0,即 a,其值为 3。

② 再执行第三个赋值语句：a=a∧b。由于 a 的值等于(a∧b),b 的值等于(b∧a∧b),因此,相当于 a=a∧b∧b∧a∧b,即 a 的值等于 a∧a∧b∧b∧b,等于 b。

a 得到 b 原来的值。

12.1.4 "取反"运算符(～)

～是一个单目(元)运算符,用来对一个二进制数按位取反,即将 0 变 1,将 1 变 0。例如,～025 是对八进制数 25(即二进制数 00010101)按位求反。

```
                0000000000010101
        (～)          ⇓
                1111111111101010
```

即八进制数 177752。因此,～025 的值为八进制数 177752。不要误认为～025 的值是－025。

下面举一例说明～运算符的应用。

若一个整数 a 为 16 位,想使最低一位为 0,可以用

```
a=a & 0177776
```

177776 即二进制数 1111111111111110,如果 a 的值为八进制数 75,a & 0177776 的运算可以表示如下：

```
                0000000000111101
        (&)     1111111111111110
                0000000000111100
```

a 的最后一个二进制位变成 0。但如果将 C 源程序移植到以 32 位存放一个整数的计算机系统(如 VAX 11/780)上,由于一个整数用 4 个字节(32 位表示),想将最后一位变成 0 就不能用 a&0177776 了。读者可以自己算一下,当 a=017776543603 时,a&0177776 的结果是什么?

为了适应以 32 位存放一个整数的计算机系统,应改用

```
a & 037777777776
```

这样改动使移植性差了,可以改用

a=a&~1

它对以 16 位和以 32 位存放一个整数的情况都适用，不必作修改。因为在以 2 个字节存储一个整数时，1 的二进制形式为 0000000000000001，~1 是 1111111111111110(注意~1 不等于-1，弄清~运算符和负号运算符的不同)。在以 4 个字节存储一个整数时，~1 是 11111111111111111111111111111110。

~运算符的优先级别比算术运算符、关系运算符、逻辑运算符和其他位运算符都高，例如，~a&b，先进行~a 运算，然后进行 & 运算。

12.1.5 左移运算符(<<)

用来将一个数的各二进制位全部左移若干位。例如：

a=a<<2

将 a 的二进制数左移 2 位，右补 0。若 a=15，即二进制数 00001111，左移 2 位得 00111100，即十进制数 60(为简单起见，用 8 位二进制数表示十进制数 15，如果用 16 位二进制数表示，结果是一样的)。

高位左移后溢出，舍弃。

左移 1 位相当于该数乘以 2，左移 2 位相当于该数乘以 $2^2=4$。上面举的例子 15<<2=60，即乘了 4。但此结论只适用于该数左移时被溢出舍弃的高位中不包含 1 的情况。例如，假设以一个字节(8 位)存一个整数，若 a 为无符号整型变量，则 a=64 时，左移一位时溢出的是 0，而左移 2 位时，溢出的高位中包含 1。

由表 12-2 可以看出，若 a 的值为 64，在左移 1 位后相当于乘 2，左移 2 位后，值等于 0。

表 12-2

a 的值	a 的二进制形式	a<<1		a<<2	
64	01000000	0	10000000	01	00000000
127	01111111	0	11111110	01	11111100

左移比乘法运算快得多，有些 C 编译程序自动将乘 2 的运算用左移一位来实现，将乘 2^n 的幂运算处理为左移 n 位。

12.1.6 右移运算符(>>)

a>>2 表示将 a 的各二进制位右移 2 位，移到右端的低位被舍弃，对无符号数，高位补 0。例如，a=017 时：

a 的值用二进制形式表示为 00001111，

a>>2 为 00000011|11

↳此 2 位舍弃

右移一位相当于除以 2，右移 n 位相当于除以 2^n。

在右移时，需要注意符号位问题。对无符号数，右移时左边高位移入 0；对于有符号的值，如果原来符号位为 0(该数为正)，则左边也是移入 0，如同上例表示的那样。如果符

号位原来为1(即负数),则左边移入0还是1,要取决于所用的计算机系统。有的系统移入0,有的系统移入1。移入0的称为"逻辑右移",即简单右移;移入1的称为"算术右移"。例如,a的值为八进制数113755:

```
a:        1001011111101101  (用二进制形式表示)
a>>1:     0100101111110110  (逻辑右移时)
a>>1:     1100101111110110  (算术右移时)
```

在有些系统上,a>>1得八进制数045766,而在另一些系统上可能得到的是145766。Turbo C和其他一些C编译采用的是算术右移,即对有符号数右移时,如果符号位原来为1,左面移入高位的是1。

12.1.7 位运算赋值运算符

位运算符与赋值运算符可以组成复合赋值运算符,例如:&=,|=,>>=,<<=,∧=等。

例如,a &= b相当于a = a & b,a <<=2相当于a = a <<2。

12.1.8 不同长度的数据进行位运算

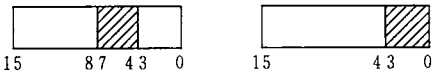
如果两个数据长度不同(例如long型和int型)进行位运算时(如a & b,而a为long型,b为int型),系统会将二者按右端对齐。如果b为正数,则左侧16位补满0;若b为负数,左端应补满1;如果b为无符号整数型,则左侧添满0。

12.2 位运算举例

例12.1 取一个整数a从右端开始的4~7位。

可以这样考虑:

① 先使a右移4位,见图12-3。图12-3(a)是未右移时的情况,(b)图是右移4位后的情况。目的是使要取出的那几位移到最右端。



(a) (b)

图 12-3

右移到右端可以用下面方法实现:

```
a >> 4
```

② 设置一个低4位全为1,其余全为0的数。可用下面方法实现:

```
~ ( ~ 0 << 4 )
```

~0的全部二进制位为1,左移4位,这样右端低4位为0,见下面所示:

```
            0:  0000…000000
           ~0:  1111…111111
      ~0<<4:  1111…110000
~(~0<<4):  0000…001111
```

③ 将上面①、②进行 & 运算。即

```
(a >> 4) & ~ ( ~ 0 << 4 )
```

根据上一节介绍的方法，与低 4 位为 1 的数进行 & 运算，就能将这 4 位保留下来。程序如下：

```
#include <stdio.h>
void main( )
  {unsigned a,b,c,d;
   scanf("%o",&a);
   b=a>>4;
   c=~(~0<<4);
   d=b&c;
   printf("%o, %d\n%o, %d\n",a,a,d,d);
  }
```

运行情况如下：

```
331↙
331, 217    (a 的值)
15, 13      (d 的值)
```

输入 a 的值为八进制数 331，即十进制数 217，其二进制形式为 11011001，经运算最后得到的 d 为 00001101，即八进制数 15，十进制数 13。

可以任意指定从右面第 m 位开始取其右面 n 位。只需将程序中的“b=a>>4”改成“b=a>>(m-n+1)”以及将“c=~(~0<<4)”改成“c=~(~0<<n)”即可。

例 12.2 循环移位。要求将 a 进行右循环移位，见图 12-4。图 12-4表示将 a 右循环移 n 位，即将 a 中原来左面(16-n)位右移 n 位，原来右端 n 位移到最左面 n 位。今假设用两个字节存放一个整数。

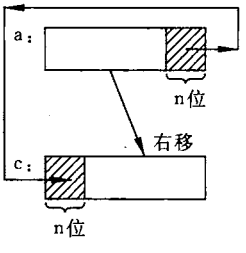


图 12-4

为实现以上目的可以用以下步骤：

① 将 a 的右端 n 位先放到 b 中的高 n 位中，可以用下面语句实现：

```
b=a<<(16-n);
```

② 将 a 右移 n 位，其左面高位 n 位补 0，可以用下面语句实现：

```
c=a>>n;
```

③ 将 c 与 b 进行按位或运算，即

```
c=c|b;
```

程序如下：

```
#include <stdio.h>
void main( )
  {unsigned a,b,c;
   int n;
   scanf("a=%o,n=%d",&a,&n);
   b=a<<(16-n);
   c=a>>n;
   c=c|b;
   printf("%o\n%o",a,c);
  }
```

运行情况如下：

```
a=157653,n=3↙
157653
75765
```

运行开始时输入八进制数 157653，即二进制数 1101111110101011，循环右移 3 位后得二进制数 0111101111110101，即八进制数 75765。

同样可以左循环位移。

12.3 位　　段

以前曾介绍过对内存中信息的存取一般以字节为单位。实际上，有时存储一个信息不必用一个或多个字节，例如，“真”或“假”用 0 或 1 表示，只需 1 位即可。在计算机用于过程控制、参数检测或数据通信领域时，控制信息往往只占一个字节中的一个或几个二进制位，常常在一个字节中放几个信息。

那么，怎样向一个字节中的一个或几个二进制位赋值和改变它的值呢？可以用以下两种方法。

(1) 可以人为地将一个整型变量 data 分为几部分。例如，a、b、c、d 分别占 2 位、6 位、4 位、4 位(见图 12-5)。如果想将 c 的值变为 12(设 c 原来为 0)，可以这样：

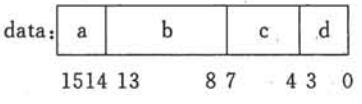


图　12-5

① 将数 12 左移 4 位，使 1100 成为右面起第 4～7 位。

② 将 data 与“12<<4” 进行“按位或” 运算，即可使 c 的值变成 12。

如果 c 的原值不为 0，应先使之为 0。可以用下面方法：

```
data=data & 0177417        (0177417 的最左边的 0 表示 177417 是八进制数)
```

$(0177417)_8$ 的二进制表示为

$$\underbrace{11}_{a}\ \underbrace{111111}_{b}\ \underbrace{0000}_{c}\ \underbrace{1111}_{d}$$

也就是使第 4～7 位全为 0，其他位全为 1。它与 data 进行 & 运算，使第 4～7 位为 0，其余各位保留 data 的原状。

这个 177417 称为“屏蔽字”，即把 c 以外的信息屏蔽起来，不受影响，只使 c 改变为 0。但要找出和记住 177417 这个数比较麻烦。可以用

```
data=data & ~ ( 15 << 4 );
```

15 是 c 的最大值，c 共占 4 位，最大值为 1111，即 15。15<<4 是将 1111 移到 4～7 位，再取反，就使 4～7 位变成 0，其余位全是 1，即

```
15:                0000000000001111
15 << 4:           0000000011110000
~ ( 15 << 4 ):     1111111100001111
```

这样可以实现对 c 清 0，而不必计算屏蔽码。

将上面几步结合起来，可以得到

```
data=data & ~ ( 15 << 4 ) | ( n & 15 ) << 4;
     ──────────
     赋给 4～7 位为 0
```

n 为应赋给 c 的值(例如 12)。n & 15 的作用是只取 n 的右端 4 位的值，其余各位置 0，即把 n 放到最后 4 位上，(n & 15) << 4 就是将 n 置在 4～7 位上，见下面：

```
data & ~(15<<4):   11011011 | 0000 | 1010
(n & 15)<<4    :   00000000 | 1100 | 0000
(按位或运算)       11011011 | 1100 | 1010
```

可见，data 的其他位保留原状未改变，而第 4～7 位改变为 12(即 1100)了。

但是用以上方法给一个字节中某几位赋值太麻烦了。可以用下面介绍的位段结构体的方法。

(2) 位段

C 语言允许在一个结构体中以位为单位来指定其成员所占内存长度，这种以位为单位的成员称为“位段”或称“位域”(bit field) 。利用位段能够用较少的位数存储数据。例如：

```
struct packed_data
{unsigned a:2;
 unsigned b:6;
 unsigned c:4;
 unsigned d:4;
 int i;
}data;
```

见图 12-6。其中 a、b、c、d 分别占 2 位、6 位、4 位、4 位，i 为整型，共占 4 个字节。也可以使各个位段不恰好占满一个字节。例如：

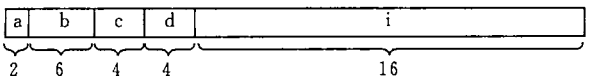


图 12-6

```
struct packed_ data
 {unsigned a:2;
  unsigned b:3;
  unsigned c:4;
  int i;
 };
struct packed_ data data;
```

见图 12-7。其中 a、b、c 共占 9 位，占 1 个字节多，不到 2 个字节，它的后面为 int 型，占 2 个字节。在 a、b、c 之后 7 位空间闲置不用，i 从另一字节开头起存放。

注意，在存储单元中位段的空间分配方向因机器而异。在微机使用的 C 系统中，一般是由右到左进行分配的，如图 12-8 所示。但用户可以不必过问这种细节。

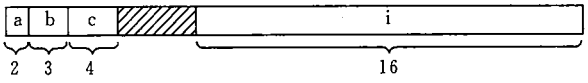


图 12-7

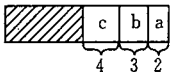


图 12-8

对位段中的数据引用的方法。例如：

```
data. a=2;
data. b=7;
data. c=9;
```

注意位段允许的最大值范围。如果写成

```
data. a=8;
```

就错了。因为 data. a 只占 2 位，最大值为 3。在此情况下，自动取赋予它的数的低位。例如，8 的二进制数形式为 1000，而 data. a 只有 2 位，取 1000 的低 2 位，故 data. a 得值 0。

关于位段的定义和引用，有几点要说明：

(1) 位段成员的类型必须指定为 unsigned 或 int 类型。

(2) 若某一位段要从另一个字开始存放，可以用以下形式定义：

```
unsigned a:1;
unsigned b:2;        } 一个存储单元
unsigned  :0;
unsigned c:3;  (另一存储单元)
```

本来 a、b、c 应连续存放在一个存储单元(字)中，由于用了长度为 0 的位段，其作用是使下一个位段从下一个存储单元开始存放。因此，现在只将 a、b 存储在一个存储单元中，c 另存放在下一个单元(上述“存储单元”可能是一个字节，也可能是 2 个字节，视不同的编译系统而异)。

(3) 一个位段必须存储在同一存储单元中,不能跨两个单元。如果第一个单元空间不能容纳下一个位段,则该空间不用,而从下一个单元起存放该位段。

(4) 可以定义无名位段。例如:

```
unsigned a :1;
unsigned   :2;  (这两位空间不用)
unsigned b :3;
unsigned c :4;
```

见图 12-9。在 a 后面的是无名位段,该空间不用。

(5) 位段的长度不能大于存储单元的长度,也不能定义位段数组。

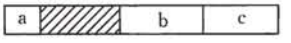


图 12-9

(6) 位段可以用整型格式符输出。例如:

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

当然,也可以用%u、%o、%x 等格式符输出。

(7) 位段可以在数值表达式中引用,它会被系统自动地转换成整型数。例如:

```
data.a+5/data.b
```

是合法的。

习　　题

12.1 编写一个函数 getbits,从一个 16 位的单元中取出某几位(即该几位保留原值,其余位为 0)。函数调用形式为 getbits(value,n1,n2)。

value 为该 16 位(两个字节)中的数据值,n1 为欲取出的起始位,n2 为欲取出的结束位。例如:

```
getbits(0101675,5,8)
```

表示对八进制 101675 这个数,取出它的从左面起第 5 位到第 8 位。

12.2 编写一函数,对一个 16 位的二进制数取出它的奇数位(即从左边起第 1、3、5、…、15 位)。

12.3 编写一程序,检查一下你所用的计算机系统的 C 编译在执行右移时是按照逻辑位移的原则,还是按算术右移原则?如果是逻辑右移,请编一函数实现算术右移?如果是算术右移,请编写一函数以实现逻辑右移。

12.4 编写一函数用来实现左右循环移位。函数名为 move,调用方法为

```
move(value,n)
```

其中 value 为要循环位移的数,n 为位移的位数。例如,n<0 表示为左移;n>0 为右移。n=4表示要右移 4 位;n=-3 表示要左移 3 位。

12.5 设计一个函数,使给出一个数的原码能得到该数的补码。

第13章 文　件

13.1 C文件概述

文件(file)是程序设计中一个重要的概念。所谓“文件”一般指存储在外部介质上数据的集合。一批数据是以文件的形式存放在外部介质(如磁盘)上的。操作系统是以文件为单位对数据进行管理的,也就是说,如果想找存在外部介质上的数据,必须先按文件名找到所指定的文件,然后再从该文件中读取数据。要向外部介质上存储数据也必须先建立一个文件(以文件名标识),才能向它输出数据。

以前各章中所用到的输入和输出,都是以终端为对象的,即从终端键盘输入数据,运行结果输出到终端上。从操作系统的角度看,每一个与主机相联的输入输出设备都看作是一个文件。例如,终端键盘是输入文件,显示屏和打印机是输出文件。

在程序运行时,常常需要将一些数据(运行的最终结果或中间数据)输出到磁盘上存放起来,以后需要时再从磁盘中输入到计算机内存。这就要用到磁盘文件。

C语言把文件看作是一个字符(字节)的序列,即由一个一个字符(字节)的数据顺序组成。根据数据的组织形式,可分为ASCII文件和二进制文件。ASCII文件又称文本(text)文件,它的每一个字节放一个ASCII代码,代表一个字符。二进制文件是把内存中的数据按其在内存中的存储形式原样输出到磁盘上存放。如果有一个整数10 000,在内存中占2个字节,如果按ASCII码形式输出,则占5个字节,而按二进制形式输出,在磁盘上只占2个字节,见图13-1。用ASCII码形式输出与字符一一对应,一个字节代表一个字符,因而便于对字符进行逐个处理,也便于输出字符。但一般占存储空间较多,而且要花费转换时间(二进制形式与ASCII码间的转换)。用二进制形式输出数值,可以节省外存空间和转换时间,但一个字节并不对应一个字符,不能直接输出字符形式。一般中间结果数据需要暂时保存在外存上,以后又需要输入到内存的,常用二进制文件保存。

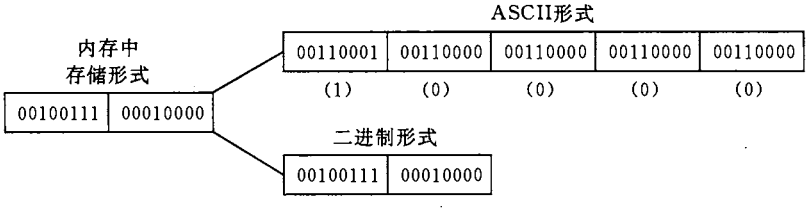


图 13-1

由前所述,一个C文件是一个字节流或二进制流。它把数据看作是一连串的字符(字节),而不考虑记录的界限。换句话说,C语言中文件并不是由记录(record)组成的(这是和PASCAL或其他高级语言不同的)。在C语言中对文件的存取是以字符(字节)

为单位的。输入输出的数据流的开始和结束仅受程序控制而不受物理符号(如回车换行符)控制。也就是说,在输出时不会自动增加回车换行符以作为记录结束的标志,输入时不以回车换行符作为记录的间隔(事实上 C 文件并不由记录构成)。把这种文件称为流式文件。C 语言允许对文件存取一个字符,这就增加了处理的灵活性。

在过去使用的 C 版本(如 UNIX 系统下使用的 C)有两种对文件的处理方法:一种叫"缓冲文件系统",一种叫"非缓冲文件系统"。所谓缓冲文件系统是指系统自动地在内存区为每一个正在使用的文件开辟一个缓冲区。从内存向磁盘输出数据必须先送到内存中的缓冲区,装满缓冲区后才一起送到磁盘去。如果从磁盘向内存读入数据,则一次从磁盘文件将一批数据输入到内存缓冲区(充满缓冲区),然后再从缓冲区逐个地将数据送到程序数据区(给程序变量),见图 13-2。缓冲区的大小由各个具体的 C 版本确定,一般为 512 字节。

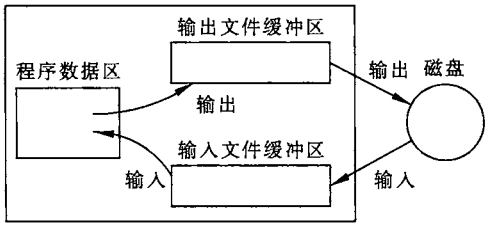


图 13-2

所谓"非缓冲文件系统"是指系统不自动开辟确定大小的缓冲区,而由程序为每个文件设定缓冲区。

在 UNIX 系统下,用缓冲文件系统来处理文本文件,用非缓冲文件系统处理二进制文件。用缓冲文件系统进行的输入输出又称为高级(或高层)磁盘输入输出(高层 I/O),用非缓冲文件系统进行的输入输出又称为低级(低层)输入输出系统。ANSI C 标准不采用非缓冲文件系统,而只采用缓冲文件系统,即既用缓冲文件系统处理文本文件,也用它来处理二进制文件,也就是将缓冲文件系统扩充为可以处理二进制文件。

在 C 语言中,没有输入输出语句,对文件的读写都是用库函数来实现的。ANSI 规定了标准输入输出函数,用它们对文件进行读写。

本章只介绍 ANSI C 规定的缓冲文件系统以及对它的读写。

13.2 文件类型指针

缓冲文件系统中,关键的概念是"文件指针"。每个被使用的文件都在内存中开辟一个区,用来存放文件的有关信息(如文件的名字、文件状态及文件当前位置等)。这些信息是保存在一个结构体变量中的。该结构体类型是由系统定义的,取名为 FILE。Turbo C 在 stdio.h 文件中有以下的文件类型声明:

```
typedef struct
{short  level;                    /* 缓冲区"满"或"空"的程度 */
 unsigned  flags;                 /* 文件状态标志 */
```

```
    char   fd;                          /* 文件描述符 */
    unsigned   char    hold;            /* 如无缓冲区不读取字符 */
    short              bsize;           /* 缓冲区的大小 */
    unsigned   char    * buffer;        /* 数据缓冲区的位置 */
    unsigned   ar      * curp;          /* 指针,当前的指向 */
    unsigned           istemp;          /* 临时文件,指示器 */
    short              token;           /* 用于有效性检查 */
}FILE;
```

有了结构体 FILE 类型之后,可以用它来定义若干个 FILE 类型的变量,以便存放若干个文件的信息。例如,可以定义以下 FILE 类型的数组:

```
FILE f[5];
```

定义了一个结构体数组 f,它有 5 个元素,可以用来存放 5 个文件的信息。

可以定义文件型指针变量。例如:

```
FILE * fp;
```

fp 是一个指向 FILE 类型结构体的指针变量。可以使 fp 指向某一个文件的结构体变量,从而通过该结构体变量中的文件信息能够访问该文件。也就是说,通过文件指针变量能够找到与它相关的文件。如果有 n 个文件,一般应设 n 个指针变量(指向 FILE 类型结构体的指针变量),使它们分别指向 n 个文件(确切地说,指向存放该文件信息的结构体变量),以实现对文件的访问。

13.3 文件的打开与关闭

和其他高级语言一样,对文件读写之前应该"打开"该文件,在使用结束之后应"关闭"该文件。

13.3.1 文件的打开(fopen 函数)

ANSI C 规定了标准输入输出函数库,用 fopen( )函数来实现打开文件。fopen 函数的调用方式通常为

FILE * fp;

fp=fopen(文件名,使用文件方式);

例如:

```
fp=fopen("a1","r");
```

表示要打开名字为 a1 的文件,使用文件方式为"读入"(r 代表 read,即读入),fopen 函数带回指向 a1 文件的指针并赋给 fp,这样 fp 就和文件 a1 相联系了,或者说,fp 指向 a1 文件。可以看出,在打开一个文件时,通知编译系统以下 3 个信息:①需要打开的文件名,也就是准备访问的文件的名字;②使用文件的方式("读"还是"写"等);③让哪一个指针变量指向被打开的文件。

使用文件方式见表 13-1。

表 13-1

文件使用方式	含　义
"r"（只读）	为输入打开一个文本文件
"w"（只写）	为输出打开一个文本文件
"a"（追加）	向文本文件尾添加数据
"rb"（只读）	为输入打开一个二进制文件
"wb"（只写）	为输出打开一个二进制文件
"ab"（追加）	向二进制文件尾添加数据
"r+"（读写）	为读写打开一个文本文件
"w+"（读写）	为读写建立一个新的文本文件
"a+"（读写）	为读写打开一个文本文件
"rb+"（读写）	为读写打开一个二进制文件
"wb+"（读写）	为读写建立一个新的二进制文件
"ab+"（读写）	为读写打开一个二进制文件

说明：

(1) 用“r”方式打开的文件只能用于向计算机输入而不能用作向该文件输出数据，而且该文件应该已经存在，不能用“r”方式打开一个并不存在的文件(即输入文件)；否则出错。

(2) 用“w”方式打开的文件只能用于向该文件写数据(即输出文件)，而不能用来向计算机输入。如果原来不存在该文件，则在打开时新建立一个以指定的名字命名的文件。如果原来已存在一个以该文件名命名的文件，则在打开时将该文件删去，然后重新建立一个新文件。

(3) 如果希望向文件末尾添加新的数据(不希望删除原有数据)，则应该用“a”方式打开。但此时该文件必须已存在；否则将得到出错信息。打开时，位置指针移到文件末尾。

(4) 用“r+”、“w+”、“a+”方式打开的文件既可以用来输入数据，也可以用来输出数据。用“r+”方式时该文件应该已经存在，以便能向计算机输入数据。用“w+”方式则新建立一个文件，先向此文件写数据，然后可以读此文件中的数据。用“a+”方式打开的文件，原来的文件不被删去，位置指针移到文件末尾，可以添加，也可以读。

(5) 如果不能实现“打开”的任务，fopen 函数将会带回一个出错信息。出错的原因可能是用“r”方式打开一个并不存在的文件；磁盘出故障；磁盘已满无法建立新文件等。此时 fopen 函数将带回一个空指针值 NULL(NULL 在 stdio.h 文件中已被定义为 0)。

常用下面的方法打开一个文件：

```
if ((fp=fopen("file1","r"))==NULL)
  {printf("cannot open this file\n");
```

```
    exit(0);
}
```

即先检查打开的操作有否出错，如果有错就在终端上输出“cannot open this file”。exit 函数的作用是关闭所有文件，终止正在执行的程序，待用户检查出错误，修改后再运行。

(6) 用以上方式可以打开文本文件或二进制文件，这是 ANSI C 的规定，用同一种缓冲文件系统来处理文本文件和二进制文件。但目前使用的有些 C 编译系统可能不完全提供所有这些功能(例如，有的只能用“r”、“w”、“a”方式)，有的 C 版本不用“r+”、“w+”、“a+”，而用“rw”、“wr”、“ar”等，请读者注意所用系统的规定。

(7) 在向计算机输入文本文件时，将回车换行符转换为一个换行符，在输出时把换行符转换成为回车和换行两个字符。在用二进制文件时，不进行这种转换，在内存中的数据形式与输出到外部文件中的数据形式完全一致，一一对应。

(8) 在程序开始运行时，系统自动打开 3 个标准文件：标准输入、标准输出、标准出错输出。通常这 3 个文件都与终端相联系。因此以前我们所用到的从终端输入或输出都不需要打开终端文件。系统自动定义了 3 个文件指针 stdin、stdout 和 stderr，分别指向终端输入、终端输出和标准出错输出(也从终端输出)。如果程序中指定要从 stdin 所指的文件输入数据，就是指从终端键盘输入数据。

13.3.2 文件的关闭(fclose 函数)

在使用完一个文件后应该关闭它，以防止它再被误用。“关闭”就是使文件指针变量不指向该文件，也就是文件指针变量与文件“脱钩”，此后不能再通过该指针对原来与其相联系的文件进行读写操作，除非再次打开，使该指针变量重新指向该文件。

用 fclose 函数关闭文件。fclose 函数调用的一般形式为

fclose(文件指针);

例如：

```
fclose(fp);
```

前面我们曾把打开文件(用 fopen 函数)时所带回的指针赋给了 fp，今通过 fp 把该文件关闭，即 fp 不再指向该文件。

应该养成在程序终止之前关闭所有文件的习惯，如果不关闭文件将会丢失数据。因为，如前所述，在向文件写数据时，是先将数据输出到缓冲区，待缓冲区充满后才正式输出给文件。如果当数据未充满缓冲区而程序结束运行，就会将缓冲区中的数据丢失。用 fclose 函数关闭文件，可以避免这个问题，它先把缓冲区中的数据输出到磁盘文件，然后才释放文件指针变量。

fclose 函数也带回一个值，当顺利地执行了关闭操作，则返回值为 0；否则返回 EOF(−1)。可以用 ferror 函数来测试(见 13.6.1 节)。

13.4 文件的读写

文件打开之后，就可以对它进行读写了。常用的读写函数如下所述。

13.4.1 fputc 函数和 fgetc 函数(putc 函数和 getc 函数)

1. fputc 函数

把一个字符写到磁盘文件上去。其一般调用形式为

fputc(ch,fp);

其中 ch 是要输出的字符,它可以是一个字符常量,也可以是一个字符变量。fp 是文件指针变量。fputc(ch,fp)函数的作用是将字符(ch 的值)输出到 fp 所指向的文件中去。fputc 函数也带回一个值:如果输出成功,则返回值就是输出的字符;如果输出失败,则返回一个 EOF(即-1)。EOF 是在 stdio.h 文件中定义的符号常量,值为-1。

在第 4 章介绍过 putchar 函数,其实 putchar 是从 fputc 函数派生出来的。putchar(c)是在 stdio.h 文件中用预处理命令 #define 定义的宏:

```
#define putchar(c) fputc(c,stdout)
```

前面已叙述,stdout 是系统定义的文件指针变量,它与终端输出相联。fputc(c,stdout)的作用是将 c 的值输出到终端。用宏 putchar(c)比写 fputc(c,stdout)简单一些。从用户的角度,可以把 putchar(c)看作函数而不必严格地称它为宏。

2. fgetc 函数

从指定的文件读入一个字符,该文件必须是以读或读写方式打开的。fgetc 函数的调用形式为

ch=fgetc(fp);

fp 为文件型指针变量,ch 为字符变量。fgetc 函数带回一个字符,赋给 ch。如果在执行 fgetc 函数读字符时遇到文件结束符,函数返回一个文件结束标志 EOF(即-1)。如果想从一个磁盘文件顺序读入字符并在屏幕上显示出来,可以用:

```
ch=fgetc(fp);
while(ch!=EOF)
  {putchar(ch);
   ch=fgetc(fp);
  }
```

注意:EOF 不是可输出字符,因此不能在屏幕上显示。由于字符的 ASCII 码不可能出现-1,因此 EOF 定义为-1 是合适的。当读入的字符值等于-1(即 EOF)时,表示读入的已不是正常的字符而是文件结束符。但以上只适用于读文本文件的情况。现在 ANSI C 已允许用缓冲文件系统处理二进制文件,而读入某一个字节中的二进制数据的值有可能是-1,而这又恰好是 EOF 的值。这就出现了需要读入有用数据而却被处理为“文件结束”的情况。为了解决这个问题,ANSI C 提供一个 feof 函数来判断文件是否真的结束。feof(fp)用来测试 fp 所指向的文件当前状态是否“文件结束”。如果是文件结束,函数 feof(fp)的值为 1(真);否则为 0(假)。

如果想顺序读入一个二进制文件中的数据,可以用:

```
while(!feof(fp))
  {c=fgetc(fp);
   ⋮
  }
```

当未遇文件结束，feof(fp)的值为 0，!feof(fp)的值为 1，读入一个字节的数据赋给整型变量 c，并接着对其进行所需的处理。直到遇文件结束，feof(fp)值为 1，!feof(fp)值为 0，不再执行 while 循环。

这种方法也适用于文本文件。

3. fputc 和 fgetc 函数使用举例

在掌握了以上几种函数以后，可以编制一些简单的使用文件的程序。

例 13.1 从键盘输入一些字符，逐个把它们送到磁盘上去，直到输入一个“#”为止。

```
#include <stdio.h>
#include <stdlib.h>
void main( )
{FILE *fp;
 char ch,filename[10];
 scanf("%s",filename);
 if ((fp=fopen(filename,"w"))==NULL)
   {printf("cannot open file\n");
    exit(0);}          /*终止程序*/
 ch=getchar();         /* 此语句用来接收在执行 scanf 语句时最后输入的回车符 */
 ch=getchar( );        /* 接收输入的第一个字符 */
 while(ch!='#')
    {
     fputc(ch,fp);putchar(ch);
     ch=getchar();
    }
 putchar(10);          /*向屏幕输出一个换行符*/
 fclose(fp);
}
```

运行情况如下：

```
file1.c↙                (输入磁盘文件名)
computer and c#↙        (输入一个字符串)
computer and c          (输出一个字符串)
```

文件名由键盘输入，赋给字符数组 filename。fopen 函数中的第一个参数“文件名”可以直接写成字符串常量形式(如 file1.c)，也可以用字符数组名，在字符数组中存放文件名(如本例所用的方法)。本例运行时，从键盘输入磁盘文件名“file1.c”，然后输入要写入该磁盘文件的字符“computer and c”，“#”是表示输入结束，程序将“computer and c”写到以“file1.c”命名的磁盘文件中，同时在屏幕上显示这些字符，以便核对。exit 是标准 C 的库

函数，作用是使程序终止，用此函数应当加入 stdlib 头文件。

可以用 DOS 命令输出 file1.c 文件中的内容：

```
C>type file1.c↙
computer and c
```

证明了在 file1.c 文件中已存入了“computer and c”的信息。

例 13.2 将一个磁盘文件中的信息复制到另一个磁盘文件中。

```
#include <stdio.h>
#include <stdlib.h>
void main( )
 {FILE *in, *out;
  char ch,infile[10],outfile[10];
  printf("Enter the infile name:\n");
  scanf("%s",infile);
  printf("Enter the outfile name:\n");
  scanf("%s",outfile);
  if((in=fopen(infile,"r"))==NULL)
    {printf("cannot open infile\n");
     exit(0);
    }
  if((out=fopen(outfile,"w"))==NULL)
    {printf("cannot open outfile\n");
     exit(0);
    }
  while(! feof(in)) fputc(fgetc(in),out);
  fclose(in);
  fclose(out);
 }
```

运行情况如下：

```
Enter the infile name:
file1.c↙          (输入原有磁盘文件名)
Enter the outfile name:
file2.c↙          (输入新复制的磁盘文件名)
```

程序运行结果是将 file1.c 文件中的内容复制到 file2.c 中去。可以用下面 DOS 命令验证：

```
C>type file1.c↙
computer and c          (file1.c 中的信息)
C>type file2.c↙
computer and c          (file2.c 中的信息)
```

以上程序是按文本文件方式处理的。也可以用此程序来复制一个二进制文件，只需将两个 fopen 函数中的“r”和“w”分别改为“rb”和“wb”即可。

也可以在输入命令行时把两个文件名一起输入。这时要用到 main 函数的参数。程

序可改为：

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc,char *argv[])
 {FILE *in,*out;
  char ch;
  if (argc!=3)
    {printf("You forgot to enter a filename\n");
     exit(0);
    }
  if ((in=fopen(argv[1],"r"))==NULL)
    {printf("cannot open infile\n");
     exit(0);
    }
  if ((out=fopen(argv[2],"w"))==NULL)
    {printf("cannot open outfile\n");
     exit(0);
    }
  while(!feof(in)) fputc(fgetc(in),out);
  fclose(in);
  fclose(out);
 }
```

假若本程序的源文件名为 a.c，经编译连接后得到的可执行文件名为 a.exe，则在 DOS 命令工作方式下，可以输入以下的命令行：

```
C>a file1.c file2.c↙
```

即在输入可执行文件名后，再输入两个参数 file1.c 和 file2.c，分别输入到 argv[1]和 argv[2]中，argv[0]的内容为 a，argc 的值等于 3(因为此命令行共有 3 个参数)。如果输入的参数少于 3 个，则程序会输出："你忘了输入一个文件名"。程序执行结果是将 file1.c 中的信息复制到 file2.c 中。可以用以下命令验证：

```
C>type file1.c↙
computer and c
```

(这是 file1.c 文件中的信息)

```
C>type file2.c↙
computer and c
```

(这是 file2.c 文件中的信息。可见 file1.c 已复制到 file2.c 中了)。

最后说明一点，为了书写方便，系统把 fputc 和 fgetc 定义为宏名 putc 和 getc：

```
#define putc(ch,fp) fputc(ch,fp)
#define getc(fp) fgetc(fp)
```

这是在 stdio.h 中定义的。因此，用 putc 和 fputc 及用 getc 和 fgetc 是一样的。一般可以把它们作为相同的函数来对待。

13.4.2 fread 函数和 fwrite 函数

用 getc 和 putc 函数可以用来读写文件中的一个字符。但是常常要求一次读入一组数据(例如,一个实数或一个结构体变量的值),ANSI C 标准提出设置两个函数(fread 和 fwrite),用来读写一个数据块。它们的一般调用形式为:

fread(buffer,size,count,fp);

fwrite(buffer,size,count,fp);

其中:

buffer:是一个指针。对 fread 来说,它是读入数据的存放地址。对 fwrite 来说,是要输出数据的地址(以上指的是起始地址)。

size:要读写的字节数。

count:要进行读写多少个 size 字节的数据项。

fp:文件型指针。

如果文件以二进制形式打开,用 fread 和 fwrite 函数就可以读写任何类型的信息,例如:

```
fread(f,4,2,fp);
```

其中 f 是一个实型数组名。一个实型变量占 4 个字节。这个函数从 fp 所指向的文件读入 2 个 4 个字节的数据,存储到数组 f 中。

如果有一个如下的结构体类型:

```
struct student_type
{char name[10];
 int num;
 int age;
 char addr[30];
}stud[40];
```

结构体数组 stud 有 40 个元素,每一个元素用来存放一个学生的数据(包括姓名、学号、年龄、地址)。假设学生的数据已存放在磁盘文件中,可以用下面的 for 语句和 fread 函数读入 40 个学生的数据:

```
for(i=0;i<40;i++)
   fread(&stud[i],sizeof(struct student_type),1,fp);
```

同样,以下 for 语句和 fwrite 函数可以将内存中的学生数据输出到磁盘文件中去:

```
for(i=0;i<40,i++)
  fwrite(&stud[i],sizeof(struct student_type),1,fp);
```

如果 fread 或 fwrite 调用成功,则函数返回值为 count 的值,即输入或输出数据项的完整个数。

下面写出一个完整的程序。

例 13.3 从键盘输入 4 个学生的有关数据,然后把它们转存到磁盘文件上去。

```
#include <stdio.h>
#define SIZE 4
struct student_type
{char name[10];
 int num;
 int age;
 char addr[15];
}stud[SIZE];
void save( )
 {FILE *fp;
  int i;
  if ((fp=fopen("stu_list","wb"))==NULL)
    {printf("cannot open file\n");
     return;
    }
  for(i=0;i<SIZE;i++)
    if (fwrite(&stud[i],sizeof(struct student_type),1,fp)!=1)
      printf("file write error\n");
  fclose(fp)
 }
void main()
 {int i;
  for(i=0;i<SIZE;i++)
    scanf("%s%d%d%s",stud[i].name,&stud[i].num,&stud[i].age,stud[i].addr);
  save( );
 }
```

在 main 函数中，从终端键盘输入 4 个学生的数据，然后调用 save 函数，将这些数据输出到以“stu_list”命名的磁盘文件中。fwrite 函数的作用是将一个长度为 29 字节的数据块送到 stu_list 文件中（一个 student_type 类型结构体变量的长度为它的成员长度之和，即 10+2+2+15=29）。

运行情况如下：

输入 4 个学生的姓名、学号、年龄和地址：

```
Zhang 1001 19 room_101↙
Fun 1002 20 room_102↙
Tan 1003 21 room_103↙
Ling 1004 21 room_104↙
```

程序运行时，屏幕上并无输出任何信息，只是将从键盘输入的数据送到磁盘文件上。为了验证在磁盘文件“stu_list”中是否已存在此数据，可以用以下程序从“stu_list”文件中读入数据，然后在屏幕上输出。

```
#include <stdio.h>
#define SIZE 4
struct student_type
```

```
{char name[10];
 int num;
 int age;
 char addr[15];
}stud[SIZE];
void main( )
{int i;
 FILE *fp;
 fp=fopen("stu_list","rb");
 for(i=0;i<SIZE;i++)
   {fread(&stud[i],sizeof(struct student_type),1,fp);
    printf("%-10s %4d %4d %-15s\n",stud[i].name,stud[i].num,stud[i]. age,stud[i].addr);
   }
 fclose (fp);
}
```

程序运行时不需从键盘输入任何数据。屏幕上显示出以下信息：

```
Zhang   1001   19   room_101
Fun     1002   20   room_102
Tan     1003   21   room_103
Ling    1004   21   room_104
```

请注意输入输出数据的状况。从键盘输入 4 个学生的数据是 ASCII 码，也就是文本文件。在送到计算机内存时，回车和换行符转换成一个换行符。再从内存以“wb”方式（二进制写）输出到“stu_list”文件，此时不发生字符转换，按内存中存储形式原样输出到磁盘文件上。在上面验证程序中，又用 fread 函数从“stu_list”文件向内存读入数据，注意此时用的是“rb”方式，即二进制方式，数据按原样输入，也不发生字符转换。也就是这时候内存中的数据恢复到第一个程序向“stu_list”输出以前的情况。最后在验证程序中，用 printf 函数输出到屏幕，printf 是格式输出函数，输出 ASCII 码，在屏幕上显示字符。换行符又转换为回车加换行符。

如果企图从“stu_list”文件中以“r”方式读入数据就会出错。

fread 和 fwrite 函数一般用于二进制文件的输入输出。因为它们是按数据块的长度来处理输入输出的，在字符发生转换的情况下很可能出现与原设想的情况不同。

例如，如果写

```
fread(&stud[i],sizeof(struct student_type),1,stdin);
```

企图从终端键盘输入数据，这在语法上并不存在错误，编译能通过。如果用以下形式输入数据：

```
Zhang 1001 10 room_101↙
   ⋮
```

由于 fread 函数要求一次输入 29 个字节（而不问这些字节的内容），因此输入数据中的空格也作为输入数据而不作为数据间的分隔符了。连空格也存储到 stud[i]中了，显然是不

对的。

这个题目要求的是从键盘输入数据，如果已有的数据已经以二进制形式存储在一个磁盘文件“stu_dat”中，要求从其中读入数据并输出到“stu_list”文件中，可以编写一个load函数，从磁盘文件中读二进制数据。

```
void load( )
 {FILE * fp;
  int i;
  if ((fp=fopen("stu_dat","rb"))==NULL)
    {printf("cannot open infile\n");
     return;}
  for(i=0;i<SIZE;i++)
    if (fread(&stud[i],sizeof(struct student_type),1,fp)!=1)
      {if(feof(fp)) {fclose(fp); return;}
       printf("file read error\n");
      }
  fclose (fp);
 }
```

将load函数加到本题原来的程序文件中，并将main函数改为

```
main( )
{
 load( );
 save( );
}
```

即可实现题目要求。

13.4.3 fprintf函数和fscanf函数

fprintf函数、fscanf函数与printf函数、scanf函数作用相仿，都是格式化读写函数。只有一点不同：fprintf和fscanf函数的读写对象不是终端而是磁盘文件。它们的一般调用方式为：

fprintf(文件指针，格式字符串，输出表列)；

fscanf (文件指针，格式字符串，输入表列)；

例如：

```
fprintf(fp,"%d,%6.2f",i,t);
```

它的作用是将整型变量i和实型变量t的值按%d和%6.2f的格式输出到fp指向的文件上。如果i=3，t=4.5，则输出到磁盘文件上的是以下的字符串：

```
3,  4.50
```

同样，用以下fscanf函数可以从磁盘文件上读入ASCII字符：

```
fscanf(fp,"%d,%f",&i,&t);
```

磁盘文件上如果有以下字符：

```
3, 4.5
```

则将磁盘文件中的数据 3 送给变量 i，4.5 送给变量 t。

用 fprintf 和 fscanf 函数对磁盘文件读写，使用方便，容易理解，但由于在输入时要将 ASCII 码转换为二进制形式，在输出时又要将二进制形式转换成字符，花费时间比较多。因此，在内存与磁盘频繁交换数据的情况下，最好不用 fprintf 和 fscanf 函数，而用 fread 和 fwrite 函数。

13.4.4 其他读写函数

1. putw 和 getw 函数

大多数 C 编译系统都提供另外两个函数：putw 和 getw，用来对磁盘文件读写一个字（整数）。例如：

```
putw(10,fp);
```

它的作用是将整数 10 输出到 fp 指向的文件。而

```
i=getw(fp);
```

的作用是从磁盘文件读一个整数到内存，赋给整型变量 i。

如果所用的 C 编译的库函数中不包括 putw 和 getw 函数，可以自己定义这两个函数。putw 函数如下：

```
putw(int i,FILE * fp)
{char * s;
 s=&i;
 putc(s[0],fp);putc(s[1],fp);
 return(i);
}
```

当调用 putw 函数时，如果用“putw(10,fp);”语句，形参 i 得到实参传来的值 10，在 putw 函数中将 i 的地址赋予指针变量 s，而 s 是指向字符变量的指针变量，因此 s 指向 i 的第 1 个字节，s+1 指向 i 的第 2 个字节。由于 *(s+0)就是 s[0]，*(s+1)就是 s[1]，因此，s[0]、s[1]分别对应 i 的第 1 字节和第 2 字节。顺序输出 s[0]、s[1]就相当于输出了 i 的两个字节中的内容，见图 13-3。

getw 函数如下：

```
getw(FILE * fp)
{char * s;
 int i;
 s=char * &i;          /* 使 s 指向 i 的起始地址 */
 s[0]=getc(fp);
 s[1]=getc(fp);
 return(i);
}
```

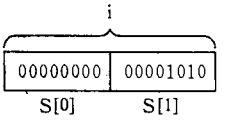


图 13-3

putw 和 getw 并不是 ANSI C 标准定义的函数。许多 C 编译都提供这两个函数，但有的 C 编译可能不以 putw 和 getw 命名此两函数，而用其他函数名，请用时注意。

2. 读写其他类型数据

如果用 ANSI C 提供的 fread 和 fwrite 函数，读写任何类型数据都是十分方便的。如果所用的系统不提供这两个函数，用户只好自己定义所需函数。例如，可以定义一个向磁盘文件写一个实数(用二进制方式)的函数 putfloat：

```
putfloat(float num,FILE *fp)
{char *s;
 int count;
 s=(char *)&num;
 for(count=0;count<4;count++)
   putc(s[count],fp);
}
```

同样可以编写出读写任何类型数据的函数。

3. fgets 函数和 fputs 函数

fgets 函数的作用是从指定文件读入一个字符串。例如：

```
fgets(str,n,fp);
```

n 为要求得到的字符，但只从 fp 指向的文件输入 n－1 个字符，然后在最后加一个'\0'字符，因此得到的字符串共有 n 个字符，把它们放到字符数组 str 中。如果在读完 n－1 个字符之前遇到换行符或 EOF，读入即结束。fgets 函数返回值为 str 的首地址。

fputs 函数的作用是向指定的文件输出一个字符串。例如：

```
fputs("China",fp);
```

把字符串"China"输出到 fp 指向的文件。fputs 函数中第一个参数可以是字符串常量、字符数组名或字符型指针。字符串末尾的'\0'不输出。若输出成功，函数值为 0；失败时，为 EOF。

这两个函数类似以前介绍过的 gets 和 puts 函数，只是 fgets 和 fputs 函数以指定的文件作为读写对象。

13.5 文件的定位

文件中有一个位置指针，指向当前读写的位置。如果顺序读写一个文件，每次读写一个字符，则读写完一个字符后，该位置指针自动移动指向下一个字符位置。如果想改变这样的规律，强制使位置指针指向其他指定的位置，可以用后面介绍的有关函数。

13.5.1 rewind 函数

rewind 函数的作用是使位置指针重新返回文件的开头，此函数没有返回值。

例 13.4 有一个磁盘文件，第一次将它的内容显示在屏幕上，第二次把它复制到另

一文件上。

```
#include<stdio.h>
void main()
 {FILE *fp1,*fp2;
  fp1=fopen("file1.c","r");
  fp2=fopen("file2.c","w");
  while(!feof(fp1)) putchar(getc(fp1));
  rewind(fp1);
  while(!feof(fp1)) putc(getc(fp1),fp2);
  fclose(fp1);fclose(fp2);
 }
```

在第一次将文件的内容显示在屏幕以后，文件 file1.c 的位置指针已指到文件末尾，feof 的值为非零(真)。执行 rewind 函数，使文件的位置指针重新定位于文件开头，并使 feof 函数的值恢复为 0(假)。

13.5.2 fseek 函数和随机读写

对流式文件可以进行顺序读写，也可以进行随机读写，关键在于控制文件的位置指针。如果位置指针是按字节位置顺序移动的，就是顺序读写；如果能将位置指针按需要移动到任意位置，就可以实现随机读写。所谓随机读写，是指读写完上一个字符(字节)后，并不一定要读写其后续的字符(字节)，而可以读写文件中任意位置上所需要的字符(字节)。

用 fseek 函数可以实现改变文件的位置指针。

fseek 函数的调用形式为

fseek(文件类型指针,位移量,起始点)

"起始点"用 0、1 或 2 代替，0 代表"文件开始"，1 为"当前位置"，2 为"文件末尾"。ANSI C 标准指定的名字如表 13-2 所示。

表 13-2

起始点	名　字	用数字代表
文件开始	SEEK_SET	0
文件当前位置	SEEK_CUR	1
文件末尾	SEEK_END	2

"位移量"指以"起始点"为基点，向前移动的字节数。ANSI C 和大多数 C 版本要求位移量是 long 型数据。这样当文件的长度大于 64KB 时不致出问题。ANSI C 标准规定在数字的末尾加一个字母 L，就表示是 long 型。

fseek 函数一般用于二进制文件，因为文本文件要发生字符转换，计算位置时往往会发生混乱。

下面是 fseek 函数调用的几个例子：

```
fseek(fp,100L,0);  将位置指针移到离文件头 100 个字节处
```

```
fseek(fp,50L,1);    将位置指针移到离当前位置 50 个字节处
fseek(fp,-10L,2);  将位置指针从文件末尾处向后退 10 个字节
```

利用 fseek 函数就可以实现随机读写了。

例 13.5 在磁盘文件上存有 10 个学生的数据。要求将第 1、3、5、7、9 个学生数据输入计算机，并在屏幕上显示出来。

程序如下：

```
#include <stdlib.h>
#include<stdio.h>
struct student_type
{
 char name[10];
 int num;
 int age;
 char sex;
}stud[10];

void main()
 { int i;
   FILE *fp;
   if ((fp=fopen("stud_dat","rb"))==NULL)
     {printf("can not open file\n");
      exit(0);}
   for(i=0;i<10;i+=2)
     { fseek(fp,i*sizeof(struct student_type),0);
       fread(&stud[i], sizeof(struct student_type),1,fp);
       printf("%s %d %d %c\n",stud[i].name,stud[i].num,stud[i].age,stud[i].sex);
     }
   fclose(fp);
 }
```

13.5.3 ftell 函数

ftell 函数的作用是得到流式文件中的当前位置，用相对于文件开头的位移量来表示。由于文件中的位置指针经常移动，人们往往不容易知道其当前位置。用 ftell 函数可以得到当前位置。如果 ftell 函数返回值为－1L，表示出错。例如：

```
i=ftell(fp);
if(i==-1L)   printf("error\n");
```

变量 i 存放当前位置，如调用函数时出错(如不存在 fp 文件)，则输出“error”。

13.6 出错的检测

C 标准提供一些函数用来检查输入输出函数调用中的错误。

13.6.1 ferror 函数

在调用各种输入输出函数(如 putc、getc、fread、fwrite 等)时,如果出现错误,除了函数返回值有所反映外,还可以用 ferror 函数检查。它的一般调用形式为

ferror(fp);

如果 ferror 返回值为 0(假),表示未出错;如果返回一个非零值,表示出错。应该注意,对同一个文件每一次调用输入输出函数,均产生一个新的 ferror 函数值,因此,应当在调用一个输入输出函数后立即检查 ferror 函数的值,否则信息会丢失。

在执行 fopen 函数时,ferror 函数的初始值自动置为 0。

13.6.2 clearerr 函数

clearerr 的作用是使文件错误标志和文件结束标志置为 0。假设在调用一个输入输出函数时出现错误,ferror 函数值为一个非零值。在调用 clearerr(fp)后,ferror(fp)的值变成 0。

只要出现错误标志,就一直保留,直到对同一文件调用 clearerr 函数或 rewind 函数,或任何其他一个输入输出函数。

13.7 文件输入输出小结

在本节中将以上介绍过的输入输出函数作一概括性的小结,以一目了然,便于查阅。表 13-3 列出常用的缓冲文件系统函数。

表 13-3 常用的缓冲文件系统函数

分 类	函 数 名	功 能
打开文件	fopen( )	打开文件
关闭文件	fclose ( )	关闭文件
文件定位	fseek( )	改变文件位置指针的位置
	rewind ( )	使文件位置指针重新置于文件开头
	ftell( )	返回文件位置指针的当前值
文件读写	fgetc ( ),getc( )	从指定文件取得一个字符
	fputc ( ),putc ( )	把字符输出到指定文件
	fgets ( )	从指定文件读取字符串
	fputs ( )	把字符串输出到指定文件
	getw ( )	从指定文件读取一个字(int 型)
	putw ( )	把一个字(int 型)输出到指定文件
	fread ( )	从指定文件中读取数据项
	fwrite ( )	把数据项写到指定文件
	fscanf ( )	从指定文件按格式输入数据
	fprintf ( )	按指定格式将数据写到指定文件中
文件状态	feof ( )	若到文件末尾,函数值为“真”(非 0)
	ferror ( )	若对文件操作出错,函数值为“真”(非 0)
	clearerr ( )	使 ferror 和 feof 函数值置零

文件这一章的内容是很重要的，许多可供实际使用的C程序都包含文件处理。本章只介绍一些最基本的概念，由于篇幅所限，不可能举复杂的例子。希望读者在实践中掌握文件的使用。

习　题

13.1　对C文件操作有些什么特点？什么是缓冲文件系统？什么是非缓冲文件系统？这二者的缓冲区有什么区别？

13.2　什么是文件型指针？通过文件指针访问文件有什么好处？

13.3　对文件的打开与关闭的含义是什么？为什么要打开和关闭文件？

13.4　从键盘输入一个字符串，将其中的小写字母全部转换成大写字母，然后输出到一个磁盘文件“test”中保存。输入的字符串以“!”结束。

13.5　有两个磁盘文件“A”和“B”，各存放一行字母，今要求把这两个文件中的信息合并(按字母顺序排列)，输出到一个新文件“C”中去。

13.6　有5个学生，每个学生有3门课程的成绩，从键盘输入学生数据(包括学号，姓名，3门课程成绩)，计算出平均成绩，将原有数据和计算出的平均分数存放在磁盘文件“stud”中。

13.7　将题13.6“stud”文件中的学生数据，按平均分进行排序处理，将已排序的学生数据存入一个新文件“stu_sort”中。

13.8　将题13.7已排序的学生成绩文件进行插入处理。插入一个学生的3门课程成绩，程序先计算新插入学生的平均成绩，然后将它按成绩高低顺序插入，插入后建立一个新文件。

13.9　题13.8结果仍存入原有的“stu_sort”文件而不另建立新文件。

13.10　有一磁盘文件“employee”，内存放职工的数据。每个职工的数据包括职工姓名、职工号、性别、年龄、住址、工资、健康状况、文化程度。今要求将职工名、工资的信息单独抽出来另建一个简明的职工工资文件。

13.11　从题13.10的“职工工资文件”中删去一个职工的数据，再存回原文件。

13.12　从键盘输入若干行字符(每行长度不等)，输入后把它们存储到一磁盘文件中。再从该文件中读入这些数据，将其中小写字母转换成大写字母后在显示屏上输出。

第14章　常见错误和程序调试

C语言的功能强，使用方便灵活，所以得到广泛的使用，它使程序设计人员有发挥聪明才智、显示编程技巧的机会。一个有经验的C程序设计人员可以编写出能解决复杂问题的、运行效率高的、占内存少的高质量程序。C是函数式的语言，利用标准库函数和自己设计的函数可以完成许多功能。善于利用函数，可以实现程序的模块化，将许多函数组织成一个大的程序。正因为如此，C语言受到愈来愈广泛的重视，从初学者到高级软件人员，都在学习C，使用C。

但是要真正学好C，用好C，并不容易，“灵活”固然是好事，但也使人难以掌握，尤其是初学者往往出了错还不知怎么回事。C编译程序对语法的检查不如其他高级语言那样严格(这是为了给程序人员留下“灵活”的余地)。因此，往往要由程序设计者自己设法保证程序的正确性。调试一个C程序要比调试一个PASCAL或FORTRAN程序更困难一些，需要不断积累经验，提高程序设计和调试程序的水平。

C语言有些语法规定和其他高级语言不同，学习过其他高级语言的读者往往按照使用其他高级语言的习惯来写C程序，这也是出错的一个原因。

14.1　常见错误分析

下面将初学者在学习和使用C语言(不包括C++)时容易犯的错误列举出来，以起提醒的作用。这些内容在以前各章中大多已谈到，为便于查阅，在本章中集中列举，供初学者参考，以此为鉴。

(1) 忘记定义变量。例如：

```
void main( )
 {
  x=3;
  y=6;
  printf("%d\n",x+y);
 }
```

C要求对程序中用到的每一个变量都必须定义其类型，上面程序中没有对x、y进行定义。应在函数体的开头加

```
int x,y;
```

这是学过BASIC和FORTRAN语言的读者写C程序时常见的一个错误。在BASIC语言中，可以不必先定义变量类型就可直接使用。在FORTRAN中，未经定义类型的变量按隐含的I—N规则决定其类型，而C语言则要求对用到的每一个变量都要在本函数中

定义(除非已定义为外部变量)。

(2) 输入输出的数据的类型与所用格式说明符不一致。例如,若 a 已定义为整型,b 已定义为实型:

```
a=3;b=4.5;                    /* 对 a 和 b 赋值 */
printf("%f  %d\n",a,b);
```

编译时不给出出错信息,但运行结果将与原意不符,输出为

```
0.000000 16402
```

它们并不是按照赋值的规则进行转换(如把 4.5 转换成 4),而是将数据在存储单元中的形式按格式符的要求组织输出(如 b 占 4 个字节,只把最后 2 个字节中的数据按%d 作为整数输出)。

(3) 未注意 int 型数据的数值范围。Turbo C 等编译系统,对一个整型数据分配 2 个字节。因此一个整数的范围为-2^{15}～$2^{15}-1$,即−32768～32767。常见这样的程序段:

```
int num;
num=89101;
printf("%d",num);
```

得到的却是 23565,原因是 89101 已超过 32767。2 个字节容纳不下 89101,则将高位截去,见图 14-1,即将超过低 16 位的数截去,也即将 89101 减去 2^{16}(即 16 位二进制所形成的模):89101−65536=23565。

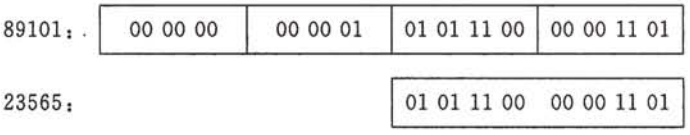


图 14-1

有时还会出现负数。例如:

```
num=196607;
```

输出得−1。因为 196607 的二进制形式为:

00 00 00 00 00 00 00 10	11 11 11 11 11 11 11 11

去掉高位 10,低 16 位的值是−1(−1 的补码是 1111111111111111)。

对于超过整数范围的数,要用 long 型,即改为:

```
long int num;
num=89101;
printf("%ld",num);
```

请注意,如果只定义 num 为 long 型,而在输出时仍用"%d"说明符,也会出现以上错误。

(4) 在输入语句 scanf 中忘记使用变量的地址符。例如:

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

这是许多初学者刚学习 C 语言时一个常见的疏忽，或者说是习惯性的错误，因为在其他语言中在输入时只需写出变量名即可，而 C 语言要求指明“向哪个地址标识的单元送值”。应写成

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

(5) 输入数据的形式与要求不符。用 scanf 函数输入数据，应注意如何组织输入数据。假如有以下 scanf 函数：

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

有人按下面的方法输入数据：

```
3,4↙
```

这是错的。数据间应该用空格(或 Tab 键，或回车符)来分隔。读者可以用

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

来验证一下。应该用以下方法输入：

```
3 4↙
```

如果 scanf 函数为

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

对 scanf 函数中格式字符串中除了格式说明符外，对其他字符必须按原样输入。因此，应按以下方法输入：

```
3,4↙
```

此时如果用“3 4”反而错了。还应注意，不能企图用

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

想在屏幕上显示一行信息：

```
input a & b:
```

然后在其后输入 a 和 b 的值，这是不行的。这是由于有的读者以为 scanf 具有 BASIC 语言中的 INPUT 语句的功能(先输出一个字符串，再输入变量的值)。如果想在屏幕上得到所需的提示信息，可以另加一个 printf 函数语句：

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

(6) 误把“=”作为“等于”运算符。在许多高级语言中，用“=”符号作为关系运算符“等于”。例如，在 BASIC 或 PASCAL 程序中都可以写

```
if(a=b) then…
```

但在 C 语言中，“=”是赋值运算符，“==”才是关系运算符“等于”。如果写成

```
if(a=b) printf("a equal to b");
```

C 编译系统将(a=b)作为赋值表达式处理，将 b 的值赋给 a，然后判断 a 的值是否为零，若为非零，则作为“真”；若为零作为“假”。如果 a 的值为 3，b 的值为 4，a≠b，按原意不应输出“a equal to b”。而现在先将 b 的值赋给 a，a 也为 4，赋值表达式的值为 4。if 语句中的表达式值为真(非零)，因此输出“a equal to b”。

这种错误在编译时是检查不出来的，但运行结果往往是错的。而且由于习惯的影响，程序设计者自己往往也不易发觉。

(7) 语句后面漏分号。C 语言规定语句末尾必须有分号。分号是 C 语句不可缺少的一部分。这也是和其他语言不同的。有的初学者往往忘记写这一分号。例如：

```
a=3
b=4
```

编译时，编译程序在“a=3”后面未发现分号，就把下一行“b=4”也作为上一行的语句的一部分，这就出现语法错误。有时编译时指出某行有错，但在该行上并未发现错误，应该检查上一行是否漏了分号。

如果用复合语句，有的学过 PASCAL 语言的读者往往漏写最后一个语句的分号，例如：

```
{t=a;
 a=b;
 b=t
}
```

在 PASCAL 中分号是两个语句间的分隔符而不是语句的一部分，而在 C 语言中，没有分号的就不是语句。

(8) 在不该加分号的地方加了分号。例如：

```
if (a>b);
  printf("a is larger than b\n");
```

本意为当 a>b 时输出“a is larger than b”的信息。但由于在 if (a>b)后加了分号，因此 if 语句到此结束。即当 a>b 为真时，执行一个空语句。本来想 a≤b 时不输出上述信息，但现在 printf 函数语句并不从属于 if 语句，而是与 if 语句平行的语句。不论 a>b 还是 a≤b，都输出“a is larger than b”。

又如：

```
for(i=0;i<10;i++);
  {scanf("%d",&x);
   printf("%d\n",x*x);
  }
```

本意为先后输入 10 个数，每输入一个数后输出它的平方值。由于在 for( )后加了一个分号，使循环体变成了空语句。只能输入一个整数并输出它的平方值。

总之，在 if、for、while 语句中，不要画蛇添足，多加分号。

(9) 对应该有花括号的复合语句,忘记加花括号。例如:

```
sum=0;
i=1;
while(i<=100)
   sum=sum+i;
   i++;
```

本意是实现1+2+…+100,即 $\sum_{i=0}^{100} i$,但上面的语句只是重复了sum+i的操作,而且循环永不终止,因为i的值始终没有改变。错误在于没有写成复合语句形式。因此,while语句的范围到其后第一个分号为止。语句"i++;"不属于循环体范围之内。应改为

```
while(i<=100)
   {sum=sum+i;
    i++;
   }
```

(10) 括号不配对。当一个语句中使用多层括号时常出现这类错误,纯属粗心所致。例如:

```
while((c=getchar( )!='#')
  putchar(c);
```

少了一个右括号。

(11) 在用标识符时,忘记了大写字母和小写字母的区别。例如:

```
void main( )
 {int a,b,c;
  a=2;b=3;
  C=A+B;
  printf("%d+%d=%d",A,B,C);
 }
```

编译时出错。编译程序把a和A认作是两个不同的变量名处理,同样b和B、c和C都分别代表两个不同的变量。

(12) 引用数组元素时误用了圆括号。例如:

```
void main( )
 {int i,a[10];
  for(i=0;i<10;i++)
    scanf("%d",&a(i));
 }
```

C语言中对数组的定义或引用数组元素时必须用方括号。

(13) 在定义数组时,将定义的"元素个数"误认为是"可使用的最大下标值"。例如:

```
void main( )
 { int a[10]={1,2,3,4,5,6,7,8,9,10};
```

```
    int i;
    for(i=1;i<=10;i++)
      printf("%d",a[i]);
}
```

想输出 a[1]到 a[10]。这是一些初学者常犯的错误。C 语言规定定义时用 a[10],表示 a 数组有 10 个元素,而不是可以用的最大下标值为 10。数组只包括 a[0]到 a[9]这 10 个元素,因此用 a[10]就超出 a 数组的范围了。

(14) 对二维或多维数组的定义和引用的方法不对。例如:

```
void main( )
  {int a[5,4];
     ⋮
   printf("%d",a[1+2,2+2]);
     ⋮
  }
```

对二维数组和多维数组在定义和引用时必须将每一维的数据分别用方括号括起来。上面 a[5,4]应改为 a[5][4],a[1+2,2+2]应改为 a[1+2][2+2]。根据 C 的语法规则,在一个方括号中的是一个维的下标表达式,a[1+2,2+2]方括号中的"1+2,2+2"是一个逗号表达式,它的值是第二个数值表达式的值,即 2+2 的值为 4。所以 a[1+2,2+2]相当于 a[4],而 a[4]是 a 数组的第 4 行的首地址。因此执行 printf 函数输出的结果并不是 a[3][4]的值,而是 a 数组第 4 行的首地址。

(15) 误以为数组名代表数组中全部元素。例如:

```
void main( )
 {int a[4]={1,3,5,7};
  printf("%d%d%d%d\n",a);
 }
```

企图用数组名代表全部元素。在 C 语言中,数组名代表数组首地址,不能通过数组名输出 4 个整数。

(16) 混淆字符数组与字符指针的区别。例如:

```
void main( )
 {char str[4];
  str="Computer and c";
  printf("%s\n",str);
 }
```

编译出错。str 是数组名,代表数组首地址。在编译时对 str 数组分配了一段内存单元,因此在程序运行期间 str 是一个常量,不能再被赋值。所以,str="Computer and c"是错误的。如果把"char str[4];"改成"char * str;",则程序正确。此时 str 是指向字符数据的指针变量,str="Computer and c"是合法的,它将字符串的首地址赋给指针变量 str,然后在 printf 函数语句中输出字符串"Computer and c"。

因此应当弄清楚字符数组与字符指针变量用法的区别。

(17) 在引用指针变量之前没有对它赋予确定的值。例如：

```
void main( )
 {char * p;
  scanf("%s",p);
     ⋮
 }
```

没有给指针变量 p 赋值就引用它，编译时给出警告信息。应当改为

```
char * p,c[20];
p=c;
scanf("%s",p);
```

即先根据需要定义一个大小合适的字符数组 c，然后将 c 数组的首地址赋给指针变量 p，此时 p 有确定的值，指向数组 c。再执行 scanf 函数就没有问题了，把从键盘输入的字符串存放到字符数组 c 中。

(18) switch 语句的各分支中漏写 break 语句。例如：

```
switch(score)
{case 5:printf("Very good!");
 case 4:printf("Good!");
 case 3:printf("Pass!");
 case 2:printf("Fail!");
 default:printf("data error!");
}
```

上述 switch 语句的作用是希望根据 score(成绩)输出评语。但当 score 的值为 5 时，输出为

```
Very Good! Good! Pass! Fail! data error!
```

原因是漏写了 break 语句。case 只起标号的作用，而不起判断作用，因此在执行完第一个 printf 函数语句后接着执行第 2、3、4、5 个 printf 函数语句。应改为

```
switch(score)
{case 5:printf("Very good!");  break;
 case 4:printf("Good!");       break;
 case 3:printf("Pass!");       break;
 case 2:printf("Fail!");       break;
 default:printf("data error!");
}
```

(19) 混淆字符和字符串的表示形式。例如：

```
char sex;
sex="M";
   ⋮
```

sex 是字符变量，只能存放一个字符。而字符常量的形式是用单撇号括起来的，应改为

```
sex='M';
```

"M"是用双撇号括起来的字符串，它包括两个字符：'M'和'\0'，无法存放到字符变量sex中。

(20) 使用自加(++)和自减(--)运算符时出的错误。例如：

```
void main( )
 {int * p,a[5]={1,3,5,7,9};
  p=a;
  printf("%d", * p++);
 }
```

不少人认为"* p++"的作用是先使p加1，即指向第1个元素a[1]处，然后输出第一个元素a[1]的值3。其实应该是先执行p++，而p++的作用是先用p的原值，用完后使p加1。p的原值指向数组a的第0个元素a[0]，因此* p就是第0个元素a[0]的值1。结论是先输出a[0]的值，然后再使p加1。如果是*(++p)，则先使p指向a[1]，然后输出a[1]的值。

(21) 所调用的函数在调用语句之后才定义，而又在调用前未声明。例如：

```
void main( )
 {float x,y,z;
  x=3.5;y=-7.6;
  z=max(x,y);
  printf("%f\n",z);
 }

float max(float x,float y)
{return(z=x>y? x:y);
}
```

这个程序乍看起来没有什么问题，但在编译时有出错信息。原因是max函数是在main函数之后才定义，也就是max函数的定义位置在main函数调用max函数之后。改错的方法可以用以下二者之一：

① 在main函数中增加一个对max函数的声明，即函数的原型：

```
void main( )
 {float max(float,float);/ * 声明将要用到的max函数为实型 * /
  float x,y,z;
  x=3.5;y=-7.6;
  z=max(x,y);
  printf("%f\n",z);
 }
```

② 将max函数的定义位置调到main函数之前。即：

```
float max(float x,float y)
{return(z=x>y? x:y);}

void main()
```

```
{float x,y,z;
 x=3.5;y=-7.6;
 z=max(x,y);
 printf("%f\n",z);
}
```

这样,编译时不会出错,程序运行结果是正确的。

(22) 对函数声明与函数定义不匹配。如已定义一个 fun 函数,其首行为:

```
int fun(int x,float y,long z)
```

在主调函数中作下面的声明将出错。

```
fun(int x,float y,long z);                    /*漏写函数类型*/
float fun(int x,float y,long z);              /*函数类型不匹配*/
int fun(int x,int y,int z);                   /*参数类型不匹配*/
int fun (int x,float y);                      /*参数数目不匹配*/
int fun(int x,long z,float y);                /*参数顺序不匹配*/
```

下面的声明是正确的:

```
int fun(int x,float y,long z);
int fun(int,float,long);                      /*可以不写形参名*/
int fun(int a,float b,long c);                /*编译时不检查函数原型中的形参名*/
```

(23) 在需要加头文件时没有用#include 命令去包含头文件。例如:

程序中用到 fabs 函数,没有用#include<math.h>

程序中用到输入输出函数,没有用#include<stdio.h>

这是不少初学者常犯的错误。至于哪个函数应该用哪个头文件,可参阅本书附录 E。

(24) 误认为形参值的改变会影响实参的值。例如:

```
void main( )
 {int a,b;
  a=3;b=4;
  swap(a,b);
  printf("%d,%d\n",a,b);
 }
void swap(int x,int y)
 {int t;
  t=x;x=y;y=t;
 }
```

原意是通过调用 swap 函数使 a 和 b 的值对换,然后在 main 函数中输出已对换了值的 a 和 b。但是这样的程序是达不到目的的,因为 x 和 y 的值的变化是不传送回实参 a 和 b 的,main 函数中的 a 和 b 的值并未改变。

如果想从函数得到一个以上的变化了的值,应该用指针变量。用指针变量作函数参数,使指针变量所指向的变量的值发生变化。此时变量的值改变了,主调函数中可以利用这些已改变的值。例如:

```
void main( )
  {int a,b, * p1, * p2;
   a=3;b=4;
   p1=&a;p2=&b;
   swap(p1,p2);
   printf("%d,%d\n",a,b);          /* a 和 b 的值已对换 */
  }
void swap(int * pt1, int * pt2)
  {int t;
   t= * pt1; * pt1= * pt2; * pt2=t;
  }
```

(25) 函数的实参和形参类型不一致。例如：

```
void main( )
 {int a=3,b=4,c;
  c=fun(a,b);
    ⋮
 }
int fun(float x,float y)
{
   ⋮
}
```

实参 a、b 为整型，形参 x、y 为实型。a 和 b 的值传递给 x 和 y 时，x 和 y 得到的值并非 3 和 4，得不到正确的运行结果。C 要求实参与形参的类型一致。如果在 main 函数中对 fun 作原型声明：

```
int fun (float, float);
```

程序可以正常运行，此时，按不同类型间的赋值的规则处理，在虚实结合后 x=3.0，y=4.0。也可以将 fun 函数的位置调到 main 函数之前，也可获得正确结果。

(26) 不同类型的指针混用。例如：

```
void main( )
 {int i=3, * p1;
  float a=1.5, * p2;
  p1=&i; p2=&a;
  p2=p1;
  printf("%d,%d\n", * p1, * p2);
 }
```

企图使 p2 也指向 i，但 p2 是指向实型变量的指针，不能指向整型变量。指向不同类型的指针间的赋值必须进行强制类型转换。例如：

```
p2=(float * )p1;
```

作用是先将 p1 的值转换成指向实型的指针，然后再赋给 p2。

这种情况在 C 程序中是常见的。例如，用 malloc 函数开辟内存单元，函数返回的是

指向被分配内存空间的 void * 类型的指针。而人们希望开辟的是存放一个结构体变量值的存储单元，要求得到指向该结构体变量的指针，可以进行如下的类型转换：

```
struct student
{int num;
 char name[20];
 float score;
};
struct student student1, * p;
     ⋮
p=(struct student * )malloc(LEN);
```

p 是指向 struct student 结构体类型数据的指针，将 malloc 函数返回的 void * 类型指针转换成指向 struct student 类型变量的指针。

(27) 没有注意函数参数的求值顺序。例如，有以下语句：

```
i=3;
printf("%d,%d,%d\n",i,++i,++i);
```

许多人认为输出必然是

```
3,4,5
```

实际不尽然。在 Turbo C 和其他一些 C 系统中输出是

```
5,5,4
```

因为这些系统是采取自右至左的顺序求函数参数的值。先求出最右面一个参数(++i)的值为 4，再求出第 2 个参数(++i)的值为 5，最后求出最左面的参数(i)的值 5。

C 标准没有具体规定函数参数求值的顺序是自左至右，还是自右至左。但每个 C 编译程序都有自己的顺序，在有些情况下，从左到右求解和从右到左求解的结果是相同的。例如：

```
fun1(a+b,b+c,c+a);
```

fun1 是一个函数名，3 个实参表达式：a+b、b+c、c+a。在一般情况下，自左至右地求这 3 个表达式的值和自右至左地求它们的值是一样的，但在前面举的例子是不相同的。因此，建议最好不使用会引起二义性的用法。如果在上例中，希望输出“3,4,5”时，可以改用

```
i=3; j=i+1; k=j+1;
printf("%d,%d,%d\n",i,j,k);
```

(28) 混淆数组名与指针变量的区别。例如：

```
void main( )
 {int i,a[5];
  for(i=0;i<5;i++)
    scanf("%d",a++);
      ⋮
 }
```

企图通过 a 的改变使指针下移，每次指向欲输入数据的数组元素。它的错误在于不了解

数组名代表数组首地址，它的值是不能改变的，用 a＋＋是错误的，应当用指针变量来指向各数组元素。即：

```
int i,a[5], * p;
p=a;
for(i=0;i<5;i++)
  scanf("%d",p++);
```

或

```
int a[5], * p;
for(p=a;p<a+5;p++)
  scanf("%d",p);
```

(29) 混淆结构体类型与结构体变量的区别，对一个结构体类型赋值。例如：

```
struct worker
 {long int num;
  char name[20];
  char sex;
  int age;
 };
worker. num=187045;
strcpy(worker. name,"ZhangFun");
worker. sex='M';
worker. age=18;
```

这是错误的，只能对变量赋值而不能对类型赋值。上面只定义了 struct worker 类型而未定义变量。应改为：

```
struct worker
 {long int num;
  char name[20];
  char sex;
  int age;
 };
struct worker worker_1;
worker_1. num=187045;
strcpy(worker_1. name,"Zhang Fun");
worker_1. sex='M';
worker_1. age=18;
```

今定义了结构体变量 worker_1，并对其中的各成员赋值。

(30) 使用文件时忘记打开，或打开方式与使用情况不匹配。例如，对文件的读写，用只读方式打开，却企图向该文件输出数据，例如：

```
if ((fp=fopen("test","r"))==NULL)
  {printf("cannot open this file\n");
   exit(0);
  }
ch=fgetc(fp);
```

```
while(ch!='#')
  {ch=ch+4;
   fputc(ch,fp);
   ch=fget(fp);
  }
```

对以“r”方式(只读方式)打开的文件,进行既读又写的操作,显然是不行的。

此外,有的程序常忘记关闭文件,虽然系统会自动关闭所用文件,但可能会丢失数据。因此必须在用完文件后关闭它。

以上只是列举了一些初学者常出现的错误,这些错误大多是对于C语法不熟悉之故。对C语言使用多了,比较熟练了,犯这些错误自然就会减少了。在深入使用C语言后,还会出现其他一些更深入、更隐蔽的错误。

程序出错有3种情况:

① 语法错误。指违背了C语法的规定,对这类错误,编译程序一般能给出“出错信息”,并且告诉在哪一行出错。只要细心,是可以很快发现并排除的。

② 逻辑错误。程序并无违背语法规则,但程序执行结果与原意不符。这是由于程序设计人员设计的算法有错或编写程序有错,通知给系统的指令与解题的原意不相同,即出现了逻辑上的混乱。例如,前面第9条错误:

```
sum=0;i=1;
while(i<=100)
  sum=sum+i;
  i++;
```

语法并无错误。但while语句通知给系统的信息是当i≤100时,执行“sum=sum+i;”。C系统无法辨别程序中这个语句是否符合作者的原意,而只能忠实地执行这一指令。这种错误比语法错误更难检查。要求程序员有较丰富的经验。

③ 运行错误。程序既无语法错误,也无逻辑错误,但在运行时出现错误甚至停止运行。例如:

```
int a ,b ,c;
scanf("%d %d",&a,&b);
c=b/a;
printf("c=%d\n",c);
```

输入a和b的值,输出b/a的值,程序没有错。但是如果输入a的值为0,就会出现错误。因此程序应能适应不同的数据,或者说能经受各种数据的“考验”,具有“健壮性”。

写完一个程序只能说完成任务的一半(甚至不到一半)。调试程序往往比写程序更难,更需要精力、时间和经验。常常有这样的情况:程序花一天就写完了,而调试程序二三天也未能完。有时一个小小的程序会出错五六处,而发现和排除一个错误,有时竟需要半天,甚至更多。希望读者通过实践掌握调试程序的方法和技术。

14.2 程序调试

所谓程序调试是指对程序的查错和排错。

调试程序一般应经过以下几个步骤:

(1) 先进行人工检查,即静态检查。在写好一个程序以后,不要匆匆忙忙上机,而应对纸面上的程序进行人工检查。这一步是十分重要的,它能发现程序设计人员由于疏忽而造成的多数错误。而这一步骤往往容易被人忽视。有人总希望把一切推给计算机系统去做,但这样就会多占用机器时间。而且,作为一个程序人员应当养成严谨的科学作风,每一步都要严格把关,不把问题留给后面的工序。

为了更有效地进行人工检查,所编的程序应注意力求做到以下几点:①应当采用结构化程序方法编程,以增加可读性;②尽可能多加注释,以帮助理解每段程序的作用;③在编写复杂的程序时,不要将全部语句都写在 main 函数中,而要多利用函数,用一个函数来实现一个单独的功能。这样既易于阅读也便于调试,各函数之间除用参数传递数据这一渠道以外,数据间尽量少出现耦合关系,便于分别检查和处理。

(2) 在人工(静态)检查无误后,才可以上机调试。通过上机发现错误称动态检查。在编译时给出语法错误的信息(包括哪一行有错以及错误类型),可以根据提示的信息具体找出程序中出错之处并改正之。应当注意的是:有时提示的出错行并不是真正出错的行,如果在提示出错的行上找不到错误的话应当到上一行再找。另外,有时提示出错的类型并非绝对准确,由于出错的情况繁多而且各种错误互有关联,因此要善于分析,找出真正的错误,而不要只从字面意义上死抠出错信息,钻牛角尖。

如果系统提示的出错信息多,应当从上到下逐一改正。有时显示出一大片出错信息往往使人感到问题严重,无从下手。其实可能只有一二个错误。例如,对所用的变量未定义,编译时就会对所有含该变量的语句发出出错信息,只要加上一个变量定义,所有错误就都消除了。

(3) 在改正语法错误(包括"错误"(error)和"警告"(warning))后,程序经过连接(link)就得到可执行的目标程序。运行程序,输入程序所需数据,就可得到运行结果。应当对运行结果作分析,看它是否符合要求。有的初学者看到输出运行结果就认为没问题了,不作认真分析,这是危险的。

有时,数据比较复杂,难以立即判断结果是否正确。可以事先考虑好一批"试验数据",输入这些数据可以得出容易判断正确与否的结果。例如,解方程 $ax^2+bx+c=0$,输入 a、b、c 的值分别为 1、−2、1 时,根 x 的值是 1。这是容易判断的,若根不等于 1,程序显然有错。

但是,用"试验数据"时,程序运行结果正确,还不能保证程序完全正确。因为有可能输入另一组数据时运行结果不对。例如,用 $x=\dfrac{-b\pm\sqrt{b^2-4ac}}{2a}$ 公式求根 x 的值,当 $a\neq 0$ 和 $b^2-4ac>0$ 时,能得出正确结果,当 $a=0$ 或 $b^2-4ac<0$ 时,就得不到正确结果(假设程序中未对 $a=0$ 作防御处理以及未作复数处理)。因此应当把程序可能遇到的多种方案都一一试到。例如,if 语句有两个分支,有可能在流程经过其中一个分支时结果正确,而经过另一个分支时结果不对,必须考虑周全。

事实上,当程序复杂时很难把所有的可能方案全部都试到,选择典型的情况作试验即可。

(4) 运行结果不对,大多属于逻辑错误。对这类错误往往需要仔细检查和分析才能

发现。可以采用以下办法：

① 将程序与流程图(或伪代码)仔细对照，如果流程图是正确的话，程序写错了，是很容易发现的。例如，复合语句忘记写花括号，只要一对照流程图就能很快发现。

② 如果实在找不到错误，可以采取“分段检查”的方法。在程序不同位置设几个printf 函数语句，输出有关变量的值，逐段往下检查，直到找到在某一段中数据不对为止。这时就已经把错误局限在这一段中了。不断缩小“查错区”，就可能发现错误所在。

③ 也可以用第 9 章介绍过的“条件编译”命令进行程序调试(在程序调试阶段，若干printf 函数语句要进行编译并执行。当调试完毕，这些语句不要再编译了，也不再被执行了)。这种方法可以不必一一删去 printf 函数语句，以提高效率。

④ 如果在程序中没有发现问题，就要检查流程图有无错误，即算法有无问题，如有则改正之，接着修改程序。

⑤ 有的系统还提供 debug(调试)工具，跟踪流程并给出相应信息，使用更为方便，请查阅有关手册。

总之，程序调试是一项细致深入的工作，需要下功夫，动脑子，善于累积经验。在程序调试过程中往往反映出一个人的水平、经验和科学态度。希望读者能给予足够的重视。上机调试程序的目的决不是为了“验证程序的正确性”，而是“掌握调试的方法和技术”。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

附录 A　常用字符与 ASCII 代码对照表

ASCII 值	字符	控制字符	ASCII 值	字符	ASCII 值	字符	ASCII 值	字符	ASCII 值	字符	ASCII 值	字符	ASCII 值	字符	ASCII 值	字符
000	(null)	NUL	032	(space)	064	@	096	`	128	Ç	160	á	192	└	224	α
001	☺	SOH	033	!	065	A	097	a	129	ü	161	í	193	┴	225	β
002	●	STX	034	"	066	B	098	b	130	é	162	ó	194	┬	226	Γ
003	♥	ETX	035	#	067	C	099	c	131	â	163	ú	195	├	227	π
004	♦	EOT	036	$	068	D	100	d	132	ä	164	ñ	196	─	228	Σ
005	♣	END	037	%	069	E	101	e	133	à	165	Ñ	197	┼	229	σ
006	♠	ACK	038	&	070	F	102	f	134	å	166	ª	198	╞	230	μ
007	(beep)	BEL	039	'	071	G	103	g	135	ç	167	º	199	╟	231	τ
008	◘	BS	040	(	072	H	104	h	136	ê	168	¿	200	╚	232	Φ
009	(tab)	HT	041	)	073	I	105	i	137	ë	169	⌐	201	╔	233	θ
010	(line feed)	LF	042	*	074	J	106	j	138	è	170	¬	202	╩	234	Ω
011	(home)	VT	043	+	075	K	107	k	139	ï	171	1/2	203	╦	235	δ
012	(form feed)	FF	044	,	076	L	108	l	140	î	172	1/4	204	╠	236	∞
013	(carriage return)	CR	045	-	077	M	109	m	141	ì	173	¡	205	═	237	φ
014	♫	SO	046	.	078	N	110	n	142	Ä	174	«	206	╬	238	∈
015	☼	SI	047	/	079	O	111	o	143	Å	175	»	207	╧	239	∩
016	►	DLE	048	0	080	P	112	p	144	É	176	▒	208	╨	240	≡
017	◄	DC1	049	1	081	Q	113	q	145	æ	177	▓	209	╤	241	±
018	↕	DC2	050	2	082	R	114	r	146	Æ	178	▓	210	╥	242	≥
019	‼	DC3	051	3	083	S	115	s	147	ô	179	│	211	╙	243	≤
020	¶	DC4	052	4	084	T	116	t	148	ö	180	┤	212	╘	244	⌠
021	§	NAK	053	5	085	U	117	u	149	ò	181	╡	213	╒	245	⌡
022	▬	SYN	054	6	086	V	118	v	150	û	182	╢	214	╓	246	÷
023	↨	ETB	055	7	087	W	119	w	151	ù	183	╖	215	╫	247	≈
024	↑	CAN	056	8	088	X	120	x	152	ÿ	184	╕	216	╪	248	°
025	↓	EM	057	9	089	Y	121	y	153	Ö	185	╣	217	┘	249	∙
026	→	SUB	058	:	090	Z	122	z	154	Ü	186	║	218	┌	250	·
027	←	ESC	059	;	091	[	123	{	155	¢	187	╗	219	█	251	√
028	∟	FS	060	<	092	\	124	¦	156	£	188	╝	220	▄	252	n
029	◆	GS	061	=	093	]	125	}	157	¥	189	╜	221	▌	253	2
030	▲	RS	062	>	094	^	126	~	158	Pt	190	┘	222	▐	254	■
031	▼	US	063	?	095	_	127	⌂	159	ƒ	191	┐	223	▀	255(blank 'FF')	

注：128～255 是 IBM-PC 上专用的。表中 000～127 是标准的。

附录 B　C 语言中的关键字

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			

附录 C　运算符和结合性

优先级	运算符	含　义	要求运算对象的个数	结合方向
1	( )	圆括号		自左至右
	[ ]	下标运算符		
	−>	指向结构体成员运算符		
	.	结构体成员运算符		
2	!	逻辑非运算符	1（单目运算符）	自右至左
	~	按位取反运算符		
	++	自增运算符		
	−−	自减运算符		
	−	负号运算符		
	(类型)	类型转换运算符		
	*	指针运算符		
	&	取地址运算符		
	sizeof	长度运算符		
3	*	乘法运算符	2（双目运算符）	自左至右
	/	除法运算符		
	%	求余运算符		
4	+	加法运算符	2（双目运算符）	自左至右
	−	减法运算符		
5	<<	左移运算符	2（双目运算符）	自左至右
	>>	右移运算符		
6	< <= > >=	关系运算符	2（双目运算符）	自左至右

续表

优先级	运算符	含义	要求运算对象的个数	结合方向
7	==	等于运算符	2（双目运算符）	自左至右
	!=	不等于运算符		
8	&	按位与运算符	2（双目运算符）	自左至右
9	^	按位异或运算符	2（双目运算符）	自左至右
10	|	按位或运算符	2（双目运算符）	自左至右
11	&&	逻辑与运算符	2（双目运算符）	自左至右
12	||	逻辑或运算符	2（双目运算符）	自左至右
13	? :	条件运算符	3（三目运算符）	自右至左
14	= += -= *= /= %= >>= <<= &= ^= |=	赋值运算符	2（双目运算符）	自右至左
15	,	逗号运算符（顺序求值运算符）		自左至右

说明：

(1) 同一优先级的运算符，运算次序由结合方向决定。例如 * 与 / 具有相同的优先级别，其结合方向为自左至右，因此 3 * 5 / 4 的运算次序是先乘后除。－和＋＋为同一优先级，结合方向为自右至左，因此－ i ＋＋相当于－(i ＋＋)。

(2) 不同的运算符要求有不同的运算对象个数，如＋(加)和－(减)为双目运算符，要求在运算符两侧各有一个运算对象(如 3＋5、8－3 等)。而＋＋和－(负号)运算符是单目运算符，只能在运算符的一侧出现一个运算对象(如－a、i ＋＋、－－ i、(float) i、sizeof(int)、* p 等)。条件运算符是 C 语言中惟一的一个三目运算符，如 x ? a : b。

(3) 从上表中可以大致归纳出各类运算符的优先级：

初等运算符 ( ) [ ] -> .

↓

单目运算符

↓

算术运算符 (先乘除，后加减)

↓

关系运算符

↓

逻辑运算符 （不包括!）

↓

条件运算符

↓

赋值运算符

↓

逗号运算符

以上的优先级别由上到下递减。初等运算符优先级最高，逗号运算符优先级最低。位运算符的优先级比较分散（有的在算术运算符之前（如～），有的在关系运算符之前（如<<和>>），有的在关系运算符之后（如 &、^、|））。为了容易记忆，使用位运算符时可加圆括号。

附录 D　C 语言常用语法提要

为读者查阅方便，下面列出 C 语言语法中常用的一些部分的提要。为便于理解，没有采用严格的语法定义形式，只是备忘性质，供参考。

1. 标识符

标识符可由字母、数字和下划线组成。标识符必须以字母或下划线开头，大、小写的字母分别认为是两个不同的字符。不同的系统对标识符的字符数有不同的规定，一般允许 7 个字符。

2. 常量

可以使用：

（1）整型常量

- 十进制常数。
- 八进制常数（以 0 开头的数字序列）。
- 十六进制常数（以 0x 开头的数字序列）。
- 长整型常数（在数字后加字符 L 或 l）。

（2）字符常量

用单撇号括起来的一个字符，可以使用转义字符。

（3）实型常量（浮点型常量）

- 小数形式。
- 指数形式。

（4）字符串常量

用双撇号括起来的字符序列。

3. 表达式

(1) 算术表达式

- 整型表达式：参加运算的运算量是整型量，结果也是整型数。
- 实型表达式：参加运算的运算量是实型量，运算过程中先转换成 double 型，结果为 double 型。

(2) 逻辑表达式

用逻辑运算符连接的整型量，结果为一个整数(0 或 1)。逻辑表达式可以认为是整型表达式的一种特殊形式。

(3) 字位表达式

用位运算符连接的整型量，结果为整数。字位表达式也可以认为是整型表达式的一种特殊形式。

(4) 强制类型转换表达式

用“(类型)”运算符使表达式的类型进行强制转换，如(float)a。

(5) 逗号表达式(顺序表达式)

其形式为

表达式 1，表达式 2，…，表达式 n

顺序求出表达式 1，表达式 2，…，表达式 n 的值，结果为表达式 n 的值。

(6) 赋值表达式

将赋值号“=”右侧表达式的值赋给赋值号左边的变量。赋值表达式的值为执行赋值后被赋值的变量的值。

(7) 条件表达式

其形式为

逻辑表达式？表达式 1：表达式 2

逻辑表达式的值若为非零，则条件表达式的值等于表达式 1 的值；若逻辑表达式的值为零，则条件表达式的值等于表达式 2 的值。

(8) 指针表达式

对指针类型的数据进行运算，例如，p－2、p1－p2 等(其中 p、p1、p2 均已定义为指向数组的指针变量，p1 与 p2 指向同一数组中的元素)，结果为指针类型。

以上各种表达式可以包含有关的运算符，也可以是不包含任何运算符的初等量(例如，常数是算术表达式的最简单的形式)。

4. 数据定义

对程序中用到的所有变量都需要进行定义。对数据要定义其数据类型，需要时要指定其存储类别。

(1) 类型标识符可用

int
short

long
unsigned
char
float
double
struct　　结构体名
union　　共用体名
enum　　枚举类型名
用 typedef 定义的类型名

结构体与共用体的定义形式为：

struct　　结构体名
　{ **成员表列** }；

union　　共用体名
　{ **成员表列** }；

用 typedef 定义新类型名的形式为：

typedef　　已有类型　新定义类型；

例如：

```
typedef int COUNT;
```

(2) 存储类别可用

```
auto
static
register
extern
```

(如不指定存储类别，作 auto 处理)
变量的定义形式为：
存储类别　数据类型　变量表列；
例如：

```
static  float  a,b,c;
```

注意外部数据定义只能用 extern 或 static，而不能用 auto 或 register。

5. 函数定义

其形式为：
存储类别　数据类型　函数名(形参表列)
函数体
函数的存储类别只能用 extern 或 static。函数体是用花括号括起来的，可包括数据定义和语句。函数的定义举例如下：

```
static int max (int x,int y)
{ int z;
  z=x > y? x:y;
  return (z);
}
```

6. 变量的初始化

可以在定义时对变量或数组指定初始值。

静态变量或外部变量如未初始化，系统自动使其初值为零(对数值型变量)或空(对字符型数据)。对自动变量或寄存器变量，若未初始化，则其初值为一不可预测的数据。

7. 语句

(1) 表达式语句；
(2) 函数调用语句；
(3) 控制语句；
(4) 复合语句；
(5) 空语句。

其中控制语句包括：

(1) if(表达式)语句
或
if (表达式) 语句 1
else 语句 2
(2) while (表达式) 语句
(3) do 语句
while (表达式)；
(4) for (表达式 1;表达式 2;表达式 3)
语句
(5) 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;
}

前缀 case 和 default 本身并不改变控制流程，它们只起标号作用，在执行上一个 case 所标志的语句后，继续顺序执行下一个 case 前缀所标志的语句，除非上一个语句中最后用 *break* 语句使控制转出 switch 结构。

(6) break 语句
(7) continue 语句

(8) return　语句
(9) goto　语句

8. 预处理命令

define　宏名　字符串
define　宏名(参数 1,参数 2,…,参数 n) 字符串
undef　宏名
include　“文件名”　(或<文件名>)
if　常量表达式
ifdef　宏名
ifndef　宏名
else
endif

附录E　C库函数

库函数并不是C语言的一部分,它是由人们根据需要编制并提供用户使用的。每一种C编译系统都提供了一批库函数,不同的编译系统所提供的库函数的数目和函数名以及函数功能是不完全相同的。ANSI C标准提出了一批建议提供的标准库函数,它包括了目前多数C编译系统所提供的库函数,但也有一些是某些C编译系统未曾实现的。考虑到通用性,本书列出ANSI C标准建议提供的、常用的部分库函数。对多数C编译系统,可以使用这些函数的绝大部分。由于C库函数的种类和数目很多(例如,还有屏幕和图形函数、时间日期函数、与系统有关的函数等,每一类函数又包括各种功能的函数),限于篇幅,本附录不能全部介绍,只从教学需要的角度列出最基本的。读者在编制C程序时可能要用到更多的函数,请查阅所用系统的手册。

1. 数学函数

使用数学函数时,应该在该源文件中使用以下命令行:

include　<math.h>或#include　"math.h"

函数名	函数原型	功　能	返回值	说　明
abs	int abs (int x);	求整数 x 的绝对值	计算结果	
acos	double acos (double x);	计算 $\cos^{-1}(x)$的值	计算结果	x 应在−1 到 1 范围内
asin	double asin (double x);	计算 $\sin^{-1}(x)$的值	计算结果	x 应在−1 到 1 范围内
atan	double atan (double x);	计算 $\tan^{-1}(x)$的值	计算结果	
atan2	double atan2 (double x, double y);	计算 $\tan^{-1}(x/y)$的值	计算结果	

续表

函数名	函数原型	功　能	返回值	说　明
cos	double cos(double x);	计算 cos(x)的值	计算结果	x 的单位为弧度
cosh	double cosh(double x);	计算 x 的双曲余弦 cosh(x)的值	计算结果	
exp	double exp(double x);	求 e^x 的值	计算结果	
fabs	double fabs(double x);	求 x 的绝对值	计算结果	
floor	double floor(double x);	求出不大于 x 的最大整数	该整数的双精度实数	
fmod	double fmod(double x, double y);	求整除 x/y 的余数	返回余数的双精度数	
frexp	double frexp(double val, int * eptr);	把双精度数 val 分解为数字部分(尾数)x 和以 2 为底的指数 n，即 val＝x * 2^n，n 存放在 eptr 指向的变量中	返回数字部分 x 0.5≤x<1	
log	double log(double x);	求 $\log_e x$，即 ln x	计算结果	
log10	double log10(double x);	求 $\log_{10} x$	计算结果	
modf	double modf(double val, double * iptr);	把双精度数 val 分解为整数部分和小数部分，把整数部分存到 iptr 指向的单元	val 的小数部分	
pow	double pow(double x, double y);	计算 x^y 的值	计算结果	
rand	int rand(void);	产生－90 到 32767 间的随机整数	随机整数	
sin	double sin(double x);	计算 sin$x$ 的值	计算结果	x 单位为弧度
sinh	double sinh(double x);	计算 x 的双曲正弦函数 sinh(x)的值	计算结果	
sqrt	double sqrt(double x);	计算$\sqrt{x}$	计算结果	x 应≥0
tan	double tan(double x);	计算 tan (x)的值	计算结果	x 单位为弧度
tanh	double tanh(double x);	计算 x 的双曲正切函数 tanh (x)的值	计算结果	

2. 字符函数和字符串函数

ANSI C 标准要求在使用字符串函数时要包含头文件“string. h”，在使用字符函数时

要包含头文件“ctype.h”。有的C编译不遵循ANSI C标准的规定，而用其他名称的头文件。请使用时查有关手册。

函数名	函数原型	功　能	返回值	包含文件
isalnum	int isalnum(int ch);	检查ch是否是字母(alpha)或数字(numeric)	是字母或数字返回1；否则返回0	ctype.h
isalpha	int isalpha(int ch);	检查ch是否字母	是，返回1；不是，则返回0	ctype.h
iscntrl	int iscntrl(int ch);	检查ch是否控制字符(其ASCII码在0和0x1F之间)	是，返回1；不是，返回0	ctype.h
isdigit	int isdigit(int ch);	检查ch是否数字(0～9)	是，返回1；不是，返回0	ctype.h
isgraph	int isgraph(int ch);	检查ch是否可打印字符(其ASCII码在ox21到ox7E之间)，不包括空格	是，返回1；不是，返回0	ctype.h
islower	int islower(int ch);	检查ch是否小写字母(a～z)	是，返回1；不是，返回0	ctype.h
isprint	int isprint(int ch);	检查ch是否可打印字符(包括空格)，其ASCII码在ox20到ox7E之间	是，返回1；不是，返回0	ctype.h
ispunct	int ispunct(int ch);	检查ch是否标点字符(不包括空格)，即除字母、数字和空格以外的所有可打印字符	是，返回1；不是，返回0	ctype.h
isspace	int isspace(int ch);	检查ch是否空格、跳格符(制表符)或换行符	是，返回1；不是，返回0	ctype.h
isupper	int isupper(int ch);	检查ch是否大写字母(A～Z)	是，返回1；不是，返回0	ctype.h
isxdigit	int isxdigit(int ch);	检查ch是否一个十六进制数字字符(即0～9，或A到F，或a～f)	是，返回1；不是，返回0	ctype.h
strcat	char *strcat(char *strl, char*str2);	把字符串str2接到strl后面，strl最后面的'\0'被取消	strl	string.h
strchr	char *strchr(char *str, int ch);	找出str指向的字符串中第一次出现字符ch的位置	返回指向该位置的指针，如找不到，则返回空指针	string.h

续表

函数名	函数原型	功　能	返回值	包含文件
strcmp	int strcmp(char *str1,char *str2);	比较两个字符串 str1、str2	str1<str2,返回负数; str1=str2,返回 0; str1>str2,返回正数	string.h
strcpy	char *strcpy(char *str1,char *str2);	把 str2 指向的字符串复制到 str1 中去	返回 str1	string.h
strlen	unsigned int strlen(char *str);	统计字符串 str 中字符的个数(不包括终止符'\0')	返回字符个数	string.h
strstr	char *strstr(char *str1,char *str2);	找出 str2 字符串在 str1 字符串中第一次出现的位置(不包括 str2 的串结束符)	返回该位置的指针,如找不到,返回空指针	string.h
tolower	int tolower(int ch);	将 ch 字符转换为小写字母	返回 ch 所代表的字符的小写字母	ctype.h
toupper	int toupper(int ch);	将 ch 字符转换成大写字母	与 ch 相应的大写字母	ctype.h

3. 输入输出函数

凡用以下的输入输出函数,应该使用#include<stdio.h>把 stdio.h 头文件包含到源程序文件中。

函数名	函数原型	功　能	返回值	说明
clearerr	void clearerr(FILE *fp);	使 fp 所指文件的错误,标志和文件结束标志置 0	无	
close	int close(int fp);	关闭文件	关闭成功返回 0; 不成功,返回-1	非 ANSI 标准
creat	int creat(char *filename, int mode);	以 mode 所指定的方式建立文件	成功则返回正数; 否则返回-1	非 ANSI 标准
eof	int eof(int fd);	检查文件是否结束	遇文件结束,返回 1; 否则返回 0	非 ANSI 标准
fclose	int fclose(FILE *fp);	关闭 fp 所指的文件, 释放文件缓冲区	有错则返回非 0; 否则返回 0	
feof	int feof(FILE *fp);	检查文件是否结束	遇文件结束符返回非零值;否则返回 0	
fgetc	int fgetc(FILE *fp);	从 fp 所指定的文件中取得下一个字符	返回所得到的字符,若读入出错,返回 EOF	

续表

函数名	函数原型	功　能	返回值	说明
fgets	char * fgets(char * buf, int n, FILE * fp);	从 fp 指向的文件读取一个长度为(n-1)的字符串,存入起始地址为 buf 的空间	返回地址 buf,若遇文件结束或出错,返回 NULL	
fopen	FILE * fopen(char * filename,char * mode);	以 mode 指定的方式打开名为 filename 的文件	成功,返回一个文件指针(文件信息区的起始地址);否则返回 0	
fprintf	int fprintf(FILE * fp, char * format,args,…);	把 args 的值以 format 指定的格式输出到 fp 所指定的文件中	实际输出的字符数	
fputc	int fputc (char ch, FILE * fp);	将字符 ch 输出到 fp 指向的文件中	成功,则返回该字符;否则返回非 0	
fputs	int fputs(char * str, FILE * fp);	将 str 指向的字符串输出到 fp 所指定的文件	成功返回 0;若出错返回非 0	
fread	int fread (char * pt, unsigned size, unsigned n, FILE * fp);	从 fp 所指定的文件中读取长度为 size 的 n 个数据项,存到 pt 所指向的内存区	返回所读的数据项个数,如遇文件结束或出错返回 0	
fscanf	int fscanf(FILE * fp, char format,args,…);	从 fp 指定的文件中按 format 给定的格式将输入数据送到 args 所指向的内存单元(args 是指针)	已输入的数据个数	
fseek	int fseek(FILE * fp, long offset, int base);	将 fp 所指向的文件的位置指针移到以 base 所给出的位置为基准、以 offset 为位移量的位置	返回当前位置;否则,返回-1	
ftell	long ftell(FILE * fp);	返回 fp 所指向的文件中的读写位置	返回 fp 所指向的文件中的读写位置	
fwrite	int fwrite(char * ptr, unsigned size, unsigned n, FILE * fp);	把 ptr 所指向的 n * size 个字节输出到 fp 所指向的文件中	写到 fp 文件中的数据项的个数	
getc	int getc(FILE * fp);	从 fp 所指向的文件中读入一个字符	返回所读的字符,若文件结束或出错,返回 EOF	
getchar	int getchar(void);	从标准输入设备读取下一个字符	所读字符。若文件结束或出错,则返回-1	

续表

函数名	函数原型	功　能	返回值	说明
getw	int getw(FILE * fp);	从 fp 所指向的文件读取下一个字(整数)	输入的整数。如文件结束或出错,返回-1	非 ANSI 标准函数
open	int open(char * filename, int mode);	以 mode 指出的方式打开已存在的名为 filename 的文件	返回文件号(正数);如打开失败,返回-1	非 ANSI 标准函数
printf	int printf(char * format, args, …);	按 format 指向的格式字符串所规定的格式,将输出表列 args 的值输出到标准输出设备	输出字符的个数,若出错,返回负数	format 可以是一个字符串,或字符数组的起始地址
putc	int putc(int ch, FILE * fp);	把一个字符 ch 输出到 fp 所指的文件中	输出的字符 ch,若出错,返回 EOF	
putchar	int putchar(char ch);	把字符 ch 输出到标准输出设备	输出的字符 ch,若出错,返回 EOF	
puts	int puts(char * str);	把 str 指向的字符串输出到标准输出设备,将'\0'转换为回车换行	返回换行符,若失败,返回 EOF	
putw	int putw(int w, FILE * fp);	将一个整数 w(即一个字)写到 fp 指向的文件中	返回输出的整数,若出错,返回 EOF	非 ANSI 标准函数
read	int read(int fd, char * buf, unsigned count);	从文件号 fd 所指示的文件中读 count 个字节到由 buf 指示的缓冲区中	返回真正读入的字节个数,如遇文件结束返回 0,出错返回-1	非 ANSI 标准函数
rename	int rename(char * oldname, char * newname);	把由 oldname 所指的文件名,改为由 newname 所指的文件名	成功返回 0;出错返回-1	
rewind	void rewind(FILE * fp);	将 fp 指示的文件中的位置指针置于文件开头位置,并清除文件结束标志和错误标志	无	
scanf	int scanf(char * format, args,…);	从标准输入设备按 format 指向的格式字符串所规定的格式,输入数据给 args 所指向的单元	读入并赋给 args 的数据个数,遇文件结束返回 EOF,出错返回 0	args 为指针
write	int write(int fd, char * buf, unsigned count);	从 buf 指示的缓冲区输出 count 个字符到 fd 所标志的文件中	返回实际输出的字节数,如出错返回-1	非 ANSI 标准函数

4. 动态存储分配函数

ANSI 标准建议设 4 个有关的动态存储分配的函数，即 calloc( )、malloc( )、free( )、realloc( )。实际上，许多 C 编译系统实现时，往往增加了一些其他函数。ANSI 标准建议在“stdlib. h”头文件中包含有关的信息，但许多 C 编译系统要求用“malloc. h”而不是“stdlib. h”。读者在使用时应查阅有关手册。

ANSI 标准要求动态分配系统返回 void 指针。void 指针具有一般性，它们可以指向任何类型的数据。但目前有的 C 编译所提供的这类函数返回 char 指针。无论以上两种情况的哪一种，都需要用强制类型转换的方法把 void 或 char 指针转换成所需的类型。

函数名	函数原型	功　能	返回值
calloc	void * calloc(unsigned n, unsign size);	分配 n 个数据项的内存连续空间，每个数据项的大小为 size	分配内存单元的起始地址，如不成功，返回 0
free	void free(void * p);	释放 p 所指的内存区	无
malloc	void * malloc(unsigned size);	分配 size 字节的存储区	所分配的内存区起始地址，如内存不够，返回 0
realloc	void * realloc(void * p, unsigned size);	将 p 所指出的已分配内存区的大小改为 size，size 可以比原来分配的空间大或小	返回指向该内存区的指针

参考文献

[1] 谭浩强著．C程序设计(第二版)．北京：清华大学出版社，1999

[2] 谭浩强编著．C++程序设计．北京：清华大学出版社，2004

[3] 谭浩强，张基温，唐永炎编著．C语言程序设计教程．北京：高等教育出版社，1992

[4] 谭浩强编著．C程序设计题解与上机指导(第二版)．北京：清华大学出版社，1999

[5] 谭浩强编著．QBASIC语言教程．北京：电子工业出版社，1997

[6] 谭浩强，田淑清编著．PASCAL语言程序设计(第二版)．北京：高等教育出版社，1998

[7] C编写组编．常用C语言用法速查手册．北京：龙门书局，1995

[8] Herbert Schildt著．戴健鹏译．C语言大全(第二版)．北京：电子工业出版社，1994

[9] Herbert Schildt著．王曦若、李沛译．ANSI C标准详解．北京：学苑出版社，1994

[10] H M Peitel, P J Deitel. C How to program, second Edition. 蒋才鹏等译. C程序设计教程. 北京：机械工业出版社，2000

[11] Stephen G Kochan著. Programming in ANSI C. Hagden Books Indianapolis: Indiana, U. S. A, 1994

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引